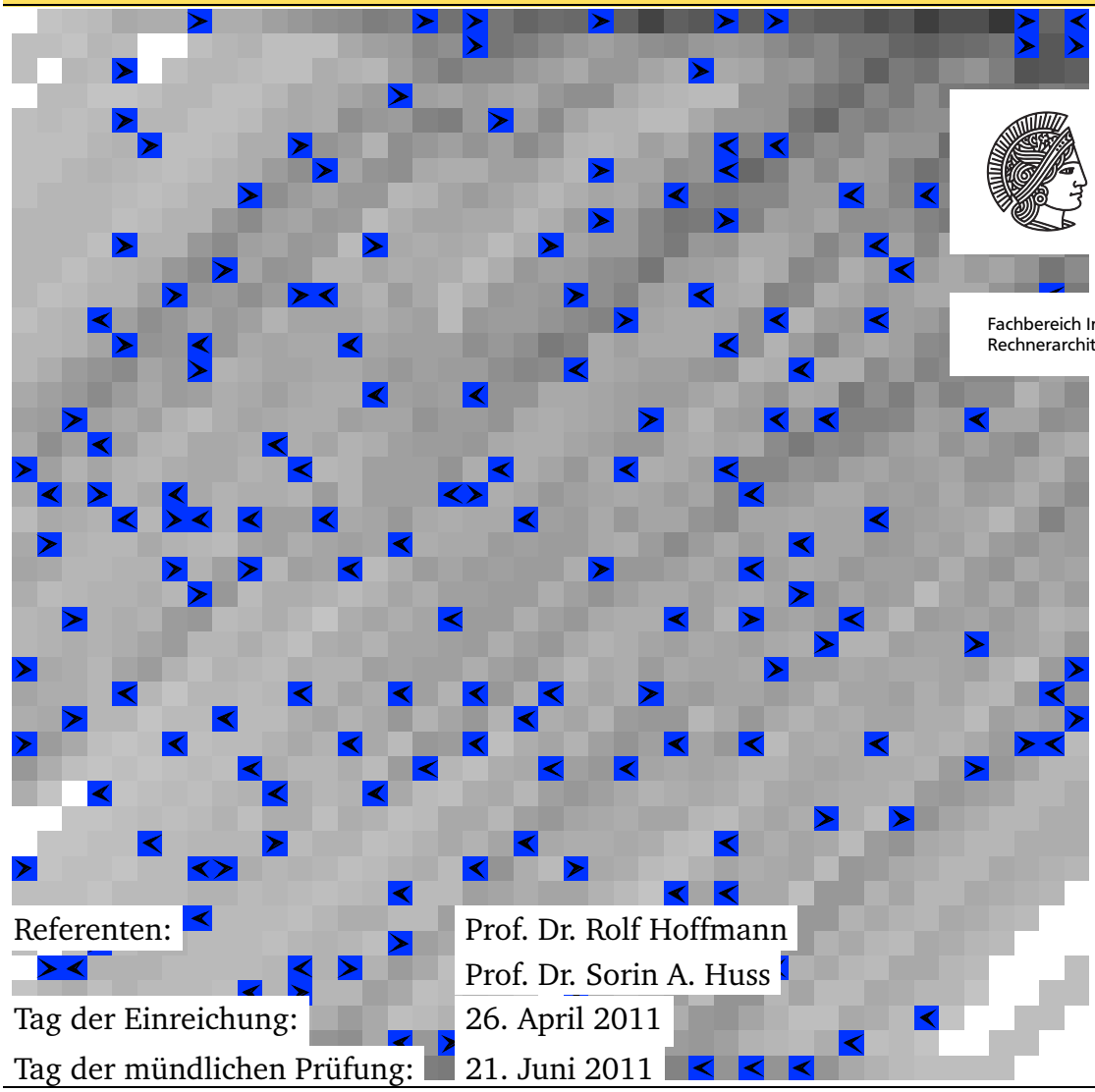
Konfigurierbare Prozessorsysteme zur hardwareunterstützten Simulation von Agentensystemen auf der Basis von Globalen Zellularen Automaten

Configurable Processing Systems for Hardware Accelerated Simulation of Agent Systems Using Global Cellular Automata

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Dipl.-Inform. Christian Alexander Schäck, geb. in Frankfurt a. M.
Juni 2011 — Darmstadt — D 17



TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Rechnerarchitektur

Referenten: Prof. Dr. Rolf Hoffmann
Prof. Dr. Sorin A. Huss
Tag der Einreichung: 26. April 2011
Tag der mündlichen Prüfung: 21. Juni 2011

Konfigurierbare Prozessorsysteme zur hardwareunterstützten Simulation von Agentensystemen auf der Basis von Globalen Zellularen Automaten
Configurable Processing Systems for Hardware Accelerated Simulation of Agent Systems Using Global Cellular Automata

genehmigte Dissertation von Dipl.-Inform. Christian Alexander Schäck, geboren in Frankfurt a. M.

1. Gutachten: Prof. Dr. Rolf Hoffmann
2. Gutachten: Prof. Dr. Sorin A. Huss

Tag der Einreichung: 26. April 2011
Tag der Prüfung: 21. Juni 2011

Darmstadt — D 17

Danksagung

Zunächst möchte ich meinen Eltern, ganz besonders meiner lieben Mutter, herzlich danken, die mich während der Anfertigung dieser Arbeit in jeder Hinsicht unterstützt und damit zum Erfolg und Gelingen erheblich beigetragen haben.

Herrn Prof. Dr. Rolf Hoffmann bin ich für die Möglichkeit zur Promotion, den anregenden Diskussionen, Beiträgen und Anmerkungen zu besonderem Dank verpflichtet. Er stand mir bei allen Fragen stets hilfreich zur Verfügung. Ich konnte mit Ihm immer offen über alle Probleme diskutieren.

Bei Herrn Prof. Dr. Sorin A. Huss möchte ich mich für die Übernahme des Korreferates bedanken. Ich bedanke mich auch bei den weiteren Mitgliedern der Prüfungskommission, den Professoren Alejandro P. Buchmann, Michael Goesele und bei Stefan Katzenbeisser für die Übernahme des Vorsitzes.

Des Weiteren haben eine Reihe von Kolleginnen und Kollegen in anregenden Diskussionen zu einem lockeren, dennoch ernsthaften Arbeitsklima ebenfalls zum Gelingen dieser Arbeit beigetragen. In alphabetischer Reihenfolge sind dies Patrick Ediger, Gudrun Harris, Dr.-Ing. Wolfgang Heenes, Johannes Jendrsczok, Thomas Paul, Hani Salah sowie alle Mitarbeiter des Fachgebiets Software Technologie der Technischen Universität Darmstadt.

Dankbar bin ich auch für die anregenden Diskussionen auf den internationalen Konferenzen und Workshops. Hieraus entstanden auch immer wieder neue Ideen und Impulse, die in diese Arbeit mit einflossen. In Erinnerung geblieben und bedeutsam für meine Arbeit ist dabei die Veröffentlichung *Algebra of Synchronization with Application to Deadlock and Semaphores*. Der dazugehörige Vortrag von Ernesto Gomez, den ich auf einer Konferenz in Japan kennen lernte, und die Diskussionen mit Ihm ergaben neue Impulse für meine Arbeit.

Zum Ausgleich neben der Arbeit an der Technischen Universität Darmstadt und meiner Promotion bin ich dem Verein *Hessen-Flieger Verein für Luftfahrt 1924 Darmstadt e. V.* beigetreten und habe dort die Lizenz für Privatpiloten erworben. Die thematische Abwechslung wie auch die zusätzliche Herausforderung haben zu den Inspirationen geführt, die wesentliche Entwicklungen dieser Arbeit begünstigten.

Kurzfassung

In dieser Arbeit werden verschiedene Hardwarearchitekturen für das GCA-Modell (engl.: Global Cellular Automata, GCA) entwickelt, bewertet und für die Simulation von Multi-Agenten-Systemen optimiert. Das GCA-Modell besteht aus einer Menge von Zellen, die ihren Zustand synchron-parallel abhängig von den Zuständen der Nachbarzellen ändern. Damit ist es ein massiv-paralleles Berechnungsmodell, bei dem, im Gegensatz zum CA-Modell, die Nachbarschaft nicht fest und lokal, sondern global und variabel ist. Das GCA-Modell eignet sich gut für die Umsetzung von Multi-Agenten-Systemen, da u. a. auch mit einfachen Zellregeln komplexes Verhalten simuliert werden kann und die Zellregel unabhängig von der Anzahl der Prozessoren ist. Die Programmierung kann einfach gehalten werden, da keine komplexen Synchronisationskonstrukte verwendet werden müssen. Es wird die unterschiedliche Leistungsfähigkeit verschiedener Architekturen dargestellt und aufgezeigt, in welcher Art und Weise diese weiter optimiert werden können bzw. wurden. Die Auswertung der Architekturen erfolgt auf verschiedenen FPGAs (Field Programmable Gate Array) mit unterschiedlichen Testdaten. Obwohl die in dieser Arbeit gezeigten Architekturen allgemein einsetzbar sind, liegt der Fokus auf der beschleunigten Simulation von Multi-Agenten-Systemen. Zunächst wurde eine *allgemeine Hardwarearchitektur* für das GCA-Modell entwickelt und dabei verschiedene Verbindungsnetzwerke untersucht und optimiert. Als Verbindungsnetzwerke wurden das Busnetzwerk mit zwei verschiedenen Arbitrierungsmöglichkeiten, das Omeganetzwerk mit unterschiedlichen Optimierungen und das Ringnetzwerk ebenfalls mit einer Optimierung, um die Zugriffe zu beschleunigen, realisiert und ausgewertet. Um die Simulation von Multi-Agenten-Anwendungen weiter zu beschleunigen, wurde eine *Architektur mit Hashfunktionen* implementiert, bei der leere Zellen von der Berechung ausgeschlossen werden. Diese Architektur erweist sich als sehr leistungsfähig, obwohl die Skalierbarkeit stark begrenzt ist und die maximale Taktfrequenz eher gering ist. In einer weiteren *speicheroptimierten Architektur* wurde der gleiche Ansatz zugrunde gelegt, aber die Skalierbarkeit verbessert. Durch die nun höhere maximale Taktfrequenz und den einfacheren Aufbau der Architektur waren weitere Beschleunigungen möglich. Die allgemeine Hardwarearchitektur eignet sich für die Berechnung von Multi-Agenten-Systemen, die sehr viele Agenten beinhalten. Multi-Agenten-Systeme mit weniger Agenten, dafür aber sehr großen Agentenwelten, können dafür sehr gut auf der Hardwarearchitektur mit Hashfunktionen simuliert werden, da hier die Größe der Agentenwelt irrelevant ist und lediglich die Anzahl der Agenten durch die Speichergröße limitiert ist. Die speicheroptimierte Hardwarearchitektur zeichnet sich durch eine sehr hohe Simulationsgeschwindigkeit aus. Da der Begriff des Agenten in der Literatur unterschiedlichste Verwendung findet, erfolgt zuerst eine Definition des Agenten, wie er in dieser Arbeit verstanden wird. Für die Darstellung der Agentenwelten sowie für die Konfiguration der FPGAs ist ein Simulationsprogramm (AgentSim) in Java entwickelt worden. Die Hauptfunktionen des Simulationsprogramms sind die Darstellung, Auswertung, Fehleranalyse, Programmierung und Simulation verschiedenster Agentenwelten. Die Anwendungen für Multi-Agenten-Systeme sind sehr vielfältig und erstrecken sich u. a. über Gebiete der Biologie, Soziologie, Verkehrsphysik, Evakuierungssimulation, Computergraphik, Filmtechnik sowie Wirtschaftssimulation.

Abstract

In this work several different hardware architectures for the GCA-model (Global Cellular Automata, GCA) have been developed, evaluated and optimized for the simulation of multi-agent systems. The GCA-model consists of a set of cells which change their states synchronously parallel depending on the states of the neighbor cells. Therefore it is a massively parallel computation model. In differentiation to the CA-model the neighbors are neither fixed nor local, instead they are variable and global. The GCA-model is suitable for implementing multi-agent systems because complex behavior can be simulated with rather simple cell rules and the cell rule is independent of the amount of processors used in the hardware architecture. Programming can be kept simple as no complex synchronization patterns have to be used. The capability of each architecture is pointed out and it is shown in which way they can be further optimized. The evaluation of the architectures has been done using different FPGAs (Field Programmable Gate Array) with varying test sets. Although the architectures can be generally used, the focus lies within the accelerated simulation of multi-agent systems. Starting with a general architecture for the GCA-model several networks have been examined and optimized. A bus network with two arbitration techniques, an omega network with several optimizations and a ring network with an optimization to accelerate the accesses through the network have been realized and evaluated. To further accelerate the simulation of multi-agent systems a architecture with hash functions, thus excluding empty cells from computation, has been implemented. This architecture has been proven to be very efficient, even though the scalability and the maximum clock frequency are rather low. In another memory optimized architecture the same approach was kept but the scalability was improved. With a higher clock frequency and a simpler design of the architecture further accelerations were reached. The general architecture is suitable for calculating multi-agent systems that involve many agents. Multi-agent systems with less agents but a very large agent world can be simulated very well with the architecture using hash functions, as the size of the agent world is irrelevant and only the amount of agents is limited by the size of the memory. The memory optimized architecture reaches a very high simulation speed. As the term agent is used in a very divers way in the literature it is necessary to first give a definition of how the term is used in this work. To display the agent worlds and to configure the FPGA a Java simulation application (AgentSim) has been developed. The main features of the simulation application are to display, evaluate, error analysis, programming and simulation of different agent worlds. The applications for multi-agent simulation are versatile range from topics such as biology, sociology, traffic physics, evacuation simulation, computer graphics, film technique to economy simulation.

Inhaltsverzeichnis

Variablen- und Abkürzungsverzeichnis

In dieser Arbeit werden einige Abkürzungen, Variablen sowie Symbole verwendet, die in der folgenden Tabelle aufgelistet sind. Auf Grund der Vielzahl an verwendeten Variablen ist eine eindeutige Zuordnung über die ganze Arbeit hinweg nicht möglich. Variablen werden daher teilweise mehrfach verwendet. Die Verwendung wird an der entsprechenden Stelle erklärt und ergibt sich zudem aus dem entsprechenden Kontext.

Abkürzungen	
Globaler Zellularer Automat	GCA
Einarmige universelle GCA-Architektur	MPA
Multiarmige speicherkonfigurierbare universelle GCA-Architektur	MPAB
Spezialisierte GCA-Architektur mit Hashfunktionen	HA
Speicherreduzierte GCA-Architektur	DAMA
Cell-Update-Rate	CUR
Agent-Update-Rate	AUR
Generation-Update-Rate	GUR
Busnetzwerk mit dynamischer Arbitrierung	BDPA
Busnetzwerk mit Round-Robin Arbitrierung	BRRA
Ringnetzwerk	RN
Ringnetzwerk mit Forwarding	RNFF
Omeganetzwerk	ON
Omeganetzwerk mit Registerstufe	ONR
Omeganetzwerk mit unsynchronisierten Lesezugriffen ohne Registerstufe	ONU
Omeganetzwerk mit unsynchronisierten Lesezugriffen mit Registerstufe	ONUR
Omeganetzwerk mit FIFO-Shuffleelementen	ONP
Multi-Agenten-System	MAS
Architekturspezifisch	
Anzahl der Prozessoren	p
Anzahl der Zellen, die gespeichert werden können	q
Anzahl der Zellen	n
Anzahl der Generationen	g
Aktuelle Generation	u
Linkfeld	L
Datenfeld	D
Anzahl der Linkfelder	m
Anzahl der Datenfelder	n
Block	B
Agentenweltspezifisch	
Ausführungszeit	t
Feldgröße in X-Richtung	X
Feldgröße in Y-Richtung	Y

X-Position eines Agenten/einer Zelle	x
Y-Position eines Agenten/einer Zelle	y
Agentendichte	d
Agenten (allgemein)	a_g
Agenten (aktiv, bewegend)	a_m
Hindernis	h
Agent bewegt sich	b
Agent bleibt stehen	s
Typ des Agenten	T
Agentenzelle	A
Leere Zelle	E
Hinderniszelle	H
Agentenblickrichtung	D
Blickrichtung des Agenten nach Norden	NORD
Blickrichtung des Agenten nach Süden	SÜD
Blickrichtung des Agenten nach Osten	OST
Blickrichtung des Agenten nach Westen	WEST
Allgemein	
Zellindex 1	i
Zellindex 2	k
Zelle mit Zellindex i	C_i
Hashfunktion	f_h
Rehash Variable	r_h
Radius der Nachbarschaft	rad
Dimension des Zellfeldes	dim
Menge der ganzen Zahlen	$\mathbb{Z}$
Menge der natürlichen Zahlen	$\mathbb{N}$
Anzahl an Threads	τ
Anzahl an Rechenkernen	k
Taktanzahl	T
Problemgröße	N
Zugriff auf Block	W
Agentenfunktion	f_a
Agentenfunktion für neue Blickrichtung	f_t
Agentenfunktion für die Evaluierung der Nachbarschaft	f_e
Agentenfunktion für die Evaluierung der Nachbarschaft	f_{free}
Verkehrssimulation	
Nagel-Schreckenberg Geschwindigkeit	v
Nagel-Schreckenberg Geschwindigkeit Zwischenwert	v_1
Nagel-Schreckenberg Geschwindigkeit Zwischenwert	v_2
Nagel-Schreckenberg Geschwindigkeit Zwischenwert	v_3
Nagel-Schreckenberg Trödelwahrscheinlichkeit	π
Nagel-Schreckenberg Zufallszahl	r
Nagel-Schreckenberg Maximalgeschwindigkeit	v_{max}
Nagel-Schreckenberg Abstand	Δ
Nagel-Schreckenberg Distanz zu leerer Zelle	α

Nagel-Schreckenberg Grenze für Zufallszahlenbereich	Γ
Nagel-Schreckenberg Reduktion	R
Nagel-Schreckenberg Zeiger	L
Nagel-Schreckenberg neue Geschwindigkeit	z
Nagel-Schreckenberg Zeiger für mehrspurige Realisierung	BL, BR, FL, FR
Mehrkörperproblem	
Mehrkörperproblem Masse	$Body_m$
Mehrkörperproblem X-Position	$Body_x$
Mehrkörperproblem Y-Position	$Body_y$
Mehrkröperproblem Z-Position	$Body_z$
Mehrkörperproblem Beschleunigung in X-Richtung	$Body_{ax}$
Mehrkörperproblem Beschleunigung in Y-Richtung	$Body_{ay}$
Mehrkörperproblem Beschleunigung in Z-Richtung	$Body_{az}$

Parameterverzeichnis

Die in dieser Arbeit aufgeführten Quellcodes verwenden eine Vielzahl an Parametern. Hier werden die Parameter aus den Quellcodes dargestellt und erläutert.

`GENS`	Anzahl der Generationen
`LOCL_CELLS`	Anzahl der Zellen im Datenspeicher der Verarbeitungseinheit
`SC`	Adresse der ersten Zelle im Adressspeicher (globaler virtueller Adressraum)
`DAT_CELLS`	Insgesamte Anzahl der Zellen in der Architektur
`MAXX`	Dimension der Agentenwelt in X-Richtung
`MAXY`	Dimension der Agentenwelt in Y-Richtung
`CELL_OBSTACLE`	Zellwert eines Hindernisses
`CELL_AGENT`	Zellwert eines Agenten
`CELL_FREE`	Zellwert einer freien/leeren Zelle
`NXTG_KEEP`	Generationswechsel ohne die alte Generation zu löschen (DAMA)
`NXTG_DELETE`	Generationswechsel und Löschen der alten Generation (DAMA)

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Diese Dissertation beschäftigt sich mit der hardwareunterstützten Simulation von Multi-Agenten-Systemen (MAS) unter Verwendung des Modells des Globalen Zellularen Automaten (engl.: Global Cellular Automata, GCA). In der folgenden Einführung werden zunächst die wichtigsten Begriffe, wie sie in dieser Dissertation verwendet werden, erläutert. Die genaue Definition erfolgt dann in dem jeweiligen Kapitel. Dann wird ein Überblick in die Thematik gegeben. Dabei werden aktuelle Anwendungen und Forschungsergebnisse berücksichtigt. Im Anschluss an diese Einführung folgt die Zielsetzung dieser Arbeit.

1.1 Einführung in das Themengebiet

Zunächst sind die Begriffe Multi-Agenten-System, Multi-Agenten-Simulation, Agent und Multi-Agenten-Anwendung einzuordnen.

Ein *Multi-Agenten-System (MAS)* beschreibt die Gesamtheit eines zu simulierenden Systems (Abb. 1.1). Das System beinhaltet dabei alle Objekte, deren Umwelt, Eigenschaften und Parameter, die für die Beschreibung und Simulation des Systems notwendig sind. Die eigentliche Durchführung der Simulation eines Multi-Agenten-Systems in diskreten Zeitschritten wird mit *Multi-Agenten-Simulation* bezeichnet. Für die Durchführung der Simulation kommt dazu ein *Multi-Agenten-Simulations-System* zum Einsatz. Es besteht aus den erforderlichen Recheneinheiten, die für die Simulation benötigt werden. Die *Multi-Agenten-Anwendung* bestimmt dabei das Verhalten der Agenten, deren Interaktionen und somit letztendlich das Simulationsergebnis.

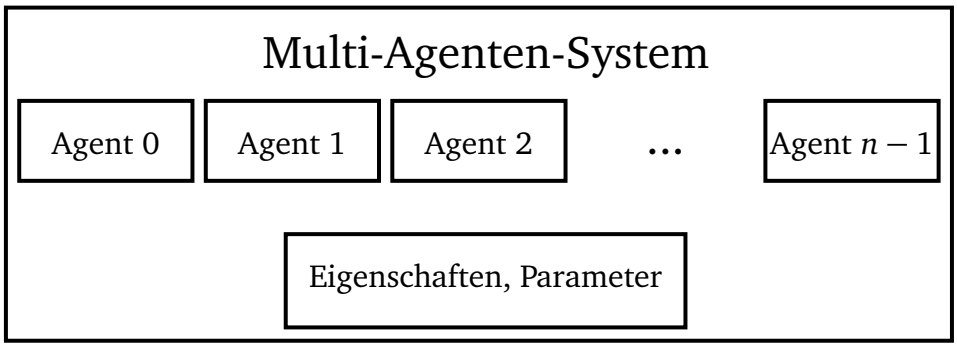


Abbildung 1.1: Übersicht über das Multi-Agenten-System

Der Begriff des *Agenten* wird in vielen Bereichen der Informatik verwendet. Aber auch in anderen Fachgebieten findet der Begriff des Agenten Anwendung. Er beinhaltet dabei sowohl real existierende u. U. autonom agierende Roboter als auch virtuelle Agenten (sogenannte Softwareagenten [GK94]). Agenten werden auch in den Wirtschaftswissenschaften verwendet. So beschreibt [Vet06] ein Multi-Agenten-System zur Verhandlungsautomatisierung in elektronischen

Märkten. Agenten können aber auch reale Lebewesen repräsentieren, so z. B. Menschen oder Tiere. Ein Beispiel für die Simulation von Fußgängerverhalten ist in [DTJ00] beschrieben. Die Fußgängersimulation findet häufig Anwendung im Kontext der Evakuierung, z. B. von Gebäuden [BKK08, Kir02]. In [STC06, TSC06] wird mit Hilfe der Multi-Agenten-Simulation das Verhalten von Honig- und Sammelbienen untersucht. Ameisen bzw. Ameisenspuren [CGNS02, JSCN04] werden auch durch Multi-Agenten-Systeme simuliert. Dabei gewonnene Erkenntnisse können dann in andere Multi-Agenten-Simulationen einfließen, wie z. B. die Verkehrssimulation. Mit eine der ältesten Simulationen für biologische Systeme ist Wa-Tor [Dew84], eine Simulation, die zu den sogenannten Räuber-Beute-Modellen gehört. Agenten eignen sich aber auch für die Simulation von Straßenverkehr [NS92, Nag94, Sch99b] oder Zugverkehr [Ste01]. Das bekannte Nagel-Schreckenberg-Modell [NS92] verwendet ursprünglich zwar keine Agenten für die Simulation, diese sind aber aus Gründen der Vereinheitlichung und Abstraktion für die Verwendung eines Multi-Agenten-Systems sinnvoll. Durch die Vereinheitlichung und Abstraktion ist das MAS unabhängig von der zu simulierenden Anwendung. Das Verhalten der Agenten wird durch die Anwendung bestimmt, so repräsentiert z. B. ein Agent bei der Verkehrssimulation ein Auto. Die Verwendung von Agenten in der Medizin wird in [ZL05] gezeigt. Durch eine Multi-Agenten-Simulation kann z. B. die Ausbreitung von Viren simuliert werden. Das 1970 von John Conway entworfene „Game of Life“ [Gar70] kann ebenfalls als ein Multi-Agenten-System aufgefasst werden. Ursprünglich wurde es aber für Zellulare Automaten entworfen. Im Bereich der künstlichen Intelligenz verwendet man ebenfalls Agenten [Wei99, Seite 28-29]. Auch innerhalb dieses Gebiets besteht keine einheitliche Verwendung des Begriffs Agent, weshalb er in [Wei99] als ein autonomes Computersystem innerhalb einer Umgebung definiert wird.

Die Definition eines Agenten in verschiedenen Gebieten ist sehr unterschiedlich und reicht von kompletten autonomen Systemen bis hin zu Softwareagenten, die verschiedene Dienste ausführen. In [Klü01, Seite 13] wird auch auf die Problematik der Mehrfachverwendung dieses Begriffes eingegangen und eine Einordnung vorgenommen. Viele Problemstellungen können durch Agenten aufgefasst, dargestellt und gelöst werden. Aufgrund der genannten Problematik wird zunächst der Begriff des Agenten definiert.

In dieser Arbeit wird unter einem *Agenten* eine aktive Einheit verstanden. Ein Agent besitzt bestimmte, definierte Eigenschaften und besitzt ein bestimmtes Verhalten. Dabei kann er gemeinschaftlich mit anderen Agenten agieren, kommunizieren und dabei ein übergeordnetes Ziel verfolgen. Das Ziel muss dabei nicht zwangsläufig allen oder überhaupt einem Agenten „bewusst“ sein. Die Agenten können aber auch egoistische Verhaltensmuster aufweisen und in wechselseitiger Konkurrenz stehen. Es ist möglich, alle Agenten mit den gleichen Eigenschaften und Verhalten (homogene Agenten) oder Agenten mit unterschiedlichen Eigenschaften und Verhalten (inhomogene Agenten) auszustatten.

Die Agenten des MAS haben kein vordefiniertes Verhalten, es wird durch die Anwendung definiert. Die *Multi-Agenten-Anwendung* (kurz: Anwendung) beschreibt das Verhalten (homogen/inhomogen) der Agenten innerhalb eines Multi-Agenten-Systems.

Die Verwendung von Agenten für die Simulation sagt nichts über die Art, Effizienz und Geschwindigkeit der Simulation aus. Für die Simulation bieten sich eine Vielzahl von Modellen und

Hardwarearchitekturen an. Abbildung 1.2 stellt den Zusammenhang zwischen Multi-Agenten-Systemen, Modellen und Architekturen dar.

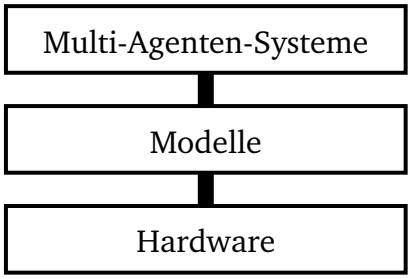


Abbildung 1.2: Übersicht: Multi-Agenten-Systeme, Modelle und Hardware

Für die Beschreibung und Realisierung von Multi-Agenten-Systemen können verschiedene Modelle verwendet werden. Die eigentliche Simulation der Modelle und damit auch der Multi-Agenten-Systeme kann auf verschiedenen Hardwarearchitekturen durchgeführt werden. Die Verwendung eines Modells trifft keine Aussage darüber, auf welcher Hardware dieses Modell zu simulieren ist oder wie dieses Vorgehen zu bewerten ist. Es stehen also vielfältige Möglichkeiten für die Multi-Agenten-Simulation zur Verfügung. Anzumerken ist, dass die Differenzierung zwischen Hardwarearchitektur und Modell in der Literatur oft nicht konsequent behandelt wird. Auf eine genaue Differenzierung der Begriffe wird zu einem späteren Zeitpunkt eingegangen.

1.2 Motivation

In der Einführung wurden einige Anwendungen für Multi-Agenten-Systeme aufgeführt. Für diese und weitere Anwendungen ist es sinnvoll, Multi-Agenten-Systeme zu verwenden. Dies erlaubt den Einsatz eines gemeinsamen Simulationssystems für eine große Menge von unterschiedlichen Multi-Agenten-Anwendungen. Davon profitieren auch Anwendungen, die zunächst nicht als Multi-Agenten-System klassifiziert wurden, wie z. B. [Gar70, Dew84, NS92]. Anstatt für jedes Multi-Agenten-System eine eigene Simulationsplattform zu entwickeln, ist es sinnvoll, ein gemeinsames Simulationssystem zu entwerfen, das für eine große Menge von Agentenanwendungen verwendet werden kann und gleichzeitig eine möglichst schnelle und effiziente Simulation ermöglicht. Dedizierte Simulationssysteme, die nur für eine Multi-Agenten-Anwendung konzipiert werden, haben den Vorteil, die Simulation der Anwendung maximal zu beschleunigen und berücksichtigen nur die notwendigen Elemente. Solche Systeme sind meist sehr aufwändig zu realisieren und nur für eine Anwendung zu verwenden. Systeme, die viele Multi-Agenten-Anwendungen simulieren können, haben dafür zum Nachteil, nicht so schnell zu sein, da alle Eigenschaften der zu simulierenden Anwendungen berücksichtigt werden müssen. Das Ziel ist also, ein Simulationssystem zu entwickeln, das die Multi-Agenten-Anwendungen möglichst schnell simuliert, aber auch für möglichst viele Anwendungen verwendbar ist. In dieser Arbeit werden Möglichkeiten aufgezeigt, wie solche Systeme aussehen, welche Vorteile und Nachteile sie mit sich bringen und wie Anwendungen dafür entwickelt werden können.

Grundsätzlich können für die Multi-Agenten-Simulation auch herkömmliche Standardhardwarearchitekturen oder Rechnersysteme verwendet werden. Hierbei bieten sich diverse Möglichkeiten, angefangen von handelsüblichen Prozessoren über Programmierschnittstellen [Ope97,

Ope08] bis hin zu programmierbaren Grafikkarten [NVI10a, SN09b, SN09a, DLR07]. Ebenfalls denkbar sind komplexe Rechnersysteme oder Cluster unter Verwendung von Protokollen zum Nachrichtenaustausch [Mes, Mes09].

Diese Arbeit untersucht die Verwendung des Modells des Globalen Zellularen Automaten (engl.: Global Cellular Automata, GCA) [HVW00, HVWH01] für die Simulation von Multi-Agenten-Systemen. Der Globale Zellulare Automat stellt eine Erweiterung des Zellularen Automaten (engl.: Cellular Automata, CA) [Neu66] dar. Ein Zellularer Automat besteht aus einer regelmäßigen n-dimensionalen Anordnung von *Zellen*. Jede Zelle besitzt einen *Zustand*, der über eine *Zustandsübergangsfunktion* in einen *Nachfolgezustand* überführt wird. Alle Zellen wechseln ihren Zustand in diskreten Zeitschritten, den sogenannten *Generationen*, parallel. Der Begriff Generation beschreibt dabei den Zustand aller Zellen zu einem bestimmten Zeitpunkt. Im Zellularen Automaten erfolgt der Zustandsübergang in Abhängigkeit von den direkten Nachbarzellen. Im Gegensatz zum Zellularen Automaten können im Globalen Zellularen Automaten Zugriffe auf Nachbarzellen wahlfrei und dynamisch zur Laufzeit erfolgen. Bei beiden Modellen handelt es sich um räumlich und zeitlich diskrete Modelle.

Da für das Modell des Globalen Zellularen Automaten keine direkte Umsetzung in Hardware im Sinne einer allgemeingültigen Architektur oder Plattform, wie z. B. die Compute Unified Device Architecture (CUDA), zur Verfügung steht und verschiedene Architekturvarianten untersucht werden sollen, werden für die Realisierung der verschiedenen Architekturen FPGAs (Field Programmable Gate Array) eingesetzt. Für eine effiziente Multi-Agenten-Simulation können so auch Anpassungen am GCA-Modell vorgenommen werden. Für die Verwendung des GCA-Modells spricht auch die gute Realisierbarkeit in Hardware. Da es nur Lesezugriffe auf Nachbarzellen gibt, entfällt eine aufwändige Implementierung für einen Schreibarbiter. Die Arbitrierung für die Lesezugriffe fällt einfacher aus und kann auch direkt in das Verbindungsnetzwerk integriert werden. Die Generationssynchronisation kann ebenfalls einfach realisiert werden.

Das Modell des Zellularen Automaten wurde schon erfolgreich in [NS92] für die Verkehrssimulation angewendet. Der Globale Zellulare Automat ermöglicht es, bisherige Anwendungen für den Zellularen Automaten ohne Änderung auszuführen. Eine Anpassung auf den Globalen Zellularen Automaten kann zu Geschwindigkeitsvorteilen bei der Ausführung der Anwendung oder einfacherer Programmierung führen [SHH10b, SHH11a].

Die Bedeutung einer schnellen Simulation in Zellularen Automaten sowie die Motivation für Spezial-Hardware wird in [Sch98] hervorgehoben:

> *„Die Simulation eines komplexen zellularen Automaten stellt hohe Anforderungen an ein Simulationssystem. Der Aufwand der Berechnung hängt von der Anzahl der Zellen und der Komplexität der Zellberechnung ab.*
> ...
> *Eine Beschleunigung des Berechnungsprozesses wird gerade bei Realzeit-Simulationen gefordert, in denen die Simulationsdauer eine bestimmte Zeit nicht überschreiten darf. Außerdem ist eine Beschleunigung bei großen ZAs mit mehreren Millionen Zellen notwendig.“ [Sch98, Seite 3]*

1.3 Ziele der Arbeit

Der Globale Zellulare Automat, als Erweiterung des klassischen Zellularen Automaten, soll für die Simulation von Multi-Agenten-Systemen verwendet, erweitert und untersucht werden (Betrachtung auf der Modellebene). Hierzu sind im speziellen die Ergebnisse von Vorarbeiten im Hinblick auf den Einsatz von Multi-Agenten-Anwendungen zu betrachten. Auf der Basis der Vorarbeiten sind die Ziele definiert.

Die Ziele dieser Arbeit sind im Einzelnen:

1. **Implementierung allgemeiner Hardwarearchitekturen**
   Es sind allgemeine Multiprozessorarchitekturen zur Unterstützung des GCA-Modells für klassische GCA-Anwendungen zu entwerfen, zu realisieren und zu bewerten. Als klassische Anwendung ist u. a. das Bitonische Mischen und Sortieren zu zählen. Dabei sind im Hinblick auf Simulation von Multi-Agenten-Anwendungen Optimierungsmöglichkeiten zu untersuchen.

2. **Entwurf einer Hierarchie**
   Es soll eine Hierarchie entworfen werden, die die zu untersuchenden Anwendungen (Multi-Agenten-Anwendungen) unter Zuhilfenahme von Modellen (u. a. GCA-Modell) auf die Architekturebene abbildet. Durch die Hierarchie soll es ermöglicht werden, die Anwendungen ohne größere Modifikationen auf verschiedenen Hardwarearchitekturen ausführen zu können.

3. **Realisierung der Optimierungsmöglichkeiten**
   Es sind optimierte und erweiterte Multiprozessorarchitekturen für die Multi-Agenten-Simulation zu entwerfen und zu realisieren. Dabei ist die allgemeine Programmierbarkeit zu erhalten, damit die Architekturen für viele Multi-Agenten-Anwendungen verwendet werden können. Um die Leistungsfähigkeit der Architekturen bewerten und vergleichen zu können, sind verschiedene Multi-Agenten-Anwendungen zu modellieren und zu implementieren.

4. **Einsatz von Spezialarchitekturen**
   Der Einsatz von Spezialarchitekturen für spezielle Multi-Agenten-Anwendungen ist zu untersuchen. Die besonderen Eigenschaften der Multi-Agenten-Anwendung sowie die Auswirkungen auf die Hardwarearchitektur sind zu ermitteln.

5. **Untersuchung der Verbindungsnetzwerke**
   Für die Kommunikation der Komponenten, die bei Multiprozessorarchitekturen notwendig sind, sollen verschiedene Verbindungsnetzwerke untersucht werden. Dabei ist es notwendig, anders als bei klassischen GCA-Anwendungen, nicht nur regelmäßige Zugriffsstrukturen zu unterstützen. Die Auswirkungen der verschiedenen Verbindungsnetzwerke sind zu evaluieren und zu bewerten.

6. **Auswertung und Vergleich der Hardwarearchitekturen**
   Die implementierten Architekturen sind auszuwerten und miteinander zu vergleichen. Dabei sind neben der erreichten Beschleunigung auch weitere Vor- und Nachteile der Architekturen zu diskutieren.

1.4 Notationen

In dieser Arbeit werden verschiedene Hervorhebungen und Notationen verwendet, die im Folgenden erklärt werden.

Für diese Arbeit inhaltlich wichtige Gedankengänge, welche aus fremden Arbeiten übernommen wurden, werden wie folgt hervorgehoben:

> *„Das ist ein Zitat.“*

Auszüge aus Quelltexten, die auszugsweise im Text näher erklärt werden, sind wie folgt dargestellt:

```
#define a 3
```

Dateinamen und Funktionsnamen, die nicht in Quelltexten vorkommen, werden durch die folgende Hervorhebung gekennzeichnet:

DATEI.DAT

Neu eingeführte Definitionen werden wie folgt kenntlich gemacht:

Definition 1.1 (Neue Definition) *Eine neue Definition.*

Einfache Definitionen, die keiner ausführlichen Erklärung bedürfen, werden direkt im Text definiert und wie folgt hervorgehoben:

Einfache Definition

1.5 Struktur der Arbeit

Diese Arbeit befasst sich mit sehr unterschiedlichen Themenkomplexen und erfordert daher zunächst eine tiefere Auseinandersetzung mit den einzelnen Themengebieten und eine entsprechende Literaturrecherche. Dazu wird in Kapitel 2 bis Kapitel 5 der aktuelle Forschungsstand aufgezeigt und die betroffenen Themengebiete dargestellt sowie für das weitere Vorgehen eingeordnet.

Zunächst wird in Kapitel 2 das Modell des Zellularen Automaten sowie dessen Erweiterung, das Modell des Globalen Zellularen Automaten, definiert und erläutert. Die Vor- und Nachteile dieses Modells sowie mögliche Einschränkungen und die historische Entwicklung werden dargelegt. Zum besseren Verständnis werden Beispielanwendungen und deren Umsetzung aufgeführt.

In Kapitel 3 wird die Struktur und der Aufbau des Multi-Agenten-Systems definiert. Auf Grund der unterschiedlichen Definitionen, die in der Literatur vorherrschen, werden zunächst die wichtigsten Begriffe und deren Verwendung sowie Bedeutung in dieser Arbeit definiert. Dazu werden

die unterschiedlichen Begriffsdefinitionen in der Literatur untersucht und deren Relevanz für die Arbeit herausgearbeitet. Ein Bezug zwischen Multi-Agenten-Systemen und Zellularen Automaten wird hergestellt und bereits Ansätze für Hardwarearchitekturen dargestellt.

In Kapitel 4 wird auf Multi-Agenten-Anwendungen eingegangen. Hierzu werden verschiedene Multi-Agenten-Anwendungen vorgestellt und erläutert. Einige dieser Anwendungen wurden auf den entwickelten Hardwarearchitekturen implementiert und ausgewertet.

Der aktuelle Forschungsstand und alternative Möglichkeiten zur Realisierung von Multi-Agenten-Systemen wird in Kapitel 5 dargestellt. Hierzu zählen existierende Architekturen und Systeme, wie z. B. die Compute Unified Device Architecture (CUDA), Realisierungen auf Mehrkernprozessoren unter der Verwendung unterschiedlicher Softwarekonstrukte sowie dedizierte Hardwarearchitekturen auf Field Programmable Gate Arrays (FPGA).

Die implementierten Multiprozessorarchitekturen für Multi-Agenten-Systeme unter Verwendung des GCA-Modells sind in Kapitel 6 erklärt und ausgewertet. Dazu werden zunächst Bewertungskriterien erarbeitet und dargestellt. Danach wird der allgemeine Aufbau der Architekturen sowie die einzelnen Komponenten erläutert. Im Anschluss werden die verschiedenen Architekturen im Detail beschrieben und unter der Verwendung unterschiedlicher Anwendungen ausgewertet. Zum Vergleich aller Architekturen wird eine künstliche Multi-Agenten-Anwendung auf allen Harwarearchitekturen ausgewertet. Es werden Vergleiche zu bereits bestehenden Hardwarearchitekturen dargestellt. Das Kapitel schließt mit einer theoretischen Betrachtung bezüglich der Skalierbarkeit der Architekturen und führt weitere Optimierungs- und Entwicklungsmöglichkeiten auf.

In Kapitel 7 werden Spezialarchitekturen vorgestellt, die im Rahmen von mir betreuten Diplomarbeiten und Masterarbeiten entstanden sind. Dabei werden unterschiedliche Aspekte betrachtet, z. B., wie eine GCA-Architektur mit einem Universalrechner gekoppelt werden kann. Für das Nagel-Schreckenberg-Modell entstand eine Spezialarchitektur auf der Basis der vorangegangenen Veröffentlichung [SHH10b]. Ebenfalls wird der Einsatz der *Global Cellular Automata Experimental Language* (GCA-L) untersucht. Die Ziele, Umsetzung sowie die Ergebnisse der Arbeiten werden vorgestellt und mögliche Optimierungen und weitere Entwicklungen vorgeschlagen.

Die Arbeit schließt in Kapitel 8 mit einer Zusammenfassung und einem Ausblick ab. Weitere für die Arbeit relevante Details sind im Anhang (Kapitel 9) aufgeführt.

Erkenntnisse aus dieser Arbeit wurden bereits auf internationalen Konferenzen und in Journalen veröffentlicht [SHH09a, SHH09b, SHH09c, SHH10a, SHH10b, SHH11a, SHH11b].

2 Zellularer Automat und Globaler Zellularer Automat

In diesem Kapitel wird zuerst auf das Modell des Zellularen Automaten eingegangen. Dieses Modell bildet die Grundlage für den darauf aufbauenden Globalen Zellularen Automaten. Der Globale Zellulare Automat bildet die Grundlage für die Architekturen, die in Kapitel 6 präsentiert werden. Für beide Modelle werden Anwendungen vorgestellt, die die Verwendung zellularer Modelle motivieren. Eine Abgrenzung zu anderen parallelen Modellen oder Architekturen wird nicht vorgenommen. Dies wurde ausführlich in [Hee07] behandelt.

2.1 Modell und Hardwarearchitektur

Die Realisierung von Multi-Agenten-Simulations-Systemen kann auf verschiedene Arten und auf verschiedenen Ebenen erfolgen. Allgemein erstrecken sich die Möglichkeiten von reinen Softwareimplementierungen bis hin zu spezialisierten Hardwarerealisierungen. Um die verschiedenen Ebenen einordnen zu können, werden die wichtigsten Begriffe zuerst definiert. Dies ist notwendig, da in der Literatur keine einheitliche Verwendung dieser Begriffe herrscht.

Die für diese Arbeit wichtigen voneinander abzugrenzenden Begriffe betreffen das Modell und die (Hardware-)Architektur.

In [Nee91] wird ein Modell wie folgt definiert:

> *„A model is a simplified representation of a system (or process or theory) intended to enhance our ability to understand, predict, and possibly control the behaviour of the system.“ [Nee91, Seite 30]*

Das Modell ist nur eine Repräsentation des Systems und nicht das System selbst, es enthält nicht notwendigerweise alle Attribute des Systems und ist ein künstliches Gebilde. In Anlehnung an [Sta73, Seite 128-133] und [Nee91, Seite 30-66] wird folgende Definition eines Modells herangezogen:

Definition 2.1 (Modell) *Ein Modell ist eine pragmatische, verkürzte Abbildung eines Systems.*

Das Modell beschreibt ein System und dessen Funktionsweise auf gleicher logischer Ebene, nicht aber, wie einzelne oder alle Funktionen umzusetzen sind. Es ist dabei nicht notwendigerweise vollständig, da es vom Gesamtsystem abstrahieren kann. Die allgemeine Modelldefinition kann für bestimmte Bereiche und Anwendungsfälle weiter spezifiziert werden. So z. B. das *Verarbeitungsmodell* (auch Ausführungsmodell oder Berechnungsmodell) [Den97] oder das *Programmiermodell*. Unter einem Verarbeitungsmodell versteht man, wie Daten in dem Modell verarbeitet werden. Das Programmiermodell beschreibt die Konstrukte, z. B. eine Programmiersprache,

die ein bestimmtes Verarbeitungsmodell unterstützen. Die obige Definition eines Modells ist sehr allgemein gefasst. In dieser Arbeit werden nur Modelle im Kontext der Informatik betrachtet. Die (Hardware)-Architektur wird in dieser Arbeit wie folgt definiert.

Definition 2.2 ((Hardware)-Architektur) *Die Architektur ist die Umsetzung eines Modells im Ganzen oder von einzelnen Teilen, sofern diese ein nach dem Modell funktionierendes Gesamtsystem ergeben.*

Die Architektur ist eine Umsetzung eines Modells. Zu einem Modell gibt es potenziell mehrere Architekturen. Die Art und Weise, wie ein Modell zu realisieren ist, ist a priori nicht festgelegt.

2.2 Das Modell des Zellularen Automaten

Zellulare Automaten (engl.: Cellular Automata, CA) wurden erstmals 1940 von Stanislaw Ulam vorgestellt [BSW85]. John von Neumann erweiterte den Zellularen Automaten zu einem universellen Berechnungsmodell mit 29 Zuständen [Neu66]. Sein Ziel war es, ein sich selbst replizierendes System zu entwickeln. Hierfür verwendete er den Zellularen Automaten. Dieser besteht aus einer Anordnung von Zellen in einem orthogonalen Gitter. Jede Zelle beinhaltet dabei den gleichen endlichen Zustandsautomaten. Der Zustand jeder Zelle ist gegeben durch den Zustand des Zustandsautomaten zu einem bestimmten Zeitpunkt. Jede Zelle ist zudem mit ihren direkten Nachbarzellen verbunden. Die Anzahl der Nachbarzellen ist fest aber beliebig wählbar. In der ursprünglichen Definition ist jede Zelle auch ihr eigener Nachbar. Aus den Zustandsinformationen der Nachbarzellen wird dann der Folgezustand der Zelle berechnet [Cod86]. Konrad Zuse veröffentlichte 1969 sein Buch „Rechnender Raum“ [Zus70]. Er beschreibt darin das Universum als einen großen Zellularen Automaten. Dabei geht er davon aus, dass die Naturgesetze diskreten Regeln folgen. Die Beliebtheit von Zellularen Automaten kann auf ihre Einfachheit bei gleichzeitig großem Potential für die Modellierung von komplexen Systemen zurückgeführt werden. Erneut populär wurden Zellulare Automaten in den 1970ern durch das „Game of Life“ von John Horton Conway [Gar70]. Das „Game of Life“ ist ein Zellularer Automat, in dem jede Zelle acht Nachbarn (die Zelle selbst ist hier nicht als Nachbar definiert) besitzt. Dabei kann der Zustand einer Zelle „lebend“ oder „tot“ sein. Der Zustand einer Zelle ändert sich durch die Anwendung der Zellregel in Abhängigkeit von der Anzahl an „lebenden“ Nachbarzellen. In diesen „klassischen“ Zellularen Automaten berechnen alle Zellen ihren Nachfolgezustand anhand der gleichen Berechnungsvorschrift. Der Nachfolgezustand ist also nur ortsabhängig, d. h. abhängig von den ausgewählten Nachbarzellen. Bei der Ausführung unterschiedlicher Berechnungsvorschriften ist der Nachfolgezustand orts- und ggf. zeitabhängig (vgl. uniform und non-uniform CA auf Seite 12). Anfang der 80er Jahre untersuchte Stephen Wolfram einfache eindimensionale Zellulare Automaten. Diese Automaten sind heute als Wolframregeln bekannt [Wol86] und zeigen, dass selbst mit einfachen Regeln komplexes Verhalten simuliert werden kann.

Die Autoren von [GSD+03] geben eine Übersicht über die folgenden Typen von Zellularen Automaten. Seit dem Bekanntwerden von Zellularen Automaten wurden viele Variationen dieses Automaten vorgestellt. Die ursprüngliche Struktur des 1940 vorgestellten Zellularen Automaten von John von Neumann hatte 29 Zustände [Neu66]. Der von Codd vorgestellte Zellulare Automat besaß acht Zustände [Cod86], wohingegen Banks mit vier Zuständen je Zelle arbeitete [Ban71]. Alle diese Zellularen Automaten arbeiteten mit einer Nachbarschaft von fünf Zellen

(die eigene Zelle sowie die vier orthogonalen Nachbarzellen). Das „Game of Life“ funktioniert hingegen mit zwei Zuständen je Zelle, verwendet aber eine Nachbarschaft von neun Zellen. Auch dieser Automat kann für universelle Berechnungen verwendet werden [Smi76].

2.2.1 Definition des Zellularen Automaten

Der Zellulare Automat [Neu66, Zus70] besteht aus einer *n*-dimensionalen Anordnung von *Zellen*. Jede Zelle ist charakterisiert durch einen *Zustand*. Der Zellzustand entspricht dem Zustand des endlichen Zustandsautomaten, der in jeder Zelle vorhanden ist. Über eine Zustandsüberführungsfunktion (auch Zellregel genannt) kann der aktuelle Zustand in einen Nachfolgezustand überführt werden. Dabei wechseln alle Zellen ihren Zustand gleichzeitig. Das Modell wird deshalb auch als massiv-parallel bezeichnet. Eine Zelle kann den Zustand ihrer Nachbarzellen lesen, nicht aber ändern. Die Nachbarzellen sind dabei statisch und lokal. Jede Zelle kann nur ihren eigenen Zustand anhand der Zustandsüberführungsfunktion ändern. Dabei findet die Berechnung der Zellzustände in diskreten Zeitschritten, den *Generationen*, statt. Eine Generation ist also eine Momentaufnahme aller Zellzustände zu einem bestimmten Zeitpunkt. Das Modell des Zellularen Automaten ist ein zeitlich sowie räumlich diskretes Modell, was aus der Verwendung von Generationen und Zellen ersichtlich ist. Angelehnt an die verschiedenen Definitionen von Zellularen Automaten wird dieser entsprechend [Mei05] wie folgt definiert.

Definition 2.3 (Zellularer Automat) *Ein Zellularer Automat ist ein 4-Tupel (G,E,U,f):*

- *reguläres Gitter G*
- *endliche Menge von Elementarzuständen E*
- *endliche Menge von Umgebungsindizes U für die gilt:* $c \in U, r \in G, r + c \in G$
- *lokale Zustandsüberführungsfunktion (Zellregel)* $f : E^n \rightarrow E$ *für* $n = |U|$

Zusammenfassend werden die folgenden Begriffe für einen Zellularen Automaten definiert und verwendet:

- **Zelle:** Eine Zelle ist das kleinste Element in einem Zellularen Automaten. Eine Zelle besteht aus einem endlichen Automaten.
- **Zellulares Feld:** Ein unendlich großes Gitter bestehend aus regelmäßig angeordneten Zellen.
- **Zellregel:** Eine Berechnungsvorschrift, die den aktuellen Zustand der Zelle in einen Folgezustand überführt. Im Zusammenhang mit Multi-Agenten-Systemen spricht man auch vom Agentenverhalten. Ist nicht nur die Zellregel einer Zelle gemeint, sondern die globale Berechnung des Feldes, spricht man allgemein auch von Anwendung, Applikation oder globalem Verhalten.
- **Nachbarschaft:** Die Nachbarzellen, die eine Zelle umgeben und von der Zellregel für die Berechnung verwendet werden. Im Gegensatz zur ursprünglichen Definition gehört nach der hier aufgeführten Definition die Zelle selbst nicht zur Nachbarschaft dazu.

- **Generation:** Eine Momentaufnahme aller Zellzustände zu einem bestimmten Zeitpunkt. Im zeitlichen Verlauf der Berechnung wird allen Generationen eine eindeutige Nummer zugeordnet.

Die Berechnung der Zellen kann durch verschiedene Überführungsschemata erfolgen [Hee07, Hoc98]. Das Überführungsschema besagt, wie die Zellregel auf die Zelle angewandt wird. Es wird beispielsweise unterschieden:

- **Synchron paralleles Überführungsschema:** Alle Zellen berechnen gleichzeitig ihren Nachfolgezustand durch Anwendung der Zellregel. Dieses Überführungsschema ist das klassische Überführungsschema für das Modell des Zellularen Automaten.
- **Asynchron sequenzielles Überführungsschema:** Die Zellen berechnen ihren Nachfolgezustand in einer bestimmten Reihenfolge. Änderungen werden deshalb sofort sichtbar und beeinflussen die Berechnung der nachfolgenden Zellen. Durch die Verwendung eines Zwischenspeichers kann verhindert werden, dass die Änderungen der zuerst berechneten Zellen direkt sichtbar werden. Dies entspricht einer Simulation des synchron parallelen Überführungsschemas.
- **Asynchron stochastisches Überführungsschema:** Die Zellen werden in zufälliger Reihenfolge ausgewählt und ihr Nachfolgezustand wird sofort berechnet.

Ein Überführungsschema ist u. a. immer dann notwendig, wenn ein CA auf einer Architektur ausgeführt werden soll, die z. B. nicht alle Zellen parallel berechnen kann.

Eine wichtige Eigenschaft des CA-Modells ist die synchrone Berechnung aller Zellen in diskreten Zeitschritten. Eine asynchrone Berechnung der Zellen führt zwangsläufig zu einem anderen Verhalten des Zellularen Automaten. Für einige Anwendungen ergibt sich dann ein konvergierendes Verhalten, im Gegensatz zu dem komplexen Verhalten bei synchroner Berechnung. Die Begründung für einen CA mit asynchroner Berechnung wird mit der Simulation natürlicher Systeme begründet, da dort kein globaler Takt vorgegeben ist [Sip97, Seite 66-67].

Abhängig von der Gestaltung der Zellregel wird heute ein Zellularer Automat weiter untergliedert. Es wird dann von einem *uniform CA* oder *non-uniform CA* gesprochen [Sip97, Cam, BV09].

- **uniform CA:** Ein uniform CA enthält für alle Zellen die gleiche Zellregel. Es ist somit der klassische Zellulare Automat gemeint.
- **non-uniform CA:** Bei einem non-uniform CA kann für jede Zelle eine eigene Zellregel vorgesehen werden. Dabei werden die grundsätzlichen Eigenschaften des CA nicht verändert. Diese Zellregel kann sich auch über die Generationen hinweg ändern. Der uniform CA kann als Sonderfall des non-uniform CA angesehen werden.

2.2.2 Nachbarschaften in Zellularen Automaten

Für die Berechnung des Nachfolgezustands können Informationen der direkten Nachbarzellen einfließen. Auf welche Nachbarzellen Zugriff besteht, kann im Einzelfall definiert werden. Die Nachbarschaft ist statisch und lokal, d. h. es kann nur auf die direkten Nachbarzellen zugegriffen werden und die Auswahl der Nachbarzellen muss a priori festgelegt werden. Üblich sind die Moore-Nachbarschaft, sie umfasst alle acht Nachbarzellen, und die Von-Neumann-Nachbarschaft, sie umfasst die vier Nachbarzellen Nord, Süd, Ost und West (Abb. 2.1). Ausgehend von der Zelle mit den Koordinaten (x, y) umfasst die Menge der Moore-Nachbarschaft (MN) die Zellen $MN = \{(x-1, y+1),\ (x, y+1),\ (x+1, y+1),\ (x-1, y),\ (x+1, y),\ (x-1, y-1),\ (x, y-1),\ (x+1, y-1)\}$, die Von-Neumann-Nachbarschaft (VNN) umfasst die Zellen $VNN = \{(x, y+1),\ (x-1, y),\ (x+1, y),\ (x, y-1)\}$.

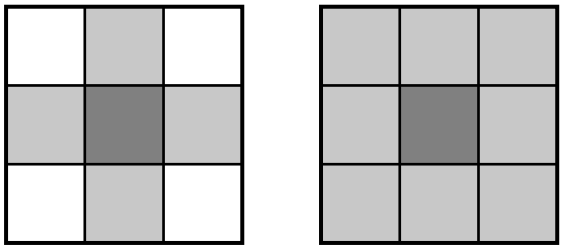


Abbildung 2.1: Nachbarschaften in Zellularen Automaten: Von-Neumann-Nachbarschaft (l.), Moore-Nachbarschaft (r.)

Für den Zellularen Automaten kann über einen Radius rad eine *erweiterte Nachbarschaft* definiert werden [Wol84, You84]. Diese erstreckt sich dann entsprechend der Größe von rad über die Nachbarzellen und erlaubt den Zugriff auf weiter entfernte Zellen. Die Verwendung der erweiterten Nachbarschaft, insbesondere für große rad, macht meist nur auf der Modellebene Sinn. Auf der Modellebene sind alle Zugriffe gleich effizient. Für reale Architekturen oder Implementierungen trifft dies nicht mehr zu.

Allgemein kann die Nachbarschaft wie in [Kar05] definiert werden. Gegeben sei ein zellulares Feld $\mathbb{Z}^{dim}$ mit der Dimension dim. Die Nachbarzellen der Zelle $\vec{x} \in \mathbb{Z}^{dim}$ sind gegeben durch einen Offsetvektor $\vec{y} \in \mathbb{Z}^{dim}$. Für die Von-Neumann-Nachbarschaft gilt für $\vec{y}$: $\left\|\vec{y}\right\|_1 < 1$, wobei

$$\left\|(y_1, y_2, \ldots, y_{dim})\right\|_1 = \left|y_1\right| + \left|y_2\right| + \left|y_3\right| + \ldots + \left|y_{dim}\right|$$

die Manhattendistanz ist.

Der Offsetvektor $\vec{y}$ für die Moore-Nachbarschaft beinhaltet alle Zellen für die $\left\|\vec{y}\right\|_\infty \leq 1$, wobei

$$\left\|(y_1, y_2, \ldots, y_{dim})\right\|_\infty = \max\left\{\left|y_1\right|, \left|y_2\right|, \ldots, \left|y_{dim}\right|\right\}$$

die Maximumsnorm ist. Für die allgemeine Moore-Nachbarschaft mit Radius $rad \in \mathbb{N}$ gilt $\left\|\vec{y}\right\|_\infty \leq rad$. Die Nachbarschaft der Zelle $\vec{x}$ ergibt sich durch $\vec{x} + \vec{y}$.
Neben diesen grundlegenden Nachbarschaften, gibt es noch Nachbarschaften in partitionierten Zellularen Automaten. In einem partitionierten Zellularen Automaten werden Gruppen von Zellen gebildet, z. B. als 2×2-Gruppe. Diese Gruppe kann dann mit jeder Generation über das

Zellfeld bewegt werden. Die Zellzustände werden dann immer pro Gruppe berechnet. Ein Beispiel für solch eine Nachbarschaft ist die Margolus-Nachbarschaft [Goo98] [TM87, Seite 119 - Seite 138]. Sie besteht aus einer 2×2-Gruppe, die in jeder Generation diagonal um eine Zelle verschoben wird. Eine derartige Nachbarschaft findet u. a. bei der Simulation von Gittergasmodellen Anwendung.

2.2.3 Einschränkungen des zellularen Felds bezüglich praktischer Anwendungen

Das ursprüngliche Modell des Zellularen Automaten geht von einem unendlich großen zellularen Feld aus. Für die Realisierung von Zellularen Automaten oder darauf aufbauenden Modellen als Hardwarearchitektur muss die Feldgröße beschränkt werden. Die Größe des zellularen Feldes ist letztendlich durch die zur Verfügung stehende Speichergröße der Architektur begrenzt. Die Begrenzung des zellularen Feldes kann aber auch durch die Anwendung vorgegeben sein. Meist reichen aber auch kleinere zellulare Felder aus. Die Begrenzung des zellularen Feldes auf eine endliche Größe bedarf der gesonderten Betrachtung der *Randzellen*. Randzellen sind jene Zellen, die an sich weniger als acht direkt angrenzende Zellen ($\neq$ Nachbarzellen) besitzen. Zu den angrenzenden Zellen zählen alle Zellen, die entweder eine gemeinsame Kante oder eine gemeinsame Ecke mit der betrachteten Zelle besitzen. Unter angrenzenden Zellen werden alle Zellen verstanden, die an eine andere Zelle angrenzen (unabhängig von der gewählten Nachbarschaft). Dieser Umstand muss gesondert betrachtet werden, wobei verschiedene Ansätze gewählt werden können [Goo98, Seite 79]. Die Randzellen können durch eine gesonderte Zellregel berechnet werden, die die fehlende Nachbarschaft ausgleicht. Dies kann auch durch Nachbarn mit konstanten Zuständen realisiert werden. Es kann aber auch durch zyklisches Schließen des zellularen Feldes (engl.: wrap-around) die Nachbarschaft wieder hergestellt werden. Eine Anpassung der Zellregel ist dann normalerweise nicht notwendig. Die dritte Möglichkeit besteht darin, die Nachbarschaft am Rand des zellularen Feldes zu spiegeln.

Für praktische Realisierungen (Hardwarearchitekturen) von Zellularen Automaten stehen meist nicht genügend Recheneinheiten zur Verfügung. Somit können nicht alle Zellen anhand des synchron parallelen Überführungsschemas gleichzeitig berechnet werden. Durch die Verwendung von Zwischenspeichern und der Verwendung des asynchron sequenziellen Überführungsschemas auf der Hardwareebene kann das synchron parallele Überführungsschema auf der Modellebene simuliert werden.

2.2.4 Anwendungen für Zellulare Automaten

Für den Zellularen Automaten gibt es eine Reihe von Anwendungen. Als eine der bekanntesten Anwendungen wird im Folgenden das „Game of Life“ genauer erklärt. Zellulare Automaten eignen sich aber für viele Anwendungen in den unterschiedlichsten Gebieten. Dazu zählen das Synchronisationsproblem (firing squad synchronization problem), Diskretisierung von Differentialgleichungen, Partikelsimulation, Gittergase, Verkehrssimulation [NS92] und viele mehr [Goo98].

Einer der bekanntesten Zellularen Automaten ist das „Game of Life“ von John Conway [Gar70] [TM87, Seite 19 - Seite 26]. Hierbei kann jede Zelle einen von zwei möglichen Zuständen annehmen. Eine Zelle ist entweder „lebend“ oder „tot“. Über folgende Zellregel wird unter Verwendung der Moore-Nachbarschaft der neue Zustand einer Zelle bestimmt.

- Eine „tote“ Zelle wird „lebendig“, wenn sie genau drei „lebende“ Nachbarn besitzt.
- Eine „tote“ Zelle bleibt „tot“, wenn sie nicht drei „lebende“ Nachbarn besitzt.
- Eine „lebende“ Zelle „stirbt“, wenn sie weniger als zwei oder mehr als drei „lebende“ Nachbarn besitzt
- Eine „lebende“ Zelle bleibt „lebend“, wenn sie zwei oder drei „lebende“ Nachbarn besitzt.

Das interessante am „Game of Life“ sind die vielfältigen scheinbar lebenden Objekte, die entstehen können. Trotz der einfachen Zellregel lässt sich vorab nicht bestimmen, ob eine gegebene Konfiguration stirbt.

Trotz der sich scheinbar bewegenden Objekte, gibt es im „Game of Life“ keine Objekte, die sich entweder von sich aus bewegen oder durch äußere Einflüsse bewegt werden. Die Bewegung der Objekte ist nur eine Interpretation des jeweiligen Beobachters. Eine Zelle ändert ihren Status zwischen den zwei Möglichkeiten. Es wird kein Objekt und auch keine Information von einer benachbarten Zelle kopiert bzw. „bewegt“ . Auf die Umsetzung aktiver Einheiten in Zellularen Automaten wird in Abschnitt 3.5.1 eingegangen.

2.3 Das Modell des Globalen Zellularen Automaten

Das Modell des Globalen Zellularen Automaten (engl.: Global Cellular Automata, GCA) [HVW00, HVWH01, Hee07] stellt eine Erweiterung zum klassischen CA-Modell (Vgl. Zellularer Automat in Abschnitt 2.2) dar. Im GCA-Modell gibt es *keine* statische Nachbarschaft mehr, jede Zelle kann Nachbarzelle jeder anderen Zelle sein. Die Nachbarschaft im GCA-Modell ist des weiteren wahlfrei und dynamisch, d. h. die Nachbarschaft kann in jeder Generation wechseln.

Der Zustand jeder Zelle besteht allgemein aus einer Anzahl n an Datenfeldern D und einer Anzahl m an Pointer-, Auswahl- oder Linkfeldern L. Die Datenfelder enthalten die relevanten Daten der Zelle für die aktuelle Generation. Über die Auswahlinformationen L können die Datenfelder beliebiger frei wählbarer Zellen gelesen werden. Die Auswahlinformationen können sich dabei in jeder Generation ändern. Die Anzahl der Auswahlinformationen beschreibt, wieviele Nachbarzellen in einer Generation parallel gelesen werden können. Bei m Auswahlinformationen spricht man auch von einem m-armigen Globalen Zellularen Automaten. Wenn ein Feld als Daten- und Auswahlfeld verwendet wird oder die Abgrenzung in Daten- und Auswahlfeld nicht notwendig ist, kann man diese auch allgemein zu $m + n$ Feldern oder Blöcken B zusammenfassen.
Das Prinzip der Berechnung eines neuen Zellzustandes ist in Abbildung 2.2 dargestellt. Der Zellzustand besteht aus zwei Auswahlfeldern (L_1, L_2) und einem Datenfeld D. Über die beiden Auswahlinformationen L_1 und L_2 werden die zwei aktuellen Datenfelder der zwei ausgewählten Nachbarzellen gelesen. Die drei Datenfelder, eigenes Datenfeld und die zwei Datenfelder

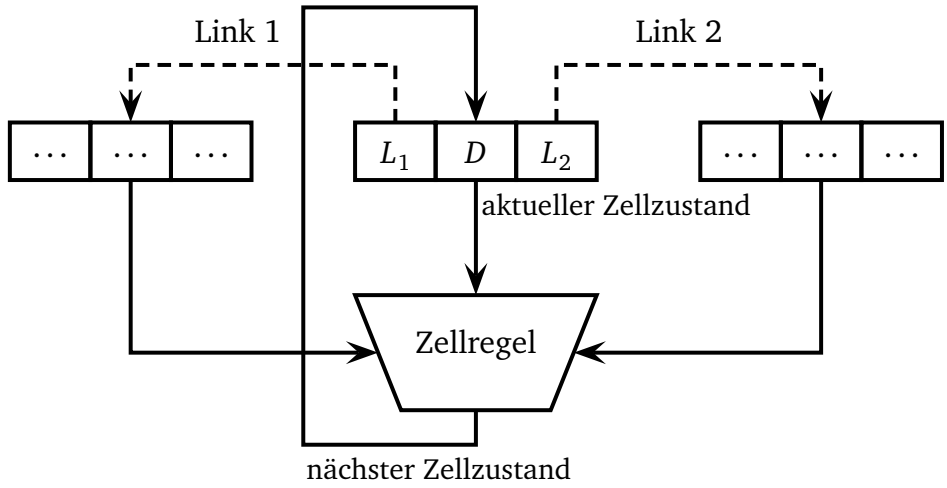


Abbildung 2.2: Prinzipielle Funktionsweise des GCA-Modells [HVW00, Seite 3]

der Nachbarzellen, werden für die Berechnung des neuen Zellzustandes verwendet. Die Zellregel berechnet den neuen Zellzustand der Zelle.

Für das GCA-Modell werden in der Literatur [Hof10a, Seite 5-6] und [Hof10b, Seite 2-3] drei verschiedene Varianten genannt. Der Unterschied der drei Varianten (basic, general und condensed) liegt in der Auswahlberechnung der Nachbarzelle:

- **Basic:** Jede Zelle verwendet direkt die Auswahlinformation L, um die Nachbarzelle zu selektieren. Der Folgezustand der Zelle bestimmt sich aus den Daten D der Zelle und der Nachbarzelle sowie dem Auswahlfeld L der Zelle und der Nachbarzelle. Alle Zellen werden synchron parallel aktualisiert.

- **General:** Jede Zelle verwendet eine Funktion zur Berechnung der Nachbarzelle. Die Nachbarzelle wird aus dem Datenfeld D und dem Auswahlfeld L bestimmt. Die Selektion der Nachbarzelle ist somit abhängig von der Auswahlinformation L und der Dateninformation D. Es wird also zuerst die Adresse der zu selektierenden Nachbarzelle berechnet und danach auf diese zugegriffen.

- **Condensed:** Anstatt eines Auswahlfeldes L und eines Datenfeldes D wird nur ein kombiniertes Feld verwendet. Das kombinierte Feld enthält die Daten und wird gleichzeitig zur Selektion der Nachbarzellen verwendet. Für die Selektion wird eine Selektionsfunktion verwendet.

Für die Zugriffe auf Nachbarzellen im GCA-Modell wird in [Hee07] das Zugriffsmuster und die Zugriffsstruktur definiert:

Definition 2.4 (Zugriffsmuster) *Das Zugriffsmuster ist die Gesamtheit aller Zugriffe in einer Generation.*

Definition 2.5 (Zugriffsstruktur) *Die Zugriffsstruktur bezeichnet die Gesamtheit der Zugriffsmuster einer Anwendung.*

Die Zugriffsstruktur wird weiter unterteilt in die statische Zugriffsstruktur und die dynamische Zugriffsstruktur. Eine Zugriffsstruktur ist statisch, wenn über alle Generationen das gleiche Zugriffsmuster angewandt wird. Im Gegensatz dazu ist eine Zugriffsstruktur dynamisch, wenn in mindestens einer Generation ein zu den anderen Generationen unterschiedliches Zugriffsmuster vorliegt.

Der Zusammenhang zwischen dem GCA-Modell und dem CROW-PRAM-Modell wird in dieser Arbeit nicht behandelt. Hierzu sei auf die Literatur [OK09] verwiesen.

2.3.1 Anwendungen für Globale Zellulare Automaten

Im Folgenden wird eine kurze Übersicht über Anwendungen bzw. Algorithmen für das GCA-Modell gegeben. Diese Beispielanwendungen sollen die Mächtigkeit und Funktionsfähigkeit des GCA-Modells verdeutlichen. Eine umfassendere Übersicht und Auswertung findet sich in [Hee07], in dessen Anlehnung auch der Begriff GCA-Anwendung definiert ist [Hee07, Seite 56].

Definition 2.6 (GCA-Anwendung) *Eine GCA-Anwendung ist eine Folge von Generationsberechnungen im GCA-Modell.*

Bitonisches Mischen und Sortieren

Zu den klassischen Anwendungen für das GCA-Modell gehört das Bitonische Mischen und Sortieren [SHH09b, SHH09c]. Das Bitonische Sortieren ist ein Sortierverfahren, das sich sehr gut für eine Hardwarerealisierung eignet [Bat68]. Im Gegensatz zu anderen Sortierverfahren (z. B. Quicksort [Hoa62], Heapsort [Wil64, Flo64]) ist die Reihenfolge der Vergleiche beim Bitonischen Sortieren unabhängig von den Daten. Man spricht in diesem Fall auch von einem Sortiernetz [CLRS01, Knu98]. Das Bitonische Mischen und Sortieren eignet sich auf Grund seiner Eigenschaften sehr gut für eine Realisierung im GCA-Modell [Hee07, Seite 94-119] [HVH01]. Nachteilig ist allerdings die Komplexität des Bitonischen Sortierens, die mit $Nlog^2N$ angegeben werden kann. Allerdings ist die Laufzeit des Bitonischen Sortierens für eine bestimmte Problemgröße konstant und datenunabhängig. Die Grundlage des Bitonischen Sortierens bildet das Bitonische Mischen.

Zunächst wird der Begriff einer bitonischen Folge definiert (u. a. [Bat68, Seite 309]).

Definition 2.7 (Bitonische Folge) *Eine Folge $A = a_0, \ldots, a_n$ heißt bitonische Folge, wenn entweder*

1. *ein $k \in [0, n)$ existiert, so dass $a_0, \ldots, a_k$ monoton wächst und $a_{k+1}, \ldots, a_n$ monoton fällt, oder*
2. *eine zyklische Verschiebung existiert, so dass 1. erfüllt ist*

Das Bitonische Sortieren für eine beliebige Zahlenfolge besteht im wesentlichen aus zwei Schritten. Im ersten Schritt wird die Zahlenfolge in eine bitonische Folge transferiert. Dieser Schritt

wird als Vorsortieren bezeichnet. Der zweite Schritt, das Bitonische Mischen, transferiert die bitonische Zahlenfolge in eine sortierte Zahlenfolge. Wenn man davon ausgehen kann, dass nur bitonische Folgen vorliegen, kann auf das Vorsortieren verzichtet werden.

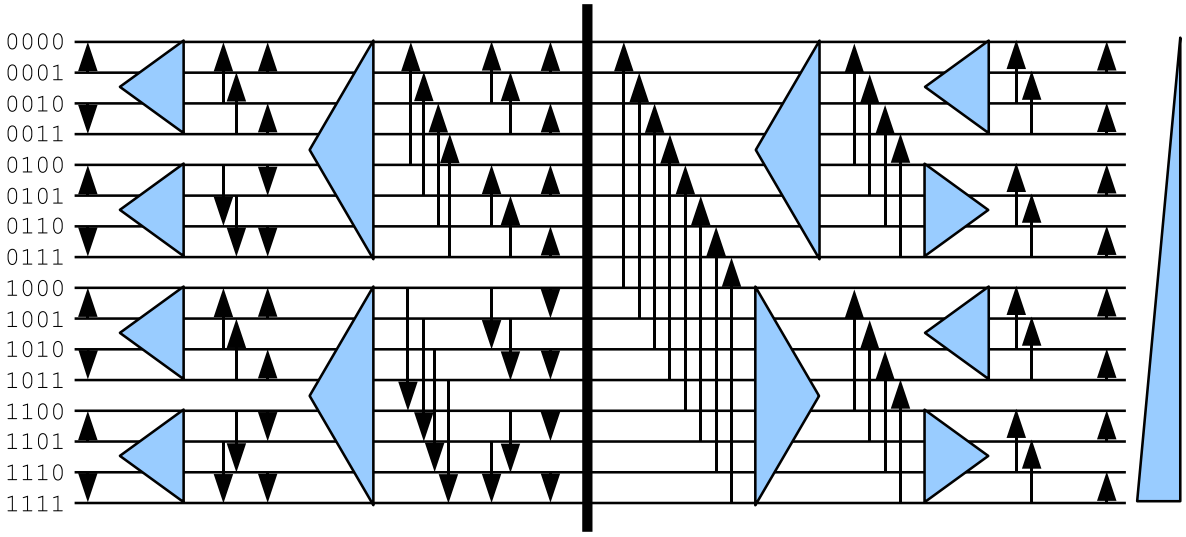


Abbildung 2.3: Funktionsweise des Bitonischen Sortierens bestehend aus den beiden Schritten: Vorsortieren (links) und Bitonisches Mischen (rechts). Abbildung auf der Basis von [Knu98, Seite 235 Abbildung 56] und [Hof10c, Seite 38]

Die schwarzen Pfeile in Abbildung 2.3 stellen die externen Zugriffe auf die Zellen dar. Es ist nur eine Zugriffsrichtung dargestellt. Die Dreiecke stellen die bitonischen Teilfolgen dar. Die externen Zugriffe können als Zweierpotenz angegeben werden.

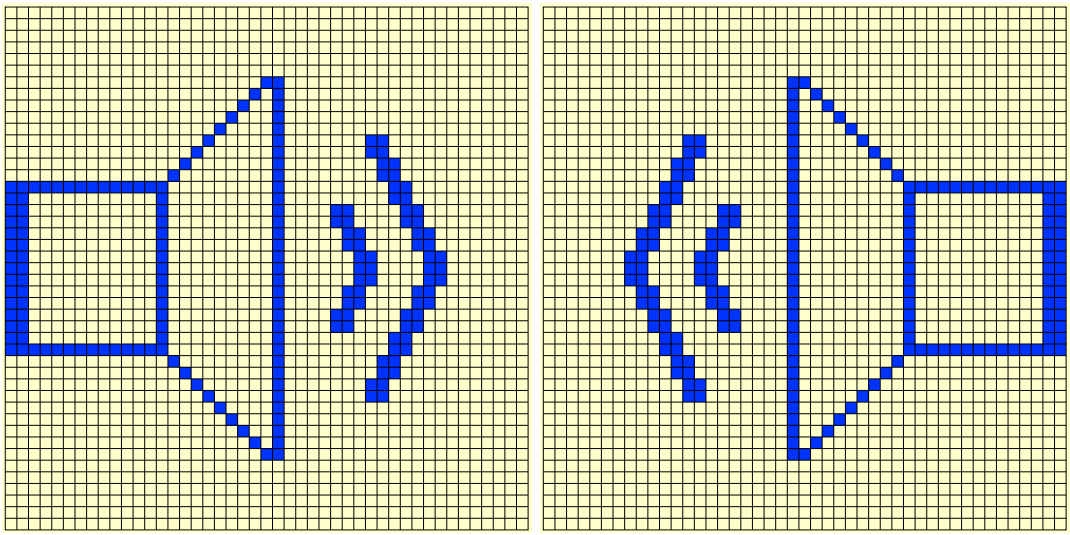
Mehrkörperproblem

Unter dem Mehrköperproblem (engl.: N-body problem) versteht man die Berechnung des Bahnverlaufs von n-Körpern unter dem Einfluss der Gravitation. Alle n-Körper ziehen sich gegenseitig bedingt durch ihre Masse an. Um den Bahnverlauf eines Körpers zu bestimmen, muss der Einfluss der Gravitation von allen anderen Körpern berechnet werden. Der Rechenaufwand steigt damit für viele Körper stark an. Für $n = 2$ spricht man auch vom Zweikörperproblem. Dieses ist durch die Keplerschen Gesetze lösbar. Für $n = 3$ spricht man auch vom Dreikörperproblem. Für $n > 3$ ist das Mehrkörperproblem nicht mehr mit elementaren Funktionen lösbar.

Der Beschleunigungsvektor (Kraftberechnung) [Mon05, Seite 81-94] ist gegeben durch:

$$\vec{a_i} = G\left(\sum_{j \neq i} \frac{m_j}{|r|^3}\vec{r}\right), r = \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2 + \left(z_j - z_i\right)^2}$$

Der Beschleunigungsvektor für das Mehrkörperproblem im GCA-Modell wurde unter anderem in [JHL09] umgesetzt. Dazu wurde eine spezielle datenparallele Architektur aus der Zellregelbeschreibung generiert, die dann auf einem FPGA ausgeführt wurde. Eine Beschreibung des



(a) Bild vor dem Spiegeln (b) Bild nach dem Spiegeln

Abbildung 2.4: Spiegeln einer Grafik im GCA-Modell

N-Body Problems und Simulation auf einer Grafikkarte ist in [NHP07] gegeben. Die Umsetzung erfolgt dabei aber nicht unter Verwendung des GCA-Modells, sondern wurde speziell für diese Simulation optimiert.

Bildverarbeitung

Unter den Begriff der Bildverarbeitung fallen verschiedene Algorithmen zur Extraktion oder Modifikation von Bilddaten. So können etwa Kanten in einem Bild durch Anwendung des Laplace-Filters

$$L(x,y) = \nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2}$$

extrahiert werden [Tho00]. Durch die Eigenschaften des Filters kann dieser sehr gut auf Globalen Zellularen Automaten umgesetzt werden. Der Laplace-Filter kann auch auf einem Zellularen Automaten umgesetzt werden, da für die Berechnung nur die direkte lokale Nachbarschaft benötigt wird. Ebenso ist die Umsetzung anderer Filter, z. B. zur Rauschunterdrückung, auf Zellularen Automaten gut umsetzbar [SH09]. Eine andere Anwendung in der Bildverarbeitung ist das Spiegeln eines Bildes (Abb. 2.4). Für eine derartige Bearbeitung des Bildes eignet sich der Globale Zellulare Automat am besten. Die Spiegelung des Bildes kann in nur einer Generation berechnet werden. Wird das Bild, so wie in Abbildung 2.4, an der vertikalen Achse (mittig zentriert) gespiegelt, ergibt sich der Nachbarzugriff durch die Entfernung der Zelle zur Achse. Der Zustand der Zelle repräsentiert dabei den Farbwert des Pixels[1]. Für die Spiegelung muss jede Zelle nur den entsprechenden Zellwert der gegenüberliegenden Zelle kopieren (gespiegelt an der Mittelachse).

[1] engl.: picture element

2.4 Einordnung von (Globalen) Zellularen Automaten

Michael J. Flynn hat 1966 eine Unterteilung von Rechnerarchitekturen [Fly72, Dun90] veröffentlicht. Die Rechnerarchitekturen werden dabei nach der Anzahl ihrer Befehle und der Anzahl an Daten, die parallel verarbeitet werden, eingeordnet. Die vier Möglichkeiten sind in Tabelle 2.1 aufgeführt.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Tabelle 2.1: Flynnsche Klassifikation [Fly72]

- SISD (**S**ingle **I**nstruction **S**ingle **D**ata):
  Ein Prozessor verarbeitet eine Instruktion und wendet diese auf ein Datum an (z. B. Von-Neumann-Architektur).
- SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata):
  Ein Prozessor verarbeitet eine Instruktion und wendet diese auf mehrere Daten an (Vektor- und Feldrechner).
- MISD (**M**ultiple **I**nstruction **S**ingle **D**ata):
  Mehrere Prozessoren verarbeiten mehrere Instruktionen und wenden diese auf ein Datum an.
- MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata):
  Mehrere Prozessoren verarbeiten mehrere Instruktionen und wenden diese auf mehrere Daten an (Multiprozessorsysteme). Die Kategorie wird noch feiner unterteilt in:
  - SPMD (**S**ingle **P**rogram, **M**ultiple **D**ata):
    Mehrere Prozessoren verarbeiten simultan das gleiche Programm und wenden dieses auf mehrere Daten an.
  - MPMD (**M**ultiple **P**rogram, **M**ultiple **D**ata):
    Mehrere Prozessoren verarbeiten simultan verschiedene Programme und wenden diese auf mehrere Daten an.

  Der größeren Portabilität der Programme bei gleichzeitig einfacherer Verwaltung in SPMD stehen die größere Flexibilität und ein größerer Verwaltungsaufwand in MPMD gegenüber.

Zellulare Automaten lassen sich anhand dieser Klassifikation in die Kategorie *Multiple Instruction Multiple Data (MIMD)* einordnen. In der feineren Unterteilung gehören sie der Kategorie *Single Program, Multiple Data (SPMD)* an. Alle Zellen berechnen parallel auf unterschiedlichen Daten (Zellen) ihren Nachfolgezustand.

2.5 Bewertung der Leistungsfähigkeit (Globaler) Zellularer Automaten

Für die Bewertung von Rechnersystemen und Rechnerarchitekturen sind objektive Bewertungskriterien notwendig. Doch auch objektive Bewertungskriterien ermöglichen nicht in allen Fällen einen fairen Vergleich. Eine Bewertung, die alle denkbaren Aspekte berücksichtigt, ist nicht vorhanden und wäre zudem unpraktikabel. Es haben sich jedoch einige einfachere Kriterien für die Bewertung etabliert. Ein wichtiger Aspekt ist dabei, welche Systeme bewertet werden sollen. Sollen zwei Systeme miteinander verglichen werden, so sind beide nach den gleichen Kriterien zu bewerten. Dabei ist darauf zu achten, dass das Bewertungskriterium beide Systeme in gleicher Weise berücksichtigt.

Für die in dieser Arbeit verwendeten Hardwarearchitekturen und Softwareimplementierungen werden die folgenden Kriterien für die Beurteilung der Leistungsfähigkeit verwendet:

- Taktfrequenz
- Millionen Instruktionen pro Sekunde (engl.: Million Operations per Second, MIPS)
- Speedup
- Ausführungszeit
- Cell-Update-Rate (CUR)
- Generation-Update-Rate (GUR)
- Agent-Update-Rate (AUR)

2.5.1 Definition des Speedups

Der Speedup eines Algorithmus auf einer Multiprozessorarchitektur wird im Vergleich zur sequenziellen Implementierung auf einer Einprozessorarchitektur angegeben [Hee07, Seite 28]. Die Laufzeit der sequenziellen Variante ist mit $T_s(N)$ angegeben. Die Laufzeit der parallelen Variante mit p Prozessoren ist mit $T_p(N)$ angegeben. Damit ergibt sich der Speedup durch:

$$S(N) = \frac{T_s(N)}{T_p(N)}$$

Im allgemeinen gilt $S < p$. Für $S = p$ spricht man auch von linearem Speedup.

Für die Bewertung einer Architektur mit unterschiedlicher Prozessoranzahl wird der Speedup als Leistungsindex herangezogen. Mit $t(N,p)$ wird die Zeit für die Ausführung auf einer Architektur mit p Prozessoren für eine Problemgröße N angegeben. Der Speedup gibt den Leistungsunterschied als Faktor an:

$$Speedup(N,p) = \frac{t(N,1)}{t(N,p)}$$

Der Speedup dient also der Bewertung einer Architektur mit unterschiedlicher Prozessoranzahl. Es wird damit die Leistungsfähigkeit in Abhängigkeit von der Prozessoranzahl angegeben. Allgemein erwartet man mit einer Erhöhung der Prozessoranzahl auch eine Leistungssteigerung.

Der Speedup beinhaltet zum einen die Anzahl an Taktzyklen, die für die Ausführung der Implementierung benötigt wird, zum anderen aber auch die Taktfrequenz der Hardwarearchitektur. Der Speedup kann aber auch bezüglich der Taktzyklen angegeben werden. Da für die Bewertung die Anzahl der Taktzyklen sowie die Taktfrequenz bedeutend ist, werden beide für den Speedup herangezogen. Dieser Speedup wird zur Unterscheidung auch als *wahrer Speedup* bezeichnet. Der Speedup, der nur die Taktzyklen berücksichtigt, wird als *Taktzyklen-Speedup* bezeichnet. Wird lediglich von Speedup gesprochen, so ist immer der wahre Speedup gemeint.

2.5.2 Definition der Generation-Update-Rate

Die Generation-Update-Rate (GUR) gibt die Anzahl an Generationen pro Zeiteinheit an. Diese Maßeinheit kann für alle Hardwarearchitekturen verwendet werden, da alle Hardwarearchitekturen den Wechsel der Generationen synchronisieren müssen. Die Prozessoranzahl fließt nicht direkt in die Berechnung ein, ist aber indirekt in der Zeit $t\left(n,p\right)$ enthalten.

Die Generation-Update-Rate für p Prozessoren und n Zellen wird wie folgt berechnet:

$$GUR\left(n,p\right) = \frac{g}{t\left(n,p\right)}$$

2.5.3 Definition der Cell-Update-Rate

Die Cell-Update-Rate (CUR) ist eine Maßeinheit, die die Anzahl von Zellupdates pro Zeiteinheit angibt. Sie ist unabhängig von der Anzahl der ausgeführten Generationen. Die Größe (Dimension) des Zellfeldes $n = X \cdot Y$ fließt in die Berechnung ein. Hardwarearchitekturen, die eine Generation unabhängig von der Größe des Zellfeldes berechnen, werden von der Cell-Update-Rate nicht korrekt berücksichtigt.

Die Cell-Update-Rate für p Prozessoren und n Zellen wird wie folgt berechnet:

$$CUR\left(n,p\right) = \frac{g \cdot n}{t\left(n,p\right)} = GUR\left(n,p\right) \cdot n$$

2.5.4 Definition der Agent-Update-Rate

Die Agent-Update-Rate (AUR) ist eine Maßeinheit ähnlich der Cell-Update-Rate. Die Agent-Update-Rate gibt die Anzahl der Agenten an, die pro Zeiteinheit berechnet werden. Mit der Agent-Update-Rate können alle Hardwarearchitekturen bewertet und verglichen werden. Die Agent-Update-Rate bezieht sich generell auf alle Agenten a_g und besteht aus den beweglichen Agenten a_m und den Hindernissen h: $a_g = a_m + h$.

Die Agent-Update-Rate für p Prozessoren und $a_g \leq n$ Agenten wird durch die folgende Formel berechnet:

$$AUR\left(a_g, p\right) = \frac{g \cdot a_g}{t\left(a_g, p\right)} = GUR\left(a_g, p\right) \cdot a_g$$

2.6 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Modelle, auf der diese Arbeit basiert, definiert. Zunächst wurde die geschichtliche Entwicklung des Modells des Zellularen Automaten vorgestellt. Im Anschluss daran wurde der Zellulare Automat definiert. Dies war u. a. notwendig, da in der Literatur unterschiedliche Definitionen verwendet werden, die sich aus dem geschichtlichen Verlauf ergeben haben. Es wurde danach auf die unterschiedlichen Nachbarschaften in Zellularen Automaten eingegangen. Bevor auf konkrete Anwendungen für das Modell des Zellularen Automaten eingegangen wurde, wurden Einschränkungen bezüglich praktischer Anwendungen behandelt. Im Anschluss daran wurde das Modell des Globalen Zellularen Automaten als Erweiterung zum Modell des Zellularen Automaten vorgestellt. Das Modell des Globalen Zellularen Automaten bildet die Grundlage für die in Kapitel 6 entwickelten Hardwarearchitekturen. Auch für das Modell des Globalen Zellularen Automaten wurden Beispielanwendungen angegeben. Abschließend wurden die (Globalen) Zellularen Automaten klassifiziert und für die spätere Bewertung der Leistungsfähigkeit der Hardwarearchitekturen Bewertungskriterien definiert und erarbeitet.

3 Struktur und Aufbau eines Multi-Agenten-Systems

Diese Kapitel behandelt und definiert die Eigenschaften von Agenten. Abhängig von der Disziplin, in der Agenten eingesetzt werden, besitzen diese unterschiedlichste Eigenschaften und Fähigkeiten. Es wird deshalb zuerst ein Überblick über die in der Literatur genannten Eigenschaften gegeben. Es fällt dabei auf, dass den Agenten im Laufe der Zeit immer mehr Eigenschaften und Fähigkeiten zugesprochen werden. Somit sind allgemein die Komplexität sowie die Ansprüche an einen Agenten gewachsen. Allerdings besteht keine Einigkeit über die generellen oder minimalen Eigenschaften und Funktionen, die ein Agent aufweisen sollte. Diese Arbeit erhebt nicht den Anspruch eine allgemeine oder minimale Definition dieser Eigenschaften aufzustellen. Es werden die in der Literatur üblichen Eigenschaften und Fähigkeiten von Agenten verwendet, die für diese Arbeit von Bedeutung sind.
Der Begriff des Multi-Agenten-Systems, der für diese Arbeit von zentraler Bedeutung ist, wird im Anschluss an den Begriff des Agenten definiert. Zu dem Multi-Agenten-System gehören neben dem Agentenbegriff noch weitere, wie z. B. der Begriff der (Agenten-)Umwelt.
Der Aufbau des Simulationssystems wird diskutiert und der Zusammenhang sowie die Umsetzung des GCA-Modells und damit einhergehende Probleme bzw. wichtige Aspekte aufgezeigt.

3.1 Charakteristische Eigenschaften von Agenten

Um verstehen zu können, wie ein Multi-Agenten-System verwendet werden kann, um Probleme zu lösen, muss zunächst der Begriff des Agenten definiert werden. Dies gestaltet sich jedoch als nicht triviale Aufgabe. Dies liegt daran, dass der Begriff des Agenten interdisziplinär verwendet wird. Doch auch innerhalb der Informatik herrscht keine einheitliche Definition über den Agentenbegriff. Je nach Anforderung werden so einem Agenten unterschiedliche Eigenschaften und Fähigkeiten zugesprochen. Auch gibt es keine gemeinsame Basis an Eigenschaften, die jeder Definition zu Grunde liegen. Einzig die Eigenschaft der Autonomität scheint allen Definitionen gleich zu sein [Woo02, Seite 15]. James Ingham stellt in seinem Beitrag „What is an Agent?“ [Ing97] verschiedene Definitionen gegenüber. Auch er kann keine einheitliche Definition liefern, versucht aber ein gemeinsames Vokabular aufzustellen. Einleitend sei eine informelle und allgemeine Definition, übernommen und angepasst aus [WJ95a], entnommen aus Wooldridge und Jennings [Woo02, Seite 15] gegeben.

> *„An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.“ [Woo02, Seite 15]*

Dieser Definition ist zu entnehmen, dass es sich bei einem Agenten um ein eigenständiges, autonomes System handelt. Der Agent handelt also von sich aus selbständig, ohne Intervention von außen. Des weiteren befindet sich der Agent in einer Umgebung, über dessen genaue Gestaltung

keine weitere Aussage getroffen wird. Laut der Definition verhält der Agent sich dabei so, dass er die ihm vorgegebenen Ziele verfolgt und erreicht.
Auf eine ähnliche Weise wird der Agent auch von Franziska Klügl definiert [Klü06]:

> *„Im Prinzip ist ein Agent eine Einheit ..., die sich in einer Umwelt befindet und in der Lage ist, in dieser autonome Aktionen durchzuführen, um ihre Ziele zu erreichen." [Klü06, Seite 412]*

In einer formaleren Definition unterscheiden Wooldridge und Jennings zwei Begriffsklassen, die „schwache" und die „starke" Begriffsklasse [WJ95b]. Die schwache Notation beinhaltet dabei die folgenden Eigenschaften [WJ95b, Seite 116]:

- **Autonomität (engl.: autonomy):** Agenten agieren autonom, d. h. ohne Eingriff von anderen Systemen. Sie haben Kontrolle über ihre Aktionen und ihren internen Zustand. [Cas95]
- **Soziale Fähigkeiten (engl.: social ability):** Agenten interagieren mit anderen Agenten. [GK94]
- **Reaktivität (engl.: reactivity):** Agenten nehmen ihre Umgebung wahr und reagieren in einer angemessenen Zeit auf diese Änderungen.
- **Pro-Aktivität (engl.: pro-activeness):** Agenten reagieren nicht nur auf ihre Umwelt, sie weisen zielgerichtetes Verhalten auf und ergreifen die Initiative.

Diese formalere Definition beinhaltet die schon erwähnte Eigenschaft der Autonomität. Ab wann ein Agent autonom handelt, ist in der Literatur umstritten. Teilweise wird ein Agent als autonom angesehen, wenn er ohne Intervention von außen, eigenständig handelt [Cas95]. Andere sehen einen Agenten erst als autonom an, wenn er auf Grund seiner Erfahrung Entscheidungen selbständig trifft [RN04]. Die weiteren Eigenschaften betreffen soziale Fähigkeiten, Reaktivität und Pro-Aktivität. Die sozialen Fähigkeiten besagen, dass ein Agent mit einem anderen Agenten kommunizieren kann. Diese Kommunikation kann wiederum Einfluss auf das Verhalten des Agenten nehmen. Die Reaktivität besagt, dass der Agent seine Umwelt erkennen und sein Verhalten entsprechend anpassen kann. Das Verhalten eines Agenten ist demnach nicht fest vorgegeben, sondern wird durch die Umwelt (z. B. andere Agenten) beeinflusst. Unter der Pro-Aktivität versteht man, dass ein Agent ein bestimmtes Ziel verfolgt. Sein Verhalten wird nicht nur von den Umwelteinflüssen geprägt, sondern auch durch sein zielgerichtetes Handeln beinflusst.

Die starke Notation erweitert diese Eigenschaften mit mentalen Eigenschaften, wie z. B. Wissen, Glaube und Intention. Eine formale Definition hierfür ist nicht gegeben. In [Bat94, BLR94] werden den Agenten sogar emotionale Eigenschaften zugesprochen.

Wooldridge und Jennings führen in [WJ95b, Seite 117] und [WJ95a] noch weitere Eigenschaften von Agenten auf:

- **Mobilität (engl.: mobility):** Agenten können sich von einem System zu einem anderen System bewegen [Mil99].
- **Aufrichtigkeit (engl.: veracity):** Ist die Annahme, dass ein Agent nicht wissentlich falsche Informationen kommuniziert.

- **Wohlwollen (engl.: benevolence):** Ist die Annahme, dass Agenten keine widersprüchlichen Ziele verfolgen. [RG85].
- **Vernunft (engl.: rationality):** Ein Agent agiert so, dass er seine Ziele erreicht. Er verhält sich nicht so, dass die Erreichung der Ziele verhindert wird.

Unter der Eigenschaft der Mobilität wird verstanden, dass ein Agent nicht örtlich gebunden ist. Er kann sich in der Umwelt bewegen. Die Aufrichtigkeit besagt, dass ein Agent nur korrekte Informationen an andere Agenten weitergibt. Er versucht demnach nicht, sich durch falsche Kommunikation Vorteile gegenüber anderen Agenten zu verschaffen. Wohlwollen besagt, dass Agenten keine widersprüchlichen Ziele verfolgen. Jeder Agent versucht demnach immer, seine Aufgabe zu erfüllen. Der letzte Punkt, die Eigenschaft der Vernunft, ist die Annahme, dass ein Agent sich so verhält, dass er sein Ziel erreicht. Er wird nicht versuchen, zu verhindern, dass das Ziel erreicht wird. Diese Eigenschaft kann durch andere Eigenschaften eingeschränkt sein, z. B. durch das „Wissen“ des Agenten.

Richard Goodwin schlägt zwei Agentenkategorien vor [Goo93]. Insgesamt spricht er den Agenten die folgenden Eigenschaften zu [Goo93, Seite 3]:

- **erfolgreich (engl.: successful):** Wenn er eine vorgegebene Aufgabe in einer vorgegebenen Umgebung löst.
- **geeignet (engl.: capable):** Wenn er die Effektoren besitzt, um die Aufgabe zu erfüllen.
- **wahrnehmend (engl.: perceptive):** Wenn er die Charakteristiken der Welt unterscheiden kann, die ihm erlauben, die Effektoren einzusetzen, um die Aufgabe zu erfüllen.
- **reaktiv (engl.: reactive):** Wenn er ausreichend schnell auf Ereignisse in der Welt reagiert.
- **reflexiv (engl.: reflexive):** Wenn er sich nach dem „stimulus-response“-Konzept verhält.
- **voraussagend (engl.: predictive):** Wenn sein Modell von der Welt genau genug ist, um Vorhersagen darüber zu treffen, wie er sein Ziel erreichen kann.
- **interpretierend (engl.: interpretive):** Wenn er seine Sensordaten korrekt interpretieren kann.
- **rational (engl.: rational):** Wenn er vorhergesagte Befehle ausführt, die zielorientiert sind.
- **gut fundiert (engl.: sound):** Wenn er vorraussagend, interpretierend und rational ist.

Die Eigenschaft der Reaktivität und Rationalität überdecken sich mit den vorher definierten Eigenschaften. Richard Goodwin definiert weitere Eigenschaften für einen Agenten, die in der Literatur sonst nicht verbreitet sind.

3.1.1 Übersicht über die Eigenschaften von Agenten

Im Folgenden wird zusammenfassend eine Übersicht über die Eigenschaften von Agenten gegeben (Tabelle 3.1). Dazu werden die Eigenschaften auf Grund ihrer Häufigkeit in drei Kategorien unterteilt: Die Basiseigenschaften, erweiterten Eigenschaften und zusätzliche Eigenschaften. Die Basiseigenschaften sind die grundlegenden Eigenschaften, die am häufigsten im Kontext von Multi-Agenten-Systemen vorkommen. Die erweiterten Eigenschaften beschreiben oft genutzte Eigenschaften und die zusätzlichen Eigenschaften stellen selten verwendete Eigenschaften dar.

Basiseigenschaften	Literatur
reactivity	[Woo02, WJ95b, Goo93, WJ95a, Wei99, Klü01]
pro-activeness	[Woo02, WJ95b, WJ95a, Wei99]
social ability	[Woo02, WJ95b, GK94, Wei99, Klü01]
autonomy	[WJ95b, Cas95, WJ95a, Wei99, Klü01]
Erweiterte Eigenschaften	**Literatur**
mobility	[WJ95b, Mil99, Ing97]
veracity	[WJ95b, Ing97]
benevolence	[WJ95b, RG85, Ing97]
rationality	[WJ95b, Goo93, Ing97, Klü01]
Zusätzliche Eigenschaften	**Literatur**
successful	[Goo93]
capable	[Goo93]
perceptive	[Goo93]
reflexive	[Goo93]
predictive	[Goo93]
interpretive	[Goo93]
sound	[Goo93]

Tabelle 3.1: Übersicht über die Eigenschaften von Agenten

3.2 Definition des Agenten

Nach dem Überblick über die möglichen und in der Literatur genannten Eigenschaften ist die Verwendung des Begriffs des Agenten für diese Arbeit zu erläutern. Auf Grund einer fehlenden, allgemein anerkannten Agentendefinition (die u. U. gar nicht erarbeitet werden kann) und den vielen unterschiedlichen Eigenschaften, die Agenten zugesprochen werden können, ist es notwendig, diese im Kontext dieser Arbeit einzugrenzen.

Die Multi-Agenten-Simulation dient der Simulation von unterschiedlichsten Multi-Agenten-Anwendungen. Abhängig von der zu simulierenden Anwendung werden daher auch unterschiedlichste Eigenschaften benötigt. Für die unterschiedlichen Anwendungen können deshalb jeweils nur die benötigten Eigenschaften umgesetzt werden. Das Multi-Agenten-Simulations-System muss jedoch die Möglichkeit bieten, alle vorher definierten Eigenschaften umsetzen zu können. Die Wahl der Eigenschaften ist aber auch von dem zu simulierenden Detailgrad abhängig.

So wird z. B. bei der Verkehrssimulation auf Autobahnen ein gröberer Detailgrad als bei der Verkehrssimulation beim Stadtverkehr verwendet. Zwar kann beides mit dem gleichen Detailgrad simuliert werden, was aber nicht zwangsläufig zu einer Verbesserung der Simulationsergebnisse führen muss und u. U. nur die Simulationszeit erhöht.

Die Eigenschaften der hier verwendeten Agenten sind deshalb letztendlich nur von der Architektur und den Erfordernissen der Anwendung abhängig.

3.3 Die Agentenumwelt

Die Agentenumwelt (kurz Umwelt) definiert die Umgebung, in der der Agent agiert. Stuart Russell und Peter Norvig haben die wichtigsten Eigenschaften für diese Umwelt herausgearbeitet [RN04] und auch in [Woo02] aufgegriffen:

- **Vollständig beobachtbar im Vergleich zu teilweise beobachtbar:** Die Umgebung ist vollständig beobachtbar, wenn der Agent zu jedem beliebigen Zeitpunkt Zugriff auf den vollständigen Zustand der Umgebung hat.
- **Deterministisch im Vergleich zu stochastisch:** Die Umgebung ist deterministisch, wenn sie vollständig aus dem aktuellen Zustand und vollständig aus den ausgeführten Aktionen der Agenten festgelegt ist. Anderenfalls ist die Umgebung als stochastisch anzusehen. Ist die Umgebung bis auf die Aktionen anderer Agenten deterministisch, wird die Umgebung als strategisch bezeichnet.
- **Episodisch im Vergleich zu sequenziell:** In einer episodischen Umgebung sind die Erfahrungen der Agenten in sogenannte Episoden unterteilt. Jede Episode ist als unabhängiger Teil zu betrachten. Eine Entscheidung in einer Episode hat keinen Einfluss auf die Entscheidungen in einer anderen Episode. In sequenziellen Umgebungen können alle aktuellen Entscheidungen zukünftige Entscheidungen beeinflussen.
- **Statisch im Vergleich zu dynamisch:** Ändert sich die Umgebung während ein Agent eine Entscheidung trifft, so ist die Umgebung dynamisch. In allen anderen Fällen ist die Umgebung statisch.
- **Diskret im Vergleich zu stetig:** Diese Unterscheidung betrifft den Zustand der Umgebung, die Zeit, die Wahrnehmung und die Aktionen des Agenten.
- **Einzelagent im Vergleich zu Multiagent:** Einzelagent- und Multiagentumgebungen unterscheiden sich erst einmal in der Anzahl der Agenten. Die komplexere Fragestellung besteht darin, die Agenten in dem System zu erkennen und zu definieren. Unter Umständen besteht eine gleichwertige, einfachere Umgebung mit weniger Agenten.

Wichtig ist, zu unterscheiden, aus welcher Perspektive die Umwelt betrachtet wird. Die Umwelt kann als Gesamtheit betrachtet werden. Damit beinhaltet sie insbesondere auch alle Agenten. Im Gegensatz dazu kann die Umwelt auch aus der Sicht eines Agenten betrachtet werden. Ein Agent hat nicht zwangsläufig Zugriff auf die komplette Umwelt. Vielmehr agiert und reagiert er nur mit einem Teil der Umwelt.

3.4 Aufbau eines Multi-Agenten-Systems

Das Multi-Agenten-System (MAS) bildet die Gesamtheit des zu simulierenden Systems. Dazu zählen einerseits alle Objekte (u. a. Agenten), die Umwelt, die notwendigen Eigenschaften und Parameter, aber auch das konkrete Verhalten der Agenten.

Definition 3.1 (Multi-Agenten-System) *Ein Multi-Agenten-System (MAS) beschreibt die Gesamtheit eines zu simulierenden Systems, alle Objekte, deren Umwelt, Eigenschaften und Parameter sowie das Verhalten der Agenten.*

Multi-Agenten-Systeme bestehen aus einer Population von Agenten. Die Agenten befinden sich in einer definierten Umwelt in der sie einzelne oder globale übergeordnete Ziele verfolgen (Abb. 3.1). Die Eigenschaften der Agenten sind vorab festzulegen. Ein MAS ist stets vollständig zu definieren. Dabei wird ein MAS als *vollständig* definiert angesehen, wenn alle Bedingungen für eine zielgerichtete Simulation erfüllt sind.

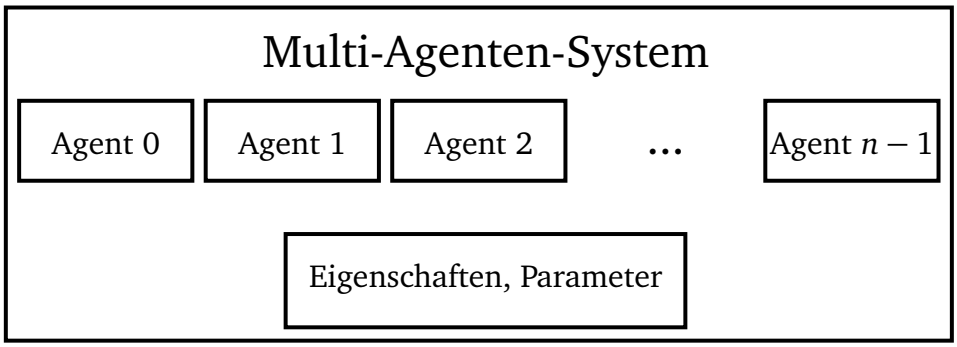


Abbildung 3.1: Übersicht über das Multi-Agenten-System

Das Multi-Agenten-System besteht aus den folgenden Komponenten:

- **Agent:** Die Agenten bilden die Grundlage des Multi-Agenten-Systems.
  - **Agententypen:** Definiert die unterschiedlichen Typen von Agenten für die Simulation von inhomogenen MAS.
  - **Agentenverhalten:** Bezeichnet das konkrete Verhalten eines einzelnen Agenten.
- **Agentenumwelt:** Bezeichnet aus der globalen Perspektive die gesamte Umwelt und beinhaltet damit insbesondere die Agenten.
- **Multi-Agenten-Anwendung:** Beschreibt das Verhalten aller Agenten im Multi-Agenten-System. In bestimmten Fällen kann die Multi-Agenten-Anwendung identisch mit dem Agentenverhalten sein.
- **Eigenschaften / Parameter:** Anzahl der Agenten, Größe der Umwelt, Simulationszeit (Generationen) usw.

Aus der Sichtweise des MAS sind alle Objekte als Agenten anzusehen. Somit sind im Speziellen auch Hindernisse als Agenten zu betrachten. Betrachtet man die gleiche Agentenwelt aus der Perspektive der Anwendung, würde man Hindernisse nicht als Agenten zählen, sondern nur die sich bewegenden und agierenden Objekte als Agenten auffassen.

Die Durchführung der Simulation des MAS wird als Multi-Agenten-Simulation bezeichnet. Dazu werden alle Agenten in diskreten Zeitschritten bewegt. Im ersten Schritt berechnet jeder Agent seine neue Position. Hierbei werden evtl. auftretende Konflikte aufgelöst. Nachdem die neue Position aller Agenten bekannt ist, werden diese quasi parallel auf ihre neue Position bewegt.

Definition 3.2 (Multi-Agenten-Simulation) *Die Durchführung der Simulation des MAS in diskreten Zeitschritten.*

3.5 Prinzipieller Aufbau des Multi-Agenten-Simulations-Systems

Für die Durchführung der Multi-Agenten-Simulation kommt ein Multi-Agenten-Simulations-System zum Einsatz. Dieses stellt die erforderlichen Ressourcen für die Simulation bereit. Hierzu können u. a. auch für die Simulation angepasste und spezialisierte Erweiterungen zum Einsatz kommen.

Definition 3.3 (Multi-Agenten-Simulations-System) *System, das für die Durchführung der Multi-Agenten-Simulation benötigt wird.*

Die Multi-Agenten-Simulation kann auf verschiedensten Hardwarearchitekturen ausgeführt werden. Für das GCA-Modell wurden bereits unterschiedlichste Architekturen entwickelt [HHJ06, JEH08b, Hee07], allerdings nicht mit dem Ziel, die Multi-Agenten-Simulation effizient zu unterstützen. Mit dem Fokus der Multi-Agenten-Simulation können die Architekturen erweitert und/ oder begrenzt werden. Dadurch sind die resultierenden Architekturen nicht mehr zwangsläufig in der Lage, alle GCA-Anwendungen auszuführen. Dafür erreicht man für die Simulation von MAS eine höhere Performance. Da die Architekturen für viele verschiedene umfangreiche Simulationen (Kapitel 4) verwendet werden können, ist es sinnvoll, eine gemeinsame Schnittstelle einzuführen. Als Schnittstelle dienen in dieser Arbeit Custom Instructions (CI) (Abschnitt 6.5.1.1). Die CIs erweitern den Befehlssatz der in den Hardwarearchitekturen verwendeten Prozessoren. So können zum einen das GCA-Modell als auch Multi-Agenten-Systeme realisiert werden. Eine weitere Möglichkeit besteht darin, die *Global Cellular Automata Experimental Language* (GCA-L) [JEH08a] als Schnittstelle einzusetzen. Ausgehend von einer Zellregel in GCA-L können entweder direkt Hardwarearchitekturen oder Zellregeln für Multiprozessorarchitekturen erstellt werden. Die Umsetzung auf die korrekten CIs wird dann durch die Übersetzung von GCA-L auf C-Code mit CIs vorgenommen. Über die Schnittstelle kann dann das Multi-Agenten-System verwendet werden. Das Multi-Agenten-System definiert dabei, wie die Agentenwelt zu simulieren ist. Es verwendet dazu das GCA-Modell oder eine modifizierte Version davon. Über das Multi-Agenten-System kann das GCA-Modell erweitert oder eingeschränkt werden und es wird eine Abbildung des Agentenverhaltens auf die Zellregel vorgenommen. Zusätzlich können oft verwendete Funktionen in Hardware oder in Software hinzugefügt werden. Abbildung 3.2 zeigt die Hierarchie der unterschiedlichen Abstraktionslevel von der Anwendungsebene über die Modellebene bis zur Architekturebene.

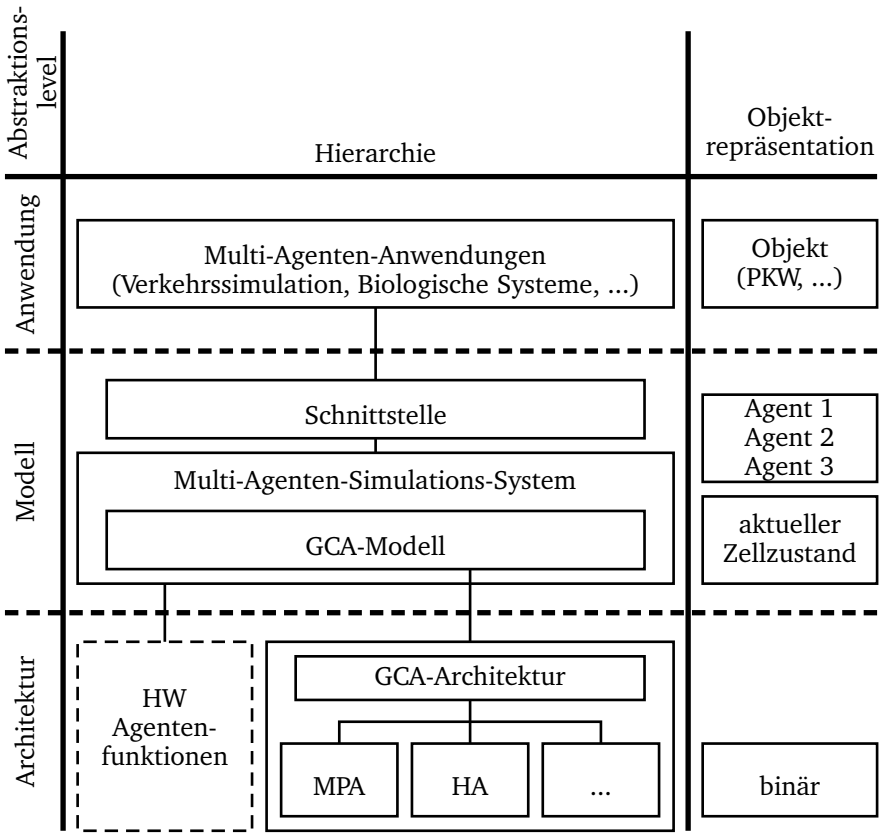


Abbildung 3.2: Anwendungs-, Modell- und Architekturhierarchie

Auf der Anwendungsebene wird die zu simulierende Anwendung, d. h. Agentenwelt, umgesetzt. Hierzu zählt neben den Agenten hauptsächlich deren Verhalten. Zur besseren Verständlichkeit und einfacheren Veranschaulichung können die Agenten durch die konkrete Objektrepräsentation ersetzt werden. So kann z. B. für die Umsetzung einer Verkehrssimulation direkt die Objektrepräsentation der Fahrzeuge (PKW, LKW) verwendet werden, was die Implementierung erleichtert. Über die Modellebene werden die einzelnen Fahrzeuge den Agenten zugeordnet. Da der Simulation das GCA-Modell zugrunde liegt, werden die Agenten für das Modell als Zellzustände interpretiert. Auf der Architekturebene repräsentieren schließlich binäre Werte die Zellzustände. Die GCA-Architektur und evtl. ergänzte Hardware Agentenfunktionen führen dann die Simulation aus.

Die Modellierung auf der Anwendungsebene und das Verständnis über die Agenten als aktive Einheiten erleichtert die Umsetzung des Agentenverhaltens. Zellzustände repräsentieren die aktiven Eigenschaften von Agenten weniger anschaulich. Die Schnittstelle sorgt dafür, dass unabhängig von der verwendeten Architektur möglichst einheitliche Funktionen zur Verfügung stehen. Dazu sind evtl. vorhandene Hardwarefunktionen einer Architektur auf einer anderen Architektur in Software zu simulieren. Darüber hinaus kann z. B. der Speicherbereich für die Zellen so weit verkleinert werden, dass nicht mehr alle Zellen gespeichert werden können. Da die Art der Verwaltung der Zellen für die Implementierung einer Anwendung unerheblich ist, kann auch hiervon abstrahiert werden.

Der Einsatz des GCA-Modells für die Simulation von MAS eignet sich aus verschiedenen Gründen. Für das GCA-Modell bzw. auf dessen Basis können verschiedene Architekturen entworfen werden, die alle die entsprechenden Zellregeln ausführen können. Der Grad der Parallelität ergibt sich direkt aus der verwendeten Hardware. Die Zellregel muss dafür nicht angepasst werden. Je nach Implementierung ist nur die Anpassung weniger Parameter notwendig. Für die Simulation von MAS hat dies die gleichen Vorteile. Es wird lediglich die Zellregel für einen Agenten aus dessen Sicht umgesetzt. Die Anwendung auf alle Agenten und das damit u. U. entstehende komplexe Verhalten sowie die parallele Ausführung ergibt sich durch die Hardwarearchitektur. Es ist keine explizite Behandlung der Parallelität notwendig. Nur die Generationssynchronisation muss an der entsprechenden Stelle in der Zellregel vorgenommen werden. Durch die Verwendung der Hierarchie können zusätzliche Funktionen in Software sowie in Hardware bereitgestellt werden. Ebenso kann das GCA-Modell dadurch angepasst oder erweitert werden. Durch eine gemeinsame Schnittstelle werden diese Änderungen vom Programmierer verdeckt. So können z. B. Architekturen entworfen werden, die nicht alle Zellen der Agentenwelt abspeichern können. Dies hat aber keine Auswirkungen auf das Simulationsergebnis und bleibt über die Schnittstelle u. U. ganz verborgen. Dadurch erreicht man eine höhere Flexibilität und Effizienz für die Hardwarearchitekturen und auch die Multi-Agenten-Simulation.

3.5.1 Unterstützung aktiver Einheiten im GCA-Modell

Im GCA-Modell werden die einzelnen Zellen in Generationen von einem Zustand in einen Nachfolgezustand überführt. In diese Überführungsfunktion oder Berechnung fließen Zustandsinformationen von ausgewählten Zellen ein. Somit berechnet jede Zelle für sich ihren Folgezustand. Für die Unterstützung aktiver Einheiten (Agenten) im GCA-Modell ist es unabdingbar, dass zwei Zellen zusammenarbeiten. Bewegt sich ein Agent von einer Zelle zur Nachbarzelle, muss eine Kollaboration zwischen diesen zwei Zellen bestehen. Auf Grund des fehlenden Schreibzugriffs auf Nachbarzellen kann der Agent sich nicht selbst auf die Nachbarzelle bewegen. Diese Bewegung muss durch einen „copy&move“-Vorgang umgesetzt werden.

> *„In a cellular automaton, to transport a particle from here to there one has to make a copy of it there and erase it here. Though performed at different places, these two actions must take place as the two halves of an indivisible operation - lest particles multiply or vanish - and thus must be carefully coordinated, ...“ [TM87, Seite 102]*

Die Zelle, auf der der Agent aktuell steht, löscht den Agenten. Die freie Nachbarzelle kopiert den Agenten. Eine Agentenbewegung setzt also immer voraus, dass eine Zelle den Agenten löscht und gleichzeitig eine andere Zelle diesen Agenten kopiert. Die Kollaboration bezieht sich einerseits auf eine leere Zelle und eine Agentenzelle andererseits aber auch auf die gesamte Nachbarschaft der leeren Zelle. Ist für eine Anwendung ausgeschlossen, dass sich zwei Agenten auf einer Zelle befinden können, muss in bestimmten Situationen eine Auswahl getroffen werden. Haben zwei Agenten die gleiche leere Zelle als Zielzelle bestimmt, bestehen zwei Möglichkeiten. Entweder bewegt sich einer der Agenten auf die Zielzelle oder aber keiner der Agenten bewegt sich auf die Zielzelle. Unabhängig von der Selektion des Agenten besteht eine Kollaboration zwischen diesen drei Zellen.

Der „copy&move“-Vorgang beschränkt die Handlungsweise eines Agenten in keiner Weise. Der Agent agiert immer noch selbständig. Das GCA-Modell erfordert lediglich eine andere Umsetzung der aktiven Eigenschaft von Agenten.

In der Arbeit von [CW06] wird auf die Verwendung von Zellularen Automaten für die Simulation von Agenten (aktiven Einheiten) im Speziellen für die Verkehrssimulation eingegangen. Die Autoren merken an, dass das Modell des Zellularen Automaten in vielen Arbeiten nicht im ursprünglichen, strikten Sinn eingehalten wird. Die Anwendung findet dann nicht auf den Zellen statt, sondern auf den Elementen, die sich auf dem Zellfeld befinden. Aus diesem Grund wird teilweise in der Literatur eine Unterscheidung in zwei Ebenen vorgeschlagen. Eine Ebene für Zellulare Automaten (räumlich verteilte Elemente) und eine Ebene für Multi-Agenten-Systeme (bewegliche Elemente). Die Arbeitsweise von Zellularen Automaten wird für natürliche kollektive Systeme als nachteilig beschrieben, da in diesen immer alle Zellen synchron berechnet werden. Realistischere Ergebnisse seien mit einem asynchronen oder zufälligen Berechnungsschema zu erreichen. Das Agentenverhalten in einem MAS ist direkt mit den Agenten verbunden und kann in eine Zellregel für Zellulare Automaten überführt werden. In [CW06] wird diese Überführung ebenfalls als nachteilig beschrieben, da die Regel des Zellularen Automaten komplexer sei, als die des Agenten eines MAS. Dies wird hauptsächlich mit der begrenzten Nachbarschaft im Zellularen Automaten begründet. Aus diesem Grund bietet sich dazu das GCA-Modell an.

Abschließend stellen die Autoren von [CW06] fest, dass sich Zellulare Automaten nicht für die Simulation von beweglichen Elementen eignen und für eine realistischere Simulation asynchrone Zustandsübergänge verwendet werden sollten. Diese Feststellungen beziehen sich zwar nur auf Zellulare Automaten und die Auswirkungen einer größeren Nachbarschaft, können aber leicht behoben werden. Das Problem der begrenzten Nachbarschaft kann durch die Verwendung des Globalen Zellularen Automaten gelöst werden. Die synchronen Eigenschaften Zellularer Automaten können durch die Verwendung von Zufallszahlen aufgehoben werden. Dem in [CW06] gezogenen Fazit kann also nicht zugestimmt werden.

3.5.2 Zufallszahlen im GCA-Modell

Die Umsetzung von Zufallszahlen im GCA-Modell erfordert eine gesonderte Betrachtung, insbesondere für die Multi-Agenten-Simulation. Wie in Abschnitt 3.5.1 beschrieben, wird die Bewegung eines Agenten auf dem zellularen Feld durch einen „copy&move“-Vorgang umgesetzt. Agenten mit komplexem Verhalten unter der Verwendung von Zufallszahlen können nicht ohne weiteres im GCA-Modell simuliert werden. Der „copy&move“-Vorgang erfordert das Zusammenspiel von zwei Zellen zur gleichen Zeit[1]. Damit dieses Zusammenspiel möglich ist, müssen sich beide Zellen auf eine Zufallszahl einigen. Die Generierung einer Zufallszahl für jede Zelle während der Abarbeitung der aktuellen Generation ist nicht möglich. Die Zufallszahl stellt einen Teil des Zustandes der Zelle dar. Somit bedeutet die Generierung einer Zufallszahl während der Abarbeitung der Zellregel eine Veränderung des Zellzustandes. Lesezugriffe von anderen Zellen würden somit, abhängig vom Zeitpunkt des Zugriffes, unterschiedliche Zellzustände erhalten. Alternativ kann man global eine Zufallszahl für alle Zellen bereitstellen. Die Bereitstellung ei-

[1] Genaugenommen steht die gesamte Nachbarschaft einer Zelle mit der durchgeführten Aktion im Zusammenhang.

ner globalen Zufallszahl für alle Zellen erfüllt dabei aber nicht den Anspruch für individuelles Agentenverhalten.

Um für jede Zelle eine Zufallszahl bereitstellen zu können und gleichzeitig die Kollaboration der Zellen zu gewährleisten, müssen die Zufallszahlen im Voraus generiert werden. Die Zufallszahlen werden in der Generation t erzeugt und können dann in der Generation $t + 1$ verwendet werden. Somit steht jeder Zelle eine Zufallszahl bereit, die über die ganze Generation bestehen bleibt. Für die Bewegung eines Agenten in Abhängigkeit einer Zufallszahl müssen sich die betreffenden Zellen nur auf eine Zufallszahl einigen. Bewegt sich ein Agent in Abhängigkeit einer Zufallszahl eine Zelle vorwärts, so kann für diese Bewegung die Zufallszahl der Agentenzelle oder der leeren Zelle verwendet werden. Wichtig dabei ist nur, dass sowohl die leere Zelle als auch die Agentenzelle die gleiche Zufallszahl verwenden. Damit ist der „copy&move“-Vorgang auch für Bewegungen in Abhängigkeit einer Zufallszahl eindeutig.

Die Umsetzung von Zufallszahlen in der Architektur ist in Abschnitt 6.7.1 erklärt.

3.6 Zusammenfassung

Um zu verstehen, wie der Begriff des Agenten in dieser Arbeit verwendet und verstanden wird, wurde zunächst, auf der Grundlage unterschiedlichster Literaturen der Agentenbegriff definiert. Dazu wurden charakteristische Eigenschaften von Agenten untersucht. Es erfolgte daraufhin eine Unterteilung in Basiseigenschaften, erweiterte Eigenschaften und zusätzliche Eigenschaften. Der ebenso wichtige Begriff sowie die Eigenschaften der Agentenumwelt (allgemein Umwelt) wurden herausgearbeitet. Auf der Basis dieser Begriffe konnte dann der Aufbau des Multi-Agenten-Systems beschrieben werden. Der prinzipielle Aufbau des für die Simulation von Multi-Agenten-Systemen notwendige Multi-Agenten-Simulations-System wurde dargestellt. Auch die Problematik bei der Unterstützung aktiver Einheiten im GCA-Modell sowie die Möglichkeit der Umsetzung von Zufallszahlen im GCA-Modell wurde erläutert. Die Definitionen aus diesem Kapitel sind notwendig, um die Multi-Agenten-Anwendungen aus Kapitel 4 einordnen zu können. Ebenso werden die Begriffe im Zusammenhang mit den Hardwarearchitekturen in Kapitel 6 benötigt.

4 Multi-Agenten-Anwendungen

In diesem Kapitel werden verschiedene Multi-Agenten-Anwendungen mit dem Hintergrund des GCA-Modells vorgestellt. Diese Anwendungen können mit Hilfe des GCA-Modells aus Abschnitt 2.3 und den in Kapitel 6 vorgestellten Architekturen effizient simuliert werden. Mehrere dieser Anwendungen wurden implementiert und deren Umsetzung wird in den entsprechenden Kapiteln zusammen mit den Architekturen beschrieben. Die in diesem Kapitel behandelten Anwendungen stellen einen Bezug zu diskreten Modellen, vor allem dem Zellularen Automaten, dar. Eine der bekanntesten Anwendungen hierfür ist das Nagel-Schreckenberg-Modell für die Verkehrssimulation. Andere Anwendungen betreffen biologische Systeme oder die Evakuierung von Gebäuden, Flugzeugen oder Schiffen.

4.1 Biologie

In der Biologie kommen Multi-Agenten-Systeme zur Simulation von Ökosystemen vor. Die Agenten sind in diesem Fall Tiere, Insekten oder andere Lebewesen. Die Agentenwelt bildet einen Teil der realen Welt ab. In der Simulation können dann verschiedene Fragestellungen geklärt werden, wie z. B., was passiert bei einer Verdopplung der Agentenanzahl? Wie wirken sich Änderungen der Agentenwelt auf die Agenten aus? In einer Simulation mit heterogenen Agenten, also mehreren verschiedenen Agententypen, können auch Auswirkungen der verschiedenen Agenten untereinander beobachtet und geklärt werden.

4.1.1 Multi-Agenten-Simulation einer Honigbienenkolonie

In [STC06, TSC06] wurde die Sammelentscheidung (Verhalten) von Honigbienenkolonien untersucht. Die verschiedenen Bienenarten (Spurbienen, Sammelbienen, Tanzbienen, ...) sind als Agenten modelliert. Das Verhalten der Agenten wurde als Zustandsautomat realisiert [STC06, Seite 20]. Die Zustandsübergänge unterliegen hierbei teilweise fixen Wahrscheinlichkeiten, so dass ein individuelles Verhalten der Agenten gewährleistet ist. Des weiteren besteht innerhalb eines gewissen Kommunikationsradius eine Interaktion zwischen den Agenten. Jeder Agent besitzt einen eigenen Stoffwechsel. Der Energieverbrauch des Agenten ist abhängig von der Bewegung, dem Gewicht und anderen Parametern.
Mit dieser Simulation wurden Umweltfluktuationen auf die Sammelstrategie der Honigbienen untersucht. Die Umwelten wurden mit verschiedenen Fluktuationsmustern simuliert. Dafür wurde die Simulationsplattform „**ho**neybee **fo**rger **sim**ulator“(HoFoSim) verwendet. Jeder Agent ist als abgekoppelter Prozess realisiert und wird sequenziell simuliert.
Die Simulation der Honigbienenkolonien ist für die Multi-Agenten-Simulation sehr gut geeignet und wurde in [STC06] schon als solche realisiert. Es lässt sich somit gut auf das GCA-Modell übertragen, da bereits eine Zellstruktur zugrunde gelegt ist. Unter Verwendung einer der in Kapitel 6 vorgestellten Architekturen kann das Problem dann echt parallel simuliert werden.

Somit ist entweder eine beschleunigte Simulation oder bei gleich bleibender Simulationsdauer eine komplexere Simulation (komplexeres Agentenverhalten, komplexere Agentenwelt, ...) möglich.

4.1.2 Das Ökosystem Wa-Tor

Weitere Anwendungen für Multi-Agenten-Systeme in der Biologie sind Ökosysteme. Dabei wird ein bestimmtes ökologisches System simuliert und dabei die Auswirkungen von Störfaktoren beobachtet. Das System kann z. B. durch äußere Einflüsse gestört werden. Auch kann die Entwicklung des Ökosystems als solches beobachtet werden.

In [Dew84] wird ein Räuber-Beute-Modell bestehend aus Fischen und Haien auf dem Planeten (in der Form eines Torus) Wa-Tor (Water-Torus) simuliert. Die gesamte Oberfläche des Planeten, der in gleichgroße Zellen unterteilt ist, ist dabei mit Wasser bedeckt. Die Fische ernähren sich von Plankton, das unbegrenzt verfügbar ist. Die Haie ernähren sich wiederum von den Fischen. Die Simulation dieser Welt ist eng mit Zellularen Automaten verwandt. Jede Zelle kann dabei einen von drei möglichen Zuständen annehmen. Die Zelle kann dabei entweder leer oder durch einen Hai oder einen Fisch belegt sein. Auf jeder Zelle kann sich maximal ein Lebewesen aufhalten. Für die Simulation (Abb. 4.1) können den drei Zuständen Farben zugewiesen werden.

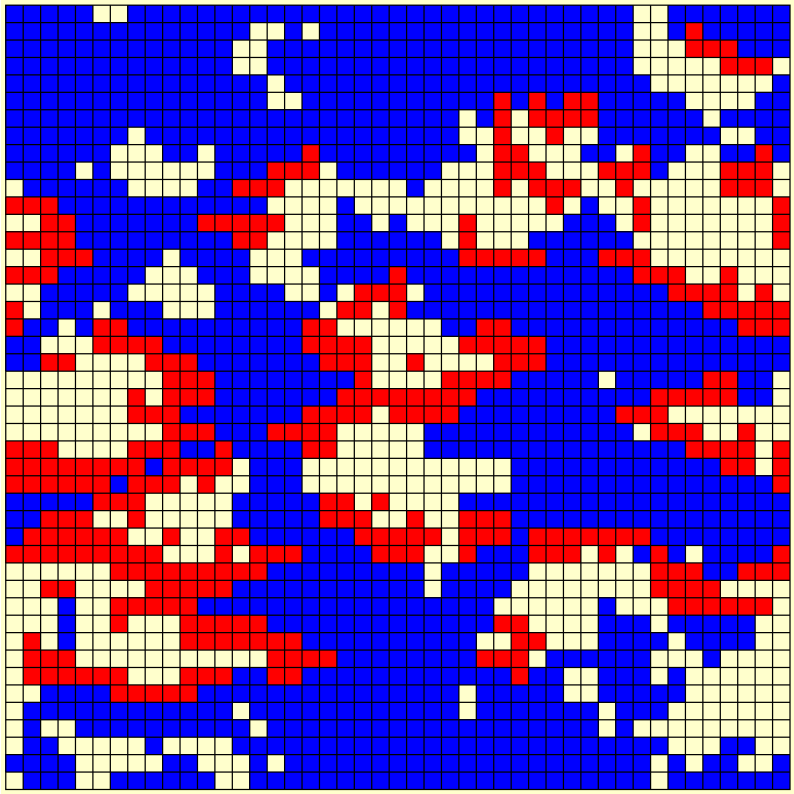


Abbildung 4.1: Simulation von Wa-Tor: Fische (blau, ■), Haie (rot, ■), leere Zellen (hellgelb, ■)

Für das Verhalten von Fischen und Haien gibt es einfache Regeln. Diese Regeln basieren auf den jeweiligen Eigenschaften des Lebewesens. Das Verhalten und die Eigenschaften der Fische sind wie folgt definiert [Dew84]:

- Ein Fisch schwimmt zufällig auf eine der freien angrenzenden Zellen.
- Überschreitet das Alter eines Fisches die „Geburtenzeit“, wird auf einer angrenzenden freien Nachbarzelle ein neuer Fisch erzeugt.

Die Eigenschaften, sowie das etwas komplexere Verhalten der Haie ergeben sich wie folgt:

- Ein Hai frisst einen Fisch, der sich auf einer benachbarten Zelle befindet.
- Befindet sich in keiner der angrenzenden Zellen ein Fisch, schwimmt der Hai zufällig auf eine der angrenzenden Zellen.
- Ein Hai muss innerhalb der „Todeszeit“ einen Fisch fressen, ansonsten stirbt er.
- Überschreitet das Alter eines Haies die „Geburtenzeit“, wird auf einer angrenzenden freien Nachbarzelle ein neuer Hai erzeugt.

Abhängig von der initialen Belegung der Zellen sowie der Parameter („Geburtenzeit“ für Fisch und Hai, „Todeszeit“) ergeben sich drei mögliche Simulationsverläufe. Einerseits können die Haie aussterben, wodurch die ganze Welt von Fischen bevölkert wird. Andererseits können auch die Fische aussterben, was zur Folge hat, dass auch die Haie aussterben. Es kann aber auch ein Gleichgewicht entstehen, in dem die Populationsgröße von Fischen und Haien immer wieder schwankt.

4.1.3 Ameisensimulation

Die Simulation von Ameisenverhalten, u. a. auf Zellularen Automaten, ist eine weitere wichtige Multi-Agenten-Anwendung. Schadschneider verwendet für die Simulation des Verhaltens von Ameisen u. a. Zellulare Automaten [JSCN06, JKN+07, JSCN09]. Durch die Simulation der Ameisen kann das kollektive Verhalten der Ameisen besser untersucht und verstanden werden. Diese und weitere wichtige Erkenntnisse sind dann übertragbar auf die Verkehrssimulation und damit interessant für die Verkehrssteuerung und Entstehung sowie Vermeidung von Staus.

4.2 Physik und Verkehrsphysik

Multi-Agenten-Systeme eignen sich auch für die Simulation in der Physik. Als Beispiel sei das Mehrkörperproblem (Abschnitt 2.3.1) genannt, welches sich sehr gut mit dem GCA-Modell [JHL09] realisieren lässt. Aus dem Bereich der Verkehrsphysik gibt es diverse Anwendungen, die sich für die Agentensimulation eignen. Mögliche Anwendungen sind die Simulation von Kraftfahrzeugen [NS92, Nag94], die Fußgängersimulation [Kir02] oder die Evakuierungssimulation.

4.2.1 Verkehrssimulation

Simulationsmodelle für die Verkehrssimulation

Für die Verkehrssimulation kommen, je nach Fragestellung, verschiedene Modelle in Frage. Diese Modelle können auch für andere Simulationen angewendet werden, sind aber für die Verkehrssimulation von besonderer Bedeutung. Eine Übersicht über die Modelle ist in [Ric05] gegeben. Demnach sind die drei folgenden Kategorien zu unterscheiden [Ric05, Seite 8]:

- **Nachfragemodelle:** Prognose der zu erwartenden Verkehrsnachfrage.
- **Umlegungsmodelle:** Vorhersage der Verkehrsstärke innerhalb eines Straßennetzwerkes.
- **Flussmodelle:** Detallierte Abläufe innerhalb von Fahrzeugströmen.

Im Kontext dieser Arbeit sind auf Grund des Detailgrades des GCA-Modells nur die Flussmodelle von besonderer Bedeutung. Das Nachfragemodell und das Umlegungsmodell werden in dieser Arbeit deshalb nicht weiter betrachtet. Das Flussmodell wird in Bezug auf den Detailgrad noch weiter untergliedert.

Die weitere Untergliederung des Flussmodells nach [Ric05, Seite 9-11]:

- **Makroskopisches Flussmodell:** Diesem Modell liegen einfach strukturierte Ansätze mit relativ geringem Auflösungsgrad zugrunde. Die Modellierung findet anhand ganzer Fahrzeugkollektiven statt. Die benötigte Rechenleistung ist auf Grund des geringen Detaillierungsgrades gering. Das makroskopische Flussmodell eignet sich deshalb gut für die Vorhersage der Spitzenbelastungen auf Autobahnen zu bestimmten Reisewellen. [Ric05, Seite 9-10]
- **Mesoskopisches Flussmodell:** Eine Erweiterung des makroskopischen Flussmodells mit zusätzlichen Teilaspekten bildet das mesoskopische Flussmodell. Eine makroskopische Modellierung des Autobahnverkehrs mit individuellen Reisezeiten für jedes Fahrzeug ist als mesoskopisches Modell anzusehen. [Ric05, Seite 10]
- **Mikroskopisches Flussmodell:** In diesem Modell findet die Modellierung auf Fahrzeugebene statt. Hierbei können für jedes Fahrzeug individuelle Parameter wie Geschwindigkeit, Reaktionszeit usw. eingestellt werden. Auf Grund des hohen Detailgrades wird für die mikroskopische Modellsimulation viel Rechenleistung benötigt. Das mikroskopische Modell ist trotz seiner Probleme (Modell und Realität stimmen nicht immer überein) sehr verbreitet. Dieses Modell findet u. a. in der Stauforschung Anwendung [BKS99, NS92]. Durch die heutzutage größere Rechenleistung von Systemen kann der hohe Detailgrad und damit der hohe Rechenaufwand bewältigt werden. [Ric05, Seite 10-11]
- **Submikroskopisches Flussmodell:** Den höchsten Detailgrad erreicht man durch den Einsatz von submikroskopischen Flussmodellen. Hierbei werden nicht nur die Fahrzeuge, sondern zusätzlich einzelne Fahrzeugkomponenten modelliert. [Ric05, Seite 11]

Mikroskopische Simulationsmethoden

Für die mikroskopische Verkehrssimulation wurden unterschiedliche Ansätze entwickelt. Darunter zählt auch der Ansatz der Simulation auf Zellularen Automaten. Dieser Ansatz wird auch in dieser Arbeit verwendet und findet in einer erweiterten Fassung im GCA-Modell Anwendung. Die Simulation findet dabei in einzelnen Zeitschritten statt.

Die Simulationsmethoden nach [Ric05, Seite 12-16]:

- **„Follow-The-Leader"-Modell:** Die Verkehrssimulation wird in Anlehnung an Regelsysteme modelliert. Die Simulation bildet eine zentrale Formel basierend auf der Masse und Position des Fahrzeuges, der Reaktionszeit und der Sensitivität. [Ric05, Seite 12]
- **„Karlsruher Schule":** Dieser Ansatz beruht auf einem psycho-physischen Fahrzeugmodell. Über eine Wahrnehmungsschwelle wird bestimmt, wann ein Fahrzeug verzögert oder beschleunigt werden kann. Dieser Ansatz soll für die detallierte Analyse konkreter Phänomene im Straßenverkehr geeignet sein. [Ric05, Seite 12-14]
- **Verkehrsmodell nach Gipps:** Das Modell von Gipps besteht aus einer Fahrzeugsimulation und enthält eine Beschreibung für die Simulation von Spurwechseln [Gip81]. Für die Umsetzung der Spurwechsel wurde dazu ein komplexer, mathematischer Entscheidungsprozess formuliert. [Ric05, Seite 14-15]
- **Zellulare Automaten:** Die Verkehrssimulation auf Zellularen Automaten findet im Gegensatz zu den anderen vorgestellten Simulationsmethoden raumdiskret statt. Die Straße ist in äquidistante Stücke (entspricht den Zellen im Zellularen Automaten) unterteilt (Abschnitt 2.2). Die bekannteste und grundlegendste Umsetzung einer solchen Simulation ist das Nagel-Schreckenberg-Modell [NS92]. Dieses Modell wird als Basis für die Simulation auf dem GCA-Modell verwendet. [Ric05, Seite 15-16]
- **Regelbasierter Ansatz:** Ein umfangreicher Ansatz, der ein Regelwerk basierend auf Fuzzy-Logic [Zad65] relativ sprachnah umsetzt. [Ric05, Seite 16]

Straßenverkehr anhand des Nagel-Schreckenberg-Modells

Das Nagel-Schreckenberg-Modell [NS92, SSN96, Nag94, Sch99b] ist ein bekanntes Modell zur Verkehrssimulation. Es ist ursprünglich für die Simulation von Autobahnverkehr entwickelt worden. Mit diesem Modell konnte erstmals das Phänomen des Staus aus dem Nichts erklärt werden [BKS99]. Für die Simulation wird die Straße in äquidistante, diskrete Bereiche unterteilt. Diese Bereiche entsprechen den Zellen im Zellularen Automaten. Auf jeder Zelle kann sich maximal ein Fahrzeug befinden. Die Bewegung der Fahrzeuge findet ebenfalls in diskreten Zeitschritten statt. Diese entsprechen den Generationen im Zellularen Automaten. Damit lässt sich dieses Modell direkt auf Zellularen Automaten anhand einfacher Bewegungsvorschriften simulieren. Im ursprünglichen Modell ist kein Überholen, Unfälle oder individuelles Verhalten von Fahrzeugen vorgesehen. Alle Fahrzeuge bewegen sich nach der gleichen Bewegungsvorschrift und besitzen auch die gleiche maximale Geschwindigkeit.

Die aktuelle Geschwindigkeit eines Fahrzeuges ist durch v gekennzeichnet. Die maximale Geschwindigkeit ist durch v_{max}, der Abstand zum vorausfahrenden Fahrzeug durch Δ gegeben. Damit besteht die Anzahl der freien Zellen vor dem Fahrzeug aus $\Delta - 1$ Zellen. Die Bewegungsvorschrift [NS92, Seite 16] ist durch die folgenden vier Schritte definiert:

- **Beschleunigen:** Die Geschwindigkeit v eines Fahrzeuges wird um eins erhöht $(v_1 = v + 1)$, sofern sie niedriger als die maximale Geschwindigkeit v_{max} ist.
- **Bremsen:** Wenn der Abstand Δ zum vorausfahrenden Fahrzeug kleiner ist als die Geschwindigkeit v_1, dann wird die Geschwindigkeit auf $\Delta - 1$ reduziert $v_2 = \Delta - 1$.
- **Trödeln:** Die Geschwindigkeit v_2 wird mit der Wahrscheinlichkeit π um eins reduziert $v_3 = v_2 - 1$.
- **Bewegen:** Alle Fahrzeuge werden anhand der Geschwindigkeit v_3 auf die neue Zelle bewegt. Die Zielzelle bestimmt sich durch den Index der Ausgangszelle, zu dem die Geschwindigkeit addiert wird. Die neue Geschwindigkeit des Fahrzeuges ist $v = v_3$.

Das Nagel-Schreckenberg-Modell wurde in der Zwischenzeit vielfach erweitert. In [ES97] wird ein mikroskopisches Flussmodell basierend auf einem Zellularen Automaten beschrieben. Hiermit kann auch mehrspuriger Autobahnverkehr simuliert werden. In [Sch98, Seite 164] wird die Verkehrssimulation als weiteres Beispiel für eine Anwendung für Zellulare Automaten vorgestellt. Auf eine genaue Umsetzung wird aber nicht mehr eingegangen, diese ist aber in [Hoc98, Seite 37-42] beschrieben. Ebenso wurden in dieser Arbeit auch Autobahnauffahrten und Kreuzungen realisiert. Die mehrspurige Verkehrssimulation wurde auch von [TSn02, RNSL96] umgesetzt.

Einen wichtigen Effekt beschreibt Schadschneider in [Sch99a]. Dabei handelt es sich um Konfigurationen, die durch die Dynamik niemals erreicht werden können. Diese Konfigurationen, werden auch als *paradiesische Zustände* oder *Garten Eden Zustände* bezeichnet und können nur als Anfangskonfigurationen auftreten. Befindet sich z. B. ein Fahrzeug auf einer Zelle mit der Geschwindigkeit 1 und direkt davor ein Fahrzeug mit der Geschwindigkeit 2, so hätten beide Fahrzeuge von der gleichen Startzelle aus losfahren müssen. Dies ist laut Modell nicht zulässig, weshalb es sich hierbei um einen paradiesischen Zustand handelt.

Die Umsetzung der Verkehrssimulation im Modell des Zellularen Automaten ist eine gute Grundlage für die Umsetzung auf dem GCA-Modell und den Einsatz von Multi-Agenten-Systemen. Die Fahrzeuge können als Agenten interpretiert werden. Für verschiedene Arten von Fahrzeugen (z. B. PKW oder LKW) werden verschiedene Agententypen verwendet. Die Umsetzung der Verkehrssimulation anhand des Nagel-Schreckenberg-Modells als Multi-Agenten-System auf einer Hardwarearchitektur unter Verwendung des GCA-Modells wird in Abschnitt 6.7.5 behandelt. Alternativ kann die Verkehrssimulation unter der Verwendung von CUDA[1] auch auf aktuellen Grafikkarten durchgeführt werden [SN09a, SN09b]. Im Gegensatz zum Nagel-Schreckenberg-Modell verwenden die Autoren auch hier den Ansatz der Multi-Agenten-Simulation. Die Programmierung von Multi-Agenten-Systemen auf CUDA gestaltet sich allerdings teilweise als umständlich und schwierig, wie die Autoren von [DLR07] bestätigen.

[1] Compute Unified Device Architecture

Ein Modell für die effiziente Simulation von Straßenverkehr unter der Verwendung des GCA-Modells wird in [LDS10] beschrieben. Für die Repräsentation mehrerer paralleler Fahrspuren ist unter Verwendung des GCA-Modells die Umsetzung als eindimensionaler Globaler Zellularer Automat möglich. Hierbei werden die parallelen Fahrspuren aneinander gehängt.

4.2.2 Fußgänger- und Evakuierungssimulationen

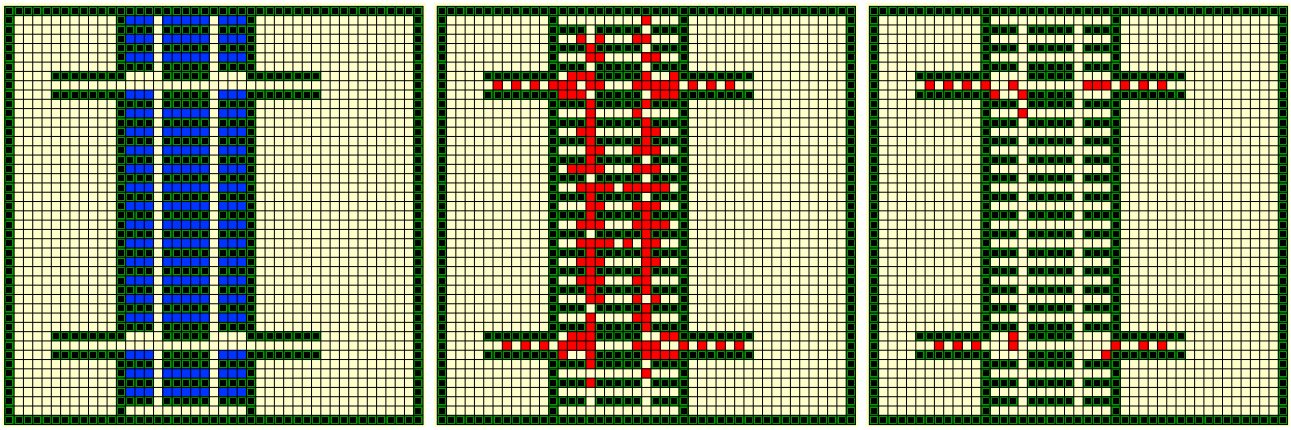
Auch die Bewegung von Fußgängern lässt sich als Multi-Agenten-System simulieren. Die Fußgänger und deren spezifisches Verhalten werden als Agenten modelliert. Das Verhalten der Agenten, die Simulationsmethoden sowie auftretende Phänomene wurden von Kirchner [Kir02] und Kretz [Kre07] ausführlich untersucht. Kirchner stellt zudem fest, dass die Zellularen Automaten für die Verkehrssimulation eine wichtige Klasse von Modellen darstellt. Ziel seiner Arbeit ist es:

> *„ein Zellularautomaten-Modell (CA-Modell) zur Beschreibung von Fußgänger-Dynamik vorzustellen, das in der Lage ist, die wichtigsten kollektiven Effekte im Verhalten von Fußgängern zu reproduzieren und andererseits die Zahl der Wechselwirkungsterme pro Zeitschritt auf $O(N)$ zu reduzieren.“ [Kir02, Seite 33]*

In seiner Arbeit [Kir02, Seite 33-35] stellt er die folgenden kollektiven Phänomene der Dynamik von Fußgängern vor:

- **Staubildung:** Verlassen viele Menschen einen Raum, kommt es an der Tür zu Staubildung.
- **Spurbildung:** Bei hoher Dichte und entgegengesetzter Bewegungsrichtung von Menschen bilden sich dynamisch variierende Spuren in die selbe Richtung.
- **Oszillationen:** An einer Engstelle (z. B. Tür) kommt es zu oszillierenden Bewegungsmustern. Geht ein Fußgänger durch die Tür, folgen ihm weitere in gleicher Bewegungsrichtung. Bei einer Lücke kann sich die Bewegungsrichtung ändern, wenn es einem Fußgänger gelingt, die Tür in Gegenrichtung zu passieren.
- **Muster an Kreuzungen:** An Kreuzungen kann es zu spontanen und kurzlebigen Bewegungsmustern kommen, die die Fußgängerströme effizienter machen können.
- **Ausbildung von Trampelpfaden:** Trampelpfade stellen Abkürzungen zwischen existierenden Wegen dar.
- **Panik:** In Paniksituationen kommt es zu irrationalen kollektiven Handlungen.

Für die Simulation verwendet Kirchner ein zweidimensionales Quadratgitter. Jede Zelle des Gitters ist dabei etwa $40 \times 40\ cm^2$ groß, was den typischen Platzbedarf eines Menschen darstellt. Die zeitliche Simulation erfolgt in diskreten Zeitschritten. Die Modellierung stellt somit einen Zellularen Automaten dar. Um die Bewegung der Menschen möglichst realistisch umzusetzen, kommmt ein statisches und ein dynamisches Grundfeld zum Einsatz. Das statische Grundfeld gibt dabei den Weg zum nächstgelegenen Ausgang vor. Jeder Zelle wird dabei eine Zahl zugeordnet, die die Entfernung zum nächsten Ausgang codiert. Somit kann sich ein Mensch entscheiden, auf welche Nachbarzelle er sich im nächsten Schritt bewegen möchte. Das dynamische Grundfeld gibt Auskunft über die vergangenen Bewegungen anderer Menschen. Somit bildet sich eine



(a) Startbelegung: Passagiere (blau, ■), Flugzeug (grün, ■)

(b) Simulation nach 35 Generationen: Passagiere (rot, ■), Flugzeug (grün, ■)

(c) Simulation nach 99 Generationen: Passagiere (rot, ■), Flugzeug (grün, ■)

Abbildung 4.2: Evakuierungssimulation eines Flugzeuges mit vier Notrutschen zu drei verschiedenen Zeitpunkten.

virtuelle Spur, die die anderen Menschen in ihrer Bewegung beeinflusst. Eine Simulation anhand des statischen Grundfeldes ist in Abbildung 4.2 zu drei verschiedenen Zeitpunkten dargestellt.
Die Simulation von Fußgängern auf Zellularen Automaten bietet sich auf Grund der Einteilung der Bewegungsfläche in ein Quadratgitter direkt an. Darüber hinaus können damit die beschriebenen kollektiven Phänomene realitätsnahe umgesetzt und simuliert werden. Für die Konfliktbehandlung schlägt [Kre07, Seite 28-29] verschiedene Varianten vor. So kann ein Agent nicht nur die Zelle blockieren, auf der er sich aktuell befindet, sondern auch zusätzliche benachbarte Zellen. Somit wird ein bestimmter Abstand zu anderen Agenten eingehalten, die sich nicht auf die blockierten Zellen bewegen können.

Ein wichtiger Anwendungsfall für die Fußgängersimulation ist die Evakuierungssimulation. Immer wieder kommt es bei großen Menschenansammlungen (z. B. bei Konzerten) oder bei der Evakuierung von Flugzeugen [Klü03, Seite 91][PKJFMC09], Schiffen [Klü03, Seite 110] und Gebäuden [Klü03, Seite 74-90] [GKSA10, GSA08] zu tödlichen Zwischenfällen. Mit einer realistischen und effizienten Simulationsmethode können dann im Vorfeld verschiedene Szenarien simuliert, Probleme erkannt und beseitigt werden. Auch unter der Verwendung von Zellularen Automaten können komplexere Simulationen modelliert werden. So ist z. B. die Evakuierungssimulation eines mehrstöckigen Passagierschiffes auf Zellularen Automaten ohne Einschränkungen möglich [Klü03, Seite 110]. Auch die Evakuierung von Flugzeugen kann im Vorfeld unter der Verwendung des Zellularen Automaten simuliert werden [PKJFMC09].

4.2.3 Eisenbahnverkehrssimulation

In seiner Arbeit [Ste01] stellt Stebens die Simulation von Eisenbahnverkehr vor. Hierbei spielen, im Vergleich zur Simulation des Straßenverkehrs, andere Aspekte eine Rolle. Diese betreffen u. a. die Streckenauslastung, Überlastungen von Strecken und Knoten, die Fahrplangestaltung, das Routing und die Disposition. Für die Züge sowie die Strecken gibt es dabei verschiedenste

Parameter, wie z. B. die Höchstgeschwindigkeit, die Beschleunigung und Verzögerung, die Länge sowie der Zustand und die Ausstattung der Strecke. Ein wichtiges Kriterium sind die Fahrpläne, die nach Möglichkeit eingehalten werden sollten. Verspätungen wirken sich direkt auf den Zug, aber auch indirekt auf andere Züge, aus.

Für die Simulation des Zugverkehrs kommt in [Ste01] das Modell des Zellularen Automaten zum Einsatz. Dabei wird eine Strecke in gleichgroße Teilstücke aufgeteilt und den Zellen zugeordnet. Ein Zug wird aber nicht komplett durch eine Zelle repräsentiert, sondern jeder Waggon bzw. jede Lok durch eine Zelle. Somit belegt ein Zug bestehend aus einer Lok und zwei Waggons insgesamt drei Zellen. Für die Simulation schlägt Stebens drei Modelle vor. Das Zug-Modell, das Signal-Modell und das Dispositions-Modell.

4.3 Brandentwicklung

Die Simulation von sich ausbreitendem Feuer in unterschiedlichen Szenarien stellt eine wichtige Aufgabe in der Brandbekämpfung dar. Dabei ist es einerseits wichtig, existierende Gebiete in Bezug auf mögliche Brände hin zu untersuchen, andererseits auch, wie sich Änderungen auf ein mögliches Brandszenario auswirken. Für die Simulation derartiger Szenarien kommen u. a. auch Zellulare Automaten zum Einsatz. Dabei repräsentiert eine Zelle ein bestimmtes festgelegtes Gebiet. Die Anwendungsfälle erstrecken sich über viele Anwendungsgebiete, z. B. Waldbrand [BSv06, LM01]. Der Einsatz von Zellularen Automaten bietet sich an und wurde auch in weiteren Anwendungsfällen bereits gezeigt [TCKT00, QMIG10]. Das GCA-Modell sowie die Verwendung von Agenten bietet sich hier ebenso an. Das Feuer breitet sich zum einen lokal aus, d. h. dass Gebiete, die an ein Feuer angrenzen, als nächstes anfangen werden, zu brennen. Zum anderen gibt es aber auch globalere Effekte, wie z. B. umherfliegende Funken, die auch weiter entferntere Gebiete betreffen. Diese entfernteren Gebiete können direkt durch das GCA-Modell unterstützt werden. Des weiteren zeigt dieses Beispiel den Einsatz von inhomogenen MAS auf. Lokal agierende Agenten definieren, wie sich das Feuer auf benachbarte Zellen ausbreitet, während global agierende Agenten den Funkenflug simulieren.

Die verschiedenen Modelle zur Ausbreitung von Feuer können nach [BSv06, Seite 1] in die folgenden Gruppen eingeteilt werden:

- Empirische oder statistische Modelle
- Semi-empirische Modelle (Labormodelle)
- Physikalische Modelle (theoretisch oder analytisch)

Die Autoren von [BSv06] ordnen den Zellularen Automaten der Gruppe der semi-empirischen Modelle zu. Für die Umsetzung der Brandsimulation kann eine Zelle einen von vier Zuständen annehmen. Die vier möglichen Zustände sind brennend, verbrannt, wachsend und entzündbar. Zusätzlich besitzt jede Zelle einen Wert, der darüber Auskunft gibt, ob auf dieser Zelle ein Baum wächst.
Die Simulation erfolgt anhand von vier einfachen Regeln. Der Zustand der Zelle ändert sich dabei, außer im ersten Schritt, in jeder Generation. Eine Simulation von Funkenflug ist hierbei

nicht direkt vorgesehen. Die Simulation von Windeinflüssen ist durch eine erweiterte Nachbarschaft des Zellularen Automaten umgesetzt worden. Die vier Schritte der Zellregel sind [BSv06]:

1. Eine entzündbare Zelle bleibt entzündbar, bis eine der direkten Nachbarzellen brennt. Brennt mindestens eine direkte Nachbarzelle, wechselt der neue Zellzustand auf brennend.
2. Eine brennende Zelle wechselt ihren Zustand auf verbrannt.
3. Eine verbrannte Zelle wechselt ihren Zustand auf wachsend.
4. Eine wachsende Zelle wechselt ihren Zustand auf entzündbar.

Ein anderer Anwendungsfall ist die Simulation von Buschfeuern. In [LM01] werden die Auswirkungen und die Entwicklungen eines Feuers anhand verschiedener initialer Bedingungen untersucht. Dabei werden für die Simulation eine Reihe von Faktoren berücksichtig. Dazu gehören die Wachstumsdichte sowie die Brennbarkeit der Büsche. Ebenso berücksichtigt wird die Temperatur, die Höhe der Landschaft und die Windeinflüsse. Über eine individuelle Zuordnung dieser Eigenschaften kann eine realistische Simulation erreicht werden. Die Wachstumsdichte zeigt, abhängig von der Wahl der Nachbarschaft, wie einzelne Gebiete von Feuer vernichtet werden und wie diese Gebiete optimal angeordnet oder voneinander getrennt werden können bzw. sollten. Da die Vegetation nicht auf allen Zellen identisch ist, wird über die Brennbarkeit der einzelnen Zellen eine genauere Simulation der Ausbreitung des Feuers möglich. Äußere Umwelteinflüsse, wie z. B. die Windrichtung und Windstärke, haben Einfluss auf die Ausbreitung des Feuers. Es sind dabei nicht alle Nachbarzellen als gleichwertig anzusehen. Die Zellen, die in Windrichtung an eine brennende Zelle angrenzen, werden mit einer höheren Wahrscheinlichkeit, als die gegenüberliegende Nachbarzelle, anfangen zu brennen. Die Autoren von [LM01] geben aber auch zu bedenken, dass nur lokale Einflüsse bei der Simulation berücksichtigt werden. Funkenflug, der weiter entfernte Zellen entzünden kann, ist bei dieser Simulation nicht vorgesehen. Ein derartiger Effekt lässt sich durch eine individuelle und zufällige Nachbarschaft, wie sie im GCA-Modell möglich ist, umsetzen.

Die Simulation von Bränden kann auch für bewohnte Gebiete verwendet werden. Dass auch mit einer Simulation unter Verwendung von Zellularen Automaten realistische Ergebnisse erzielt werden können, zeigt [TCKT00]. Hier wird die Feuerausbreitung in Hyogo-ken Nambu von 1995 in der Folge eines Erdbebens simuliert und mit vorhandenen Aufzeichnungen des realen Verlaufs der Feuerausbreitung verglichen. Das Feuer konnte sich gut ausbreiten, da hier viele Häuser aus Holz gebaut waren. Die Simulation mit Zellularen Automaten erlaubt, selbst mit einfachen Zellregeln gute Ergebnisse bei gleichzeitig schneller Simulation zu erzielen. Die Unterschiede des Simulationsverlaufs im Vergleich zum realen Verlauf des Feuers beziehen die Autoren auf zusätzliche Einflüsse, wie z. B. Wind und der Austritt von Gas, als Folge des Erdbebens. Die Simulation des Feuers benötigt auf einem Pentium II Xeon 450 MHz 34 Sekunden [TCKT00, Seite 7].

4.4 Künstliche Agenten

Neben der Simulation von Anwendungen aus der realen Welt besteht auch die Möglichkeit der Simulation von fiktiven oder nur bedingt realitätsnahen Multi-Agenten-Systemen. Eine solche

Anwendung ist die Testapplikation (Abschnitt 6.6.4.2) für die Bewertung der Hardwarearchitekturen in Kapitel 6 [SHH09a]. Derartige Systeme können zur Untersuchung von Algorithmen oder Kommunikationsmustern verwendet werden. Auswirkungen auf das Verbindungsnetzwerk der verwendeten Architektur können so gezielt untersucht werden.

Eine weitere Anwendung für Agenten ist das „Creature Exploration Problem“, das u. a. in [Hal08] untersucht wurde. Bei diesem Problem ist ein zweidimensionaler zellularer Automat mit Agenten (oder Kreaturen) gegeben. Die globale Aufgabe der Agenten ist es, alle freien Zellen der Agentenwelt in einer minimalen Anzahl von Generationen mindestens einmal zu besuchen. Die Agenten besitzen dabei nur eine lokale Sicht und haben keine Kenntnis über das globale Ziel. Daher verhalten sich die Agenten auch nur lokal. Für die Aktionen der Agenten ist eine Kollisionsbehandlung notwendig. Existierende Hindernisse sowie andere Agenten müssen erkannt und eine Kollision verhindert werden. Eine optimale Lösung bei einer vorgegebenen Anzahl von Agenten ist für verschiedene Anwendungen interessant und besitzt praktische Relevanz. Das schnellstmögliche Erkunden eines Gebietes, Informationen effizient verteilen oder eine Wiese schnellstmöglich zu mähen, sind nur einige Anwendungsfälle.

In [EH08] ist die Aufgabe der Agenten, Informationen auszutauschen. Dazu erhält jeder Agent eine von den anderen Agenten disjunkte Information. Das Ziel jedes Agenten ist es, alle Informationen zu sammeln. Hierfür gibt es unterschiedliche Implementierungsmöglichkeiten. Zum einen kann definiert werden, dass ein Agent eine Information nur von dem Agenten erhalten kann, der diese Information ursprünglich besaß. Das bedeutet, ein Agent kann keine Informationen anderer Agenten weitergeben. Ein anderer Ansatz, der auch in [EH08] verwendet wurde, erlaubt diesen Fall. Insgesamt hat dies hauptsächlich Auswirkung auf die Anzahl der Generationen, die zur Lösung dieser Aufgabe benötigt werden, also bis jeder Agent alle Informationen besitzt. Zusätzlich muss definiert sein, wann zwei Agenten ihre Informationen austauschen können. Dies kann z. B. sein, wenn sich zwei Agenten in Bewegungsrichtung gegenüberstehen. Möglich sind aber auch andere Fälle oder gar der parallele Informationsaustausch von mehreren Agenten gleichzeitig. Derartige Simulationen können dazu verwendet werden, die Ausbreitung von Krankheiten zu simulieren [KAANS10].

Die Aufgabe der Agenten in [EH09b] besteht darin, eine dedizierte Zielzelle aufzufinden. Jeder Agent befindet sich zu Beginn der Simulation auf einer Zelle und bewegt sich über den Verlauf der Simulation zu der ihm zugeordneten Zielzelle. Auch hier bestehen verschiedene Implementierungsfreiheiten. Ein Agent könnte sich z. B. zum Simulationsbeginn bereits auf der Zielzelle befinden. Im nächsten Schritt könnte er entweder entfernt werden, da er sich bereits auf der Zielzelle befindet, oder sich sogar vom Ziel weg bewegen. Des weiteren kann für jeden Agenten ein persönliches Ziel definiert werden oder für eine Gruppe von Agenten, u. U. alle, eine gemeinsame Zielzelle. Bei der Bewegung der Agenten von ihrer Startzelle zu ihrer Zielzelle entstehen teilweise interessante Muster. Die Verwendung von derartigen Zielzellen für die Agentensimulation kann auch bei der Simulation aus Abschnitt 4.2.2 angewendet werden. Bei der Evakuierungssimulation bewegen sich nicht alle Agenten zum nächstgelegenen Ausgang. Über das Bewegen einiger Agenten zu einem anderen Ausgang (äquivalent der Zielzelle) kann ein Erinnerungsvermögen realisiert werden.

In [EH09a] können die Agenten Objekte, die auf jeder Zelle vorhanden sind, beeinflussen. Das globale Ziel der Agenten ist es, alle Objekte in eine Richtung auszurichten. Es hat allerdings keiner der Agenten die Information über das globale Ziel. In [EH09a] können die Agenten ein Objekt um 90° links herum, um 90° rechts herum drehen oder es in der aktuellen Ausrichtung belassen.

4.5 Ausbreitung von Krankheiten

In [BMS03] werden „Situated Cellular Agents (SCA)“ vorgestellt. Diese sind definiert als Systeme bestehend aus reaktiven, heterogenen Agenten. Das Verhalten der Agenten wird durch den Zustand und den Typ der benachbarten („adjacent and at-a-distance“) Agenten beeinflusst. Den Agenten liegt somit eine Art zellulares Feld zugrunde. Eine der vorgestellten Anwendungen ist die Simulation des Immunsystems. Die Autoren belegen, dass dies auf Zellularen Automaten gut umzusetzen ist. Das Ziel der „Situated Cellular Agents“ ist es, ein flexibleres und detaillierteres Tool für die Modellierung von Immunsystemen zu erstellen.

Die Ausbreitung von Viren in einer größeren Gruppe von Personen wurde in [RIP99] untersucht. Ziel der Simulation ist das Verstehen der grundlegenden Mechanismen von Viren. Damit können verschiedene Fragestellungen untersucht werden: Verhält sich die Infektionsrate linear? Wie schnell breitet sich das Virus aus? Die Mechanismen der Virusausbreitung sind aber auch grundlegend für weitere Anwendungen, wie z. B. die Verbreitung von Modetrends oder die Verbreitung von Informationen. Für die Simulation der Virusausbreitung kommen verschiedene Agententypen zum Einsatz: gesunde Personen, infizierte Personen, Ärzte. Die Simulation der Agenten erfolgt regelbasiert. Die Regeln bestehen dabei aus zwei Teilen, Bedingungen und Aktionen.

Ein weiteres Modell für die Ausbreitung von Krankheiten wird in [FM04] vorgestellt. Die Simulation ermöglicht es, Experimente durchzuführen, die in der Realität so nicht durchzuführen sind. Dazu wird das Modell des Zellularen Automaten verwendet. Im Gegensatz zu traditionellen Zellularen Automaten können in diesem Modell pro Zelle mehrere „hosts“ untergebracht sein. Der Zustand einer Zelle ist definiert durch die anfälligen, die infizierten und die gesunden hosts. Jede Zelle hat einen individuellen Parameter, der die Anzahl der hosts pro Zelle begrenzt. Die Landschaft ist durch das zellulare Feld gegeben und in die diskreten Zellen aufgeteilt. Die Anzahl der hosts pro Zelle kann sich im Simulationsverlauf ändern. Hosts können von einer Zelle zu anderen Zellen migrieren. Weitere Effekte, wie Geburten und Tod, sind ebenso berücksichtigt. In dem Modell von [FM04] kann eine Zelle kontaminiert werden, in dem ein infizierter host auf eine andere Zelle migriert und damit dort weitere hosts infiziert oder in dem der Erreger durch seinen eigenen Verbreitungsalgorithmus andere Zellen infiziert. Während diese Simulationen eher abstrakt sind und das ganze zellulare Feld allgemein zur Verfügung steht, wird in [Hai10] die Simulation in Gebäuden vorgenommen. Im Speziellen werden Gebäude mit größeren Hallen oder Gängen untersucht. Dabei wird, anders als in [FM04], nicht von Zellularen Automaten Gebrauch gemacht. Die Simulation soll dabei in Echtzeit durchgeführt werden. Wie die genaue Umsetzung erfolgte, ist nicht weiter ausgeführt worden.

4.6 Zusammenfassung

Dieses Kapitel behandelte unterschiedlichste Multi-Agenten-Anwendungen und zeigt damit die Bedeutung der Thematik auf. Die Thematik erstreckt sich über unterschiedlichste Themengebiete. Dargestellt und näher beschrieben wurden einige wichtige Anwendungen, welche teilweise auch auf den in Kapitel 6 entwickelten Hardwarearchitekturen umgesetzt wurden. Die Verkehrssimulation ist dabei hervorzuheben, da sich diese sehr gut auf dem Multi-Agenten-System umsetzen lässt und auch die Eigenschaften des Globalen Zellularen Automaten entsprechend ausnutzt. Wie der Literatur zu entnehmen ist, ist die Umsetzung und Weiterentwicklung der vorgestellten Multi-Agenten-Anwendungen bis heute aktuell.

5 Forschungsstand

In diesem Kapitel wird der aktuelle Forschungsstand bezüglich der Umsetzung von Zellularen und Globalen Zellularen Automaten betrachtet. Einige Aussagen beziehen sich allgemein auf Zellulare sowie Globale Zellulare Automaten. Um solche Aussagen von jenen, die nur Zellulare Automaten betreffen, abzugrenzen, wird die Abkürzung GCA/CA verwendet, wenn sowohl Zellulare Automaten als auch Globale Zellulare Automaten gemeint sind.
GCA/CA können auf verschiedene Weisen implementiert werden. Die einfachste und naheliegendste Möglichkeit ist die Implementierung als Software zur Ausführung auf einem Prozessor. Diese Möglichkeit ist besonders beim Einsatz von Multicoreprozessoren interessant. Die Hardware ist hierbei schon verfügbar und eine parallele Ausführung lässt sich durch verschiedene vorhandene Softwarekonstrukte realisieren. Ob durch diese Softwarekonstrukte immer eine effizientere Ausführung ermöglicht wird, hängt von vielen verschiedenen Faktoren ab. Eine weitere interessante Alternative bieten Grafikprozessoren. Hier steht ebenfalls schon entsprechende Hardware zur Verfügung und der Grad der möglichen Parallelität ist um ein Vielfaches größer. Ebenfalls möglich ist die Realisierung von neuer und für die Ausführung von GCA/CA angepasster Hardware, z. B. durch Verwendung von FPGAs (Field Programmable Gate Array). Die Realisierung dieser Hardware ist sehr aufwändig, verspricht aber eine größere Leistung zu erzielen. Dies ist u. a. dadurch möglich, da der entstehende Kommunikationsaufwand der Komponenten untersucht und direkt Einfluss auf die Architektur genommen werden kann. Wichtig für die Umsetzung von GCA/CA ist auch die Umsetzung der Generationssynchronisation. Diese muss durch entsprechende Konstrukte möglichst effizient umgesetzt werden können. Die verschiedenen Möglichkeiten für die Umsetzung von GCA/CA, insbesondere des GCA-Modells sowie der aktuelle Stand der Forschung, werden im Folgenden dargestellt.

5.1 GCA-Hardwarearchitekturen

Auf Grund der bestehenden Problematik bei der effizienten Umsetzung von GCA/CA auf Grafikprozessoren und der großen Anzahl von Anwendungen für GCA/CA ist die Entwicklung einer angepassten Hardwarearchitektur naheliegend. Mit dem Fokus auf die Simulation von MAS unter Verwendung des GCA-Modells ergeben sich dadurch noch vielfältigere Optimierungsmöglichkeiten. Des weiteren ist der Vorteil einer Implementierung auf Grafikprozessoren gegenüber einer Softwareimplementierung und Ausführung auf Prozessoren bei optimierter Implementierung nicht sehr groß oder nicht mehr gegeben [LKC+10].

Bisher wurden verschiedene Architekturen (vollparallele, teilparallele/datenparallele und Multiprozessorarchitekturen) für das GCA-Modell in der Arbeit von Heenes [Hee07] behandelt. Diese Architekturen dienen als Grundlage für diese Arbeit. Die grundlegenden Architekturen sind im Einzelnen:

a) **Vollparallele Architektur:** Alle Zellen inklusive der globalen Verbindungen sind in Hardware realisiert. Die Ausführungszeit ist minimiert. Die Skalierbarkeit ist durch die vorhandenen Ressourcen eingeschränkt, weshalb nur eine begrenzte Anzahl an Zellen verwendet werden kann.

b) **Datenparallele Architektur:** p Pipelines berechnen parallel die Zellregel. Damit ist die Architektur leistungsstark und skalierbar. Unterschiedliche Zellstrukturen sind schwierig umzusetzen.

c) **Multiprozessorarchitektur:** p Prozessoren sind mit einem einfachen Netzwerk (nur lesender Zugriff) verbunden. Die Architektur ist sehr flexibel, anpassbar und einfach programmierbar, aber nicht ganz so leistungsfähig wie eine Spezialarchitektur. Zusätzliche Befehle können einfach hinzugefügt werden und die Ausführung beschleunigen.

Die Leistungsfähigkeit einer Multiprozessorarchitektur unter Verwendung eines Multiplexernetzwerks für das Bitonische Mischen von 128 Zahlen (entspricht 128 Zellen) wurde in [HHJ06] ausgewertet. Die Ausführungszeit liegt zwischen 0,951 ms (ein Prozessor) und 0,052 ms (16 Prozessoren)[1]. Im Vergleich zu einer Softwareimplementierung auf einem Prozessor ist die Beschleunigung gering, wenn überhaupt gegeben. Die Anzahl der Zellen ist mit 128 Zellen gering und somit auch der Einsatz der Architektur für praktische Anwendungen. Neuere Architekturen auf dieser Basis sollten daher ein größeres Zellfeld abdecken können und somit auch die praktische Anwendbarkeit verbessern. Zusätzlich lassen sich dann weitere und u. U. präzisere Aussagen über die Architektur und deren zukünftige Entwicklung treffen. Ein weiterer wichtiger Aspekt ist die Skalierbarkeit der Architektur. Einerseits soll mit der Architektur ein hoher Parallelitätsgrad erreicht werden, andererseits steigen damit die benötigten Ressourcen stark an. Der Einfluss der Skalierbarkeit unter unterschiedlichen Vorbedingungen soll deshalb hier noch genau untersucht werden. Für die Zugriffe innerhalb der Architektur ist das Verbindungsnetzwerk von zentraler Bedeutung. Ein Netzwerk, das alle Anforderungen bezüglich des GCA-Modells optimal erfüllt, ist nicht realisierbar. Zu diesen Anforderungen zählt einerseits ein schneller Zugriff. In diesem Kontext bedeutet schnell, dass ein Zugriff in möglichst wenig Taktzyklen abgeschlossen sein sollte. Andererseits soll die Taktfrequenz des Gesamtsystems möglichst hoch sein. Dabei hat das Verbindungsnetzwerk einen großen Einfluss auf die maximal erreichbare Taktfrequenz. Mit einer zunehmenden Anzahl an Prozessoren bzw. Verarbeitungseinheiten sollte das Verbindungsnetzwerk auch einen parallelen Anteil besitzen, also mehrere Anfragen parallel abarbeiten können. Damit steigt im Allgemeinen auch die Komplexität des Verbindungsnetzwerks. Ein Vorteil der Multiprozessorarchitektur liegt jedoch darin, dass vom strikten Modell des GCA/CA abgewichen werden kann. Somit können z. B. kostspielige externe Lesezugriffe[2] über das Verbindungsnetzwerk vermieden werden, wenn auf Grund anderer vorliegender Daten Aussagen über die extern zu ladenden Daten getroffen werden können.

Eine andere Möglichkeit ist die Umsetzung des GCA-Modells als datenparallele Architektur [JEH08b] unter der Verwendung mehrerer Speicherbänke. Die Leistungsfähigkeit dieser Architektur wurde anhand verschiedener Anwendungen gezeigt (Mehrkörperproblem [JHL09],

[1] Auf einer ersten neuen GCA-Architektur (Abschnitt 6.6) benötigt das Bitonische Mischen von 128 Zahlen 0,217 ms (ein Prozessor) bis 0,035 ms (16 Prozessoren). Die Architektur ist somit insgesamt bereits ohne die später zusätzlich vorgenommenen Optimierungen (u. a. Optimierung der Taktfrequenz der Architektur in Abschnitt 6.9.6) leistungsfähiger als die vorherige Multiprozessorarchitektur aus [HHJ06].

[2] Der Unterschied zwischen internen und externen Lesezugriffen wird in Kapitel 6 definiert.

Jacobi-Verfahren [JHE09], Graphenalgorithmen [JHK07, JHK08]). Für das Mehrkörperproblem lässt sich mit acht Pipelines eine Leistung von 5,94 GFlops erreichen. Dabei besteht jede Pipeline aus 17 Gleitkommaeinheiten mit jeweils 150 Pipelinestufen [JHL09].

Eine alternative, nicht näher beschriebene Architektur haben die Autoren von [CCH11] entwickelt. Hierzu werden „Processing Elements“ zur Berechnung der Agentenwelt verwendet. In diesem Fall wird das „Game of Life“ implementiert, das in Abschnitt 2.2.4 vorgestellt wurde. Wie in Kapitel 3 ausgeführt, zähle ich diese Anwendung nicht zu den Multi-Agenten-Anwendungen, da sich hier im eigentlichen Sinn keine Agenten bewegen. Deshalb gibt es in [CCH11] faktisch keinen Unterschied zwischen einer Zelle und einem Agenten. Für die Simulation einer 32×32 großen Agentenwelt mit 1024 Agenten wurde auf einem Xilinx Virtex-5 FPGA bei einer Taktfrequenz von 150 MHz im Vergleich zu einem AMD Athlon 2,9 GHz Vierkern CPU eine Beschleunigung von 40 erreicht. Allerdings ist die genaue Umsetzung der Vergleichssoftware unklar.

Die effiziente Simulation von Agenten auf Zellularen Automaten wurde in [Hal08, HH07] behandelt. Die Realisierung einer Hardwarearchitektur zur Simulation von Agenten gestaltet sich auf Zellularen Automaten einfacher als auf Globalen Zellularen Automaten. So wird in [Hal08] eine massiv parallele Hardwarearchitektur vorgestellt, die eine Generation in einem Taktzyklus berechnet. Dafür ist es notwendig, dass jeder Zelle eine Berechnungseinheit in Hardware zur Verfügung steht und die Kollisionsbehandlung für Agenten kombinatorisch stattfindet. Eine derartige Kollisionsbehandlung ist in Zellularen Automaten möglich, da nur die benachbarten Zellen berücksichtigt werden müssen. In Globalen Zellularen Automaten ist eine derartige Umsetzung nicht mehr möglich. Die Nachbarschaft wird in jeder Generation dynamisch neu bestimmt. Eine Kollisionsbehandlung einer Zelle, als kombinatorisches Schaltnetz ausgeführt, müsste alle Zellen des Globalen Zellularen Automaten berücksichtigen. Aus Gründen der Skalierbarkeit ist dies nicht praktikabel. In Tabelle 5.1 werden die wichtigsten Unterschiede zwischen Agenten auf Zellularen Automaten [Hal08] und Agenten auf Globalen Zellularen Automaten aufgeführt. Für den Globalen Zellularen Automaten wird eine Realisierung unterstellt, deren kleinste Recheneinheit keine Zelle ist. Ansonsten müssten zwischen allen Zellen auf Grund der Möglichkeit, globale Zugriffe zu erlauben, Verbindungen realisiert werden.

Eigenschaften	Zellularer Automat	Globaler Zellularer Automat
Nachbarschaft	lokal	global, dynamisch, wahlfrei
Kollisionsbehandlung ↦Umsetzung	einfach kombinatorisch, Schaltnetz	komplex getaktet
Recheneinheit	Zelle	Prozessor
Skalierungsverhalten	schlecht	gut
Parallelitätsgrad	sehr gut	beschränkt
Ressourcenabhängigkeit	Zellanzahl, Agentenanzahl	Prozessoranzahl (teilw. Zellanzahl)

Tabelle 5.1: Architekturen für die Multi-Agenten-Simulation auf Zellularen Automaten und Globalen Zellularen Automaten im Vergleich

Die Architekturen in [Hal08] benötigen relativ viele Hardwareressourcen. Dies liegt an dem massiv parallelen Aufbau. Einerseits ist damit eine schnelle Ausführung möglich, andererseits ist die Skalierbarkeit und damit die Anzahl der Zellen begrenzt. Für eine größere Anzahl an Zellen sind andere Architekturen notwendig. Für das GCA-Modell sind zellenbasierte Architekturen (d. h. jede Zelle besitzt eine Recheneinheit für die Ausführung der Zellregel) ebenso wenig geeignet. Auf Grund der dann viel komplexeren Kollisionsbehandlung ist eine noch schlechtere Skalierbarkeit gegeben. Deshalb wird nicht jeder Zelle eine Recheneinheit zur Verfügung gestellt, sondern mehrere Zellen teilen sich eine Recheneinheit.

Bisher wurden die genannten Architekturen für die Simulation lokaler Agenten verwendet [Hal08, HH07]. Diese nutzen aber nicht die Eigenschaften des Globalen Zellularen Automaten, sondern agieren auf Zellularen Automaten mit lokaler Nachbarschaft. Auf Grund der ressourcenintensiven Realisierung war die Zellanzahl sowie die Skalierbarkeit meist stark begrenzt. Die Wahl der möglichen Parameter hat zudem großen Einfluss auf die Leistungsfähigkeit der jeweiligen Architektur. Für die Simulation von MAS können andere Eigenschaften von Bedeutung sein. Die Wahl des Verbindungsnetzwerks einer Multiprozessorarchitektur kann in Bezug auf die Simulation von MAS ganz andere Strukturen aufweisen. Ebenso stellen MAS eine ganz andere Klasse von Anwendungen dar. Üblicherweise beinhaltet bei Zellularen und Globalen Zellularen Automaten jede Zelle einen oder mehrere Werte. Für MAS gibt es leere Zellen, also Zellen, die in diesem Zustand u. U. gar keinen Einfluss auf die Berechnung haben. Berücksichtigt man derartige Unterschiede, ergeben sich andere Optimierungsansätze für die Architekturen.

5.2 Compute Unified Device Architecture

Unter der Compute Unified Device Architecture (CUDA) versteht man eine Technik der Firma NVIDIA [NVI10c], um moderne Grafikprozessoren (engl.: Graphics Processing Unit, GPU) für weitere Berechnungen zu verwenden. Das Ziel ist, die Anwendung durch die zur Verfügung stehende Leistung des Grafikprozessors zu beschleunigen. Dies geht unter der Verwendung von Direct3D oder OpenGL[3] prinzipiell auch ohne CUDA, ist aber mit einer komplexeren Programmierung verbunden. Eine detaillierte Beschreibung von CUDA ist u. a. in [NVI10b, NVI09] gegeben.

CUDA wurde 2007 von der Firma NVIDIA vorgestellt. Damit kann die Leistung von modernen Grafikprozessoren (engl.: Graphics Processing Unit, GPU) auch für allgemeine Anwendungen und Berechnungen verwendet werden. Das Hauptziel von CUDA ist die Beschleunigung wissenschaftlicher und technischer Berechnungen. Diese Leistungssteigerung wird durch das SIMD-Prinzip erreicht. Durch eine Vielzahl an Recheneinheiten wird ein Befehl auf mehrere Daten gleichzeitig angewandt. Um die Leistung der Grafikprozessoren nutzen zu können, muss die Anwendung allerdings neu programmiert bzw. angepasst werden. Damit ist die Anwendung für die Ausführung auf der GPU optimiert und kann nicht mehr auf jedem Computer ausgeführt werden [SSH08, NVI09]. Für eine möglichst schnelle Beschleunigung muss sich der Algorithmus gut parallelisieren lassen. Des weiteren müssen die Daten in die zur Verfügung stehenden Zwischenspeicher passen, um langwierige Speicherzugriffe zu vermeiden. Das Speichermodell der Architektur muss dazu beachtet werden. Für die Ausführung allgemeiner Anwendungen

[3] Direct3D und OpenGL sind Programmierschnittstellen für die Entwicklung von Computergrafik

und Berechnungen auf Grafikprozessoren hat sich der Begriff *General Purpose Graphics Processing Unit* (GPGPU) geprägt.

Zellulare Automaten werden seit dem Aufkommen einfach programmierbarer Grafikkarten auf diesen implementiert und untersucht [vSv10, Kal09]. In jüngster Zeit wird dazu häufig CUDA verwendet. Bei der Umsetzung Zellularer Automaten fällt auf, dass die Lokalität der Nachbarschaft für die Umsetzung eine bedeutende Rolle spielt. Durch diese Lokalität lassen sich die Eigenschaften der Grafikprozessoren ideal ausnutzen, da die Zellen der Reihe nach aus den Speichern gelesen werden können und die Nachbarzellen gleichzeitig zur Verfügung stehen. Aus diesem Grund wurden auch viele eindimensionale Zellulare Automaten umgesetzt [vSv10]. Bei der Verwendung mehrdimensionaler Zellularer Automaten, u. a. mit einer zusätzlich größeren Nachbarschaft, ist die Umsetzung weniger leistungsstark [TJL04]. Es ist demnach zu erwarten, dass eine Umsetzung des allgemeinen GCA-Modells auf Grafikprozessoren noch komplexer und weniger leistungsstark sein wird. Da die Nachbarzellen dynamisch und in jeder Generation neu bestimmt werden, ist ein effizientes Laden der Nachbarzellen allgemein vermutlich nicht möglich. Für einzelne, speziell angepasst und optimierte Anwendungen sind dennoch Leistungssteigerungen möglich [vSv10, Kal09, TJL04].

Eine Umsetzung von GCA-Anwendungen auf CUDA wurde in [MB11] gezeigt. Dazu wurde das Bitonische Mischen und eine Simulation der Partikeldiffusion implementiert. Für das Bitonische Mischen werden Beschleunigungen um den Faktor 232 und 147 im Vergleich zu einer Softwareimplementierung auf einem Prozessor bzw. einer parallelen Implementierung auf vier Rechenkernen erreicht. Für die Simulation der Partikeldiffusion lagen die Beschleunigungen mit den Faktoren 40 und 13 geringer als für das Bitonische Mischen. Ein Grund könnte in der Zugriffsstruktur der beiden Anwendungen liegen. Während das Bitonische Mischen eine regelmäßige Zugriffsstruktur aufweist, existiert bei der Simulation der Partikeldiffusion ein unregelmäßiges Muster. Dieses unregelmäßige Muster entsteht, da eine Nachbarzelle aus einer Menge von 12 möglichen Nachbarzellen zufällig ausgewählt wird. Im Vergleich zu einer optimierten Umsetzung auf einem FPGA aus [JEH08b] wird eine Beschleunigung um den Faktor 1,5 erreicht. Ein Problem, das bei der Umsetzung mit CUDA entsteht, ist, dass es nach Angaben der Autoren keine geeignete Synchronisationsfunktion gibt. In [KCS+09] werden verschiedene Synchronisationsmethoden, nicht nur für CUDA, aufgeführt. Die `__syncthreads()` Methode kam demnach nicht zum Einsatz und die Generationssynchronisation des GCA-Modells wurde auf andere Weise umgesetzt. In [MB11] wurde daher immer nur eine Generation berechnet und die Synchronisation über den Prozessor abgewickelt. Diese Vorgehensweise, die Umsetzung der Generationssynchronisation durch die implizite globale Barriere, wurde auch in [KCS+09] angewendet.

Die auf CUDA implementierten Algorithmen erhalten häufig große Beschleunigungsfaktoren im Vergleich zu einer üblichen Implementierung auf Prozessoren. Deshalb wird die Implementierung von aufwändigen und rechenintensiven Algorithmen bevorzugt auf Grafikprozessoren implementiert. Somit sind die meisten Anwendungen speziell auf die Architektur der Grafikprozessoren angepasst und nutzen die Leistungsfähigkeit der Grafikprozessoren optimal aus. In [LKC+10] wurden Vergleiche zwischen Implementierungen auf Grafikprozessoren unter der Verwendung von CUDA und der Implementierung auf Prozessoren angestellt. Die Implementierung auf den Prozessoren wurde ebenfalls auf die Hardware optimiert. Bei diesen Vergleichen

kam ein NVIDIA GTX280 Grafikprozessor und ein Intel Core i7 960 zum Einsatz. Es wurden 14 verschiedene Anwendungen verglichen, wobei der Grafikprozessor im Allgemeinen nur eine Beschleunigung um den Faktor 2,5 erreichte.

Die Umsetzung eines zweidimensionalen Zellularen Automaten mit Moore-Nachbarschaft auf Grafikprozessoren zur Phasentrennung (engl.: phase separation process) wurde in [Kal09] vorgenommen. Die Phasentrennung wurde mit einer einfachen Zellregel ermittelt. Der neue Zustand einer Zelle bestimmt sich durch

$$C_i = \begin{cases} 0, & S = 5 \text{ or } S < 4 \\ 1, & \text{sonst} \end{cases}$$

wobei S die Summe der Nachbarzellen (inklusive der Zelle selbst) ist. Dabei wurden auch die Effekte der unterschiedlichen Speicherhierarchien untersucht. Im Vergleich zu einem Prozessor (Core2, ein Core) sind Beschleunigungen um den Faktor 2,36 bis 42,72 zu erreichen. Der Beschleunigungsfaktor hängt dabei stark von dem verwendeten Speicher (global oder gemeinsam) ab. Unter Verwendung des globalen Speichers sind nur geringe Beschleunigungen zu erreichen, wohingegen unter der Verwendung des gemeinsamen Speichers höhere Beschleunigungen erzielt werden können. Ein weiteres Ergebnis der Untersuchung ist, dass einzelne zufällige Speicherzugriffe sehr lange dauern, bis zu 10 mal länger als reguläre Speicherzugriffe [Kal09, Seite 8]. Eine Implementierung mit regulären Speicherzugriffen erreicht dadurch eine höhere Performance.

In [TJL04] wird ebenfalls auf die Auswirkungen der Nachbarschaft auf eine effiziente Implementierung von Zellularen Automaten auf Grafikprozessoren eingegangen. Die Autoren beschreiben die begrenzte Nachbarschaft, die für eine effiziente Implementierung von Zellularen Automaten auf Grafikprozessoren genutzt wird, als unzureichend.

> *„They implemented the CML[4] on early programmable graphics hardware and achieved speedups of about 25x over a roughly equivalent CPU implementation. However, their simulations required sampling only the four direct nearest neighbors. For many CA simulations, this is insufficient; for example, cells may query a non-fixed radius of neighbors. The resulting large and varying neighborhoods eliminate some of the simplicity exploited in Harris et al.'s GPU implementation." [TJL04, Seite 1]*

Die Verwendung des GCA-Modells löst die Problematik der lokalen Nachbarschaft, erschwert aber eine effiziente und allgemeine Umsetzung auf vorhandenen Hardwarearchitekturen, z. B. Grafikprozessoren, und erfordert andere Strategien für die Umsetzung. Für die Umsetzung des Gerhard Modells [GST90] zur Simulation von sich ausbreitenden Wellen erreichten die Autoren von [TJL04] eine Beschleunigung unter Verwendung eines Grafikprozessors um den Faktor 7 bis 50, abhängig von der jeweiligen Implementierung. Dafür verwendeten sie eine Nachbarschaft von 7×7 Zellen. Das Fehlen von Operatoren für effiziente bitweise Manipulationen erschwert zudem die Umsetzung des CA-Modells.

[4] Anmerkung: CML (engl.: coupled map lattice)

In [SN09b, SN09a] wurde CUDA für die Verkehrssimulation eingesetzt. Durch eine entsprechende Implementierung wird somit eine Beschleunigung der Anwendung im Vergleich zu einer Simulation auf einem Prozessor erreicht. In [CMSS07] werden mehrere Anwendungen (Verkehrssimulation, Wärmesimulation, K-Means) mit CUDA realisiert. Dabei konnte eine Beschleunigung bis zum 40-fachen einer CPU[5]-Implementierung erreicht werden. Mit Hilfe von CUDA wurden schon über 2 Millionen Agenten mit einer Auflösung von 256x1024 simuliert [DLR07]. Dabei wurden über 50 Aktualisierungen pro Sekunde erreicht. Die Autoren bemängeln allerdings, dass die Programmierung sehr umständlich ist, betrachten diese deshalb als nicht trivial und kommen zu dem Ergebnis, dass neue Algorithmen benötigt werden.

5.3 Mehrkernprozessoren

Unter einem Mehrkernprozessor (auch Multicoreprozessor) versteht man einen Prozessor, der mehrere vollwertige Recheneinheiten besitzt und damit die parallele Ausführung von mehreren Anwendungen unterstützt. Die Ausführung auf einem Prozessor scheint zunächst keine effiziente Simulation von MAS zu erlauben. Häufig wird für die Ausführung rechenintensiver und parallelisierbarer Probleme CUDA (Abschnitt 5.2) herangezogen und damit teilweise enorme Beschleunigungen in der Ausführung erreicht. Ein direkter Vergleich zwischen mehreren verschiedenen Anwendungen optimiert für die Ausführung auf einem Mehrkernprozessor sowie für die Ausführung auf einer Grafikkarte zeigt jedoch, dass im Allgemeinen der Grafikprozessor lediglich 2,5 mal schneller als die Ausführung auf einem Mehrkernprozessor ist [LKC+10]. Als Gründe für die teilweise enormen Beschleunigungsfaktoren werden fehlerhafte Vergleiche und unoptimierter Code angeführt.

> *„However, we found that with careful multithreading, reorganization of memory access patterns, and SIMD optimizations, the performance on both CPUs and GPUs is limited by memory bandwith and the gap is reduced to only 5X.“ [LKC+10, Seite 458]*

Für die Implementierung und parallele Ausführung von Anwendungen, damit auch für die Implementierung des GCA-Modells für die Simulation von MAS, kommen verschiedene Techniken in Frage. Die Umsetzung der Generationssynchronisation kann durch die in [KCS+09] aufgeführten Mechanismen umgesetzt werden. So steht z. B. in Java5 [GJSB05] durch `CyclicBarrier` eine expliziete Barriere zur Verfügung. Wie effizient derartige Synchronisationsmethoden umgesetzt sind oder umgesetzt werden können, ist unterschiedlich. In den folgenden Abschnitten werden die verschiedenen Techniken bzw. Umgebungen erläutert, die für die Umsetzung von GCA/CA auf Mehrkernprozessoren verwendet werden können.

5.3.1 Open Multi-Processing

Die Open Multi-Processing Programmierschnittstelle (OpenMP) [Ope97, Ope08] wird seit 1997 entwickelt und soll auf Systemen mit gemeinsamem Hauptspeicher Anwendung finden. Hierfür wurden spezielle Befehle definiert, die auf Threadebene die Abarbeitung parallelisieren.

[5] Central Processing Unit, Hauptprozessor

Damit nutzen die Programme Multiprozessorsysteme oder Multicoreprozessoren aus. [Ope97] beschreibt den Einsatz von OpenMP:

> *„The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.“ [Ope97]*

Der Vorteil von OpenMP ist, dass bereits bestehender Code einfach parallelisiert und damit beschleunigt werden kann. Der Grad der Parallelität ist dabei durch die Anzahl der Rechenkerne der verwendeten CPU abhängig. Dies bedeutet auch, dass ein entsprechender Prozessor vorhanden sein muss, um überhaupt parallele Abarbeitung zu ermöglichen. OpenMP muss nicht zwangsläufig zu einer Beschleunigung führen. Die Hardware, hauptsächlich Prozessor und Hauptspeicher, sind unverändert und unter Umständen unpassend für die parallele Ausführung. Eine Umsetzung von GCA/CA unter der Verwendung von OpenMP ist prinzipiell möglich. Der Vorteil ist die einfache und schnelle Umsetzung von GCA/CA, allerdings ist der Grad der Parallelität durch die verwendete Hardware (Prozessor, Speicher) begrenzt.

Für die Multi-Agenten-Simulation kann OpenMP durchaus eingesetzt werden. Der Parallelitätsgrad ist dabei durch den Prozessor eingeschränkt. Ein weiteres Problem ist in der Speicheranordnung zu sehen. Bei aktuellen Computersystemen gibt es meistens einen Hauptspeicher. Bei einer ungünstigen Datenanordnung und vielen Rechenkernen in einer CPU kann dies zu einer Verlangsamung der Simulation führen. Die Speicherbandbreite reicht dann nicht mehr aus, um alle Rechenkerne mit den erforderlichen Daten zu versorgen.

Ein Beispiel für die Umsetzung von Algorithmen mit OpenMP, darunter auch eine Multi-Agenten-Simulation, wird in [DVT00] gezeigt. Für das parallele Programmierparadigma haben die Autoren als regelmäßiges Berechnungsproblem die Umsetzung von künstlichen neuronalen Netzen und als unregelmäßiges Berechnungsproblem ein MAS entwickelt. In diesem MAS, bestehend aus Robotern, Minen, Fabriken und Hindernissen, müssen die Roboter auf einem quadratischen Gitternetz Eisenerz von einer Mine zu einer Fabrik transportieren. Die Bewegung der Roboter wird über ein Potenzialfeld, das die Entfernungen kodiert, gesteuert. Das MAS ist nach Aussage der Autoren das komplexere System. Die Anwendung ist auf Grund der Agenten dynamisch. Für die Parallelisierung können die Agenten und die Umgebung gewählt werden. Probleme bei der Parallelisierung bereiten die unterschiedlichen Aktionen der Roboter. Ein Roboter, der sich nicht bewegt und keine Aktion auswählt, benötigt wenig Rechenzeit, während er für andere Aktionen viel Rechenzeit benötigt. Eine effiziente Aufteilung ist schwer zu finden und am ehesten über eine dynamische Zuordnung zu erreichen. Um eine schnelle Simulation zu erreichen, sollte die komplette Umgebung in den Cache des Prozessors passen. In [DVT00] wurde eine 512×512 große Welt mit 4096 Agenten gewählt. Bezüglich der Programmierbarkeit und Verwendbarkeit von OpenMP für MAS ist das folgende Zitat aufgeführt.

> *„For irregular applications the higher-level of OpenMP leads to difficulties in programming. Sometimes, the use of foreign parallel functions is necessary, so we think that OpenMP still remains limited to regular computations.“ [DVT00, Seite 7]*

Ein komplettes Simulationsframework für MAS wird in [RH10] vorgestellt. Obwohl hierbei der Fokus nicht auf einer optimierten Simulation liegt, wurde auch dieser Punkt diskutiert [RH10,

Seite 4]. Die Autoren sehen die Multi-Agenten-Simulation als sehr aufwändige Berechnungsaufgabe an. Als Optimierungen wird die parallele Simulation mit Threads und eine räumliche Partitionierung der Welt vorgeschlagen. Datenabhängigkeiten müssen jedoch, wie z. B. bei der Kollisionsbehandlung, berücksichtig werden. Als erstes und einfaches Mittel wurde die Berechnung mit OpenMP parallelisiert. Um den Bedingungen an Echtzeitanwendungen zu genügen, erwarten die Autoren eine Aktualisierungsrate von 25 Hz. Damit können ohne Optimierungen bis zu 730 Agenten und mit der OpenMP Parallelisierung auf einem Prozessor mit zwei Rechenkernen 1.000 Agenten simuliert werden. Dies entspricht einem Beschleunigungsfaktor von etwa 1,37. Mit weiteren Optimierungen, u. a. zur Vermeidung unnötiger Berechnungen, können bis zu 2.000 Agenten simuliert werden. Für die Umsetzung des Verhaltens der Agenten kam dazu ein endlicher Automat (engl.: Finite State Machine, FSM) zum Einsatz. Damit wurde das Verhalten von Personen umgesetzt und verschiedene Szenarien, z. B. Evakuierung von Räumen, simuliert.

5.3.2 Java Threads auf Mehrkernprozessoren

Durch die seit etwa 2006 im Endverbrauchermarkt zunehmende Verbreitung von Mehrkernprozessoren besteht die Möglichkeit, einen Standard PC zur Lösung einzusetzen. Die Zellen werden dabei auf die Kerne k des Prozessors verteilt und parallel berechnet. Die Zellregel wird dann in Software ausgeführt. Jeder Prozessor erhält dabei, wie bei den Multiprozessorarchitekturen, einen Teil des gesamten Zellfeldes. Die in der Hardware vorhandenen Prozessoren werden durch Softwarethreads ersetzt bzw. simuliert. Jeder Thread τ berechnet hierbei einen Teil des gesamten Zellfeldes. Durch den Einsatz von Mehrkernprozessoren werden die Threads parallel ausgeführt und somit wie bei der Multiprozessorarchitektur eine Leistungssteigerung erreicht.

Die Leistungsfähigkeit der Implementierung mit Threads wurde anhand von vier Testfällen (Tabelle 5.2) untersucht und bewertet. Diese wurden auf einem Intel Xeon E5335 (Codename: Clovertown) [Int07] mit 2 GHz und 4 GB DDR2 RAM ermittelt. Hier stehen vier Rechenkerne zur Verfügung, womit vier Threads echt parallel ausgeführt werden können. Das dabei simulierte Verhalten entspricht dem aus Abschnitt 6.6.4.2. Die Ergebnisse der vier Testfälle sind in Tabelle 5.3 aufgeführt. Die Umsetzung erfolgte in der Programmiersprache Java [GJSB05].

Testfall	Zellanzahl	Agentenanzahl	Generationen	Rand
A	2.048	185	20.000	ja
B	409.600	40.704	20.000	ja
C	409.600	20.000	20.000	ja
D	409.600	200.000	20.000	ja

Tabelle 5.2: Testfälle der Agentenwelt für die Softwaresimulation

Für die Testfälle ist für acht Threads keine Beschleunigung mehr zu erreichen. Dies liegt zum einen daran, dass der verwendete Prozessor nur vier Rechenkerne besitzt, also auch nur vier Threads echt parallel ausführen kann. Zum anderen erhöht sich mit der Anzahl der Threads auch der Kommunikationsaufwand unter den Threads. Da nicht mehr als vier Threads echt parallel ausgeführt werden können, ist somit, im Vergleich zu vier Threads, nur der Kommuni-

Testfall	τ	Zeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$	AUR $\left[\frac{1}{ms}\right]$
A	1	1.157	-	35.401	3.197
A	2	875	1,32	46.811	4.228
A	4	1.140	1,02	35.929	3.245
A	8	2.125	0,54	19.275	1.741
B	1	255.375	-	32.078	3.187
B	2	143.547	1,78	57.068	5.671
B	4	72.375	3,53	113.188	11.248
B	8	75.766	3,37	108.122	10.744
C	1	224.828	-	36.436	1.779
C	2	109.375	2,06	74.898	3.657
C	4	62.219	3,61	131.663	6.428
C	8	71.125	3,16	115.177	5.623
D	1	376.547	-	21.755	10.622
D	2	189.359	1,99	43.261	21.123
D	4	108.109	3,48	75.775	36.999
D	8	111.000	3,39	73.801	36.036

Tabelle 5.3: Ergebnisse der Agentensimulation mit τ Threads für die Testfälle A bis D auf einem Intel Xeon E5335

kationsaufwand erhöht worden. Eine Beschleunigung ist deshalb nicht möglich.

Die Testfälle B bis D wurden alle auf einer gleichgroßen Agentenwelt simuliert. Lediglich die Anzahl der Agenten wurde variiert. Diese zeigen die Auswirkung der Agentenanzahl auf die Ausführungszeit sowie die Prozessorauslastung.
Direkten Einfluss auf den Grad der Parallelität haben:

- Anzahl der Threads (τ)
- Die Anzahl der zu simulierenden Generationen (g)
- Die Größe der Agentenwelt (n)
- Die Komplexität der Zellregel

Um die maximale Parallelität, d. h. 100% Prozessorauslastung zu erreichen, müssen folgende Bedingungen erfüllt sein:

- Die Anzahl der auszuführenden Threads τ muss größer gleich der Anzahl der Prozessorkerne k sein ($\tau \geq k$).
- Die Feldgröße muss einer bestimmten Mindestgröße entsprechen und/oder die anzuwendende Zellregel besitzt eine bestimmte Mindestkomplexität (hierzu zählt auch die Anzahl der Generationen, die zu simulieren sind).

Um eine möglichst große Parallelität erreichen zu können, muss die Datenmenge pro Thread ausreichend groß sein. Ist dies nicht der Fall, dann ist der Aufwand für die Generationssynchronisationen zu groß und es bleibt zu viel Rechenzeit ungenutzt. Des weiteren steigt die Anzahl der

externen Lesezugriffe immer weiter an. Der Anteil der Ausführungszeit, der für die eigentlichen Berechnungen verwendet wird, wird immer geringer. Liegen genügend Daten bzw. Zellen vor, kann es durchaus angebracht sein, mehr Threads zu verwenden, als Rechenkerne vorhanden sind. Es können dann zwar nicht alle Threads mit der maximalen Performance parallel ausgeführt werden, aber die Wartezeit eines Threads (externes Lesen, Generationssynchronisation) kann durch Berechnungen eines anderen Threads ausgenutzt werden.

Die Testfälle A und B wurden zusätzlich noch auf einem HP ProLiant DL385 G7 Performance Rack Server mit zwei Opteron 6174 Prozessoren ausgeführt. Hier stehen zwei Prozessoren mit jeweils zwei Dies, die sechs Rechenkerne beinhalten, zur Verfügung. Insgesamt können 24 Rechenkerne verwendet werden. Damit lässt sich die Anzahl der Threads τ um ein Vielfaches erhöhen und das Verhalten hinsichtlich der parallelen Ausführung genauer untersuchen. Die Ergebnisse für die Testfälle A und B sind in der Tabelle 5.4 aufgeführt. Bei der Ausführung wurde jeder Thread einem Rechenkern zugeordnet, sofern noch freie Rechenkerne zur Verfügung standen. Während für den Testfall A nur für zwei Threads eine leichte Performancesteigerung erreicht werden konnte, wurden für mehr als zwei Threads nur langsamere Ausführungszeiten erreicht. Im Vergleich zum Testfall B, bei dem eine weitaus größere Agentenwelt verwendet wurde, zeigt sich, dass für den Testfall A die Zellanzahl zu gering ist, um durch eine parallele Ausführung eine Geschwindigkeitssteigerung erreichen zu können. Für den Testfall B ergibt sich bis 16 Threads ein Speedup von 12,58. Bei mehr als 16 Threads ergab sich dagegen keine weitere Geschwindigkeitssteigerung. Die Auslastung der CPUs während der Ausführung ist in Abbildung 5.1 dargestellt. Auf Grund des hohen Kommunikationsaufwands zum Lesen von Zellen und der häufigen Generationssynchronisation wird keine volle Auslastung der Rechenkerne erreicht.

Testfall	τ	Zeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$	AUR $\left[\frac{1}{ms}\right]$
A	1	961	-	42.622	3.850
A	2	932	1,03	43.948	3.969
A	4	1.328	0,72	30.843	2.786
A	8	2.420	0,40	16.925	1.528
A	16	4.846	0,20	8.452	763
A	32	10.050	0,10	4.075	368
A	64	22.830	0,04	1.794	162
B	1	228.566	-	35.840	3.561
B	2	104.815	2,18	78.156	7.766
B	4	53.458	4,28	153.241	15.228
B	8	28.706	7,96	285.375	28.359
B	16	18.174	12,58	450.753	44.793
B	32	33.625	6,80	243.628	24.210
B	64	36.503	6,26	224.419	22.301

Tabelle 5.4: Ergebnisse der Agentensimulation des Testfalls A und B mit Threads

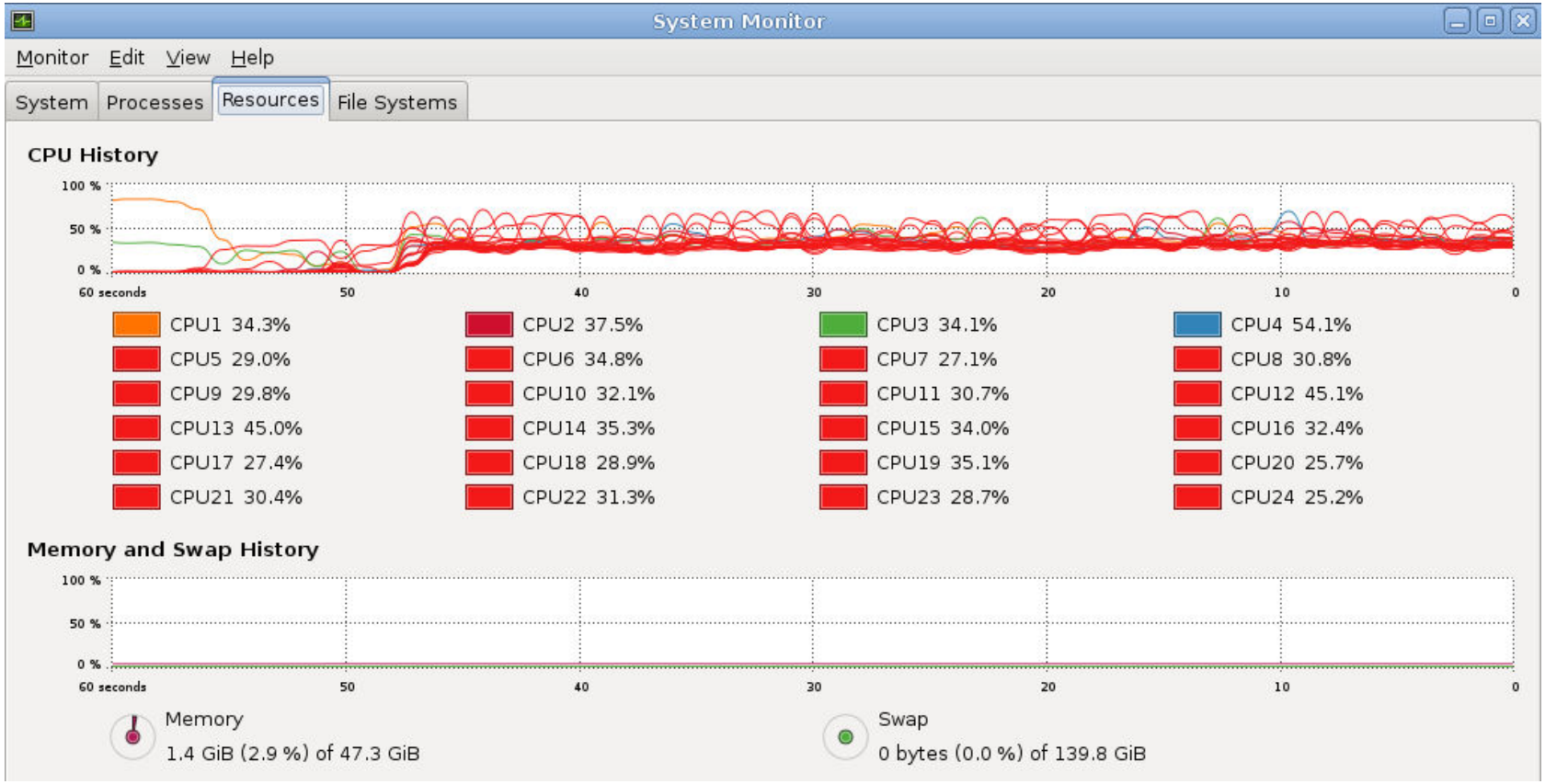


Abbildung 5.1: CPU Auslastung während der Ausführung der Agentensimulation auf zwei Opteron Prozessoren mit insgesamt 24 Rechenkernen

5.4 Message Passing Interface

Das Message Passing Interface (MPI) [Mes09] bietet, im Gegensatz zu OpenMP, die Möglichkeit, die Ausführung der Zellregel auf mehrere Rechensysteme zu verteilen. Dadurch kann der Grad der Parallelität einfach gesteigert werden. Die Kommunikation, d. h. Lesezugriffe auf Nachbarzellen, die sich auf einem anderen System befinden, müssen über ein Netzwerk abgewickelt werden. Je nach Umsetzung der GCA/CA sind die Lesezugriffe zu langsam und es kann gar keine Beschleunigung erreicht werden. Da GCA/CA's feingranulare Architekturen erfordern bzw. sind, eignen sich derartige Umsetzungen nur, wenn der Kommunikationsaufwand gering ist oder eine schnelle Kommunikation über das Netzwerk möglich ist. Eine testweise Implementierung in Java [GJSB05] unter Verwendung von zwei und vier vernetzten Computern bestätigte diese Vermutung. Zu einem ähnlichen Resultat kommen u. a. auch die Autoren von [CCH11] und [APS10].

In [CCH11] wird auf die spezifischen Eigenschaften von Multi-Agenten-Systemen auf verteilten Ressourcen eingegangen. Teilweise können demnach gute Beschleunigungen erreicht werden. Bedingung dafür ist aber, dass zwischen den Prozessen keine oder nur wenige Interaktionen stattfinden.

> *„One of the earlier researches in accelerating MAS is based on cluster with speedups between 11 and 14 using a cluster of 16 workstations [PVR+03]. But the power of clusters resides in high speed computing with non-interactive workloads, making it less efficient when the interaction among agents occupies a significant part of system overhead, which is common in MAS.“ [CCH11]*

Auf die Problematik der Umsetzung von Multi-Agenten-Systemen wird auch in [APS10] eingegangen. Dazu schreiben die Autoren:

> *„For example, threads within a block of NVIDIA's Common Unified Data Architecture (CUDA) have very fast access to a shared memory segment, whereas Message Passing Interface (MPI)-based communication across GPU nodes typically consumes hundreds of microseconds.*
> *The challenge is compounded by the fact that computation within each agent's state update in an ABMS[6] can be very fine-grained, taking little more than a few microseconds. When states are decomposed across the hierarchies, synchronization across time-stepped updates to the partitioned states can become a significant source of overhead.“ [APS10, Seite 1]*

Die dennoch großen Beschleunigungsraten, die in [APS10] erreicht werden, sind auf den dort verwendeten Ansatz zurückzuführen. Die Aktionen der Agenten auf einem zellularen Feld werden dort nur lokal behandelt. Somit lässt sich das zellulare Feld in gleichgroße Teile zerlegen. Dadurch entsteht nur an den Rändern der einzelnen Teile ein Kommunikationsaufwand. Die Kommunikation zwischen den Zellen innerhalb eines dieser Teile wird lokal umgesetzt. Diese Möglichkeit der Aufteilung des zellularen Feldes besteht nur im CA-Modell. Das GCA-Modell ermöglicht zwar auch diese Aufteilung des zellularen Feldes, da aber eine globale Kommunikation

[6] Anmerkung: agent-based model simulations

zwischen allen Zellen möglich ist, kann der Kommunikationsaufwand nicht auf die Grenzen der einzelnen Teile begrenzt werden. Eine derartige Umsetzung eignet sich demnach nur für GCA/CA mit lokaler Kommunikation.

5.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Möglichkeiten für die Umsetzung von Zellularen Automaten und Globalen Zellularen Automaten dargestellt. Die dabei erzielten Ergebnisse sind teilweise sehr unterschiedlich. So lassen sich mit Implementierungen auf Grafikkarten gute Ergebnisse erzielen. Allerdings sind die Ergebnisse von der Zugriffsstruktur abhängig und damit für Globale Zellulare Automaten schlechter als für Zellulare Automaten geeignet.

Mit GCA-Spezialarchitekturen, z. B. realisiert auf einem FPGA, können auch für Globale Zellulare Automaten gute Ergebnisse erzielt werden. Die Architekturen können für das Modell optimiert und die Zugriffsstrukturen beim Entwurf der Architekturen berücksichtigt werden.

Umsetzungen von GCA/CA auf Mehrkernprozessoren sind sehr leicht auch für größere zellulare Felder zu realisieren. Die erreichbaren Beschleunigungen hängen von vielen Faktoren ab. Zum einen, wie auch bei Grafikkarten, von der Zugriffsstruktur, zum anderen auch von der Größe des zellularen Feldes. Zugriffe auf den Hauptspeicher verlangsamen die Ausführung. Die Speicherhierarchie muss deshalb bei der Implementierung berücksichtigt werden. Auch ist die Skalierbarkeit eingeschränkt. Mit zunehmender Anzahl an Rechenkernen steigt der Kommunikationsaufwand stark an. Die Rechenkerne können dann nicht mehr optimal genutzt werden und werden durch die langsame Kommunikation blockiert.

Eine Umsetzung von GCA/CA auf verteilten Rechensystemen ist in den meisten Fällen, vor allem aber für Globale Zellulare Automaten, nicht zu empfehlen. Die Kommunikation ist im Vergleich zu Netzwerken innerhalb eines Chips zu langsam, wodurch schon ein geringer Kommunikationsaufwand die Leistung des Gesamtsystems stark mindert.

Nach dieser einleitenden Übersicht über die verschiedenen Möglichkeiten zur Umsetzung eines Multi-Agenten-Systems möchte ich das Kapitel zusammenfassend mit einem Zitat aus [APS10], das die Anforderungen an ein Multi-Agenten-System bzw. an eine dedizierte Hardwareplattform stellt, abschließen.

> *„A solution is needed to simultaneoulsy address the challenges of latency spectrum, hierarchical organization as well as heterogeneity. Ideally, a single, unified, parameterized solution would be useful that can be easily instantiated, customized, and auto-tuned for any given, specific compound computational platform instance.“ [APS10, Seite 1]*

6 GCA-Multiprozessorarchitekturen für Multi-Agenten-Systeme

In diesem Kapitel werden verschiedene Realisierungen von Multiprozessorarchitekturen für das GCA-Modell (Abschnitt 2.3) vorgestellt. Einleitend wird die grundlegende Systemstruktur der Multiprozessorarchitekturen diskutiert und die in den darauf folgenden Kapiteln verwendeten Begriffe definiert und erläutert. Für alle diese Multiprozessorarchitekturen ist es zwingend notwendig, ein Verbindungsnetzwerk einzusetzen, um damit den externen Lesezugriff des GCA-Modells zu realisieren. Aus diesem Grund verwenden alle im Folgenden vorgestellten Netzwerke eine einheitliche Schnittstelle und werden in einem eigenen Kapitel erläutert. In den darauf folgenden Kapiteln werden dann die gesamten Architekturen für das GCA-Modell beschrieben. Eine theoretische Auswertung und Bewertung der Netzwerke befindet sich in Abschnitt 6.4. Die Implementierung der Netzwerke befindet sich im Anhang (Abschnitt 9.8). Die Hardwarearchitekturen wurden in der Hardwarebeschreibungssprache Verilog HDL [IEE04] umgesetzt.

Architektur	Netz-werk	Block-anzahl	FPGA [Abschnitt]	Anwendung [Abschnitt]	zusätzliche Hardware	Anmerkung
Einarmige universelle GCA-Architektur (MPA)	ON	-	Cyclone II [6.6.2.1]	Bitonisches Mischen [6.6.2.2]	-	Auswertung der Wartezyklen
	ON	-	Stratix II [6.6.2.1]	Bitonisches Mischen [6.6.2.2]	-	
	ONR	-	Cyclone II [6.6.3.1]	Bitonisches Mischen [6.6.3.2]	-	
	ONR	-	Stratix II [6.6.3.1]	Bitonisches Mischen [6.6.3.2]	-	
	ONP	-	Cyclone II [6.6.4.1]	Bitonisches Mischen [9.4]	-	
	RN	-	Cyclone II [6.6.4.1]	Bitonisches Mischen [9.4]	-	
	RNFF	-	Cyclone II [6.6.4.1]	Bitonisches Mischen [9.4]	-	
	BDPA	-	Cyclone II [6.6.4.1]	Bitonisches Mischen [9.4]	-	
	BRRA	-	Cyclone II [6.6.4.1]	Bitonisches Mischen [9.4]	-	
	ONR	-	Cyclone II [6.6.3.1]	Bitonisches Sortieren [6.6.3.3]	-	Vergleich mit Quicksort
	ONR	-	Cyclone II [6.6.3.1]	Bitonisches Sortieren [6.6.3.3]	-	
	RN	-	Cyclone II [6.6.4.1]	Bitonisches Sortieren [9.4]	-	
	RNFF	-	Cyclone II [6.6.4.1]	Bitonisches Sortieren [9.4]	-	
	BDPA	-	Cyclone II [6.6.4.1]	Bitonisches Sortieren [9.4]	-	
	BRRA	-	Cyclone II [6.6.4.1]	Bitonisches Sortieren [9.4]	-	
	ONR	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	Auswertung
	ONP	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	der Netzwerke
	RN	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	auf dem FPGA
	RNFF	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	
	BDPA	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	Verwendung der
	BRRA	-	Cyclone II [6.6.4.1]	Agentenwelt [6.6.4.2]	-	`AUTO_READ` Funktionen
Multiarmige speicher-konfigurierbare universelle GCA-Architektur (MPAB)	BDPA	2	Cyclone II [6.7.4.1]	Verkehrssimulation (CA) [6.7.5.3]	LFSR	
	BDPA	4	Cyclone II [6.7.4.1]	Verkehrssimulation (GCA) [6.7.5.3]	LFSR	
	ONR	7	Stratix II [6.7.4.2]	Mehrkörperproblem [6.7.6]	Gleitkommaeinheit	
	ONUR	7	Stratix II [6.7.4.2]	Mehrkörperproblem [6.7.6]	Gleitkommaeinheit	Zufallszahlen, optional
	RN	7	Stratix II [6.7.4.2]	Mehrkörperproblem [6.7.6]	Gleitkommaeinheit	Gleitkommaeinheit
	BDPA	7	Stratix II [6.7.4.2]	Mehrkörperproblem [6.7.6]	Gleitkommaeinheit	
Spezialisierte GCA-Architektur mit Hashfunktionen (HA)	RN	2	Cyclone II [6.8.2]	Agentenwelt [6.8.3]	-	aktive, inaktive Zellen
	RNFF	2	Cyclone II [6.8.2]	Agentenwelt [6.8.3]	-	aktive, inaktive Zellen
	BDPA	2	Cyclone II [6.8.2]	Agentenwelt [6.8.3]	-	aktive, inaktive Zellen
	BRRA	2	Cyclone II [6.8.2]	Agentenwelt [6.8.3]	-	aktive, inaktive Zellen
Agentenbasierte speicheroptimierte GCA-Architektur (DAMA)	RNFF	2	Cyclone II [6.9.3]	Agentenwelt [6.9.4]	Vier-Nachbarn	
	BDPA	2	Cyclone II [6.9.3]	Agentenwelt [6.9.4]	Vier-Nachbarn	
	RNFF	3	Cyclone II [6.9.3]	Informationsverteilung [6.9.5]	Vier-Nachbarn	
	RNFF	3	Cyclone II [6.9.3]	Informationsverteilung [6.9.5]	Vier-Nachbarn	zusätzlich versch. Anzahl von Agenten ausgewertet
	RNFF	2	Cyclone II [6.9.6]	Agentenwelt [6.9.6]	Vier-Nachbarn	Optimierung der Taktfrequenz
	RNFF	2	Stratix V [6.9.7]	Agentenwelt [6.9.7]	Vier-Nachbarn	Skalierbarkeit der Architektur
	RNFF	2	Stratix V [6.9.8]	Agentenwelt [6.9.8]	Vier-Nachbarn	Simulation unterschiedlicher Anzahl von Agenten
Gegenüberstellung der Architekturen	-	-	[6.10]	[6.10]	-	Hardware und Java Multithread im Vergleich
	-	-			-	

6.1 Systemstruktur der Multiprozessorarchitekturen

Die Multiprozessorarchitekturen, die das GCA-Modell realisieren, verwenden den Ansatz der Parallelverarbeitung. Das Gesamtproblem wird in mehrere kleinere Probleme zerteilt und diese dann parallel auf jeweils einer eigenen Prozessoreinheit verarbeitet. Die Teillösungen werden am Ende zur Gesamtlösung zusammengefügt. Der Prozess des Aufteilens und Zusammenfügens kann auch mehrfach während der Abarbeitung erfolgen. Dies ist im GCA-Modell der Fall. In jeder Generation wird von jeder Prozessoreinheit ein Teil des Gesamtproblems gelöst. Die Aufteilung ist hierbei impliziet durch das Modell gegeben. Da im GCA-Modell keine Schreibzugriffe auf Nachbarzellen erlaubt sind, muss keine Behandlung von Schreibkonflikten erfolgen. Dies erleichtert und beschleunigt zusätzlich die Ausführung von Algorithmen, da während der Ausführung keine Ausnahmefälle auftreten können und nur für jeden Generationswechsel synchronisiert werden muss.

Grundsätzlich bestehen die Multiprozessorarchitekturen aus p NIOS II/f Prozessoren. Jeder NIOS II/f Prozessor hat einen eigenen Programmspeicher. In dem Programmspeicher ist die auszuführende Zellregel enthalten. Jeder NIOS II/f Prozessor ist zudem einem Datenspeicher zugeordnet. Jeder Datenspeicher enthält dabei einen Teil des gesamten Zellfeldes. Jeder der Prozessoren kann nur auf den ihm zugeordneten Speicher schreibend zugreifen. Lesezugriff besteht grundsätzlich auf alle Datenspeicher, entweder intern oder extern. Ein *interner Lesezugriff* ist gegeben, wenn ein Prozessor lesend auf den ihm zugeordneten Speicher zugreift. Ein *externer Lesezugriff* ist ein Lesezugriff auf einen Datenspeicher, der einem anderen Prozessor zugeordnet ist. Derartige Lesezugriffe werden über ein Verbindungsnetzwerk abgewickelt, das alle Prozessoren und Datenspeicher miteinander verbindet. Es besteht immer die Möglichkeit, einen internen Lesezugriff durch einen externen Lesezugriff zu realisieren, nicht aber umgekehrt. Diese Möglichkeit kann bei der Programmierung der Zellregel ausgenutzt werden, um diese möglichst einfach zu halten. Ein externer Lesezugriff benötigt jedoch immer mehr Taktzyklen als ein interner Lesezugriff. Eine derartige Ersetzung der Lesezugriffe zur Vereinfachung der Zellregel ist nur angebracht, wenn die Anzahl der ersetzten Lesezugriffe sehr gering ist. Ansonsten empfiehlt es sich, die Zugriffe anhand der zu lesenden Adresse in externe und interne Lesezugriffe zu unterscheiden. Diese Möglichkeit besteht auf Softwareebene innerhalb der Zellregel, aber auch auf Hardwareebene.

Für die Programmierung der Zellregel wurden dem Standardbefehlssatz des Prozessors daher zwei spezielle Befehle hinzugefügt. Ein Befehl setzt die internen Lesezugriffe um, der zweite die externen Lesezugriffe. Für die Verwendung der Befehle gelten die folgenden Regeln:

- Der Befehl für interne Lesezugriffe ist zu verwenden, wenn für alle Lesezugriffe nur der zugeordnete Speicher verwendet wird.
- Der Befehl für externe Lesezugriffe ist zu verwenden, wenn mindestens ein Lesezugriff auf einen anderen als den zugeordneten Speicher verwendet wird.

Abhängig von der Architektur und von den erzielten Ergebnissen wurden noch weitere spezielle Befehle hinzugefügt. Diese sind bei der jeweiligen Hardwarearchitektur aufgeführt und erklärt.

Für die im Folgenden vorgestellten Multiprozessorarchitekturen wird der Begriff der *Verarbeitungseinheit* (VE) verwendet. Eine Verarbeitungseinheit beschreibt dabei mehrere Komponenten (u. a. Prozessor, Speicher) der Architektur. Je nach Architektur enthält eine Verarbeitungseinheit weitere Komponenten. Die gesamte Architektur wird dann aus mehreren Verarbeitungseinheiten gebildet. Welche Komponenten zu einer Verarbeitungseinheit gezählt werden, ist für jede Architektur einzeln definiert. Da allerdings jede Verarbeitungseinheit immer einem Prozessor zugeordnet ist, werden die Begriffe Verarbeitungseinheit und Prozessor teilweise synonym verwendet. Wird z. B. von der Anzahl der Prozessoren p gesprochen, so ist damit die Anzahl der Verarbeitungseinheiten gemeint. Diese Synonyme Verwendung der Begriffe steht auch im Einklang mit der Literatur.

Für die Auswertung der Architekturen wird neben der erzielbaren Beschleunigung bei der Ausführung von GCA-Anwendungen und Multi-Agenten-Simulationen auch deren Platzbedarf ermittelt. Auf dem Cyclone II wird der Platzbedarf in Logikelementen (engl.: Logic Elements, LE) angegeben [Alta]. Für das Stratix II wird der Platzbedarf in ALMs (engl.: Adaptive Logic Module) und ALUTs (engl.: Adaptive Look-up Table) angegeben [Alth]. Details zu den verwendeten FPGAs sind in Abschnitt 9.7 aufgelistet.

6.2 Verteilung der Zellen auf die Verarbeitungseinheiten

Die Agentenwelten werden auf Hardwarearchitekturen, die das GCA-Modell ganz oder in Teilen umsetzen, simuliert. Die Hardwarearchitekturen sind dabei als Multiprozessorarchitekturen mit unterschiedlichsten Ausprägungen umgesetzt. Für die Aufteilung der Zellen auf die einzelnen Prozessoren gibt es verschiedenste Aufteilungsmöglichkeiten. Die gebräuchlichsten Aufteilungsmöglichkeiten sind in Abbildung 6.1 dargestellt.

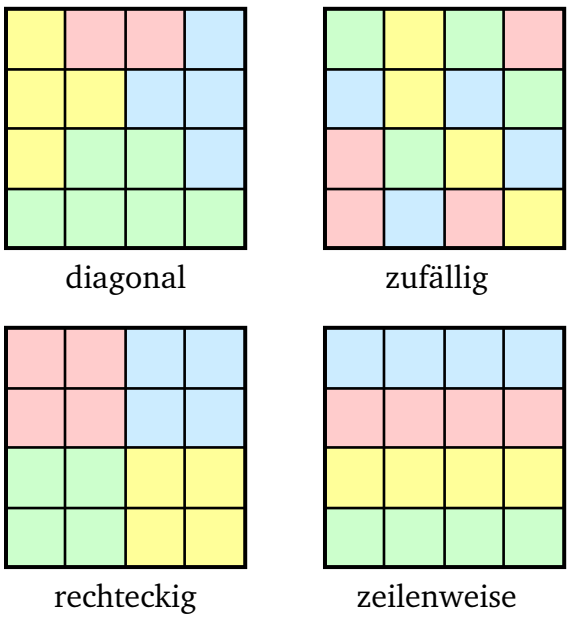


Abbildung 6.1: Mögliche Verteilungen der Zellen auf die Verarbeitungseinheiten (VE 1 , VE 2 , VE 3 , VE 4)

Am einfachsten und anschaulichsten lässt sich das Zellfeld rechteckig oder zeilenweise auf die Verarbeitungseinheiten aufteilen. Ebenso denkbar ist eine diagonale oder zufällige Verteilung der Zellen auf die Verarbeitungseinheiten. Die Art der Verteilung der Zellen auf die Verarbeitungseinheiten hat direkt Einfluss auf die Geschwindigkeit der Simulation. Auf Grund der Aufteilung können die Lesezugriffe auf andere Verarbeitungseinheiten minimiert, aber auch Kollisionen im Netzwerk minimiert werden. Die Verteilung ist aber auch von der Anwendung, also dem Verhalten der Agenten abhängig. Eine allgemeine und optimale Verteilung kann demnach also nicht bestimmt werden. Für bestimmte Gruppen von Agentenanwendungen, z. B. mit lokalen Bewegungsmustern, sind rechteckige bzw. zeilenweise Verteilungen besser geeignet. Zusätzlich muss betrachtet werden, ob eine Agentenwelt mit oder ohne Weltrand verwendet wird. Hierbei entfallen bei geeigneter Wahl der Verteilung Zugriffe auf andere Verarbeitungseinheiten, wenn z. B. davon ausgegangen wird, dass Agenten sich nicht über den Weltrand hinausbewegen können.

Die Hardwarearchitektur an sich muss ebenfalls betrachtet werden. Kommt eine Architektur zum Einsatz, die für die Simulation von Multi-Agenten-Systemen optimiert ist, können sich andere Zugriffsmuster ergeben.

Grundsätzlich ist jede Permutation der Zellen auf die Verarbeitungseinheiten denkbar. Für die praktische Umsetzung einer Hardwarearchitektur sollte diese im Bezug auf geringe Hardwareressourcen und geringe Systemkomplexität gewählt werden.

6.3 Verbindungsnetzwerke

Als Verbindungsnetzwerke für die Verbindung der Verarbeitungseinheiten untereinander kommen diverse Netzwerke in Frage. Über diese Netzwerke werden die externen Lesezugriffe realisiert, indem alle Prozessoren und alle Speicher der Verarbeitungseinheiten mit dem Netzwerk verbunden sind.
Die Netzwerke werden in dieser Arbeit allgemein in die folgenden drei Klassen eingeordnet:

- applikationsspezifisch
- vollkommen allgemein
- allgemein mit applikationsspezifischen Teilen

applikationsspezifisch: Die applikationsspezifischen Netzwerke stellen nur Verbindungen zwischen je zwei Zellen zur Verfügung, die auch von der ausgeführten Anwendung verwendet werden. Jede realisierte Verbindung wird von der ausgeführten Anwendung mindestens einmal verwendet. Alle Verbindungen, die von der Anwendung nie verwendet werden, werden in der Architektur nicht realisiert.

vollkommen allgemein: Die vollkommen allgemeinen Netzwerke stellen alle Verbindungen zwischen je zwei Zellen auf gleichwertige Weise zur Verfügung. Es sind alle Verbindungen realisiert, unabhängig davon, ob diese von der ausgeführten Anwendung verwendet werden. Alle benötigten Verbindungen sind auf die gleiche Art und Weise realisiert.

allgemein mit applikationsspezifischen Teilen: Die allgemeinen Netzwerke mit applikationsspezifischen Teilen verbinden die Möglichkeiten der beiden voran genannten Netzwerkklassen. Es können, wie bei den vollkommen allgemeinen Netzwerken, alle Verbindungen zwischen je zwei Zellen hergestellt werden. Zusätzlich dazu ist es möglich, bestimmte Zugriffskombinationen besonders gut zu realisieren. Gut bedeutet, dass die Verbindungen schneller oder ohne Zugriffskonflikte parallel hergestellt werden können. Bewegen sich z. B. die Agenten nur lokal, reicht es aus, den Zugriff nur auf benachbarte Zellen zur Verfügung zu stellen. Ein weiter entfernter Zugriff ist dann immer noch möglich, aber mit einer größeren Zugriffszeit verbunden.

In [Reg87, Seite 91-114] wird die Bedeutung von Kommunikations- und Verbindungsnetzwerken hervorgehoben. Diese sind demnach besonders für massiv parallele Systeme von Bedeutung. Die Verbindungsstruktur wird als neue Komponente der Rechnerarchitektur angesehen, da die Aktivitäten der einzelnen Einheiten (meist Prozessoren) aufeinander abgestimmt werden müssen. Bei den Netzwerken unterscheidet man zwischen statischen und dynamischen. Unter statischen Netzwerken versteht man solche, die eine feste Zuordnung zwischen den Eingängen und den Ausgängen haben. Bei dynamischen Netzwerken kann über Steuersignale die Zurodnung zwischen Eingängen und Ausgängen verändert werden und somit verschiedene Verbindungen hergestellt werden. Können nicht mehrere Verbindungen gleichzeitig hergestellt werden, wird dies Blockierung genannt. So können zwei Prozessoren nicht gleichzeitig über einen gemeinsam genutzten Bus Zugriffe abwickeln, weshalb einer der beiden Zugriffe blockiert ist.

6.3.1 Die Netzwerkschnittstelle

Alle im Folgenden vorgestellten Netzwerke für die GCA-Architekturen verwenden eine einheitliche Schnittstelle für die Verbindung der Prozessoren und der Speicher mit den Netzwerken (Abb. 6.2, 6.3).

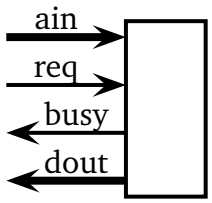


Abbildung 6.2: Übersicht über die Netzwerkschnittstelle der Prozessoren

Für jede Verarbeitungseinheit existieren als Netzwerkeingangssignale eine Requestleitung (req) und eine Adressleitung (ain). Als Netzwerkausgangssignale existieren eine Busyleitung (busy) und eine Datenleitung (dout). Die Netzwerkschnittstelle der Prozessoren ist in Abbildung 6.2 dargestellt. Die Requestleitung zeigt dem Netzwerk an, dass die anliegende Adresse gültig ist und dass die Anfrage noch nicht bearbeitet wurde. Das Busysignal zeigt dem Prozessor an, ob die Anfrage fertig bearbeitet wurde und damit auch, ob die Daten auf der Datenleitung gültig sind. Die Bitbreite der Daten und Adressleitung sind an die Größe der vorhandenen Speicher angepasst. Die obersten Bits der Adressleitung wählen über das Netzwerk den Zielspeicher aus, die unteren Bits die jeweilige Zelle.

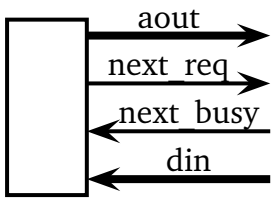


Abbildung 6.3: Übersicht über die Netzwerkschnittstelle der Speicher

Für jeden Speicher einer Verarbeitungseinheit existieren als Netzwerkeingangssignale eine Nextbusyleitung (next_busy) und ein Dateneingang (din), als Netzwerkausgangssignale eine Adressleitung (aout) und eine Nextrequestleitung (next_req). Die Netzwerkschnittstelle der Speicher ist in Abbildung 6.3 dargestellt. Zwischen dem Speicher und dem Netzwerk existiert ein Speichercontroller, der die Steuerleitungen verarbeitet. Die Nextrequestleitung zeigt dem Speichercontroller an, dass die anliegende Adressleitung gültig ist und das dazugehörige Datum aus dem Speicher gelesen werden soll. Über die Nextbusyleitung zeigt der Speichercontroller dem Netzwerk an, dass die anliegenden Daten gültig sind und der Lesevorgang somit abgeschlossen ist.

6.3.2 Das Busnetzwerk

Das Busnetzwerk verbindet alle Prozessoren und alle Speicher über einen gemeinsamen Bus (die Implementierung des Busnetzwerks befindet sich im Anhang Abschnitt 9.8.1). Dies bedeutet, dass keine parallelen Zugriffe möglich sind, dafür aber jeder einzelne Zugriff schnell abgearbeitet werden kann. Die Problematik des Busnetzwerkes liegt zum einen in der Realisierung auf einem FPGA und zum anderen in einer fairen und schnellen Arbitrierung. Das Fehlen von Tristatepuffern im FPGA erfordert die Umsetzung des Busnetzwerks mit Multiplexern. Als Arbitrierungsmöglichkeiten wurden eine dynamische Arbitrierung (engl.: Bus Dynamic Prioritized Arbitration, BDPA) und eine Round-Robin (engl.: Bus Round Robin Arbitration, BRRA) Arbitrierung untersucht.

6.3.2.1 Dynamische Arbitrierung (BDPA)

Bei der dynamischen Arbitrierung (BDPA) werden die ID's[1] aller Verarbeitungseinheiten, die ein aktives Requestsignal gesetzt haben, über einen Vergleicherbaum (Abb. 6.4) ausgewertet. Die Verarbeitungseinheit mit der höchsten aktiven ID erhält die Berechtigung, den Lesezugriff über das Netzwerk durchzuführen. Führt eine Verarbeitungseinheit mehrere aufeinanderfolgende Zugriffe aus, so können diese schneller abgearbeitet werden, da die Arbitrierung in diesem Fall schon die entsprechende Verarbeitungseinheit ausgewählt hat. Es ist dabei nicht wichtig, wie schnell hintereinander diese Zugriffe erfolgen, sondern nur, dass diese Zugriffskette nicht von einer anderen Verarbeitungseinheit unterbrochen wird.

[1] Jeder Verarbeitungseinheit wird eine eindeutige ID (engl.: Identification) zugewiesen.

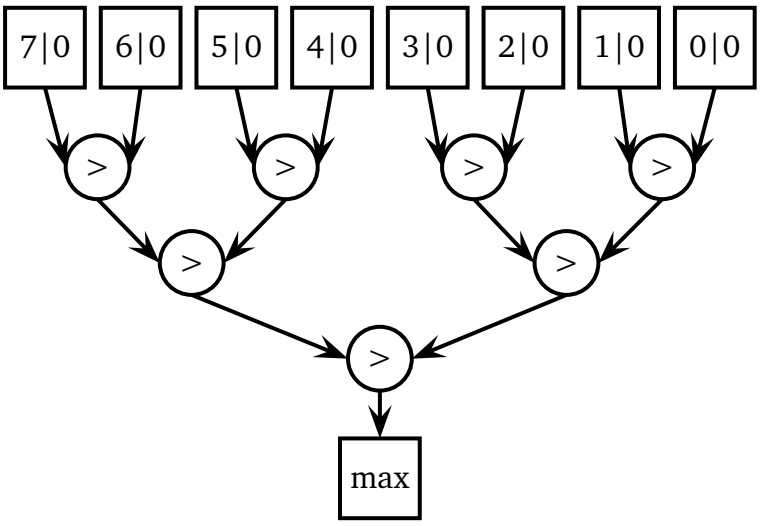


Abbildung 6.4: Vergleicherbaum der dynamischen Busarbitrierung für acht VE

Bei der dynamischen Arbitrierung können einzelne Zugriffe schnell über das Netzwerk realisiert werden. Sobald die Anzahl an Zugriffen steigt, besteht die Gefahr, dass die Verarbeitungseinheiten mit einer niedrigen ID länger warten müssen. Es ist jedoch auf Grund der Generationssynchronisation sichergestellt, dass alle Verarbeitungseinheiten konsistente Daten lesen und auch nicht komplett verhungern.

6.3.2.2 Round-Robin Arbitrierung (BRRA)

Der Ansatz der Round-Robin Arbitrierung (BRRA) ist im Gegensatz zur dynamischen Arbitrierung fairer. Die Arbitrierung durchläuft der Reihe nach alle Verarbeitungseinheiten. Erreicht die Arbitrierung eine Verarbeitungseinheit mit einer ausstehenden Leseanfrage, so wird diese abgearbeitet. Danach wird mit der nächsten Verarbeitungseinheit fortgefahren. Somit ist sichergestellt, dass die Verarbeitungseinheiten der Reihe nach Leseanfragen ausführen können.

6.3.3 Das Ringnetzwerk

Das Ringnetzwerk verbindet alle Verarbeitungseinheiten über in Ringform angeordnete Registerstufen (die Implementierung des Ringnetzwerks befindet sich im Anhang Abschnitt 9.8.3). Die Datenpakete werden pro Taktzyklus im Ring von einer Registerstufe zur nächsten Ringstufe weitergeleitet. Somit sind parallele Zugriffe auf die Speicher anderer Verarbeitungseinheiten möglich. Damit eine neue Anfrage auf die Registerstufe der Verarbeitungseinheit gestellt werden kann, muss diese zu diesem Zeitpunkt leer sein. Somit entstehen für die Anfrage u. U. Wartezyklen. Begrenzt man die Anzahl der Anfragen pro Verarbeitungseinheiten auf eins, sind damit per Definition Deadlocks im Ringnetzwerk ausgeschlossen. Anderenfalls wären andere Mechanismen notwendig. Im GCA-Modell ist nicht vorgesehen, dass eine Anfrage gestellt wird bevor die vorherige Anfrage abgearbeitet wurde. Dies ist vermutlich auf die Tatsache zurückzuführen, dass die Zugriffe aus Speicherzugriffen bestehen. Im Folgenden werden zwei Implementierungen von Ringnetzwerken vorgestellt.

6.3.3.1 Standard Ringnetzwerk (RN)

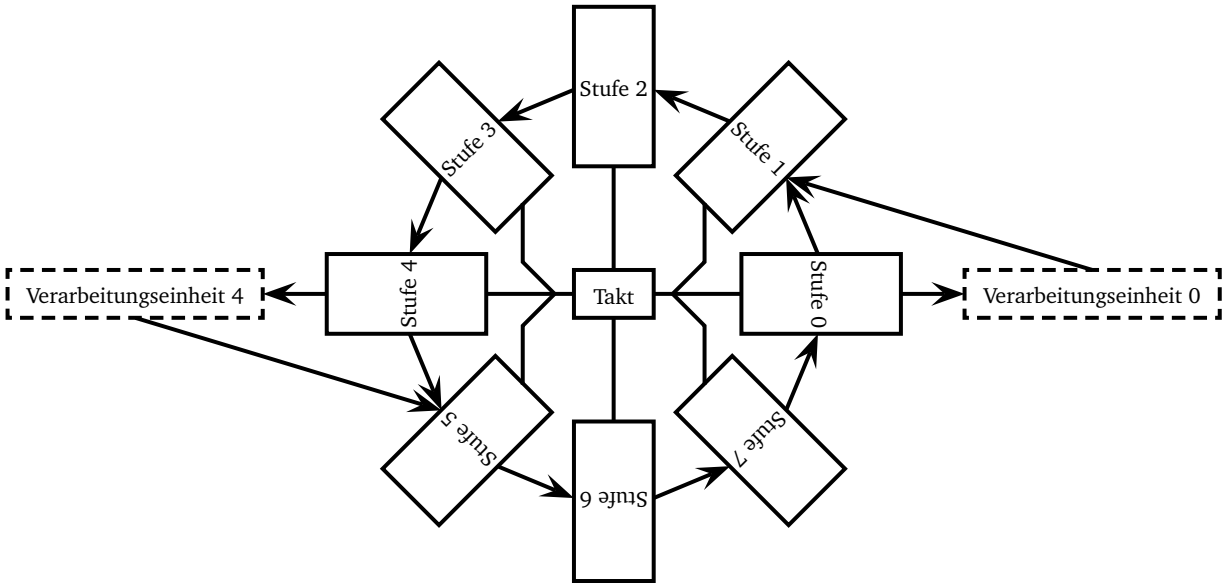


Abbildung 6.5: Standard Ringnetzwerk (RN) für acht Verarbeitungseinheiten mit Lesezugriff von Verarbeitungseinheit 0 auf Verarbeitungseinheit 4

Das Ringnetzwerk (engl.: Ring Network, RN), dargestellt in Abbildung 6.5, verbindet alle Verarbeitungseinheiten über als Ring angeordnete Registerstufen. Jede Verarbeitungseinheit besitzt eine eigene Registerstufe im Ringnetzwerk. In jedem Takt werden die Daten im Ring um eine Registerstufe weitertransportiert. Dabei überprüft jede Verarbeitungseinheit, ob die Daten in der ihr zugeordneten Registerstufe vom Ring entfernt werden müssen. Dies trifft sowohl für Speicheranfragen anderer Verarbeitungeinheiten als auch für Datenantworten anderer Verarbeitungseinheiten zu. Anfragen einer Verarbeitungseinheit werden auf die nachfolgende Registerstufe geschrieben. Jede Verarbeitungseinheit kann zu jedem Zeitpunkt maximal eine Leseanfrage stellen. Somit sind bei n Verarbeitungseinheiten mit n Registerstufen maximal n Anfragen auf dem Ring. Deadlocks sind somit nicht möglich. Das Ringnetzwerk ermöglicht den Verarbeitungseinheiten, das Netzwerk parallel zu verwenden, indem die Datenanfragen und Datenantworten in den Registerstufen gespeichert werden. Die Registerstufen ermöglichen gleichzeitig eine hohe Taktfrequenz für das Netzwerk, indem die langen Datenpfade unterbrochen sind. Allerdings muss dafür jede Datenanfrage, inklusive der Datenantwort, einmal durch den gesamten Ring laufen. In Sonderfällen können Datenanfragen auch mehrfach durch den Ring laufen bis sie abgearbeitet werden können.

6.3.3.2 Ringnetzwerk mit Forwarding (RNFF)

Das Ringnetzwerk mit Forwarding (engl.: Ring Network Fast Forward, RNFF), dargestellt in Abbildung 6.6, ist eine Erweiterung zum Standard Ringnetzwerk aus Abschnitt 6.3.3.1. Durch Hinzufügen weiterer Verbindungen und durch Auslassung einer Registerstufe können Datenanfragen und Datenantworten schneller zur Zielverarbeitungseinheit transportiert werden. Dies ist vor allem bei einer großen Anzahl an Registerstufen nützlich. Das Einfügen weiterer Verbindun-

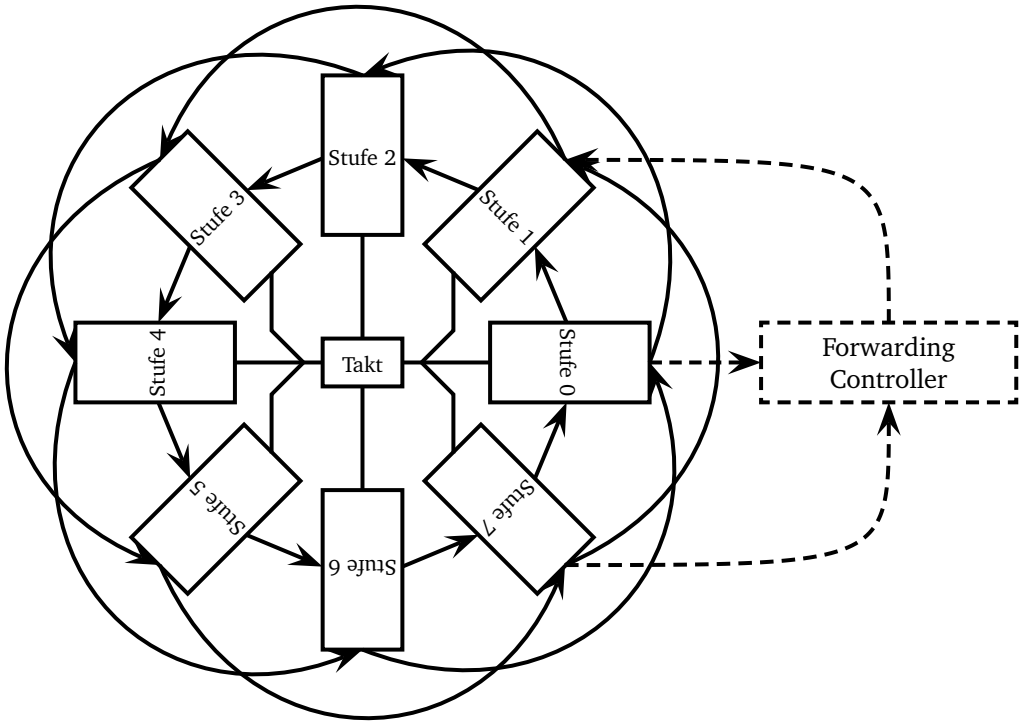


Abbildung 6.6: Ringnetzwerk mit Forwarding (RNFF) und Forwarding Controller der Verarbeitungseinheit 0

gen und das Auslassen von Registerstufen erfordert auf jeder Registerstufe einen Forwarding Controller. Der Controller hat die Aufgabe, das Überspringen einer Registerstufe zu steuern und Blockierungen im Netzwerk zu erkennen. Durch das Vorreichen kann es bei bestimmten Anfragekombinationen dazu führen, dass keine der Anfragen an ihr Ziel gelangt. Deshalb wird das Vorreichen im gesamten Ring für eine Runde deaktiviert, falls der Controller Datenpakete erkennt, die den Ring ein zweites mal durchlaufen. Die Abbildung 6.6 zeigt das Ringnetzwerk mit Forwarding und den Controller für eine Registerstufe. Die Verarbeitungseinheiten sind aus Gründen der Übersichtlichkeit weggelassen.

6.3.4 Das Omeganetzwerk

Das Omeganetzwerk ist ein mehrstufiges, kombinatorisches Netzwerk, das aus einfachen Schaltelementen aufgebaut wird. Dieses wird im Folgenden in verschiedenen Varianten vorgestellt, die für diese Arbeit implementiert wurden. Um auch für eine große Anzahl an Verarbeitungseinheiten eine möglichst hohe Taktfrequenz des Geamtsystems zu erhalten, wurde in einer Variante eine Registerstufe eingefügt. In einer anderen Variante wurde die Möglichkeit von unsynchronisierten Zugriffen hinzugefügt. Eine weitere Variante sieht in jedem Schaltelement Speicher für die Weiterleitung vor. Die verschiedenen Varianten werden im Folgenden detailiert beschrieben.

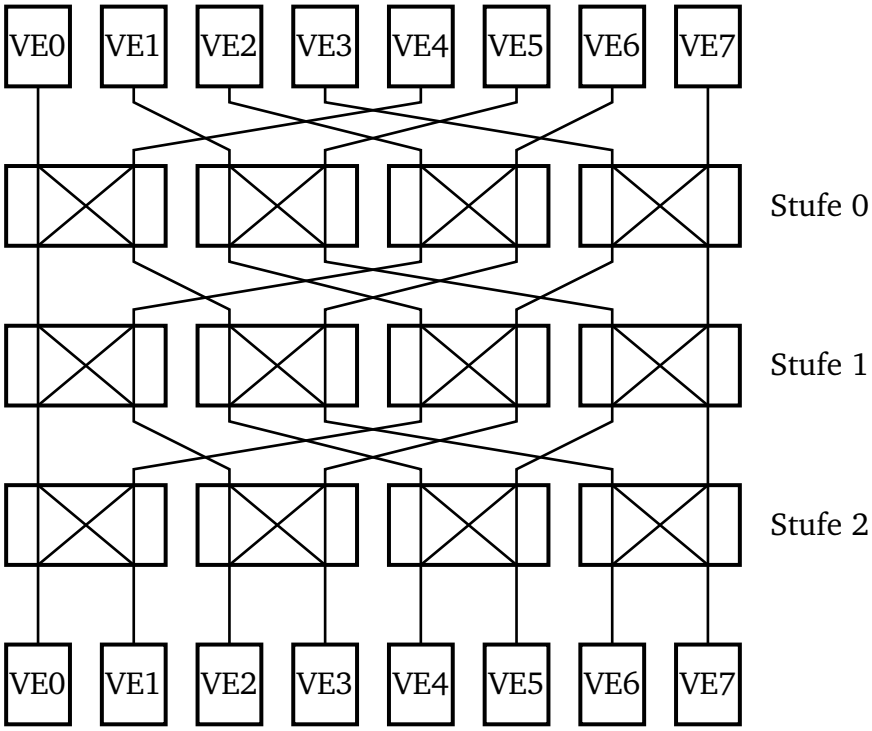


Abbildung 6.7: Das Omeganetzwerk (ON) dargestellt für acht Verarbeitungseinheiten

6.3.4.1 Standard Omeganetzwerk (ON)

Das Omeganetzwerk (engl.: Omega Network, ON), dargestellt in Abbildung 6.7, ist ein mehrstufiges, rein kombinatorisches Netzwerk und benötigt daher keinen globalen Takt (die Implementierung des Omeganetzwerks befindet sich im Anhang Abschnitt 9.8.2). Das Netzwerk besteht aus mehreren Shuffleelementen. Jedes Shuffleelement hat genau zwei Eingänge und genau zwei Ausgänge. Anhand einer mitgeführten Adresse werden die Eingänge mit den Ausgängen verbunden, entweder direkt oder über Kreuz. Als Spezialfall kann auch ein Eingang auf beide Ausgänge geschaltet werden (Broadcast). Werden mehrere dieser Shuffleelemente in mehreren Stufen miteinander verschaltet, ist es grundsätzlich möglich, jeden Eingang mit jedem Ausgang zu verbinden. Es ist jedoch nicht möglich, alle Verbindungspermutationen zeitgleich herzustellen. Die Verschaltung der Shuffleelemente erfolgt nach folgender Formel, bei der jede Stufe den i-ten Eingang mit dem j-ten Ausgang für $p = 2^k$ Prozessoren verbindet:

$$j = \begin{cases} 2 \cdot i & \text{für } 0 \leq i \leq p/2 - 1 \\ 2 \cdot i + 1 - p & \text{für } p/2 \leq i \leq p - 1 \end{cases}$$

Da das Omeganetzwerk ein rein kombinatorisches Netzwerk ist, müssen die Zugriffe über das Netzwerk organisiert werden. Dies ist vor allem von den verwendeten Prozessoren abhängig. Die in diesem Kapitel vorgestellten Architekturen verwenden alle den NIOS II/f Softcoreprozessor [Alte, Altf] von Altera. Die NIOS II Softcoreprozessoren führen als Zellregel kompilierten C-Code aus und besitzen, je nach Ausführung, eine bis zu sechsstufige Pipeline. Dies führt dazu, dass die

Verarbeitungseinheiten bei der Abarbeitung der Zellregel nicht synchron laufen und somit auch die Zugriffe über das Netzwerk nicht synchron stattfinden. Dies macht es bei der Verwendung des Omeganetzwerkes notwendig, die Zugriffe zu synchronisieren, um die Datenkonsistenz zu gewährleisten. Andernfalls könnten anstehende Leseanfragen von höherpriorisierten Leseanfragen unterbrochen werden. Die Folge sind entweder fehlerhafte Berechnungen der Zellen oder ein Deadlock der gesamten Architektur. Interessant ist in diesem Zusammenhang der Einfluss der Synchronisation auf die Ausführungszeit. Die Synchronisation ist über UND-Gatter realisiert, die die Lesezugriffe so lange blockieren, bis alle Prozessoren eine Leseanfrage gestellt haben. Dies bedeutet aber auch, dass alle Prozessoren gleich viele Leseanfragen an das Netzwerk stellen müssen. Das Ausführen unterschiedlicher Zellregeln ist somit erschwert. Die Wartezyklen für die Synchronisation der externen Lesezugriffe und deren Einfluss auf die effiziente Ausführung wurde in Abschnitt 6.6.2.2 für das Bitonische Mischen ausgewertet. Das Omeganetzwerk, wie es auch in [Hee07] vorgestellt wurde, wird als *Standard Omeganetzwerk* verwendet. Alternativ wird es mit einer Registerstufe und schließlich als dritte Version mit FIFO-Elementen (engl.: First In First Out, FIFO) erweitert. Ein paketorientiertes Omeganetzwerk wurde in [BAM+10] realisiert. Dieses benötigt im Vergleich zum Omeganetzwerk weniger Ressourcen.

6.3.4.2 Omeganetzwerk mit Registerstufe (ONR)

Das Omeganetzwerk mit Registerstufe (engl.: Omega Network Register, ONR), dargestellt in Abbildung 6.8, ist eine Erweiterung des Omeganetzwerks (die Implementierung des Omeganetzwerks mit Registerstufe befindet sich im Anhang Abschnitt 9.8.2). Das Omeganetzwerk, als rein kombinatorisches Netzwerk, hat den Nachteil, dass bei einer großen Anzahl an Verarbeitungseinheiten die Pfadlänge im Netzwerk steigt. Ebenso erhöht sich die Anzahl der Shuffleelemente und somit auch die Zugriffszeit jedes Pfades. Für jede Verdopplung der Verarbeitungseinheiten wird eine neue Zeile mit Shufflelementen notwendig. Jeder Pfad von einer Verarbeitungseinheit zu einer anderen Verarbeitungseinheit geht dann durch ein zusätzliches Shuffleelement. Der maximale Gesamttakt verringert sich dadurch.

Das Omeganetzwerk mit Registerstufe setzt an diesem Punkt an. Alle Pfade gehen in der Mitte des Omeganetzwerkes durch eine Registerstufe. Theoretisch ist somit eine doppelt so hohe Taktfrequenz möglich. Die höhere mögliche Taktfrequenz verdoppelt aber auch die Anzahl an Taktzyklen jedes Zugriffs. Eine Beschleunigung gegenüber dem ursprünglichen Omeganetzwerk ist also nur dann möglich, wenn die höhere Taktfrequenz einen größeren Einfluss hat, als die Erhöhung der Taktzyklen.

6.3.4.3 Omeganetzwerk mit unsynchronisierten Lesezugriffen (ONU)

Das in Abschnitt 6.3.4.1 vorgestellte Omeganetzwerk erfordert, dass die Lesezugriffe der Prozessoren an das Netzwerk synchronisiert werden. Um auch unsynchronisierte Lesezugriffe zu ermöglichen, muss sichergestellt sein, dass bestehende Leseanfragen nicht unterbrochen werden können, weshalb eine Zugriffseinheit die Zugriffe auf das Netzwerk kontrolliert. Das Omeganetzwerk ohne Registerstufe mit unsynchronisierten Lesezugriffen wird mit ONU (engl.: Omega Network Unsynchronized) abgekürzt. Wird eine Registerstufe eingesetzt, wird die Abkürzung

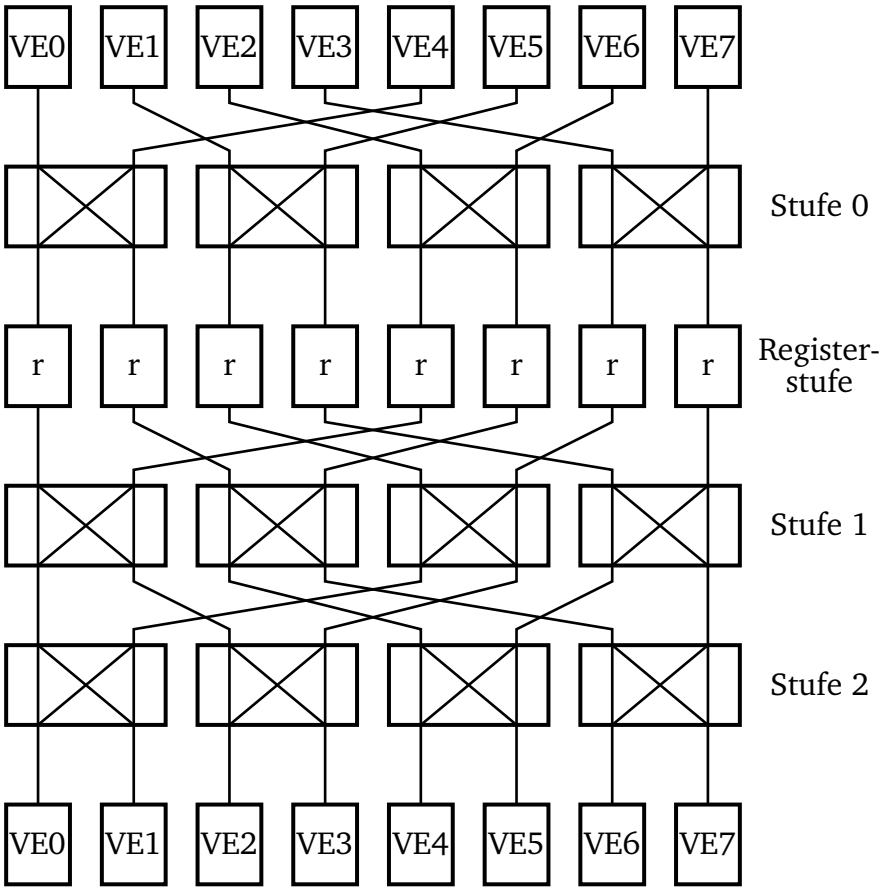


Abbildung 6.8: Das Omeganetzwerk mit Registerstufe (ONR) dargestellt für acht Verarbeitungseinheiten und der Registerstufe (r)

ONUR (engl.: Omega Network Unsynchronized Register) verwendet. Auf Grund der Vorteile der Registerstufe innerhalb des Omeganetzwerks wurde die unsynchronisierte Variante ohne Registerstufe nicht mehr evaluiert.

Die Zugriffseinheit leitet eingehende Leseanfragen an das Netzwerk weiter, sofern zu diesem Zeitpunkt keine Leseanfragen bearbeitet werden. Wird das Netzwerk noch zur Abarbeitung vorher gestellter Leseanfragen verwendet, werden alle neu eingehenden Leseanfragen so lange blockiert, bis das Netzwerk wieder verfügbar ist. Diese Vorgehensweise ermöglicht es jedem Prozessor, unabhängig von den anderen Prozessoren auf das Netzwerk zuzugreifen. Es ist aber auch weiterhin möglich, parallele Zugriffe auszuführen. Dadurch wird auch die Programmierung erleichtert. Es muss keine Rücksicht mehr auf das verwendete Netzwerk genommen werden und die Zellregel muss nicht für das Omeganetzwerk angepasst werden. Somit können über das Netzwerk sowohl einzelne als auch mehrere parallele Anfragen behandelt werden. Die bei parallelen Zugriffen u. U. auftretenden Lesekonflikte werden wie im Omeganetzwerk über priorisierte Eingänge behandelt.

6.3.4.4 Omeganetzwerk mit FIFO-Shuffleelementen (ONP)

Als Weiterentwicklung des Omeganetzwerks mit Registerstufe wurde das Omeganetzwerk mit FIFO-Shuffleelementen (engl.: Omega Network Pipeline, ONP), dargestellt in Abbildung 6.9, implementiert (die Implementierung des Omeganetzwerks mit FIFO-Shuffleelementen befindet sich im Anhang Abschnitt 9.8.4). Anstatt nur eine Registerstufe innerhalb des gesamten Omeganetzwerkes zu realisieren, wird in jedem Shuffleelement eine Registerstufe eingeführt (vgl. Omeganetzwerk mit Registerstufe). Eine Anfrage benötigt nun einen Taktzyklus, um von einem zum darauffolgenden Shuffleelement weitergeleitet zu werden. Dafür werden die einzelnen Pfade in kleine Stücke zerlegt und erlauben eine höhere Taktfrequenz. Durch die Aufteilung der Pfade ist es ungünstig, den kompletten Pfad von einer Verarbeitungseinheit zu einer anderen Verarbeitungseinheit für die Dauer der Anfrage aufrecht zu erhalten. Die Shuffleelemente können deshalb in jedem Takt umgeschaltet werden. Eine Anfrage wird immer nur zur nächsten Stufe weitergereicht, ohne dass der gesamte Pfad aufrecht erhalten wird. Dadurch entfällt die Synchronisation bei Zugriffen auf das Netzwerk und Wartezyklen entfallen. Das Prinzip ähnelt dem der Pipeline in einem Prozessor. Wartezyklen treten nur noch auf, wenn beide Eingänge im gleichen Taktzyklus auf den gleichen Ausgang weitergeschaltet werden sollen. Dies ist nicht möglich, weshalb einer der beiden Eingänge um einen Takt verzögert weitergeleitet wird. Eine ähnliche Realisierung wurde in [AMAD07] vorgenommen. Das Netzwerk wurde allerdings nur simuliert und für die Auswertung die Effekte einer Hardwarerealisierung nicht ausreichend beachtet. Des weiteren ist es möglich, dass Nachrichten verloren gehen, da hier ein paketorientierter Ansatz verwendet wurde. Im Vergleich dazu besitzt das ONP Leitungen, die die einzelnen Stufen auch rückwärtig verbinden. Somit kann von den Ausgängen zu den Eingängen in umgekehrter Richtung die Blockierung des Netzwerks angezeigt werden. Die einzelnen Stufen und letztendlich die Verarbeitungseinheiten senden dann keine weiteren Daten, bis die Blockierung im Netzwerk aufgehoben ist. Diese Realisierung ist komplexer und benötigt daher auch mehr Hardwareressourcen. Es gehen dafür aber keine Informationen im Netzwerk verloren.

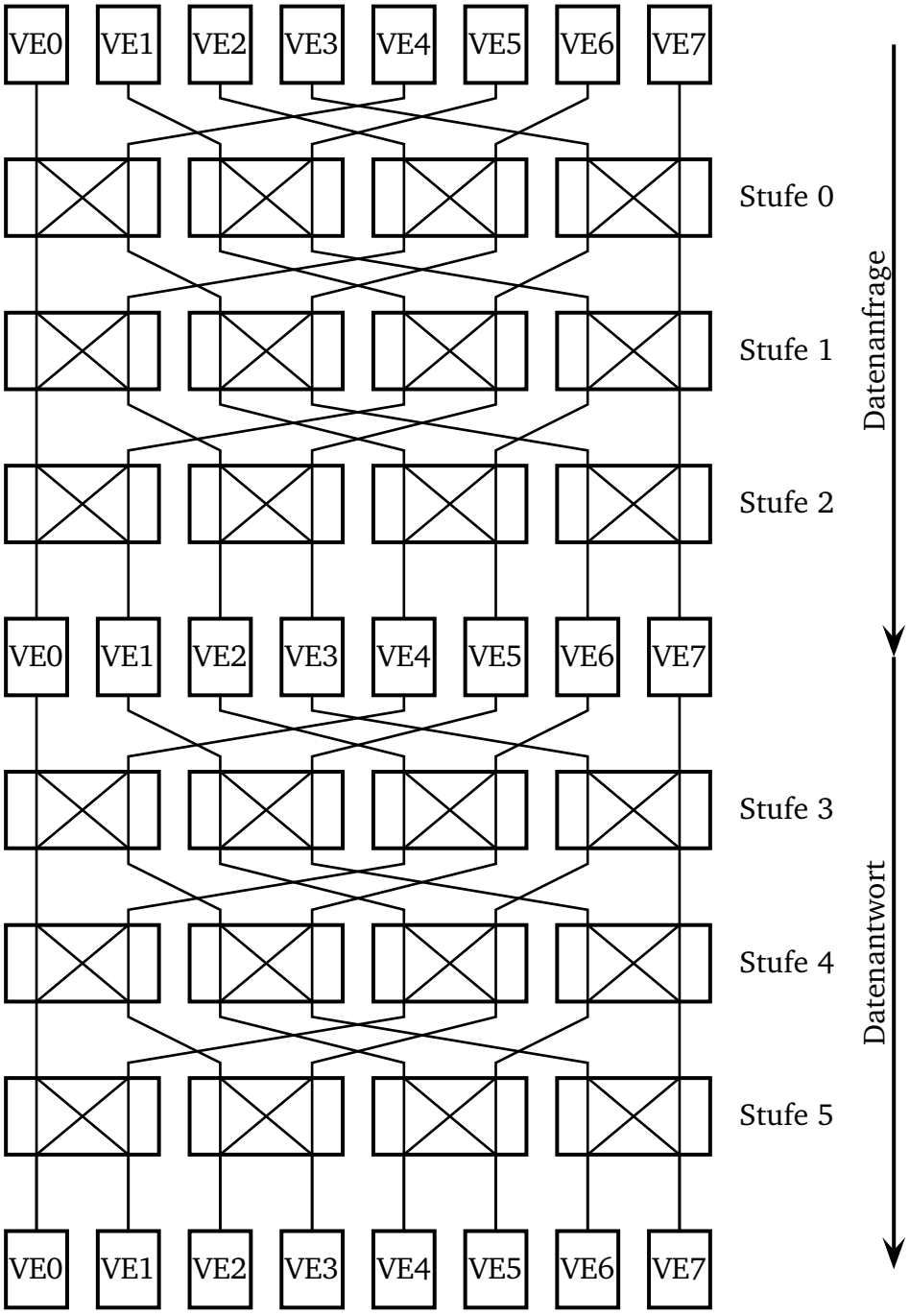


Abbildung 6.9: Das Omeganetzwerk mit FIFO-Shuffleelementen (ONP) dargestellt für acht Verarbeitungseinheiten bestehend aus zwei getrennten Omegenetzwerken für die Datenanfrage und die Datenantwort

Um die angeforderten Daten zurücksenden zu können, ist ein zweites Netzwerk notwendig. Durch die Aufteilung des gesamten Pfades in Segmente steht der Pfad für die Rücksendung der Daten nicht mehr zur Verfügung. Die vorhandenen Pfade können auch nicht ohne weiteres für die Rücksendung der Daten verwendet werden, da jederzeit neue Anfragen durch das Netzwerk laufen können. Für die Rücksendung der Daten wird ein weiteres Omeganetzwerk verwendet (Stufe 3 bis 5 in Abb. 6.9), das vom Aufbau her dem ersten Teil gleicht (Stufe 0 bis 2 in Abb. 6.9). Für die Rücksendung der Daten werden zusätzliche Datenleitungen benötigt. Die Requestleitungen werden durch Acknowledgeleitungen ersetzt. Die Shufflelemente für die Hin- und Rückrichtung sind in Abbildung 6.11 dargestellt. Durch diesen Aufbau ist die Skalierung des Netzwerkes stark eingeschränkt, da zwei Netzwerke benötigt werden.

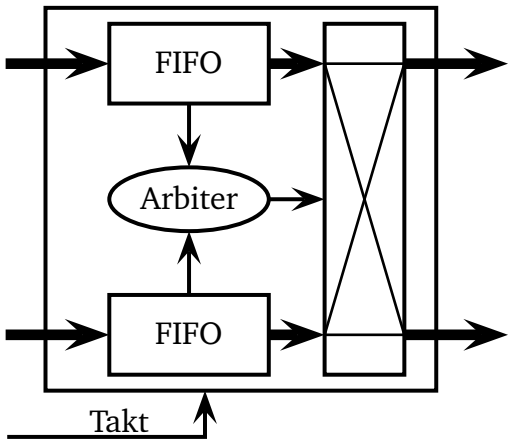


Abbildung 6.10: Aufbau des FIFO-Shuffleelements

Die Register in den Shuffleelementen sind als FIFO's realisiert (Abb. 6.10). Ein Shuffleelement kann in den FIFO's somit mehrere Anfragen (für die Auswertung bis zu vier Anfragen je FIFO) zwischenspeichern. Dies hat den Vorteil, dass die Anfragen aus den vorherigen Shuffleelementen schneller in die nächste Stufe weitergereicht werden. Anfragehäufungen an einem Shuffleelement können somit besser ausgeglichen werden und eine mögliche Stauung breitet sich nicht sofort rückwärts im Netzwerk aus. Damit keine Daten verloren gehen, müssen die Shuffleelemente über Steuersignale in entgegengesetzter Richtung verbunden werden. Die Steuersignale des Shuffleelementes der Stufe i zeigen dem Shuffleelement der Stufe $i-1$ an, ob dieses noch Daten annehmen kann. Durch diese zusätzlich notwendigen Steuerleitungen steigt auch der Verdrahtungsaufwand. Die verwendeten Steuersignale der Shuffleelemente sind in Abbildung 6.11 dargestellt.

FIFO 1	FIFO 2	Arbitrierung
leer	leer	undefiniert
leer	nicht leer	FIFO 2
nicht leer	leer	FIFO 1
nicht leer	nicht leer	toggle

Tabelle 6.1: Funktionsweise des Arbiters eines FIFO-Shuffleelements

Die Priorisierung der beiden Ein- und Ausgänge der Shuffleelemente kann nicht wie im standard Omeganetzwerk vorgenommen werden. Ein Eingang hätte dann eine höhere Priorität, was

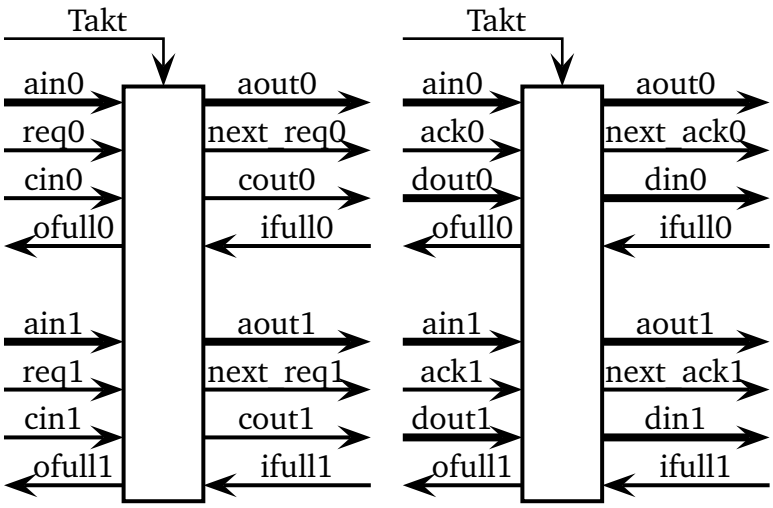


Abbildung 6.11: Datenleitungen der FIFO-Shuffleelemente, Datenanfrage (l.), Datenantwort (r.)

abhängig von den Datenanfragen zu einer Ungleichbehandlung der beiden Ports führt. Ein Arbiter sorgt deshalb für eine gerechte aber auch effiziente Weiterleitung der Anfragen. Ist einer der beiden FIFO's leer, kann der Arbiter alle Anfragen aus dem zweiten FIFO weiterleiten. Nur wenn beide FIFO's mit Anfragen gefüllt sind, wird abwechselnd aus einem der beiden FIFO's eine Anfrage weitergeleitet. Die Tabelle 6.1 zeigt das Verhalten des Arbiters für die vier möglichen Fälle. Im Gegensatz zu dem Arbiter aus [AMAD07] ist der hier verwendete Arbiter effizienter. In [AMAD07] wechselt der Arbiter mit jedem Takt zwischen den beiden Eingängen und kann somit nicht beschleunigt Daten aus einem Eingang weiterleiten.

6.4 Experimentelle Auswertung der Verbindungsnetzwerke

Um die verwendeten Netzwerke, unabhängig von der verwendeten Architektur, bewerten zu können, wurden diese ohne Verarbeitungseinheiten auf einem Cyclone II FPGA (Abschnitt 9.7) realisiert. Neben dem Realisierungsaufwand, also dem Platzbedarf des jeweiligen Netzwerks, ist auch der Einfluss des Netzwerks auf die maximal erreichbare Taktfrequenz interessant. Die Taktfrequenz für die Netzwerke ist in Abbildung 6.12 dargestellt. Das Busnetzwerk (BDPA) erreicht für vier Verarbeitungseinheiten die höchste Taktfrequenz. Diese sinkt aber schon bei 16 Verarbeitungseinheiten stark ab. Nur das Ringnetzwerk (RN) hält die Taktfrequenz relativ stabil und sinkt bis 128 Verarbeitungseinheiten nur leicht. Das Omeganetzwerk (ON) ist in Abbildung 6.12 nicht enthalten, da es sich hierbei um ein kombinatorisches Schaltnetz handelt. Als Vergleich kann das Omeganetzwerk mit Registerstufe (ONR) herangezogen werden. Da hier die Registerstufe die langen Pfade im Netzwerk auftrennt, kann davon ausgegangen werden, dass beim Omeganetzwerk die Taktfrequenz noch geringer ist. Das Omeganetzwerk mit FIFO-Shuffleelementen (ONP) konnte nur bis 16 Verarbeitungseinheiten ausgewertet werden, da die notwendigen Ressourcen auf dem FPGA für mehr Verarbeitungseinheiten nicht ausgereicht haben. Der notwendige Aufwand für die Realisierung der Netzwerke ist in Abbildung 6.13 dargestellt. Abgesehen von dem ONP Netzwerk benötigte das ON die meisten Ressourcen. Das Busnetzwerk mit den beiden Arbitrierungsmöglichkeiten sowie das Ringnetzwerk benötigen relativ wenig Ressourcen. Um den Zusammenhang zwischen den benötigten Ressourcen

und der maximal erreichbaren Taktfrequenz bestimmen zu können, wurde die Taktfrequenz in Abhängigkeit von den Ressourcen bestimmt $\left(\frac{Taktfrequenz}{Logikelemente}\right)$. Das Ergebnis ist in Abbildung 6.14 dargestellt. Es gibt Auskunft darüber, wie die Taktfrequenz von den Ressourcen (Logikelementen) abhängt bzw. wie effizient die Ressourcen für eine möglichst hohe Taktfrequenz verwendet werden. Am besten schneiden dabei das Ringnetzwerk (RN) und die Busnetzwerke ab. In der Gegenüberstellung fallen die Busnetzwerke und das Ringnetzwerk zusammen, obwohl die Ergebnisse aus Ressourcenverbrauch und maximaler Taktfrequenz unterschiedlich sind. Das Ringnetzwerk benötigt zwar mehr Ressourcen (für die Realisierung der Registerstufen und die Arbitrierung an jeder Registerstufe), erreicht aber auch eine entsprechend höhere Taktfrequenz.

Unberücksichtigt bleiben bei dieser Gegenüberstellung Eigenschaften, wie die Dauer, bis eine Anfrage durch das Netzwerk abgearbeitet wurde oder mögliche parallele Eigenschaften. Ein Netzwerk, das viele Anfragen parallel abarbeiten kann, kann dadurch eine geringere Taktfrequenz ausgleichen. Dies ist darüber hinaus zusätzlich von der Anwendung und der Anzahl der Netzwerkzugriffe abhängig. Die Anbindung des Netzwerks an das Gesamtsystem ist ein weiterer wichtiger Aspekt. Läuft das Gesamtsystem mit der gleichen Taktfrequenz wie das Netzwerk, kann das System u. U. ausgebremst werden. Dies ist aber auch wieder stark von der Anwendung abhängig. Allgemein empfiehlt es sich, bei einem großen Unterschied in der Taktfrequenz das Netzwerk vom Gesamtsystem zu trennen und beide Teilsysteme mit unterschiedlichen Taktfrequenzen zu betreiben.

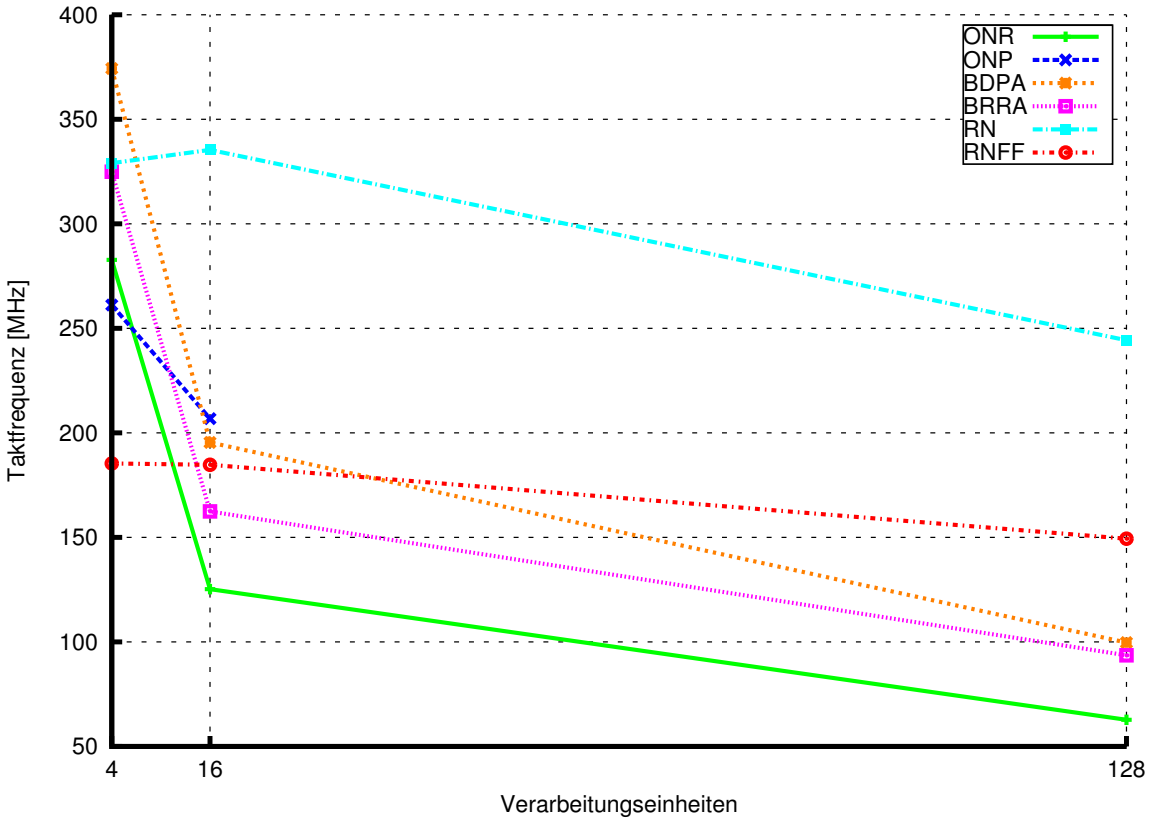


Abbildung 6.12: Auswertung der Verbindungsnetzwerke: Taktfrequenz der Netzwerke

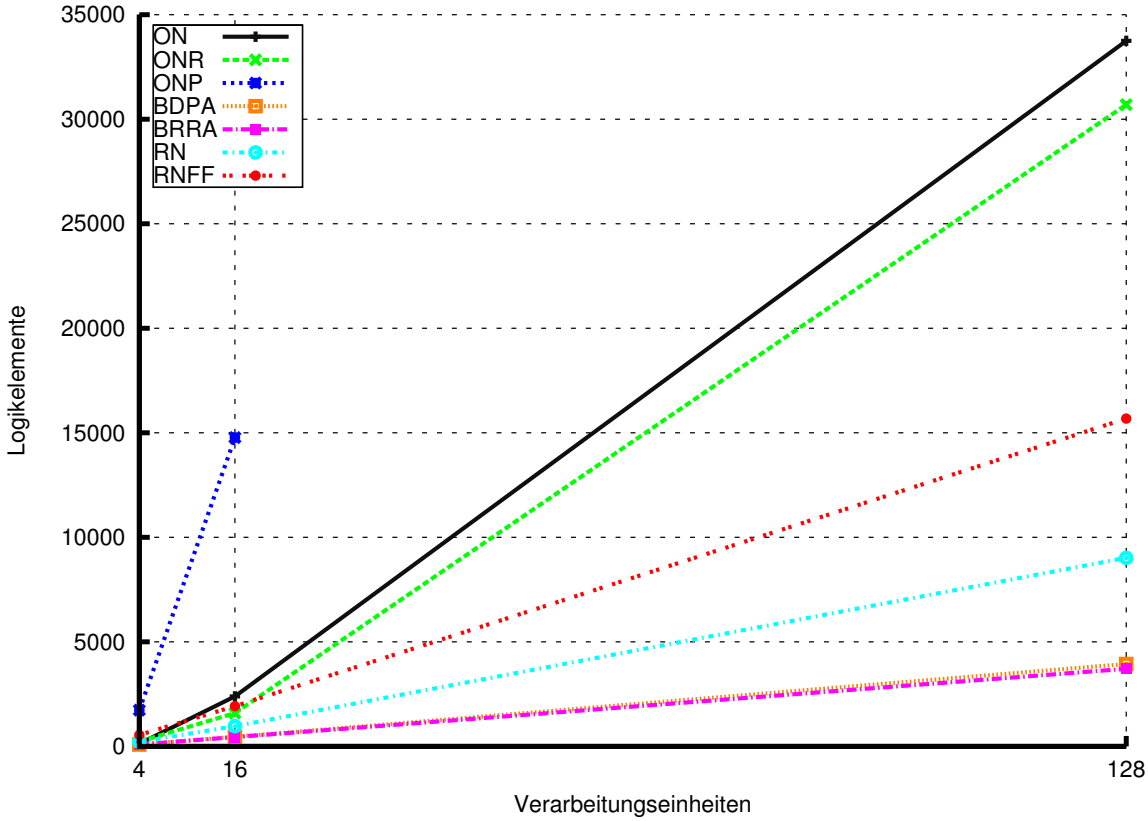


Abbildung 6.13: Auswertung der Verbindungsnetzwerke: Logikelemente der Netzwerke

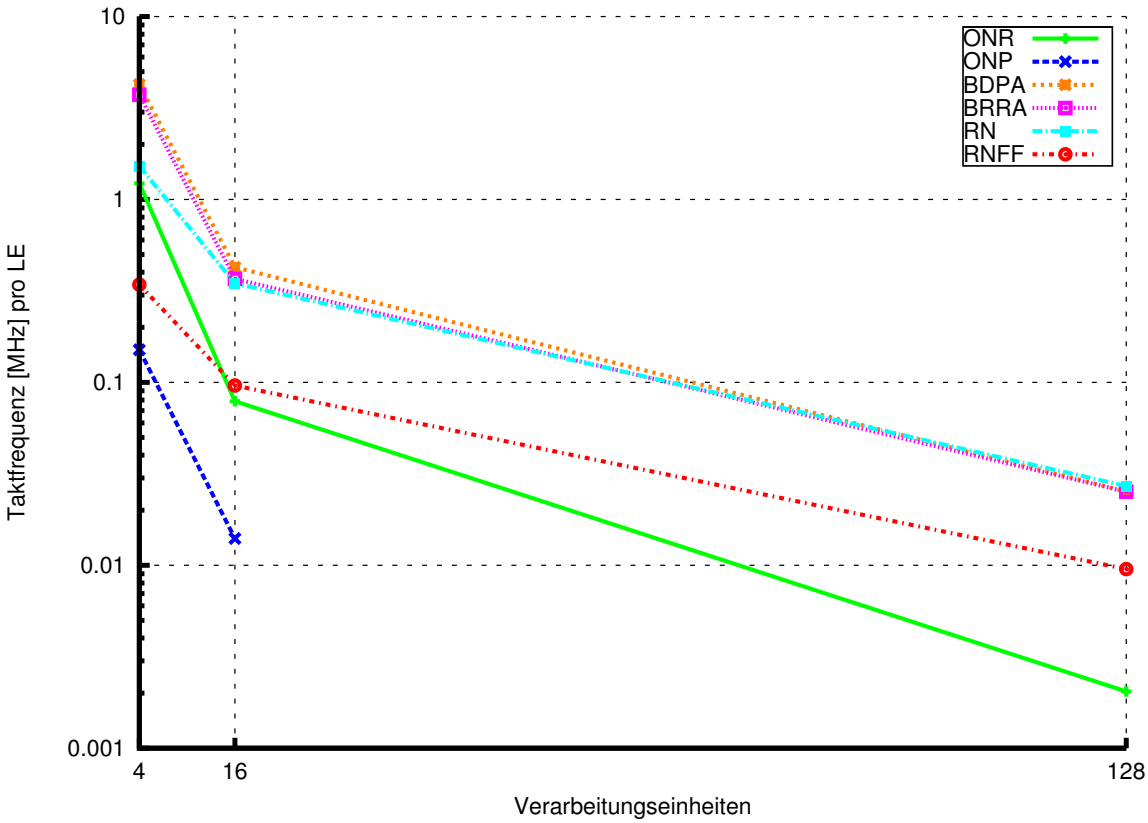


Abbildung 6.14: Auswertung der Verbindungsnetzwerke: Taktfrequenz pro Logikelement

In Tabelle 6.2 sind die Netzwerke bezüglich der Parallelität der Zugriffe und der erreichbaren maximalen Taktfrequenz auch bei einer großen Anzahl von Verarbeitungseinheiten bewertet. Das Bewertungsschema reicht von sehr gut $++$, gut $+$, durchschnittlich $\oslash$, gering $-$ bis sehr gering $--$. So weist z. B. das BDPA sehr gute Eigenschaften für einzelne Zugriffe auf. Parallele Zugiffe können hingegen nur sequenziell abgearbeitet werden. Die erreichbare Taktfrequenz ist hauptsächlich von der Buslänge und der Anordnung der Verarbeitungseinheiten abhängig und wurde daher mit $\oslash$ bewertet. In die Bewertung der Zugriffe fließt zum einen die Zeitdauer (Taktzyklen) ein bis ein Lesezugriff abgearbeitet ist, zum anderen auch die Möglichkeit, parallel Zugriffe durchzuführen. So können im Ringnetzwerk ebenso wie im Omeganetzwerk parallele Zugriffe durchgeführt werden. Allerdings ist auch der Zeitpunkt, zu dem die Leseanfragen gestellt werden, zu berücksichtigen.

Verbindungsnetzwerk	Zugriffe		Taktfrequenz
	einzeln	parallel	
Busnetzwerk (dynamisch)	$++$	$--$	$\oslash$
Busnetzwerk (Round-Robin)	$\oslash$	$--$	$\oslash$
Ringnetzwerk	$\oslash$	$+$	$++$
Ringnetzwerk (Forwarding)	$+$	$+$	$++$
Omeganetzwerk	$--$	$++$	$--$
Omeganetzwerk (Register)	$--$	$++$	$-$
Omeganetzwerk (FIFO-Shuffleelemente)	$\oslash$	$+$	$+$

Tabelle 6.2: Bewertung der untersuchten Verbindungsnetzwerke

6.5 Softcoreprozessoren

Unter Softcoreprozessoren versteht man Prozessoren, die in einer Hardwarebeschreibungssprache vorliegen. Diese werden üblicherweise auf FPGAs implementiert. Der Vorteil von Softcoreprozessoren besteht darin, dass diese für die verschiedensten Anwendungen individuell konfiguriert und erweitert werden können. So können z. B. besonders häufig ausgeführte Funktionen in Hardware implementiert werden. Softcoreprozessoren gibt es von vielen Anbietern. Zu den bekanntesten Softcoreprozessoren gehört der MicroBlaze [Xila] und der PicoBlaze [Xilb] beide von der Firma Xilinx [Xil10] sowie der NIOS [Altb, Altc] und NIOS II, die beide von der Firma Altera [Alt10b] stammen. Für die in dieser Arbeit verwendeten Hardwarearchitekturen wird der NIOS II/f Prozessor verwendet. Die Wahl fiel auf den NIOS II/f Prozessor, da dieser in der Altera Design Suite integriert ist und diese auf Grund der verwendeten FPGAs ohnehin Verwendung findet. Die gute Erweiterbarkeit, Anpassbarkeit sowie die vorhandene Toolunterstützung sprechen für den Einsatz dieses Softcoreprozessors.

6.5.1 NIOS II Prozessor

Der NIOS II/f Prozessor ist ein Sofcoreprozessor der Firma Altera [Alt10b] und wird in drei Geschwindigkeitsstufen angeboten. Der große Vorteil von Softcoreprozessoren besteht darin, einerseits einen fertigen, optimierten Prozessor zur Verfügung zu haben, andererseits diesen

aber an verschiedene Anwendungen anpassen und erweitern zu können. Der Softcoreprozessor kann somit exakt an die Anforderungen angepasst werden. Zusätzliche Module, z. B. für Fließkommaberechnungen, können einfach hinzugefügt werden. Gleichzeitig können Anwendungen für den Prozessor, unabhängig von der gewählten Konfiguration, in der Programmiersprache C entwickelt werden. Eine gute Übersicht über die NIOS II Prozessoren gibt Pat Mead in [Mea05]. Die wichtigsten Eigenschaften des NIOS II/f Prozessors im Überblick [Alte]:

- 32 Bit NIOS II/f Kern [Alte, Seite 5-4]
- 6-stufige Pipeline
- dynamische Sprungvorhersage
- unterstützt Custom Instructions [Altd], siehe Abschnitt 6.5.1.1
- 32 universale Register
- 218 DMIPS [Wei02] bei 185 MHz

Die Vorgehensweise zur Umsetzung des NIOS II Prozessors als Multiprozessorsystem wird in [Alt10a, Elk08] beschrieben[2]. Obwohl dies, wie beschrieben, grundsätzlich möglich ist, gestaltet sich die Umsetzung vieler NIOS II Prozessoren in einem Multiprozessorsystem als schwierig, da die dafür eingesetzten Programme (im weitesten Sinne: Synthesesoftware) für derartige Systeme nicht ausgelegt zu sein scheinen. So ist die Erstellung von Systemen mit vielen Prozessoren nach dieser Vorgehensweise eingeschränkt. Auf eine detaillierte Beschreibung der Probleme sowie diese gelöst werden können wird in Abschnitt 6.9.6 eingegangen. Anhand der dort beschriebenen Hardwarearchitektur wird gezeigt, wie die Taktfrequenz des Gesamtsystems deutlich verbessert werden kann. Diese neue Vorgehensweise war zu Beginn der Arbeit unbekannt und wurde parallel zu den Hardwarearchitekturen erarbeitet.

Ebenfalls problematisch ist die Simulation dieser Systeme [Dur07, Alt08].

6.5.1.1 Custom Instruction

Die verwendeten NIOS II Prozessoren können über Custom Instructions (CI) [Altd] erweitert werden. Bei den Custom Instructions handelt es sich um benutzerspezifische oder applikationsspezifische Hardwarefunktionen, die zu den vorhandenen Funktionen des NIOS II Prozessors hinzugefügt werden können. Dazu wird der Befehlssatz des NIOS II Prozessors um zusätzliche Befehle erweitert. Wie die Custom Instructions verwendet werden können, um Anwendungen zu beschleunigen, und welche Probleme dabei entstehen, wurde in [CFHZ04] behandelt. Die Custom Instructions eignen sich besonders gut, um die im GCA-Modell zusätzlich benötigten Funktionen zu realisieren. Dazu wird für jede zusätzlich benötigte Funktion eine Custom Instruction hinzugefügt, so etwa eine, um alle NIOS II Prozessoren zu synchronisieren.

[2] Die Verwendung von Mutexen kommen bei der Umsetzung des GCA-Modells nicht zum Einsatz.

Die Custom Instructions stehen in verschiedenen Arten zur Verfügung [Altd, Tabelle 1-1, Seite 1-4 - 1-5]:

- **kombinatorisch:** Erweiterung für *eine* Funktion, deren Berechnung genau *einen* Taktzyklus dauert.
- **mehrzyklisch:** Erweiterung für *eine* Funktion, deren Berechnung *mehrere* Taktzyklen dauert.
- **erweitert:** Erweiterung für *mehrere* Funktionen, deren Berechnung *mehrere* Taktzyklen dauern.
- **internes Register:** Erweiterung mit weiteren Steuerleitungen zur Ansteuerung von Speichern.
- **externes Interface:** Erweiterung mit benutzerspezifischen Steuerleitungen.

Um den NIOS II Prozessor mit den nach dem GCA-Modell zusätzlich benötigten Funktionen zu erweitern, wurde die erweiterte Custom Instruction gewählt. Mit dieser Custom Instruction können darüber hinaus auch noch zusätzliche architekturspezifische Custom Instructions hinzugefügt werden. Die erweiterte Custom Instruction gewährleistet, dass die benötigte Flexibilität für unterschiedliche Architekturen erhalten bleibt. Eine Custom Instruction, deren Berechnungsdauer sich über mehrere Taktzyklen erstrecken kann, ist notwendig, da die Dauer der Berechnung vorab unbekannt ist (z. B. Synchronisation aller Prozessoren).
Die Custom Instruction wird im C-Code über drei Parameter aufgerufen (ein 8 Bit und zwei 32 Bit Parameter) und enthält einen Rückgabewert (32 Bit). Der erste Parameter definiert dabei die auszuführende Funktion, die beiden 32 Bit Parameter werden für die Übergabe von Adressen und Daten verwendet. Der Rückgabewert enthält die gelesenen Daten oder Kontrollinformationen. Je nach verwendeter Architektur unterscheiden sich die Funktionen in ihrer Komplexität, Ausführungszeit, Menge und Verwendbarkeit.

6.5.2 Generationssynchronisation

Für die Umsetzung der Multiprozessorhardwarearchitekturen wird der NIOS II Softcoreprozessor verwendet. Der Prozessor wendet die Zellregel auf die einzelnen Zellen an und berechnet deren Folgezustand. Üblicherweise besteht das zu berechnende Zellfeld aus mehr Zellen als Prozessoren zur Verfügung stehen. Dies hat zur Folge, dass jeder Prozessor mehrere Zellen nacheinander berechnen muss. Die Berechnung des Nachfolgezustands jeder Zelle ist abhängig von ihrem aktuellen Zustand. Daraus folgt, dass die Berechnungsdauer jeder Zelle unterschiedlich lange ist. Dieser Umstand wird durch den Aufbau des Prozessors verstärkt. Eigenschaften, wie z. B. Sprungvorhersage, führen dazu, dass die Prozessoren selbst bei gleichen Daten unterschiedlich lange für die Berechnung benötigen. Es ist also notwendig, die Prozessoren nach der Berechnung einer Generation zu synchronisieren. Der Begriff der Synchronisation wird in der Literatur oft unterschiedlich verwendet. Eine boolesche Algebra zur Einordnung von unterschiedlichen Synchronisationsmechanismen wird in [GS10] vorgeschlagen. Durch die Synchronisation ist gewährleistet, dass alle Prozessoren in jeder Generation auf gültigen Daten arbeiten. Bei der Generationssynchronisation kommt es daher zu Wartezyklen. Prozessoren, die

ihre Berechnung früher als andere beendet haben, müssen warten, bis diese ihre Berechnung ebenfalls abgeschlossen haben. Insgesamt führt die Generationssynchronisation zu einer Reduzierung der Leistungsfähigkeit der Architektur. Die Generationssynchronisation wird über eine Custom Instruction umgesetzt, die den Prozessor so lange anhält, bis sich alle Prozessoren in der Synchronisationsphase befinden. Hierzu kommt eine Barrieren-Synchronisation zum Einsatz [SK10, And95, KCS+09].

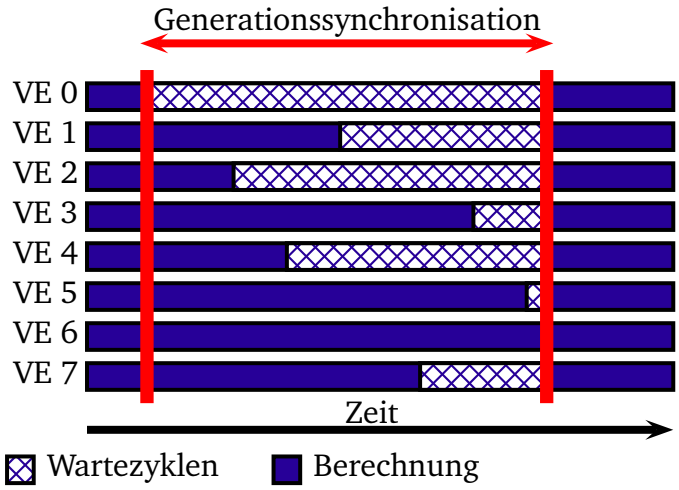


Abbildung 6.15: Generationssynchronisation für acht Verarbeitungseinheiten (VE)

In [SK10, Seite 28-29] werden drei grundlegende Synchronisierungsmechanismen (Barrieren) unterschieden:

- **Zentrale Barrieren:** Alle Prozessoren senden bei Erreichen der Barriere eine Nachricht an eine zentrale Einheit. Die zentrale Einheit beantwortet diese Nachrichten, sobald alle Prozessoren die Barriere erreicht haben. In einer Hardwareimplementierung kann dies durch ein UND-Gatter realisiert werden.
- **Dezentrale Barrieren:** Jeder Prozessor, der eine Barriere erreicht, sendet eine Nachricht an alle anderen Prozessoren. Jeder Prozessor wertet dann, bei Erreichen der Barriere, diese Nachrichten aus. Sobald alle Prozessoren eine Nachricht von allen anderen Prozessoren erhalten haben, kann die Barriere verlassen werden.
- **Hierarchische Barrieren:** In hierarchischen Barrieren synchronisiert sich ein Prozessor mit einem Teil der restlichen Prozessoren. Die Synchronisation wird dann an die nächst höhere Ebene weitergereicht bis die globale Synchronisation sichergestellt ist.

Für die Synchronisation der Prozessoren wird die zentrale Barriere verwendet. Die Umsetzung erfolgt über ein UND-Gatter als zentrale Einheit. Die Wartezyklen der Generationssynchronisation werden durch die Zellregel, die Anzahl an Zellen, die Zellzustände und die Anzahl der Prozessoren beeinflusst. Die Dauer der Generationssynchronisation ist u. a. datenabhängig. Somit ist nicht generell davon auszugehen, dass die Anzahl der Wartezyklen mit zunehmender Prozessoranzahl steigt. Auf Grund der Zuteilung der Zellen auf die Prozessoren ergibt sich für die Laufzeit innerhalb einer Generation ein konstanter, ein datenabhängiger und ein netzwerkabhängiger Anteil. Abbildung 6.15 zeigt den zeitlichen Synchronisationsprozess für acht Verarbeitungseinheiten.

6.6 Einarmige universelle GCA-Architektur (MPA)

Als Erstes wurde eine einarmige universelle GCA-Architektur (Abb. 6.16), abgekürzt MPA (engl.: Multiprocessor Architecture), entworfen [SHH09a, SHH09b, SHH09c]. Diese Architektur wird, unabhängig vom verwendeten Netzwerk, als MPA (Multiprozessorarchitektur) bezeichnet. Einarmig bedeutet, dass eine Zelle im GCA-Modell auf maximal eine Nachbarzelle zugreifen kann. Jede Zelle besitzt also einen Speicherplatz, in dem ein Zeiger auf eine beliebige Nachbarzelle abgespeichert werden kann[3]. Die GCA-Architektur ist universell, da sie nicht nur für die Simulation von Multi-Agenten-Systemen verwendet werden kann. In dieser Architektur besteht jede Zelle aus zwei 16 Bit großen Speicherplätzen (L und D). In D wird der aktuelle Zustand der Zelle gespeichert, in L ein Zeiger auf den aktuellen Nachbarn.

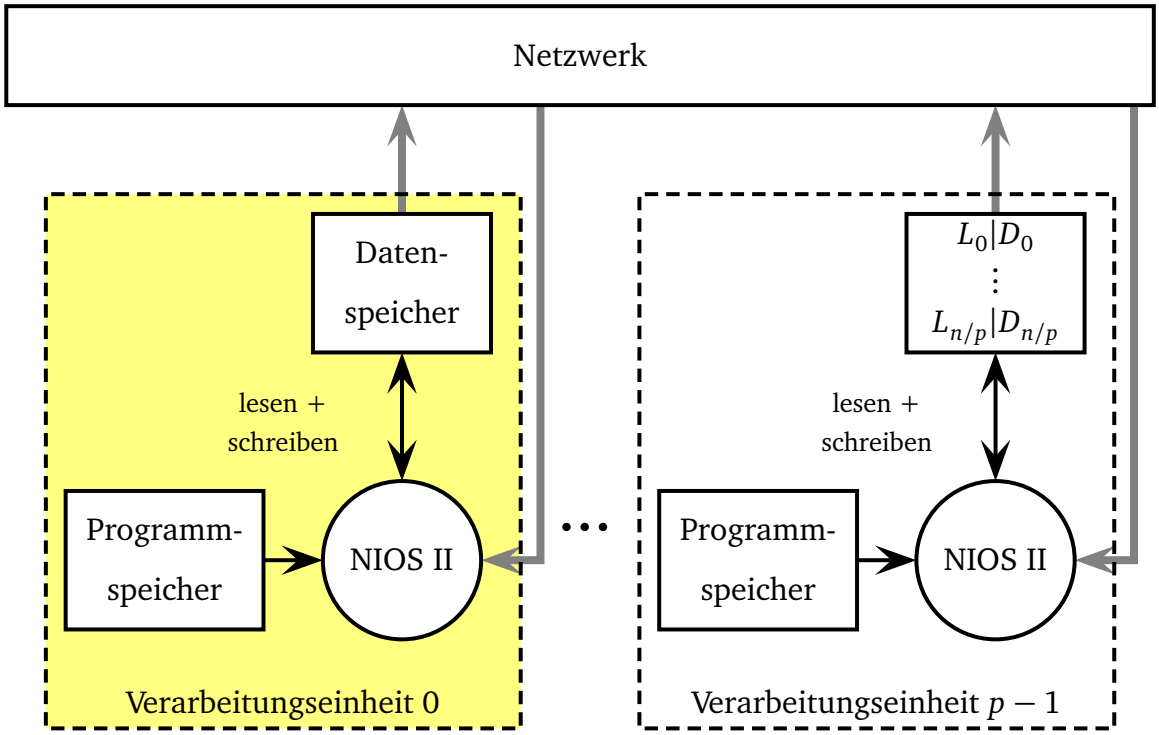


Abbildung 6.16: Einarmige universelle GCA-Architektur (MPA)

Die Architektur besteht aus p NIOS II Prozessoren. Jeder der Prozessoren besitzt einen eigenen Programmspeicher, in dem die Zellregel enthalten ist. Des weiteren besitzt jeder Prozessor einen Datenspeicher, in dem ein Teil des gesamten Zellfeldes gespeichert ist. Diese drei Komponenten (Programmspeicher, Datenspeicher und Prozessor) bilden eine Verarbeitungseinheit (VE). Mehrere dieser Verarbeitungseinheiten bilden zusammen die GCA-Architektur. Die Verarbeitungseinheiten sind über eines der in Abschnitt 6.3 beschriebenen Netzwerke miteinander verbunden und erlauben somit jeder Verarbeitungseinheit lesenden Zugriff auf die Datenspeicher der anderen Verarbeitungseinheiten. Der Datenspeicher besitzt dafür zwei Ports. Ein Port für die Lese- und Schreibzugriffe des ihm zugeordneten NIOS II Prozessors und einen für die externen Lesezugriffe anderer Verarbeitungseinheiten.

[3] Obwohl nur ein Speicherplatz für einen Zeiger vorhanden ist, können während der Abarbeitung der Zellregel weitere Zeiger berechnet werden.

6.6.1 Definierte Custom Instructions

Um die Zugriffe entsprechend dem GCA-Modell in der Architektur zu realisieren, wurden zehn Custom Instructions (CI), auch Funktionen genannt, definiert. Die Custom Instructions erlauben die Durchführung von Lesezugriffen (intern und extern), internen Schreibzugriffen und der Generationssynchronisation. Da in dieser Architektur jede Zelle aus einem L und einem D Feld besteht, gibt es für jedes der beiden Felder einen internen und einen externen Lesezugriff. Der Schreibzugriff besteht nur für die internen Zellen. Für das L Feld gibt es einen weiteren Schreibzugriff, der das L Feld vor dem Schreiben um ein Bit nach rechts verschiebt. Diese kombinierte Operation ist für eine Teilmenge von GCA-Anwendungen (Bitonisches Mischen und Sortieren in Abschnitt 2.3.1, Umsetzung des Bitonischen Mischens in Abschnitt 6.6.2.2, Umsetzung des Bitonisches Sortierens in Abschnitt 6.6.3.3) nützlich. Für die effiziente Multi-Agenten-Simulation wurden nachträglich noch zwei Funktionen hinzugefügt (AUTO_GET_P und AUTO_GET_L). Diese zwei Befehle entscheiden anhand eines Adressvergleichs in Hardware, ob der Lesezugriff intern oder extern durchzuführen ist. Dies beschleunigt die Zellregel, wenn für einen Lesezugriff nicht feststeht oder nicht bestimmt werden kann, ob dieser immer intern oder immer extern durchzuführen ist. Somit wird auch die Programmierung der Zellregel erleichtert und von internen bzw. externen Lesezugriffen abstrahiert.

Die definierten Custom Instructions dieser Architektur:

- INTERN_GET_P: Interner Lesezugriff auf den Zeiger L
- INTERN_GET_D: Interner Lesezugriff auf das Datum D
- EXTERN_GET_P: Externer Lesezugriff auf den Zeiger L unter Verwendung des Netzwerks
- EXTERN_GET_D: Externer Lesezugriff auf das Datum D unter Verwendung des Netzwerks
- SET_PS: Schreibt den Zeiger L um ein Bit nach rechts verschoben ($L >> 1$)
- SET_P: Schreibt den Zeiger L
- SET_D: Schreibt das Datum D
- NEXTGEN: Generationssynchronisation
- AUTO_GET_P[4] : Lesezugriff auf den Zeiger L. Über einen kombinatorischen Adressvergleich wird der Lesezugriff, wenn möglich intern, ansonsten extern, ausgeführt.
- AUTO_GET_L[4] : Lesezugriff auf das Datum D. Über einen kombinatorischen Adressvergleich wird der Lesezugriff, wenn möglich intern, ansonsten extern, ausgeführt.

Für Lesezugriffe wird als Parameter die Adresse der zu lesenden Zelle übergeben. Wird eine (interne) Zelle beschrieben, wird zusätzlich zur Adresse das Datum übergeben. Die Details der Custom Instructions sind in Abschnitt 6.5.1.1 beschrieben. Ist für eine Funktion unerheblich, ob ein Lesezugriff auf einen Zeiger L oder auf ein Datum D stattfindet, so wird nur der Funktionsname verwendet (z. B. AUTO_READ).

[4] Diese Funktion wurde nachträglich für die Multi-Agenten-Simulation hinzugefügt und kann nicht ohne Vorkehrungen auf das Omeganetzwerk und das Omeganetzwerk mit Registerstufe angewendet werden.

6.6.2 Architektur mit kombinatorischem Omeganetzwerk

6.6.2.1 Realisierungsaufwand auf einem FPGA

Die Realisierung der Architektur mit einem kombinatorischen Omeganetzwerk (Abschnitt 6.3.4.1) erfolgte auf einem Cyclone II FPGA und auf einem Stratix II FPGA (die genauen Eigenschaften der FPGAs sind in Abschnitt 9.7 aufgeführt) [SHH09b]. Die maximale Anzahl an Prozessoren p beträgt auf dem Cyclone II 16. Auf dem Stratix II können 32 Prozessoren realisiert werden. Mehr Prozessoren sind auf Grund von Einschränkungen der Synthesesoftware derzeit nicht möglich. Die Tabelle 6.3 zeigt u. a. die Anzahl der Logikelemente (LE), Register und maximale Taktfrequenz für die Realisierung der Architektur auf dem Cyclone II. Die Tabelle 6.4 zeigt die Werte für die Realisierung auf dem Stratix II.

Die abnehmende Taktfrequenz ist durch mehrere Faktoren bedingt. Zum einen durch das Verbindungsnetzwerk, zum anderen durch die Realisierung auf einem FPGA. Jede Verarbeitungseinheit bildet eine abgeschlossene Einheit. Die einzige Verbindung unter den Verarbeitungseinheiten stellt das Verbindungsnetzwerk dar. Die maximale Taktfrequenz wird also durch die Wahl des Verbindungsnetzwerkes beschränkt. Doch auch ohne Verbindungsnetzwerk sinkt die maximale Taktfrequenz ab. Dies ist begründet durch die Struktur des FPGAs (Abschnitt 9.7.1). Die Speicher im FPGA sind spaltenartig zwischen den Logikelementen angeordnet. Bei einer größeren Anzahl an Verarbeitungseinheiten steigt auch der Bedarf an Speichern für die Programm- und Datenspeicher. Es muss also eine geeignete Zuordnung von Speichern, Prozessoren und Netzwerk gefunden werden. Für eine große Anzahl an Verarbeitungseinheiten ist es unter Umständen nicht mehr möglich, den Prozessor jeder Verarbeitungseinheit direkt an einen Speicher zu platzieren. Die Platzierung der Verarbeitungseinheiten hat dann wiederum Einfluss auf das verwendete Verbindungsnetzwerk.

Bei der hier dargestellten GCA-Architektur ist die maximale Taktfrequenz durch das verwendete Omeganetzwerk limitiert. Durch jede Verdopplung der Verarbeitungseinheiten wird eine weitere Verteilerstufe im Omeganetzwerk notwendig. Dies begrenzt die maximale Taktfrequenz mit zunehmender Anzahl an Verarbeitungseinheiten, da hierdurch die Pfadlänge im Omeganetzwerk immer größer wird. Diese Einschränkung kann durch die Verwendung eines alternativen Netzwerks aufgehoben werden. Die Betrachtung anderer Netzwerke findet in Abschnitt 9.4 statt. Der Realisierungsaufwand für das Omeganetzwerk ist gering (Tabelle 6.3 und Tabelle 6.4). Es werden nur wenig Logikelemente für die Shuffleelemente benötigt.

6.6.2.2 Bitonisches Mischen

Als klassische Anwendung für das GCA-Modell und zur Bewertung der Leistungsfähigkeit der Architektur ist zunächst das Bitonische Mischen realisiert worden. Später wurde dieses erweitert um so das Bitonische Sortieren umzusetzen. Diese Sortiernetzwerke werden in [Bat68] beschrieben. Die beiden Algorithmen lassen sich effizient im GCA-Modell umsetzen (Abschnitt 2.3.1).

p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
		LEs	Register			
1	2.824	-	-	195.296	1.585	145
4	10.165	154	0	289.568	4.910	105
8	19.985	866	0	415.264	9.351	75
16	40.878	2.145	0	666.656	18.244	55

Tabelle 6.3: Realisierungsaufwand der einarmigen universellen GCA-Architektur mit Omeganetzwerk (ON) auf einem Cyclone II FPGA

p	ALMs	ALUTs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			ALUTs	Register			
1	1.203	1.592	-	-	195.296	1.520	200
4	4.147	5.354	279	0	289.568	4.813	133
8	8.397	10.634	818	0	415.264	9.200	100
16	17.448	21.439	2.344	0	666.656	17.994	75
32	36.131	43.815	6.092	0	1.169.440	35.591	55

Tabelle 6.4: Realisierungsaufwand der einarmigen universellen GCA-Architektur mit Omeganetzwerk (ON) auf einem Stratix II FPGA

Auf der Architektur wurde das Bitonische Mischen aus Abschnitt 2.3.1 als klassische Anwendung für das GCA-Modell umgesetzt [SHH09b]. Hierfür wird die Zellregel als C-Programm implementiert (Quellcode 6.1) und durch die NIOS II/f Prozessoren ausgeführt. Die im GCA-Modell notwendigen Nachbarzugriffe werden durch externe Lesezugriffe über das Netzwerk realisiert.

Die Zellregel für das Bitonische Mischen ist durch die folgenden Formeln gegeben. Jede Zelle C_i besteht aus einem Datenfeld D und einem Linkfeld L also: $C_i = (D, L)$. In den Datenfeldern sind dabei die zu mischenden Zahlen abgespeichert. Die Linkfelder speichern die auszuwählenden Nachbarzellen ab. Dabei ist n die Anzahl der Zellen, g die Anzahl der auszuführenden Generationen und u die aktuelle Generation. Für das Linkfeld der Zelle mit dem Zellindex i wird $L(i)$ und für das Datenfeld $D(i)$ geschrieben. Mit $\div$ ist die Ganzzahldivision gemeint.

$$L^+(i) := \begin{cases} 2^{(u-g)}, & u \geq g \\ 0, sonst \end{cases}$$

$$D^+(i) := \begin{cases} D(L(i) \mod n), & (D(L(i) \mod n) < D(i)) \wedge \left(i \div \frac{n}{2^{(g+1)}} = 0\right) \\ D(i), & (D(L(i) \mod n) \geq D(i)) \wedge \left(i \div \frac{n}{2^{(g+1)}} = 0\right) \\ D(L(i) \mod n), & (D(L(i) \mod n) > D(i)) \wedge \left(i \div \frac{n}{2^{(g+1)}} = 1\right) \\ D(i), & (D(L(i) \mod n) \leq D(i)) \wedge \left(i \div \frac{n}{2^{(g+1)}} = 1\right) \end{cases}$$

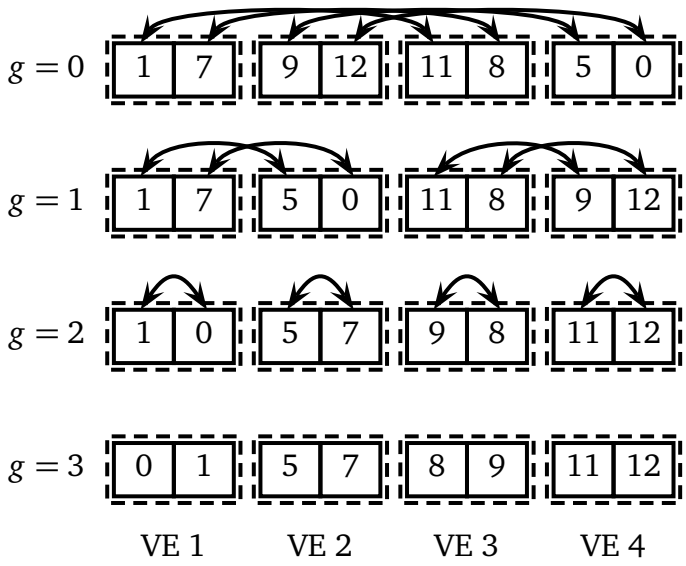


Abbildung 6.17: Externe und interne Lesezugriffe beim Bitonischen Mischen [HVWH01, Seite 7]. Darstellung analog [Hee07, Seite 182]

Abbildung 6.17 zeigt die Veränderung der Zugriffsmuster jeder Generation für vier Verarbeitungseinheiten und acht Zellen. Jeder Verarbeitungseinheit sind damit zwei Zellen zugeordnet. Für die ersten beiden Generationen ($g = 0$, $g = 1$) werden nur externe Lesezugriffe verwendet (pro Generation acht externe Zugriffe). In der letzten Generation werden nur interne Lesezugriffe verwendet. Abhängig von der Anzahl der Zellen und Verarbeitungseinheiten liegt die Anzahl der internen Lesezugriffe so hoch, dass es sich aus Effizienzgründen lohnt, diese von den externen Lesezugriffen zu unterscheiden. Im Quellcode 6.1 wird diese Unterscheidung in den Zeilen 18-21 in Abhängigkeit vom Generationszähler vorgenommen. Obwohl es sich hierbei um eine Softwareüberprüfung handelt, die zusätzliche Takte benötigt, lohnt sich die Überprüfung, da die externen Lesezugriffe noch mehr Takte benötigen. Ab welcher Generation von externen auf interne Lesezugriffe gewechselt werden kann, ist abhängig von der Anzahl der Verarbeitungseinheiten und der Anzahl der Zellen. Der Wechsel muss vorab bestimmt werden. Der Quellcode 6.1 zeigt die Zellregel für das Bitonische Mischen. Hervorgehoben sind die zusätzlichen Befehle und Custom Instructions (CI).
Vor der eigentlichen Ausführung der Zellregel werden in Zeile 6 und 7 alle Verarbeitungseinheiten synchronisiert. Die doppelte Generationssynchronisation ist notwendig, da mit der Synchronisation auch zwischen den beiden Generationen, die in den Datenspeichern abgespeichert sind, umgeschaltet wird. Während der Abarbeitung der Zellregel wird aus einem Teil des Speichers nur gelesen, während in den anderen Teil des Speichers nur geschrieben wird. Zeitgleich mit der Generationssynchronisation wird die Lese- und Schreibrichtung umgedreht. So kann nun aus dem Teil des Speichers gelesen werden, der die soeben berechnete Generation beinhaltet. Zu Beginn der Zellregel ist nur ein Teil des Speichers initialisiert. Dies erfordert eine doppelte Generationssynchronisation. Mit der ersten Generationssynchronisation werden alle Verarbeitungseinheiten zu Beginn der Berechnung synchronisiert. Da hierbei auch die Schreib- und Leserichtung für die beiden Teile des Speichers, die die beiden Generationen abspeichern, vertauscht werden, ist eine zweite Synchronisation notwendig. Durch die zweite Synchronisation wird für den initialisierten Teil des Speichers die Leserichtung eingestellt.

```
1  int main(){
2   int i,j,ii,k;
3   int neighbour;
4   int N,I;
5
6   CI(NEXTGEN,0,0);                                    //Generationssynchronisation
7   CI(NEXTGEN,0,0);
8
9   for(j=0;j<GENS;j++)
10  {
11   for(i=0;i<LOCL_CELLS;i++)
12   {
13    ii=i+SC;
14    k=CI(INTERN_GET_P,ii,0);
15    if((ii & k)==0) neighbour=ii+k;          //Adressberechnung der Nachbarzelle
16    else neighbour=ii-k;
17
18    if(j>1)                                                          //N = D(L(i))
19     N = CI(INTERN_GET_D,neighbour,0);         //Laden (extern) der Nachbarzelle
20    else
21     N = CI(EXTERN_GET_D,neighbour,0);         //Laden (intern) der Nachbarzelle
22
23    I = CI(INTERN_GET_D,ii,0);                  //Laden des Zellwertes I = D(i)
24
25    if((ii & k)==0)                             //Bestimme Vertauschungsrichtung
26     if(N<I) CI(SET_D,ii,N);
27     else CI(SET_D,ii,I);
28    else
29     if(I<N) CI(SET_D,ii,N);
30     else CI(SET_D,ii,I);
31
32    CI(SET_PS,ii,k);
33   }
34
35   CI(NEXTGEN,0,0);                                    //Generationssynchronisation
36  }
37  return 0;}
```

Quellcode 6.1: Zellregel des Bitonischen Mischens in C

Die Auswertung der Zellregel für das Bitonische Mischen auf dem Cyclone II ist in Tabelle 6.5, die Auswertung für das Stratix II FPGA ist in Tabelle 6.6 dargestellt. Für die Auswertung wurden 2048 Zahlen bitonisch gemischt. Dafür werden insgesamt 11 Generationen benötigt. Die absoluten Taktzyklen für die Ausführung der Zellregel sind bei gleicher Anzahl an Verarbeitungseinheiten für beide FPGAs identisch. Die Ausführungszeit ist auf dem Stratix II FPGA aufgrund der höheren Taktfrequenz geringer. Der Speedup beträgt für eine Architektur mit 32 Verarbeitungseinheiten auf dem Stratix II 8,2.

Die Taktzyklen, die für die Generationssynchronisation sowie für die externen Lesezugriffe benötigt werden, wurden direkt auf der Hardware gemessen. Die Wartezyklen für die Generationssynchronisation und die Dauer der externen Lesezugriffe fallen mit bis zu 3,5% auf dem Stratix II gering aus. Mit steigender Anzahl an Verarbeitungseinheiten steigen die Wartezyklen, da immer mehr externe Lesezugriffe durchgeführt werden müssen. Des weiteren müssen auch

mehr Verarbeitungseinheiten synchronisiert werden. Der prozentuale Anteil aller Wartezyklen (Generationssynchronisation und externe Lesezugriffe) ist in der Spalte $\sum\%$ angegeben. Mit steigender Anzahl an Verarbeitungseinheiten steigt auch die Anzahl an Wartezyklen an. Für einen Vergleich und die Bewertung der Architekturen wurde die CUR berechnet. Auf dem Stratix II mit 32 Verarbeitungseinheiten können somit 47.269 Zellen pro Millisekunde berechnet werden. Auf Grund der abnehmenden Taktfrequenz und der erhöhten Anzahl an externen Lesezugriffen steigt die CUR unterlinear an.

p	Taktzyklen	Ausführungszeit [ms]	Speedup	Wartezyklen			CUR $\left[\frac{1}{ms}\right]$
				Synchronisation	externes Lesen	$\sum\%$	
1	781.567	5,39	-	-	-	-	4.179
4	199.878	1,90	2,83	3.620	1.187	2,4	11.834
8	101.279	1,35	3,99	1.626	966	2,6	16.682
16	51.776	0,94	5,73	763	740	2,9	23.930

Tabelle 6.5: Auswertung des Bitonischen Mischens auf der MPA mit einem Omeganetzwerk (ON) auf einem Cyclone II FPGA

p	Taktzyklen	Ausführungszeit [ms]	Speedup	Wartezyklen			CUR $\left[\frac{1}{ms}\right]$
				Synchronisation	externes Lesen	$\sum\%$	
1	781.567	3,91	-	-	-	-	5.764
4	199.878	1,50	2,60	3.620	1.187	2,4	14.990
8	101.279	1,01	3,86	1.626	966	2,6	22.243
16	51.776	0,69	5,66	763	740	2,9	32.632
32	26.212	0,48	8,20	333	578	3,5	47.269

Tabelle 6.6: Auswertung des Bitonischen Mischens auf der MPA mit einem Omeganetzwerk (ON) auf einem Stratix II FPGA

Zum Vergleich der Leistungsfähigkeit dieser Architektur wurden einige Vergleichswerte ermittelt. Diese können den Ergebnissen der Multiprozessorarchitektur aus der Arbeit [HHJ06] gegenübergestellt werden. In dieser Arbeit wurden 128 Zahlen bitonisch gemischt und es liegt der selbe Algorithmus zugrunde. Die Anzahl der Zellen wurde von 2048 auf 128 reduziert, um möglichst aussagekräftige Vergleichswerte zu erhalten. Jedoch ist auf Grund der Architektur des NIOS II Prozessors davon auszugehen, dass die Leistungsfähigkeit bei einer zu geringen Zellanzahl pro Prozessor leicht sinkt. Ebenso wurde ein anderes Verbindungsnetzwerk verwendet, was die Vergleichbarkeit geringfügig einschränkt. Die NIOS II Architektur verwendet ein RNFF, während in [HHJ06] ein Multiplexernetzwerk verwendet wird. Vergleichswerte wurden für eine Architektur mit einem Prozessor und für eine Architektur mit 16 Prozessoren ermittelt. Die Architektur aus [HHJ06] benötigt bei einem Prozessor 15.288 und bei 16 Prozessoren 772 Instruktionen. Jede Instruktion benötigt sechs Taktzyklen für die Ausführung. Damit ergeben sich 91.728 bzw. 4.632 Taktzyklen. Mit der entsprechenden Taktfrequenz umgerechnet, ergibt sich eine Laufzeit von 0,951 ms (1 Prozessor) bzw. 0,0516 ms (16 Prozessoren). Die vorgestellte NIOS II Architektur benötigt bei einem Prozessor 31.489 und bei 16 Prozessoren 2.721 Takte. Mit der Taktfrequenz umgerechnet ergibt sich eine Laufzeit von 0,217 ms (1 Prozessor) und 0,035 ms (16 Prozessoren). Die Anzahl der benötigten Takte ist bei der NIOS II Architektur in beiden Fällen geringer. Die Taktfrequenz liegt bei der Architektur mit einem NIOS II Prozessor bei

145,00 MHz, bei 16 NIOS II Prozessoren bei 76,92 MHz. Im Vergleich der beiden Architekturen für einen Prozessor und 16 Prozessoren ergibt sich für einen Prozessor eine Beschleunigung von 4,38 und für 16 Prozessoren eine Beschleunigung von 1,46. Der Vergleich mit einem Prozessor zeigt, dass die Wahl für den NIOS II Prozessor bestätigt wurde. Dem Vergleich für 16 Prozessoren kann entnommen werden, dass die Leistungsfähigkeit mit zunehmender Prozessoranzahl absinkt. Dies ist hauptsächlich auf die stark sinkende Taktfrequenz zurückzuführen.

6.6.3 Architektur mit Omeganetzwerk mit Registerstufe

6.6.3.1 Realisierungsaufwand auf einem FPGA

Die Resultate der ersten Architektur mit einem Omeganetzwerk sind durch die Struktur des Netzwerkes begrenzt. Das Omeganetzwerk ist rein kombinatorisch aufgebaut. Dieser Umstand führt bei einer hohen Anzahl an Verarbeitungseinheiten zu einem komplexeren Netzwerk mit einer hohen Leitungslänge und vielen Shuffleelementen. Dabei ist der Platzbedarf für die Realisierung weniger relevant. Die insgesamt ansteigende Leitungslänge mindert die maximal mögliche Taktfrequenz des Gesamtsystems. Damit werden alle Berechnungen auf den Prozessoren der Verarbeitungseinheiten langsamer ausgeführt und die Leistungsfähigkeit des Systems gemindert. Um diese Problematik zu umgehen, wurde eine Registerstufe in das Omeganetzwerk integriert (Abschnitt 6.3.4.2). Die externen Lesezugriffe benötigen dadurch eine höhere Taktanzahl; dafür kann die gesamte Architektur mit einem höheren Systemtakt betrieben werden. Um die Leistungsfähigkeit mit dem bisherigen Omeganetzwerk anstellen zu können, wurde die Architektur mit dem Omeganetzwerk mit Registern ebenfalls auf dem Cyclone II und Stratix II FPGA realisiert. Der Realisierungsaufwand ist in Tabelle 6.7 und Tabelle 6.8 dargestellt.

p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
		LEs	Register			
1	2.824	-	-	195.296	1.585	145
4	10.029	416	181	289.568	5.091	105
8	19.699	1.113	361	415.264	9.712	90
16	40.445	3.276	721	666.656	18.965	70

Tabelle 6.7: Realisierungsaufwand der einarmigen universellen GCA-Architektur mit Omeganetzwerk mit Registerstufe (ONR) auf einem Cyclone II FPGA

Ein externer Lesezugriff benötigt bei dieser Architektur sechs Taktzyklen. Diese setzen sich zusammen aus zwei Taktzyklen für die Leseanfrage durch das Netzwerk und aus zwei Taktzyklen für den eigentlichen Speicherzugriff. Das Zurücksenden der Datenantwort benötigt weitere zwei Taktzyklen. Im Vergleich zum klassischen Omeganetzwerk benötigt jeder externe Lesezugriff auf Grund der Registerstufe zwei Taktzyklen mehr. Die Taktfrequenz konnte dadurch für mehr als vier Verarbeitungseinheiten um 15 MHz auf dem Cyclone II und um 20 MHz auf dem Stratix II gesteigert werden. Der zusätzliche Ressourcenbedarf für die Implementierung ist dagegen nur leicht angestiegen.

p	ALMs	ALUTs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			ALUTs	Register			
1	1.203	1.592	-	-	195.296	1.520	200
4	4.355	5.437	326	184	289.568	4.995	133
8	8.344	10.780	922	368	415.264	9.568	120
16	17.638	21.935	2.616	736	666.656	18.726	95
32	36.577	44.441	6.637	1.472	1.169.440	37.047	75

Tabelle 6.8: Realisierungsaufwand der einarmigen universellen GCA-Architektur mit Omeganetzwerk mit Registerstufe (ONR) auf einem Stratix II FPGA

6.6.3.2 Bitonisches Mischen

Das Bitonische Mischen wurde mit der gleichen Zellregel (Quellcode 6.1), wie in Abschnitt 6.6.2.2 dargestellt, auf der Hardwarearchitektur ausgeführt. Die Ergebnisse der Ausführung auf dem Cyclone II und dem Stratix II sind in Tabelle 6.9 und Tabelle 6.10 aufgeführt.

p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
1	781.567	5,39	-	4.179
4	201.926	1,92	2,80	11.714
8	102.932	1,14	4,71	19.697
16	52.442	0,75	7,19	30.070

Tabelle 6.9: Auswertung des Bitonischen Mischens auf der MPA mit einem Omeganetzwerk mit Registerstufe (ONR) auf einem Cyclone II FPGA

p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
1	781.567	5.764	-	5.764
4	201.926	14.838	2,57	14.838
8	102.932	26.263	4,56	26.263
16	52.442	40.810	7,08	40.810
32	26.899	62.812	10,90	62.812

Tabelle 6.10: Auswertung des Bitonischen Mischens auf der MPA mit einem Omeganetzwerk mit Registerstufe (ONR) auf einem Stratix II FPGA

Im Vergleich zu den Ergebnissen bei Verwendung des klassischen Omeganetzwerks bestätigen sich die Erwartungen, dass die Registerstufe erst bei einer größeren Anzahl an Verarbeitungseinheiten einen Geschwindigkeitsvorteil bei der Ausführung der Zellregel bietet. Für vier Verarbeitungseinheiten ist das klassiche Omeganetzwerk ohne Registerstufe bezüglich des Ressourcenbedarfs auf dem FPGA und auch bezüglich der Ausführungszeit der Zellregel besser geeignet. Die Taktfrequenz der Architektur ist in beiden Fällen identisch. Die zusätzlich benötigten Taktzyklen für einen externen Lesezugriff mindern die Leistungsfähigkeit der Architektur.
Für mehr als vier Verarbeitungseinheiten macht sich die zusätzliche Registerstufe im Omeganetzwerk positiv bemerkbar. Mit der nun höheren Taktfrequenz kann die Zellregel schneller

ausgeführt werden. Die Anzahl der externen Lesezugriffe ist im Vergleich zu der Anzahl an auszuführenden Operationen gering. Die für einen externen Lesezugriff zusätzlich benötigten Taktzyklen mindern die Systemleistung deshalb nur in geringem Umfang. Insgesamt erreicht man für eine größere Anzahl an Verarbeitungseinheiten deshalb eine Leistungssteigerung der Architektur. Die CUR für 32 Verarbeitungseinheiten steigt unter Verwendung der gleichen Zellregel (Quellcode 6.1 aus Abschnitt 6.6.2.2) von 47.269 auf jetzt 62.812.

6.6.3.3 Bitonisches Sortieren

Das Bitonische Sortieren aus Abschnitt 2.3.1 zum Sortieren einer Zahlenmenge wurde, ebenso wie das Bitonische Mischen, als Zellregel implementiert. Um die Leistungsfähigkeit der Architektur abschätzen zu können, wurde das Quicksortverfahren [Hoa62] implementiert. Die Komplexität des Bitonischen Sortierens ist mit $Nlog^2N$ zwar größer als die mittlere Komplexität des Quicksortverfahrens mit $NlogN$. Dafür lässt sich das Bitonische Sortieren einfach und effizient im GCA-Modell abbilden. Das Bitonische Sortieren hat den weiteren Vorteil, dass die Komplexität der Laufzeit unabhängig von den zu sortierenden Daten ist. Somit kann für jede Anzahl N an Eingangsdaten unabhängig von deren Permutation im Voraus die benötigte Laufzeit angegeben werden. Beim Quicksortverfahren ist die Laufzeit abhängig von der Permutation der Daten und der Wahl des Pivotelementes. Beim klassischen Quicksortverfahren beträgt die Komplexität für den schlechtesten anzunehmenden Fall N^2, für den idealen Fall $NlogN$. Um einen fairen Vergleich herstellen zu können, wurde daher das Quicksortverfahren in zwei Varianten implementiert. Die erste Variante ist das klassiche Quicksortverfahren mit einem Pivotelement. Die zweite Variante verwendet das mittlere von drei Pivotelementen.

Die Zellregel für das Bitonische Sortieren ist im Quellcode 6.2 dargestellt. Die Umsetzung basiert auf dem Bitonischen Mischen. Der Parameter `DAT_CELLS` gibt dabei die Anzahl der Zellen in der Architektur an. Die Zellregel ist unabhängig von einem Generationszähler implementiert und es muss nur die Anzahl der zu sortierenden Zellen angegeben werden.

Die Sortierung von 2048 Zahlen wurde auf dem Cyclone II und auf dem Stratix II durchgeführt. Die Ergebnisse sind für den Cyclone II in Tabelle 6.11 und für den Stratix II in Tabelle 6.12 aufgeführt. Für den Stratix II ergibt sich bei 32 Verarbeitungseinheiten eine CUR von 55.518 Zellen pro Millisekunde. Im Vergleich zum Bitonischen Mischen sinkt die Leistung der Architektur. Dies liegt hauptsächlich an der komplexeren Zellregel und an dem höheren Anteil an externen Lesezugriffen.

p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
1	4.971.726	34,29	-	3.942
4	1.443.694	13,75	2,49	9.830
8	726.336	8,07	4,25	16.748
16	366.650	5,24	6,55	25.805

Tabelle 6.11: Auswertung des Bitonischen Sortierens auf der MPA mit einem Omeganetzwerk mit Registerstufe (ONR) auf einem Cyclone II FPGA

```
1  int main(){
2   int ii,j,k,js;
3   int N,I;
4   int neighbour;
5
6   CI(NEXTGEN,0,0);                                    //Generationssynchronisation
7   CI(NEXTGEN,0,0);
8
9   for(j=1;j<DAT_CELLS;j=j<<1)
10  {
11   k=j;
12   js=j<<1;
13
14   while(k>=1)
15   {
16    for(ii=SC;ii<LOCL_CELLS+SC;ii++)
17    {
18     if(j!=k)
19     k=CI(INTERN_GET_P,ii,0);
20
21     if((ii & k)==0) neighbour=ii+k;          //Adressberechnung der Nachbarzelle
22     else neighbour=ii-k;
23
24     N = CI(EXTERN_GET_D,neighbour,0);
25     I = CI(INTERN_GET_D,ii,0);
26
27     if((ii & k)==0)                        //Bestimmung der Vertauschungsrichtung
28      if(!(ii&js)==(N<I)) CI(SET_D,ii,N);
29      else CI(SET_D,ii,I);
30     else
31      if(!(ii&js)==(I<N)) CI(SET_D,ii,N);
32      else CI(SET_D,ii,I);
33     CI(SET_PS,ii,k);
34    }
35
36    CI(NEXTGEN,0,0);                                  //Generationssynchronisation
37    k=CI(INTERN_GET_P,SC,0);
38   }
39  }
40  return 0;}
```

Quellcode 6.2: Zellregel des Bitonischen Sortierens in C

p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
1	4.971.726	24,86	-	5.437
4	1.443.694	10,85	2,29	12.452
8	726.336	6,05	4,11	22.331
16	366.650	3,86	6,44	35.022
32	182.598	2,43	10,21	55.518

Tabelle 6.12: Auswertung des Bitonischen Sortierens auf der MPA mit einem Omeganetzwerk mit Registerstufe (ONR) auf einem Stratix II FPGA

Die Vergleichswerte des Quicksortverfahrens sind in Tabelle 6.13 aufgeführt. Für die Auswertung wurde eine Architektur bestehend aus einem NIOS II/f Prozessor mit getrenntem Daten- und Programmspeicher verwendet. Die Taktfrequenz dieser Architektur beträgt 137,5 MHz. Der Datenspeicher dieser Architektur benötigt keine gesonderten Lesezugriffsbefehle.

	sortierte Daten		zufällige Daten	
Quicksort	Taktzyklen	Ausführungszeit [ms]	Taktzyklen	Ausführungszeit [ms]
Klassisch	23.314.342	169,56	502.411	3,65
Median	420.197	3,06	542.098	3,94

Tabelle 6.13: Taktzyklen und Ausführungszeiten für zwei Implementierungen des Quicksortalgorithmus auf einem NIOS II/f Prozessor

Die Ausführungszeiten der klassischen Quicksortimplementierung schwanken stark in Abhängigkeit von den zu sortierenden Daten. Im schlechtesten anzunehmenden Fall ist eine sortierte Liste zu sortieren. Hierfür werden 169,56 ms benötigt. Geht man davon aus, dass dieser Fall in der praktischen Anwendung nicht oft auftritt oder wird eine optimierte Variante eingesetzt, beträgt die Ausführungszeit zwischen 3 ms und 4 ms. Im direkten Vergleich mit dem Bitonischen Sortieren zeigt sich, dass für etwa die gleiche Ausführungszeit dafür 16 Prozessoren (Stratix II) benötigt werden.

6.6.4 Architektur mit anderen Netzwerken für Agenten

6.6.4.1 Realisierungsaufwand auf einem FPGA

Die bisherigen Architekturen mit Omeganetzwerk setzen das GCA-Modell um. Als Anwendung wurde das Bitonische Mischen und das Bitonische Sortieren beschrieben. Für diese Anwendungen ist auf Grund der Zugriffsstruktur das Omeganetzwerk sehr gut geeignet[5]. Alle externen Lesezugriffe können damit kollisionsfrei umgesetzt werden. Für die Realisierung von Agentenanwendungen (dem Fokus dieser Arbeit) unter Verwendung des GCA-Modells ist es im Hinblick auf die Architektur interessant, andere Netzwerkstrukturen zu untersuchen. Da sich die Zugriffsstrukturen je nach Anwendung, Anzahl der Agenten und Größe der Agentenwelt stark unterscheiden, scheint ein spezialisiertes Netzwerk nicht allgemein gut geeignet zu sein. Zusätzlich zu dem Omeganetzwerk mit verschiedenen Optimierungen und Erweiterungen wurden noch ein Busnetzwerk und ein Ringnetzwerk auf dem Cyclone II untersucht. In Tabelle 6.14 sind die benötigten Ressourcen und realisierbaren Taktfrequenzen für die verschiedenen Netzwerke aufgelistet. Dabei stehen die Abkürzungen ONR für das Omeganetzwerk mit Registerstufe, ONP für das Omegenetzwerk mit FIFO-Shuffleelementen, RN für das Ringnetzwerk, RNFF für das Ringnetzwerk mit Forwarding, BDPA für das Busnetzwerk mit dynamischer priorisierter Arbitrierung und BRRA für das Busnetzwerk mit Round-Robin Arbitrierung. Die höchsten Taktfrequenzen erreicht man mit dem Busnetzwerk und mit dem Ringnetzwerk. Der Verbrauch an Logikelementen ist bis auf das ONP für alle Netzwerke ähnlich. Der größere Verbrauch an

[5] Als Vergleich wurde das Bitonische Mischen und das Bitonische Sortieren unter der Verwendung weiterer Netzwerke evaluiert. Die Daten hierzu befinden sich im Anhang (Abschnitt 9.4)

Logikelementen und Registern ist durch den komplexeren Aufbau des Netzwerkes begründet (vgl. Abschnitt 6.3.4.4).

Netzwerk	p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			LEs	Register			
-	1	2.853	-	-	195.296	1.583	140,00
ONR	2	5.069	102	0	226.720	2.643	133,33
ONR	4	9.827	419	181	289.568	4.947	110,00
ONR	8	19.114	1.112	361	415.264	9.376	90,00
ONR	**16**	**38.662**	**3.053**	**721**	**666.656**	**18.245**	**70,00**
ONP	2	5.450	468	397	226.720	3.038	137,51
ONP	4	11.402	1.926	1.602	289.568	6.366	107,15
ONP	8	24.039	5.944	4.852	415.264	13.865	92,86
ONP	**16**	**52.097**	**16.158**	**13.064**	**666.656**	**30.586**	**70,84**
RN	2	5.135	147	124	226.720	2.765	135,00
RN	4	9.711	300	248	289.568	5.012	112,50
RN	8	18.573	588	496	415.264	9.509	90,00
RN	**16**	**37.040**	**1.200**	**992**	**666.656**	**18.514**	**75,00**
RNFF	2	5.250	292	204	226.720	2.845	140,00
RNFF	4	10.011	596	328	289.568	5.092	112,51
RNFF	8	19.098	1.090	576	415.264	9.589	91,67
RNFF	**16**	**38.017**	**2.063**	**1.072**	**666.656**	**18.594**	**73,81**
BDPA	2	5.051	48	2	226.720	2.643	135,72
BDPA	4	9.389	94	3	289.568	4.767	112,50
BDPA	8	18.200	234	4	415.264	9.017	93,75
BDPA	**16**	**35.949**	**501**	**5**	**666.656**	**17.527**	**75,00**
BRRA	2	5.006	48	2	226.720	2.643	140,00
BRRA	4	9.474	94	3	289.568	4.767	110,00
BRRA	8	18.178	232	4	415.264	9.017	90,00
BRRA	**16**	**36.135**	**490**	**5**	**666.656**	**17.527**	**70,00**

Tabelle 6.14: Realisierungsaufwand der einarmigen universellen GCA-Architektur auf einem Cyclone II FPGA für die Netzwerke: Omeganetzwerk mit Registerstufe (ONR), Omeganetzwerk mit FIFO-Shuffleelementen (ONP), Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA), Busnetzwerk mit Round-Robin Arbitrierung (BRRA).

Die Anzahl der benötigten Taktzyklen für einen externen Lesezugriff sind abhängig von dem verwendeten Netzwerk und der Zugriffe. Teilweise können mehrere Zugriffe parallel von einem Netzwerk verarbeitet werden. Für alle Netzwerke gilt, dass im ungünstigsten Fall ein externer Zugriff einer Verarbeitungseinheit blockiert ist, bis alle anderen Verarbeitungseinheiten die Generationssynchronisation erreicht haben. Spätestens zu diesem Zeitpunkt kann der externe Lesezugriff durchgeführt werden. Die Wahrscheinlichkeit, dass dieser Fall in praktischen Anwendungen auftritt, ist als sehr gering anzunehmen. Die externen Lesezugriffe sind nur ein sehr geringer Anteil der gesamten Zellregel bzw. des Agentenverhaltens. Deshalb können blo-

ckierte Zugriffe nach wenigen zusätzlichen Taktzyklen ausgeführt werden. Ein positiver Effekt der Blockierungen liegt darin, dass die einzelnen Verarbeitungseinheiten unterschiedlich lange blockiert werden. Die Abarbeitung der Zellregel und somit auch die externen Lesezugriffe auf den einzelnen Verarbeitungseinheiten sind damit zeitlich so verteilt, dass Blockierungen weitestgehend vermieden werden. Dieser Effekt wird durch eine größere Anzahl an Verarbeitungseinheiten gemindert.

Für das BDPA lässt sich die Anzahl an benötigten Takten T für einen externen Lesezugriff wie folgt abschätzen:

$$T = \begin{cases} 2, & VE_{last} = VE_{now} \\ 3, & VE_{last} \neq VE_{now} \end{cases}$$

Dabei bedeutet $VE_{last} = VE_{now}$, dass die Verarbeitungseinheit, die den letzten externen Lesezugriff durchgeführt hat (VE_{last}), bei einem erneuten externen Lesezugriff (VE_{now}) nur zwei Taktzyklen benötigt. Führt eine andere Verarbeitungseinheit einen externen Lesezugriff $(VE_{last} \neq VE_{now})$ aus, so dauert dieser drei Taktzyklen.

Für $1 < i \leq p$ gleichzeitige Lesezugriffe kann die Anzahl der benötigten Takte T mit $2 \leq T \leq 2 + 3 \cdot (i - 1)$ abgeschätzt werden.

Die Anzahl der benötigten Takte T für einen externen Lesezugriff unter Verwendung des BRRA werden in Abhängigkeit von der Anzahl $1 \leq i \leq p$ an gleichzeitig stattfindenden Lesezugriffen mit $T \leq z + 2 \cdot n$ Takte abgeschätzt, wobei $z \in \{\mathbb{Z}_0 | 0 < z < p\}$ ist. Somit kann die maximal benötigte Taktanzahl für einen externen Zugriff bestimmt werden.

Für das RN kann die Anzahl an benötigten Takten T mit $p + 3 \leq T \leq \infty$ abgeschätzt werden. Dabei bedeutet ∞, dass der externe Lesezugriff bis zur Generationssynchronisation blockiert sein kann.

Zusätzlich zu der Taktanzahl, die für einen externen Lesezugriff benötigt wird, muss der Parallelitätsgrad des Netzwerks berücksichtigt werden. Bei dem Busnetzwerk kann jeweils nur ein externer Lesezugriff abgearbeitet werden, während bei dem Ringnetzwerk bis zu p parallel abgearbeitet werden können.

Für die Betrachtung des kombinatorischen Omeganetzwerks (ON) sei auf [Hee07] verwiesen.

Für das BDPA, das BRRA und das RN wurden die Zugriffsmuster direkt auf der Hardware untersucht. Hierzu wurden die Zugriffsstrukturen während der Berechnung direkt vom FPGA aufgezeichnet. Für alle drei Verbindungsnetzwerke wurde die gleiche Anwendung verwendet (eine Agentenanwendung aus Abschnitt 6.6.4.2). Die verwendete Anwendung ist für die Zugriffsstrukturen aber nur von untergeordneter Bedeutung. Ein Auszug der aufgenommenen Daten ist in Abbildung 6.18 dargestellt, beginnend mit Taktzyklus 47. Dargestellt sind die Anfragesignale und die Antwortsignale für eine Architektur mit vier Verarbeitungseinheiten. Die Anfragesignale sind über die gesamte Dauer der Anfrage für jeden Taktzyklus als aktiv markiert. Die Antwortsignale sind nur in dem Takt als aktiv gekennzeichnet, in dem die Antwort bei

der anfragenden Verarbeitungseinheit angekommen sind. Taktzyklen, in denen keine Anfrage ansteht, wurden ausgelassen. Die erste Verarbeitungseinheit stellt in dem gezeigten Ausschnitt keine Anfragen an das Netzwerk, da der zu bearbeitende Bereich durch interne Lesezugriffe durchgeführt werden kann.

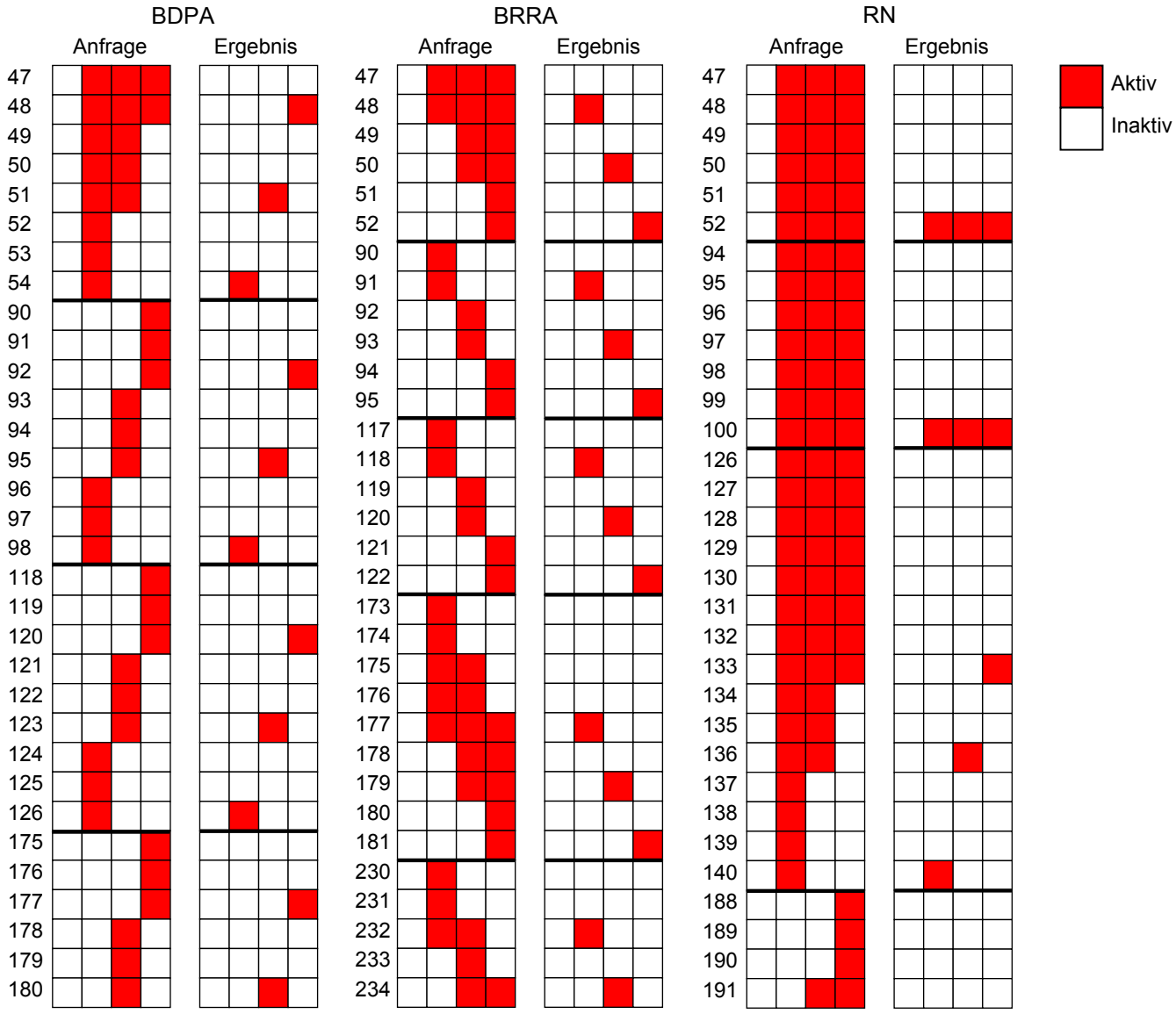


Abbildung 6.18: Zugriffsstruktur (Auszug) im Vergleich: Busnetzwerk mit dynamischer Arbitrierung (BDPA), Busnetzwerk mit Round-Robin Arbitrierung (BRRA) und Ringnetzwerk (RN) für vier Verarbeitungseinheiten

Wie in der Abbildung 6.18 dargestellt, bietet die dynamische Arbitrierung des Busnetzwerks eine konstante Zugriffszeit (bei einem Zugriff). Mehrere parallele Zugriffe, wie zu Beginn der Aufzeichnung zu sehen, werden zeitlich verteilt abgearbeitet. Dieses zeitlich versetzte Zugriffsmuster wird sehr gut über die gesamte Aufzeichnung hinweg gehalten. Abhängig vom Status des Arbiters benötigt ein Zugriff zwei oder drei Taktzyklen (ohne Wartezyklen). Die Round-Robin Arbitrierung für das Busnetzwerk hat variierende Zugriffszeiten. Dies kann den Zeilen zum Zeitpunkt 90 und 173 entnommen werden. Zum Zeitpunkt 90 wird der externe Lesezugriff genau so

schnell wie ein interner Lesezugriff abgearbeitet. Zum Zeitpunkt 173 hingegen werden für den externen Lesezugriff fünf Taktzyklen benötigt: davon drei Taktzyklen für die Arbitrierung. Für einzelne Lesezugriffe ist deshalb das Busnetzwerk mit dynamischer Arbitrierung besser geeignet, da alle Zugriffe schneller oder gleichschnell abgearbeitet werden. Ein Vergleich der ersten Zeile zeigt zudem das unterschiedliche Arbitrierungsverhalten, wobei die Round-Robin Arbitrierung auch ein anderes Muster aufzeigen könnte, abhängig vom Zeitpunkt des Zugriffs und des internen Status des Arbiters zum Zeitpunkt des Zugriffs. Der Vergleich mit dem Ringnetzwerk zeigt, dass es schneller als die Round-Robin, aber langsamer als die dynamische Arbitrierung ist. Dies stimmt aber nicht für die Gesamtperformance. Im Gegensatz zu dem Busnetzwerk können im Ringnetzwerk mehrere Anfragen parallel bearbeitet werden. Dies zeigen die Antwortsignale zum Zeitpunkt 52. Die ersten Zugriffe benötigen sechs Taktzyklen, was die best mögliche Zugriffszeit ist. Vier Taktzyklen werden für die Anfrage und die Bestätigung benötigt, um im Ring weitergereicht zu werden und zwei Taktzyklen für den Speicherzugriff. Die danach folgenden Anfragen benötigen mehr als sechs Taktzyklen, was zusätzliche Wartezyklen auf Grund eines vollen Rings anzeigt.

6.6.4.2 Eine Agentenanwendung

Für die Auswertung der entwickelten Architekturen wurde eine Agentenwelt (Abb. 6.19) definiert. Sie dient dem Vergleich der unterschiedlichen Architekturen untereinander. Die Agentenwelt besteht aus $n = 45 \times 45 = 2025$ Zellen. Die Architektur unterstützt ein Zellfeld von $n = 2^{11} = 2048$ Zellen. Die 23 Zellen, die nicht von der Agentenwelt verwendet werden, werden mit Hindernissen aufgefüllt. In der Agentenwelt existieren zusammengefasst $a_g = a_m + h = 185 + 176 = 361$ Agenten. Diese sind aufgeteilt in $a_m = 185$ sich bewegende Agenten und $h = 176$ Hindernisse (nicht bewegbare Agenten). Im Folgenden sind mit Agenten nur die sich bewegenden Agenten gemeint.

Die Agenten wurden zufällig auf dem Zellfeld angeordnet. Damit wird verhindert, dass bestimmte Eigenschaften einer Architektur besonders gut oder besonders schlecht ausgenutzt werden. Hierbei sind vor allem Zugriffskonflikte im Verbindungsnetzwerk von Bedeutung. Das Verhalten der Agenten (Abb. 6.20) ist durch folgende Regel, die aus zwei grundlegenden Aktionen besteht, gegeben:

1. Der Agent bewegt (b) sich um ein Feld nach vorne und dreht sich anschließend. Die Blickrichtung des Agenten wechselt dabei zwischen Nord und Ost bzw. zwischen Süd und West.

2. Der Agent bewegt sich nicht (s), dreht sich aber. Der Agent dreht sich so lange im Uhrzeigersinn, bis er sich wieder bewegen kann.

Durch dieses Agentenverhalten laufen die Agenten diagonal in der Agentenwelt. Im Laufe der Simulation sammeln sich daher alle Agenten um die Diagonale der Agentenwelt. Dort kommt es dann vermehrt zu Blockierungen zwischen den Agenten. Für die Überprüfung der Blickrichtung der Agenten sind dann weitere Nachbarzugriffe und Überprüfungen notwendig. Die zu überprüfenden Nachbarzellen einer leeren Zelle sowie die zu überprüfenden Nachbarzellen einer Agentenzelle sind in Abbildung 6.21 dargestellt. Die leere Zelle prüft die vier angrenzenden Nachbarzellen auf einen Agenten in Blickrichtung der leeren Zelle. Die Agentenzelle prüft nur

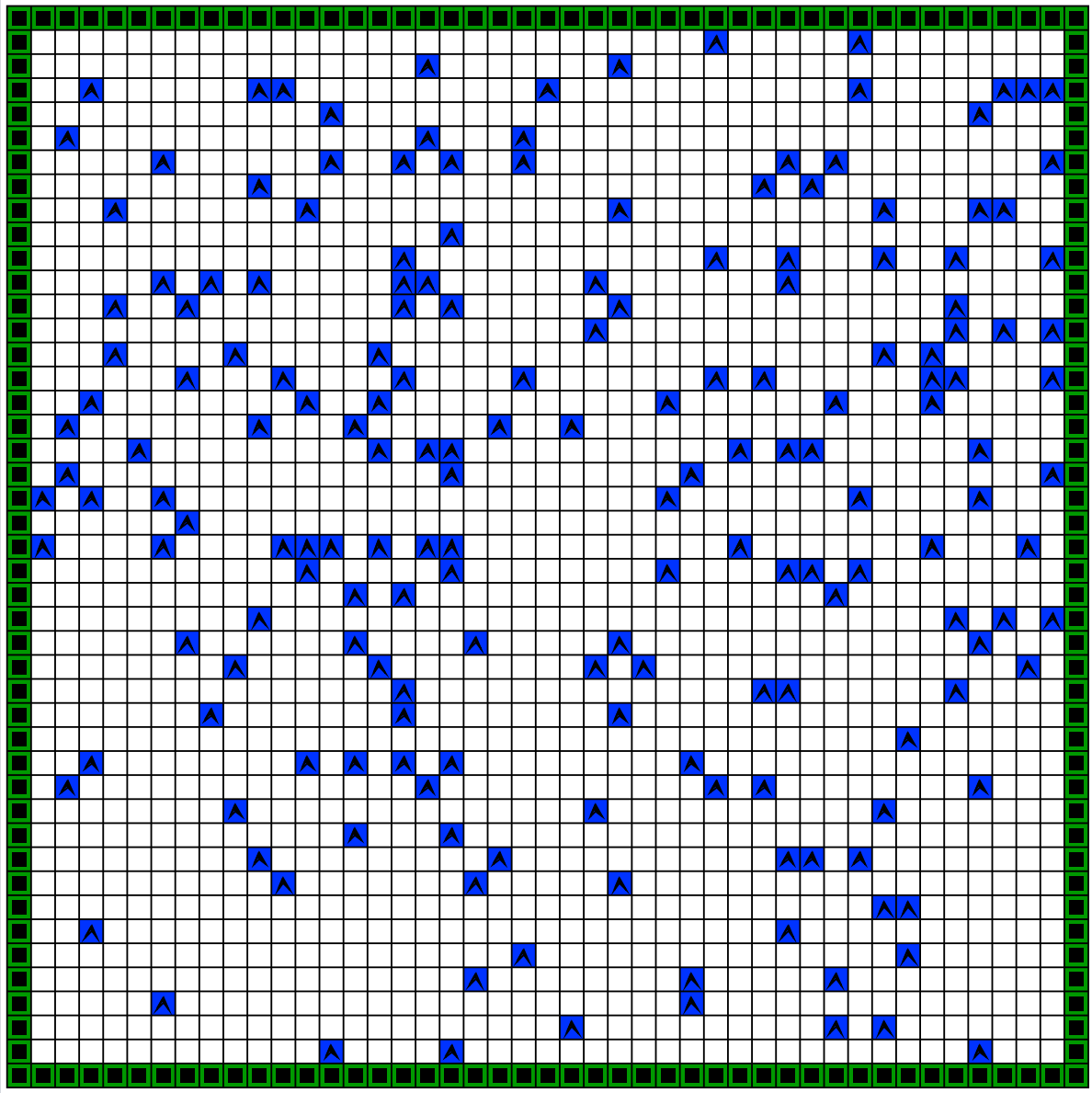


Abbildung 6.19: Simulation einer Agentenwelt: Agenten mit Richtung (blau, ■), Hindernisse (grün, ■), freie Zellen (weiß, □)

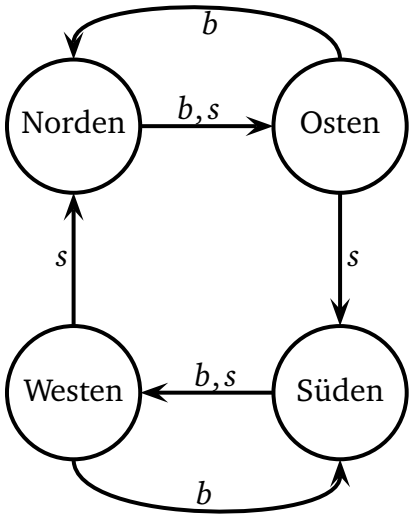


Abbildung 6.20: Das Agentenverhalten, dargestellt als endlicher Automat, mit den vier Blickrichtungen als Zustände. Die möglichen Aktionen: bewegen (b), stehen bleiben (s)

die Zellen in Blickrichtung des Agenten. Die Frontzelle muss nur auf das Vorhandensein eines Agenten überprüft werden. Die an die Frontzelle angrenzenden Zellen müssen auf Agenten mit Blickrichtung auf die Frontzelle überprüft werden.

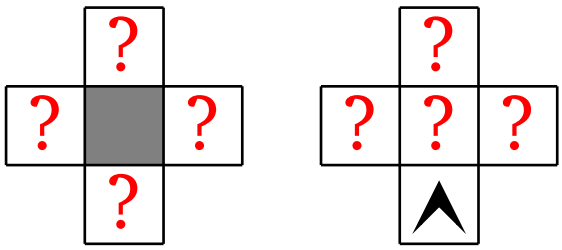


Abbildung 6.21: Zellüberprüfungen einer leeren Zelle (links) und einer Agentenzelle (rechts)

Die Hindernisse bilden einen Rand um die Welt und können von den Agenten nicht bewegt oder modifiziert werden. Eine gesonderte Betrachtung für den Rand entfällt somit. Denkbar wäre auch, die Welt zyklisch zu schließen. Durch den Rand verringert sich die Feldgröße, auf der sich die Agenten bewegen können, auf 44×44.

Jede Zelle C_i mit dem Index i speichert den Agententyp T und für einen beweglichen Agenten zusätzlich die Blickrichtung D des Agenten $\big(C_i = (T, D)\big)$. Im Folgenden wird vereinfacht $T(i)$ geschrieben, wenn der Agententyp der Zelle C_i gemeint ist. Mit $D(i)$ ist die Blickrichtung des Agenten der Zelle C_i gemeint.

Die Zellregel, d. h. das Agentenverhalten, ist durch die folgenden Formeln definiert. Die Funktion f_a wertet dabei aus, ob auf einer Zelle ein Agent mit einer bestimmten Blickrichtung vorhanden ist. Diese Funktion wird von der Funktion f_{free} verwendet, um die Nachbarschaft einer leeren Zelle E bzw. die einer Agentenzelle A zu prüfen. Die neue Blickrichtung des Agenten wird

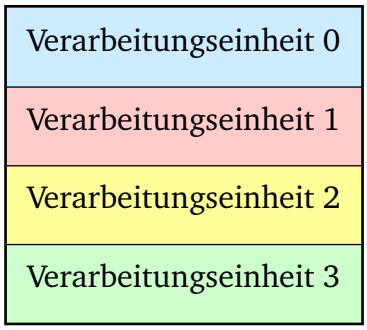


Abbildung 6.22: Aufteilung der Zellen der Agentenwelt auf die Verarbeitungseinheiten der Hardwarearchitekturen

mit der Funktion D^+ und der neue Agententyp mit der Funktion T^+ bestimmt. Die Funktion f_e dient nur zur vereinfachten Notierung der übrigen Funktionen. Die Blickrichtungen der Agenten sind mit: Norden: NORD, Osten: OST, Süden: SÜD und Westen: WEST definiert. Für den Agententyp sind definiert: Agent: A, Hindernis: H und leere Zelle: E. Der neue Zustand einer Zelle C_i^+ ergibt sich durch Anwendung der beiden Funktionen D^+ und T^+ auf diese Zelle.

$$f_a(i,k) := \begin{cases} 1, & (T(i)) = A \wedge (D(i) = k) \\ 0, & sonst \end{cases}$$

$$f_t(k) := \begin{cases} \text{NORD}, & k = \text{OST} \\ \text{OST}, & k = \text{SÜD} \\ \text{SÜD}, & k = \text{WEST} \\ \text{WEST}, & k = \text{NORD} \end{cases}$$

$$f_{free}(i) := \begin{cases} \infty, & T(i) = A \\ f_a(i-X, \text{SÜD}) + f_a(i+1, \text{WEST}) + f_a(i+X, \text{NORD}) + f_a(i-1, \text{OST}), & sonst \end{cases}$$

$$f_e(i) := \begin{cases} f_{free}(i-X), & D(i) = \text{NORD} \\ f_{free}(i+X), & D(i) = \text{SÜD} \\ f_{free}(i-1), & D(i) = \text{WEST} \\ f_{free}(i+1), & D(i) = \text{OST} \end{cases}$$

$$D^+(i) := \begin{cases} \text{NORD}, & (T(i) = A) \wedge (D(i) = \text{OST}) \wedge \left(f_{free}(i+1) = 1\right) \\ \text{OST}, & (T(i) = A) \wedge (D(i) = \text{NORD}) \wedge \left(f_{free}(i-X) = 1\right) \\ \text{SÜD}, & (T(i) = A) \wedge (D(i) = \text{WEST}) \wedge \left(f_{free}(i-1) = 1\right) \\ \text{WEST}, & (T(i) = A) \wedge (D(i) = \text{SÜD}) \wedge \left(f_{free}(i+X) = 1\right) \\ f_t(D(i)), & (T(i) = A) \wedge \left(f_e(i) \neq 1\right) \\ -, & sonst \end{cases}$$

$$T^+(i) := \begin{cases} E, & (T(i) = A) \wedge \left(f_e(i) = 1\right) \\ A, & (T(i) = E) \wedge \left(f_{free}(i) = 1\right) \\ A, & (T(i) = A) \wedge \left(f_e(i) \neq 1\right) \\ E, & (T(i) = E) \wedge \left(f_{free}(i) \neq 1\right) \\ H, & (T(i) = H) \end{cases}$$

Die Auswertung für 100 Generationen unter Verwendung der Netzwerke ONR, ONP, RN, RNFF, BDPA und BRRA ist in Tabelle 6.15 wiedergegeben. Die Zellen der Agentenwelt werden dabei wie in Abbildung 6.22 dargestellt, zeilenweise auf die Verarbeitungseinheiten aufgeteilt. Die Berechnung der CUR zeigt, dass das BDPA mit einer CUR von 11.255 Zellen pro Millisekunde die Agentenwelt am schnellsten simulieren kann ($p = 16$). Das ONR schneidet mit einer CUR von 4.473 Zellen pro Millisekunden für $p = 16$ am schlechtesten ab. Dies liegt an dem zusätzlichen Synchronisierungsaufwand für die Lesezugriffe. Dadurch kann auch die `AUTO_READ` Funktion nicht verwendet werden. Im direkten Vergleich mit dem ONP mit einer CUR von 10.137 Zellen pro Millisekunde für $p = 16$ zeigen sich die Auswirkungen der `AUTO_READ` Funktion. Das RNFF schneidet, trotz der vielen Registerstufen auf Grund der Möglichkeit parallele Zugriffe durchzuführen, gut ab. Das RN zeigt ähnliche CURs wie das BRRA. Die schnelleren einzelnen Zugriffe des BRRA sind damit ähnlich leistungsstark wie die parallelen, dafür länger andauernden Zugriffe des RN.

Die Ergebnisse in Tabelle 6.15 sind spezifisch für die simulierte Agentenwelt. Abhängig von der Komplexität der Zellregel, den Zugriffsstrukturen, der Größe der Agentenwelt, der Anzahl der Agenten und weiterer Parameter ergeben sich andere Beschleunigungen (Speedup) und andere CURs. Wie im Verlauf dieser Arbeit gezeigt wird, ist der Speedup relativ unabhängig von der Agentenanwendung. So wurde auf der in Abschnitt 6.9 beschriebenen Hardwarearchitektur die gleiche Agentenwelt, wie hier beschrieben, und zusätzlich eine erweiterte Version (Eine Agentenanwendung mit Informationsverbreitung in Abschnitt 6.9.5) simuliert. Der Speedup für 16 Verarbeitungseinheiten unterscheidet sich nur um 0,43 wobei die komplexere Zellregel den größeren Speedup erlangte.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
-	1	18.874.706	134,82	-	1.519
ONR*	2	18.986.231	142,40	0,95	1.438
ONR*	4	11.031.769	100,29	1,34	2.042
ONR*	8	5.879.344	65,33	2,06	3.135
ONR*	**16**	**3.204.694**	**45,78**	**2,94**	**4.473**
ONP	2	9.785.927	71,17	1,89	2.877
ONP	4	5.029.474	46,94	2,87	4.363
ONP	8	2.640.628	28,44	4,74	7.201
ONP	**16**	**1.431.096**	**20,20**	**6,67**	**10.137**
RN	2	9.791.713	72,53	1,86	2.823
RN	4	5.040.248	44,80	3,01	4.571
RN	8	2.677.569	29,75	4,53	6.883
RN	**16**	**1.544.556**	**20,59**	**6,55**	**9.944**
RNFF	2	9.791.806	69,94	1,93	2.928
RNFF	4	5.029.877	44,71	3,02	4.581
RNFF	8	2.642.482	28,83	4,68	7.104
RNFF	**16**	**1.460.967**	**19,79**	**6,81**	**10.346**
BDPA	2	9.775.095	72,02	1,87	2.843
BDPA	4	4.997.752	44,42	3,03	4.610
BDPA	8	2.586.247	27,59	4,89	7.423
BDPA	**16**	**1.364.675**	**18,20**	**7,41**	**11.255**
BRRA	2	9.777.226	69,84	1,93	2.932
BRRA	4	5.004.627	45,50	2,96	4.501
BRRA	8	2.616.640	29,07	4,64	7.044
BRRA	**16**	**1.442.007**	**20,60**	**6,54**	**9.941**

Tabelle 6.15: Auswertung der Agentenwelt auf der MPA auf einem Cyclone II FPGA für die Netzwerke: Omeganetzwerk mit Registerstufe (ONR), Omeganetzwerk mit FIFO-Shuffleelementen (ONP), Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA), Busnetzwerk mit Round-Robin Arbitrierung (BRRA).
*Verwendet nicht die `AUTO_READ` Funktion

Der Quellcode 6.3 zeigt die Zellregel, die das Agentenverhalten umsetzt. Die Funktion `evaluateField` in Zeile 1 bis 26 dient dabei der Kollisionsbehandlung. Ausgehend von einer Zelle mit der Adresse `fieldADR` werden alle vier Nachbarzellen (Norden, Süden, Osten, Westen) auf mögliche Agenten geprüft. In dem Fall, dass auf einer Nachbarzelle, z. B. Norden, ein Agent vorhanden ist, muss zusätzlich dessen Blickrichtung geprüft werden. Nur wenn die Blickrichtung des Agenten in diesem Fall nach Süden ausgerichtet ist, ist eine Kollisionssituation gegeben. Die Anzahl der Agenten, die sich auf die Zelle mit der Adresse `fieldADR` bewegen möchten, wird mit `count` gezählt. Ein Agent kann sich auf die vor ihm befindliche leere Zelle bewegen, wenn die Auswertung der Funktion den Wert eins ergibt. Die Funktion ist so ausgelegt, dass sie sowohl von einer leeren als auch von einer Agentenzelle verwendet werden kann. Für eine Agentenzelle muss dafür die Adresse der Zelle, die sich in Bewegungsrichtung des Agenten vor ihm befindet, verwendet werden. Für eine leere Zelle wird die Adresse der leeren Zelle verwendet.

Der Zellinhalt einer Zelle wird über vordefinierte Werte festgelegt. Eine freie Zelle wird mit `CELL_FREE`, eine Agentenzelle mit `CELL_AGENT` und ein Hindernis mit `CELL_OBSTACLE` definiert. Die Blickrichtungen sind über die Konstanten `NORTH` für Norden, `SOUTH` für Süden, `WEST` für Westen und `EAST` für Osten definiert. Für die Berechnung der Nachbarzellen muss die Dimension der Agentenwelt bekannt sein. Die Breite der Agentenwelt ist durch `MAXX` und die Höhe durch `MAXY` gegeben.

Im Unterschied zum Bitonischen Mischen und Sortieren und ähnlichen Algorithmen folgen die Lesezugriffe bei der Simulation von Agentensystemen keinem bestimmtem Muster (vgl. Abschnitt 6.6.2.2 und Abschnitt 6.6.3.3). Die Zugriffsmuster sind von zu vielen Parametern abhängig, als dass ein generelles Zugriffsmuster entstehen könnte. Die Zellregel kann auch mit dem Omeganetzwerk verwendet werden. Dafür müssen die `AUTO_GET` Befehle durch die externen Lesebefehle ersetzt werden.

```
1  int evaluateField(int fieldADR, int *src){                //Prüfe Nachbarzellen
2   if(CI(AUTO_GET_D,fieldADR,0)!=CELL_FREE)
3    {return 9999;}                                        //Zielzelle ist nicht frei
4   int count=0, N,S,E,W;
5
6   N=fieldADR-MAXY;                                                 //Zelle im Norden
7   if((CI(AUTO_GET_D,N,0)==CELL_AGENT) && (CI(AUTO_GET_P,N,0)==SOUTH))
8    {count++; *src=N;}
9
10  S=fieldADR+MAXY;                                                 //Zelle im Süden
11  if((CI(AUTO_GET_D,S,0)==CELL_AGENT) && (CI(AUTO_GET_P,S,0)==NORTH))
12   {count++; *src=S;}
13
14  E=fieldADR+1;                                                    //Zelle im Osten
15  if((CI(AUTO_GET_D,E,0)==CELL_AGENT) && (CI(AUTO_GET_P,E,0)==WEST))
16   {count++; *src=E;}
17
18  W=fieldADR-1;                                                  //Zelle im Westen
19  if((CI(AUTO_GET_D,W,0)==CELL_AGENT) && (CI(AUTO_GET_P,W,0)==EAST))
20  {
21   if(count>=1) return 9999;
22   count++; *src=W;
```

```
23  }
24
25  return count;
26 }
27
28 int main(){
29  int g;                                                  //Generationszähler
30  int ii;                                                 //Lokaler Adresszähler
31  int newADR, direction, source, c;
32  int CellContentD;                                                 //Zellinhalt
33
34  CI(NEXTGEN,0,0);
35  CI(NEXTGEN,0,0);
36
37  for(g=0;g<GENS;g++)
38  {
39   for(ii=SC;ii<LOCL_CELLS+SC;ii++)
40   {
41    CellContentD=CI(INTERN_GET_D,ii,0);                          //Lade Zellinhalt
42
43    if(CellContentD==CELL_FREE){                                  //Zelle ist frei
44
45     c=evaluateField(ii, &source);                              //Kollisionsbehandlung
46     direction=CI(AUTO_GET_P,source,0);
47
48     if(c==1){
49      switch(direction){
50       case NORTH: direction=EAST;  break;
51       case SOUTH: direction=WEST;  break;
52       case EAST:  direction=NORTH; break;
53       case WEST:  direction=SOUTH; break;
54      }
55
56      CI(SET_D,ii,CELL_AGENT);
57      CI(SET_P,ii,direction);
58     }
59     else{
60      CI(SET_D,ii,CELL_FREE);
61     }
62    }
63    else{
64     if(CellContentD==CELL_AGENT){                               //Zelle ist Agent
65      direction=CI(INTERN_GET_P,ii,0);
66
67      switch(direction){
68       case NORTH: newADR=ii-MAXY; break;
69       case SOUTH: newADR=ii+MAXY; break;
70       case EAST:  newADR=ii+1;    break;
71       case WEST:  newADR=ii-1;    break;
72      }
73
74      c=evaluateField(newADR, &source);                         //Kollisionsbehandlung
75
76      if(c==1){
77       CI(SET_D,ii,CELL_FREE);
```

```
78      }
79      else{
80       CI(SET_D,ii,CELL_AGENT);
81       if(direction==WEST) direction=NORTH; else direction+=2;
82       CI(SET_P,ii,direction);
83      }
84     }
85     else{                                                    //Zelle ist Hindernis
86      CI(SET_D,ii,CELL_OBSTACLE);
87     }
88    }
89   }
90   CI(NEXTGEN,0,0);                                        //Generationssynchronisation
91  }
92 return 0;}
```

Quellcode 6.3: Zellregel der Agentenanwendung für die MPA

6.6.5 Bewertung der Hardwarearchitektur

Mit der MPA steht die erste Hardwarearchitektur auf der Basis von NIOS II Prozessoren für das GCA-Modell zur Verfügung. Die Auswertung wurde zunächst für das Bitonische Mischen und das Bitonische Sortieren vorgenommen. Als Netzwerk wurde ein Omeganetzwerk verwendet. Die Verwendung des Omeganetzwerks zeigte, dass durch die große kombinatorische Logik die Taktfrequenz stark sinkt. Durch das Einfügen einer Registerstufe kann der Abfall der Taktfrequenz begrenzt werden. Die Architektur wurde für bis zu 32 Prozessoren ausgewertet. Für eine größere Anzahl an Verarbeitungseinheiten sind dann noch weitere Registerstufen notwendig. Aus diesem Grund und auch im Hinblick auf die Entwicklung von Multi-Agenten-Anwendungen wurden noch weitere Netzwerke untersucht. Die Architektur erlaubt zudem nur das Abspeichern von zwei Werten pro Zelle. Damit ist die allgemeine Verwendbarkeit der Architektur eingeschränkt. Die wichtigsten Werte der Software sowie der Hardwarearchitektur sind zusammengefasst in Tabelle 6.16 aufgeführt.

	Software				Hardware		
Netzwerk	p	Taktzyklen	Speedup	CUR $\left[\frac{1}{ms}\right]$	LEs	Register	max. Takt [MHz]
-	1	18.874.706	-	1.519	2.853	1.583	140,00
ONR†	16	3.204.694	2,94	4.473	38.662	18.245	70,00
ONP	16	1.431.096	6,67	10.137	52.097	30.586	70,84
RN	16	1.544.556	6,55	9.944	37.040	18.514	75,00
RNFF	16	1.460.967	6,81	10.346	38.017	18.594	73,81
BDPA	16	1.364.675	7,41	11.255	35.949	17.527	75,00
BRRA	16	1.442.007	6,54	9.941	36.135	17.527	70,00

Tabelle 6.16: Zusammenfassung der Ergebnisse der Agentensimulation auf der MPA.
†Verwendet nicht die `AUTO_READ` Funktion

6.7 Multiarmige speicherkonfigurierbare universelle GCA-Architektur (MPAB)

Die Architektur aus Abschnitt 6.6 unterstützt zwei 16 Bit große Speicherplätze (L und D). Diese zwei Speicherplätze sind für viele Agentenanwendungen nicht ausreichend, so dass eine erweiterte, multiarmige speicherkonfigurierbare universelle Hardwarearchitektur, abgekürzt MPAB (engl.: Multiprocessor Architecture Blocks), entwickelt wurde [SHH10b, SHH11a]. Des weiteren hat sich die Unterscheidung in ein Linkfeld L und ein Datenfeld D auf Modell- sowie auf Architekturebene als nicht zweckmäßig erwiesen, weshalb die Speicherplätze nun *Block B* genannt werden. Ein Block ist ein Speicherplatz mit einer einstellbaren Größe von 1 Bit bis zu 32 Bit der für beliebige Daten- und Zeigerinformationen verwendet werden kann. Die tatsächliche Verwendung eines Blocks wird durch die jeweilige Anwendung bestimmt. Die Anzahl der Blöcke ist über einen Parameter einstellbar und kann frei konfiguriert werden. Für komplexeres Agentenverhalten wurde zusätzlich die Möglichkeit der Zufallszahlengenerierung in die Architektur integriert. Im GCA-Modell können Zufallszahlen nicht während der Abarbeitung der Zellregel generiert werden (siehe hierzu Abschnitt 3.5.2). Eine gesonderte Bereitstellung zum generieren von Zufallszahlen muss in der Architektur vorhanden sein. Die Verwendung von Zufallszahlen ermöglicht individuelles und komplexes Agentenverhalten bei identischer Zellregel. Die so erweiterte Architektur ist in Abbildung 6.23 dargestellt.

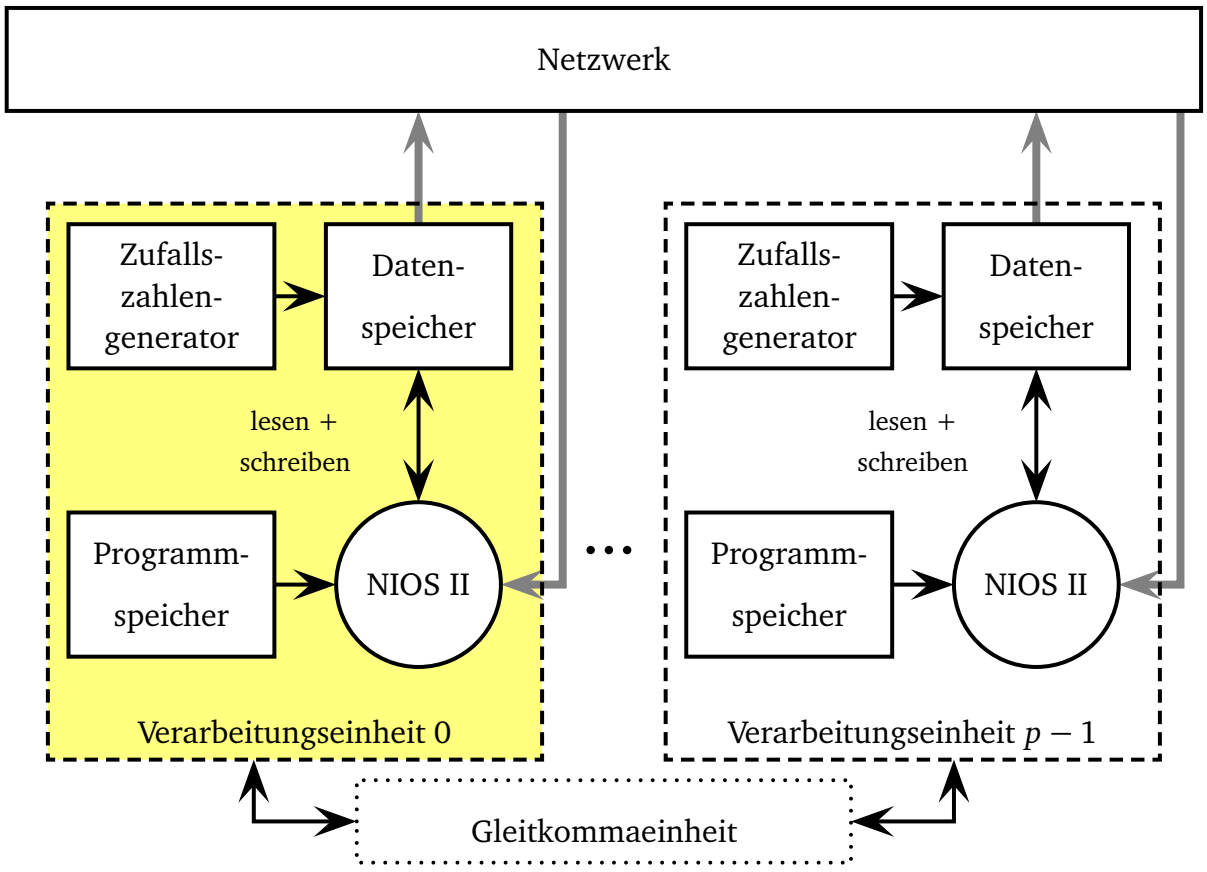


Abbildung 6.23: Multiarmige speicherkonfigurierbare universelle GCA-Architektur (MPAB) mit optionaler Gleitkommaeinheit

Die Architektur besteht, wie zuvor, aus p NIOS II Prozessoren. Jeder Prozessor ist mit einem Programmspeicher verbunden und besitzt Lese- und Schreibzugriff auf einen Teil des gesamten Zellfeldes bzw. der Agentenwelt, die in den Datenspeichern hinterlegt ist. Auf die Zellen, die in anderen Datenspeichern gespeichert sind, besteht über ein Verbindungsnetzwerk (siehe Abschnitt 6.3) lesender Zugriff. In den Datenspeichern sind die Blöcke jeder Zelle gespeichert. Ein Zufallszahlengenerator erzeugt Zufallsdaten, die im Datenspeicher abgespeichert werden. Die genaue Funktionsweise und Problematik bei Zufallszahlen im GCA-Modell und die Umsetzung in der Architektur wird in Abschnitt 6.7.1 behandelt. Eine Verarbeitungseinheit (VE) besteht bei dieser Architektur aus einem Programmspeicher, dem NIOS II Prozessor, einem Datenspeicher und einem Zufallszahlengenerator. Des weiteren kann optional eine Gleitkommaeinheit nach dem IEEE754 Standard [IEE85] für die Architektur hinzugefügt werden.

6.7.1 Umsetzung von Zufallszahlen

Die Erzeugung und Verwendung von Zufallszahlen im GCA-Modell erfordert eine gesonderte Betrachtung. Die Erklärung, wie Zufallszahlen im GCA-Modell umgesetzt werden können, ist in Abschnitt 3.5.2 aufgeführt. Im Gegensatz dazu wird im Folgenden die technische Realisierung erläutert.

Die Zufallszahlen werden über ein linear rückgekoppeltes Schieberegister (engl.: Linear Feedback Shift Register, LFSR) erzeugt [Tex96, Alf96]. Jede Verarbeitungseinheit enthält dafür ein LFSR, das während eines Schreibzugriffs auf eine Zelle eine neue Zufallszahl erzeugt. Das Abspeichern der Zufallszahl erzeugt dabei keine Verzögerung des Schreibzugriffs. Die Zufallszahlen werden für jede Zelle in einen Block geschrieben. Nach Abschluss des Schreibvorgangs enthält jede Zelle in einem Block eine Zufallszahl, die von der Anwendung in der nächsten Generation verwendet werden kann. Die Agenten erhalten dadurch bei gleicher Zellregel ein individuelles Verhalten.

6.7.2 Umsetzung von Gleitkommaoperationen

Die Gleitkommaeinheit unterstützt die Berechnung (Addition, Subtraktion, Division, Multiplikation und Quadratwurzel) von Gleitkommazahlen mit einfacher Genauigkeit nach dem IEEE754 Standard [IEE85]. Der NIOS II Prozessor unterstützt zwar auch optional die Behandlung von Gleitkommazahlen, allerdings werden dann für jeden Prozessor Hardwareressourcen benötigt, was die Skalierbarkeit zusätzlich weiter einschränkt. Da die Gleitkommaoperationen nur im Pipelinemodus arbeiten ($\pm$: 7 Taktzyklen, $*$: 5 Taktzyklen, $\div$: 6 Taktzyklen, $\sqrt{}$: 16 Taktzyklen), wurde der Ansatz einer zentralen Gleitkommaeinheit gewählt. Bei gleichzeitigem Zugriff auf die gleiche Gleitkommaoperation von mehreren Prozessoren selektiert der Arbiter der Gleitkommaeinheit nacheinander die Prozessoren. Für p parallele Zugriffe entsteht damit für den letzten Zugriff eine Verzögerung von $p - 1$ Taktzyklen. Eine parallele Verwendung verschiedener Gleitkommaoperationen durch verschiedene Prozessoren ist möglich. Insgesamt wird die Auslastung der Gleitkommaeinheit bei geringem Performanceverlust gesteigert und die Skalierbarkeit erhalten.

6.7.3 Definierte Custom Instructions

Die Zugriffe auf die einzelnen Blöcke sind über verschiedene Custom Instructions (CI) realisiert. Dabei gibt es für jeden Block einer Zelle einen eigenen Lese- und Schreibzugriff. Die RD_B{W} Funktion entscheidet dabei in Hardware, ob der Lesezugriff intern oder extern auszuführen ist.

Für die Generierung der Zufallszahlen existiert keine Custom Instruction. Der Speicherplatz der Zufallszahl wird bei jedem Schreibzugriff auf diese Zelle mit einer neuen Zufallszahl überschrieben. Da jede Zelle mindestens einmal geschrieben wird, ist sichergestellt, dass jeder Zelle für die nächste Generation eine neue Zufallszahl zur Verfügung steht. Die generierten Zufallszahlen können über eine der Lesefunktionen, wie jeder andere Block auch, gelesen werden. Somit besteht auch Lesezugriff auf die Zufallszahlen anderer Zellen.

Für die Gleitkommaberechnung steht für jede Funktion eine separate Custom Instruction zur Verfügung. Die Gleitkommaoperationen arbeiten im Pipelinemodus und benötigen unterschiedlich viele Taktzyklen für die Ausführung. Hinzu kommen evtl. noch Wartezyklen bei gleichzeitigen Zugriffen.

Die definierten Custom Instructions dieser Architektur für B Blöcke sind:

- RD_B{W}: Lesezugriff auf den Block W ($W \in \{0, ..., B-1\}$). Über einen kombinatorischen Adressvergleich wird der Lesezugriff wenn möglich intern ansonsten extern ausgeführt.
- EXT_RD_B{W}: Externer Lesezugriff auf den Block W ($W \in \{0, ..., B-1\}$) unter Verwendung des Netzwerks.
- INT_RD_B{W}: Interner Lesezugriff auf den Block W ($W \in \{0, ..., B-1\}$).
- WR_B{W}: Schreibt den Block W ($W \in \{0, ..., B-1\}$).
- NEXTGEN: Generationssynchronisation
- FP_SQRT: Berechnung der Quadratwurzel als Gleitkommazahl mit einfacher Genauigkeit.
- FP_DIV: Berechnung der Ganzzahldivision als Gleitkommazahl mit einfacher Genauigkeit.
- FP_MULT: Berechnung der Multiplikation als Gleitkommazahl mit einfacher Genauigkeit.
- FP_SUB: Berechnung der Subtraktion als Gleitkommazahl mit einfacher Genauigkeit.
- FP_ADD: Berechnung der Addition als Gleitkommazahl mit einfacher Genauigkeit.

6.7.4 Realisierungsaufwand auf einem FPGA

6.7.4.1 Realisierung ohne Gleitkommaeinheit

Die Architektur wurde auf einem Cyclone II FPGA realisiert. Auf Grund der Erfahrungen der vorangegangenen Hardwarearchitekturen kommt hier ein Busnetzwerk mit dynamischer priorisierter Arbitrierung zum Einsatz. Es wurde davon ausgegangen, dass auch bei dieser Architektur

(bei bis zu 16 Verarbeitungseinheiten) das Busnetzwerk (BDPA) optimale Ergebnisse liefert, da die Architektur zu großen Teilen mit der Architektur aus Abschnitt 6.6 übereinstimmt. Zusätzlich lag dieses Mal der Fokus stärker auf der Testapplikation (Verkehrssimulation) und deren optimalen Umsetzung. Die Testapplikation verwendet innerhalb der Zellregel Zufallszahlen, die einen detaillierten Vergleich einschränken. Als Vergleich wurde Testweise das RNFF für 16 Verarbeitungseinheiten ausgewertet. Damit ist zwar die Simulation geringfügig schneller, allerdings ist die Testapplikation für eine detaillierte Auswertung ungeeignet, da hier Zufallszahlen zum Einsatz kommen und dadurch bei jeder Ausführung andere Zugriffsstrukturen entstehen und zusätzlich ist der Unterschied zu gering, um fundierte Aussagen treffen zu können. Der Ressourcenbedarf für zwei Blöcke ist in Tabelle 6.17 und der Ressourcenbedarf für vier Blöcke in Tabelle 6.18 aufgestellt. Somit kann der Speicherbedarf der Zellen optimal an die Anwendung angepasst werden. Auf Grund der Verdopplung der Anzahl an Blöcken reduziert sich die Taktfrequenz leicht. Die zusätzlichen Blöcke müssen für alle Verarbeitungseinheiten über das Netzwerk lesbar sein, wodurch sich die Anzahl der Adressleitungen erhöht und das Netzwerk komplexer wird. Die Architekturen mit zwei und vier Blöcken werden in Abschnitt 6.7.5 für die Verkehrssimulation verwendet.

p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
		LEs	Register			
1	3.390	-	-	195.296	1.932	140,00
2	6.135	50	2	226.720	3.357	127,78
4	11.712	116	3	289.568	6.211	107,15
8	22.898	157	4	415.264	11.921	92,86
16	45.184	354	5	666.656	23.351	75,00

Tabelle 6.17: Realisierungsaufwand der multiarmigen speicherkonfigurierbaren universellen GCA-Architektur mit Busnetzwerk mit dynamischer Arbitrierung (BDPA) mit **2** Blöcken ohne Gleitkommaeinheit auf einem Cyclone II FPGA

p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
		LEs	Register			
1	3.421	-	-	326.368	1.932	131,25
2	6.251	83	2	357.792	3.357	120,85
4	12.005	198	3	420.640	6.211	103,34
8	23.473	415	4	546.336	11.921	88,89
16	46.605	892	5	797.728	23.353	73,08

Tabelle 6.18: Realisierungsaufwand der multiarmigen speicherkonfigurierbaren universellen GCA-Architektur mit Busnetzwerk mit dynamischer Arbitrierung (BDPA) mit **4** Blöcken ohne Gleitkommaeinheit auf einem Cyclone II FPGA

6.7.4.2 Realisierung mit Gleitkommaeinheit

Die Auswertung mit Gleitkommaeinheit erfolgte auf einem Stratix II FPGA. Da diese Architekturen für die Kraftberechnung des Mehrkörperproblems (Abschnitt 6.7.6) verwendet werden, sind insgesamt sieben Blöcke zu je 32 Bit notwendig. Unter anderem verringert sich aus diesem Grund die Taktfrequenz bereits bei einem Prozessor erheblich (Tabelle 6.19). Die Adressierung wird bei einer Erhöhung der Blockanzahl komplexer, denn alle Blöcke müssen über das Netzwerk gelesen werden können.
Ein anderer Aspekt ist die Integration der Gleitkommaeinheit. Für die Verwendung aller Gleitkommaoperationen sind zusätzliche Befehle notwendig. Die Anbindung und dafür notwendige Arbitrierung erhöhen die Systemkomplexität zusätzlich.
Damit bei dem erhöhten Speicheraufwand auch Architekturen mit mehreren Prozessoren realisiert werden können, ist die Zellanzahl auf 256 Zellen begrenzt.

Netzwerk	p	ALUTs	ALMs	Netzwerk ALUTs	Netzwerk Register	Speicherbits	Register	max. Takt [MHz]
-	1	3.076	2.312	-	-	183.740	2.774	92,00
ONR	2	4.751	3.646	96	0	215.164	3.937	87,00
ONR	4	7.809	6.021	388	180	278.012	6.381	79,00
ONR	8	13.985	11.011	1.036	360	403.708	11.097	71,00
ONUR	2	4.732	3.618	61	0	215.164	3.870	87,00
ONUR	4	7.735	6.051	291	180	278.012	6.252	79,00
ONUR	8	13.888	10.999	812	360	403.708	10.836	71,00
RN	2	4.709	3.653	193	120	215.164	3.988	87,00
RN	4	7.567	5.902	425	240	278.012	6.307	86,00
RN	8	13.371	10.823	895	480	403.708	10.952	75,00
BDPA	2	4.721	3.615	179	2	215.164	3.870	87,00
BDPA	4	7.461	5.897	311	3	278.012	6.074	83,00
BDPA	8	13.109	10.618	676	5	403.708	10.476	72,00

Tabelle 6.19: Realisierungsaufwand der multiarmigen speicherkonfigurierbaren universellen GCA-Architektur auf einem Stratix II FPGA mit **7** Blöcken für die Netzwerke: Omeganetzwerk mit Registerstufe (ONR), Omeganetzwerk mit unsynchronisierten Lesezugriffen (ONUR), Ringnetzwerk (RN), Busnetzwerk mit dynamischer Arbitrierung (BDPA).

6.7.5 Verkehrssimulation anhand des Nagel-Schreckenberg-Modells

In Abschnitt 4.2.1 wurde das Nagel-Schreckenberg-Modell für die Verkehrssimulation vorgestellt. Es werden nun zwei verschiedene Möglichkeiten der Realisierung dieses Modells auf dem GCA vorgestellt. Dazu wird das ursprünglich für den CA entworfene Modell als Agentensystem aufgefasst. Die Agenten stellen dabei die Fahrzeuge dar, die Anhand der Regel aus [NS92], wie in Abschnitt 4.2.1 aufgeführt, bewegt werden. Die verschiedenen Agentenwelten wurden mit dem AgentSim Simulator (Abschnitt 9.2) in Software und Hardware (GCA-Architekturen aus

Kapitel 6) simuliert. Im Folgenden ist unter dem Begriff Fahrzeug ein bestimmter Agententyp zu verstehen, der über die Anwendung definiert wird.

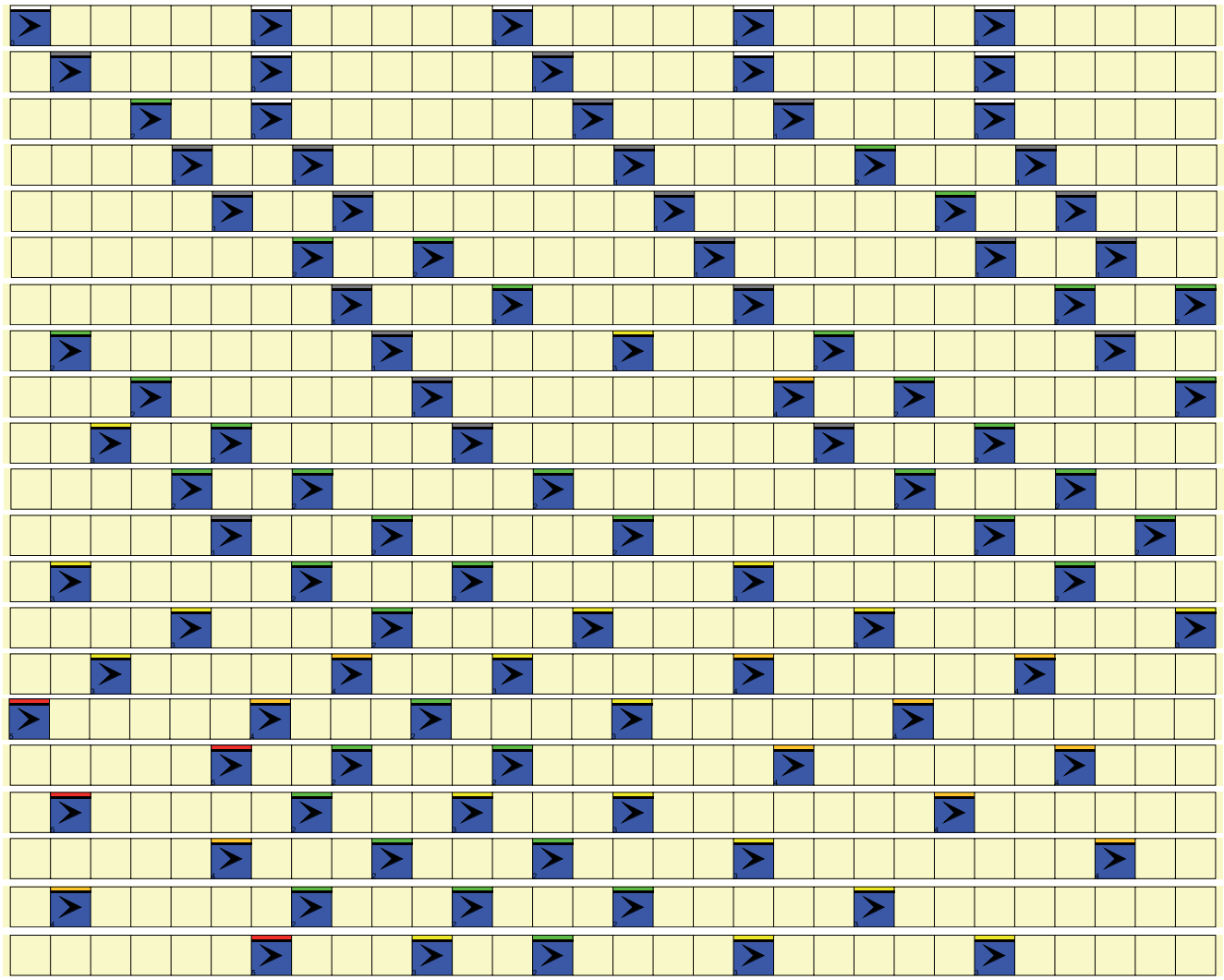


Abbildung 6.24: Nagel-Schreckenberg-Simulation für 20 Generationen (Zeilen von oben nach unten) mit $X = 30,\ Y = 1,\ \pi = 0,6$. Fahrzeuge (blau, ■), freie Straßenabschnitte (hellgelb, ■)

Die Abbildung 6.24 zeigt beispielhaft die Simulation von fünf gleichmäßig verteilten Fahrzeugen auf einer Fahrspur bestehend aus 30 Zellen. Die Fahrspur ist als Kreis realisiert und die Fahrzeuge bewegen sich von links nach rechts. Die maximale Geschwindigkeit v_{max} beträgt 5 Zellen $v_{max} = 5$ bei einer Trödelwahrscheinlichkeit π von $\pi = 0,6$. Die Agentenwelt wurde für 20 Generationen simuliert, was den einzelnen Zeilen der Abbildung entspricht. Die Abbildung zeigt somit das typische Zeitdiagramm, das bei der Verkehrssimulation anhand des Nagel-Schreckenberg-Modells entsteht. Die Fahrzeuge in dieser Simulation enthalten einen farbigen Geschwindigkeitsstreifen (Tabelle 6.20), der die aktuelle Geschwindigkeit des Fahrzeugs in Zellfeldern angibt.

Es werden zwei Implementierungen des Nagel-Schreckenberg-Modells betrachtet. Die erste Implementierung (Abschnitt 6.7.5.1) setzt einen klassischen Zellularen Automaten voraus. Hierbei besitzt jede Zelle lediglich Lesezugriff auf die direkten Zellnachbarn. Eine Fahrzeugbewegung

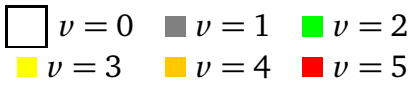


Tabelle 6.20: Bedeutung der Farben des Geschwindigkeitsstreifens in der Nagel-Schreckenberg-Simulation

von mehr als einer Zelle pro Generation ist nicht direkt möglich. Das Verhalten kann aber durch das Einführen von Untergenerationen simuliert werden. Es werden dann pro Generation v_{max} Untergenerationen ausgeführt und in jeder Untergeneration die Fahrzeuge mit entsprechend hoher Geschwindigkeit um eine Zelle bewegt. Da hier die GCA-Architektur aus Abschnitt 6.7 zum Einsatz kommt, wurde diese Methode in abgewandelter Form realisiert. Anstatt Untergenerationen zu verwenden, werden die Zellen in Bewegungsrichtung des Fahrzeugs abgesucht. Erst nach Beendigung des Suchvorganges werden die Fahrzeuge auf die Zielzelle bewegt.

Die zweite Implementierung (Abschnitt 6.7.5.2) setzt einen globalen Zellularen Automaten voraus. Nun besitzt jede Zelle Lesezugriff auf jede beliebige andere Zelle. Damit ist es möglich, die Fahrzeuge untereinander zu verketten. Jedes Fahrzeug besitzt nun die Kenntnis über seinen direkten Vordermann und damit die Distanz. Bei dieser Vorgehensweise entfallen viele externe Lesezugriffe, da der Abstand Δ und die Distanz α nicht mehr bestimmt werden müssen. Die Aktualisierung der Verkettung, die auf Grund der Bewegungen der Fahrzeuge notwendig ist, kann über wenige einfache Berechnungen erfolgen.

Zur genauen Bestimmung der Zellen wird bei der Beschreibung der beiden folgenden Abschnitte die Abkürzung C_i für eine Zelle mit dem Zellindex i verwendet. Der Abstand zwischen zwei Fahrzeugen wird mit Δ, die Distanz zwischen einer leeren Zelle und einem Fahrzeug mit α bezeichnet. Als Abkürzung wird eine leere Zelle auch mit E und eine Zelle mit einem Fahrzeug (entspricht einem Agenten) mit A bezeichnet.

6.7.5.1 Modellierung mit Suchen

Die Zellen im CA-Modell haben eine bestimmte, definierte Nachbarschaft. Bei dieser Anwendung beträgt die Nachbarschaft v_{max} Zellen nach rechts (Fahrzeuge bewegen sich nur von links nach rechts) für eine Fahrzeugzelle. Für eine leere Zelle beträgt die Nachbarschaft v_{max} Zellen nach links und zwei Zellen nach rechts.

Das Modell mit Suchen benötigt zwei Blöcke pro Zelle $C_i = (r, v)$, wobei v die aktuelle Geschwindigkeit des Fahrzeuges und r eine Zufallszahl bezeichnet. Eine leere Zelle ist durch eine Geschwindigkeit v größer als die maximale Geschwindigkeit v_{max} definiert $(v \geq v_{max})$. Eine Fahrzeugzelle ist durch eine Geschwindigkeit von $0 \leq v \leq v_{max}$ definiert.

Zuerst erhöht ein Fahrzeug seine Geschwindigkeit $v_1 = max\,(v + 1, v_{max})$, sofern die maximale Geschwindigkeit nicht überschritten wird. Nun wird die Größe des Abstands Δ zum vorausfahrenden Fahrzeug bestimmt. Dazu müssen die v_{max} Zellen in Fahrtrichtung abgesucht werden. Als Optimierung kann die Bestimmung des Abstands nach v_1 Zellen abgebrochen werden, da das Fahrzeug keine größere Geschwindigkeit aufweist. Dafür werden nacheinander die Zel-

len im Abstand $1,\ldots,v_1$ abgesucht. Die Geschwindigkeit v_1 muss kleiner als der Abstand Δ sein $\left(v_2 = min\left(v_1, \Delta - 1\right)\right)$, wenn auf den abgesuchten Zellen ein weiteres Fahrzeug gefunden wurde. Ansonsten wird die Geschwindigkeit v_1 nicht verändert. Mit der Wahrscheinlichkeit π wird die Geschwindigkeit um eins verzögert und es ergibt sich eine neue Geschwindigkeit v_3 für das Fahrzeug $v_3 = min(v_2 - 1, 0)$. Wenn $v_3 = 0$ ist, dann bewegt sich das Fahrzeug nicht und der Zellzustand bleibt erhalten. In allen anderen Fällen wird die Zelle leer. Es ist nicht möglich, dass sich in der gleichen Generation ein neues Fahrzeug auf diese Zelle bewegt. Die Reduktion der Geschwindigkeit über die Wahrscheinlichkeit π wird in der Hardwareimplementierung über eine 16 Bit Zufallszahl r, die gegen eine gesetzte Grenze Γ geprüft wird, umgesetzt. Für eine Wahrscheinlichkeit von 50% wird die Grenze Γ somit auf $\Gamma = \frac{2^{16}}{2}$ gesetzt. Die Reduktion ergibt sich dann durch $v_3 = min\left(v_2 - R, 0\right)$ mit:

$$R = \begin{cases} 0, & r < \Gamma \\ 1, & r \geq \Gamma \end{cases}$$

Die Berechnung der Agentenzelle in Pseudocode ist im Quellcode 6.4 dargestellt.

```
1  v' := min(v_max, v + 1);                                  //Beschleunigen
2
3  Δ := 1; repeat Δ++ until Δ = v'                           //Abstand bestimmen
4   if (C_{i+Δ} = A) then v' := Δ - 1
5   break
6
7  v' := v' - R                                                //Trödeln
8
9  if (v' > 0) then                                          //Neuer Zellwert
10  C_i := E
11 else
12  C_i := A
```

Quellcode 6.4: Pseudocode für die Berechnungen einer Agentenzelle

Die Zellregel für eine leere Zelle C_i ist komplexer. Zuerst werden die Zellen nach links (entgegen der Bewegungsrichtung der Fahrzeuge) durchsucht, bis die Distanz α die maximale Geschwindigkeit v_{max} übersteigt oder ein Fahrzeug an der Position $i - \alpha$ gefunden wurde. Anschließend wird die Bewegung des Fahrzeuges nachvollzogen, um zu bestimmen, ob sich das Fahrzeug auf diese leere Zelle bewegt oder nicht. Bewegt sich das Fahrzeug auf diese Zelle, so ist der aktuelle Zustand der Fahrzeugzelle zu kopieren und gleichzeitig in den Folgezustand zu überführen. Als Optimierung kann das Absuchen der Zellen von der Fahrzeugzelle bis zur leeren Zelle (in Fahrtrichtung) ausbleiben, da diese bei der Suche (entgegen der Fahrtrichtung) von der leeren Zelle zur Fahrzeugzelle schon geprüft wurden. Nur wenn der Abstand Δ die Größe $\alpha + 1$ oder $\alpha + 2$ hat, könnte das Fahrzeug überhaupt die leere Zelle als Zielzelle bestimmen. Aus diesem Grund ist es ausreichend, nur die zwei Zellen rechts von der leeren Zelle C_i zu prüfen. Das Fahrzeug wird nur auf diese Zelle kopiert, wenn die Geschwindigkeit v_3 äquivalent ist mit der Distanz α. Die optimierten Berechnungen für eine leere Zelle sind im Quellcode 6.5 gegeben.

```
1  α := 1; repeat α++ until (C_{i−α} = A) or (α = v_max);
2   if (C_{i−α} = E) then continue loop                    //Suchen einer Agentenzelle
3
4  v' := min(v_max, v+1);
5  if (v' < α) then      //Überprüfung ob die Geschwindigkeit größer der Distanz ist
6   return
7  if ((v' > α) and (C_{i+1} = A)) or (v' = α) then      //Kollisionsbehandlung und Trödeln
8   v' := α − R
9  else if ((v' = α+1) and (C_{i+1} = E)) or ((v' > α+1) and (C_{i+1} = E) and (C_{i+2} = A)) then
10  v' := α + 1 − R
11
12 if (v' = α) then                                        //Neuer Zellzustand
13  C_i := A
14 else
15  C_i := E
```

Quellcode 6.5: Pseudocode für die Berechnungen einer leere Zelle

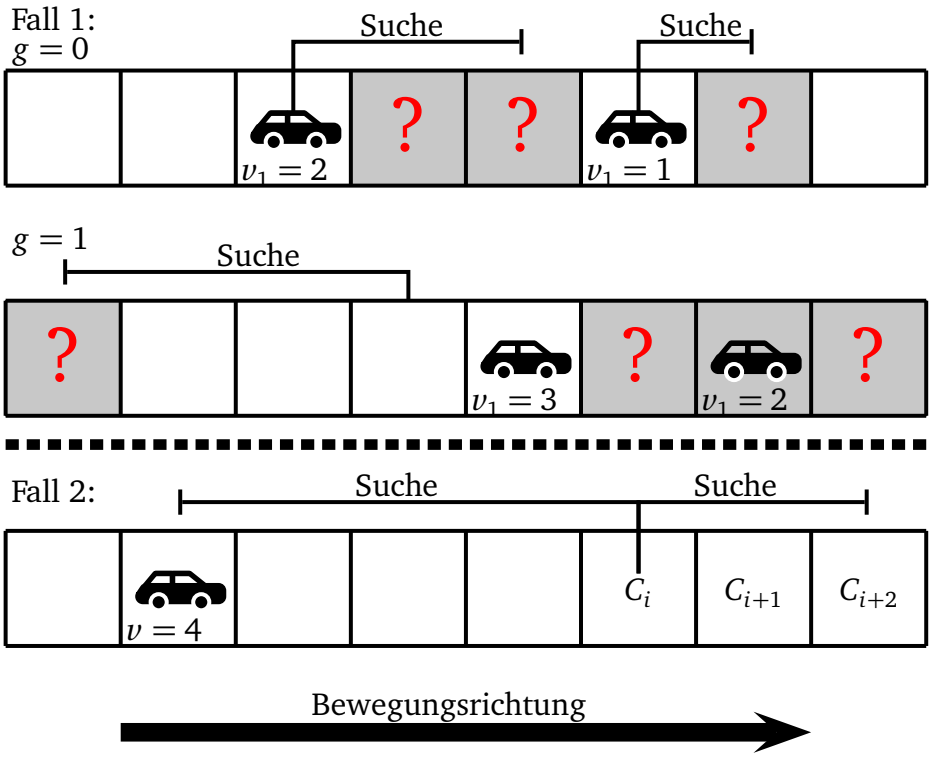


Abbildung 6.25: Nagel-Schreckenberg-Simulation mit Suchen (CA). Fall 1: Suchprozess der Fahrzeuge mit Transition einer Generation $(g = 0)$ in die Folgegeneration $(g = 1)$ Fall 2: Suchprozess einer leeren Zelle.

In Abbildung 6.25 werden zwei Fälle dargestellt. Im ersten Fall wird die Bewegung von zwei Fahrzeugen zu den Generationen $g = 0$ und $g = 1$ dargestellt. Die grauen Zellen stellen Zellen dar, die geprüft werden müssen. Das linke Fahrzeug hat eine Geschwindigkeit von zwei $v_1 = 2$, das rechte Fahrzeug hat eine Geschwindigkeit von eins $v_1 = 1$. Angegeben sind die Geschwindigkeiten nach der Beschleunigung, denn diese sind relevant für die Bestimmung des Abstandes Δ. In der Generation $g = 1$ hat das linke Fahrzeug die Geschwindigkeit $v_1 = 3$, doch die Suche wird nach zwei Zellen abgebrochen, da auf der zweiten Zelle in Fahrtrichtung ein Fahrzeug gefunden wurde. Die Anzahl der zu prüfenden Zellen variiert zwischen null und v_{max}.

Im zweiten Fall wird die Funktionsweise einer leeren Zelle gezeigt. Die aktuelle Geschwindigkeit v ist vier und wird auf fünf erhöht. Die leere Zelle prüft alle Zellen nach links, bis die Distanz α den Wert von v_{max} erreicht hat oder bis ein Fahrzeug gefunden wurde. In diesem Fall wird ein Fahrzeug gefunden. Die Zielzelle des Fahrzeugs ist entweder C_{i+1} oder C_i, abhängig davon, ob die Geschwindigkeit durch das Trödeln reduziert wird oder nicht. Nur wenn die Zielzelle C_i ist, wird das Fahrzeug kopiert.
In einem anderen Fall könnte sich das Fahrzeug z. B. auch auf die Zelle C_{i+2} bewegen wollen. Ist aber auf C_{i+2} ein anderes Fahrzeug vorhanden, kann als Zielzelle maximal die Zelle C_{i+1} in Betracht kommen. Eine weitere Reduktion durch Trödeln würde dann Zelle C_i als Zielzelle ergeben. Deshalb müssen die zwei Zellen nach rechts (C_{i+1}, C_{i+2}) überprüft werden, wenn die Geschwindigkeit v_3 größer ist als die Distanz α.

6.7.5.2 Modellierung mit verketteten Zellen

Die Zellen im GCA-Modell haben keine feste Nachbarschaft. Die Nachbarschaft ist dynamisch und wahlfrei und kann zu einer Optimierung der Verkehrssimulation verwendet werden. Diese Eigenschaft wird dazu verwendet, um die Fahrzeuge untereinander zu verketten.

Das Modell mit verketteten Zellen benötigt vier Blöcke pro Zelle $C_i = (r, v, L, z)$. Eine Fahrzeugzelle ist für eine Geschwindigkeit $v < v_{max}$ definiert. Für Geschwindigkeiten größer als v_{max} ist eine Zelle als leer definiert. Der Block z speichert einen relativen Wert, der die zukünftige Geschwindigkeit des Fahrzeugs angibt. In der Generation g wird dann die Geschwindigkeit z als neue aktuelle Geschwindigkeit v übernommen. Gleichzeitig wird die Geschwindigkeit für die Generation $g + 2$ berechnet und in z gespeichert. Über den Zeiger L sind die Fahrzeuge in Bewegungsrichtung miteinander verkettet. Leere Zellen haben über den Zeiger L Zugriff auf das nächste Fahrzeug, das in Fahrtrichtung die leere Zelle passieren wird. Der Wert des Zeigers L ist dabei eine absolute Adresse einer Zelle. Dabei bezeichnet C_L die Zelle, die durch den Zeiger L adressiert wird. Die Zufallszahl wird, wie in Abschnitt 6.7.5.1 auf Seite 118 beschrieben, in dem Block r gespeichert.

Die Verkettung der Fahrzeuge ist über den Zeiger L realisiert. Für Fahrzeugzellen zeigen alle Zeiger in Bewegungsrichtung auf das nächste Fahrzeug. Die Zeiger aller leeren Zellen zeigen entgegen der Bewegungsrichtung auf das nächste Fahrzeug. Dabei ist es unerheblich, ob sich das Fahrzeug tatsächlich auf diese leere Zelle bewegen, vor dieser Zelle bleiben oder sich über die Zelle hinweg bewegen wird. Es wird angenommen, dass es keine Begrenzung am Ende der Fahrbahn gibt und diese zyklisch mit dem Anfang verbunden ist. Somit ergibt sich über die Zeiger eine Kette in der Form eines Kreises.
Der Block z enthält für Fahrzeugzellen die zukünftige Geschwindigkeit, die das Fahrzeug in der nächsten Generation erhalten wird. Für leere Zellen wird dieser Block nicht verwendet. Mit der Position eines Fahrzeuges, dem Zeiger L und dem Wert von z kann die nächste leere Zelle bestimmt werden, auf der sich das Fahrzeug in der nächsten Generation befinden wird. Somit kann ohne ein Absuchen der Zellen zwischen zwei Fahrzeugen direkt die nächste freie Zelle bestimmt werden. Für die Verkettung der Fahrzeuge und der initialen Bestimmung der zukünftigen Geschwindigkeit z wird eine zusätzliche Generation benötigt. Die initiale Generation wird als gegeben betrachtet.

Die Berechnung der Geschwindigkeit v, des Zeigers L und der zukünftigen Geschwindigkeit z ist durch die folgenden Regeln 1 bis 15 gegeben. Zur Vereinfachung und besseren Lesbarkeit wird der Zeiger L der Zelle C_i mit $L(i)$ bezeichnet. Zusätzlich zur Definition des Typs einer Zelle über die Geschwindigkeit wird vereinfacht auch $C_i = A$ geschrieben, wenn es sich um eine Agentenzelle (in diesem Fall ein Fahrzeug) handelt. Für den Fall, dass sich ein Fahrzeug bewegt, werden für die Fahrzeugzelle die Regeln 5, 6, 10 ($A \to E$) und für eine leere Zelle die Regeln 4, 9, 13 ($E \to A$) angewendet.
Wenn sich ein Fahrzeug von der Position i auf die Position k bewegt (von einer Fahrzeugzelle C_i auf eine leere Zelle C_k), wird $C_i := E$ und $C_k := A$. Die Bedingung ($E \to A$) für die Zelle k ist $L(k) + z(L(k)) = k$. Dies bedeutet, dass die leere Zelle E mit dem Zeiger L auf ein Fahrzeug A zeigt, welches mit dem Offset der zukünftigen Geschwindigkeit z auf die leere Zelle E zeigt. Die entsprechende Bedingung für die Fahrzeugzelle ist $L(i + z(i)) = i$. Über den Offset der Geschwindigkeit z wird die leere Zelle (Zielzelle) ermittelt. Der Zeiger L dieser leeren Zelle zeigt wiederum zurück auf das Fahrzeug. Für die Bedingung als solche, ist es allerdings ausreichend, wenn $z \neq 0$ vorliegt. Damit bewegt sich das Fahrzeug auf eine andere Zelle. Für die freiwerdende Zelle ist das Ziel dabei unerheblich.

Ein Fahrzeug bewegt sich nicht ($A \to A$), wenn die zukünftige Geschwindigkeit $z = 0$ ist (Regeln 7, 11, 15). Eine leere Zelle bleibt leer, wenn das Fahrzeug diese Zelle nicht erreichen wird (Regeln 3, 8) oder diese überfährt (Regeln 1, 2, 8).

Im GCA-Modell sind einfache Zugriffe auf Nachbarzellen vorgesehen (z. B. $L(i)$). Nicht standardmäßig vorgesehen sind doppelte indirekte Zugriffe (z. B. $z(L(L(i)))$). Diese Erweiterung kann dem GCA-Modell hinzugefügt werden und auch von einer Hardwarearchitektur unterstützt werden.

Die Zellregel bzw. das „Verhalten“ der Fahrzeuge wird durch die folgenden drei Formeln definiert. Mit diesen drei Formeln wird der Zeiger L, die Geschwindigkeit v und die zukünftige Geschwindigkeit z berechnet.

$$L^+(i) := \begin{cases} L(L(i)-1) + z(L(L(i)-1)), & (E \to E) \wedge \text{auslassen } (C_i) \wedge \left(C_{L(i)-1} = E\right) \quad (1) \\ L(i) - 1, & (E \to E) \wedge \text{auslassen } (C_i) \wedge \left(C_{L(i)-1} = A\right) \quad (2) \\ L(i) + z(L(i)), & (E \to E) \wedge \text{nicht ausgelassen } (C_i) \quad (3) \\ L(L(i)) + z(L(L(i))), & (E \to A) \quad (4) \\ L(i-1) + z(L(i-1)), & (A \to E) \wedge \left(C_{i-1} = E\right) \quad (5) \\ i - 1, & (A \to E) \wedge \left(C_{i-1} = A\right) \quad (6) \\ L(i) + z(L(i)), & (A \to A) \quad (7) \end{cases}$$

$$v^+(i) := \begin{cases} E, & (E \to E) \quad (8) \\ z(L(i)), & (E \to A) \quad (9) \\ E, & (A \to E) \quad (10) \\ z(i), & (A \to A) \quad (11) \end{cases}$$

$$z^{+}(i) := \begin{cases} -, & (E \to E) \quad (12) \\ \max\left[\min\left[z\left(L\left(i\right)\right)+1, L\left(L\left(i\right)\right)+z\left(L\left(L\left(i\right)\right)\right)-i-1\right]-R, 0\right], & (E \to A) \quad (13) \\ -, & (A \to E) \quad (14) \\ \max\left[\min\left[1, L\left(i\right)+z\left(L\left(i\right)\right)-i-1\right]-R, 0\right], & (A \to A) \quad (15) \end{cases}$$

Die Bedingungen für die einzelnen Fälle sind wie folgt definiert:

$$\text{Bedingung } (E \to E) : L\left(i\right)+z\left(L\left(i\right)\right) \neq i$$

$$\text{Bedingung } (E \to A) : L\left(i\right)+z\left(L\left(i\right)\right) = i$$

$$\text{Bedingung } (A \to E) : z\left(i\right) \neq 0$$

$$\text{Bedingung } (A \to A) : z\left(i\right) = 0$$

$$auslassen\left(C_i\right) : L\left(i\right)+z\left(L\left(i\right)\right) > i$$

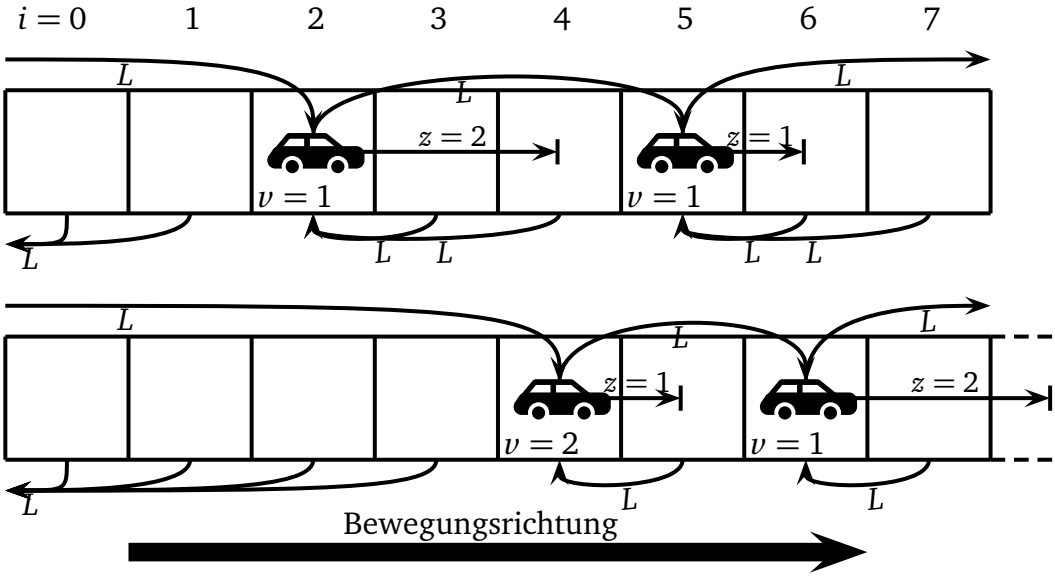


Abbildung 6.26: Nagel-Schreckenberg-Simulation mit verketteten Zellen (GCA). Transition einer Generation $(g = 0)$ in die Folgegeneration $(g = 1)$

In Abbildung 6.26 ist die Bewegung der Fahrzeuge und die Veränderung der Zeiger L, der Geschwindigkeit v und der zukünftigen Geschwindigkeit z dargestellt. In der Generation $g = 0$ hat der linke Agent eine Geschwindigkeit von $v = 1$ und eine zukünftige Geschwindigkeit von $z = 2$. Die zukünftige Geschwindigkeit z bestimmt die leere Zelle, auf die sich das Fahrzeug bewegen wird. Das Fahrzeug wird von der leeren Zelle kopiert $(g = 1)$, d. h. der Zustand den

das Fahrzeug derzeit hat $(g = 0)$, wird um die Bewegung verändert und ist dann in der Zielzelle gespeichert $(g = 1)$. Die neue Geschwindigkeit z wurde auf den Abstand der beiden Fahrzeuge reduziert.

Die leeren Zellen verwenden den Zeiger L dazu, um entgegengesetzt der Fahrtrichtung auf das nächste Fahrzeug zu zeigen. Die leere Zelle C_3 wird übersprungen, was durch die Anwendung der Regel eins berücksichtigt wird. Die leere Zelle C_7 wird nicht übersprungen und Regel drei kommt zur Anwendung.

Die Zellregel (Quellcode 6.6) beinhaltet die Behandlung der Fahrzeugzellen, der leeren Zellen und die Überprüfung von Sonderfällen (kein Fahrzeug vorhanden, nur ein Fahrzeug vorhanden sowie ungültige Zellzustände). Die Zellindizes, die über den vorhandenen Speicherbereich hinauslaufen, werden über die Funktion `recalcIndexInLine` in Zeile 1 auf gültige Indizes abgebildet. Somit wird das zkylische Schließen der Fahrspur realisiert. Die Berechnung für eine Agentenzelle findet in den Zeilen 60 bis 89, die für eine leere Zelle in den Zeilen 18 bis 58 statt. Zur Fehlerbehandlung werden alle ungültigen Zellzustände als Hindernis markiert (Zeile 91).

```
1  int recalcIndexInLine(int p, int pInLine)  //Zyklisches Schließen des Zellraums
2   {return (p%MAXX+(pInLine/MAXX)*MAXX);}
3
4  inline int abs(int a) {return (a>=0) ? a : -a;}                //Betragsfunktion
5
6  inline int min(a,b) {return (a<b) ? a : b;}                   //Minimumsfunktion
7
8  int main(){                                                  //Beginn der Zellregel
9   int g, i, speedtype, L, Z, tmp;
10
11  CI(NEXTGEN,0,0);                                          //Anfangssynchronisation
12  CI(NEXTGEN,0,0);
13
14  for(g=0;g<GENS;g++){
15   for(i=SC;i<LOCL_CELLS+SC;i++){
16    speedtype = CI(RD_B0,i,0);
17
18    if(speedtype==CELL_FREE)                                        //Leere Zelle
19    {
20     L = CI(RD_B1,i,0);
21     if(L!=i)                                  //Mindestens ein Fahrzeug pro Zeile
22     {
23      Z = CI(RD_B2,L,0);
24      if(recalcIndexInLine(L+Z,i)==i){ //Kopiere den Agenten, aktualisiere L,z,v
25       CI(WR_B0,i,Z);
26       tmp=recalcIndexInLine(
27           CI(RD_B1,L,0)+CI(RD_B2,CI(RD_B1,L,0),0),i);
28
29       CI(WR_B1,i,tmp);
30       if(i<tmp)
31        tmp=min(min(Z+1, (tmp-i)-1),MAXSPEED);
32       else
33        tmp=min(min(Z+1,(MAXX-abs(tmp-i))-1),MAXSPEED);
34
35       CI(WR_B2,i,(tmp>0 && CI(RD_B3,L,0)<P) ? tmp-1: tmp);
36      }
```

```
37      else{                                   //Zelle bleibt leer, aktualisiere L
38       CI(WR_B0,i,CELL_FREE);
39       tmp=L+Z;
40                                      //Prüfe ob leere Zellen ausgelassen werden
41       if( (i>L && i<tmp) || (i+MAXX>L && i+MAXX<tmp) ){
42        tmp=recalcIndexInLine(L-1+MAXX,i);
43        if(CI(RD_B0,tmp,0)==CELL_FREE){               //Vorgängerzelle ist leer
44         L = CI(RD_B1,tmp,0);
45         CI(WR_B1,i,recalcIndexInLine(L+CI(RD_B2,L,0),i));
46        }
47        else
48         CI(WR_B1,i,tmp);
49       }
50       else
51        CI(WR_B1,i,recalcIndexInLine(tmp,i));
52      }
53     }
54     else{                                          //komplette Zeile ist leer
55      CI(WR_B0,i,CELL_FREE);
56      CI(WR_B1,i,i);
57     }
58    }
59    else{
60     if(speedtype<=CELL_AGENT)                                 //Fahrzeugzelle
61     {
62      Z = CI(RD_B2,i,0);
63      if(Z>0){                                              //Fahrzeug bewegen
64       CI(WR_B0,i,CELL_FREE);
65
66       tmp=recalcIndexInLine(i-1+MAXX,i);
67
68       if(CI(RD_B0,tmp,0)==CELL_FREE){               //Vorgängerzelle ist leer
69        L = CI(RD_B1,tmp,0);              //Kopiere neuen Link L des Vorgängers
70        CI(WR_B1,i,recalcIndexInLine(L+CI(RD_B2,L,0),i));
71       }
72       else                 //Vorgänger ist ein Fahrzeug, setze L auf diese Zelle
73        CI(WR_B1,i,tmp);
74      }
75      else{                                          //Fahrzeug nicht bewegen
76       CI(WR_B0,i,Z);
77
78       L=CI(RD_B1,i,0);
79       tmp=recalcIndexInLine(L+CI(RD_B2,L,0),i);
80       CI(WR_B1,i,tmp);
81
82       if(i<tmp)
83        tmp=min(1, (tmp-i)-1);
84       else
85        tmp=min(1, (MAXX-abs(tmp-i))-1);
86
87       CI(WR_B2,i,(tmp>0 && CI(RD_B3,i,0)<P) ? tmp-1: tmp);
88      }
89     }
90     else                    //Undefinierter Zelltyp wird als Hindernis behandelt
91      CI(WR_B0,i,CELL_OBSTACLE);
```

```
92      }
93     }
94     CI(NEXTGEN,0,0);                                    //Generationssynchronisation
95    }
96  return 0;}
```

Quellcode 6.6: Zellregel der Nagel-Schreckenberg-Implementierung mit verketteten Zellen

6.7.5.3 Auswertung der Implementierungen

Für die Auswertung wurde ein Fahrstreifen bestehend aus 2048 Zellen verwendet. Dabei sind jedem Prozessor $\frac{2048}{p}$ Zellen zugeordnet. Um den Effekt, den eine unterschiedliche Anzahl an Fahrzeugen auf die Simulation hat, untersuchen zu können, wurden in der ersten Simulation 204 (entspricht 10% der vorhandenen Zellen) und in der zweiten Simulation 1024 (entspricht 50% der vorhandenen Zellen) Fahrzeuge verwendet. Die maximale Geschwindigkeit beträgt $v_{max} = 5$. Die Trödelwahrscheinlichkeit ist auf $\pi = 0,5$ gesetzt.

In Tabelle 6.21 sind die Ergebnisse für 10% und 50% Agenten dargestellt. Die Implementierung mit Suchen ist der Implementierung mit verketteten Zellen gegenübergestellt. Des weiteren kann die Skalierbarkeit der Architektur anhand der Verkehrssimulation betrachtet werden. Für 16 Prozessoren ergibt sich, unabhängig von der Implementierungsart, ein Speedup von 8,2. Die Ausführungszeiten unterscheiden sich für die beiden Varianten etwa um den Faktor 2. Die Implementierung mit verketteten Zellen erreicht einen Speedup von 8,7 während die Implementierung mit Suchen einen Speedup von 7,7 erreicht ($p = 16$). Der Unterschied der Ausführungszeiten nimmt bei einer höheren Verkehrsdichte ab. Dies ist darauf zurückzuführen, dass der Suchprozess bei einer höheren Verkehrsdichte häufiger früher abgebrochen werden kann. Dies führt insgesamt dazu, dass weniger Zellen und damit auch weniger Zugriffe stattfinden. Den Unterschied der beiden Varianten und somit die erreichbare Beschleunigung kann man somit bei einer geringen Verkehrsdichte am besten erzielen.

Eine Auswertung der möglichen Beschleunigung für 10% und für 50% Agenten ist in Abbildung 6.27 gegeben. Für 10% Agenten beträgt die Beschleunigung für 1, 2, 4, 8 und 16 Verarbeitungseinheiten etwa Faktor 2. Für 50% Agenten ist die mögliche Beschleunigung auf Grund der höheren Verkehrsdichte geringer und liegt abhängig von der Anzahl an Verarbeitungseinheiten zwischen 1,1 und 1,3. Die maximale Beschleunigung von 1,3 wurde mit 8 Verarbeitungseinheiten erreicht. Die Ergebnisse zeigen die Auswirkungen unterschiedlicher Anzahl von Agenten bei gleicher maximaler Geschwindigkeit.

In einer weiteren Auswertung wurde die Auswirkung unterschiedlicher maximaler Geschwindigkeiten untersucht. Dazu wurden 20 Agenten (entspricht 1% aller Zellen) mit den maximalen Geschwindigkeiten $v_{max} = \{5, 10, 20, 40, 80\}$ ausgewertet. Die Erhöhung der maximalen Geschwindigkeit scheint auf den ersten Blick nicht sinnvoll und nicht im Sinne des ursprünglichen Modells [NS92] zu sein. Höhere maximale Geschwindigkeiten können aber einerseits dazu verwendet werden, auch andere Verkehrsmittel, die schneller als PKWs sind, zu simulieren. Andererseits kann eine höhere maximale Geschwindigkeit dazu verwendet werden, die Auflösung der Zellen zu verfeinern. In [NS92] wurde die Länge einer Zelle mit 7,5 m festgelegt. Somit belegen

p	CA Algorithmus (10% Agenten)			
	Taktzyklen pro Generation	Taktzyklen-Speedup	Ausführungs-zeit pro Generation [ms]	wahrer Speedup
1	400.731	-	2,862	-
2	201.399	1,99	1,576	1,82
4	102.250	3,92	0,954	2,99
8	51.450	7,79	0,554	5,17
16	26.179	**15,31**	0,349	**8,20**
p	GCA Algorithmus (10% Agenten)			
	Taktzyklen pro Generation	Taktzyklen-Speedup	Ausführungs-zeit pro Generation [ms]	wahrer Speedup
1	187.791	-	1,431	-
2	95.203	1,97	0,788	1,82
4	49.077	3,83	0,475	3,01
8	24.980	7,52	0,281	5,09
16	12.729	**14,75**	0,174	**8,21**
p	CA Algorithmus (50% Agenten)			
	Taktzyklen pro Generation	Taktzyklen-Speedup	Ausführungs-zeit pro Generation [ms]	wahrer Speedup
1	237.911	-	1,699	-
2	121.983	1,95	0,955	1,78
4	62.271	3,82	0,581	2,92
8	34.502	6,90	0,372	4,57
16	16.591	**14,34**	0,221	**7,68**
p	GCA Algorithmus (50% Agenten)			
	Taktzyklen pro Generation	Taktzyklen-Speedup	Ausführungs-zeit pro Generation [ms]	wahrer Speedup
1	201.793	-	1,537	-
2	101.783	1,98	0,842	1,83
4	50.882	3,97	0,492	3,12
8	25.638	7,87	0,288	5,33
16	12.973	**15,55**	0,178	**8,66**

Tabelle 6.21: Gegenüberstellung der Verkehrssimulation des Nagel-Schreckenberg-Modells (Eine Fahrspur mit 10% und 50% Agenten): Modellierung mit Suchen (CA) und Modellierung mit verketteten Zellen (GCA). Auswertung auf der MPAB auf einem Cyclone II FPGA für das Busnetzwerk mit dynamischer Arbitrierung (BDPA).

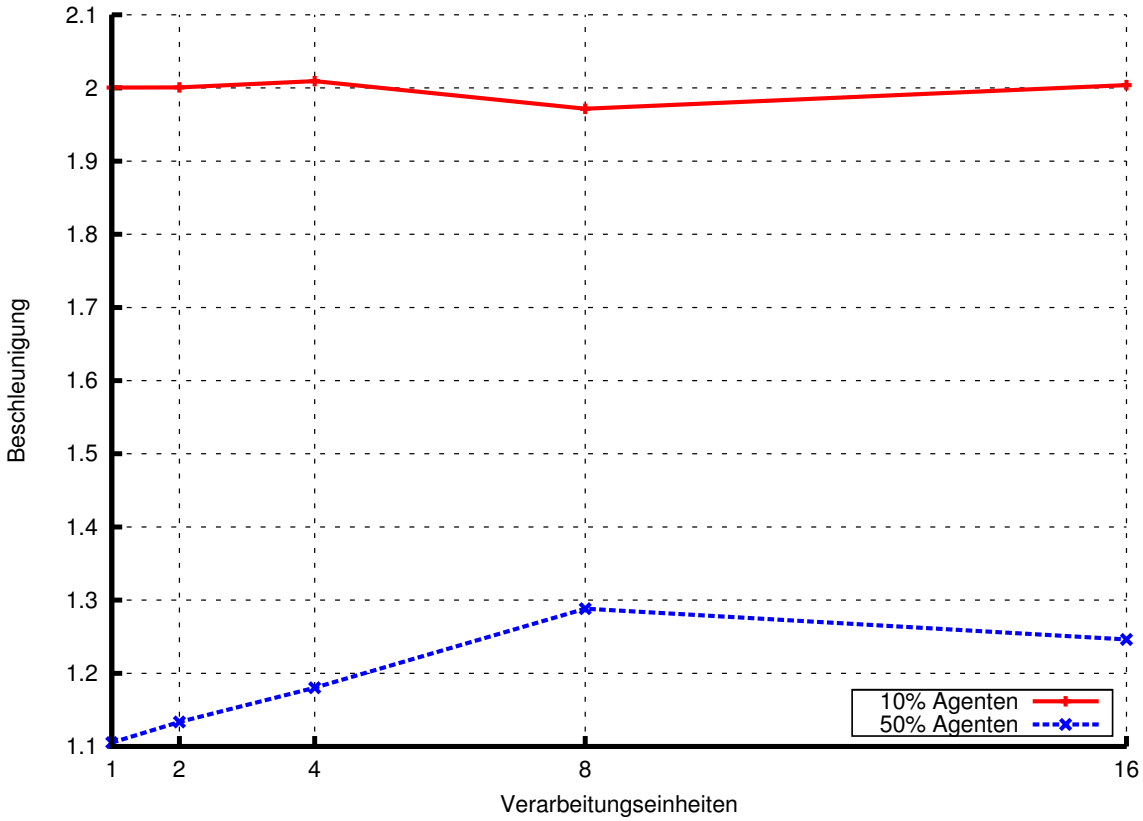


Abbildung 6.27: Beschleunigung des GCA-Algorithmus gegenüber dem CA-Algorithmus. $n = 2048$, $v_{max} = 5$, $\pi = 0,5$

auch alle Fahrzeuge, unabhängig von ihrer realen Länge diesen Platz. Mit einer feineren Auflösung können unter anderem unterschiedlich große Fahrzeuge berücksichtigt werden. Mit einer GCA-Implementierung (Modellierung mit verketteten Zellen) kann die Verkehrssimulation unter Verwendung der feineren Auflösung zudem ohne Mehraufwand durchgeführt werden. Die zusätzlichen leeren Zellen werden übersprungen und müssen von der Zellberechnung nicht ausgewertet werden. Die Auswertung für die maximalen Geschwindigkeiten $v_{max} = \{5, 10, 20, 40, 80\}$ ergab Beschleunigungen von 2,27; 4,12; 7,86; 14,90; 29,34;.

Die Ergebnisse zeigen, dass mit der Implementierung mit verketten Zellen (GCA-Modell) im Vergleich zur Implementierung mit Suchen (CA-Modell) enorme Beschleunigungen bei der Simulation erreicht werden können. Die erzielten Ergebnisse hängen zum einen von der Verkehrsdichte, zum anderen von der maximalen Geschwindigkeit ab. Der Unterschied ist am größten, wenn die Verkehrsdichte gering ist und die maximale Geschwindigkeit groß ist. Doch auch für die Werte des ursprünglichen Modells [NS92] mit einer maximalen Geschwindigkeit von $v_{max} = 5$ erreicht man abhängig von der Verkehrsdichte Beschleunigungen vom Faktor 1,1 bis etwa 2. Bei einer sehr großen Verkehrsdichte fallen die beiden Implementierungen zusammen, d. h., dass keine Beschleunigung mehr zu erreichen ist und Laufzeitunterschiede hauptsächlich auf die Implementierung zurückzuführen sind. Ob die direkte Nachbarzelle über einen Link selektiert oder durch einen Suchlauf gefunden wird, ist - von der Ausführungszeit gesehen - nahezu äquivalent.

6.7.5.4 Erweiterung für mehrspurige Modellierungen

Die Realisierung von verketteten Zellen ist nicht nur auf eine Fahrspur beschränkt. Mit zusätzlichen Verkettungen kann das Modell auch für mehrere Fahrspuren verwendet werden. Hierfür benötigt jede Zelle acht Blöcke $C_i = (r, v, L, z, BL, BR, FL, FR)$. Für die benachbarten Fahrspuren gibt es jeweils zwei zusätzliche Zeiger. Für die in Fahrtrichtung linke benachbarte Fahrspur gibt es zwei Zeiger BL, FL. Mit BL kann auf das nachfolgende Fahrzeug auf der linken Fahrspur zugegriffen werden. Somit kann ein Fahrzeug bestimmen, ob ein Fahrspurwechsel auf die linke Fahrspur möglich ist oder ob ein nachfolgendes Fahrzeug dabei behindert würde. Der Zeiger FL dient dazu, auf das vorausfahrende Fahrzeug auf der linken Fahrspur zuzugreifen. Somit kann ein Fahrzeug bestimmen, ob es sich lohnt, die Fahrspur zu wechseln. Mit beiden Informationen zusammen kann die Größe der Lücke zwischen zwei Fahrzeugen auf der linken Fahrspur bestimmt werden. Bei großer Verkehrsdichte kann ein Fahrzeug damit ermitteln, ob es möglich ist, die Fahrspur zu wechseln. Für die rechte Fahrspur stehen analog die Zeiger BR, FR zur Verfügung. Abbildung 6.28 zeigt die Verwendung der Zeiger. Die Zeiger L sind aus Gründen der Übersichtlichkeit nicht dargestellt.
Eine leere Zelle (in Abb. 6.28 Zelle C_i) verwendet die Zeiger L, BL, BR, um auf das nächste Fahrzeug auf jeder Fahrspur zuzugreifen. Von jeder der drei Fahrspuren kann sich in der nächsten Generation ein Fahrzeug auf die leere Zelle bewegen. Eine Kollision von zwei Fahrzeugen, die sich jeweils von der linken und der rechten Spur auf die gleiche leere Zelle bewegen möchten, kann über eine Kollisionserkennung ausgeschlossen werden. Von der leeren Zelle aus besteht über die Zeiger direkter Zugriff auf alle drei in Frage kommenden Fahrzeuge.

Für die Fahrspuren ganz links außen und ganz rechts außen gibt es keine benachbarte Fahrspur. Deshalb fallen hier die Zeiger nach links bzw. die Zeiger nach rechts weg. Um für alle

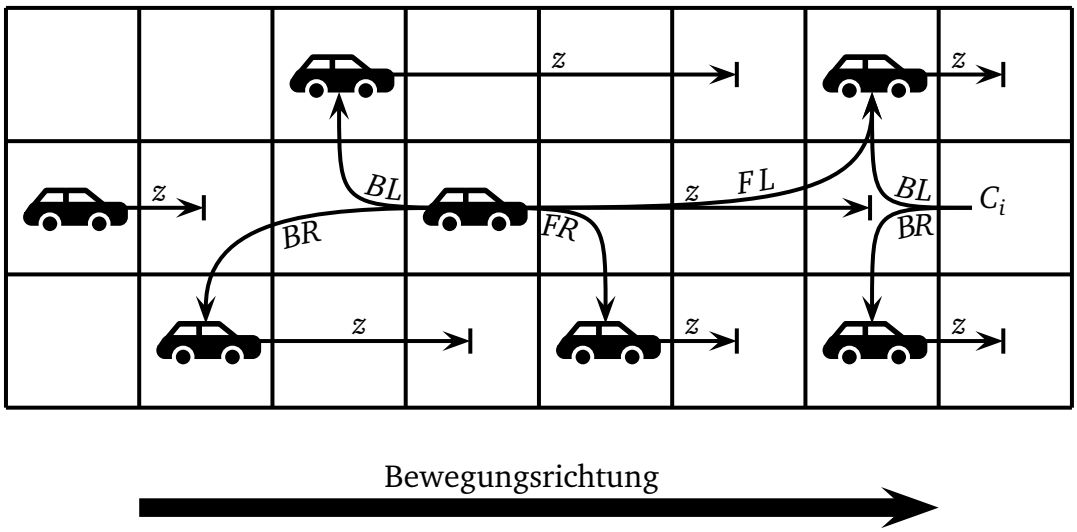


Abbildung 6.28: Mehrspurige Nagel-Schreckenberg-Simulation mit verketteten Zellen (GCA). Die Zeiger L sind nicht dargestellt.

Zellen die gleiche Zellregel verwenden zu können, ist es notwendig, dass die Zeiger an den Fahrbahnrändern keine Fahrspurwechsel ermöglichen. Dies kann z. B. dadurch realisiert werden, indem die Zeiger an den Rändern auf ein globales Fahrzeugobjekt zeigen und somit eine volle Fahrspur simulieren. Ein Fahrspurwechsel kann dann außgeschlossen werden und es ist kein Simulationsaufwand für die Randbetrachtungen notwendig.

6.7.6 Kraftberechnung des Mehrkörperproblems

Die Zellregel der Kraftberechnung des Mehrkörperproblems [JHL09] (Abschnitt 2.3.1) wurde in der *Global Cellular Automata Experimental Language* (GCA-L) beschrieben [JEH08a]. Die Zellregel wird von GCA-L (aufgeführt im Anhang Abschnitt 9.6 Quellcode 9.4) in C-Code (aufgeführt im Anhang Abschnitt 9.6 Quellcode 9.5) übersetzt, der danach auf der Hardwarearchitektur ausgeführt werden kann. Auf Grund dieser Übersetzung ist der generierte C-Code nicht so effizient wie direkt programmierter C-Code (aufgeführt im Anhang Abschnitt 9.6 Quellcode 9.6). GCA-L ist durch zusätzliche Konstrukte direkt auf das GCA-Modell abgestimmt und erlaubt eine sehr einfache und effiziente Beschreibung von Zellregeln. Ausgehend von der Zellregelbeschreibung in GCA-L können auch direkt Spezialarchitekturen erstellt werden.

Die Auswertung der Kraftberechnung für die aus GCA-L generierte Zellregel für die verschiedenen Netzwerke der Hardwarearchitektur sind in Tabelle 6.22 aufgeführt. Auf Grund des Designs der Architektur und der geringen Taktfrequenz sowie der Anwendung von Gleitkommazahlen ergibt sich eine im Vergleich zu den bisher vorgestellten Zellregeln geringe CUR. Die Kraftberechnung des Mehrkörperproblems basiert auf vielen Gleitkommaberechnungen und erfordert zudem eine größere Anzahl an externen Netzwerkzugriffen.

Für den optimierten C-Code (aufgeführt im Anhang Abschnitt 9.6 Quellcode 9.6) erhält man bei acht Verarbeitungseinheiten und unter der Verwendung des BDPA eine Ausführungszeit von

34,06 ms. Der Speedup beträgt 6,28. Die automatische Generierung des C-Codes aus GCA-L ist also noch nicht optimal, liefert aber schon gute Ergebnisse.
In jeder Zelle werden sieben 32 Bit Gleitkommazahlen gespeichert. Diese sieben Werte speichern die Masse $(Body_m)$, Position $\left(Body_x, Body_y, Body_z\right)$ und Beschleunigung $\left(Body_{ax}, Body_{ay}, Body_{az}\right)$ im Raum jedes Körpers.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
-	1	19.692.481	214,05	-	1,20
ONR‡	2	11.431.633	131,34	1,63	1,95
ONR‡	4	6.834.498	86,51	2,47	2,96
ONR‡	8	3.977.009	56,01	3,82	4,57
ONUR	2	9.353.387	107,46	1,99	2,38
ONUR	4	5.342.949	67,63	3,16	3,79
ONUR	8	3.030.281	42,68	5,02	6,00
RN	2	9.615.844	110,48	1,94	2,32
RN	4	5.322.251	61,88	3,46	4,14
RN	8	3.192.624	42,57	5,03	6,01
BDPA	2	9.307.076	106,93	2,00	2,39
BDPA	4	5.128.066	61,78	3,46	4,14
BDPA	8	2.912.912	40,46	5,29	6,33

Tabelle 6.22: Auswertung der Kraftberechnung des Mehrkörperproblems auf der MPAB auf einem Stratix II FPGA für die Netzwerke: Omeganetzwerk mit Registerstufe (ONR), Omeganetzwerk mit unsynchronisierten Zugriffen (ONUR), Ringnetzwerk (RN), Busnetzwerk mit dynamischer Arbitrierung (BDPA).
‡Verwendet nicht die `AUTO_READ` Funktion

Für die Kraftberechnung von 256 Körpern wurden die internen und externen Lesezugriffe sowie die Schreibzugriffe bestimmt. Für die Anzahl externer Lesezugriffe eines Prozessors gilt:

$$\frac{4n^2\left(p-1\right)}{p^2}$$

Die Anzahl der internen Lesezugriffe pro Prozessor bestimmt sich durch:

$$\frac{4n^2+10n^2p}{p^2}$$

Die Anzahl der Schreibzugriffe pro Prozessor kann über die folgende Formel ermittelt werden:

$$7\frac{n^2}{p}$$

Die Zahl vier gibt dabei die Anzahl der Blöcke an, die von einer externen Verarbeitungseinheit gelesen werden müssen $\left(Body_x, Body_y, Body_z, Body_m\right)$. Die Gesamtanzahl der Blöcke ist sieben und alle sieben Blöcke werden bei jeder Berechnung neu geschrieben. Die Verteilung der internen und externen Zugriffe sowie die Anzahl der Schreibzugriffe sind in Tabelle 6.23 aufgeführt.

p	Zugriffe		Schreibzugriffe
	extern	intern	
2	65.536	393.216	229.376
4	49.152	180.224	114.688
8	28.672	86.016	57.344

Tabelle 6.23: Verteilung der internen und externen Lesezugriffe bei der Kraftberechnung von 256 Körpern bezogen auf eine Verarbeitungseinheit sowie die Anzahl der Schreibzugriffe einer Verarbeitungseinheit.

6.7.7 Bewertung der Hardwarearchitektur

Mit der MPAB steht eine erweiterte Hardwarearchitektur zur Verfügung. Durch die Verwendung von Blöcken können pro Zelle beliebig viele Datenwerte gespeichert werden. Damit ist die Architektur für die unterschiedlichsten Anwendungen flexibel einsetzbar. Der Einsatz der Architektur wurde anhand der Verkehrssimulation auf der Basis des Nagel-Schreckenberg-Algorithmus aufgezeigt und evaluiert. Darüber hinaus wurde eine neue Zellregel für das GCA-Modell implementiert, die die Simulation der Anwendung erheblich beschleunigt. Zusätzlich ist es möglich, Gleitkommaoperationen in der Architektur hinzuzufügen. Da die Gleitkommaoperationen immer mehrere Taktzyklen für eine Berechnung benötigen, verwenden alle Verarbeitungseinheiten die gleiche Gleitkommaeinheit. Somit werden Ressourcen gespart und der zeitliche Verlust ist gering. Es ist für eine Leistungssteigerung durchaus möglich, jedem Prozessor eine eigene Gleitkommaeinheit zur Verfügung zu stellen. Ob die damit verbundene Leistungssteigerung den erhöhten Platzbedarf rechtfertigt, ist vom Einzelfall und von der Anwendung abhängig.

6.8 Spezialisierte GCA-Architektur mit Hashfunktionen (HA)

Mit der in Abschnitt 6.6 vorgestellten Architektur (MPA), die das GCA-Modell umsetzt, können einerseits klassische GCA-Anwendungen ausgeführt werden, aber auch Multi-Agenten-Anwendungen simuliert werden. Um die Simulation von Multi-Agenten-Anwendungen besser zu unterstützen und auch komplexere Verhalten der Agenten zu ermöglichen, wurde in Abschnitt 6.7 eine erweiterte Architektur (MPAB) vorgestellt.

Bei der Simulation von Multi-Agenten-Anwendungen zeigt sich, im Gegensatz zur Ausführung von klassischen GCA-Anwendungen, dass nicht immer alle Zelldaten für die Berechnung relevant sind. Ausgehend vom GCA-Modell werden in einer Generation alle Zellen berechnet. Für Multi-Agenten-Anwendungen ist die Berechnung aller Zellen jedoch nicht notwendig. Aus dieser Tatsache ergibt sich für den Entwurf neuer Hardwarearchitekturen eine Optimierungsmöglichkeit zur Beschleunigung der Simulation von Multi-Agenten-Anwendungen.

Um die Notwendigkeit einer Optimierung und den möglichen Geschwindigkeitsvorteil einer Simulation abschätzen zu können, wurde die Simulation der Agentenwelt aus Abschnitt 6.6.4.2 auf der Architektur aus Abschnitt 6.6 mit der Simulation einer leeren Welt (d. h. nur der Weltrand ist vorhanden) auf der gleichen Architektur ($p = 4$) verglichen. Die Simulation der Agentenwelt benötigt dabei 0,44 ms pro Generation. Für die Simulation der leeren Welt werden 0,37 ms pro Generation benötigt. Somit werden etwa 83 % der Ausführungszeit für die Simulation der leeren Zellen benötigt. Betrachtet man die Anzahl der Zellen der Agentenwelt, die durch Agenten belegt sind im Vergleich zu den leeren Zellen, kommt man auf eine Dichte von $d = 17,8\%$. Eine optimierte Architektur verspricht somit eine beschleunigte Simulation.

Für die Umsetzung der Optimierung werden die Zellen in zwei Gruppen eingeteilt [SHH10a, SHH11b]:

- **aktive Zellen:** Eine aktive Zelle beinhaltet für die Simulation (d. h. für die Ausführung der Zellregel) relevante Daten. Die Daten der Zelle werden in einem Datenspeicher vorgehalten und müssen bei der Simulation berücksichtigt werden. Im Kontext der Multi-Agenten-Simulation handelt es sich hierbei generell um Agentenzellen jeglicher Art (Agenten, Hindernisse, ...).
- **inaktive Zellen:** Eine inaktive Zelle beinhaltet keine für die Simulation relevanten Daten. Diese Zelle muss daher in der Simulation nicht berücksichtigt werden. Es ist auch nicht notwendig, Speicherplatz für diese Zelle vorzusehen. Eine inaktive Zelle ist deshalb in der Architektur gar nicht vorhanden. Im Kontext der Multi-Agenten-Simulation sind inaktive Zellen alle leeren Zellen.

Die Unterscheidung in aktive und inaktive Zellen erlaubt die Realisierung von neuartigen Architekturen. Da inaktive Zellen in der Architektur nicht vorkommen, muss ein Mechanismus vorgesehen werden, um feststellen zu können, ob eine Zelle vorhanden ist oder nicht und welchen Status sie besitzt. Zu diesem Zweck werden Hashfunktionen $f_h(\ldots)$ eingesetzt.

Wie die Agenten (bzw. die aktiven Zellen) auf die einzelnen Datenspeicher der Architektur verteilt werden, ist in Abbildung 6.29 dargestellt. Die Position jedes Agenten wird durch eine Hashfunktion auf den vorhandenen Speicherbereich umgerechnet. An dieser Stelle wird dann der Agent abgespeichert. Evtl. auftretende Kollisionen werden durch rehashing aufgelöst.

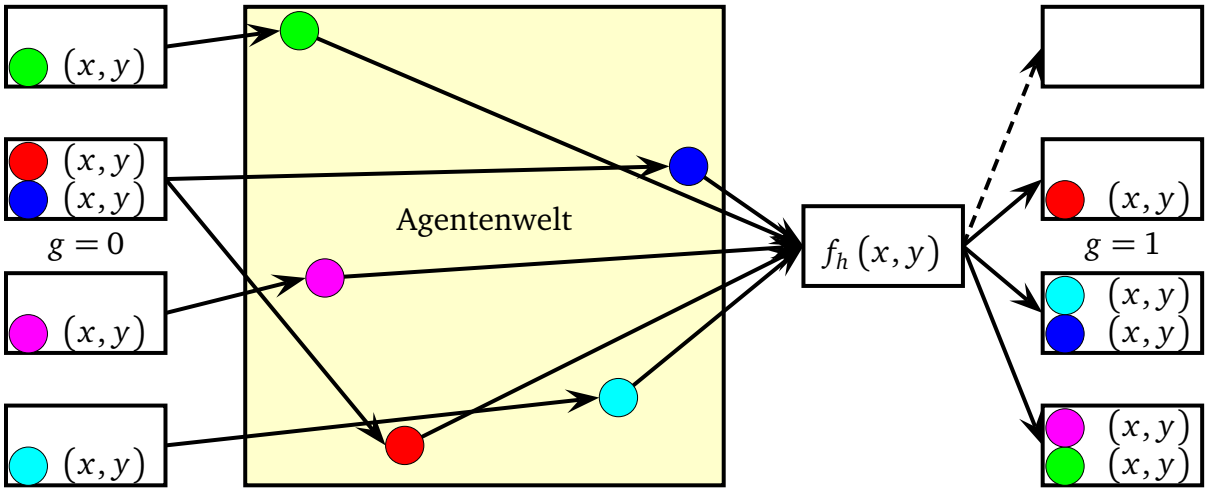


Abbildung 6.29: Verteilung der Agenten auf die Datenspeicher der HA

Der Ansatz aktiver und inaktiver Zellen wurde in [Sch98] in Bezug auf Zellulare Automaten untersucht. In [Sch98, Seite 12-15] wird der Begriff der *zellularen Aktivität* eingeführt. Dabei wird zwischen *eigenaktiven* und *fremdaktiven* Zellen unterschieden.

> *„Eine Zelle ist eigenaktiv, wenn sie eine Zustandsänderung erfahren hat. Eine Zelle wird als fremdaktiv bezeichnet, wenn sie selbst nicht eigenaktiv ist, aber sich in ihrer Nachbarschaft eine eigenaktive Zelle befindet. Beim Vorliegen von Eigen- oder Fremdaktivität wird die Zelle der Generationsberechnung zugeführt.“ [Sch98, Seite 13]*

Für die entwickelte Hardwarearchitektur ist eine derart feine Unterteilung nicht notwendig. Die Unterteilung in aktive und inaktive Zellen ist vollkommen ausreichend. In der implementierten Hardwarearchitektur werden nach der Definition von [Sch98] nur eigenaktive Zellen berechnet. Ein Wechsel findet immer zwischen zwei Zellen statt, die je nach Anforderung ihren Status „tauschen“.

Unter Berücksichtigung der genannten Aspekte ergibt sich eine neuartige Hardwarearchitektur, abgekürzt HA (engl.: Hash Architecture), die für $p = 4$ in Abbildung 6.30 dargestellt ist [SHH10a, SHH11b]. Hierbei besteht eine Verarbeitungseinheit (VE) aus einem Datenspeicher, einem Lese- und Schreibarbiter (R/W), einem Flagregister (Flag), einem Lesepuffer (RB), einem Prozessor (NIOS II), einer Registerstufe (r), einem Schreibpuffer (WB), der Hashfunktion $f_h(i)$, einer Ladefunktion (LADE) und einer Prüffunktion (PRÜFE). Der Verilogcode (Quellcode 9.11) dieser Architektur befindet sich im Anhang Kapitel 9.

Der Datenspeicher jeder VE kann dabei zwei Generationen speichern, wobei der Speicherplatz aller VE so groß sein muss, dass alle aktiven Zellen (Agenten, Hindernisse, ...) gespeichert werden können. Somit lässt sich der benötigte Speicherplatz auf die tatsächlich zu speichernden Agenten reduzieren. Der Ansatz der Blöcke aus Abschnitt 6.7 wird auch in dieser Architektur

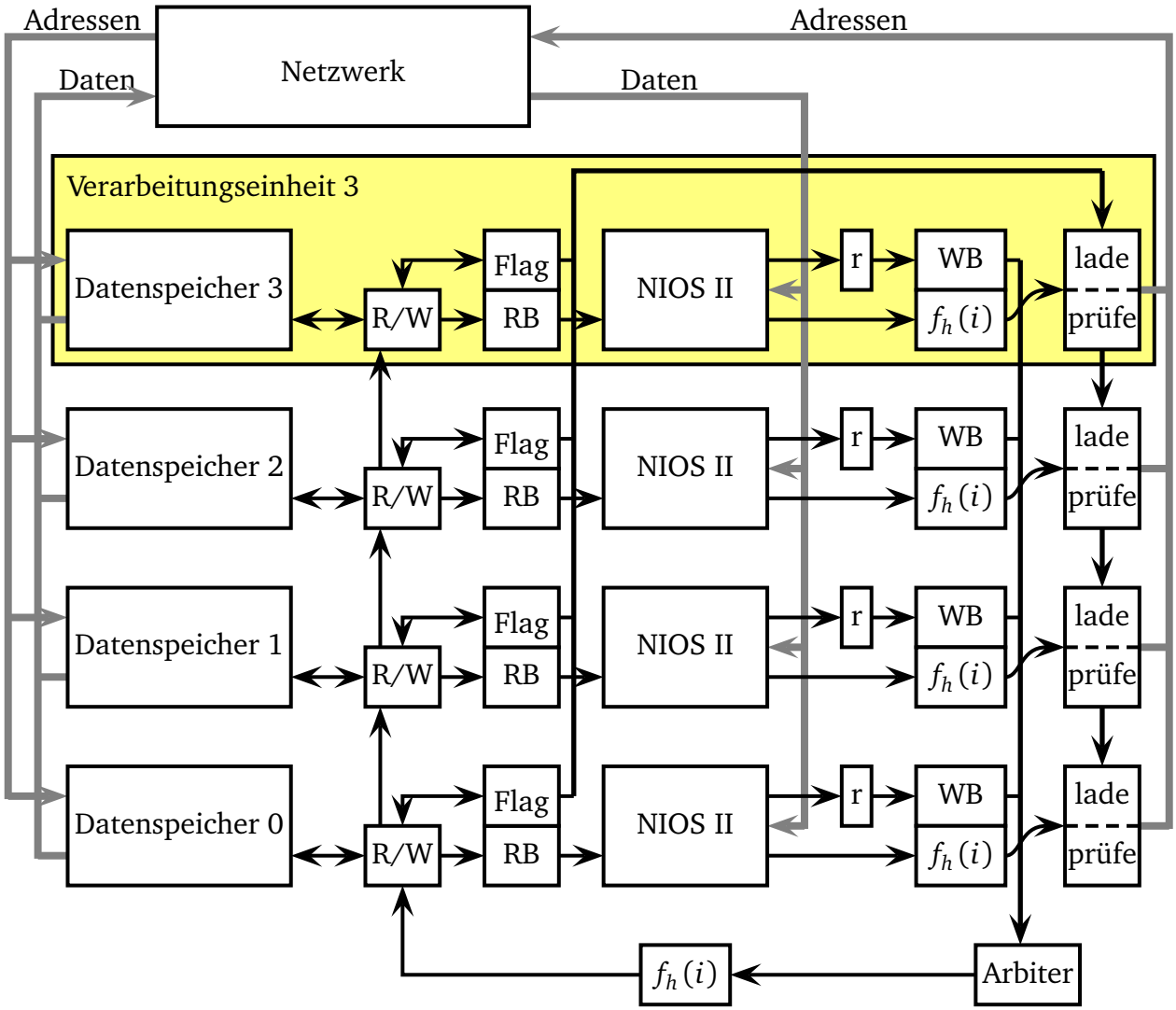


Abbildung 6.30: GCA-Architektur mit Hashfunktion (HA)

verwendet. Weil bei dieser Architektur nicht mehr alle Zellen durch Speicher in der Hardware repräsentiert werden, ist die Position eines Agenten nicht mehr von der Adresse des Speicherplatzes bestimmbar. Aus diesem Grund muss die Position i bzw. (x, y) des Agenten im Speicher mit abgespeichert werden.

Da die Verteilung der Agenten in den Datenspeichern nicht bekannt ist und auch in jeder Generation wechselt, durchsucht der Lese- und Schreibarbiter (R/W) den Datenspeicher und schreibt die Agenten in den Lesepuffer (RB). Um den Suchvorgang zu beschleunigen, ist in dem Flagregister (Flag, 1 Bit je Speicherposition) vermerkt, welche Speicherposition belegt ist. Der NIOS II Prozessor greift direkt auf den Lesepuffer (RB) zu und kann effizient nacheinander auf die Agenten zugreifen und deren neue Position bestimmen. Die einzelnen Blöcke werden nacheinander für entsprechende Custom Instructions in Register (r) geschrieben. Mit einer zusätzlichen Custom Instruction werden dann alle Blöcke aus dem Register (r) in den Schreibpuffer (WB) übertragen. Ein Arbiter selektiert einen Schreibpuffer und berechnet aus der neuen Position i bzw. (x, y) des Agenten unter Verwendung der Hashfunktion $f_h(i)$ eine gültige Speicheradresse. Hierbei treten u. U. Kollisionen durch bereits belegte Speicherplätze auf. Der Arbiter sorgt dann durch einen rehash für eine gültige neue Adresse. Der Lese- und Schreibarbiter (R/W) der entsprechenden VE übernimmt dann die Aufgabe des Schreibens der neuen Daten in den Datenspeicher. Da das Schreiben von Daten über einen globalen Arbiter abgewickelt wird, sind die Lese- und Schreibarbiter (R/W) für das Schreiben von Daten priorisiert. Die Puffer (RB) und (WB) sorgen dafür, dass alle Prozessoren Agenten bearbeiten können und nicht auf das Auslesen des Datenspeichers oder das Zurückschreiben der Agenten warten müssen.

Die Berechnung der Bewegung eines Agenten erfordert die Überprüfung von weiteren Zellen, u. a. der Zielzelle. Diese Nachbarzugriffe werden über spezielle Funktionen (LADE und PRÜFE) abgewickelt. Der Prozessor berechnet dabei die zu prüfende Nachbarzelle und übergibt diese über eine Custom Instruction an die Hardware. Die Position wird durch eine Hashfunktion in eine gültige Speicheradresse umgerechnet. Unter Verwendung der PRÜFE Funktion wird über ein Netzwerk der entsprechende Datenspeicher angesprochen und es wird überprüft, ob an dieser Position Daten vorhanden sind. Um unnötige Zugriffe über das Netzwerk zu vermeiden und die Überprüfung zu beschleunigen, haben beide Funktionen Zugriff auf die Flag Register. Ist das Flag Register für die betreffende Speicherposition gesetzt, werden die Daten über das Netzwerk überprüft. Dies ist notwendig, da auf Grund der Hashfunktion mehrere Daten potenziell an dieser Speicheradresse gespeichert werden können. Durch einen Vergleich der Position wird sichergestellt, dass es sich um die richtigen Daten handelt. Mehrfache Lesezugriffe über das Netzwerk können notwendig sein. Die PRÜFE Funktion dient nur dazu, zu überprüfen, ob Daten an der angefragten Position vorhanden sind und sie übernimmt zusätzlich die Kommunikation mit dem Netzwerk sowie die Vergleiche der Positionen. Das eigentliche Lesen der Daten geschieht über die LADE Funktion. Hierbei wird die Position über die Hashfunktion in eine gültige Adresse umgerechnet und dann über das Netzwerk auf den entsprechenden Datenspeicher zugegriffen. Auch hierbei können, abhängig von der Datenverteilung, mehrfache Lesezugriffe notwendig sein. Vor dem Aufruf der LADE Funktion ist immer die PRÜFE Funktion zu verwenden, da ansonsten nicht sichergestellt werden kann, dass die korrekten Daten gelesen werden. Für den Fall, dass an der angegebenen Position keine Daten vorhanden sind, d. h. eine leere Zelle vorliegt, ist ein Zugriff über die LADE Funktion undefiniert. Die beiden Funktionen übernehmen die komplexen Vorgänge der Überprüfung bzw. des Ladens der Daten einer Zelle.

Aus der Sicht des Prozessors entsteht somit eine *agentenbasierte* Verarbeitung. Das bedeutet, dass anstatt alle Zellen der Reihe nach abzuarbeiten, nun die Agenten der Reihe nach abgearbeitet werden. Somit werden die leeren Zellen von der Berechnung ausgeschlossen. Damit ergibt sich aus der Sicht des Prozessors ein anderer Verarbeitungsansatz. Dieser ist auch entsprechend in der Zellregel zu berücksichtigen.

Die Hashfunktion $f_h(\ldots)$ für das Speichern der Daten und die Lesezugriffe über das Netzwerk ist grundsätzlich frei wählbar. In dieser Architektur wurde die folgende Hashfunktion verwendet:

$$f_h(i) = \left(i + 2 \cdot r_h\right) \mod q$$

Dabei wird der Zellindex i durch die Hashfunktion $f_h(\ldots)$ auf den verfügbaren Speicherplatz q abgebildet. Ist die so errechnete Speicherposition schon durch einen anderen Agenten belegt, muss eine neue Speicherposition berechnet werden. Dazu wird die Rehashvariable r_h für jede belegte Speicherposition um eins erhöht, bis ein freier Speicherplatz gefunden wird. An diesem wird dann der Agent abgespeichert. Für Lesezugriffe wird die gleiche Hashfunktion verwendet. Beim Lesen von Agenten muss deren abgespeicherte Position mit der angefragten Position verglichen werden. Ist diese ungleich, wird ebenfalls die Rehashvariable r_h so lange um eins erhöht, bis der Agent mit der richtigen Position gefunden wurde.

6.8.1 Definierte Custom Instructions

Als Custom Instruction (CI) werden zusätzliche Funktionen realisiert, die das Lesen des Lesepuffers (RB), der Register (r), des Schreibpuffers (WB) und der externen Lesezugriffe umsetzen. Der Prozessor kann so transparent von der Hardware Agenten lesen, Nachbarzellen prüfen und den Agenten an seine neue Position schreiben. Es wird dabei nicht auf eine Nachbarzelle geschrieben, sondern nur die Position des Agenten angepasst. Eine Behandlung der leeren Zellen entfällt, da diese in der Architektur nicht berücksichtigt werden.

Die definierten Custom Instructions dieser Architektur für B Blöcke:

- GET_NET_D{W}: Externer Lesezugriff auf den Block W ($W \in \{0, \ldots, B-1\}$) unter Verwendung des Netzwerks. Setzt die LADE Funktion um.
- WRITE_TO_HASH: Übernahme aller Blöcke aus dem Register (r) in den Schreibpuffer (WB). Nach der Übernahme in den Schreibpuffer werden die Daten in den Datenspeicher geschrieben.
- GET_IT_NEXT: Prüft, ob noch mindestens ein weiterer Agent im Lesepuffer ist und schaltet den Lesepuffer ggf. weiter.
- SET_BLOCK_D{W}: Schreibt den Block W ($W \in \{0, \ldots, B-1\}$) in das Register (r).
- GET_NEXT_FIELD_A: Lädt die Position des aktuellen Agenten aus dem Lesepuffer.
- GET_NEXT_BLOCK_D{W}: Lädt den Block W ($W \in \{0, \ldots, B-1\}$) des aktuellen Agenten aus dem Lesepuffer (RB).

- CONTAINS: Setzt die Funktionsweise der PRÜFE Funktion um. Es wird geprüft, ob ein Agent mit der angegebenen Position vorhanden ist.
- NEXTGEN: Generationssynchronisation

6.8.2 Realisierungsaufwand auf einem FPGA

Die Architektur wurde für die Netzwerke RN, RNFF, BDPA und BRRA auf einem Cyclone II FPGA realisiert. Auf Grund der nun für Multi-Agenten-Simulationen angepassten Architektur, muss das Busnetzwerk nicht mehr das Netzwerk mit den schnellsten Simulationsergebnissen sein. Deshalb wurde die Architektur mit den Netzwerken, die gute Simulationsergebnisse bei geringem Ressourcenbedarf versprechen, realisiert. Die Syntheseergebnisse sind in Tabelle 6.24 aufgeführt. Die Skalierbarkeit der Architektur ist, unabhängig vom verwendeten Netzwerk und im Vergleich zu den bisherigen Architekturen stark eingeschränkt. Eine Realisierung mit acht Verarbeitungseinheiten ist nicht möglich. Unter Verwendung des BDPA-Netzwerks wurden zusätzlich die theoretischen Syntheseergebnisse aufgeführt. Die schlechtere Skalierbarkeit der Architektur liegt in ihrem komplexeren Aufbau. Dies betrifft hauptsächlich das Flag Register. Dieses ermöglicht den gleichzeitigen Zugriff verschiedener Hardwarekomponenten auf unterschiedliche Adressen. Zusätzlich benötigen die Funktionen PRÜFE und LADE Zugriff auf alle Flag Register. Dadurch entstehen komplexe kombinatorische Verschaltungen die letztendlich die Ursache für die schlechte Skalierbarkeit darstellen.

Netzwerk	p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			LEs	Register			
-	1	13.487	-	-	153.664	3.802	75,00
RN	2	15.169	204	92	186.464	5.149	73,81
RN	4	22.902	416	184	252.064	7.757	73,08
RNFF	2	15.315	337	156	186.464	5.213	73,81
RNFF	4	23.005	565	248	252.064	7.821	70,00
BDPA	2	14.993	62	2	186.464	5.059	76,19
BDPA	4	22.579	85	3	252.064	7.576	70,84
BDPA[§]	*8*	*103.333*	*143*	*4*	*383.264*	*12.614*	-
BRRA	2	14.997	63	2	186.464	5.059	76,19
BRRA	4	22.571	84	3	252.064	7.576	71,88

Tabelle 6.24: Realisierungsaufwand der spezialisierten GCA-Architektur mit Hashfunktionen auf einem Cyclone II FPGA mit **2** Blöcken für die Netzwerke: Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA), Busnetzwerk mit Round-Robin Arbitrierung (BRRA).
[§] Die Skalierbarkeit dieser Architektur ist eingeschränkt. Dargestellt sind die Syntheseergebnisse. Der Ressourcenbedarf ist für das verwendete FPGA zu groß.

Der komplexe Aufbau der Architektur spiegelt sich auch in der geringen Taktfrequenz wieder, der für eine Verarbeitungseinheit bei 75 MHz liegt. Das Verbindungsnetzwerk sorgt daher nicht mehr für eine zusätzliche Abnahme der Taktfrequenz. Obwohl die Hardwarearchitektur keine guten Skalierungseigenschaften aufweist, sind dennoch die Simulationsergebnisse von Multi-

Agenten-Anwendungen interessant, um den generellen Ansatz für eine derartige Architektur bewerten zu können.

6.8.3 Eine Agentenanwendung

Die Auswertung der Architektur erfolgte mit der gleichen Agentenwelt, die auch in Abschnitt 6.6.4.2 verwendet wurde, für 100 Generationen. Auf Grund der unterschiedlichen Architektur ist die Zellregel entsprechend angepasst. Durch die Simulation der gleichen Agentenwelt, lassen sich die Simulationsergebnisse gut miteinander vergleichen. Allerdings ist ein Vergleich über die CUR ungeeignet, da die Architektur mit Hashfunktionen nicht mehr alle Zellen berechnet. Deshalb wurde, zusätzlich zur CUR, auch die AUR berechnet. Die AUR gibt dabei an, wie viele Agenten pro Zeiteinheit berechnet werden können. Die Ergebnisse der Simulation sind in Tabelle 6.25 für die Netzwerke RN, RNFF, BDPA und BRRA aufgelistet.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
-	1	3.022.117	40,29	-	896	5.025
RN	2	2.288.114	31,00	1,30	1.164	6.532
RN	4	1.744.145	23,86	1,69	1.512	8.484
RNFF	2	2.288.031	30,99	1,30	1.164	6.532
RNFF	4	1.594.874	22,78	1,77	1.584	8.887
BDPA	2	1.810.657	23,77	1,70	1.519	8.520
BDPA	4	1.260.010	17,79	2,27	2.029	11.384
BRRA	2	1.730.362	22,71	1,77	1.589	8.916
BRRA	4	1.249.669	17,39	2,32	2.076	11.647

Tabelle 6.25: Auswertung der Agentenwelt auf der HA auf einem Cyclone II FPGA für die Netzwerke: Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA), Busnetzwerk mit Round-Robin Arbitrierung (BRRA).

Die Zellregel für das Agentenverhalten, das auch in Abschnitt 6.6.4.2 verwendet wurde, ist für die neue Architektur angepasst, im Quellcode 6.7 aufgeführt. Im Gegensatz zu den vorherigen Architekturen, muss nicht mehr auf leere Zellen überprüft werden. Des weiteren werden nicht die Zellen, sondern die Agenten der Reihe nach durchlaufen. Dies wird über die Custom Instruction `GET_IT_NEXT` in Zeile 8 umgesetzt. Diese erlaubt es, die Zellregel so lange anzuwenden, so lange noch Agenten für die Abarbeitung vorliegen. Durch den Aufruf `CI(GET_IT_NEXT,0,0)` wird der Lesepuffer RB weitergeschaltet und die neuen Agentendaten können über die Blockfunktionen (z. B. `GET_NEXT_BLOCK_D0` in Zeile 11) gelesen werden. Da nun nicht mehr die Zellen abgearbeitet werden, sondern die Agenten, kann die Position des Agenten über die Funktion `GET_NEXT_FIELD_A` in Zeile 10 gelesen werden. Das Verhalten des Agenten wird nun analog zu Abschnitt 6.6.4.2 umgesetzt. Die Funktion PRÜFE ist durch die Custom Instruction `CI(CONTAINS,Adresse,0)` (z. B. in Zeile 22) und der Lesezugriff über das Netzwerk auf benachbarte Verarbeitungseinheiten z. B. auf den Block 0 durch die Custom Instruction `CI(GET_NET_D0,Adresse,0)` (u. a. in Zeile 26) realisiert. Die Übernahme der neuen Agentendaten in die Hardwareregister wird über die Funktionen `SET_BLOCK_D0` (z. B. in Zeile 53) und

SET_BLOCK_D1 (z. B. in Zeile 54) umgesetzt. Nach dem die neu berechneten Daten alle in die Hardwareregister übernommen worden sind, werden alle Hardwareregister (entspricht dem gesamten Zellzustand) mit der Funktion WRITE_TO_HASH in einen der Speicher zurückgeschrieben (z. B. in Zeile 55).

Da die leeren Zellen nicht mehr betrachtet werden, gibt es nur noch Agenten (CELL_AGENT) und Hindernisse zu berücksichtigen. Für die Hindernisse wurde keine gesonderte Überprüfung vorgesehen. Daher wird alles, was keinen Agenten darstellt, als Hindernis interpretiert, was die Zellregel vereinfacht.

Auf Grund der Verwendung des R/W Arbiters, der den Speicher durchsucht, müssen alle Lesezugriffe über das Netzwerk, also extern, abgewickelt werden.

```
1  int main(){
2
3   int gen, CellAdr, CellType, CellDir, CellAdrNew, cond;
4   CI(NEXTGEN,0,0);
5
6   for(gen=0;gen<GENS;gen++)
7   {
8    while(CI(GET_IT_NEXT,0,0))          //Prüfe ob weitere Agenten vorhanden sind
9    {
10    CellAdr = CI(GET_NEXT_FIELD_A,0,0);                    //Lade Zellindex i
11    CellType = CI(GET_NEXT_BLOCK_D0,0,0);                      //Lade Zelltyp
12
13    if(CellType==CELL_AGENT){                               //Zelltyp ist Agent
14     CellDir = CI(GET_NEXT_BLOCK_D1,0,0);   //Lade die Blickrichtung des Agenten
15     switch(CellDir){                                       //Berechne Zielzelle
16      case NORTH: CellAdrNew=CellAdr-MAXY; break;
17      case SOUTH: CellAdrNew=CellAdr+MAXY; break;
18      case EAST: CellAdrNew=CellAdr+1; break;
19      case WEST: CellAdrNew=CellAdr-1; break;
20     }
21
22     if(CI(CONTAINS,CellAdrNew,0)==0){                     //Zielzelle ist frei
23      cond=0;
24                                                           //Prüfe Nachbarzellen
25      if(CI(CONTAINS,CellAdrNew-MAXY,0)
26      && CELL_AGENT==CI(GET_NET_D0,CellAdrNew-MAXY,0)
27      && SOUTH==CI(GET_NET_D1,CellAdrNew-MAXY,0))             //Zelle im Norden
28      {cond++;}
29
30      if(CI(CONTAINS,CellAdrNew+MAXY,0)
31      && CELL_AGENT==CI(GET_NET_D0,CellAdrNew+MAXY,0)
32      && NORTH==CI(GET_NET_D1,CellAdrNew+MAXY,0))              //Zelle im Süden
33      {cond++;}
34
35      if(CI(CONTAINS,CellAdrNew+1,0)
36      && CELL_AGENT==CI(GET_NET_D0,CellAdrNew+1,0)
37      && WEST==CI(GET_NET_D1,CellAdrNew+1,0))                  //Zelle im Osten
38      {cond++;}
39
40      if(CI(CONTAINS,CellAdrNew-1,0)
```

```
41      && CELL_AGENT==CI(GET_NET_D0,CellAdrNew-1,0)
42      && EAST==CI(GET_NET_D1,CellAdrNew-1,0))                  //Zelle im Westen
43      {cond++;}
44     }
45     else{                                                  //Zielzelle ist nicht frei
46      cond=999;
47     }
48
49     if(cond>1){
50                                                          //Keine Bewegung möglich
51      if(CellDir==WEST) CellDir=NORTH;  else CellDir+=2;
52
53      CI(SET_BLOCK_D0,CellAdr,CellType);//Agent bewegt sich nicht, alte Position
54      CI(SET_BLOCK_D1,CellAdr,CellDir);                              //speichern
55      CI(WRITE_TO_HASH,0,0);
56     }
57     else{
58                                                        //Bewegung möglich
59      switch(CellDir){
60       case NORTH: CellDir=EAST; break;
61       case SOUTH: CellDir=WEST; break;
62       case EAST: CellDir=NORTH; break;
63       case WEST: CellDir=SOUTH; break;
64      }
65
66      CI(SET_BLOCK_D0,CellAdrNew,CellType);       //Agent an berechnete Position
67      CI(SET_BLOCK_D1,CellAdrNew,CellDir);                           //speichern
68      CI(WRITE_TO_HASH,0,0);
69     }
70    }
71    else{                                                           //Hindernis
72     CI(SET_BLOCK_D0,CellAdr,CellType);
73     CI(WRITE_TO_HASH,0,0);
74    }
75   }
76   CI(NEXTGEN,0,0);                                  //Generationssynchronisation
77  }
78 return 0;}
```

Quellcode 6.7: Zellregel der Agentenanwendung für die HA

6.8.4 Bewertung der Hardwarearchitektur

In der HA wurden neue Ideen umgesetzt. Der zunächst einfache Ansatz, nur aktive Zellen zu berechnen, stellt sich in der Umsetzung als schwierig heraus, wenn gleichzeitig der Platzbedarf (also die Skalierbarkeit), eine hohe Simulationsgeschwindigkeit und andere Kriterien berücksichtigt werden müssen. In der HA wird die Berechnung nicht mehr ausgehend von den Zellen durchgeführt. Es wird ein *agentenbasierter* Ansatz verwendet. Das bedeutet, dass nicht die einzelnen Zellen berechnet werden, sondern die Agenten. Die Verwendung der Zellen wird dabei zu einer weiteren Abstraktionsebene. Das Problem des agentenbasierten Ansatzes ist, dass die Zellstruktur nicht mehr direkt in der Hardware gegeben ist. Somit muss eine Lösung gefunden werden, wie auf andere Zellen zugegriffen werden kann, bzw. auf die Agenten, die sich dort

potenziell aufhalten können. Ein Durchsuchen des Speichers für jeden externen Lesezugriff ist eine triviale und gleichzeitig schlechte Lösung, da dies sehr lange dauern würde. Das Durchsuchen des Speichers wird daher nur für die internen Lesezugriffe auf die Agenten verwendet. Eine Leseeinheit durchsucht parallel zur Berechnung der Zellregel im NIOS II Prozessor den Speicher. Die gefundenen Agenten werden dann in einem Puffer zwischengespeichert. Somit kann der NIOS II Prozessor sehr schnell auf die Agenten im eigenen Datenspeicher zugreifen. In der HA werden für die externen Lesezugriffe Hashfunktionen eingesetzt. Über die Hashfunktion wird die Adresse im Zellfeld auf eine reale Adresse im Speicher abgebildet. Somit besteht ein schneller Zugriff auf alle Zellen. Zusätzlich zu diesen beiden Mechanismen stehen Flagregister zur Verfügung. Über diese Register kann innerhalb eines Taktzykluses geprüft werden, ob eine Speicherstelle belegt oder frei ist. Somit werden die sonst notwendigen Speicherzugriffe zusätzlich beschleunigt. Ein weiterer Effekt ist, dass die Datenspeicher nun nicht mehr alle Zellen der Agentenwelt speichern können müssen, sondern nur noch alle Agenten. Somit ist die Simulation sehr großer Agentenwelten möglich, ohne dass weiterer Speicher benötigt oder die Simulationszeit erhöht wird. Nachteilig wirkt sich dagegen aus, dass die Lesezugriffe nun aus zwei Teilen bestehen. Somit werden doppelt so viele Lesezugriffe notwendig. Mit dem ersten Lesezugriff wird geprüft, ob die Zielzelle ein berechenbares Objekt, also eine aktive Zelle, enthält. Mit dem zweiten Lesezugriff werden dann die eigentlichen Daten gelesen. Dieser Prozess kann aber mit zusätzlicher Logik verhindert werden, indem die beiden Lesezugriffe zusammengefasst werden. Ein weiterer Nachteil der Architektur ist, dass auf Grund der Hashfunktion die Agenten ungleich in den Datenspeichern verteilt liegen. Somit kann es sein, dass die Verarbeitungseinheiten unterschiedlich viele Agenten berechnen müssen. Im Extremfall würde eine Verarbeitungseinheit alle Agenten berechnen, während die restlichen Verarbeitungseinheiten keine Agenten berechnen. Dieser Fall kann durch die Wahl der Speichergrößen und der Hashfunktion gut verhindert werden. Dennoch ergibt sich immer eine ungleiche Verteilung, wodurch die Verarbeitungseinheiten nicht optimal genutzt werden. Die Verteilung der Agenten ist dabei aber nicht konstant und ändert sich in jeder Generation. Im Durchschnitt kann man eine relativ gute Verteilung annehmen, sofern die Datenspeicher in Bezug auf die Anzahl der Agenten nicht unverhältnismäßig groß gewählt werden.

Wie die Ergebnisse der Simulation gezeigt haben, ist mit dieser Architektur auch mit wenigen Verarbeitungseinheiten eine große Beschleunigung möglich. Dies zeigen die Vergleiche mit den vorherigen Architekturen (MPA und MPAB). Die schlechte Skalierbarkeit wirkt sich aber negativ aus. Der Platzbedarf ist zu groß und die Taktfrequenz zu niedrig. Dennoch ist der Ansatz vielversprechend und insbesondere die Möglichkeit der Simulation großer Agentenwelten mit geringen Datenspeichern interessant. Dies zeigt auch wieder, dass ein allgemeiner Vergleich zwischen den Architekturen nur schwer durchzuführen ist. Für die Simulation einer sehr großen Agentenwelt mit wenigen Agenten ist die HA trotz der erwähnten Nachteile sehr gut geeignet.

Die HA ist generell für Multi-Agenten-Anwendungen optimiert. Somit lassen sich verschiedene Agentenwelten simulieren. Durch entsprechende Einstellungen kann die HA aber auch als generelle GCA-Architektur eingesetzt werden. Die Zellregel der Anwendung muss ebenso auf die HA angepasst werden. Dann könnten auch Anwendungen wie das Bitonische Mischen oder das Bitonische Sortieren auf der HA ausgeführt werden. Dazu ist es aber zwingend notwendig, dass alle Zellen abgespeichert werden können, weshalb die Datenspeicher entsprechend groß gewählt werden müssen. Eine Unterscheidung in aktive und inaktive Zellen entfällt. Es

ist allerdings zu erwarten, dass die HA für klassische GCA-Anwendungen keine Beschleunigungen erreichen wird. Dies liegt daran, dass im Vergleich zur MPA mehr Zugriffe über das Verbindungsnetzwerk abgewickelt werden müssen und zusätzliche Kollisionen durch die Verwendung einer Hashfunktion behandelt werden müssen[6]. Des weiteren ist die Taktfrequenz auch für wenige Verarbeitungseinheiten zu gering. Die HA ist nicht für den Einsatz von klassischen GCA-Anwendungen gedacht.

[6] In diesem Fall bietet es sich an, als Hashfunktion die Identitätsfunktion zu verwenden. Es wäre dann auch nicht mehr notwendig vor einem Lesezugriff die PRÜFE Funktion zu verwenden.

6.9 Agentenbasierte speicheroptimierte GCA-Architektur (DAMA)

Die Multi-Agenten-Simulationen auf der HA haben gezeigt, dass durch die vorgenommene Optimierung die Simulation stark beschleunigt werden kann (Ein Vergleich aller Hardwarearchitekturen ist in Abschnitt 6.10 durchgeführt worden). Da die Skalierbarkeit der HA auf $p = 4$ Prozessoren begrenzt ist, wird eine neue, besser skalierbare Architektur, abgekürzt DAMA (engl.: Dedicated Agent Memory Architecture), vorgestellt [SHH10a, SHH11b]. Dabei sollen die Vorteile der HA erhalten bleiben. Zusätzlich zu den generellen Problemen, die beim Entwurf einer Architektur auftreten, wird der Schwerpunkt der Multi-Agenten-Simulation noch stärker betont. Es wird dazu eine Hardwarefunktion *Vier-Nachbarn* entworfen, die lokale Bewegungen von Agenten unterstützt. Diese, von Multi-Agenten-Anwendungen häufig benötigte Funktion beschleunigt die Simulation zusätzlich. Dadurch entfallen kostspielige der ansonsten nötigen Funktionsaufrufe der Custom Instructions (CI). Die Funktion Vier-Nachbarn, überprüft dabei die vier Nachbarzellen (Nord, Ost, Süd, West) einer Zelle C_i. Der Verilogcode (Quellcode 9.12) dieser Architektur befindet sich im Anhang Kapitel 9.

Die Verarbeitungseinheit (VE) der Architektur (Abb. 6.31) besteht aus der Hardwarefunktion Vier-Nachbarn, einem Agentenspeicher, einem Zellspeicher, dem Programmspeicher sowie dem NIOS II Prozessor. Alle Verarbeitungseinheiten sind über ein Netzwerk, über das die Lesezugriffe realisiert werden, miteinander verbunden. Alle Zellspeicher enthalten zusammen das gesamte Zellfeld, also die Agentenwelt. Hierin werden aber keine Daten der Zellen abgespeichert. Der Zellspeicher enthält nur Zeiger auf die Agentenspeicher. Die Agentenspeicher enthalten neben der Position des Agenten die eigentlichen Agentendaten. Über die Zellspeicher kann so die Position jedes Agenten direkt bestimmt werden. Leere Zellen werden durch einen ungültigen Zeiger im Zellspeicher realisiert. Sowohl der Zellspeicher als auch der Agentenspeicher können für die Abarbeitung der Zellregel zwei Generationen speichern. Der Programmspeicher enthält die Zellregel, also das Verhalten der Agenten.

Jeder Prozessor bearbeitet nur die Agenten in dem ihm zugeordneten Agentenspeicher. Alle Agentenspeicher enthalten gleichviele Agenten. Die eigentlichen Agentendaten bleiben immer in dem jeweiligen Agentenspeicher und werden nicht über das Netzwerk in andere Agentenspeicher kopiert. Schreibzugriffe werden direkt auf dem Agentenspeicher ausgeführt. Zusätzlich zu den Schreibzugriffen auf dem Agentenspeicher müssen auch die Zeiger der Zellspeicher angepasst werden. Da die gesamte Agentenwelt auf alle VE verteilt ist, kann der Zeiger in einer anderen VE sein wie der Agent selbst. Die neue Zeigerinformation muss deshalb an die korrekte VE weitergeleitet werden. Dazu sind alle VE über ein weiteres Ringnetzwerk (PF) verbunden, das verwendet wird, um die neue Zeigerinformation an die betreffende VE weiterzuleiten. Diese Weiterleitung und das Schreiben auf den richtigen Zellspeicher sind transparent von der Abarbeitung der Zellregel und benötigen keine Interaktion mit den Prozessoren.

Für die Berechnung eines Agenten lädt der NIOS II Prozessor die relevanten Daten aus dem Agentenspeicher. Durch Anwendung der Zellregel wird die neue Position des Agenten bestimmt und evtl. weitere agentenspezifische Daten. Die Position sowie die Agentendaten werden direkt in den Agentenspeicher geschrieben. Die Aktualisierung der Position im Zellspeicher ist dagegen etwas komplexer. Bewegt sich der Agent auf einer Zelle, die dem Zellspeicher der eigenen VE

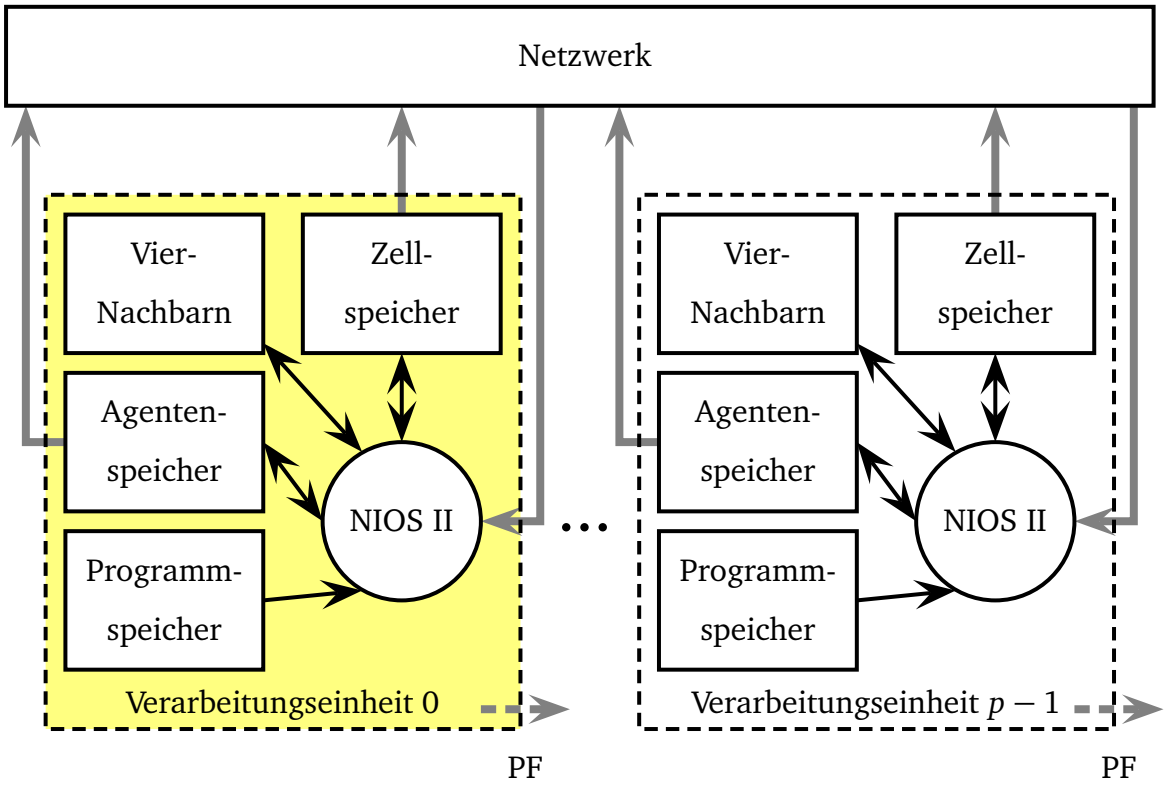


Abbildung 6.31: Agentenbasierte speicheroptimierte GCA-Architektur (DAMA)

zugeordnet ist, so wird ein Zeiger auf die entsprechende Stelle des Agenten im Agentenspeicher in den Zellspeicher geschrieben. Die Zelle im Zellspeicher, die der neuen Position des Agenten entspricht, zeigt nun auf den Agenten im Agentenspeicher. Bewegt sich der Agent auf eine Zelle, die im Zellspeicher einer anderen VE gespeichert ist, so wird die Zeigerinformation über das zusätzliche Ringnetzwerk weitergeleitet. Die entsprechende VE liest dann die Daten und schreibt diese an die korrekte Stelle im Zellspeicher. Nun zeigt die Zelle im Zellspeicher einer anderen VE auf den Agenten im Agentenspeicher. Die für die Bewegung der Agenten notwendigen Lesezugriffe auf andere Zellen werden zur Kollisionsprüfung über das Netzwerk abgewickelt. Über das Netzwerk besteht für jede VE Zugriff auf alle Zellspeicher und alle Agentenspeicher. Somit kann jede Zelle und auch jeder Agent gelesen werden. Die eigenen Speicher können auch direkt gelesen werden und erfordern keinen Netzwerkzugriff. Um andere Zellen zu überprüfen, ist ein Zugriff auf den Zellspeicher anderer VE notwendig. Damit kann aber immer nur bestimmt werden, ob die geprüfte Zelle leer ist oder nicht. Soll zusätzlich z. B. die Blickrichtung eines Agenten bestimmt werden, so muss zuerst der Zeiger aus dem Zellspeicher gelesen werden, um mit diesem Zeiger die benötigten Daten aus dem Agentenspeicher zu lesen. In diesen einfachen Leseprozess können insgesamt drei VE involviert sein. Eine VE fragt die Daten an. Die zweite VE besitzt die Zeigerinformation in ihrem Zellspeicher und die dritte VE enthält die eigentlichen Agentendaten in ihrem Agentenspeicher.

6.9.1 Definierte Custom Instructions

Die Custom Instructions (CI) realisieren die Lesezugriffe auf die Speicher (Agentenspeicher und Zellspeicher) und die Schreibzugriffe auf die internen Speicher. Das Schreiben des Zeigers in den richtigen Speicher wird von der Hardware übernommen. Für die Lesezugriffe entscheidet die Hardware anhand eines Adressvergleiches, ob Zugriffe intern oder extern auszuführen sind. Getrennte Befehle für internes oder externes Lesen sind in dieser Architektur nicht vorhanden und werden auch nicht benötigt. Wie bei der HA werden nur die Agenten berechnet. Leere Zellen können über einen ungültigen Zeiger im Zellspeicher identifiziert werden. Die Zellregel wird nicht auf dem Zellspeicher ausgeführt, sondern auf dem Agentenspeicher. Der Zellspeicher dient nur zur Adressauflösung für Lesezugriffe auf andere Zellen.

Die definierten Custom Instructions dieser Architektur für B Blöcke sind:

- RD_AGT_ADR: Lesezugriff auf die Zellspeicher der VE. Liefert den gespeicherten Zeiger zu einem Zellindex i.
- RD_B{W}: Liest den Block W ($W \in \{0, ..., B-1\}$) eines Agenten. Der Zugriff wird sofern möglich intern, ansonsten extern abgewickelt.
- GET_NXT_AGT: Setzt die Adresse des zu lesenden Agenten. Hiermit ist ein Durchlaufen aller Agenten sowie wahlfreier Zugriff auf den Agentenspeicher möglich.
- RD_NXT_AGT_B{W}: Liest den Block W ($W \in \{0, ..., B-1\}$) des aktuell ausgewählten Agenten. Diese Funktion liefert die Daten des ausgewählten Agenten schneller als die Funktion RD_BW.
- WR_SEND: Übergibt den Zeiger zum Schreiben in den richtigen Zellspeicher an die Hardware. Die Funktion terminiert nach einem Taktzyklus, sofern das Netzwerk zum Schreiben nicht belegt ist.
- WR_B{W}: Schreibt den Block W ($W \in \{0, ..., B-1\}$) in den Agentenspeicher.
- CHECK_CELL: Startet eine Hardwarefunktion zur Überprüfung der angrenzenden Zellen und gibt die Anzahl der Agenten mit Blickrichtung auf die Adresse der übergebenen Adresse zurück. Eine detaillierte Beschreibung der Funktion ist in Abschnitt 6.9.2 zu finden.
- NEXTGEN: Generationssynchronisation

6.9.2 Agenten Hardware-Funktion: Vier-Nachbarn

Zur Ausführung der Zellregel bzw. des Agentenverhaltens werden, wie beschrieben, Custom Instructions eingesetzt. Jede Custom Instruction bedeutet eine Übergabe von Daten aus der Zellregel (Software) an die Architektur (Hardware). Bevor eine Custom Instruction ausgeführt werden kann, sind drei Übergaberegister mit den entsprechenden Werten zu belegen. Die drei Werte sind die Funktion, die die Custom Instruction ausführen soll und zwei Datenwerte. Jeder Aufruf einer Custom Instruction ist also mit einem Mehraufwand verbunden, der die Simulation unnötig verlangsamt. Das Laden und Schreiben von Daten in die Datenspeicher kann nicht

weiter optimiert werden. Sofern einige Datenwerte nicht zwingend geschrieben werden müssen, können diese entfallen, dafür sind aber meistens Überprüfungen in Software notwendig, so dass insgesamt keine Beschleunigung erreicht werden kann.

Für die Bewegung der Agenten sind aber weitere Aufrufe von Custom Instructions notwendig. Es muss für jede Bewegung überprüft werden, ob die Zielzelle frei ist. Ebenso müssen alle Nachbarzellen überprüft werden, denn es kann sich immer nur ein Agent auf eine Zelle bewegen[7]. Dazu müssen Kollisionen ausgeschlossen werden. Für eine einfache Geradeausbewegung eines Agenten bedeutet dies, dass vier Zellen überprüft werden müssen. Somit sind also auch vier Custom Instruction Aufrufe notwendig, für die wiederum vier mal die entsprechenden Register geladen werden müssen. Durch das Einfügen einer Hardwarefunktion, die über die Custom Instruction `CHECK_CELL` aufgerufen wird, ist nur noch ein Aufruf einer Custom Instruction notwendig. Die Hardware übernimmt dann die Überprüfung der vier Zellen.

Damit die Funktion allgemein, also sowohl für leere Zellen als auch für Agentenzellen verwendet werden kann, ist ein allgemeiner Prüfmechanismus vorzusehen. Ausgehend von einer Zelle C_i werden die vier Nachbarzellen geprüft. Für eine leere Zelle wird die Funktion mit dem Zellindex der leeren Zelle aufgerufen. Für eine Agentenzelle wird die Funktion mit dem Zellindex der Zelle, die in Bewegungsrichtung des Agenten liegt, aufgerufen. Als Ergebnis liefert die Funktion die Anzahl der Agenten, die sich auf die Frontzelle bewegen wollen. Ruft ein Agent die Funktion auf, so ist der Rückgabewert mindestens 1 (der Agent selbst möchte auf die Frontzelle). Die Hardwarefunktion berücksichtigt auch die Bewegungsrichtung der Agenten. Agenten auf benachbarten Zellen mit einer Blickrichtung, die nicht auf die zu prüfende Zelle zeigt, werden dementsprechend nicht berücksichtigt, da diese keine Kollision verursachen können.

6.9.3 Realisierungsaufwand auf einem FPGA

Zur Auswertung verschiedener Multi-Agenten-Simulationen wurde die Architektur für zwei Netzwerke, die bei den bisher benutzten Architekturen die besten Ergebnisse lieferten, auf einem Cyclone II FPGA realisiert. Die Syntheseergebnisse für das RNFF und das BDPA Netzwerk sind in Tabelle 6.26 gegeben. Im Vergleich zur HA wurde die Skalierbarkeit wieder hergestellt. Es können bis zu $p = 16$ Prozessoren auf dem verwendeten FPGA realisiert werden. Da das Verbindungsnetzwerk sowohl für die Zellspeicher als auch für die Agentenspeicher verwendet wird, ergibt sich ein höherer Ressourcenbedarf für die Verbindungen. Die benötigten Speicherbits mit der HA zu vergleichen, ist nicht einfach. Die HA benötigt keinen zusätzlichen Speicher, um die Positionen der Agenten zu speichern. Allerdings muss der Speicher größer, als zum Abspeichern der Agenten notwendig, gewählt werden, um Kollisionen durch die Hashfunktion zu verhindern. Die DAMA benötigt dafür in Form des Zellspeichers mehr Speicherplatz. Beim Vergleich der beiden realisierten Architekturen stellt man fest, dass der Speicherbedarf bei beiden Architekturen in etwa gleich groß ist.

[7] In den in dieser Arbeit behandelten Multi-Agenten-Anwendungen trifft dies zu. Es ist aber durchaus möglich Multi-Agenten-Anwendungen zu definieren, bei denen mehr als ein Agent pro Zelle zugelassen ist.

Netzwerk	p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			LEs	Register			
-	1	3.748	-	-	137.952	2.075	138,89
RNFF	2	7.121	355	161	169.376	3.807	131,25
RNFF	4	13.611	592	257	232.224	7.025	105,00
RNFF	8	26.395	1.101	449	357.920	13.441	90,91
RNFF	16	52.330	2.230	833	609.312	26.284	71,88
BDPA	2	6.874	58	2	169.376	3.648	116,67
BDPA	4	13.080	135	3	232.224	6.771	95,00
BDPA	8	25.652	172	4	357.920	12.996	80,56
BDPA	16	50.886	644	5	609.312	25.441	67,50

Tabelle 6.26: Realisierungsaufwand der agentenbasierten speicheroptimierten GCA-Architektur auf einem Cyclone II FPGA mit **2** Blöcken für die Netzwerke: Ringnetzwerk mit Forwarding (RNFF) und Busnetzwerk mit dynamischer Arbitrierung (BDPA).

6.9.4 Eine Agentenanwendung

Die Agentenwelt aus Abschnitt 6.6.4.2, die auch für die HA Architektur aus Abschnitt 6.8 verwendet wurde, erlaubt den direkten Vergleich der Architekturen. Die Zellregel ist auf die neue Architektur angepasst, berechnet aber das gleiche Agentenverhalten, ebenfalls für 100 Generationen. Die zusätzlich vorhandene Hardwarefunktion Vier-Nachbarn erhöht zum einen die Simulationsgeschwindigkeit und verbessert andererseits die Lesbarkeit des Quellcodes der Zellregel. Zusätzlich zur Lesbarkeit verringert sich auch der benötigte Quellcode. Damit können Multi-Agenten-Anwendungen schneller entwickelt und umgesetzt werden. Zudem wird weniger Speicherplatz für die Zellregel benötigt. Die Ergebnisse der Simulation für die Netzwerke RNFF und BDPA sind in Tabelle 6.27 aufgeführt. Diese Architektur erreicht die bisher höchste CUR bzw. AUR. Deshalb ist diese Architektur von den bisher untersuchten Architekturen für die Multi-Agenten-Simulation am besten geeignet.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
-	1	2.158.566	15,54	-	2.322	13.029
RNFF	2	1.146.565	8,74	1,78	4.132	23.180
RNFF	4	649.371	6,18	2,51	5.837	32.743
RNFF	8	392.219	4,31	3,60	8.367	46.936
RNFF	**16**	267.863	3,73	**4,17**	9.687	54.340
BDPA	2	1.093.222	9,37	1,66	3.852	21.611
BDPA	4	577.739	6,08	2,56	5.936	33.297
BDPA	8	391.640	4,86	3,20	7.425	41.654
BDPA	**16**	341.348	5,06	**3,07**	7.138	40.043

Tabelle 6.27: Auswertung der Agentenwelt auf der DAMA auf einem Cyclone II FPGA für die Netzwerke: Ringnetzwerk mit Forwarding (RNFF) und Busnetzwerk mit dynamischer Arbitrierung (BDPA).

Die Zellregel (Quellcode 6.8) ist analog zu der Zellregel der HA aufgebaut. Mit der CI GET_NXT_AGT in Zeile 11 können die einzelnen Agenten geladen werden. Über einen Zähler können so alle Agenten der Reihe nach abgearbeitet werden. Es sind aber auch andere Schemata denkbar. Die Daten der einzelnen Blöcke werden über die Blockfunktionen (z. B. RD_NXT_AGT_B0 in Zeile 13) geladen. Die Position, d. h. der Zellindex eines Agenten, wird über die Funktion RD_AGT_ADR in Zeile 26 geladen. Die Position wird mit jedem Agenten gespeichert und ist identisch mit der Position im Zellspeicher. Der Zellspeicher wird für Lesezugriffe auf andere Zellen bzw. Agenten verwendet. Über den darin gespeicherten Zeiger gelangt man somit von dem Zellindex über den Zeiger zu den Agentendaten. Über die Funktion RD_AGT_ADR besteht Zugriff auf die Zellspeicher und den dort gespeicherten Zeiger. Über die Blockfunktionen, z. B. RD_B1, können unter Verwendung des Zeigers die Daten des Agenten auf einer anderen Zelle gelesen werden. Die Zellregel im Quellcode 6.8 verwendet diese Funktionen nicht, da hier die Hardwarefunktion Vier-Nachbarn verwendet wird. Diese wird über die CI CHECK_CELL in Zeile 27 ausgeführt. Die einzelnen Blöcke des Agenten können über die Schreibfunktionen, z. B. WR_B0 in Zeile 35, geschrieben werden. Damit wird aber nicht die Position im Zellspeicher eingetragen. Dies geschieht über WR_SEND z. B. in Zeile 37. Die komplette Zellregel ist im Quellcode 6.8 dargestellt.

```
1  int main()
2  {
3   int g, AGENT_NR, AGENT_BLOCK0, AGENT_BLOCK1, NEW_AGENT_ADR, cond;
4
5   CI(NEXTGEN,0,NXTG_KEEP);
6   CI(NEXTGEN,0,NXTG_KEEP);
7
8   for(g=0;g<GENS;g++)
9   {
10   AGENT_NR=0;                                    //Agentenzähler initialisieren
11   while(CI(GET_NXT_AGT,AGENT_NR,0)!=0)//Prüfe ob weitere Agenten vorhanden sind
12   {
13    AGENT_BLOCK0 = CI(RD_NXT_AGT_B0,0,0);                  //Lade Zellindex i
14    AGENT_BLOCK1 = CI(RD_NXT_AGT_B1,0,0);                      //Lade Zelltyp
15
16    if(AGENT_BLOCK1>CELL_BLOCK)                            //Beweglicher Agent
17    {
18     switch(AGENT_BLOCK1)                                 //Berechne Zielzelle
19     {
20      case CELL_AGENT_N: NEW_AGENT_ADR = AGENT_BLOCK0-MAXY; break;
21      case CELL_AGENT_S: NEW_AGENT_ADR = AGENT_BLOCK0+MAXY; break;
22      case CELL_AGENT_E: NEW_AGENT_ADR = AGENT_BLOCK0+1; break;
23      case CELL_AGENT_W: NEW_AGENT_ADR = AGENT_BLOCK0-1; break;
24     }
25                                      //Prüfe Nachbarzellen mit Hardwarefunktion
26     if(CI(RD_AGT_ADR,NEW_AGENT_ADR,0)==-1)
27     {cond=CI(CHECK_CELL,NEW_AGENT_ADR,MAXY);}
28     else
29     {cond=999;}
30
31     if(cond>1)
32     {                                                 //Bewegung nicht möglich
33      if(AGENT_BLOCK1==CELL_AGENT_W)
34       AGENT_BLOCK1=CELL_AGENT_N; else AGENT_BLOCK1+=2;
```

```
35      CI(WR_B0,0,AGENT_BLOCK0);          //Agent an berechnete Position speichern
36      CI(WR_B1,0,AGENT_BLOCK1);
37      CI(WR_SEND,AGENT_BLOCK0,0);
38     }
39     else
40     {                                                    //Bewegung möglich
41      switch(AGENT_BLOCK1)
42      {
43       case CELL_AGENT_N: AGENT_BLOCK1=CELL_AGENT_E; break;
44       case CELL_AGENT_S: AGENT_BLOCK1=CELL_AGENT_W; break;
45       case CELL_AGENT_E: AGENT_BLOCK1=CELL_AGENT_N; break;
46       case CELL_AGENT_W: AGENT_BLOCK1=CELL_AGENT_S; break;
47      }
48      CI(WR_B0,0,NEW_AGENT_ADR);         //Agent an berechnete Position speichern
49      CI(WR_B1,0,AGENT_BLOCK1);
50      CI(WR_SEND,NEW_AGENT_ADR,0);
51     }
52
53    }
54    else
55    {
56     CI(WR_B0,0,AGENT_BLOCK0);      //Andere Zelltypen an alte Position speichern
57     CI(WR_B1,0,AGENT_BLOCK1);
58     CI(WR_SEND,AGENT_BLOCK0,0);
59    }
60
61    AGENT_NR++;                                          //Agentenzähler
62   }
63
64   CI(NEXTGEN,0,NXTG_DELETE);                    //Generationssynchronisation
65  }
66 return 0;}
```

Quellcode 6.8: Zellregel der Agentenanwendung für die DAMA

6.9.5 Eine Agentenanwendung mit Informationsverbreitung

Als weitere Testanwendung wurde eine erweiterte Version der Agentenanwendung aus Abschnitt 6.6.4.2 simuliert. Bei dieser Anwendung ist das Agentenverhalten identisch mit der vorherigen Agentenanwendung. Zusätzlich besitzt jeder Agent eine individuelle, von den anderen Agenten disjunkte Information. Diese Information ist als positive ganze Zahl im „Gedächtnis“ des Agenten hinterlegt. Die Simulation ist beendet, wenn alle Agenten das Maximum der vorhandenen Informationen besitzen. Um an andere Informationen zu gelangen, müssen die Agenten die Informationen austauschen. Ein Agent kann die Information eines anderen Agenten lesen, wenn sich dieser auf der direkten Nachbarzelle in Bewegungsrichtung des ersten Agenten befindet. Somit bewegt sich ein Agent oder liest die Information eines anderen Agenten. Ein Agent kopiert eine Information, wenn der Wert größer ist als der Wert, den der Agent schon besitzt. Ein Agent nimmt somit nur größere Informationswerte auf. Im Verlauf der Simulation (Abb. 6.32) verbreiten sich so Informationen, die durch einen größeren Wert repräsentiert sind. Nach diesem Muster können z. B. auch andere Simulationen ablaufen. Die Ausbreitung von Krankheiten oder Viren lässt sich somit gut realisieren. Die genauen Details der Zellregel über

den Austausch und Verbleib des Informationswertes können den Anforderungen entsprechend angepasst werden.
Der Informatinsgehalt der Agenten wird durch den Farbwert repräsentiert (Abb. 6.33). Ein geringer Informationsgehalt wird durch die Farbe weiß dargestellt, ein großer Informationsgehalt wird durch die Farbe blau dargestellt. Anhand der farblichen Abstufung sind alle Zwischenwerte dargestellt. Die maximale Information ist durch einen roten Farbwert hervorgehoben.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
-	1	8.065.475	61,45	-	1.997	11.204
RNFF	2	4.279.450	32,10	1,91	3.824	21.451
RNFF	4	2.430.984	24,31	2,53	5.049	28.322
RNFF	8	1.434.116	15,93	3,86	7.703	43.208
RNFF	**16**	947.199	13,37	**4,60**	9.180	51.492

Tabelle 6.28: Auswertung der Agentenwelt mit Informationsverteilung auf der DAMA auf einem Cyclone II FPGA für das Ringnetzwerk mit Forwarding (RNFF).

Die Simulationsergebnisse sind in Tabelle 6.28 dargestellt. Die benötigten Hardwareressourcen für diese Architektur sowie die verwendete Zellregel sind in Abschnitt 9.5 aufgeführt. Die besten Ergebnisse wurden mit dem RNFF erzielt. Lediglich für zwei Verarbeitungseinheiten erreichte das BDPA leicht bessere Ergebnisse. Die Simulation dieser Agentenwelt wurde für 340 Generationen durchgeführt. So lange dauert es in dieser Agentenwelt, bis sich die maximale Information auf alle Agenten verteilt hat. In Abbildung 6.32 ist der Verlauf der Ausbreitung für vier unterschiedliche Zeitpunkte dargestellt.

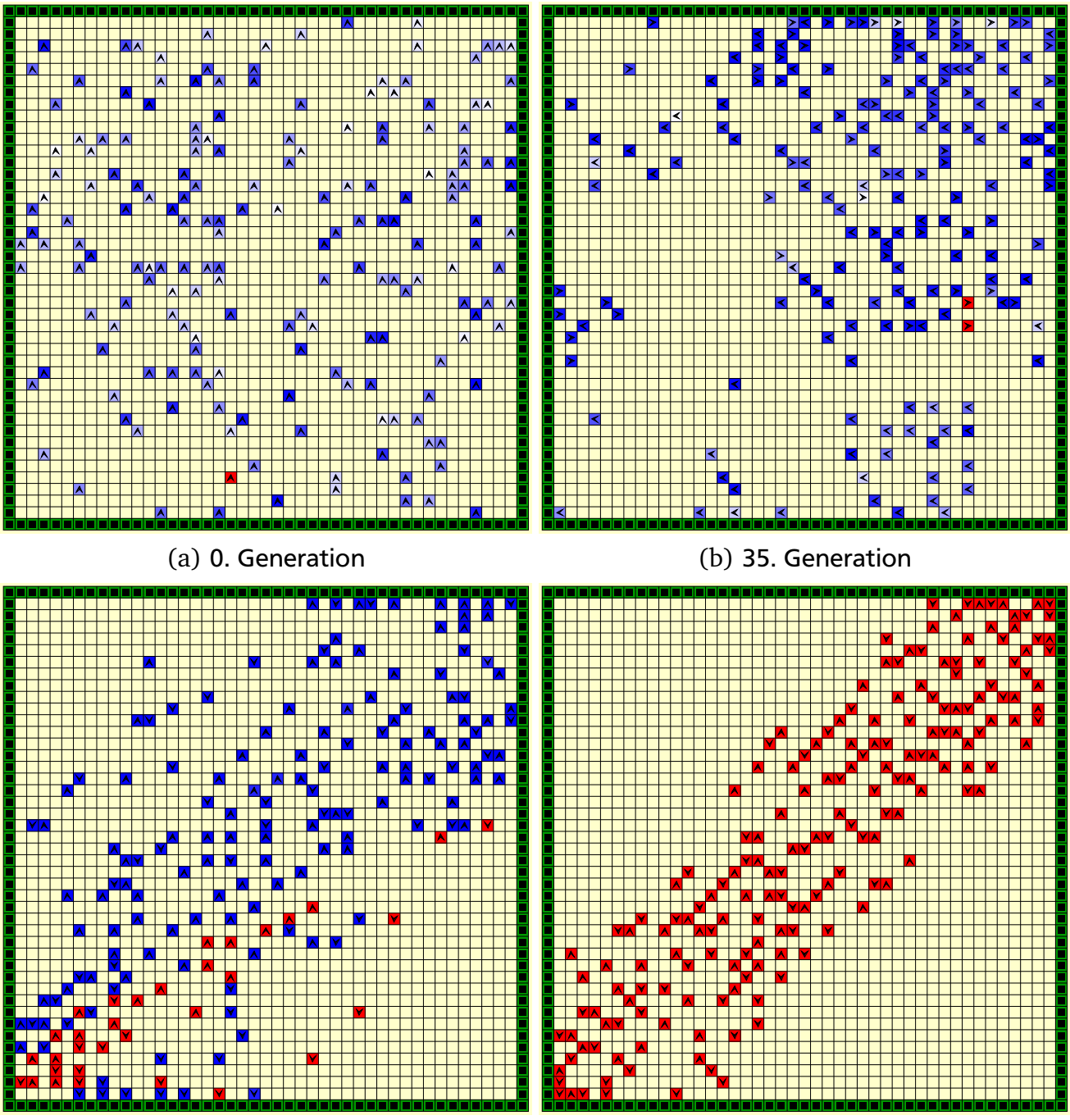
Die Simulation für diese Agentenwelt erfordert, dass jeder Agent (bzw. jede Zelle) drei Informationen abspeichern kann. Der Zustand jeder Zelle besteht also aus drei einzelnen Informationen:

1. Der Richtung des Agenten
2. Dem Typ des Agenten (bzw. der Zelle)
3. Und der disjunkten Information im Gedächtnis des Agenten

Für die Simulation dieser Agentenwelt ist es angebracht, die Architektur für die Unterstützung von drei Blöcken zu erweitern. Damit können die drei Teilinformationen, die insgesamt den Zellzustand ergeben, abgespeichert werden. Dies ist zwar nicht zwingend erforderlich, erleichtert aber die Programmierung und zeigt zudem die Flexibilität der Architektur sowie die Auswirkungen von komplexeren Zellregeln. Der zusätzliche Block erzeugt weitere Lesezugriffe, die dann auch über das Verbindungsnetzwerk abgewickelt werden müssen[8].

Vergleicht man die Werte aus Tabelle 6.28 mit den Werten aus Tabelle 6.27, also die identische Agentenanwendung einmal mit und einmal ohne Informationsaustausch, so fällt auf, dass sowohl die AUR als auch die CUR etwa gleich gute Werte aufweisen. Bei einem Vergleich der

[8] Alternativ könnte die maximale Information als Agent interpretiert werden, was einen zusätzlichen Block in der Architektur vermeiden würde.



(a) 0. Generation (b) 35. Generation

(c) 150. Generation (d) 340. Generation

Abbildung 6.32: Verbreitung der Information(en) nach: (a) 0 Generationen, (b) 35 Generationen, (c) 150 Generationen, (d) 340 Generationen

Abbildung 6.33: Die Information der Agenten wird durch den Farbwert repräsentiert. Rot repräsentiert die maximale Information.

Ausführungszeiten muss beachtet werden, dass die Simulation der Agenten mit Informationsaustausch für 340 Generationen simuliert wurde, wohingegen die Agenten zuvor für 100 Generationen simuliert wurden. Die Leistungsfähigkeit der Architektur mit drei Blöcken ist insgesamt etwas geringer. Dies liegt daran, dass diese komplexer ist und damit eine geringere maximale Taktfrequenz erreicht. Des weiteren steigen, je nach Agentenanwendung, die externen Lesezugriffe. Allerdings ist der Unterschied gering, was der Vergleich der AUR und der CUR zeigt. Die AUR sowie die CUR berücksichtigen bereits die unterschiedliche Anzahl an simulierten Generationen.

Um die Auswirkungen einer unterschiedlichen Agentenanzahl beurteilen zu können, wurde diese Agentenanwendung zusätzlich mit 20, 50 und 100 Agenten für 340 Generationen simuliert. Da die Agentenanzahl verschieden ist, ergeben sich auch unterschiedliche Simulationszeiten (in Generationen) bis zum Abschluss des Informationsaustausches. Die 340 Generationen reichen damit nicht unbedingt aus. Zu Zwecken der Vergleichbarkeit wurden die Agentenwelten aber alle 340 Generationen lange simuliert, unabhängig davon, ob der Informationsaustausch bereits abgeschlossen war oder noch weitere Generationen notwendig gewesen wären. Für die Simulationen wurde auch wieder das RNFF Netzwerk verwendet. U. U. erweisen sich für unterschiedlich viele Agenten verschiedene Netzwerke als optimal. Dieser Aspekt wurde aber nicht weiter untersucht, da die Unterschiede in der Simulationsdauer aus der bisherigen Erfahrung als sehr gering angesehen werden und zudem ein möglichst allgemeines Agentensystem gesucht wird. Die Ergebnisse der Simulationen für 20, 50 und 100 Agenten auf der Architektur mit 16 Verarbeitungseinheiten sind in Tabelle 6.29 aufgeführt. Die AUR ist für 185 Agenten geringer als beispielweise für 20 Agenten. Dies liegt daran, das nicht nur die 185 bzw. 20 Agenten simuliert werden müssen, sondern auch die Hindernisse, die den Weltrand bilden. Die Hindernisse können viel schneller simuliert werden, also die Agenten. Ein Hindernis muss nur in die nächste Generation kopiert werden wohingegen ein Agent ein komplexeres Verhalten halt. Dieser muss prüfen ob er sich bewegen kann, wie die neue Blickrichtung ist und ob ein Informationsaustausch stattzufinden hat. Deshalb sinkt mit steigender Agentenanzahl auch die AUR.

Netzwerk	Agenten	Taktzyklen	Ausführungszeit [ms]	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
RNFF	20	328.624	4,64	14.365	148.417
RNFF	50	400.318	5,65	13.598	121.836
RNFF	100	608.934	8,60	10.917	80.096
RNFF	185	947.199	13,37	9.180	51.492

Tabelle 6.29: Auswertung der Agentenwelt mit Informationsverteilung auf der DAMA für unterschiedliche Agentenanzahlen mit 16 Verarbeitungseinheiten auf einem Cyclone II FPGA für das Ringnetzwerk mit Forwarding (RNFF).

6.9.6 Optimierung der Taktfrequenz der Architektur

Ein Problem aller bisher vorgestellten Architekturen ist die stark fallende Taktfrequenz mit steigender Anzahl an Prozessoren. Die Multiprozessorarchitekturen wurden entsprechend der Literatur [Alt10a, Elk08] erstellt. Die Verwendung des Omeganetzwerks erwies sich zunächst

als Schwachstelle der Architektur, da die langen Pfade innerhalb des Netzwerks die maximale Taktfrequenz limitieren. Dieser Effekt macht sich mit steigender Prozessoranzahl immer stärker bemerkbar. Durch entsprechende Optimierungen und den Einsatz alternativer Netzwerke gelang es, die Taktfrequenz weiter zu steigern. Damit verschob sich die Begrenzung der Taktfrequenz vom Verbindungsnetzwerk auf das Prozessorsystem an sich. Verschiedene alternative Vorgehensweisen zur Erstellung der NIOS II Prozessoren wurden durchgeführt. Letztendlich gelang es so, die Taktfrequenz des Gesamtsystems zu steigern.

In der klassischen Vorgehensweise [Alt10a] werden alle benötigten Prozessoren in einem SOPC-Projekt[9] angelegt. Dieses Gesamtmodul besitzt entsprechende Schnittstellen, die für die Umsetzung des GCA-Modells verwendet werden. Durch die Verwendung dieses Gesamtmoduls wird die Taktfrequenz negativ beeinflusst, auch wenn keine Verbindungen zwischen den Prozessoren bestehen. Jeder Prozessor kann getrennt von den anderen über die NIOS II IDE[10] programmiert werden. Somit können einfach verschiedene Speicherbereiche den Prozessoren zugeordnet werden.

Um die negativen Einflüsse zu reduzieren, wird ein SOPC-Projekt mit nur einem NIOS II Prozessor erstellt. Von diesem Modul werden dann so viele Instanzen erstellt, wie für die Architektur benötigt werden. Diese Vorgehensweise hat verschiedene Nachteile, die durch andere Mechanismen umgangen oder ausgeglichen werden müssen. Da es sich nun nicht mehr um mehrere Prozessoren handelt, sondern um mehrere Instanzen des selben Prozessors, führen auch alle Prozessoren die gleichen Instruktionen aus. Die Programmierung über die NIOS II IDE erfolgt nur noch für einen Prozessor. Die einzelnen Instruktionsspeicher der Prozessoren können erst einmal nicht mehr voneinander unterschieden werden, und beinhalten somit identische Programme. Somit besteht keine elegante Möglichkeit mehr, z. B. den Adressraum innerhalb der Zellregel aufzuteilen. Dies lässt sich am einfachsten durch eine zusätzliche Custom Instruction lösen, durch die zur Laufzeit jeder Prozessor eine ID lesen kann, die während der Synthese jedem Prozessor zugeteilt wird. Damit wird die spätere Programmierung von neuen Zellregeln erheblich vereinfacht.

Netzwerk	p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			LEs	Register			
-	1	2.869	-	-	96.960	1.628	140,02
RNFF	2	6.187	331	161	120.192	3.391	135,01
RNFF	4	12.207	614	257	166.656	6.662	135,01
RNFF	8	24.184	1.154	449	259.584	13.181	131,25
RNFF	16	48.064	2.332	833	445.440	26.220	127,78

Tabelle 6.30: Realisierungsaufwand der optimierten agentenbasierten speicheroptimierten GCA-Architektur auf einem Cyclone II FPGA mit **2** Blöcken für das Ringnetzwerk mit Forwarding (RNFF)

[9] System On A Programmable Chip
[10] Die NIOS II IDE wird durch die NIOS II Software Build Tools for Eclipse ersetzt

Mit Hilfe der gewonnenen Erkenntnisse wurde die Architektur DAMA mit dem RNFF Netzwerk neu erstellt. Die Ergebnisse der Architektur sind in Tabelle 6.30 aufgelistet. Die Taktfrequenz sinkt jetzt nur noch gering. Der Platzbedarf ist deutlich geringer und wirkt sich positiv auf die Skalierbarkeit aus. Die Leistungsfähigkeit der Architektur steigt damit insgesamt enorm an. Die neuen Ergebnisse der Simulation der Agentenwelt aus Abschnitt 6.9.4 sind in Tabelle 6.31 aufgeführt.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
-	1	2.158.566	15,42	-	2.341	13.135
RNFF	2	1.146.565	8,49	1,82	4.250	23.844
RNFF	4	649.371	4,81	3,21	7.505	42.101
RNFF	8	392.219	2,99	5,16	12.080	67.763
RNFF	16	267.863	2,10	**7,35**	17.220	96.599

Tabelle 6.31: Auswertung der Agentenwelt auf der optimierten DAMA auf einem Cyclone II FPGA für das Ringnetzwerk mit Forwarding (RNFF).

6.9.7 Untersuchung der Skalierbarkeit der Architektur

Um die Skalierbarkeit und die Leistungsfähigkeit der Architektur für eine größere Anzahl an Verarbeitungseinheiten auswerten zu können, wurde die DAMA auf einem Stratix V (5SGSMB8I4H35I4) synthetisiert [Alti]. Auf diesem FPGA können bis zu 128 Verarbeitungseinheiten umgesetzt werden. Als Verbindungsnetzwerk wurde das RNFF gewählt. Der Ansatz der Optimierung der Taktfrequenz aus Abschnitt 6.9.6 ist dabei bereits berücksichtigt. Die Resultate der Synthese sind in Tabelle 6.32 zu sehen. Die Taktfrequenz sinkt im Vergleich zu der Implementierung auf dem Cyclone II FPGA stärker ab (vgl. Tabelle 6.30). Dies liegt daran, dass auf dem Stratix V eine viel größere Taktfrequenz für eine Verarbeitungseinheit erreicht wird. Allgemein ist die Taktfrequenz auf dem Stratix V FPGA aber immer höher als die auf dem Cyclone II FPGA. Aus diesem Grund ergeben sich für die Architektur auf dem Stratix V für die gleiche Anzahl an Verarbeitungseinheiten auch geringere Speedups.

p	ALMs	ALUTs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			ALUTs	Register			
1	1.875	1.712	-	-	96.960	1.674	216,73
8	15.327	13.997	670	449	259.584	13.475	166,67
16	30.468	27.822	1.275	833	445.440	26.785	150,02
32	60.206	55.281	2.560	1.601	817.152	53.375	120,00
64	119.854	110.446	5.158	3.137	1.560.576	106.493	107,15
128	239.699	221.675	10.404	6.209	3.047.424	212.603	95,47
256	*Platzbedarf zu groß*						

Tabelle 6.32: Realisierungsaufwand der agentenbasierten speicheroptimierten GCA-Architektur auf einem Stratix V FPGA mit **2** Blöcken für das Ringnetzwerk mit Forwarding (RNFF).

Die Auswertung musste mit ModelSim V6.4a [Men] durchgeführt werden, da weder das Stratix V FPGA noch ein Vergleichbares zur Verfügung standen. Alle bisherigen Architekturen wurden hingegen direkt auf einem FPGA ausgewertet. Als Agentenanwendung diente für Vergleichszwecke die Anwendung aus Abschnitt 6.6.4.2. Die Simulation der Architektur gestaltet sich auf Grund der Verwendung der NIOS II Prozessoren komplexer. Hinzu kommt, dass die große Anzahl an Prozessoren, die verwendete Art der Instanziierung sowie die ständige Weiterentwicklung der Synthesesoftware zusätzliche Probleme erzeugen. Aus diesem Grund wird hier kurz auf die Simulation der Hardwarearchitekturen eingegangen. Als Grundlage für die Simulation dienten [Alt08, Dur07].

Um die Simulation zu starten, müssen evtl. vorhandene Speicherinhalte vorab initialisiert werden. Die Schritte zum starten der Simulation sind:

1. Generieren des SOPC Systems mit Simulationsunterstützung und den compilierten NIOS II Programmcode einbinden.
2. In ModelSim die Simulationsumgebung über `do setup_sim.do` starten.
3. Über das Kommando `s` die Übersetzung und die Simulation des SOPC Systems starten.
4. Die Simulation des SOPC Systems beenden.
5. Zusätzliche Verilog HDL [IEE04] Dateien übersetzen und die Simulation des Hauptmoduls starten.

Die über die Simulation mit ModelSim ermittelten Daten sind in Tabelle 6.33 aufgelistet. Auf Grund einer anderen Initialisierung der NIOS II Prozessoren ergeben sich leicht höhere Taktzyklen für die Simulation. Bezüglich der Taktzyklen wird die beste Leistung mit 64 Verarbeitungseinheiten erzielt. Bei mehr Verarbeitungseinheiten steigt die Anzahl der Taktzyklen wieder an. Für 128 Verarbeitungseinheiten ist die Anzahl an Agenten pro Verarbeitungseinheit gering. Daraus ergeben sich sehr viele externe Zugriffe und die Bedeutung des Verbindungsnetzwerks nimmt zu. Die externen Lesezugriffe benötigen bei einer Erhöhung der Verarbeitungseinheiten unter Verwendung des Ringnetzwerks (RNFF) auch mehr Taktzyklen. Aus diesem Grund sind Architekturen mit vielen Verarbeitungseinheiten wahrscheinlich nur effizient einsetzbar, wenn die Anzahl an Agenten pro Verarbeitungseinheit ein bestimmtes Minimum erfüllt.

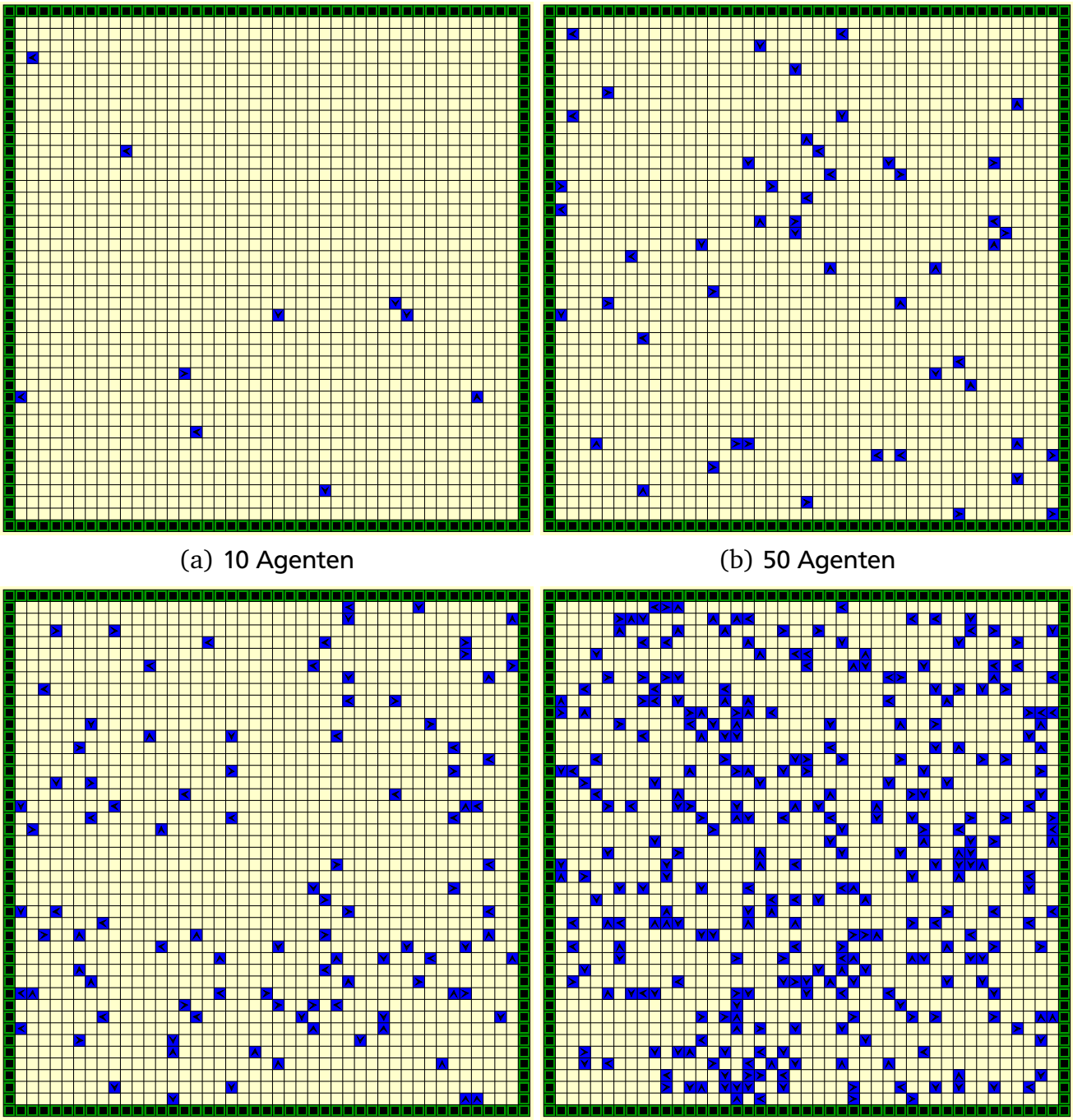
Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	AUR $\left[\frac{1}{ms}\right]$	CUR $\left[\frac{1}{ms}\right]$
-	1	2.164.424	9,99	-	3.615	20.277
RNFF	8	419.852	2,52	3,96	14.331	80.387
RNFF	16	309.843	2,07	4,84	17.479	98.047
RNFF	32	255.806	2,13	4,68	16.935	94.994
RNFF	64	235.754	2,20	4,54	16.407	92.036
RNFF	128	269.848	2,83	3,53	12.772	71.643
RNFF	256	352.531	-	-	-	-

Tabelle 6.33: Auswertung der Agentenwelt auf der optimierten DAMA auf einem Stratix V FPGA für das Ringnetzwerk mit Forwarding (RNFF).

6.9.8 Auswirkungen unterschiedlicher Agentenanzahlen

Die Anzahl der Agenten, die simuliert werden, hat Einfluss auf die Leistungsfähigkeit der Architektur. Durch eine höhere Anzahl an Agenten, steigen potenziell auch die externen Lesezugriffe. Ebenso steigen die externen Lesezugriffe, wenn die Anzahl der Verarbeitungseinheiten erhöht wird. Im Extremfall hat jeder Agent eine eigene Verarbeitungseinheit und alle Lesezugriffe finden extern statt. Die effiziente Abwicklung der externen Lesezugriffe wird also mit steigender Anzahl an Prozessoren immer wichtiger. Um diese Auswirkungen näher zu untersuchen, wurden mehrere Simulationen durchgeführt. Dazu wurde je eine Agentenwelt mit 10, 50, 100 und 300 Agenten (mit Weltrand ergeben sich 186, 226, 276, 476 zu simulierende Objekte, siehe Abbildung 6.34) auf der DAMA mit 8, 64 und 128 Verarbeitungseinheiten simuliert. Das Verhalten der Agenten war dabei immer identisch und entsprach, ebenso wie die Weltgröße, dem Verhalten der Agenten aus Abschnitt 6.6.4.2. Die Ergebnisse sind in Abbildung 6.35 dargestellt.

Zunächst fällt auf, dass für die Verdopplung der Verarbeitungseinheiten von 64 auf 128 die Anzahl der Taktzyklen gleich bleibt oder steigt. Somit ist mit keiner Geschwindigkeitssteigerung bei der Simulation zu rechnen. Für 8 und 64 Verarbeitungseinheiten ist eine Simulation von mehr Agenten immer mit einer größeren Anzahl an Taktzyklen verbunden. Für die Simulationen auf 128 Verarbeitungseinheiten stimmt dies nicht mehr. So ist die Simulation von 50 Agenten schneller, als die Simulation von 10 Agenten. Die Datenmenge pro Verarbeitungseinheit ist für 10 Agenten zu gering und die Wartezyklen können nicht ausgeglichen werden. Auffallend ist auch, dass für 50 und 300 Agenten die Simulation auf 64 und 128 Verarbeitungseinheiten etwa gleich viele Taktzyklen benötigt, während die Simulation von 10 und 100 Agenten auf 128 Verarbeitungseinheiten mehr Taktzyklen benötigen als auf 64 Verarbeitungseinheiten.



(a) 10 Agenten (b) 50 Agenten

(c) 100 Agenten (d) 300 Agenten

Abbildung 6.34: Agentenwelten mit unterschiedlicher Anzahl von Agenten

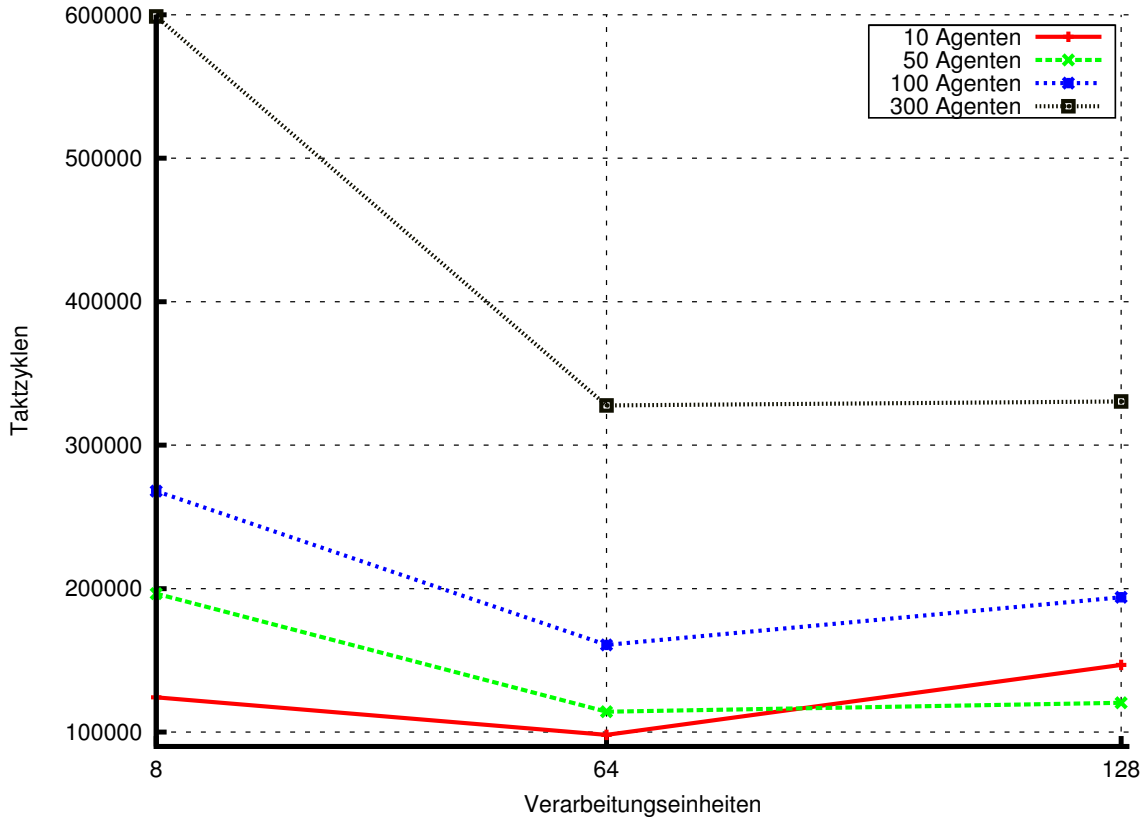


Abbildung 6.35: Auswirkungen unterschiedlicher Agentenanzahl und unterschiedlicher Anzahl an Verarbeitungseinheiten auf die Taktzyklen

6.9.9 Bewertung der Hardwarearchitektur

Aus den Ideen, die bereits bei der HA umgesetzt wurden, entstand die DAMA. Das Hauptziel dabei war, die Vorteile der HA zu erhalten, also den agentenbasierten Ansatz, aber die Nachteile zu beseitigen. Dies betrifft hauptsächlich die Skalierbarkeit, die ungleiche Verteilung der Agenten auf die Agentenspeicher sowie die Taktfrequenz. In der DAMA wurde dazu keine Hashfunktion mehr verwendet, sondern ein zusätzlicher Speicher eingeführt, der die Position der Agenten speichert. Damit existiert ein Agentenspeicher, der die Agentendaten enthält und ein Zellspeicher, der zu jeder Zelle einen Zeiger auf den Agentenspeicher enthält. Die Agentendaten können somit lückenlos in den Agentenspeichern abgespeichert werden und ein Durchsuchen des Speichers sowie das anschließende Zwischenspeichern der Agentendaten entfällt. Durch den einfacheren Aufbau der Architektur ist die Skalierbarkeit wieder hergestellt und die damit benötigten Hardwareressourcen entsprechend geringer. Ein weiterer Vorteil ist, dass die Agentendaten in den Agentenspeicher nicht umkopiert werden. Jede Verarbeitungseinheit bekommt gleich viele Agenten zugeordnet und behält diese über die gesamte Simulationsdauer. Lediglich die Position wird über die Zellspeicher hinweg ausgetauscht. Für zusätzliche Beschleunigungen wurde eine Hardwarefunktion hinzugefügt, die komplexere Auswertungen und mehrere externe Lesezugriffe kombiniert. Nachteilig wirkt sich der nun benötigte Zellspeicher aus. Um auf einen anderen Agenten zugreifen zu können, muss zunächst über den Zellspeicher geprüft werden, ob die Zielzelle von einem Agenten belegt ist. Ist auf der Zielzelle ein Agent vorhanden, so kann mit der dort gespeicherten Adresse auf den Agentenspeicher zugegriffen werden. Somit sind auch in der DAMA weitere Lesezugriffe hinzugekommen. Da die Zellspeicher über die Verarbeitungseinheiten verteilt sind und jede Verarbeitungseinheit nur ihren eigenen Zellspeicher beschreiben kann, ist ein weiteres Netzwerk notwendig. Über dieses Netzwerk werden zu schreibende Positionsdaten an die entsprechende Verarbeitungseinheit weitergereicht. Da die zu schreibenden Positionsdaten auf Grund der generationsweisen Berechnung nicht sofort geschrieben werden müssen, ist es möglich, die Schreibzugriffe parallel zur Zellberechnung abzuwickeln. Es muss lediglich sichergestellt sein, dass die Daten bis zur Generationssynchronisation geschrieben sind. Im Vergleich zu der HA konnte die Architektur noch einmal verbessert werden. Die Grundidee konnte dabei beibehalten und die meisten negativen Eigenschaften der HA beseitigt werden. Da in der DAMA aber ein Zellspeicher zum Einsatz kommt, ist es nicht mehr möglich, größere Welten zu simulieren. Für jede Zelle muss im Zellspeicher Platz zum Abspeichern eines Zeigers vorgehalten werden. Somit hängt die Weltgröße von der Größe des Agentenspeichers und auch von der Größe des Zellspeichers ab. Die Simulationsgeschwindigkeit konnte aber erneut gesteigert werden.

6.10 Gegenüberstellung und Auswertung der Multiprozessorarchitekturen

Vergleiche der Simulationen auf den erstellten Hardwarearchitekturen gegenüber einer Softwareimplementierung gestalten sich aus vielen Gründen als schwierig. Bei den Hardwarearchitekturen handelt es sich um keine hochgradig optimierten Architekturen. Auch ist die Architektur der verwendeten Prozessoren komplett unterschiedlich. Zusätzliche Parameter, wie z. B. die Cachegröße, erschweren einen Vergleich zusätzlich. Eine Möglichkeit besteht aber darin, den Unterschied zwischen der Ausführung eines Threads und der Ausführung auf einem Prozessor der Hardwarearchitektur zu bestimmen. Der Faktor der Ausführungszeiten kann dann dazu ver-

wendet werden, um die Ausführung mit mehreren Threads bzw. die Ausführung auf mehreren Prozessoren anzupassen und so die unterschiedlichen Charakteristika der Prozessoren auszugleichen. Dieser Vergleich ist zwar auch nicht fair, liefert aber dennoch gute Anhaltspunkte über die Leistungsfähigkeit der Hardwarearchitektur. Die Vergleichswerte wurden auf einem Intel Xeon E5335 (Codename: Clovertown) [Int07] mit 2 GHz und 4 GB DDR2 RAM ermittelt.

Testfall	p	Zeit [ms]	Speedup
A	1	26.942	1,00
A	4	8.806	3,06
A	8	5.376	5,01
A	16	3.493	7,71

Tabelle 6.34: Agentensimulation des Testfalls A auf der Hardwarearchitektur (MPA, Abschnitt 6.6)

Der Testfall A aus Abschnitt 5.3.2 wurde für den Vergleich auf der Hardwarearchitektur aus Abschnitt 6.6 (Einarmige universelle GCA-Architektur (MPA)) ausgeführt. Die Ergebnisse sind in Tabelle 6.34 aufgeführt.

Testfall	p| τ	Java Threads Zeit [ms]	virtuelle Hardware Zeit[ms]
A	1	1.157	1.157
A	2	875	-
A	4	1.140	378
A	8	2.125	230
A	16	-	150

Tabelle 6.35: Gegenüberstellung der Ausführungszeiten der Agentensimulation des Testfalls A mit Java Threads und der Hardwarearchitektur aus Abschnitt 6.6. Die Ausführungszeiten der Hardwarearchitektur sind auf die Leistung des Intel Xeon E5335 Prozessors angepasst.

Mit der Ausführungszeit für einen Thread bzw. einen Prozessor ergibt sich ein Unterschied vom Faktor 23,29. Das bedeutet, dass die Ausführungszeit auf einem Thread auf dem Prozessor etwa um den Faktor 24 schneller ist als die Ausführung auf der Hardwarearchitektur mit einem NIOS II Prozessor. Dieser Faktor ist hauptsächlich auf den großen Unterschied in der Taktfrequenz zurückzuführen. Teilt man die Ausführungszeit für einen NIOS II Prozessor durch diesen Faktor, erhält man die gleiche Ausführungszeit wie für einen Thread. Der Unterschied der Architekturen ist damit ausgeglichen. Die Ausführungszeiten aus Tabelle 6.34 können nun alle durch diesen Faktor geteilt werden, um die Unterschiede der Architekturen auszugleichen. Die Gegenüberstellung der Ausführungszeiten ist in Tabelle 6.35 aufgeführt. Bei der Simulation der gleichen Agentenwelt unter Berücksichtigung des Unterschiedsfaktors zeigt sich die Hardwarearchitektur als leistungsfähiger. Die Erhöhung der Prozessoranzahl spiegelt sich auf Grund der Architektur direkt in den Ausführungszeiten wieder. Der Vergleich wurde mit der zuerst neu entwickelten Hardwarearchitektur durchgeführt. Die im weiteren Verlauf umgesetzten Optimierungen, vor allem die Optimierung der Taktfrequenz der Architektur aus Abschnitt 6.9.6, sind darin noch

nicht enthalten. Insgesamt steigt die Leistungsfähigkeit der Hardwarearchitekturen damit noch weiter an.

Anstatt die Ausführungszeit für eine Verarbeitungseinheit bzw. für einen Thread anzugleichen, kann auch eine weitere Vergleichsmethode herangezogen werden. Dazu wird angenommen, dass eine Verarbeitungseinheit mit der gleichen Taktfrequenz betrieben werden kann, wie auch der Vergleichsprozessor. In diesem Fall bedeutet das, dass die Architektur mit einer Verarbeitungseinheit eine Taktfrequenz von 2 GHz hätte. Der Unterschiedsfaktor zwischen dem Vergleichsprozessor und der Architektur mit einer Verarbeitungseinheit liegt bei etwa 14,3. Mit diesem Faktor werden nun auch die Taktfrequenzen für die Architektur mit mehreren Verarbeitungseinheiten umgerechnet. Schlussendlich ergeben sich dann die in Tabelle 6.36 dargestellten Werte. Bei diesem Vergleich schneidet die Hardwarearchitektur für eine Verarbeitungseinheit schlechter ab. Für mehr Verarbeitungseinheiten bzw. Threads steigt aber die Leistungsfähigkeit der Architektur an, obwohl die Werte insgesamt schlechter sind wie in dem vorangegangenen Vergleich.

Testfall	p| τ	Java Threads Zeit [ms]	virtuelle Hardware Zeit[ms]
A	1	1.157	1.886
A	2	875	-
A	4	1.140	616
A	8	2.125	376
A	16	-	245

Tabelle 6.36: Gegenüberstellung der Ausführungszeiten der Agentensimulation des Testfalls A mit Java Threads und der Hardwarearchitektur aus Abschnitt 6.6. Die Taktfrequenz der Hardwarearchitektur ist auf die Taktfrequenz des Intel Xeon E5335 Prozessors angepasst.

Um die entwickelten Hardwarearchitekturen aus Kapitel 6 vergleichen und die Auswirkungen der Optimierungen bewerten zu können, wurde auf allen Architekturen die Agentenanwendung aus Abschnitt 6.6.4.2 ausgeführt. Die Ergebnisse der Simulation sind in Abbildung 6.36 gegenübergestellt. Da die Änderungen der Architekturen auch Auswirkungen auf das verwendete Netzwerk haben, ist nicht für alle Architekturen das gleiche Netzwerk optimal. In Abbildung 6.36 sind daher die Ergebnisse für das jeweils schnellste Netzwerk dargestellt. Für die MPA und die HA ist das BDPA am schnellsten. Für die DAMA ist das RNFF am schnellsten. Die MPAB ist nicht dargestellt, da sich die Änderungen der Architektur auf die allgemeine Benutzbarkeit beziehen und keine Optimierungen bezüglich einer effizienteren Simulation beinhalten. Die DAMA ist zusätzlich in einer optimierten Version dargestellt. Die Optimierung bezieht sich auf die Taktfrequenz des Systems, wie in Abschnitt 6.9.6 beschrieben. Durch diese Optimierung ist die Architektur für 16 Verarbeitungseinheiten noch einmal um den Faktor 1,78 schneller als die Architektur ohne Taktoptimierung. Die erreichbaren Beschleunigungen, die durch die neuen Architekturen möglich sind, sind in Tabelle 6.37 dargestellt. So ist z. B. die DAMA 4,88 mal schneller als die MPA (bei 16 VE und der Simulation der Agentenwelt). Der Abfall der Beschleunigung der DAMA gegenüber der MPA liegt an den zusätzlichen externen Lesezugriffen, die bei

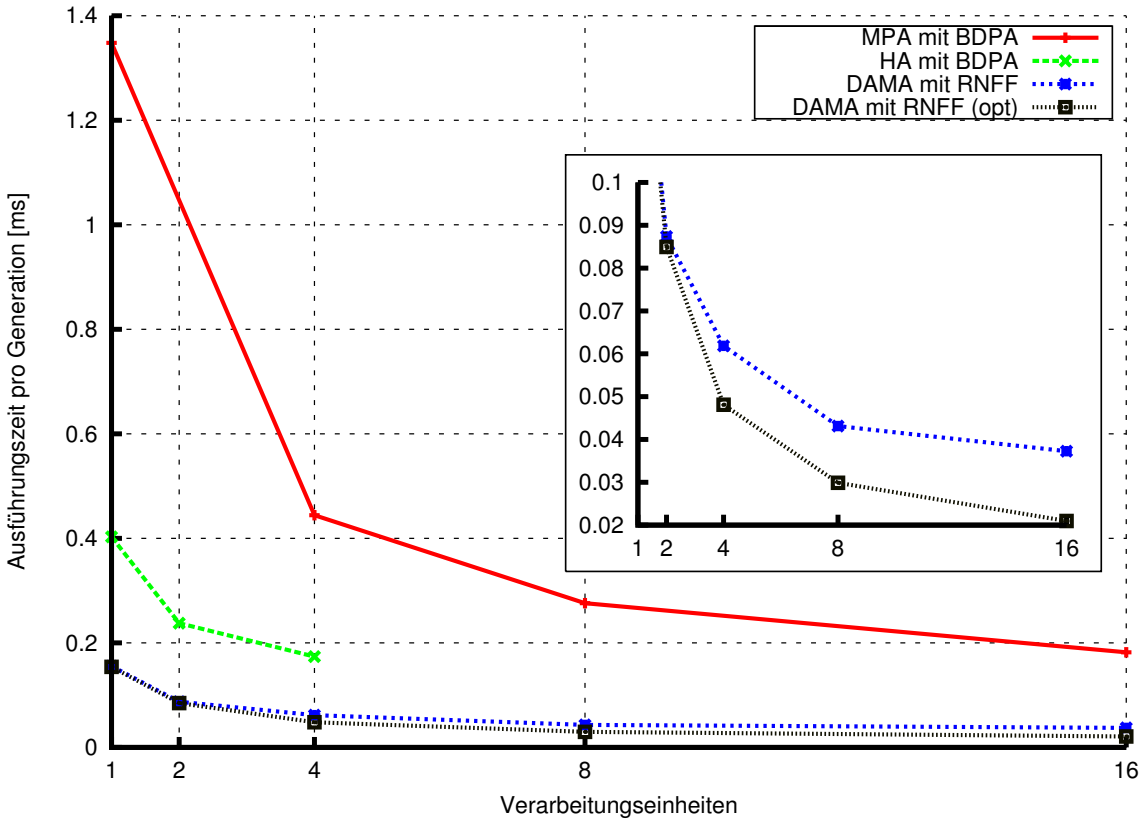


Abbildung 6.36: Gegenüberstellung der entwickelten Hardwarearchitekturen

der DAMA benötigt werden. Dadurch steigt die Anzahl der externen Lesezugriffe für viele VE stark an. Mit DAMA(opt) ist die DAMA mit der Optimierung der Taktfrequenz gemeint.

p	HA gegenüber MPA	DAMA gegenüber MPA	DAMA gegenüber HA	DAMA(opt) gegenüber DAMA
1	3,35	8,67	2,59	1,01
2	3,03	8,24	2,72	1,03
4	2,50	7,18	2,88	1,29
8	-	6,39	-	1,44
16	-	4,88	-	1,78

Tabelle 6.37: Erreichbare Beschleunigungen der Architekturen im Vergleich

In Tabelle 6.38 sind die positiven und die negativen Eigenschaften der Hardwarearchitekturen bezogen auf eine effiziente Multi-Agenten-Simulation gegenübergestellt. Die Bewertung erfolgte dabei unabhängig vom verwendeten Verbindungsnetzwerk. Es ist dabei unterstellt, dass das jeweils optimalste Netzwerk verwendet wird.

Hardware-architektur	positive Eigenschaften	negative Eigenschaften
MPA Einarmige universelle GCA-Architektur	• nicht nur für die Agentensimulation einsetzbar	• nicht agentenoptimiert • schlechte Simulationsgeschwindigkeit • zellbasierte Berechnung • Agentendaten müssen zwischen den VE kopiert werden
MPAB Multiarmige speicherkonfigurierbare universelle GCA-Architektur	• flexibler einsetzbar als die MPA	• schlechte Simulationsgeschwindigkeit • zellbasierte Berechnung • Agentendaten müssen zwischen den VE kopiert werden
HA Spezialisierte GCA-Architektur mit Hashfunktionen	• unbegrenzte Weltgröße • berechnet nur Agentenzellen • hohe Simulationsgeschwindigkeit • agentenbasierte Berechnung • Speichergröße minimiert	• schlechte Skalierbarkeit • begrenzt durch die Anzahl der Agenten • ungleiche Verteilung der Agenten auf die VE • komplexe Hardwarearchitektur
DAMA Agentenbasierte speicheroptimierte GCA-Architektur	• gute Skalierbarkeit • Agenten gleichmäßig auf die VE verteilt • berechnet nur Agentenzellen • sehr hohe Simulationsgeschwindigkeit • dedizierte Hardwarefunktion • agentenbasierte Berechnung • Agenten werden nicht kopiert	• begrenzt durch die Weltgröße • begrenzt durch die Anzahl der Agenten • benötigt zwei Speicher • zusätzliches „Schreibnetzwerk“ notwendig • mehrfach Lesezugriffe notwendig

Tabelle 6.38: Gegenüberstellung der Eigenschaften der Hardwarearchitekturen bezüglich der Multi-Agenten-Simulation

6.11 Zusammenfassung

In diesem Kapitel wurden verschiedene Hardwarearchitekturen vorgestellt, die alle als Multiprozessorarchitektur mit NIOS II Prozessor ausgelegt sind.

Einarmige universelle GCA-Architektur (MPA):
Angefangen von der MPA, einer Hardwarearchitektur die direkt das GCA-Modell umsetzt und noch nicht auf die Besonderheiten von Multi-Agenten-Anwendungen eingeht, bis hin zu für die Multi-Agenten-Simulation optimierten Hardwarearchitekturen (DAMA). Die Auswertung erfolgte anhand klassischer GCA-Anwendungen, wie z. B. das Bitonische Mischen und Sortieren. Im Vergleich zu einer davor implementierten Multiprozessorarchitektur lassen sich damit mehr Zellen bei einer gleichzeitig höheren Geschwindigkeit abarbeiten.

Multiarmige speicherkonfigurierbare universelle GCA-Architektur (MPAB):
Eine allgemeinere Hardwarearchitektur, die MPAB, die auch komplexere Agentenverhalten ermöglicht, wurde vorgestellt. Als Multi-Agenten-Anwendung kam die Verkehrssimulation zum Einsatz. Diese wurde für das GCA-Modell entwickelt und ausgewertet. Der Vergleich mit einer Implementierung im CA-Modell zeigt, dass hohe Beschleunigungen möglich sind. Die dafür notwendigen Bedingungen wurden ausführlich diskutiert.

Spezialisierte GCA-Architektur mit Hashfunktionen (HA):
Eine weitere Hardwarearchitektur unter der Verwendung von Hashfunktionen wurde realisiert. Die Vorteile sowie die Nachteile der Architektur wurden dargestellt. Durch den Einsatz von Hashfunktionen und den damit verbundenen Problemen ist die Skalierbarkeit der Architektur eingeschränkt. Es ergeben sich jedoch auch eine Reihe von Vorteilen. So werden nicht mehr alle Zellen berechnet, sondern nur noch Agentenzellen. Die Rechenleistung wird damit auf die Agenten konzentriert. Des weiteren sind mit dieser Architektur unbegrenzt große Agentenwelten simulierbar. Nur die Anzahl der Agenten in der Welt ist begrenzt.

Agentenbasierte speicheroptimierte GCA-Architektur (DAMA):
Mit der DAMA wurde eine Architektur entwickelt, die die Nachteile der HA aufhebt und deren Vorteile weitestgehend erhält. Die Leistung konnte weiter gesteigert und die Skalierbarkeit wieder hergestellt werden. Die Agentenwelten sind dafür bezüglich der Zellanzahl sowie der Agentenanzahl begrenzt. Die Simulation unbegrenzt großer Agentenwelten, wie dies bei der HA möglich war, ist bei der DAMA nicht mehr gegeben. Da die DAMA die bis dahin am weitesten entwickelte und auf die Multi-Agenten-Simulation optimierte Architektur darstellt, wurden eine Reihe weiterer Optimierungen und Verbesserungen untersucht und umgesetzt. Dies betrifft spezielle Hardwarefunktionen für die Multi-Agenten-Simulation sowie die Untersuchung der abnehmenden Taktfrequenz bei steigender Anzahl von Verarbeitungseinheiten. Des weiteren wurden unterschiedliche Einflüsse auf die Simulationsgeschwindigkeit untersucht. So z. B. die Auswirkungen unterschiedlicher Agentenanzahlen oder die Skalierbarkeit der verwendeten Netzwerke.

Alle Architekturen setzen einen 32 Bit Prozessor, den NIOS II Prozessor, ein. Auf Grund der Beschränkungen der verwendeten FPGAs wurden 16 Bit Datenwerte und entsprechend breite

Netzwerke eingesetzt. Somit ist der Anspruch an allgemein einsetzbare Hardwarearchitekturen gegeben. Das Kapitel schließt mit einer Gegenüberstellung der Hardwarearchitekturen ab.

7 Spezialarchitekturen und weitere durchgeführte Untersuchungen

In diesem Kapitel werden Spezialarchitekturen für die zellulare Verarbeitung, für das GCA-Modell und auch für die Multi-Agenten-Simulation, die im Rahmen von Diplom- und Masterarbeiten entstanden sind, vorgestellt. Eine Arbeit befasst sich mit der Anbindung einer GCA-Architektur an einen Universalrechner. Ziel ist es, bestimmte Anwendungen oder Teile davon zur Laufzeit auf die GCA-Architektur auszulagern, um damit die Berechnung zu beschleunigen. In einer anderen Arbeit werden spezielle Hardwarearchitekturen für die Simulation bestimmter Multi-Agenten-Anwendungen implementiert. Hierbei wurde auch der Einsatz von endlichen Zustandsautomaten betrachtet. Ebenfalls wurde der Einsatz der *Global Cellular Automata Experimental Language* (GCA-L) untersucht. Anhand einer einfachen Zellregel für den Globalen Zellularen Automaten wird gezeigt, wie eine Zellregel, die in GCA-L beschrieben wurde, in C-Code mit Custom Instructions (CI) übersetzt werden kann. Der C-Code wird dann auf den NIOS II Prozessoren der Hardwarearchitektur ausgeführt.

7.1 GCA-Architekturen in Verbindung mit Universalrechnern

In der Diplomarbeit von Kai Denker [Den09] sollte die Fragestellung gelöst werden, wie Architekturen auf der Basis des GCA-Modells mit Universalrechnern verbunden werden können. Dazu wurde in dieser Diplomarbeit 2009 der erste Entwurf einer GCA-Architektur weiterentwickelt (Abschnitt 6.6). Auf Grund der historischen und forschungsbedingten nicht immer klaren Entwicklung der Hardwarearchitekturen ergeben sich einige Inkonsistenzen, die in [Den09] zu Recht bemängelt wurden. Allerdings sind diese Aspekte für die im Vordergrund stehenden Fragestellungen irrelevant und ohne Implikation auf die Fragestellung. Die Diplomarbeit ist primär auf das GCA-Modell ausgelegt und berücksichtigt nicht die besonderen Anforderungen der Multi-Agenten-Simulation. Aus diesem Grund sind viele der hier im späteren Verlauf aufgetretenen und gelösten Fragestellungen dort nicht berücksichtigt.

Ein wichtiger Aspekt, der die Verbindungsnetzwerke betrifft, wurde in dieser Arbeit aufgegriffen. In den bisherigen Architekturen wurde immer nur ein Verbindungsnetzwerk verwendet. Wie die Untersuchungen zeigen, gibt es derzeit kein Netzwerk, das unter allen Bedingungen optimale Ergebnisse liefert. Zum einen liegt dies am Aufbau der Architekturen an sich, zum anderen an den unterschiedlichen Anwendungen. In [Den09, Seite 29] wurde der Ansatz, mehrere Netzwerke in der Architektur vorzuhalten, umgesetzt. Durch die Anwendung kann das Verbindungsnetzwerk dann ausgewählt werden. Damit kann abhängig von der Applikation das optimalste Netzwerk gewählt werden. Die Umsetzung mehrerer Netzwerke führt aber dazu, dass der Ressourcenverbrauch steigt und die Taktfrequenz weiter sinkt. Die Taktfrequenz kann nur maximal so hoch sein, wie die Taktfrequenz, die für das langsamste Netzwerk erreicht wird. Die Umsetzung mehrerer Netzwerke erfordert zusätzliche Hardware auf den Daten- und Kon-

trollpfaden für die Selektierung des gewählten Netzwerks. Somit sinkt insgesamt die Leistung des Systems weiter ab und die Leistungssteigerung, die durch die Wahl des passenden Netzwerkes erreicht werden kann, kommt nicht mehr zum Tragen. Unabhängig von den Resultaten ist der Ansatz dennoch interessant. Für eine noch effektivere Verwendung wäre eine automatische Selektion des passenden Netzwerkes denkbar. Damit würde sich das Verbindungsnetzwerk von sich aus an die Anwendung anpassen und ein Wechsel des Netzwerks während der Ausführung wäre denkbar. Unter Umständen ist es dann sinnvoll, für jedes Netzwerk eine Clockdomäne einzuführen.

Auf Basis der programmierbaren Multiprozessorarchitektur aus Abschnitt 6.6 wurde ein Universalrechner entworfen. Damit können sowohl herkömmliche als auch spezielle Anwendungen für das GCA-Modell ausgeführt werden. Die GCA-Anwendungen werden dabei direkt von der Hardware unterstützt. Der universelle Teil der Architektur besteht aus einem NIOS II Prozessor (auch Steuerungsprozessor genannt) mit Speichern (SDRAM und SSRAM). Die GCA-Architektur ist über einen Bus mit dem Prozessor verbunden [Den09, Seite 33]. Der NIOS II Prozessor kann über unterschiedliche Mechanismen auf die GCA-Architektur zugreifen und dort GCA-Anwendungen ausführen. In [Den09, Seite 52-54] werden die folgenden Vorteile und Nachteile, welche hier zusammengefasst dargestellt werden, aufgeführt.

Die Vorteile:

- Der Steuerungsprozessor kann die Firmware der GCA-Architektur während der Laufzeit austauschen und damit unterschiedliche GCA-Anwendungen ausführen.
- Der Steuerungsprozessor kann die Speicher der GCA-Architektur zur Laufzeit verändern.
- Über einen zusätzlichen Befehl kann der Steuerungsprozessor den Ablauf der Anwendungen auf der GCA-Architektur beinflussen. Es ist z. B. möglich, die Ausführung nach jeder Generation zu unterbrechen, die Daten auszulesen oder auch abzuändern.
- Die Größe des verwendeten Speichers kann für die jeweilige GCA-Anwendung angepasst werden.
- Über eine Registerbank können unterschiedliche Parameter der GCA-Architektur beeinflusst werden, so dass diese flexibel eingesetzt werden kann.

Die Nachteile:

- Die maximal erreichbare Taktfrequenz ist auf Grund des größeren Designs des Systems und der größeren Flexibilität stark gesunken. Es ist deshalb zukünftig zu untersuchen, inwiefern ein derartiger Ansatz allgemein geeignet ist oder ob andere Implementierungen bessere Umsetzungen ermöglichen.
- Der vorhandene Platz auf dem FPGA ist begrenzt und beschränkt daher die Möglichkeiten bei der Umsetzung derartiger Systeme. Es kann aber davon ausgegangen werden, dass in Zukunft größere FPGAs zur Verfügung stehen und deshalb dieser Ansatz weiter verfolgt werden sollte.

Neben der Hardwarerealisierung ist für die Ansteuerung der GCA-Architektur eine Softwarebibliothek entwickelt worden, mit der die GCA-Architektur über verschiedene Befehle gesteuert werden kann. Für die genaue Umsetzung dieser Bibliothek sei auf die Arbeit [Den09, Seite 57-73] verwiesen.

Um die Leistungsfähigkeit der Architektur zu messen, wurde das Bitonische Sortieren implementiert. Da das Bitonische Sortieren nicht für jede Zahlenmenge anwendbar ist, müssen evtl. freie Zellen mit neutralen Zahlenwerten gefüllt werden. Die Laufzeit des Bitonischen Sortierens ist nicht datenabhängig und die zugrundeliegende Datenmenge eine Zweierpotenz. Damit ergeben sich in der Auswertung an den Grenzen der Zweierpotenzen sprunghafte Anstiege. Der Vergleich mit anderen Sortierverfahren (Quicksort [Hoa62], Introspection Sort [Mus97]) gestaltet sich als nicht trivial, da die Vergleichsverfahren, u. a. je nach Implementierung, datenabhängig sind. Für einen Vergleich besteht neben dem Problem der Datenabhängigkeit auch das Problem der Datenmenge. Ist das Bitonische Sortieren für zufällige Datenwerte noch schneller als die Vergleichsverfahren, gilt dies nicht mehr für sortierte oder teilweise sortierte Datenwerte. Insbesondere das Introspection Sort lieferte im Vergleich sehr gute Werte. Zusätzlich ist anzumerken, dass für die GCA-Architektur acht Prozessoren zum Einsatz kamen, während die Vergleichsalgorithmen auf einem Prozessor liefen. Als Resümee kommt in [Den09, Seite 87] auf Grund dieser Ergebnisse ein Einsatz dieser Architektur nur bedingt in Frage. Es ist anzumerken, dass nur das Sortieren als Vergleich für die Architektur herangezogen wurde. Eine allgemeingültige Aussage ist somit nicht zu treffen. Auch können andere Designentscheidungen zu leistungsfähigeren Architekturen führen. Somit ist dieser Ansatz trotzdem vielversprechend und im Detail weiter zu untersuchen.

7.2 Hardwarearchitektur für das Nagel-Schreckenberg-Modell

Eine Spezialarchitektur für die Verkehrssimulation wurde im Rahmen einer Masterarbeit von Florian Geib [Gei10] entwickelt[1]. Dazu kam ein Stratix II FPGA mit zwei 256 MB DDR2-SDRAM Speichern zum Einsatz (Platform: PROCel180-A) [GiD07]. Um auch größere Agentenwelten simulieren zu können, war ein Bestandteil der Aufgabe die Verwendung der DDR2 Speicher. Die Speicher stellen damit einen Engpass dar, weshalb er möglichst effizient an das Gesamtsystem anzubinden war. Alle Lesezugriffe müssen über einen Bus auf die DDR2 Speicher durchgeführt werden. Allerdings lassen sich die beiden DDR2 Speicher gut nutzen, um zwei Generationen im Zellularen Automaten oder im Globalen Zellularen Automaten abzulegen. Somit kann aus einem Speicher gelesen und in den zweiten Speicher geschrieben werden.

Die Verkehrssimulation wurde auf Basis des in [SHH10b] vorgestellten Verfahrens (siehe auch Abschnitt 6.7.5) umgesetzt. Mit diesem Verfahren ist es möglich, die DDR2 Speicher noch effizienter zu nutzen. Es müssen nur die Agenten gespeichert werden. Leere Zellen werden nicht abgespeichert, weshalb noch mehr Platz für das Abspeichern von Agenten zur Verfügung steht. Für die Umsetzung der Verkehrssimulation wurde keine allgemeine Hardwarearchitektur, sondern eine dedizierte Hardwarearchitektur entworfen. Da sich die Agenten immer nur in eine Richtung fortbewegen, ist kein allgemeines Netzwerk für Nachbarzugriffe notwendig. Ein Agent

[1] Auf Grund eines Speicherfehlers in den DDR2 Speichern konnte nicht die maximale Leistung der Hardwarearchitektur erzielt werden.

muss nur Informationen über den nächsten Agenten in Bewegungsrichtung haben. Somit reicht es aus, wenn eine Berechnungseinheit mit der Berechnungseinheit verbunden ist, die den vorausfahrenden Agenten berechnet. Der Aufbau der Architektur ist in [Gei10, Seite 26] dargestellt. Jede Berechnungseinheit besitzt einen lokalen Speicher zum Zwischenspeichern von Agenten. Die lokalen Speicher sind über einen Arbiter an die DDR2 Speicher angebunden. Über den Arbiter werden die Agenten aus dem DDR2 Speicher in die lokalen Speicher der Berechnungseinheiten geladen. Um den vorausfahrenden Agenten nicht auch aus dem DDR2 Speicher laden zu müssen, sind die Berechnungseinheiten in einer Reihe miteinander verbunden. Somit kann der vorausfahrende Agent aus der Berechnungseinheit geladen werden, die diesen Agenten bearbeitet. Zusätzliche Speicherzugriffe auf den DDR2 Speicher werden vermieden, was die Leistung der Architektur insgesamt steigert. Sollen mehr Agenten simuliert werden, als Berechnungseinheiten zur Verfügung stehen, so werden diese nacheinander berechnet. Dazu werden mehrere *Runden* pro Generation mit verschiedenen Agenten berechnet.

Die entwickelte Architektur erreicht Taktfrequenzen von 160 MHz für zwei Berechnungseinheiten und skaliert bis zu 64 Berechnungseinheiten bei 90 MHz auf einem Stratix II FPGA (EP2S180) [Alth]. Die Auswertungen der Simulationen für 500.000 und 5.000.000 Generationen zeigen, dass ab acht Berechnungseinheiten nur noch geringfügige Beschleunigungen zu erreichen sind. Dies kann auf den zentralen DDR2 Speicher zurückgeführt werden. Die erforderlichen Lesezugriffe können nicht in der notwendigen Zeit verarbeitet werden, wodurch die Berechnungseinheiten auf die Abarbeitung dieser warten müssen und somit nicht zu einer Beschleunigung der Anwendung beitragen können. Für die Simulation wird ein Speedup bis zum 1,5-fachen im Vergleich zu einer Berechnungseinheit erreicht. Dabei wird unabhängig von der Anzahl der Berechnungseinheiten eine Taktfrequenz von 90 MHz verwendet. Nimmt man die maximal erreichbare Taktfrequenz, die man bei weniger Berechnungseinheiten erreichen kann, ergibt sich ein leicht geringerer Speedup bis zum 1,25-fachen.

7.3 Hardwarearchitektur für Agenten mit endlichen Zustandsautomaten

In [Gei10] wurde auch die Umsetzung eines Multi-Agenten-Systems unter der Verwendung von endlichen Zustandsautomaten (engl.: Finite State Machine, FSM) untersucht. Die Berechnungseinheiten der Hardwarearchitektur bestehen aus einem Zufallszahlengenerator, einem Programmspeicher, einer ALU (engl.: Arithmetic Logic Unit) sowie einem Multiplizierer und Vergleicher. Die Berechnungseinheiten sind über einen RamArbiter an die DDR2 Speicher angebunden. Durch die programmierbare Gestaltung der Architektur werden unterschiedliche Agentenanwendungen unterstützt. Die Architektur ist aber dadurch nicht mehr so leistungsfähig. Eigenschaften der Anwendung können nicht mehr oder nur noch teilweise berücksichtigt werden.

Die Architektur für die Simulation von Agenten auf der Basis von endlichen Automaten erreicht auf einem Stratix II FPGA (EP2S180) [Alth] Taktfrequenzen von 180 MHz bis 90 MHz abhängig von der Anzahl der Berechnungseinheiten. Hierbei können bis zu 64 Berechnungseinheiten auf dem FPGA realisiert werden. Als eine Anwendung wurde auch das Nagel-Schreckenberg-Modell umgesetzt. Da die Architektur verschiedene Agentenverhalten simulieren kann, ist sie nicht so leistungsfähig wie die Architektur aus Abschnitt 7.2, dafür aber allgemeiner verwendbar. Auch

bei dieser Architektur stellt die Speicherschnittstelle den Engpass dar, so dass auch hier ab acht Verarbeitungseinheiten keine nennenswerte Beschleunigung mehr erreicht werden kann. Erreichbar sind Beschleunigungen bis zum 2,6-fachen im Vergleich zu der Simulation auf einer Berechnungseinheit.

7.4 Global Cellular Automata Experimental Language

Für die Beschreibung von GCA-Anwendungen wurde die Sprache GCA-L (engl.: Global Cellular Automata Experimental Language) entwickelt [JEH08a]. Damit lassen sich Zellregeln auf geeignete Weise programmieren. Anhand eines einfachen Algorithmus (Informationsverteilung auf alle Zellen) ist die Verwendung von GCA-L im Folgenden erklärt. Der Einsatz von GCA-L kann auch die Beschreibung von Multi-Agenten-Anwendungen vereinfachen und als weitere Ebene in der entwickelten Hierarchie dienen.

Der Broadcastingalgorithmus wird verwendet, um eine Information aus einer Zelle auf alle anderen Zellen zu kopieren. Dabei wird die Information in jedem Schritt verdoppelt, um so schnellstmöglich auf allen Zellen zur Verfügung zu stehen. Diese Vorgehensweise verhindert, abhängig von dem eingesetzten Netzwerk der GCA-Architektur, Kollisionen im Netzwerk zu vermeiden.

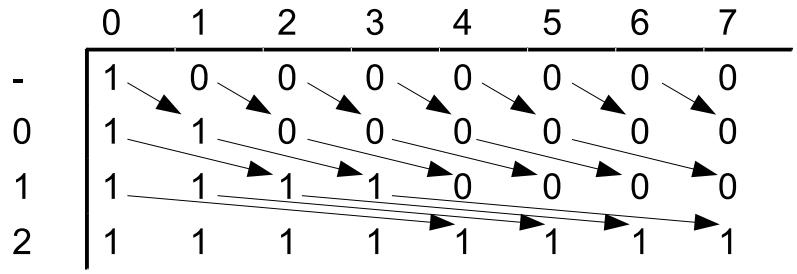


Abbildung 7.1: Darstellung des Broadcasting Algorithmus

Die Abbildung 7.1 zeigt die Vorgehensweise. Bei acht Zellen werden drei Generationen benötigt, bis die Information allen Zellen zur Verfügung steht. In jedem Schritt wird die Information verdoppelt. Dabei wechseln die Zellen in jeder Generation ihre Nachbarn. Die Beschreibung des Algorithmus in GCA-L [JEH08a] ist im Quellcode 7.1, der von Johannes Jendrsczok stammt, gegeben.

```
1  cellstructure = d;
2  neighborhood = neighbor;
3  central gen;
4  celltype cell = int;
5  cell X[8] [1];
6
7  X.d = 1,0,0,0, 0,0,0,0;
8
9  // Berechnung
10 for gen = 0 to 2 do
```

```
11  foreach X with
12   neighbor = &X[i-(1<<gen),j]
13  do
14   if (i > (1<<gen)-1) then d<=neighbor.d
15   else d <= d endif
16  endforeach
17 endfor
```

Quellcode 7.1: Zellregel des Broadcasting Algorithmus in GCA-L

Der Algorithmus wird von GCA-L durch den von Johannes Jendrsczok entwickelten Compiler nach C übersetzt und dann auf die Prozessoren der Verarbeitungseinheiten geladen. Der automatisch generierte C-Code ist im Quellcode 7.2 zu sehen. Dabei wird über die Custom Instructions `AUTO_GET_0` das Datum gelesen und über `SET_0` geschrieben. Die Generationssynchronisation wird über die Custom Instruction `NEXTGEN` umgesetzt.

```
1  int main(){
2   int i, ii;
3   int temp_i; int temp_j;
4   int LOAD_I; int LOAD_J;
5   int NB_0; int VAR_gen;
6
7   for(VAR_gen=0;VAR_gen<=2;VAR_gen++){
8    //FOREACH
9    for(i=0;i<LOCL_CELLS;i++){
10    ii = i + SC;
11    LOAD_I = (ii - 0) / 1;
12    LOAD_J = (ii - 0) % 1;
13
14    //H-Block
15    if(LOAD_I > ((1 << VAR_gen) - 1)){
16     temp_i = (LOAD_I - (1 << VAR_gen));
17     temp_j = LOAD_J ;
18    }
19    else{
20     temp_i = LOAD_I ;
21     temp_j = LOAD_J ;
22    }
23    NB_0 = CI(AUTO_GET_0, ((temp_i * 1) + temp_j + 0), 0);
24
25    //D-P-Block
26    CI(SET_0, ii, NB_0);
27   }
28   CI(NEXTGEN, 0, 0);
29  }
30  return 0;}
```

Quellcode 7.2: Übersetzte Zellregel des Broadcasting Algorithmus in C

Der C-Code aus dem Quellcode 7.2 wird für die Ausführung auf den NIOS II Prozessoren compiliert [Altd, Alte, Altf]. Der compilierte Assemblercode ist im Quellcode 7.3 aufgeführt. Einige Variablen aus dem C-Code wurden Registern des NIOS II Prozessors zugewiesen. Für folgende Register kann die Zuordnung angegeben werden: r4=i, r9=Var_gen, r11=1, r2=Rückgabewerte (teilweise NB_0).

```
1  0x0000185c <main>:    mov    r9,zero
2  0x00001860 <main+4>:  movi   r11,1
3  0x00001864 <main+8>:  movi   r8,511
4  0x00001868 <main+12>: movi   r10,2
5  0x0000186c <main+16>: sll    r7,r11,r9              // 1 << Var_gen
6  0x00001870 <main+20>: mov    r4,zero
7  0x00001874 <main+24>: addi   r6,r7,-1               // Var_gen - 1
8  0x00001878 <main+28>: addi   r3,r4,1024             // +SC
9  0x0000187c <main+32>: mov    r5,zero
10 0x00001880 <main+36>: sub    r2,r3,r7               // LOAD_I - (1 << Var_gen)
11 0x00001884 <main+40>: blt    r6,r3,0x188c <main+48>
12 0x00001888 <main+44>: mov    r2,r3
13 0x0000188c <main+48>: custom 9,r2,r2,r5             // AUTO_GET_0
14 0x00001890 <main+52>: custom 0,r2,r3,r2             // SET_0
15 0x00001894 <main+56>: addi   r4,r4,1
16 0x00001898 <main+60>: bge    r8,r4,0x1878 <main+28>
17 0x0000189c <main+64>: custom 1,r2,r5,r5             // NEXTGEN
18 0x000018a0 <main+68>: addi   r9,r9,1
19 0x000018a4 <main+72>: bge    r10,r9,0x186c <main+16>
20 0x000018a8 <main+76>: mov    r2,zero
21 0x000018ac <main+80>: ret                           // return 0
```

Quellcode 7.3: Zellregel des Broadcasting Algorithmus in NIOS II Assembler

7.5 Zusammenfassung

In diesem Kapitel wurden Spezialarchitekturen im Bezug auf das GCA-Modell vorgestellt. Die vorgestellten Architekturen reichen von Architekturen für das klassische GCA-Modell bis hin zu applikationsspezifischen Architekturen.

Zunächst wurde eine Spezialarchitektur zur Anbindung von GCA-Architekturen vorgestellt. Es wurde aufgezeigt, wie eine GCA-Architektur mit einem Universalprozessor gekoppelt werden und wie die Zellregeln in die GCA-Architektur geladen werden kann. Neben der Hardwarearchitektur steht eine Softwarebibliothek für die Programmierung der Architektur zur Verfügung.

Eine Spezialarchitektur für das Nagel-Schreckenberg-Modell auf der Basis des GCA-Modells mit Anbindung an zwei DDR2 Speicher wurde beschrieben. Durch die Anwendung ergeben sich Vereinfachungen im Verbindungsnetzwerk. Es müssen nicht alle potenziellen Verbindungen zwischen den Berechnungseinheiten vorgehalten werden, sondern nur diejenigen, die auch von der Anwendung verwendet werden.

Für allgemeine Agentenanwendungen wurde noch eine programmierbare Architektur präsentiert. Auf der Basis von endlichen Zustandsautomaten kann das Agentenverhalten programmiert werden. Zum Vergleich wurde das Nagel-Schreckenberg-Modell auf dieser Architektur umgesetzt. Auf Grund der größeren Flexibilität ist diese Architektur nicht so leistungsstark wie die Spezialarchitektur.

Die Vorgehensweise für die Umsetzung von Zellregeln, die in GCA-L beschrieben sind, wurde aufgezeigt. Dazu wurde eine einfache Zellregel von Johannes Jendrsczok mit dessen entwi-

ckeltem GCA-L Compiler in C-Code mit Custom Instructions übersetzt. Die übersetzte Zellregel wurde dann auf der Hardwarearchitektur ausgeführt.

8 Zusammenfassung und Ausblick

Zu Beginn der Arbeit wurde eine Reihe von Zielen definiert:

1. Implementierung allgemeiner Hardwarearchitekturen
2. Entwurf einer Hierarchie
3. Realisierung der Optimierungsmöglichkeiten
4. Einsatz von Spezialhardware
5. Untersuchung der Verbindungsnetzwerke
6. Auswertung und Vergleich der Hardwarearchitekturen

8.1 Ergebnisse der Arbeit

Zunächst wurde die Ausgangssituation dieser Arbeit festgestellt. Diese bezieht sich zum einen generell auf Literatur zum behandelten Themengebiet, zum anderen vor allem auch auf direkte Vorarbeiten. Im Folgenden werden die einzelnen Ergebnisse diskutiert:

1. Ausgehend vom aktuellen Standpunkt wurde eine Hardwarearchitektur (MPA) für das GCA-Modell ohne spezielle Unterstützung von Multi-Agenten-Anwendungen realisiert. Diese Architektur konnte dann unter Verwendung der identischen Anwendungen (Bitonisches Mischen und Bitonisches Sortieren) ausgewertet und direkt verglichen werden. Bereits hierbei zeigte sich eine Leistungssteigerung. Eine Gegenüberstellung mit der Vorarbeit [HHJ06], in der 128 Zahlen bitonisch gemischt wurden, lässt einen direkten Vergleich zu. Im Vergleich der beiden Architekturen für einen und 16 Prozessoren ergibt sich für einen Prozessor eine Beschleunigung von 4,38 und für 16 Prozessoren eine Beschleunigung von 1,46. Auch diese Architektur wurde für die Multi-Agenten-Simulation eingesetzt. Dadurch konnten frühzeitig Ideen für weitere Optimierungen entwickelt werden. Für die Leistungssteigerung sind der Prozessor sowie das Netzwerk verantwortlich. In [HHJ06] kam ein Kreuzschienenverteiler zum Einsatz während für die MPA ein Omeganetzwerk verwendet wurde. Die Beschränkungen der MPA (nur ein Datenfeld und nur ein Linkfeld pro Zelle) wurden in einer zweiten Hardwarearchitektur (MPAB) behoben. In dieser Hardwarearchitektur wurde auch der Einsatz von Zufallszahlen sowie eine Umsetzung von Gleitkommazahlen realisiert.

2. Um die im späteren Verlauf eingesetzten Hardwarearchitekturen in das Themengebiet einordnen und um Änderungen, die u. U. auf Grund der Optimierungen im verwendeten Modell notwendig waren, aufzeigen zu können, wurde eine Hierarchie erarbeitet. Die Hierarchie beinhaltet dabei die Anwendungsebene, die Modellebene und die Architekturebene.

Um von den Änderungen in der Anwendungsebene, die auf Grund der verschiedenen Architekturen notwendig sind, abstrahieren zu können, wurde eine Schnittstelle eingefügt. Die Idee der kompletten Umsetzung einer allgemeingültigen Schnittstelle konnte nicht bis ins Detail realisiert werden, da hierbei für jede neue Architektur Anpassungen notwendig sind. Es wurde aber aufgezeigt, dass dies mit geringem Aufwand bewerkstelligt werden kann. Eine Möglichkeit besteht im Einsatz der Global Cellular Automata Experimental Language (GCA-L). Damit kann für die jeweilige Architektur die entsprechende Zellregel generiert werden. Auch denkbar ist es, dies direkt im C-Code durch Präprozessoranweisungen umzusetzen.

3. Die Optimierungsmöglichkeiten im Hinblick auf die Multi-Agenten-Simulation führten zu neuen Hardwarearchitekturen. Zwecks Vergleichbarkeit wurde auf allen Hardwarearchitekturen eine identische Agentenwelt simuliert und die Ergebnisse gegenübergestellt. Eine der bedeutendsten Optimierungen ist, nur die aktiven Zellen, also nur die Agenten, zu simulieren. Die leeren Zellen werden von der Bearbeitung ausgeschlossen. Dieser augenscheinlich einfache Ansatz wirft bei der Umsetzung und unter der Verwendung des GCA-Modells verschiedene Schwierigkeiten auf. Zugriffe auf leere Zellen müssen einerseits gewährleistet sein und andererseits soll gleichzeitig deren Berechnung ausgeschlossen werden. Für die Multi-Agenten-Simulation müssen zudem Kollisionen ausgeschlossen werden. Somit muss ein Agent die Möglichkeit haben, festzustellen, ob er sich in eine bestimmte Richtung vorwärts bewegen kann oder nicht. Durch den Wegfall der leeren Zellen besteht zunächst damit kein direkter Zugriff mehr auf andere Zellen und die Kollisionsüberprüfung wird erschwert. Diese Problematik wurde in einer Architektur durch die Verwendung von Hashfunktionen und in einer weiteren Architektur durch den Einsatz eines zusätzlichen Speichers umgesetzt. Durch diese Optimierung konnte die Simulation deutlich beschleunigt werden. Gleichzeitig sind diese Architekturen, trotz ihres hohen Spezialisierungsgrades in der Lage, klassische GCA-Anwendungen auszuführen.

4. Eine weitere wichtige Optimierung ist der Einsatz von dedizierten Hardwarefunktionen. Für die Umsetzung von Multi-Agenten-Anwendungen werden bestimmte Funktionen oder Abfragemuster (Zugriffe auf Nachbarzellen) häufig verwendet. Dabei erfolgt die Umsetzung innerhalb der Zellregel, muss also vom NIOS II Prozessor ausgeführt werden. Eine Bereitstellung der notwendigen Funktionen beschleunigt die Anwendungen zusätzlich, erhöht die Lesbarkeit und vereinfacht die Programmierung der Zellregeln.

   Im Rahmen von Master- und Diplomarbeiten wurde der Einsatz von Spezialarchitekturen untersucht. Durch die Einschränkung auf eine bestimmte Multi-Agenten-Anwendung können die Verarbeitungseinheiten vereinfacht werden. Ebenso kann das Verbindungsnetzwerk auf die Multi-Agenten-Anwendung angepasst werden. Es wurde auch der Einsatz von größeren Speichern, auf die keine parallelen Zugriffe möglich sind, aufgezeigt. Weiterhin wurde behandelt, wie eine GCA-Architektur in einen bestehenden Universalrechner integriert werden kann.

5. Für die Multi-Agenten-Anwendungen entstehen andere Anforderungen an das Verbindungsnetzwerk als an klassische GCA-Anwendungen. Um unnötige, externe Zugriffe auf

das Verbindungsnetzwerk zu vermeiden, ist es sinnvoll, von den strengen Vorgaben des GCA-Modells abzuweichen. Dies hat auch Auswirkungen auf das Verbindungsnetzwerk. Dieses sollte einerseits in der Lage sein, parallele Zugriffe abarbeiten zu können, andererseits muss es aber auch möglich sein, einzelne Zugriffe effizient abzuarbeiten. Zudem sollte durch die Parallelität die Skalierbarkeit erhalten bleiben und die Taktfrequenz nicht zu stark sinken. Das Omeganetzwerk als kombinatorisches Netzwerk ist für Leseanfragen blockiert, so lange noch eine andere Leseanfrage abgearbeitet wird. Dies macht es erforderlich, dass die Leseanfragen, die unkoordiniert gestellt werden, entweder synchronisiert bzw. so lange blockiert werden, bis das Netzwerk wieder frei ist. Die Leistungsfähigkeit des Netzwerks wird dadurch gemindert. Für viele VE besitzt das Netzwerk zudem viele lange Datenpfade. Diese mindern die Taktfrequenz des Gesamtsystems. Durch das Einfügen einer Registerstufe kann dem entgegengewirkt werden. Eine höhere Taktfrequenz des Gesamtsystems wirkt sich stärker aus, als die zusätzlichen Taktzyklen, die nun für die Lesezugriffe benötigt werden. Auf Grund der beim Omeganetzwerk notwendigen Synchronisation der Lesezugriffe kann auch ein Busnetzwerk eingesetzt werden. Dieses ist weniger komplex, erlaubt aber, die Leseanfragen schnell weiterzuleiten. Die fehlende Parallelität wird durch die schnellere Abarbeitung ausgeglichen. Die Parallelität ist aber stark von der Menge der Lesezugriffe und deren zeitlichen Abfolge abhängig. Das Ringnetzwerk mit den vielen Registerstufen erlaubt eine hohe Taktfrequenz des Gesamtsystems, benötigt pro Zugriff aber viele Taktzyklen. Zusätzlich sind parallele, also Zugriffe von mehreren VE zur gleichen Zeit möglich. Durch ein Überspringen von Registerstufen können die Leseanfragen, insbesondere in einem großen Ring beschleunigt werden. Die Art und die Weite des Überspringens ist auch vom Design der Architektur abhängig, welches bei einer Umsetzung auf FPGAs nur begrenzt beeinflusst werden kann. Das Überspringen besitzt zudem adaptiven Charakter. Bei einem vollen Ring wird die Möglichkeit, parallele Zugriffe abzuarbeiten, voll ausgenutzt und das Überspringen in großen Teilen des Rings deaktiviert. Ist der Ring nahezu leer, können die wenigen Anfragen durch das Überspringen schneller weitergeleitet werden.

6. Die erzielten Ergebnisse der Architekturen wurden gegenübergestellt und verglichen. Aus den Untersuchungen können wichtige Erkenntnisse für zukünftige Arbeiten erlangt werden. Als Anwendung kam dazu auf allen Architekturen die Agentenanwendung aus Abschnitt 6.6.4.2 zum Einsatz. Den Werten kann entnommen werden, dass z. B. die HA gegenüber der MPA bei vier VE 2,5 mal so schnell ist. Die DAMA ist gegenüber der MPA bei vier VE sogar 7,18 mal schneller. Die Beschleunigung der DAMA gegenüber der HA beträgt 2,88. Zusätzlich skaliert die DAMA besser. Bei 16 VE erreicht die DAMA gegenüber der MPA eine Beschleunigung von 4,88. Der Wert der Beschleunigung sinkt mit steigender Anzahl an VE immer weiter. Gibt es so viele VE wie es Zellen in der Agentenwelt gibt, ist es irrelevant, ob in der MPA leere Zellen berechnet werden, da alle Zellen echt parallel berechnet werden. Der hauptsächliche Vorteil der HA und der DAMA gegenüber der MPA und gegenüber der MPAB ist, dass nicht alle Zellen berechnet werden müssen. Dieser Aspekt ist besonders für die Multi-Agenten-Simulation von Bedeutung. Die Skalierbarkeit der HA ist auf Grund der Komplexität der Hardwarearchitektur eingeschränkt. Dafür lassen sich auf dieser Architektur Agentenwelten unbegrenzter Größe simulieren. Die DAMA erreicht eine weitere Beschleunigung gegenüber der HA durch einen einfacheren Aufbau. Dadurch sind

höhere Taktfrequenzen möglich und die Skalierbarkeit ist wiederhergestellt. Die DAMA benötigt allerdings für Zugriffe auf Zellen, die in anderen VE abgespeichert sind, mehrere Lesezugriffe. Damit steigt die Anzahl der externen Lesezugriffe mit der Anzahl der VE stark an.

Nicht nur die Optimierung der Hardwarearchitekturen sorgt für eine schnelle Multi-Agenten-Simulation. Anhand des Beispiels der Verkehrssimulation wurde eine neue Zellregel erarbeitet. Durch die Transformation des ursprünglichen Zellularen Automaten in einen Globalen Zellularen Automaten ergeben sich teilweise enorme Beschleunigungen. Eine optimale Beschleunigung wird dann erreicht, wenn nicht nur die zugrundeliegende Hardwarearchitektur optimiert ist, sondern auch die Anwendung an das GCA-Modell angepasst wird.

8.2 Ausblick

Im Rahmen dieser Arbeit wurden viele Themengebiete und Ebenen bearbeitet. Behandelt wurden Probleme auf der Architekturebene, Modellebene und der Anwendungsebene. Auf Grund der Vielfältigkeit und der Komplexität bleiben noch Aspekte für zukünftige Arbeiten offen.

Auf der Anwendungsebene ist zu klären, wie bereits existierende Multi-Agenten-Anwendungen für Zellulare Automaten auf Globale Zellulare Automaten umgesetzt werden können. Diese können sicherlich direkt ausgeführt werden, doch kann erwartet werden, dass ein Teil dieser Anwendungen durch die Transformierung in das GCA-Modell beschleunigt werden können.

Auf der Architekturebene wurden verschiedene Hardwarearchitekturen implementiert. Nicht alle Optimierungsmöglichkeiten konnten umgesetzt werden und bei weitergehenden Überlegungen können sich zusätzliche Optimierungsmöglichkeiten ergeben. So können beispielsweise noch weitere Betrachtungen bezüglich des Verbindungsnetzwerks angestellt werden. Um die Taktfrequenz des Gesamtsystems auch für eine große Anzahl an Verarbeitungseinheiten möglichst hoch zu halten, wurden eine Reihe von Ideen entwickelt, die noch wissenschaftlich untersucht werden müssen. Dies gilt insbesondere für das Ringnetzwerk, das im Allgemeinen gute Ergebnisse lieferte, die Taktfrequenz der Systeme nur geringfügig negativ beeinflusst und wenig Hardwareressourcen benötigt.

Nicht zuletzt sollten auch andere Realisierungsmöglichkeiten, wovon die Wichtigsten in dieser Arbeit diskutiert wurden, nicht aus dem Auge verloren werden. Zukünftig könnten sich hier neue Impulse ergeben.

9 Anhang

9.1 Mapping der Zellregeln auf die Prozessoren

Die GCA-Architekturen bieten die Möglichkeit, jede Verarbeitungseinheit mit einer anderen Zellregel auszustatten. Für die Agentensimulation werden normalerweise alle Verarbeitungseinheiten mit der gleichen Zellregel initialisiert. Somit ist das Agentenverhalten eines Agenten innerhalb der gesamten Agentenwelt identisch. Die Verarbeitungseinheiten bearbeiten jedoch alle einen anderen Teil der gesamten Agentenwelt. Deshalb ist es notwendig, dass der globale Adressraum auf alle Verarbeitungseinheiten aufgeteilt wird, damit jede Verarbeitungseinheit einen Teil der gesamten Agentenwelt bearbeitet. Damit nicht für jede Verarbeitungseinheit eine eigene Zellregel entworfen werden muss (die sich nur in der jeweiligen Startadresse unterscheiden), geschieht dies automatisch mit Hilfe von Headerdateien.

```
1  #define PROCESSORS     4                                  //Prozessoranzahl
2  #define DAT_CELLS      2048                    //Gesamtanzahl der Datenblöcke
3  #define GENS           100                              //Agentengenerationen
4
5  #define LOCL_CELLS     DAT_CELLS / PROCESSORS       //Datenblöcke pro Prozessor
6
7  #define CPU1_SC        LOCL_CELLS*0              //Startzelle des Prozessors 1
8  #define CPU2_SC        LOCL_CELLS*1              //Startzelle des Prozessors 2
9  #define CPU3_SC        LOCL_CELLS*2              //Startzelle des Prozessors 3
10 #define CPU4_SC        LOCL_CELLS*3              //Startzelle des Prozessors 4
11 #define CPU5_SC        LOCL_CELLS*4              //Startzelle des Prozessors 5
12 #define CPU6_SC        LOCL_CELLS*5              //Startzelle des Prozessors 6
13 ...
```

Quellcode 9.1: Auszug aus der settings.h

In der Headerdatei SETTINGS.H (Quellcode 9.1) sind alle Parameter für die GCA-Architektur definiert. PROCESSORS definiert die Anzahl der Prozessoren, DAT_CELLS definiert die Gesamtzahl aller Zellen, GENS die Anzahl der auszuführenden Generationen und LOCL_CELLS definiert die Zellanzahl je Prozessor. Die Startzelle des jeweiligen Prozessors, z. B. die des zweiten Prozessors, wird mit CPU2_SC definiert. CPU2_SC gibt dabei die Adresse der ersten Zelle an, die der zweite Prozessor bearbeitet. Der Bereich an Zellen, die der zweite Prozessor zu bearbeiten hat, beginnt bei CPU2_SC und endet bei CPU2_SC + LOCL_CELLS.
Damit eine gemeinsame Zellregel für alle Verarbeitungseinheiten verwendet werden kann, ist es notwendig, dass die Zellregel beim compilieren mit der entsprechenden Startadresse versehen wird. Dazu wird in der gemeinsamen Zellregel SC verwendet, was für *StartCell* steht. Für jede Verarbeitungseinheit existiert eine C-Datei in der SC die entsprechende Startadresse zugewiesen wird (Quellcode 9.2). So wird z. B. für die zweite Verarbeitungseinheit im Quellcode 9.2 in Zeile 5 der Startzelle SC die Startzelle von CPU2_SC zugewiesen. Die gemeinsame Zellregel wird in Zeile 7 eingebunden. Für jede Verarbeitungseinheit hat SC nun die richtige Startadresse.

```
1 #include <stdio.h>
2 #include "system.h"
3 #include "../settings.h"                          //Einbinden der Headerdatei
4
5 #define SC CPU2_SC                //Initialisieren der Startzelle(SC) für CPU 2
6
7 #include "../CELLRULE.c"                            //Einbinden der Zellregel
```

Quellcode 9.2: Zellregelmapping am Beispiel der CPU 2

9.2 Der Agentensimulator AgentSim

Für die Darstellung, Simulation, Entwicklung und Auswertung von Multi-Agenten-Anwendungen wurde ein Agentensimulator (AgentSim) in Java [GJSB05] entwickelt. Neben der graphischen Visualisierung der Agentenwelten ist es möglich, diese für verschiedene Szenarien in Software zu simulieren. Für die Softwaresimulation stehen verschiedene Simulationsengines zur Verfügung. Für die Simulation mit gleichzeitiger Visualisierung steht eine Einprozessorengine, für die Entwicklung, Umsetzung und parallele Simulation von GCA-Algorithmen steht eine Multithreadengine zur Verfügung. Die Anzahl der Threads ist einstellbar und ermöglicht eine parallele Simulation der Agentenwelt auf modernen Multicoreprozessoren. Der Simulator stellt nicht nur die Simulationsergebnisse graphisch dar, er dient auch als Editor zum Erstellen und Ändern von Agentenwelten. Für die Hardwaresimulation sind entsprechende Schnittstellen implementiert, die das Schreiben, Lesen und Ansteuern der GCA-Architekturen aus dem Simulator heraus ermöglichen. Simulationsdaten sowie Statistiken werden während der Simulation angezeigt. Der Simulator inklusive Konsole ist in Abbildung 9.1 dargestellt.

9.3 Simulation mit getrennten externen Lesezugriffen

Eine weitere Möglichkeit, die Geschwindigkeit der Simulation von Multi-Agenten-Anwendungen zu erhöhen, liegt darin, die Netzwerkzugriffe zu beschleunigen. Wie bereits gezeigt wurde, können dafür unterschiedliche Verbindungsnetzwerke zum Einsatz kommen. Es ist jedoch, unabhängig vom verwendeten Verbindungsnetzwerk zu erwarten, dass mit einer steigenden Anzahl an Verarbeitungseinheiten dieses immer komplexer wird und die Zugriffe immer mehr Taktzyklen beanspruchen werden. Da die Lesezugriffe ab einem bestimmten Punkt nicht weiter optimiert werden können, ist es angebracht, die Wartezeit für einen Lesezugriff in Kauf zu nehmen. Doch anstatt auf die angefragten Daten zu warten, kann die Wartezeit durch andere Berechnungen genutzt werden. Dies erfordert aber, dass die Lesebefehle in zwei Teile aufgeteilt werden und die Zellregel entsprechend angepasst wird. Der Lesebefehl wird durch einen Anfragebefehl und einen Befehl zum Daten lesen ersetzt. Der Anfragebefehl übergibt die Leseanfrage an das Netzwerk und terminiert sofort. Der Lesebefehl kann nach dem Anfragebefehl ausgeführt werden. Er blockiert die weitere Ausführung der Zellregel so lange, bis die vorher angefragten Daten zur Verfügung stehen. Die Zeit zwischen der Ausführung dieser beiden Befehle wurde vorher als Wartezeit verschwendet und kann nun für andere Berechnungen genutzt werden. Voraussetzung ist, dass die Wartezeit groß genug ist und der zusätzliche Aufwand für die Umsetzung der beiden Befehle gering ist. In einer testweisen Implementierung war die Implementierung mit Wartezeit schneller als die mit getrennten Lesebefehlen. Die Gründe dafür sind, dass die Auftei-

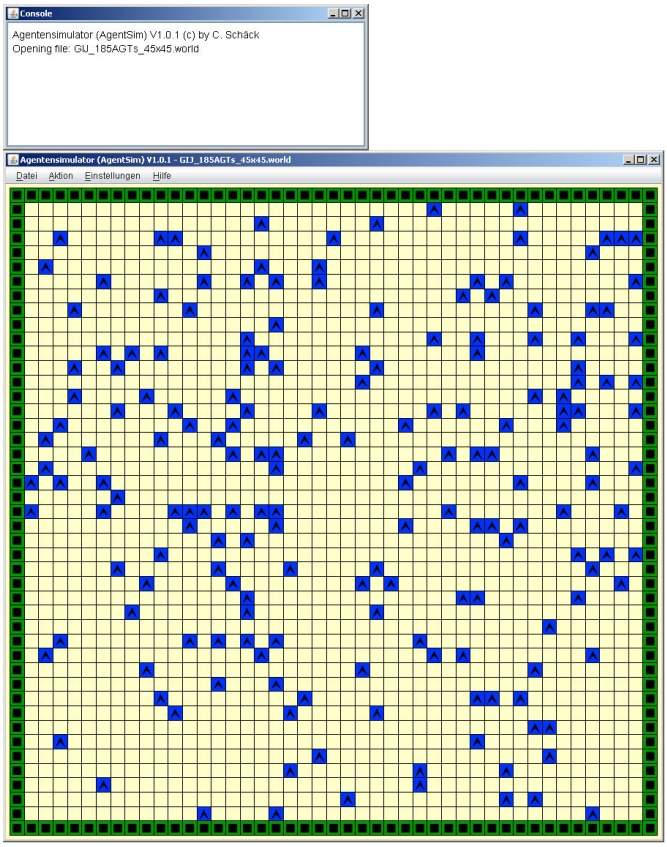


Abbildung 9.1: Oberfläche des Agentensimulators

lung der Lesebefehle zu zeitintensiv ist und einen zusätzlichen Wechsel zwischen Hardware und Software erfordert. Des weiteren ist es, abhängig von der Zellregel, nicht immer möglich, die Wartezeit mit sinnvollen Berechnungen zu füllen. Ist der weitere Ablauf der Zellregel abhängig von dem angeforderten Datenwert, kann keine der später folgenden Berechnungen vorgezogen werden.

Die Auswertung wurde auf der MPA durchgeführt. Hier stehen bis zu 16 Verarbeitungseinheiten zur Verfügung. Auf Grund der schlechten Ergebnisse wurde auf eine detaillierte Auswertung verzichtet. Es ist durchaus möglich, dass sich dieser Ansatz erst für eine noch größere Anzahl an Verarbeitungseinheiten lohnt, da hier auf Grund des komplexeren Netzwerks die Wartezeit entsprechend groß ist.

9.4 Bitonisches Mischen und Sortieren auf verschiedenen Verbindungsnetzwerken

Für die effiziente Multi-Agenten-Simulation wurden weitere Netzwerke für verschiedene Hardwarearchitekturen evaluiert. Diese Netzwerke wurden zusätzlich für das Bitonische Mischen (Tabelle 9.1) und das Bitonische Sortieren (Tabelle 9.2) ausgewertet und bieten damit eine gute Vergleichsgrundlage. Die Auswertung fand auf der Hardwarearchitektur aus Tabelle 6.14 auf dem Cyclone II statt. Um die Leistung weiter zu steigern, wurde für die Ausführung die `AUTO_READ` Funktion für alle Lesezugriffe verwendet. Dies hat für das Bitonische Mischen den Vorteil, dass der Übergang von externen zu internen Lesezugriffen nicht mehr in Software be-

stimmt werden muss, sondern von der Hardware ohne zusätzliche Taktzyklen während des Lesezugriffes bestimmt wird. Für das Bitonische Sortieren ergibt sich ein noch größerer Vorteil, da vorher immer externe Lesezugriffe durchgeführt werden mussten, auch wenn diese nicht nötig waren.

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
-	1	781.567	5,58	-	4.035
ONP	4	206.595	1,50	3,72	14.994
ONP	8	106.445	1,15	4,87	19.652
ONP	16	55.295	0,78	7,15	28.861
RN	4	208.367	1,85	3,01	12.163
RN	8	109.568	1,22	4,59	18.504
RN	16	60.628	0,81	6,91	27.868
RNFF	4	208.160	1,85	3,02	12.176
RNFF	8	106.921	1,17	4,79	19.314
RNFF	16	57.070	0,77	7,22	29.136
BDPA	4	203.485	1,81	3,09	12.454
BDPA	8	102.713	1,10	5,10	20.562
BDPA	16	71.954	0,96	5,82	23.481
BRRA	4	205.520	1,87	2,99	12.057
BRRA	8	104.083	1,16	4,83	19.479
BRRA	16	57.206	0,82	6,83	27.566

Tabelle 9.1: Auswertung des Bitonischen Mischens auf der MPA auf einem Cyclone II FPGA für verschiedene Netzwerke: Omeganetzwerk mit FIFO-Shuffleelementen (ONP), Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA) und Busnetzwerk mit Round-Robin Arbitrierung (BRRA).

Netzwerk	p	Taktzyklen	Ausführungszeit [ms]	Speedup	CUR $\left[\frac{1}{ms}\right]$
-	1		24,86	-	5.437
RN	4	1.275.804	9,59	2,59	14.090
RN	8	653.634	5,45	4,56	24.815
RN	16	346.466	3,65	6,82	37.062
RNFF	4	1.274.851	9,59	2,59	14.101
RNFF	8	650.031	5,42	4,59	24.952
RNFF	16	338.567	3,56	6,98	37.927
BDPA	4	1.269.813	9,55	2,60	14.157
BDPA	8	640.944	5,34	4,65	25.306
BDPA	16	371.192	3,91	6,36	34.593
BRRA	4	1.271.838	9,56	2,60	14.134
BRRA	8	649.528	5,41	4,59	24.972
BRRA	16	335.723	3,53	7,03	38.248

Tabelle 9.2: Auswertung des Bitonischen Sortierens auf der MPA auf einem Cyclone II FPGA für verschiedene Netzwerke: Ringnetzwerk (RN), Ringnetzwerk mit Forwarding (RNFF), Busnetzwerk mit dynamischer Arbitrierung (BDPA) und Busnetzwerk mit Round-Robin Arbitrierung (BRRA).

9.5 Realisierungsaufwand der DAMA für drei Blöcke auf einem FPGA

Der Realisierungsaufwand der DAMA für drei Blöcke ist in Tabelle 9.3 dargestellt.

Netzwerk	p	LEs	Netzwerk		Speicherbits	Register	max. Takt [MHz]
			LEs	Register			
-	1	3.784	-	-	154.336	2.091	131,25
RNFF	2	7.242	361	166	185.760	3.844	133,33
RNFF	4	13.835	655	266	248.608	7.098	100,00
RNFF	8	26.948	1.253	466	374.304	13.586	90,00
RNFF	16	53.351	2.511	866	625.696	26.575	70,84

Tabelle 9.3: Realisierungsaufwand der agentenbasierten speicheroptimierten GCA-Architektur auf einem Cyclone II FPGA mit **3** Blöcken für das Ringnetzwerk mit Forwarding (RNFF)

Die Umsetzung der Zellregel der Agenten mit Informationsaustausch ist im Quellcode 9.3 dargestellt.

```
1  int main()
2  {
3   int g, AGENT_NR, AGENT_BLOCK0, AGENT_BLOCK1, NEW_AGENT_ADR, TEMP_AGENT, cond;
4
5   CI(NEXTGEN,0,NXTG_KEEP);
6   CI(NEXTGEN,0,NXTG_KEEP);
7
8   for(g=0;g<GENS;g++)
9   {
10   AGENT_NR=0;                                    //Agentenzähler initialisieren
11   while(CI(GET_NXT_AGT,AGENT_NR,0)!=0)//Prüfe ob weitere Agenten vorhanden sind
12   {
13    AGENT_BLOCK0 = CI(RD_NXT_AGT_B0,0,0);                    //Lade Zellindex i
14    AGENT_BLOCK1 = CI(RD_NXT_AGT_B1,0,0);                        //Lade Zelltyp
15
16    if(AGENT_BLOCK1>CELL_BLOCK)                             //Beweglicher Agent
17    {
18     switch(AGENT_BLOCK1)                                 //Berechne Zieladresse
19     {
20      case CELL_AGENT_N: NEW_AGENT_ADR = AGENT_BLOCK0-MAXY; break;
21      case CELL_AGENT_S: NEW_AGENT_ADR = AGENT_BLOCK0+MAXY; break;
22      case CELL_AGENT_E: NEW_AGENT_ADR = AGENT_BLOCK0+1; break;
23      case CELL_AGENT_W: NEW_AGENT_ADR = AGENT_BLOCK0-1; break;
24     }
25                                      //Prüfe Nachbarzellen mit Hardwarefunktion
26     if(CI(RD_AGT_ADR,NEW_AGENT_ADR,0)==-1)
27     {cond=CI(CHECK_CELL,NEW_AGENT_ADR,MAXY);}
28     else
29     {cond=999;}
30
31     if(cond>1)
32     {                                                 //Bewegung nicht möglich
33      switch(AGENT_BLOCK1) //Berechnung der Frontzelle zum Informationsaustausch
34      {
```

```
35        case CELL_AGENT_N: TEMP_AGENT=AGENT_BLOCK0-MAXY; break;
36        case CELL_AGENT_S: TEMP_AGENT=AGENT_BLOCK0+MAXY; break;
37        case CELL_AGENT_E: TEMP_AGENT=AGENT_BLOCK0+1; break;
38        case CELL_AGENT_W: TEMP_AGENT=AGENT_BLOCK0-1; break;
39       }
40
41       TEMP_AGENT=CI(RD_AGT_ADR,TEMP_AGENT,0);                //Prüfe Frontzelle,
42       if(TEMP_AGENT!=-1)                                      //Agent vorhanden?
43       {                                                     //Frontzelle ist Agent
44        TEMP_AGENT=CI(RD_B2,TEMP_AGENT,0);          //Lade Information des Agenten
45        if(CI(RD_NXT_AGT_B2,0,0)>TEMP_AGENT)             //Übernehme größeren Wert
46         CI(WR_B2,0,CI(RD_NXT_AGT_B2,0,0));
47        else
48         CI(WR_B2,0,TEMP_AGENT);
49       }
50       else
51       {                                                 //Frontzelle ist kein Agent
52        CI(WR_B2,0,CI(RD_NXT_AGT_B2,0,0));               //Behalte Information bei
53       }
54
55       if(AGENT_BLOCK1==CELL_AGENT_W)
56        AGENT_BLOCK1=CELL_AGENT_N;
57       else
58        AGENT_BLOCK1+=2;
59       CI(WR_B0,0,AGENT_BLOCK0);          //Agent an berechnete Position speichern
60       CI(WR_B1,0,AGENT_BLOCK1);
61       CI(WR_SEND,AGENT_BLOCK0,0);
62      }
63      else
64      {                                                       //Bewegung möglich
65       CI(WR_B2,0,CI(RD_NXT_AGT_B2,0,0));               //Behalte Information bei
66
67       switch(AGENT_BLOCK1)
68       {
69        case CELL_AGENT_N: AGENT_BLOCK1=CELL_AGENT_E; break;
70        case CELL_AGENT_S: AGENT_BLOCK1=CELL_AGENT_W; break;
71        case CELL_AGENT_E: AGENT_BLOCK1=CELL_AGENT_N; break;
72        case CELL_AGENT_W: AGENT_BLOCK1=CELL_AGENT_S; break;
73       }
74       CI(WR_B0,0,NEW_AGENT_ADR);         //Agent an berechnete Position speichern
75       CI(WR_B1,0,AGENT_BLOCK1);
76       CI(WR_SEND,NEW_AGENT_ADR,0);
77      }
78
79     }
80     else
81     {
82      CI(WR_B0,0,AGENT_BLOCK0);      //Andere Zelltypen an alte Position speichern
83      CI(WR_B1,0,AGENT_BLOCK1);
84      CI(WR_SEND,AGENT_BLOCK0,0);
85     }
86
87     AGENT_NR++;                                              //Agnetenzähler
88    }
89
```

```
90   CI(NEXTGEN,0,NXTG_DELETE);                          //Generationssynchronisation
91  }
92 return 0;}
```

Quellcode 9.3: Zellregel der Agentenanwendung mit Informationsverteilung

9.6 Quellcodes der Kraftberechnung des Mehrkörperproblems

Die Beschreibung des Mehrkörperproblems in GCA-L, übernommen von Johannes Jendrsczok, ist im Quellcode 9.4 aufgeführt [JHL09, JEH08a]. Der, mit dem von Johannes Jendrsczok entwickelten Compiler, daraus generierte C-Code ist im Quellcode 9.5 dargestellt. Diese Version wurde dann von mir weiter optimiert. Die optimierte Version ist im Quellcode 9.6 dargestellt.

```
1  program
2   cellstructure = m, X, Y, Z, a1, a2, a3;
3   neighborhood = neighbor;
4
5   celltype cell=float, float, float, float, float, float, float;
6   cell C[16];
7
8   parameter N=16;
9   central index;
10  C.X = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
11  C.Y = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
12  C.Z = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
13  C.m = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
14  C.a1 = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
15  C.a2 = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
16  C.a3 = 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0,16.0;
17
18  for index = 1 to N do
19   foreach C with
20    neighbor = &C[(index+1)%N]
21   do
22    deltaX = neighbor.X - X;
23    deltaY = neighbor.Y - Y;
24    deltaZ = neighbor.Z - Z;
25    rq=deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ;
26    r=sqrt(rq);
27    temp = neighbor.m / (r * rq);
28    if(i != (index+1)%N)then
29     a1 <= a1 + temp * deltaX;
30     a2 <= a2 + temp * deltaY;
31     a3 <= a3 + temp * deltaZ;
32    endif
33   endforeach
34  endfor
35 endprogram
```

Quellcode 9.4: Zellregel des Mehrkörperproblems in GCA-L

```
1  int main(){
2   int i, id;
3   int temp_i; int temp_j;
```

```
4   int LOAD_I; int LOAD_J;
5   int NB_0;
6   int NB_1;
7   int NB_2;
8   int NB_3;
9   int VAR_index;
10  int deltaX;
11  int deltaY;
12  int deltaZ;
13  int rq;
14  int r;
15  int temp;
16
17  CI(NEXTGEN,0,0);
18  CI(NEXTGEN,0,0);
19
20  VAR_index = 1;
21  for(VAR_index=1;VAR_index<=256;VAR_index++){
22   //FOREACH
23   for(i=0;i<LOCL_CELLS;i++){
24    id = i + SC;
25    LOAD_I = (id - 0) / 1;
26    LOAD_J = (id - 0) % 1;
27
28    //h  block
29    temp_i = ((VAR_index + 1 ) % 256);
30    temp_j = 0 ;
31    NB_0   = CI(RD_B0, ((temp_i * 1)+ temp_j + 0), 0);
32    NB_1   = CI(RD_B1, ((temp_i * 1)+ temp_j + 0), 0);
33    NB_2   = CI(RD_B2, ((temp_i * 1)+ temp_j + 0), 0);
34    NB_3   = CI(RD_B3, ((temp_i * 1)+ temp_j + 0), 0);
35
36    //d-p block
37    deltaX = fsub(NB_1, CI(RD_B1,id,0));
38    deltaY = fsub(NB_2, CI(RD_B2,id,0));
39    deltaZ = fsub(NB_3, CI(RD_B3,id,0));
40
41    rq     = fadd(fadd(fmult(deltaX, deltaX),
42                  fmult(deltaY, deltaY)),
43                  fmult(deltaZ, deltaZ));
44    r      = fsqrt(rq);
45    temp   = fdiv(NB_0, fmult(r, rq));
46
47    CI(WR_B0, id, CI(RD_B0,id,0));//Masse
48    CI(WR_B1, id, CI(RD_B1,id,0));//X
49    CI(WR_B2, id, CI(RD_B2,id,0));//Y
50    CI(WR_B3, id, CI(RD_B3,id,0));//Z
51
52    if(id != ((VAR_index+1) % 256)){
53     CI(WR_B4, id, fadd(fmult(temp, deltaX), CI(RD_B4,id,0)));//a1
54     CI(WR_B5, id, fadd(fmult(temp, deltaY), CI(RD_B5,id,0)));//a2
55     CI(WR_B6, id, fadd(fmult(temp, deltaZ), CI(RD_B6,id,0)));//a3
56    }
57    else{
58     CI(WR_B4, id, CI(RD_B4,id,0));//a1
```

```
59      CI(WR_B5, id, CI(RD_B5,id,0));//a2
60      CI(WR_B6, id, CI(RD_B6,id,0));//a3
61     }
62    }
63   CI(NEXTGEN,0,0);
64   }
65  return 0;}
```

Quellcode 9.5: Zellregel des Mehrkörperproblems (Übersetzung aus GCA-L)

```
1   int main(){
2    int id, VAR_index;
3    int deltaX, deltaY, deltaZ;
4    int rq, r, temp;
5
6    CI(NEXTGEN,0,0);
7    CI(NEXTGEN,0,0);
8
9    for(id=SC;id<(LOCL_CELLS+SC);id++){
10    CI(WR_B0, id, CI(RD_B0,id,0));//Masse
11    CI(WR_B1, id, CI(RD_B1,id,0));//X
12    CI(WR_B2, id, CI(RD_B2,id,0));//Y
13    CI(WR_B3, id, CI(RD_B3,id,0));//Z
14   }
15
16   for(VAR_index=0;VAR_index<256;VAR_index++){
17    //FOREACH
18    for(id=SC;id<(LOCL_CELLS+SC);id++){
19
20     //d-p block
21     deltaX = fsub(CI(RD_B1, VAR_index, 0), CI(RD_B1,id,0));
22     deltaY = fsub(CI(RD_B2, VAR_index, 0), CI(RD_B2,id,0));
23     deltaZ = fsub(CI(RD_B3, VAR_index, 0), CI(RD_B3,id,0));
24
25     rq     = fadd(fadd(fmult(deltaX, deltaX),
26                  fmult(deltaY, deltaY)),
27                  fmult(deltaZ, deltaZ));
28     r      = fsqrt(rq);
29     temp   = fdiv(CI(RD_B0, VAR_index, 0), fmult(r, rq));
30
31     if(id != VAR_index){
32      CI(WR_B4, id, fadd(fmult(temp, deltaX), CI(RD_B4,id,0)));//a1
33      CI(WR_B5, id, fadd(fmult(temp, deltaY), CI(RD_B5,id,0)));//a2
34      CI(WR_B6, id, fadd(fmult(temp, deltaZ), CI(RD_B6,id,0)));//a3
35     }
36     else{
37      CI(WR_B4, id, CI(RD_B4,id,0));//a1
38      CI(WR_B5, id, CI(RD_B5,id,0));//a2
39      CI(WR_B6, id, CI(RD_B6,id,0));//a3
40     }
41    }
42    CI(NEXTGEN,0,0);
43   }
44  return 0;}
```

Quellcode 9.6: Zellregel des Mehrkörperproblems (optimiert)

9.7 Synthesesoftware und FPGAs

Im Folgenden werden Details zu der verwendeten Synthesesoftware und den verwendeten FPGAs aufgelistet.

9.7.1 Synthesesoftware

Die verschiedenen Architekturen (Kapitel 6) wurden mit der Synthesesoftware Quartus II V8.1 [Altg] von Altera [Alt10b] synthetisiert. In der Zwischenzeit gab es mehrere neue Versionen, welche für die Umsetzung der Architekturen allerdings nicht verwendet wurden, um eine möglichst gute Vergleichbarkeit zwischen den verschiedenen Architekturen zu erhalten. Für die Auswertungen auf dem Stratix V musste die Version 10.0 verwendet werden, da das Stratix V FPGA erst in neueren Versionen zur Verfügung stand. Neuere Versionen der Synthesesoftware erzielen eine höhere Taktfrequenz und benötigen gleichzeitig weniger Logikelemente.

System-on-a-Programmable-Chip

Der System-on-a-Programmable-Chip(SOPC)-Builder ist Teil der Quartus Synthesesoftware [Altg]. Über dieses Tool werden die NIOS II Prozessoren und deren zugehörige Programmspeicher generiert. Der SOPC-Builder ist nicht für eine große Anzahl an Prozessoren ausgelegt, was die Realisierung der Multiprozessorarchitekturen erschwerte.

9.7.2 Verwendete Field Programmable Gate Arrays

Für die Auswertung auf realer Hardware kamen zwei Field Programmable Gate Arrays (FPGA) zum Einsatz.

Cyclone II FPGA

Bei dem Cyclone II FPGA [Alta] handelt es sich um ein Low-Cost FPGA mit dennoch guter Leistung. Es ist in verschiedenen Größen erhältlich. Für die Architekturen wurde das größte Cyclone II (EP2C70) verwendet. Dieses besitzt u. a.:

- 68.416 Logikelemente (LEs)
- 250 M4K Speicherblöcke (entspricht 1.125 Speicherbits)
- 150 integrierte Multiplizierer
- 4 PLLs[1]

[1] Phase Locked Loop

Stratix II FPGA

Bei dem Stratix II FPGA [Alth] handelt es sich um ein High-End FPGA. Dieses FPGA besitzt einen anderen internen Aufbau. Auf dem Stratix II sind größere Schaltungen mit höherer Taktfrequenz realisierbar. Die verwendete Plattform PROCel180-A [GiD07] besitzt ein Stratix II FPGA (EP2S180) mit folgenden Eigenschaften:

- 71.760 ALMs
- insgesamt 9.383.040 Speicherbits
- 384 integrierte Multiplizierer
- 12 PLL*s*[2]

Interner Aufbau des Stratix II FPGAs

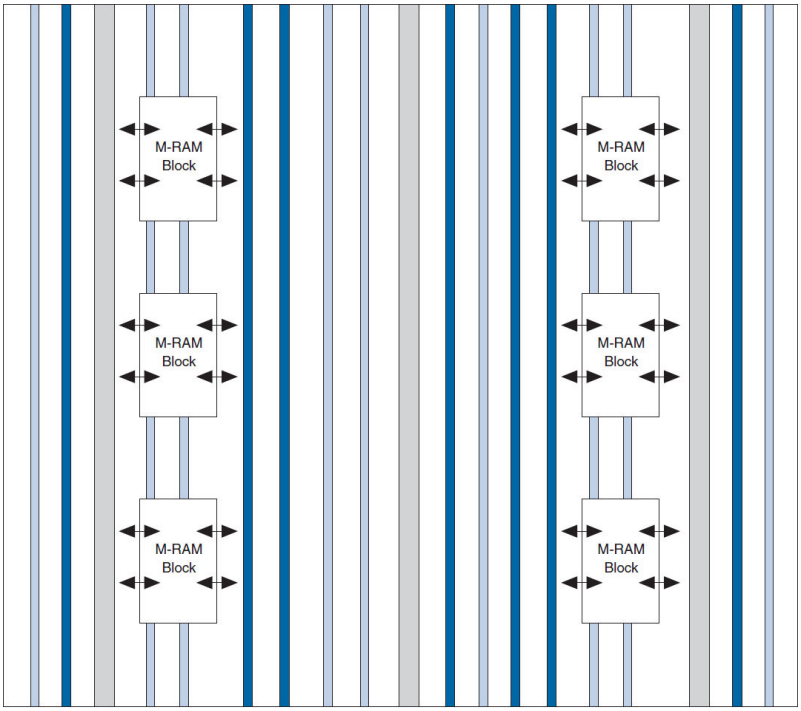


Abbildung 9.2: Interner Aufbau des Stratix II FPGAs ([Alth, Seite 2-37])

Stratix V FPGA

Für die Auswertungen, die mit ModelSim durchgeführt wurden, wurde die Architektur für das Stratix V FPGA (5SGSMB8I4H35I4) [Alti], um die maximal erzielbare Taktfrequenz bestimmen zu können, synthetisiert. Dieses bietet im Vergleich zum Cyclone II und Stratix II noch mehr Platz und erlaubt die Realisierung von größeren Architekturen mit vielen Verarbeitungseinheiten.

[2] Phase Locked Loop

9.8 Implementierung der Verbindungsnetzwerke

Im Folgenden werden die Implementierungen der Netzwerke vorgestellt.

9.8.1 Das Busnetzwerk

Die Implementierung des Busnetzwerks mit den beiden Arbitrierungsvarianten (dynamische Arbitrierung und Round-Robin Arbitrierung).

```
1  module BUS(CLOCK, din, dout, req, busy, nextreq, nextbusy, ain, aout);
2
3  parameter adr_width=3;                                          //Adressbreite
4  parameter data_width=16;                                         //Datenbreite
5  parameter processors=4;                                       //Prozessoranzahl
6  parameter log_processors=2;                                //log_2(Prozessoranzahl)
7  parameter int_adr_width=adr_width+log_processors;          //interne Adressbreite
8
9  /*****************************************************************************/
10
11 input CLOCK;
12
13 input  [((data_width*processors)-1):0]    din;
14 output [((data_width*processors)-1):0]    dout;
15
16 input  [((int_adr_width*processors)-1):0] ain;
17 output [((int_adr_width*processors)-1):0] aout;
18
19 input  [processors-1:0]                    req;
20 output [processors-1:0]                    busy;
21 output [processors-1:0]                    nextreq;
22 input  [processors-1:0]                    nextbusy;
23
24 /*****************************************************************************/
25
26 wire [data_width-1:0]    separate_din [processors-1:0];
27 wire [int_adr_width-1:0] separate_ain [processors-1:0];
28
29 wire separate_req                     [processors-1:0];
30 wire separate_nextbusy                [processors-1:0];
31
32 reg separate_nextreq                  [processors-1:0];
33 reg separate_busy                     [processors-1:0];
34
35 wire [data_width-1:0]    DATABUS;
36 wire [int_adr_width-1:0] ADRBUS;
37
38 /*****************************************************************************/
39
40 //Auftrennen und Verbinden der Inputs/Outputs
41 genvar v;
42 generate
43  for(v=0;v<processors;v=v+1)
```

```
 begin:inputs
  assign separate_din[v] = din[((v+1)*data_width)-1:v*data_width];
  assign dout[((v+1)*data_width)-1:v*data_width] = DATABUS;

  assign separate_ain[v] = ain[((v+1)*int_adr_width)-1:v*int_adr_width];
  assign aout[((v+1)*int_adr_width)-1:v*int_adr_width] = ADRBUS;

  assign separate_req[v] = req[v];
  assign separate_nextbusy[v] = nextbusy[v];

  assign nextreq[v] = separate_nextreq[v];
  assign busy[v] = separate_busy[v];
 end
endgenerate

assign DATABUS=separate_din[separate_ain[position][int_adr_width-1:adr_width]];
assign ADRBUS=separate_ain[position];

//Ansteuerung der request und busy Signale
generate
for(v=0;v<processors;v=v+1)
 begin:reqbusy
  always@*
  begin
   if(v==separate_ain[position][int_adr_width-1:adr_width])
    separate_nextreq[v] <= separate_req[position];
   else
    separate_nextreq[v] <= 1'b0;

   if(position==v)
    separate_busy[v] <= separate_nextbusy
     [separate_ain[position][int_adr_width-1:adr_width]];
   else
    separate_busy[v] <= 1'b1;
  end
 end
endgenerate

/*****************************************************************************/
//Round-Robin Busmaster
`ifdef ROUNDROBIN
reg FSM;
reg [log_processors-1:0] position;

//Busmaster, steuert den Zugriff auf den gemeinsamen Bus
always@(posedge CLOCK)
begin
 case(FSM)
 'h0:
 begin
  if(separate_req[position])
   FSM <= 'h1;
  else
  begin
   FSM <= 'h0;
```

```
99    position <= position+1;
100    end
101   end
102
103   'h1:
104   begin
105    if(~separate_nextbusy[separate_ain[position][int_adr_width-1:adr_width]])
106    begin
107     FSM <= 'h0;
108     position <= position+1;
109    end
110    else
111     FSM <= 'h1;
112   end
113   endcase
114  end
115
116  /*************************************************************************/
117  `else
118  /*************************************************************************/
119  //Busmaster mit dynamischer Arbitrierung
120
121  reg [log_processors-1:0] value [processors-1:0];
122
123  //Erzeuge Indizes für die Requestleitungen
124  generate
125  for(v=0;v<processors;v=v+1)
126   begin:idx
127    always@*
128    begin
129     value[v] <= separate_req[v]?v:0;
130    end
131   end
132  endgenerate
133
134  reg [log_processors:0] vergleichsstufe [processors/2][processors];
135
136  //Generiere Vergleicherbaum
137  genvar w;
138  generate
139   for(w=0;w<log_processors;w=w+1)
140   begin:stufe
141    for(v=0;v<processors;v=v+2)
142    begin:WerteInStufeW
143     always@*
144     begin
145      if(w==0)
146       vergleichsstufe[w][v/2] <= (value[v]>value[v+1]) ? value[v] : value[v+1];
147      else
148       vergleichsstufe[w][v/2] <=
149        (vergleichsstufe[w-1][v] > vergleichsstufe[w-1][v+1]) ?
150         vergleichsstufe[w-1][v] : vergleichsstufe[w-1][v+1];
151     end
152    end
153   end
```

```
154 endgenerate
155
156
157 reg FSM;
158 reg [log_processors-1:0] position;
159
160 //Busmaster, steuert den Zugriff auf den gemeinsamen Bus
161 always@(posedge CLOCK)
162 begin
163  case(FSM)
164  'h0:
165  begin
166   if(separate_req[position])
167    FSM <= 'h1;
168   else
169   begin
170    FSM <= 'h0;
171    position <= vergleichsstufe[log_processors-1][0];
172   end
173  end
174
175  'h1:
176  begin
177   if(~separate_nextbusy[separate_ain[position][int_adr_width-1:adr_width]])
178    FSM <= 'h0;
179   else
180    FSM <= 'h1;
181  end
182  endcase
183 end
184 /*****************************************************************************/
185 `endif
186
187 endmodule
```

Quellcode 9.7: Implementierung des Busnetzwerks mit Round-Robin und dynamischer Arbitrierung

9.8.2 Das Omeganetzwerk

Die Implementierung des Omeganetzwerks basierend auf den Vorarbeiten von [Hee07]. Über `REGISTERED_NET` kann die neu hinzugefügte Registerstufe in der Mitte des Omeganetzwerks aktiviert werden.

```
1 module OMEGA(CLOCK, din, dout, req, busy, nextreq, nextbusy, ain, aout);
2
3 parameter adr_width=3;                                          //Adressbreite
4 parameter data_width=16;                                         //Datenbreite
5 parameter processors=4;                                       //Prozessoranzahl
6 parameter log_processors=2;                                 //log_2(Prozessoranzahl)
7 parameter int_adr_width=adr_width+log_processors;          //interne Adressbreite
8
```

```
 9  /*****************************************************************************/
10
11  input CLOCK;
12
13  input  [((data_width*processors)-1):0]    din;
14  output [((data_width*processors)-1):0]    dout;
15
16  input  [((int_adr_width*processors)-1):0] ain;
17  output [((int_adr_width*processors)-1):0] aout;
18
19  input  [processors-1:0] req;
20  output [processors-1:0] busy;
21  output [processors-1:0] nextreq;
22  input  [processors-1:0] nextbusy;
23
24  /*****************************************************************************/
25
26  wire   [data_width-1:0] s_din [(log_processors-1):0][(processors-1):0],
27   s_dout [(log_processors-1):0][(processors-1):0];
28
29  wire   [int_adr_width-1:0] s_ain [(log_processors-1):0][(processors-1):0],
30   s_aout [(log_processors-1):0][(processors-1):0];
31
32  wire   [data_width-1:0]    omega_inputs     [(processors-1):0];
33  wire   [int_adr_width-1:0] omega_inputs_adr [(processors-1):0];
34
35
36  wire omega_inputs_req                       [(processors-1):0];
37  wire omega_inputs_nextbusy                  [(processors-1):0];
38
39  wire s_req      [(log_processors-1):0]      [(processors-1):0];
40  wire s_nextbusy [(log_processors-1):0]      [(processors-1):0];
41
42  wire s_nextreq [(log_processors-1):0]       [(processors-1):0];
43  wire s_busy    [(log_processors-1):0]       [(processors-1):0];
44
45  /*****************************************************************************/
46
47  reg reg_req                     [(log_processors-1):0][(processors-1):0];
48  reg reg_nextbusy                [(log_processors-1):0][(processors-1):0];
49  reg [data_width-1:0] reg_din    [(log_processors-1):0][(processors-1):0];
50  reg [int_adr_width-1:0] reg_ain [(log_processors-1):0][(processors-1):0];
51
52  /*****************************************************************************/
53
54  genvar i,j,k;
55
56  generate
57   for(j=0;j<log_processors;j=j+1)
58   begin:stage
59    for(k=0;k<(processors/2);k=k+1)
60    begin:shuffle
61
62      shuffle #(.data_width(data_width),
63      .stage(j),
```

```
64      .cid_width(log_processors),
65      .mem_width(adr_width))
66
67    net(.req0(s_req[j][2*k]),
68     .req1(s_req[j][2*k+1]),
69     .ain0(s_ain[j][2*k]),
70     .ain1(s_ain[j][2*k+1]),
71     .aout0(s_aout[j][2*k]),
72     .aout1(s_aout[j][2*k+1]),
73     .busy0(s_busy[j][2*k]),
74     .busy1(s_busy[j][2*k+1]),
75     .next_req0(s_nextreq[j][2*k]),
76     .next_req1(s_nextreq[j][2*k+1]),
77     .next_busy0(s_nextbusy[j][2*k]),
78     .next_busy1(s_nextbusy[j][2*k+1]),
79     .din0(s_din[j][2*k]),
80     .din1(s_din[j][2*k+1]),
81     .dout0(s_dout[j][2*k]),
82     .dout1(s_dout[j][2*k+1])
83      );
84    end
85   end
86  endgenerate
87
88  //Beschaltung der Stufen
89  generate
90   for(j=0;j<(log_processors-1);j=j+1)
91   begin:inner
92    for(k=0;k<(processors/2);k=k+1)
93    begin:inner1
94
95     `ifdef REGISTERED_NET
96      if(!(log_processors/2==(j+1)))
97      begin
98       assign s_req[j+1][2*k] = s_nextreq[j][k];
99       assign s_req[j+1][2*k+1] = s_nextreq[j][k+(processors/2)];
100
101      assign s_nextbusy[j][k] = s_busy[j+1][2*k];
102      assign s_nextbusy[j][k+(processors/2)] = s_busy[j+1][2*k+1];
103
104      assign s_din[j][k] = s_dout[j+1][2*k];
105      assign s_din[j][k+(processors/2)] = s_dout[j+1][2*k+1];
106
107      assign s_ain[j+1][2*k] = s_aout[j][k];
108      assign s_ain[j+1][2*k+1] = s_aout[j][k+(processors/2)];
109     end
110     else
111     begin
112      always@(posedge CLOCK)
113      begin
114       if(~s_busy[j+1][2*k] | ~reg_nextbusy[j][k])
115        reg_req[j+1][2*k] <= 1'b0;
116       else
117        reg_req[j+1][2*k] <= s_nextreq[j][k];
118
```

```
119       if(~s_busy[j+1][2*k+1] | ~reg_nextbusy[j][k+(processors/2)])
120        reg_req[j+1][2*k+1] <= 1'b0;
121       else
122        reg_req[j+1][2*k+1] <= s_nextreq[j][k+(processors/2)];
123
124       reg_nextbusy[j][k] <= s_busy[j+1][2*k];
125       reg_nextbusy[j][k+(processors/2)] <= s_busy[j+1][2*k+1];
126
127       reg_din[j][k] <= s_dout[j+1][2*k];
128       reg_din[j][k+(processors/2)] <= s_dout[j+1][2*k+1];
129
130       reg_ain[j+1][2*k] <= s_aout[j][k];
131       reg_ain[j+1][2*k+1] <= s_aout[j][k+(processors/2)];
132      end
133
134      assign s_req[j+1][2*k] = reg_req[j+1][2*k];
135      assign s_req[j+1][2*k+1] = reg_req[j+1][2*k+1];
136
137      assign s_nextbusy[j][k]=reg_nextbusy[j][k];
138      assign s_nextbusy[j][k+(processors/2)]=reg_nextbusy[j][k+(processors/2)];
139
140      assign s_din[j][k] = reg_din[j][k];
141      assign s_din[j][k+(processors/2)] = reg_din[j][k+(processors/2)];
142
143      assign s_ain[j+1][2*k] = reg_ain[j+1][2*k];
144      assign s_ain[j+1][2*k+1] = reg_ain[j+1][2*k+1];
145     end
146    `else
147     assign s_req[j+1][2*k] = s_nextreq[j][k];
148     assign s_req[j+1][2*k+1] = s_nextreq[j][k+(processors/2)];
149
150     assign s_nextbusy[j][k] = s_busy[j+1][2*k];
151     assign s_nextbusy[j][k+(processors/2)] = s_busy[j+1][2*k+1];
152
153     assign s_din[j][k] = s_dout[j+1][2*k];
154     assign s_din[j][k+(processors/2)] = s_dout[j+1][2*k+1];
155
156     assign s_ain[j+1][2*k] = s_aout[j][k];
157     assign s_ain[j+1][2*k+1] = s_aout[j][k+(processors/2)];
158    `endif
159   end
160  end
161 endgenerate
162
163 //Generiere Netz aus übergebenem
164 generate
165  for(i=0;i<processors;i=i+1)
166  begin:omega
167   assign omega_inputs[i] = din[((i+1)*data_width)-1:i*data_width];
168   assign omega_inputs_adr[i] = ain[((i+1)*int_adr_width)-1:i*int_adr_width];
169
170   assign omega_inputs_req[i] = req[i];
171   assign omega_inputs_nextbusy[i] = nextbusy[i];
172  end
173 endgenerate
```

```
174
175 //Eingänge
176 generate
177  for(k=0;k<(processors/2);k=k+1)
178  begin:left
179   assign s_ain[0][2*k] = omega_inputs_adr[k];
180   assign s_ain[0][2*k+1] = omega_inputs_adr[k+(processors/2)];
181
182   assign s_req[0][2*k] = omega_inputs_req[k];
183   assign s_req[0][2*k+1] = omega_inputs_req[k+(processors/2)];
184
185   assign busy[k] = s_busy[0][2*k];
186   assign busy[k+(processors/2)] = s_busy[0][2*k+1];
187
188   assign dout[(k*data_width) +: data_width] = s_dout[0][2*k];
189   assign dout[(k+(processors/2))*data_width +: data_width] = s_dout[0][2*k+1];
190  end
191 endgenerate
192
193 generate
194  for(k=0;k<processors;k=k+1)
195  begin:lefti
196   assign s_nextbusy[log_processors-1][k] = omega_inputs_nextbusy[k];
197  end
198 endgenerate
199
200 //Ausgänge
201 generate
202  for(k=0;k<processors;k=k+1)
203  begin:right
204   assign aout[((k+1)*int_adr_width)-1:k*int_adr_width]
205          = s_aout[log_processors-1][k];
206   assign nextreq[(k+1)-1:k] = s_nextreq[log_processors-1][k];
207   assign s_din[log_processors-1][k] = omega_inputs[k];
208  end
209 endgenerate
210
211 endmodule
```

Quellcode 9.8: Implementierung des Omeganetzwerks mit optionaler Registerstufe

9.8.3 Das Ringnetzwerk mit und ohne Forwarding

Die Implementierung des Ringnetzwerks. Über den Parameter `SKIP_ONE` kann das Forwarding aktiviert werden.

```
1 module RING_FF(CLOCK, din, dout, req, busy, nextreq, nextbusy, ain, aout);
2
3 parameter adr_width=3;                                          //Adressbreite
4 parameter data_width=16;                                         //Datenbreite
5 parameter processors=4;                                       //Prozessoranzahl
6 parameter log_processors=2;                                 //log_2(Prozessoranzahl)
7 parameter int_adr_width=adr_width+log_processors;          //interne Adressbreite
```

```
 8
 9  /*****************************************************************************/
10
11  input CLOCK;
12
13  input  [((data_width*processors)-1):0]    din;
14  output [((data_width*processors)-1):0]    dout;
15
16  input  [((int_adr_width*processors)-1):0] ain;
17  output [((int_adr_width*processors)-1):0] aout;
18
19  input  [processors-1:0] req;
20  output [processors-1:0] busy;
21  output [processors-1:0] nextreq;
22  input  [processors-1:0] nextbusy;
23
24  /*****************************************************************************/
25
26  wire RING_OCCUPIED                        [processors-1:0];
27
28  reg FSM                                   [processors-1:0];
29  reg block_request                         [processors-1:0];
30
31  reg [data_width-1:0]    RING_DATA         [processors-1:0];
32  reg [int_adr_width-1:0] RING_ADR          [processors-1:0];
33  reg RING_REQ                              [processors-1:0];
34  reg RING_ACK                              [processors-1:0];
35
36  `ifdef SKIP_ONE
37  reg [data_width-1:0]    FF_RING_DATA;
38  reg [int_adr_width-1:0] FF_RING_ADR;
39  reg FF_RING_REQ;
40  reg FF_RING_ACK;
41  reg FF_WORKING;
42  reg FF_FORWARD;
43  reg [31:0] FF_COUNTER;
44  `endif
45
46  reg memory_req                            [processors-1:0];
47  reg memory_working                        [processors-1:0];
48
49  reg [int_adr_width-1:0] memory_adr        [processors-1:0];
50  reg [int_adr_width-1:adr_width] reply_id  [processors-1:0];
51
52  reg buffer_busy                           [processors-1:0];
53
54  /*****************************************************************************/
55
56  genvar v;
57  generate
58   for(v=0;v<processors;v=v+1)
59   begin:DEF_Signals
60    assign dout[v*data_width +: data_width] = RING_DATA[v];
61    assign aout[v*int_adr_width +: int_adr_width] = memory_adr[v];
62    assign nextreq[v] = memory_req[v];
```

```
63    assign busy[v] = ~(RING_ACK[v] & RING_ADR[v][int_adr_width-1:adr_width]==v);
64
65    assign RING_OCCUPIED[v] = RING_ACK[v] | RING_REQ[v];
66   end
67  endgenerate
68
69  /*****************************************************************************/
70  //Deadlock Detektion
71  `ifdef SKIP_ONE
72  always@(posedge CLOCK)
73  begin
74   if(FF_WORKING)
75   begin
76    if( (FF_RING_DATA==RING_DATA[0]
77        & FF_RING_ADR==RING_ADR[0]
78        & FF_RING_REQ==RING_REQ[0]
79        & FF_RING_ACK==RING_ACK[0])
80    | (FF_RING_DATA==RING_DATA[processors-1]
81        & FF_RING_ADR==RING_ADR[processors-1]
82        & FF_RING_REQ==RING_REQ[processors-1]
83        & FF_RING_ACK==RING_ACK[processors-1]))
84    begin
85     FF_FORWARD <= 1'b0;
86     FF_COUNTER <= 'b0;
87    end
88    else
89     FF_COUNTER <= FF_COUNTER+'b1;
90
91    if(FF_COUNTER==(processors-1))
92    begin
93     FF_WORKING <= 1'b0;
94     FF_FORWARD <= 1'b1;
95    end
96   end
97   else
98   begin
99    if(RING_OCCUPIED[0])
100   begin
101    FF_RING_DATA <= RING_DATA[0];
102    FF_RING_ADR <= RING_ADR[0];
103    FF_RING_REQ <= RING_REQ[0];
104    FF_RING_ACK <= RING_ACK[0];
105    FF_WORKING <= 1'b1;
106    FF_COUNTER <= 1'b0;
107    FF_FORWARD <= 1'b1;
108   end
109   else
110   begin
111    FF_WORKING <= 1'b0;
112    if(FF_COUNTER==(processors-1))
113    begin
114     FF_COUNTER <= 'b0;
115     FF_FORWARD <= 1'b0;
116    end
117    else
```

```
118     begin
119       FF_COUNTER <= FF_COUNTER+'b1;
120       FF_FORWARD <= 1'b1;
121     end
122   end
123  end
124 end
125 `endif
126 /*****************************************************************************/
127
128 integer i;
129 integer from, to;
130 integer bck;
131
132 always@(posedge CLOCK)
133 begin
134  for(i=0;i<processors;i=i+1)
135  begin
136   from=i;
137
138   if(i!=processors-1)
139    to=i+1;
140   else
141    to=0;
142
143   bck=(i-1+processors)%processors;
144
145   case(FSM[from])
146    'h0:
147    begin
148     if(req[from] & ~RING_OCCUPIED[from])
149     begin
150      FSM[from] <= 'h1;
151      block_request[from] <= 1'b1;
152     end
153     else
154     begin
155      FSM[from] <= 'h0;
156      block_request[from] <= 1'b0;
157     end
158    end
159
160    'h1:
161    begin
162     if(~req[from])
163     begin
164      FSM[from] <= 'h0;
165      block_request[from] <= 1'b0;
166     end
167     else
168     begin
169      FSM[from] <= 'h1;
170      block_request[from] <= 1'b1;
171     end
172    end
```

```
173   endcase
174
175   if((RING_OCCUPIED[from] | (req[from] & ~block_request[from]))
176    & memory_working[from] & (~nextbusy[from] | ~buffer_busy[from]))
177    buffer_busy[from] <= 1'b0;
178   else
179    buffer_busy[from] <= 1'b1;
180
181   if(RING_REQ[from])
182   begin
183    if((RING_ADR[from][int_adr_width-1:adr_width]==from)
184     & ~memory_working[from] & buffer_busy[from])
185    begin
186     RING_REQ[to] <= 1'b0;
187     RING_ACK[to] <= 1'b0;
188     memory_req[from] <= 1'b1;
189     memory_working[from] <= 1'b1;
190     memory_adr[from] <= RING_ADR[from];
191     reply_id[from] <= RING_DATA[from][log_processors-1:0];
192    end
193    else
194    begin
195     memory_req[from] <= 1'b0;
196
197     `ifdef SKIP_ONE
198      if(FF_FORWARD & (RING_ADR[from][adr_width +: log_processors]!=to)
199       & ~RING_OCCUPIED[to] & ~(req[to] & ~block_request[to])
200       & ~(memory_working[to] & (~nextbusy[to] | ~buffer_busy[to])))
201      begin
202       RING_REQ[to] <= 1'b0;
203       RING_ACK[to] <= 1'b0;
204      end
205      else
206      begin
207       RING_REQ[to] <= RING_REQ[from];
208       RING_ADR[to] <= RING_ADR[from];
209       RING_ACK[to] <= RING_ACK[from];
210       RING_DATA[to] <= RING_DATA[from];
211      end
212     `else
213      RING_REQ[to] <= RING_REQ[from];
214      RING_ADR[to] <= RING_ADR[from];
215      RING_ACK[to] <= RING_ACK[from];
216      RING_DATA[to] <= RING_DATA[from];
217     `endif
218    end
219   end
220   else
221   begin
222    memory_req[from] <= 1'b0;
223    if(~RING_ACK[from])
224    begin
225     if(req[from] & ~block_request[from])
226     begin
227      RING_ADR[to] <= ain[from*int_adr_width +: int_adr_width];
```

```
228       RING_REQ[to] <= 1'b1;
229       RING_ACK[to] <= 1'b0;
230       RING_DATA[to] <= from;
231      end
232      else
233      begin
234       if(memory_working[from] & (~nextbusy[from] | ~buffer_busy[from]))
235       begin
236        RING_ADR[to][int_adr_width-1:adr_width] <= reply_id[from];
237        RING_REQ[to] <= 1'b0;
238        RING_ACK[to] <= 1'b1;
239        RING_DATA[to] <= din[from*data_width +: data_width];
240        memory_working[from] <= 1'b0;
241       end
242       else
243       begin
244        `ifdef SKIP_ONE
245         if(FF_FORWARD
246         & (RING_ADR[bck][adr_width +: log_processors]!=from)
247         & RING_OCCUPIED[bck]
248         & ~(RING_ADR[bck][int_adr_width-1:adr_width]==bck
249         & ~memory_working[bck] & buffer_busy[bck])
250         & ~(RING_ACK[bck] & RING_ADR[bck][int_adr_width-1:adr_width]==bck))
251         begin
252          RING_ADR[to] <= RING_ADR[bck];
253          RING_REQ[to] <= RING_REQ[bck];
254          RING_ACK[to] <= RING_ACK[bck];
255          RING_DATA[to] <= RING_DATA[bck];
256         end
257         else
258         begin
259          RING_REQ[to] <= 1'b0;
260          RING_ACK[to] <= 1'b0;
261         end
262        `else
263         RING_REQ[to] <= 1'b0;
264         RING_ACK[to] <= 1'b0;
265        `endif
266       end
267      end
268     end
269     else
270     begin
271      `ifdef SKIP_ONE
272       if(FF_FORWARD & (RING_ADR[from][adr_width +: log_processors]!=to)
273        & ~RING_OCCUPIED[to] & ~(req[to] & ~block_request[to])
274        & ~(memory_working[to] & (~nextbusy[to] | ~buffer_busy[to])))
275        RING_ACK[to] <= 1'b0;
276       else
277       begin
278        RING_REQ[to] <= RING_REQ[from];
279        RING_ADR[to] <= RING_ADR[from];
280
281        if(RING_ACK[from] & RING_ADR[from][int_adr_width-1:adr_width]==from)
282         RING_ACK[to] <= 1'b0;
```

```
283       else
284        RING_ACK[to] <= RING_ACK[from];
285
286       RING_DATA[to] <= RING_DATA[from];
287      end
288     `else
289      RING_REQ[to] <= RING_REQ[from];
290      RING_ADR[to] <= RING_ADR[from];
291
292      if(RING_ACK[from] & RING_ADR[from][int_adr_width-1:adr_width]==from)
293       RING_ACK[to] <= 1'b0;
294      else
295       RING_ACK[to] <= RING_ACK[from];
296
297      RING_DATA[to] <= RING_DATA[from];
298     `endif
299    end
300   end
301
302  end
303 end
304
305 endmodule
```

Quellcode 9.9: Implementierung des Ringnetzwerks mit optionalem Forwarding

9.8.4 Das Omeganetzwerk mit FIFO-Shuffleelementen

Die Implementierung des Omeganetzwerks mit FIFO-Shuffleelementen.

```
1  module NET_PIPELINE(CLOCK, din, dout, req, busy, nextreq, nextbusy, ain, aout);
2
3  parameter adr_width=3;                                          //Adressbreite
4  parameter data_width=16;                                         //Datenbreite
5  parameter processors=4;                                       //Prozessoranzahl
6  parameter log_processors=2;                                 //log_2(Prozessoranzahl)
7  parameter int_adr_width=adr_width+log_processors;        //interne Adressbreite
8
9  /*****************************************************************************/
10
11 input CLOCK;
12
13 input  [((data_width*processors)-1):0]    din;
14 output [((data_width*processors)-1):0]    dout;
15
16 input  [((int_adr_width*processors)-1):0] ain;
17 output [((int_adr_width*processors)-1):0] aout;
18
19 input  [processors-1:0] req;
20 output [processors-1:0] busy;
21 output [processors-1:0] nextreq;
22 input  [processors-1:0] nextbusy;
23
```

```
24 /*****************************************************************************/
25
26 wire [data_width-1:0] s_din [(log_processors-1):0][(processors-1):0],
27  s_dout [(log_processors-1):0][(processors-1):0];
28
29 wire [int_adr_width-1:0] s_ain [(log_processors-1):0][(processors-1):0],
30  s_aout [(log_processors-1):0][(processors-1):0];
31
32 wire s_req                          [(log_processors-1):0][(processors-1):0];
33 wire s_ack                          [(log_processors-1):0][(processors-1):0];
34
35 wire s_nextreq                      [(log_processors-1):0][(processors-1):0];
36 wire s_nextack                      [(log_processors-1):0][(processors-1):0];
37
38 wire s_ifull                        [(log_processors-1):0][(processors-1):0];
39 wire s_ofull                        [(log_processors-1):0][(processors-1):0];
40
41 wire s_ifullback                    [(log_processors-1):0][(processors-1):0];
42 wire s_ofullback                    [(log_processors-1):0][(processors-1):0];
43
44 wire [log_processors-1:0] s_cin     [(log_processors-1):0][(processors-1):0];
45 wire [log_processors-1:0] s_cout    [(log_processors-1):0][(processors-1):0];
46
47 wire [log_processors-1:0] s_cbacki  [(log_processors-1):0][(processors-1):0];
48 wire [log_processors-1:0] s_cbacko  [(log_processors-1):0][(processors-1):0];
49
50 reg  [log_processors-1:0] s_cbackibuf [(processors-1):0];
51
52 /*****************************************************************************/
53
54 genvar i,j,k;
55
56 reg FSM [(processors-1):0];
57 wire [processors-1:0] ofullx;
58
59 generate
60  for(j=0;j<log_processors;j=j+1)
61  begin:stage
62   for(k=0;k<(processors/2);k=k+1)
63   begin:shuffle
64
65   pshuffle_in #(
66     .stage(j),
67     .cid_width(log_processors),
68     .mem_width(adr_width))
69   netin(.CLOCK(CLOCK),
70    .req0(s_req[j][2*k]),
71    .req1(s_req[j][2*k+1]),
72    .ain0(s_ain[j][2*k]),
73    .ain1(s_ain[j][2*k+1]),
74    .ifull0(s_ifull[j][2*k]),
75    .ifull1(s_ifull[j][2*k+1]),
76    .ofull0(s_ofull[j][2*k]),
77    .ofull1(s_ofull[j][2*k+1]),
78    .aout0(s_aout[j][2*k]),
```

```
79    .aout1(s_aout[j][2*k+1]),
80    .cin0(s_cin[j][2*k]),
81    .cin1(s_cin[j][2*k+1]),
82    .cout0(s_cout[j][2*k]),
83    .cout1(s_cout[j][2*k+1]),
84    .next_req0(s_nextreq[j][2*k]),
85    .next_req1(s_nextreq[j][2*k+1])
86    );
87
88
89   pshuffle_out #(
90     .stage(j),
91     .cid_width(log_processors),
92     .mem_width(0),
93     .data_width(data_width))
94   netout(.CLOCK(CLOCK),
95    .ack0(s_ack[j][2*k]),
96    .ack1(s_ack[j][2*k+1]),
97    .next_ack0(s_nextack[j][2*k]),
98    .next_ack1(s_nextack[j][2*k+1]),
99    .ain0(s_cbacki[j][2*k]),
100   .ain1(s_cbacki[j][2*k+1]),
101   .aout0(s_cbacko[j][2*k]),
102   .aout1(s_cbacko[j][2*k+1]),
103   .din0(s_din[j][2*k]),
104   .din1(s_din[j][(2*k)+1]),
105   .dout0(s_dout[j][2*k]),
106   .dout1(s_dout[j][2*k+1]),
107   .ifull0(s_ifullback[j][2*k]),
108   .ifull1(s_ifullback[j][2*k+1]),
109   .ofull0(s_ofullback[j][2*k]),
110   .ofull1(s_ofullback[j][2*k+1])
111    );
112   end
113  end
114 endgenerate
115
116 //Beschaltung der Stufen
117 generate
118  for(j=0;j<(log_processors-1);j=j+1)
119  begin:inner
120   for(k=0;k<(processors/2);k=k+1)
121   begin:inner1
122    assign s_req[j+1][2*k] = s_nextreq[j][k];
123    assign s_req[j+1][2*k+1] = s_nextreq[j][k+(processors/2)];
124
125    assign s_cin[j+1][2*k] = s_cout[j][k];
126    assign s_cin[j+1][2*k+1]= s_cout[j][k+(processors/2)];
127
128    assign s_ack[j+1][2*k] = s_nextack[j][k];
129    assign s_ack[j+1][2*k+1] = s_nextack[j][k+(processors/2)];
130
131    assign s_din[j+1][2*k] = s_dout[j][k];
132    assign s_din[j+1][2*k+1] = s_dout[j][k+(processors/2)];
133
```

```
134     assign s_cbacki[j+1][2*k] = s_cbacko[j][k];
135     assign s_cbacki[j+1][2*k+1] = s_cbacko[j][k+(processors/2)];
136
137     assign s_ain[j+1][2*k] = s_aout[j][k];
138     assign s_ain[j+1][2*k+1] = s_aout[j][k+(processors/2)];
139
140     assign s_ifull[j][k] = s_ofull[j+1][2*k];
141     assign s_ifull[j][k+(processors/2)] = s_ofull[j+1][2*k+1];
142
143     assign s_ifullback[j][k] = s_ofullback[j+1][2*k];
144     assign s_ifullback[j][k+(processors/2)] = s_ofullback[j+1][2*k+1];
145    end
146   end
147  endgenerate
148
149  generate
150   for(i=0;i<processors;i=i+1)
151   begin:DEF_FSM
152    always@(posedge CLOCK)
153    begin
154     case(FSM[i])
155     'h0:
156     begin
157      if(req[i] & ~ofullx[i]) FSM[i] <= 'h1; else FSM[i] <= 'h0;
158     end
159
160     'h1:
161     begin
162      if(!busy[i]) FSM[i] <= 'h0; else FSM[i] <= 'h1;
163     end
164
165     endcase
166
167     s_cbackibuf[i] <= s_cout[log_processors-1][i];
168    end
169
170    assign busy[i] = ~s_nextack[log_processors-1][i];
171    assign dout[(i*data_width) +: data_width] = s_dout[log_processors-1][i];
172
173    assign aout[i*int_adr_width +: int_adr_width] = s_aout[log_processors-1][i];
174    assign nextreq[i] = s_nextreq[log_processors-1][i];
175    assign s_ifullback[log_processors-1][i] = 1'b0;
176   end
177  endgenerate
178
179
180  generate
181   for(k=0;k<(processors/2);k=k+1)
182   begin:left
183    assign ofullx[k] = s_ofull[0][2*k];
184    assign ofullx[k+(processors/2)] = s_ofull[0][2*k+1];
185
186    assign s_ifull[log_processors-1][k] = s_ofullback[0][2*k];
187    assign s_ifull[log_processors-1][k+(processors/2)] = s_ofullback[0][2*k+1];
188
```

```
189   assign s_ain[0][2*k] = ain[k*int_adr_width+:int_adr_width];
190   assign s_ain[0][2*k+1] = ain[(k+processors/2)*int_adr_width+:int_adr_width];
191
192   assign s_din[0][2*k] = din[k*data_width +: data_width];
193   assign s_din[0][2*k+1] = din[(k+processors/2)*data_width +: data_width];
194
195   assign s_req[0][2*k] = req[k] & FSM[k]=='h0;
196   assign s_req[0][2*k+1] = req[k+(processors/2)] & FSM[k+(processors/2)]=='h0;
197
198   assign s_cin[0][2*k] = k;
199   assign s_cin[0][2*k+1]= k+(processors/2);
200
201   assign s_cbacki[0][2*k] = s_cbackibuf[k];
202   assign s_cbacki[0][2*k+1] = s_cbackibuf[k+(processors/2)];
203
204   assign s_ack[0][2*k] = ~nextbusy[k];
205   assign s_ack[0][2*k+1] = ~nextbusy[k+(processors/2)];
206  end
207 endgenerate
208
209 endmodule
```

Quellcode 9.10: Implementierung des Omeganetzwerks mit FIFO-Shuffleelementen

9.9 Implementierung der Architekturen

9.9.1 Spezialisierte GCA-Architektur mit Hashfunktionen (HA)

```
1  `include "defines.v"
2
3  module MYNIOSSYSTEM(CLK, RESET);
4
5  parameter CMD_DONE                = 8'd17;
6  parameter CMD_GET_NET_D3          = 8'd16;
7  parameter CMD_GET_NET_D2          = 8'd15;
8  parameter CMD_GET_NET_D1          = 8'd14;
9  parameter CMD_GET_NET_D0          = 8'd13;
10
11 parameter CMD_WRITE_TO_HASH       = 8'd12;
12 parameter CMD_GET_IT_NEXT         = 8'd11;
13
14 parameter CMD_SET_BLOCK3          = 8'd10;
15 parameter CMD_SET_BLOCK2          = 8'd09;
16 parameter CMD_SET_BLOCK1          = 8'd08;
17 parameter CMD_SET_BLOCK0          = 8'd07;
18
19 parameter CMD_GET_NEXT_FIELD_A    = 8'd06;
20
21 parameter CMD_GET_NEXT_BLOCK_D3   = 8'd05;
22 parameter CMD_GET_NEXT_BLOCK_D2   = 8'd04;
23 parameter CMD_GET_NEXT_BLOCK_D1   = 8'd03;
24 parameter CMD_GET_NEXT_BLOCK_D0   = 8'd02;
```

```
25
26 parameter CMD_CONTAINS            = 8'd01;
27 parameter CMD_NEXTGEN             = 8'd00;
28
29
30
31 `ifdef PROCESSORS1
32 parameter processors=1;
33 parameter log_num_cells=0;
34 parameter PROCS="1";
35 parameter memory_adr_width=11;
36 `endif
37 `ifdef PROCESSORS2
38 parameter processors=2;
39 parameter log_num_cells=1;
40 parameter PROCS="2";
41 parameter memory_adr_width=10;
42 `endif
43 `ifdef PROCESSORS4
44 parameter processors=4;
45 parameter log_num_cells=2;
46 parameter PROCS="4";
47 parameter memory_adr_width=9;
48 `endif
49 `ifdef PROCESSORS8
50 parameter processors=8;
51 parameter log_num_cells=3;
52 parameter PROCS="8";
53 parameter memory_adr_width=8;
54 `endif
55
56
57 parameter data_width=16;
58 parameter blocks=2;
59 parameter log_blocks=1;
60
61 parameter total_data_size=2**(memory_adr_width-1+log_num_cells);
62 parameter total_memory_adr_width = log_num_cells+memory_adr_width;
63
64 parameter hash_save_adr_width=11;
65
66 input CLK;
67 input RESET;
68
69 wire        EXT_CLK_EN [processors-1:0];
70 wire [31:0] EXT_DATAA  [processors-1:0];
71 wire [31:0] EXT_DATAB  [processors-1:0];
72 wire        EXT_DONE   [processors-1:0];
73 wire        EXT_RESET  [processors-1:0];
74 reg  [31:0] EXT_RESULT [processors-1:0];
75 wire        EXT_START  [processors-1:0];
76 wire [7:0]  EXT_N      [processors-1:0];
77
78 wire [processors-1:0]        W_EXT_CLK_EN;
79 wire [((32*processors)-1):0] W_EXT_DATAA;
```

```
80  wire [((32*processors)-1):0] W_EXT_DATAB;
81  wire [processors-1:0]        W_EXT_DONE;
82  wire [(8*processors)-1:0]    W_EXT_N;
83  wire [processors-1:0]        W_EXT_RESET;
84  wire [((32*processors)-1):0] W_EXT_RESULT;
85  wire [processors-1:0]        W_EXT_START;
86
87
88  wire NEXTGEN [processors-1:0];
89
90  wire GET_IT_NEXT      [processors-1:0];
91  wire GET_NEXT_BLOCK_D [processors-1:0] [blocks-1:0];
92  wire GET_NEXT_FIELD_A [processors-1:0];
93  wire CONTAINS         [processors-1:0];
94  wire SET_BLOCK        [processors-1:0] [blocks-1:0];
95  wire WRITE_TO_HASH    [processors-1:0];
96  wire GET_NET_D        [processors-1:0] [blocks-1:0];
97  wire DONE             [processors-1:0];
98
99  reg GET_NEXT_BLOCK_ANY_D [processors-1:0];
100 reg SET_BLOCK_ANY_D      [processors-1:0];
101 reg GET_NET_ANY_D        [processors-1:0];
102
103 reg [log_blocks-1:0] BLOCK_INDEX [processors-1:0];
104
105 wire [total_memory_adr_width-1-1:0] HASH_VALUE [processors-1:0];
106 reg  [total_memory_adr_width-1-1:0] NEXT_HASH  [processors-1:0];
107
108 wire [total_memory_adr_width-1-1:0] HASH_VALUE_W;
109 reg  [total_memory_adr_width-1-1:0] NEXT_HASH_W;
110
111 reg GENERATION;
112 reg [((total_data_size/processors)*2)-1:0] FLAG_OCCUPIED [processors-1:0];
113 reg [data_width-1:0]          BUFFER_CELL   [processors-1:0] [blocks-1:0];
114 reg [hash_save_adr_width-1:0] BUFFER_ADR    [processors-1:0];
115
116 `ifdef PROCESSORS1
117  reg [0:0] WRITE_INDEX;
118 `else
119  reg [log_num_cells-1:0] WRITE_INDEX;
120 `endif
121
122 reg ALLNEXTGEN;
123
124 wire [((data_width*processors)-1):0]                                  total_din;
125 wire [((data_width*processors)-1):0]                                 total_dout;
126 wire [(((total_memory_adr_width-1+log_blocks+1)*processors)-1):0]     total_ain;
127 wire [(((total_memory_adr_width-1+log_blocks+1)*processors)-1):0]    total_aout;
128 wire [processors-1:0]                                                 total_req;
129 wire [processors-1:0]                                                total_busy;
130 wire [processors-1:0]                                             total_nextreq;
131 wire [processors-1:0]                                            total_nextbusy;
132
133
134 wire [log_num_cells-1:0]      NET_ID          [processors-1:0];
```

```
135 wire [memory_adr_width-1-1:0] NET_MEM_ADR     [processors-1:0];
136 wire [log_blocks-1:0]         NET_FIELD_INDEX [processors-1:0];
137 wire NET_GET_NEXT_FIELD_A                     [processors-1:0];
138 wire NET_REQ                                  [processors-1:0];
139 wire NET_BUSY                                 [processors-1:0];
140 wire [data_width-1:0] NET_RES                 [processors-1:0];
141
142 reg READ_DELAY       [processors-1:0];
143 reg READ_DELAY2      [processors-1:0];
144 reg GET_NEXT_NET_ADR [processors-1:0];
145
146 reg [2:0] NET_FSM     [processors-1:0];
147 reg START_NET_READ    [processors-1:0];
148 reg NET_DONE          [processors-1:0];
149 reg [31:0] NET_RESULT [processors-1:0];
150
151
152 wire                          IT_EMPTY         [processors-1:0] [blocks-1:0];
153 wire                          IT_FULL          [processors-1:0] [blocks-1:0];
154 wire [data_width-1:0]         IT_Q             [processors-1:0] [blocks-1:0];
155
156 reg                           IT_FULL_ANY      [processors-1:0];
157 reg                           IT_EMPTY_ANY     [processors-1:0];
158 reg                           IT_EMPTY_ALL     [processors-1:0];
159 reg                           IT_WR_ALL        [processors-1:0];
160
161 wire                          IT_ADR_EMPTY     [processors-1:0];
162 wire                          IT_ADR_FULL      [processors-1:0];
163 wire [hash_save_adr_width-1:0] IT_ADR_Q        [processors-1:0];
164
165 wire                          WRITER_EMPTY     [processors-1:0] [blocks-1:0];
166 wire                          WRITER_FULL      [processors-1:0] [blocks-1:0];
167 wire [data_width-1:0]         WRITER_Q         [processors-1:0] [blocks-1:0];
168 reg  [data_width-1:0]         WRITER_Q_REG     [processors-1:0] [blocks-1:0];
169
170 reg                           WRITER_FULL_ANY  [processors-1:0];
171 reg                           WRITER_EMPTY_ANY [processors-1:0];
172 reg                           WRITER_EMPTY_ALL [processors-1:0];
173 reg                           WRITER_RD_ALL    [processors-1:0];
174
175 wire                          WRITER_ADR_EMPTY [processors-1:0];
176 wire                          WRITER_ADR_FULL  [processors-1:0];
177 wire [hash_save_adr_width-1:0] WRITER_ADR_Q     [processors-1:0];
178 reg  [hash_save_adr_width-1:0] WRITER_ADR_Q_REG [processors-1:0];
179
180 reg [10:0] WRITER_FSM;
181 reg WRITER_DATA_AVAIL;
182 reg WRITER_DATA_NEXT [processors-1:0];
183 reg WRITER_DATA_ANY_NEXT;
184 reg WRITER_PROCESSING;
185
186 reg [10:0]                    ARB_FSM     [processors-1:0];
187 reg [memory_adr_width-1-1:0] ARB_GEN_ADR [processors-1:0];
188 reg ARB_FIN_RD [processors-1:0];
189
```

```
190 reg WRITE_D [processors-1:0];
191
192 reg ARBITER_NEXTGEN_READY [processors-1:0];
193
194 wire [data_width-1:0] q_a [processors-1:0] [blocks-1:0];
195 wire [data_width-1:0] q_b [processors-1:0] [blocks-1:0];
196
197 wire [hash_save_adr_width-1:0] q_adr_a [processors-1:0];
198 wire [hash_save_adr_width-1:0] q_adr_b [processors-1:0];
199
200 wire SYSTEM_CLK;
201
202 genvar i;
203 genvar j;
204
205 integer k;
206 integer m;
207 integer n;
208 integer p;
209
210 PLL PLL_inst (
211     .inclk0 (CLK),
212     .c0(SYSTEM_CLK)
213     );
214
215 generate
216  for(i=0;i<processors;i=i+1)
217  begin:DEF_DECODE
218
219   assign WRITE_TO_HASH[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_TO_HASH);
220   assign GET_IT_NEXT[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_IT_NEXT);
221
222   if(blocks>=4)
223    assign SET_BLOCK[i][3] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_SET_BLOCK3);
224   if(blocks>=3)
225    assign SET_BLOCK[i][2] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_SET_BLOCK2);
226   if(blocks>=2)
227    assign SET_BLOCK[i][1] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_SET_BLOCK1);
228   if(blocks>=1)
229    assign SET_BLOCK[i][0] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_SET_BLOCK0);
230
231   if(blocks>=4)
232    assign GET_NEXT_BLOCK_D[i][3]
233           =EXT_CLK_EN[i]&(EXT_N[i]==CMD_GET_NEXT_BLOCK_D3);
234   if(blocks>=3)
235    assign GET_NEXT_BLOCK_D[i][2]
236           =EXT_CLK_EN[i]&(EXT_N[i]==CMD_GET_NEXT_BLOCK_D2);
237   if(blocks>=2)
238    assign GET_NEXT_BLOCK_D[i][1]
239           =EXT_CLK_EN[i]&(EXT_N[i]==CMD_GET_NEXT_BLOCK_D1);
240   if(blocks>=1)
241    assign GET_NEXT_BLOCK_D[i][0]
242           =EXT_CLK_EN[i]&(EXT_N[i]==CMD_GET_NEXT_BLOCK_D0);
243
244   assign GET_NEXT_FIELD_A[i]=EXT_CLK_EN[i]&(EXT_N[i]==CMD_GET_NEXT_FIELD_A);
```

```
245
246   if(blocks>=4)
247    assign GET_NET_D[i][3] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_NET_D3);
248   if(blocks>=3)
249    assign GET_NET_D[i][2] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_NET_D2);
250   if(blocks>=2)
251    assign GET_NET_D[i][1] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_NET_D1);
252   if(blocks>=1)
253    assign GET_NET_D[i][0] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_NET_D0);
254
255   assign CONTAINS[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_CONTAINS);
256   assign NEXTGEN[i]  = EXT_CLK_EN[i] & (EXT_N[i]==CMD_NEXTGEN);
257   assign DONE[i]     = EXT_CLK_EN[i] & (EXT_N[i]==CMD_DONE);
258
259   assign HASH_VALUE[i] = (EXT_DATAA[i][hash_save_adr_width-1:0]
260                         + NEXT_HASH[i] + NEXT_HASH[i]) % total_data_size;
261  end
262 endgenerate
263
264 assign HASH_VALUE_W =(WRITER_ADR_Q_REG[WRITE_INDEX]
265                     + NEXT_HASH_W + NEXT_HASH_W) % total_data_size;
266
267 always@*
268 begin
269  for(k=0;k<processors;k=k+1)
270  begin:DEF_NEXT_FIELD_DECODE
271   for(m=0;m<blocks;m=m+1)
272   begin:DEF_NEXT_FIELD_DECODE2
273    if(m==0)
274     GET_NEXT_BLOCK_ANY_D[k]=GET_NEXT_BLOCK_D[k][m];
275    else
276     GET_NEXT_BLOCK_ANY_D[k]=GET_NEXT_BLOCK_ANY_D[k] | GET_NEXT_BLOCK_D[k][m];
277
278    if(m==0)
279     GET_NET_ANY_D[k]=GET_NET_D[k][m];
280    else
281     GET_NET_ANY_D[k]=GET_NET_ANY_D[k] | GET_NET_D[k][m];
282
283    if(m==0)
284     SET_BLOCK_ANY_D[k]=SET_BLOCK[k][m];
285    else
286     SET_BLOCK_ANY_D[k]=SET_BLOCK_ANY_D[k] | SET_BLOCK[k][m];
287
288    if(GET_NEXT_BLOCK_D[k][m] | GET_NET_D[k][m])
289     BLOCK_INDEX[k]=m;
290    else
291     BLOCK_INDEX[k]=0;
292   end
293  end
294 end
295
296 always@*
297 begin
298  ALLNEXTGEN = WRITER_EMPTY_ALL[0] & ARB_FIN_RD[0] & NEXTGEN[0]
299             & ARBITER_NEXTGEN_READY[0] & IT_EMPTY_ALL[0] & ~WRITER_PROCESSING;
```

```
300
301  for(k=1;k<processors;k=k+1)
302  begin:DEF_NEXTGEN_EMPTY_FIN
303   ALLNEXTGEN = ALLNEXTGEN & WRITER_EMPTY_ALL[k] & ARB_FIN_RD[k]
304              & NEXTGEN[k] & ARBITER_NEXTGEN_READY[k] & IT_EMPTY_ALL[k];
305  end
306 end
307
308 generate
309  for(i=0;i<processors;i=i+1)
310  begin:DEF_ProcessorNet
311   assign EXT_CLK_EN[i]=W_EXT_CLK_EN[i];
312   assign EXT_DATAA[i]=W_EXT_DATAA[((i+1)*32)-1:i*32];
313   assign EXT_DATAB[i]=W_EXT_DATAB[((i+1)*32)-1:i*32];
314   assign W_EXT_DONE[i]=EXT_DONE[i];
315   assign EXT_N[i]=W_EXT_N[((i+1)*8)-1:i*8];
316   assign EXT_RESET[i]=W_EXT_RESET[i];
317   assign W_EXT_RESULT[((i+1)*32)-1:i*32]=EXT_RESULT[i];
318   assign EXT_START[i]=W_EXT_START[i];
319  end
320 endgenerate
321
322 PROCESSORS #(.processors(processors))
323 DUT(
324 .CLK(SYSTEM_CLK),
325 .RESET(RESET),
326 .W_EXT_CLK_EN(W_EXT_CLK_EN),
327 .W_EXT_DATAA(W_EXT_DATAA),
328 .W_EXT_DATAB(W_EXT_DATAB),
329 .W_EXT_DONE(W_EXT_DONE),
330 .W_EXT_N(W_EXT_N),
331 .W_EXT_RESET(W_EXT_RESET),
332 .W_EXT_RESULT(W_EXT_RESULT),
333 .W_EXT_START(W_EXT_START)
334 );
335
336 generate
337  for(i=0;i<processors;i=i+1)
338  begin:DEF_NIOS_DONE
339   assign EXT_DONE[i]=(GET_IT_NEXT[i] & (~IT_EMPTY_ANY[i] | ARB_FIN_RD[i]))
340          |(GET_NEXT_BLOCK_ANY_D[i] | GET_NEXT_FIELD_A[i])
341          | SET_BLOCK_ANY_D[i]
342          | (WRITE_TO_HASH[i] & ~WRITER_FULL_ANY[i])
343          | ALLNEXTGEN
344          | ((GET_NET_ANY_D[i] | CONTAINS[i]) & NET_DONE[i]);
345  end
346 endgenerate
347
348
349 always@*
350 begin
351  for(k=0;k<processors;k=k+1)
352  begin:DEF_NIOS_RESULT
353
354   if(GET_NEXT_BLOCK_ANY_D[k])
```

```
355     EXT_RESULT[k]=(IT_Q[k][BLOCK_INDEX[k]] + 32'b0);
356    else
357    begin
358     if(GET_NEXT_FIELD_A[k])
359      EXT_RESULT[k]=(IT_ADR_Q[k] + 32'b0);
360     else
361     begin
362      if(GET_IT_NEXT[k])
363       EXT_RESULT[k]= {31'b0,~(ARB_FIN_RD[k] & IT_EMPTY_ANY[k])};
364      else
365      begin
366       EXT_RESULT[k]=NET_RESULT[k];
367      end
368     end
369    end
370   end
371  end
372
373  generate
374   for(i=0;i<processors;i=i+1)
375   begin:DEF_nets
376    assign total_din[((i+1)*data_width)-1:i*data_width] =
377     NET_GET_NEXT_FIELD_A[i] ? q_adr_a[i] : q_a[i][NET_FIELD_INDEX[i]];
378    assign NET_RES[i] = total_dout[((i+1)*data_width)-1:i*data_width];
379    assign total_ain[((i+1)*(total_memory_adr_width-1+log_blocks+1))-1:
380     i*(total_memory_adr_width-1+log_blocks+1)] =
381     {HASH_VALUE[i][total_memory_adr_width-1-1:0],
382     BLOCK_INDEX[i],GET_NEXT_NET_ADR[i]};
383    assign {NET_ID[i],NET_MEM_ADR[i],NET_FIELD_INDEX[i],NET_GET_NEXT_FIELD_A[i]}=
384     total_aout[((i+1)*(total_memory_adr_width-1+log_blocks+1))-1:
385     i*(total_memory_adr_width-1+log_blocks+1)];
386    assign total_req[i] = START_NET_READ[i];
387    assign NET_BUSY[i] = total_busy[i];
388    assign NET_REQ[i] = total_nextreq[i];
389    assign total_nextbusy[i] = ~READ_DELAY2[i];
390   end
391  endgenerate
392
393  `ifdef PROCESSORS1
394  generate
395   for(i=0;i<processors;i=i+1)
396   begin:DEF_PROC1_NETS
397    assign total_dout[((i+1)*data_width)-1:i*data_width] =
398    total_din[((i+1)*data_width)-1:i*data_width];
399    assign total_nextreq[i] = total_req[i];
400    assign total_busy[i] = total_nextbusy[i];
401    assign total_aout[((i+1)*(total_memory_adr_width-1+log_blocks+1))-1:
402    i*(total_memory_adr_width-1+log_blocks+1)] =
403    total_ain[((i+1)*(total_memory_adr_width-1+log_blocks+1))-1:
404    i*(total_memory_adr_width-1+log_blocks+1)];
405   end
406  endgenerate
407  `else
408  NETWORK_INST #(.adr_width(memory_adr_width-1+log_blocks+1),
409                 .data_width(data_width),
```

```
410                 .num_cells(processors),
411                 .log_num_cells(log_num_cells))
412               net(
413                 .CLOCK(SYSTEM_CLK),
414                 .din(total_din),
415                 .dout(total_dout),
416                 .req(total_req),
417                 .busy(total_busy),
418                 .nextreq(total_nextreq),
419                 .nextbusy(total_nextbusy),
420                 .ain(total_ain),
421                 .aout(total_aout)
422                 );
423
424 `endif
425
426 always@(posedge SYSTEM_CLK)
427 begin
428  for(k=0;k<processors;k=k+1)
429  begin:DEF_READ_DELAY
430   READ_DELAY[k] <= NET_REQ[k] & ~READ_DELAY2[k];
431   READ_DELAY2[k] <= READ_DELAY[k] & ~READ_DELAY2[k];
432  end
433 end
434
435
436 always@(posedge SYSTEM_CLK)
437 begin
438  for(k=0;k<processors;k=k+1)
439  begin:DEF_NETWORK_FSM
440
441   if(~RESET)
442   begin
443    NET_FSM[k]<='h0;
444    START_NET_READ[k]<=1'b0;
445    GET_NEXT_NET_ADR[k]<=1'b0;
446    NET_DONE[k]<='b0;
447   end
448   else
449   begin
450    case(NET_FSM[k])
451
452    'h0:
453    begin
454     NET_FSM[k]<='h1;
455     NET_DONE[k]<='b0;
456     NEXT_HASH[k]<='b0;
457    end
458
459    'h1:
460    begin
461     if(CONTAINS[k])
462     begin
463     `ifdef PROCESSORS1
464       if(FLAG_OCCUPIED[0][{GENERATION,HASH_VALUE[k][memory_adr_width-1-1:0]}])
```

```
465     `else
466       if(FLAG_OCCUPIED[HASH_VALUE[k][memory_adr_width+log_num_cells-1-1:
467       memory_adr_width-1]][{GENERATION,HASH_VALUE[k][memory_adr_width-1-1:0]}])
468     `endif
469       begin
470        START_NET_READ[k]<=1'b1;
471        GET_NEXT_NET_ADR[k]<=1'b1;
472        NET_DONE[k]<=1'b0;
473        NET_FSM[k]<='h3;
474       end
475       else
476       begin
477        START_NET_READ[k]<=1'b0;
478        NET_DONE[k]<=1'b1;
479        NET_RESULT[k]<='h0;
480        NET_FSM[k]<='h2;
481       end
482      end
483      else
484      begin
485       if(GET_NET_ANY_D[k])
486       begin
487        START_NET_READ[k]<=1'b1;
488        GET_NEXT_NET_ADR[k]<=1'b1;
489        NET_DONE[k]<=1'b0;
490        NET_FSM[k]<='h3;
491       end
492       else
493       begin
494        START_NET_READ[k]<=1'b0;
495        NET_DONE[k]<=1'b0;
496        NET_FSM[k]<='h1;
497       end
498      end
499     end
500
501    'h2:
502    begin
503     NEXT_HASH[k]<='b0;
504     if(CONTAINS[k] | GET_NET_ANY_D[k])
505     begin
506      NET_DONE[k]<=1'b1;
507      NET_FSM[k]<='h2;
508     end
509     else
510     begin
511      NET_DONE[k]<=1'b0;
512      NET_FSM[k]<='h1;
513     end
514    end
515
516    'h3:
517    begin
518     if(~NET_BUSY[k])
519     begin
```

```
520     if(NET_RES[k][hash_save_adr_width-1:0]
521        ==EXT_DATAA[k][hash_save_adr_width-1:0])
522     begin
523      if(CONTAINS[k])
524      begin
525       START_NET_READ[k]<=1'b0;
526       NET_FSM[k]<='h2;
527       NET_DONE[k]<=1'b1;
528       NET_RESULT[k]<='h1;
529      end
530      else
531      begin
532       START_NET_READ[k]<=1'b0;
533       GET_NEXT_NET_ADR[k]<=1'b0;
534       NET_DONE[k]<=1'b0;
535       NET_FSM[k]<='h4;
536      end
537     end
538     else
539     begin
540      NEXT_HASH[k]<=NEXT_HASH[k]+'b1;
541      START_NET_READ[k]<=1'b0;
542      GET_NEXT_NET_ADR[k]<=1'b1;
543      NET_FSM[k]<='h1;
544      NET_DONE[k]<=1'b0;
545     end
546    end
547    else
548    begin
549     START_NET_READ[k]<=1'b1;
550     NET_DONE[k]<=1'b0;
551     NET_FSM[k]<='h3;
552    end
553   end
554
555   'h4:
556   begin
557    if(~NET_BUSY[k])
558    begin
559     NET_DONE[k]<=1'b1;
560     NET_FSM[k]<='h2;
561     NET_RESULT[k]<=NET_RES[k];
562     START_NET_READ[k]<=1'b0;
563    end
564    else
565    begin
566     START_NET_READ[k]<=1'b1;
567     NET_FSM[k]<='h4;
568     NET_DONE[k]<=1'b0;
569    end
570   end
571   endcase
572
573  end
574 end
```

```
575 end
576
577 always@(posedge SYSTEM_CLK)
578 begin
579  for(k=0;k<processors;k=k+1)
580  begin
581   WRITER_ADR_Q_REG[k]<=WRITER_ADR_Q[k];
582   for(m=0;m<blocks;m=m+1)
583   begin
584    WRITER_Q_REG[k][m]<=WRITER_Q[k][m];
585   end
586  end
587 end
588
589 generate
590  for(i=0;i<processors;i=i+1)
591  begin:DEF_WRITE_BUFFER
592
593  ITERATOR_ADR WRITE_BUFFER_ADR_inst (
594               .aclr (~RESET),
595               .clock (SYSTEM_CLK),
596               .data (BUFFER_ADR[i]),
597               .rdreq (WRITER_RD_ALL[i]),
598               .wrreq ((WRITE_TO_HASH[i] & ~WRITER_FULL_ANY[i])),
599               .empty (WRITER_ADR_EMPTY[i]),
600               .full (WRITER_ADR_FULL[i]),
601               .q (WRITER_ADR_Q[i])
602               );
603
604
605   for(j=0;j<blocks;j=j+1)
606   begin:DEF_WRITE_BUFFER_NUMBER
607    ITERATOR WRITE_BUFFER_DATA_inst (
608             .aclr (~RESET),
609             .clock (SYSTEM_CLK),
610             .data (BUFFER_CELL[i][j]),
611             .rdreq (WRITER_RD_ALL[i]),
612             .wrreq ((WRITE_TO_HASH[i] & ~WRITER_FULL_ANY[i])),
613             .empty (WRITER_EMPTY[i][j]),
614             .full (WRITER_FULL[i][j]),
615             .q (WRITER_Q[i][j])
616             );
617   end
618  end
619 endgenerate
620
621
622 always@*
623 begin
624  WRITER_DATA_ANY_NEXT=1'b0;
625
626  for(k=0;k<processors;k=k+1)
627  begin:DEF_ITERATOR_FULL
628
629   WRITER_FULL_ANY[k]=WRITER_ADR_FULL[k];
```

```
630   WRITER_EMPTY_ANY[k]=WRITER_ADR_EMPTY[k];
631   WRITER_EMPTY_ALL[k]=WRITER_ADR_EMPTY[k];
632   WRITER_DATA_ANY_NEXT=WRITER_DATA_ANY_NEXT | WRITER_DATA_NEXT[k];
633
634   for(m=0;m<blocks;m=m+1)
635   begin:DEF_ITERATOR_NUMBER_FULL
636    WRITER_FULL_ANY[k]=WRITER_FULL_ANY[k] | WRITER_FULL[k][m];
637    WRITER_EMPTY_ANY[k]=WRITER_EMPTY_ANY[k] | WRITER_EMPTY[k][m];
638    WRITER_EMPTY_ALL[k]=WRITER_EMPTY_ALL[k] & WRITER_EMPTY[k][m];
639   end
640  end
641 end
642
643 always@(posedge SYSTEM_CLK)
644 begin
645  for(k=0;k<processors;k=k+1)
646  begin:DEF_SAVE_D_FIELDS
647
648   if(SET_BLOCK_ANY_D[k])
649    BUFFER_ADR[k]<=EXT_DATAA[k];
650
651   for(m=0;m<blocks;m=m+1)
652   begin:DEF_SAVE_D_FIELDS_NUMBER
653    if(SET_BLOCK[k][m])
654    begin
655     BUFFER_CELL[k][m]<=EXT_DATAB[k][data_width-1:0];
656    end
657   end
658  end
659 end
660
661 always@(posedge SYSTEM_CLK)
662 begin
663  case(WRITER_FSM)
664
665  'h0:
666  begin
667   WRITER_FSM<='h1;
668   WRITE_INDEX<='h0;
669   WRITER_DATA_AVAIL<='b0;
670   WRITER_PROCESSING<='h0;
671   for(m=0;m<processors;m=m+1)
672   begin:DEF_RESET_RD_ALL
673    WRITER_RD_ALL[m]<='h0;
674   end
675  end
676
677  'h1:
678  begin
679   NEXT_HASH_W<='b0;
680   if(~WRITER_EMPTY_ANY[WRITE_INDEX])
681   begin
682    WRITER_RD_ALL[WRITE_INDEX]<='h1;
683    WRITER_FSM<='h4;
684    WRITER_PROCESSING<='h1;
```

```
685   end
686   else
687   begin
688    `ifndef PROCESSORS1
689     WRITE_INDEX<=WRITE_INDEX+'b1;
690    `endif
691    WRITER_RD_ALL[WRITE_INDEX]<='h0;
692    WRITER_FSM<='h1;
693    WRITER_PROCESSING<='h0;
694   end
695  end
696
697  'h4:
698  begin
699   WRITER_RD_ALL[WRITE_INDEX]<='h0;
700   WRITER_FSM<='h5;
701  end
702
703  'h5:
704  begin
705   WRITER_RD_ALL[WRITE_INDEX]<='h0;
706   WRITER_FSM<='h6;
707  end
708
709  'h6:
710  begin
711   `ifdef PROCESSORS1
712    if(FLAG_OCCUPIED[0][{~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}])
713   `else
714    if(FLAG_OCCUPIED[HASH_VALUE_W[memory_adr_width+log_num_cells-1-1:
715    memory_adr_width-1]][{~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}])
716   `endif
717   begin
718    NEXT_HASH_W<=NEXT_HASH_W+'b1;
719    WRITER_FSM<='h6;
720   end
721   else
722   begin
723    WRITER_FSM<='h2;
724   end
725  end
726
727  'h2:
728  begin
729   WRITER_RD_ALL[WRITE_INDEX]<='h0;
730
731   if(WRITER_DATA_ANY_NEXT)
732   begin
733    WRITER_FSM<='h3;
734    `ifndef PROCESSORS1
735     WRITE_INDEX<=WRITE_INDEX+'b1;
736    `endif
737    WRITER_DATA_AVAIL<='b0;
738   end
739   else
```

```
740   begin
741    WRITER_FSM<='h2;
742    WRITER_DATA_AVAIL<='b1;
743   end
744  end
745
746  'h3:
747  begin
748   if(WRITER_DATA_ANY_NEXT)
749    WRITER_FSM<='h3;
750   else
751    WRITER_FSM<='h1;
752  end
753
754  endcase
755 end
756
757
758 generate
759  for(i=0;i<processors;i=i+1)
760  begin:DEF_ITERATOR
761
762   ITERATOR_ADR ITERATOR_ADR_inst (
763   .aclr (~RESET | ALLNEXTGEN),
764   .clock (SYSTEM_CLK),
765   .data (q_adr_b[i]),
766   .rdreq ((GET_IT_NEXT[i] & ~IT_EMPTY_ANY[i])),
767   .wrreq (IT_WR_ALL[i]),
768   .empty (IT_ADR_EMPTY[i]),
769   .full (IT_ADR_FULL[i]),
770   .q (IT_ADR_Q[i])
771   );
772
773
774   for(j=0;j<blocks;j=j+1)
775   begin:DEF_ITERATOR_NUMBER
776    ITERATOR ITERATOR_inst (
777    .aclr (~RESET | ALLNEXTGEN),
778    .clock (SYSTEM_CLK),
779    .data (q_b[i][j]),
780    .rdreq ((GET_IT_NEXT[i] & ~IT_EMPTY_ANY[i])),
781    .wrreq (IT_WR_ALL[i]),
782    .empty (IT_EMPTY[i][j]),
783    .full (IT_FULL[i][j]),
784    .q (IT_Q[i][j])
785    );
786   end
787  end
788 endgenerate
789
790
791 always@*
792 begin
793  for(k=0;k<processors;k=k+1)
794  begin:DEF_ITERATOR_FULL
```

```
795
796     IT_FULL_ANY[k]=IT_ADR_FULL[k];
797     IT_EMPTY_ANY[k]=IT_ADR_EMPTY[k];
798     IT_EMPTY_ALL[k]=IT_ADR_EMPTY[k];
799
800     for(m=0;m<blocks;m=m+1)
801     begin:DEF_ITERATOR_NUMBER_FULL
802       IT_FULL_ANY[k]=IT_FULL_ANY[k] | IT_FULL[k][m];
803       IT_EMPTY_ANY[k]=IT_EMPTY_ANY[k] | IT_EMPTY[k][m];
804       IT_EMPTY_ALL[k]=IT_EMPTY_ALL[k] & IT_EMPTY[k][m];
805     end
806   end
807 end
808
809
810 always@(posedge SYSTEM_CLK)
811 begin
812
813   if(ALLNEXTGEN)
814     GENERATION<=~GENERATION;
815
816   for(k=0;k<processors;k=k+1)
817   begin:DEF_ARBITER_FSM
818
819   if(~RESET)
820   begin
821     ARB_FSM[k]<='h0;
822   end
823   else
824   begin
825     case(ARB_FSM[k])
826
827     'h0:
828     begin
829       ARB_FSM[k]<='h1;
830
831       ARBITER_NEXTGEN_READY[k]<=1'b0;
832       IT_WR_ALL[k]<='h0;
833       ARB_GEN_ADR[k]<='h0;
834       ARB_FIN_RD[k]<='h0;
835
836       GENERATION<=1'b0;
837       WRITER_DATA_NEXT[k]<=1'b0;
838       for(m=0;m<((total_data_size/processors)*2);m=m+1)
839       begin:DEF_TEST
840         FLAG_OCCUPIED[k][m]<=1'b0;
841       end
842     end
843
844     'h1:
845     begin
846       IT_WR_ALL[k]<='h0;
847       ARBITER_NEXTGEN_READY[k]<=1'b1;
848
849       if(ALLNEXTGEN)
```

```
850     begin
851      WRITER_DATA_NEXT[k]<=1'b0;
852
853      if(GENERATION)
854      begin
855       for(n=(total_data_size/processors);n<((total_data_size/processors)
856        +(total_data_size/processors));n=n+1)
857       begin:DEF_RESET_FLAG_GEN1
858        FLAG_OCCUPIED[k][n]<=1'b0;
859       end
860      end
861      else
862      begin
863       for(p=0;p<(total_data_size/processors);p=p+1)
864       begin:DEF_RESET_FLAG_GEN0
865        FLAG_OCCUPIED[k][p]<=1'b0;
866       end
867      end
868
869
870      ARB_FIN_RD[k]<=1'b0;
871      ARB_GEN_ADR[k]<='b0;
872      ARB_FSM[k]<='h1;
873     end
874     else
875     begin
876     `ifdef PROCESSORS1
877      if(WRITER_DATA_AVAIL & (~IT_EMPTY_ANY[k] | ARB_FIN_RD[k]))
878     `else
879      if(WRITER_DATA_AVAIL & (HASH_VALUE_W[memory_adr_width+log_num_cells-1-1
880       :memory_adr_width-1]==k) & (~IT_EMPTY_ANY[k] | ARB_FIN_RD[k]))
881     `endif
882      begin
883       ARB_FSM[k]<='h5;
884       `ifdef PROCESSORS1
885        FLAG_OCCUPIED[0][{~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}]<=1;
886       `else
887        FLAG_OCCUPIED[HASH_VALUE_W[memory_adr_width+log_num_cells-1-1
888        :memory_adr_width-1]][{~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}]
889        <=1'b1;
890       `endif
891       WRITE_D[k]<='h1;
892       WRITER_DATA_NEXT[k]<=1'b1;
893      end
894      else
895      begin
896       WRITER_DATA_NEXT[k]<=1'b0;
897       WRITE_D[k]<='h0;
898       if(~IT_FULL_ANY[k] & ~ARB_FIN_RD[k])
899       begin
900        if(FLAG_OCCUPIED[k][{GENERATION,ARB_GEN_ADR[k]}])
901        begin
902         ARB_FSM[k]<='h3;
903        end
904        else
```

```
905         begin
906          ARB_FSM[k]<='h1;
907          ARB_GEN_ADR[k]<=ARB_GEN_ADR[k]+'b1;
908          ARB_FIN_RD[k]<=(ARB_GEN_ADR[k]=={(memory_adr_width-1){1'b1}});
909         end
910        end
911        else
912        begin
913         ARB_FSM[k]<='h1;
914        end
915       end
916      end
917     end
918
919     'h3:
920     begin
921      ARBITER_NEXTGEN_READY[k]<=1'b0;
922      IT_WR_ALL[k]<='h1;
923      ARB_FSM[k]<='h4;
924     end
925
926     'h4:
927     begin
928      ARBITER_NEXTGEN_READY[k]<=1'b0;
929      IT_WR_ALL[k]<='h0;
930      ARB_FSM[k]<='h1;
931      ARB_FIN_RD[k]<=(ARB_GEN_ADR[k]=={(memory_adr_width-1){1'b1}});
932
933      ARB_GEN_ADR[k]<=ARB_GEN_ADR[k]+'b1;
934     end
935
936     'h5:
937     begin
938      ARBITER_NEXTGEN_READY[k]<=1'b0;
939      WRITE_D[k]<='h0;
940      if(WRITER_DATA_AVAIL)
941       ARB_FSM[k]<='h5;
942      else
943       ARB_FSM[k]<='h1;
944
945      WRITER_DATA_NEXT[k]<=1'b0;
946     end
947
948
949     endcase
950    end
951    end
952   end
953
954   generate
955    for(i=0;i<processors;i=i+1)
956    begin:DEF_RAM_FIELD
957
958     RAM_FIELD_ADR RAM_FIELD_ADR_inst (
959      .address_a ({GENERATION,NET_MEM_ADR[i]}),
```

```
960     .address_b (WRITE_D[i] ? {~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}
961                        : {GENERATION,ARB_GEN_ADR[i]}),
962     .clock (SYSTEM_CLK),
963     .data_a (),
964     .data_b (WRITER_ADR_Q_REG[WRITE_INDEX]),
965     .wren_a (1'b0),
966     .wren_b (WRITE_D[i]),
967     .q_a (q_adr_a[i]),
968     .q_b (q_adr_b[i])
969     );
970
971   for(j=0;j<blocks;j=j+1)
972   begin:DEF_RAM_FIELD_NUMBER
973    RAM_FIELD RAM_FIELD_inst (
974     .address_a ({GENERATION,NET_MEM_ADR[i]}),
975     .address_b (WRITE_D[i] ? {~GENERATION,HASH_VALUE_W[memory_adr_width-1-1:0]}
976                        : {GENERATION,ARB_GEN_ADR[i]}),
977     .clock (SYSTEM_CLK),
978     .data_a (),
979     .data_b (WRITER_Q_REG[WRITE_INDEX][j]),
980     .wren_a (1'b0),
981     .wren_b (WRITE_D[i]),
982     .q_a (q_a[i][j]),
983     .q_b (q_b[i][j])
984     );
985    end
986  end
987 endgenerate
988
989 endmodule
```

Quellcode 9.11: Implementierung der HA

9.9.2 Agentenbasierte speicheroptimierte GCA-Architektur (DAMA)

```
1  `include "defines.v"
2
3  module MYNIOSSYSTEM(CLK, RESET, RESETNIOS);
4
5  parameter CELL_FREE                    = 32'b0;
6  parameter CELL_AGENT_N                 = 32'd2;
7  parameter CELL_AGENT_E                 = 32'd4;
8  parameter CELL_AGENT_S                 = 32'd6;
9  parameter CELL_AGENT_W                 = 32'd8;
10 parameter CMD_GET_ID                   = 8'd51;
11 parameter CMD_CHECK_CELL               = 8'd50;
12
13 parameter CMD_AGENT_ADR                = 8'd40;
14
15 parameter CMD_AGENT_READ_BLOCK6        = 8'd36;
16 parameter CMD_AGENT_READ_BLOCK5        = 8'd35;
17 parameter CMD_AGENT_READ_BLOCK4        = 8'd34;
18 parameter CMD_AGENT_READ_BLOCK3        = 8'd33;
19 parameter CMD_AGENT_READ_BLOCK2        = 8'd32;
```

```
20  parameter CMD_AGENT_READ_BLOCK1         = 8'd31;
21  parameter CMD_AGENT_READ_BLOCK0         = 8'd30;
22
23  parameter CMD_GET_NEXT_AGENT            = 8'd29;
24
25  parameter CMD_GET_CUR_AGENT_BLOCK6      = 8'd26;
26  parameter CMD_GET_CUR_AGENT_BLOCK5      = 8'd25;
27  parameter CMD_GET_CUR_AGENT_BLOCK4      = 8'd24;
28  parameter CMD_GET_CUR_AGENT_BLOCK3      = 8'd23;
29  parameter CMD_GET_CUR_AGENT_BLOCK2      = 8'd22;
30  parameter CMD_GET_CUR_AGENT_BLOCK1      = 8'd21;
31  parameter CMD_GET_CUR_AGENT_BLOCK0      = 8'd20;
32
33  parameter CMD_SEND_AGENT                = 8'd19;
34
35  parameter CMD_WRITE_BLOCK6              = 8'd16;
36  parameter CMD_WRITE_BLOCK5              = 8'd15;
37  parameter CMD_WRITE_BLOCK4              = 8'd14;
38  parameter CMD_WRITE_BLOCK3              = 8'd13;
39  parameter CMD_WRITE_BLOCK2              = 8'd12;
40  parameter CMD_WRITE_BLOCK1              = 8'd11;
41  parameter CMD_WRITE_BLOCK0              = 8'd10;
42
43  parameter CMD_DONE_EXE                  = 8'd2;
44  parameter CMD_TIME                      = 8'd1;
45  parameter CMD_NEXTGEN                   = 8'd0;
46
47
48  `ifdef PROCESSORS1
49  parameter processors  = 1;
50  parameter log_num_cells  = 0;
51  parameter memory_adr_width = 12;
52  parameter agent_memory_width = 10;
53  `endif
54  `ifdef PROCESSORS2
55  parameter processors  = 2;
56  parameter log_num_cells  = 1;
57  parameter memory_adr_width = 11;
58  parameter agent_memory_width = 9;
59  `endif
60  `ifdef PROCESSORS4
61  parameter processors  = 4;
62  parameter log_num_cells  = 2;
63  parameter memory_adr_width = 10;
64  parameter agent_memory_width = 8;
65  `endif
66  `ifdef PROCESSORS8
67  parameter processors  = 8;
68  parameter log_num_cells  = 3;
69  parameter memory_adr_width = 9;
70  parameter agent_memory_width = 7;
71  `endif
72  `ifdef PROCESSORS16
73  parameter processors  = 16;
74  parameter log_num_cells  = 4;
```

```
75  parameter memory_adr_width = 8;
76  parameter agent_memory_width = 6;
77  `endif
78
79  parameter dat_width             = 16;
80  parameter link_width            = log_num_cells+(agent_memory_width-1)+1;
81  parameter blocks                = 2;
82  parameter log_blocks            = 1;
83
84  (* chip_pin = "AD15" *) input CLK;
85  (* chip_pin = "T29" *) input RESET;
86  (* chip_pin = "T28" *) input RESETNIOS;
87
88  wire SYSTEM_CLK;
89
90  reg ALLNEXTGEN;
91  integer g;
92  always@(NEXTGEN)
93  begin
94   ALLNEXTGEN = NEXTGEN[0];
95
96   for(g=1;g<processors;g=g+1)
97   begin:DEF_RWN
98    ALLNEXTGEN = ALLNEXTGEN & NEXTGEN[g];
99   end
100 end
101
102 reg GENERATION;
103
104 wire WRITE[processors-1:0][blocks-1:0];
105 wire DONE_EXE[processors-1:0];
106 wire TIME;
107 wire NEXTGEN[processors-1:0];
108
109 reg ANY_WRITE[processors-1:0];
110 reg ANY_GET_CUR_AGENT_BLOCK[processors-1:0];
111 reg ANY_AGENT_READ_INTERN[processors-1:0];
112 `ifndef PROCESSORS1
113 reg ANY_AGENT_READ_EXTERN[processors-1:0];
114 `endif
115
116 wire SEND_AGENT[processors-1:0];
117 wire GET_NEXT_AGENT[processors-1:0];
118 wire AGENT_ADR[processors-1:0];
119 wire AGENT_READ[processors-1:0][blocks-1:0];
120
121 `ifdef AGENT_CHECK_CELL
122  wire CHECK_CELL[processors-1:0];
123 `endif
124
125 wire GET_ID[processors-1:0];
126
127 wire AGENT_READ_INTERN[processors-1:0][blocks-1:0];
128 `ifndef PROCESSORS1
129 wire AGENT_READ_EXTERN[processors-1:0][blocks-1:0];
```

```
130 `endif
131
132 wire AGENT_ADR_INTERN[processors-1:0];
133 `ifndef PROCESSORS1
134 wire AGENT_ADR_EXTERN[processors-1:0];
135 `endif
136
137 wire GET_CUR_AGENT_BLOCK[processors-1:0][blocks-1:0];
138
139
140 reg [log_blocks-1:0]BLOCK_ID[processors-1:0];
141
142
143 integer a;
144 integer b;
145 always@*
146 begin
147  for(a=0;a<processors;a=a+1)
148  begin:DEF_RWWS1
149   for(b=0;b<blocks;b=b+1)
150   begin:DEF_RWWS2
151    if(WRITE[a][b] | GET_CUR_AGENT_BLOCK[a][b] | AGENT_READ_INTERN[a][b]
152    `ifndef PROCESSORS1 | AGENT_READ_EXTERN[a][b] `endif
153    )
154     BLOCK_ID[a] = b;
155
156    if(b==0)
157     ANY_WRITE[a] = WRITE[a][b];
158    else
159     ANY_WRITE[a] = ANY_WRITE[a] | WRITE[a][b];
160
161    if(b==0)
162     ANY_GET_CUR_AGENT_BLOCK[a] = GET_CUR_AGENT_BLOCK[a][b];
163    else
164     ANY_GET_CUR_AGENT_BLOCK[a] = ANY_GET_CUR_AGENT_BLOCK[a]
165                                      | GET_CUR_AGENT_BLOCK[a][b];
166
167    if(b==0)
168     ANY_AGENT_READ_INTERN[a] = AGENT_READ_INTERN[a][b];
169    else
170     ANY_AGENT_READ_INTERN[a] = ANY_AGENT_READ_INTERN[a]
171                                    | AGENT_READ_INTERN[a][b];
172
173    `ifndef PROCESSORS1
174    if(b==0)
175     ANY_AGENT_READ_EXTERN[a] = AGENT_READ_EXTERN[a][b];
176    else
177     ANY_AGENT_READ_EXTERN[a] = ANY_AGENT_READ_EXTERN[a]
178                                    | AGENT_READ_EXTERN[a][b];
179    `endif
180   end
181
182   `ifndef PROCESSORS1
183     EXT_DONE[a] <=
184     `ifdef AGENT_CHECK_CELL C_RETURN[a] | `endif
```

```
185     GET_ID[a] | ALLNEXTGEN_DONE | (SEND_AGENT[a] & (ARBITER_DONE[a] ))
186     | GET_NEXT_AGENT[a] | ANY_WRITE[a] | DELAY_INTERNAL_READ[a]
187     | (~NETWORK_BUSY[a] & (ANY_AGENT_READ_EXTERN[a]
188     | AGENT_ADR_EXTERN[a])) | (TIME & (a==0));
189   `else
190     EXT_DONE[a] <=
191     `ifdef AGENT_CHECK_CELL C_RETURN[a] | `endif
192     GET_ID[a] | ALLNEXTGEN_DONE | (SEND_AGENT[a] & (ARBITER_DONE[a] ))
193     | GET_NEXT_AGENT[a] | ANY_WRITE[a] | DELAY_INTERNAL_READ[a]
194     | (TIME & (a==0));
195   `endif
196
197
198
199   if(DELAY_INTERNAL_READ[a])
200   begin
201    if(AGENT_ADR_INTERN[a])
202    begin
203     if(INTERN_FIELD_DATA[a][link_width-1]==1'b0)
204      EXT_RESULT[a] <= {32{1'b1}};
205     else
206      EXT_RESULT[a] <= INTERN_FIELD_DATA[a][link_width-1-1:0]+0;
207    end
208    else
209    begin
210     EXT_RESULT[a]<=INTERNAL_AGENT_BLOCK[a][(BLOCK_ID[a]*dat_width)+:dat_width];
211    end
212   end
213   else
214   begin
215    if(a==0 & TIME)
216    begin
217     EXT_RESULT[a] <= TIMING;
218    end
219    else
220    begin
221     if(GET_NEXT_AGENT[a])
222     begin
223      EXT_RESULT[a] <=
224       (EXT_DATAA[a][agent_memory_width-1:0] <= AMOUNT_OF_AGENTS_SAVED[a])+0;
225     end
226     else
227     begin
228      `ifndef PROCESSORS1
229      if(~NETWORK_BUSY[a] & (ANY_AGENT_READ_EXTERN[a] | AGENT_ADR_EXTERN[a]))
230      begin
231       if(AGENT_ADR_EXTERN[a] & NETWORK_DATA_OUT[a][link_width-1]==1'b0)
232       begin
233        EXT_RESULT[a] <= {32{1'b1}};
234       end
235       else
236       begin
237        if(AGENT_ADR_EXTERN[a])
238         EXT_RESULT[a] <= NETWORK_DATA_OUT[a][link_width-1-1:0] + 0;
239        else
```

```
240         EXT_RESULT[a] <= NETWORK_DATA_OUT[a] + 0;
241       end
242      end
243      else
244      begin
245      `endif
246       `ifdef AGENT_CHECK_CELL
247       if(CHECK_CELL[a])
248        EXT_RESULT[a] <= C_COND[a] +'b0;
249       else
250       `endif
251        EXT_RESULT[a] <= a;
252      `ifndef PROCESSORS1
253      end
254      `endif
255     end
256    end
257   end
258
259  end
260 end
261
262 reg DELAY_INTERNAL_READ[processors-1:0];
263
264
265 reg EXT_DONE_R[processors-1:0];
266 reg [31:0]EXT_RESULT_R[processors-1:0];
267
268 always@(posedge SYSTEM_CLK)
269 begin
270  for(a=0;a<processors;a=a+1)
271  begin:DEF_RWWC
272   DELAY_INTERNAL_READ[a]<=ANY_GET_CUR_AGENT_BLOCK[a] | ANY_AGENT_READ_INTERN[a]
273                          | (AGENT_ADR_INTERN[a] & ARBITER_DONE[a]);
274
275   EXT_DONE_R[a] <= EXT_DONE[a];
276   EXT_RESULT_R[a] <= EXT_RESULT[a];
277  end
278 end
279
280 `ifdef AGENT_CHECK_CELL
281
282 reg C_RETURN [processors-1:0];
283
284 reg [4:0]CHECK_FSM[processors-1:0];
285 reg C_BLOCK[processors-1:0];
286
287 reg [31:0] C_DAT_A [processors-1:0];
288 reg C_ENA [processors-1:0];
289 reg [7:0] C_N [processors-1:0];
290 reg C_START [processors-1:0];
291
292 reg [31:0] CELL_ADR [processors-1:0];
293 reg [15:0] MAXY [processors-1:0];
294 reg [3:0] C_COND [processors-1:0];
```

```
295
296 always@(posedge SYSTEM_CLK)                                    //Vier-Nachbarn
297 begin
298  for(a=0;a<processors;a=a+1)
299  begin:DEF_CHECK_CELL
300
301   case(CHECK_FSM[a])
302   'h0:
303   begin
304    C_RETURN[a]<=1'b0;
305    C_COND[a]<='h0;
306
307    if(CHECK_CELL[a])
308    begin
309     C_BLOCK[a]<=1'b1;
310     CHECK_FSM[a]<='h2;
311     CELL_ADR[a]<=EXT_DATAA[a];
312     MAXY[a]<=EXT_DATAB[a][15:0];
313
314
315     C_N[a]     <= CMD_AGENT_ADR;
316     C_DAT_A[a] <= EXT_DATAA[a]-EXT_DATAB[a][15:0];
317     C_START[a] <= 1'b1;
318     C_ENA[a]   <= 1'b1;
319
320    end
321    else
322    begin
323     C_BLOCK[a]<=1'b0;
324     CHECK_FSM[a]<='h0;
325     C_START[a]<=1'b0;
326     C_ENA[a]<=1'b0;
327    end
328   end
329
330   'h2:
331   begin
332    C_START[a] <= 1'b0;
333
334    if(EXT_DONE_R[a])
335    begin
336     if(EXT_RESULT_R[a]!={32{1'b1}})
337     begin
338      CHECK_FSM[a]<='h3;
339      C_DAT_A[a] <= EXT_RESULT_R[a];
340     end
341     else
342     begin
343      CHECK_FSM[a]<='h5;
344     end
345
346     C_ENA[a] <= 1'b0;
347    end
348    else
349    begin
```

```
350       C_ENA[a] <= ~EXT_DONE[a];
351       CHECK_FSM[a]<='h2;
352      end
353     end
354
355     'h3:
356     begin
357      C_N[a]     <= CMD_AGENT_READ_BLOCK1;
358      C_START[a] <= 1'b1;
359      C_ENA[a]   <= 1'b1;
360
361      CHECK_FSM[a]<='h4;
362     end
363
364     'h4:
365     begin
366      C_START[a] <= 1'b0;
367
368      if(EXT_DONE_R[a])
369      begin
370       if(EXT_RESULT_R[a]==CELL_AGENT_S)
371        C_COND[a]<=C_COND[a]+'h1;
372
373       C_ENA[a]<=1'b0;
374       CHECK_FSM[a]<='h5;
375      end
376      else
377      begin
378       C_ENA[a] <= ~EXT_DONE[a];
379       CHECK_FSM[a]<='h4;
380      end
381     end
382
383
384     'h5:
385     begin
386      C_N[a]     <= CMD_AGENT_ADR;
387      C_DAT_A[a] <= CELL_ADR[a]+MAXY[a];
388      C_START[a] <= 1'b1;
389      C_ENA[a]   <= 1'b1;
390
391      CHECK_FSM[a]<='h6;
392     end
393
394     'h6:
395     begin
396      C_START[a] <= 1'b0;
397
398      if(EXT_DONE_R[a])
399      begin
400       if(EXT_RESULT_R[a]!={32{1'b1}})
401       begin
402        CHECK_FSM[a]<='h7;
403        C_DAT_A[a] <= EXT_RESULT_R[a];
404       end
```

```
405     else
406     begin
407      CHECK_FSM[a]<='h9;
408     end
409
410     C_ENA[a] <= 1'b0;
411    end
412    else
413    begin
414     C_ENA[a] <= ~EXT_DONE[a];
415     CHECK_FSM[a]<='h6;
416    end
417   end
418
419   'h7:
420   begin
421    C_N[a]     <= CMD_AGENT_READ_BLOCK1;
422    C_START[a] <= 1'b1;
423    C_ENA[a]   <= 1'b1;
424
425    CHECK_FSM[a]<='h8;
426   end
427
428   'h8:
429   begin
430    C_START[a] <= 1'b0;
431
432    if(EXT_DONE_R[a])
433    begin
434     if(EXT_RESULT_R[a]==CELL_AGENT_N)
435      C_COND[a]<=C_COND[a]+'h1;
436
437     C_ENA[a]<=1'b0;
438     CHECK_FSM[a]<='h9;
439    end
440    else
441    begin
442     C_ENA[a] <= ~EXT_DONE[a];
443     CHECK_FSM[a]<='h8;
444    end
445   end
446
447   'h9:
448   begin
449    C_N[a]     <= CMD_AGENT_ADR;
450    C_DAT_A[a] <= CELL_ADR[a]+1;
451    C_START[a] <= 1'b1;
452    C_ENA[a]   <= 1'b1;
453
454    CHECK_FSM[a]<='hA;
455   end
456
457   'hA:
458   begin
459    C_START[a] <= 1'b0;
```

```
460
461     if(EXT_DONE_R[a])
462     begin
463      if(EXT_RESULT_R[a]!={32{1'b1}})
464      begin
465       CHECK_FSM[a]<='hB;
466       C_DAT_A[a] <= EXT_RESULT_R[a];
467      end
468      else
469      begin
470       CHECK_FSM[a]<='hD;
471      end
472
473      C_ENA[a] <= 1'b0;
474     end
475     else
476     begin
477      C_ENA[a] <= ~EXT_DONE[a];
478      CHECK_FSM[a]<='hA;
479     end
480    end
481
482    'hB:
483    begin
484     C_N[a]     <= CMD_AGENT_READ_BLOCK1;
485     C_START[a] <= 1'b1;
486     C_ENA[a]   <= 1'b1;
487
488     CHECK_FSM[a]<='hC;
489    end
490
491    'hC:
492    begin
493     C_START[a] <= 1'b0;
494
495     if(EXT_DONE_R[a])
496     begin
497      if(EXT_RESULT_R[a]==CELL_AGENT_W)
498       C_COND[a]<=C_COND[a]+'h1;
499
500      C_ENA[a]<=1'b0;
501      CHECK_FSM[a]<='hD;
502     end
503     else
504     begin
505      C_ENA[a] <= ~EXT_DONE[a];
506      CHECK_FSM[a]<='hC;
507     end
508    end
509
510
511    'hD:
512    begin
513     C_N[a]     <= CMD_AGENT_ADR;
514     C_DAT_A[a] <= CELL_ADR[a]-1;
```

```
515     C_START[a] <= 1'b1;
516     C_ENA[a]   <= 1'b1;
517
518     CHECK_FSM[a]<='hE;
519   end
520
521   'hE:
522   begin
523     C_START[a] <= 1'b0;
524
525     if(EXT_DONE_R[a])
526     begin
527       if(EXT_RESULT_R[a]!={32{1'b1}})
528       begin
529         CHECK_FSM[a]<='hF;
530         C_DAT_A[a] <= EXT_RESULT_R[a];
531       end
532       else
533       begin
534         CHECK_FSM[a]<='h11;
535       end
536
537       C_ENA[a] <= 1'b0;
538     end
539     else
540     begin
541       C_ENA[a] <= ~EXT_DONE[a];
542       CHECK_FSM[a]<='hE;
543     end
544   end
545
546   'hF:
547   begin
548     C_N[a]     <= CMD_AGENT_READ_BLOCK1;
549     C_START[a] <= 1'b1;
550     C_ENA[a]   <= 1'b1;
551
552     CHECK_FSM[a]<='h10;
553   end
554
555   'h10:
556   begin
557     C_START[a] <= 1'b0;
558
559     if(EXT_DONE_R[a])
560     begin
561       if(EXT_RESULT_R[a]==CELL_AGENT_E)
562         C_COND[a]<=C_COND[a]+'h1;
563
564       C_ENA[a]<=1'b0;
565       CHECK_FSM[a]<='h11;
566     end
567     else
568     begin
569       C_ENA[a] <= ~EXT_DONE[a];
```

```
570       CHECK_FSM[a]<='h10;
571      end
572     end
573
574     'h11:
575     begin
576      if(!EXT_DONE_R[a] & !EXT_DONE[a])
577      begin
578       C_BLOCK[a]<=1'b0;
579       C_RETURN[a]<=1'b1;
580
581       CHECK_FSM[a]<='h12;
582      end
583      else
584      begin
585       C_BLOCK[a]<=1'b1;
586       C_RETURN[a]<=1'b0;
587
588       CHECK_FSM[a]<='h11;
589      end
590     end
591
592     'h12:
593     begin
594      C_RETURN[a]<=1'b0;
595      if(!CHECK_CELL[a])
596       CHECK_FSM[a]<='h0;
597      else
598       CHECK_FSM[a]<='h12;
599     end
600
601
602     endcase
603    end
604   end
605
606
607   `endif
608
609   genvar i;
610   generate
611    for(i=0;i<processors;i=i+1)
612    begin:DEF_RWW
613     assign GET_ID[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_ID);
614
615     `ifdef AGENT_CHECK_CELL
616      assign CHECK_CELL[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_CHECK_CELL);
617     `endif
618
619     assign AGENT_ADR[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_AGENT_ADR);
620
621     `ifndef PROCESSORS1
622      assign AGENT_ADR_INTERN[i] = AGENT_ADR[i]
623      & (EXT_DATAA[i][memory_adr_width+log_num_cells-1-1:memory_adr_width-1]==i);
624     `else
```

```
625     assign AGENT_ADR_INTERN[i] = AGENT_ADR[i];
626   `endif
627
628   `ifndef PROCESSORS1
629     assign AGENT_ADR_EXTERN[i] = AGENT_ADR[i]
630      & (EXT_DATAA[i][memory_adr_width+log_num_cells-1-1:memory_adr_width-1]!=i);
631   `endif
632
633   if(blocks>=7) assign AGENT_READ[i][6] = EXT_CLK_EN[i]
634         & (EXT_N[i]==CMD_AGENT_READ_BLOCK6);
635   if(blocks>=6) assign AGENT_READ[i][5] = EXT_CLK_EN[i]
636         & (EXT_N[i]==CMD_AGENT_READ_BLOCK5);
637   if(blocks>=5) assign AGENT_READ[i][4] = EXT_CLK_EN[i]
638         & (EXT_N[i]==CMD_AGENT_READ_BLOCK4);
639   if(blocks>=4) assign AGENT_READ[i][3] = EXT_CLK_EN[i]
640         & (EXT_N[i]==CMD_AGENT_READ_BLOCK3);
641   if(blocks>=3) assign AGENT_READ[i][2] = EXT_CLK_EN[i]
642         & (EXT_N[i]==CMD_AGENT_READ_BLOCK2);
643   if(blocks>=2) assign AGENT_READ[i][1] = EXT_CLK_EN[i]
644         & (EXT_N[i]==CMD_AGENT_READ_BLOCK1);
645   if(blocks>=1) assign AGENT_READ[i][0] = EXT_CLK_EN[i]
646         & (EXT_N[i]==CMD_AGENT_READ_BLOCK0);
647
648
649   `ifndef PROCESSORS1
650   if(blocks>=7) assign AGENT_READ_INTERN[i][6] = AGENT_READ[i][6]
651    & (EXT_DATAA[i]
652    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
653   if(blocks>=6) assign AGENT_READ_INTERN[i][5] = AGENT_READ[i][5]
654    & (EXT_DATAA[i]
655    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
656   if(blocks>=5) assign AGENT_READ_INTERN[i][4] = AGENT_READ[i][4]
657    & (EXT_DATAA[i]
658    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
659   if(blocks>=4) assign AGENT_READ_INTERN[i][3] = AGENT_READ[i][3]
660    & (EXT_DATAA[i]
661    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
662   if(blocks>=3) assign AGENT_READ_INTERN[i][2] = AGENT_READ[i][2]
663    & (EXT_DATAA[i]
664    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
665   if(blocks>=2) assign AGENT_READ_INTERN[i][1] = AGENT_READ[i][1]
666    & (EXT_DATAA[i]
667    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
668   if(blocks>=1) assign AGENT_READ_INTERN[i][0] = AGENT_READ[i][0]
669    & (EXT_DATAA[i]
670    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]==i);
671   `else
672   if(blocks>=7) assign AGENT_READ_INTERN[i][6] = AGENT_READ[i][6];
673   if(blocks>=6) assign AGENT_READ_INTERN[i][5] = AGENT_READ[i][5];
674   if(blocks>=5) assign AGENT_READ_INTERN[i][4] = AGENT_READ[i][4];
675   if(blocks>=4) assign AGENT_READ_INTERN[i][3] = AGENT_READ[i][3];
676   if(blocks>=3) assign AGENT_READ_INTERN[i][2] = AGENT_READ[i][2];
677   if(blocks>=2) assign AGENT_READ_INTERN[i][1] = AGENT_READ[i][1];
678   if(blocks>=1) assign AGENT_READ_INTERN[i][0] = AGENT_READ[i][0];
679   `endif
```

```
680
681
682   `ifndef PROCESSORS1
683   if(blocks>=7) assign AGENT_READ_EXTERN[i][6] = AGENT_READ[i][6]
684    & (EXT_DATAA[i]
685    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
686   if(blocks>=6) assign AGENT_READ_EXTERN[i][5] = AGENT_READ[i][5]
687    & (EXT_DATAA[i]
688    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
689   if(blocks>=5) assign AGENT_READ_EXTERN[i][4] = AGENT_READ[i][4]
690    & (EXT_DATAA[i]
691    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
692   if(blocks>=4) assign AGENT_READ_EXTERN[i][3] = AGENT_READ[i][3]
693    & (EXT_DATAA[i]
694    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
695   if(blocks>=3) assign AGENT_READ_EXTERN[i][2] = AGENT_READ[i][2]
696    & (EXT_DATAA[i]
697    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
698   if(blocks>=2) assign AGENT_READ_EXTERN[i][1] = AGENT_READ[i][1]
699    & (EXT_DATAA[i]
700    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
701   if(blocks>=1) assign AGENT_READ_EXTERN[i][0] = AGENT_READ[i][0]
702    & (EXT_DATAA[i]
703    [agent_memory_width+log_num_cells-1-1:agent_memory_width-1]!=i);
704   `endif
705
706   assign GET_NEXT_AGENT[i] = EXT_CLK_EN[i]
707                            & (EXT_N[i]==CMD_GET_NEXT_AGENT) & EXT_START[i];
708
709   if(blocks>=7) assign GET_CUR_AGENT_BLOCK[i][6]
710    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK6);
711   if(blocks>=6) assign GET_CUR_AGENT_BLOCK[i][5]
712    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK5);
713   if(blocks>=5) assign GET_CUR_AGENT_BLOCK[i][4]
714    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK4);
715   if(blocks>=4) assign GET_CUR_AGENT_BLOCK[i][3]
716    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK3);
717   if(blocks>=3) assign GET_CUR_AGENT_BLOCK[i][2]
718    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK2);
719   if(blocks>=2) assign GET_CUR_AGENT_BLOCK[i][1]
720    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK1);
721   if(blocks>=1) assign GET_CUR_AGENT_BLOCK[i][0]
722           = EXT_CLK_EN[i] & (EXT_N[i]==CMD_GET_CUR_AGENT_BLOCK0);
723
724   assign SEND_AGENT[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_SEND_AGENT);
725
726   if(blocks>=7) assign WRITE[i][6]
727    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK6) & EXT_START[i];
728   if(blocks>=6) assign WRITE[i][5]
729    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK5) & EXT_START[i];
730   if(blocks>=5) assign WRITE[i][4]
731    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK4) & EXT_START[i];
732   if(blocks>=4) assign WRITE[i][3]
733    = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK3) & EXT_START[i];
734   if(blocks>=3) assign WRITE[i][2]
```

```
735     = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK2) & EXT_START[i];
736   if(blocks>=2) assign WRITE[i][1]
737     = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK1) & EXT_START[i];
738   if(blocks>=1) assign WRITE[i][0]
739     = EXT_CLK_EN[i] & (EXT_N[i]==CMD_WRITE_BLOCK0) & EXT_START[i];
740
741   assign DONE_EXE[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_DONE_EXE);
742   if(i==0)
743     assign TIME = EXT_CLK_EN[i] & (EXT_N[i]==CMD_TIME);
744   assign NEXTGEN[i] = EXT_CLK_EN[i] & (EXT_N[i]==CMD_NEXTGEN);
745
746  end
747 endgenerate
748
749
750 reg [31:0] TIMING;
751 always@(posedge SYSTEM_CLK)
752 begin
753  if(TIME)
754   TIMING <= 'b0;
755  else
756   TIMING <= TIMING + 'b1;
757 end
758
759 genvar v;
760
761 wire [memory_adr_width-1-1:0]NETWORK_ADDRESS [processors-1:0];
762 `ifndef PROCESSORS1
763 wire [dat_width-1:0] NETWORK_DATA_OUT [processors-1:0];
764 wire [log_num_cells-1:0] NETWORK_ID [processors-1:0];
765 wire NETWORK_SELECT_DATAIN [processors-1:0];
766
767 wire  [log_blocks-1:0] NETWORK_BLOCK_ID [processors-1:0];
768
769 wire NETWORK_REQUEST [processors-1:0];
770 wire NETWORK_BUSY [processors-1:0];
771
772 reg NETWORK_BUSY_TO_REQUEST_WAIT [processors-1:0];
773 integer c;
774 always@(posedge SYSTEM_CLK)
775 begin
776  for(c=0;c<processors;c=c+1)
777  begin:DEF_NETWAIT
778   NETWORK_BUSY_TO_REQUEST_WAIT[c]<=NETWORK_REQUEST[c];
779  end
780 end
781
782 wire [((dat_width*processors)-1):0] total_din;
783 wire [((dat_width*processors)-1):0] total_dout;
784 wire [(((memory_adr_width-1+log_num_cells+log_blocks+1)*processors)-1)
785      :0] total_ain;
786 wire [(((memory_adr_width-1+log_num_cells+log_blocks+1)*processors)-1)
787      :0] total_aout;
788 wire [processors-1:0] total_req;
789 wire [processors-1:0] total_busy;
```

```
790 wire [processors-1:0] total_nextreq;
791 wire [processors-1:0] total_nextbusy;
792
793 wire [((memory_adr_width-1+log_num_cells+log_blocks)-1):0]
794      ain_select [processors-1:0];
795
796 generate
797  for(v=0;v<processors;v=v+1)
798  begin:DEF_nets
799   assign total_din[((v+1)*dat_width)-1:v*dat_width] =
800    ~NETWORK_SELECT_DATAIN[v] ? EXTERN_FIELD_DATA[v] :
801    EXTERNAL_AGENT_BLOCK[v][(NETWORK_BLOCK_ID[v]*dat_width) +: dat_width];
802
803   assign NETWORK_DATA_OUT[v] = total_dout[((v+1)*dat_width)-1:v*dat_width];
804
805   assign ain_select[v] = ANY_AGENT_READ_EXTERN[v] ?
806   {EXT_DATAA[v][agent_memory_width+log_num_cells-1-1:agent_memory_width-1],
807   {(memory_adr_width-1)-(agent_memory_width-1){1'b0}},
808   EXT_DATAA[v][agent_memory_width-1-1:0],BLOCK_ID[v]}
809   : {EXT_DATAA[v][memory_adr_width+log_num_cells-2:0],BLOCK_ID[v]};
810
811   assign total_ain[((v+1)*(memory_adr_width-1+log_num_cells+log_blocks+1))-1
812          :v*(memory_adr_width-1+log_num_cells+log_blocks+1)]
813          = {ain_select[v],ANY_AGENT_READ_EXTERN[v]};
814   assign {NETWORK_ID[v],NETWORK_ADDRESS[v],
815    NETWORK_BLOCK_ID[v],NETWORK_SELECT_DATAIN[v]}
816          =total_aout[((v+1)*(memory_adr_width-1+log_num_cells+log_blocks+1))-1:
817          v*(memory_adr_width-1+log_num_cells+log_blocks+1)];
818
819   assign total_req[v] = ANY_AGENT_READ_EXTERN[v] | AGENT_ADR_EXTERN[v];
820
821   assign NETWORK_REQUEST[v] = total_nextreq[v];
822   assign NETWORK_BUSY[v] = total_busy[v];
823   assign total_nextbusy[v] = ~NETWORK_BUSY_TO_REQUEST_WAIT[v];
824  end
825 endgenerate
826
827
828 NETWORK_INST #(.adr_width(memory_adr_width-1+log_blocks+1),
829    .data_width(dat_width),
830    .num_cells(processors),
831    .log_num_cells(log_num_cells))
832    net(
833    .CLOCK(SYSTEM_CLK),
834    .din(total_din),
835    .dout(total_dout),
836    .req(total_req),
837    .busy(total_busy),
838    .nextreq(total_nextreq),
839    .nextbusy(total_nextbusy),
840    .ain(total_ain),
841    .aout(total_aout)
842    );
843 `endif
844
```

```
845 wire EXT_CLK_EN[processors-1:0];
846 wire [31:0] EXT_DATAA[processors-1:0];
847 wire [31:0] EXT_DATAB[processors-1:0];
848 reg EXT_DONE[processors-1:0];
849 wire EXT_RESET[processors-1:0];
850 reg [31:0]EXT_RESULT[processors-1:0];
851 wire EXT_START[processors-1:0];
852 wire [7:0] EXT_N [processors-1:0];
853
854
855 wire [processors-1:0] W_EXT_CLK_EN;
856 wire [((32*processors)-1):0] W_EXT_DATAA;
857 wire [((32*processors)-1):0] W_EXT_DATAB;
858 wire [processors-1:0] W_EXT_DONE;
859 wire [(8*processors)-1:0] W_EXT_N;
860 wire [processors-1:0] W_EXT_RESET;
861 wire [((32*processors)-1):0] W_EXT_RESULT;
862 wire [processors-1:0] W_EXT_START;
863
864
865 generate
866  for(v=0;v<processors;v=v+1)
867  begin:DEF_ProcessorNet2
868   `ifndef AGENT_CHECK_CELL
869   assign EXT_CLK_EN[v]=W_EXT_CLK_EN[v];
870   assign EXT_DATAA[v]=W_EXT_DATAA[((v+1)*32)-1:v*32];
871   assign EXT_DATAB[v]=W_EXT_DATAB[((v+1)*32)-1:v*32];
872   assign W_EXT_DONE[v]=EXT_DONE[v];
873   assign EXT_N[v]=W_EXT_N[((v+1)*8)-1:v*8];
874   assign EXT_RESET[v]=W_EXT_RESET[v];
875   assign W_EXT_RESULT[((v+1)*32)-1:v*32]=EXT_RESULT[v];
876   assign EXT_START[v]=W_EXT_START[v];
877   `else
878   assign EXT_CLK_EN[v]=C_BLOCK[v] ? C_ENA[v] : W_EXT_CLK_EN[v];
879   assign EXT_DATAA[v]=C_BLOCK[v] ? C_DAT_A[v] : W_EXT_DATAA[((v+1)*32)-1:v*32];
880   assign EXT_DATAB[v]=W_EXT_DATAB[((v+1)*32)-1:v*32];
881   assign W_EXT_DONE[v]=C_BLOCK[v] ? 1'b0 : EXT_DONE[v];
882   assign EXT_N[v]=C_BLOCK[v] ? C_N[v] : W_EXT_N[((v+1)*8)-1:v*8];
883   assign EXT_RESET[v]=W_EXT_RESET[v];
884   assign W_EXT_RESULT[((v+1)*32)-1:v*32]=EXT_RESULT[v];
885   assign EXT_START[v]=C_BLOCK[v] ? C_START[v] : W_EXT_START[v];
886   `endif
887  end
888 endgenerate
889
890
891 PROCESSORS #(.processors(processors))
892 DUT(
893 .CLK(SYSTEM_CLK),
894 .RESET(RESET | RESETNIOS),
895 .W_EXT_CLK_EN(W_EXT_CLK_EN),
896 .W_EXT_DATAA(W_EXT_DATAA),
897 .W_EXT_DATAB(W_EXT_DATAB),
898 .W_EXT_DONE(W_EXT_DONE),
899 .W_EXT_N(W_EXT_N),
```

```
900  .W_EXT_RESET(W_EXT_RESET),
901  .W_EXT_RESULT(W_EXT_RESULT),
902  .W_EXT_START(W_EXT_START)
903  );
904
905  reg [agent_memory_width-1-1:0] READ_NEXT_AGENT_ADR [processors-1:0];
906  reg [agent_memory_width-1-1:0] AMOUNT_OF_AGENTS_SAVED [processors-1:0];
907
908  always@(posedge SYSTEM_CLK)
909  begin
910   for(a=0;a<processors;a=a+1)
911   begin:DEF_READ_NEXT
912    if(GET_NEXT_AGENT[a]
913    begin
914     READ_NEXT_AGENT_ADR[a]=EXT_DATAA[a][agent_memory_width-1-1:0];
915    end
916   end
917  end
918
919  wire [memory_adr_width-2:0] WRITE_CACHE_ADR [processors-1:0];
920  wire [log_num_cells-1:0] WRITE_ID [processors-1:0];
921  wire [link_width-1-1:0] WRITE_AGENT_LINK [processors-1:0];
922
923  wire WRITE_INTERN [processors-1:0];
924
925  generate
926   for(v=0;v<processors;v=v+1)
927   begin:DEF_WRITE2
928    assign {WRITE_ID[v],WRITE_CACHE_ADR[v]}
929     = EXT_DATAA[v][memory_adr_width-2+log_num_cells:0];
930    assign WRITE_AGENT_LINK[v] = {v,READ_NEXT_AGENT_ADR[v]};
931    assign WRITE_INTERN[v] = (WRITE_ID[v]==v);
932   end
933  endgenerate
934
935  reg [(blocks*dat_width)-1:0] WRITE_CACHE_CELL [processors-1:0];
936
937  reg [memory_adr_width-1:0] CELL_FIELD_ADDRESS [processors-1:0];
938  reg CELL_FIELD_WRITE_ACTIVE [processors-1:0];
939  reg [link_width-1:0] CELL_FIELD_WRITE_DATA [processors-1:0];
940  reg ARBITER_DONE [processors-1:0];
941
942  reg [memory_adr_width-1:0] CF_ADR_CACHE [processors-1:0];
943  reg [link_width-1:0] CF_DATA_CACHE [processors-1:0];
944  reg [log_num_cells-1:0] CF_ID [processors-1:0];
945  reg CF_OCCUPIED [processors-1:0];
946  reg ANY_CF_OCCUPIED;
947
948  always@*
949  begin
950   for(a=0;a<processors;a=a+1)
951   begin:DEF_CF_OCC
952    if(a==0)
953     ANY_CF_OCCUPIED = CF_OCCUPIED[a];
954    else
```

```
955     ANY_CF_OCCUPIED = ANY_CF_OCCUPIED | CF_OCCUPIED[a];
956    end
957   end
958
959   always@(posedge SYSTEM_CLK)
960   begin
961    for(a=0;a<processors;a=a+1)
962    begin:DEF_WRITE
963     if(ANY_WRITE[a])
964     begin
965      WRITE_CACHE_CELL[a][(BLOCK_ID[a]*dat_width) +: dat_width]
966      <= EXT_DATAB[a][dat_width-1:0];
967     end
968
969     if(!ARBITER_DONE[a]&((WRITE_INTERN[a]&SEND_AGENT[a])|AGENT_ADR_INTERN[a]))
970     begin
971      if(!AGENT_ADR_INTERN[a])
972       CELL_FIELD_ADDRESS[a] <= {~GENERATION,WRITE_CACHE_ADR[a]};
973      else
974       CELL_FIELD_ADDRESS[a] <= {GENERATION,EXT_DATAA[a][memory_adr_width-1-1:0]};
975
976      CELL_FIELD_WRITE_DATA[a] <= {1'b1,WRITE_AGENT_LINK[a]};
977      CELL_FIELD_WRITE_ACTIVE[a] <= WRITE_INTERN[a] & SEND_AGENT[a];
978
979      ARBITER_DONE[a]<=1'b1;
980     end
981     else
982     begin
983      if(!ARBITER_DONE[a] & !CF_OCCUPIED[(a+1)%processors]
984         & !WRITE_INTERN[a] & SEND_AGENT[a])
985      begin
986       CF_ADR_CACHE[(a+1)%processors] <= {~GENERATION,WRITE_CACHE_ADR[a]};
987       CF_DATA_CACHE[(a+1)%processors] <= {1'b1,WRITE_AGENT_LINK[a]};
988       CF_ID[(a+1)%processors] <= WRITE_ID[a];
989       CF_OCCUPIED[(a+1)%processors] <= 1'b1;
990
991       CELL_FIELD_WRITE_ACTIVE[a] <= 1'b0;
992       ARBITER_DONE[a]<=1'b1;
993      end
994      else
995      begin
996       if(CF_OCCUPIED[a])
997       begin
998        if(CF_ID[a]==a)
999        begin
1000        CELL_FIELD_ADDRESS[a] <= CF_ADR_CACHE[a];
1001        CELL_FIELD_WRITE_DATA[a] <= CF_DATA_CACHE[a];
1002        CELL_FIELD_WRITE_ACTIVE[a] <= 1'b1;
1003
1004        CF_OCCUPIED[a] <= 1'b0;
1005       end
1006       else
1007       begin
1008        if(CF_OCCUPIED[(a+1)%processors]==1'b0)
1009        begin
```

```
1010        CF_ADR_CACHE[(a+1)%processors] <= CF_ADR_CACHE[a];
1011        CF_DATA_CACHE[(a+1)%processors] <= CF_DATA_CACHE[a];
1012        CF_ID[(a+1)%processors] <= CF_ID[a];
1013        CF_OCCUPIED[(a+1)%processors] <= 1'b1;
1014
1015        CF_OCCUPIED[a] <= 1'b0;
1016       end
1017       CELL_FIELD_WRITE_ACTIVE[a] <= 1'b0;
1018      end
1019      ARBITER_DONE[a]<=1'b0;
1020     end
1021     else
1022     begin
1023      CELL_FIELD_WRITE_ACTIVE[a] <= 1'b0;
1024      ARBITER_DONE[a]<=1'b0;
1025     end
1026    end
1027   end
1028
1029  end
1030 end
1031
1032
1033
1034 reg[1:0] FSM_DELETE;
1035 reg [memory_adr_width-1-1:0] ADR_DEL;
1036 reg ADR_WRITE;
1037 reg ALLNEXTGEN_DONE;
1038
1039
1040 always@(posedge SYSTEM_CLK)
1041 begin
1042
1043 case(FSM_DELETE)
1044
1045  0:
1046  begin
1047   if(ALLNEXTGEN & !ANY_CF_OCCUPIED)
1048   begin
1049    FSM_DELETE<='h1;
1050    ADR_DEL<=0;
1051    ADR_WRITE<=1;
1052   end
1053   else
1054   begin
1055    FSM_DELETE<='h0;
1056    ADR_WRITE<=0;
1057   end
1058   ALLNEXTGEN_DONE<=0;
1059  end
1060
1061  1:
1062  begin
1063   if(ADR_DEL<{memory_adr_width-1{1'b1}})
1064   begin
```

```
1065     ADR_DEL<=ADR_DEL+1;
1066     ADR_WRITE<=EXT_DATAB[0][0];
1067     FSM_DELETE<='h1;
1068     ALLNEXTGEN_DONE<=0;
1069    end
1070    else
1071    begin
1072     ADR_DEL<=0;
1073     ADR_WRITE<=0;
1074     FSM_DELETE<='h2;
1075     ALLNEXTGEN_DONE<=1;
1076     GENERATION<=~GENERATION;
1077    end
1078   end
1079
1080   2:
1081   begin
1082    FSM_DELETE<='h0;
1083    ADR_WRITE<=0;
1084    ALLNEXTGEN_DONE<=0;
1085   end
1086  endcase
1087
1088
1089  end
1090
1091
1092  wire [(blocks*dat_width)-1:0] INTERNAL_AGENT_BLOCK [processors-1:0];
1093  wire [(blocks*dat_width)-1:0] EXTERNAL_AGENT_BLOCK [processors-1:0];
1094
1095  wire [link_width-1:0] INTERN_FIELD_DATA [processors-1:0];
1096  wire [link_width-1:0] EXTERN_FIELD_DATA [processors-1:0];
1097
1098  genvar Mem1;
1099  genvar BLKS;
1100  generate
1101   for(Mem1=0;Mem1<processors;Mem1=Mem1+1)
1102   begin:DEF_Memory1
1103    CELL_FIELD_MEM CELL_FIELD_MEM_inst (
1104     .address_a ({GENERATION,NETWORK_ADDRESS[Mem1][memory_adr_width-1-1:0]}),
1105     .address_b (ADR_WRITE ? {GENERATION,ADR_DEL} : CELL_FIELD_ADDRESS[Mem1]),
1106     .clock (SYSTEM_CLK),
1107     .data_a (),
1108     .data_b (ADR_WRITE ? {link_width{1'b0}} : CELL_FIELD_WRITE_DATA[Mem1]),
1109     .wren_a (1'b0),
1110     .wren_b (ADR_WRITE | CELL_FIELD_WRITE_ACTIVE[Mem1]),
1111     .q_a (EXTERN_FIELD_DATA[Mem1]),
1112     .q_b (INTERN_FIELD_DATA[Mem1])
1113     );
1114
1115    DATAMEM DATAMEM_inst (
1116     .address_a ({GENERATION,NETWORK_ADDRESS[Mem1][agent_memory_width-1-1:0]}),
1117     .address_b (ANY_AGENT_READ_INTERN[Mem1] ?
1118     {GENERATION,EXT_DATAA[Mem1][agent_memory_width-1-1:0]}
1119     :({(SEND_AGENT[Mem1]?~GENERATION : GENERATION),READ_NEXT_AGENT_ADR[Mem1]})),
```

```
1120    .clock (SYSTEM_CLK),
1121    .data_a (),
1122    .data_b (WRITE_CACHE_CELL[Mem1]),
1123    .wren_a (1'b0),
1124    .wren_b (SEND_AGENT[Mem1]),
1125    .q_a (EXTERNAL_AGENT_BLOCK[Mem1]),
1126    .q_b (INTERNAL_AGENT_BLOCK[Mem1])
1127    );
1128  end
1129 endgenerate
1130
1131 PLL PLL_inst (
1132 .inclk0 (CLK),
1133 .c0(SYSTEM_CLK)
1134 );
1135
1136 endmodule
```

Quellcode 9.12: Implementierung der DAMA

Literaturverzeichnis

[Alf96] Alfke, Peter: *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators*. http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf. Version: July 1996. – Xilinx Application Note Version 1.1

[Alta] Altera Corporation: *Cyclone II Device Handbook, Volume 1*. http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf

[Altb] Altera Corporation: *Nios Embedded Processor 16-Bit Programmer's Reference Manual*. http://www.altera.com/literature/manual/mnl_nios_programmers16.pdf

[Altc] Altera Corporation: *Nios Embedded Processor Software Development Reference Manual*. http://www.altera.com/literature/manual/mnl_niossft.pdf

[Altd] Altera Corporation: *NIOS II Custom Instruction User Guide*. http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

[Alte] Altera Corporation: *NIOS II Processor Reference Handbook*. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[Altf] Altera Corporation: *NIOS II Software Developer's Handbook*. http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf

[Altg] Altera Corporation: *Quartus II Handbook Version 8.1*

[Alth] Altera Corporation: *Stratix II Device Handbook, Volume 1&2*. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf

[Alti] Altera Corporation: *Stratix V Device Handbook, Volume 1&2&3*. http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf

[Alt08] Altera Corporation: *Simulating Nios II Embedded Processor Designs*, November 2008. http://www.altera.com/literature/an/an351.pdf. – AN 351

[Alt10a] Altera Corporation: *Creating Multiprocessor Nios II Systems Tutorial*, February 2010. http://www.altera.com/literature/tt/tt_nios2_multiprocessor_tutorial.pdf. – Document Version: 1.4

[Alt10b] Altera Corporation: *Website*. http://www.altera.com. Version: 2010

[AMAD07] Aydi, Yassine ; Meftali, Samy ; Abid, Mohamed ; Dekeyser, Jean-Luc: Dynamicity Analysis of Delta MINs for MPSOC Architectures. In: *Conference internationale des sciences et technique de l'automatique (ICM'07)*, 2007

[And95] Anderson, James R.: Simulation and Analysis of Barrier Synchronization Methods / University of Minnesota, High-Performance Parallel Computing Research Group. Version: August 1995. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.224&rep=rep1&type=pdf. 1995 (HPPC-95-04). – Forschungsbericht

[APS10] Aaby, Brandon G. ; Perumalla, Kalyan S. ; Seal, Sudip K.: Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, 2010 (SIMUTools '10). – ISBN 978–963–9799–87–5, 29:1-29:10

[BAM+10] Baklouti, Mouna ; Aydi, Yassine ; Marquet, Philippe ; Dekeyser, Jean-Luc ; Abid, Mohamed: Scalable mpNoC for Massively Parallel Systems - Design and Implementation on FPGA. In: *Journal of Systems Architecture* 56 (2010), July, 278-292. http://dx.doi.org/10.1016/j.sysarc.2010.04.001. – DOI 10.1016/j.sysarc.2010.04.001

[Ban71] Banks, Edwin R.: *Information Processing and Transmission in Cellular Automata*. Cambridge, MA, USA, Massachusetts Institute of Technology, Diss., January 1971. http://www.bottomlayer.com/bottom/banks/banks_thesis_1971.pdf

[Bat68] Batcher, Kenneth E.: Sorting Networks and their Applications. In: *Spring Joint Computer Conference, AFIPS* Bd. 32, 1968, 307-314

[Bat94] Bates, Jospeh: The Role of Emotion in Believable Agents. In: *Communication of the ACM* 37 (1994), Nr. 7, 122-125. http://dx.doi.org/10.1145/176789.176803. – DOI 10.1145/176789.176803. – ISSN 0001–0782

[BKK08] Bansal, Vidit ; Kota, Ramachandra ; Karlapalem, Kamalakar: System Issues in Multi-agent Simulation of Large Crowds. In: *Multi-Agent-Based Simulation VIII* Bd. 5003, Springer Verlag, 2008. – ISSN 0302–9743 (Print) 1611–3349 (Online), 8-19

[BKS99] Barlović, R. ; Kerner, B. S. ; Schreckenberg, Michael: *Eine Theorie des Staus aus dem Nichts*. http://www.ptt.uni-duisburg.de/fileadmin/docs/paper/1999/vdifinal6.pdf. Version: 1999. – Statusseminar - Technische Anwendungen von Erkenntnissen der Nichtlinearen Dynamik, VDI Technologiezentrum Physikalische Technologien Düsseldorf

[BLR94] Bates, Joseph ; Loyall, A. B. ; Reilly, W. S.: An Architecture for Action, Emotion, and Social Behavior. In: Carbonell, J. G. (Hrsg.) ; Siekmann, J. (Hrsg.): *Artificial Social Systems* Bd. 830, Springer Berlin / Heidelberg, 1994. – ISBN 978–3–540–58266–3, 55-68

[BMS03] Bandini, Stefania ; Manzoni, Sara ; Simone, Carla: Situated Cellular Agents in Non-uniform Spaces. In: *Parallel Computing Technologies* Bd. 2763, Springer Berlin / Heidelberg, 2003 (Lecture Notes in Computer Science), 10-19

[BSv06] Bodrožić, Ljiljana ; Stipaničev, Darko ; Šerić, Marijo: Forest fires spread modeling using cellular automata approach. In: *Modern trends in control*

(2006), 23-33. http://www.fesb.hr/~ljiljana/radovi/Ljiljana_Bodrozic_ceepus2006_2.pdf

[BSW85] Beyer, William A. ; Sellers, Peter H. ; Waterman, Michael S.: Stanislaw M. Ulam's Contributions to Theoretical Theory. In: *Letters in Mathematical Physics* 10 (1985), June, Nr. 2-3, 231-242. http://dx.doi.org/10.1007/BF00398163. – DOI 10.1007/BF00398163

[BV09] Bidlo, Michal ; Vasicek, Zdenek: Comparison of the Uniform and Non-Uniform Cellular Automata-Based Approach to the Development of Combinational Circuits. In: Suess, Martin (Hrsg.) ; Arslan, Tughrul (Hrsg.) ; Keymeulen, Didier (Hrsg.) ; Stoica, Adrian (Hrsg.) ; Erdogan, Ahmet T. (Hrsg.) ; Merodio, David (Hrsg.): *Adaptive Hardware and Systems / NASA/ESA*, 2009. – ISBN 978–0–7695–3714–6, 423-430

[Cam] Camara, Daniel: *Non-Uniform Cellular Automata a Review*. http://www.eurecom.fr/~camara/files/NonUniformCellularAutomataReview.pdf. – University of Maryland

[Cas95] Castelfranchi, Cristiano: Guarantees for Autonomy in Cognitive Agent Architecture. In: Carbonell, J. G. (Hrsg.) ; Siekmann, J. (Hrsg.): *Intelligent Agents* Bd. 890, Springer Berlin / Heidelberg, January 1995. – ISBN 978–3–540–58855–9, 56-70

[CCH11] Cui, Lintao ; Chen, Jing ; Hu, Bryan: *Acceleration of Multi-Agent Simulation based on FPGA*. http://research.microsoft.com/en-us/um/cambridge/events/date2011/acceleration%20of%20multi-agent%20simulation%20based%20on%20fpga.pdf. Version: March 2011. – W2 Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing

[CFHZ04] Cong, Jason ; Fan, Yiping ; Han, Guoling ; Zhang, Zhiru: Application-specific instruction generation for configurable processor architectures. In: *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 2004, 183–189

[CGNS02] Chowdhury, Debashish ; Guttal, Vishwesha ; Nishinari, Katsuhiro ; Schadschneider, Andreas: A cellular-automata model of flow in ant-trails: non-monotonic variation of speed with density. In: *Journal de Physique* A35 (2002), Nr. L573, 7. http://xxx.lanl.gov/pdf/cond-mat/0201207v3

[CLRS01] Cormen, Thomas H. ; Leiserson, Charles E. ; Riverst, Ronald L. ; Stein, Clifford: *Introduction to Algorithms*. 2nd Edition. Cambridge, Massachusetts; London, England : Massachusetts Institute of Technology (MIT Press), 2001. – 1184 S. – ISBN 978–0262531962 / 0262531968

[CMSS07] Che, Shuai ; Meng, Jiayuan ; Sheaffer, Jeremy W. ; Skadron, Kevin: A Performance Study of General Purpose Applications on Graphics Processors. In: *First Workshop On General Purpose Processing On Graphics Processing Units*, 2007

[Cod86] Codd, E. F. ; Ashenhurst, Robert L. (Hrsg.): *Cellular Automata*. New York and London 1968 : Academic Press, 1986

[CW06] Correia, Luís ; Wehrle, Thomas: Beyond Cellular Automata, Towards More Realistic Traffic Simulators. In: El Yacoubi, Samira (Hrsg.) ; Chopard, Bastien (Hrsg.) ; Bandini, Stefania (Hrsg.): *Cellular Automata* Bd. 4173, Springer Berlin / Heidelberg, 2006, 690-693

[Den97] Dennis, J.: A Parallel Program Execution Model Supporting Modular Software Construction. In: *MPPM '97: Proceedings of the Conference on Massively Parallel Programming Models*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–8186–8427–5, 50-61

[Den09] Denker, Kai: *Entwurf und FPGA-Implementierung von parallelen Architekturen mit Spezialbefehlen zur Unterstützung des GCA-Rechenmodells*. Technische Universität Darmstadt, Oktober 2009. – Diplomarbeit

[Dew84] Dewdney, Alexander K.: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. In: *Scientific American* 251 (1984), Dezember, Nr. 6, 14-22. http://home.cc.gatech.edu/biocs1/uploads/2/wator_dewdney.pdf

[DLR07] D'Souza, R.M. ; Lysenko, M. ; Rahmani, K.: SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates. In: *Proceedings of Agent 2007 Conference*. Chicago, IL, 2007

[DTJ00] Dijkstra, J. ; Timmermans, H. J. P. ; Jessurun, A. J.: A Multi-Agent Cellular Automata System for Visualising Simulated Pedestrian Activity. In: Bandini, S. (Hrsg.) ; Worsch, T. (Hrsg.): *Theoretical and Practical Issues on Cellular Automata - Proceedings on the 4th International Conference on Cellular Automata for research and Industry*. London, UK : Springer Verlag, 2000, 29-36

[Dun90] Duncan, Ralph: A Survey of Parallel Computer Architectures. In: *Computer* Bd. 23, 1990. – ISSN 0018–9162, 5-16

[Dur07] Duran, Ray: *Practical Hardware Debugging: Quick Notes On How to Simulate Altera's Nios II Multiprocessor Systems Using Mentor Graphics' ModelSim*. http://www.altera.com/literature/cp/cp-01031.pdf. Version: March 2007. – Staff Design Specialist FAE, Altera Corporation

[DVT00] Dedu, Eugen ; Vialle, Stéphane ; Timsit, Claude: Comparison of OpenMP and Classical Multi-Threading Parallelization for Regular and Irregular Algorithms. In: Fouchal, Hacène (Hrsg.) ; Lee, Roger Y. (Hrsg.): *Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*. Reims, France : Association for Computer and Information Science, May 2000, 53-60

[EH08] Ediger, Patrick ; Hoffmann, Rolf: Optimizing the creature's rule for all-to-all communication. In: Adamatzky, Andrew (Hrsg.) ; Alonso-Sanz, Ramon (Hrsg.) ; Lawniczak, Anna (Hrsg.) ; Martinez, Genaro J. (Hrsg.) ; Morita, Kenichi (Hrsg.) ; Worsch, Thomas (Hrsg.): *AUTOMATA-2008 Theory and Applications of Cellular Automata*, Luniver Press, 2008. – ISBN 1–905986–16–5, 398-412

[EH09a] Ediger, Patrick ; Hoffmann, Rolf: Alignment of Particles by Agents with Evolved Behaviour. In: *1. Workshop Innovative Rechnertechnologien*, 2009

[EH09b] Ediger, Patrick ; Hoffmann, Rolf: CA Models for Target Searching Agents. In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 252 (2009), October, 41-54. http://dx.doi.org/10.1016/j.entcs.2009.09.013. – DOI 10.1016/j.entcs.2009.09.013

[Elk08] Elkeelany, Omar: *Implementing a Multiprocessor System on an FPGA using Altera Nios II Processors & SOPC*. http://iweb.tntech.edu/oelkeelany/6170/lectures/multi.pdf. Version: 2008. – Tennessee Tech University

[ES97] Esser, Jörg ; Schreckenberg, Michael: Microscopic Simulation Of Urban Traffic Based On Cellular Automata. In: *International Journal of Modern Physics C (IJMPC)* 8 (1997), Nr. 5, 1025-1036. http://dx.doi.org/10.1142/S0129183197000904. – DOI 10.1142/S0129183197000904

[Flo64] Floyd, Robert W.: Algorithm 245. Treesort. In: *Communication of the ACM* 7 (1964), 701. http://dx.doi.org/10.1145/355588.365103. – DOI 10.1145/355588.365103

[Fly72] Flynn, Michael J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* Bd. C-21, 1972. – ISSN 0018–9340, 948-960

[FM04] Fu, Shih C. ; Milne, George: A Flexible Automata Model for Disease Simulation. In: Sloot, Peter M. A. (Hrsg.) ; Chopard, Bastien (Hrsg.) ; Hoekstra, Alfons G. (Hrsg.): *Cellular Automata - 6th International Conference on Cellular Automata for Research and Industry (ACRI)* Bd. 3305, Springer Berlin / Heidelberg, October 2004 (Lecture Notes in Computer Science), 642-649

[Gar70] Gardner, Martin: The fantastic combinations of John Conway's new solitaire game "life". In: *Scientific American* 223 (1970), 120-123. http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

[Gei10] Geib, Florian: *Hardwarearchitekturen zur Simulation von Multiagentensystemen*. Technische Universität Darmstadt, August 2010. – Masterarbeit

[GiD07] GiDEL: *PROCel*. http://www.gidel.com. Version: Juni 2007

[Gip81] Gipps, P. G.: A Behavioural Car-Following Model for Computer Simulation. In: *Transportation Research Part B: Methodological* 15 (1981), Nr. 2, 105-111. http://dx.doi.org/10.1016/0191-2615(81)90037-0. – DOI 10.1016/0191–2615(81)90037–0. – ISSN 0191–2615

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification Third Edition*. Addison-Wesley, 2005 http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf. – ISBN 0–321–24678–0

[GK94] Genesereth, Michael R. ; Ketchpel, Steven P.: Software Agents. In: *Communications of the ACM* 37 (1994), 48-53. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.1311&rep=rep1&type=pdf

[GKSA10] Georgoudas, Ioakeim G. ; Kyriakos, P. ; Sirakoulis, Georgios C. ; Andreadis, Ioannis T.: An FPGA implemented cellular automaton crowd evacuation model inspired by the electrostatic-induced potential fields. In: *Microprocessors and Microsystems* 34 (2010), June, Nr. 7-8, 285-300. http://dx.doi.org/10.1016/j.micpro.2010.06.001. – DOI 10.1016/j.micpro.2010.06.001. – ISSN 0141–9331

[Goo93] Goodwin, Richard: Formalizing Properties of Agents / School of Computer Science - Carnegie Mellon University. Version: May 1993. http://www.agent.ai/download.php?ctag=download&docID=5. Pittsburgh, PA 15213, May 1993. – Forschungsbericht

[Goo98] Goos, Prof. Dr. G.: *Vorlesungen über Informatik - Band 4 Paralleles Rechnen und nicht-analytische Lösungsverfahren*. Springer Berlin / Heidelberg, 1998. – ISBN 3–540–60650–5

[GS10] Gomez, Ernesto ; Schubert, Keith: Algebra of Synchronization with Application to Deadlock and Semaphores. In: Fujita, Satoshi (Hrsg.) ; Ibarra, Oscar (Hrsg.) ; Ichikawa, Shuichi (Hrsg.) ; Nakano, Koji (Hrsg.) ; Zomaya, Albert (Hrsg.): *The First International Conference on Networking and Computing (ICNC)*. Higashi Hiroshima, Japan, November 2010, S. 58

[GSA08] Georgoudas, Ioakeim G. ; Sirakoulis, Georgios C. ; Andreadis, Ioannis T.: Potential Field Approach of a Cellular Automaton Evacuation Model and Its FPGA Implementation. Version: 2008. http://dx.doi.org/10.1007/978-3-540-79992-4_73. In: Umeo, Hiroshi (Hrsg.) ; Morishita, Shin (Hrsg.) ; Nishinari, Katsuhiro (Hrsg.) ; Komatsuzaki, Toshihiko (Hrsg.) ; Bandini, Stefania (Hrsg.): *Cellular Automata* Bd. 5191. Springer Berlin / Heidelberg, 2008. – DOI 10.1007/978–3–540–79992–4_73, 546-549

[GSD+03] Ganguly, Niloy ; Sikdar, Biplab K. ; Deutsch, Andreas ; Canright, Geoffrey ; Chaudhuri, P P.: A Survey on Cellular Automata / Dresden University of Technology, Bengal Engineering College, Telenor Research and Development1331 Fornebu. Version: 2003. http://www.cs.unibo.it/bison/publications/CAsurvey.pdf. 2003. – Forschungsbericht

[GST90] Gerhardt, Martin ; Schuster, Heike ; Tyson, John J.: A Cellular Automaton Model of Excitable Media Including Curvature and Dispersion. In: *Science* 247 (1990), 1563-1566. http://dx.doi.org/10.1126/science.2321017. – DOI 10.1126/science.2321017

[Hai10] Haines, Tyler: *Simulating the Spread of a Virus Spread in a Modern Building Environment*. http://www.tjhsst.edu/~rlatimer/techlab10/Per5/FourthQuarter/HainesPaperQ4-10.pdf. Version: 2009-2010. – TJHSST Senior Research Project

[Hal08] Halbach, Mathias: *Algorithmen und Hardwarearchitekturen zur optimierten Aufzählung von Automaten und deren Einsatz bei der Simulation künstlicher Kreaturen*, Technische Universität Darmstadt, Diss., 2008. http://tuprints.ulb.tu-darmstadt.de/1181/1/essay.pdf

[Hee07] HEENES, Wolfgang: *Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten*, Technische Universität Darmstadt, Diss., 2007. http://www.ra.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_RA/papers/dissertation.pdf

[HH07] HALBACH, Mathias ; HOFFMANN, Rolf: Parallel Hardware Architecture to Simulate Movable Creatures in the CA Model. In: MALYSHKIN, Victor (Hrsg.): *Parallel Computing Technologies* Bd. 4671, Springer Berlin / Heidelberg, 2007 (Lecture Notes in Computer Science), 418-431

[HHJ06] HEENES, Wolfgang ; HOFFMANN, Rolf ; JENDRSCZOK, Johannes: A Multiprocessor Architecture for the Massively Parallel Model GCA. In: *20th IEEE International Parallel and Distributed Processing Symposium*, 2006. – ISBN 1–4244–0054–6, 8

[Hoa62] HOARE, C. A. R.: Quicksort. In: *The Computer Journal* 5 (1962), January, Nr. 1, 10-16. http://dx.doi.org/10.1093/comjnl/5.1.10. – DOI 10.1093/comjnl/5.1.10

[Hoc98] HOCHBERGER, Christian: *CDL - Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen*. Darmstadt, Technische Universität Darmstadt, Diss., Januar 1998

[Hof10a] HOFFMANN, Rolf: GCA-w Algorithms for Traffic Simulation. In: *Summer Solstice 2010, International Conference on Discrete Models of Complex Systems*. Nancy, France, June 2010

[Hof10b] HOFFMANN, Rolf: The Massively Parallel Computing Model GCA. In: *HPPC 2010 - 4th Workshop on Highly Parallel Processing on a Chip*. Ischia - Naples, Italy, August 2010, S. 7. – http://www.par.univie.ac.at/workshop/hppc/HPPC10-Preproc.pdf

[Hof10c] HOFFMANN, Rolf: *Parallele Algorithmen*. http://www.ra.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_RA/ra/03_G_ParalleleAlgorithmen.pdf. Version: 2010

[HVH01] HOFFMANN, Rolf ; VÖLKMANN, Klaus-Peter ; HEENES, Wolfgang: Globaler Zellularautomat (GCA): Ein neues massivparalleles Berechnungsmodell. In: *Mitteilungen - Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen*, 2001. – ISSN 0177–0454

[HVW00] HOFFMANN, Rolf ; VÖLKMANN, Klaus-Peter ; WALDSCHMIDT, Stefan: Global Cellular Automata GCA, An Universal Extension of the CA Model. In: WORSCH, Thomas (Hrsg.): *ACRI*, 2000

[HVWH01] HOFFMANN, Rolf ; VÖLKMANN, Klaus-Peter ; WALDSCHMIDT, Stefan ; HEENES, Wolfgang: GCA: Global Cellular Automata, A Flexible Parallel Model. In: *6th International Conference on Parallel Computing Technologies (PaCT)* Bd. 2127, Springer Verlag, 2001, 66-73

[IEE85] *IEEE Standard for Binary Floating-Point Arithmetic*. http://dx.doi.org/10.1109/IEEESTD.1985.82928. Version: March 1985

[IEE04] IEEE Computer Society: *IEEE Standard Verilog Hardware Description Language*. http://dx.doi.org/10.1109/IEEESTD.2001.93352. Version: January 2004. – IEEE Std 1364-2001

[Ing97] Ingham, James: What is an Agent? / Centre for Software Maintenance University of Durham. Version: 1997. http://www.lsi.upc.es/~bejar/aia/aia-web/ingham99what.pdf. Durham, 1997 (6/99). – Forschungsbericht

[Int07] Intel Corporation: *Quad-Core Intel Xeon Processor 5300 Series Datasheet*, September 2007. http://www.intel.com/Assets/PDF/datasheet/315569.pdf

[JEH08a] Jendrsczok, Johannes ; Ediger, Patrick ; Hoffmann, Rolf: The Global Cellular Automata Experimental Language GCA-L (Version 2) / TU Darmstadt, Rechnerarchitektur. Version: August 2008. http://www.ra.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_RA/papers/JEH08b.pdf. 2008 (RA-1-2007). – Forschungsbericht

[JEH08b] Jendrsczok, Johannes ; Ediger, Patrick ; Hoffmann, Rolf: A Scalable Configurable Architecture for the Massively Parallel GCA Model. In: *22th IEEE International Parallel and Distributed Processing Symposium*, 2008. – ISBN 978–1–4244–1693–6, 1-8

[JHE09] Jendrsczok, Johannes ; Hoffmann, Rolf ; Ediger, Patrick: A Generated Data Parallel GCA Machine for the Jacobi Method. In: *3rd HiPEAC Workshop on Reconfigurable Computing*, 2009, S. 73–82

[JHK07] Jendrsczok, Johannes ; Hoffmann, Rolf ; Keller, Jörg: Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton. In: *9th Workshop on Advances in Parallel and Distributed Computational Models*, 2007. – ISBN 1–4244–0910–1, 1-8

[JHK08] Jendrsczok, Johannes ; Hoffmann, Rolf ; Keller, Jörg: Implementing Hirschberg's PRAM-Algorithm for Connected Components on a Global Cellular Automaton. In: *International Journal of Foundations of Computer Science* 19 (2008), December, Nr. 6, 1299-1316. http://www.worldscinet.com/ijfcs/19/preserved-docs/1906/S0129054108006297.pdf

[JHL09] Jendrsczok, Johannes ; Hoffmann, Rolf ; Lenck, Thomas: Generated Horizontal and Vertical Data Parallel GCA Machines for the N-Body Force Calculation. In: Berekovic, Mladen (Hrsg.) ; Müller-Schloer, Christian (Hrsg.) ; Hochberger, Christian (Hrsg.) ; Wong, Stephan (Hrsg.): *Architecture of Computing Systems – ARCS 2009* Bd. 5455/2009, Springer Berlin / Heidelberg, February 2009. – ISBN 978–3–642–00453–7, 96-107

[JKN+07] John, A. ; Kunwar, A. ; Namazi, A. ; Chowdhury, D. ; Nishinari, K. ; Schadschneider, A.: Traffic on bi-directional ant-trails. Version: 2007. http://dx.doi.org/10.1007/978-3-540-47064-9_44. In: Waldau, Nathalie (Hrsg.) ; Gattermann, Peter (Hrsg.) ; Knoflacher, Hermann (Hrsg.) ; Schreckenberg, Michael (Hrsg.): *Pedestrian and Evacuation Dynamics 2005*. Springer Berlin Heidelberg, 2007. – DOI 10.1007/978–3–540–47064–9_44. – ISBN 978–3–540–47064–9, 465-470

[JSCN04] John, Alexander ; Schadschneider, Andreas ; Chowdhury, Debashish ; Nishinari, Katsuhiro: Collective effects in traffic on bi-directional ant-trails. In: *Journal of Theoretical Biology* 231 (2004), 279. http://xxx.lanl.gov/pdf/cond-mat/0409458v1

[JSCN06] John, Alexander ; Schadschneider, Andreas ; Chowdhury, Debashish ; Nishinari, Katsuhiro: Traffic Patterns and Flow Characteristics in an Ant Trail Model. Version: 2006. http://dx.doi.org/10.1007/11839088_27. In: Dorigo, Marco (Hrsg.) ; Gambardella, Luca (Hrsg.) ; Birattari, Mauro (Hrsg.) ; Martinoli, Alcherio (Hrsg.) ; Poli, Riccardo (Hrsg.) ; Stützle, Thomas (Hrsg.): *Ant Colony Optimization and Swarm Intelligence* Bd. 4150. Springer Berlin / Heidelberg, 2006. – DOI 10.1007/11839088_27, 306-315

[JSCN09] John, Alexander ; Schadschneider, Andreas ; Chowdhury, Debashish ; Nishinari, Katsuhiro: Trafficlike Collective Movement of Ants on Trails: Absence of a Jammed Phase. In: *Physical Review Letters* 102 (2009), March, Nr. 10, 108001. http://dx.doi.org/10.1103/PhysRevLett.102.108001. – DOI 10.1103/PhysRevLett.102.108001

[KAANS10] Khalil, Khaled M. ; Abdel-Aziz, M. ; Nazmy, Taymour T. ; Salem, Abdel-Badeeh M.: An Agent-Based Modeling for Pandemic Influenza in Egypt. In: *7th International Conference on Informatics and Systems (INFOS)*, 2010. – ISBN 978–1–4244–5828–8, 1-7

[Kal09] Kalgin, Konstantin: Implementation of Fine-Grained Algorithms on Graphical Processing Unit. In: Malyshkin, Victor (Hrsg.): *Parallel Computing Technologies* Bd. 5698, Springer Berlin / Heidelberg, 2009, 207-215

[Kar05] Kari, Jarkko: Theory of Cellular Automata: A Survey. In: *Theoretical Computer Science* 334 (2005), April, Nr. 37, 3-33. http://dx.doi.org/10.1016/j.tcs.2004.11.021. – DOI 10.1016/j.tcs.2004.11.021

[KCS+09] Karmani, Rajesh K. ; Chen, Nicholas ; Su, Bor-Yiing ; Shali, Amin ; Johnson, Ralph: Barrier Synchronization Pattern. In: *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009

[Kir02] Kirchner, Ansgar: *Modellierung und statistische Physik biologischer und sozialer Systeme*, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, Diss., 2002. http://www.thp.uni-koeln.de/zitt/phys/data/diss_aki.pdf

[Klü01] Klügl, Franziska: *Multiagentensimulation. Konzepte, Werkzeuge, Anwendung*. Addison-Wesley, 2001. – 234 S. – ISBN 3–8273–1790–8

[Klü03] Klüpfel, Hubert L.: *A Cellular Automaton Model for Crowd Movement and Egress Simulation*. Duisburg, Universität Duisburg-Essen, Diss., Juli 2003. http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-5477/Diskluepfel.pdf

[Klü06] Klügl, Franziska: Multiagentensimulation. In: *Informatik-Spektrum* 29 (2006), December, Nr. 6, 412-415. http://dx.doi.org/10.1007/s00287-006-0115-7. –

DOI 10.1007/s00287–006–0115–7. – ISSN 0170–6012 (Print) 1432–122X (Online)

[Knu98] KNUTH, Donald E.: *The Art of Computer Programming, Volume 3, Sorting and Searching*. Bd. 3. 2nd Edition. Amsterdam : Addison-Wesley Longman, 1998. – 800 S. – ISBN 978–0201896855 / 0201896850

[Kre07] KRETZ, Tobias: *Pedestrian Traffic - Simulation and Experiments*, Universität Duisburg-Essen, Diss., Februar 2007. http://deposit.d-nb.de/cgi-bin/dokserv?idn=983420513&dok_var=d1&dok_ext=pdf&filename=983420513.pdf

[LDS10] LAWNICZAK, A. T. ; DI STEFANO, B. N.: Digital Laboratory of Agent-based Highway Traffic Model. In: *Acta Physica Polonica B Proceedings Supplement* Bd. 3, 2010, 479-453

[LKC+10] LEE, Victor W. ; KIM, Changkyu ; CHHUGANI, Jatin ; DEISHER, Michael ; KIM, Daehyun ; NGUYEN, Anthony D. ; SATISH, Nadathur ; SMELYANSKIY, Mikhail ; CHENNUPATY, Srinivas ; HAMMARLUND, Per ; SINGHAL, Ronak ; DUBEY, Pradeep: Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In: *37th annual internatinal symposium on computer architecture*, 2010. – ISBN 978–1–4503–0053–7, 451-460

[LM01] LI, Xiaodong ; MAGILL, William: Modeling fire spread under environmental influence using a cellular automaton approach. In: *Complexity International* 8 (2001). http://www.complexity.org.au/ci/vol08/li01/li01.pdf

[MB11] MILDE, Benjamin ; BÜSCHER, Niklas: Implementing the Massively Parallel GCA model on CUDA. In: *wird veröffentlicht in: Mitteilungen - Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen*, 2011

[Mea05] MEAD, Pat: Softcore-Prozessoren in Low-Cost-FPGAs. In: *elektronik industrie* 4 (2005), 20-22. http://all-electronics.de/ai/resources/e6dc618b937.pdf

[Mei05] MEINEKE, Frank: *Einführung in die Medizinische Informatik und Bioinformatik - Zellularautomaten*. http://www.imise.uni-leipzig.de/Lehre/Semester/2005/EMIB/Zellularautomaten2.pdf. Version: 2005

[Men] MENTOR GRAPHICS: *ModelSim*. http://model.com/

[Mes] MESSAGE PASSING INTERFACE FORUM: *Message Passing Interface*. http://www.mpi-forum.org/

[Mes09] MESSAGE PASSING INTERFACE FORUM: *MPI: A Message-Passing Interface Standard Version 2.2*, September 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[Mil99] MILOJICIC, Dejan S.: Trend Wars: Mobile agent applications. In: *IEEE Concurrency* 7 (1999), Nr. 3, 80-90. http://dlib.computer.org/pd/books/pd1999/pdf/p3080.pdf

[Mon05] Montenbruck, Oliver: *Grundlagen der Ephemeridenrechnung*. 7. Auflage. Spektrum Akademischer Verlag, 2005. – 173 S. http://www.springerlink.com/content/k00xt1/?p=b15d423912ce4b6791b4c1c1431faeb5&pi=0. – ISBN 978–3–8274–2291–0 (Print) 978–3–8274–2292–7 (Online)

[Mus97] Musser, David R.: Introspective Sorting and Selection Algorithms. In: *Software Practice and Experience* 27 (1997), August, 983-993. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=251DCA7E383E79147EBE1E1DFB9ED87E?doi=10.1.1.14.5196&rep=rep1&type=pdf. – ISSN 0038–0644

[Nag94] Nagel, Kai: *High-speed microsimulations of traffic flow*, Mathematisches Institut, Universität zu Köln, Diss., 1994. http://www.zaik.uni-koeln.de/~paper/unzip.html?file=zpr95-183.ps. – 201 S.

[Nee91] Neelamkavil, Francis: *Computer Simulation and Modelling*. Chichester, New York, Brisbane, Toronto, Singapore : John Wiley & Sons Ltd., 1991. – ISBN 0–471–91129–1

[Neu66] Neumann, John von ; Burks, Arthur W. (Hrsg.): *Theory of Self-Reproducing Automata*. Urbana and London : University of Illinois Press, 1966 http://www.wjzeng.net/Ref/VonNeumann_TheoryOfSelfReproducingAutomata.pdf

[NHP07] Nyland, Lars ; Harris, Mark ; Prins, Jan: Chapter 31, Fast N-Body Simulation with CUDA. Version: 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.156.7082&rep=rep1&type=pdf. In: Nguyen, Hubert (Hrsg.): *GPU Gems 3*. Addison-Wesley, 2007. – ISBN 978–0–321–51526–1, 677-695

[NS92] Nagel, Kai ; Schreckenberg, Michael: A cellular automaton model for freeway traffic. In: *Journal de Physique I* 2 (1992), Nr. 115, 2221-2229. http://www.ptt.uni-duisburg.de/fileadmin/docs/paper/1992/origca.pdf

[NVI09] NVIDIA Corporation: *NVIDIA CUDA Architecture - Introduction & Overview*. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf. Version: April 2009. – Version 1.1

[NVI10a] NVIDIA Corporation: *NVIDIA CUDA C Programming Guide*. Version 3.1, Mai 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[NVI10b] NVIDIA Corporation: *Was ist CUDA?* http://www.nvidia.de/object/what_is_cuda_new_de.html. Version: 2010

[NVI10c] NVIDIA Corporation: *Website*. http://www.nvidia.de. Version: 2010

[OK09] Osterloh, Andre ; Keller, Jörg: Das GCA-Modell im Vergleich zum PRAM-Modell / FernUniversität in Hagen. Version: März 2009. http://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/pv/gca_1_.pdf. 2009 (350). – Forschungsbericht

[Ope97] OpenMP Architecture Review Board: *Open Multi-Processing*. http://www.openmp.org/. Version: 1997

[Ope08] OpenMP Architecture Review Board: *OpenMP Application Program Interface*. Version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf

[PKJFMC09] Poudel, Minesh ; Kaffa-Jackou, Rakia ; França, Felipe M. G. ; Mora-Camino, Félix: Modelling of Aircraft Emergency Evacuation: A Multiagent Approach. In: *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. – ISBN 978–963–9799–45–5, 1-5

[PVR+03] Popov, Konstantin ; Vlassov, Vladimir ; Rafea, Mahmoud ; Holmgren, Fredrik ; Brand, Per ; Haridi, Seif: Parallel Agent-Based Simulation on a Cluster of Workstations. In: Kosch, Harald (Hrsg.) ; Böszörményi, László (Hrsg.) ; Hellwagner, Hermann (Hrsg.): *Euro-Par 2003 Parallel Processing* Bd. 2790, 2003 (Lecture Notes in Computer Science), 470-480

[QMIG10] Quartieri, Joseph ; Mastorakis, Nikos E. ; Iannone, Gerardo ; Guarnaccia, Claudio: A Cellular Automata Model for Fire Spreading Prediction. In: Jha, M. (Hrsg.): *Latest Trends on Urban Planning and Transportation*, 2010. – ISBN 978–960–474–204–2, 173-179. – 3rd WSEAS International Conference on Urban Planning and Transportation (UPT '10)

[Reg87] Regenspurg, Gert: *Hochleistungsrechner - Architekturprinzipien*. McGraw-Hill Book Company GmbH, 1987. – 195 S. – ISBN 3–89028–099–4

[RG85] Rosenschein, Jeffrey S. ; Genesereth, Michael R.: Deals Among Rational Agents. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence* Bd. 1. Los Angeles, California : Morgan Kaufmann Publishers Inc., 1985. – ISBN 0–934613–02–8, 91-99

[RH10] Rossmann, Jürgen ; Hempe, Nico: A Flexible Framework for General, Multi-Agent Based Real-Time Crowd Simulation. In: Baralt, J. (Hrsg.) ; Callaos, N. (Hrsg.) ; Chu, H-W. (Hrsg.) ; Savoie, M. J. (Hrsg.) ; Zinn, C. D. (Hrsg.): *The International Multi-Conference on Complexity, Informatics and Cybernetics (IIIS IMCIC 2010)* Bd. I. Orlando, Florida, April 2010. – ISBN 978–1–934272–87–9, 300-306

[Ric05] Richter, Andreas: *Geschwindigkeitsvorgabe an Lichtsignalanlagen: technische Aspekte und volkswirtschaftlicher Nutzen*. 1. Deutscher Universitäts-Verlag, Wiesbaden, 2005. – 199 S. – ISBN 978–3824408283 / 3–8244–0828–7

[RIP99] Repenning, Alexander ; Ioannidou, Andri ; Phillips, Jonathan: Building a Simulation of the Spread of a Virus - Using the AgentSheets simulation-authoring tool. In: *Learning Technology Review*, 1999, 56-72

[RN04] Russell, Stuart ; Norvig, Peter: *Künstliche Intelligenz*. 2. Auflage. Pearson Education, Inc, publishing as Prentice Hall, 2004. – ISBN 3–8273–7089–2 / 978–3–8273–7089–1. – Authorized translation from the English language edition, entitled ARTIFICIAL INTELLIGENCE: A MODERN APPROACH

[RNSL96] Rickert, Marcus ; Nagel, Kai ; Schreckenberg, Michael ; Latour, Andreas: Two Lane Traffic Simulations using Cellular Automata. In: *Physica A* 231 (1996), 534-550. http://arxiv.org/pdf/cond-mat/9512119v1

[Sch98] Schneider, Ralf: *Forschungsergebnisse zur Informatik*. Bd. 41: *Verfahren zur effizienten Zellularen Verarbeitung*. Dr. Kovač, 1998. – 1–169 S. – ISBN 3–86064–880–2

[Sch99a] Schadschneider, Andreas: *Diskrete stochastische Systeme in niedrigen Dimensionen: Die Physik des Straßenverkehrs*. http://www.thp.uni-koeln.de/~as/Mypage/PSfiles/habil.pdf. Version: März 1999. – Habilitationsschrift

[Sch99b] Schadschneider, Andreas: The Nagel-Schreckenberg model revisited. In: *The European Physical Journal B - Condensed Matter and Complex Systems* 10 (1999), August, Nr. 3, 573-582. http://dx.doi.org/10.1007/s100510050888. – DOI 10.1007/s100510050888. – ISSN 1434–6028 (Print) 1434–6036 (Online)

[SH09] Selvapeter, P. J. ; Hordijk, Wim: Cellular Automata for Image Noise Filtering. In: *Nature Biologically Inspired Computing*, 2009. – ISBN 978–1–4244–5053–4, 193-197

[SHH09a] Schäck, Christian ; Heenes, Wolfgang ; Hoffmann, Rolf: GCA Multi-Softcore Architecture for Agent Systems Simulation. In: Fischer, Stefan (Hrsg.) ; Maehle, Erik (Hrsg.) ; Reischuk, Rüdiger (Hrsg.): *Informatik 2009 Im Focus das Leben* Bd. P-154, 2009 (Lecture Notes in Informatics). – ISSN 1617–5468, 278; 2268-82

[SHH09b] Schäck, Christian ; Heenes, Wolfgang ; Hoffmann, Rolf: A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA. In: Bertels, Koen (Hrsg.) ; Dimopoulos, Nikitas (Hrsg.) ; Silvano, Cristina (Hrsg.) ; Wong, Stephan (Hrsg.): *Embedded Computer Systems: Architectures, Modeling, and Simulation* Bd. 5657, Springer Berlin / Heidelberg, July 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–03137–3, 98-107

[SHH09c] Schäck, Christian ; Heenes, Wolfgang ; Hoffmann, Rolf: Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA. In: *Mitteilungen - Gesellschaft für Informatik e. V., Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware* Bd. 26 Wolfgang Karl and Rolf Hoffmann and Wolfgang Heenes, 2009. – ISSN 0177–0454, S. 48–57

[SHH10a] Schäck, Christian ; Heenes, Wolfgang ; Hoffmann, Rolf: Multiprocessor Architectures Specialized for Multi-Agent Simulation. In: Fujita, Satoshi (Hrsg.) ; Ibarra, Oscar (Hrsg.) ; Ichikawa, Shuichi (Hrsg.) ; Nakano, Koji (Hrsg.) ; Zomaya, Albert (Hrsg.): *The First International Conference on Networking and Computing (ICNC)*. Higashi Hiroshima, Japan, November 2010. – ISBN 978–1–4244–8918–3, 232-236

[SHH10b] Schäck, Christian ; Hoffmann, Rolf ; Heenes, Wolfgang: Efficient Traffic Simulation using the GCA Model. In: *24th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum IPDPSW*, 2010. – ISBN 978–1–4244–6532–3. – 12th Workshop on Advances in Parallel and Distributed Computational Models (APDCM), IEEE Catalog Number: CFP1051J-CDR

[SHH11a] Schäck, Christian ; Hoffmann, Rolf ; Heenes, Wolfgang: Efficient Traffic Simulation Using Agents within the Global Cellular Automata Model. In: *International Journal of Networking and Computing* 1 (2011), January, Nr. 1, 2-20. http://ijnc.org/lib/exe/fetch.php/v01n01p01.pdf

[SHH11b] Schäck, Christian ; Hoffmann, Rolf ; Heenes, Wolfgang: Specialized Multicore Architectures Supporting Efficient Multi-Agent Simulations. In: *International Journal of Networking and Computing* (2011)

[Sip97] Sipper, Moshe ; Goos, Gerhard (Hrsg.) ; Hartmanis, Juris (Hrsg.) ; Leeuwen, Jan van (Hrsg.): *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Bd. 1194. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1997. http://dx.doi.org/10.1007/3-540-62613-1. http://dx.doi.org/10.1007/3-540-62613-1. – ISBN 3540626131

[SK10] Sartori, John ; Kumar, Rakesh: Low-Overhead, High-Speed Multi-core Barrier Synchronization. In: Patt, Yale (Hrsg.) ; Foglia, Pierfrancesco (Hrsg.) ; Duesterwald, Evelyn (Hrsg.) ; Faraboschi, Paolo (Hrsg.) ; Martorell, Xavier (Hrsg.): *High Performance Embedded Architectures and Compilers* Bd. 5952, Springer Berlin / Heidelberg, 2010 (Lecture Notes in Computer Science), 18-34

[Smi76] Smith, Alvy R.: Introduction to and Survey of Cellular Automata or Polyautomata Theory. In: Lindenmayer, A. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Automata, Languages, Development*. New York : North-Holland Publishing Co., 1976, 405-422

[SN09a] Strippgen, David ; Nagel, Kai: Multi-Agent traffic simulation with CUDA. In: *High Performance Computing & Simulation (HPCS)*. Leipzig, Germany, 2009, 106-114

[SN09b] Strippgen, David ; Nagel, Kai: Using common graphics hardware for multi-agent traffic simulation with CUDA. In: Dalle, Olivier (Hrsg.) ; Wainer, Gabriel A. (Hrsg.) ; Perrone, L. F. (Hrsg.) ; Stea, Giovanni (Hrsg.): *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST, Brussels, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 1-8

[SSH08] Stratton, John A. ; Stone, Sam S. ; Hwu, Wen-mei W.: MCDUA: An Efficient Implementation of CUDA Kernels on Multi-cores / University of Illinois at Urbana-Champaign Center for Reliable and High-Performance Computing. Version: April 2008. http://www.crhc.uiuc.edu/IMPACT/ftp/report/impact-08-01-mcuda.pdf. 2008. – IMPACT Technical Report - IMPACT-08-01

[SSN96] Schreckenberg, Michael ; Schadschneider, Andreas ; Nagel, Kai: Zellularautomaten simulieren Straßenverkehr. In: *Physikalische Blätter* 52 (1996), Nr. 5, 460-462. http://www.ptt.uni-duisburg.de/fileadmin/docs/paper/1996/physbl.pdf

[Sta73] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Wien, New York : Springer Verlag, 1973. – ISBN 3–211–81106–0 / 0–387–81106–0

[STC06] Schmickl, Thomas ; Thenius, Ronald ; Crailsheim, Karl: Kollektive Sammel-Entscheidungen: Eine Multi-Agenten-Simulation einer Honigbienenkolonie. In:

Entomologica Austriaca 13 (2006), März, 15-24. http://www.biologiezentrum.at/pdf_frei_remote/ENTAU_0013_0015-0024.pdf. – ISSN 1681–0406

[Ste01] Stebens, André: *Simulation von Eisenbahnverkehr auf der Basis von Zellularautomaten*, Fachbereich Physik, Universität Duisburg, Diss., 2001. http://www.ub.uni-duisburg.de/ETD-db/theses/available/duett-10142001-214332/unrestricted/Stebens_Dissertation.pdf

[TCKT00] Takizawa, Atsushi ; Camada, Atsushi ; Kawamura, Hiroshi ; Tani, Akinori: Simulation of spreads of fire on city site by stochastic cellular automata. In: *12th World Conference on Earthquake Engineering*, 2000, 1-8. – Ref. No. 2334

[Tex96] Texas Instruments: *What's an LFSR*. http://focus.ti.com/lit/an/scta036a/scta036a.pdf. Version: December 1996. – SCTA036A

[Tho00] Thomas, Christopher D.: *Evolution of Cellular Automata for Image Processing*. http://coldfusion2.net/_doc/research.pdf. Version: April 2000. – University of Birmingham

[TJL04] Tran, John ; Jordan, Don ; Luebke, David: *New Challenges for Cellular Automata Simulation on the GPU*. http://www.cs.virginia.edu/johntran/cagpu/. Version: August 2004. – ACM Workshop on General Purpose Computing on Graphics Processors

[TM87] Toffoli, Tommaso ; Margolus, Norman ; Gannon, Dennis (Hrsg.): *Cellular Automata Machines - A new environment for modeling*. MIT Press Series in Scientific Computation, 1987

[TSC06] Thenius, Ronal ; Schmickl, Thomas ; Crailsheim, Karl: Einfluß der Individualität bei Sammelbienen (Apis mellifera L) auf den Sammelerfolg. In: *Entomologica Austriaca* 13 (2006), März, 25-29. http://www.biologiezentrum.at/pdf_frei_remote/ENTAU_0013_0025-0029.pdf. – ISSN 1681–0406

[TSn02] Tabares, Winfer C. ; Saldaña, Rafael P.: A Cellular Automata-Based Study of Vehicular Traffic Dynamics. In: *2nd Philippine Computing Science Congress*, 2002

[Vet06] Vetter, Dr.-Ing. M.: *Ein Multiagentensystem zur Verhandlungsautomatisierung in elektronischen Märkten*, Universität Stuttgart, Diss., 2006

[vSv10] Žaloudek, Luděk ; Sekanina, Lukáš ; Šimek, Václav: Accelerating Cellular Automata Evolution on Graphics Processing Units. In: *International Journal on Advances in Software* 3 (2010), Nr. 1, 294–303. http://www.fit.vutbr.cz/research/view_pub.php?id=9315. – ISSN 1942–2628

[Wei99] Weiss, Gerhard (Hrsg.): *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Cambridge, Massachusetts : Massachusetts Institute of Technology (MIT Press), 1999. – ISBN 978–0262731317 / 0–2627–3131–2

[Wei02] Weiss, Alan R.: *Dhrystone Benchmark - History, Analysis, "Scores" and Recommendations*. http://www.johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf. Version: November 2002

[Wil64] Williams, J. W. J.: Algorithm 232: Heapsort. In: *Communication of the ACM* Bd. 7, 1964, S. 347–348

[WJ95a] Wooldridge, Michael ; Jennings, Nicholas R.: Agent Theories, Architectures, and Languages: A Survey. In: Carbonell, J. G. (Hrsg.) ; Siekmann, J. (Hrsg.): *Intelligent Agents* Bd. 890, Springer Berlin / Heidelberg, January 1995. – ISBN 978–3–540–58855–9, 1-39

[WJ95b] Wooldridge, Michael ; Jennings, Nicholas R.: Intelligent Agents: Theory and Practice. In: *The Knowledge Engineering Review* 10 (1995), Nr. 2, 115-152. http://www.csc.liv.ac.uk/~mjw/pubs/ker95.pdf

[Wol84] Wolfram, Stephen: Universality and Complexity in Cellular Automata. In: *Physica D: Nonlinear Phenomena* 10 (1984), January, 1-35. http://dx.doi.org/10.1016/0167-2789(84)90245-8. – DOI 10.1016/0167–2789(84)90245–8

[Wol86] Wolfram, Stephen: *Theory and Applications of Cellular Automata*. World Scientific, 1986. – ISBN 9971–50–123–6 / 9971–50–124–4

[Woo02] Woolridge, Michael: *Introduction to MultiAgent Systems*. John Wiley & Sons, Ltd, 2002. – 348 S. – ISBN 0–471–49691–x

[Xila] Xilinx Corporation: *MicroBlaze Processor Reference Guide Embedded Development Kit EDK 10.1i*. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[Xilb] Xilinx Corporation: *PicoBlaze 8-bit Embedded Microcontroller User Guide for Spartan-3, Spartan-6, Virtex-5, and Virtex-6 FPGAs*. http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf

[Xil10] Xilinx Corporation: *Website*. http://www.xilinx.com. Version: 2010

[You84] Young, David A.: A Local Activator-Inhibitor Model of Vertebrate Skin Patterns. In: *Mathematical Bioscience* 72 (1984), 51-58. http://dx.doi.org/10.1016/0025-5564(84)90060-9. – DOI 10.1016/0025–5564(84)90060–9

[Zad65] Zadeh, Lotfali A.: Fuzzy Sets. In: *Information and Control* 8 (1965), Nr. 3, 338-353. http://dx.doi.org/10.1016/S0019-9958(65)90241-X. – DOI 10.1016/S0019–9958(65)90241–X. – ISSN 0019–9958

[ZL05] Zhang, Shiwu ; Liu, Jiming: A Massively Multi-agent System for Discovering HIV-Immune Interaction Dynamics. In: *Massively Multi-Agent Systems I* Bd. 3446, Springer Verlag, 2005. – ISSN 0302–9743 (Print) 1611–3349 (Online), 161-173

[Zus70] Zuse, Konrad: Calculating Space / Massachusetts Institute of Technology. Cambridge, Mass. 02139, February 1970. – Forschungsbericht

Glossar

Agent Bewegliche Einheit im Multi-Agenten-System. 30

Agentenumwelt Umgebung, in der ein Agent agiert. 29

Generation Momentaufnahme aller Zellzustände zu einem Zeitpunkt. 12

Modell Ein Modell ist eine pragmatische, verkürzte Abbildung eines Systems. 9

Multi-Agenten-Anwendung Verhalten aller Agenten als Gesamtsystem. 30

Multi-Agenten-Simulation Durchführung der Agentensimulation. 30

Multi-Agenten-Simulations-System Durchführung der Agentensimulation. 31

Multi-Agenten-System Gesamtheit eines zu simulierenden Systems. 29

Nachbarschaft Zellen, die eine andere Zelle umgeben. 11

non-uniform CA Zellularer Automat mit unterschiedlichen Zellregeln für die Zellen. 12

uniform CA Zellularer Automat mit der gleichen Zellregel für alle Zellen. 12

Zelle kleinstes Element eines Zellularen Automaten. 11

Zellregel Berechnungsvorschrift. 11

Zellulares Feld Unendlich großes regelmäßig angeondetes Gitter aus Zellen. 11

Zugriffsmuster Gesamtheit aller Zugriffe einer Generation. 16

Zugriffsstruktur Gesamtheit der Zugriffsmuster einer Anwendung. 16

Wissenschaftlicher Werdegang

Personalien:	Christian Alexander Schäck geboren am 21. Juli 1982 in Frankfurt am Main
Schulbildung:	1989-1993 Grundschule 1993-2002 Gymnasium
Schulabschluss:	Abitur (Allgemeine Hochschulreife) Leistungskurse: Mathematik, Informatik
Betriebspraktika:	1997 Fraport AG, Frankfurt 2002 Pan Dacom Networking AG, Dreieich
Studium:	2002-2008 Diplomstudiengang der Informatik an der Technischen Universität Darmstadt, Abschluss Diplom-Informatiker 2008-2011 Promotionsstudium an der Technischen Universität Darmstadt
Auszeichnungen:	2004 Programmierwettbewerb, Technische Universität Darmstadt, 3. Platz 2009 Nachwuchspreis der Fachgruppe Parallel -Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS), eine gemeinsame Fachgruppe der Gesellschaft für Informatik e.V. (GI) und der Informationstechnischen Gesellschaft (ITG) für die Arbeit „Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA“
Berufstätigkeit:	2008-2010 Studentische Hilfskraft, Fachgebiet Rechnerarchitektur, Technische Universität Darmstadt Wissenschaftliche Hilfskraft mit Abschluss, Fachgebiet Rechnerarchitektur, Technische Universität Darmstadt Wissenschaftlicher Mitarbeiter, Fachgebiet Rechnerarchitektur, Technische Universität Darmstadt Betreuung von Übungen zur Einführung in Computer Microsystems Betreuung des Bachelorpraktikums Betreuung von Master- und Diplomarbeiten