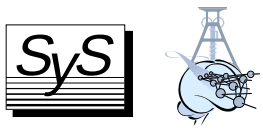
Diplomarbeit

Evolvable Hardware

Eine Analyse intrinsischer Hardware-Evolution digitaler Schaltkreise

Axel W. Dietrich



Lehrstuhl (XI) für Systemanalyse
Fachbereich Informatik
Universität Dortmund

Lehrstuhl für theoretische Biologie
Institut für Neuroinformatik
Ruhr-Universität Bochum

11. August 1999

Betreuer

Prof. Dr. Wolfgang Banzhaf
Prof. Dr. Werner von Seelen

meinen Eltern

Mein Dank gilt Herrn Professor Banzhaf und Herrn Professor von Seelen für die Übernahme des Gutachtens dieser Arbeit. Peter Dittrich, Martin Kreutz, Bernhard Sendhoff sowie Christian Igel möchte ich für ihre Bemühungen danken, mich trotz vereinzelter Gegenwehr auf dem rechten Weg zu halten.

Zusammenfassung—Diese Arbeit beschäftigt sich mit Evolvable Hardware (EHW), einem Teilgebiet der Evolutionary Electronics. Evolvable Hardware stellt dabei eine Methodik dar, die sich das Ziel gesetzt hat durch die Zusammenführung Evolutionärer Algorithmen und programmierbarer Mikro-Chips, neue Wege bei dem Entwurf elektronischer Schaltkreise zu beschreiten. In dieser Arbeit wurde ein Evolvable Hardware System, bestehend aus einem XC6216 FPGA der Firma XILINX und Software in Form Genetischer Algorithmen, aufgebaut, das den evolutionären Entwurf boolescher Schaltkreise ermöglicht. Mit Hilfe dieses Systems erfolgten Experimente zur Evolvierung digitaler Schaltkreise, die in der Lage sind bestimmte boolesche Funktionen wie bspw. n-Parity, $n:m$-Multiplexer, etc. zu berechnen. Im Hinblick auf die Kodierung der Genotyp/Phänotyp Abbildung des Genetischen Algorithmus werden schrittweise Verbesserungen dieser Kodierung vorgestellt und die Ergebnisse im Hinblick auf Nutzen und Anwendbarkeit von Mutations- und Rekombinations-Operatoren untersucht. Die Analysen zeigen, daß dem Rekombinations-Operator eine vernachlässigbare Bedeutung im Rahmen des evolutionären Schaltkreisentwurfes zukommt; die alleinige Verwendung eines Mutations-Operators führt zu einer deutlich schnelleren Konvergenz des Genetischen Algorithmus. Bei der Untersuchung verschiedener Selektionsmechanismen stellt sich heraus, daß mit der q-Tournament Selektion die besten Resultate erzielt werden können. Neben der Tatsache, daß sich die Komplexität der betrachteten Probleme durch eine Verfeinerung der Kodierung steigern läßt, bleibt festzustellen, daß der evolutionäre Entwurf digitaler Schaltkreise außerordentlich schwierig ist.

Stichwörter—Evolvable Hardware, EHW, Hardware Evolution, Genetische Algorithmen, Field Programmable Gate Array, FPGA, Reconfigurable Programming Unit, RPU, XILINX XC6216.

Inhaltsverzeichnis

1 Zielsetzung

Welchen Leser ich wünsche? Den unbefangensten, der mich, sich und die Welt vergißt, und in dem Buche nur lebt.

– Johann Wolfgang von Goethe

Evolvable Hardware (EHW) hat seit Anfang der 90er Jahre zunehmendes Interesse durch die Verfügbarkeit einfach zu rekonfigurierender Hardware, bspw. in Form von Field Programmable Gate Arrays (FPGA), erfahren. Dabei verspricht die EHW einerseits einen neuartigen und unkonventionellen Ansatz für den Entwurf komplexer Schaltkreise darzustellen. Andererseits mag sie aber auch die Grundlage neuer, sich selbst an eine veränderliche Umgebung anpassender, Hardware sein. Derzeit verwenden die meisten Evolvable Hardware Systeme biologisch motivierte Berechnungsmodelle, die sog. Evolutionären Algorithmen (EA), als haupsächlichen Adaptationsmechanismus. Generell wird dabei zwischen *intrinsischer* und *extrinsischer* Hardware-Evolution unterschieden. Während sich die letztgenannte auf Softwaresimulationen beschränkt, bei der das Verhalten der Hardware modelliert und nur die endgültige Lösung in eine FPGA o. ä. geladen wird, nutzt die intrinsische Evolution einen *realen* Chip.

Ein solches intrinsisches Evolvable Hardware System wird normalerweise unter Zuhilfenahme (re)konfigurierbarer Chips, den sog. Programmable Logic Devices (PLD), implementiert; häufig kommen dabei FPGAs zum Einsatz. Ein solcher FPGA kann durch die Angabe einer Menge von Konfigurationsbits beliebig oft programmiert werden, so daß er einen bestimmten Schaltkreis darstellt. In Abhängigkeit des Verwendungszwecks kann ein EHW-System damit als die Anwendung evolutionärer Techniken zum Zwecke der Schaltkreissynthese verstanden werden: Die simulierte Evolution wird zur Bestimmung einer Menge von Konfigurationsbits verwendet, so daß der durch die Bits beschriebene Schaltkreis des FPGA vorgegebenen Kriterien entspricht. Diese Kriterien können u. a. die Funktion eines Schaltkreises oder, für den Fall digitaler Bausteine, die Anzahl der verwendeten Logik-Zellen auf dem FPGA sein.

In dieser Arbeit wurde zunächst ein Evolvable Hardware System aufgebaut und implementiert, das unter Verwendung eines XILINX XC6216 FPGA sowie Evolutionärer Algorithmen die Synthese von digitalen Schaltkreisen erlaubt. Ziel war eine adäquate Übertragung der Konfigurationsbits eines FPGA auf Bitstrings zu finden, so daß diese als Individuen einer Population für einen Genetischen Algorithmus (GA) verwendet werden können. Während der komplette GA auf einem Host-Rechner (PC) abläuft, wird der XC6216 zur Instantiierung der Bitstrings verwendet. Darüberhinaus wurde die gewählte Kodierung zusammen mit den verwendeten genetischen Operatoren dahingehend analysiert, ob eine schnelle Konvergenz des GA im Hinblick auf die Problemstellung gewährleistet ist.

Durch eine schrittweise Verbesserung ebenjener Kodierung, bei Evolutionären Algorithmen wird von der Genotyp/Phänotyp Abbildung gesprochen, konnten zunehmend komplexere Probleme gelöst werden: Während zu Beginn der Versuche lediglich Schaltkreise für das 3-Parity Problem gefunden wurden, war mit der besten Kodierung bereits der evolutionäre Entwurf von 8-Parity Schaltkreisen möglich. Anhand der Evolvierung eines Halbaddierer- und Multiplixer-Schaltkreises werden die Grenzen der Kodierung festgestellt.

In Kapitel 2 erfolgt die thematische Einführung in die Arbeit. Neben der Vorstellung grundlegender Prinzipien der Evolutionären Algorithmen sowie Programmable Logic Devices wird ausführlich erläutert wie die Symbiose dieser beiden Disziplinen zur Evolvable Hardware führt. Den Abschluß dieses Kapitels bildet ein Überblick aktueller Forschungsarbeiten auf dem Gebiet der EHW.

Ein wichtiges Standbein dieser Arbeit stellt der XC6216 Field Programmable Gate Array von XILINX dar, dessen Programmierung ausführlich in Kapitel 3 erläutert wird. Zusammen mit den im Anhang vorgestellten Besonderheiten des XC6216 sowie den dort ebenfalls befindlichen Beispielen sollte dieses Kapitel ein tieferes Verständnis der Programmierung des XC6216 vermitteln.

Kapitel 4 setzt sich mit dem Versuchsaufbau des in dieser Arbeit implementierten EHW Systems auseinander. Es wird die verwendete Hard- und Software vorgestellt sowie eine detaillierte Übersicht des implementierten Systems gegeben. Weiterhin wird der Begriff der Kodierung im Hinblick auf das biologische Vorbild, einer adäquaten Repräsentation in Genetischen Algorithmen und auch bzgl. der Kausalität von Genotyp/Phänotyp Abbildungen diskutiert.

Die durchgeführten Experimente zur Variation der Repräsentation von Genotypen wird in Kapitel 5 vorgestellt. Dabei werden die Verfahren zur Analyse erläutert, entsprechende Ergebnisse dokumentiert und abschließend diskutiert.

Im letzten Kapitel 6 werden die Ergebnisse zusammengefaßt und es erfolgt ein Ausblick auf der weiterführende Ideen und Analysen.

2 Thematische Einführung

Ich war sehr befriedigt, daß ich prompt antworten konnte. Ich sagte, ich wüßte es nicht.
– Mark Twain

Im Forschungsgebiet der **Evolutionary Electronics** finden sich verschiedene Disziplinen zusammen, die alle unter Zuhilfenahme Evolutionärer Algorithmen die Optimierung bestimmter Eigenschaften elektronischer Schaltkreise verfolgen. Zu diesen Eigenschaften können u. a. das Layout (Routing, Struktur) eines Schaltkreises oder auch die Parameter eines einzelnen Transistors innerhalb einer Schaltung gehören; nach Yao und Higuchi (1999) wird dieses Vorgehen jedoch treffender als kombinatorische Optimierung mit Evolutionären Algorithmen bezeichnet: „They [. . .] optimize certain aspects of electronic circuit boards, e. g., cell placement, and compaction of symbolic layout. In essence, such work is better described as combinatorial optimization by EA's" (Yao und Higuchi 1999, Seite 88).

Bei der in dieser Arbeit betrachteten **Evolvable Hardware** (EHW) wird als Eigenschaft die **Funktion** eines Schaltkreises optimiert. EHW gibt damit dem Schaltkreisdesigner ein Verfahren an die Hand, das für ein gegebenes Problem, bspw. der Berechnung der n-Parity Funktion oder eines $n:m$-Multiplexers, automatisch einen passenden Schaltkreis als Lösung findet.

Zum Verständnis der Funktionsweise von Evolvable Hardware werden in den nächsten Abschnitten einige grundlegende Begriffe geklärt. Der folgende Abschnitt befaßt sich mit dem Gebiet der Evolutionären Algorithmen unter besonderer Berücksichtigung der in dieser Arbeit verwendeten Genetischen Algorithmen. Abschnitt 2.2 gewährt einen kurzen Überblick über aktuelle Technologien für programmierbare Mikro-Chips, wobei hier das Hauptaugenmerk auf den Field Programmable Gate Arrays (FPGA) liegen wird, da diese in der vorliegenden Arbeit genutzt werden. Abschnitt 2.3 wird die Idee der EHW anhand der Fusion von Evolutionären Algorithmen und FPGAs näher beleuchten und in diesem Zusammenhang auch auf die Unterschiede zwischen der EHW und dem herkömmlichen Schaltkreisentwurf eingehen.

2.1 Evolutionäre Algorithmen

„Darwinism [..] in the scientific sense, accepts with Darwin that the key mechanism of evolutionary change is natural selection or (as it was also called) the survival of the fittest—a mechanism that depends crucially on the struggle between organisms for food and space“ (Keller et al. 1992, Seite 78).

Evolutionäre Algorithmen (EA) sind eine Klasse direkter, probabilistischer Such- und Optimierungsalgorithmen, deren Funktionsweise auf dem erstmals von Charles R. Darwin formulierten Modell der natürlichen biologischen Evolution basiert. Die Hauptvertreter Evolutionärer Algorithmen sind die **Genetischen Algorithmen** (Holland 1975, Goldberg 1989), die **Evolutionsstrategien** (Rechenberg 1994, Schwefel 1995), sowie das **Evolutionäre Programmieren** (Fogel 1994), die zwar alle unabhängig voneinander entwickelt wurden jedoch mit verwandten Prinzipien arbeiten. Später kam als weiterer Vertreter noch das **Genetische Programmieren** hinzu (Koza 1994, Banzhaf et al. 1998). Eine empfehlenswerte Betrachtung der drei erstgenannten Disziplinen ist in Bäck (1996) zu finden.

Der folgende Abschnitt soll den biologischen Hintergrund Evolutionärer Algorithmen kurz beleuchten während im Anschluß Abschnitt 2.1.2 das Grundprinzip aller EA vorstellt.

2.1.1 Biologischer Hintergrund

Genotyp: die Gene [..] durch deren gemeinsames Wirken [..] der Phänotyp geprägt wird. Phänotyp: die genetisch kontrollierte Eigenschaft oder das gesamte Erscheinungsbild eines Indviduums zu einem best. Zeitpunkt seiner Entwicklung als Ergebnis der kombin. Wirkung von Genotypen u. Umwelt. Aus Urban und Schwarzberg (1995).

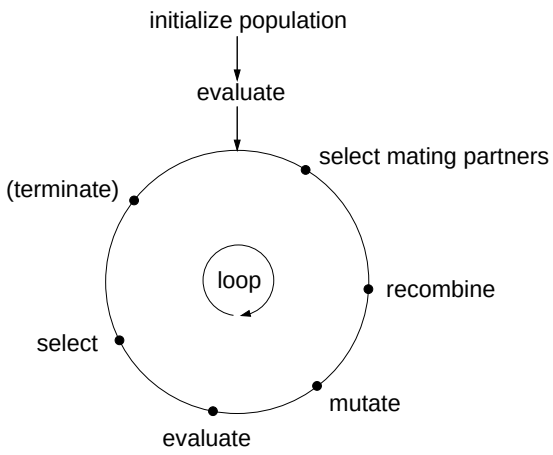
Gemeinsam ist allen Evolutionären Algorithmen die Anlehnung an das biologische Modell natürlicher Evolution. Dieses erklärt gemäß Darwins Evolutionstheorie die adaptive Änderung einer Art durch die Prinzipien der natürlichen **Selektion**, bei der genau jene Spezies überleben und sich weiterentwickeln wird, die bestmöglich an ihren Lebensraum angepaßt ist. Neben der Selektion ist ein weiterer wichtiger Faktor für Evolution das Auftreten von kleinen, anscheinend zufälligen und nicht-gerichteten Variationen zwischen Phänotypen. Solche sog. **Mutationen** bleiben durch die Selektion erhalten, wenn sie sich als nützlich im Hinblick auf den aktuellen Lebensraum erwiesen haben, andernfalls verschwinden sie. Die (nicht aktive, s. u.) treibende Kraft der Selektion stellt die Zeugung von **Nachkommen** dar. Diese Darstellung makroskopischer Evolution wurde durch neue Erkenntnisse der Biochemie und Genetik im Bereich der Vererbungslehre zur synthetischen Theorie der Evolution (Neo-Darwinismus) erweitert. Diese Theorie basiert auf **Genen** als Träger der Erbinformationen sowie den **Chromosomen**, die aus den Genen aufgebaut sind. Gene können Mutationen unterliegen. Selektion erfolgt unter den **Individuen**, den Trägern der Gene, einer **Population**. Im Rahmen natürlicher Evolution kann die **Fitneß** eines Individuums nur indirekt gemessen werden, bspw. über seine Reproduktionsrate im Vergleich zu anderen Individuen. Die natürliche Selektion ist keine aktive treibende Kraft an sich, sondern sie ist das Resultat von Überleben und Reproduktion eines Individuums innerhalb einer Population. Somit ist Fitneß als Bestandteil des Terminus „survival of the fittest“, der erstmals in Spencer (1864) als Synonym für Darwins „natürliche Selektion“ auftauchte, schwierig

zu definieren und bedarf im Rahmen natürlicher Evolution weiterer Klärung: „The precise meaning of 'fitness' has yet to be settled [. . .] After all, evolutionary theory itself is still in flux." (Keller und Lloyd 1992, Seite 115).

Weitergehende Informationen zu den biologischen Grundlagen natürlicher Evolution finden sich u. a. in Hirsch-Kaufmann und Schweiger (1992) oder Keller und Lloyd (1992).

2.1.2 Das Grundprinzip der Evolutionären Algorithmen

Das Grundprinzip aller Evolutionären Algorithmen beruht auf der Verwendung einer Population von Individuen, die Positionen oder Punkte in einem Suchraum repräsentieren. Neue Individuen werden dabei unter Zuhilfenahme molekularbiologischer Prinzipien erzeugt. EAs arbeiten mit sog. **genetischen Operatoren**, vornehmlich solchen zur Mutation und Rekombination von Individuen. Als Rekombinationsoperator kommt hierbei häufig das **Crossover** zum Einsatz, bei dem Teile von zwei oder mehr Individuen mit dem Ziel der Informationsübertragung ausgetauscht werden. Im Laufe einer **Generation** werden die Individuen einer Population einer Fitneßbewertung unterzogen.




```
t:=0
initialize(P(t))
evaluate(P(t))
while not terminate(P(t)) do
   recombine(P(t))
   mutate(P(t))
   evaluate(P(t))
   P(t+1):=select(P(t))
   t:=t+1
od
```

Abbildung 1 Links: Evolutionsschleife die den meisten Evolutionären Algorithmen zugrunde liegt, aus Schwefel und Kursawe (1998). Rechts: Algorithmische Beschreibung eines EA in sehr vereinfachter Form nach Bäck (1996), Seite 66. P(t) bezeichnet die Population zum Zeitpunkt t.

Die Fitneß eines Individuums nimmt dabei Einfluß auf die Wahrscheinlichkeit, mit der ein Individuum an der Reproduktion für die nächste Generation teilnimmt. Dieser Vorgang wird, in Analogie zur natürlichen Evolution, Selektion genannt. Abbildung 1 (links) verdeutlicht diesen Vorgang anhand einer „evolutionären Schleife", die den grund-

legenden Ablauf der meisten Evolutionären Algorithmen darstellt. Zu einem Zeitpunkt $t = 0$ wird eine Population, meist zufällig, initialisiert und im Anschluß evaluiert, d. h. es wird die Fitneß eines jeden Individuums dieser Population bestimmt. Im Anschluß daran wird die evolutionäre Schleife betreten, in der die Schritte Selektion, Rekombination, Mutation und wiederum Fitneßbewertung ausgeführt werden, bis eine bestimmte Abbruchbedingung, bspw. ein hinreichend guter Fitneßwert eines Individuums der Population, erfüllt ist.

2.1.3 Genetische Algorithmen

In diesem Abschnitt sollen die in dieser Arbeit verwendeten Genetischen Algorithmen vorgestellt werden, die in ihrer klassischen Form in den 60er Jahren von John Holland entwickelt wurden, um einige Eigenschaften der natürlichen, biologischen Evolution zu imitieren (Holland 1975). Beim Einsatz Genetischer Algorithmen werden analog zur Natur **Chromosomen** betrachtet. Jedes Chromosom stellt eine mögliche (kodierte) Lösung eines gegebenen Problems dar. Chromosomen sind Symbolstrings mit fester oder variabler Länge; zumeist handelt es sich um Bitstrings aus $\{0,1\}^l$ mit fester Länge l. Verglichen mit herkömmlichen Suchmethoden arbeiten Genetische Algorithmen mit einer Menge von Lösungskandidaten gleichzeitig oder, um die aus der Biologie entlehnten Begriffe zu verwenden, mit einer Population von Individuen, so daß das Prinzip der **multidirektionalen Suche** unterstützt wird. Die auf der rechten Seite von Abbildung 1 gezeigte algorithmische Beschreibung eines allgemeinen EA ist im Falle der Genetischen Algorithmen direkt übertragbar.

Bei der **Initialisierung** der Startpopulation $P(0)$, vgl. Abbildung 1 (rechts), werden alle Individuen mit zufälligen Werten belegt. Auch die Vorbelegung der Individuen mit auf das Problem angepaßten „sinnvollen“ Werten ist möglich, wobei es sich in diesem Fall jedoch bereits um eine Daten-Vorverarbeitung aufgrund von Wissen über das zu lösende Problem handelt.

Innerhalb der evolutionären Schleife (Abbildung 1, links) werden Individuen einer Generation durch **Selektion** für die Folgegeneration ausgewählt. Dabei erfolgt die Selektion zu einem gewissen Teil fitneßabhängig, d. h. bessere Individuen werden bevorzugt in die Nachfolgepopulation übernommen. Weiterhin erfolgt die Selektion häufig auch zu einem gewissen Teil stochastisch, so daß zufällig gewählte Individuen zur Erhaltung der Diversität einer Population, Eltern der nächsten Generation werden können. Verbreitet sind sog. Elitistenstrategien, bei denen eine bestimmte Anzahl von besten Individuen (**Elitist**) einer Population stets von einer Generation zur nächsten Generation weitergegeben wird, um ihr „gutes“ Erbmaterial, bspw. während der Rekombination, weitergeben zu können.

Im Anschluß erfolgt die **Rekombination** von zwei oder mehr Individuen mit einer gewissen Wahrscheinlichkeit p_c, der, in Anlehnung an das biologische Vorbild, sog. **Crossover-Rate**. Das Crossover dient dem Datenaustausch zwischen Individuen, wobei die Hoffnung besteht durch den Austausch möglicherweise besonders „nützlicher" Teile der Individuuen, größere Sprünge im Suchraum in Richtung einer Lösung des Problems zu machen. Abbildung 2 zeigt Beispiele zum z-Punkt Crossover, bei dem z Bruchstellen innerhalb der Individuen existieren, an denen Chromosomenabschnitte ausgetauscht werden.

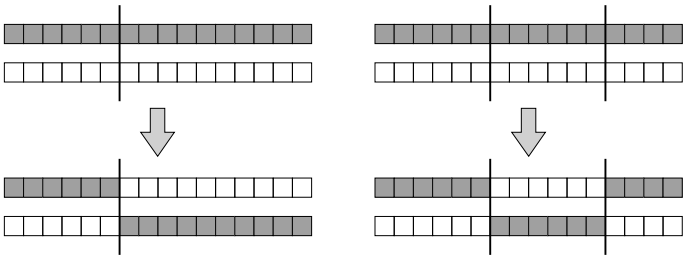


Abbildung 2 Darstellung des z-Punkt Crossover. Links für $z = 1$, rechts für $z = 2$.

Weitere Varianten der Rekombination bilden das Uniform sowie das Multi-Parent Crossover bei denen zum einen an jeder Position des Bitstrings jeweils eines der Gene der beteiligten Eltern-Individuen gewählt wird, zum anderen die Anzahl der an der Rekombination beteiligten Individuen ≥ 2 ist.

Zuletzt erfolgt die zufällige (und ungerichtete) **Mutation** einzelner Bits eines Individums. Dabei wird jeder Position eines Chromosoms die Wahrscheinlichkeit p_m (die sog. **Mutationsrate**) zugeordnet, mit der es mutiert werden kann. Im Falle einer Mutation wird aus der Menge der zulässigen Symbole eines zufällig ausgewählt und der entsprechenden Position des Genoms zugewiesen. Im Falle binärer Strings ist das Vorgehen einfacher: Das Bit an der entsprechenden Position wird invertiert (geflippt) oder mit einem aus der Menge $\{0, 1\}$ zufällig gewählten Wert belegt.

2.2 Programmierbare Logik-Geräte

Durch die anhaltenden Fortschritte bei der Fertigung hochintegrierter Schaltkreise und der regelmäßigen Steigerung der Chip-Leistung ist es mittlerweile möglich nicht nur VLSI-, sondern auch programmierbare Logik-Schaltkreise herzustellen. Solche **Programmable Logic Devices** (PLD) stellen eine ganze Familie von programmierbaren Schaltkreisen dar, die i. d. R. aus einem Array (Feld) von einfachen **Logikelementen** wie bspw. AND, OR, FLIP-FLOP, etc. bestehen. Ein solches Gerät ist programmierbar, wenn seine Funktionsweise durch den Anwender konfiguriert werden kann; ist dies beliebig oft möglich, so spricht man von einer **Reconfigurable Programming Unit** (RPU). Die Funktion, die ein derartiger Chip erfüllen soll, wird durch einen Bit-String kodiert, der die Konfiguration des Chips repräsentiert. Der Unterschied zwischen der Programmierung eines Mikroprozessors und der Programmierung eines solchen Schaltkreises ist damit offensichtlich: Mikroprozessoren werden durch die Spezifikation einer Folge von Instruktionen (Algorithmus) programmiert während bei programmierbaren Schaltkreisen die Maschine selbst, zumeist auf Gatter-Ebene, konfiguriert wird.

VLSI = Very Large Scale Integration: Höchstintegration von Schaltkreisen mit 50 000+ Gatter/Chip (Kolla, Molitor, und Osthof 1989). Seit 1986 existiert die Ultra Large Scale Integration (ULSI) mit mehr als 1 000 000 Gatter/Chip. Weitere Schritte zu höheren Integrationsdichten umfassen die Wafer Scale Integration (WSI), bei der über den ganzen Wafer integriert wird (Chapman 1997).

Wie eingangs erwähnt enthält die Familie der PLDs verschiedene Klassen, so u. a. ASICs, PLAs, PALs, PROMs sowie FPGAs, welche im folgenden eingehender diskutiert werden sollen. Insbesondere die letztgenannte Klasse wird einer ausführlicheren Betrachtung unterzogen, da die in dieser Arbeit verwendete RPU, ein XC6216 der Firma XILINX, zur Klasse der FPGAs gehört. Nähere Informationen zu PLDs, deren Aufbau und Programmierung liefert u. a. das Buch von Tietze und Schenk (1989).

2.2.1 ASIC

ASICs sind **Application Specific Integrated Circuits**, die zwar programmiert werden, jedoch im Gegensatz zu anderen PLDs sowohl digitale, analoge als auch eine Mischung von digitalen und analogen Funktionen bereitstellen können. Die Programmierung von Anwenderseite ist möglich, jedoch nicht sehr verbreitet; zumeist erfolgt die Programmierung anhand von Anwender-Spezifikationen beim Hersteller des ASICs. Haupteinsatzgebiet sind kleine Serien hochintegrierter, sehr anwendungsbezogener Schaltkreise.

2.2.2 Programmierbare Festwertspeicher (PROM)

Unter PROMs, den **Programmable Read Only Memory** Chips, versteht man Festwertspeicher, deren Inhalte vom Anwender programmierbar sind. Die ersten nicht-löschbaren PROMs verwandten als programmierbare Bauelemente meist Schmelzsicherungen, die in den integrier-

ten Schaltungen durch besonders dünne Metallisierungsbrücken realisiert wurden. Alternativen bilden die UV-löschbaren Festwertspeicher (Erasable PROMs = EPROMs), die aber den Nachteil haben, daß stets der komplette Chip neu programmiert werden muß, selbst wenn nur kleine Änderungen des Schaltkreises erfolgen sollen. Um diesem Nachteil entgegenzuwirken wurden später die elektrisch löschbaren Festwertspeicher (Electrically Erasable PROMs = EEPROMs) entwickelt, deren Programmierung byteweise möglich ist, d. h. durch Anlegen einer Speicheradresse des EEPROMs kann die entsprechende Zelle mit einem Wert beschrieben werden.

Den jüngsten Trend in der PROM-Technologie stellen die Flash-Speicher oder auch Flash-EEPROMs dar, die ein Realisierung zwischen EPROMs und EEPROMs sind. Sie sind wie die EEPROMs elektrisch löschbar, jedoch müssen sie wie die EPROMs stets komplett neu beschrieben werden – Änderungen einzelner Bytes sind nicht möglich. Trotzdem werden Flash-Speicher vermehrt in Verbraucher-Produkten wie bspw. BIOSs von Motherboards, Modems, etc. eingesetzt, da sie auch dem unversierten Anwender das vergleichsweise einfache Aktualisieren produktspezifischer Programme ermöglichen.

2.2.3 PLAs und PALs

Die **Programmable Logic Arrays** (PLAs) und **Programmable Array Logics** (PALs) unterscheiden sich vornehmlich in der Flexibilität ihrer Programmierbarkeit. Bei der Realisierung logischer Funktionen mit Hilfe von PLAs oder den, aufgrund der einfacheren Programmierung, besonders populären PALs, macht man sich die Tatsache zunutze, daß jede beliebige boolesche Funktion f in die **Disjunktive Normalform** (DNF) überführt werden kann. Die DNF ist die eindeutige Darstellung von f als Disjunktion der Minterme ihrer einschlägigen Indizes. Angenommen f ist in Form einer Funktionstabelle wie zur Rechten gegeben, dann sind die Minterme genau jene $\wedge$-Verknüpfung von x_1, x_2 und x_3 bei denen die Funktion f eine Eins liefert (einschlägiger Index). Für die gegebene Funktion $f = x_1 \oplus x_2 \overline{\wedge} x_3$ sind somit die Minterme $\overline{x_1}\overline{x_2}\overline{x_3}$, $\overline{x_1}\overline{x_2}x_3$, $\overline{x_1}x_2\overline{x_3}$, $x_1\overline{x_2}\overline{x_3}$, $x_1x_2\overline{x_3}$ und $x_1x_2x_3$ Die DNF lautet dann $f = \overline{x_1}\overline{x_2}\overline{x_3} \vee \overline{x_1}\overline{x_2}x_3 \vee \overline{x_1}x_2\overline{x_3} \vee x_1\overline{x_2}\overline{x_3} \vee x_1x_2\overline{x_3} \vee x_1x_2x_3$. Diese DNF stellt eine vollständige Beschreibung der Funktion f dar, sie ist jedoch nicht die *minimale* Darstellung. Ein Vorteil der Normalform ist, daß sie nur zwei logische Stufen hat und damit bei Verwendung der richtigen Basis mit unbeschränktem Fan-In in Tiefe zwei realisierbar ist. Man beschränkt sich hierbei i. d. R. auf die Klasse der Tiefe-2-Schaltkreise über der vollständigen Basis $U = \{\neg, \wedge, \vee\}$, die für den Fall der Disjunktiven Normalform $\sum_2$-Schaltkreise heißen, da der Summentyp $\vee$ auf der zweiten Ebene realisiert wird. Abbildung 3 zeigt den internen Aufbau von PLAs (links) und PALs (rechts), die beide auf dem Prinzip

x_1	x_2	x_3	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Für die Minimierung boolescher Funktionen stehen verschiedene Verfahren wie bspw. der Quine/McCluskey Algorithmus zur Verfügung, vgl. Wegener (1989). Ein Minimalpolynom für f lautet z. B. $f = \overline{x_1x_2} \vee x_1x_2 \vee \overline{x_3}$.

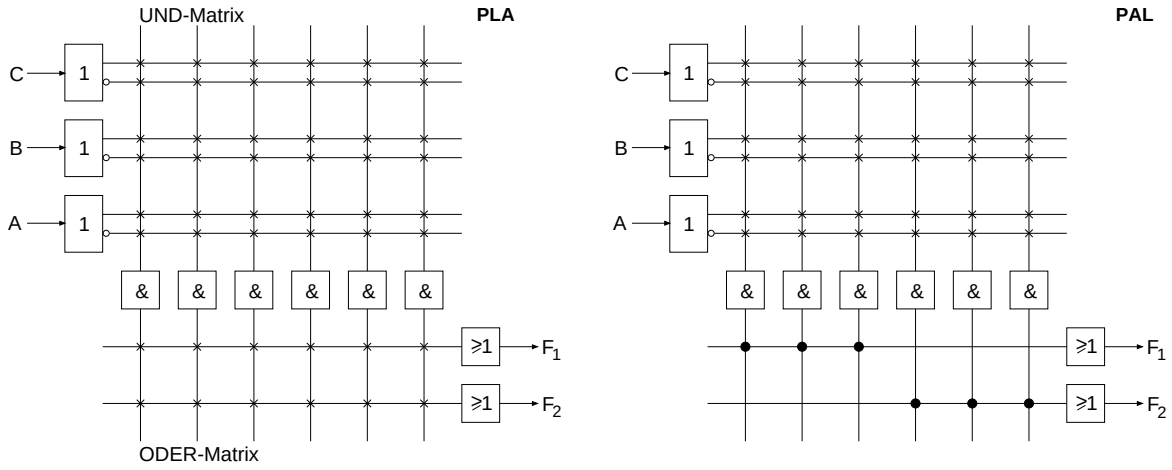


Abbildung 3 Schematischer Aufbau von PLAs und PALs. Programmierbare Verbindungen sind durch ein $\times$ gekennzeichnet, ein $\bullet$ markierte feste Verbindungen.

eines konfigurierbaren $\sum_2$-Schaltkreises basieren. Das Maß der Konfigurierbarkeit bestimmt dabei die Komplexität der Programmierung. Je mehr Freiheiten bei den Verbindungen herrschen, desto komplizierter wird die Programmierung des PLDs. Bei den Programmable Logic Arrays sind sowohl die UND-Matrix der ersten Stufe als auch die ODER-Matrix der zweiten Stufe programmierbar. PLAs sind somit sehr flexibel, allerdings erfordert ihre Programmierung einen höheren Aufwand als dies bei den Programmable Array Logics der Fall ist. Diese besitzen ebenfalls eine programmierbare UND-Matrix in der ersten Stufe, jedoch sind die Verbindungen der ODER-Matrix festgelegt, vgl. Abbildung 3 (rechts). Sie bieten auf gleichem Raum wie die PLAs weniger Flexibilität, haben aber den entscheidenden Vorteil durch diese Beschränkung dem Anwender die Programmierung zu erleichtern.

Weiterführende Informationen zu PLAs und PALs liefern Tietze und Schenk (1989).

2.2.4 FPGAs

Ein **Field Programmable Gate Array** (FPGA) ist ein elektrisch programmierbares Feld (Array) von Zellen, das sehr hohe Gatterdichten, hohe Geschwindigkeit, benutzerdefinierte Ein- und Ausgaben sowie ein flexibles Konnektierungsschema miteinander verbindet. FPGAs sind nicht auf eine UND-ODER-Matrix beschränkt, wie dies bspw. bei den PALs oder PLDs der Fall ist, sondern sie besitzen eine Matrix programmierbarer **Configurable Logic Blocks** (CLB) und einen umgebenden Ring von **Input/Output Blocks** (IOB), der das Interface zu den Anschlußpins des FPGA darstellt. Jeder IOB kann unabhängig von allen anderen als Eingabe-, Ausgabe- oder auch bidirektionaler Pin konfiguriert werden. Ein CLB enthält eine programmierbare

kombinatorische Logikeinheit sowie eine unterschiedliche Anzahl von Speicherregistern (Flip-Flop). Die kombinatorische Einheit kann eine beliebige boolesche Funktion auf ihren Eingaben berechnen. Verbindungen zwischen einzelnen CLBs erfolgen durch ein Netzwerk aus horizontalen und vertikalen Linien zwischen den CLBs. Sowohl die Verbindung der IOBs untereinander als auch die Funktion, die ein CLB berechnen soll, kann beliebig oft (re)konfiguriert werden. FPGAs gehören damit zur Familie der **Reconfigurable Programming Units** (RPU).

Bei der in dieser Arbeit verwendeten RPU handelt es sich um einen Field Programmable Gate Array (FPGA) aus der XC6200 Serie der Firma XILINX Inc., genauer um einen XC6216. Der XC6216 besteht aus einem **Gate-Array** programmierbarer Logik-Zellen, die durch eine bestimmte Infrastruktur miteinander verbunden sind und auf drei unterschiedlichen Ebenen programmiert werden können:

- Funktion der einzelnen Logik-Zellen,
- Verbindungen zwischen den Zellen (Infrastruktur),
- Inputs und Outputs.

Alle drei Ebenen werden durch einen Bit-String konfiguriert, der von einer externen Quelle (hier von einem Host-Rechner, in dem der FPGA-Chip auf einer PCI-Karte, installiert ist) in den Chip geladen wird.

Der XC6216 besteht aus einem großen Array von einfachen, konfigurierbaren Zellen, vgl. Abbildungen 4 (links) und 5. Jede Basis-Zelle enthält eine Berechnungs-Einheit die sowohl eine beliebige Logik-Funktion aus einer Menge von Funktionen als auch eine sog. **Routing Area** zur Verbindung einzelner Zellen bereitstellt. Die Komplexität jeder Zelle bestimmt die Strukturgröße des Arrays: Feinstrukturierte Chips enthalten nur einfache Logik-Zellen (bspw. XILINX XC6200 Serie), während Chips mit groben Strukturen (bspw. multiplexerbasierte Entwürfe von ALCATEL) ein weitaus größeres Funktionspotential aufweisen.

Die XC6200 Serie der Firma XILINX ist sehr verbreitet bei der Anwendung für EHW. RPUs aus dieser Reihe werden weltweit zur Forschung im Bereich der EHW eingesetzt. Weiterentwicklung und Support der XC6000-Serie wurde zugunsten der neuen Virtex-Familie von XILINX eingestellt.

Die Konfiguration einer XC6200 Einheit wird in einem integralen **Konfigurations-Speicher** (**Control Store**) gespeichert. Dadurch kann ein XC6200 schnell und beliebig oft rekonfiguriert werden. Der Konfigurations-Speicher kann in den Adress-Bereich eines Host-Computers eingeblendet werden; zusätzlich ist das das schnelle Rekonfigurieren von Teilen oder einer kompletten XC6200 Einheit möglich. Bei der Rekonfiguration von Teilen einer Einheit werden Schaltkreise, die in anderen Sektionen des FPGAs arbeiten, nicht beeinflußt.

Ausgabesignale der Funktions- oder Berechnungs-Einheiten eines XC6200 können durch einen Prozessor und Einsatz geeigneter Software ausgelesen werden. Prozessoren können Register einer Einheit lesen

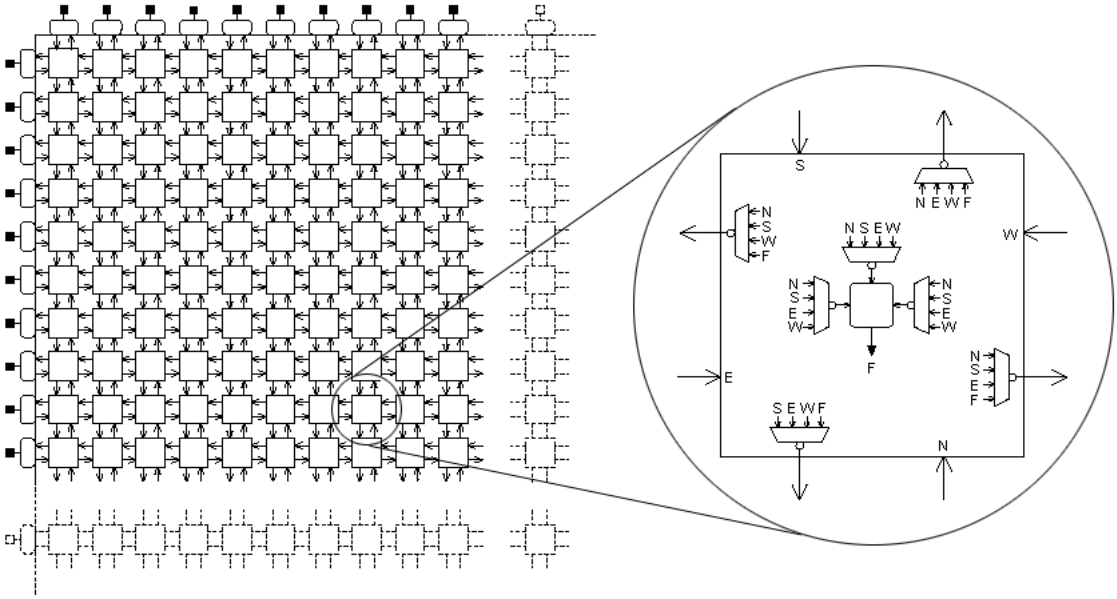


Abbildung 4 Vereinfachte Ansicht eines XC6216 FPGA nach Thompson (1998) und Xilinx Inc. (1997). Zur linken ist ein 10×10 Ausschnitt des kompletten 64×64 Zell-Arrays dargestellt. Auf der rechten Seite ist der Inhalt einer FPGA Zelle abgebildet: Die Funktions-Einheit befindet sich in der Mitte während die Function- und Routing-Multiplexer mit einem Kreis an ihrem Ausgang dargestellt sind.

und schreiben, wobei der Datentransfer mit 8, 16 oder 32 Bit über den Datenbus erfolgt.

Jede Zelle des XC6216 Zellarrays ist individuell programmierbar: Entweder als D-Type Register (Flip-Flop), als eine Logik-Funktion berechnende Zelle wie bspw. Multiplexer, Gatter, etc. oder aber als kombinatorische Funktion ohne Register. Weiterhin kann eine Zelle so konfiguriert werden, daß sie als einzige Funktion die Signale benachbarten Zellen weiterleitet. Tabelle 7 zeigt eine Übersicht aller verfügbaren Logik-Funktionen, die von einer Basis-Zelle berechnet werden können.

Die Struktur des Arrays selber ist hierarchisch aufgebaut. Benachbarte und miteinander verbundene Zellen sind in Gruppen zu 4×4 Zellen angeordnet, wobei diese Gruppen ihrerseits wieder ein Array bilden, indem diese 4×4 Gruppen mit anderen 4×4 Gruppen kommunizieren, vgl. Abbildung 5 (links). Ein 4×4 Array solcher 4×4 Blöcke stellt einen 16×16 Block dar. Im XC6216 Chip besteht der zentrale Array aus 64×64 Zellen, welcher aus einem 4×4 Array von 16×16 Blöcken besteht. Abbildung 5 (mitte) zeigt die Anordnung des 4×4 Arrays von 16×16 Blöcken. Daraus folgt, daß das Gate-Array des XC6216 aus insgesamt $4 \cdot 16 \times 4 \cdot 16 = 64 \times 64 = 4096$ Basis-Zellen besteht.

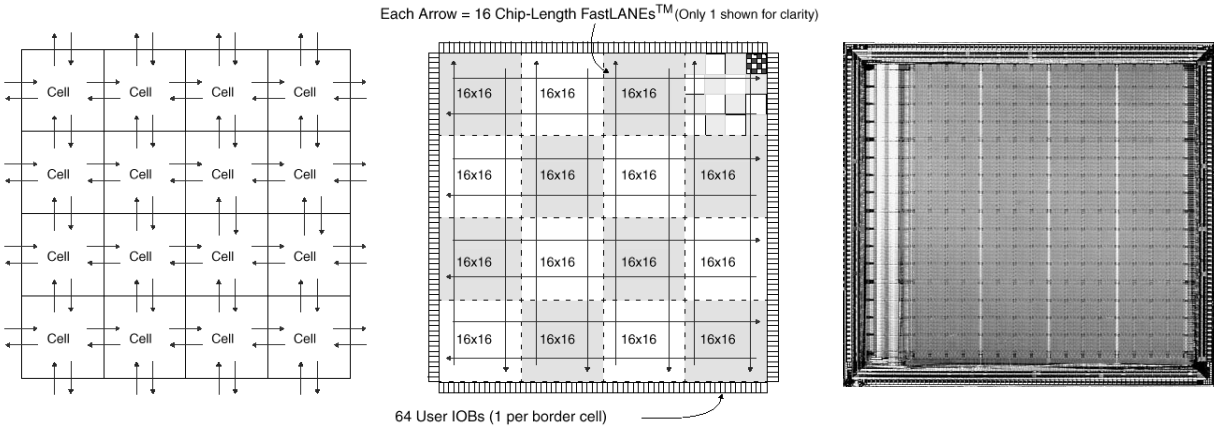


Abbildung 5 Schematische Darstellung der Zellanordnung in einem XC6216 FPGA (aus Xilinx Inc. 1997). Links: Ein 4×4 Block aus Basis-Zellen. Mitte: Gröbere Ansicht des kompletten 64×64 Array mit den den Chip umgebenden I/O-Pads. Rechts: Mikrophotographie der Zellstruktur eines „realen“ XC6216 FPGA in 20-facher Vergrößerung.

Einen detaillierten Einblick in den Aufbau einer Basis-Zelle gibt Abbildung 6. Die Eingänge benachbarter Zellen sind mit N, E, S und W gekennzeichnet, Ausgänge mit N_{out}, E_{out}, S_{out} und W_{out}. Weiterhin ist ein Takt- sowie ein asynchroner Clear-Eingang für das D-Type Register (Flip-Flop) der Berechnungs-Einheit vorhanden. Der Ausgang der Berechnungs-Einheit ist mit F gekennzeichnet.

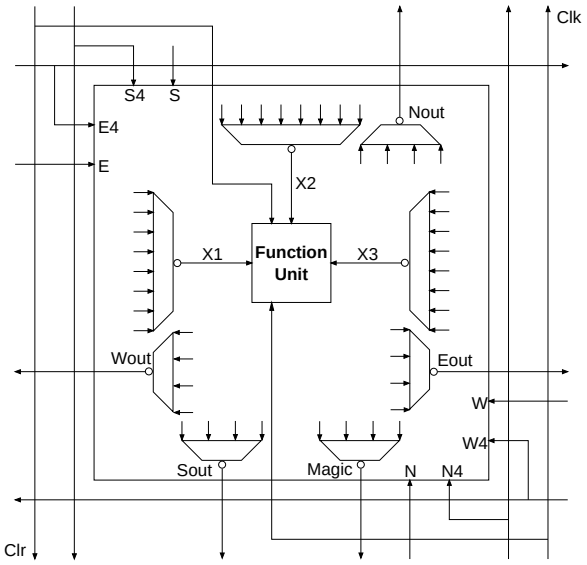


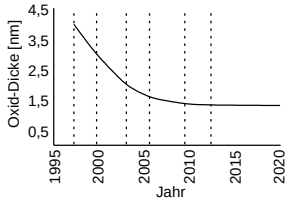
Abbildung 6 Detaillierte Darstellung einer XC6200 Basis-Zelle.

2.3 Evolvable Hardware

Dieser Abschnitt erläutert das Vorgehen zum evolutionären Entwurf digitaler Schaltkreise. Insbesondere sollen hier die Unterschiede zum herkömmlichen Schaltkreisentwurf verdeutlicht und den Eigenheiten des evolutionären Entwurfs gegenübergestellt werden.

Beim **herkömmlichen Schaltkreisentwurf** wird die Arbeit des Schaltkreisdesigners dadurch vereinfacht, daß von den realen Gegebenheiten der einem Entwurf zugrundeliegenden Hardware abstrahiert wird – der Designer braucht auf das analoge Verhalten der Transistoren keine Rücksicht zu nehmen, sondern er betrachtet sie als Zellen mit den logischen Zuständen 0 oder 1. Da der Schaltkreisdesigner nicht nur auf der Ebene von Transistoren arbeitet, sondern viel häufiger mit komplexen aber dennoch grundlegenden Logik-Bausteinen (zumeist aus einer Logik-Bibliothek) wie bspw. AND, OR, Multiplexer etc. arbeitet, ist es verständlich, daß dasselbe hohe Abstraktionsniveau auch für diese Elemente gilt. Durch diese Herangehensweise ergeben sich jedoch Probleme, wenn eine Umsetzung in entsprechende Hardware erfolgen soll. Nun müßen alle Eigenschaften von denen während des Entwurfs abstrahiert wurde, wie bspw. parasitäre Kapazitäten oder der nichtlineare Schaltverlauf von Transistoren, berücksichtigt werden. Es besteht somit der Wunsch nach einer Vereinfachung des Schaltkreisentwurfs, da dieser infolge stetig zunehmender Komplexität der Mikrochips immer schwieriger zu handhaben und fehleranfälliger wird. Auch die physikalischen Grenzen bei der zunehmenden Miniaturisierung von Mikrochips bereiten Probleme und das Ende des Mooreschen Gesetzes (Verdopplung der Chip-Komplexität alle 18 Monate, siehe Moore 1975) ist zumindest für Silizium-Chips in Sicht. Nach einem Bericht der Nature (Muller et al. 1999) haben Forscher von Lucent (Bell Labs) nachgewiesen, daß die Isolationsschichten von Silizium-Dioxid SiO_2 nicht beliebig verkleinert werden können. Im Moment sind Schichtdicken von etwa 4 nm gebräuchlich, das entspricht ca. 25 Atomlagen. Bei Schichtdicken von vier Atomlagen verliert das Oxid jedoch seine isolierende Wirkung, so daß gemäß der Chip-Roadmap der amerikanischen SIA im Jahr 2012 zumindest SiO_2 als Isolator nicht mehr verwendbar sein dürfte.

Fehler im Design der Fließkommaeinheit des INTEL Pentium Prozessors führten zu fehlerhaften Berechnungen und zwangen INTEL zu einer großangelegten Umtauschaktion (Stiller 1997).



Siliziumdioxid-Roadmap nach Stiller (1999).

Damit wird verständlich warum nicht nur der Miniaturisierunprozeß an sich einer stetigen Optimierung unterliegt, sondern daß auch bei dem *Entwurf* von Mikrochips neue Wege erforscht werden müssen. Als vergleichsweise junger Trend zum Schaltkreisentwurf steht dem herkömmlichen Design seit geraumer Zeit ein evolutionär motivierter Ansatz gegenüber — die Evolvable Hardware. Bei diesem **evolutionären Schaltkreisentwurf** können unter Verwendung der in Abschnitt 2.2 vorgestellten rekonfigurierbaren Hardware sowie Evolutionärer Algorithmen (vgl. Abschnitt 2.1) automatisch Schaltkreise ge-

funden werden, die ein vorgegebenes Problem lösen. Der evolutionäre Schaltkreisentwurf läßt sich nach Tomassini und Sipper (1997) anhand der Verwendung der beteiligten Hardware in Form von bspw. FPGAs in vier Klassen einteilen:

(1) **Offline Evolution.** Bei dieser – auch Simulated Circuit Evolution – genannten Technik des evolutionären Schaltkreisentwurfes werden alle Operationen durch Software ausgeführt, wobei die endgültige Lösung, also der evolutionär gefundene Schaltkreis, in einem FPGA o. ä. realisiert wird. Dies ist möglicherweise eine brauchbare Entwurfs-Technik, die allerdings in den Bereich der herkömmlichen Evolutions-Techniken fällt.

(2) **Semi Online Evolution.** Während des Evolutions-Prozesses wird ein echter Hardware-Schaltkreis benutzt, jedoch wird die Population in einem externen Rechner (Host-PC) gespeichert, der auch die restlichen Operationen wie Rekombination, Mutation und Selektion kontrolliert.

(3) **Online Evolution.** In dieser Klasse werden sowohl die Operationen des Evolutionären Algorithmus als auch die Fitneßbewertung in der Hardware durchgeführt, also ohne jeden Kontakt zu einem externen Rechner. Allerdings ist diese Form der Evolution nicht *open-ended*, d. h. es gibt ein fest vorgegebenes Ziel und keine dynamische Umgebung.

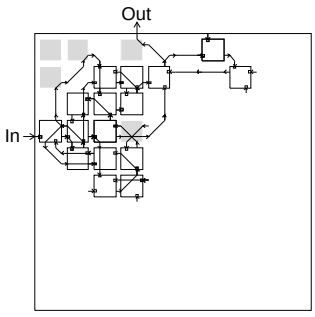
(4) **True Online Evolution.** Die letzte Klasse der Evolvable Hardware verfügt über eine Population von Hardware-Entitäten, die in einer open-ended Umgebung evolvieren sollen. Im Gegensatz zu derzeitigen Techniken künstlicher Evolution, die durch ihre Art der Fitneßbewertung in Zusammenhang mit der zu lösenden Aufgabe eine Form zielgerichteter Evolution darstellen, steht die *open-ended* Evolution, wie sie in der Natur vorkommt. Diese verfügt über kein explizit auferlegtes, sondern ein implizites, aufstrebendes und dynamisches Fitneßkriterium. Nach Tomassini et al. kann nur eine solche Form der Evolution innovativ sein: „Open-ended, undirected evolution is the only form of evolution known to produce such devices as eyes, wings, and nervous systems, and to give rise to the formation of species“ (Tomassini et al. 1997, Seite 10).

Im Rahmen dieser Arbeit wurde ein System in der Semi Online Evolution Klasse implementiert, nähere Informationen zum experimentellen Aufbau befinden sich in Kapitel 5.

2.4 Überblick

Der *Call for Papers* des diesjährigen *NASA/DoD Workshop on Evolvable Hardware* läßt vermuten, daß diese Technologie einen nicht unerheblichen Einfluß auf die Belange der Raumfahrt- als auch der Rüstungsindustrie haben wird: „The emerging field of Evolvable Hardware is expected to have major impact on deployable systems for space missions and defense applications [. . .]“ (NASA/DoD 1999). Aber nicht nur in diesen Bereichen scheint der EHW eine vielversprechende Zukunft bevorzustehen, sondern auch bei kommerziellen Applikationen die von „human-oriented hardware interfaces“ über „internet adpative hardware“ bis hin zu „automotive applications“ reichen.

In diesem Abschnitt werden einige Arbeiten verschiedener EHW-Gruppen als Vertreter aktueller Forschungstätigkeit im Bereich der Evolvable Hardware vorgestellt. Dabei kann und soll es sich nur um einige wenige interessante Auszüge handeln, da, trotzdem es sich bei der Evolvable Hardware um ein vergleichsweises junges Forschungsgebiet handelt, bereits jetzt die Zahl der Arbeiten nahezu unüberschaubar ist. Einen Blick in die zukünftige Entwicklung Evolvierbarer Systeme wagt nicht nur die NASA bzw. das DoD sondern auch bspw. Kitano (1996) oder Yao et al. (1999).



Als einer der ersten Forscher auf dem Gebiet der Evolvable Hardware konnte **Adrian Thompson** an der Universität von Sussex eine „non toy application“ für den evolutionären Schaltkreisentwurf vorstellen. Er war in der Lage durch Verwendung eines Genetischen Algorithmus zusammen mit einem rekonfigurierbaren FPGA der Firma XILINX (ein XC6216) einen Schaltkreis zu evolvieren, der zwischen zwei Frequenzen von 1 kHz und 10 kHz, unterscheiden kann (Thompson 1996, Thompson 1998); die Abbildung zur linken zeigt diesen Schaltkreis. Das besondere an Thompsons Vorgehen ist die Art und Weise, in der ein Schaltkreis die Eigenschaften bzw. Eigenheiten des Chips verwenden darf, auf dem er instantiiert wird: „[. . .] there is a fundamental decision to be made: whether the evolutionary process is free to explore any possible design, or whether it is constrained to encourage ‘sensible’ circuits more like those arising from conventional design methods“ (Thompson und Layzell 1999, Seite 1). Während Thompson den erstgenannten Ansatz verwandte, war er in der Lage einen Schaltkreis zu evolvieren, der zwar eine Lösung für das gegebene Problem der Frequenzunterscheidung lieferte, der jedoch weder in anderen Bereichen ein und desselben Chips noch auf anderen Chips instantiiert lauffähig war. Sogar das Verändern bestimmter, eigentlich nicht vom Schaltkreis verwendeter Zellen – wie bspw. der grau dargestellten in der Abbildung (links) – kann zu Störungen beim Betrieb des entsprechenden Schaltkreises führen. Darüberhinaus zeigte der Schaltkreis keine hohe

Temperaturtoleranz und dessen Funktionsweise war zu alledem auch noch äußerst schwierig zu analysieren, siehe Thompson und Layzell (1999).

Als Konsequenz dieser Entdeckung versuchte Thompson durch gleichzeitiges Evolvieren auf vier verschiedenen Chips, die unterschiedlichen äußeren Einflüssen wie Hitze, Kälte etc. ausgesetzt sind, tolerantere Schaltkreise zu finden. Diese Toleranz sollte u. a. eine größere Betriebsstabilität bei Temperaturschwankungen sowie eine gewisse Unabhängigkeit vom verwendeten Chip sicherstellen.

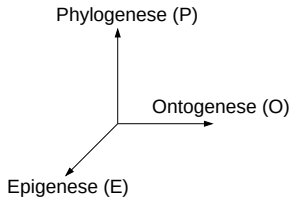
In jüngster Zeit untersuchte Adrian Thompson die Anwendung evolutionärer Techniken im Hinblick auf neue Technologien. In Thompson (1999) präsentiert er die Ergebnisse einer einfachen Studie: Die Evolution eines NOR-Gatters in einem Meso-/Nano-Scale Medium basierend auf „few/single-electron effects in tunnel junctions." Die Ergebnisse sind vielversprechend, da ein funktionierendes NOR-Gatter evolutionär gefunden werden konnte. Jedoch wurden die Experimente ausschließlich simuliert und es wurde von vereinfachenden Vorraussetzungen ausgegangen, wie bspw. der Simulation bei 0 K oder der Nichtbeachtung von Co-Tunneling. „This experiment is very preliminary, but illustrates one way that evolutionary design techniques may help novel technologies to become viable" (Thompson 1999).

An einem ähnlichen Projekt zum evolutionären Design von Schaltkreisen im nano-meter Bereich (Nanotechnologie) arbeiten **Adrian Stoica** et al. (1999) und **Didier Keymeulen** vom Jet Propulsion Laboratory am California Institute of Technology. Dabei legen sie die verbreitete Definition für Nanotechnologie als „the synthesis and the integration of materials and process devices at the level of molecule" (Stoica et al. 1999, Seite 1273) für ihre Experimente zugrunde. Einer der wichtigsten Aspekte der Nanotechnologie ist dabei die Ausnutzung der Materialunterschiede auf atomarer Ebene, die die Grundlage für die quantenmechanische Funktionalität derartiger Geräte wie bspw. „Resonant Tunneling Diodes (RTD)" darstellt. Ziel ihrer Arbeit ist die Untersuchung der Effekte struktureller Veränderungen sowie „doping variations" im Hinblick auf den Elektronentransport in nanotechnologischen Geräten; in Stoica et al. (1999) wurden diese Untersuchungen durch Simulationen anhand von RTDs durchgeführt.

In Sankt Augustin an der GMD (Gesellschaft für Mathematik und Datenforschung) entwickeln **John McCaskill**, **Uwe Tangen** und **Bernd Senf** am Institut für molekulare Biotechnologie (IMB) einen rekonfigurierbaren Parallel-Rechner mit Namen 'Polyp'. Dieser Computer soll in der Lage sein Schaltkreise zu evolvieren, er wird jedoch später für die Untersuchung der Evolution von Molekülen verwendet. Das Ziel

dieser Forschung soll durch Verzahnen der elektronischen und chemischen Evolution ein „Evolvierbarer DNA-Rechner“ sein, der in der Lage ist Moleküle mit einem bestimmten Verhalten, aber vollständig unbekannter Struktur zu entwerfen (Tangen und McCaskill 1998).

Am Logic Systems Laboratory (LSL) des Swiss Federal Institute of Technologiy in Lausanne arbeiten u. a. **Moshe Sipper** und **Gianluca Tempesti** an der Entwicklung *ontogenetischer* Hardware. Diese Hardware repräsentiert eine Achse des von Sanchez et al. (1997) und Sipper et al. (1997) eingeführten POE Modells biologisch inspirierter Systeme. Dieses Modell basiert auf der Beobachtung, daß bei der Entwicklung zwischen drei Organisationsebenen unterschieden werden kann: Die phylogenetische Ebene bezieht sich auf die zeitliche Entwicklung des genetischen Programms innerhalb von Individuen und Spezien. Die ontogenetische Ebene beschreibt den Entwicklungsprozess eines Organismus durch Zellteilungen angefangen bei einer einzelnen Mutterzelle, der Zygote, bis hin zum vollständigen Organismus. Die epigenetische Ebene stellt den Lernprozess während der Lebensspanne eines Individuums dar.

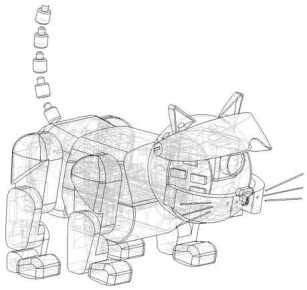


POE-Modell nach Sipper, Mange, und Stauffer (1997).

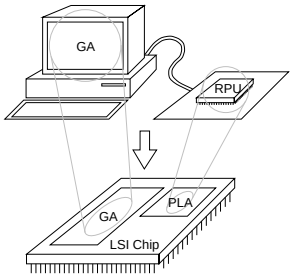
Unter Verwendung rekonfigurierbarer Schaltkreise (FPGAs) und Beschränkung auf die ontogenetische Achse ihres POE Modells, vgl. Abbildung zur Linken, konnten Sipper et al. (1997) zunächst das von Neumann Modell sich selbst replizierender zellulärer Automaten in Hardware implementieren. Später zeigten Mange und Stauffer (1994) ebenfalls vom LSL, wie die Implementierung einer künstlichen Zelle, genannt *Biodule* (Biological Module), auf FPGAs möglich ist. Solche Biodule Zellen werden, in einer als *Empryonics* bezeichneten Architektur, als elementare Einheiten verwendet, aus denen multizelluläre Organismen aufgebaut werden können, die sich ontogenetisch entwickeln, um eine Aufgabe zu erfüllen. Dieser Ansatz wurde von Tempesti (1995) zu einem in Hardware implementierten zellulären Automaten weiterentwickelt, der nicht nur zur Selbstreproduktion in der Lage ist, sondern an den auch ein Programm angehängt werden kann, welches dupliziert und in jeder Kopie ausgeführt wird. Bis 1997 konnten mehrere Applikationen der Biodules vorgestellt werden, darunter ein Zufallszahlengenerator, eine universelle Turingmaschine sowie eine „Biowatch“, die einen sich selbst-reproduzierenden und selbst-reparierenden Systems aus vier Biodules darstellt und Minuten sowie Sekunden zählen kann.

Hugo de Garis forscht derzeit am Advanced Telecommunications Research Institute (ATR), einem japanischen Forschungszentrum für Computer- und Kommunikationstechnik in Kyoto (Japan). de Garis, der sich selber als „Brain Builder“ bezeichnet, plant die Entwicklung künstlicher Gehirne mit Milliarden von Nervenzellen. Solche Artilects (Artificial Intellects) sollen auf der Basis zellulärer Automaten im Rahmen einer CBM, der CAM-Brain Machine, verwirklicht werden. In Korkin et al. (1997) wurde die CBM als ein Hardware System vorgestellt „[. . .] capable of evolving thousands of neural network modules in a matter of minutes and running a simulation of a million neuron modular artificial brain in real time“ (Seite 1). Die CBM besteht im wesentlichen aus mehreren dutzend Xilinx XC6264 FPGAs auf denen einige tausend zelluläre 3D-Automaten (CA) Zellen mit etwa 100 Neuronen je FPGA implementiert werden. Ohne auf weitere Details der CBM einzugehen, ist in Korkin et al. (1997) von de Garis Plänen der Entwicklung einer Roboter-Katze (Robokoneko) auf der Basis des CBM Hardware-Systems zu lesen: „Once the kitten brain architecture is ready, and the CBM as well, the year 1998 will be spent in evolving the modules and testing the brain inside the kitten robot“ (Korkin et al. 1997, Seite 503). Die ehemals hoch gesteckten Ziele mußten jedoch infolge der Komplexität des geplanten Unterfangens korrigiert werden. In einem zur Veröffentlichung stehenden Paper schreibt de Garis: „The real challenge will very likely be the creation of artificial brain architectures.“ (de Garis et al. 1999), und tatsächlich existieren bisher lediglich Simulationen des CBM Modells: „This article details how the authors are simulating and evolving the neural network controlled motions of a life sized kitten robot [. . .]“ (de Garis et al. 1999, Seite 1). Es bleibt abzuwarten, ob de Garis Ansatz in einigen Jahren zum Erfolg führen wird. Fest steht jedoch, daß bereits jetzt die verwendete Hardware (Xilinx XC6264) veraltet ist, so daß bis bis zum Jahr 2001 mit dem Nachfolger Virtex der XC6000-Reihe von XILINX eine zweite Version der CBM entwickelt werden soll, die dann bis zu 40 Millionen künstliche Neuronen besitzen kann.

Sowohl Hugo de Garis als auch Tetsuya Higuchi beanspruchen für sich 1992 erster mit der Idee von Evolvable Hardware gewesen zu sein. Interessant ist die Tatsache, daß Higuchi und de Garis vor ’92 zunächst zusammenarbeiteten, ab dann jedoch ihre Vorstellungen von EHW und deren Realisierung getrennt weiterverfolgten.



Tetsuya Higuchi leitet die größte Forschungsabteilung im Bereich evolvierender Chips am Evolvable Systems Lab des Electrotechnical Laboratory (ETL) in Tsukuba (Japan). Dabei verwendet er keine standardisierten FPGAs wie bspw. die XC6000er Familie von XILINX, sondern er entwickelt speziell auf seine Projekte zugeschnitte EHW-Chips. In dieser Forschungsabteilung werden verschiedene Projekte bearbeitet, so u. a. ein autonom navigierender Roboter, der einen Ball verfolgen kann (Keymeulen et al. 1998); eine Handprothese die sich durch Verwendung von EHW an seinen Benutzer anpaßt (Kajitani et al. 1999); oder auch ein spezieller EHW-Chip, der sowohl die Operationen eines Genetischen Algorithmus als auch die rekonfigurierbare Hardware auf einem einzigen Chip vereint.



Letztgenannte Applikation stellt einen interessanten Ansatz zur Integration *aller* Komponenten eines Evolvable Hardware Systems auf einem hochintegrierten Chip dar, vgl. Abbildung zur Linken. Dabei führt eine erfolgreiche Umsetzung dieses Ansatzes zu einem EHW-System, daß, bezogen auf die in Abschnitt 2.3 vorgestellten Evolutionstypen, in der Klasse der Online Evolution einzuordnen wäre. Nicht nur die Fitneßbewertung würde in diesem Fall mit Schaltgeschwindigkeiten im Nanosekundenbereich arbeiten, sondern der komplette Evolutionäre Algorithmus. In Kajitani et al. (1999) wird das Vorgehen zur Einbettung von EHW auf einem Chip näher beschrieben, wobei die Autoren auch auf die Implementierungsprobleme für den effektiven Einsatz Genetischer Algorithmen auf Chips eingehen: Allen voran steht der begrenzte Speicherplatz auf einem LSI-Chip, der sich nachteilig sowohl auf die Anzahl der speicherbaren Individuen als auch bspw. auf die Art der Selektionsstrategie auswirkt: „Generational selection strategies which are used in simple GAs need extra memory to temporarily preserve selected individuals“ (Kajitani et al. 1998, Seite 3). Für den Crossover-Operator wird darüberhinaus ein Zufallszahlengenerator benötigt, dessen Implementierungsgröße von der Länge der betrachteten Individuen abhängt. Anhand einer Handprothese wurde von Kajitani et al. (1999) die erfolgreiche Anwendung eines derart konstruierten EHW-Chips für ein „human-oriented hardware interface“ gezeigt. Keymeulen et al. (1998) konnten zeigen, daß auch in der Robotik die erfolgreiche Anwendung eines solchen EHW-Systems für einen Robot-Controller möglich ist.

3 Die Programmierung der XC6216 RPU

> *We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers.*
>
> *– Dijkstra (1972)*

Zum Verständnis der Programmierung des XC6216 läßt sich zunächst feststellen, daß zwar ausreichend Dokumentation in Form eines **Logic Programmable Data Book** (Xilinx Inc. 1997) vorhanden ist, sich diese jedoch eher an den versierten Schaltkreisdesigner wendet, denn an Novizen auf diesem Gebiet. Darüberhinaus wird schnell klar, daß die Programmierung des XC6216 „zu Fuß“ ungleich komplexer ist, als der Entwurf eines Schaltkreises unter Verwendung einer Beschreibungssprache wie bspw. VHDL und dem anschließenden automatischen Generieren einer entsprechenden Bitfolge, die dann direkt in den FPGA geladen werden kann. Zwar werden von XILINX verschiedene Bibliotheken mitgeliefert, die das Programmieren der RPU vereinfachen sollen, jedoch ist die Dokumentation der Schnittstellen nicht nur dürftig, sondern die Bibliotheken enthalten mitunter auch Fehler.

Aus diesem Grund bestand ein Teil dieser Arbeit darin herauszufinden, wie sich der XC6216 konfigurieren läßt, d. h. insbesondere welche Daten an welche Orte des FPGA geschrieben werden müssen, um bspw. in einer beliebigen Zelle des Array eine bestimmte Zellfunktion berechnen lassen zu können. Auch bei der Eingabe von Signalen in einen Schaltkreis, ebenso wie beim Auslesen von Berechnungsergebnissen der Funktionseinheiten galt es Besonderheiten zu beachten, die in Anhang A näher erläutert werden. In diesem Abschnitt sollen die grundlegenden Schritte vorgestellt werden, mit denen die Konfiguration des XC6216 von einem Host-Rechner (PC) aus möglich ist. Zwei einfache aber ausführliche Beispiele inklusive des entsprechenden C++ Codes sind in Anhang B zu finden.

3.1 Grundsätzliches

Die binären Daten, mit denen der XC6216 konfiguriert werden soll, müssen auf irgendeine Art und Weise in den XC6216-Chip gelangen. Hierzu existieren generell zwei verschiedene Möglichkeiten: Zum einen können die Daten über ein serielles Interface in die RPU übertragen werden, zum anderen erlauben es die von XILINX mitgelieferten Hardware-Treiber, direkt in den **Control Store** (Konfigurations-SRAM, Kontrollspeicher) des XC6216 zu schreiben. Bei der letztgenannten Variante, die auch in dieser Arbeit verwendet wurde, erfolgt ein **Mapping** des Control Store in den Adressraum des Host-Prozessor, vgl. Abbildung 7.

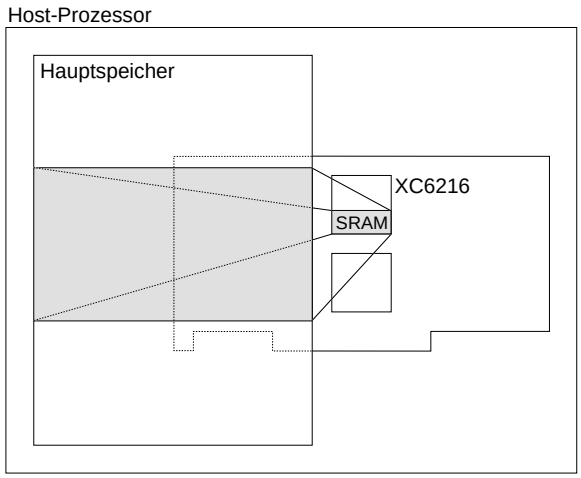


Abbildung 7 Der Kontrollspeicher (SRAM) des XC6216 wird durch die von XILINX gelieferten Treiber in den Hauptspeicher des Host-Prozessors abgebildet.

Schreiben und Lesen des Kontrollspeichers stellen sich somit als gewöhnliche Zugriffe in den Hauptspeicher des Hosts dar. Der Zugriff auf den gemappten Speicherbereich erfolgt über ein paralleles CPU Interface, das von XILINX mit **FastMAP** benannt wurde. FastMAP erlaubt den transparenten Umgang sowohl mit dem Konfigurations-SRAM als auch mit den Logik-Zellen, da beides dem Anwender wie konventioneller Hauptspeicher (des Host-Prozessors) erscheint.

3.2 Der Bus

Trotz der im vorigen Kapitel beschriebenen Abbildung (Mapping) des Kontrollspeichers in den Hauptspeicher des Hostprozessors müssen alle Daten – sowohl jene die aus dem XC6216 gelesen als auch in ihn geschrieben werden sollen – in den Kontrollspeicher der RPU gelangen, welche auf dem PCI-Board angesiedelt ist. Dies geschieht über den

PCI-Bus des PCs, d. h. die für den Anwender transparenten Lese- und Schreibvorgänge von und in den Teil des Hauptspeichers, in den der Control Store des XC6216 gemapped wurde, führen i. d. R. dazu, daß Daten über den PCI-Bus übertragen werden. Zugriffe auf den Kontrollspeicher erfolgen daher bei weitem nicht so schnell wie auf den Hauptspeicher, jedoch im Hinblick auf diese Arbeit mit befriedigender Geschwindigkeit.

```
**** Performance Test 1
**** Demonstrates:
**** Data transfer rate from PC to PCI/6200 SRAM.
**** Data transfer rate from PCI/6200 SRAM to PC.
****
**** Copyright (c) 1997 by Xilinx
****
CMD: reset
CMD: bctl 0
**** Write speed, using 4096 byte blocks
wspd 0 1000
Measuring PCI write bandwidth to RAM: blocksize = 4096
Wrote 16000 KB in 7.430000 s => 2153.432032 KB/s
**** Read speed, using 4096 byte blocks
rspd 0 1000
Measuring PCI read bandwidth from RAM: blocksize = 4096
Read 3200 KB in 1.843000 s => 1736.299512 KB/s
```

Abbildung 8 Bildschirmausgabe des `PCITestNT.exe` Programms. Als Eingabe wurde die Kommandodatei `prf1_tst.cmd` verwendet.

Abbildung 8 zeigt die Beispielausgabe eines Schreib-/Lese-Tests unter Windows NT 4.0 mit AMD K6/300MHz Prozessor. Gemittelt über 10 Läufe ergibt sich eine Transferrate von 2,1 MB/s zum Schreiben und 1.7 MB/s zum Lesen von Datenblöcken der Größe 4 kB. Interessant ist der Vergleich mit Windows 95; hier liegen die gemessenen Werte im Mittel um 4 MB/s (Schreiben) und 3 MB/s (Lesen), beide Transferraten sind etwa doppelt so schnell wie bei Windows NT. Über die Gründe für dieses Verhalten lassen sich ohne eingehende Analyse nur Vermutungen anstellen. Nichtsdestoweniger wurde für diese Arbeit Windows NT als Entwicklungsplattform eingesetzt, da zugunsten der größeren Stabilität des Betriebsystems auf höchste Transferraten verzichtet werden konnte.

3.3 Adressierung des Kontrollspeichers

In Abschnitt 3.1 wurde beschrieben, daß der Kontrollspeicher (Control Store) in den Hauptspeicher des Host-Prozessor abgebildet wird. Es stellt sich nun die Frage, wie der Kontrollspeicher adressiert werden

muß, um bestimmte Teile der RPU konfigurieren zu können. Ausgehend von der Architektur des XC6216 sind die primären Einheiten für Schreib- bzw. Lese-Operationen:

- Zellen des Array,
- Zustände der D-Type Register,
- East/West Switches,
- North/South Switches,
- Input/Output Blocks (IOBs),
- Input/Output Pads,
- Device Control Register (DCR).

„Unter einer Adresse versteht man in der Digitaltechnik eine binärcodierte Information zur eindeutigen Auswahl (Adressierung) eines bestimmten Speicherwortes, eines Registers oder einer Schnittstelle als Quelle oder Ziel eines Datentransportes“ (Bähring 1994).

Der Bereich innerhalb des Kontrollspeichers ein jeder dieser Einheiten beginnt ab einer bestimmten Adresse im (gemappten) Hauptspeicher des Host-Prozessors, so daß der erste Schritt zunächst die Berechnung dieser Adresse darstellt, um dann im zweiten Schritt Daten an die entsprechenden Stelle schreiben oder von ihr lesen zu können. Das **Adressformat** lt. Xilinx Inc. (1997), Seite 20, sieht dabei wie in Tabelle 1 dargestellt aus.

Tabelle 1 Adressformat des XC6216

Mode[15:14]	Column[13:8]	Column Offset[7:6]	Row[5:0]

Dabei wird über die **Mode-Bits** angegeben, welcher Bereich des Kontrollspeichers angesprochen werden soll, also welche der o. g. Einheiten mit einem nachfolgenden Befehl beschrieben oder gelesen wird. Alle weiteren Bits für **Column**, **Row** sowie **Column Offset** liefern spezielle Informationen in Bezug auf den ausgewählten Adressmodus.

Tabelle 2 Belegungen der Mode-Bits einer Adresse.

Mode[15:14]	Column[13:8]	Column Offset[7:6]	Row[5:0]

00	Cell Mode	`XCAddr::CELLMODE`
01	East/West Switch oder IOB	`XCAddr::EWMODE`
10	North/South Switch oder IOB	`XCAddr::NSMODE`
11	Device Control Register (DCR)	`XCAddr::CONMODE`

Die vier möglichen Belegungen der Mode-Bits haben die in Tabelle 2 angegebene Bedeutung; Spalte 3 zeigt die entsprechenden Konstanten aus der Datei `XCAddr.h`. Wird also bspw. Mode[15:14]= 00 gesetzt, so kann im nächsten Schritt eine Zelle (re)konfiguriert oder deren Zustand (Inhalt ihres D-Type Registers) geschrieben werden. Die Bedeutung der Column, Row und Column Offset Felder richtet sich nach dem gewählten Adressmodus, so daß diese zusammen mit den einzelnen Adressmodi im folgenden diskutiert werden.

3.3.1 Cell Mode – 00

Über den **Zellmodus** läßt sich zum einen die komplette Funktionalität sowie Verknüpfung einer jeden Zelle des Zellarrays konfigurieren, zum anderen kann schreibend auf das D-Type Register und lesend auf das Berechnungsergebnis einer Zelle zugegriffen werden. Im Zellmodus werden die folgenden Teile einer Zelle konfiguriert:

- **Neighbour Routing**. Das Neighbour Routing legt fest welche Signale an die vier Ausgänge einer Zelle weitergeleitet werden. Dies können einerseits Eingabesignale sein, die auf diese Weise durch eine Zelle propagiert werden. Zum anderen kann jeder der vier Ausgänge einer Zelle das Funktionsergebnis der Berechnungseinheit weiterleiten. In Abbildung 9 (links) sind genau die Teile einer Zelle hervorgehoben, die durch Adressierung des Neighbour Routing angesprochen werden: Multiplexer sowie Ein- und Ausgabeleitungen.
- **Function Routing**. Mit dem Function Routing wird festgelegt, welche Signale auf die Eingabeleitungen $X1, X2$ und $X3$ einer Funktionseinheit geleitet werden. Die hier angelegten Signale dienen jedoch nicht nur als Eingabe, sondern sie legen zusammen mit den Belegungen für $Y2$ und $Y3$ (vgl. Function Unit)auch gleichzeitig fest, *welche* Funktion eine Zelle berechnet; siehe hierzu auch Tabelle 7. Abbildung 9 (mitte) zeigt die vom Function Routing betroffenen Multiplexer sowie Ein- und Ausgabeleitungen.
- **Function Unit**. Das Konfigurationswort der Function Unit legt abschließend fest, welche Funktion eine Zelle berechnet. Hierbei ist jedoch zu beachten, daß die zu berechnende Funktion aus der Belegung von $X1, X2$ und $X3$ sowie $Y2$, $Y3$ und CS erfolgt. Was eine einzelne Zelle berechnet hängt dadurch sowohl vom Function Routing als auch von der Konfiguration des Function Unit-Wortes ab. Einzig das Neighbour Routing ist von dem was eine Zelle berechnet abgekoppelt. Abbildung 9, rechts, stellt die vom Function Unit Konfigurationswort angesprochenen Teile einer Zelle dar. $Y1$ und $Y2$ sind Teil der Function Unit und somit nicht separat dargestellt, vgl. Abbildung 11.
- **State Access**. Jede Zelle enthält ein D-Type Register (Flip-Flop), das zur Speicherung beliebiger Daten benutzt werden kann. Das Speichern von (binären) Daten in ein D-Type Register erfolgt über das State Access Konfigurationswort. Näheres zu den Besonderheiten beim Speichern und Auslesen von Daten in und aus einer Zelle ist in Abschnitt A.2 nachzulesen.

Die Adressierung eines dieser Konfigurationsworte erfolgt über die **Column Offset**-Bits des Adresswortes; Tabelle 3 zeigt die entsprechenden Belegungen.

Tabelle 3 Die Belegungen für Column Offset[7:6]. In der dritten Spalte ist die entsprechende Konstante aus der Datei `XCAddr.h` aufgeführt.

Mode[15:14] Column[13:8] **Column Offset[7:6]** Row[5:0]

00	Neighbour Routing	`XCAddr::CELLNR`
01	Function Routing	`XCAddr::CELLFR`
10	Function Unit	`XCAddr::CELLFU`
11	State Access	`XCAddr::CELLAC`

Die verbleibenden beiden Teile der Adresse, also **Column** und **Row**, dienen im Falle der Cell Mode Konfiguration zur Spezifikation dessen, *welche* der 4096 Zellen des Zellarrays konfiguriert werden soll. Dabei werden die Column- und Row-Angaben als Punkte in einem **kartesischen Koordinatensystem** mit dem Ursprung $(0, 0)$ in der unteren linken Ecke des Zellarrays interpretiert, vgl. Abbildung 10.

Eine Adresse, die z. B. mit der Bitkombination `00.00001.01.000010` (vgl. Adressformat in Tabelle 1) berechnet wurde, greift auf das Function Routing-Konfigurationswort der Zelle an Position $(1, 2)$ zu.

Im folgenden werden die Belegungen der drei Konfigurationsworte Neighbour Routing, Function Routing und Function Unit vorgestellt. Beginnend mit dem **Neighbour Routing** zeigt Tabelle 4 welche Bitkombinationen an den jeweiligen Positionen des 8-Bit Wortes zu welcher Konfiguration des entsprechenden North-, East-, West- und South-Multiplexers innerhalb einer Zelle führt. Entsprechend Tabelle 4 würde das Konfigurationswort `00.01.00.11` zu folgender Einstellung der Routing Multiplexer führen: Sowohl der North- (oder auch N_{out}) als auch der West-Multiplexer liefern das Berechnungsergebnis `F` der

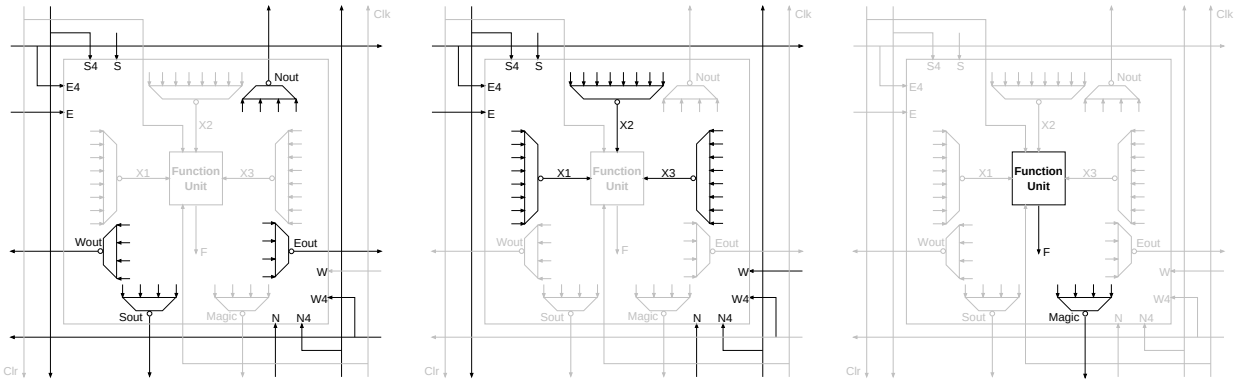


Abbildung 9 Die hervorgehobenen Bereiche zeigen jene Teile einer Zelle, die durch das Neighbour Routing- (links), Function Routing- (mitte) oder Function Unit- (rechts) Konfigurationswort angesprochen werden.

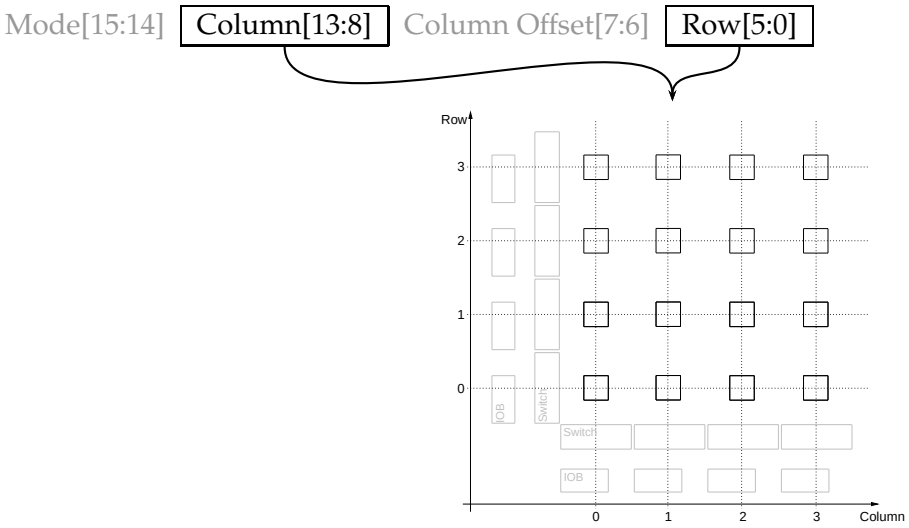


Abbildung 10 Mit den Column- und Row-Feldern wird im Cell Mode eine bestimmte Zelle als Punkt eines kartesischen Koordinatensystems mit Ursprung $(0,0)$ in der linken unteren Ecke des Zellarrays adressiert.

Zelle, während $E_{out} = N_{in}$ und $S_{out} = S_{in}$ gilt, d. h. der East-Ausgang der Zelle leitet das von Norden und der South-Ausgang das von Süden kommende Signal weiter.

Das **Function Routing**-Konfigurationswort setzt sich aus den in Tabelle 5 gezeigten Bits zusammen. Dabei wird mit dem CS-Bit ausgewählt, welches Signal die Funktionseinheit einer Zelle als Funktionsergebnis F liefert. Für $CS = 1$ ist dies das sog. kombinatorische (Combinatorial) Signal, also das Ergebnis einer Funktionsberechnung wie bspw. $A.B$ oder $A+B$. Falls $CS = 0$, so liefert die Funktionseinheit den Inhalt ihres D-Type Registers (Sequential) als Funktionsergebnis F.

Tabelle 4 Neighbour Routing.

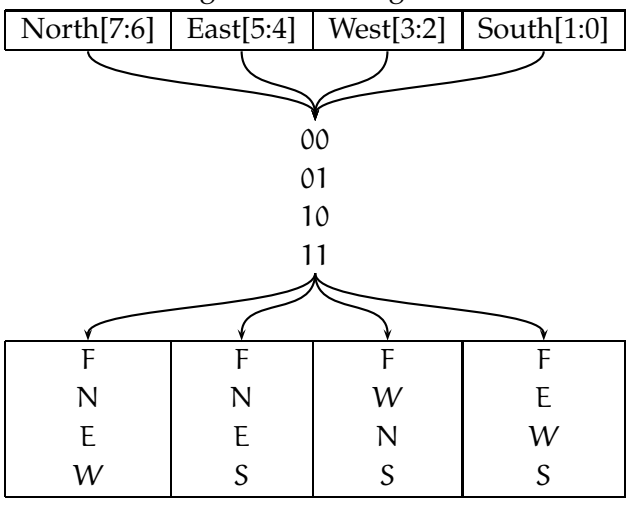
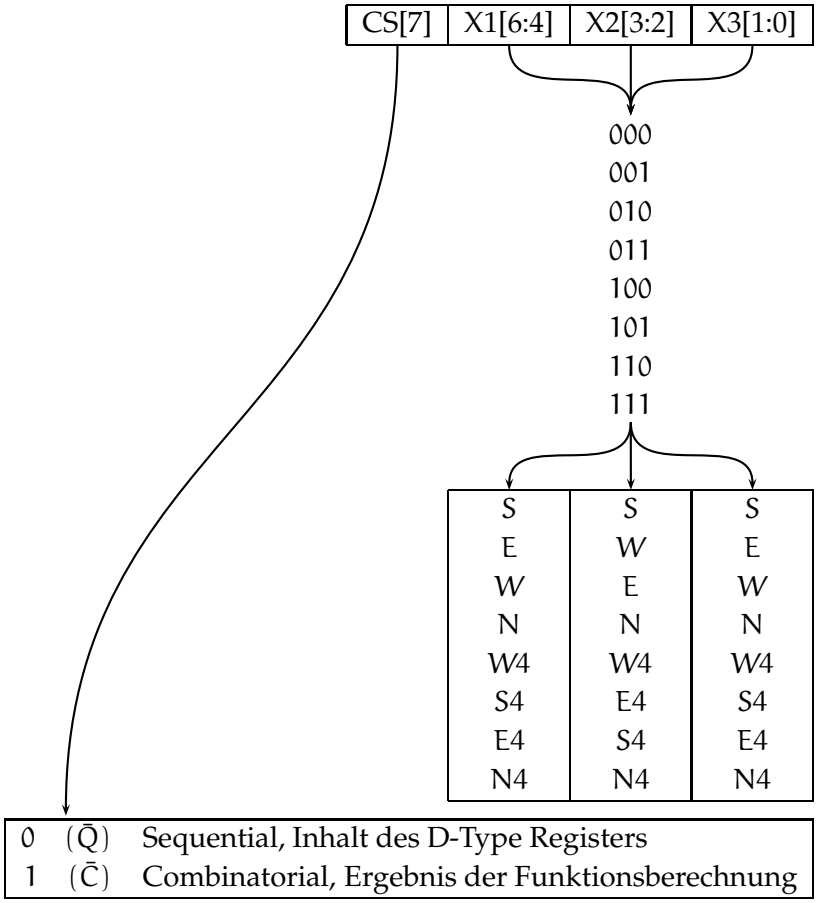


Tabelle 5 Function Routing.



Zur Veranschaulichung dessen, was bei der Änderung des CS-Bits innerhalb einer Zelle passiert, zeigt Abbildung 11 (vgl. Xilinx Inc. 1997, Abbildung 6 XC6200 Function Unit, Seite 6) wie die Berechnungseinheit einer Zelle aus verschiedenen Multiplexern und einem D-Type Register aufgebaut ist. Das CS-Bit des Neighbour Routing Wortes konfiguriert damit den CS Multiplexer am Ausgang der Zelle, der in Abhängigkeit seiner Konfiguration entweder den Inhalt Q des D-Type Registers oder das Ergebnis der Funktionsberechnung als Funktionsergebnis F weiterleitet. Es sei darauf hingewiesen, daß bei der *Weitergabe* von Signalen als Funktionsergebnis F, diese durch den CS Multiplexer invertiert werden. Diese Invertierung fällt beim *Auslesen* von F von einem Host-Prozessor aus nicht auf, da der sog. **Readback Circuit** das entsprechende Signal *vor* dem CS Multiplexer und somit vor der Invertierung abgreift (vgl. Abbildung 11, schwarze Punkte vor CS Multiplexer) und an den Host-Prozessor zurückliefert. Weitere Informationen zu internen Signalinvertierungen befinden sich in Anhang A.1.

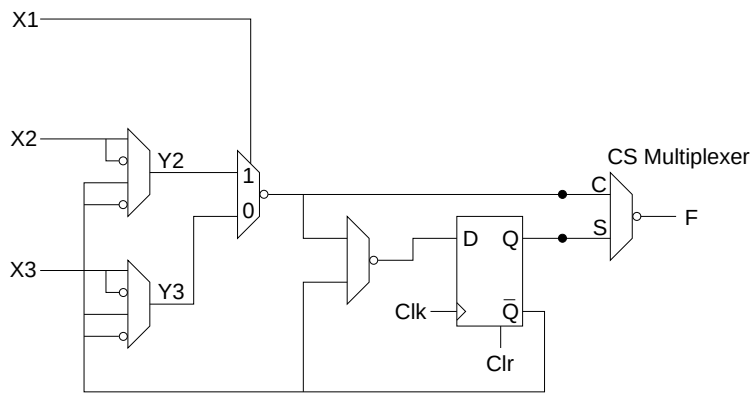


Abbildung 11 Der Aufbau einer Zell-Funktionseinheit des XC6200. Beim Lesen von F wird der an den Host-Prozessor gelieferte Wert an den durch die schwarzen Punkte markierten Stellen ausgelesen. Dadurch kommt es nicht zu einer Invertierung des entsprechenden Signals.

Die verbleibenden Bits des Function Routing Konfigurationswortes dienen zur Einstellung der Signale, die an die Eingabeleitungen X1, X2 und X3 der Berechnungseinheit einer Zelle geleitet werden. Tabelle 5 zeigt die möglichen Belegungen für $X1[6:4], X2[3:2]$ und $X3[1:0]$. Es fällt auf, daß zur Konfiguration jeder Eingabeleitung drei Bit und damit $2^3 = 8$ verschiedene Kombinationen zur Verfügung stehen, jedoch im Function Unit Konfigurationswort nur für die Eingabeleitung X1 drei Bits reserviert wurden. Aus Platzgründen wurde das jeweils höchstwertigste (dritte) Bit der Leitungen X2 und X3 im Function Unit Konfigurationswort, das im nächsten Paragraphen beschrieben wird, untergebracht. Somit ist Tabelle 5 im Zusammenhang mit der Beschreibung des Function Unit Wortes zu betrachten.

Das Wort `1.001.11.01` würde, vorausgesetzt die fehlenden Bits der Eingabeleitungen X2 und X3 sind im Function Unit Wort auf 0 gesetzt, zu folgender Konfiguration führen: An F wird, da $CS = 1(\bar{C})$, vom CS Multiplexer das Berechnungsergebnis der Funktionseinheit weitergeleitet. Weiterhin gilt, daß sowohl X1 als auch X3 das East-Eingabesignal erhalten $X1 = X3 = E_{in}$, während X2 mit dem Nord-Eingabesignal belegt wird $X2 = N_{in}$.

Fast Gates sind Gatter, die das D-Type Register einer Zelle verwenden, um eine schnellere Berechnung der entsprechenden Logik-Funktion durchführen zu können.

Das **Function Unit** Wort legt, zusammen mit den Belegungen der Eingabeleitung des Function Routing Wortes, fest, welche Funktion eine Zelle berechnet. Darüberhinaus wird mit dem RP-Bit bestimmt, ob das D-Type Register einer Zelle für sog. **Fast Gate**-Versionen von Zellfunktionen verwendet werden kann oder nicht. Das Function Unit Wort stellt weiterhin eine zusätzliche Routing Möglichkeit in Form der **Magic**-Leitung zur Verfügung und enthält die fehlenden höchstwertigen Bits der Eingabeleitungen X2 und X3 des Function Routing Konfigurationswortes.

Tabelle 6 Function Unit

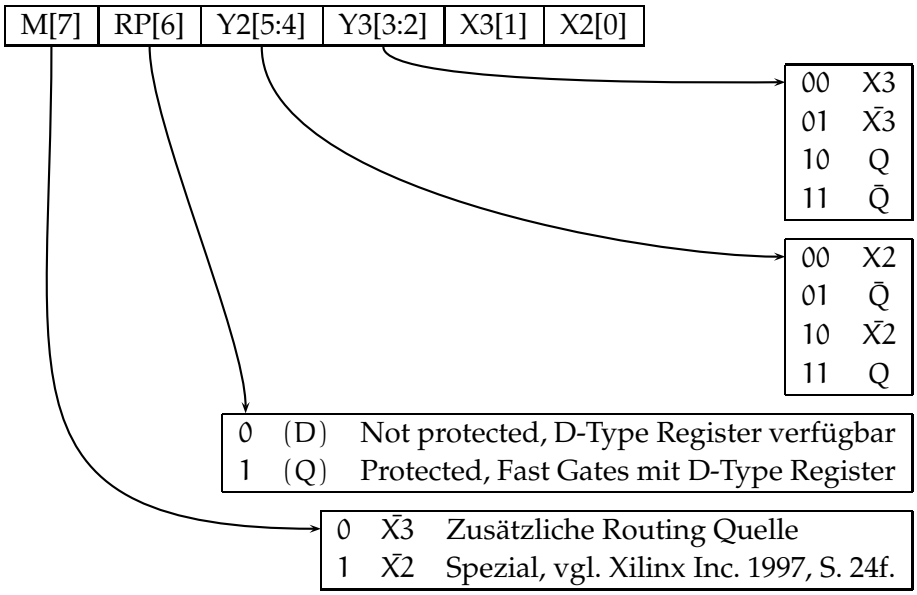


Tabelle 6 zeigt das Function Unit Wort mit seinen möglichen Belegungen. $Y2, Y3, \mathrm{RP}$ und CS sowie der Inhalt des D-Type Registers Q entscheiden in Abhängigkeit der Belegung von $X1, X2$ und X3, welche Funktion eine Zelle berechnen soll. Tabelle 7 listet alle möglichen Funktionen auf, die eine Zelle berechnen kann (nach Xilinx Inc. 1997, Tabelle 2 Function Derivation, Seite 8).

Unter der Annahme, daß für die Belegung der Eingabeleitungen $X1 = X3 = \mathrm{N_{in}}$ und $X2 = W_{\mathrm{in}}$ gilt, berechnet eine Zelle, deren Function Unit mit dem Wort `0.0.10.00.0.0` konfiguriert wurde die Funktion $\overline{\mathrm{N_{in}}} + W_{\mathrm{in}}$.

Das **State Access** Wort erlaubt schreibende Zugriffe auf das D-Type Register einer Zelle oder lesende Zugriffe auf das Funktionsergebnis einer Zelle. In diesem Fall werden die Werte, die auf den Datenbus gelangen sollen, anhand des **Map Registers** festgelegt. Zugriffe erfolgen stets auf Zellen einer Spalte, so daß der in Row gespeicherte Wert unberücksichtigt bleibt. Abschnitt A.2 gibt einen detaillierten Überblick wie das Schreiben und Lesen von Zellein- und ausgaben mit Hilfe des Map Registers funktioniert.

Tabelle 7 Funktionsableitung anhand der für X1, X2, X3, Y2, Y3, RP, CS sowie Q gesetzten Werte. A und B sind beliebige Eingangssignale, bspw. $A = S_{in}$ und $B = W_{in}$. C steht für Combinatorial und bedeutet, daß die entsprechende Zelle eine kombinatorische Funktion berechnet.

Function	X1	X2	X3	Y2	Y3	RP	CS	Q
0	A	A	A	X2	$\overline{X3}$	X	C	X
1	A	A	A	$\overline{X2}$	X3	X	C	X
BUF (Fast)	A	X	X	Q	$\overline{Q}$	Q	C	0
BUF	X	A	A	$\overline{X2}$	$\overline{X3}$	X	C	X
INV (Fast)	A	X	X	$\overline{Q}$	Q	Q	C	0
INV	X	A	A	X2	X3	X	C	X
A.B (Fast)	A	B	X	$\overline{X2}$	$\overline{Q}$	Q	C	0
A.B	A	B	A	$\overline{X2}$	$\overline{X3}$	X	C	X
$\overline{A}$.B (Fast)	A	X	B	$\overline{Q}$	$\overline{X3}$	Q	C	0
$\overline{A}$.B	A	A	B	X2	$\overline{X3}$	X	C	X
$\overline{A.B}$ (Fast)	A	B	X	X2	Q	Q	C	0
$\overline{A.B}$	A	B	A	X2	X3	X	C	X
A+B (Fast)	A	X	B	Q	$\overline{X3}$	Q	C	0
A+B	A	A	B	$\overline{X2}$	$\overline{X3}$	X	C	X
$\overline{A}$+B (Fast)	A	B	X	$\overline{X2}$	Q	Q	C	0
$\overline{A}$+B	A	B	A	$\overline{X2}$	X3	X	C	X
$\overline{A+B}$ (Fast)	A	X	B	$\overline{Q}$	X3	Q	C	0
$\overline{A+B}$	A	A	B	X2	X3	X	C	X
A⊕B	A	B	B	X2	$\overline{X3}$	X	C	X
$\overline{A⊕B}$	A	B	B	$\overline{X2}$	X3	X	C	X
M2_1	SEL	A	B	$\overline{X2}$	$\overline{X3}$	X	C	X
M2_1B1A	SEL	A	B	X2	$\overline{X3}$	X	C	X
M2_1B1B	SEL	A	B	$\overline{X2}$	X3	X	C	X
M2_1B2	SEL	A	B	X2	X3	X	C	X

3.3.2 Switches und IOBs – 01/10

Multiplexer in Form sog. **Switches** befinden sich jeweils an 4×4 Grenzen des kompletten Zellarrays und dienen dem Routing von Clock- und Clear-Signalen in die einzelnen Zellen, als zusätzliche Routing Quellen für die sog. Fast Lanes sowie als Mittler zwischen den **Input/Output Blocks** (IOB) an den Grenzen des Zellarrays. In Abbildung 37 auf Seite 107, sind sowohl die Switches als auch die IOBs dargestellt. Die spezielle Funktion der IOBs liegt in der Weiterleitung von Signalen in den Zellarray, die an einzelne Pins des FPGA-Chips angelegt werden können. Verschiedene Pins leiten ihre Eingaben an die sog. **Pads** weiter, wobei die Pads ihrerseits Signale an einen IOB weitergeben. Nicht zu jedem IOB existiert ein Pad, so daß es sog. „padless IOBs“ gibt, die ihre Signale von und zu benachbarten IOBs mit Pads

Thompson (1996) verwandte diese Methode, um das Signal eines Funktionsgenerators in evolvierte Schaltkreise einzuleiten.

her- bzw. weiterleiten. Die Konfiguration der IOBs und Pads zur Einspeisung externer Signale in einen Schaltkreis ist kompliziert, und wird in Xilinx Inc. (1997), Seite 11ff. beschrieben.

In dieser Arbeit war jedoch nur die Konfiguration der verschiedenen Switches nötig, um die als Eingaberegister dienenden Zellen mit dem Takt-Signal der Global Clock zu versorgen. An dieser Stelle soll auf die Konfiguration der Switches nicht weiter eingegangen werden, da Abschnitt A.2.2 alle im Rahmen dieser Arbeit benötigten Informationen in Verbindung mit der Ein-/Ausgabe von Signalen bereitstellt.

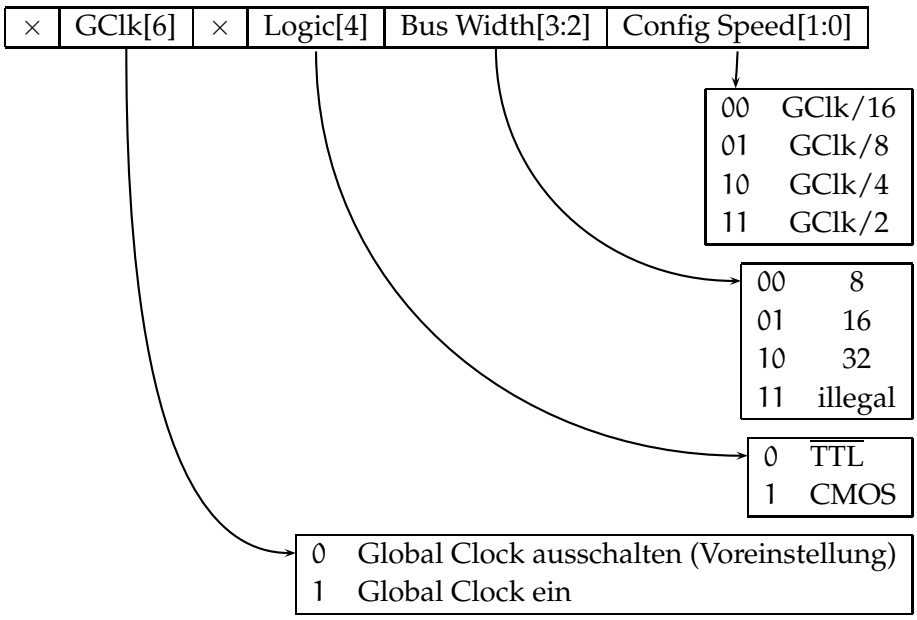
3.3.3 Control Register – 11

Im Adressmodus 11 kann auf die verschiedenen Kontroll-Register des XC6200 zugegriffen werden. Neben dem Mask Register, welches hier nicht diskutiert werden soll, da es in dieser Arbeit keine Verwendung fand (nähere Informationen in Xilinx Inc. 1997, Seite 19f.), existieren noch das Map-, das Device Configuration- sowie das Device Identification-Register. Das Map Register ist beim Lesen und Schreiben von Daten in bzw. aus dem Zellarray wichtig und wird ausführlich im Abschnitt A.2 behandelt.

Das **Device Configuration Register** (DCR) codiert globale Funktionen bzw. Modi des XC6200. Es ist, ebenso wie bspw. die Neighbour Routing oder Function Routing Konfigurationswörter, ein 8 Bit Wort von denen jedoch zwei Bits keine Bedeutung zukommt. Mit den verbleibenden Bits werden die folgenden Funktionen eingestellt: die Baud Rate der seriellen Programmierschnittstelle, die Breite des Daten-Busses in Bits, der Input Logic Threshold Level aller Ein- und Ausgaben sowie ein Flag mit dem die Global Clock (GClk) des Zellarrays gestartet oder angehalten werden kann.

Tabelle 8 zeigt das DCR mit den möglichen Belegungen und deren Bedeutung. Das Global Clock-Bit kann jederzeit verändert werden, da der XC6200 so konstruiert ist, daß die Uhr stets auf eine „safe, glitch free manner“ (Xilinx Inc. 1997, Seite 26) gestartet bzw. gestoppt wird. An der Baud Rate mußten im Rahmen dieser Arbeit keine Änderungen vorgenommen werden, da die Programmierung des XC6200 nicht über das serielle sondern über das parallele FastMap CPU Interface erfolgte. Dasselbe gilt für den Input Logic Threshold, der als Standardeinstellung stets auf $\overline{\text{TTL}}$ gesetzt blieb. Ein wichtiger Teil des DCR stellt die Einstellung der Breite des externen Datenbusses dar; Beispiel-Code 4 und Beispiel-Code 5 zeigen wie die Busbreite mit der Methode `setBusWidth()` gesetzt werden kann. Das Starten/Stoppen der globalen Uhr kann notwendig werden, wenn während des Betriebes eines Schaltkreises bestimmte Teile rekonfiguriert und inkorrekte Übergangszustände vermieden werden sollen.

Tabelle 8 Device Configuration Register



Das **Device Identification Register** (DIR) kann im Prinzip als eine Art Ein-/Ausschalter für den XC6216 betrachtet werden. Wurde ein Schaltkreis in der RPU instantiiert, so werden die Ausgabeeinheiten des FPGA erst aktiviert, wenn in den 16-Byte Speicher des DIR die korrekte Geräteidentifikation (Device ID) eingespeichert wurde. Dies stellt eine Art Test für das „Programming Interface“ dar, um sicherzustellen, daß keine möglicherweise schädlichen Ausgabeeinheiten aktiviert werden. Jedes Byte des DIR muß dabei mit einem bestimmten ASCII Code beschrieben werden, mit Ausnahme des letzten Bytes, welches eine dem FPGA entsprechende Gerätenummer zugewiesen bekommen muß (vgl. Xilinx Inc. 1997, Tabelle 26, Control Register Memory Map For XC6209 and XC6216). Beispiel-Code 1 zeigt eine Funktion, die im Rahmen dieser Arbeit verwendet wurde, um die für den XC6216 korrekte Byte-Folge in das DIR einzuschreiben bzw. aus ihm zu löschen.

In der XC6200-Familie wurden vier FPGAs mit unterschiedlicher Größe der Zellarrays produziert: XC6209, XC6216, XC6236 und XC6264. Jedes FPGA trägt eine eindeutige Gerätenummer.

Mit Hilfe des DIR können alle Ausgabeeinheiten des FPGA gleichzeitig deaktiviert werden, indem nur ein einziges der 16 Bytes auf einen unzulässigen Wert gesetzt wird. Nach Korrektur des entsprechende Bytes sind alle Ausgabeeinheiten wieder aktiviert.

3.3.4 Adressberechnung

Die Berechnung der Adresse, die sich aus den in den vorigen Kapitel vorgestellten Komponenten Adress-Modus, Column, Column Offset sowie Row zusammensetzt, ist ein wenig mühselig, so daß sich die Definition einer eigenen Funktion für diesen Zweck anbietet. In Beispiel-Code 2 sind zwei Varianten einer Funktion, die genau diesen Zweck erfüllt, abgebildet.

Beispiel-Code 1

```
// Write a valid device id to the DIR and let the circuit run.
void writeValidDeviceID( )
{
  word adr, i;
  unsigned char XC6000_ID[] = "Xilinx XC6000  ";

  for ( adr= IDREG0, i= 0; i<15; ++i, ++adr )
    fpga.write6200( adr, XC6000_ID[ i ] );
  fpga.write6200( 0xc03f, 0x1 ); // set device id of xc6216 (=1)
}

// Make sure chip is disabled by writing an invalid device id to the DIR.
void writeInvalidDeviceID( )
{
  word adr, i;

  for ( adr= IDREG0, i= 0; i < IDREGSIZE; ++i, ++adr )
    fpga.write6200( adr, 0x0 );
}
```

Beispiel-Code 2

```
// Returns the FPGA address composed of the fields given in the parameters.
// addr1 is hand-crafted, addr2 uses methods from Xilinx' library.
unsigned addr1( unsigned mode,   unsigned col,
                unsigned colOff, unsigned row  )
{
  unsigned address= 0;
  address|= ( mode   << 14 );
  address|= ( col    << 8  );
  address|= ( colOff << 6  );
  address|= row;
  return( address );
}

unsigned addr2( unsigned mode, unsigned col,
                unsigned colOff, unsigned row )
{
  word address= 0;
  XCAddr::setMode  ( address, mode   );
  XCAddr::setCol   ( address, col    );
  XCAddr::setColOff( address, colOff );
  XCAddr::setRow   ( address, row    );
  return( address );
}
```

4 Versuchsaufbau und Methode

The cosmos evolves.
– Michael Ruse

The darkness dissolves.
– Yello

Zur Durchführung der Experimente und Analysen auf der Basis eines Evolvable Hardware Systems war ein bestimmter Versuchsaufbau erforderlich, der in Abschnitt 4.1.2 vorgestellt wird.

Es zeigte sich, daß ein wichtiger Aspekt des vorgestellten EHW-Systems, gerade im Hinblick auf die in Kapitel 5 vorgestellten Experimente und Ergebnisse, eine angemessene Genotyp/Phänotyp-Abbildung ist, insbesondere im Hinblick auf die **Repräsentation des Genotyps**. Abschnitt 4.2 wird sich aus diesem Grund intensiv mit dem Begriff der **Kodierung** auseinandersetzen und dessen Einordnung in den Kontext dieser Arbeit klären.

4.1 Versuchsaufbau

Zur Motivation des gewählten Versuchaufbaus wird die von Tomassini et al. (1997) vorgeschlagene und in Abschnitt 2.3 bereits diskutierte Einteilung des evolutionären Schaltkreisentwurfes in vier Klassen herangezogen. Diese Klassen unterscheiden sich im wesentlichen durch die Art und Weise in der (re)konfigurierbare Mikrochips (RPU) zur Anwendung kommen. Handelt es sich um eine Simulation der Schaltkreissynthese auf einem Host-Rechner, ohne das ein Mikrochip beteiligt ist, so wird von der Klasse der Offline Evolution gesprochen. In der Klasse der Semi Online Evolution werden RPUs unterstützend eingesetzt, während die Klassen Online sowie True Online Evolution ausschließlich eine RPU ohne jede Beeinflußung durch einen Host-Rechner verwenden.

Wie im Überblick des Abschnitts 2.4 bereits erwähnt gibt es Bemühungen, bspw. von Kajitani et al. (1998, 1999), Evolvable Hardware in der Online Evolution Klasse zu implementieren, d. h. daß sowohl

die Anwendung der genetischen Operatoren als auch die Speicherung einer kompletten Population in der Hardware stattfindet. Jedoch bereitet die Realisierung dieses Vorhabens, auch wenn bereits erhebliche Fortschritt zu verzeichnen sind, aus verschiedenen zumeist technischen Gründen noch immer Probleme. Die für diese Arbeit zur Verfügung stehenden Mittel erlaubten den Aufbau eines EHW-Systems, daß von der Definition her in der **Semi Online Evolution** Klasse einzuordnen ist und im wesentlichen aus einem FPGA sowie einer Softwarerealisierung Evolutionärer Algorithmen besteht. Dieses System soll hier vorgestellt werden.

4.1.1 Verwendete Hard- und Software

Die verwendete RPU, ein XILINX XC6216 FPGA, wurde auf einem PCI-Board der Firma Annapolis Micro System, Inc. geliefert und war bereits in Abschnitt 2.2.4 Gegenstand näherer Betrachtung. In einem PC mit AMD K6/300MHz Prozessor und Windows NT 4.0 als Betriebssystem konnte dieses Board erfolgreich installiert werden. Die von XILINX mitgelieferte Software enthält, neben Tools zur Synthese von Schaltkreisen (Velab) und dem Autorouten/-placen (XACT6000) von Schaltkreisen, eine C++ Software-Bibliothek, die den direkten Zugriff auf den XC6216 erleichtern soll, sowie Hardware-Treiber für Windows95/NT. Sowohl die Treiber für Windows 95, als auch für NT liefen zufriedenstellend, wobei die Arbeitsgeschwindigkeit unter Windows 95 höher als unter NT zu sein schien, vgl. hierzu auch Abschnitt 3.2. Alle Experimente fanden jedoch zugunsten größerer Stabilität unter Windows NT statt, als Compiler kam Visual C++ 5.0 von Microsoft zum Einsatz.

4.1.2 Das Evolvable Hardware System

Im Rahmen dieser Arbeit war der Aufbau eines EHW-Systems notwendig, das den evolutionären Entwurf von Schaltkreisen erlaubt, die ein vorgegebenes Ein-/Ausgabeverhalten aufweisen. Ein Schaltkreis soll für eine Testmustervektor M mit x Mustern $m_i \in M, i = 1, \ldots, x$ die entsprechende Ausgabe o_i aus einer zu M gehörigen Ausgabemenge O produzieren. Um solche Schaltkreise zu finden wird im Rahmen eines Genetischen Algorithmus (vgl. Abschnitt 2.1.3) eine Population P von N schaltkreiskodierenden Individuen $p_i, i \in \{1, \ldots, N\}$ verwendet. Ein Individuum repräsentiert dabei eine mögliche Zusammenstellung der Konfigurationsbits einer bestimmten Anzahl von Zellen des FPGA-Zellarrays – jedes Individuum der Population ist somit ein Repräsentant eines physikalischen Schaltkreises. Die Konfigurationsbits des FPGA werden Bit für Bit direkt auf den Bitstring eines Individuums abgebildet, so daß es sich hier um eine **direkte Kodierung** des Genotyps eines Individuums handelt. Weil der Repräsentation von Genotypen als auch der Genotyp/Phänotyp Abbildung in dieser Arbeit eine

zentrale Rolle zukommt, wird Abschnitt 4.2 ausführlich dieses Thema diskutieren. Die gewählten Kodierungen werden dann in Kapitel 5 im Zusammenhang mit den durchgeführten Experimenten und Ergebnissen vorgestellt.

Ein **evolutionärer Lauf**, dessen schematischer Aufbau in Abbildung 12 gezeigt ist, beginnt mit der **Initialisierung der Startpopulation** $P(t)$ zum Zeitpunkt $t = 0$. Diese Initialisierung kann entweder zufällig, d. h. in der Startpopulation existieren N zufällige Bitstrings, oder aber gezielt erfolgen, indem die Startpopulation bspw. nur Individuen enthält, die ausschließlich nicht-konstante Ausgaben als Schaltkreisberechnung liefern. Letztgenannte Möglichkeit kann insbesondere dann von Nutzen sein, wenn ein programmierbarer Schaltkreis (PLD) benutzt wird der auch Konfigurationen zuläßt, die dem PLD möglicherweise schaden können. So lassen bestimmte PLDs, ein entsprechend fehlerhaftes Design vorausgesetzt, das Zusammenschalten von Ein- und Ausgabeleitungen zu, so daß es zu Kurzschlüssen und damit zur Zerstörung des PLDs kommen kann. Bei dem in dieser Arbeit verwendeten XC6216 FPGA von XILINX gibt es jedoch keine solchen „gefährlichen“ Konfigurationen, so daß in den meisten Fällen eine zufällige Initialisierung der Startpopulation erfolgte.

Nach der Initialisierung erfolgt die **Fitneßbewertung** aller Individuen aus $P(t)$, indem jeweils ein Individuum p auf dem FPGA instantiiert wird und dann nacheinander alle Testmuster $m_i \in M$ als Eingaben an den Schaltkreis angelegt werden. Zu jedem Testmuster m_i soll die zugehörige Ausgabe o_i aus der Ausgabemenge als Ergebnis der Schaltkreisberechnung e_i ausgegeben werden. Es wird

$$\Delta = \sum_{i=1}^{n} |e_i - o_i|$$

berechnet und dem Individuum, welches diesen Schaltkreis codiert, eine entsprechende Fitneß zugewiesen, die umso höher (=besser) sein wird, je kleiner Δ ist. Es gilt also einen Schaltkreis zu finden, der Δ minimiert bzw. für den mit einem bestimmten, zuvor festzulegenden Schwellenwert S die Gleichung $\Delta \leq S$ erfüllt ist.

Im Hinblick auf Abbildung 1 folgen nach der initialen Fitneßbewertung die Schritte Rekombination, Mutation sowie Selektion; dieser Weg ist in Abbildung 12 durch die gestrichelten Pfeile dargestellt. Im weiteren Verlauf des Algorithmus wird die mit den durchgezogenen Pfeilen markierte Schleife durchlaufen, bis ein Individuum gefunden wird, welches $\Delta \leq S$ erfüllt.

Bei der **Rekombination** werden jeweils zwei zufällig ermittelte Individuen aus $P(t)$ durch Anwendung des Crossover-Operators rekombiniert. Das Crossover ermöglicht an zufällig ermittelten Bruchstellen das Austauschen von Abschnitten gleicher Länge zwischen Individuen. In Abbildung 12 (unten) ist das 2-Punkt Crossover dargestellt, gene-

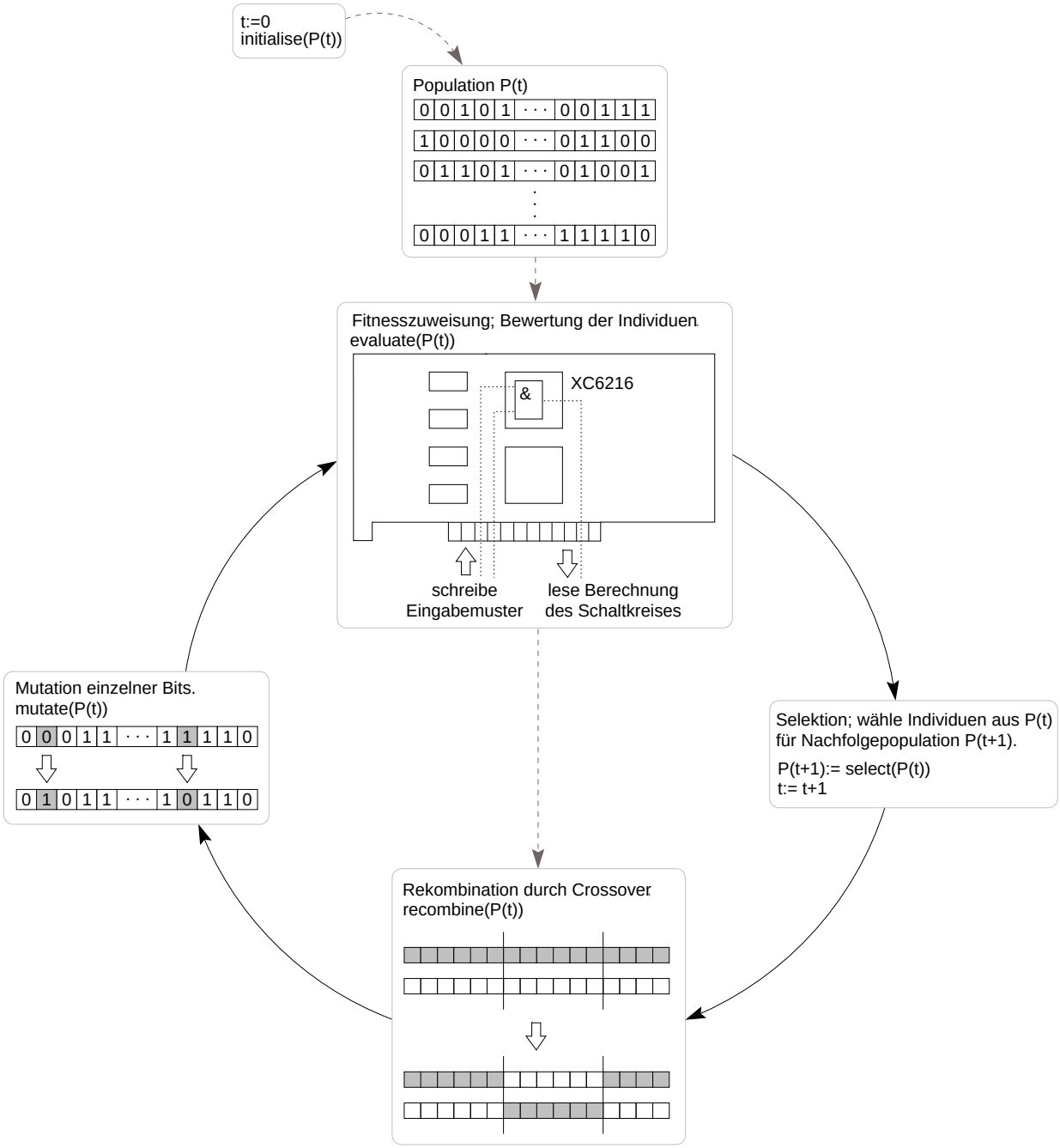


Abbildung 12 Visualisierung des evolutionären Prozesses eines EHW-Systems in der Semi Online Evolution Klasse. Der gestrichelte Pfad zeigt, in Analogie zu der in Abbildung 1 (links) dargestellten Schleife, die Eintrittsphase des Genetischen Algorithmus in den evolutionären Kreislauf. Der durch die durchgezogenen Pfeile markierte Pfad stellt die Hauptphasen des GA dar, die bis zur Erfüllung des Abbruchkriteriums in der angezeigten Richtung durchlaufen werden.

rell existieren beim z-Punkt Crossover z Bruchstellen. Die Wahrscheinlichkeit mit der zwei Individuen rekombiniert werden, wird mit der Crossover-Rate p_c angegeben, siehe auch Algorithmus auf Seite 4.2.4. Typische Werte für p_c liegen im Bereich von $[0.6, 1.0]$, vgl. Bäck (1996). Als Crossover wurden in dieser Arbeit zunächst das 2-Punkt und 8-Punkt Crossover, später auch das Uniform Crossover implementiert. Details zur Implementierung befinden sich in Kapitel 5.

Empirischen Studien zufolge, stellt $z = 8$ nicht nur den optimalen Wert für das z-Punkt Crossover in Evolutionären Algorithmen dar, sondern es ist auch die obere Schranke für die maximale Anzahl von Crossover-Punkten in der Natur. Trotzdem hat sich das 2-Punkt Crossover als Implementierungsstandard etabliert, „[...] probably due to the wide dissemination of Grefenstette's implementation." Aus Bäck (1996), Seite 115.

Im Anschluß an die Rekombination folgt die **Mutation**, die eine zufällige und ungerichtete Invertierung von Bits auf den Bitstrings der Individuen bewirkt. Die Mutationsrate p_m gibt die Wahrscheinlichkeit an, mit der ein einzelnes Bit eines Bitstrings invertiert wird. Mit $p_i(k)$ sei die k-te Bit-Position eines Individuums $p_i \in P(t)$ bezeichnet. Für alle Individuen $p_{i=1,\ldots,N} \in P(t)$ wird für jede Bit-Position k ein $\chi_k \in [0, 1]$ gleichwahrscheinlich gezogen. Falls $\chi_k > p_m$ gilt, bleibt das Bit an Position $p_i(k)$ unverändert. Für den Fall, daß $\chi_k \leq p_m$ ist, erfolgt die Invertierung von $p_i(k)$ durch $p_i(k) = 1 - p_i(k)$. Die meisten evolutionären Läufe in dieser Arbeit wurden mit einer konstanten Mutationsrate von $p_m = 1/l$ durchgeführt, wobei l die Länge der betrachteten Individuen ist; durch diese Wahl von p_m wird es im Mittel eine Bit-Mutation pro Individuum geben. Abweichungen von $1/l$ werden an entsprechender Stelle erwähnt. Abbildung 12 (links) zeigt ein Individuum bei dem es nach Anwendung des Mutations-Operators zu Invertierungen an zwei Stellen kam.

Abschließend werden durch einen **Selektion**sprozess Individuen für die nachfolgende Population ausgewählt, vgl. Abbildung 12 (rechts). Verschiedene Vorversuche haben gezeigt, daß mit der EP q-Tournament-Selektion nach Fogel (1995) die besten Resultate im Hinblick auf die Konvergenz des hier implementierten GA erzielt wurden, vgl. Abschnitt 5.1.6.

4.2 Repräsentation

Im vorhergehenden Abschnitt wurde bereits kurz auf den Begriff der Kodierung eingegangen. Dabei wurde erwähnt, daß die Individuen einer Population Repräsentanten physikalischer Schaltkreise sind. Jedes Individuum stellt einen direkt kodierten Bitstring der Konfigurationsbits genau jener Zellen dar, die einen kompletten Schaltkreis beschreiben. Die in jedes Individuum kodierten Informationen in Form der Konfigurationsbits stellen somit den genetischen Bauplan (Genotyp) eines Schaltkreises und seines Verhaltens (Phänotyp) dar und es stellt sich die Frage, wie der Genotyp idealerweise repräsentiert wird, damit im Rahmen eines Genetischen Algorithmus schnell die gewünschten Lösungen gefunden werden. Zur weiteren Erläuterung dieses Repräsentations-Problem wird es nötig sein, zunächst einen Blick auf das biologische Modell der (De)Kodierung von Informationen zu werfen.

4.2.1 Das biologische Modell der Kodierung

Der Bauplan aller Lebewesen ist in der Desoxyribonukleinsäure (DNS) kodiert, die eine lineare Kette von Nukleotiden darstellt. Die am Aufbau der DNS beteiligten Nukleotide sind aus Zucker, Phosphat und vier verschiedenen Basen zusammengesetzt; diese Basen stellen das Alphabet der genetischen Information dar. Die DNS ist eine Doppelhelix bestehend aus zwei schraubenförmig ineinander verwundenen Ketten solcher Basen. Jeweils drei aufeinanderfolgende Basen bilden ein Codon. Eine Sequenz von 100 bis 1000 Codons bildet ein Gen. Der **Genotyp** eines Organismus wird von Richard C. Lewontin als „[. . .] the class of which it is member based upon the postulated state of its internal hereditary factors, the genes“ (Keller und Lloyd 1992, Seite 146) bezeichnet. Einfacher gesagt stellt der Genotyp eines Individuums (Organismus) die Gesamtheit seiner Erbanlagen in Form der Gene dar. Durch die sog. Genexpression ist der Phänotyp, also das Erscheinungsbild des Individuums in der Gesamtheit seiner Merkmale, definiert. Lewontin beschreibt den **Phänotyp** eines Organismus als „[. . .] the class of which it is a member based upon the observable physical qualities of the organism, including its morphology, physiology, and behaviour at all levels of description“ (Keller und Lloyd 1992, Seite 146). Die Erzeugung eines Organismus durch seine genetische Blaupause, und damit der Genotyp/Phänotyp Abbildung, ist ein sehr komplizierter Prozess, der bis heute nicht vollständig verstanden ist. Im folgenden wird eine stark vereinfachte Variante dargestellt, ausführlichere Informationen zu diesem Thema finden sich bspw. in Hirsch-Kaufmann und Schweiger (1992).

Diese Basen heißen Adenin, Guanin, Cytosin und Thymin. Sie werden als Vertreter eines Alphabetes mit ihren Anfangsbuchstaben abgekürzt.

Die Menge der geerbten Gene, sowohl im Nukleus als auch in verschiedenen zytoplasmatischen Partikeln wie Mitochondrien und Chloroplasten, bilden das Genom eines Individuums.

Während das Genom eines Individuums den Träger der Erbinformation darstellt, sind die Träger der Phänotypen die Proteine. Proteine sind mehrfach gefaltete Makromoleküle, die im wesentlichen aus einer langen Kette von Aminosäuren bestehen. Die Sequenz einer Aminosäurekette wird dabei wie folgt kodiert: Drei aufeinanderfolgende Basenpaare (die o. g. Codons) kodieren genau eine von 20 verschiedenen Aminosäuren. Damit ist der genetische Code redundant, da $4^3 = 64$ verschiedene Symbole für die Kodierung der Aminosäuren zur Verfügung stehen. Als Folge werden mehrere verschiedene Codons auf dieselbe Aminosäure abgebildet. Die Redundanz im genetischen Code bedeutet Robustheit gegenüber Basenmutationen oder Ablesefehlern: Wird durch eine Mutation eines der drei Nukleotide eines Codons verändert, so muß dies nicht zwangsläufig die Übersetzung in eine andere als die ursprünglich vorgesehene Aminosäure zur Folge haben. Die Proteinbiosynthese beginnt mit der Transkription eines Gens in die m-RNS (messenger-Ribonukleinsäure). Die m-RNS dient dem Transport der genetischen Information zu den Ribosomen, wo sie für die Synthese des entsprechenden Proteins verwendet wird. Die Ri-

bosomen übernehmen in der Translationsphase die Übersetzung der Basen-Triplets (Codons) in Aminosäuren und die Zusammensetzung der abgelesenen Aminosäuren zum Protein.

Der Einfluß eines oder mehrerer Gene auf die phänotypischen Eigenschaften eines Organismus kann nicht durch eine einfache Eins-zu-Eins Beziehung beschrieben werden. Verschiedene Gene können auf ein einziges phänotypisches Merkmal Einfluß nehmen, während auch der umgekehrte Fall möglich ist, indem sich ein Gen auf mehrere phänotypische Eigenschaften auswirkt. Darüberhinaus existieren, neben Unterbrechersequenzen zwischen den Genen, auch nichtkodierende Sequenzen in den Genen der DNS, sog. Introns, die wahrscheinlich überhaupt keine genetischen Informationen für den Phäntotyp transportieren. Somit herrscht ein Informationsfluß ausschließlich vom Genotyp zum Phänotyp über den Weg DNS $\rightarrow$ RNS $\rightarrow$ Protein.

Genau genommen, ergibt das bloße Aneinanderreihen von Aminosäuren zu einer Sequenz noch kein Protein. Der Übergang von einer Aminosäuresequenz zu einem Protein, auch 2. Genetischer Code genannt, erfolgt durch einen Faltungsprozeß der zum sog. Native State des Proteins führt und noch immer in weiten Teilen unverstanden ist.

„Yet other fractions of DNA such as introns and intervening sequences have yet looser and more poorly understood relations to the biosynthesis of proteins.“ (Keller und Lloyd 1992, Seite 134).

Während der Reproduktionsphase wird der Genotyp durch Mutation und Rekombination (Crossover) verändert. Jedes Individuum existiert in einer Umgebung mit begrenzten Ressourcen und unterliegt einem Selektionsdruck: Je angepaßter ein Individuum an seine Umwelt ist, desto größer sind sowohl seine Chancen zu überleben als auch in dieser Zeit Nachkommen zu zeugen, an die es Teile seines Genotyps weitergibt. Durch diese nicht-deterministische Reproduktion kommt es zu permanenter Variation des genetischen Materials und damit zu stets unterschiedlichen Nachkommen.

Anhand dieser äußert kurzen Einführung in die Abbildung des biologischen Genotyps in einen Phänotyp wird klar, daß das einfache Kopieren dieses Prinzips zum Zwecke der algorithmischen Nutzbarmachung nur schwer realisierbar scheint. Es gilt daher Vereinfachungen zu finden und nur jene Teile, bspw. im Rahmen eines Evolutionären Algorithmus, aus dem biologischen Vorbild zu übernehmen, die für die gegebene Aufgabe angemessen scheinen. Der folgende Abschnitt wird sich daher mit der Repräsentation in Genetischen Algorithmen befassen.

4.2.2 Repräsentation in Genetischen Algorithmen

In Evolutionären Algorithmen wird allgemein zwischen dem Genotyp-Raum $\mathcal{G}$ als Repräsentant des Suchraums und dem Phänotyp-Raum $\mathcal{P}$ als Abbildraum unterschieden (die Notation folgt Droste und Wiesmann (1998) sowie Bäck (1996)). Somit steht die Abbildung $h' : \mathcal{G} \rightarrow \mathcal{P}$ für eine Abstraktion des Prozesses der Proteinbiosynthese und der Entwicklung des Phänotyps, mithin stellt sie also die Genotyp/Phänotyp Abbildung eines einzelnen Indvidiuums dar. Während sowohl Rekombination als auch Mutation auf dem Genotyp-Raum arbeiten, findet die Bewertung der Fitneß eines Individuums anhand einer Fitneßfunktion $f : \mathcal{P} \rightarrow \mathcal{W}$ mit bpsw. $p = \mathbb{N}$ auf dem Abbildraum statt. Es kann

gezeigt werden, daß zwischen $\mathcal{G}$, $\mathcal{P}$ und h' sowie der Mutation, Rekombination und der Selektion starke Verbindungen und Abhängigkeiten herrschen, die zu der Frage führen wie $\mathcal{G}, h'$, Rekombinations- und Mutations-Operator gewählt werden sollten, wenn ein bestimmter Phänotypraum $\mathcal{P}$ und eine Fitneßfunktion f gegeben sind. Nach Droste und Wiesmann (1998) besteht eine Möglichkeit darin einen Standard Genotyp-Raum, wie bspw. $\mathcal{G} = \{0,1\}^l = \mathbb{B}^l$ für Genetische Algorithmen, zu wählen und dann eine passende Genotyp/Phänotyp Abbildung $h' : \mathcal{G} \to \mathcal{P}$ zu entwerfen. Jedoch ist die Wahl einer passenden Abbildung h' nicht unproblematisch und es kann zu Schwierigkeiten bei der Zuverlässigkeit und Konvergenz des Evolutionären Algorithmus kommen (Bäck 1996). Eine mögliche Erklärung für dieses Verhalten ist die Reduktion der Kausalität zwischen Genotyp und Fitneßfunktion durch die verwendete Abbildungs- oder Kodierungsfunktion h'.

Aufgrund der Bedeutung der Kausalität für den Suchprozeß in Evolutionären Algorithmen insbesondere im Zusammenhang mit dem Mutations-Operator, wird sich der folgende Abschnitt eingehender mit diesem Thema befassen.

4.2.3 Kausalität

Sendhoff et al. (1997) haben die Forderung nach lokaler starker Kausalität des Suchprozesses im Hinblick auf den Mutations-Operator präzisiert und formale Maße angegeben. Diese besagen, daß kleine Änderungen im Genotyp-Raum als Folge von Mutationen zu kleinen Änderungen im Phänotyp-Raum führen sollen. Dadurch bleibt die Nachbarschafts-Struktur unter der Abbildung $h' : \mathcal{G} \to \mathcal{P}$ erhalten. Abbildung 13 zeigt Beispiele für stark-, schwach- und nicht-kausale Abbildungen unter dem Einfluß von Mutationen. Auf der linken Seite von Abbildung 13 sind drei Genotypen g_i, g_j und g_k abgebildet, von denen g_j und g_k durch Mutation von g_i enstehen. Nach Abbildung in den Phänotyp-Raum (rechts) gilt für die oberste Darstellung ein stark kausaler Zusammenhang der Phänotypen p_i, p_j und p_k – die Nachbarschafts-Struktur blieb unter der Abbildung h' erhalten. Bereits bei der mittleren Darstellung handelt es sich nurmehr um eine schwache Kausalität, die Nachbarschaftsstruktur ist zerstört. Bei der nichtkausalen Abbildung im unteren Drittel von Abbildung 13 gilt dies ebenfalls allerdings sind die Störungen in der Nachbarschaftsstruktur zusätzlich nicht-deterministisch.

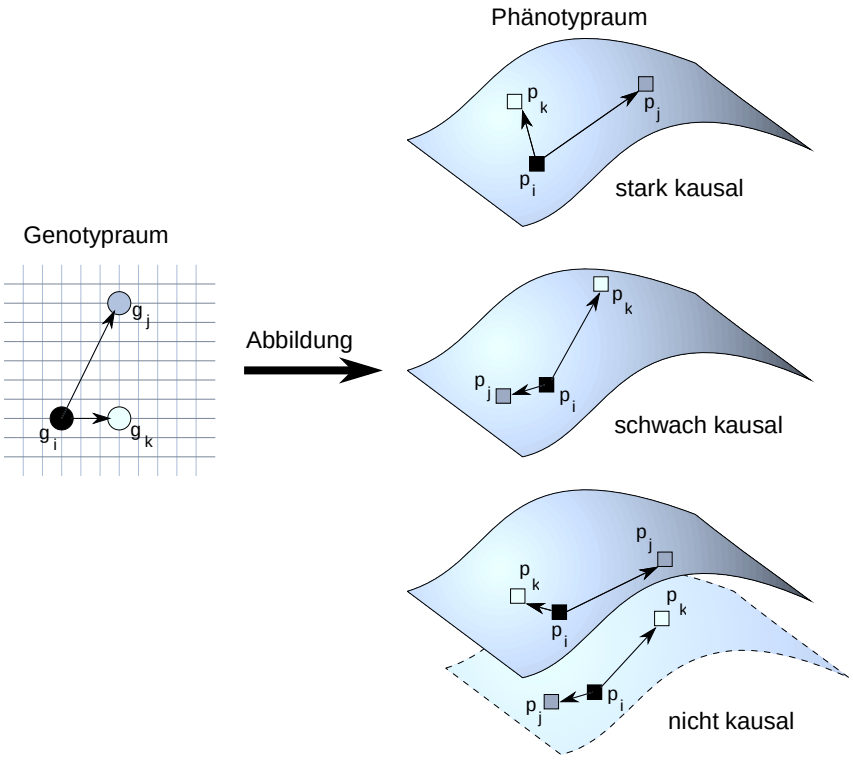


Abbildung 13 Beispiele für stark-, schwach- und nicht-kausale Abbildungen unter dem Einfluß von Mutationen.

Die Forderung nach starker Kausalität in Evolutionären Algorithmen begründen Sendhoff et al. (1997) wie folgt:

- Zum einen sollen kleine, kontrollierte Schritte im Phänotyp-Raum als Folge kleiner Schritte im Genotyp-Raum möglich sein. Insbesondere in der Nähe eines Optimums sind kleine Schritte zur langsamen Annäherung an das Optimum notwendig.
- Zum anderen ist die Möglichkeit zur Selbst-Adaption von Strategieparametern gewünscht: „[. . .] since with the lack of strong causality the information about the past is meaningless and adaption is impossible.“ (Sendhoff et al. 1997, Seite 74).

Für den Fall eines kanonischen GA zur Parameteroptimierung, mit $h' : \mathcal{G} \to \mathcal{P}$, $\mathcal{G} = \mathbb{B}^l$, $\mathcal{P} = \mathbb{N}_0$, $h'(g) = \sum_{n=0}^{l-1} g^n 2^n$ und g^n sei das n-te Bit des Genotyps g konnten Sendhoff et al. (1997) unter Verwendung eines probabilistischen Kausalitätsmaßes zeigen, daß die Anwendung des Mutations-Operators (kleine Änderung des Genotyps) in etwa der Hälfte der Fälle zu einer kleinen Änderung des Phänotyps führte. Für die umgekehrte Richtung, daß also eine kleine Änderung eines Phänotyps durch eine geringfügige Mutation eines Genotyps hervorgerufen wurde, gilt dies für etwas mehr als die Hälfte der Mutationen. Damit

zeigt sich, daß ein kanonischer Genetischer Algorithmus der zur Parameteroptimierung eingesetzt wird, die Abbildung h' nicht stark kausal ist.

Es ist somit wünschenswert eine Kombination der Abbildung h' und des Mutations-Operators m zu finden, so daß kleine Schritte im Genotypraum infolge von Mutationen auch zu kleinen Schritten im Phänotypraum führen, um bspw. die Gefahr des Überspringens des globalen Minimums zu vermeiden. Es wird sich zeigen, daß die Forderung nach starker Kausalität für das evolutionäre Design von Schaltkreisen schwierig zu realisieren ist.

Oder natürlich auch des Maximums, je nach Art der Optimimierungsaufgabe.

4.2.4 Verwendete Repräsentation und Fitneßbewertung

In dieser Arbeit wurden ausschließlich digitale Schaltkreise zur Berechnung boolescher Funktionen betrachtet. Das Hauptaugenmerk wurde bei den durchgeführten Versuchen, insbesondere im Hinblick auf die beiden vorangegangenen Abschnitte, auf das Ermitteln einer geeigneten Genotyp/Phänotyp Abbildung $h' : \mathcal{G} \rightarrow \mathcal{P}$ gelegt, die das schnelle und zuverlässige Auffinden von digitalen Schaltkreisen für boolesche Funktionen erlaubt.

Der Genotyp-Raum $\mathcal{G}$ besteht aus Binärstrings der Länge l, es gilt $\mathcal{G} = \mathbb{B}^l$. Ein Genotyp stellt eine lineare Anordnung der Konfigurationswörter von Zellen des FPGA dar, vgl. Abbildung 14. Drei Konfigurationsworte pro Zelle beschreiben die Konfiguration des Neighbour Routing, des Function Routing und der Function Unit einer Zelle.

In Analogie zur Biologie können die drei Konfigurationsworte einer Zelle auf dem Genotyp auch als Gene eines Chromosoms betrachtet werden.

Die Abbildung zwischen hintereinanderliegenden Zellen des Genotyps und ihrer Position auf dem Zellarray des FPGA übernimmt die Funktion h' bei der Übertragung der Genotyp-Information in einen Schaltkreis. Es wird stets ein rechteckiger Bereich von $n_{cols} \times n_{rows}$ Zellen in der linken unteren Ecke des FPGA zur Instantiierung eines Schaltkreises verwendet, wobei n_{cols} die Anzahl der Zellen pro Spalte und n_{rows} die Anzahl der Zellen pro Zeile angibt. Insgesamt enthält der Genotyp damit Informationen über $n_{cells} = n_{cols} \times n_{rows}$ Zellen. Sei $g(k)$ die k-te Zelle eines Genotyps $g \in \mathcal{G}$ mit $k \in \{1, \ldots, n_{cells}\}$. Dann genügt eine einfach Hilfsfunktion $pos : \mathcal{G} \rightarrow \mathbb{N} \times \mathbb{N}$ für die Berechnung der entsprechenden Position im kartesischen Koordinatensystem des Zellarrays $pos(g(k-1)) = (k \mod n_{cols}, \lfloor k/n_{rows} \rfloor)$, siehe auch Abbildung 10, Seite 31.

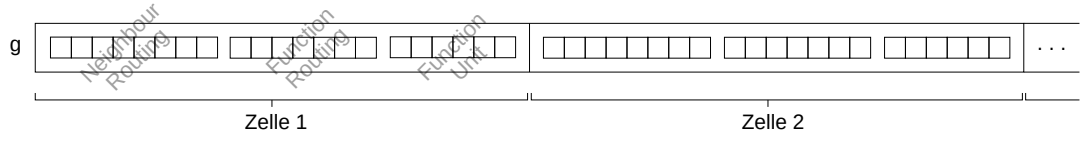


Abbildung 14 Lineare Anordnung der Zellen eines Schaltkreises in einem Genotyp g.

Der Phänotyp-Raum $\mathcal{P}$ wird durch den FPGA zusammen mit allen Schaltkreisen die auf ihm instantiiert werden können inklusive deren Verhalten dargestellt. Die Bestimmung der Fitneß eines Inviduums erfolgt durch:

- Übersetzen des Genotyps in einen Konfigurationsbitstring.
- Einschreiben des Bitstrings in den FPGA.
- Anlegen aller Testmuster aus M und Lesen des Ergebnisvektors $\vec{e}$.
- Berechnung der Hamming-Distanz zur Bestimmung eines Fitneßwertes.

Bei 4096 Zellen des XC6216 und 24 Konfigurationsbits zur vollständigen Beschreibung einer Zelle ergibt sich die astronomische Zahl von $4096^{2^{24}} \approx 10^{60605343}$ möglichen Kombinationen, wobei die Konfiguration der IOBs noch unberücksichtigt blieb.

In Anbetracht der Tatsache, daß zunächst nur boolesche Schaltkreise mit Fan-Out 1 betrachtet werden sollen, wird die **Hamming-Distanz** zur Bestimmung eines Fitneßwertes f eingesetzt, vgl. Anhang E. Die Hamming-Distanz stellt ein Maß für die Ähnlichkeit zweier Bitstrings dar, indem sie die Anzahl der Stellen zählt an denen sich diese Bitstrings unterscheiden. Die Zuweisung eines Fitneßwertes zu einem Individuum wird hier wie folgt realisiert: Das Anlegen aller Testmuster des Testmustervektors M mit Kardinalität $|M|$, vgl. Abschnitt 4.1.2, führt zu $|M|$ binären Ausgabewerten als Ergebnis der Schaltkreisberechnung. Diese $|M|$ Ausgabewerte bilden einen Bitstring als Ergebnisvektor $\vec{e}$ aus $\mathbb{B}^{|M|}$ der mit dem gewünschten Zielvektor $\vec{o}$ unter Verwendung der Hamming-Distanz verglichen wird. Das Ergebnis der Berechnung $d_H(\vec{e},\vec{o})$ wird umso kleiner sein, je näher der instantiierte Schaltkreis einer gesuchten Lösung kommt, die nämlich gerade die exakte Reproduktion der Abbildung $M \rightarrow O$ ist. Eine Lösung ist genau dann gefunden, wenn Ergebnis- und Zielvektor übereinstimmen, die Hamming-Distanz also Null ist $d_H(\vec{e},\vec{o}) = 0$.

Je Testmuster $m_i \in M$ gibt es natürlich nur dann genau eine binäre Ausgabe, wenn ausschließlich Schaltkreise mit Fan-Out 1 betrachtet werden.

Am Beispiel des 3-Parity Problems (XOR_3), bei dem genau dann eine Eins ausgegeben wird, wenn der Eingabevektor eine ungerade Anzahl von Einsen enthält, soll dieses Vorgehen veranschaulicht werden. Die Abbildung zur Rechten stellt die Wahrheitstabelle für die XOR-Verknüpfung dreier Eingabebits dar. Der Zielvektor $\vec{o}$ entspricht somit der von oben nach unten gelesenen Spalte der Ausgabemenge O, damit $\vec{o} = 01101001$. Angenommen der Schaltkreis der von einem Individuum aus $p \in P(t)$ instantiiert wurde liefert für die Eingaben $000, 001, \ldots, 111 \in M$ als Berechnungsergebnis den Vektor $\vec{e} = 01001011$, so ergibt die Berechnung der Hamming-Distanz $d_H(\vec{e},\vec{o}) = 2$; weil $d_H > 0$ stellt dieser Schaltkreis keine Lösung für das XOR_3-Problem dar. Das Individuum bekommt genau diese Distanz als Fitneßwert zugewiesen $f_p = d_H$. Auf diese Art und Weise wird verfahren bis jedem Individuum aus $p_{i=1,\ldots,N} \in P(t)$ ein Fitneßwert zugewiesen wurde. Es gilt somit für die Definition des Testmustervektors für das allgemeine n-Parity Problem $M = (bin(0), bin(1), \ldots, bin(2^n-1))$

M	O
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

und dem zugehörigen Ausgabevektor $\vec{o} = (m_0(0) \oplus m_0(1) \oplus \cdots \oplus m_0(n-1), m_1(0) \oplus m_1(1) \oplus \cdots \oplus m_1(n-1), \ldots, m_j(0) \oplus m_j(1) \oplus \cdots \oplus m_j(n-1))$, wobei $bin(i)$ die Binärdarstellung von i berechnet und $m_j(k) \in M$ das k-te Bit des j-ten Testmusters aus M bezeichnet.

Auf der gegenüberliegenden Seite ist der Ablauf des in dieser Arbeit verwendeten Genetischen Algorithmus dargestellt. Weitere Details und Besonderheiten des GA werden im folgenden Kapitel diskutiert.

Genetischer Algorithmus

// Bewerte Fitneß aller Individuen in der Population $P(0)$.
func evaluate(P)
  for $i := 1, \ldots, N$ **do**
    $P(0) \leftarrow$ writeIndividualToFPGA($p_i \in P(0)$)
    for $i = 1, \ldots, |M|$ **do**
      writePatternToFPGA($m_i \in M$)
      $\vec{e}_i :=$ readOutputFromFPGA()
    od
    $f_{p_i} := d_H(\vec{o}, \vec{e})$
  od
cnuf

// Initialisiere und validiere Population $P(0)$.
for $i := 1, \ldots, N$ **do** $p_i = \beta_i$ **od**
$P(0) \leftarrow$ validateAndRepair($P(0)$)

// Bewerte Fitneß aller Individuen in $P(0)$.
$P(0) \leftarrow$ evaluate($P(0)$)

// Betrete evolutionäre Schleife.
$t := 0$
while $\nexists p \in P(t) : f_p = 0$ **do**
  $P'(t) = P(t)$

  // Rekombiniere $P'(t)$.
  for $i := 1, \ldots, N$ **step** 2 **do**
    if $\chi_i \leq p_c$ **then**
      $(p_i, p_{i+1}) \leftarrow$ recombine(p_i, p_{i+1})
    fi
  od

  // Mutiere und validiere $P'(t)$
  for $i := 1, \ldots, N$ **do**
    for $j := 1, \ldots, l$ **do**
      if $\chi_i \leq p_m$ **then**
        $p_i(j) := 1 - p_i(j)$ **fi**
      fi
    od
  od
  $P'(t) \leftarrow$ validateAndRepair($P'(t)$)

  // Bewerte Fitneß aller Individuen in $P'(t)$.
  $P'(t) \leftarrow$ evaluate($P'(t)$)

  // Selektion. Nach der Selektion ist die Population unsortiert.
  $P(t+1) :=$ select($P'(t)$)

  $t := t + 1$
od

P	Population mit N Individuen
p	ein Individuum aus P
$p(i)$	das i-te Bit eines Individuums
l	Länge eines Individuums in Bits
f_p	Fitneß eines Individuums
p_m	Mutations-Wahrscheinlichkeit
p_c	Crossover-Wahrscheinlichkeit
M	Testmustermenge
m	ein Muster aus der Menge M
$\vec{o}$	Lösungsvektor
$\vec{e}$	Ergebnisvektor
$\beta_i \in \mathbb{B}^l$	zufällig gezogene Zahl aus $\mathbb{B}^l$
$\chi_i \in [0, 1]$	zufällig gezogene Zahl aus $[0, 1]$
$d_H(\vec{x}, \vec{y})$	Hamming Distanz von $\vec{x}$ und $\vec{y}$

5 Experimente und Ergebnisse

A world without string is chaos.
– R. Smuntz, Mousehunt

In diesem Kapitel werden die in dieser Arbeit durchgeführten Experimente beschrieben und die dabei gewonnen Ergebnisse analysiert. Bei der Experiment-Beschreibung handelt es sich vornehmlich um Details bzgl. der gewählten Genotyp/Phänotyp Abbildung $h' : \mathcal{G} \to \mathcal{P}$. Insbesondere die Repräsentation der Genotypen $g \in \mathcal{G}$ und deren Verbesserung im Hinblick auf die betrachteten Probleme wird hier ausführlich behandelt.

Die meisten der durchgeführten Experimente zielten auf das Auffinden von Schaltkreisen für das boolesche n-Parity Problem ab, vgl. auch Abschnitt 4.2.4, wobei eine Abbildung h' gesucht wurde, die das evolutionäre Finden von Schaltkreisen für möglichst große n zuläßt. Die hier vorgestellte schrittweise Verbesserung einer anfänglich sehr „groben“ Kodierung hin zu einer stetig „feiner“ werdenden, liefert eine gute Darstellung für den Gewinn an Vorwissen über das betrachtete Problem: Die scheinbar sehr beschränkten Möglichkeiten des implementierten EHW-Systems mit dem sich zunächst ausschließlich Schaltkreise als Lösung für das 3-Parity Problem finden ließen (18 Bit Kodierung), waren in Wirklichkeit nicht dem System als solches anzulasten. Vielmehr konnte durch geschickt eingesetzte Restriktionen, bspw. bei der Repräsentation der Genotypen also mithin bei der Kodierung, die Komplexität der aufgefundenen Lösungen bis hin zum 8-Parity Problem gesteigert werden.

5.1 Direkte Kodierung mit 18 Bit pro Zelle

5.1.1 Beschreibung der Kodierung

Bei der ersten gewählten Repräsentation des Genotyps wurden von den insgesamt $3 \cdot 8 = 24$ zur Verfügung stehenden Konfigurationsbits einer Zelle des XC6216 insgesamt 18 Bit verwendet. In Abbildung 15 sind genau jene Bits der drei Konfigurationswörter Neighbour Routing,

Neighbour Routing

N_{out}		E_{out}		W_{out}		S_{out}	
×	×	×	×	×	×	×	×

Function Routing

CS	X1		X2		X3		
1	0	×	×	×	×	×	×

Function Unit

M	RP	Y2		Y3		X3[2]	X2[2]
0	0	×	×	×	×	0	0

Abbildung 15 Von den drei 8 Bit-Konfigurationswörtern Neighbour Routing, Function Routing und Function Unit werden jeweils die mit einem × gekennzeichneten Bits in der 18 Bit Kodierung verwendet. Alle anderen Bits sind konstant 0 oder 1.

Function Routing und Function Unit mit einem × gekennzeichnet, die für die Kodierung einer Zelle in den Genotyp übernommen wurden.

Das Weiterleiten des D-Type Register Inhaltes als Funktionsergebnis F einer Zelle macht nur in solchen Fällen Sinn, in denen das Flip-Flop zuvor auch beschrieben wurde bzw. in denen es überhaupt benutzt werden soll. Aufgrund des in Anhang A.2 geschilderten Vorgehens zum Beschreiben des D-Type Registers einer Zelle und dem gewählten Procedere zur Expression eines Genotyps, können während eines evolutionären Laufes keine Werte in dem Flip-Flop einer Zelle eingeschrieben, ausgelesen oder gespeichert werden. Als Folge kann im Function Routing Wort stets $CS = 1$ gesetzt werden, so daß eine Zelle ausschließlich das kombinatorische Ergebnis ihrer Berechnungseinheit als Funktionsergebnis F liefert.

Der konstante Wert $RP = 0$ des Register Protect-Bit im Function Unit Wort wirkt sich ebenfalls auf die Berechnungseinheit einer Zelle aus, indem keine sog. Fast Gates von Zellfunktionen zugelassen werden. Diese Fast Gates verwenden das auf einen Konstanten Wert gesetzte D-Type Register einer Zelle, um eine schnellere Variante der verschiedenen Zellfunktionen anbieten zu können. Außer als Register für Schaltkreiseingaben sollten die Flip-Flops während des evolutionären Laufes nicht verwendet werden, so daß das Register Protect Bit stets auf 0 gesetzt ist.

Ebenso wie die Fast Gates finden auch die sog. Magic-Wires als zusätzliche Routing-Funktionen einer Zelle keine Anwendung. Daher wird das Magic-Wire Bit einer jeden Zelle als Standardeinstellung auf $M = 0$ gesetzt.

Während die drei Konfigurationsbits der Eingabeleitung X1 vollständig im Function Routing Wort Platz fand, gilt dies nicht für das jeweils höchstwertige Bit von X2 und X3. Diese wurden von den Chip-Designern des XC6216 an den Bit-Positionen 0 und 1 des Function Unit Wortes eingefügt.

Weil das Weiterleiten von Ausgabesignalen einer Zelle über die Grenzen von 4×4 Zellblöcken nicht erwünscht war, sondern als Eingaben der Funktionseinheit einer Zelle ausschließlich die Ausgabesignale *direkt benachbarter* Zellen zulässig sein sollten, bedurfte es weiterer Einschränkungen. Bei den drei letzten Bits, die auf einen konstanten Wert gesetzt werden, handelt es sich um das jeweils höchstwertige Bit der X1, X2 und X3 Input Multiplexer. Anhand von Tabelle 5 lassen sich die Folgen dieser Einschränkung leicht ablesen: An die Eingabeleitungen X1, X2 und X3 der Funktionseinheit einer Zelle können ausschließlich die Ausgaben N, E, S und W der direkt benachbarten Zellen weitergeleitet werden; das Routen von Signalen über 4×4 Blöcke von Zellen hinweg ist somit ausgeschlossen.

5.1.2 Vermeintliche Lösungen mit optimaler Fitneß

Die ersten Probeläufe zur Evolvierung von Schaltkreisen, die das 3-Parity Problem lösen, erfolgten mit einer Populationsgröße von $N = 500$ Individuen. Die Mutations-Rate betrug $p_m = 1/l$ und es wurde das 2-Punkt Crossover mit einer Crossover-Rate von $p_c = 0.7$ sowie proportionaler Selektion mit einem Elitisten verwendet; die Anzahl der zur Verfügung stehenden Zellen wurde auf eine quadratische 3×3 Fläche von Arrayzellen beschränkt – daraus folgt eine Genotyp-Länge von $l = n_{cols} \cdot n_{rows} \cdot 18 = 162$ Bit. Die Ergebnisse waren zunächst überraschend aber auch enttäuschend. Überrascht hat vor allem die schnelle Konvergenz des Genetischen Algorithmus, denn es wurden schon nach kurzer Laufzeit des GA Lösungen gefunden. Bei der Erprobung der evolvierten Schaltkreise wurde jedoch festgestellt, daß diese in Wirklichkeit *keine* Lösungen für das 3-Parity darstellen, obwohl die entsprechenden Individuen im Rahmen des Genetischen Algorithmus einen Fitneßwert von 0 zugewiesen bekamen, vgl. Abschnitt 4.2.4. Abbildung 16 zeigt einen solchen Schaltkreis, der fälschlicherweise als Lösung präsentiert wurde.

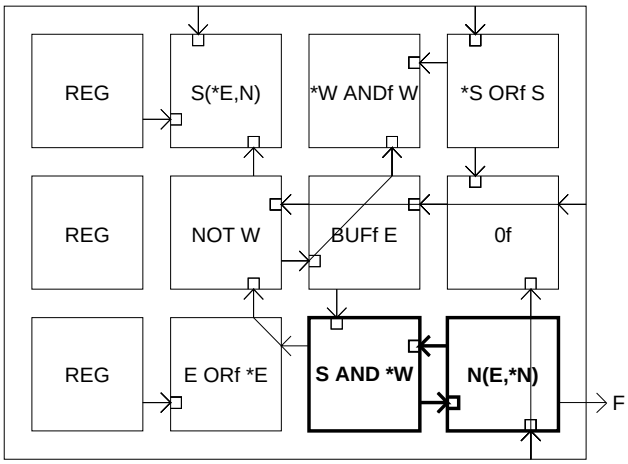


Abbildung 16 Ein Schaltkreis der, aufgrund des durch die fett gezeichneten Zellen gebildeten Zyklus, als vermeintliche Lösung für das XOR_3-Problem ausgegeben wurde.

Fernab der Tatsache, daß dieser Schaltkreis nicht die gewünschte Lösung darstellen kann, da eines der drei Eingabebits überhaupt nicht in den Schaltkreis eingespeist wird, ergibt eine genauere Analyse, daß es sich bei diesem Schaltkreis aufgrund von Rückkopplungen in Wirklichkeit um ein **Schaltwerk** handelt, d. h. die Ausgabesignale bestimmter Logik-Gatter G_j in Spalte j des Schaltkreises werden als Eingaben an Gatter G_i mit $i < j$ weitergeleitet. Aufgrund dieses Aufbaus kann es zu **instabilen Zuständen** kommen, bei denen Signale im Schaltwerk

Ein Beispiel für ein Schaltwerk mit instabilen Zuständen ist das RS-Flip Flop, bei dem die Eingabe $(1, 1)$ verboten werden muß, um eine determinstische Funktionsweise des Flip Flops zu gewährleisten.

zirkulieren und dadurch an einem dezidierten Ausgabegatter G_{out} niemals ein konstanter Wert gelesen werden kann, wenn dieses Gatter Teil des Zyklus ist. Genau dieses Verhalten läßt sich an den als Lösung ausgegebenen Schaltkreisen feststellen. Eingehende Tests des Schaltkreises zeigen, daß stets unterschiedliche Ausgaben für das Funktionsergebnis F der Ausgabezelle in der unteren rechten Ecke des Zellarrays, mithin für die Berechnung des kompletten Schaltkreises, ausgelesen werden. Dieses Verhalten ist darüberhinaus völlig unabhängig davon welche Signale an den Eingaberegistern der ersten Spalte anliegen oder zu welchen Zeitpunkten F gelesen wird. Es stellt sich heraus, daß die Ausgabezelle zusammen mit der Zelle an Position $(2, 0)$ einen **Zyklus** bildet, der Ursache des beschriebenen Verhaltens ist. Die Linke der beiden Zellen berechnet $S.\bar{W} = \bar{W}$, da die Eingabe S stets 1 ist, und wird somit das von Westen kommende Signal invertieren. Die im Westen liegende Ausgabezelle ist ihrerseits ein Multiplexer, der als Steuerleitung das Signal aus Norden verwendet und somit, da für das vom **Pad** kommende Signal stets $N = 1$ gilt, die Eingabe aus dem Osten wiederum als Funktionsergebnis liefern. Somit wird ein Signal zwischen diesen beiden Zellen kreisen und bei jedem Durchwandern der linken Zelle invertiert werden. Da die Taktung der Zellen durch den Oszillator des PCI-Boards erfolgt, werden die Signale von Zelle zu Zelle mit einer Geschwindigkeit im Megahertzbereich weitergegeben. Damit ist klar, daß aufeinanderfolgendes Lesen von F stets unterschiedliche Ergebnisse zur Folge haben wird und es wird auch klar, warum der obige Schaltkreis als Lösung des XOR_3-Problems klassifiziert wurde: Durch „Zufall" wurde an F der korrekte Lösungsvektor für das 3-Parity Problem gelesen und der Schaltkreis damit als Lösung ausgegeben.

Zyklen stellen, für die in dieser Arbeit behandelten Probleme, ein eher ungewolltes Phänomen dar. Sie lassen sich jedoch durchaus in anderem Rahmen gewinnbringend einsetzen: Durch den Nichtdeterminismus beim Auslesen von F handelt es sich hier um einen echten Zufallszahlengenerator, der Bitfolgen mit beliebiger nicht reproduzierbarer Reihenfolge erzeugen kann.

5.1.3 Verbesserte Fitneßfunktion für die 18 Bit Kodierung

Um diesem Problem entgegenzuwirken wurden Zyklen zunächst *nicht* unterbunden, da dies eine Einschränkung der Freiheitsgrade der verwendeten Methode bedeutet hätte, sondern es erfolgte eine Änderung bei der Fitneßbewertung. Anstatt nur ein einziges Mal die Testmuster $m_i \in M$ anzulegen und somit auch nur einen einzigen Ergebnisvektor $\bar{e}$ pro Fitneßbewertung zu erhalten, erfolgte dies nun fünf Mal hintereinander – der Fitneßwert eines Individuums $p \in P(t)$ berechnet sich damit als die Summe der Hamming-Distanzen der jeweiligen Ergebnisvektoren $\bar{e}_{i=1,\ldots,5}$ mit dem Lösungsvektor $\bar{o}$, also $f_p = d_H(\bar{e}_1, \bar{o}) + \cdots + d_H(\bar{e}_5, \bar{o})$. Ein Individuum repräsentiert also eine Lösung, wenn $d_H(\bar{e}_i, \bar{o}) = 0, \forall i$ gilt; in diesem Fall wird $f_p = 0$ sein. Die weiteren Versuche zeigten, daß dieses Vorgehen den gewünschten Effekt erzielt: Es werden nun korrekte Schaltkreise als Lösung für das 3-Parity Problem (XOR_3) gefunden werden.

Die Zahl fünf wurde empirisch ermittelt. Die Wahrscheinlichkeit, daß mit dem beschriebenen Verhalten der Zyklen, mehrere Male hintereinander der korrekte Lösungsvektor gelesen wird, scheint bei fünf Wiederholungen, wie die weiteren Versuche zeigten, genügend klein zu sein.

5.1.4 Simulationsergebnisse

In Abbildung 17 sind von 50 evolvierten XOR_3 Schaltkreisen exemplarisch vier beliebige Schaltkreise, entsprechend den Läufen 18, 22, 40 und 49, mit ihren Fitneßverläufen dargestellt. Die Graphen zeigen von oben nach unten gesehen den Fitneßwert (y-Achse) des schlechtesten Individuums einer Population je Generation (x-Achse), die durchschnittliche Fitneß aller Individuen, den Fitneßwert des jeweils besten Individuums sowie die Standardabweichung σ (vgl. Anhang E) der Fitneßwerte.

Der Fitneßverlauf der jeweils besten Individuen einer Population zeigt in allen vier Fällen keine kontinuierliche Verbesserung der Fitneß, sondern zumeist abrupte Sprünge die von langen Phasen gleichbleibender Fitneß dominiert werden. Es ist weiterhin zu beobachten, daß es zunächst, wie bspw. im ersten Graphen von oben, in der Anfangsphase zu einer schnelleren Verbesserung der Fitneß kommen kann (bis zur 100. Generation), während im weiteren Verlauf nur noch kleine Verbesserungen der Fitneß erfolgen. Umgekehrt kann die Fitneß jedoch auch geraume Zeit unverändert bleiben (ebenfalls bis zur 100. Generation), bis sie dann plötzlich absinkt (vgl. vierter Graph) und der GA eine Lösung findet. Auffällig bei dem ersten und vierten Graphen ist der Anstieg der Standardabweichung kurz bevor eine Lösung gefunden wird. Dies läßt auf eine Zunahme der Diversität der Population zum Zeitpunkt der Konvergenz des GA schließen. Die Graphen zwei und vier lassen jedoch vermuten, daß dieses Phänomen kein notwendiges Kriterium für das Auffinden einer Lösung darstellt. Die visuelle Überprüfung der verbleibenden Läufe hinsichtlich dieses Verhaltens ergab jedoch, daß bei insgesamt 90% der Schaltkreise die Standardabweichung zum Zeitpunkt der Konvergenz bzw. vor einer Fitneßverbesserung ansteigt, vgl. Anhang D.

In Abbildung 18 wurde die Fitneß des jeweils besten Individuums einer Population über 50 Läufe gemittelt gegen die Anzahl der benötigten Generationen aufgetragen; die durchschnittliche Anzahl von Generationen ($= 635$) ist durch die gestrichelte Linie markiert. Die Fitneßverbesserung im Rahmen dieses Genetischen Algorithmus ist kein kontinuierlicher Prozeß, sondern es kommt zu sprunghaften Fitneßverbesserungen, die von längeren Phasen gleichbleibender Fitneß gesäumt sind. Es ist darüberhinaus zu erkennen, daß eine Fitneßverbesserung häufig in mehreren aufeinanderfolgenden Schritten erfolgt, bis wiederum ein Plateau mit gleichbleibender Fitneß erreicht ist.

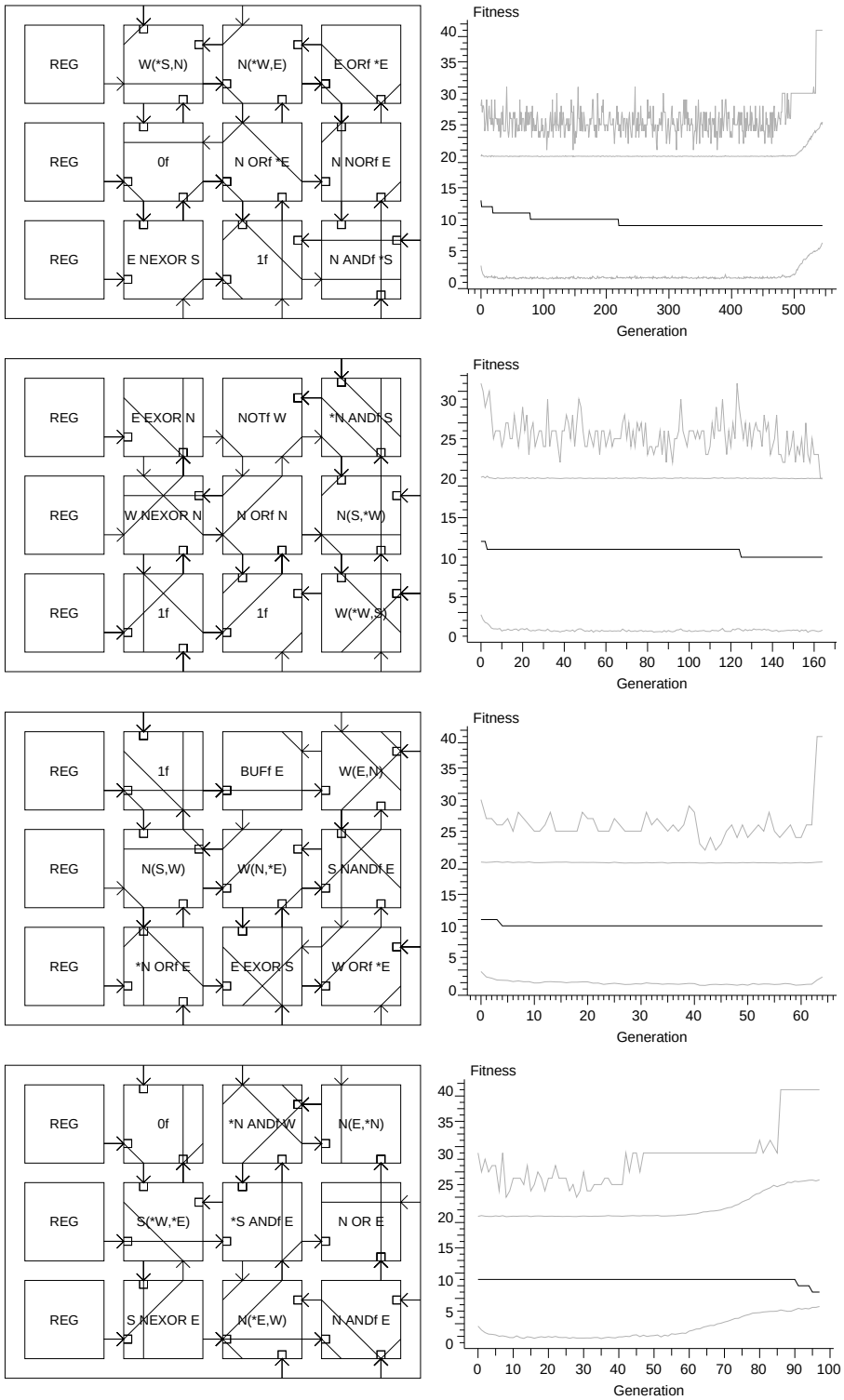


Abbildung 17 Vier evolvierte XOR_3 Schaltkreise und deren Fitneßverläufe. Die Graphen zeigen, von oben nach unten gesehen, die Fitneß des schlechtesten Individuums einer Population, die durchschnittliche Fitneß aller Individuen, die Fitneß des besten Individuums sowie die Standardabweichung der Fitneßwerte.

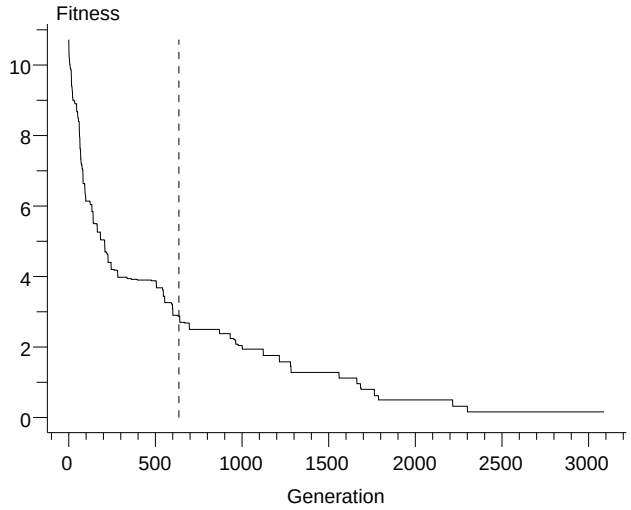


Abbildung 18 In dem Graphen ist die Fitneß des jeweils besten Individuums einer Population gegen die Generation gemittelt über 50 Läufe evolvierter XOR_3-Schaltkreise auf. Die durchschnittlich Anzahl benötigter Generationen (635) je evolviertem Schaltkreis wird durch die gestrichelte Linie markiert.

In Anbetracht der inakzeptabel langen Rechenzeit von etwa 9 Stunden für 50 Läufe bei einer mittleren Anzahl von 635 Generationen je Lauf, stellt sich die Frage, wie der evolutionäre Prozeß beschleunigt werden kann. Auch zeigt sich, daß das Evolvieren von Schaltkreisen, die das 4-Parity (XOR_4) Problem lösen, nicht mehr zu bewerkstelligen ist. In Abbildung 19 sind drei Ausschnitte des Kurvenverlaufes eines evolutionären Laufes für das XOR_4-Problem dargestellt, der bei einer Laufzeit von mehreren Tagen nach etwa 377000 Generationen abgebrochen wurde.

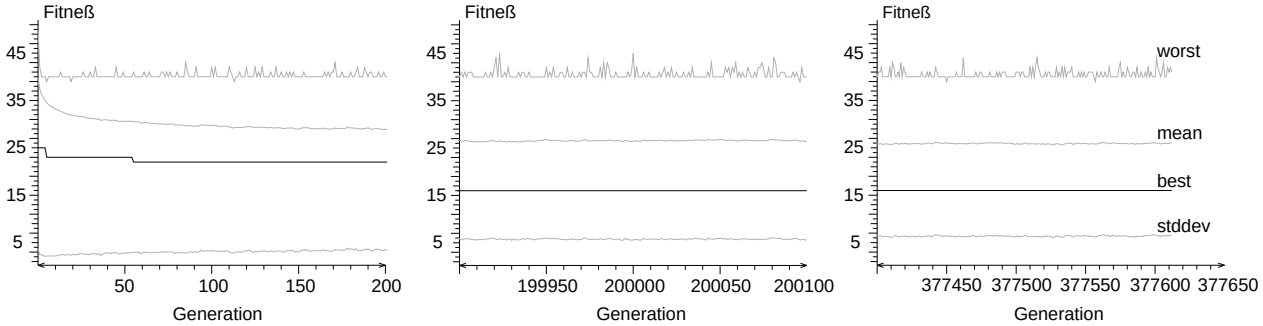


Abbildung 19 Fitneßverläufe einer XOR_4-Evolution, die nach ca. 377000 Generationen erfolglos abgebrochen wurde. Die Kurven geben von oben nach unten die gleichen Daten wie in Abbildung 17 wieder. Anhand des linken Graphen ist ein leichter Anstieg der Standardabweichung während der ersten 200 Generationen zu beobachten. Dieser Trend setzt sich fort, bis sich die Standardabweichung bei einem Wert von etwa 7 einpendelt.

Im Graphen auf der linken Seite von Abbildung 19 ist zu erkennen, wie die Fitneß des besten Individuums einer Population während der ersten 200 Generationen zunächst fällt, dann jedoch bis zum Abbruch des Laufes bei einem Wert von 16 verbleibt. Weiterhin ist zu beobachten, daß die Standardabweichung (unterste Kurve) in den ersten 200 Generationen ansteigt, sich im weiteren Verlauf jedoch ebenfalls nicht mehr ändert – die Diversität der Population scheint ab einem bestimmten Zeitpunkt nicht weiter zuzunehmen. Zusammen mit den Beobachtungen bei der Evolvierung von XOR_3-Schaltkreisen wird gefolgert, daß der erfolgreiche evolutionäre Entwurf von n-Parity Schaltkreisen einen gewissen Erhalt der Unterschiedlichkeit der Individuen einer Population benötigt. In Anbetracht der nicht zufriedenstellenden Komplexität der bisher betrachteten Schaltkreise, muß der evolutionäre Prozeß dahingehend verfeinert werden, daß er zum einen schneller eine Lösung findet. Zum anderen sollen jedoch auch komplexere Schaltkreise in akzeptabler Zeit evolviert werden können. Versuche in diese Richtung umfaßten dabei die Erprobung alternativer Selektionsmechanismen im Hinblick auf den Diversitätserhalt innerhalb der Population und die Variation des Rekombinationsoperators.

5.1.5 Untersuchung des Rekombinations-Operators

Die Versuche des vorhergehenden Abschnitts erfolgten allesamt mit einem einfachen 2-Punkt Crossover Operator. Bei den weiteren Versuchen konnte die Rekombination als zeitkritischer Hemmschuh für das evolutionäre Design digitaler Schaltkreise entlarvt werden. Neben der Untersuchung des z-Punkt Crossover für $z = 2$ und $z = 8$ wurden zwei neue Rekombinations-Operatoren entworfen, die beim Ermitteln der Bruchstellen eines Individuums bestimmten Restriktionen unterliegen.

Das **herkömmliche Crossover** c_{norm} betrachtet den Genotyp eines Individuums als lineare Abfolge von Bits – die z Bruchstellen werden zufällig über die gesamte Länge des Strings ermittelt. Die beiden **neuen Crossover-Operatoren** c_{NNF} und c_{cells} basieren auf der Idee nur zwischen funktionell zusammenhängenden Einheiten des Genotyps zu schneiden. In Anbetracht der Repräsentation eines Genotyps, vgl. Abbildung 14, können solche Einheiten zum einen die Neighbour Routing, Function Routing und Function Unit Abschnitte sein, zum anderen aber auch die komplette Beschreibung einer Zelle. Somit läßt der c_{NNF} Operator ausschließlich an den Grenzen der für die einzelnen Konfigurationswörter zuständigen Bereiche auf einem Genotyp zu. Alternativ dazu tauscht c_{cells} nur komplette Zellen während der Rekombination aus. Abbildung 20 veranschaulicht dieses Vorgehen anhand dreier Bitstrings.

Sowohl c_{NNF} als auch c_{cells} wurden im Rahmen eines 2-Punkt Crossover verwendet. Darüberhinaus erfolgten zwei weitere evolutionäre

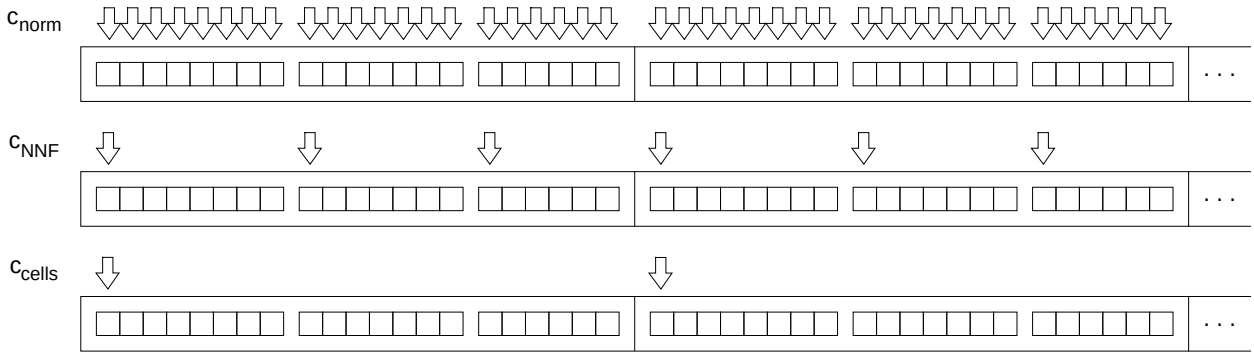


Abbildung 20 Für den herkömmlichen Rekombinations-Operator c_{norm} ist jede Bitposition als Bruchstelle zulässig während die beiden anderen Operatoren c_{NNF} und c_{cells} an die Bitpositionen funktioneller Einheiten des Individuums gebunden sind.

Läufe: Zum einen kam nochmals c_{norm} jedoch mit $z = 8$ zum Einsatz, zum anderen wurde das Uniform Crossover angewendet. Beim Uniform Crossover, das sich nach Syswerda (1989) besser für bestimmte Testfunktionen wie bspw. pseudobooleschen eignet, wird an jeder Bitposition zufällig entschieden, welches Bit der beteiligten Individuen in den Nachkommen übernommen wird. Alle Versuche wurden mit einer Crossover-Rate von $p_c = 0.7$ durchgeführt. In Abbildung 21 sind die Ergebnisse anhand der Fitneßverläufe gemittelt über jeweils 50 Läufe dargestellt. Die kleinen Graphen zeigen für alle 50 Läufe die Anzahl der Generationen, die jeweils für einen Schaltkreis benötigt wurden. Die durschnittliche Anzahl benötigter Generationen wird jeweils durch eine gestrichelte Linie markiert.

Dies entspricht im Mittel einem $l/2$-Punkt Crossover, wenn die Länge der Bitstrings l beträgt.

Es fällt unmittelbar auf, daß das qualitative Aussehen aller Fitneßverläufe der evolutionären Prozesse, die mit einem Rekombinationsoperator arbeiten, sehr ähnlich ist. Im Hinblick auf die Anzahl der Generationen, die jeweils für einen einzelnen Lauf benötigt wurden (vgl. kleine Grafiken in Abbildung 21) weisen alle Graphen einen mehr oder weniger stark zerklüfteten Verlauf auf. Einzig in dem Lauf, bei dem ohne Crossover gearbeitet wurde (unten rechts), zeigt sich ein vergleichsweise ruhiger Verlauf dieser Kurve. Aus diesem Graphen läßt sich erkennen, daß bei der Nichtverwendung eines Crossover-Operators eine weniger starke Streuung bzgl. der Anzahl an Generationen, die für die Evolvierung eines XOR_3 Schaltkreises benötigt wird, herrscht. Dieses Ergebnis ist erstaunlich, da dem Rekombinations-Operator in Standard Genetischen Algorithmen normalerweise die größte Bedeutung zugemessen wird. Es zeigt sich jedoch, daß durch die alleinige Anwendung des Mutations-Operator eine schnellere Konvergenz des GA erzielt werden kann. In Tabelle 9 sind die Ergebnisse aus Abbildung 21 noch einmal tabellarisch aufgeführt.

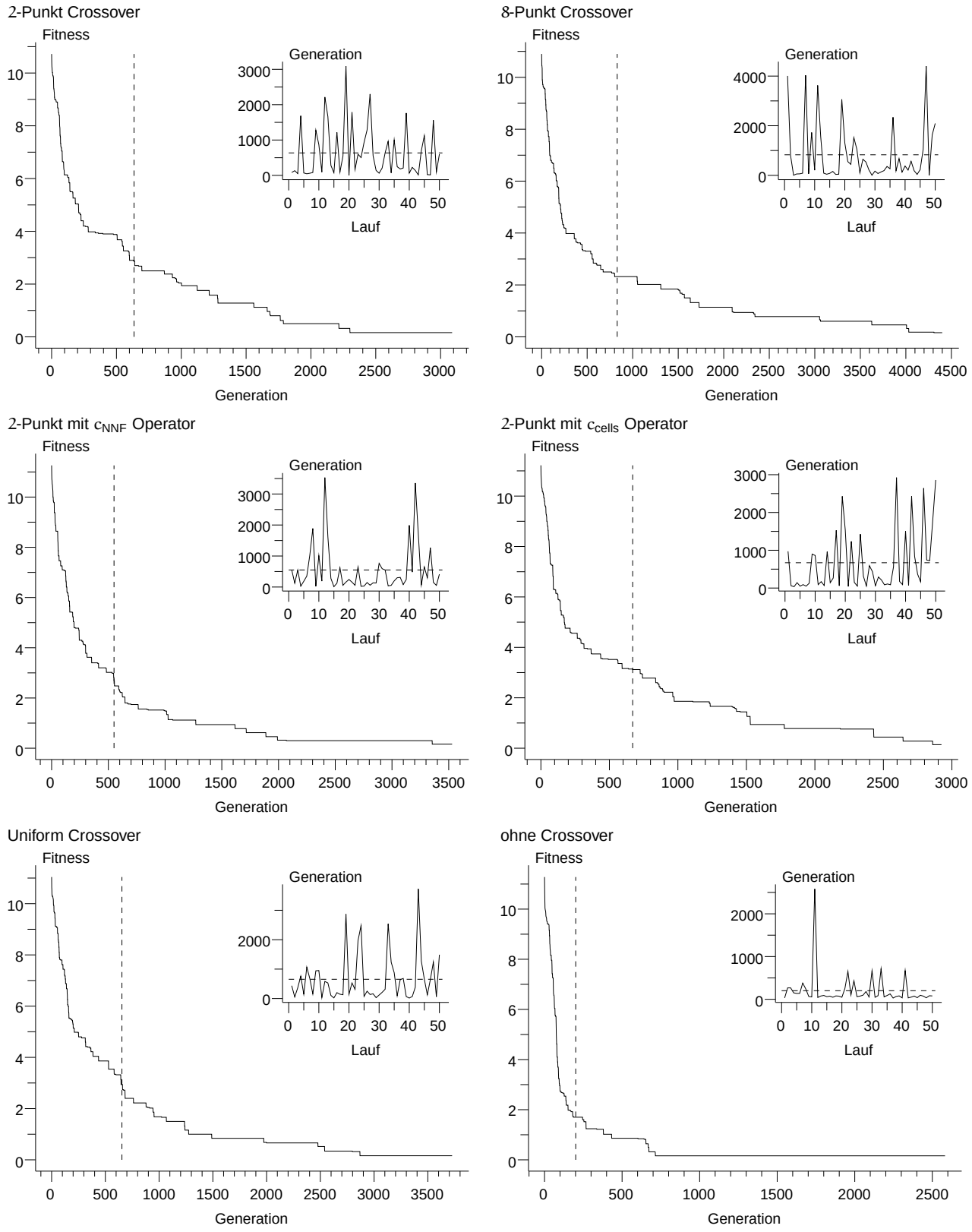


Abbildung 21 Fitneßverlauf des jeweils besten Individuums einer Population bei Verwendung verschiedener Crossover-Operatoren gemittelt über jeweils 50 Läufe. In den kleinen Graphen ist die Anzahl der benötigten Generationen je Lauf für alle Läufe von 1 bis 50 aufgeführt. Die durchschnittliche Anzahl benötigter Generationen pro evolviertem Schaltkreis ist jeweils durch die gestrichelte Linie markiert.

Tabelle 9 Ergebnisse der Untersuchung verschiedener Rekombinationsoperatoren. Die durschnittliche Anzahl benötigter Generationen $\bar{g}$ für einen Lauf wurde über insg. 50 Läufe gemittelt. Der Standardfehler ist mit $m_{\bar{g}}$ angegeben.

Operator	$\bar{g}$	$m_{\bar{g}}$
2-Punkt c_{norm}	635.54	135.89
8-Punkt c_{norm}	829.20	201.09
Uniform	654.70	146.25
2-Punkt c_{NNF}	550.12	133.46
2-Punkt c_{cells}	670.72	148.90
n/a	201.10	61.06

Um den Einfluß des Crossover-Operators c_{norm}, im Vergleich zum Mutations-Operator, auf die Fitneßveränderung besser beurteilen zu können, wurde anhand des ersten in Abbildung 21 (oben/links) durchgeführten Laufes der Fitneß-Gain (Igel und Chellapilla 1999) untersucht, d. h. es wurde ermittelt in wievielen Fällen das Crossover zweier Eltern $p_i, p_j \in P$ einen Nachkommen p_k erzeugte, dessen Fitneß besser, gleich oder schlechter als die seiner Eltern war. Von den 3^2 möglichen Kombinationen wie sich die Fitneß von p_k gegenüber der von p_i und p_j darstellen kann, ist in Abbildung 22 ein Ausschnitt der ersten 100 Generationen für den Kuvenverlauf im Falle

- gleichbleibender Fitneß $f_{p_k} = f_{p_i} = f_{p_j}$ (gestrichtelte Linie),
- sich verschlechternder Fitneß $(f_{p_k} > f_{p_i}) \wedge (f_{p_k} > f_{p_j})$ (gepunktete Linie),
- sich verbessernder Fitneß $(f_{p_k} < f_{p_i}) \wedge (f_{p_k} < f_{p_j})$ (durchgezogene Linie),
- einer besseren Fitneß des Nachkommen einem der Eltern gegenüber $(f_{p_k} < f_{p_i}) \vee (f_{p_k} < f_{p_i})$ (Strichpunkt-Linie)

abgebildet. Für jede Generation wird die Anzahl der Crossover-Operationen des entsprechenden Typs dargestellt.

Von den insgesamt $5.57 \cdot 10^6$ Crossover-Operationen die während der 50 Läufe ausgeführt wurden, ergaben etwa 91% überhaupt keine Fitneßverbesserung, d. h. die Fitneß des Nachkommen ist gleich der beider Eltern. Anhand der Strichpunkt-Linie ist zu beobachten wie die Anzahl der Crossover-Operationen, bei denen der Nachkomme zumindest einem der Eltern gegenüber eine besserer Fitneß aufweist, in den ersten ca. 100 Generationen zunächst deutlich über den beiden Kurven liegt, die die Verbesserung/Verschlechterung gegenüber beiden Eltern anzeigen. Ab etwa der 100. Generation nimmt jedoch die Anzahl der Crossover-Operationen dieses Typs ebenfalls ab. Es ergibt sich, daß in

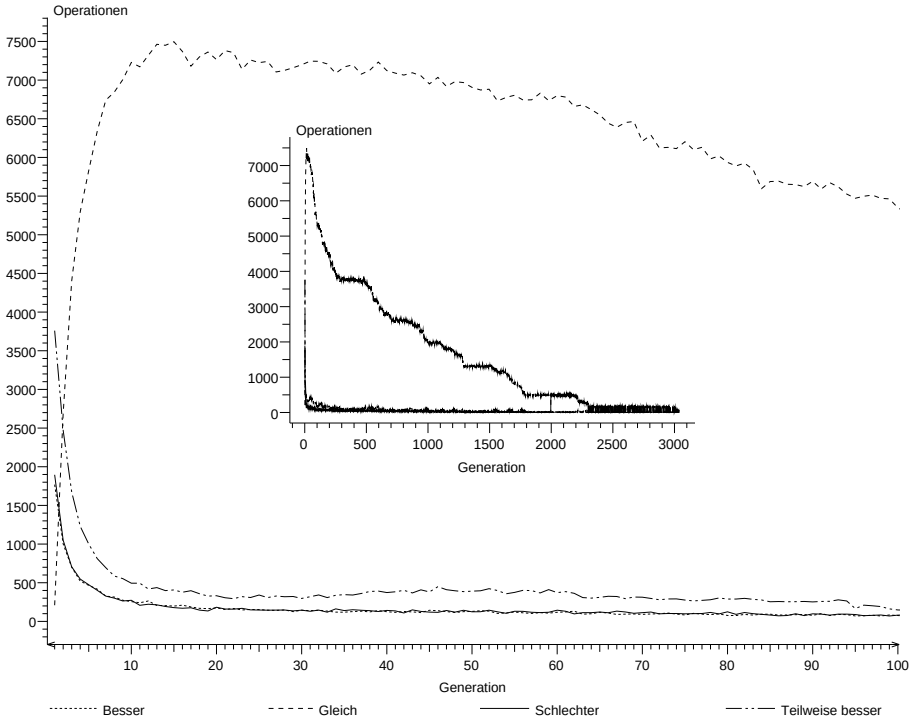


Abbildung 22 Der große Graph führt die Anzahl an Crossover-Operationen entsprechend des jeweiligen Typs für jede Generation gezählt über 50 Läufe an; es wird ein Ausschnitt der Generationen 0–100 gezeigt. Im kleinen Graph ist der Verlauf für alle Generationen dargestellt. Die obere Kurve scheint zu sinken, da es weniger Läufe gibt, die besonders viele Generationen benötigt haben und somit bei diesen im Mittel auch weniger Mutations- und Crossover-Operationen stattfanden.

Die restlichen Prozentpunkte verteilen sich auf die fünf hier nicht betrachteten Fälle, wie bpsw. $(f_{p_k} > f_{p_i}) \wedge (f_{p_k} = f_{p_i})$ etc.

etwa 3.3% der Fälle der Nachkomme zumindest einem der Eltern gegenüber eine bessere Fitneß hat. Bei jeweils ca. 1.5% der Crossover-Operationen verbessert und verschlechter sich die Fitneß des Nachkommen gegenüber *beiden* Eltern. Nur ein vergleichsweise geringer Prozentsatz an Crossover-Operationen führt somit überhaupt zu einer Fitneßverbesserung; gleiches gilt jedoch auch für die Fitneßverschlechterung. Das Gros stellen neutrale Rekombinationen dar, bei denen, zumindest im Hinblick auf den Fitneßwert, nichts passiert.

Im Vergleich dazu wurde der Mutations-Operator auf ähnliche Weise untersucht. Es lassen sich nur die drei Fälle, daß eine Fitneßverbesserung, eine Fitneßverschlechterung oder gar keine Änderung der Fitneß nach Anwendung des Mutations-Opertors auftrat, unterscheiden. In Abbildung 23 werden die entsprechenden Kurvenverläufe analog zu Abbildung 22 dargestellt. Von ca. 10^7 während der 50 Läufe durchgeführten Mutations-Operationen stellen ein Großteil von 95.57% neutrale Mutationen dar. Dies sind Mutationen, die zwar den Bitstring verändern, sich jedoch nicht in der Fitneß des Individuums nieder-

schlagen. Zu einer Verschlechterung der Fitneß kam es in 1.97% der Fälle, während für 2.46% eine Fitneßverbesserung zu verzeichnen war.

Dieses Ergebnis zeigt, daß, verglichen mit dem Crossover-Operator, nur ein Prozentpunkt mehr an durchgeführten Mutations-Operationen zu einer Fitneßverbesserung führt. In Relation zur Gesamtanzahl durchgeführter Operationen muß auch beachtet werden, daß nahezu doppelt so viele Mutations- ($1.00 \cdot 10^7$) wie Crossover-Operationen ($5.57 \cdot 10^6$) stattfanden. Wird dieses Verhältnis berücksichtigt, so bleiben von den Mutations-Operationen noch etwa 1.4%, die einen Fitneßzuwachs erzielten. Dies entspricht fast exakt dem Wert für die Crossover-Operationen, bei denen der Nachkomme besser als beide Eltern ist.

Ausgehend von der Analyse des Fitneß-Gain sollte sich der Einsatz beider Operatoren ähnlich auf den evolutionären Prozeß auswirken. Dies ist jedoch nicht so, wie Tabelle 9 belegt. Die alleinige Verwendung des Mutations-Operators führt zu einer deutlich niedrigeren durchschnittlichen Anzahl an Generationen, die für einen Lauf benötigt werden. Aufgrund dieser Beobachtungen wurden alle weiteren Versuche ohne Rekombinations-Operator durchgeführt.

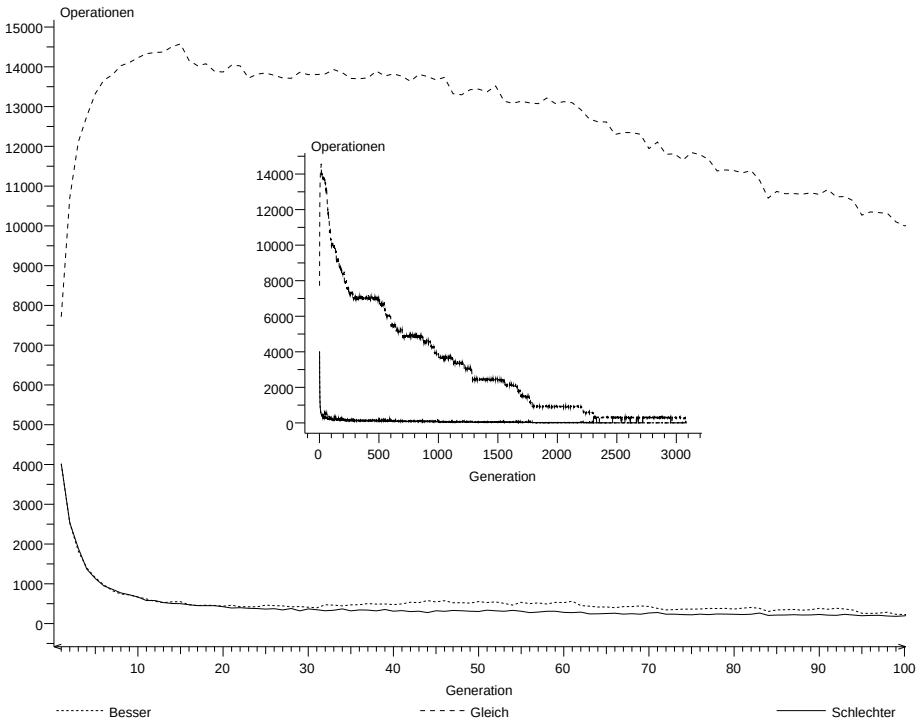


Abbildung 23 Die Kurven des großen Graphen zeigen (von oben nach unten) die Anzahl der Mutations-Operationen, die zu keiner Fitneßverbesserung führten, die zu einer besseren Fitneß führten und die eine Fitneßverschlechterung zur Folge hatten; gezählt wurden die Mutations-Operationen von insg. 50 Läufen.

5.1.6 Vergleich verschiedener Selektionsmechanismen

In Anbetracht der Vermutung, daß sich eine möglichst große Diversität in der Population günstig auf den evolutionären Prozeß auswirkt, wurden einige Untersuchungen mit verschiedenen Selektionsmechanismen durchgeführt. Dabei wurden die folgenden Selektionsverfahren verglichen:

- **Proportional**. Bei diesem Selektionsverfahren ist die Wahrscheinlichkeit eines Individuums in die Nachfolgepopulation aufgenommen zu werden proportional zu seinem Fitneßwert. Um der Gefahr zu entgehen, daß das beste Individuum einer Population sich mit diesem Selektionsmechanismus nicht in der Folgepopulation weitervermehrt, wird im Rahmen einer sog. Elitisten-Strategie stets das beste Individuum einer Population in die nächste Population übernommen
- **Whitley's Linear Ranking**. Dies ist ein rangbasiertes Verfahren. Individuen werden mit einer gewissen Wahrscheinlich selektiert die auf der Basis einer (nicht wachsenden) Zuweisungsfunktion, die ausschließlich auf dem Rang eines Individuums in der Population beruht, bestimmt wird. Dabei wird davon ausgegangen, daß die Individuen bgzl. ihres Fitneßwertes absteigend sortiert sind. Die Zuweisungsfunktion für Whitley's Linear Ranking gibt dabei direkt den Index des Individuums zurück, das selektiert werden soll. Mit einem Parameter η_{max} läßt sich der Selektionsdruck variieren; je größer η_{max} ist, desto größer ist auch der Selektionsdruck.
- q**-Tournament**. Bei der q-Tournament Selektion wird aus einer Gruppe von q zufällig gewählten Individuen das Beste ausgewählt. Dieser Prozeß wird so oft wiederholt, bis die nötige Anzahl an Individuen in der Nachfolgepopulation erreicht ist.
- **EP** q**-Tournament**. Die EP q-Tournament-Selektion ist eine Mischung aus deterministischer und stochastischer Selektion, bei der jedes Individuum aus der Elterpopulation $P(t)$ und Nachfolge-Population $P(t+1)$ mit q zufällig ausgewählten Individuen aus der Vereinigungsmenge von $P(t)$ und $P(t+1)$ in einem Turnier antritt. Wenn der Fitneßwert des aktuellen Individuums größer oder gleich eines seiner q Kontrahenten ist, bekommt das entsprechende Individuum einen Gewinn gutgeschrieben. Die m Individuen mit den meisten Gewinnen formen dann die nächste Population. Falls zwei Individuen dieselbe Anzahl von Gewinnen aufweisen, wird jenes mit dem besseren Fitneßwert in die Nachfolgepopulation übernommen. Bei diesem Selektionsmechanismus handelt es sich um eine Elitistenstrategie.

Abbildung 24 zeigt eine vergleichende Übersicht der Fitneßverläufe des jeweils besten Individuums einer Population gemittelt über 50

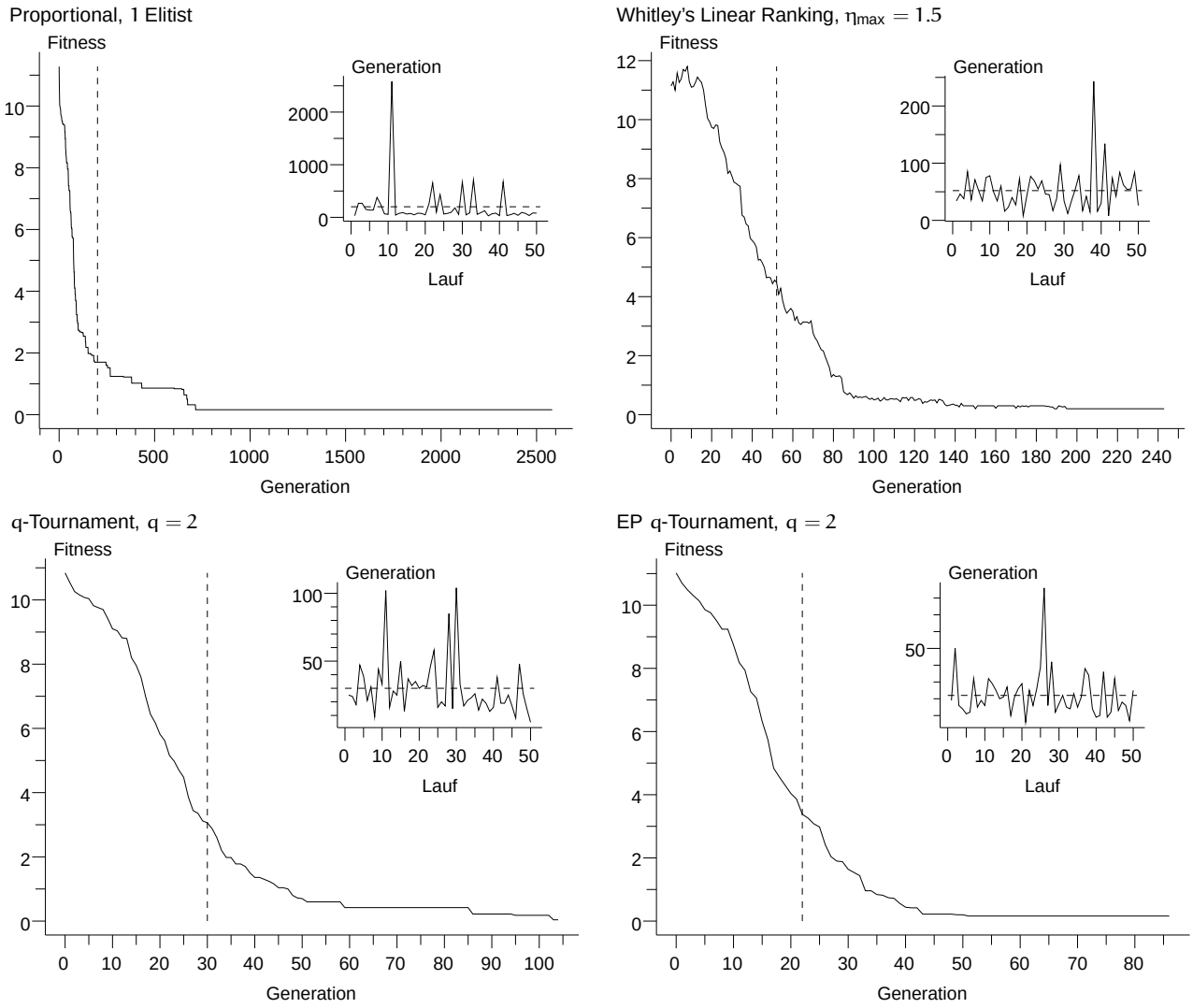


Abbildung 24 Fitneßverlauf des jeweils besten Individuums einer Population bei Verwendung verschiedener Selektionsmechanismen gemittelt über jeweils 50 Läufe. In den kleinen Graphen ist die Anzahl der benötigten Generationen je Lauf für alle Läufe von 1 bis 50 aufgeführt. Die durchschnittliche Anzahl benötigter Generationen pro evolviertem Schaltkreis ist jeweils durch die gestrichelte Linie markiert

Läufe. Auffallend ist die Tatsache, daß unter Verwendung der drei Selektionsverfahren q-Tournament, EP q-Tournament und Whitle's Linear Ranking der evolutionäre Prozeß deutlich schneller zu einer Lösung konvergiert als dies mit der fitneßproportionalen Selektion erfolgt. Die kleinen Graphen stellen die Anzahl der benötigten Generationen eines jeden der 50 Läufe dar – die durchschnittliche Anzahl an Generationen je Schaltkreis wird durch die gestrichelte Linie markiert.

In Tabelle 10 wird für jeden Selektionsmechanismus die durchschnittliche Anzahl an Generationen $\bar{g}$, die für die Evolvierung eines

Tabelle 10 Selektionsverfahren und ihr Einfluß auf die Konvergenzgeschwindigkeit bei der evolutionären Suche nach XOR_3-Schaltkreisen. $\bar{g}$ ist die durchschnittliche Fitneß des besten Individuums einer Population gemittelt über 50 Läufe, $m_{\bar{g}}$ der zugehörige Standardfehler.

Selektionsverfahren	Parameter	$\bar{g}$	$m_{\bar{g}}$
Proportional	1 Elitist	201.1	61.06
Whitley's Linear Ranking	$\eta_{max} = 1.5$	52.7	9.08
q-Tournament	$q = 2$	30.14	5.12
EP q-Tournament	$q = 2$	22.24	3.63
EP q-Tournament	$q = 10$	100.14	53.23

Schaltkreises benötigt wird, zusammen mit dem mittleren Fehler $m_{\bar{g}}$ des Mittelwertes (Standardfehler, vgl. Anhang E) für alle Selektionsverfahren gegenübergestellt. Angesichts der hohen Konvergenzgeschwindigkeit die durch die Verwendung der EP q-Tournament Selektion zu beobachten ist (durchschnittlich 22 Generationen bis eine Lösung gefunden wird) wurde mit dieser Strategie erneut versucht eine Lösung für das 4-Parity Problem (XOR_4) zu finden. Jedoch scheiterte auch dieser Versuch mit einem Fitneßverlauf ähnlich dem in Abbildung 19.

Für alle weiteren Versuche wurde aufgrund der ermittelten Daten ausschließlich EP q-Tournament als Selektionsverfahren eingesetzt.

5.1.7 Evolution at work?

Die Frage, ob bei dem evolutionären Schaltkreisentwurf wie er bisher praktiziert wurde wirklich eine Art „Evolution" stattfindet, ist nicht ganz unberechtigt. So könnte doch vermutet werden, daß, insbesondere beim Weglassen des Rekombinations-Operators und der alleinigen Exploration des Suchraumes durch kleine Mutationsschritte, eine Art zufällige Suche stattfindet. Die Zielrichtung eines Evolutionären Algorithmus wird einzig durch die Bewertung von Individuen mit einem Fitneßwert und der anschließenden Selektion zur Produktion von Nachkommen erwirkt. Es stellt sich die Frage, ob nicht eine rein zufällig Suche mit der **Monte Carlo Methode** ebenfalls Lösungen für die hier betrachteten Probleme liefern kann.

Zum Vergleich mit den bereits erzielten Ergebnissen wurde die Monte Carlo Methode zur Suche nach einem Schaltkreis eingesetzt der das 3-Parity Problem löst. Dazu wurden die Konfigurationsbits eines Individuums zufällig generiert, dieses Individuum als Schaltkreis im FPGA instantiiert, dessen Fitneß bewertet und, falls das Individuum keine Lösung darstellt, wiederum mit der zufälligen Belegung begonnen. Es handelt sich um ein Verfahren, bei dem bereits „gezogene" Individuen zurücklegt werden, ein und derselbe Schaltkreis somit mehrmals instantiiert werden kann.

Im Laufe des Versuches konnten tatsächlich 23 XOR_3-Schaltkreise zufällig gefunden werden. Allerdings wurde das Experiment nach 2 Tagen während des 24. Laufs abgebrochen, da zu diesem Zeitpunkt bereits ca. $5 \cdot 10^6$ zufällige Schaltkreise getestet wurden. Bei einer Größe des Suchraumes von $2^{n_{cells} \cdot n_{bitsPerCell}} = 2^{3 \cdot 3 \cdot 18} = 2^{162} \approx 5.84 \cdot 10^{48}$ entspricht dies allerdings nur einem Bruchteil von $8.61 \cdot 10^{-42}$% aller mit der gegebenen Kodierung möglichen Schaltkreise, die ausprobiert wurden. Gemittelt über die 23 gefunden Schaltkreise, wurden im Durchschnitt nur alle ca. 210000 Züge eine Lösung gefunden. In Tabelle 11 ist diese Anzahl der Züge der Anzahl an Fitneßbewertungen gegenübergestellt, die für die Lösung der einzelnen Probleme mit der 18 Bit Kodierung benötigt wurden. Es stellt sich heraus, daß die Monte Carlo Methode im Mittel schneller einen Schaltkreis findet, der eine Lösung für das 3-Parity Problem darstellt, als dies bei *allen* evolutionären Läufen der Fall ist, in denen ein Rekombinations-Operator eingesetzt wurde. Allerdings ist die zufällige Suche auch deutlich schlechter, als ein evolutionärer Prozeß, der ausschließlich mit einem Mutations-Operator arbeitet.

Zum Vergleich der Größenordnung: Die geschätzte Anzahl an H_2O Molekülen in den Weltmeeren beträgt $4 \cdot 10^{46}$.

Tabelle 11 Gegenüberstellung der durschnittlichen Anzahl von Zügen bei der Monte Carlo Methode zur Auffindung von XOR_3-Schaltkreisen mit der Anzahl der im Durchschnitt benötigten Fitneßauswertungen für die einzelnen Läufe mit der 18 Bit Kodierung. Weil die Populationsgröße bei den evolutionären Läufen stets $N = 500$ betrug gilt für die durchschnittliche Anzahl an Fitneßbewertungen $\bar{f} = \bar{g} \cdot 500$.

Methode	Crossover	Selektion	$\bar{f}(\bar{g})$
Monte Carlo	n/a	n/a	$21 \cdot 10^4$ (n/a)
18 Bit Kodierung	2-Punkt c_{norm}	Proportional	$31 \cdot 10^4$ (635)
·	8-Punkt c_{norm}	·	$41 \cdot 10^4$ (829)
·	Uniform	·	$32 \cdot 10^4$ (654)
·	2-Punkt c_{NNF}	·	$27 \cdot 10^4$ (550)
·	2-Punkt c_{cells}	·	$33 \cdot 10^4$ (670)
·	n/a	·	$10 \cdot 10^4$ (201)
·	n/a	Linear Ranking	$2 \cdot 10^4$ (52)
·	n/a	q-Tournament	$1 \cdot 10^4$ (30)
·	n/a	EP q-Tournament	$1 \cdot 10^4$ (22)
·	n/a	EP q-Tournament	$5 \cdot 10^4$ (100)

5.2 Die 16 Bit Kodierung

5.2.1 Beschreibung der Kodierung

Wie die Experimente mit der 18 Bit Kodierung zeigen, ist das alleinige Weglassen des Rekombinations-Operators sowie die Variation des Selektionsmechanismus nicht ausreichend, um die Komplexität des betrachteten n-Parity Problems von $n = 3$ auf $n = 4$ steigern zu können. Um dieses Ziel dennoch zu erreichen, werden die Freiheitsgrade der verwendeten Methode dahingehend eingeschränkt werden, daß Zyklen in Schaltkreisen nicht mehr zulässig sind.

Dies wird erreicht, indem die Ausgaben einer Zelle des FPGA ausschließlich in die Richtungen *North*, *East* und *South* weitergegeben werden dürfen. Signalweiterleitungen in die *West* Richtung sind verboten. Ein Zyklus von einer Spalte des Zell-Arrays zur nächsten wie er in Abbildung 16, Seite 57, dargestellt wurde ist damit nicht mehr möglich. Ohne weitere Einschränkungen könnte jedoch eine Schleife *innerhalb* einer Spalte auftreten, wie sie in Abbildung 25 dargestellt ist.

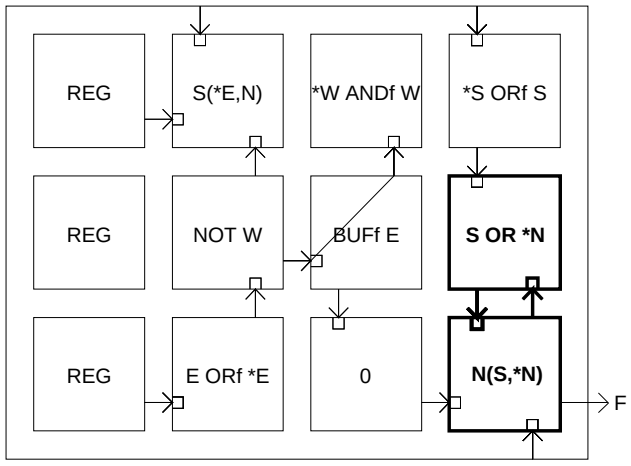


Abbildung 25 Ein Zyklus in der 4. Spalte, der durch die beiden fett gezeichneten Zellen gebildet wird.

Die beiden fett gezeichneten Zellen der letzten Spalte bilden einen Zyklus; die Signale $\mathtt{F}$, die von der Ausgabezelle rechts unten gelesen werden sind zufällig, vgl. Abschnitt 5.1.2. Um auch solche Zyklen zu vermeiden, bedarf es einer entsprechenden Initialisierung der Startpopulation $\mathtt{P}(0)$ sowie der Validierung eines Individuums nachdem es mutiert wurde.

Um dieses Vorhaben realisieren zu können wird eine neue Genotyp/Phänotyp Abbildung $\mathtt{h}'$ verwendet. In Abbildung 26 sind die Konfigurationswörter einer Zelle abgebildet, wobei alle mit einem $\times$ gekennzeichneten Bits in der neuen Genotyp/Phänotyp Abbildung für die Kodierung einer FPGA-Zelle verwendet werden. Im Vergleich zur 18 Bit Kodierung, vgl. Abbildung 15, fielen 2 Bits im Function Unit

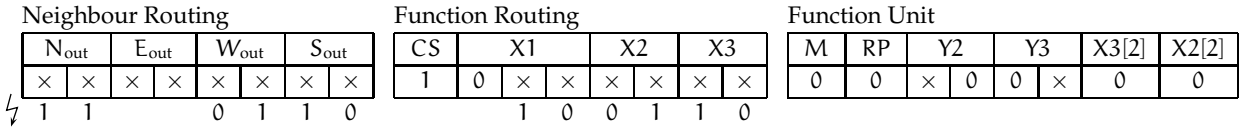


Abbildung 26 Die in der 16 Bit Kodierung verwendeten Bits der Zell-Konfigurationswörter sind mit einem $\times$ gekennzeichnet. Unzulässige Bitkombinationen bei der Belegung sind in der mit einem Blitz markierten Zeile aufgeführt; bspw. ist die Belegung der S_{out} Bits mit $(1,0)$ nicht erlaubt, da hierdurch $S_{out} = W_{in}$ gelten würde und eine solche Konfiguration zu Zyklen führen kann.

Konfigurationswort weg, so daß nurmehr 16 Bit für die Kodierung einer Zelle verbleiben. Darüberhinaus sind für die Bits des Neighbour und Function Routing Wortes nicht alle Belegungen zulässig; in Abbildung 26 sind diese Bitkombinationen in der mit einem Blitz gekennzeichneten Zeile zu sehen: Beim Neighbour Routing Wort ist die Bitkombination $(1,1)$ als Belegung für N_{out} nicht zulässig, da dies $N_{out} = W_{in}$ bedeuten würde (vgl. Tabelle 4). W_{in} darf jedoch nicht durch eine Zelle geleitet werden. Aus diesem Grund sind die Eingaben $(0,1)$ fur S_{out} und $(0,1)$ für W_{out} verboten, da sie ebenfalls das Signal W_{in} weiterleiten würden.

Die unzulässigen Bitkombinationen der Eingabeleitungen X1, X2 sowie X3 im Function Routing Konfigurationswort werden aus dem gleichen Grund verboten, da die Signale W_{in} zusätzlich auch nicht als Eingabesignale der Funktionseinheit verwendet werden dürfen. Damit werden also genau jene Bitkombinationen ausgeschlossen, die zu den Belegungen $X1 = X2 = X3 = W_{in}$ führen würden, vgl. Tabelle 5.

Im Function Unit Wort können die Bits Y2[0] und Y3[1] auf den konstanten Wert 0 gesetzt werden, da keine Fast Gates verwendet werden und somit für Y2 nur die Bitkombinationen $(0,0)$ und $(1,0)$ sowie für Y3 nur $(0,0)$ und $(0,1)$ sinnvoll sind, vgl. Tabelle 6.

Der Genotyp besteht somit nach wie vor aus einer linearen Abfolge von Bits von denen jeweils 16 Bit eine Zelle des FPGA beschreiben, siehe auch Abbildung 14 auf Seite 50.

5.2.2 Validierung und Reparatur von Individuen

Bei dem hier eingesezten Genetischen Algorithmus werden die Individuen der Startpopulation $P(0)$ zufällig initialisiert. Da jedes Individuum aus $P(0)$ ein linearer Bitstring der Länge l ist, erfolgt diese Initialisierung durch die zufällige Wahl eines Wertes aus $\{0,1\}$ für jede Bitposition des Strings. Durch dieses Vorgehen ist es sehr wahrscheinlich, daß einige der Individuen die in Abbildung 26 aufgezeigten unzulässigen Bitkombinationen an den entsprechenden Stellen des Strings aufweisen. Auch sind Kombinationen möglich, die zu Zyklen

wie in Abbildung 25 dargestellt führen können. Es ist daher notwendig die Individuen nach der zufälligen Initialsierung einer Validierung zu unterziehen, die dafür sorgt, daß weder unzulässige Bitkombinationen noch Zyklen der genannten Form vorliegen. Weiterhin ist diese Validierung nach jeder Mutation eines Individuums erforderlich, da die Mutationsstellen auf dem Bitstring zufällig gewählt werden und es dadurch zu unzulässigen Bitkombinationen kommen kann.

Die Validierung des Neighbour Routings ist dabei vergleichsweise einfach durchzuführen. Die unzulässige Bitkombination $(1, 1)$ für N_{out} wird korrigiert, indem eines der beiden Bits zufällig auf 0 gesetzt wird – es gilt dann entweder $N_{out} = N_{in}$ oder $N_{out} = E_{in}$. Die Korrektur verbotener Bitkombinationen für W_{out} und S_{out} erfolgt durch die Invertierung eines der beiden Bits, so daß entweder S_{in} oder F weitergeleitet wird.

Die Überprüfung des Function Routing gestaltet sich schwieriger. Neben der Unzulässigkeit von W_{in} für die Eingableitungen der Funktionseinheit, muß dafür Sorge getragen werden, daß innerhalb einer Spalte keine Zyklen auftreten können. Dies wird wie folgt erreicht: Ein Zelle darf das Signal S_{in} nicht als Eingabe für X1, X2 oder X3 verwenden, wenn sie gleichzeitig ihr Funktionsergebnis F nach N_{out} leitet *und* sie darf das Signal N_{in} nicht als Eingabe für X1, X2 oder X3 verwenden, wenn sie gleichzeitig ihr Funktionsergebnis F nach S_{out} weiterleitet. Für den Fall, daß eine nicht zulässige Bitkombination vorliegt, wird die entsprechende Eingangsleitung der Funktionseinheit mit dem stets gültigen Eingabesignal E_{in} belegt. Mit diesem Vorgehen lassen sich erfolgreich Zyklen innerhalb einer Spalte unterbinden.

5.2.3 Simulationsergebnisse

Zunächst wurde die Brauchbarkeit der implementierten Kodierung anhand des 3-Parity Problems überprüft. Die Parameter des Genetischen Algorithmus wurden dabei wie folgt eingestellt: Die Populationsgröße beträgt $N = 500$ Individuen. Als Chipfläche stehen $n_{cells} = 3 \cdot 3$ Zellen zur Verfügung. Die Mutationsrate beträgt $1/l$ wobei $l = n_{cells} \cdot 16 = 144$ Bit die Länge der Bitstrings ist, so daß im Mittel eine Mutation je Individuum durchgeführt wird. Ein Rekombinations-Operator kommt nicht zur Anwendung. Als Selektionsmechanismus wird EP q-Tournament mit einer Turniergröße von $q = 10$ eingesetzt.

In Abbildung 27 (links) ist ein mit dieser Kodierung evolvierter Schaltkreis abgebildet. Es ist zu sehen, daß die Bedingungen für die Zyklenfreiheit wie sie im vorherigen Abschnitt erläutert wurden erfüllt sind.

Auf der rechten Seite von Abbildung 27 ist ein entsprechender Fitneßverlauf gemittelt über 50 Evolutions-Läufe dargestellt. Die durchschnittliche Anzahl benötigter Generationen $\bar{g} = 41$ je Lauf wird durch

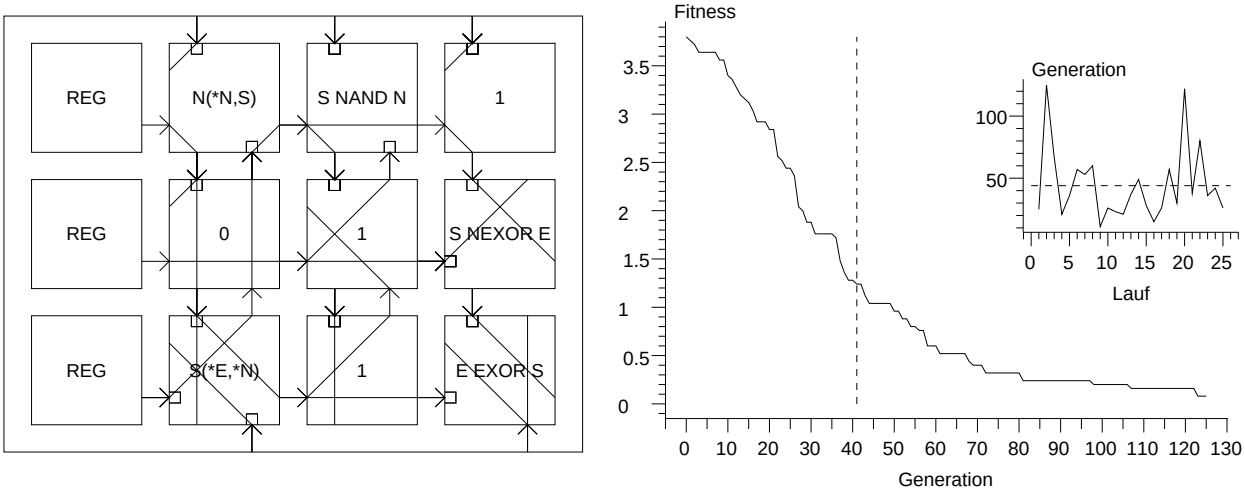


Abbildung 27 Links: XOR_3-Schaltkreis, der mit der 16 Bit Kodierung evolviert wurde. Rechts: Der zugehörige Fitneßverlauf des jeweils besten Individuums einer Population gemittelt über 50 Läufe. Der kleine Graph zeigt die Anzahl benötigter Generationen für jeden Lauf. Die durchschnittliche Anzahl benötigter Generation ($= 41$) wird durch die gestrichelte Linie markiert.

die gestrichelte Linie markiert. Der kleine Graph zeigt, wieviele Generationen für jeden einzelnen der 50 Läufe benötigt wurden (Standardfehler $m_{\bar{g}} = 6.94$). Verglichen mit der 18 Bit Kodierung bei gleicher Wahl der Parameter des Genetischen Algorithmus, fällt auf, daß die Konvergenzgeschwindigkeit ähnlich ist. Bei der 18 Bit Kodierung wurden im Durchschnitt 22 Generationen zur Evolvierung eines XOR_3-Schaltkreises benötigt, während dies bei der 16 Bit Kodierung 41 Generationen sind. Ein Zuwachs bei der Konvergenzgeschwindigkeit konnte mit der neuen Kodierung für das XOR_3-Problem somit nicht erreicht werden.

Allerdings war es möglich mit dieser Kodierung die Komplexität des n-Parity Problems von $n = 3$ auf $n = 4$ zu steigern; Abbildung 28 (links) zeigt einen der evolvierten XOR_4-Schaltkreise mit dem entsprechenden, über 50 Läufe gemittelten, Fitneßverlauf zur Rechten. Es fällt auf, daß die durschnittlich benötigte Anzahl an Generationen je Lauf ungleich höher ist, als dies bei der Evolvierung des XOR_3-Schaltkreises der Fall war. Hier werden nun im Mittel $\bar{g} = 399$ Generationen mit einem Standardfehler von $m_{\bar{g}} = 83$ benötigt. Weil sich die Größe des Suchraumes nach der Anzahl der verwendeten Zellen und natürlich der Anzahl der Bits je Zelle richtet, steigt die Größe des Suchraumes bei dem Schritt vom 3-Parity zum 4-Parity Problem von $2^{3 \cdot 3 \cdot 16} \approx 10^{48}$ auf $2^{4 \cdot 4 \cdot 16} \approx 10^{77}$ an.

Für das 5-Parity Problem konnte mit dieser Kodierung keine Lösung mehr gefunden werden. Insbesondere zeigte sich, daß bei ei-

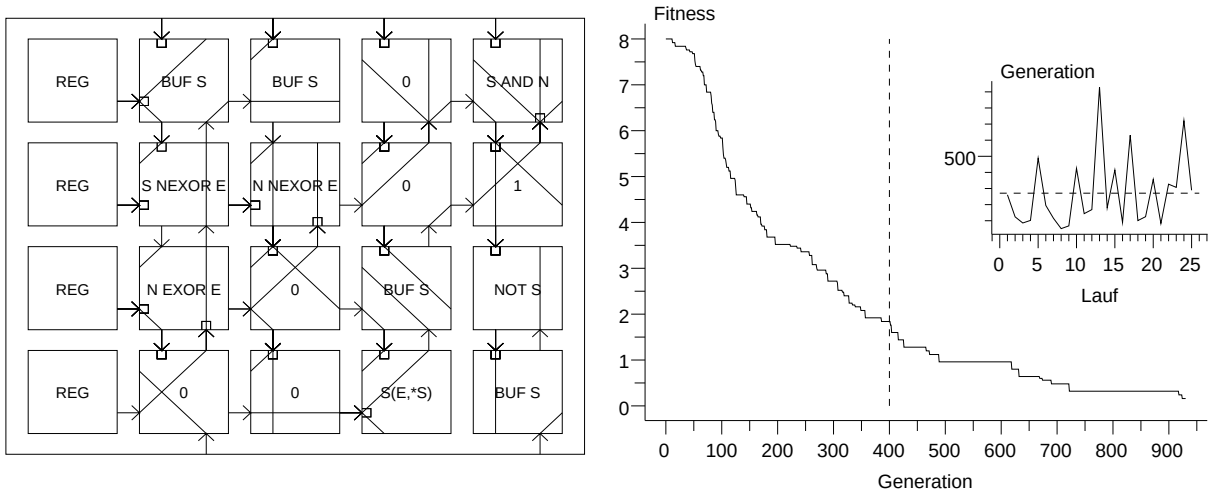


Abbildung 28 Links: XOR_4-Schaltkreis, der mit der 16 Bit Kodierung evolviert wurde. Rechts: Der zugehörige Fitneßverlauf des jeweils besten Individuums einer Population gemittelt über 50 Läufe. Der kleine Graph zeigt die Anzahl benötigter Generationen für jeden Lauf. Die durchschnittliche Anzahl benötigter Generation (399.6 ± 83.06) wird durch die gestrichelte Linie markiert.

nem entsprechenden Lauf der nach der 8000. Generation abgebrochen wurde, überhaupt keine Fitneßverbesserung zu verzeichnen war. Darüberhinaus wiesen die Fitneß des besten und schlechtesten Individuums einer Population sowie die durchschnittliche Fitneß aller Individuen keine Unterschiede auf – alle hatten denselben Wert, der sich über die Generationen auch nicht änderte. Dies läßt darauf schließen, daß die meisten, wenn nicht sogar alle, Individuen schon nach kürzester Zeit identisch waren; hier wäre jedoch ein genauerer Blick in die Population notwendig. Dieser Mangel an Diversität in der Population scheint die Ursache für die schlechte Leistung des GA zu sein. Wenn alle Individuen sehr ähnlich oder gar identisch sind, so können keine größeren Sprünge im Suchraum vorgenommen werden – der GA ist nicht mehr in der Lage aus einem lokalen Minimum zu entfliehen. Auch die Verwendung eines Rekombinations-Operators hilft an dieser Stelle nicht weiter, da er stets nur Nachkommen produzieren würde, die nahezu identisch zu seinen Eltern sind. Möglicherweise könnte die Verwendung einer adaptiven Mutationsrate dazu führen, daß bei (scheinbarer) Konvergenz des GA gegen ein lokales Minimum, durch die Erhöhung der Mutationsrate aus diesem entkommen werden kann.

5.3 Die 9 Bit Kodierung

5.3.1 Beschreibung der Kodierung

Während mit der Kodierung des vorhergehenden Abschnitts, die auf der Vermeidung von Zyklen basiert, das 4-Parity Problem zu lösen war, wird mit der im folgenden vorgestellten Kodierung die Evolvierung von n-Parity Schaltkreisen mit $n \in \{3, \ldots, 8\}$ möglich sein. Mit dieser neuen Kodierung, die auf weiteren Einschränkungen bei der Signalweiterleitung basiert, können somit XOR_8-Schaltkreise evolutionär entworfen werden. Die Idee dabei ist, daß Ausgabesignale bzw. weitergeleitete Signale einer Zelle ausschließlich nach E_{out} und S_{out} propagiert werden dürfen. Der Unterschied zur 16 Bit Kodierung besteht damit in dem Verbot der Verwendung von N_{out}.

In Abbildung 29 sind die Bits der drei Konfigurationswörter einer Zelle die nun verwendet werden sollen mit einem $\times$ gekennzeichnet. Im Gegensatz zur 16 Bit Kodierung sind nur noch an zwei Stellen des Neighbour Routing Wortes unerlaubte Belegungen möglich. Aufgrund der Beschränkung auf E_{out} und S_{out} als Signalleitungen konnten weitere Bits der drei Wörter auf Konstante Werte gesetzt werden – es verbleiben somit 9 Bit für die Kodierung einer Zelle des FPGA.

5.3.2 Validierung von Individuen

Es ist wiederum eine Validierung von Individuen sowohl bei der Initialisierung der Startpopulation als auch nach der Mutation eines Individiuums notwendig, da es zu unerlaubten Belegungen der Konfigurationsbits kommen kann. Die Überprüfung der Korrektheit ist jedoch, verglichen mit der Validierung bei der 16 Bit Kodierung, einfacher.

Die beiden einzigen unerlaubten Bitkombinationen können beim Neighbour Routing auftreten, vgl. Abbildung 29. Dabei ist die Belegung $(0, 1)$ für den E_{out} Ausgang einer Zelle nicht zulässig, da in diesem Fall $E_{out} = N_{in}$ gelten würde. Dies wird durch zufällige Invertierung eines der beiden Bits korrigiert. Die zwei zulässigen Bitkombinationen sind damit: $(1, 1)$ oder $(0, 0)$, die zu $E_{out} = S_{in}$ oder $E_{out} = F$ führen.

Neighbour Routing

N_{out}		E_{out}		W_{out}		S_{out}	
0	0	×	×	0	0	×	×
		0	1			1	0

Function Routing

CS	X1			X2		X3	
1	0	0	×	×	0	0	×

Function Routing

M	RP	Y2		Y3		X3[2]	X2[2]
0	0	×	0	0	×	0	0

Abbildung 29 Bei der 9 Bit Kodierung sind nur noch an zwei Stellen des Neighbour Routing Wortes unerlaubte Belegungen möglich. Weil Das Routing von Ausgabesignalen nur nach E_{out} und S_{out} zulässig sein soll, ist bspw. die Belegung der E_{out} Bits mit $(0, 1)$ verboten, da in diesem Fall – vgl. Tabelle 4 – $E_{out} = N_{in}$ gelten würde und diese Konfiguration nicht zulässig ist.

Für den S_{out} Ausgang ist die Bitkombination $(1,0)$ verboten, da aus ihr $S_{out} = W_{in}$ folgt und W_{in} ebenso wie in der 18 Bit Kodierung nicht verwendet werden sollen. Auch hier wird zur Korrektur eines der beiden Bits zufällig invertiert. Es folgen die beiden Bitkombinationen $(1,1)$ oder $(0,0)$, die zu den zulässigen Verschaltungen $S_{out} = S_{in}$ oder $S_{out} = F$ führen.

5.3.3 Simulationsergebnisse

Mit der neuen Kodierung werden zum einen erstaunlich schnell Schaltkreise evolviert, zum anderen können *deutlich komplexere* n-Parity Probleme gelöst werden als dies mit der 18 oder 16 Bit Kodierung möglich ist. Abbildung 30 (rechts/oben) zeigt die Fitneßverläufe des n-Parity Problems für $n = 3, \ldots, 8$ gemittelt über jeweils 50 Läufe. Zur besseren Übersicht wurde der Bereich zwischen Generation 0 und 100 vergrößert dargestellt. Jeweils ein Beispiel für einen XOR_3- und XOR_8-Schaltkreis ist links/oben und im unteren Bereich von Abbildung 30 zu sehen. An den beiden Beispielen ist das Routing-Konzept der neuen Kodierung gut zu erkennen – Signale werden ausschließlich nach S_{out} und E_{out} weitergeleitet. Die Signale der Eingaberegister werden treppenförmig nach unten geleitet, wobei das Berechnungsergebnis des Schaltkreises wiederum an der Zelle in der unteren rechten Ecke der benutzten Chipfläche abgelesen wird. In Tabelle 12 ist für jedes betrachtete Problem die durchschnittliche Anzahl benötigter Generationen $\bar{g}$ je Lauf gemittelt über 50 Läufe sowie der zugehörige Standardfehler $m_{\bar{g}}$ der einzelnen Messung angegeben.

Tabelle 12 Durchschnittliche Anzahl von Generationen $\bar{g}$ für jeden mit der 9 Bit Kodierung durchgeführten evolutionären Prozeß für eines der n-Parity Probleme mit $n = 3, \ldots, 8$ gemittelt über jeweils 50 Läufe. $m_{\bar{g}}$ gibt den Standardfehler der jeweiligen Messung an.

Problem	$\bar{g}$	$m_{\bar{g}}$
XOR_3	5.58	1.07
XOR_4	22.12	3.53
XOR_5	152.36	24.29
XOR_6	71.46	10.66
XOR_7	122.00	18.65
XOR_8	211.30	23.71

Auffallend an den Werten in Tabelle 12 ist, daß für das XOR_5-Problem im Mittel $\bar{g} = 152.36$ Generationen benötigt wurden, und damit mehr als für das XOR_7-Problem. Weil $\bar{g}$ qualitativ anscheinend exponentiell anwächst, wird die Messung für XOR_5 als Ausreißer betrachtet.

Wie sich zeigt konnte, durch die Beschränkung der Freiheitsgrade bei der Verbindung von benachbarten Zellen untereinander, ein enormer Zuwachs bei der Komplexität der zu lösenden Probleme erzielt werden. Es wird vermutet, daß sich die gewählte Art der Kodierung besonders gut gerade für die Evolvierung von n-Parity Problemen eignet, da sich der bereits erwähnte treppenförmige Abstieg der Eingabesignale von den Registern der ersten Spalte bis zur Ausgabezelle in

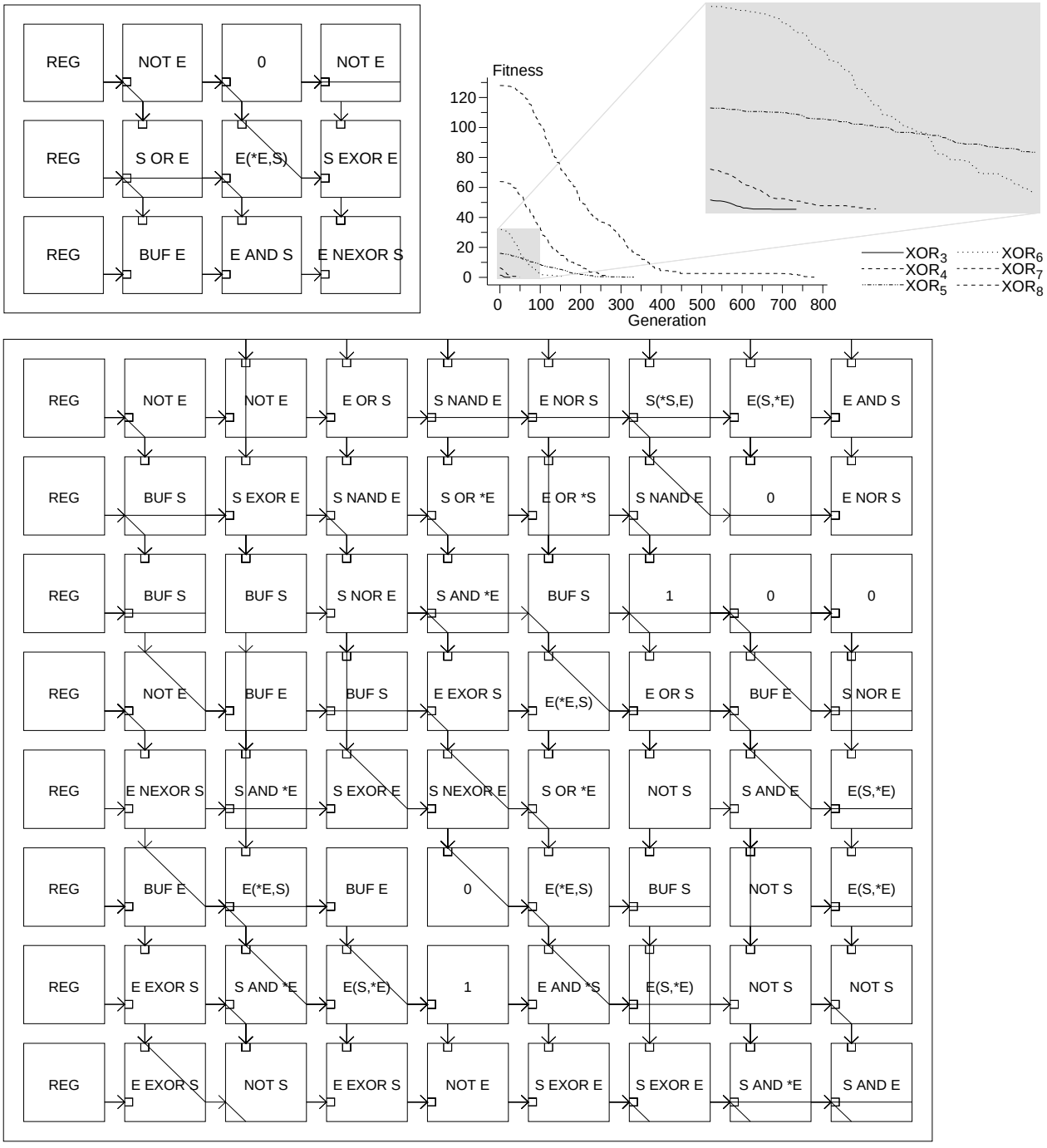


Abbildung 30 Links/oben und unten: Jeweils ein XOR_3-Schaltkreis und XOR_8-Schaltkreis. Rechts/oben: Fitneßverlauf des jeweils besten Individuums einer Population gemittelt über 50 Läufe. Es werden die Kurven für die n-Parity Probleme mit $n = 3, \ldots, 8$ gezeigt. Der schwer zu erkennende Bereich von Generation 0 bis 100 ist vergrößert dargestellt.

der letzten Spalte für eine kaskadenartige Berechnung anbietet. Dem n-Parity Problem ist gerade eine solche Berechnungsweise inhärent, weil

nämlich aus $y = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ leicht $z = x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus x_{n+1}$ zu ermitteln ist, da $z = y \oplus x_{n+1}$ gilt. Jede beliebige Teilformel von z läßt sich auf diese Weise bestimmen und am Ende das Ergebnis in der beschriebenen Weise zusammensetzen. Es wird vermutet, daß bei der Weiterleitung der Signale durch den Schaltkreis ein vergleichbares Berechnungsschema vorliegt. Tatsächlich zeigt eine genauere Betrachtung der Zellen in der unteren linken Ecke des XOR_8-Schaltkreises in Abbildung 30 (unten), daß zunächst $x_2 \oplus x_3$ und dann $x_1 \oplus (x_2 \oplus x_3)$ berechnet wird. Parallel dazu erfolgt die Berechnung von $x_4 \oplus x_5$, während dieses Ergebnis dann später in die Berechnung von $(x_1 \oplus (x_2 \oplus x_3)) \oplus (x_4 \oplus x_5)$ einfließt.

Die Eingaberegister seien von unten nach oben mit $x_1, \ldots, x_8$ bezeichnet.

5.3.4 $XOR_{n\geq 3}$-Schaltkreise ohne XOR_2-Bausteine

Bei den bisher evolvierten Schaltkreisen, die das XOR_n mit $n = 3, \ldots, 8$ Problem lösen, wurden als Zellfunktionen neben AND, OR, etc. auch die XOR-Funktion selber als Logikbaustein zugelassen. Es stellt sich die Frage, ob die Evolvierung von n-Parity Schaltkreisen auch *ohne* Verwendung von XOR als „Building Block" möglich ist und wenn ja, bis zu welcher Komplexität das betrachtete Problem reichen kann. Die n-Parity Funktion gilt bei der Realisierung in Hardware, bspw. als Logikbaustein auf einem Chip, als schwierig, da meist nur eine feste Basis einfacher Funktionen wie etwa $U = \{\neg, \wedge, \vee\}$ benutzt werden darf. Angenommen es soll ein XOR_3-Logikbaustein entwickelt werden, der auf der Basis U die Funktion $y = x_1 \oplus x_2 \oplus x_3$ berechnet. Dann lautet die Formel für y ohne Verwendung des $\oplus$-Operators

„[...] $\oplus$-Bausteine mit großem Fan-in [sind] wesentlich schwieriger zu bauen als $\wedge$- oder $\vee$-Bausteine mit großem Fan-in" (Wegener 1989, Seite 30).

$$y = \left[((x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge x_2)) \wedge \bar{x_3}\right] \vee \left[\neg((x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge x_2)) \wedge x_3\right].$$

Schon für kleine n ist die resultierende Formel vergleichsweise komplex. Der evolutionäre Entwurf eines XOR_3-Schaltkreises mit der 9 Bit Kodierung *ohne* Verwendung der Zellfunktionen $A \oplus B$ und $\overline{A \oplus B}$, vgl. Tabelle 7, scheint, in Anbetracht der Komplexität von y und der Tatsache, daß eine Chipfläche des FPGA von 3×3 Zellen verwendet wird, zum Scheitern verurteilt. Zwar können Teilformeln von y wie bspw. $(x_1 \wedge \bar{x_2})$ oder $(\bar{x_1} \wedge x_2)$ jeweils doppelt verwendet werden und darüberhinaus stehen dem evolutionären Entwurf mehr Funktionen zur Verfügung als ausschließlich die der Basis U. Jedoch allein die eingeschränkte Anzahl von 9 Zellen und die Tatsache, daß bei der 9 Bit Kodierung, das Berechnungsergebnis stets im unteren rechten Bereich an der Ausgabezelle vorliegen muß, läßt an der Lösbarkeit zweifeln. Abbildung 31 veranschaulicht den Datentransport anhand einer Chipfläche von $64 = 8 \times 8$ Zellen. Dabei wird angenommen, daß durch die Beschränkung der Signalweiterleitung bei der 9 Bit Kodierung auf E_{out} und S_{out}, die Wahrscheinlichkeit der Benutzung von grau hinterlegten Zellen mit abnehmender Helligkeit kleiner wird.

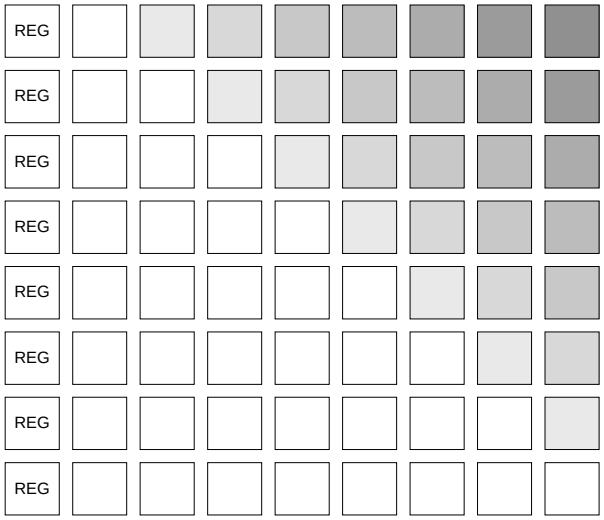


Abbildung 31 Am Beispiel einer 8×8 Chipfläche wird gezeigt, welche Zellen bei der 9 Bit Kodierung vermutlich eher genutzt werden als andere. Je dunkler die Zelle dargestellt ist, desto unwahrscheinlicher ist deren Verwendung im evolvierten Schaltkreis.

Die experimentellen Ergebnisse bestätigen die Vermutung, daß auf einer Fläche von 3×3 Zellen ohne Verwendung des $\oplus$-Operators als Logikbaustein keine Lösung für das XOR_3 Problem gefunden werden kann. Auch die Erhöhung der Anzahl zur Verfügung stehender Zellen durch Hinzunahme weiterer Spalten führt nicht zum Erfolg. Mit der gegebenen Art der Signalweiterleitung wäre es wahrscheinlich erfolgversprechender nicht nur die Spaltenzahl sondern auch gleichzeitig die Anzahl an Zeilen zu erhöhen. Die Einspeisung der Eingabesignale über die Register der ersten Spalte würde dann im „oberen Teil" der Chipfläche erfolgen, bei 10×10 Zellen also bspw. in den Zeilen 7, 8 und 9 oder versetzt in den Zeilen 0, 4 und 8.

Für die Evolvierung von XOR_3-Schaltkreisen wurden Chipflächen von 3×3, 4×3, 5×3 und 6×3 Zellen verwendet.

5.3.5 Redundanz der Kodierung

Während der Arbeit mit der 9 Bit Kodierung wurde festgestellt, daß die Auswahl der Konfigurationsbits, die in den Genotyp übernommen wurden, eine gewisse Redundanz der Kodierung zur Folge hatte.

Die Funktion F, die von einer Logikzelle i des FPGA berechnet wird, ist vollständig durch das Tupel $F_i = (X1, X2, X3, Y2, Y3)$ bestimmt, vgl. Tabelle 7. Für den Fall, daß die Anzahl der möglichen Eingänge für $X1, X2$ und $X3$ auf zwei beschränkt wird und darüberhinaus stets nur dieselben Eingänge zulässig sind, bspw. ausschließlich A und B, zeigt Tabelle 13 (links) alle 2^5 möglichen Belegungen der funktionsbestimmenden Eingänge $X1, X2, X3, Y2$ und $Y3$ einer Zelle und die aus diesen Belegungen resultierenden Zellfunktionen. Bei der hier betrachteten 9 Bit Kodierung, werden 5 der insgesamt 9 Bits für die Bestimmung der

Zellfunktion verwendet, vgl.Abbildung 29 auf Seite 77. Dies ergibt 2^5 verschiedene Bitkombinationen und würde bedeuten, daß 32 verschiedene Funktionen von einer Zelle berechnet werden können. Weil eine Zelle des XC6216 jedoch nur die in Tabelle 7 gezeigten 24 verschiedene Funktionen berechnen kann, muß bei der gegebenen Kodierung Redundanz vorliegen. Dies bedarf einer genaueren Untersuchung.

Es finden weder schnelle Zellfunktionen, die sog. Fast-Gates, noch die zwei Multiplexer $M2\_1(S, A, B)$ und $M2\_1B2(S, A, B)$ Verwendung. Während die schnellen Versionen der Zellfunktionen ausdrücklich unterbunden wurden, indem die Eingaben Q und $\bar{Q}$ für Y2 und Y3 nicht zulässig sind, resultiert die Nichtverwendung der Multiplexer aus der Beschränkung des Neighbour-Routings auf E_{out} und S_{out}. Falls nämlich $S = A$ oder $S = B$ gilt, was bei der 9 Bit Kodierung stets der Fall ist, da nur zwei verschiedene Eingänge möglich sind, so gilt $M2\_1(S, A, B) \equiv A.B$ und $M2\_1B2(S, A, B) \equiv \overline{A.B}$. Damit ergibt sich, daß von den ursprünglich 24 Funktionen die eine Logikzelle berechnen kann, tatsächlich nur 14 verschiedene zur Verfügung stehen und sich die daraus resultierende Redundanz bei 2^5 Tabelleneinträgen in der mehrfachen Verwendung von ein und derselben Funktion niederschlägt. Das bedeutet: verschiedene Belegungen von X1, X2, X3, Y2 und Y3 werden auf *dieselbe Zellfunktion* abgebildet. Anhand von Tabelle 13 (rechts) läßt sich diese Mehrfachverwendung leicht ermitteln und es zeigt sich, daß die 14 Funktionen mit den in der Tabelle zur Linken gezeigten Häufigkeiten auftreten. Sowohl *Buffer* als auch *Inverter* tauchen jeweils viermal, alle anderen Funktionen zweimal oder sogar nur einmal als Instanz einer Belegung auftauchen. Die Wahrscheinlichkeit, daß nach Mutation eines Individuums *Buffer* oder *Inverter* als Zellfunktionen entstehen, ist damit weitaus größer als dies bspw. für die Multiplexer der Fall ist.

Funktion	Häufigkeit
BufS	2
BufE	2
InvS	2
InvE	2
0	2
1	2
$S + E$	2
$\overline{S + E}$	2
$S.E$	2
$\overline{S.E}$	2
$S \oplus E$	2
$\overline{S \oplus E}$	2
$\bar{S} + E$	1
$\bar{E} + S$	1
$\bar{S}.E$	1
$\bar{E}.S$	1
M2_1A1A	1
M2_1B1B	1
M2_1A1B	1
M2_1B1A	1

Da sich eine solche Gewichtung (Bias) zugunsten bestimmter Zellfunktionen möglicherweise negativ auf den evolutionären Prozeß auswirken könnte wird im nächsten Schritt eine Kodierung vorgestellt, bei der nach Mutation eines Individuums jede Zellfunktion gleichwahrscheinlich ist.

Tabelle 13 Links: Zellfunktionen bei der Beschränkung auf zwei Eingänge A und B. Rechts: Die 2^5 möglichen Zellfunktionen bei der 9 Bit Kodierung.

X1	X2	X3	Y2	Y3	Funktion
A	A	A	$\overline{X2}$	$\overline{X3}$	*BufA*
A	A	A	$\overline{X2}$	X3	1
A	A	A	X2	$\overline{X3}$	0
A	A	A	X2	X3	*InvA*
A	A	B	$\overline{X2}$	$\overline{X3}$	$A + B$
A	A	B	$\overline{X2}$	X3	M2_1B1B
A	A	B	X2	$\overline{X3}$	$\bar{A}.B$
A	A	B	X2	X3	$\overline{A + B}$
A	B	A	$\overline{X2}$	$\overline{X3}$	A.B
A	B	A	$\overline{X2}$	X3	$\bar{A} + B$
A	B	A	X2	$\overline{X3}$	M2_1B1A
A	B	A	X2	X3	$\overline{A.B}$
A	B	B	$\overline{X2}$	$\overline{X3}$	*BufB*
A	B	B	$\overline{X2}$	X3	$\overline{A \oplus B}$
A	B	B	X2	$\overline{X3}$	$A \oplus B$
A	B	B	X2	X3	*InvB*
B	A	A	$\overline{X2}$	$\overline{X3}$	*BufA*
B	A	A	$\overline{X2}$	X3	$\overline{A \oplus B}$
B	A	A	X2	$\overline{X3}$	$A \oplus B$
B	A	A	X2	X3	*InvA*
B	A	B	$\overline{X2}$	$\overline{X3}$	A.B
B	A	B	$\overline{X2}$	X3	$\bar{A} + B$
B	A	B	X2	$\overline{X3}$	M2_1B1A
B	A	B	X2	X3	$\overline{A.B}$
B	B	A	$\overline{X2}$	$\overline{X3}$	$A + B$
B	B	A	$\overline{X2}$	X3	M2_1B1B
B	B	A	X2	$\overline{X3}$	$\bar{A}.B$
B	B	A	X2	X3	$\overline{A + B}$
B	B	B	$\overline{X2}$	$\overline{X3}$	*BufB*
B	B	B	$\overline{X2}$	X3	1
B	B	B	X2	$\overline{X3}$	0
B	B	B	X2	X3	*InvB*

X1[0]	X2[1]	X3[0]	Y2[1]	Y3[0]	Funktion
0	0	0	0	0	*InvS*
0	0	0	0	1	0
0	0	0	1	0	1
0	0	0	1	1	*BufS*
0	0	1	0	0	$\overline{S + E}$
0	0	1	0	1	$\bar{S}.E$
0	0	1	1	0	M2_1E1E
0	0	1	1	1	$S + E$
0	1	0	0	0	$\overline{S.E}$
0	1	0	0	1	M2_1E1S
0	1	0	1	0	$\bar{S} + E$
0	1	0	1	1	S.E
0	1	1	0	0	*InvE*
0	1	1	0	1	$S \oplus E$
0	1	1	1	0	$\overline{S \oplus E}$
0	1	1	1	1	*BufE*
1	0	0	0	0	*InvS*
1	0	0	0	1	$S \oplus E$
1	0	0	1	0	$\overline{S \oplus E}$
1	0	0	1	1	*BufE*
1	0	1	0	0	$\overline{S.E}$
1	0	1	0	1	M2_1E1S
1	0	1	1	0	$\bar{E} + S$
1	0	1	1	1	S.E
1	1	0	0	0	$\overline{S + E}$
1	1	0	0	1	$\bar{E}.S$
1	1	0	1	0	M2_1E1E
1	1	0	1	1	$S + E$
1	1	1	0	0	*InvE*
1	1	1	0	1	0
1	1	1	1	0	1
1	1	1	1	1	*BufE*

5.4 Die Integer-Kodierung

Es wird eine Kodierung vorgestellt, die die gleichwahrscheinliche Verteilung von Zellfunktionen nach Initialisierung der Startpopulation bzw. nach der Mutation eines Individuums erreicht. Diese Kodierung wird aufgrund ihrer Implementierung **Integer-Kodierung** genannt.

Im Vergleich zu den Ergebnissen mit der 9 Bit Kodierung läßt sich feststellen, daß diese Kodierung für Probleme geringerer Komplexität (bis XOR_5) schneller entsprechende Lösungen findet. Für das XOR_6 Problem liegen beide Kodierungen gleichauf, während die Integer-Kodierung für die XOR_7- und XOR_8-Probleme der 9 Bit Kodierung deutlich unterlegen ist.

5.4.1 Beschreibung der Kodierung

Jeder der 14 verschiedenen Funktionen aus Tabelle 13 (rechts) wird eine Zahl aus $\{1, \ldots, 14\}$ zugewiesen, welche Funktion welche Zahl erhält ist dabei beliebig. Im Hinblick auf die Beschränkungen des Neighbour Routing bei der 9 Bit Kodierung, existieren nur 9 verschiedene Möglichkeiten, wie die Signale innerhalb einer Zelle weitergeleitet werden können. Tabelle 14 (rechts) zeigt die Belegungen der E_{out} und S_{out} Bits unter Berücksichtigung der verbotenen Bitkombinationen. Die grau dargestellten Bereiche zeigen die auf konstante Werte gesetzten Teile des vollständigen Neighbour Routing Konfigurationswortes, vgl. Abbildung 29.

In Abbildung 32 sind die 9 möglichen Routing-Varianten noch einmal graphisch dargestellt. Dabei fällt auf, daß in 5 Fällen das Ergebnis der Funktionseinheit an mindestens einen der Ausgänge E_{out} oder S_{out} weitergeleitet wird. Die restlichen 4 Fälle bilden Routing-Zellen, welche ausschließlich dem Weiterleiten eingehender Signale dienen. Die Funktionseinheit solcher Zellen wird nicht benötigt.

Tabelle 14 Bei der 9 Bit Kodierung werden vom Neighbour Routing Wort die 4 Bit für E_{out} und S_{out} verwendet. Jede mögliche Belegung unter Berücksichtigung der verbotenen Bitkombinationen, führt zur Weiterleitung der in den Spalten $E_{out} =$ und $S_{out} =$ angegebenen Signale.

Nr.	N_{out}	E_{out}	W_{out}	S_{out}	$E_{out} =$	$S_{out} =$
1	00	00	00	00	F	F
2	00	00	00	01	F	E_{in}
3	00	00	00	11	F	S_{in}
4	00	10	00	00	E_{in}	F
5	00	10	00	01	E_{in}	E_{in}
6	00	10	00	11	E_{in}	S_{in}
7	00	11	00	00	S_{in}	F
8	00	11	00	01	S_{in}	E_{in}
9	00	11	00	11	S_{in}	S_{in}

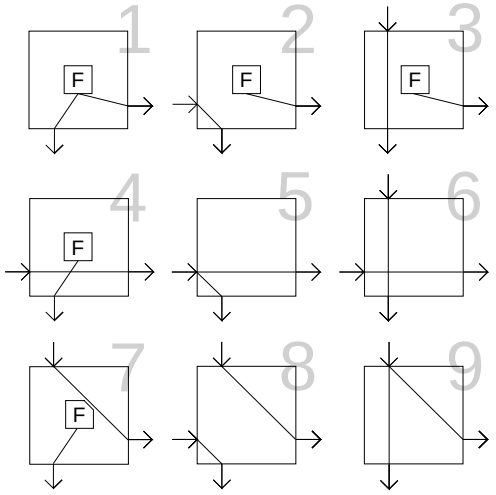


Abbildung 32 Die möglichen Konfigurationen einer Zelle bei allen Bitkombinationen, die mit den in der 9 Bit Kodierung verwendeten 4 Bits des Neighbour Routing Konfigurationswortes unter Berücksichtigung der verbotenen Zustände erzielt werden können, vgl. Abbildung 29. Die Zahlen an den einzelnen Zellen entsprechen denen in Tabelle 14.

Zusammengenommen ergeben sich aus 14 verschiedenen Funktionen und 9 Routing-Varianten 126 Kombinationen, wie mit der 9 Bit Kodierung eine Zelle konfiguriert werden kann. Dabei ist zu beachten, daß nicht jede dieser Kombinationen wirklich einen Sinn ergibt: Angenommen eine Zelle berechnet die Funktion $E + S$, das Neighbour Routing wurde aber so konfiguriert, daß $E_{out} = E_{in}$ und $S_{out} = S_{in}$ gilt (Fall 6 in Tabelle 14). Dann wird das Ergebnis der Berechnung von $E + S$ niemals an eine der Nachbarzellen weitergeleitet werden, da diese Zelle ausschließlich als Routing-Zelle dient. Andererseits besteht natürlich die Möglichkeit, daß eine nachfolgende Mutation genau dieser Zelle bspw. die Konfiguration $E_{out} = F$ zur Folge hätte und dies gerade der letzte benötigte Schritt bei der Auffindung einer Lösung war. Somit ist es (im Moment) nicht wünschenswert ein derartiges Verhalten zu unterbinden.

In Anbetracht der 126 verschiedenen Kombinationen von Zellfunktion und Routing, werden nicht länger Genotypen aus Bitstrings betrachtet, sondern jedes Individuum besteht aus einer Abfolge von n_{cells} Integer-Werten aus der Menge $\{1, \ldots, (126 \cdot 2)\}$. Jeder Integer-Wert repräsentiert die Konfiguration einer Zelle. Mithilfe einer „Lookup-Tabelle“, wird jeder Wert einer entsprechenden Bitkombinationen zur Konfiguration einer Zelle zugeordnet. Einige der Funktionen tauchen in zwei Varianten auf, so bspw. Buf E und Buf S. Andere Funktionen sind nicht symmetrisch; z. B. berechnet $\bar{E}.S$ etwas anderes als $\bar{S}.E$. Es muß also bei der Übertragung eines Individuums in einen Schaltkreis entschieden werden, welche der beiden Varianten einer Funktion gewählt werden soll. Diese Entscheidung wird ebenfalls auf der

Grundlage der Integer-Werte $v \in \{1, \ldots, (126 \cdot 2)\}$ getroffen. Falls nämlich $v \leq 126$ ist, so wird die entsprechende Funktion aus der Lookup-Tabelle gewählt. Für den Fall $v > 126$ ergibt $v \mod 126$ den Index in der Lookup-Tabelle und damit die Funktion, allerdings werden, da $v > 126$ ist, die Eingabesignale vertauscht.

5.4.2 Simulationsergebnisse

Die Versuche wurden mit denselben Parametern wie bei der 9 Bit Kodierung durchgeführt, mit einer Ausnahme beim Mutations-Operator: Weil die Individuen nicht mehr aus Bits bestehen, die mutiert werden könnten, muß eine neue Mutationsrate festgelegt werden. Mutationen bei der Integer-Kodierung werden durchgeführt, indem zufällige Zellen (= Integer-Werte) des Individuums ausgewählt und durch eine neue Zahl aus $\{1, \ldots, (126 \cdot 2)\}$ ersetzt werden. Die Wahrscheinlichkeit, daß eine Zelle mutiert wird, bestimmt die Mutatitionsrate p_m. Durch das Austauschen kompletter Zellen, bewirken Mutationen bei der Integer-Kodierung wesentlich mehr Änderungen, als wenn bei der 9 Bit Kodierung einzelne Bits invertiert werden. Für die Integer-Kodierung wird daher eine Mutationsrate bestimmt, die der Wahrscheinlichkeit entspricht, mit der *alle Bits einer Zelle* bei der 9 Bit Kodierung mutiert werden können. Bei der 9 Bit Kodierung wurde die Mutationsrate stets auf $1/l$ mit $l = n_{\text{cells}} \cdot n_{\text{bitsPerCell}}$ Länge des Bitstrings festgelegt. Dies ist die Wahrscheinlichkeit, daß *ein* Bit mutiert wird. Dann ist $1 - 1/l$ die Wahrscheinlichkeit, daß *kein* Bit mutiert wird und $(1-1/l)^{n_{\text{bitsPerCell}}}$ die Wahrscheinlichkeit, daß $n_{\text{bitsPerCell}}$ Bits keiner Mutation unterliegen. Mit einer Wahrscheinlichkeit von $1-\left((1-1/l)^{n_{\text{bitsPerCell}}}\right)$ werden dann alle Bits einer Zelle mutiert. Dies entspricht der gesuchten Mutationsrate p_m für die Integer-Kodierung.

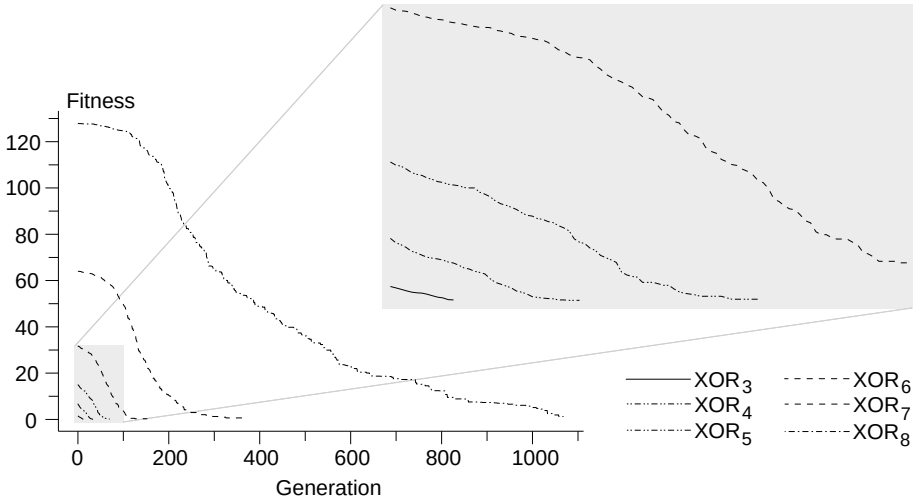


Abbildung 33 Fitneßverlauf des jeweils besten Individuums einer Population gemittelt über 50 Läufe. Es werden die Kurven für die n-Parity Probleme mit $n = 3, \ldots, 8$ gezeigt. Der schwer zu erkennende Bereich von Generation 0 bis 100 ist vergrößert dargestellt.

Tabelle 15 Durchschnittliche Anzahl von Generationen $\bar{g}$ für jeden mit der Integer-Kodierung durchgeführten evolutionären Prozeß für eines der n-Parity Probleme mit $n = 3, \ldots, 8$ gemittelt über jeweils 50 Läufe. $m_{\bar{g}}$ gibt den Standardfehler der jeweiligen Messung an.

Problem	$\bar{g}$	$m_{\bar{g}}$
XOR_3	4.46	0.83
XOR_4	20.12	2.96
XOR_5	43.14	6.21
XOR_6	81.74	11.85
XOR_7	171.06	25.24
XOR_8	438.98	70.52

In Abbildung 33 sind alle Fitneßverläufe für die Probleme $XOR_{3,\ldots,8}$ dargestellt. Es wird die Fitneß des jeweils besten Individuums einer Population gemittelt über jeweils 50 Läufe gezeigt; der Ausschnitt auf der rechten Seite stellt eine Vergrößerung des schwer erkennbaren Bereiches von Generation 0 bis 100 dar. Tabelle 15 zeigt die mit dieser Kodierung ermittelte durschnittliche Anzahl an Generationen σ (gemittelt über 50 Läufe), die für die Evolvierung eines Schaltkreises in der entsprechenden Problemklasse benötigt wurden.

Beim Vergleich der Tabellen 12 und 15 läßt sich feststellen, daß durch die gleichwahrscheinliche Auswahl von Zellfunktionen nach Mutation die Probleme $XOR_{3,\ldots,5}$ im Mittel schneller gelöst werden können. Bei dem XOR_6 Problem verhalten sich die beiden Kodierungen sehr ähnlich. Für die Probleme XOR_7 und XOR_8 gilt jedoch, daß die Integer-Kodierung, trotz des hohen Standardfehlers insbesondere bei XOR_8, deutlich schlechter als die 9 Bit Kodierung abschneidet. Das XOR_9 Problem kann auch mit der Integer-Kodierung nicht gelöst werden.

Es wird vermutet, daß das bessere Konvergenzhalten der 9 Bit Kodierung bei den Problemen XOR_7 und XOR_8 nicht nur auf die Tatsache zurückzuführen ist, daß *irgendeine* Gewichtung bzgl. bestimmter Zellfunktionen vorliegt, sondern daß gerade $A \oplus B$ und $\overline{A \oplus B}$ zu den häufiger gewählten Funktionen gehören, vgl. Tabelle auf Seite 82. Mit zunehmender Komplexität des Problems scheint also die bevorzugte Verwendung von XOR_2-Bausteinen als Building Blocks für den evolutionären Prozeß hilfreich zu sein, während dies für die einfacheren Probleme aber nicht gilt.

5.5 Grenzen der 9 Bit Kodierung

Aufgrund der guten Ergebnisse bei Verwendung der 9 Bit Kodierung im Hinblick auf die Evolvierung von n-Parity Schaltkreise für große n werden einige weitere boolesche Probleme betrachtet, um die Grenzen der 9 Bit Kodierung festzustellen. Zum einen wird die Evolvierung eines **2-Bit Halbaddierers** untersucht, zum anderen die einiger $n : m$-**Multiplexer** unterschiedlicher Komplexität.

5.5.1 2-Bit Halbaddierer

Ein n-Bit Halbaddierer (engl. Halfadder) berechnet aus zwei Binärzahlen $x = (x_{n-1}, \ldots, x_0)$ und $y = (y_{n-1}, \ldots, y_0)$ die Summe z und den Übertrag c (das sog. Carry-Bit). Es wird versucht einen Halbaddierer für $n = 2$ zu evolvieren; die Tabelle zur Linken zeigt eine entsprechende Wertetabelle. Die drei Ergebnisbits bestehend aus c, z_1 und z_0 können als Integer-Wert im Bereich $\{0, \ldots, 6\}$ interpretiert wird. Darauf aufbauend wird eine Fitneßfunktion implementiert, die nicht mehr Bit-Ergebnisvektoren unter Zuhilfenahme der Hamming-Distanz vergleicht, sondern die nun den Abstand zweier Vektoren mit der Euklidischen Distanz berechnet, vgl. Anhang E. Werden also die drei Ergebnisbits der linken Tabelle als Dezimalwerte interpretiert und von oben nach unten gelesen, so erhalten wir den Lösungsvektor $\bar{o} = (0,1,2,3,1,2,3,4,2,3,4,5,3,4,5,6)$. Ein Individuum $p_i \in P(t)$, das nach Expression und Anlegen aller 2^4 Eingabebits den Ergebnisvektor $\bar{e}_i = (0,1,2,2,2,4,6,6,6,1,0,3,3,0,0,0)$ liefert, wird mit der Euklidischen Distanz schlechter bewertet als bspw. $p_j \in P(t)$ mit $\bar{e}_j = (0,1,2,3,1,2,3,4,6,6,5,5,3,4,5,6)$, da $d_e(\bar{o}, \bar{e}_i) = 11.66 > d_e(\bar{o}, \bar{e}_j) = 5.09$ ist.

x_1	x_0	y_1	y_0	c	z_1	z_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Wie sich zeigt, ist das Evolvieren eines vergleichsweise einfachen 2-Bit Halbaddierers mit der gegebenen Kodierung auf einer Chipfläche von 4×4 Zellen nicht möglich. Es wird vermutet, daß hier, wie schon bei dem Versuch der Evolvierung von n-Parity schaltkreisen ohne XOR_2-Bausteine (vgl. Abschnitt 5.3.4), die zu kleine Anzahl an verfügbaren Zellen zusammen mit der Beschränkung bei der Weiterleitung von Signalen, das Hauptproblem darstellt. Zitiert nach Wegener (1989) konnte Redkin zeigen, „daß kein Additionsschaltkreis mit weniger als $5n - 3$ Bausteinen auskommt“ (Seite 76). Für den hier betrachteten 2-Bit Halbaddierer werden also mindestens sieben Logik-Zellen (mit Funktionen aus der Basis U) benötigt. Weil aber maximal 16 Zellen zur Verfügung stehen und davon mindestens sechs, nach der in Abschnitt 5.3.4 geäußerten Theorie, möglicherweise seltener als alle anderen Zellen verwendet werden, ist die Zahl der wirklich verfügbaren Zellen schon nahe dem absoluten Minimum für einen Additionsschaltkreis. Es handelt sich also um ein Problem, für dessen Lösung wahrscheinlich mehr Zellen zur Verfügung stehen müßten.

5.5.2 $n:m$-Multiplexer

Ein $n:m$-Multiplexer (Selektor) leitet, in Abhängigkeit von k Steuerleitungen, m seiner n Datenleitungen als Ausgabe weiter. In Tabelle 16 sind die Wahrheitstabellen dreier Multiplexer gezeigt. Auf der linken Seite die eines einfachen 2:1-Multiplexers, bei dem x sowohl Steuer- als auch Datenleitung ist. In der Mitte ist derselbe Multiplexer dargestellt, allerdings besitzt er einen separaten Steuereingang s. Zur Rechten ist ein aufwendigerer 3:2-Multiplexer mit zwei Steuerleitungen s_1 und s_2 zu sehen.

Der evolutionäre Entwurf von Schaltkreisen für die beiden 2:1-Multiplexer aus Tabelle 16 ist mit der 9 Bit Kodierung erfolgreich. Insbesondere ist dies auf einer, verglichen mit dem n-Parity Problem, sehr kleinen Chipfläche möglich. Der erste Multiplexer, vgl. Tabelle 16 (links), benötigt nur 2 Zellen. Dies ist jedoch nicht erstaunlich, da zur Lösung nur die Funktion $x \wedge \bar{y}$ berechnet werden muß. Auch für den mittleren Multiplexer sieht die zu berechnende Funktion nicht wesentlich komplizierter aus, hier lautet die Lösung $\bar{s}x \vee sy$; für diesen Multiplexer genügen 6 Zellen des FPGA.

Für den dritten Multiplexer hingegen läßt sich mit der 9 Bit Kodierung kein Schaltkreis auf Chipflächen von 25, 30 und 35 Zellen finden. Auch dieses ist im Rahmen der bisher gemachten Beobachtungen nicht ungewöhnlich, da die vom Schaltkreis zu berechnenden Funktionen für die Ausgabeleitungen v und w kompliziert sind:

$$\begin{aligned} v &= s_1\bar{s_2}\bar{x}\bar{y}\bar{z} \vee s_1\bar{s_2}yz \vee \bar{s_1}x \vee xy \vee s_2x \\ w &= \bar{s_1}\bar{s_2}y \vee s_1s_2\bar{x} \vee s_1\bar{s_2}z \vee \bar{s_1}s_2z. \end{aligned}$$

Die Minimalpolynome für v und w wurden durch Minimierung der DNF mit dem Quine/McCluskey-Algorithmus berechnet, siehe Abschnitt 2.2.3 und Wegener (1989).

In Anbetracht der Komplexität von v und w könnte durch Vergrößerung der Chipfläche möglicherweise dennoch eine Lösung mit der 9 Bit Kodierung für diesen Multiplexer gefunden werden.

Tabelle 16 Links: Ein $2:1$-Multiplexer; x ist gleichzeitig Steuer- und Datenleitung. Mitte: Ein 2:1-Multiplexer ähnlich wie links, jedoch mit separater Steuerleitung s; Es wird x ausgegeben falls $s = 0$ gilt, sonst y. Rechts: Ein 3:2-Multiplexer mit zwei Steuerleitung s_1 und s_2 und drei Datenleitungen x, y und z.

x	y	$2:1$
0	0	0
0	1	0
1	0	0
1	1	1

s	x	y	$3:1$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

s_1	s_2	v	w
0	0	x	y
0	1	x	z
1	0	y	z
1	1	x	$\bar{x}$

5.6 Diskussion der Ergebnisse

Die in Abschnitt 5.1 vorgestellte Repräsentation eines Genotyps mit 18 Bit je Zelle hat sich im Hinblick auf die Verwendung für den evolutionären Entwurf digitaler Schaltkreise als wenig brauchbar erwiesen. Bei dieser Kodierung wird der Evolution das größtmögliche Maß an Freiheiten bzgl. der Verbindungen benachbarter Zellen gewährt, indem Signaleinspeisung aus und -weiterleitung in jede der vier Richtungen N_{out}, E_{out}, S_{out} und W_{out} zulässig ist. Diese Freiheit führt jedoch zu Zyklen innerhalb von Schaltkreisen, so daß derartige Schaltkwerke als vermeintliche Lösung für ein gegebenes Problem, bspw. 3-Parity, vom Genetischen Algorithmus präsentiert werden. Dieses Problem konnte zwar durch mehrfaches Auswerten der Testmuster gelöst werden und auch Thompson (1998) konnte zeigen, daß ein ein solcher „unconstrained approach" für die Evolvierung eines Frequency Discriminators, der die physikalischen Eigenheiten der auf dem FPGA angesiedelten Elektronik nutzt, erfolgreich war. Jedoch empfiehlt sich für den evolutionären Entwurf *kombinatorischer Schaltkreise* die ausschließliche Verwendung vorwärtsgerichteter Schaltkreise. Miller und Thomson (1998) gehen noch einen Schritt weiter, indem sie die Verwendung vorwärtsgerichteter Schaltkreise sogar als „vital" bezeichnen.

Es wurde beobachtet, vgl. Anhang D, daß bei 50 evolutionären Läufen zur Evolvierung von 3-Parity Schaltkreisen, in 90% der Fälle ein Anstieg der Standardabweichung vor dem Auffinden einer Lösung vorliegt. Dies läßt auf eine Zunahme der Diversität in der Population schließen. Weiterhin ist festzustellen, daß Lösungen meist „plötzlich" nach langen Phasen gleichbleibender Fitneßwerte (des besten Individuums einer Population) gefunden werden. Im Hinblick auf die Untersuchungen von Harvey (1992a) bzgl. Species Adaption Genetic Algorithms (SAGA) und dem Fitneßverlauf des jeweils besten Individuums einer Population, vgl. Anhang D, wird vermutet, daß der GA nach einigen anfänglichen Fitneßverbesserungen vorzeitig zu einem lokalen Minimum konvergiert; dies würde der Plateauphase der Fitneßverläufe entsprechen. In diesem Zustand weisen alle Individuen einer Population eine gewisse genotypische Homogenität auf. Es wird angenommen, daß ein bestes Individuum als sog. Master Sequenz (oder Wildtyp) an der Spitze des lokalen Optimums (eines Berges) sitzt, während die anderen Mitglieder der Population bzgl. ihrer Hamming-Distanz zumindest nahe bei der Master Sequenz zu finden sind. Mutationen dieser Mitglieder wird sie eher von der Master Sequenz entfernen, als sie anzunähern. Der diesem Druck zu kontinuierlicher Verschlechterung der Mitglieder einer Population entgegensetzte Vorgang stellt die Selektion dar, die vorzugsweise den Wildtyp und seine nächsten Nachbarn reproduziert. Idealerweise herrscht ein gewisses Gleichgewicht zwischen Mutationen und Selektion, das aber trotzdem erlaubt Außenseiter auf

dem Bergkamm (engl. ridge) zu erreichen, die eine höhere Hamming-Distanz zur Master Sequenz aufweisen und möglicherweise die Flucht aus einem lokalen Minimum erlauben: „If any such outliers reach a second hill that climbs away from the ridge, then parts of the population can climb this hill. Depending on the difference in fitness and the spread of the population, it will either move *en masse* to the new hill as a better local optimum, or share itself across both of them“ (Harvey 1992a, Seite 4). Bei den zu beobachtenden Fitneßverläufen scheint ein solches oder zumindest ähnliches Verhalten vorzuliegen, da die geringe Standardabweichung auf eine geringe Diversität in der Population schließen läßt. Ab einem bestimmten Zeitpunkt wird durch genügend Mutationen (und evtl. Rekombinationen) ein Berg erreicht, der einen Weg zu höherer Fitneß darstellt, mithin zum globalen Optimum. Dies ist der Punkt, an dem die Diversität in der Population zunimmt.

Als Ergebnis der Untersuchung verschiedener Rekombinations-Operatoren läßt sich festhalten, daß die Rekombination beim evolutionären Entwurf digitaler Schaltkreise eher hinderlich ist. Dies gilt sowohl für den Standard z-Punkt Crossover, bei dem die z Bruchstellen zufällig bestimmt werden, als auch für die auf die Repräsentation des Genotyps angepaßten Rekombinations-Operatoren c_{NNF} und c_{cells}, die nur an funktionell zusammengehörigen Teilen eines Individuums Bruchstellen erlauben. Dieses Ergebnis stellt einen Widerspruch zur gängigen Meinung über die Anwendung von Rekombination in Genetischen Algorithmen dar – generell wird dieser nämlich als Schwerpunkt-Operator für GAs angehsehen. Im Zusammenhang mit der Betrachtung des Species Adaption Genetic Algorithm (SAGA) von Harvey (1992b) stellte jedoch auch Thompson (1996) fest, daß für den evolutionären Schaltkreisentwurf andere Prioritäten gelten: Hier kommt dem Crossover-Operator im Gegensatz zum Mutations-Operator eine untergeordnete Bedeutung zu. Harvey (1992a) stellt darüber hinaus fest: „Recently in the GA community there has been some discussion of the surprising success (in some circumstances) of what has come to be called *Naive Evolution*; i.e., mutation only, contrary to normal GA folk-lore which emphasises the significance of crossover“ (Seite 5). Dies wird auch durch eine Untersuchung von Vassilev et al. (1999) belegt, bei der die Fitneßlandschaft eines auf einer Chipfläche von 4×4 Zellen evolvierten 2-Bit Multiplizierers analysiert wird. Dabei ergaben die Messungen der Autokorrelation von Zeitreihen bei der zufälligen Wanderung (Random Walk) auf der Mutations- und Rekombinations-Landschaft unterschiedliche Ergebnisse: Die Korrelationen der Crossover-Landschaften sind gering, während jene für den Mutations-Operator relativ hoch sind. Vassilev et al. (1999) folgern daraus, daß die Fitneßlandschaft des Rekombinations-Operators zerklüftet ist und diese Landschaften demzufolge schwierig für die evolutionäre Suche sind: „It agress with Miller *et al.* (1997) who suggested that

the crossover operator might be dropped in order to improve search“ (Seite 1303). Diese Resultate stimmen somit mit den in dieser Arbeit gewonnenen Erkenntnissen über den Nutzen des Rekombinations-Operators überein. Im Gegensatz zum Rekombinations-Operator wird der Mutations-Operator generell als „Background Operator“ im Rahmen Genetischer Algorithmen angesehen. Jüngeren Erkenntnissen zufolge wird diesem jedoch eine zunehmend größere Aufmerksamkeit geschenkt: „Originally mutation was only of small importance within GAs. [. . .] But recent studies have shown, that the importance of mutation should not be underrated“ (Droste und Wiesmann 1998, Seite 10).

„The mutation operator of Genetic Algorithms was introduced by Holland as a 'background operator' that occasionally changes single bits of individuals by inverting them“ (Bäck 1996, Seite 113).

Bei der Untersuchung des Fitneß-Gain, siehe hierzu auch Igel und Chellapilla (1999), nach Anwendung entweder des Crossover- oder des Mutations-Operators stellte sich heraus, daß in etwa gleich vielen Fällen bei beiden Operatoren ein Fitneßzuwachs zu verzeichnen ist. Dies ist ein erstaunliches Ergebnis, da der evolutinäre Prozeß ohne Verwendung eines Rekombinations-Operators deutlich schneller zu einer Lösung konvergiert. Das Maß an Fitneßzuwachs nach Anwendung eines der genetischen Operatoren kann somit nicht allein entscheidend für die Konvergenzgeschwindigkeit des Genetischen Algorithmus sein. Neben der o. g. Ergebnisse bzgl. der zerklüfteten Fitneßlandschaft von Vassilev et al. (1999) könnte ein weiterer Grund für dieses Verhalten sein, daß durch die Anwendung des z-Punkt Crossover zu viele Störungen in die Nachkommen eingebracht werden, so daß bereits evolvierte „gute“ Strukturen immer wieder zerstört werden. Das gilt anscheinend ebenso für die speziellen Crossover-Operatoren c_{NNF} und c_{cells}, die in der Hoffnung implementiert wurden, daß durch den alleinigen Austausch funktioneller Einheiten eines Individuums gerade die „guten“ Strukturen an die Nachkommen weitergegeben werden.

Durch die Untersuchung verschiedener Selektionsmechanismen konnte die Vermutung belegt werden, daß eine höhere Diversität in der Population zu besseren Ergebnissen bzgl. des Konvergenzverhaltens und der Konvergenzgeschwindigkeit des Genetischen Algorithmus führt. Durch den stochastischen Anteil bei der q-Tournament Selektion begründet sich ihr deutlicher Einfluß auf den GA bspw. im Vergleich zur fitneßproportionalen Selektion. Wie Goldberg et al. (1990) zitiert nach Harvey (1992a) zeigen konnten, ist die q-Tournament Selektion äquivalent zu einem rangbasierten Selektions-Schema, was eine Erklärung für das ähnlich gute Abschneiden des q-Tournament Verfahrens mit Whitley's Linear Ranking ist. Auch zeigt sich, daß von den unterschiedlichen Forschergruppen im Bereich der EHW gehäuft die Tournament-Selektion eingesetzt wird, so bspw. von Thompson (1996), Miller et al. (1998) oder Vassilev et al. (1999). Es zeigt sich also, daß die Wahl des Selektionsmechanismus zwar maßgeblich die Geschwindigkeit mit der eine Lösung gefunden wird beeinflußt, sich jedoch nicht positiv auf die Komplexität der betrachteten Probleme auswirkt.

Dafür ist die Erhöhung der Komplexität durch die Verfeinerung der gewählten Genotyp/Phänotyp Abbildung möglich. Im Gegensatz zur 18 Bit Kodierung, mit der lediglich das 3-Parity Problem gelöst werden konnte, ließ die 16 Bit Kodierung die Evolvierung von Schaltkreisen für das 4-Parity Problem zu. Dies wurde erreicht, indem Zyklen verboten wurden. Erst die 9 Bit Kodierung, bei der Signale im Schaltkreis nur noch in die Richtungen S_{out} und E_{out} weitergeleitet werden konnten, ermöglichte den evolutionäre Entwurf von Schaltkreisen bis hin zum 8-Parity Problem. Dabei ist es sehr wahrscheinlich, daß die Lösbarkeit der betrachteten Probleme nicht nur von der Art und Weise des Neighbour Routings abhängt, sondern natürlich auch von der Größe des Suchraums. Bezogen auf eine Chipfläche von 3×3 Zellen, die stets für das XOR_3-Problem verwendet wurden, beträgt die Suchraumgröße mit der 18 Bit Kodierung $\approx 10^{48}$, mit der 16 Bit Kodierung $\approx 10^{46}$ und mit der 9 Bit Kodierung nur noch $\approx 10^{24}$. Schon ab dem XOR_4-Problem gilt jedoch auch für die 9 Bit Kodierung bereits $\approx 10^{46}$ und für das XOR_8-Problem sogar $\approx 10^{173}$. Trotz der immensen Größe des Suchraums beim XOR_8-Problem lassen sich mit der 9 Bit Kodierung aber dennoch entsprechende Schaltkreise als Lösung finden. Die beiden Einheiten Größe des Suchraums und Kodierung der Genotyp/Phänotyp Abbildung können somit nicht losgelöst voneinander betrachtet werden.

Die Evolvierung von n-Parity Schaltkreisen *ohne* XOR_2-Bausteine stellte sich als unlösbar heraus. Wie bereits in Abschnitt 5.3.4 diskutiert liegt dies möglicherweise an der eingeschränkten Anzahl von Zellen, die für den evolutionären Prozeß zur Verfügung stehen. Durch die gegebene Kodierung, insbesondere der Art und Weise wie Signale weitergeleitet werden, ist es sehr wahrscheinlich, daß bestimmte Teile der zur Verfügung stehenden Chipfläche weniger genutzt werden als andere, so daß es zur weiteren Verminderung von Zellen kommt, die für den Schaltkreis verwendet werden können.

Aufgrund der Tatsache, daß bei der 9 Bit Kodierung Redundanz in der Repräsentation des Genotyps festgestellt wurde, die zu einer bevorzugten Auswahl bestimmter Zellfunktionen bei der Initialisierung der Startpopulation und nach Mutationen führte, konnte eine neue Kodierung auf der Basis von Integer-Werten implementiert werden, die dieses Verhalten nicht zeigt. Erstaunlicherweise zeigt diese Kodierung ein besseres Konvergenzverhalten für die weniger Komplexen n-Parity Probleme mit $n = 3, \ldots, 5$. Für $n = 6$ benötigen beide Kodierungen im Durchschnitt etwa gleich viele Generationen für die Evolvierung eines entsprechenden Schaltkreises, während die 9 Bit Kodierung für $n = 7$ und $n = 8$ deutlich besser abschneidet. Als direkte Folge der Gewichtung zugungsten bestimmter Zellfunktionen, fällt bei der 9 Bit Kodierung auf, daß zu den Funktionen deren Verwendung wahrscheinlicher ist die XOR_2-Bausteine zählen. Es scheint also, als ob sich deren be-

vorzugte Verwendung mit Zunahme der Komplexität des betrachteten Problems positiv auswirkt, während es im umgekehrten Fall genau anders herum ist: Je einfacher das Problem, desto besser scheint eine gleichmäßige Verteilung der zur Verfügung stehenden Zellfunktionen zu sein.

Bei der Untersuchung der Grenzen der 9 Bit Kodierung stellt sich heraus, daß der evolutionäre Entwurf von Schaltkreisen weder für einen 2-Bit Halbaddierer noch für einen aufwendigen Multiplexer möglich ist. Sehr wohl können jedoch einfache Multiplexer evolviert werden. In vergleichbaren Arbeiten bspw. von Vassilev, Miller, und Fogarty (1999) oder Miller und Thomson (1998), bei denen sehr spezifische Repräsentationen des Genotyps gewählt wurden, waren die Autoren in der Lage zumindest für 2-Bit Multiplizierer entsprechende Schaltkreise evolutionär zu finden. Dies läßt vermuten, daß die Evolvierung von Schaltkreisen auch für einen 2-Bit Halbaddierer prinzipiell möglich ist, die 9 Bit Kodierung für diese Aufgabe jedoch ungeeignet ist. Eine der Haupteinschränkungen dürfte die Art und Weise sein, in der Ausgabesignale weitergeleitet werden. Bei den o. g. Arbeiten wird das Weiterleiten von Signalen nicht nur in die Richtungen E_{out} und S_{out} zugelassen, sondern auch nach N_{out} – dies entspricht dem Routing wie es in der 18 Bit Kodierung praktiziert wird.

6 Zusammenfassung und Ausblick

Er sehnte sich nach einer schönen langen zähen Arbeit, die ihn in Gefangenschaft nahm.
– aus Dr. Schiwago (Pasternak)

Im Rahmen dieser Arbeit wurde ein Evolvable Hardware System beschrieben und implementiert, das den evolutionären Entwurf von digitalen Schaltkreisen ermöglicht. Durch das Zusammenspiel eines rekonfigurierbaren Mikrochips – ein XC6216 FPGA der Firma XILINX – und einer Variante der Evolutionären Algorithmen, war es möglich dieses System in der Klasse der Semi Online Evolution zu realisieren. Dadurch wurden die Grundlagen für die Analyse intrinsischer Hardware Evolution geschaffen, d. h. es konnten Experimente durchgeführt werden, in denen ein realer Chip zur Instantiierung von Schaltkreisen verwendet wurde.

Es wurden Versuche zur Evolvierung von Schaltkreisen durchgeführt, die in der Lage sind bestimmte boolesche Funktionen wie bspw. n-Parity, $n : m$-Multiplexer, etc. zu berechnen. In Übereinstimmung mit aktuellen Forschungsergebnissen konnte festgestellt werden, daß die Verwendung eines Rekombinations-Operators im Zusammenhang mit der Evolvierung kombinatorischer Schaltkreise nicht angebracht ist. Hingegen wurde gezeigt, daß dem Mutations-Operator eine größere Bedeutung beizumessen ist, als dies generell der Fall ist. Darüberhinaus ergab der Vergleich verschiedener Selektionsmechanismen, daß mit der Tournament-Selektion die besten Ergebnisse im Hinblick auf schnelle und zuverlässige Konvergenz des Genetischen Algorithmus erreicht wurden. Hinsichtlich der Lösbarkeit und Komplexität bestimmter boolescher Probleme wurde mit einer schrittweisen Verbesserung der Kodierung für die Genotyp/Phänotyp Abbildung deutlich gemacht, daß sich die Komplexität mit einer problemangepaßten Kodierung deutlich steigern läßt. Dieses Ergebnis deckt sich mit Erkenntnissen aktueller Forschungsergbnisse: „To evolve a circuit to perform a computational task a suitable genotype representation needs to be chosen [. .]. In the process of modeling circuit evolution this is probably the

most important stage which predetermines the sucess in solving this engineering problem“ (Vassilev et al. 1999, Seite 1299). In Anhang C sind die experimentellen Ergebnisse dieser Arbeit noch einmal tabellarisch dargestellt.

Es existiert eine Vielzahl an Möglichkeiten zur weiteren Anpassung der Genotyp/Phänotyp Abbildung an die gegebenen Probleme. So scheint eine einfache Vergrößerung der zur Verfügung stehenden Chipfläche für die in Abschnitt 5.5 vorgestellten Probleme 2-Bit Halbaddierer und Multiplexer, deren Lösbarkeit wahrscheinlicher werden – insbesondere im Hinblick auf die beschränkten Routing-Möglichkeiten der 9 Bit Kodierung. Als Erweiterung dieses Ansatzes wäre die Implementierung von Genotypen mit variabler Länge denkbar, bei denen die Evolution selbständig die Anzahl benötigter Zellen des FPGA herausfinden kann. Um den Einschränkungen, die sich aus dem begrenzten Routing-Potential der 9 Bit Kodierung ergibt, entgegenzuwirken, könnte eine Kodierung gefunden werden, die ein abwechselndes Weiterleiten von Signalen nach S_{out} und E_{out} oder N_{out} und E_{out} erlaubt. Auf diese Weise würden weder Zyklen auftreten, noch gäbe es einen festgelegten Fluß von Signalen. In einer vergleichbaren Arbeit zeigten Miller und Thomson (1998), daß bei einer Aufteilung der Zellen eines FPGA in Routing- und Logik-Zellen, die entweder in einem Schachbrettmuster oder zeilen- und spaltenweise abwechselnd angeordnet sind, die Evolvierung eines 2-Bit Multiplizierers möglich ist.

Während bei der Integer-Kodierung eine ähnliche Unterteilung in Routing- und Logikzellen im Hinblick auf die Repräsentation des Genotyps bereits festgestellt wurde, vgl. Abschnitt 5.4, wäre eine sinnvolle Erweiterung dieser Kodierung die Zuordnung einer bestimmten Wahrscheinlichkeit zu jedem der 9 Zelltypen, mit der dieser im Fall einer Mutation auftreten kann. Im zweiten Schritt könnten diese Wahrscheinlichkeiten mit in den evolutionären Prozeß aufgenommen werden, sodaß diese an das gegebene Problem automatisch adaptiert werden.

Im Hinblick auf die wünschenswerte Eigenschaft einer stark kausalen Genotyp/Phänotyp Abbildung zum Erhalt der Nachbarschaftsrelationen nach Mutation, ließe sich bezogen auf die kombinatorische Funktion, die eine Zelle berechnet, bspw. die 9 Bit Kodierung wie folgt anpassen. Es wird eine Tabelle von Übergangswahrscheinlichkeiten angelegt, in der vermerkt ist, mit welcher Wahrscheinlichkeit eine Logikfunktion in eine andere übergeht. Ziel ist es, daß diese Wahrscheinlichkeiten umso höher sind, je ähnlicher sich zwei Logikfunktionen bezogen auf das Ergebnis ihrer Berechnung sind. In der Tabelle zur Linken sind die Berechnungsergebnisse von 7 binären booleschen Funktionen dargestellt. Nach der Mutation eines Individuums soll nun kein Übergang einer Zellfunktion von bspw. $\wedge \rightarrow \vee$ möglich sein, wie es derzeit bei der 9 Bit Kodierung der Fall ist, sondern es sollen ausschließlich „weiche“ Übergänge wie bspw. $\wedge \rightarrow \overline{\oplus}$ oder $\overline{\vee} \rightarrow \overline{x}$ erfolgen.

x	y	$\wedge$	$\overline{\oplus}$	$\overline{\vee}$	$\overline{x}$	$\overline{\wedge}$	$\oplus$	$\vee$
0	0	0	1	1	1	1	0	0
0	1	0	0	0	1	1	1	1
1	0	0	0	0	0	1	1	1
1	1	1	1	0	0	0	0	1

Daß der evolutionäre Entwurf von Schaltkreisen, ob analog oder digital, außerordentlich schwierig ist, zeigen nicht nur die in dieser Arbeit durchgeführten Experimente, sondern vor allem die intensiven Bemühungen auf dem Forschungsgebiet der Evolvable Hardware. Und dennoch: Trotz dieser Bemühungen stehen große Erfolge noch aus, denn der evolutionäre Schaltkreisentwurf bietet zwar eine Alternative zum herkömmlichen Schaltkreisdesign. Jedoch sind die evolvierten Schaltkreise zum einen für den Menschen schwer zu verstehen, vgl. Thompson und Layzell (1999), und zum anderen von beschränkter Komplexität, so daß sie lediglich als „Building Blocks“ für größere Schaltungen in Frage kommen können.

Dabei stellt sich natürlich die Frage, ob Schaltkreise bzw. Geräte, von denen man nicht genau weiß wie sie funktionieren, mit gutem Gewissen an Orten eingesetzt werden sollten oder gar dürfen, an denen es auf höchste Sicherheit ankommt. So sei an dieser Stelle noch einmal der geplante Einsatz von Evolvable Hardware im militärischen Sektor oder der Raumfahrt erwähnt. Gerade beim letztgenannten Bereich können allerdings schon die kleinsten Fehler, sowohl bei der Software als aber auch bei der Hardware katastrophale Folgen haben. Die durch Software-Fehler bereits bestehenden Gefahren, könnten durch den Einsatz „unberechenbarer“, weil unverstandener Hardware, nur noch potenziert werden.

Die andere – positive – Seite der Medaille darf natürlich nicht unerwähnt werden: Die Aussicht auf Hardware-Entitäten, die kleinere Defekte selbständig „on-chip“ reparieren können, die Hoffnung auf neuartige Prothesen, die Behinderten einen angemesseneren Umgang mit einer Umwelt ermöglichen, die vornehmlich für nichtbehinderte Menschen konstruiert wurde, oder auch die Vorstellung von nicht einmal sichtbaren Kleinstrobotern im Nanometerbereich, die im menschlichen Körper Reparaturen durchführen, ist verführerisch. So lassen es sich einige Forscher im Bereich der EHW auch nicht nehmen einen mitunter weiten Blick in die Zukunft zu wagen.

> „Looking (and dreaming) toward the future, one can imagine nano-scale (bioware) systems becoming a reality, which will be endowed with evolutionary, reproductive, regenerative, and learning capabilities. Such systems could give rise to novel species which will coexist alongside carbon-based organisms.“ (Tomassini und Sipper 1997, S. 10).

A Einige Besonderheiten der XC6216 RPU

Down that path lies madness. On the other hand, the road to hell is paved with melting snowballs.

– Larry Wall

Während der Arbeit mit der XC6216 RPU zeigten sich einige Besonderheiten, die in diesem Kapitel näher betrachtet werden. Diese Informationen mögen als Hinweise zur Vermeidung unnötiger und vor allem zeitraubender Fehler dienen.

A.1 Signalinvertierungen

Eine Eigenheit des XC6216, die in Xilinx Inc. (1997) nicht die gebührende Beachtung erfährt, die ihrer Wichtigkeit eigentlich zukommen sollte, stellen die Ausgabe-Signalinvertierungen der Routing Multiplexer einer jeden Zelle dar. Diese Eigenschaft wird im Kapitel *Understanding The Configuration Bits* des Programmable Logic Data Book wie folgt abgehandelt: „Note that most of the routing multiplexers invert their outputs to reduce propagation delays. This has not been shown in earlier figures." (Xilinx Inc. 1997, Seite 21). Die Implikationen dieser Aussage sind jedoch mannigfaltig, insbesondere da das Berechnungsergebnis einer Zelle von ihren Eingaben abhängt und diese Eingaben ihrerseits in den meisten Fällen Ausgaben benachbarter Zellen sein werden. Liegen diese Ausgaben also invertiert vor, so berechnet eine Zelle die als $A.B$ Gatter konfiguriert wurde (lt. Tabelle 2 Function Derivation, Xilinx Inc. 1997, Seite 8) nicht, wie ursprünglich gedacht, $A.B$ sondern $\bar{A}.\bar{B} = \overline{A+B}$. Zur Verdeutlichung des Vorgangs zeigt Abbildung 34 die Routing Multiplexer als Ursache der Signalinvertierungen sowie das wechselnde Funktionsergebnis einer Zelle bei unterschiedlicher Verschaltung mit ihren Nachbarzellen.

Für die sowohl auf der linken als auch der rechten Seite von Abbildung 34 dargestellten Verschaltungen gilt, daß die oberste Zelle als Ergebnis ihrer Funktionseinheit und damit als Ausgabesignal konstant 0 liefert. In beiden Fällen wird dieses Signal als Eingabe für die

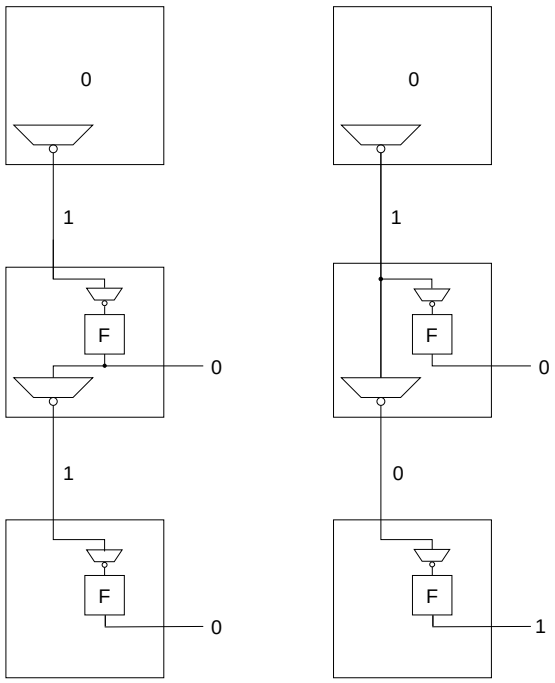


Abbildung 34 Signalinvertierungen und deren Auswirkung auf das Funktionsergebnis einer Zelle. Auf der linken Seite wird das Ausgabesignal der obersten Zelle als Eingabesignal für die Funktionseinheit der mittleren Zelle verwendet wird. Die mittlere Zelle leitet dann ihr Berechnungsergebnis (hier die Identitätsfunktion) an die untere Zelle weiter. Rechts hingegen wird das Ausgabesignal der obersten Zelle direkt an die unterste Zelle weiteregeleitet – es erfolgt eine Invertierung weniger als auf der linken Seite. Als Folge kommt es zu unterschiedlichen Ausgaben der untersten Zelle.

Funktionseinheit der mittleren Zelle verwendet. Ein entscheidender Unterschied ist jedoch, daß das Eingabesignal (konstant 0) der mittleren Zelle auf der rechten Seite *direkt* an die darunter liegende Zelle weitergeleitet wird. Im Gegensatz dazu leitet die mittlere Zelle auf der linken Seite das *Ergebnis ihrer Funktionseinheit* an die darunter liegende Zelle weiter. Dadurch unterscheiden sich die an der unteren Zelle ankommenden Signale. Während die untere Zelle auf der linken Seite als Eingabesignal eine konstante 1 erhält und somit ihre Ausgabe 0 lautet, gilt für die unterste Zelle auf der rechten Seite eine Eingabe von konstant 0 und damit als Ausgabe der Wert 1. Im Fall zur Rechten erfolgt somit eine Invertierung weniger und das ursprüngliche Signal wurde invertiert.

Die Invertierung der Ein-/Ausgabesignale durch die entsprechenden Multiplexer einer jeden Zelle stellt somit kein Problem dar, solange die Ausgaben benachbarter Zellen *direkt* als Eingaben für die Funktionseinheit einer Zelle verwendet werden. Im Hinblick auf das Wei-

terleiten eines Signals treten jedoch Komplikationen auf, da an jedem Übergang von einer Zelle zur nächsten eine Invertierung dieses Signals stattfindet.

A.2 Schaltkreisein- und ausgabe

Zwei entscheidende Fragen bei der Arbeit mit dem XC6216 waren: Wie gelangen Eingabesignale in den instantiierten Schaltkreis und wie kann das Berechnungsergebnis eines Schaltkreises wieder ausgelesen werden? Deren Beantwortung war schwieriger als zunächst erwartet, denn es gibt weder einfache Methoden bei der mitgelieferten Software, die es erlauben bspw. die Eingaben einer beliebigen Zelle des Arrays auf bestimmte Werte zu setzen, noch die Ausgaben beliebiger Zellen direkt auszulesen. Gerade der letzte Fall läßt sich jedoch vergleichsweise einfach realisieren, so daß dieser zuerst behandelt werden soll.

A.2.1 Lesen einer Zellausgabe

Für das Lesen des Funktionsergebnisses einer oder mehrerer Zellen gibt es eine wesentliche Einschränkung: Es können lediglich solche Zellen gleichzeitig gelesen werden, die in derselben Spalte liegen, wobei für das Lesen einer kompletten Spalte eine entsprechende Methode in der mitgelieferten Funktionsbibliothek zu finden ist. Das Funktionsergebnis von Zellen wird unter Zuhilfenahme des sog. **Map Register** gelesen. Dieses 64 Bit große Register stellt eine Maske dar, die dafür sorgt, daß nur jene Funktionsergebnisse von Zellen beim Lesen einer Spalte des Zellarrays auf den Datenbus gelegt werden, für die im Map Register eine 0 steht. Jedes Bit des Registers steht somit für eine der 64 Zellen einer Spalte des FPGA. Dabei entspricht die Bit-Position 0 (niederwertigstes Bit) des Map Registers der Zelle an Position $(x, 0)$ mit $x \in 0..63$, also der jeweils südlichsten Zelle des Gate-Arrays. Zur Verdeutlichung des Zusammenhangs stellt Abbildung 35 eine vereinfachte Ansicht des *8-Bit Data Bus Example* aus Xilinx Inc. (1997) dar (Figure 25 Internal Register Access, Seite 18). Das Funktionsergebnis genau jener Zellen einer Spalte, deren Bit im Map Register auf 0 gesetzt wurde, gelangt beim Lesen der entsprechenden Spalte auf den Datenbus.

Die Vorgehensweise zum Lesen von Zell-Funktionsergebnissen gliedert sich damit in die folgenden drei Schritte:

A.2.1.1 Map Register setzen. Das Map Register muß zur Maskierung der gewünschten Zellen einer Spalte gesetzt werden. Für das Beispiel in Abbildung 35 müßte das Setzen des Map Registersm, wie in Beispiel-Code 4, angegeben gesetzt werden.

A.2.1.2 Spalte lesen. Im Beispiel der Abbildung 35 befinden sich die Zellen in Spalte 2 des Gate-Array. Ein Befehl wie `fpga.getColumn(2)`

Da es 64 Zellen je Spalte gibt, über den Datenbus jedoch maximal 32 Bit übertragen werden können, sind zum Auslesen *aller* Zellen einer Spalte zwei Lesevorgänge nötig zwischen denen jeweils das Map Register umgestellt werden muß.

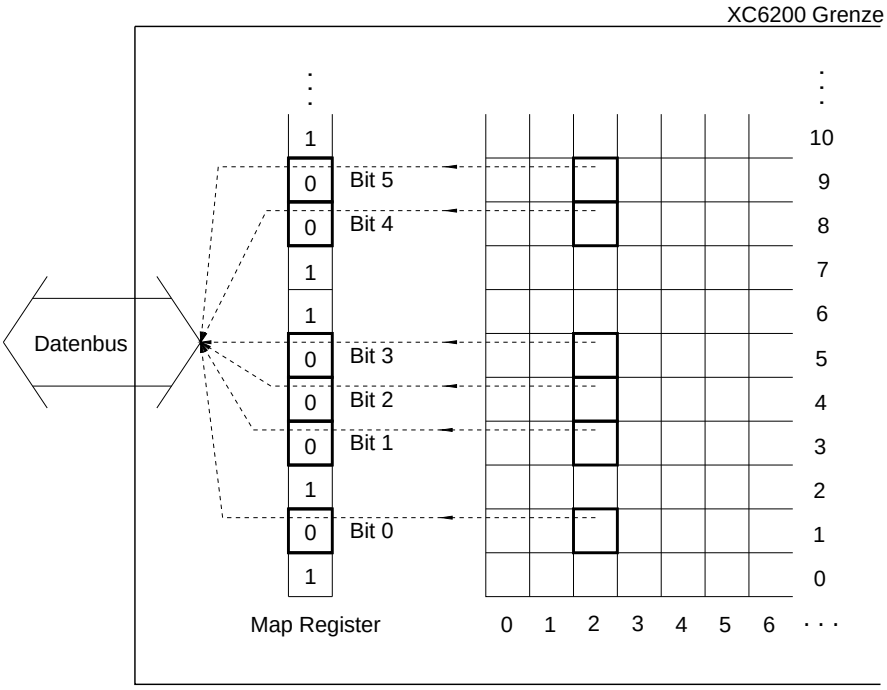


Abbildung 35 Zuordnung der Map Register Bits zu Zellen einer Spalte des Arrays.

Beispiel-Code 4

```
XC6200 fpga;                          // Instantiiere XC6200 Objekt
fpga.setBusWidth( 32 );               // Busbreite auf 32 Bit
fpga.setMap( 0, 0xfffff4c5L );        // Map[31:0] = 111...10011000101
fpga.setMap( 1, 0xffffffffL );        // Map[63:32] = 111...1111111111
```

genügt nun, um die Bits entsprechend der eingestellten Busbreite aus der angegebenen Spalte zu lesen. In obigem Beispiel sollen die Berechnungsergebnisse von sechs Zellen ($\widehat{=}$ 6 Bit) gelesen werden, somit wird die Busbreite zunächst auf 8 Bit eingstellt und im Anschluß Spalte 2 gelesen; Beispiel-Code 5 zeigt dieses Vorgehen.

Angenommen, alle sechs Zellen liefern als Funktionsergebnis ihrer Berechnung 0, dann enthielte die Variable `col` nach der Zuweisung in Beispiel-Code 5 den Wert `11000000` in binärer Darstellung. Die ersten beiden Bits des Rückgabewertes `col`, die durch das Map Register nicht maskiert werden konnten, wurden auf 1 gesetzt; dies geschieht für alle

Beispiel-Code 5

```
unsigned col;
fpga.setBusWidth( 8 );                // Busbreite auf 8 Bit
col= fpga.getColumn( 2 );             // Lese 8 Bit aus Spalte 2
```

Bits, die vom Map Register nicht maskiert werden. Für den Fall, daß das Funktionsergebnis der Zellen von oben nach unten gelesen `011010` lautete, ergäbe das Lesen der Spalte den Wert `11011010`.

A.2.1.3 Maskieren von Bits. Der Vollständigkeit halber sei noch kurz das Ausmaskieren von Bits erwähnt, um auf das Ergebnis einer spezifischen Zelle zugreifen zu können. Hierzu sind einige einfache Schiebe- und Maskieroperationen notwendig, die aber Brot und Butter des redlichen Programmierers darstellen und in keinster Weise spezifisch für den Umgang mit der XC6216 RPU sind. Der Zugriff auf das Funktionsergebnis der dritten Zelle von oben in Abbildung 35 könnte bspw. mit der Anweisung `unsigned res=(fpga.getColumn(2)>>3)&1` erfolgen.

A.2.2 Schreiben von Zelleingaben

Im Gegensatz zum Lesen des Funktionsergebnisses einer Zelle stellt sich das Anlegen von Eingabesignalen an die Funktionseinheit einer Zelle aufwendiger dar. Es ist nämlich nicht möglich eine oder mehrere der drei Eingabeleitungen $X1, X2$ und $X3$ einer Funktionseinheit direkt vom Host aus zu beschreiben. Zwar existiert eine Methode zum Schreiben von Zellen `setColumn(col,val)` als Pendant zur Methode zum Lesen, jedoch erlaubt diese nur das Setzen des **D-Type Registers** einer jeden Zelle. Da es möglich ist eine Zelle so zu konfigurieren, daß sie den Inhalt ihres Registers als Funktionsergebnis liefert, können über diesen Umweg Eingaberegister konstruiert werden, die den Inhalt ihrer D-Type Register als Eingabe an den Schaltkreis weiterleiten. Abbildung 36 zeigt einen einfachen $A.B$ Schaltkreis bei dem dieses Prinzip umgesetzt wurde.

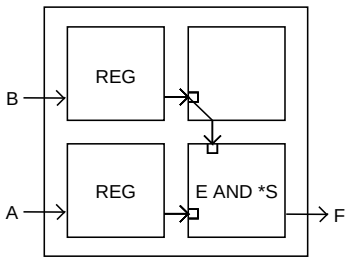


Abbildung 36 Ein Schaltkreis der $F = A.B$ berechnet.

Zwei als Eingaberegister konfigurierte Zellen in der ersten Spalte leiten ihr Funktionsergebnis, also den Inhalt ihres D-Type Registers, jeweils in die benachbarten Zellen der folgenden Spalte weiter. Die Zelle oben rechts dient zur Weiterleitung des Eingabesignals nach Süden, wobei es zur Invertierung der Eingabe, wie bereits in Anhang A.1 beschreiben,

kommt. Aufgrund dieser Invertierung wurde die Funktionseinheit der Ausgabezelle unten rechts so konfiguriert, daß sie $A.\bar{B}$ berechnet, denn als Eingaben liegen in Wirklichkeit A und $\bar{B}$ vor. Das Funktionsergebnis F wird mit dieser Konfiguration nur für die Eingaben $A = 1$ und $B = 1$ eins sein, in allen anderen Fällen gilt $F = 0$, was der korrekten Berechnung von $A.B$ entspricht.

Eine weitere Möglichkeit zur Eingabe von Daten in einen Schaltkreis stellt die Verwendung der IOBs dar, also das direkte Einspeisen von Signalen über die Anschlußpins des FPGA. Thompson (1998) verwandte diesen Ansatz, um ein externes analoges Signal in seinem Frequency Discriminator verwenden zu können.

Damit ergibt sich, daß die Hauptarbeit bei der Einspeisung von Signalen in einen Schaltkreis bei der Konfiguration der Eingaberegister liegt. Beispiel-Code 6 zeigt, wie die untersten n Zellen der ersten Spalte als Register konfiguriert und beschrieben werden. Dabei sei `fpga` eine Instanz der Klasse `XC6200` und `addr1()` die in Beispiel-Code 2 vorgestellte Mehode, die aus ihren Parametern eine zulässige Adresse des FPGA berechnet, vgl. Xilinx Inc. (1997), Address Bus Format, Seite 20ff. Alle verwendeten Konstanten befinden sich in der Headerdatei der Klasse `XCAddr`.

Ein entscheidender Punkt bei der Register-Konfiguration stellt der Inhalt des **CS-Bits** des **Function Routing** Konfigurationswortes dar. Dieses Bit wird im Beispiel-Code 6 für jede der n Zellen der ersten Spal-

Beispiel-Code 6

```
00 for ( i=0; i<n; ++i ) {
01  // CS=0(*Q), X1=X2=X3=W4in
02  fpga.write6200( addr( CELLMODE, 0, CELLFR, i ), 0x40 );
03  // M=0(*X3), RP=1(ON), Y2=X2, Y3=X3, X3[2]=1, X2[2]=1
04  fpga.write6200( addr( CELLMODE, 0, CELLFU, i ), 0x43 );
05 }
06
07 // Register writes require clocking, so clock all cells. Set PrimaryClk[1:0]
08 // in first config byte (col. offset 00) of all north switches to GClk.
09 // PrimaryClk[1:0]=GClk, N[2]=*F
10 for ( c=0; c<FPGA_SIZE; ++c )
11  for ( r=0; r<FPGA_SIZE; r+=4 )
12   fpga.write6200( addr( NSMODE, c, NSREG0, r+3 ), 0x6 );
13
14 // Set IOB below column 0 to use the global clock as clocking source.
15 fpga.write6200( addr( NSMODE,   // address mode
16                       0,        // switch column
17                       NSREG0,   // iob reg0
18                       1,        // south iob [5:2]=0000, [1:0]=01
19                       0x20 )    // Clk[1:0]=10, rest to 0
20 );
21
22 // Set missing 3rd bit for GClk source in IOB reg1.
23 fpga.write6200( addr( NSMODE, 0, NSREG1, 1 ), 0x10 );
```

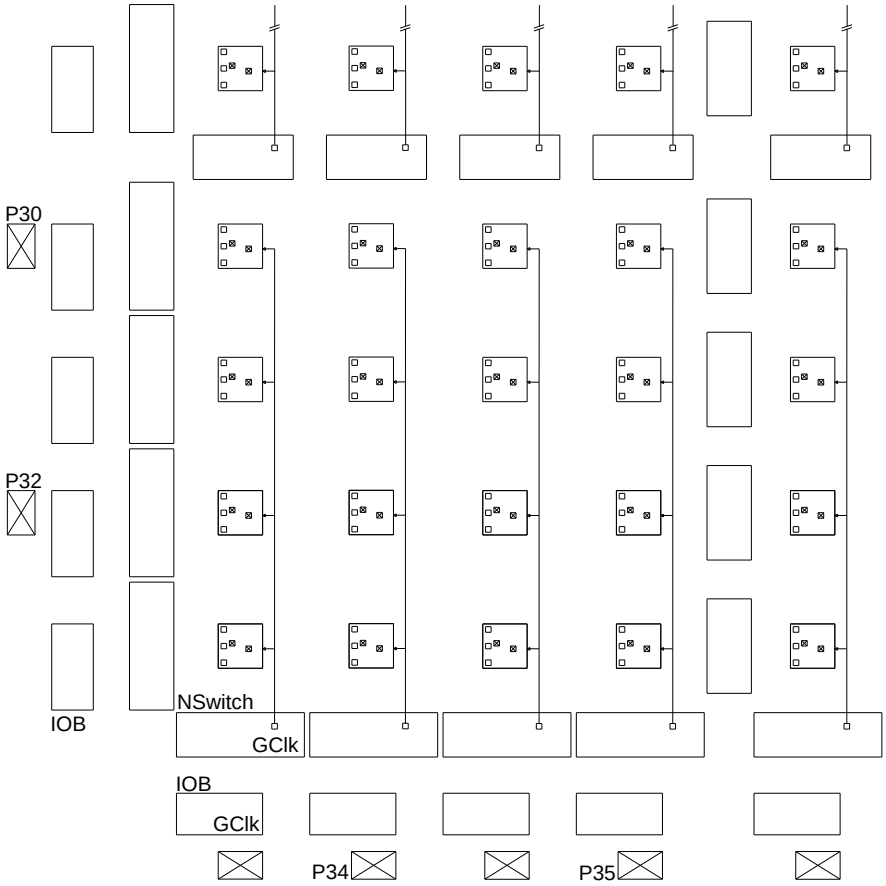


Abbildung 37 Jeweils vier untereinanderliegende Zellen in jeder Spalte verwenden das Global Clock-Signal der North Switches zur Taktung ihrer D-Type Register.

te innerhalb der Schleife auf $0(\bar{Q})$ gesetzt (Konfigurationswort ist 0x40). Die Ausgabe der Funktionseinheit dieser Zelle ist somit der Inhalt des D-Type Registers Q und nicht, wie es für $CS = 1$ der Fall wäre, das Ergebnis einer Berechnung der Funktionseinheit.

Darüberhinaus gibt es eine Besonderheit bei der Verwendung der Flip-Flops einer Zelle zu beachten. Zum Schreiben der Register müssen diese **getaktet** werden. Dies geschieht durch Setzen der **Primary Clock** aller North Switches an 4×4 Zellgrenzen auf **GClk**, d. h. jeweils 4 untereinanderliegende Zellen einer jeden Spalte verwenden die **Global Clock** als Taktgeber für ihre D-Type Register.

Die Zeilen 7–12 des Beispiel-Code 6 zeigen, wie die North Switches des kompletten Arrays konfiguriert werden müssen, damit die D-Type Register einer jeden Zelle mit dem Global Clock Signal getaktet werden. In den meisten Fällen wird es jedoch nicht nötig sein *alle* North Switches des gesamten Zellarrays zu konfigurieren, sondern nur jene, die genau den Zellen zugeordnet sind, die als Eingaberegister fungieren sollen.

Mit den bisher erfolgten Konfigurationen können die Eingaberegister jedoch noch immer nicht als solche verwendet werden, da abschließend das Global Clock Signal vom **GClk I/O Pad** des FPGA in den Schaltkreis geleitet werden muß. Dieser Schritt ist notwendig, da als Taktgeber für ein D-Type Register verschiedene Quellen zum Einsatz kommen können, so bspw. ein Taktsignal vom Hostrechner, ein im Schaltkreis selbst erzeugtes Taktsignal oder eben der **Taktgenerator** (Oszillator) des **PCI-Boards** auf dem der FPGA montiert ist. Die Zeilen 14–23 im Beispiel-Code 6 zeigen, wie der IOB unter der ersten Spalte im Westen des FPGA konfiguriert werden muß, damit er das Global Clock Signal des GClk I/O Pads in den Schaltkreis leitet.

Es sei noch einmal erwähnt, daß dieses Vorgehen ausschließlich zur Verwendung der D-Type Register einer Zelle nötig ist. Die Berechnung von Zellfunktionen oder das Auslesen von Berechnungsergebnissen funktioniert auch ohne Konfiguration der North Switches oder IOBs.

B Beispiele zur Programmierung des XC6216

bit: a boring tool
– Webster

In diesem Abschnitt soll anhand zweier einfacher Beispiele gezeigt werden, wie ein Schaltkreis zu Fuß in den XC6216 eingeschrieben werden kann. „Zu Fuß“ bedeutet in diesem Fall, daß die zur Konfiguration der einzelnen Zellen, Switches, etc. nötigen Bits mit Hilfe des in Kapitel 3 vorgestellten Vorgehens mit dem Taschenrechner berechnet wurden.

Zunächst wird ein sehr einfacher AND_2-Schaltkreis vorgestellt, der die Funktion $F = A.B$ berechnet und im Anschluß ein etwas komplexerer OR_5-Schaltkreis, der $A + B + C + D + E$ als Funktionsergebnis F liefert. Beide Schaltkreise werden in der linken unteren Ecke des FPGA eingeschrieben.

B.1 AND_2 Schaltkreis

Aufgabe ist es, einen Schaltkreis in den XC6216 einzuschreiben, der auf seinen zwei Eingabe-Bits A und B die AND-Funktion berechnet. Der gewünschte Aufbau soll dazu wie in Abbildung 38 dargestellt aussehen.

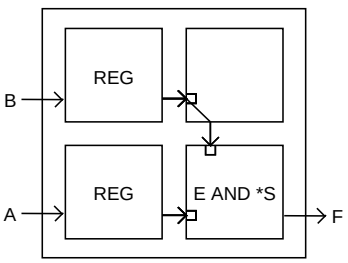


Abbildung 38 Ein Schaltkreis der $F = A.B$ berechnet.

Die zwei untersten Zellen der ersten Spalte dienen als Eingaberegister (vgl. Abschnitt A.2). Die Zelle an Position $(1, 0)$, die gleichzeitig als

Ausgabezelle des Funktionsergebnisses F dient, wird die Funktion A.B berechnen, während die Zelle an Position $(1,1)$ lediglich als Routing-Zelle fungiert, um das Eingabesignal B an die darunterliegende Ausgabezelle weiterzuleiten. Bei dieser Weiterleitung kommt es zur Invertierung von B (vgl. Abschnitt A.1), so daß als Zellfunktion in Wirklichkeit $A.\bar{B}$ instantiiert wird. Der folgende kommentierte Beispiel-Code stellt, unter Verwendung der in Beispiel-Code 1 und 2 vorgestellten Funktionen, ein lauffähiges Program zum Einschreiben des AND_2-Schaltkreises in den XC6216 dar.

Beispiel-Code 7

```
#include <stdlib.h>
#include <stdio.h>
#include "XC6200DS.h"
#include "XCAddr.h"

int main( )
{
  void writeCircuit( XC6200 &obj );          // function prototype
  XC6200 fpga;                               // instantiate FPGA-object
  unsigned int j;

  fpga.reset( );                             // reset the FPGA
  fpga.clear( );                             // erase all registers, etc.
  fpga.writeInvalidDeviceID( );              // disable output devices
  writeCircuit( fpga );                      // write and2 circuit into FPGA
  fpga.writeValidDeviceID( );                // enable output devices
  fpga.setBusWidth( 32 );                    // set the external bus width to 32 bit
  fpga.setMap( 0, 0xfffffffcL );             // set lo-byte of map register
  fpga.setMap( 1, 0xffffffffL );             // set the hi-byte
  fpga.enableClock( );                       // start global clock
  for ( j= 0; j < 4; ++j ) {                 // test the circuit
     fpga.setColumn( 0, j );                 // write j into 1st end
     printf( "Wrote %i to (0,1) and ", (fpga.getColumn( 0 )&3 )>>1 );
     printf( "%i to (0,0) ", fpga.getColumn( 0 )&1 );
     printf( "- result from output-cell at pos " );
     printf( "(1,0) is %i.\n", fpga.getColumn( 1 )&1 );
  }
}

void writeCircuit( XC6200 &fpga )
{
  // Configure cells at pos (0,0) and (0,1) to act as input registers.
  // CS = 0(Q̄), X1 = X2 = X3 = W4_in
  // M = 0(X̄3), RP = 1(ON), Y2 = X2, Y3 = X3, X3[2] = 1, X2[2] = 1
  fpga.write6200( addr1( XCAddr::CELLMODE, 0, XCAddr::CELLFR, 0 ), 0x40 );
```

Fortsetzung Beispiel-Code 7

```
  fpga.write6200( addr1( XCAddr::CELLMODE, 0, XCAddr::CELLFU, 0 ), 0x43 );
  fpga.write6200( addr1( XCAddr::CELLMODE, 0, XCAddr::CELLFR, 1 ), 0x40 );
  fpga.write6200( addr1( XCAddr::CELLMODE, 0, XCAddr::CELLFU, 1 ), 0x43 );

  // Configure North Switch below column 0 so the four cells underneath the switch
  // will use the GClk signal as clocking source. PrimaryClk[1:0]=GClk, N[2]=F̄.
  fpga.write6200( addr1( XCAddr::NSMODE, col, XCAddr::NSREG0, 0x3 ), 0x6  );

  // Configure South IOB below column 0 to use the global clock (GClk) as clocking source
  fpga.write6200( addr1( XCAddr::NSMODE, 0, 0, 1 ), 0x20 ); // Clk[1:0]=10, rest to 0
  // We need three bits. Missing 3rd bit is set in reg1 of IOB.
  fpga.write6200( addr1( XCAddr::NSMODE, 0, 1 , 1 ), 0x10 );

  // Cell at pos (1,1) will be configured to route the output signal from left neighbour
  // cell E_in at pos (0,1) to lower neighbour cell at (1,0). The latter one implementing the
  // A.B̄ gate and thus the register input in register (0,1) will be one of the two inputs
  // for the A.B̄ gate in (1,0). Configure additionally as a buffer storing input from E_in.
  // S_out=E_in, N_out = E_out = W_out = F
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLNR, 1 ), 0x1  );
  // CS = 1(C), X1 = W4_in, X2 = X3 = Ē_in
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLFR, 1 ), 0xc9 );
  // M = X̄3, RP = 0(OFF), Y2 = X̄2, Y3 = X̄3, X3[2] = 0, X2[2] = 0
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLFU, 1) , 0x24 );

  // Configure cell at (1,0) to be an A.B̄ gate with two inputs. First input comes from
  // left neighbour cell and therefore is the output of the register at (0,0). Second input
  // comes from the upper neighbour cell at (1,1) which is nothing more than a router that
  // feeds the cell output from register at (0,1) to S_out which in turn is the second input
  // for the A.B̄ gate at pos (1,0).
  // N_out = E_out = W_out = S_out = F
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLNR, 0 ), 0x0  );
  // CS = 1(C̄), X1 = X2 = Ē_in, X3 = S̄_in
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLFR, 0 ), 0x98 );
  // M = X̄3, RP = 0(OFF), Y2 = X2, Y3 = X̄3, X3[2] = 0, X2[0] = 0
  fpga.write6200( addr1( XCAddr::CELLMODE, 1, XCAddr::CELLFU, 0 ), 0x4  );
}
```

B.2 OR_5 Schaltkreis

In diesem zweiten und abschließenden Beispiel soll gezeigt werden, wie ein OR_5-Schaltkreis in den XC6216 eingeschrieben werden kann. Der Schaltkreis soll die Funktion $F = A + B + C + D + E$ berechnen, d. h. es gibt fünf Eingabe- und ein Ausgabe-Bit. Der Entwurf ist sehr ähnlich zu dem im vorigen Abschnitt B.1 vorgestellten AND_2-Schaltkreis; wiederum werden einige Zellen der ersten Spalte als Eingaberegister verwendet und es existiert eine dezidierte Ausgabezelle, an der das Funktionsergebnis F gelesen wird. Abbildung 39 zeigt den Aufbau des OR_5-Schaltkreises.

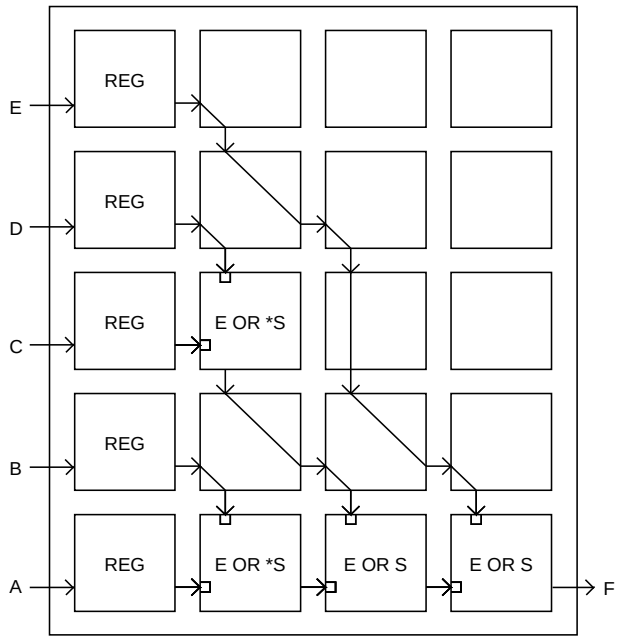


Abbildung 39 Ein OR_5-Schaltkreis, der $F = A + B + C + D + E$ berechnet.

Eine nähere Erläuterung der Funktionsweise des Schaltkreises ist aufgrund des einfachen Aufbaus nicht notwendig. Interessant sind, ebenso wie beim AND2 Schaltkreis des vorangegangen Abschnitts, die internen Signalinvertierungen, die bei der Weiterleitung von Signalen durch Zellen auftreten. Bei den Zellen an Position $(1, 2)$ und $(1, 0)$ führt dies dazu, daß die Funktion $E+S$ durch $E+\bar{S}$ ersetzt werden muß, um diesen Invertierungen entgegenzuwirken. Das Ergebnis der Funktionsberechnung des Schaltkreises wird von der Ausgabezelle an Position $(3, 0)$ gelesen.

Im folgenden zeigt der kommentierte Beispiel-Code, ebenfalls unter Verwendung der in den Beispiel-Codes 1 und 2 vorgestellten Funktionen, ein lauffähiges Program zum Einschreiben des OR_5-Schaltkreises in den XC6216. Insbesondere in der `main()` Funktion treten im Vergleich zum AND_2 Beispiel des vorigen Abschnitts kaum Unterschiede auf, so daß nur die geänderten Zeilen aufgeführt sind.

Beispiel-Code 8

```
#include <stdlib.h>
#include <stdio.h>
#include "XC6200DS.h"
#include "XCAddr.h"

int main( )
{
     ⋮
  fpga.setMap( 0, 0xfffffffe0L );            // set lo-byte of map register
  fpga.setMap( 1, 0xffffffffL );             // set the hi-byte
  fpga.enableClock( );                       // start global clock
  for ( j= 0; j < 32; ++j ) {                // test the or5 circuit
     fpga.setColumn( 0, j );                 // write j into 1st end
     printf( "Wrote %i to column 0 - result from output-cell ", j );
     printf( "at pos (3,0) is %i - \n", fpga.getColumn( 3 )&1 );
  }
}

void writeCircuit( XC6200 &fpga )
{
  const unsigned CM= XCAddr::CELLMODE;
  const unsigned NR= XCAddr::CELLNR;
  const unsigned FR= XCAddr::CELLFR;
  const unsigned FU= XCAddr::CELLFU;
  unsigned i;

  // Set the five cells in column 0 to act as input registers for the or5 circuit.
  // CS = 0(Q̄), X1 = X2 = X3 = W4_in
  // M = 0(X̄3), RP = 1(ON), Y2 = X2, Y3 = X3, X3[2] = 1, X2[2] = 1
  for ( i= 0; i < 5; ++i ) {
    fpga.write620000( addr1( CM, 0, FR, i ), 0x40 );
    fpga.write620000( addr1( CM, 0, FU, i ), 0x43 );
  }

  // Configure North Switches of 4 × 4 block 0 and 1 in column 0 so the cells underneath the switches
  // will use the GClk signal as clocking source. PrimaryClk[1:0]=GClk, N[2]=F̄.
  fpga.write620000( addr1( XCAddr::NSMODE, col, XCAddr::NSREG0, 0x3 ), 0x6  );
  fpga.write620000( addr1( XCAddr::NSMODE, col, XCAddr::NSREG0, 0x7 ), 0x6  );

  // Configure South IOB below column 0 to use the global clock (GClk) as clocking source.
  fpga.write620000( addr1( XCAddr::NSMODE, 0, 0, 1 ), 0x20 );
  // We need three bits. Missing 3rd bit is set in reg1 of IOB.
  fpga.write620000( addr1( XCAddr::NSMODE, 0, 1 , 1 ), 0x10 );

  // Configure remaining cells. 1st part of comment denotes position, next part function of cell.
  // (1,0) E + S̄
  fpga.write620000( addr1( CM, 1, NR, 0 ), 0x0  );
  fpga.write620000( addr1( CM, 1, FR, 0 ), 0x91 ); // X2 = S̄_in, X1 = X3 = E_in
  fpga.write620000( addr1( CM, 1, FU, 0 ), 0x20 ); // Y2 = X̄2, Y3 = X3
```

Fortsetzung Beispiel-Code 8

```
  // (2,0) E.S → Ē.S̄ → E + S
  fpga.write620000( addr1( CM, 2, NR, 0), 0x0  );
  fpga.write620000( addr1( CM, 2, FR, 0), 0x91 ); // X1 = X3 = E_in, X2 = S_in
  fpga.write620000( addr1( CM, 2, FU, 0), 0x0  ); // Y2 = X2, Y3 = X3

  // (3,0) E.S → Ē.S̄ → E + S
  fpga.write620000( addr1( CM, 3, NR, 0 ), 0x0  );
  fpga.write620000( addr1( CM, 3, FR, 0 ), 0x91 ); // X1 = X3 = E_in, X2 = S_in
  fpga.write620000( addr1( CM, 3, FU, 0 ), 0x0  ); // Y2 = X2, Y3 = X3

  // (1,1) routing cell
  fpga.write620000( addr1( CM, 1, NR, 1 ), 0x31 ); // S_out = E_in, E_out = S_in, N_out = W_out = F̄
  fpga.write620000( addr1( CM, 1, FR, 1 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 1, FU, 1 ), 0x3  ); // X3[2] = X2[2] = 1

  // (2,1) routing cell, same as (1,1)
  fpga.write620000( addr1( CM, 2, NR, 1 ), 0x31 );
  fpga.write620000( addr1( CM, 2, FR, 1 ), 0x40 );
  fpga.write620000( addr1( CM, 2, FU, 1 ), 0x3  );

  // (3,1) routing cell
  fpga.write620000( addr1( CM, 3, NR, 1 ), 0x1  ); // S_out = E_in, E_out = N_out = W_out = F̄
  fpga.write620000( addr1( CM, 3, FR, 1 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 3, FU, 1 ), 0x3  ); // X3[2] = X2[2] = 1

  // (1,2) E + S̄
  fpga.write620000( addr1( CM, 1, NR, 2 ), 0x0  );
  fpga.write620000( addr1( CM, 1, FR, 2 ), 0x91 ); // X1 = X2 = E_in, X3 = S_in
  fpga.write620000( addr1( CM, 1, FU, 2 ), 0x20 ); // Y2 = X̄2, Y3 = X̄3

  // (2,2) routing cell
  fpga.write620000( addr1( CM, 2, NR, 2 ), 0x3  ); // S_out = S_in, N_out = E_out = W_out = F̄
  fpga.write620000( addr1( CM, 2, FR, 2 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 2, FU, 2 ), 0x3  ); // X3[2] = X2[2] = 1

  // (1,3) routing cell
  fpga.write620000( addr1( CM, 1, NR, 3 ), 0x31 ); //S_out=E_in, E_out=S_in, N_out=W_out=F̄
  fpga.write620000( addr1( CM, 1, FR, 3 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 1, FU, 3 ), 0x3  ); // X3[2] = X2[2] = 1

  // (2,3) routing cell
  fpga.write620000( addr1( CM, 2, NR, 3 ), 0x1  ); // S_out = E_in, E_out = N_out = W_out = F̄
  fpga.write620000( addr1( CM, 2, FR, 3 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 2, FU, 3 ), 0x3  ); // X3[2] = X2[2] = 1

  // (1,4) routing cell
  fpga.write620000( addr1( CM, 1, NR, 4 ), 0x1  ); // S_out = E_in, E_out = N_out = W_out = F̄
  fpga.write620000( addr1( CM, 1, FR, 4 ), 0x40 ); // X1 = X2 = X3 = W4_in
  fpga.write620000( addr1( CM, 1, FU, 4 ), 0x3  ); // X3[2] = X2[2] = 1
}
```

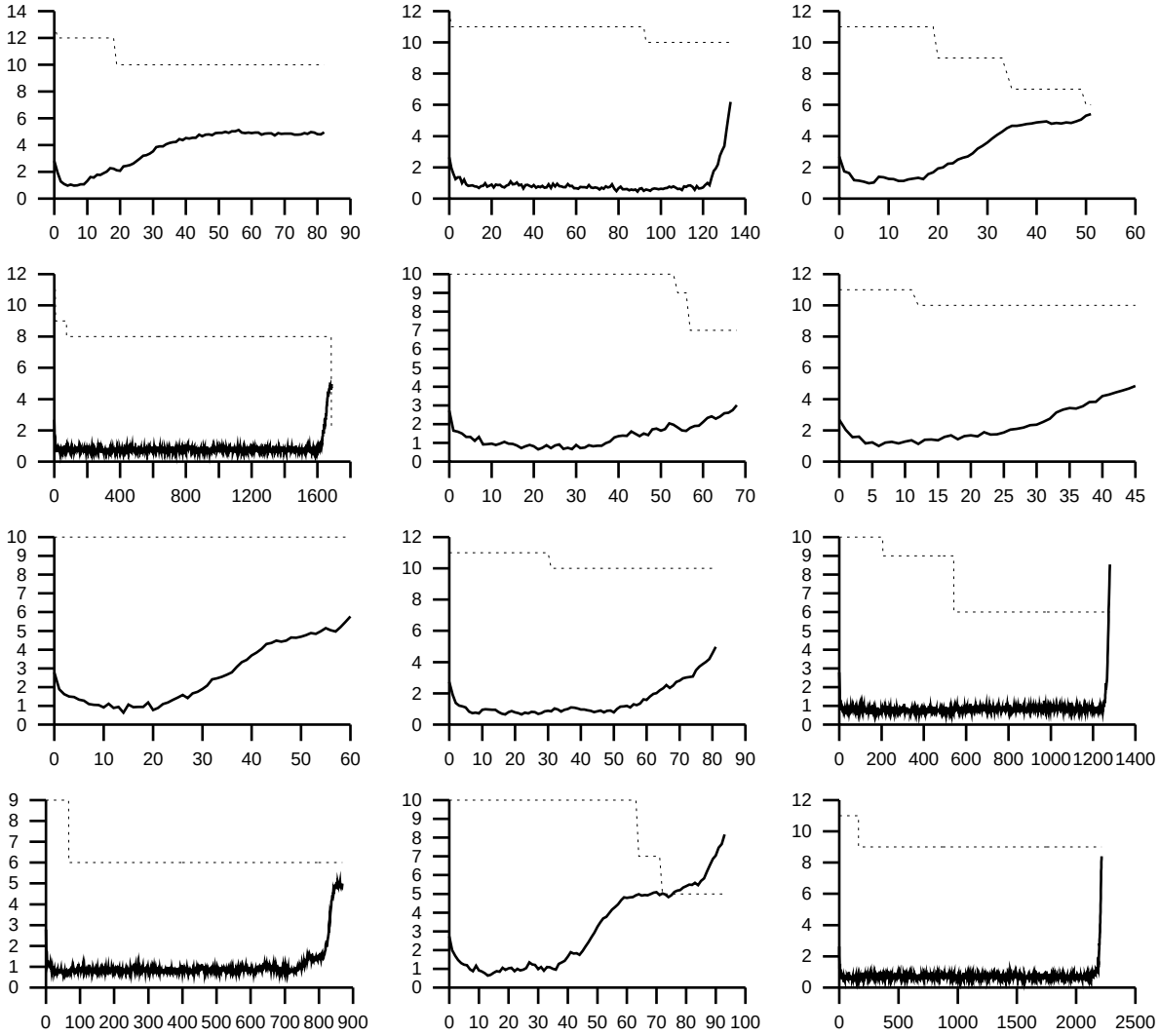
C Tabellarische Zusammenfassung der Ergebnisse

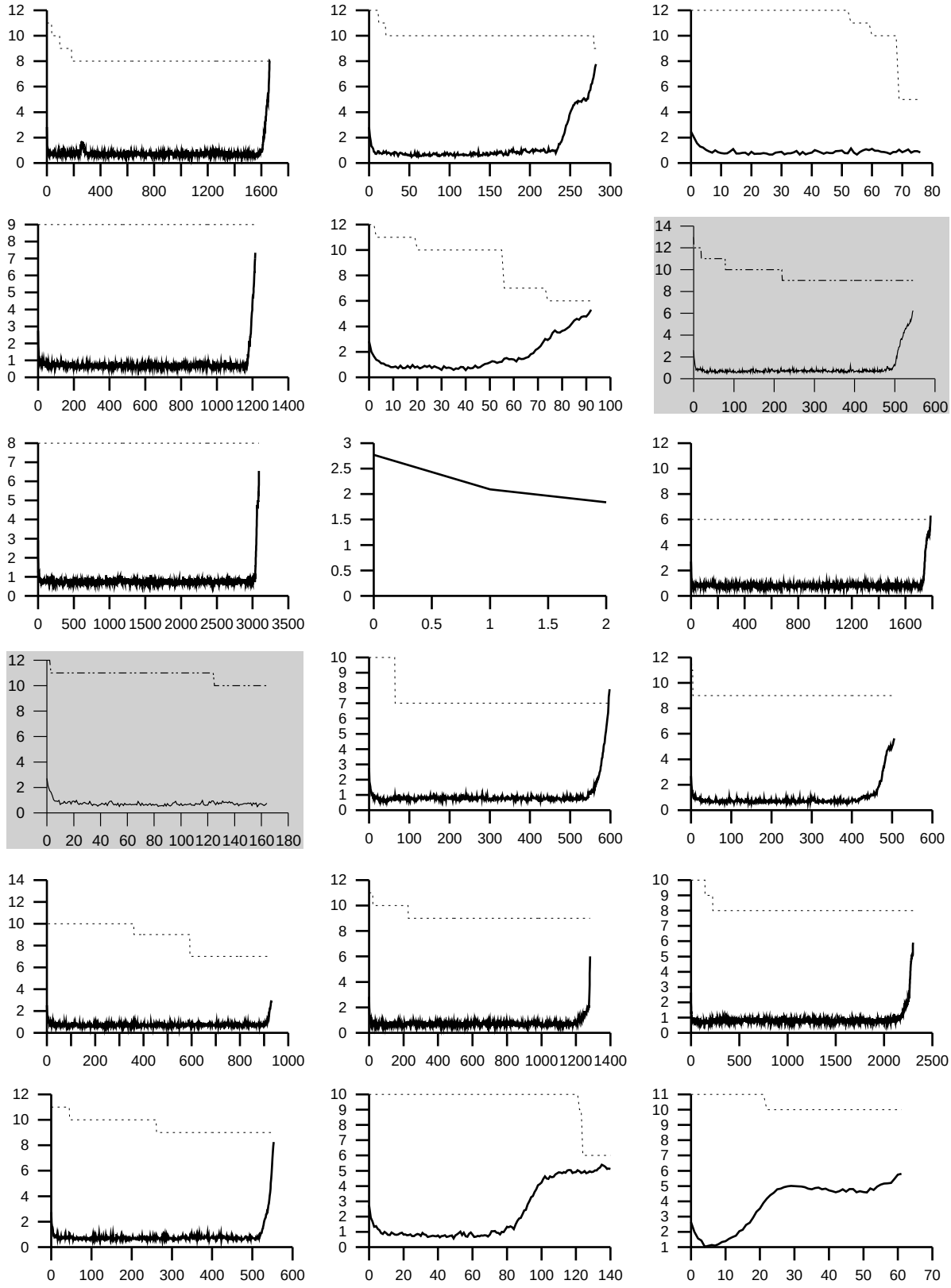
In der Tabelle ist eine Zusammenfassung der experimentellen Ergebnisse abgebildet. Alle Ergebnisse wurden über jeweils 50 Läufe gemittelt; jeder Lauf verwendete eine Populationsgröße von 500 Individuen. Die durschnitlliche Anzahl von Generationen je Lauf wird mit $\bar{g}$ bezeichnet, der Standardfehler mit $m_{\bar{g}}$. $\bar{t}$ gibt die Zeit in Minuten an, die im Mittel für einen Lauf benötigt wurde.

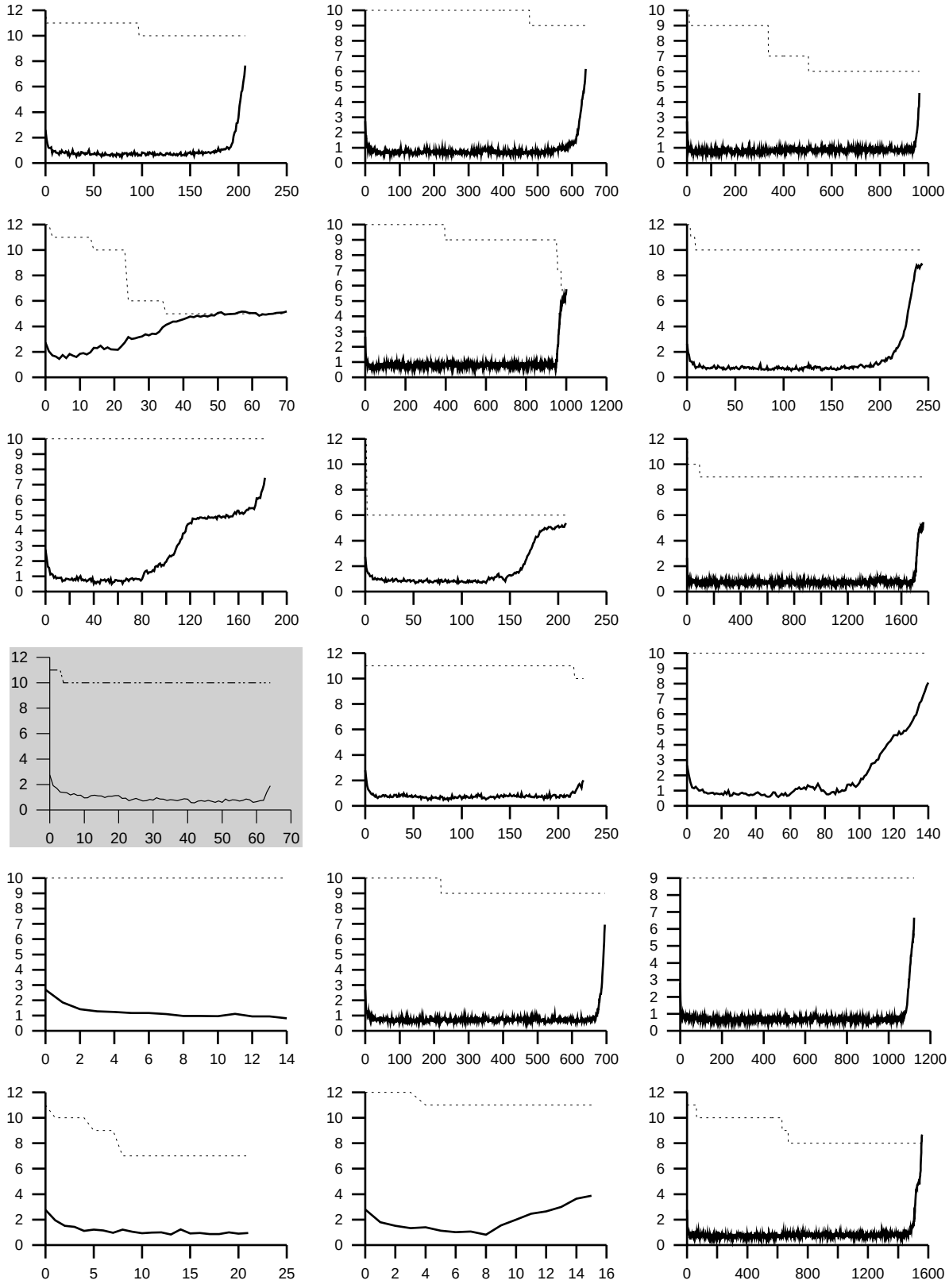
Kodierung	Problem	Crossover-operator	Selektions-mechanismus	$\bar{g}$	$m_{\bar{g}}$	$\bar{t}$
18 Bit	XOR_3	2-Punkt	Proportional, 1 Elitist	635.54	135.89	10.77
·	·	8-Punkt	·	829.20	201.09	14.03
·	·	Uniform	·	654.70	146.25	4.32
·	·	c_{NNF}	·	550.12	133.46	3.65
·	·	c_{Cells}	·	670.72	148.90	4.35
·	·	–	·	201.10	61.06	1.59
·	·	–	Linear Ranking, $\eta_{max} = 1.5$	52.70	9.08	0.74
·	·	–	q-Tournament, $q = 2$	30.14	5.12	0.63
·	·	–	EP q-Tournament, $q = 2$	22.24	3.63	0.59
·	·	–	EP q-Tournament, $q = 10$	100.14	53.23	1.07
16 Bit	XOR_3	–	EP q-Tournament, $q = 10$	41.34	6.94	0.19
·	XOR_4	–	·	399.6	83.06	2.78
9 Bit	XOR_3	–	EP q-Tournament, $q = 10$	5.58	1.07	0.05
·	XOR_4	–	·	22.12	3.53	0.24
·	XOR_5	–	·	152.36	24.29	0.11
·	XOR_6	–	·	71.46	10.66	1.58
·	XOR_7	–	·	122.00	18.65	1.47
·	XOR_8	–	·	211.30	23.71	4.60
·	$2{:}1MUX_1$	–	·	0.00	0.00	0.002
·	$2{:}1MUX_2$	–	·	14.32	2.69	0.04
Integer	XOR_3	–	EP q-Tournament, $q = 10$	4.46	0.83	0.01
·	XOR_4	–	·	20.12	2.96	0.05
·	XOR_5	–	·	43.14	6.21	0.16
·	XOR_6	–	·	81.74	11.85	0.46
·	XOR_7	–	·	171.06	25.24	1.54
·	XOR_8	–	·	438.98	70.52	6.78

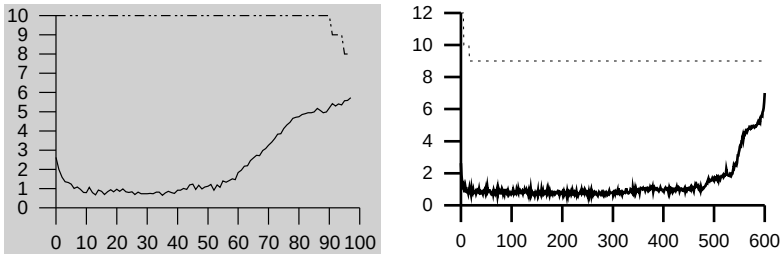
D Graphen

Bei dem Vergleich verschiedener Crossover-Operatoren in Abschnitt 5 wurde erwähnt, daß bei 5 von 50 durchgeführten Läufen in denen jeweils ein XOR_3-Schaltkreis evolutionär gefunden wurde, ein Anstieg der Standardabweichung σ zum Zeitpunkt der Konvergenz des GA beobachtet werden konnte. Die nachfolgenden Graphen zeigen die Fitneßverläufe jedes dieser 50 Evolutionsläufe. Jeder Graph stellt für jeweils einen der 50 Schaltkreise den Fitneßverlauf des jeweils besten Individuums einer Population als gestrichtelte Linie und den Verlauf der Standarbweichung σ als durchgezogene Linie dar.






0 50 100 150 200 250
0 100 200 300 400 500 600 700
0 200 400 600 800 1000
0 10 20 30 40 50 60 70
0 200 400 600 800 1000 1200
0 50 100 150 200 250
0 40 80 120 160 200
0 50 100 150 200 250
0 400 800 1200 1600
0 10 20 30 40 50 60 70
0 50 100 150 200 250
0 20 40 60 80 100 120 140
0 2 4 6 8 10 12 14
0 100 200 300 400 500 600 700
0 200 400 600 800 1000 1200
0 5 10 15 20 25
0 2 4 6 8 10 12 14 16
0 400 800 1200 1600



Die grau hinterlegten Graphen entsprechen den Läufen 18, 22, 40 und 49 der in Abbildung 17 auf Seite 60 bereits dargestellten XOR_3-Schaltkreise.

E Formeln

Hamming Distanz

Für zwei Bitvektoren (Bitstrings) $\vec{x}$ und $\vec{y}$ gleicher Länge l berechnet sich die Hamming-Distanz gemäß

$$d_H(\vec{x}, \vec{y}) = \sum_{k=1}^{l} |\vec{x}_k - \vec{y}_k|$$

wobei x_k die k-te Bitposition des Strings $\vec{x}$ bezeichnet.

Euklidische Distanz

Gegeben eine Punktmenge aus $\mathbb{N}^n$ mit allen n-Tupeln $x = (\xi_1, \ldots, \xi_N)$ ganzer Zahlen ξ_i. Ferner sei $y = (\eta_1, \ldots, \eta_n)$. Dann ist die euklidische Distanz definiert als

$$d_E(x, y) = \sqrt{\sum_{i=1}^{n} (\xi_i - \eta_i)^2}$$

und $\mathbb{N}^n$ wird bzgl. d_E zu einem metrischen Raum.

Standardabweichung

Für eine Meßreihe $x_1, \ldots, x_n$ ist der empirische Mittelwert definiert als

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

und für die Berechnung der Standardabweichung σ ergibt sich

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}.$$

Standardfehler

Der mittlere Fehler des Mittelwertes (Standardfehler) wird durch

$$m_{\bar{x}} = \frac{\sigma}{\sqrt{n}} = \sqrt{\frac{1}{n(n-1)} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

berechnet.

Abkürzungen

ASIC	Application Specific Integrated Circuit
BIOS	Basic Input/Output System
CLB	Configurable Logic Block
DCR	Device Control Register
DIR	Device Identification Register
DNS	Desoxyribonukleinsäure
DNF	Disjunktive Normalform
DoD	Department of Defense
EA	Evolutionärer Algorithmus
EEPROM	Electrically Eraseable PROM
EHW	Evolvable Hardware
EPROM	Erasable PROM
FPGA	Field Programmable Gate Array
GA	Genetischer Algorithmus
GClk	Global Clock
GP	Genetisches Programmieren (engl. Genetic Programming)
IOB	Input/Ouput Block
LSI	Large Scale Integration
NASA	National Aeronautics and Space Administration
PROM	Programmable Read Only Memory
PAL	Programmable Array Logic
PC	Personal Computer
PLA	Programmable Logic Array
PLD	Programmable Logic Device
RAM	Random Access Memory
RNS	Ribonukleinsäure
ROM	Read Only Memory
RPU	Reconfigurable Programming Unit
SAGA	Species Adaption Genetic Algorithm
SIA	Semiconductor Industry Association
SRAM	Static Random Access Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration

Referenzen

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press. ISBN 0-19-509971-0.

Bähring, H. (1994). *Mikrorechnersysteme* (2. Auflage ed.). Springer Verlag. ISBN 3-540-58362-9.

Banzhaf, W., P. Nordin, R. Keller, und F. Francone (1998). *Genetic Programming – An Introduction*. Academic Press/Morgan Kauffmann, San Francisco.

Chapman, G. H. (1997). Centre for systems science. Update newsletter: *Wafer scale integration*. http://fas.sfu.ca/css/update.html. vol. 2, no. 7.

de Garis, H., N. Petroff, M. Korkin, G. Fehr, und E. Nawa (1999). An artificial brain. *Connection Science Journal*. To be submitted.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM 15*(10), 859–866.

Droste, S. und D. Wiesmann (1998, Juli). On representation and genetic operators in evolutionary algorithms. Technical report, Research Center 531 „Computational Intelligence“ CI-41/98, Universität Dortmund.

Fogel, D. B. (1994). Evolutionary programming: An introduction and some current directions. *Statistics and Computing 4*, 113–129.

Fogel, D. B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Reading, MA. Addison Wesley.

Goldberg, D. E. und K. Deb (1990). A comparative analysis of selection schemes used in genetic algorithms. Technical Report TCGA-90007, TCGA, University of Alabama.

Graham, R., O. Patashnik, und D. E. Knuth (1994, März). *Concrete Mathematics* (2nd ed.). Addison-Wesley. ISBN 0201558025.

Harvey, I. (1992a). Evolutionary robotics and SAGA: The case for hill crawling and tournament selection. In C. Langton (Ed.), *Artificial Life 3*, Band XVI. Santa Fe Institute.

Harvey, I. (1992b). Species adaption genetic algorithms: A basis for a continuing SAGA. In F. J. Varela und P. Bourgine (Eds.), *Towards a Practice of Autonomous Systems*, Proceedings 1st Eur. Conference on Artificial Life, S. 346–354. MIT Press.

Hirsch-Kaufmann, M. und M. Schweiger (1992, 2. Auflage). *Biologie für Mediziner und Naturwissenschaftler*. Stuttgart: Georg Thieme Verlag.

Holland, J. H. (1975). *Adaption in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press.

Igel, C. und K. Chellapilla (1999). Fitness distributions: Tools for designing efficient evolutionary computations. In L. Spector, W. B. Langdon, U.-M. O'Reilly, und P. J. Angeline (Eds.), *Advances in Genetic Programming 3*, Chapter 9. MIT Press.

Kajitani, I., T. Hoshino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, M. Iwata, D. Keymeulen, und T. Higuchi (1998). A gate-level EHW chip: Implementing GA operations and reconfigurable hardware on a single LSI. In *Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science 1478, S. 1–12.

Kajitani, I., M. Murakawa, D. Nishikawa, H. Yokoi, N. Kajihara, M. Iwata, D. Keymeulen, H. Sakanashi, und T. Higuchi (1999). An evolvable hardware chip for prosthetic hand controller. In *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy, and Bio-Inspired Systems (MicroNeuro99)*, S. 179–186.

Keller, E. F. und E. A. Lloyd (Eds.) (1992). *Keywords in evolutionary biology*. Harvard University Press.

Keymeulen, D., K. Konaka, M. Iwata, Y. Kuniyoshi, und T. Higuchi (1998). Robot learning using gate-level evolvable hardware. In A. Birk und J. Demiris (Eds.), *Proceedings of the 6th European Workshop on Learning Robots*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

Kitano, H. (1996). Challenges of evolvable systems: Analysis and future directions. In *International Conference On Evolvable Systems*, S. 125–135. ICES96.

Kolla, R., P. Molitor, und H. G. Osthof (1989). *Einführung in den VLSI-Entwurf*. Teubner.

Korkin, M., H. de Garis, F. Gers, und H. Hemmi (1997). CBM (CAM-Brain Machine). In *Genetic Programming 1997*, S. 498–503.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing 4*, 87–112.

Kreutz, M. (1999). *EALib: A C++ class library for evolutionary algorithms*. ftp://ftp.neuroinformatik.ruhr-uni-bochum.de/pub/software/EA.

Lamport, L. (1994). *LaTeX, A Document Preparation System, User's Guide and Reference*. Addison-Wesley Publishing Company, 2nd ed.

Mange, D. und A. Stauffer (1994). Introduction to embryonics: Towards new self-repairing and self-reproducing hardware based on biological-like properties. *Artificial Life and Virtual Reality*, 61–72.

Miller, J. F. und P. Thomson (1998, September). Aspects of digital evolution: Geometry and learning. In M. Sipper, D. Mange, und A. Perez-Uribe (Eds.), *2nd International Conference on Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science (LCNS1478), Lausanne, Switzerland, S. 25–35. Springer Verlag.

Moore, G. E. (1975). Technical digest 7511. *IEEE International Electronic Devices Mtg (IEDM)*. Piscataway, New Jersey.

Muller, D. A., T. Sorsch, S. Moccio, F. H. Baumann, K. Evans-Lutterodt, und G. Timp (1999). The electronic structure at the atomic scale of ultrathin gate oxides. *Nature 399*, 758–761. Letters to Nature.

NASA/DoD (1999, 7). 1st NASA/DoD workshop on evolvable hardware (EH99). http://cism.jpl.nasa.gov/events/nasa_eh. Call for Papers.

Rechenberg, I. (1994). *Evolutionsstrategie '94*, Band 1 von *Werkstatt Bionik und Evolutionstechnik*. Frommann-Holzboog.

Sanchez, E., D. Mange, M. Sipper, M. Tomassini, A. Perez-Uribe, und A. Stauffer (1997). Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In T. Higuchi, M. Iwata, und L. Weixin (Eds.), *International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, Heidelberg, S. 35–54. Springer-Verlag.

Schwefel, H. P. (1995). *Evolution and Optimum Seeking*. 6th-Generation Computer Technologiy Series. New York: Wiley & Sons Inc.

Schwefel, H.-P. und F. Kursawe (1998, May 4-9). On natural life tricks' to survive and evolve. In *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC'98)*, Anchorage, S. 1–8. IEEE.

Sendhoff, B., M. Kreutz, und W. von Seelen (1997). A condition for the genotype-phenotype mapping: Causality. In T. Bäck (Ed.), *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA97)*, San Francisco, S. 73–80. Morgan Kauffmann.

Sipper, M., D. Mange, und A. Stauffer (1997). Ontogenetic hardware. *BioSystems* (44), 193–207.

Sipper, M., E. Sanchez, D. Mange, M. Tmassini, A. Perez-Uribe, und A. Stauffer (1997). A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Trans. Evol. Comput. 1*(1), 83–97.

Spencer, H. (1864). *Principles of Biology*. London: Williams and Norgate. 2 vols.

Stiller, A. (1997). Bug-Wahn, Pentium: Absturz-Loch. *c't 14*, 14.

Stiller, A. (1999). Prozessorgeflüster. *c't 14*, 24.

Stoica, A., G. Klimeck, C. Salazar-Lazaro, D. Keymeulen, und A. Thakoor (1999, Juli). Evolutionary design of electronic devices and circuits. In *1999 Congress on Evolutionary Computation*, Band 2, Mayflower Hotel, Washington D.C., S. 1271–1278.

Syswerda, G. (1989, 4.–7. Juli). Uniform crossover in genetic algorithms. In J. D. Schaffer (Ed.), *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA'89)*, San Mateo, CA, S. 2–9. Morgan Kaufmann Publishers, Inc.

Tangen, U. und J. S. McCaskill (1998). Hardware evolution with a massively parallel dynamicallly reconfigurable computer: POLYP. In *Proceedings 2nd International Conference on Evolvable Systems (ICES98)*, S. 364–371. Springer LNCS.

Tempesti, G. (1995). A new self-reproducing cellular automaton capable of constuction and computation. In F. Moran, A. Moreno, J. J. Merelo, und P. Chacon (Eds.), *ECAL'95: 3rd Eur. Conference on Artificial Life*, S. 555–563. Springer LNCS (929). Proc 3rd ECAL (ECAL95).

Thompson, A. (1996). An evolved circuit, intrinsic in silicon, entwined with physics. In *Proceedings 1st International Conference on Evolvable Systems (ICES96)*. Springer LNCS.

Thompson, A. (1998). *Hardware Evolution. Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. Ph. D. thesis, School of Cognitive and Computing Sciences, University of Sussex. Springer-Verlag London Ltd.

Thompson, A. (1999, 2. Juni). Evolutionary design for novel technologies. In *IEE Colloquium on Evolutionary Hardware Systems*, Savoy Place, London, S. 4/1.

Thompson, A. und P. Layzell (1999, April). Analysis of unconventional evolved electronics. *Communications of the ACM 42*(4), 71–79.

Tietze, U. und C. Schenk (1989). *Halbleiter-Schaltungstechnik*. Springer-Verlag Berlin.

Tomassini, M. und M. Sipper (1997, 3). An introduction to evolvable hardware. *EvoNews Issue 3*, 8–10.

Urban und Schwarzberg (1995, 11). Roche Lexikon Medizin. CD-ROM Version 3.5.

Vassilev, V. K., J. F. Miller, und T. C. Fogarty (1999). Digital circuit evolution and fitness landscapes. In *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC'99)*, Band 2, S. 1299–1306. IEEE.

Wegener, I. (1989). *Effiziente Algorithmen für grundlegende Funktionen*. Stuttgart: Teubner.

Xilinx Inc. (1997, 4). *Programmable Logic Data Book* (v1.10 ed.). Xilinx Inc. XC6200 Field Programmable Gate Arrays.

Yao, X. und T. Higuchi (1999, February). Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews 29*(1), 87–97.

Colophon

Diese Arbeit wurde mit dem LaTeX Textsatz-System in Version LaTeX 2_ε vom 1. Juni 1996 erstellt (Lamport 1994). Die Grundlage allen visuellen Übels bildete der Dokument-Stil article, wobei einige Eingriffe bzgl. der Randeinstellungen, Kopfzeilen, etc. notwendig waren, um dieses Layout zu verwirklichen. Bei dem verwendeten Zeichensatz handelt es sich nicht um eine Times (New) Roman Variante, sondern um ein Kind der Palatino-Familie. Alle mathematischen Schriften und Symbole werden mit dem erstmals im Buch Concrete Mathematics von Graham et al. (1994) vorgestellten Euler-Zeichensatz dargestellt. Als Erweiterungspakete kamen graphicx, euler, chicago, algorithm, amsmath, pstricks, pstnode, hhline, ifthen, calc, wrapfig sowie makeidx zum Einsatz. Weiterhin das Paket code, welches eine leicht abgewandelte Form des algorithm Paketes darstellt und zur Visualisierung der C++ Beispiele dient.

Der Großteil aller Abbildungen und Grafiken wurde mit dem vektororientierten Zeichenprogramm `xfig` in Version 3.2 (patchlevel 2) erstellt. Graphen und Plots sind das Ergebnis von PAGODE v2.0β, einem Paket zur Datenvisualisierung im zwei- und dreidimensionalen Raum sowie `gnuplot` von Thomas Williams et al. in der Linux Version 3.7.

Zur Extraktion und zum Nachbearbeiten einiger Grafiken aus dem nur als PDF-Dokument vorliegenden *Programmable Logic Data Book* von Xilinx Inc. (1997) wurde folgender Ansatz gewählt: Konvertierung des kompletten Data Book ins Postscript-Format, z. B. mit Ghostscript von Aladdin Enterprises. Abspeichern der Seite, die die gewünschte Grafik enthält, in ein separates Postscript-File (Frontend Ghostview oder `gv` für Ghostscript, alternativ `psselect`). Anschließend Transformierung der einzelnen Seite mittels `pstoedit` (mit Backend Ghostscript in Version 5.10) von Wolfgang Glunz in das `fig`-Format des vektororientierten Sketch-Programmes `xfig`. Mit `xfig` kann die Grafik beliebig nachbearbeitet und in verschiedenen Formaten, bspw. Postscript, gespeichert werden.

Die Implementierung des Genetischen Algorithmus erfolgte unter Zuhilfenahme der von Martin Kreutz (1999) entwickelten EALib, einer ausgesprochen elaborierten C++ Bibliothek für die Programmierung Evolutionärer Algorithmen. Adrian Thompson stellte freundlicherweise seinen C-Code zur Visualisierung instantiierter Schaltkreise zur Verfügung.

Index

Notizen