Institut für Informatik
Universität Würzburg

Ein generisches Konzept zur Modellierung und Bewertung feldprogrammierbarer Architekturen

Dissertation

zur Erlangung des naturwissenschaftlichen Doktorgrades

der Bayerischen Julius-Maximilians-Universität Würzburg

vorgelegt von

Frank Wolz

aus

Wertheim am Main

Würzburg 2003

Eingereicht am 25. August 2003
bei der Fakultät für Mathematik und Informatik

Erstgutachter: Prof. Dr. Reiner Kolla
Zweitgutachter: Prof. Dr. Paul Molitor

Tag der mündlichen Prüfung: 4. Februar 2004

Ernst ist die Kunst – heiter das Leben.
Max Reger (1873-1916) an seinen Lehrer Adalbert Lindner.

Danksagung

Mein besonderer Dank gilt zunächst meinem Lehrer, Herrn Prof. Dr. Reiner Kolla, für die fachliche Betreuung dieser Arbeit. Im Laufe ihrer Entstehung hatte er stets ein offenes Ohr für meine Fragen; nie hat er mich abgewiesen, wenn ich ihn um seinen fachlichen Rat gebeten hatte. Dabei ließ er dennoch durch eine distinguiert ausgeübte Zurückhaltung meiner Kreativität großen Raum zur Entfaltung, was maßgeblich beitragend zu einem Arbeitsumfeld war, in dem ich mich stets neu motivieren konnte, war ich doch in diesem umfangreichen Projekt von Anfang an als Einzelkämpfer unterwegs.

Teil dieses Arbeitsumfeldes bildeten auch meine geschätzten Kollegen, allen voran Markus Wild, der mich seit Beginn meiner Tätigkeit am Lehrstuhl für Technische Informatik auf meinem Weg begleitete und, obgleich thematisch auf einem anderen Pfad unterwegs, durch sein häufiges Interesse an meiner Arbeit in mir etliche neue Ideen auslöste. Auch meine Kollegen Holger Englert, Dr. Winfried Nöth und Dr. Franz Duckstein, mit denen ich mehr oder weniger kürzere Zeitabschnitte zusammenarbeiten durfte, lieferten mir immer wieder wertvolle Impulse, insbesondere auch, was meine Tätigkeit in der Lehre anbetraf. Danken möchte ich auch Frau Gisela Hoppe, die neben ihrer Verwaltungstätigkeit im Sekretariat des Lehrstuhls stets für ein freundliches Wort zu erreichen war und mich damit den Blick auch auf die menschliche Seite meines Umfeldes nicht verlieren ließ.

Abschließend möchte ich noch den vielen Menschen aus meinem privaten Kreis danken, die mir bewußt oder unbewußt seelische Unterstützung in den letzten Jahren haben zuteil werden lassen. Meinem Freundeskreis, unter ihnen mein langjähriger *spiritus rector*, Herr Pfarrer Reinhard Hausmann, meine liebe Freundin Anja, vor allem aber zwei Personen: meinen Eltern, dank deren uneingeschränkten familiären Rückhalt, sowie deren ideeller, finanzieller und seelischer Unterstützung ich den Weg bis dahin überhaupt gehen konnte.

Inhaltsverzeichnis

Einleitung

Digitalfernsehen, DVD-Spieler, Mobiltelefon und Handheld Computer – Beispiele aus unserer heutigen digitalen Welt der Medien, bei deren Konstruktion man nicht mehr auf sie verzichtet: *rekonfigurierbare Mikrochips*. Unter der Bezeichnung *Field-Programmable Gate Arrays (FPGAs)* bekannt, haben sie seit ihrer Einführung, etwa um das Jahr 1985, einen wahren Siegeszug angetreten, wobei deren eigentliche Ära erst vor einigen Jahren, nicht zuletzt forciert durch den Aufstieg der digitalen Kommunikation, richtig begonnen hat. Als die ersten FPGA-Prototypen in den 1980er Jahren gefertigt wurden, ahnte noch niemand, wie wichtig programmierbare Logik für den technologischen Fortschritt in der heutigen Zeit sein wird. Mancher sieht gar eine innovative High-Tech-Entwicklung ohne FPGAs gar nicht mehr realisierbar.

In der Tat wird die Rekonfigurierbarkeit als die „Kultur des Hardwareentwurfes“ des Jahrzehnts betrachtet, welches sich nach der *Makimoto'schen Welle*[1] gegenwärtig aufspannt. Die Makimoto'sche Welle, benannt nach dem Unternehmensberater des japanischen Sony-Konzerns, Dr. Tsugio Makimoto, der sie im Jahre 1991 formulierte, beschreibt, beginnend im Jahre 1957, die zyklische Natur der Halbleiterentwicklung, welche zwischen Standardisierung und Kundenorientierung in Zehnjahresrhythmen fluktuiert (Abbildung 1).

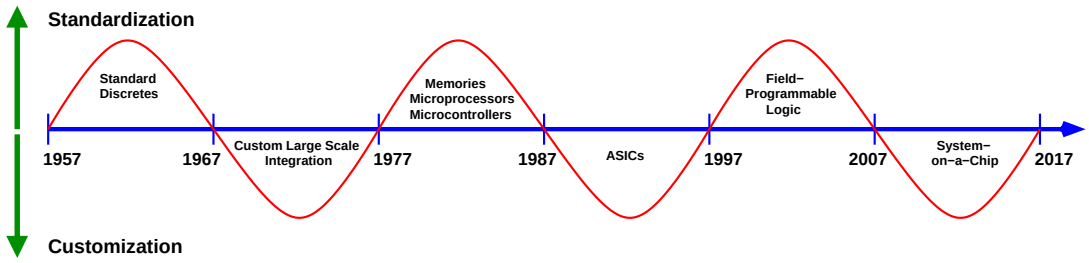


Abbildung 1: Die Makimoto'sche Welle

Von 1957 bis 1967 wurde demnach Hardwareentwurf mit diskreten Bauelementen, wie Bipolartransistoren, betrieben. Von 1967 bis 1977 wurden kleine anwendungsspezifische Mikrochips beispielsweise in Fernsehern und Taschenrechnern eingesetzt. In der Dekade bis 1987 traten wiederum standardisierte Mikroprozessoren und Speicher in den Vordergrund. Mit der bis dahin fortgeschrittenen Entwicklung der VLSI-Chiptechnologie *(Very Large Scale Integration)*, begann 1987 die Ära der *Application Specific Inte-*

[1] Tsugio Makimoto: *The Rising Wave of Field-Programmability*, Keynote-Vortrag im Rahmen der Konferenz *Field-Programmable Logic and Applications*, Villach, Österreich, 2000

grated Circuits (ASICs). Seit 1997 befindet sich die Halbleiterentwicklung gemäß Makimoto's Theorie nun in der Phase der Feldprogrammierbarkeit, welche hinsichtlich Fertigung zwar durch Standardisierung, hinsichtlich der Anwendung jedoch kundenspezifisch individuell geprägt ist.

Betrachtet man zumindest den Halbleitermarkt der kundenspezifischen *Gate-Array*-Chips und der FPGAs über die letzten 13 Jahre hinweg, scheint sich die Theorie, obgleich mit etwas Verzögerung, doch wieder zu bewahrheiten (siehe Abbildung 2, Zahlen von 2000 bis 2003 geschätzt). Was Marktanalysten Anfang der 1990er Jahre noch

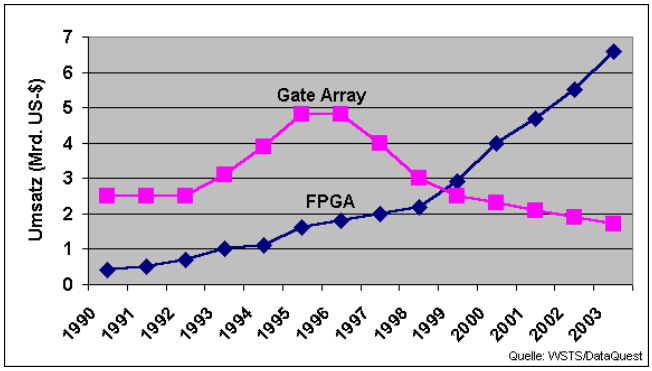


Abbildung 2: Marktentwicklung von Gate Arrays und FPGAs

geschätzt hatten, nämlich daß der jährliche Umsatz der programmierbaren Logik gegen Ende des Jahrtausends die Grenze von einer Milliarde US-Dollar erreichen wird, wurde weitaus übertroffen.

Doch rekonfigurierbaren Architekturen scheint auch weiterhin die Zukunft zu gehören, denn selbst nach dem Ende der Phase der feldprogrammierbaren Logikchips in Makimoto's Welle, werden diese immer noch eine zentrale Rolle spielen, als Komponente der dann vorherrschenden Technologie, der *Configurable Systems-on-a-Chip (CSoC)*.

Problemstellung

Im Laufe des sicher mittlerweile weit über 20 Jahren andauernden Entwicklungszeitraumes, wurden zahlreiche feldprogrammierbare Architekturen konzipiert und produziert, wobei sich auf kommerzieller Ebene schließlich nur wenige wirklich durchsetzten. Bei diesen wenigen Architekturen änderte sich dann auch deren grundlegende Struktur kaum noch, sondern sie wurden im wesentlichen nur in kapazitärer Hinsicht erweitert, soweit durch bessere Fertigungsprozesse mehr Chipfläche zur Verfügung stand, beziehungsweise der Markt über anvisierte Anwendungen größere Chipvarianten erforderlich machte. Daneben wurden insbesondere Anfang bis Mitte der 1990er Jahre Untersuchungen über die Skalierung der Strukturen der Architekturen durchgeführt, inwieweit sie auf Platz- und Performanzwerte Einfluß besitzt. Doch erst in einer jüngeren Studie [91] wurde die Aussagekraft gerade solcher Architekturstudien kritisch untersucht und zum Teil auch in Frage gestellt.

Nicht zuletzt aufgrund dessen, daß sich viele der publizierten Studien über feldprogrammierbare Architekturen ohnehin stets mit einer, derselben kommerziellen Archi-

tektur beschäftigen, war diesem Forschungsbereich bislang immer eine besondere Problematik zueigen: Im wesentlichen wurden Verfahren zur Realisierung von Logikschaltkreisen in FPGAs unter dem Gesichtspunkt entwickelt, die Ressourcen einer speziellen, gegebenen FPGA-Architektur am besten auszunutzen. Fragestellungen hinsichtlich des tatsächlichen Bedarfs an struktureller Flexibilität unterblieben, der Vergleich entsprechend verschiedener Architekturen fehlte.

Tabelle 1 zeigt exemplarisch anhand der jährlich stattfindenen internationalen Konferenz *FPGA* eine Kategorisierung der dort in den Jahren 1995 bis 2003 publizierten Architekturstudien. Die Spalte *Allg.* der Tabelle enthält die absolute, sowie prozentuale Anzahl von Arbeiten, die sich allgemein mit dem Entwurf von Architekturen befaßten, also auf keiner konkreten Architektur basierten. Die folgenden beiden Spalten beziehen sich auf die Studien, denen Architekturen zweier bekannter kommerzieller Anbieter zugrundegelegt wurden. In der fünften Spalte sind die Publikationen über andere Architekturen zusammengefaßt. Insgesamt wurden im Betrachtungszeitraum 214 Arbeiten, davon 45 Architekturstudien, bei der Konferenz vorgestellt. Die beiden Zeilen der prozentualen Werte beziehen sich jeweils auf diese beiden Summen.

	Allg.	Kommerz. 1	Kommerz. 2	Andere
Anzahl	5	21	4	15
Arch.stud. (%)	11.1	46.7	8.9	33.3
Gesamt (%)	2.3	9.8	1.9	7

Tabelle 1: Publizierte Architekturstudien bei der internat. Konferenz „FPGA“

Die Zahlen sprechen eine deutliche Sprache: mehr als die Hälfte der Architekturstudien befaßten sich mit den Produkten kommerzieller Anbieter, lediglich ein Drittel der Arbeiten führte neue Architekturen ein. Mit rund elf Prozent dagegen relativ gering stellt sich der Anteil allgemeiner Betrachtungen zum Entwurf von Architekturen dar.

Auf dem Hintergrund der oben beschriebenen Beobachtungen entstand schließlich die Idee zum Thema der vorliegenden Arbeit, deren Zielsetzung denn auch in der Bereitstellung eines Architekturmodells für feldprogrammierbare Architekturen, sowie der Entwicklung von, auf dem Modell basierenden, generischen Verfahren zur Realisierung von Schaltkreisen besteht und ferner auch die Definition geeigneter Metriken zur Bewertung von Architekturen umfaßt.

Die Anwendung eines *generischen* Layout-Werkzeuges für Schaltkreise soll bei empirischen Untersuchungen verschiedener feldprogrammierbarer Architekturen für eine gewisse „Konstanz der Bedingungen“ sorgen, insofern Initialstruktur der Schaltkreise, Layout-Strategien und deren Parametrisierung, experimentelle und Architektur-Voraussetzungen unverändert bleiben.

Vorgehensweise

Die vorliegende Arbeit gliedert sich in die folgenden Einzelkapitel: Kapitel 1 führt zunächst in die grundlegenden und zum Verständnis der vorliegenden Arbeit notwendigen Begriffe ein, gibt dann einen kurzgefaßten Überblick über die Technologien integrierter Schaltkreise, sowie Problematiken beim Hardwareentwurf und führt dann in den Bereich der feldprogrammierbaren Logik ein. Abschließend wird der in dieser Arbeit verfolgte, neue Ansatz genauer motiviert und erläutert.

Ein auf der Basis konfigurierbarer Multiplexer-Netzwerke basierendes, neues Modell für feldprogrammierbare Architekturen wird in Kapitel 2 vorgestellt, wobei auch die Realisierung von Schaltkreisen auf solchen Architekturen begrifflich eingeführt wird. Den Abschluß des Kapitels bildet ein Abschnitt, der verschiedene Bewertungsmetriken motiviert.

Die beiden Kapitel 3 und 4 beschäftigen sich mit algorithmischen Verfahren als Komponenten eines generischen Layoutsystems für Logikschaltkreise. Zunächst wird eine neue Partitionierungsmethode für Schaltkreise, sowie ein neuer *Route-then-Place*-Ansatz zur Generierung von Subschaltkreis-Realisierungen vorgestellt, während in Kapitel 4 eine neue Floorplanning-Methode und eine Heuristik für Subschaltkreis-Verdrahtungen eingeführt wird.

Eine Reihe von insgesamt neun Architekturen wird schließlich in Kapitel 5 vorgestellt, welche sodann mittels der in Kapitel 2 definierten Bewertungsmetriken unter Zuhilfenahme des entwickelten generischen Layoutsystems einander gegenübergestellt werden. Den Abschluß des Kapitels und der vorliegenden Arbeit bildet ein zusammenfassender Überblick, sowie ein Ausblick über mögliche künftige Arbeiten in diesem Bereich.

Abbildung 3 skizziert nochmals das Gesamtkonzept der vorliegenden Arbeit, welches schrittweise in den einzelnen Kapiteln realisiert wird.

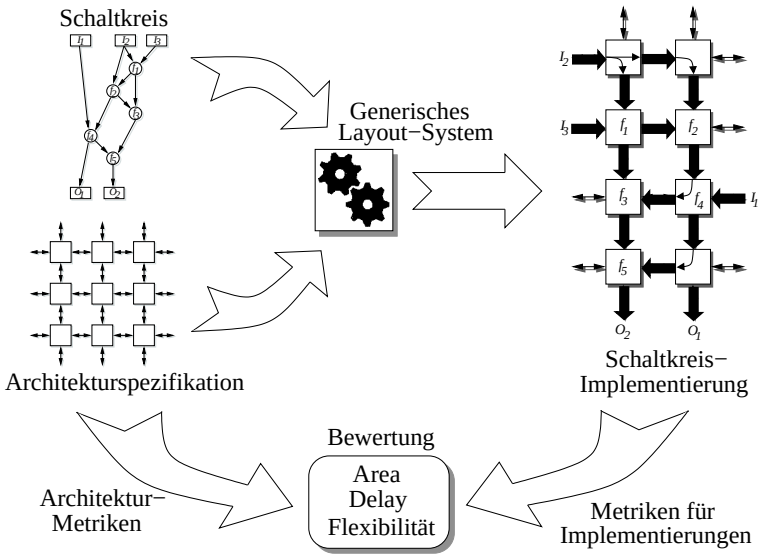


Abbildung 3: Konzeption zur Bewertung feldprogrammierbarer Architekturen

Kapitel 1

Grundlagen

Dieser Abschnitt führt in die wichtigsten Definitionen und Notationen ein, die im weiteren Verlauf der Arbeit zur Anwendung kommen werden. Die einzelnen Sachverhalte werden hierbei jedoch nicht Gegenstand genauerer Untersuchungen sein, sondern lediglich im Hinblick auf ihren späteren Einsatz dargestellt. Im Einzelfall wird auf entsprechende Standardwerke im Literaturverzeichnis verwiesen.

1.1 Probleme und deren Komplexität

Definition 1.1 (Algorithmisches Problem)
Ein *algorithmisches Problem* Π ist eine Abbildung $\Pi : I \to \mathcal{P}(S)$, wobei I die Menge der Probleminstanzen und S die Menge der Konfigurationen ist.

Zu einem Problem[1] $p \in I$ ist $\Pi(p)$ die Menge der *zulässigen Lösungen*. I und S können auch unendlich große Mengen sein. Falls $\#S = 1$, heißt Π ein *Entscheidungsproblem*.

Definition 1.2 (Optimierungsproblem)
Ein *Optimierungsproblem* ist ein algorithmisches Problem, für das eine Kostenfunktion $c : S \to \mathbb{N}$ existiert. Zu einem Problem $p \in I$ heißt $s^{\text{opt}} \in \Pi(p)$ eine *optimale Lösung*, wenn $c(s^{\text{opt}})$ ein globales Extremum ist.

Im folgenden gehen wir, wenn nichts anderes gesagt ist, stets von Minimierungsproblemen aus. Ein Algorithmus A *löst* ein algorithmisches Problem Π, wenn er auf die Eingabe einer Codierung einer Probleminstanz $p \in I$ die Codierung einer zulässigen Lösung berechnet. Die zur Berechnung benötigte Zeit wird mit $T(A, p)$, der zur Berechnung benötigte Speicherplatz wird mit $S(A, p)$ bezeichnet.

Die Bestimmung der Komplexitätsgrößen $T(A, p)$ und $S(A, p)$ ist ein im allgemeinen sehr schwieriges Problem. Deshalb charakterisiert man das Verhalten eines Algorithmus bezüglich Speicherbedarf und Laufzeit durch Funktionen, welche man auf eine oder mehrere Kenngrößen der Probleminstanzen anwendet. Solche Kenngrößen stehen für gewöhnlich eng in Beziehung zur Länge der Codierung einer Probleminstanz.

[1] Die Definitionen orientieren sich hier an [50].

Dadurch erhält man eine Aussage, wie sich die Komplexität eines Problems verändert, wenn seine Größe zunimmt. Eine mittlerweile sehr gängige Charakterisierung wurde 1976 von Donald E. Knuth ([50], Lit. 237) vorgestellt, sie wird auf Laufzeit und Speicherbedarf gleichermaßen angewandt:

Definition 1.3 ($\mathcal{O}, \Omega, \Theta$)
Sei R die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}^+$. Dann definiert eine Funktion $g \in R$ die folgenden Funktionsklassen:

$$\begin{aligned}
\mathcal{O}(g) &= \{\, f \in R \mid \exists_{c\in\mathbb{R}^+}\, \exists_{n_0\in\mathbb{N}}\, \forall_{n\geq n_0} \quad f(n) \leq c \cdot g(n) \,\} \\
\Omega(g) &= \{\, f \in R \mid \exists_{c\in\mathbb{R}^+}\, \exists_{n_0\in\mathbb{N}}\, \forall_{n\geq n_0} \quad f(n) \geq c \cdot g(n) \,\} \\
\Theta(g) &= \mathcal{O}(g) \cap \Omega(g)
\end{aligned}$$

Demnach bezeichnet man einen Algorithmus als von *polynomieller Komplexität* bzgl. Laufzeit oder Speicherplatzbedarf, wenn die Funktion f der Laufzeit bzw. des Speicherplatzbedarfs durch ein Polynom nach oben beschränkt ist, also ein Polynom g existiert, so daß $f \in \mathcal{O}(g)$. Ein Algorithmus heißt hingegen von *exponentieller Komplexität*, falls $f \in \mathcal{O}(2^g)$. Hat man nun eine Funktion f, so daß die Komplexität eines Algorithmus für alle Eingaben der Länge n durch $f(n)$ nach oben beschränkt ist, so bezeichnet man den Algorithmus als von der *worst-case Komplexität* $\mathcal{O}(f)$.

Wie bestimmt sich nun eine solche Funktion f? In der Vergangenheit wurden hierzu verschiedene Berechnungsmodelle, wie z.B. Turing-Maschinen eingeführt, die allerdings — in Analogie zur *These von Church* — durch Simulation unter höchstens polynomiellem Mehraufwand sämtlichst ineinander überführbar sind. So insbesondere auch das Berechnungsmodell der RAM (*Random Access Machine*), welches sich wohl aufgrund seiner Eigenschaft einer akkurateren Darstellung heutiger Algorithmen durchgesetzt hat [61]. Da jedoch das klassische Modell der RAM in einem Berechnungsschritt Operationen mit unendlich großen Zahlen durchführen kann (*unit-cost RAM*), gelang der Äquivalenzbeweis zur Turing-Maschine erst, indem die Operanden mit der Länge ihrer Codierung, also dem Logarithmus angesetzt wurden (*log-cost RAM*). In der Praxis der Komplexitätsanalyse ist die Anwendung des einfacheren Unit-Cost-Modells jedoch gerechtfertigt, da, gerade im Hinblick auf heutige Rechenanlagen, eine polynomielle Beschränktheit der Codierungslänge der Operanden in der Regel ohnehin gegeben ist.

Während das Berechnungsmodell der *deterministischen RAM* die sequentielle Abarbeitung einer gegebenen Folge von Anweisungen realisiert, ist das Modell der *nichtdeterministischen RAM* in der Lage, in jedem Berechnungsschritt die korrekte nachfolgende Anweisung, welche zur Lösung des Problems führt, zu raten.

Unter Zugrundelegung des Berechnungsmodells der RAM sind wir nun in der Lage, Entscheidungsprobleme bezüglich ihrer Laufzeitkomplexität zu klassifizieren:

Definition 1.4 (Komplexitätsklassen P und NP)
Die Klasse P ist definiert als die Menge aller Entscheidungsprobleme, die mittels einer deterministischen (log-cost) RAM in polynomieller Laufzeit gelöst werden können. Die Klasse NP ist definiert als die Menge aller Entscheidungsprobleme, die mittels einer nichtdeterministischen (log-cost) RAM in polynomieller Laufzeit gelöst werden können.

Diese Klassifikation läßt sich jedoch auch auf Optimierungsprobleme $\Pi : I \to \mathcal{P}(S)$ mit Kostenfunktion $c : S \to \mathbb{N}$ übertragen, indem entsprechende *Schwellwertprobleme* $\Pi_{\text{TH}} : I \times \mathbb{N} \to \mathbb{B}$ konstruiert werden, wobei dann für jedes $p \in I$ gilt:

$$\exists_{k \in \mathbb{N}}\ \Pi_{\text{TH}}(p,k) = 1 \iff \exists_{s \in \Pi(p)}\ c(s) \leq k$$

Ein zwar anschaulicher, jedoch auch wichtiger Zusammenhang zwischen den Klassen P und NP besteht darin, daß zu jedem Problem aus der Menge NP ein Polynom q existiert, so daß das Problem durch einen deterministischen Algorithmus in Laufzeit $\mathcal{O}(2^q)$ lösbar ist. Ist zu einem Problem bekannt, daß es in P liegt, so bezeichnet man es als „leicht". Probleme aus NP, von denen dies nicht bekannt ist, werden gemeinhin als „schwer" oder als „nicht traktabel", mit anderen Worten, als „unlösbar" bezeichnet. Derart „unlösbare" Probleme werden in der Praxis stattdessen durch spezielle deterministische Polynomialzeitverfahren, *Heuristiken*, bearbeitet, die versuchen, Lösungen zu approximieren.

Interessant ist nun jedoch, wie man entscheiden kann, ob ein neues Problem einen traktablen oder einen nicht traktablen Hintergrund besitzt. Mit einem Ansatz über die *relative Berechenbarkeit* fand man einen geeigneten Weg, Probleme in ihrer Komplexität direkt vergleichen zu können, nämlich, indem sie durch Transformationen von höchstens polynomiellen Mehraufwand ineinander überführt wurden:

Definition 1.5 (Polynomialzeit-Reduzierbarkeit)
Ein Entscheidungsproblem $\Pi_1 : I_1 \to \mathbb{B}$ heißt *Polynomialzeit-reduzierbar* auf ein Entscheidungsproblem $\Pi_2 : I_2 \to \mathbb{B}$, falls es einen Algorithmus $A : I_1 \to I_2$ mit polynomieller Laufzeit gibt, so daß für jedes $p \in I_1$ gilt:

$$\Pi_1(p) = \Pi_2(A(p))$$

Ein Problem, auf welches ein Problem Π Polynomialzeit-reduzierbar ist, kann somit intuitiv als „mindestens so schwer" wie Π bezeichnet werden. Eine Klasse der schwierigsten Probleme in NP bildet nun die Menge der *NP-vollständigen* Probleme:

Definition 1.6 (NP-vollständig)
Ein Problem Π heißt NP-vollständig, wenn jedes Problem aus NP auf Π Polynomialzeit-reduzierbar ist.

Nachdem mit dem *Erfüllbarkeitsproblem* im Jahre 1971 die Existenz des ersten NP-vollständigen Problems durch Stephen Cook bewiesen wurde, konnte jenes auf zahlreiche andere Probleme reduziert werden. Probleme Π, auf die ein NP-vollständiges Problem reduziert werden kann, bezeichnet man als *NP-hart*, d.h. als mindestens so schwer, wie ein NP-vollständiges Problem. Ist Π selbst aus NP, so folgt mit der Abgeschlossenheit der Menge NP gegenüber der Polynomialzeit-Reduktion, daß Π ebenfalls NP-vollständig sein muß.

Im wesentlichen aufgrund von Optimierungsproblemen, die nicht durch die Mengen P und NP zufriedenstellend klassifiziert werden konnten, konstruierte man die Klasse *PSPACE*.

Definition 1.7 (Komplexitätsklasse PSPACE)
Die Klasse PSPACE ist definiert als die Menge aller Entscheidungsprobleme, die mittels einer deterministischen (log-cost) RAM unter polynomiellem Speicherbedarf gelöst werden können.

Die Beziehung NP⊆PSPACE kann man sich intuitiv so vorstellen, daß nichtdeterministische Maschinen in Anbetracht ihres jeweiligen maximalen nichtdeterministischen Verzweigungsgrades m während ihrer polynomiellen Laufzeit auch höchstens polynomiell in m viel Speicher benötigen können und damit von einer deterministischen Maschine mit ebensoviel Speicherbedarf, jedoch beliebig hohem, aber endlichem Zeitbedarf, simuliert werden können. Der Vollständigkeits-Begriff gilt bei PSPACE in analogem Sinne wie bei NP. Als elementares PSPACE-vollständiges Problem gilt das *quantifizierte Erfüllbarkeitsproblem.*

Soweit die für diese Arbeit notwendigen Grundlagen der Komplexitätstheorie. An weiterführender Literatur sei der Leser auf [30], [61] und [10] verwiesen.

1.2 Mengen und Relationen

In diesem Unterabschnitt werden einige grundlegende Begriffe der Mengentheorie eingeführt, die zwar weitläufig bekannt sind, deren Notation sich jedoch in der Literatur gelegentlich unterscheidet.

Für gewöhnlich geht man bei einer *Menge* von einer ungeordneten Sammlung von Elementen aus, in der jedes Element höchstens einmal vorkommt. Hingegen handelt es sich bei einem *Tupel* um eine geordnete Liste von Elementen, in der ein Element auch mehrfach vorkommen kann. Im Zuge der späteren Definition von Graphen werden jedoch auch ungeordnete Mengen mit mehrfach auftretenden Elementen benötigt:

Definition 1.8 (Multimenge)
Als *Multimenge* S bezeichnet man eine Menge, in der ein Element mehr als einmal vorkommen kann. Jedem Element $s \in S$ ist eine *Multiplizität* $\mu(s)$ zugeordnet, so daß für die Kardinalität von S gilt:

$$\#S = \sum_{s \in S} \mu(s)$$

Eine elementare Struktur, welche Beziehungen zwischen Elementen einer Menge beschreibt, ist die *Relation.*

Definition 1.9 (Relation)
Sei M eine Menge. Dann heißt die Menge $R \subseteq M \times M$ eine (binäre) *Relation* über der Menge M. Die Relation R heißt:

- *reflexiv,* falls gilt: $\forall_{m \in M}\ (m, m) \in R$
- *irreflexiv,* falls gilt: $\forall_{m \in M}\ (m, m) \notin R$
- *transitiv,* falls für $m_1, m_2, m_3 \in M$ gilt:
  $(m_1, m_2) \in R$ und $(m_2, m_3) \in R \implies (m_1, m_3) \in R$

- *symmetrisch,* falls für $m_1, m_2 \in M$ gilt:
  $(m_1, m_2) \in R \iff (m_2, m_1) \in R$
- *antisymmetrisch,* falls für $m_1, m_2 \in M$ gilt:
  $(m_1, m_2) \in R$ und $(m_2, m_1) \in R \implies m_2 = m_1$

Eine reflexive, transitive und symmetrische Relation heißt *Äquivalenzrelation.* Eine reflexive, transitive und antisymmetrische Relation heißt *partielle Ordnung.*

Für die *Konkatenation* $R \circ S$ („R nach S“) zweier Relationen R, S über M gilt:

$$R \circ S = \{(m_1, m_2) \mid \exists_{m \in M}\ (m_1, m) \in S \text{ und } (m, m_2) \in R\}$$

Nun sei definiert:

$$\begin{aligned} R^0 &= \{(m, m) \mid m \in M\} \\ R^1 &= R \\ R^n &= R \circ R^{n-1} \end{aligned}$$

Der *reflexiv-transitive Abschluß* von R ist definiert als:

$$R^* = \bigcup_{n \geq 0} R^n$$

Der *transitive Abschluß* von R ist definiert als:

$$R^+ = \bigcup_{n > 0} R^n$$

1.3 Graphen

Graphen gehören zu den fundamentalen Strukturen in der Informatik. Sie stellen eine abstrakte Möglichkeit dar, Beziehungen zwischen Objekten der realen oder der mathematischen Welt auszudrücken und bilden eine formale Basis zur Definition algorithmischer Probleme, sowie eine Hilfe bei der Suche nach und Beschreibung von Lösungsverfahren.

Eine zentrale Rolle spielen Graphen insbesondere auch bei der Modellierung von Netzwerken, wie zum Beispiel Schaltkreisen oder Verdrahtungsstrukturen. Auf Netzwerken, wie diesen, stellen sich häufig Probleme, Signale unter bestimmten Optimierungsbedingungen von einem gegebenen Startort zu einem oder mehreren Zielorten zu leiten, wobei die Verbindungen im Netzwerk nur in einer Richtung passiert werden dürfen. Probleme dieser Art lassen sich durch die Klasse der *gerichteten Graphen* beschreiben, welche im Vordergrund der Betrachtungen der vorliegenden Arbeit stehen werden. Begonnen werden soll jedoch zunächst mit der Klasse der *ungerichteten Graphen,* die zur Modellierung einiger Subprobleme ebenfalls vonnöten sein werden.

1.3.1 Ungerichtete und gerichtetete Graphen

Alle Graphenformen, die im folgenden betrachtet werden, sind in mancher Literatur unter dem Begriff *Multigraphen* spezifiziert.

Definition 1.10 (Ungerichteter Graph)
Ein *ungerichteter Graph* (oder einfach: Graph) $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Multimenge E von Kanten. Eine Kante $e_i \in E$ wird repräsentiert durch eine Multimenge von zwei Knoten, auch als *Rand* $\rho(e_i)$ der Kante bezeichnet.

Ein Graph heißt *endlich*, wenn $\#(V \cup E) < \infty$. Als *Grad deg*(v) eines Knotens v wird die Zahl der Kanten bezeichnet, in denen v enthalten ist. Zwei Knoten $v, w \in V$ heißen *benachbart*, wenn $\{v, w\} \in E$. Eine Kante e mit $\rho(e) = \{v, v\}$ für ein $v \in V$ heißt *Schlinge*. Ein Graph heißt *zusammenhängend*, wenn seine Knotenmenge nicht in zwei nichtleere Teilmengen partitioniert werden kann, so daß jede Kante ganz innerhalb einer der Teilmengen liegt.

Graphen, die Kanten e besitzen mit $\#\rho(e) > 2$, werden als *Hypergraphen* bezeichnet. Hypergraphen werden später vor allem in ihrer gerichteten Variante eine wichtige Rolle spielen:

Definition 1.11 (Gerichteter Hypergraph)
Ein *gerichteter Hypergraph* $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Multimenge E von Kanten. Eine Kante $e_i \in E$ wird repräsentiert durch einen Startknoten $v_i^+ \in V$ und eine Multimenge $V_i^- \subseteq V$ von Zielknoten.

Definition 1.12 (Gerichteter Graph)
Ein *gerichteter Graph* $G = (V, E)$ ist ein gerichteter Hypergraph, wobei für alle Kanten $(v_i^+, V_i^-) \in E$ gilt: $\#V_i^- = 1$. Statt V_i^- notiert man vereinfacht auch v_i^-.

Ist $e = (v, w)$ Kante eines gerichteten Graphen $G = (V, E)$, so heißt $Q(e) = v$ *Quellknoten* und $Z(e) = w$ *Zielknoten* der Kante. v ist *direkter Vorgänger* von w und w ist *direkter Nachfolger* von v bezüglich der Kante e. Die Menge der direkten Vorgänger eines Knotens v bezeichnen wir mit *pred*(v), die Menge der direkten Nachfolger eines Knotens v bezeichnen wir mit *succ*(v). Die reflexiv-transitiven Abschlüsse dieser Vorgänger- und Nachfolgerrelationen bezeichnen wir mit *pred*$^*(v)$ bzw. *succ*$^*(v)$, die transitiven Abschlüsse mit *pred*$^+(v)$ bzw. *succ*$^+(v)$. Es sei E_v^- die Menge der in v einlaufenden und E_v^+ die Menge der von v ausgehenden Kanten, insgesamt bezeichnen wir mit E_v die Menge der zu v adjazenten Kanten.

Als *Ingrad deg*$^-(v)$ eines Knotens v bezeichnet man die Zahl der in v einlaufenden Kanten, als *Ausgrad deg*$^+(v)$ die Zahl der aus v auslaufenden Kanten. Kanten e_1 und e_2 nennt man *parallel*, falls gilt: $Q(e_1) = Q(e_2)$, $Z(e_1) = Z(e_2)$. Man nennt sie *antiparallel*, falls gilt: $Q(e_1) = Z(e_2)$, $Q(e_2) = Z(e_1)$.

Es gibt verschiedene Darstellungsmöglichkeiten von Graphen. Zu den mathematischen Darstellungen gehören die, im allgemeinen jedoch nicht sonderlich platzeffizienten

Matrizendarstellungen: *Adjazenzmatrix* und *Inzidenzmatrix*. Wir führen beide lediglich für gerichtete Graphen ein:

Definition 1.13 (Adjazenzmatrix, Inzidenzmatrix)
Sei $G = (V, E)$ ein schlingenfreier gerichteter Graph mit $n = \#V$ und $m = \#E$. Dann sind die *Adjazenzmatrix* $A_G = (a_{jk}) \in \mathbb{N}_0^{n \times n}$ und die *Inzidenzmatrix* $I_G = (i_{jk}) \in \{0, +1, -1\}^{n \times m}$ definiert wie folgt:

$$a_{jk} = \#\{e \in E \mid Q(e) = v_j \text{ und } Z(e) = v_k\}$$

$$i_{jk} = \begin{cases} +1, & \text{falls } Q(e_k) = v_j \\ -1, & \text{falls } Z(e_k) = v_j \\ 0, & \text{sonst} \end{cases}$$

Eine alternative, platzeffizientere Darstellung bilden Adjazenzlisten, die zu jedem Knoten eine Liste der Nachbarknoten speichern oder etwa zu jeder Kante die Liste der Randknoten.

Definition 1.14 (Pfad, Zyklus, Länge)
Ein *Pfad* p (oder auch: *gerichteter Weg*) auf einem gerichteten Graphen ist eine endliche Folge $p = (v_0, e_1, v_1, e_2, v_2, \ldots, e_{n-1}, v_{n-1})$ mit $Q(e_i) = v_{i-1}$ und $Z(e_i) = v_i$ für $i \in \{1 \ldots n-1\}$. Falls $v_0 = v_{n-1}$, nennt man den Pfad einen *Zyklus*. Als *Länge* $l(p)$ des Pfades p bezeichnen wir die Zahl seiner Kanten: $n - 1$.

Ist jeder Kante $e \in E$ eines Graphen ein *Distanzwert*[2] $d : E \to \mathbb{R}^+$ zugeordnet, so bezeichnen wir als *Distanz* eines Pfad p gerade die Summe der Distanzwerte seiner Kanten:

$$d(p) = \sum_{e \in p} d(e)$$

Zur Berechnung von Pfaden minimaler Distanz existieren effiziente Algorithmen mit Laufzeiten von $\mathcal{O}(\#E \log \#V)$ bis $\mathcal{O}(\#E + \#V \log \#V)$, welche an dieser Stelle jedoch nicht wiederholt werden sollen. Stattdessen sei der Leser beispielsweise auf [50] verwiesen.

Ein Knoten w heißt von v aus *erreichbar*, wenn es im Graphen einen Pfad mit Startknoten v und Endknoten w gibt. Ist jeder Knoten von jedem Knoten aus erreichbar, so nennt man den Graphen *streng zusammenhängend*.

Definition 1.15 (Aufgespannter Subgraph)
Sei $G = (V, E)$ ein (gerichteter) Graph und $U \subseteq V$. Dann heißt $S_U = (V_U, E_U)$ der durch U *aufgespannte Subgraph* von G, falls gilt: $V_U = U$ und $E_U = \{e \in E \mid \rho(e) \subseteq U\}$.

Bei der Modellierung kombinatorischer Schaltkreise, wie auch später bei der Definition feldprogrammierbarer Architekturen spielt die Klasse der gerichteten azyklischen Graphen eine wesentliche Rolle:

[2] Hinsichtlich der Distanzwerte betrachten wir in dieser Arbeit aus praktischen Gründen ausschließlich nichtnegative Werte.

Definition 1.16 (Gerichteter azyklischer Graph, DAG)
Ein *gerichteter azyklischer Graph* (oder: directed acyclic graph, DAG) ist ein gerichteter Graph, der keine Zyklen enthält.

Ein Knoten v mit $deg^-(v) = 0$ heißt *Quelle*, ein Knoten v mit $deg^+(v) = 0$ heißt *Senke* des Graphen.

1.3.2 Periodische Graphen

Die für diese Arbeit wichtigste Klasse von Graphen stellen die sogenannten *periodischen Graphen* dar. Periodische Graphen spielen eine entscheidende Rolle bei der Beschreibung regelmäßiger Systeme, wie zum Beispiel Datenabhängigkeitsgraphen verschachtelter Programmschleifen, regelmäßige Prozessorfelder oder regelmäßige Layouts. Während periodische Graphen, zum Teil unter der Bezeichnung *dynamic graphs*, bereits relativ früh im Rahmen solcher konkreter Anwendungsfälle betrachtet wurden, begann man offensichtlich erst Mitte der 1980er Jahre, ihre strukturellen Eigenschaften zu erforschen. In dieser Einführung werden jedoch nur einige, im späteren Verlauf dieser Arbeit benötigte Resultate angegeben. Ansonsten sei der Leser vor allem verwiesen auf [60], [21], sowie [76], [31], [8], [32] und [33].

In der vorliegenden Arbeit wird die Klasse der periodischen Graphen als strukturelles Modell feldprogrammierbarer Architekturen eingesetzt, denn periodische Graphen stellen ein flexibles Instrument dar, um die in repetitivem Muster angeordnete konfigurierbare Funktionalität feldprogrammierbarer Architekturen sehr kompakt darzustellen und Architekturen hinsichtlich ihrer Ausdehnung variabel konstruieren und verändern zu können. Zur kompakten Beschreibung periodischer Graphen dient der Typus des *statischen Graphen (static graph, dependence graph)*:

Definition 1.17 (Statischer Graph)
Ein *statischer Graph* $G = (V, E, \tau)$ ist ein gerichteter Graph mit einer *Transitionsabbildung* $\tau : E \to \mathbb{Z}^d$. Hierbei wird $d \in \mathbb{N}$ als die *Dimension* des statischen Graphen bezeichnet.

Ein endlicher, statischer Graph G bildet nun eine eindeutige Konstruktionsvorschrift für einen unendlichen periodischen Graphen G^∞. Während in den meisten Publikationen von solchen unendlichen periodischen Graphen ausgegangen wird, ist für uns ausschließlich die endliche Variante von Interesse, also periodische Graphen von endlicher *Dilatation*. In weiterer Abweichung, da aus anwendungsspezifischen Gründen vorteilhaft, werden wir auch Kanten e mit $\#\rho(e) = 1$ betrachten, das heißt deren Quell- oder Zielknoten nicht definiert ist.

Definition 1.18 (Endlicher periodischer Graph)
Ein *endlicher periodischer Graph* $G^x = (G, x)$ ist ein gerichteter Graph, der beschrieben wird durch einen statischen Graphen $G = (V, E, \tau)$ und einen *Dilatationsvektor* $x \in \mathbb{N}^d$, wobei d gerade der Dimension des statischen Graphen entspricht. In der Knotenmenge des periodischen Graphen ist stets ein *Nilknoten* v_{nil} enthalten.

Zu einem Vektor $x \in \mathbb{N}^d$ bezeichne *span*$(x) \subset \mathbb{N}^d$ die Menge der *Gitterpunkte* des durch x aufgespannten d-dimensionalen Gitters:

$$\textit{span}(x) = \{y \in \mathbb{N}^d \mid \forall_{i=0\ldots d-1}\ 0 \leq y_i < x_i\}$$

Die Konstruktion eines endlichen periodischen Graphen G^x, auch zuweilen als „Abrollen“ des statischen Graphen bezeichnet, wird wie folgt durchgeführt:

Für die Knotenmenge V^x des periodischen Graphen G^x gilt:

$$V^x = V \times \textit{span}(x)$$

Eine Kante $e^x \in E^x$ des periodischen Graphen ist ein Paar $(e,z) \in E \times \mathbb{N}^d$. Für Kanten $e \in E$ und Gitterpunkte $z \in \mathbb{N}^d$ seien die beiden Abbildungen $Q^x : E^x \to V^x$ und $Z^x : E^x \to V^x$ wie folgt definiert:

$$Q^x\big((e,z)\big) = \begin{cases} (Q(e),z) & : \text{ falls } z \in \textit{span}(x) \\ v_{\text{nil}} & : \text{ sonst} \end{cases}$$

$$Z^x\big((e,z)\big) = \begin{cases} (Z(e),z+\tau(e)) & : \text{ falls } z+\tau(e) \in \textit{span}(x) \\ v_{\text{nil}} & : \text{ sonst} \end{cases}$$

Damit ist auch die Kantenmenge E^x des periodischen Graphen G^x eindeutig definiert als:

$$E^x = \{\ (e,z) \in E \times \mathbb{N}^d \mid Q^x\big((e,z)\big) \neq v_{\text{nil}} \text{ oder } Z^x\big((e,z)\big) \neq v_{\text{nil}}\ \}$$

Man beachte, daß die Menge E^x aus Paaren (e,z) über $E \times \mathbb{N}^d$ und nicht über $E \times$ *span*(x) konstruiert ist. Somit gilt in Abweichung zur Knotenmenge des periodischen Graphen im allgemeinen hier die Beziehung: $E^x \supseteq E \times \textit{span}(x)$.

Die Menge

$$N^x = \{\ (e,z) \in E^x \mid Q^x\big((e,z)\big) = v_{\text{nil}} \text{ oder } Z^x\big((e,z)\big) = v_{\text{nil}}\ \}$$

der zum Knoten v_{nil} adjazenten Kanten heißt Menge der *Nullkanten* des periodischen Graphen.

Eine Kante $(e,z) \in E^x$ heißt *Internkante*, falls $\tau(e) = \vec{0}$, ansonsten *Externkante*. Für Knoten $v \in N^x$ liefere die Abbildung *statnode*(v) jenen Knoten des statischen Graphen, aus dem v erzeugt wurde. Analog liefere die Abbildung *statedge*(e) für Kanten $e \in E^x$ die Kante des statischen Graphen, aus der e erzeugt wurde.

Abbildung 1.1 zeigt links einen statischen Graphen und rechts daneben den aus ihm konstruierten periodischen Graphen der Dilatation $(3,4)$.

Zur Betrachtung von einzelnen Gitterpunkten zugeordneten Subgraphen eines endlichen periodischen Graphen erweitern wir den in Definition 1.15 eingeführten Begriff des durch eine gegebene Knotenmenge U aufgespannten Subgraphen $S_U = (V_U, E_U)$ dahingehend, daß zu E_U noch die Externkanten hinzutreten, welche adjazent zu einem der Knoten in V_U sind. Aus der Sicht dieser speziellen Art von Subgraphen, welche wir als *Module* bezeichnen, sind alle Externkanten zugleich auch Nullkanten:

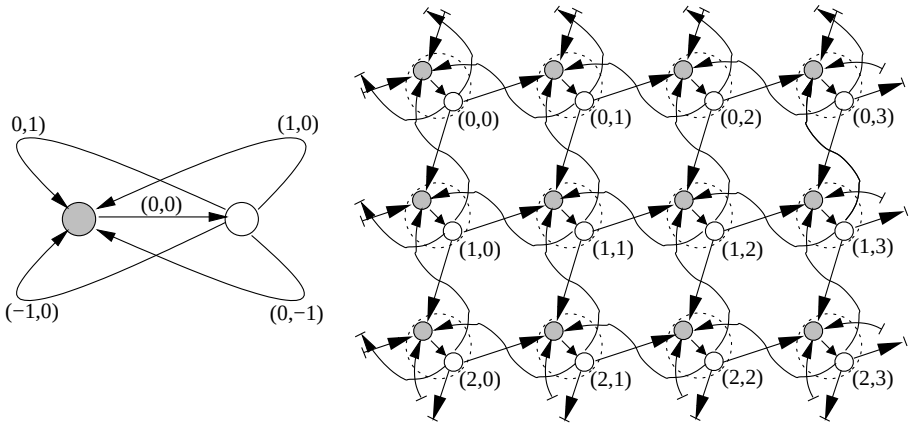


Abbildung 1.1: Statischer und periodischer Graph

Definition 1.19 (Modul)
Sei $G^x = (G, x)$ ein periodischer Graph mit statischem Graphen $G = (V, E)$. Für ein $z \in$ *span*(x) sei $U_z = \{(v, z) \in V^x \mid v \in V\}$. Ist $S_{U_z} = (V_{U_z}, E_{U_z})$ der durch U_z aufgespannte Subgraph, so heißt der gerichtete Graph $M_z = (V_z, E_z)$ ein *Modul* von G^x, falls gilt: $V_z = V_{U_z}$ und $E_z = E_{U_z} \cup \{e \in E^x \mid (Q^x(e) \in V_z$ und $Z^x(e) = v_{\text{nil}})$ oder $(Z^x(e) \in V_z$ und $Q^x(e) = v_{\text{nil}})\}$.

Offensichtlich sind alle Module eines periodischen Graphen isomorph. Mit Ausnahme der Externkanten sind Module auch isomorph zum zugrundeliegenden statischen Graphen. Diese bestehende Isomorphie ist Ausgangspunkt der folgenden Definitionen:

Definition 1.20 (Korrespondierender Knoten)
Für $y, z \in$ *span*(x) seien M_y und M_z zwei Module eines periodischen Graphen G^x mit statischem Graph $G = (V, E)$. Ist v ein Knoten des Moduls M_y, dann heißt $w =$ *corr*(z, v) der zu v *korrespondierende Knoten* des Moduls M_z, falls gilt:

$$\exists_{u \in V}\ v = (u, y) \text{ und } w = (u, z)$$

Korrespondierende Knoten eines periodischen Graphen werden also aus demselben Knoten des statischen Graphen erzeugt. Bei Kanten gestaltet sich die Definition allerdings etwas diffiziler, da jene Externkanten, die keine Nullkanten sind, zwei Modulen zugeordnet sein könnten. Erst eine zusätzliche Unterscheidung hinsichtlich der relativen Orientierung von Kanten ermöglicht eine Definition:

Definition 1.21 (Korrespondierende/duale Kante)
Für $y, z \in$ *span*(x) seien M_y und M_z zwei Module eines periodischen Graphen G^x mit statischem Graph $G = (V, E)$ und sei e_i eine Kante des Moduls M_y. Die Kante $e_j =$ *corr*(z, e_i) heißt die zu e_i *korrespondierende Kante* des Moduls M_z, falls *statedge*$(e_i) =$ *statedge*(e_j) und mindestens eine der beiden folgenden Aussagen gilt:

$$\begin{aligned} \exists_{u \in V}\ Q^x(e_i) = (u, y) &\implies Q^x(e_j) = (u, z) \\ \exists_{u \in V}\ Z^x(e_i) = (u, y) &\implies Z^x(e_j) = (u, z) \end{aligned}$$

Die Kante $e_j = \textit{dual}(z, e_i)$ heißt die zu e_i *duale Kante* des Moduls M_z, falls $\textit{statedge}(e_i) = \textit{statedge}(e_j)$ und mindestens eine der beiden folgenden Aussagen gilt:

$$\exists_{u \in V}\ Q^x(e_i) = (u, y) \implies \exists_{v \in V}\ Z^x(e_j) = (v, z)$$
$$\exists_{u \in V}\ Z^x(e_i) = (u, y) \implies \exists_{v \in V}\ Q^x(e_j) = (v, z)$$

Abbildung 1.2 illustriert die Aussagen von Definition 1.21 anhand eines einfachen Beispiels. Die Abbildung zeigt links einen eindimensionalen statischen Graphen und rechts einen daraus konstruierten periodischen Graphen. Bezüglich des Moduls M_z ist die Kante e_k die korrespondierende Kante und die Kante e_j die duale Kante zur Kante e_i des Moduls M_y.

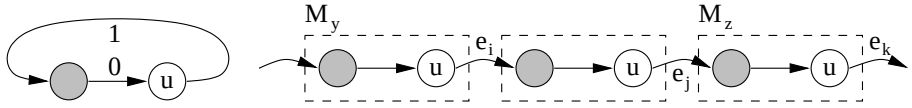


Abbildung 1.2: Korrespondierende und duale Kanten

Zum Abschluß des Unterkapitels über periodische Graphen sei noch ein Blick auf die Bestimmung von Wegen mit minimaler Distanz geworfen, um auch auf komplexitätstheoretischer Seite eine Einschätzung der periodischen Graphen im Hinblick auf gewöhnliche Graphen zu skizzieren.

Definition 1.22 (Minimaler m-Pfad)
Sei $G = (V, E, \tau)$ ein statischer Graph und $d : E \to \mathbb{N}$ eine Distanzfunktion. Dann heißt $p = (v_0, e_1, v_1, \ldots, e_{n-1}, v_{n-1})$ ein minimaler m-Pfad für ein $m \in \mathbb{Z}^d$, falls $d(p)$ minimal ist und es gilt:

$$\tau(p) = \sum_{i=1}^{n-1} \tau(e_i) = m$$

Im Falle von unendlichen periodischen Graphen G^∞ ist bekannt, daß dieses Problem zwar NP-vollständig, unter fixierter Dimension jedoch in pseudo-polynomieller Zeit lösbar ist [31].

Für endliche periodische Graphen G^x wurde hingegen gezeigt, daß das Problem des minimalen m-Pfades sogar PSPACE-vollständig ist [76]. Durch die Einschränkung auf eindimensionale Transitionsvektoren τ liegt es dann in der Klasse der NP-vollständigen Probleme, wenn zusätzlich $\#V = 1$ gilt und in der Klasse P, wenn lediglich $\tau \in \{-1, 0, 1\}$ [60].

1.4 Sequentielle Look-Up-Table-Schaltkreise

Die Komponenten digitaler Systeme bilden Schaltkreise, deren Funktionalität primär über *Boolesche Funktionen* definiert wird.

Definition 1.23 (Boolesche Funktion)
Eine *m-stellige Boolesche Funktion f* ist eine Abbildung

$$f : \mathbb{B}^n \to \mathbb{B}^m$$

Im folgenden betrachten wir ausschließlich einstellige Boolesche Funktionen, d.h. es sei $m = 1$. Eine Darstellungsmöglichkeit für Boolesche Funktionen liefert die Auflistung ihres Wertebereichs, also der Funktionswerte unter allen möglichen Eingangswerten, traditionell bekannt unter dem Begriff *Wahrheitstafel*. Die technische Realisierung einer Wahrheitstafel wird als *Look-Up-Table* (LUT) bezeichnet. Im allgemeinen sind Look-Up-Tables aufgrund ihres in Abhängigkeit der Zahl der Eingangswerte exponentiellen Platzbedarfs keine sonderlich effiziente Realisierung Boolescher Funktionen, jedoch haben sie sich für Funktionen von bis zu fünf Variablen zumindest im Bereich feldprogrammierbarer Architekturen durchgesetzt, im wesentlichen, da sie hinsichtlich der rechnergestützten Synthese- und Layoutverfahren weitaus vorteilhafter als beispielsweise konfigurierbare Gatterkombinationen sind.

Als elementares Objekt eines Schaltkreises definieren wir den *Pin*, als Bezugspunkt für die Informationsträger des Schaltkreises die sogenannten *Signale*, welche an ein zeitliches Verhalten gebunden sind.

Definition 1.24 (Signal)
Ein *(digitales) Signal* ist eine Funktion $\sigma : \mathbb{R} \to \mathbb{B}$, wobei $\sigma(t)$ der Wert des Signals zu einem Zeitpunkt t ist.

Die Einführung von Signalwerten lediglich über $\mathbb{B}$ ist insofern gerechtfertigt, als wir später ausschließlich unidirektionale Netze mit eindeutigem Treiber betrachten werden. Der besseren Lesbarkeit wegen interpretieren wir für gewöhnlich Signale mit ihrem Wert. Für einen Pin p und einen Zeitpunkt $t \in \mathbb{R}$ definieren wir $\sigma_p(t)$ als der Wert des zum Zeitpunkt t am Pin p befindlichen Signals.

Zur Definition der Look-Up-Table genügt uns die Zugrundelegung eines idealisierten Verzögerungsmodells, welches unabhängig von Signalwerten und Lasten arbeitet.

Definition 1.25 (Look-Up-Table)
Eine *Look-Up-Table* (LUT) F ist ein Quadrupel (I, o, δ, α), wobei I die Menge der Eingänge, o der Ausgang, $\delta \in \mathbb{R}$ die Verzögerungszeit (*delay*) und $\alpha \in \mathbb{R}$ der Platzverbrauch (*area*) ist. Als *Breite* $b(F)$ der Look-Up-Table definieren wir die Kardinalität von I. Die *von F realisierte Boolesche Funktion* sei $\varphi(F)$.

Für jede Look-Up-Table F gelte:

$$\sigma_{o_F}(t) = \varphi(F)(\sigma_{i_1}(t - \delta_F), \ldots, \sigma_{i_{b(F)}}(t - \delta_F)) \quad \text{wobei } i_j \in I_F$$

Definition 1.26 (Netz)
Ein *Netz* N ist ein Paar (s, T), wobei s ein Pin und T eine Menge von Pins ist. Dabei werden $\mathit{drv}(N) = s$ als *Treiber* und $\mathit{tgts}(N) = T$ als Menge der *Ziele* des Netzes bezeichnet.

Für jedes Netz N gelte:

$$\forall_{p \in \mathit{tgts}(N)}\ \sigma_p(t) = \sigma_{\mathit{drv}(N)}(t)$$

Spezielle Pins eines Schaltkreises sind dessen *primäre Eingänge* und *primäre Ausgänge*, gelegentlich auch als *Pads* bezeichnet:

Definition 1.27 (Primärer Eingang/Ausgang)
Ein *primärer Eingang* ist ein Pin, der Treiber mindestens eines Netzes und kein Ziel ist. Ein *primärer Ausgang* ist ein Pin, der Ziel genau eines Netzes und kein Treiber ist.

Jeder sequentielle Schaltkreis besitzt ein ausgezeichnetes Netz, das *Clock*-Netz, dessen Ziele ausschließlich Verzögerungselemente, die sogenannten *Latches*, sind. Wir modellieren sie an dieser Stelle, ebenfalls idealisiert, ohne Verzögerungszeit:

Definition 1.28 (Latch)
Seien d, c und q Pins, dann heißt ein Tripel $L = (d, c, q)$ (flankengesteuertes Daten-) *Latch*, falls gilt:

$$\sigma_q(t) = \begin{cases} \sigma_d(t-\varepsilon), & \text{falls } \exists_{\varepsilon>0}\ \forall_{\delta\leq\varepsilon,\delta>0}\ \sigma_c(t-\delta) = 0 \neq \sigma_c(t+\delta) \\ \sigma_q(t-\varepsilon), & \text{falls } \exists_{\varepsilon>0}\ \forall_{\delta\leq\varepsilon,\delta>0}\ \sigma_c(t-\delta) = \sigma_c(t+\delta) \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Pin d heißt *Dateneingang* und Pin c heißt *Clock*-Eingang.

Wir nennen primäre Eingänge und Ausgänge, sowie Look-Up-Tables und Latches die *Komponenten* eines sequentiellen Look-Up-Table-Schaltkreises.

Auf der Basis der vorangegangenen Definitionen kann nun das formale Modell der *sequentiellen Look-Up-Table-Schaltkreise* eingeführt werden, wobei wir jedoch aufgrund der Gleichwertigkeit aller Eingänge einer Look-Up-Table von den Pins auf das gesamte Look-Up-Table-Objekt abstrahieren. Ferner gehen wir stets von der Präsenz genau eines Clock-Netzes aus, weshalb wir dieses nicht weiter betrachten. Dieser Schritt ermöglicht aber nunmehr auch eine Abstraktion von den Pins der Latches, so daß wir künftig Netze nicht mehr über Pins, sondern nur noch über Komponenten des Schaltkreise betrachten müssen.

Definition 1.29 (Sequentieller LUT-Schaltkreis)
Ein *sequentieller LUT-Schaltkreis* $\mathcal{C}$ wird beschrieben durch einen gerichteten Hypergraphen $\mathcal{C} = (V, E)$. Die Knotenmenge V partitioniert sich in vier Untermengen $V = I \dot\cup O \dot\cup F \dot\cup L$, wobei I die Menge der primären Eingänge, O die Menge der primären Ausgänge, F die Menge der Look-Up-Tables und L die Menge der Latches ist. Die Kantenmenge E ist eine Menge von Netzen.

Knoten aus $I \cup L$ bezeichnen wir als *Quellen*, Knoten aus $O \cup L$ bezeichnen wir als *Senken* des Schaltkreises $\mathcal{C}$. Zu einem Knoten $v \in V$ liefere *pred*(v) die Menge der Vorgängerknoten, d.h. jene Knoten, die ein Netz treiben, das einen Eingang von v bedient. Entsprechend sei *succ*(v) die Menge der Nachfolgerknoten, d.h. jene Knoten, die einen Eingang besitzen, der vom Netz bedient wird, welches v treibt. Es bezeichne $N_{\mathcal{C}}$ die Menge aller Netze des Schaltkreises $\mathcal{C}$.

Von besonderem Interesse bei Schaltkreisen ist häufig die Signalverzögerung *(Delay)*, welche über kombinatorische Pfade des Schaltkreises hinweg berechnet wird.

Definition 1.30 (Kombinatorischer Schaltkreis-Pfad)
Sei $\mathcal{C} = (V, E)$ ein sequentieller LUT-Schaltkreis mit Knotenpartitionierung $V = I \dot\cup O \dot\cup F \dot\cup L$. Dann heißt ein Pfad $(v_0, e_1, v_1, \ldots, e_{n-1}, v_{n-1})$ auf dem Hypergraphen (V, E) ein *kombinatorischer Pfad* des Schaltkreises $\mathcal{C}$, wenn gilt: $v_0 \in I \cup L$ und $v_{n-1} \in O \cup L$ und $\forall_{i=1\ldots n-2}\ v_i \in F$.

1.5 Diskrete Optimierung

1.5.1 Lineare und Integer Programme

Definition 1.31 (Diskretes/Allgemeines Optimierungsproblem)
Ein Optimierungsproblem $\Pi : I \to \mathcal{P}(S)$ heißt *diskret*, wenn der Lösungsraum $\mathcal{P}(S)$ für alle Probleminstanzen $p \in I$ endlich ist. Π heißt *allgemein*, wenn $\mathcal{P}(S)$ unendlich oder nicht abzählbar ist.

Eine spezielle Form diskreter Optimierungsprobleme liegt vor, wenn die Lösungsbedingungen, sowie die Kostenfunktion als lineare Funktionen über dem Körper $\mathbb{R}$ geschrieben werden können. Man bezeichnet diese Probleme auch als *lineare Programme* (LP):

Definition 1.32 (Lineares Programm)
Ein *lineares Programm* ist gegeben durch eine Matrix $A \in \mathbb{R}^{m\times n}$, einen Vektor $b \in \mathbb{R}^m$ und eine Kostenfunktion $c \in \mathbb{R}^n$. Die Lösung des linearen Programmes sind alle Vektoren $x \in \mathbb{R}^n$ mit $x \geq 0$, welche die Bedingung $Ax \leq b$ erfüllen und $c^T x$ minimieren.

In geometrischer Hinsicht bildet die Lösungsmenge eines LPs ein konvexes Polytop im $\mathbb{R}^n$, wobei alle Optimallösungen auf dessen Ecken liegen. Ein klassisches Lösungsverfahren für LPs bildet das 1947 von George B. Dantzig vorgeschlagene *Simplexverfahren*, welches auf einer sequentiellen Durchmusterung der Ecken basiert und aufgrund der Konvexität des Lösungsraumes stets das globale Minimum der Kostenfunktion findet.

Eine Ecke des Lösungspolytops wird beschrieben als Linearkombination einer Basis. Ausgehend von einer zulässigen Basis, also einer beliebigen Startecke, bewegt man sich durch Anwendung von Pivotschritten, geometrisch betrachtet, zu den Nachbarecken. Indem sich die Pivotoperationen quasi auch auf die Kostenfunktion erstrecken, wird anhand der Nichtexistenz eines gültigen Pivotelementes erkannt, daß keine günstigere Nachbarecke mehr existiert. Aufgrund der Konvexität des Lösungsraumes wurde damit ein globales Optimimum erreicht. Für die Wahl des Pivotelementes gibt es verschiedene Strategien. Eine häufig verwendete, da im allgemeinen sehr effizient implementierbare Methode, ist die des steilsten Anstiegs, im vorliegenden Falle also die größtmögliche Reduktion der Kosten. Alternative Pivotregeln und ein Vergleich von Konvergenzeigenschaften, insbesondere hinsichtlich „entarteter Ecken“, können an dieser Stelle allerdings nicht betrachtet werden. Stattdessen sei der Leser auf einschlägige Literatur verwiesen [59]. In der Regel ist mit dem Ausgangsproblem jedoch noch keine zulässige Basislösung gegeben. Zur Ermittlung einer Startecke schaltet die sogenannte *2-Phasen-Strategie* dem eigentlichen Simplexverfahren ein weiteres Simplexverfahren vor. Hierbei wird durch die Einführung von Schlupfvariablen eine künstliche Basis geschaffen

und mit einer Minimierungsfunktion, welche die Summe der Schlupfvariablen in der Basis beschreibt, letztere „hinausminimiert". Gelingt dies nicht, ist das Problem unbeschränkt und es existiert keine Lösung.

Eine praktische Implementierung des Verfahrens führt die Operationen auf einer Tabelle, dem sogenannten Simplextableau, durch, wobei man in der Regel das *revidierte Simplexverfahren* anwendet, welches lediglich mit einer impliziten Darstellung der Basis (*working basis*) arbeitet. Obwohl das Simplexverfahren im worst case von exponentieller Laufzeit ist, da das Lösungspolytop bis zu $\binom{n}{m}$ Ecken besitzen kann, erweist es sich bei vielen Problemen in der Praxis als relativ effizient. Erst viel später konnte bewiesen werden, daß das Entscheidungsproblem linearer Programme dennoch in der Klasse P liegt (1979: Ellipsoid-Methode von Khachian, 1984: Innere-Punkt-Methode von Karmarkar) [2].

Oftmals treten auch Nebenbedingungen und Lösungsraum in ganzzahliger Form auf. Dieser Fall ganzzahliger linearer Programme bezeichnet man auch als *Integer Programme* (IP, ILP):

Definition 1.33 (Integer Programm)
Ein *Integer Programm* ist gegeben durch eine Matrix $A \in \mathbb{Z}^{m \times n}$, einen Vektor $b \in \mathbb{Z}^m$ und eine Kostenfunktion $c \in \mathbb{R}^n$. Die Lösung des Integer Programmes sind alle Vektoren $x \in \mathbb{Z}^n$ mit $x \geq 0$, welche die Bedingung $Ax \leq b$ erfüllen und $c^T x$ minimieren.

Als *lineare Relaxation* eines Integer Programms nennt man jenes lineare Programm, welches sich durch Weglassen der Ganzzahligkeitsbedingung ergibt. Aus geometrischer Sicht bildet die Relaxation eine (konvexe) Hülle um den Lösungsraum des ursprünglichen Integer Programms. Obwohl bereits hier ein starker Zusammenhang zwischen linearen und Integer Programmen deutlich wird, ist das Entscheidungsproblem für Integer Programme jedoch NP-vollständig [30]. Andererseits gibt es Spezialfälle, in denen die Lösungsmenge eines Integer Programms mit jener seiner linearen Relaxation übereinstimmt:

Definition 1.34 (Vollständig unimodulare Matrix)
Eine Matrix A heißt *vollständig unimodular*, wenn die Determinante jeder quadratischen nichtsingulären Untermatrix von A die Werte 1 oder -1 hat.

Über die Klasse der vollständig unimodularen Matrizen liefert nun der Satz von Hoffmann und Kruskal ein entscheidendes Ergebnis [2]:

Satz 1.1 (Hoffmann-Kruskal)
Sei $A \in \mathbb{Z}^{m \times n}$. A ist vollständig unimodular genau dann, wenn für alle $b \in \mathbb{Z}^m$ gilt: Jede Ecke des Polytops $\{x \mid x \geq 0 \text{ und } Ax \leq b\}$ ist ganzzahlig.

Integer Programme, die auf vollständig unimodularen Matrizen basieren, werden deshalb auch als *einfache Integer Programme* bezeichnet. Typische Anwendungsfälle aus der Graphentheorie liefern beispielsweise Inzidenzmatrizen von gerichteten oder ungerichteten bipartiten Graphen. In dieser Arbeit werden sie in Kapitel 4 eine Rolle bei der Abschätzung von kürzesten Wegen in periodischen Graphen spielen.

1.5.2 Beschränkte lineare Optimierung

In der Praxis häufig auftretende Varianten linearer Programme sind *beschränkte lineare Optimierungsprobleme* (BLOP). In ihrer Standardform besitzen sie lineare Nebenbedingungen der Form $Ax = b$, mit oberen Schranken für den Lösungvektor x.

Definition 1.35 (Beschränktes lineares Optimierungsproblem)
Ein *beschränktes lineares Optimierungsproblem* ist gegeben durch eine Matrix $A \in \mathbb{R}^{m \times n}$, einen Vektor $b \in \mathbb{R}^m$, einen Vektor $u \in \mathbb{R}^n$ und eine Kostenfunktion $c \in \mathbb{R}^n$. Die Lösung des beschränkten linearen Programmes sind alle Vektoren $x \in \mathbb{R}^n$ mit $0 \leq x \leq u$, welche die Bedingung $Ax = b$ erfüllen und $c^T x$ minimieren.

Lineare Programme der Form $Ax \leq b$ sind durch die Einführung von Schlupfvariablen leicht auf die Standardform $Ax = b$ bringen: Sei $\tilde{A} = (A, I_m)$, wobei I_m die $m \times m$-Einheitsmatrix ist. Dann ist $\tilde{x} = (x, s)^T \in \mathbb{R}^{n+m}$ für ein $s \in \mathbb{R}^m$ mit $s \geq 0$ genau dann eine Lösung von $\tilde{A}\tilde{x} = b$, wenn x eine Lösung von $Ax \leq b$ ist.

Das im vorigen Unterabschnitt erläuterte Simplexverfahren könnte nun leicht für BLOP erweitert werden, indem die Beschränkungen $x \leq u$ als Nebenbedingungen reformuliert werden. Dies vergrößert jedoch das Problem um entsprechend viele Hilfsvariablen und Bedingungen. Es ist jedoch möglich, im Zuge der Bestimmung des Pivotelementes die Beschränkungen per Fallunterscheidung zu berücksichtigen, so daß die Zahl der Variablen und Bedingungen unverändert bleibt [59].

1.6 Entwurf integrierter Schaltungen

1.6.1 Strukturierung des Entwurfsraumes

Der Entwurf digitaler Systeme ist heute durch eine hohe und ständig wachsende Komplexität gekennzeichnet. Auf der anderen Seite nehmen auch die Anforderungen an die Leistungsfähigkeit dieser Systeme ständig zu. In technologischer Hinsicht konkurrieren beispielsweise Erfordernisse wie minimaler Platzbedarf, hohe Performanz, geringe Leistungsaufnahme und große Zuverlässigkeit gegeneinander, um schließlich auch in Relation zu wirtschaftlichen Aspekten, wie Markteffizienz und Entwicklungskosten („Time-To-Market“) gesetzt zu werden.

Eine Bewältigung der Kombination dieser gegenläufigen Ziele erfordert eine sinnvolle Strukturierung des Entwurfsraumes, sowie ein strategisches und systematisches Vorgehen auf demselben. Eine Strukturierung digitaler Systeme kann zunächst hierarchisch, nach Entwurfsebenen gegliedert, erfolgen. Abbildung 1.3 illustriert dies am Beispiel eines einfachen Mikroprozessors: Die Systemebene stellt den Prozessor lediglich als „black box“ dar, spezifiziert die Eingänge und Ausgänge und den Zweck des Bausteins. Erst die algorithmische Ebene beschreibt seine Funktionsweise. Auf der RTL-Ebene (Register-Transfer-Level) wird das Design in einzelne Module mit zugewiesener Funktionalität gegliedert und der Datenfluß zwischen den Modulen definiert. Auf der Gatterebene steht die Realisierung der Module anhand Boolescher Funktionen und Speicher-

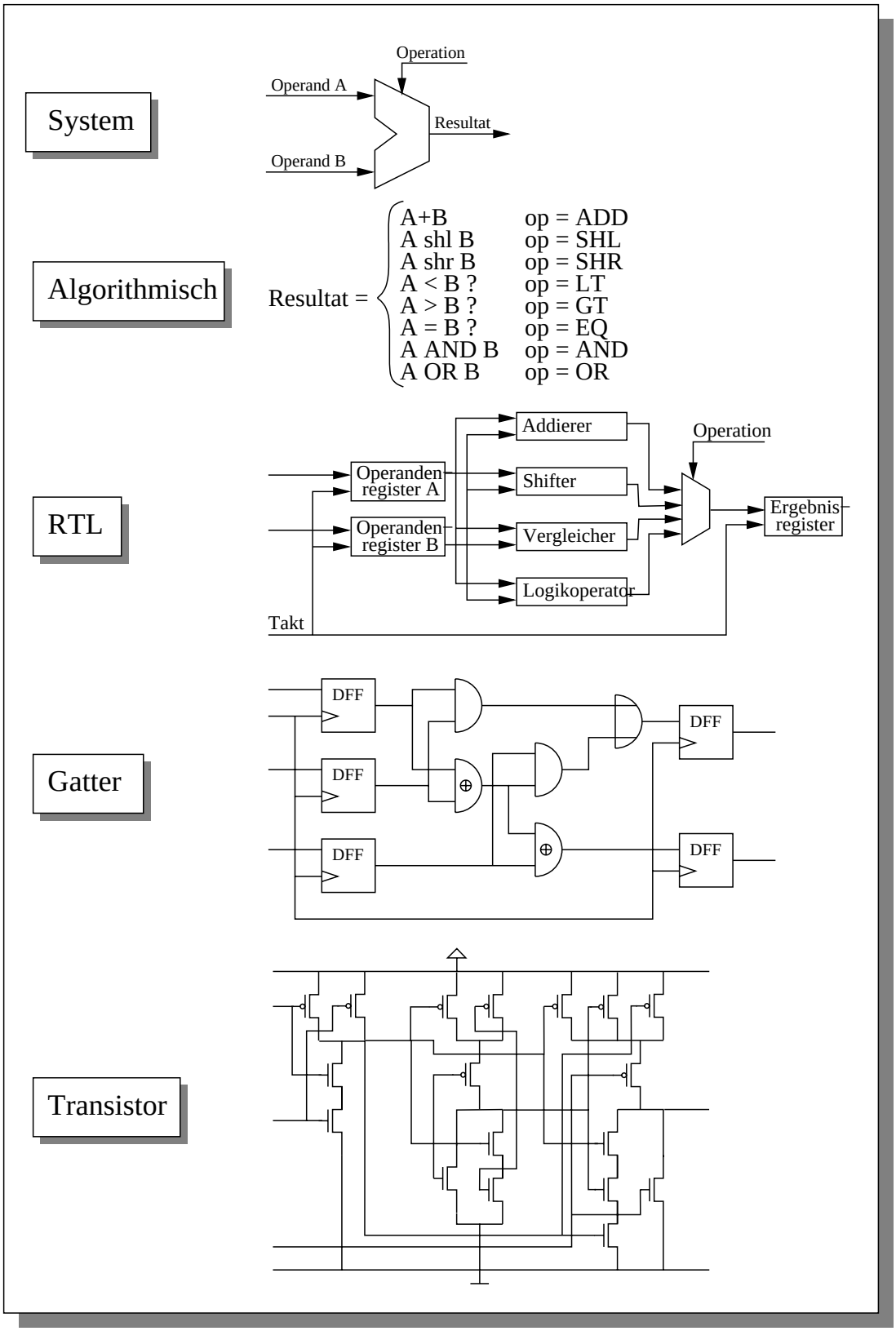


Abbildung 1.3: Entwurfsebenen digitaler Systeme

elementen im Mittelpunkt. Die physikalische Realisierung von Schaltkreisen betrachtet man auf der untersten, der Transistorebene.

Das Design eines digitalen Systems besteht nun stets aus einem stufenweisen Durchlaufen der Entwurfsebenen, wobei aufgrund der Komplexität der Systeme spezielle Softwarewerkzeuge eingesetzt werden müssen, welche die Übergänge unter Berücksichtigung gegebener Optimierungsziele vollziehen. Im Zuge der Konkretisierung eines digitalen Systems von der abstrakten funktionalen Beschreibung bis hin zur Schaltungsebene, können unterschiedliche Sichtweisen betrachtet werden [29].

Gajski und Walker kategorisierten diese Sichtweisen in ihrem bekannten *Y-Diagramm*, siehe Abbildung 1.4. In jeder der drei Sichtweisen *Verhalten*, *Struktur* und *Geometrie*

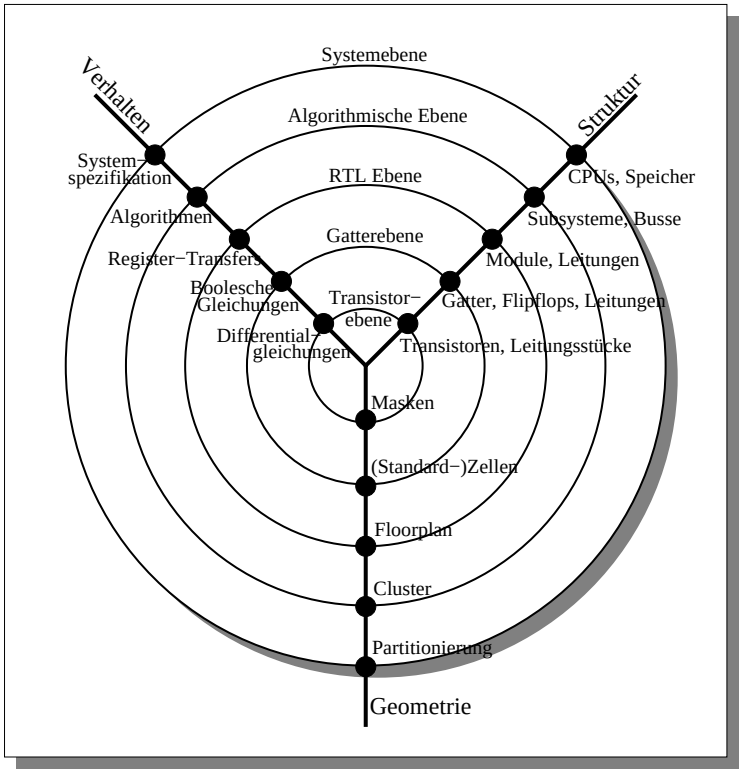


Abbildung 1.4: Y-Diagramm nach Gajski und Walker

kann ein digitales System vollständig beschrieben werden. Ferner ist es möglich und oft auch erforderlich, auf einer Entwurfsebene mehrere Sichtweisen zu betrachten, so beispielsweise bei plazierungsorientierter Logiksynthese, wenn ein Schaltkreis mit initial vorgegebener Gatterstruktur unter Berücksichtigung seines späteren Layouts hinsichtlich Verdrahtungsaufwandes zu optimieren ist.

1.6.2 Zur Problematik des rechnergestützten Entwurfs

In Abschnitt 1.6.1 wurde bereits illustriert, wie die Realisierung komplexer digitaler Systeme, bzw. seiner Komponenten (Schaltkreise), in einem schrittweisen Durchlaufen von Entwurfsebenen charakterisierbar ist. Ausgehend von einer initialen Spezifikati-

on, ob aus struktureller, verhaltensbasierter oder geometrischer Sicht, stellen sich im Zuge der Konkretisierung von der Systembene bis hin zur Transistorebene die verschiedensten Optimierungsprobleme, für die, aufgrund ihrer immensen Komplexität, bereits recht früh nach rechnergestützten Lösungsmöglichkeiten gesucht wurde. Der explosive technologische Fortschritt in der Halbleiterproduktion trieb die Komplexität digitaler Systeme voran und damit auch den Bedarf nach traktablen Methoden zur zufriedenstellenden Bewältigung der dementsprechend mitgewachsenen Probleme.

Der konventionelle Designablauf am Beispiel der RTL-Spezifikation eines Systems mit Standardzellen-Zielarchitektur stellt sich wie in Abbildung 1.5 aufgeführt, dar. Die Natur der Teilprobleme des Design Flows ist in der Regel diskreter Art, Grundlage der Lösungsverfahren sind kombinatorische Algorithmen. Allerdings sind die Probleme allesamt NP-schwer. Prinzipiell lassen sich die Teilprobleme in drei Problemkreise zusammenfassen, welche allerdings hinsichtlich Methodik und Verhalten bei der Optimierung untereinander stark zusammenhängen: *Logiksynthese*, *Logikoptimierung* und *Schaltkreislayout*.

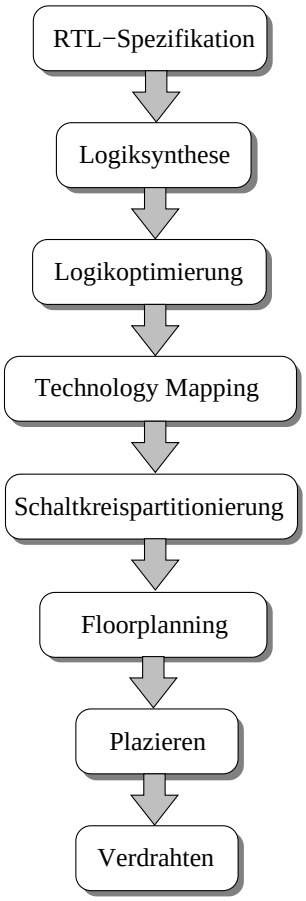


Abbildung 1.5: Standardzellen Design Flow

Der erste Problemkreis behandelt die Konstruktion effizienter, formaler Darstellungen von Schaltkreisen auf der Basis der gegebenen Modul-Spezifikation eines digitalen Systems. Zur zweiten Klasse gehören Probleme, die auf der erhaltenen Repräsentation eines Schaltkreises strukturelle Transformationen nach Optimierungskriterien wie Lei-

stungsaufnahme, Platzverbrauch oder Signalverzögerung, durchführen, dabei jedoch dessen funktionale Äquivalenz erhalten. Hierzu gehört auch die Überführung der Repräsentation auf die technologiespezifischen Gatter- und Transistorebenen im *Technology Mapping* Schritt. Die dritte und in der Zahl der Teilprobleme größte Kategorie beinhaltet Fragestellungen von geometrischer Natur. Dabei geht es im wesentlichen um die physikalische Anordnung der erhaltenen Netzwerke auf einem Träger, hier also einem Halbleiter-Mikrochip. Software-Werkzeuge, die diese Tätigkeiten leisten, werden unter dem Begriff *CAD-Tools (Computer-Aided Design Tools)* zusammengefaßt.

Bei der Darstellung und Optimierung von Schaltkreisen weicht man in der Regel auf attributierte verhaltensbeschreibende Modelle aus. Solche theoretischen Modelle bilden beispielsweise Repräsentationen durch Entscheidungsdiagramme, disjunktive Formen oder auch Boolesche Netzwerke, für die zum einen bereits eine Anzahl effizienter Transformationsmethodiken bekannt sind und die zum anderen auch die notwendige Abstraktion zur Entwicklung weiterer traktabler Verfahren liefern kann. Mit diesem Gedanken ist bereits der erste Aspekt des erwähnten Zusammenhangs der drei Problemkreise charakterisiert: durch die Festlegung der Repräsentation, des Modells, sind also Sichtweise und Möglichkeiten des Vorgehens bezüglich der durchzuführenden Transformationen in gewisser Weise fixiert. Ähnlich wie die oft vorhandene gegenseitige Beeinflussung der diversen Kostenfunktionen bei multikriteriellen Optimierungsproblemen, bestehen in der Regel auch Zusammenhänge zwischen den Optimierungskriterien von Problemen aus verschiedenen der genannten Kategorien. Ein Beispiel, in dem Logikoptimierungs- und Layoutkriterien unter Umständen mit- und auch gegeneinander agieren können, stellt die Optimierung eines Schaltkreises nach Platzverbrauch, Signalverzögerung, sowie die Minimierung der Leitungslänge im Rahmen seiner Layout-Realisierung dar: Platz- und Laufzeitkriterien arbeiten häufig gegeneinander, wohingegen beide Kriterien jedoch wiederum Einfluß auch auf die Optimierung des geometrischen Problems nehmen können.

Obgleich man sich der genannten Zusammenhänge bewußt ist, beschränkte man die gegenseitige Beeinflussung der Teilschritte des Design-Prozesses in der Regel auf eine entsprechende Ausrichtung ihrer jeweiligen Kostenfunktionen, verfolgte also eine strikte Trennung beispielsweise von Logikoptimierung zu Schaltkreispartitionierung, zu Plazierung und zu Verdrahtung — interaktive Verfahren, wie zum Beispiel die Integration von Floorplanning und (globalem) Verdrahten, wurden stattdessen aus Komplexitätsgründen nur wenige entwickelt.

Ein letzter wichtiger Aspekt des rechnergestützten Entwurfs wurde noch nicht angesprochen: die *Design Verifikation*. Man unterscheidet hier grundsätzlich zwischen *logischer Verifikation*, welche eine synthetisierte und optimierte Schaltung auf das Entsprechen ihrer initialen Spezifikation überprüft, und zwischen *physikalischer Verifikation*, die anhand eines erstellten Schaltkreislayouts geometrische Anforderungen *(Design Rule Check, DRC)*, oder elektrische Vorgaben *(Electrical Rule Check, ERC)* verifiziert. Viele Probleme der Design Verifikation besitzen eine weitaus höhere Komplexität, als die genannten Probleme der Design Konstruktion, manche davon sind nicht einmal entscheidbar im Sinne der Berechenbarkeitstheorie. Da die Thematik der Verifikation jedoch nicht Gegenstand der vorliegenden Arbeit ist, sondern nur aufgrund ihrer praktischen Relevanz erwähnt werden sollte, sei an dieser Stelle verwiesen auf geeignete Literatur: [78].

1.6.3 Technologien integrierter Schaltkreise

Zur Realisierung digitaler Systeme in Mikrochips auf Halbleiterbasis bieten sich unterschiedliche Möglichkeiten. Im Rahmen des *kundenspezifischen Designs* werden Mikrostrukturen im wesentlichen anwendungsorientiert neu entworfen oder zusammengefügt, wohingegen bei der *programmierbaren Logik* das realisierende Mikrosystem bereits komplett vorgefertig ist und lediglich noch anwendungsspezifisch programmiert wird (*Personalisierung*).

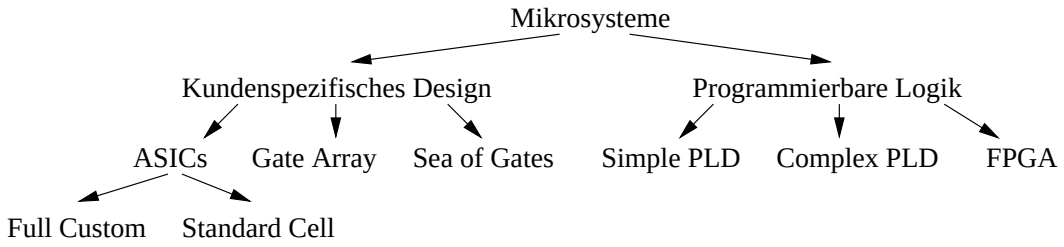


Abbildung 1.6: Klassifikation der Mikrosysteme

Hinsichtlich kundenspezifischer Designs unterscheidet man im allgemeinen zwischen *ASIC-*, *Gate-Array-* und *Sea-of-Gates*-Konzepten, wobei sich die ASIC-Konzepte[3] weiter in *Standard Cell* und *Full Custom* Design untergliedern lassen. Während beim Full Custom Design das komplette Layout des Chips entsprechend des zu realisierenden Schaltkreises neu entworfen wird, zieht man beim Standard Cell Design eine Bibliothek vorgefertigter Standard- und Komplexgatter heran, aus welcher Komponenten auf dem Chip plaziert und verdrahtet werden. Gate Arrays und Sea of Gates enthalten hingegen bereits vorentworfene, voll funktionsfähige Standardgatter, die lediglich noch nicht untereinander verdrahtet sind. Bei Gate Arrays werden die Verbindungen in speziellen Verdrahtungskanälen realisiert, bei Sea of Gates erfolgt die Verdrahtung „über die Zellen hinweg“ in separaten Verdrahtungslayern.

Mikrosysteme der programmierbaren Logik kann man heute grob in *Simple PLDs*, *Complex PLDs* und *FPGAs* klassifizieren. Zur Klasse der Simple PLDs gehören die produktterm-orientierten PLAs, aber auch EPROMs. In komplexen PLDs sind mehrere Simple PLDs zusammengefaßt und durch programmierbare Verdrahtungsstrukturen vernetzt. FPGAs, hingegen, stellen heute einen eigenen Typus dar, der sich in einer Vielfalt programmierbarer Logik- und Verdrahtungsressourcen manifestiert. Da FPGA-Architekturen Hauptgegenstand der Betrachtungen dieser Arbeit sein werden, wird sich Abschnitt 1.7 einer eingehenderen Charakterisierung widmen.

Doch bereits aus der vorangegangenen Darstellung der Klassifikation von Mikrosystemen werden zwei grundlegende technologische Unterscheidungsmerkmale offenbar: zum einen, daß Programmierbarkeit stets einher geht mit einer geringeren *Logikdichte* und zum anderen in einer höheren *Signalverzögerung* resultiert. Der erste Fall geht aus der Tatsache hervor, daß die Fähigkeit der Personalisierung eines Mikrosystems auf rein elektronischem Wege durch die Implementierung eines weiteren Schaltkreissystems auf dem Chip, der sogenannten *Programmierlogik*, erreicht wird. Die höhere Signalverzögerung bei programmierbarer Logik im Gegensatz zu kundenspezifischem

[3] *Application Specific Integrated Circuits*

Design ergibt sich aus den elektrischen Eigenschaften von Schaltern im programmierbaren Verdrahtungsnetzwerk.

Andererseits lokalisieren sich die Hauptvorteile programmierbarer Logik im wesentlichen in der kostengünstigeren Herstellung, da sie als Massenware produziert werden können, d.h. es fließen keine kundenspezifischen Herstellungsdaten ein, sowie aus der Flexibilität bei Redesign und Reimplementierung eines digitalen Systems, welche nicht die Fertigung eines neuen Chips nach sich ziehen. Bei der Entwurfsentscheidung, ob kundenspezifische oder programmierbare Logik, spielen heute, insbesondere im Hinblick auf Leistungsfähigkeit programmierbarer Logik, im allgemeinen kurze Time-to-Market-Zeiten eine wichtigere Rolle, als die Einsparung von ein paar Quadratmillimetern Chipfläche.

1.7 Field-Programmable Gate Arrays

Die Architekturen von Field-Programmable Gate Arrays (FPGAs) sind im wesentlichen jenen der gewöhnlichen Gate Arrays nachempfunden: die Komponenten sind hier jedoch einheitliche *Logikblöcke*, die auf die Berechnung einer Booleschen Funktion programmiert werden können, und die umgeben sind von Verdrahtungskanälen, in denen bereits Leitungen realisiert sind, auf welche Signale von und zu den Logikblöcken programmierbar aufgeschaltet werden können. Hauptmerkmal der FPGAs ist somit das komplette Vorhandensein von Logik- und Verdrahtungsressourcen, sowie deren Programmierbarkeit auf elektrischem Wege beim Anwender. Bis auf wenige Ausnahmen sind FPGAs auch rekonfigurierbar, so daß bereits erfolgte Konfigurationen durch neue überschrieben werden können.

1.7.1 Evolution der Rekonfigurierbarkeit

Die Ursprünge der rekonfigurierbaren Implementierung von Schaltfunktionen reicht bereits bis zu den Anfängen der Technologie integrierter Schaltkreise, etwa Ende der 1950er Jahre, zurück. Jedoch erst im Zuge der Einführung der Serienproduktion für integrierte Schaltkreise wurden erste Chips mit Feldern programmierbarer Logikzellen und Verdrahtungen, die sogenannten *Cellular Arrays*, hergestellt [56, 37, 36]. Die Hauptmotivation bei der Entwicklung der cellular arrays in den 1960ern bestand in der Reduktion der Chipkosten und der besseren Testbarkeit von Schaltkreisen für diskrete MSI Chips.

Die meisten cellular arrays bestanden, wie beispielsweise das in Abbildung 1.7 dargestellte Cutpoint Cellular Array, aus ein- oder zweidimensionalen Anordnungen diskreter Logikzellen. Klassifiziert man Architekturen heute hinsichtlich ihrer Logikflexibilität üblicherweise als grob- oder feingranular (coarse/fine grain), so wurden auch damals Logikzellen mit wenig Gattern (typischerweise bis fünf Gatter) als mikrozellulare Arrays, darüber hinaus als makrozellulare Arrays bezeichnet. Kaskaden von Logikzellen gleichen Typs auf den Arrays besaßen in der Regel einen unidirektionalen Logikfluß, d.h. von oben nach unten und von links nach rechts. Die Ausgänge an den Seiten dienten als Feedback-Eingänge zur Realisierung synchroner Arrays oder als Eingänge für benachbarte Arrays.

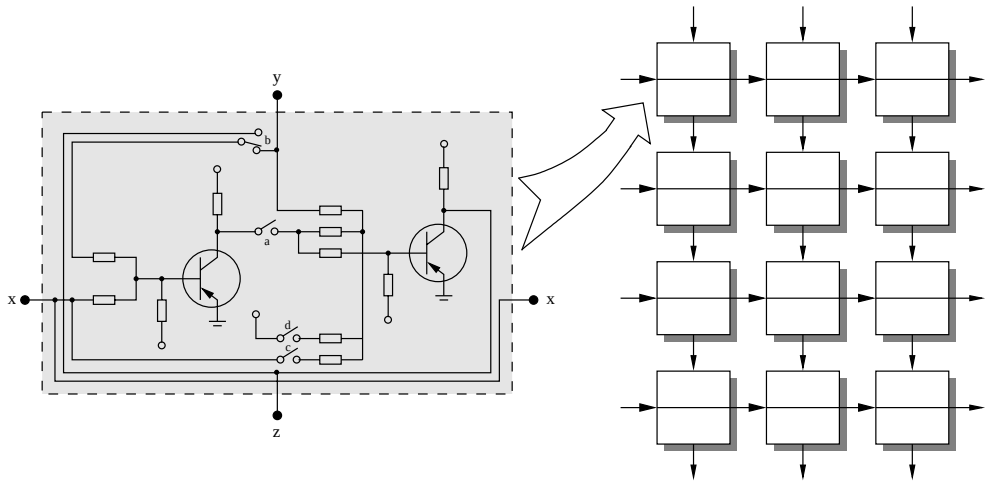


Abbildung 1.7: Cutpoint Cellular Array

Der Entwurf von Architekturen zellularer Arrays kennzeichnete sich durch eine starke Tendenz zur Feingranularität, wofür als Hauptgrund nicht zuletzt die damaligen technologischen Standards zu sehen sind [73]. In der Tat waren die zellularen Arrays Wegbereiter für spätere Technologien, wie PLAs, systolische Arrays und anwendungsspezifische zellulare Prozessoren. Diese frühen rekonfigurierbaren Architekturen fristeten aufgrund ihrer niedrigen Logikkapazitäten jedoch nur ein Dasein als glue logic inmitten der übrigen SSI- und MSI-Bausteine. Erst durch die Einführung der VLSI-Technologie, mit der Chipkapazitäten von mehr als einer Million Gattern möglich wurden, waren die Voraussetzungen gegeben, das erste feldprogrammierbare Gate Array zu entwickeln. Es wurde im Jahre 1986 von der amerikanischen Firma Xilinx vorgestellt und prägte den Typus der zweidimensionalen Architekturen mit Insel-Struktur, welcher noch die Grundlage vieler heutiger kommerzieller Architekturen bildet.

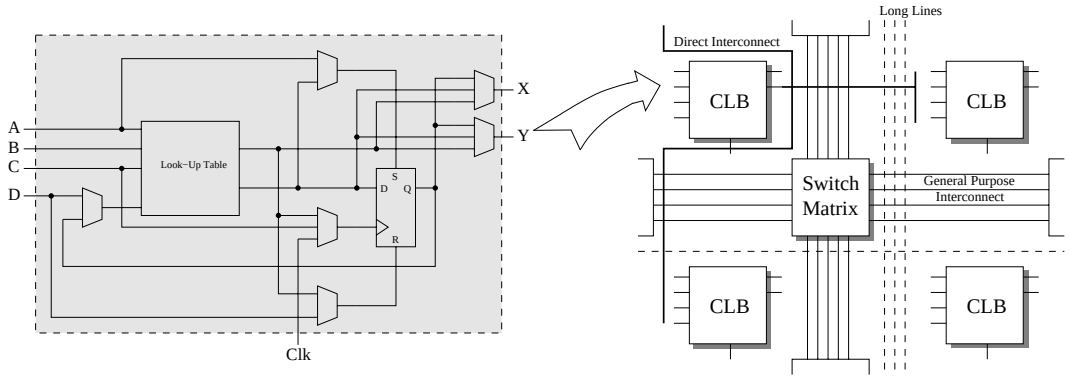


Abbildung 1.8: Xilinx XC2000

Während die zellularen Arrays noch aus Blöcken bestanden, in die Logik- und/oder Verdrahtungsaufgaben programmierbar waren, wiesen die meisten späteren FPGAs eine deutliche Trennung von Logik- und Verdrahtungsressourcen auf. Dies lag im wesentlichen an der Wiederverwendbarkeit bekannter und mitunter bereits sehr ausgefeilter Algorithmen zur Plazierung und Verdrahtung von Standardzellen. Vereinzelt gab es

dennoch Ansätze, die Architekturen von routingintegrierenden Basisblöcken anwandten, wie z.B. das 1991 vorgestellte Labyrinth-FPGA [11], dessen Technologie in zweiter Generation später in ein kommerzielles FPGA einging. Zu diesen Ansätzen ist ferner auch das auf derselben Konferenz vorgestellte Triptych-FPGA zu zählen [24], welches mit segmentierten vertikalen Verdrahtungskanälen einen Hybridansatz verfolgte. 1995 wurde das inzwischen patentierte Triptych-Konzept ein weiteres Mal publiziert, nun ausgestattet mit einem Erweiterungsvorschlag, mit dem es zur ersten reprogrammierbaren Architektur für asynchrone Schaltkreise wurde [14]. Ebenfalls bereits im Jahre 1995 wurden die ersten Ansätze dreidimensionaler FPGA-Architekturen vorgestellt [4]. Kurze Zeit später hatten Leeser et al. die Triptych-Architektur als Grundlage für ihre dreidimensionale FPGA-Architektur Rothko adaptiert [48].

Doch worin lag die Motivation, Forschung mit Architekturen zu betreiben, für die es weitaus schwieriger war, geeignete CAD-Systeme zu entwickeln, als bei grobgranulareren, zwischen Logik und Verdrahtung trennenden Ansätzen? Erstaunlicherweise wurde zu einem Zeitpunkt, an dem die FPGAs gerade erst begannen, ihre strukturellen Konzepte zu festigen und im wesentlichen nur hinsichtlich der Kapazität eine Weiterentwicklung voran zu treiben, bemerkt, daß viele der Logik- und Verdrahtungsressourcen bekannter Architekturen nach der Programmierung ungenutzt blieben. Man suchte nach Architekturen, welche sich den FPGAs am häufigsten vorgeworfenen Problemen entgegenstellten: nämlich der relativ geringen Logikdichte, der relativ geringen Ausnutzung und der relativ geringen Performanz.

1.7.2 FPGA-Technologien — State-of-the-Art

Seit der Einführung der FPGAs Mitte der 1980er Jahre, wurden zum Teil sehr unterschiedliche Architekturen entwickelt. Einige der Entwurfsschemata haben sich bis heute durchgesetzt, wurden dank verfeinerter Herstellungstechnologien hinsichtlich Logik- und Verdrahtungsflexibilität nach und nach erweitert und zu sehr erfolgreichen kommerziellen Produkten gemacht. In diesem Abschnitt sollen die beiden heute marktführenden Architekturkonzepte kurz vorgestellt werden, als typisches Beispiel, wie sich die Architektur-Situation heute darstellt. Für eine Übersicht über die wichtigsten unterschiedlichen Ansätze sei der Leser verwiesen auf [17, 16, 79].

Das Xilinx Virtex FPGA

Abbildung 1.9 zeigt links die Detailskizze eines konfigurierbaren Logikblocks (CLB) des Virtex-E FPGAs der Firma Xilinx. Neben beliebigen vierstelligen Booleschen Funktionen sind mittels der Look-Up-Tables auch RAM oder Shift-Register realisierbar. Ferner erlaubt der CLB durch die XOR-Gatter eine direkte Konfiguration von Volladdierern und auch zwei Carry-Ketten werden unterstützt. Zudem enthält der CLB noch zwei flexible D-Flipflops. Rechts in der Abbildung ist der Aufbau des gesamten Virtex-E schematisch dargestellt. Das zweidimensionale Array aus alternierenden Spalten von CLBs und RAM-Blöcken ist umgeben von ringförmigen Verdrahtungsressourcen, die Verdrahtbarkeit insbesondere hinsichtlich *pin locking*, also fixierter Anordnung von I/O-Signalen, garantieren soll [88].

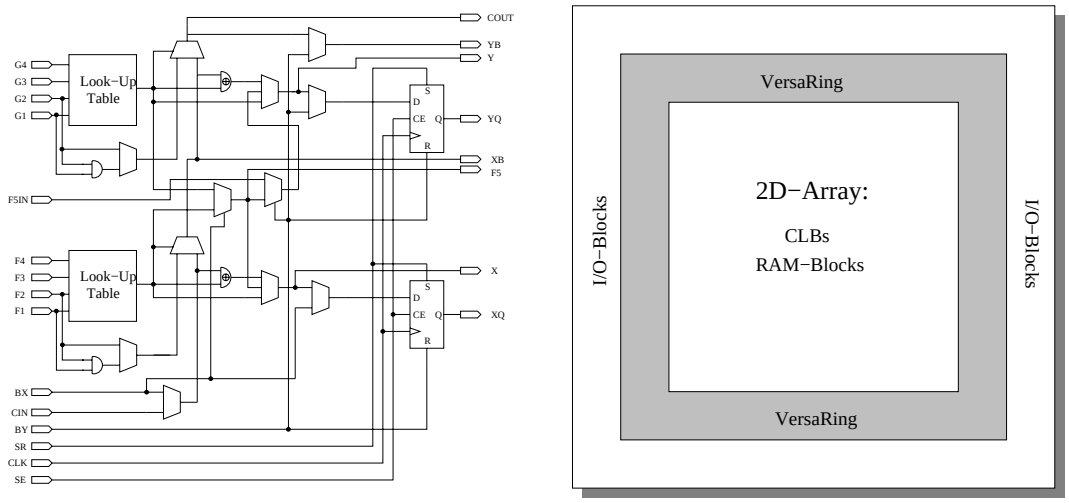


Abbildung 1.9: Xilinx Virtex-E FPGA

Das Altera Apex II FPGA

Der Aufbau eines Apex II FPGAs der Firma Altera ist in Abbildung 1.10 dargestellt. Die Abbildung zeigt links ein sogenanntes Logikelement (LE), das neben einer Look-Up-Table für vierstellige Boolesche Funktionen ebenfalls zwei Carry-Ketten unterstützt. Das D-Flipflop des LEs kann mittels einer Reihe externer Steuersignale geladen, und zurückgesetzt werden. Jeweils zehn LEs bilden zusammen mit einer lokalen Verdrahtungsstruktur einen *Logic Array Block (LAB)*, von welchen wiederum mehrere zu einem *MegaLAB* mit umspannenden Verdrahtungsresourcen zusammengefaßt sind. Das Apex II besteht nun aus einer zweidimensionalen Anordnung von MegaLABs, die in das chipumspannende *Fast Interconnect* Netzwerk eingebettet sind. Damit besitzt das Apex FPGA, im Gegensatz zum Virtex, mehrstufige, *hierarchische* Verdrahtungsstrukturen [7].

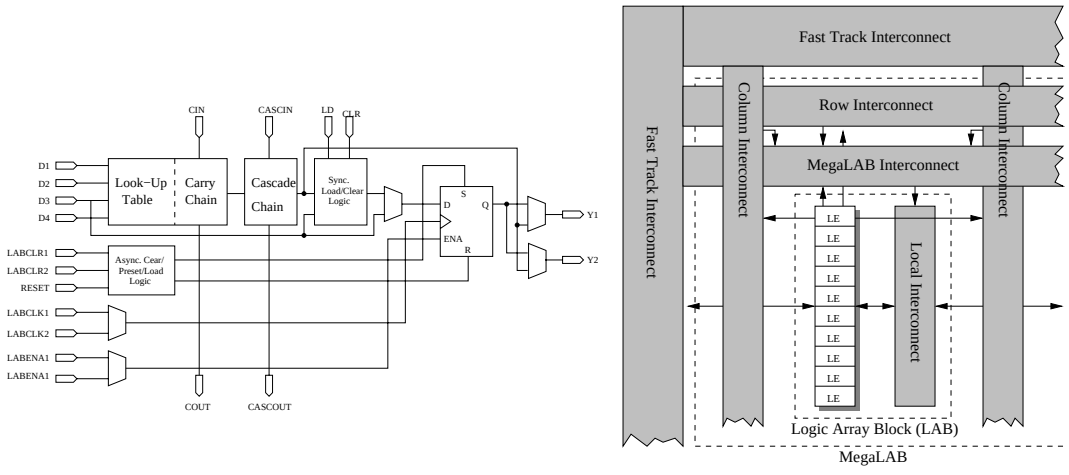


Abbildung 1.10: Altera Apex II FPGA

Configurable System-on-a-Chip

Vor wenigen Jahren trat erstmals eine neue Generation von Mikrosystemen auf, die immer mehr im Bereich der Eingebetteten Systeme (ES) zum Einsatz kommt und deren Entwicklung mehr und mehr vorangetrieben wird: die sogenannten *Configurable Systems-on-a-Chip (CSoC)*. Nach nunmehr über 15 Jahren haben die FPGAs eine Trendwende erfahren. Sie avancierten von ihrem einstigen GlueLogic-Image über den anwendungsspezifischen Prozessor hin zum „in-System“ -rekonfigurierbaren System, indem neben der konfigurierbaren Logik auch gebrauchsfertige Einheiten bekannter Mikroprozessoren, Speicher und Interface-Komponenten auf einem Chip untergebracht wurden. Ein aktuelles Beispiel stellt hier das Virtex-II Pro FPGA dar, das neben der oben bereits vorgestellten FPGA-Architektur des Virtex-E auch zwei Einheiten der IBM PowerPC 405 RISC CPU und ein Gigabit Netzwerk-Interface vereinigt [89].

1.7.3 CAD-Werkzeuge

Wie im Abschnitt 1.7.1 bereits angeführt, war es möglich, unter einer Trennung von Logik- und Verdrahtungsressourcen hinsichtlich der rechnergestützten Implementierung von Schaltkreisen in FPGAs auf die bekannten Verfahren für Standardzellen zurück zu greifen, wobei sich auch theoretische Studien diese „Kongruenz“ zunutze machten [18, 25]. Der klassische FPGA Design Flow stellte sich somit wie in Abbildung 1.11 dar.

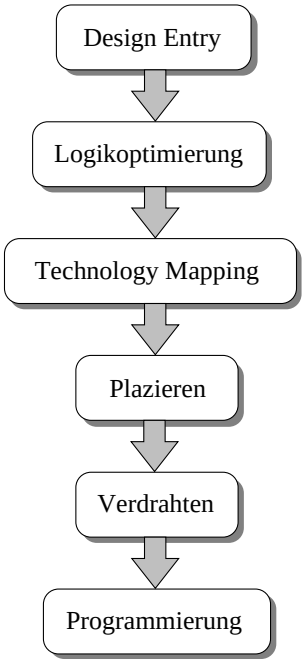


Abbildung 1.11: FPGA Design Flow

Die Spezifikation eines zu implementierenden Schaltkreises erfolgt im sogenannten Design-Entry-Schritt. Möglichkeiten der Spezifikation sind i.a. die graphische Schaltplaneingabe als hierarchisches Netzwerk von Bausteinen, ferner die Strukturbeschreibung mittels einer Hardwarebeschreibungssprache oder auch Spezialeditoren, wie sie

beispielsweise für endliche Automaten in kommerziellen Hardwareentwurfswerkzeugen häufig implementiert sind. Die recht schnell zunehmende Komplexität der Schaltkreise trieb jedoch die Entwicklung ausgefeilter Werkzeuge zur Logiksynthese voran, so daß etwa seit Beginn der 1990er Jahre ein deutlicher Trend eher zur textuellen Eingabe mittels Hardwarebeschreibungssprachen beobachtet werden konnte. Ausgehend vom Design-Entry-Schritt, dessen Resultat in aller Regel auf unterster Ebene eine Netzliste darstellt, werden logikoptimierende Verfahren eingesetzt, die den Schaltkreis nach bestimmten Kriterien, beispielsweise hinsichtlich Tiefe und Knotengrad bearbeiten. Der Technology-Mapping-Schritt untersucht den Schaltkreis unter Berücksichtigung der Logikressourcen der Ziel-FPGA-Architektur und faßt einzelne Schaltkreisknoten zusammen, wenn sie in einem Logikblock realisierbar sind bzw. spaltet Knoten auf, wenn sie nicht in einem Logikblock realisierbar sind. Hierbei wurden in Abhängigkeit der Logikblock-Technologie und der Architektur die unterschiedlichsten Ansätze entwikkelt. Beim Placement-Schritt werden die im Mapping Prozeß konstruierten Schaltkreiskomponenten nach Kriterien der Verdrahtbarkeit und der zu erwartenden Verdrahtungslänge konkreten Logikressourcen des FPGAs zugeordnet. In der Routing-Phase wird nun für jedes Netz, unter Berücksichtigung der im vorangegangenen Schritt erhaltenen Positionierung seiner Endpunkte, ermittelt, welche Verdrahtungsressourcen zur Realisierung entsprechender Routen geeignet zu programmieren sind. Die Menge der aus dem Placement- und dem Routing-Schritt erhaltenen Belegungen der Programmierbits wird als eine Konfiguration des FPGAs bezeichnet. Das Laden eines FPGAs mit einer Konfiguration erfolgt in der Regel entweder über ein spezielles Programmiergerät oder die inzwischen standardisierte JTAG-Schnittstelle.

1.8 Problemstellung und Konzept

Ziel und Gegenstand der vorliegenden Arbeit ist es, flexibel spezifizierbare feldprogrammierbare Architekturen mit repetitiver Struktur zu modellieren und sie hinsichtlich ihres Verhaltens in Platzverbrauch und Signalverzögerung zu untersuchen. Verschiedene Architekturtypen werden hinsichtlich Granularität und Flexibilität skaliert und mittels Plazierung und Verdrahtung von Modell- sowie Benchmark-Schaltkreisen bewertet, wobei insbesondere das Verhältnis zwischen Logik- und Verdrahtungsflexibilität beobachtet wird.

Erscheint diese kurzgefaßte Beschreibung der Problemstellung auf den ersten Blick relativ klar, so erweist sich Forschung auf diesem Gebiet dennoch als recht schwierig – und dies selbst aus unterschiedlichen Perspektiven. In den folgenden Unterabschnitten soll versucht werden, dem Leser hierüber einen kompakten Überblick zu liefern.

1.8.1 Zur Problematik der Architekturforschung

Frühere Arbeiten

Hinsichtlich der Quellensituation bietet sich beim Thema „neue FPGA-Architekturen“ ein eher gemischtes Bild. Zwar sind Publikationen über neue FPGAs bis heute beinahe an der Tagesordnung, denn die große Zahl technischer Anwendungen bringt noch im-

mer neue Anforderungen und Erfahrungen mit sich, aus denen Neuerungen und Verbesserungen an Architekturen entwickelt werden. Ungeachtet dessen hat man es bei genauerem Hinsehen dennoch mit zwei Problemkomplexen zu tun:

Zum einen behandeln die meisten Publikationen lediglich kommerzielle Architekturen, wie in der Einleitung bereits gezeigt wurde und auch nachfolgend noch zu sehen sein wird. Derzeit gibt es zwei große Konkurrenten, die sich etwa zwei Drittel des Marktes teilen und die gemeinsam mit privaten und öffentlichen Stiftungen die Forschung an universitären Einrichtungen vornehmlich in Kanada und den USA viele Jahre unterstützt haben und noch heute unterstützen.

Ein weiteres Problem hinsichtlich von Quellen liegt in der Tatsache begründet, daß Details über existierende Architekturen, insbesondere auch Entwurfsmotivationen, welche vorrangig von den Unternehmen selbst spezifiziert sind, nicht veröffentlicht werden. Zu kommerziellen Devices frei verfügbar ist praktisch ausschließlich Dokumentation für Anwender.

Anfang bis Mitte der 1990er Jahre erschienen jedoch zahlreiche Arbeiten, die sich zum Teil um theoretische, in den meisten Fällen aber um empirische Betrachtungen bemühten. Einige davon seien im folgenden erwähnt:

- Prädiktionsverfahren für die Kanalausnutzung (Verdrahtbarkeit) [19, 87, 18]
- Splitting von Look-Up-Tables in den Logikzellen (Erhöhung der Logikdichte) [34]
- Auswirkung der Anzahl der Look-Up-Table-Eingänge der Logikzellen auf die Platzeffizienz. [65]
- Auswirkung des Logikzellen-Aufbaus auf die Performanz des FPGAs [72]
- Auswirkung der Verdrahtungsflexibilität auf die Verdrahtbarkeit [66, 64]
- Auswirkung der Segmentierung von Leitungen auf die Performanz [16]

Insbesondere, akzeptiert man die Selbstverständlichkeit, mit der in obigen Publikationen allgemein von „FPGAs“ gesprochen wird, so scheint auf den ersten Blick das Thema Architekturen von allen Seiten gründlichst durchleuchtet und die „beste“ Architektur somit schon seit langem gefunden. Leider liegt jedoch ausnahmslos allen oben genannten Studien ein und dieselbe kommerzielle Architektur zugrunde. Nur wenige Publikationen betrachteten andere Architekturen [20, 44]. Alle bis dato bekannten architekturübergreifenden Studien bestehen überwiegend in empirischen Vergleichen lediglich kommerzieller Architekturen. So wurde auch beispielsweise für die bereits vorgestellte, zweidimensionale inselstrukturierte FPGA-Architektur etwa ab Mitte der 1990er Jahre das bislang einzige adaptive Plazierungs- und Verdrahtungswerkzeug zum Zwecke des Vergleichs unterschiedlicher Ausprägungen der Logik- und Verdrahtungsresourcen dieses Architekturtypus entwickelt: *Versatile Place and Route (VPR)* [13, 12].

Insgesamt beurteilt, gab es in der Vergangenheit relativ wenige Studien, die sich, losgelöst von derartigen industriellen Vorgaben, dem Vergleich unterschiedlicher Architekturen und neuer Architekturtypen widmeten, also nach Möglichkeiten zur Modellierung allgemeiner feldprogrammierbarer Strukturen, sowie Wege zu deren Bewertung suchten.

Zielsetzung der Bewertung

Grundlegende Aspekte der Bewertung feldprogrammierbarer Architekturen stellen Fragestellungen dar, wie sich bei der Implementierung von Schaltkreisen auf einer gegebenen FPGA-Architektur die Anordnung und Verhältnisse der Ressourcen auf bestimmte Kostenmaße auswirken. Im Mittelpunkt der Betrachtungen steht dabei, auf welche Weise ein bestimmter Architekturtyp verbessert werden kann hinsichtlich Zielsetzungen, wie:

- Erhöhung der Logikdichte (Platzeffizienz)
- Erhöhung des Grades der Ausnutzung
- Verbesserung der Performanz

Diese drei Ziele sind offensichtlich konkurrierende Ziele, denn die niedrige Logikdichte von FPGAs hat ihre prinzipielle Ursache in der Präsenz ausgedehnter Systeme von Verdrahtungsressourcen, welche andererseits jedoch ausreichend Möglichkeiten zur Realisierung kurzer Verdrahtungswege bzw. zur Verdrahtbarkeit überhaupt bieten sollen. Architekturen mit stark eingeschränkten Logik- und Verdrahtungsressourcen resultieren hingegen in einem niedrigen Nutzungsgrad. Eine Möglichkeit zur Optimierung eines gegebenen Architekturtyps besteht somit in der Ermittlung eines geeigneten Tradeoffs zwischen Logik- und Verdrahtungsressourcen.

Als Ansatzpunkte bieten sich zunächst die Logikzellen, deren Flexibilität zur Realisierung von Logik neben der Chipfläche auch zumindest die lokalen Verdrahtungsressourcen mitbeinflussen. Ferner sind die Verdrahtungsressourcen skalierbar hinsichtlich Zahl der Leitungen, deren Unterteilung in Segmente, Flexibilität der Anbindung an die Logikzellen und Flexibilität in der Programmierung von Verdrahtungspfaden.

Während sich die genannten Zielsetzungen im wesentlichen auf strukturelle Eigenschaften einer Architektur stützen, gibt es auch Kostenmaße, die mehr von der technischen Realisierung der Architektur-Ressourcen beeinflußt sind, wie beispielsweise die Optimierung des Stromverbrauchs (*power consumption*). Da sich die vorliegende Arbeit jedoch in erster Linie mit strukturelle Eigenschaften von FPGA-Architekturen und deren Auswirkungen befaßt, werden derartige Kostenmaße nicht weiter betrachtet.

1.8.2 Neuer Ansatz

Die grundlegende Idee zu einer neuen Entwurfsphilosophie stammt aus einer früheren Arbeit des Verfassers [79], in der Verhältnismäßigkeiten gängiger FPGA-Architekturen kritisch hinterfragt wurden, wobei durch den Vergleich mit einer Reihe neu vorgeschlagener, feingranularer Minimalarchitekturen ein ungünstiges Ressourcenverhältnis der kommerziellen Architekturen nachgewiesen werden konnte. Während diese Vergleiche empirisch anhand manueller Realisierungen kleiner Schaltkreise erfolgten, wurden auch Bewertungsmaße für Basisblöcke definiert, die Logik- und Verdrahtungsflexibilitäten von Architektursegmenten ins Verhältnis zu den jeweiligen Chipflächenkosten setzten. Allerdings wurde auch gezeigt, daß die Fortsetzbarkeit solcher *relativen Bewertungsmaße* auf Architekturen im allgemeinen nicht gegeben ist, wodurch klar war, daß

eine theoretische Bewertung von Architekturen eine empirische Bewertung durch die tatsächliche Realisierung von Schaltkreisen nicht ersetzen kann.

Ein großer Teil der vorliegenden Arbeit befaßt sich sodenn auch mit der Entwicklung eines generischen Systems zum rechnergestützten Layout (Plazierung und Verdrahtung) von Schaltkreisen auf flexibel spezifizierbaren feldprogrammierbaren Architekturen. Der Begriff der Generizität der entwickelten Verfahren besitzt in diesem Falle die Bedeutung, daß die Algorithmen nicht spezialisiert sind auf einen bestimmten Architekturtypus, um beispielsweise durch geeignete Abstraktion auf eine festgelegte Geometrie der Strukturen eingehen zu können. Ziel ist vielmehr, ein System zu kreieren, das in der Lage ist, alle spezifizierten Ressourcen einer Architektur auszunutzen. Die Grundkonzeption baut sich deshalb auf zwei Säulen auf: Zum einen sollen eher *quasi-erschöpfende Methoden* eingesetzt werden, deren Suchraum im wesentlichen lediglich durch den vorgegebenen Plazierungsraum einer Architektur beschränkt ist. Der zweite Eckpfeiler der Konzeption besteht in der Entwicklung *integrierter Verfahren*, die mehrere Phasen des Design Flow vereinigen, um die Determiniertheit von Lösungen durch Ergebnisse früherer Phasen zu senken.

Diese Grundkonzeption zur Generizität bringt natürlich große Probleme mit sich. Wie in den folgenden Kapiteln noch gezeigt werden wird, sind bereits die meisten Teilprobleme des Entwurfszyklus NP-hart. Neben exakten Methoden werden deshalb auch Heuristiken entwickelt, wobei die Kostenfunktionen hinsichtlich Schaltkreis und Architektur adaptiv gewählt werden, indem sie beider Struktur zu berücksichtigen suchen. Dennoch kann in den meisten Fällen nur „motiviert“ werden, wie die Kostenfunktion überhaupt auf das jeweilige gedachte Optimum zielt. Insbesondere in diesen Fällen ist es dann oft unmöglich, eine geeignete Schranke für die Güte der Approximation zu ermitteln – ein generelles Hauptproblem bei Optimierungen.

Als Konsequenz hiervon ist festzustellen, daß die „Performanz“ eines Layout-Systems kaum absolut bewertet werden kann, sondern allenfalls relativ zu einem anderen System. Obwohl dies bei Verfahren innerhalb einer einzelnen Entwurfsphase durchaus möglich ist und auch so gehandhabt werden wird, kann dies bei dem im vorliegenden Falle zu entwickelnden Gesamtsystem keine Rolle spielen, da es schlichtweg das bislang einzige seiner Art sein wird. Die vorliegende Arbeit wird also auf einem völlig anderen, als sonst üblichen Hintergrund zu betrachten sein: nämlich nicht der Bewertung von Methoden, sondern der Bewertung von Architekturen. Es soll und kann also nicht Ziel dieser Arbeit sein, Verfahren zu liefern, die jene bereits existierender, architektur-spezialisierter CAD-Systeme überbieten oder die andererseits zu beliebigen Benchmark-Schaltkreisen in möglichst kurzer Zeit ein Layout generieren.

Ein generisches Layout-System

Der erste Schritt hin zu einem generischen Layout-System erfordert zunächst ein geeignetes graphentheoretisches Modell für FPGA-Architekturen, welches deren inhärent repetitive Struktur unmittelbar unterstützt, sowohl hinsichtlich einer Expansion des Plazierungsraumes, als auch einer Relokation bereits plazierter Schaltkreisteile. Ferner soll auch auf die Einschränkung der zweidimensionalen Geometrie verzichtet werden, die bei den kommerziellen FPGA-Typen sonst üblich ist.

Ein weiteres, wichtiges Kennzeichen des neuen Ansatzes besteht im Prinzip der *Identifikation von Logik und Verdrahtung*. Während bei bisherigen Architekturen streng zwischen Logik- und Verdrahtungsressourcen getrennt wird, sollen diese in unserem Modell konkurrieren. Wir sprechen hierbei von *routingintegrierenden Basisblöcken*. Dies ist so zu verstehen, daß ein Basisblock mit Logikressourcen auch Verdrahtungsaufgaben erfüllen kann, die exklusiv zu den Logikaufgaben genutzt werden können. Hintergrund ist das Erreichen einer höheren Flexibilität beim Verdrahten durch den Einsatz ungenutzter Logikressourcen für Verdrahtungsaufgaben. Andererseits kann die Nutzung von Logik auch bestimmte Verdrahtungswege einschränken. Dieses Konzept wurde gewählt, da insbesondere ressourcenminimalistische und feingranulare Architekturtypen Gegenstand der Betrachtung sein sollen.

Eine Folge dieser Konkurrenz von Logik und Verdrahtung ist, daß konventionelle Vorgehensweisen nicht mehr anwendbar sind, bei denen zunächst alle Logikblöcke plaziert und beim nachfolgenden Verdrahten lediglich mittels der maximalen Anzahl der die Routingkanäle nutzenden Netze auf Verdrahtbarkeit getestet wird. Dies stellt im übrigen auch den Hauptgrund dar, weshalb das oben bereits genannte *VPR*-System nicht an das geplante Architekturmodell angepaßt werden kann: die Plazierung hat bei *VPR* lediglich Auswirkung auf die Kanaldichte, aber keine Auswirkung auf die unmittelbare Konfigurierbarkeit von Verdrahtungsaufgaben.

Abbildung 1.12 zeigt ein Schema des generischen Layout-Systems. Ausgangspunkt werden die Spezifikationen einer Architektur und eines Schaltkreises sein, wobei davon ausgegangen wird, daß der Schaltkreis bereits in einer Netzliste vorliegt und zuvor in einem Technology-Mapping-Schritt auf die verfügbaren Logikressourcen der Zielarchitektur (Look-Up-Tables) angepaßt wurde.

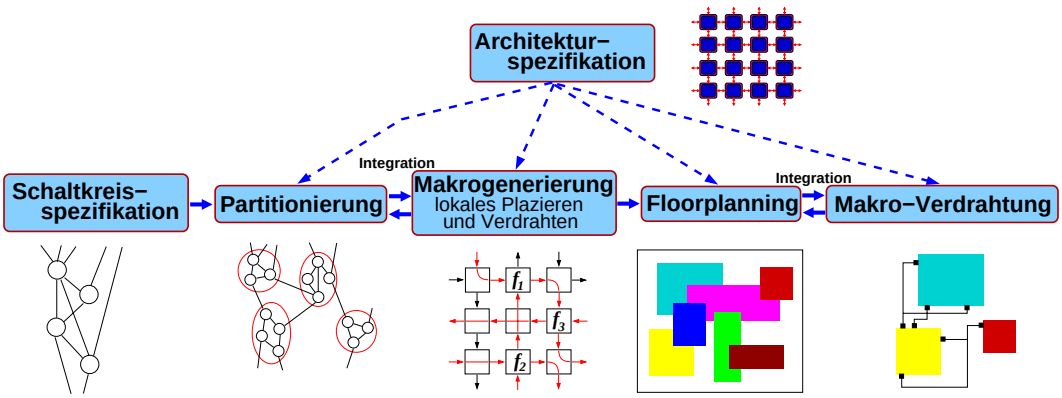


Abbildung 1.12: Schema des generischen Layout-Systems

Aufgrund der hohen Komplexität der generischen Berechnung von Schaltkreis-Layouts, wurde als Verfahren nun ein Makro-Konzept gewählt, nach dem der Schaltkreis zunächst partitioniert und aus den einzelnen Partitionen entsprechend der vorgegebenen Spezifikation der Zielarchitektur plaziert und verdrahtet wird. Diese Vorgehensweise läßt im übrigen auch die Einbeziehung von im Rahmen des Schaltkreisentwurfs genutzten Makros aus Komponenten-Bibliotheken zu.

In der nachfolgenden Floorplanning-Phase werden die Makros im Plazierungsraum angeordnet, um schließlich deren offene Pins in einem letzten Schritt zu verdrahten.

Bei dieser Konzeption sind die die Phasen *Partitionierung* und *Makrogenerierung,* sowie die Phasen *Floorplanning* und *Makro-Verdrahtung* integrativ ausgelegt. Das heißt beispielsweise, daß die Generierung der Makros sukzessive mit der Partitionierung des Schaltkreises erfolgt, die Algorithmen der beiden Phasen also gegenseitig aufeinander Einfluß nehmen. Gleiches gilt für den Floorplanning und den Makro-Verdrahtungs-Schritt.

1.8.3 Vorgehensweise

Der weitere Verlauf dieser Arbeit wird sich wie folgt gestalten: Im nachfolgenden Kapitel 2 wird zunächst ein Modell für allgemeine konfigurierbare Verdrahtungsnetzwerke vorgestellt und auf deren Eigenschaften, sowie Optimierung beim Entwurf eingegangen. Danach werden formale Modelle für konfigurierbare Zellen, Architekturen und Routen vorgestellt. Schließlich werden noch Modelle für Platzverbrauch und Performanz angegeben, mit deren Hilfe später Architekturen zu bewerten sind. Das Kapitel 3 betrachtet das integrative Verfahren zur Schaltkreispartitionierung und Makrogenerierung. Das in unserem Layout-System angewandte, neue Floorplanning-Konzept wird in Kapitel 4 vorgestellt. Bei der Integration des inkrementellen Floorplanning-Verfahrens mit dem Verdrahtungsschritt, spielt insbesondere auch die Abschätzung der Routenlängen eine wichtige Rolle. Ferner wird eine Heuristik zur Endverdrahtung der Makros angegeben. Dem letzten Kapitel ist die Vorstellung verschiedener Architekturen vorbehalten, sowie deren Bewertung mittels unseres generischen Layout-Systems.

Kapitel 2

Modellierung feldprogrammierbarer Architekturen

Dieses Kapitel behandelt die Frage nach Möglichkeiten einer formalen Modellierung feldprogrammierbarer Architekturen vor dem Hintergrund des Entwurfs generischer Verfahren zur Implementierung sequentieller Look-Up-Table-Schaltkreise. Dabei soll der Anspruch an eine Modellierung hinsichtlich des Konkretisierungsgrades möglichst hoch gewählt werden, das heißt Struktur und Fähigkeiten einer Architektur sollen sich über eine möglichst konkrete technische Realisierung deren Komponenten definieren.

Im nachfolgenden Abschnitt werden zunächst die Aspekte dargelegt, die zur schließlich gewählten Modellierung mittels periodischer Graphen auf der Basis zellgekapselter Multiplexer-Netzwerke führten. Die Abschnitte danach enthalten im wesentlichen die notwendigen formalen Definitionen, sowie Darstellungen wichtiger Charakteristika. Den Abschluß dieses Kapitels bildet ein Abschnitt, in dem Bewertungsmetriken für Architekturen hinsichtlich Kapazität, Flexibilität und Performanz vorgestellt werden.

2.1 Aspekte der Modellierung

Feldprogrammierbare Architekturen sind digitale Systeme zur Realisierung sequentieller Schaltkreise. Die Struktur ihrer Ressourcen ist für den Anwender vollständig festgelegt, wobei mittels einer geeignet konsistenten Programmierung elektrischer Schalter die Realisierung des zu implementierenden Schaltkreises definiert wird.

Betrachtet man die Entwurfsebenen digitaler Systeme (vgl. Abbildung 1.3), so stellen sich Transistor- und Gatterebene als die konkretesten Stufen dar. Da wir jedoch nicht an einer physikalischen sondern eher einer logik-strukturellen Sichtweise interessiert sind, ferner ein zu implementierender Schaltkreis ebenfalls als Graph über boolesche Funktionen repräsentierende Knoten vorliegt, wurde als die für unsere Zwecke geeignetste Stufe einer möglichst konkreten Modellierung die Gatterebene gewählt.

Die Komponenten einer feldprogrammierbaren Architektur sind die Basisblöcke, sowie eine Verdrahtungsstruktur, in welche sie eingebettet sind. Wie in Abschnitt 1.8.2 bereits motiviert, werden wir hinsichtlich des Entwurfs von Basisblöcken eine Integra-

tion von Logik- und Verdrahtungsaufgaben zugrunde legen und – im Gegensatz zu den meisten kommerziellen Architekturen – die gesamte Feldprogrammierbarkeit in diese Basisblöcke verlagern, so daß die sie umgebende Verdrahtungsstruktur über lediglich Punkt-zu-Punkt-Verbindungen definierbar ist.

Damit bleibt also noch die Frage, wie die Funktionalität eines solchen Basisblockes spezifiziert werden kann. Damit durch eine Kombination mehrerer Basisblöcke nun sequentielle Schaltkreise realisiert werden können, sind als notwendige Komponenten eines Basisblockes noch festzulegen:

a) Einheiten zur Berechnung beliebiger boolescher Funktionen

b) Haltebausteine für Signale (D-Latches)

c) Programmierbares Netzwerk für Routingaufgaben

Funktionsgeneratoren

Auf Gatterebene könnte der Frage der Wahl von Berechnungseinheiten einerseits durch die Bereitsstellung einer Menge von (Komplex-)Gattern, die eine (im Postschen Sinne) *vollständige Menge* Boolescher Funktionen realisiert, genügt werden. Diese, jeweils eine feste Boolesche Funktion realisierenden Gatter, würden auf einen oder mehrere Basisblöcke verteilt, so daß die Programmierbarkeit der Gesamtarchitektur alleine von der Flexibilität des Verdrahtungsnetzwerkes abhinge. Bei diesem Konzept hätte jedoch das Ergebnis des Technology-Mapping-Schrittes entscheidenden Einfluß auf die Einbettbarkeit eines Schaltkreises. Denn, geht man realistischerweise von einer nichtvollständigen Vernetzung aller Gatter aus, so determiniert eine lokal stark eingeschränkte Logikflexibilität große Distanzen und damit auch einen hohen Anteil ungenutzter Gatter.

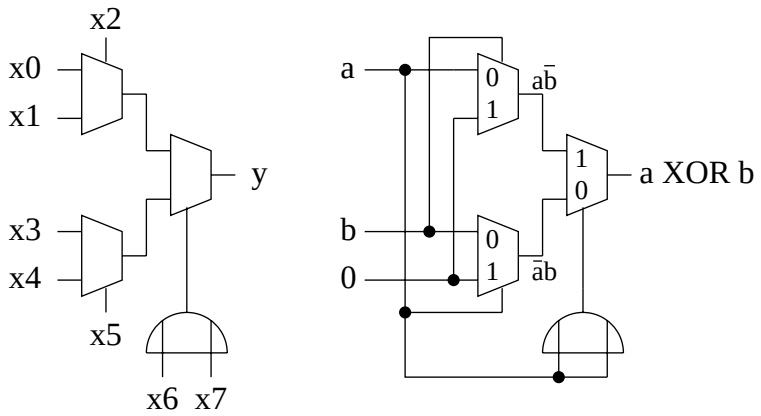


Abbildung 2.1: Basisblock *Actel Act-1* und Beispiel

Eine weitere, hinsichtlich Logikflexibilität aber verbesserte Möglichkeit stellt der Einsatz universell-konfigurierbarer Logikmodule dar, welche durch Kopplungen und Konstantsetzungen ihrer Eingänge in der Lage sind, statt einzelner nun eine bestimmte Menge Boolescher Funktionen zu realisieren. Dieses Konzept wurde beispielsweise in

den FPGAs der Firma Actel Inc. realisiert. Abbildung 2.1 zeigt links den Aufbau eines Basisblocks der Architektur *Actel Act-1* [26] und rechts das Beispiel einer in einem solchen Basisblock implementierten Funktion, die zweistellige Exklusiv-Oder-Verknüpfung. Der Act-1-Basisblock besitzt acht Eingänge, einen Ausgang und ist in der Lage, alle zweistelligen Booleschen Funktionen, die meisten dreistelligen, einige vierstellige und so fort, jedoch insgesamt 554 Funktionen [79], zu realisieren.

In einer früheren Arbeit des Verfassers [79] wurden darüberhinaus noch weit minimalistischere Ansätze vorgestellt – beispielsweise die sogenannte *UF-Zelle* (siehe Abbildung 2.2) in welcher die Universalität des if-then-else-Operators, realisiert durch einen 2:1-Multiplexer, ausgenutzt wurde, um jede zweistellige Boolesche Funktion programmieren zu können. Die UF-Zelle besitzt zwei Eingänge und zwei Ausgänge, kann alle zweistelligen Booleschen Funktionen und alle Verdrahtungskombinationen zwischen Eingängen und Ausgängen realisieren. In der Abbildung stellt jeder Kreis einen programmierbaren Schalter dar, ausgefüllte Kreise im Implementierungsbeispiel deuten hierbei eine Verbindung an.

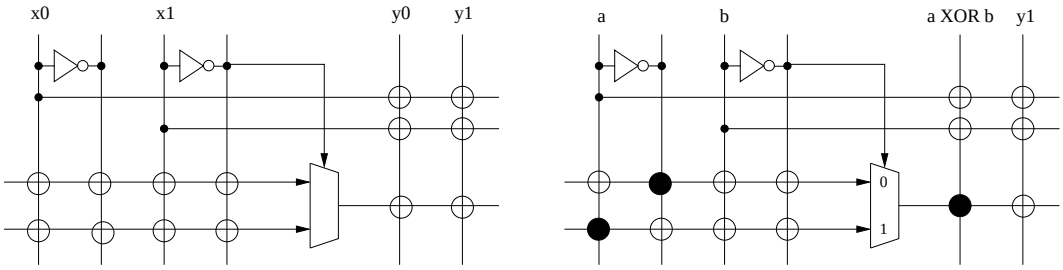


Abbildung 2.2: Basisblock *UF-Zelle* und Beispiel

Im Hinblick auf die geplanten Untersuchungen, soll unser Architekturmodell jedoch von den in einem Basisblock realisierbaren Booleschen Funktionen abstrahieren, da vor allem die strukturellen Eigenschaften konfigurierbarer Architekturen im Zentrum der Betrachtungen stehen werden. Immerhin liefert hier jedoch das Konzept der UF-Zelle bereits eine Basisblock-Architektur, die einen Teil der ursprünglich geforderten Eigenschaften erfüllt: die Realisierbarkeit einer beliebigen (hier: zweistelligen) Booleschen Funktion, gemeinsam mit Verdrahtungsaufgaben. Mit dem Einsatz des Multiplexers enthält die UF-Zelle also die konkrete Realisierung eines Funktionsgenerators, welche noch dazu relativ platzeffizient ist. Für universell-konfigurierbare Logikmodule höherer Ordnung, die Funktionen mit mehr als zwei Eingängen realisieren können, ist dies im allgemeinen nicht der Fall. Andererseits wollen wir bei unseren Untersuchungen auch hinsichtlich des Platzverbrauches von einem eindeutigen, mit der Zahl der Eingänge skalierbaren Kostenmodell ausgehen können. Deshalb modellieren wir die Logikressourcen von Basisblöcken durch Look-Up-Tables (LUTs) mit n Eingängen und einem Ausgang, die wir abstrakt als Generator für n-stellige Boolesche Funktionen interpretieren und wobei alle Eingänge einer Look-Up-Table gleichwertig sind, was die Realisierbarkeit von Funktionen anbetrifft. Während ein formales Modell bereits mit Definition 1.25 eingeführt wurde, wird in Abschnitt 2.5.2 noch ein Kostenmodell für Look-Up-Tables betrachtet werden.

Halteschaltungen

Als Halteschaltungen betrachten wir flankengesteuerte Daten-Latches (D-Latches). Ein D-Latch besitzt einen Daten-Eingang, einen Clock-Eingang und einen Ausgang, wobei der Signalpegel am Daten-Eingang bei steigender Clock-Flanke auch am Ausgang angenommen wird. Zur Vermeidung von *Clock Skews* aufgrund hoher Lastwiderstände realisiert man im allgemeinen ein für alle Haltebausteine gemeinsames Clock-Signal auf einem separaten Clock-Netzwerk von spezieller Topologie. Bei unseren Architekturen gehen wir deshalb von der Existenz eines fest installierten Clock-Netzwerkes aus, so daß wir eine Modellierung desselben unterlassen können. Ein formales Modell für Latches wurde bereits mit Definition 1.28 gegeben.

Verdrahtungsnetzwerk

Auch in Bezug auf das Verdrahtungsnetzwerk skizziert die in Abbildung 2.2 dargestellte UF-Zelle einen ersten Ansatz zur Modellierung. Wie bereits erwähnt, symbolisieren die in der Abbildung dargestellten Kreise jeweils einen programmierbaren Schalter. Solche Schalter werden in der Regel mittels Tristate-Elementen bzw. Pass-Transistoren technisch realisiert. Nicht unerwähnt bleiben soll auch die von den ursprünglichen PLAs herstammende Fusing-Technologie, welche sich in der Praxis bei den FPGAs allerdings nicht durchgesetzt hat.

Eine weitere, nicht weniger mächtige Möglichkeit, eine gegebene Menge von Eingangssignalen kontrolliert auf eine gegebene Menge von Ausgängen zu leiten, stellen sogenannte konfigurierbare Multiplexer-Netzwerke dar [83]. Während in der Praxis bei Verdrahtungsstrukturen grobgranularer Architekturen in der Regel die Tristate- und Pass-Transistor-Variante zur Anwendung kommt, finden sich bei der Realisierung feingranularer Architekturen, sowie beim Design konfigurierbarer Basisblöcke überwiegenderweise Multiplexer-Netzwerke wieder, da Multiplexer eine im Verhältnis zur Zahl ihrer Eingänge minimale Redundanz hinsichtlich der Codierung einer Auswahl besitzen, d.h. eine wesentlich geringere Anzahl von Programmierbits erfordern. Hingegen verursachen Tristate-Realisierungen im allgemeinen eine geringere Signalverzögerung als Multiplexer mit einer Stufenstruktur.

Für unser Architekturmodell wollen wir konfigurierbare Multiplexer-Netzwerke nutzen, um Verdrahtungsaufgaben strukturell exakt beschreiben zu können und insbesondere Verträglichkeiten einer gleichzeitigen Konfiguration mehrerer Verdrahtungsaufgaben effizient bestimmen zu können. Ferner soll das Konzept der konfigurierbaren Multiplexer-Netzwerke ein skalierbares Modell zur allgemeinen Bewertung von Basisblöcken liefern. Wir betrachten sie deshalb anschließend genauer in Abschnitt 2.2.

2.2 Konfigurierbare Multiplexer-Netzwerke

Dieser Abschnitt betrachtet die Verdrahtungsflexibilität und die Optimierung konfigurierbarer Multiplexer-Netzwerke (KMN). Nach einer formalen Modellierung der Netzwerke und der Verdrahtungsaufgaben, werden Eigenschaften hinsichtlich der Flexibilität einer gleichzeitigen Realisierung mehrerer Tasks gezeigt. Schließlich wird die Möglichkeit der Minimierung von KMN nach den Kostenkriterien Platzverbrauch und Signalverzögerung, unter Erhaltung der Verträglichkeit von Taskmengen, untersucht.

2.2.1 Definitionen

Multiplexer-Netzwerke

Die Elementarbausteine eines konfigurierbaren Multiplexer-Netzwerkes sind Auswahlschaltungen (Multiplexer), die aus einer gegebenen Menge von Eingangssignalen ein Signal zu ihrem Ausgang durchschalten:

Definition 2.1 (Multiplexer, MUX)
Ein *Multiplexer* f ist eine boolesche Funktion $f : \mathbb{B}^{n+m} \to \mathbb{B}$ mit Eingangssignalen $I_f = \{x_0 \dots x_{n-1}\}$, Steuersignalen $S_f = \{s_0 \dots s_{m-1}\}$, wobei $m = \lceil \log_2 n \rceil$ und f definiert ist wie folgt[1]:

$$f(x_0 \dots x_{n-1}, s_0 \dots s_{m-1}) = x_{\mathrm{u}(s_0 \dots s_{m-1})}$$

Eine Belegung der Steuersignale in S_f bezeichnen wir als *Konfiguration* des Multiplexers.

Die Abbildung 2.3 zeigt das Schaltsymbol eines Multiplexers.

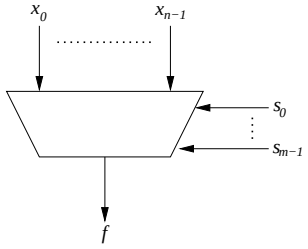


Abbildung 2.3: Multiplexer

Netzwerke von Multiplexern lassen sich nun relativ einfach graphentheoretisch modellieren.

Definition 2.2 (Konfigurierbares Multiplexer-Netzwerk, KMN)
Ein *konfigurierbares Multiplexer-Netzwerk* $\mathcal{N}$ wird beschrieben durch einen gerichteten, azyklischen Graphen $\mathcal{N} = (V, E)$, wobei V eine Menge von Knoten und $E \subset V \times V$ eine Menge von gerichteten Kanten ist. Die Knotenmenge V partitioniert sich in eine Menge von Eingängen $I(\mathcal{N})$, Ausgängen $O(\mathcal{N})$ und Multiplexern $M(\mathcal{N})$, wobei gilt:

$$\begin{aligned}
&\mathit{deg}^-(v) = 0, \mathit{deg}^+(v) > 0, \text{falls } v \in I(\mathcal{N}) \\
&\mathit{deg}^-(v) = 1, \mathit{deg}^+(v) = 0, \text{falls } v \in O(\mathcal{N}) \\
&\mathit{deg}^-(v) = \#I_v, \mathit{deg}^+(v) > 0, \text{falls } v \in M(\mathcal{N})
\end{aligned}$$

Jeder Kante $e \in E$, die in einen Multiplexerknoten $v \in M(\mathcal{N})$ einläuft, sei eindeutig ein Eingang $i_v(e) \in \{0, \dots, \mathit{deg}^-(v) - 1\}$ zugeordnet. Eine *Konfiguration* des Multiplexer-Netzwerks ist eine Abbildung $\chi : M(\mathcal{N}) \to \mathbb{N}$, wobei für alle Multiplexerknoten $v \in M(\mathcal{N})$ gilt: $0 \leq \chi(v) \leq \mathit{deg}^-(v) - 1$ und $\chi(v)$ ist eine Konfiguration für v.

[1] Es bezeichne $u(s_0 \dots s_{m-1})$ die Interpretation der Bitfolge $s_0 \dots s_{m-1}$ als nichtnegative Binärzahl.

Die Abbildung 2.4 illustriert die Modellierung eines konfigurierbaren Multiplexer-Netzwerkes.

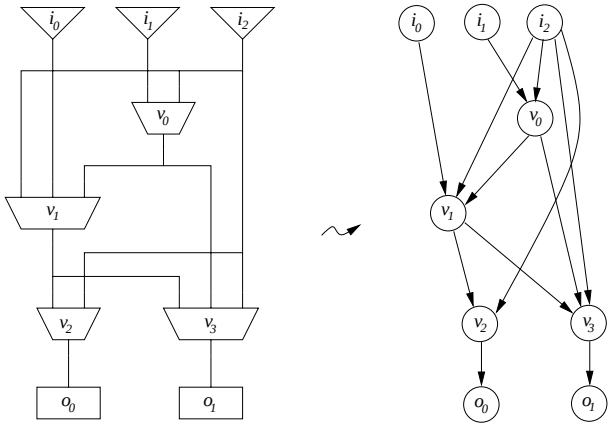


Abbildung 2.4: Konfigurierbares Multiplexer-Netzwerk

Jeder Ausgangsknoten $v \in O(\mathcal{N})$ besitzt also genau einen Vorgängerknoten w.

Routen und R-Tasks

Eine Verdrahtungsaufgabe besteht informal darin, einen gerichteten Weg von einem gegebenen Eingang über eine Kette von Multiplexern zu einem gegebenen Ausgang zu finden. Die Lösung einer Verdrahtungsaufgabe bezeichnen wir als *Route*.

Definition 2.3 (Route)
Eine *Route* R durch ein KMN $\mathcal{N} = (V, E)$ wird beschrieben durch einen gerichteten Weg $R = (v_0, e_0 \ldots v_r, e_r, v_{r+1})$ im Graphen (V, E) mit $v_i \in V$ und $e_i \in E$, wobei gilt: $v_0 \in I(\mathcal{N})$, $v_{r+1} \in O(\mathcal{N})$ und für $i = 1 \ldots r$ ist $v_i \in M(\mathcal{N})$. Die *Länge* von R sei definiert als $l(R) \stackrel{\text{def}}{=} r$. Als *Quelle* der Route bezeichnen wir $Q(R) = v_0$, als *Ziel* der Route $Z(R) = v_{r+1}$. Die Route R heißt auf dem KMN $\mathcal{N}$ *konfigurierbar* genau dann, wenn es eine Konfiguration χ des KMN gibt, so daß gilt:

$$\forall_{i=1\ldots r} \;\; Q(i_v^{-1}(\chi(v_i)) = v_{i-1} \text{ und } v_{r+1} \in \mathit{succ}(v_r)$$

D.h. Vorgängerknoten von v_i über den Eingang $\chi(v_i)$ ist gerade v_{i-1}. Die *Konfiguration* K_R der Route R sei definiert als die Menge aller Konfigurationen des KMN $\mathcal{N}$, vermöge derer R auf $\mathcal{N}$ konfigurierbar ist.

Benutzen zwei Routen in einem KMN ausschließlich verschiedene Multiplexer-Knoten, so sind sie gleichzeitig realisierbar. In diesem Sinne führen wir den Begriff der *Verträglichkeit* von Routen ein.

Definition 2.4 (Verträglichkeit von Routen)
Seien $R_1 = (v_0, e_0 \ldots v_r, e_r, v_{r+1})$ und $R_2 = (v'_0, e'_0 \ldots v'_s, e'_s, v'_{s+1})$ zwei Routen durch ein KMN $\mathcal{N}$. Dann sind R_1 und R_2 *verträglich* genau dann, wenn sie knotendisjunkt sind, d.h. wenn gilt: $\forall_{i \in \{1\ldots r\}} \; \forall_{j \in \{1\ldots s\}} \; v_i \neq v'_j$

Die Konfiguration $K_{R_1R_2}$ zweier verträglicher Routen R_1 und R_2 ergibt sich damit durch den Schnitt ihrer jeweiligen Konfigurationen:

Definition 2.5 (Konfiguration zweier verträglicher Routen)
Sind R_1 und R_2 verträgliche Routen, so ist ihre gemeinsame Konfiguration $K_{R_1R_2}$ definiert wie folgt:

$$K_{R_1R_2} = K_{R_1} \cap K_{R_2}$$

Eine Verdrahtungsaufgabe auf einem KMN, wie sie oben bereits informal definiert wurde, kann natürlich verschiedene Lösungen besitzen. Dementsprechend fassen wir formal einen *Routing-Task* auf als die Menge aller Routen mit identischen Start- und identischen Zielknoten.

Definition 2.6 (Routing-Task, R-Task)
Sei $\mathcal{N}$ ein KMN, $x \in I(\mathcal{N})$ und $y \in O(\mathcal{N})$. Ein *Routing-Task* T_{xy} ist die Menge aller Routen R mit $Q(R) = x$ und $Z(R) = y$. Wir setzen die Operatoren $Q(\cdot)$ und $Z(\cdot)$ auf Routing-Tasks fort durch: $Q(T_{xy}) = x$ und $Z(T_{xy}) = y$. Als *Konfig-Set* $C(T_{xy})$ des Routing-Tasks bezeichnen wir die Vereinigung der Konfigurationen K_R aller Routen $R \in T_{xy}$.

Besitzen nun zwei R-Tasks jeweils verträgliche Routen, so lassen sich die entsprechenden Teil-Konfigurationen, die die Verträglichkeit garantieren, durch den Schnitt der Konfig-Sets der R-Tasks zusammenfassen und somit den Begriff der Verträglichkeit von Routen auf R-Tasks fortsetzen.

Definition 2.7 (Schnitt von Konfig-Sets)
Seien C_1 und C_2 die Konfig-Sets zweier Tasks T_1 und T_2. Dann ist der *Schnitt* der Konfig-Sets definiert als:

$$C_1 \cap C_2 = \{\ K_{R_1R_2} \quad | \quad R_1 \in T_1, R_2 \in T_2 \text{ und } R_1, R_2 \text{ verträglich } \}$$

Wir sprechen jedoch bezüglich R-Tasks (statt von Verträglichkeit) im folgenden nur noch von *Task-Kompatibilität*.

Definition 2.8 (Task-Kompatibilität)
Seien T_1 und T_2 zwei Routing-Tasks mit $Z(T_1) \neq Z(T_2)$. Dann heißen T_1 und T_2 *kompatibel* genau dann, wenn Routen $R_1 \in T_1$ und $R_2 \in T_2$ existieren, die verträglich sind.

Durch die Definitionen 2.7 und 2.8 wird nunmehr eine Verallgemeinerung des Kompatibilitätsbegriffs auf eine Menge von R-Tasks möglich. Wir sprechen dabei von einer *konsistenten* Menge von R-Tasks.

Definition 2.9 (Konsistente Task-Menge)
Eine Menge M von Routing-Tasks heißt *konsistent* genau dann, wenn gilt:

$$\bigcap_{T \in M} C(T) \neq \emptyset$$

Aus einer konsistenten Task-Menge M ist eine Konfiguration für das KMN ermittelbar, vermöge jener alle Tasks in M realisiert werden, nämlich indem ein beliebiges Element aus dem Konfig-Set $\bigcap_{T \in M} C(T)$ gewählt wird.

Eine weitere, recht anschauliche Möglichkeit der Interpretation von Konfig-Sets soll an dieser Stelle noch erwähnt werden. Führt man nämlich für jeden Steuereingang jedes Multiplexers des KMN eine Boolesche Variable ein und faßt man die Bilder der Konfiguration $\chi \in K_R$ einer Route R als Exponenten dieser Variablen auf, so läßt sich jede Route als Boolesches Produkt über den Variablen der Multiplexer darstellen. Das Konfig-Set eines Tasks kann demnach durch eine Boolesche Funktion in disjunktiver Form dargestellt werden, welche genau dann auf Eins auswertet, wenn eine der Routen des Tasks auf dem KMN realisiert wird. Der Schnitt von Konfig-Sets entspricht dann einer Konjunktion der jeweiligen disjunktiven Formen und eine Taskmenge ist genau dann konsistent, wenn die resultierende Boolesche Funktion nicht konstant Null ist.

2.2.2 Eigenschaften

Um die Verdrahtungsflexibilität eines gegebenen KMN zu charakterisieren, betrachten wir Eigenschaften konsistenter Task-Mengen. Wir definieren jedoch zunächst die beiden folgenden Beziehungen zwischen Knoten v eines KMN, Tasks T und Routen $R \in T$: Es ist $v \in R$ genau dann, wenn der Knoten v in der Route R vorkommt, ferner gilt $v \in T$ genau dann, wenn der Knoten v in mindestens einer Route des Tasks T auftritt.

Definition 2.10 (Konfigurierbare/blockierte Tasks)
Sei M eine konsistente Task-Menge und T ein Task. Dann heißt T *konfigurierbar* genau dann, wenn $M \cup \{T\}$ wieder eine konsistente Task-Menge ist, ansonsten heißt T *blockiert*.

Aus der Konsistenz einer Task-Menge folgt also die paarweise Kompatibilität der Tasks, die Umkehrung gilt i.a. jedoch nicht, wie Lemma 2.1 zeigt.

Lemma 2.1
Sei $M = \{T_1 \dots T_n\}$ eine Menge von Tasks, wobei $n > 2$. Unter paarweiser Kompatibilität aller Tasks ist M nicht notwendigerweise konsistent.

Beweis: Gegenbeispiel: Betrachte den KMN in Abbildung 2.5. Die Tasks T_{ux}, T_{vy} und T_{wz} sind paarweise kompatibel, allerdings benutzt T_{vy} stets entweder einen Knoten der Routen von T_{wz} oder einen Knoten der Routen von T_{vy}. □

Eine triviale Eigenschaft zur Kompatibilität zweier Tasks zeigt Lemma 2.2.

Lemma 2.2
Für beliebige Eingänge u, v und für einen beliebigen Ausgang y eines KMN gilt: T_{uy} ist nicht kompatibel zu T_{vy}, falls die beiden Tasks existieren.

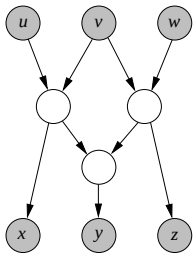


Abbildung 2.5: Gegenbeispiel zu Lemma 2.1

Beweis: Jeder Ausgangsknoten y eines KMN besitzt genau einen Vorgängerknoten, der von T_{uy} und T_{vy} gemeinsam genutzt würde. Somit sind die beiden Tasks nicht kompatibel. □

Faßt man den Begriff der Task-Kompatibilität als Relation auf, ergeben sich folgende Eigenschaften.

Lemma 2.3 (Eigenschaften der Task-Kompatibilität)
Die Relation der Task-Kompatibilität ist:

a) irreflexiv
b) symmetrisch
c) nicht-transitiv

Beweis:

a) Ein Task T ist nicht kompatibel zu sich selbst, da alle seine Routen zumindest den Vorgängerknoten von $Z(T)$ gemeinsam haben und somit nicht verträglich mit sich selbst sind (Lemma 2.2).
b) Trivial.
c) Gegenbeispiel (siehe Abbildung 2.6): T_{uy} und T_{wz} sind kompatibel, wie auch T_{wz} und T_{vy}, allerdings sind T_{uy} und T_{vy} nicht kompatibel, da ihre Routen einen gemeinsamen Knoten nutzen.

□

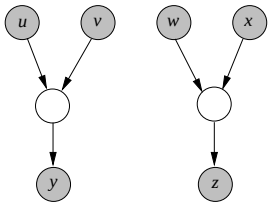


Abbildung 2.6: Gegenbeispiel zu Lemma 2.3c

Aus der Existenz von Tasks kann auch die Existenz weiterer Tasks gefolgert werden. Satz 2.1 gibt einen entsprechenden, zwar leicht einzusehenden, jedoch wichtigen Zusammenhang wieder.

Satz 2.1 (Implikative Task-Existenz)
Existieren in einem KMN zwei nicht-kompatible Tasks T_1 und T_2, so existieren auch Tasks T_1' und T_2' mit:

$$Q(T_1') = Q(T_1), Z(T_1') = Z(T_2) \text{ und } Q(T_2') = Q(T_2), Z(T_2') = Z(T_1)$$

Beweis: Aus der Nicht-Kompatibilität folgt, daß zu jedem Paar von Routen aus T_1 und T_2 mindestens ein Multiplexerknoten v existiert, den die Routen gemeinsam nutzen. Also sind sowohl $Z(T_1)$, als auch $Z(T_2)$ von v aus erreichbar und T_1' bzw. T_2' existieren vermöge der Ersetzung der Konfigurationen aller Nachfolgerknoten von v bzgl. T_2 bzw. T_1. □

Aus dem vorangegangenen Satz 2.1 ergibt sich auch der folgende:

Satz 2.2
Sei $\mathcal{N}$ ein KMN mit den Eingängen u und v, sowie den Ausgängen x und y. Es existieren die nicht-kompatiblen Tasks T_{ux} und T_{vy}. Sei M eine konsistente Task-Menge und auch die Mengen $M \cup \{T_{ux}\}$, sowie $M \cup \{T_{vy}\}$ seien konsistent. Dann existiert ein Task T_{uy} und $M \cup \{T_{uy}\}$ ist ebenfalls eine konsistente Menge.

Beweis: Die Existenz von Task T_{uy} folgt sofort aus Satz 2.1, ebenso wie die gesamte Behauptung des Satzes für den Fall $M = \emptyset$.

Sei daher $M \neq \emptyset$. Wir betrachten die beiden folgenden Mengen:

$$V_{ux} = \{w \in M(\mathcal{N}) \mid \exists_{R \in T_{ux}} K_R \cap C(M \cup \{T_{ux}\}) \neq \emptyset \ \wedge\ w \in R\}$$

$$V_{vy} = \{w \in M(\mathcal{N}) \mid \exists_{R \in T_{vy}} K_R \cap C(M \cup \{T_{vy}\}) \neq \emptyset \ \wedge\ w \in R\}$$

Die beiden Mengen enthalten also gerade die Knoten aller Routen, welche die Tasks T_{ux} bzw. T_{vy} jeweils gemeinsam mit den Tasks aus der Menge M realisieren. Nach Voraussetzung, $M \cup \{T_{ux}\}$ und $M \cup \{T_{vy}\}$ konsistent, sind die Mengen V_{ux} und V_{vy} nicht leer.

Annahme:

$$\begin{aligned} & V_{ux} \cap V_{vy} = \emptyset \\ \iff \quad & \forall_{R_1 \in T_{ux}} \forall_{R_2 \in T_{vy}} \\ & \text{mit } K_{R_1} \cap C(M \cup \{T_{ux}\}) \neq \emptyset \text{ und } K_{R_2} \cap C(M \cup \{T_{vy}\}) \neq \emptyset \\ & \text{gilt } R_1 \cap R_2 = \emptyset \end{aligned}$$

Dies würde jedoch bedeuten, es gäbe disjunkte Routen für T_{ux} und T_{vy}, was einen Widerspruch zur Voraussetzung der Inkompatibilität der beiden Tasks darstellt. Somit muß gelten: $V_{ux} \cap V_{vy} \neq \emptyset$ woraus folgt, daß $M \cup \{T_{ux}\}$ konsistent ist. Denn für jeden Knoten $w \in V_{ux} \cap V_{vy}$ existiert dann eine Konfiguration auf $\mathcal{N}$, die eine gleichzeitig mit den Tasks aus M realisierbare Route von x über w nach y liefert. □

Obwohl Satz 2.2 auf den ersten Blick relativ unscheinbar wirkt, birgt er einen wichtigen Nachweis für die Anwendbarkeit klassischer Algorithmen zur Suche nach kürzesten Pfaden in einem System von KMN. Denn wird im Zuge der Kürzeste-Wege-Suche ein KMN mehrmals durchlaufen, ist eine Kompatibilität der später gleichzeitig zu realisierenden Tasks zum Zeitpunkt der Suche nicht effizient berechenbar, sondern erst bei der Rekonstruktion des kürzesten Pfades. Abbildung 2.7 illustriert diesen Zusam-

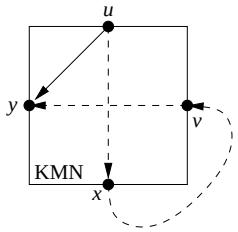


Abbildung 2.7: Beispiel zu Satz 2.2

menhang: beinhaltet ein kürzester Pfad beispielsweise eine Verbindung von u nach y, so wird ein entsprechender Algorithmus den direkten, da kostengünstigeren Task T_{uy} wählen, dessen Existenz in jedem Falle sichergestellt wäre, wenn die alternativen Tasks T_{ux} und T_{vy} inkompatibel wären. Man beachte, daß damit jedoch auch eine Voraussetzung an das Kostenmodell des KMN bzw. dessen Umgebung (Architektur), in die es eingebettet ist, gegeben ist:

$$d(T_{uy}) < d(T_{ux}) + d(T_{ext}) + d(T_{vy})$$

Auf eine besondere Eigenschaft hinsichtlich der Kompatibilität von Tasks soll mit dem abschließenden Lemma noch hingewiesen werden.

Lemma 2.4 (Indirekte Task-Inkompatibilität)
Sei $\mathcal{N}$ ein KMN mit den Eingängen u und v, sowie den Ausgängen x und y. Es existieren die Tasks T_{ux} und T_{vy}. Sei M eine konsistente Task-Menge und auch die Mengen $M \cup \{T_{ux}\}$, sowie $M \cup \{T_{vy}\}$ seien konsistent. Dann ist die Menge $M \cup \{T_{ux}, T_{vy}\}$ nicht notwendigerweise konsistent.

Beweis: Falls die Tasks T_{ux} und T_{vy} nicht kompatibel sind, folgt die Behauptung sofort. Doch auch wenn T_{ux} und T_{vy} kompatibel sind, folgt mit dem Beweis zu Lemma 2.1, daß T_{ux} „indirekt inkompatibel“ zu T_{vy} sein kann. □

Die Konsistenzeigenschaft von Task-Mengen läßt sich leicht mittels dynamischer Programmierung überprüfen, wie wir später noch sehen werden (Algorithmus 2.4). Allerdings hat man es im worst case natürlich mit einer exponentiellen Anzahl von Teilproblemen zu tun.

2.2.3 Optimierung von KMN

Unter der Optimierung eines gegebenen KMNs verstehen wir eine Umformung, die dessen konsistente Task-Mengen unverändert läßt. Nach der formalen Definition eines aus geeigneten Umformungen resultierenden KMNs, geben wir bzgl. dieser Eigenschaft invariante Transformationen an.

Transformationen

Definition 2.11 (Routing-verwandtes KMN)
Sei $\mathcal{N}$ ein KMN und sei $\mathcal{T}(\mathcal{N})$ die Menge der R-Tasks auf $\mathcal{N}$. Ein KMN $\mathcal{N}'$ heißt *routing-verwandt* genau dann, wenn die folgenden Eigenschaften gelten:

a) $I(\mathcal{N}) = I(\mathcal{N}')$ und $O(\mathcal{N}) = O(\mathcal{N}')$

b) $T \in \mathcal{T}(\mathcal{N}) \quad \Longrightarrow \quad \exists_{T' \in \mathcal{T}(\mathcal{N}')}\, Q(T') = Q(T) \ \wedge\ Z(T') = Z(T)$

c) Sei $M = \{T_1 \ldots T_k\} \subseteq \mathcal{T}(\mathcal{N})$ eine beliebige Menge von R-Tasks, die auf $\mathcal{N}$ konsistent sind. Dann existiert zu jedem R-Task $T_i \in M$ ein R-Task T_i' auf $\mathcal{N}'$ mit gleicher Quelle und gleichem Ziel, und die Menge $M' = \{T_1' \ldots T_k'\}$ ist auf $\mathcal{N}'$ ebenfalls konsistent.

Aus Lemma 2.3 folgt insbesondere, daß Routing-Verwandtschaft keine Äquivalenzrelation sein kann. Denn bereits mit der fehlenden Eigenschaft der Transitivität hinsichtlich der Kompatibilität von Tasks wird offensichtlich, daß bei routing-verwandten KMN-Paaren ($\mathcal{N}_1$,$\mathcal{N}_2$) und ($\mathcal{N}_2$,$\mathcal{N}_3$) durchaus konsistente Task-Mengen in $\mathcal{N}_1$ existieren können, die in $\mathcal{N}_3$ nicht konsistent sind.

Andererseits stellt Routing-Verwandtschaft jedoch auch keine partielle Ordnung dar, da gegenseitig routing-verwandte KMN nicht notwendigerweise identisch sein müssen, wie das Beispiel in Abbildung 2.8 zeigt.

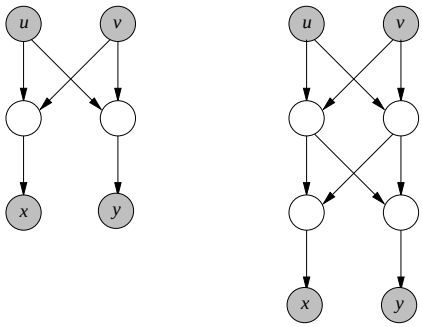


Abbildung 2.8: Gegenseitig routing-verwandte KMN

Mit dem folgenden Satz 2.3 geben wir nun Transformationen auf KMN an, die die Menge der konsistenten Task-Mengen nicht verringern.

Satz 2.3 (Verwandtschafts-Transformationen)
Ist $\mathcal{N} = (V, E)$ ein KMN, so führen die folgenden Transformationen stets zu einem routing-verwandten KMN:

a) **Knoten-Splitting**
Sei $u \in V$ ein Multiplexerknoten und sei $P_1 \ldots P_n$ mit $P_i \subseteq E_u^+$ eine Partitionierung der in u einlaufenden Kanten. Dann kann u ersetzt werden durch einen Knoten v und Vorgängerknoten $u_1 \ldots u_n$, wobei u_i neuer Zielknoten der Kanten in P_i ist.

b) **Knoten-Verschmelzung**
Seien $u, v \in V$ zwei Multiplexerknoten, wobei v einziger Nachfolger von u sei. Dann kann v samt seinen adjazenten Kanten eliminiert werden, wenn Kanten von allen Knoten $w \in \mathit{pred}(v) \setminus \{u\}$ nach u gezogen werden und Kanten von u zu allen Knoten $z \in \mathit{succ}(v)$.

c) **Triviale Knoten**
Besitzt ein Multiplexerknoten v lediglich eine einlaufende Kante e, kann er samt seinen adjazenten Kanten entfernt werden, indem neue Kanten von $Q(e)$ zu allen Nachfolgern von v gezogen werden. Ein Multiplexerknoten ohne ausgehende Kante kann mitsamt seinen einlaufenden Kanten entfernt werden.

d) **Bypass**
Seien $u, v \in V$ zwei Multiplexerknoten. Existiert eine Route von u nach v, so kann eine Kante (u, v) hinzugefügt werden.

e) **Redundante Knoten**
Sei $u \in V$ ein Multiplexerknoten. Existieren für alle $v \in \mathit{pred}(u)$ Kanten zu allen Nachfolgern $w \in \mathit{succ}(u)$, so kann u samt seinen adjazenten Kanten entfernt werden.

f) **Redundante Kanten**
Seien $u, v \in V$ zwei Knoten in einem KMN und es existieren zwei knotendisjunkte Pfade von u nach v, wobei alle durchlaufenen Knoten der beiden Pfade stets einen Fanout von 1 besitzen. Sei e die erste Kante eines der beiden Pfade. Dann kann e entfernt werden.

Beweis:

a) Das Einfügen der Zwischenknoten u_i schränkt die Verträglichkeit der bisher existierenden Routen nicht ein.

b) Da u nur eine Ausgangskante besitzt, können dessen Eingangssignale ausschließlich nach v geroutet werden. Direktes Einleiten der Signale in v ändert somit die Verträglichkeit der Routen nicht.

c) Trivial.

d) Das Hinzufügen der Kante schränkt die Verträglichkeit aller bisherigen Routen nicht ein, sondern erweitert sie lediglich.

e) Mit den Kanten von Knoten aus $\mathit{pred}(u)$ nach Knoten aus $\mathit{succ}(u)$ existieren für alle Signale verträgliche Routen, die u nicht benutzen. Somit kann u entfernt werden.

f) Betrachte eine beliebige Konfiguration des KMN. Ist ein Pfad von u nach v geschaltet, so kann der alternative Pfad, da alle seine Knoten einen Fanout von 1 besitzen, nicht von einer anderen Route genutzt werden. Somit ist die Streichung von e unkritisch. Ist andererseits kein Pfad von u nach v geschaltet, kann e auf jeden Fall gestrichen werden.

□

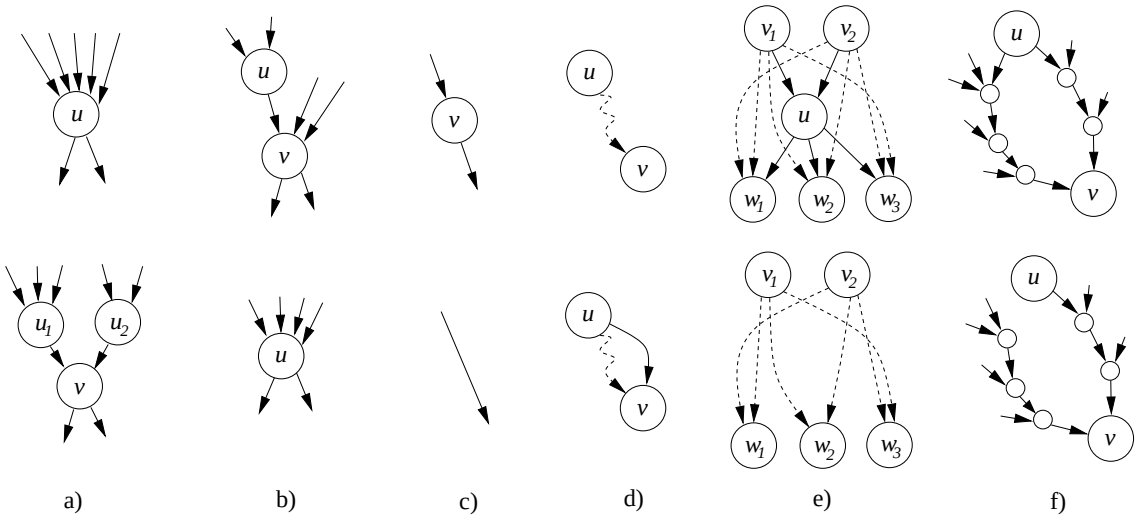


Abbildung 2.9: Verwandschafts-Transformationen

In Abbildung 2.9 sind alle Transformationen nochmals graphisch skizziert. Die obere Graphik zeigt jeweils den Zustand vor, die untere Graphik nach der Transformation in den einzelnen Fällen.

Es sei an dieser Stelle nochmals darauf hingwiesen, daß Verwandtschafts-Transformationen nur „unidirektional“ invariante Transformationen hinsichtlich der gleichzeitigen Realisierbarkeit von Routing-Tasks auf einem KMN darstellen.

Definition 2.12 (Triviales KMN)
Sei $\mathcal{N} = (V, E)$ ein KMN mit n Eingängen und m Ausgängen. Dann wird das zu $\mathcal{N}$ gehörende *triviale KMN* $\mathcal{N}' = (V', E')$ wie folgt konstruiert:

- $V' = I(\mathcal{N}) \cup O(\mathcal{N}) \cup \{v_1 \dots v_m\}$, wobei v_i Multiplexerknoten sind.
- Jeder Knoten v_i ist Vorgänger genau eines Ausgangsknotens in V'.
- Für jedes Paar $(x, y_i) \in I(\mathcal{N}) \times O(\mathcal{N})$ gilt: falls in $\mathcal{N}$ eine Route von Eingang x zum Ausgang y_i existiert, ziehe eine Kante von x zum zu y_i gehörigen Multiplexerknoten v_i.

Die Abbildung 2.10 zeigt ein KMN und rechts daneben sein zugöriges triviales KMN.

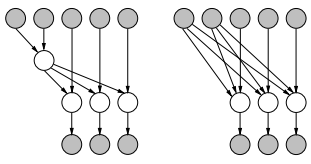


Abbildung 2.10: KMN und zugehöriges triviales KMN

Lemma 2.5
Sei $\mathcal{N}$ ein KMN und $\mathcal{N}'$ sein zugehöriges triviales KMN. Dann ist $\mathcal{N}'$ routingverwandt zu $\mathcal{N}$.

Beweis: Offensichtlich läßt sich das triviale KMN $\mathcal{N}'$ durch Anwendung von Transformationen aus Satz 2.3 konstruieren. Somit ist es routing-verwandt zu $\mathcal{N}$. □

Delayminimierung

Das Problem der Minimierung des Delays von KMN stellt sich wie folgt dar: Sei $d : M(\mathcal{N}) \to \mathbb{R}$ eine Kostenfunktion für das Delay von Multiplexerknoten des KMN $\mathcal{N}$. Finde ein delayminimales, routing-verwandtes KMN $\mathcal{N}'$, d.h. es gilt:

$$\max_{R \text{ Route in } \mathcal{N}'} \sum_{v \in R \cap M(\mathcal{N}')} d(v) \quad \leq \quad \max_{R \text{ Route in } \mathcal{N}} \sum_{v \in R \cap M(\mathcal{N})} d(v)$$

Die Delayfunktion d bestimmt sich durch die Tiefe der Realisierung der Multiplexerbausteine. Unter der Annahme von delayminimal konstruierten Multiplexerbausteinen, d.h. logarithmischer Tiefe bzgl. der Zahl der Eingangssignale, erhält man mit dem zugehörigen trivialen KMN also auch ein delayminimales, routing-verwandtes KMN.

Platzminimierung

Im Gegensatz zur Delayminimierung existiert für das Problem der Platzminimierung im allgemeinen keine triviale Optimallösung. Sei $c : M(\mathcal{N}) \to \mathbb{R}$ eine Kostenfunktion für den Platzverbrauch von Multiplexerknoten des KMN $\mathcal{N}$. Finde ein kostenminimales, routing-verwandtes KMN $\mathcal{N}'$, d.h. es gilt:

$$\sum_{v \in M(\mathcal{N}')} c(v) \quad \leq \quad \sum_{v \in M(\mathcal{N})} c(v)$$

Hängt die Kostenfunktion c im wesentlichen von der Zahl der Eingangssignale eines Multiplexers ab, so ist das Problem äquivalent zur Minimierung der Ingrad-Summe unter Routing-Verwandschafts erhaltenden Transformationen.

Bei fanoutfreien KMN läßt sich die Ingrad-Summe durch Konstruktion des zugehörigen trivialen KMNs minimieren. Die Ingrad-Summe ist hierbei minimal, da sie sich mit jeder Kaskadierung von Multiplexerknoten über die Zahl der dadurch entstehenden Zwischensignale nur erhöhen würde.

Bei KMN mit Fanoutgrad größer als eins gilt dies hingegen nicht, wie das Beispiel in Abbildung 2.10 zeigt: Das gegebene, linke KMN routet fünf Signale nach drei Ausgängen. Sein zugehöriges triviales KMN (rechts) besitzt jedoch offensichtlich eine höhere Ingrad-Summe.

Algorithmus 2.1 stellt ein einfaches Greedy-Verfahren dar, um die Ingrad-Summe eines gegebenen KMN zu reduzieren. Während die Knotenverschmelzung, sowie die Detektion trivialer und redundanter Knoten jeweils mittels einer Durchmusterung aller Knoten des KMNs durchführbar sind, wäre das KMN bei der Ermittlung redundanter Kanten auf knotendisjunkte Pfade zu untersuchen. Dies ist ein bekanntermaßen NP-schweres Problem [30]. Aufgrund zweier gegebener Einschränkungen läßt sich jedoch

auch dieses Problem in Polynomialzeit lösen: zunächst werden nur disjunkte Pfade gesucht, deren Knoten Fanout 1 besitzen, ferner können solche Pfade aufgrund der zuvor durchgeführten Knotenverschmelzung nur noch ab den Eingangsknoten des KMNs auftreten.

Algorithmus 2.1 (Reduktion der Ingrad-Summe)

```
redindeg( )
{
    do
    {
        Knoten-Verschmelzung (Satz 2.3b)
            // Eliminiere durch Knotenverschmelzung alle Multiplexer-
             knoten, die einziger Nachfolger eines Multiplexerknotens sind.
        Eliminierung von Redundanzen (Satz 2.3e, f)
            // Eliminiere redundante Knoten und Kanten.
        Eliminierung trivialer Knoten (Satz 2.3c)
            // Eliminiere triviale Multiplexerknoten.
        Eliminierung faninfreier Multiplexerknoten
        samt ihren ausgehenden Kanten
    }
    while (Transformationen durchgeführt);
}
```

Wie am bereits erwähnten Beispiel (Abbildung 2.10) zu sehen ist, handelt es sich beim gefundenen Minimum also um ein *lokales* Minimum, denn auf triviale KMNs sind keinerlei die Routing-verwandtschaft erhaltenden Transformationen anwendbar.

Eine untere Schranke für die Berechnungskomplexität des genannten Minimierungsproblems erhält man bereits mit der Komplexität der Überprüfung auf die Existenz verträglicher Routen. Dies ist ein disjunkte-Wege-Problem, welches selbst bei direkt konstruktiven Verfahren der Ausgangspunkt wäre.

2.3 Feldprogrammierbare Architekturen

2.3.1 KMN-Einbettungen und Tasks

Die Grundlage unseres Modells für feldprogrammierbare Architekturen bildet nun die Einbettung von Look-Up-Tables und Latches in konfigurierbare Multiplexer-Netzwerke. Dies liefert schließlich ein Modell, das eine Spezifikation beliebiger Basisblöcke ermöglicht, mittels welchen eine feldprogrammierbare Gesamtarchitektur generisch konstruierbar ist. Ferner können wir aus den skalierbaren Komponenten ein Kostenmodell für Basisblöcke ableiten, welches anhand einer fixierten Technologie konkrete Kostenmaße für Platzverbrauch und die Signalverzögerung liefern kann, die auf die Gesamtarchitektur fortsetzbar sind.

Definition 2.13 (KMN-Einbettung)
Eine *KMN-Einbettung* wird repräsentiert durch $\mathcal{E} = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{F}, \mathcal{L}, \eta_1, \eta_2)$, wobei $\mathcal{N}_1$ und $\mathcal{N}_2$ zwei KMN sind, $\mathcal{F}$ eine Menge von Look-Up-Tables und $\mathcal{L}$ eine Menge von Latches. Sei $\mathcal{I}$ die Menge der Eingänge und $\mathcal{O}$ die Menge der Ausgänge aller Look-Up-Tables und Latches in $\mathcal{F} \cup \mathcal{L}$. Dann definieren die Abbildungen $\eta_1 : O(\mathcal{N}_1) \to 2^{\mathcal{I} \cup I(\mathcal{N}_2)}$ und $\eta_2 : \mathcal{O} \to 2^{I(\mathcal{N}_2)}$ Identifikationen von Ausgängen mit Mengen von Eingängen, wobei für $o_1, o_2 \in O(\mathcal{N}_1)$ mit $o_1 \neq o_2$ gilt: $\eta_1(o_1) \cap \eta_1(o_2) = \emptyset$. Gleiches gilt für η_2.

Die Abbildung 2.11 skizziert Definition 2.13 nochmals graphisch.

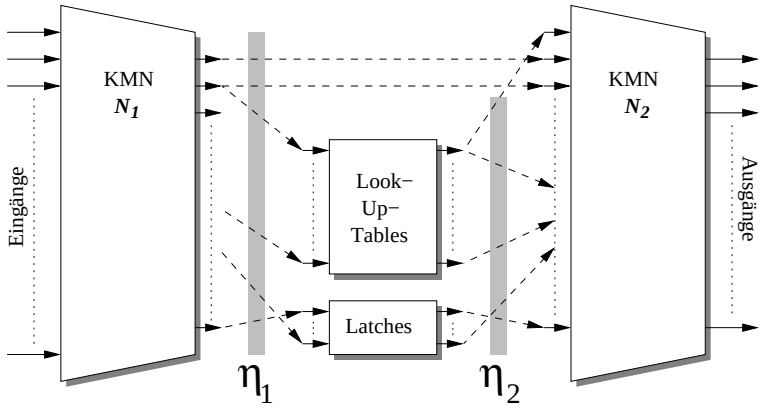


Abbildung 2.11: KMN-Einbettung

Die Vereinigung der Knotenmengen von $\mathcal{N}_1$, $\mathcal{N}_2$, zuzüglich der als Knotenmengen interpretierten Mengen $\mathcal{F}$ und $\mathcal{L}$, sowie die Vereinigung der Kantenmengen von $\mathcal{N}_1$ und $\mathcal{N}_2$, zuzüglich der durch die Abbildungen η_1 und η_2 implizierten Kanten, liefert offensichtlich einen gerichteten azyklischen Graphen $\mathcal{G}$. Wir bezeichnen $\mathcal{G}$ als *Graphen der KMN-Einbettung*.

Definition 2.14 (Task)
Sei $\mathcal{E} = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{F}, \mathcal{L}, \eta_1, \eta_2)$ eine KMN-Einbettung. Dann ist ein *Task* auf $\mathcal{E}$ definiert wie folgt:

- *Routing-Task (R-Task)* T_{xy} für einen Eingang $x \in I(\mathcal{N}_1)$ und einen Ausgang $y \in O(\mathcal{N}_2)$:

$$T_{xy} = \Big\{ (T_1, T_2) \;\Big|\; T_i \text{ R-Task in } \mathcal{N}_i \text{ für } i \in \{1,2\} \text{ und } Q(T_1) = x, Z(T_2) = y, Q(T_2) \in \eta_1(Z(T_1)) \Big\}$$

- *Logik-Task (L-Task)* T_F für eine Look-Up-Table $F \in \mathcal{F}$:

$$T_F = \Big\{ (\mathcal{T}_1, T_2) \;\Big|\; \mathcal{T}_1 \text{ konsistente Menge von R-Tasks auf } \mathcal{N}_1 \text{ mit } I_F \subseteq \bigcup_{T \in \mathcal{T}_1} \eta_1(Z(T)) \text{ und } Q(T_2) \in \eta_2(o_F) \Big\}$$

- *Speicher-Task (S-Task)* T_L für ein Latch $L \in \mathcal{L}$:

$$T_L = \Big\{ (T_1, T_2) \quad \Big| \quad T_i \text{ R-Task in } \mathcal{N}_i \text{ für } i \in \{1,2\} \text{ und } d_L \in \eta_1(Z(T_1)), Q(T_2) \in \eta_2(q_L) \Big\}$$

Bei jedem Task einer KMN-Einbettung, ob Routing-, Logik- oder Speichertask, sind also R-Tasks der KMN $\mathcal{N}_1$ und $\mathcal{N}_2$ beteiligt, wobei im Falle von Logiktasks speziell eine Menge *konsistenter* R-Tasks gefordert wird, um die Eingangssignale der zu berechnenden Funktion multiplexerdisjunkt zu den Eingängen der Look-Up-Table zu leiten. Sind hingegen Funktionen in einer geringeren Zahl von Eingängen, als die Look-Up-Table besitzt, zu realisieren, so müssen dennoch Routen von Eingängen der Zelle zu allen Eingängen der Look-Up-Table konfiguriert werden, wie dies im übrigen auch bei einer technischen Realisierung der Fall wäre.

Wir können nun die Begriffe der Kompatibilität von Tasks (Definition 2.8) und der Konsistenz von Taskmengen (Definition 2.9) von KMN auf KMN-Einbettungen fortsetzen, indem wir die *Konfiguration* χ einer KMN-Einbettung einfach als Konkatenation der Konfigurationen der KMN $\mathcal{N}_1$ und $\mathcal{N}_2$ interpretieren

$$\chi \stackrel{\text{def}}{=} (\chi_{\mathcal{N}_1}, \chi_{\mathcal{N}_2})$$

und auch Schnitte über Konfig-Sets nun komponentenweise definieren. In obiger Mengendefinition für Tasks wurde ferner berücksichtigt, daß ein Task mehrere Konfigurationen auf einer KMN-Einbettung besitzen kann, durch welche er realisiert wird. Wir werden in Kürze hierzu ein Beispiel sehen.

Bevor wir die Mächtigkeit des Modells der KMN-Einbettung genauer betrachten, sollen noch einige Bezeichnungen eingeführt werden, welche später häufiger benötigt werden. Sei T Task auf einer KMN-Einbettung $\mathcal{E}$. Es bezeichne o_T den durch Konfigurieren von T angesprochenen Ausgang auf $\mathcal{E}$. Falls T ein Logik-Task ist, bezeichne I_T die Menge seiner Eingänge, im Falle von Verdrahtungs- und Latch-Tasks bezeichne i_T den (eindeutigen) Eingang des Tasks. Die Kompatibilität zweier Tasks T_1 und T_2 drücken wir aus durch $T_1 \sim T_2$.

Ermittlung der Taskmenge

Zu einer gegebenen KMN-Einbettung $\mathcal{E} = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{F}, \mathcal{L}, \eta_1, \eta_2)$ kann die Menge aller Tasks, sowie deren zugehörige Konfigurationsmengen mittels einer Rückwärtstraverse auf dem DAG $\mathcal{G} = (V, E)$ der KMN-Einbettung berechnet werden. Im folgenden geben wir einen entsprechenden Algorithmus an.

Für jeden Knoten $v \in V$ halten wir dabei eine (zu Beginn noch leere) Menge L_v von Paaren (K, T), wobei K eine Konfiguration der KMN-Einbettung und T einen Tasktypen (Routing-, Logik- oder Speichertask) bezeichne. Für einen Multiplexer-Knoten $v \in M(\mathcal{N}_1) \cup M(\mathcal{N}_2)$ des KMNs $\mathcal{N}_1$ oder $\mathcal{N}_2$ bezeichne K_v gerade die im Zuge der Rückwärtstraverse auf $\mathcal{G}$ bis zum Knoten v ermittelte Menge von (Teil-) Konfigurationen.

Algorithmus 2.2 (Berechnung der Taskmenge einer KMN-Einbettung)

Initialisierung:
    Liste Q := $\emptyset$;
    Für alle Ausgangsknoten $v \in O(\mathcal{N}_2)$:
        Initialisiere L_v mit dem Paar $((x \dots x), \text{ROUTE})$;
        Q.append(v);
while(Q $\neq \emptyset$)
{
    v := Q.pop();
    // Propagiere Tasksets erst, wenn Unterbaum abgearbeitet:
    Falls $w \in$ *succ*(v) existiert und w nicht markiert:
        Q.append(v);
        continue;
    sonst:
        markiere v;
    Falls $v \in I(\mathcal{N}_1)$: *// Eingangsknoten*
        continue;
    Falls $v \in O(\mathcal{N}_2)$: *// Ausgangsknoten*
        Für alle $w \in$ *pred*(v):
            $L_w := L_w \cup L_v$;
            Falls $w \notin Q$: Q.append(w);
        $L_v := \emptyset$;
    Falls $v \in \mathcal{L}$: *// Latchknoten*
        Für alle $(K,T) \in L_v$: setze $T :=$ LATCH;
        Sei $w =$ *pred*(v);
        $L_w := L_w \cup L_v$;
        Falls $w \notin Q$: Q.append(w);
        $L_v := \emptyset$;
    Falls $v \in \mathcal{F}$: *// LUT-Knoten*
        Für alle $(K,T) \in L_v$: setze $T :=$ LUT;
        Für alle $w \in$ *pred*(v):
            $L_w := L_w \cup L_v$;
            Falls $w \notin Q$: Q.append(w);
        $L_v := \emptyset$;
    Sonst: *// Multiplexer-Knoten*
        Sei $\{w_0 \dots w_{k-1}\} :=$ *pred*(v);
        Für $i = 0 \dots k-1$:
            Für alle $(K,T) \in L_v$:
                Setze $K_i := K|_{u(K_v)=i}$;
                $L_{w_i} := L_{w_i} \cup (K_i, T)$;
            Falls $w_i \notin Q$: Q.append(w_i);
        $L_v := \emptyset$;
}

Nach der Terminierung von Algorithmus 2.2 enthält die Vereinigung der Mengen L_v aller Eingangsknoten v der KMN-Einbettung die Konfigurationen aller realisierbaren Tasks, wobei durch x bezeichnete Positionen in den Konfigurationen „Don't Care"-Belegungen darstellen. Die erhaltene Taskliste kann ferner hinsichtlich äquivalenter Tasks noch strukturiert werden. Ein Beispiel hierzu ist in Abbildung 2.12 dargestellt. Der Algorithmus findet unter anderem die Paare (abcd,T) = `(1x0x,ROUTE)` und `(x01x,ROUTE)`, welche jedoch zu einem gemeinsamen Routing-Task $x_1 \rightarrow y_0$ gehören.

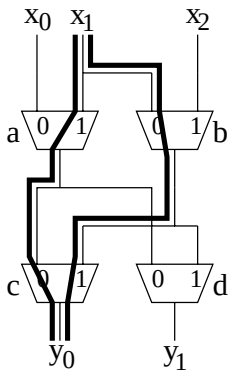


Abbildung 2.12: Beispiel äquivalenter Tasks

Auf den ersten Blick erscheint die Berechnung mit Algorithmus 2.2 wohl relativ effizient, denn jeder Knoten $u \in V$ des Graphen wird höchstens $deg^+(u) \leq \#E$ mal in die Liste Q eingefügt und somit wäre die Zahl der Schleifendurchläufe durch $\#V \cdot \#E$ nach oben begrenzt. Allerdings kann die Zahl der Paare (K,T) in einem Knoten im worstcase der Anzahl von Wegen bis zu einer Senke des Graphen entsprechen. Dies wäre ein exponentieller Faktor in der Länge des längsten Weges. Andererseits können wir mit den Beobachtungen aus Abschnitt 2.2.3 wiederum folgern, daß Routing-Netzwerke sehr hoher Flexibilität nur eine geringe Tiefe besitzen. Im Extremfall maximaler Flexibilität, das heißt wenn man für $\mathcal{N}_1$ und $\mathcal{N}_2$ ihr zugehöriges triviales KMN wählt, beträgt die maximale Tiefe des Routing-Netzwerkes sogar nur zwei.

Wir abstrahieren jedoch im folgenden über Konfigurationen, bewegen uns künftig also vielmehr auf der Begriffsebene kompatibler Tasks bzw. konsistenter Taskmengen und wenden uns nun endlich der formalen Definition feldprogrammierbarer Architekturen zu.

2.3.2 Konfigurierbare Zellen

Die kleinsten Einheiten einer feldprogrammierbaren Architektur bilden die Basisblöcke. Wie bereits motiviert wurde, soll diese Klasse konfigurierbarer Zellen nach unserem Architekturmodell in sich sowohl Logik- als auch Verdrahtungsaufgaben vereinigen. Ein geeignetes Modell für derartige Basisblöcke liefern uns nun die im vorigen Abschnitt eingeführten KMN-Einbettungen.

Definition 2.15 (Konfigurierbare Zelle)
Eine *konfigurierbare Zelle* P wird beschrieben durch $P = (\mathcal{E}, \mathcal{T}, \delta)$, wobei $\mathcal{E}$ eine KMN-Einbettung, $\mathcal{T}$ die Menge aller konfigurierbaren Tasks auf $\mathcal{E}$ und δ die Delayfunktion bezeichnet.

Die Delayfunktion $\delta : \mathcal{T} \to \mathbb{R}$ ordnet jedem konfigurierbaren Task $T \in \mathcal{T}$ eine maximale Signalverzögerung zu, welche zwischen den Eingangs- und den Ausgangssignalen von T auftritt, wenn die Zelle auf den Task T konfiguriert wird.

Die Definition konfigurierbarer Zellen auf der Basis von KMN-Einbettungen sieht lediglich ausgangsorientierte Tasks vor. Dies bedeutet beispielsweise auch, daß keine Kaskadierungen von Look-Up-Tables bzw. Latches innerhalb einer Zelle, sondern nur durch Hintereinanderschaltung von Zellen konstruiert werden können. Diese Tatsache wird sich später hinsichtlich der Implementierung von Schaltkreisen, insbesondere bei Entscheidungen hinsichtlich gültiger Zuordnungen, noch als durchaus positiv erweisen.

Zur späteren Bewertung der Performanz einer Schaltkreisimplementierung wird ferner ein Verzögerungsmodell (*Delaymodell*) benötigt. Hier wurden in der Vergangenheit insbesondere für die Gatterebene zahlreiche Ansätze publiziert, die von einer Differenzierung der Gattereingänge über Schaltintervalle bis hin zu statischen Aspekten, wie etwa eine Prozeßveränderungen durch Migration oder die Arbeitstemperatur, und dynamischen Faktoren, wie beispielsweise das Übersprechen von Signalen oder der Einfluß von Schleifen, berücksichtigen. Eine Übersicht hierzu gibt beispielsweise [67]. Eine generische Implementierung der Auswertung derartig vielschichtiger Modelle resultiert offensichtlich in einem relativ hohen Berechnungsaufwand, der insbesondere im Zuge der zu entwickelnden Plazierungs- und Verdrahtungsverfahren leicht zu einem nicht unwesentlichen Faktor werden kann. Für unsere Zwecke benötigen wir vielmehr ein einfaches, effizient auswertbares Delaymodell, das in Anlehnung an das eingeführte Modell für konfigurierbare Zellen mit *taskorientierten*, fixierten Delaygrößen $\delta : \mathcal{T} \to \mathbb{R}$ arbeitet.

Wir beginnen die Definition unseres Delaymodells bei der konfigurierbaren Zelle, und setzen das Modell später auf Architekturen fort.

Definition 2.16 (Delaymodell für konfigurierbare Zellen)
Sei $P = (\mathcal{E}, \mathcal{T}, \delta)$ eine konfigurierbare Zelle und $U \subseteq \mathcal{T}$ eine konsistente Taskmenge auf $\mathcal{E}$. Sei ferner zu jedem Eingangspin $p \in I_P$ eine Verzögerung $\beta_U(p) \in \mathbb{R}$ gegeben. Dann ist die *Verzögerung (das Delay)* $\beta_U(o) \in \mathbb{R}$ für einen Ausgangspin $o \in O_P$ unter der Konfiguration U definiert als:

$$\beta_U(o) \stackrel{\text{def}}{=} \begin{cases} \delta(T), & \text{falls } \exists_{T \in U, o = o_T}\ T \text{ Speicher-Task} \\ \delta(T) + \max_{p \in I_T} \beta_U(p), & \text{falls } \exists_{T \in U, o = o_T}\ T \text{ Logik-Task} \\ \delta(T) + \beta_U(i_T), & \text{falls } \exists_{T \in U, o = o_T}\ T \text{ Routing-Task} \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Natürlich ist $\beta_U(p)$ mit $p \in I_P$ unabhängig von U. Ist P Instanz einer programmierbaren Zelle, so bezeichne $\beta_P(p)$ das Delay am Pin p unter der aktuellen Konfiguration von P.

2.3.3 Architekturen

Eine feldprogrammierbare Architektur besteht prinzipiell aus einer repetitiven Anordnung konfigurierbarer Zellen, sowie festgelegten Verbindungen zwischen deren Eingängen und Ausgängen. Der repetitive Aufbau einer Architektur erlaubt offensichtlich eine kompakte Beschreibung durch Definition eines entsprechenden *Architektur-Segmentes*. Dieses setzt sich zusammen aus einer Menge von Instanzen konfigurierbarer Zellen über einer Zelltypmenge, sowie gerichteten Verbindungen, die segment-interne, als auch das Architektur-Segment verlassende Verdrahtungen darstellen können.

Definition 2.17 (Architekturbeschreibung)
Das Oktupel $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ stellt eine *Architekturbeschreibung* dar, wenn $\mathcal{P}$ eine Menge[2] konfigurierbarer Zellen ist, V die Zellinstanzenmenge, E die Menge von Verbindungen zwischen Zellinstanzen, $\varphi : V \to \mathcal{P}$ die Typenabbildung, Q und Z zwei bijektive Randabbildungen, $r \in \mathbb{N}$ die Dimension und $\tau : E \to \mathbb{Z}^r$ die Transitionsabbildung bezeichnet.

Während die Menge V also Instanzen konfigurierbarer Zellen enthält, gibt die Abbildung $\varphi : V \to \mathcal{P}$ jeweils deren Zelltyp an. Die Menge E enthält Objekte, die feste Leitungen (Verbindungen, *Links*) zwischen den Ausgängen und Eingängen der konfigurierbaren Zellen aus V repräsentieren. Wir definieren den *Rand* einer Verbindung $e \in E$ über (geordneten) Paaren von Pins der konfigurierbaren Zellen:

Sei I_v bzw. O_v die Menge der Eingangs– bzw. Ausgangspins der konfigurierbaren Zelle $v \in V$ und $e \in E$ eine Verbindung der Architekturbeschreibung. Dann liefern die Abbildungen $Q : E \to V \times \bigcup_{v \in V} O_v$ bzw. $Z : E \to V \times \bigcup_{v \in V} I_v$ zu jedem Element $e \in E$ ein Zellinstanz/Pin-Tupel (v, p). Mit der Forderung der Bijektivität für beide Randabbildungen erhält man also konkreterweise eine Identifikation von Eingangs- und Ausgangspins der Instanzen programmierbarer Zellen. Sie resultiert zwangsläufig als Konsequenz aus unserer Forderung nach vollständiger Kapselung aller Logik- und Verdrahtungs-Funktionalität in den programmierbaren Zellen, unter welche insbesondere auch Signalverzweigungen einzuordnen sind.

Für den Zugriff auf die Komponenten des Randes einer Verbindung führen wir noch die folgenden Abbildungen ein: Sei $e \in E$ eine Verbindung mit $Q(e) = (v_1, p_1)$ und $Z(e) = (v_2, p_2)$. Dann sei $Q^{\text{CELL}}(e) \stackrel{\text{def}}{=} v_1$, $Z^{\text{CELL}}(e) \stackrel{\text{def}}{=} v_2$, $Q^{\text{PIN}}(e) \stackrel{\text{def}}{=} p_1$ und $Z^{\text{PIN}}(e) \stackrel{\text{def}}{=} p_2$.

Aus der obigen Definition einer Architekturbeschreibung läßt sich nun direkt eine feldprogrammierbare Architektur konstruieren, denn $G = (V, E, r, \tau)$ bildet offensichtlich einen statischen Graphen, durch dessen Abrollen man einen periodischen Graphen erhält und damit das gesuchte Modell für Architekturen. Wir bezeichnen im folgenden daher Zellinstanzen auch als *Knoten* und Verbindungen auch als *Kanten* einer Architektur.

Definition 2.18 (Architektur)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung und $x \in \mathbb{N}^r$. Eine *Architektur* $\mathcal{A}^x$ ist ein periodischer Graph mit Dilatation x, dessen Konstruktionsvorschrift mit dem statischen Graphen der Architekturbeschreibung $\mathcal{A}$ gegeben ist.

[2] Genauer: eine Menge von Typen programmierbarer Zellen.

Nun können wir auch das mit Definition 2.16 eingeführte Verzögerungsmodell der konfigurierbaren Zellen auf Architekturen fortsetzen. Aufgrund der Bijektivität der Randabbildungen Q und Z der Architekturbeschreibung, welche beim Übergang vom statischen auf den periodischen Graph der Architektur (bis auf Nullkanten) ja erhalten bleibt, kann der Begriff des *Delays* von Eingangs- und Ausgangspins auf Kanten übertragen werden:

Definition 2.19 (Delaymodell für Architekturen)
Sei $\mathcal{A}^x$ eine Architektur zur Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ und $x \in \mathbb{N}^r$. Dann ist das Delay für Kanten $e \in E^x$ wie folgt definiert:

$$\beta(e) \stackrel{\text{def}}{=} \begin{cases} \beta_{Q^{\text{CELL}}(e)}\big(Q^{\text{PIN}}(e)\big), & \text{falls } Q(e) \in V^x \text{ existiert} \\ 0, & \text{sonst} \end{cases}$$

Mit dieser Definition wird also, basierend auf einer gegebenen Konfiguration einer Architektur, die Verzögerung eines Signals von seiner Quelle bis zu einer gegebenen Kante des Architekturgraphen eindeutig beschrieben, wobei den Kanten des Architekturgraphen selbst keine Verzögerung zugeordnet ist. Sie besitzen also lediglich eine zuordnende Bedeutung.

2.3.4 Konfigurierbare Pfade und Bäume

Der Verlauf von Signalen auf einer Architektur wird durch Pfade bestimmt, die sich über der Konfigurierbarkeit der Architektur bestimmen. In der nachfolgenden Definition modellieren wir solche *konfigurierbaren Pfade*.

Definition 2.20 (Konfigurierbarer Pfad, K-Pfad)
Ein *K-Pfad* W der Länge n auf einer Architektur $\mathcal{A}^x$ mit Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ und $x \in \mathbb{N}^r$ ist eine Folge $W = (e_1 \dots e_n)$ paarweise verschiedener Kanten $e_i \in E^x$, wobei gilt:

$$\forall_{i=2\dots n}\ \exists_{T_i \in \mathcal{T}_{Q^{\text{CELL}}(e_i)}}\ T_i \text{ Routing-Task mit:}$$

(1) $Z^{\text{PIN}}(e_{i-1}) = i_{T_i}$

(2) $Q^{\text{PIN}}(e_i) = o_{T_i}$

(3) $\forall_{j \in \{2\dots n\}}\ Q^{\text{CELL}}(e_i) = Q^{\text{CELL}}(e_j) \Longrightarrow T_i \sim T_j$

Während die Bedingungen (1) und (2) der Definition die Existenz eines Routing-Tasks zwischen Kanten e_{i-1} und e_i sicherstellen, drückt Bedingung (3) die Forderung nach Kompatibilität von Tasks aus, falls der K-Pfad mehr als einmal durch eine konfigurierbare Zelle verläuft.

Es sei bemerkt, daß Definition 2.20 die K-Pfade nicht etwa als eine Folge von Routing-Tasks betrachtet, also keine explizite Konfiguration einer Architektur enthält, sondern vielmehr über Verdrahtungsaufgaben abstrahiert. In Konsequenz hiervon kann es sich also durchaus auch um *Mengen* von (äquivalenten) Routing-Tasks handeln, die zwei Kanten eines K-Pfades verbinden. Dementsprechend sei die aus der Kantenfolge eines K-Pfades W implizierte Folge von Routing-Task-Mengen mit $\mathcal{R}_W$ bezeichnet. Aus der

Definition folgt schließlich noch, daß ein K-Pfad keine Verzweigungen besitzt und ferner stets azyklisch ist.

Die Abbildungen Q und Z setzen wir auf einem K-Pfad $W = (e_1 \dots e_n)$ wie folgt fort: $Q(W) \stackrel{\text{def}}{=} Q(e_1)$ und $Z(W) \stackrel{\text{def}}{=} Z(e_n)$, falls Quelle beziehungsweise Ziel existieren. Entsprechende Bedeutung besitzen Q^{CELL}, Z^{CELL}, Q^{PIN} und Z^{PIN}.

K-Pfade können im übrigen auch unter Einschränkung auf eine Architekturbeschreibung betrachtet werden, wenn die Nichtzyklizität lediglich auf durch Externkanten begrenzten Abschnitten des K-Pfades gefordert wird. Sie stellen formal eine Verfeinerung der in der Literatur bereits betrachteten *m-Pfade* auf periodischen Graphen dar [31].

Mit dem Modell der K-Pfade ist es nun möglich, Implementierungen von Netzen mit jeweils einem Start- und Zielterminal (*Zweipunktnetze*) zu beschreiben. Zur Implementierung von Multiterminalnetzen eines Schaltkreises genügen jedoch einfache K-Pfade nicht. Wir kombinieren daher K-Pfade zu einer neuen Struktur, den *konfigurierbaren Bäumen*, wobei zuvor noch einige Begriffe festzulegen sind.

Zwei K-Pfade W_1 und W_2 werden als *disjunkt* bezeichnet, wenn sie keine gemeinsamen Kanten haben. Sie heißen ferner *kompatibel*, wenn sie disjunkt und alle ihre Routing-Task-Mengen aus $\mathcal{R}_{W_1}$ und $\mathcal{R}_{W_2}$ kompatibel sind.

Definition 2.21 (Konkatenation von K-Pfaden)
Zwei K-Pfade $W_1 = (e_{1,1} \dots e_{1,n(1)})$ und $W_2 = (e_{2,1} \dots e_{2,n(2)})$ heißen *konkateniert*, wenn sie disjunkt sind und ferner Routing-Tasks T_1, T_2 (sogenannte *Brückentasks*) existieren, so daß genau eine der beiden folgenden Bedingungen gilt:

(1) $\exists_{T_1 \in \mathcal{T}_{Q^{\text{CELL}}(e_{2,1})}} \; i_{T_1} = Z^{\text{PIN}}(e_{1,n(1)}), o_{T_1} = Q^{\text{PIN}}(e_{2,1})$

(2) $\exists_{T_1, T_2 \in \mathcal{T}_{Q^{\text{CELL}}(e_{2,1})}} \; i_{T_1} = i_{T_2}, o_{T_1} = Q^{\text{PIN}}(e_{1,1}), o_{T_2} = Q^{\text{PIN}}(e_{2,1}), T_1 \sim T_2$

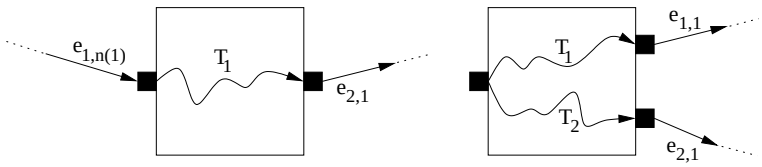


Abbildung 2.13: Konkatenation von K-Pfaden

Abbildung 2.13 illustriert nochmals die beiden Möglichkeiten der Konkatenation Paares von K-Pfaden. Man beachte, daß durch diese Definition eine Konkatenation mehrerer K-Pfade stets rekonvergenzfrei bleibt.

Eine Menge konkatenierter K-Pfade wird als *kompatibel* bezeichnet, wenn die K-Pfade selbst und ihre Brückentasks kompatibel sind.

Definition 2.22 (Konfigurierbarer Baum, K-Baum)
Sei $\mathcal{A}^x$ eine Architektur mit Beschreibung $\mathcal{A}$. Dann ist ein *K-Baum* eine Menge zusammenhängender konkatenierter K-Pfade auf $\mathcal{A}^x$.

Zwei K-Bäume heißen *kompatibel* genau dann, wenn alle ihre K-Pfade kompatibel sind.

Hinsichtlich der Definition von Quellen- und Ziel-Abbildungen eines K-Baumes B gehen wir nun jedoch von den bisherigen Zelle/Pin-Tupeln auf Kanten über. So bezeichne $Q(B)$ die Menge der Startkanten jener K-Pfade von B, die keinen Vorgängerpfad besitzen. Nach Konstruktion der K-Bäume besteht dann $Q(B)$ entweder lediglich aus einer Kante oder alle Kanten in $Q(B)$ gehen von derselben konfigurierbaren Zelle aus und besitzen kompatible Brückentasks ab demselben Eingangspin der Zelle. Es bezeichne $Q^{\text{CELL}}(B)$ diese eindeutige Zelle und $Q^{\text{PIN}}(B)$ die Menge der entsprechenden Pins, falls die Zelle existiert, was lediglich im Falle $Q(B) = \{e\}$ mit Nullkante e nicht auftreten muß. Analog sei $Z(B)$ die Menge aller Endkanten jener K-Pfade von B, die keinen Nachfolgerpfad besitzen. Für alle Kanten in $Z(B)$, die keine Nullkanten sind, liefere Z^{CELL} beziehungsweise Z^{PIN} die Mengen der entsprechenden Ziele.

Mit der Definition der K-Pfade und K-Bäume kann nun eine wichtige Charakterisierung der im Zentrum der späteren Betrachtungen stehenden Architekturen getroffen werden. Ausgeschlossen werden sollen nämlich Architekturen, die aufgrund unidirektional eingeschränkter Verdrahtungsstrukturen nicht in der Lage sind, sequentielle LUT-Schaltkreise beliebiger Struktur zu implementieren. Hinsichtlich der Erreichbarkeit von Netzterminalen führen wir deshalb den Begriff der *vernünftigen Architektur* ein:

Definition 2.23 (Vernünftige Architektur)
Sei $\mathcal{A}^x$ eine Architektur zur Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ mit $x \in \mathbb{N}^r$. Die Architektur $\mathcal{A}^x$ heißt *vernünftig* genau dann, wenn für jeden beliebigen Ausgangspin o und jede beliebige Menge von Eingangspins I stets ein K-Baum B existiert mit $Q^{\text{PIN}}(B) = o$ und $Z^{\text{PIN}}(B) = I$.

2.3.5 Implementierung von Schaltkreisen

Nachdem im vorangegangenen Unterabschnitt bereits von „Implementierungen eines Schaltkreises“ gesprochen wurde, soll der Begriff der Realisierung eines sequentiellen Look-Up-Table-Schaltkreises auf einer feldprogrammierbaren Architektur im folgenden modelliert werden.

Definition 2.24 (Implementierbarkeit von Schaltkreisknoten)
Sei $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionen $V_\mathcal{C} = I \dot\cup O \dot\cup F \dot\cup L$ und sei $P = (\mathcal{E}, \mathcal{T}, \delta)$ eine konfigurierbare Zelle. Ein Look-Up-Table-Knoten $f \in F$ heißt in P *implementierbar* genau dann, wenn es mindestens einen Logik-Task $T \in \mathcal{T}$ gibt mit $\#I_T \geq \#I_f$. Ein Latch-Knoten $f \in L$ heißt in P *implementierbar* genau dann, wenn es mindestens einen Speicher-Task in P gibt.

Als *Implementierung* eines Schaltkreisknotens $f \in F \cup L$ bezeichnen wir die Menge $\mathcal{T}_f \subseteq \mathcal{T}$ aller Tasks, vermöge derer f implementierbar ist. Eine Menge $\mathcal{F} \subseteq F \cup L$ von Schaltkreisknoten heißt in der konfigurierbaren Zelle P implementierbar, wenn gilt:

(1) $\forall_{f \in \mathcal{F}}$ f ist in P implementierbar

(2) $\forall_{f \in \mathcal{F}} \exists^1_{T_f \in \mathcal{T}_f} \bigcup_{f \in \mathcal{F}} T_f$ ist eine konsistente Taskmenge

Eine Implementierung funktionaler Knoten eines Schaltkreises beinhaltet also zwar eine konkrete Bindung an eine konfigurierbare Zelle der Architektur, jedoch nicht notwendigerweise eine Bindung an eine konkrete Konfiguration der Zelle.

Eine Sonderrolle bei der Implementierung von Schaltkreis-Komponenten spielen die übrigen Knoten eines Schaltkreises: seine primären Eingänge und Ausgänge. Ausgehend von unserem architekturstrukturellen Ansatz, lassen sich die Anforderungen an Ein- und Ausgaben eines Schaltkreises auf die externe Verfügbarkeit der entsprechenden Signale reduzieren. Wir definieren daher eine Implementierung von Knoten $v \in I \cup O$ als injektive Abbildung auf die Menge der Nullkanten des Architekturgraphen.

Definition 2.25 (Implementierbarkeit von Netzen)
Sei $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionen $V_\mathcal{C} = I \dot\cup O \dot\cup F \dot\cup L$ und $N \in E_\mathcal{C}$ ein Netz mit Quelle $v_s \in V_\mathcal{C}$ und Zielen $v_1 \ldots v_n \in V_\mathcal{C}$. Sei $\mathcal{A}^x$ eine feldprogrammierbare Architektur mit Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$. Dann heißt N auf $\mathcal{A}^x$ *implementierbar*, falls ein konfigurierbarer Baum B auf $\mathcal{A}^x$ existiert, wobei gilt:

$$\begin{array}{lrcl} (1) & v_s \in I & \Longrightarrow & Q(B) = \{e\} \text{ und } e \text{ ist Nullkante} \\ (2) & v_s \in F \cup L & \Longrightarrow & Q^{\text{CELL}}(B) \text{ definiert und} \\ & & & v_s \text{ implementierbar in } Q^{\text{CELL}}(B) \end{array}$$

und für $i = 1 \ldots n$:

$$\begin{array}{lrcl} (3) & v_i \in O & \Longrightarrow & \exists^1_{e \in Z(B)}\ e \text{ ist Nullkante} \\ (4) & v_i \in F \cup L & \Longrightarrow & \exists^1_{c \in Z^{\text{CELL}}(B)}\ v_i \text{ ist implementierbar in } c \end{array}$$

Implementierungen von Netzen, also K-Bäume, erstrecken sich damit im allgemeinen über mehrere konfigurierbare Zellen hinweg, wobei in den durchlaufenen Zellen entsprechende Routing-Tasks verlangt werden. Sind konfigurierbare Zellen nun durch Implementierungen mehrerer Netze, Teile von Netzen oder gar Knoten eines Schaltkreises genutzt, so sprechen wir von einer *Kompatibilität der Implementierungen*, wenn für jede konfigurierbare Zelle gilt, daß die Vereinigung der von ihr geforderten Tasks eine konsistente Taskmenge bildet.

Nachdem wir nun die Implementierungen der Komponenten eines sequentiellen Look-Up-Table-Schaltkreises modelliert haben, gestaltet sich der Übergang auf den Schaltkreis selbst relativ einfach:

Definition 2.26 (Schaltkreis-Implementierung)
Sei $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionen $V_\mathcal{C} = I \dot\cup O \dot\cup F \dot\cup L$ und sei $\mathcal{A}^x$ eine feldprogrammierbare Architektur mit Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$. Dann stellt $\mathcal{I} = (K, \rho^{\text{FUNC}}, \rho^{\text{NET}})$ eine *Implementierung* von $\mathcal{C}$ auf $\mathcal{A}^x$ dar, wobei K eine Menge konfigurierbarer Bäume ist und $\rho^{\text{FUNC}} : F \cup L \to V^x$, sowie $\rho^{\text{NET}} : E_\mathcal{C} \to K$ Implementierungsabbildungen. Eine *gültige* Schaltkreis-Implementierung liegt vor, wenn gilt:

(1) $\forall_{f \in F \cup L}\ f$ ist implementierbar in $\rho^{\text{FUNC}}(f)$
(2) $\forall_{N \in E_\mathcal{C}}\ N$ ist implementierbar auf $\mathcal{A}^x$ vermöge $\rho^{\text{NET}}(N)$
(3) Alle Knoten- und Netz-Implementierungen sind kompatibel.

Die im Abschnitt 2.3 eingeführten Modelle für feldprogrammierbare Architekturen wurden in Form eines C^{++}-Klassensystems objektorientiert realisiert. Zum besseren Verständnis der in den späteren Kapiteln vorgestellten Verfahren wird im folgenden Abschnitt eine kleine Übersicht über die Implementierung der speziellen Datenstrukturen und Algorithmen gegeben.

2.4 Eine objektorientierte Implementierung

2.4.1 Konfigurierbare Zellen und Architekturen

Ein illustratives Beispiel

Die Abbildung 2.14 zeigt rechts das Beispiel der graphischen Spezifikation einer einfachen konfigurierbaren Zelle. Die Zelle besitzt zwei Eingänge x_0, x_1 und zwei Ausgänge y_0, y_1, sowie eine Look-Up-Table (LUT) mit zwei Eingängen und ein Daten-Latch. Sie ist mittels der über fünf *Programming Bits (pbits)* konfigurierbaren Multiplexer in der Lage, verschiedene Routing-, Logik- und Speichertasks zu erfüllen, wobei maximal zwei Tasks gleichzeitig konfigurierbar sind. Links in der Abbildung ist die, aus der strukturellen Spezifikation mittels des bereits vorgestellten Algorithmus 2.2 berechnete Taskmenge dargestellt – jeweils mit Signalverzögerungswerten, sowie Konfigurationen. Die Konfigurationsabbildung wurde auf der Basis einer beliebig festgelegten Reihenfolge der konfigurierbaren Multiplexer, sowie deren Eingänge bestimmt. Ihre Werte sind als Bitstrings über der Menge {`0,1,x`} dargestellt, wobei `x` für Konfigurationen mit `0` oder `1` an dieser Position verwendet wird. Die Verzögerungswerte sind in diesem Beispiel allesamt auf den konstanten Wert 1.00 festgesetzt.

```
.progcell CSR0202V001

.inputs 2
.outputs 2
.pbits 5
.latches 1
.luts 1

@0:  ROUTE(0)     1.00 <x00xx>
@0:  ROUTE(1)     1.00 <x01xx>
@0:  LUT0(0,1)    1.00 <x10xx>
@0:  LATCH(0)     1.00 <011xx>
@0:  LATCH(1)     1.00 <111xx>
@1:  ROUTE(1)     1.00 <xxx00>
@1:  ROUTE(0)     1.00 <xxx01>
@1:  LUT0(0,1)    1.00 <xxx10>
@1:  LATCH(0)     1.00 <0xx11>
@1:  LATCH(1)     1.00 <1xx11>
```

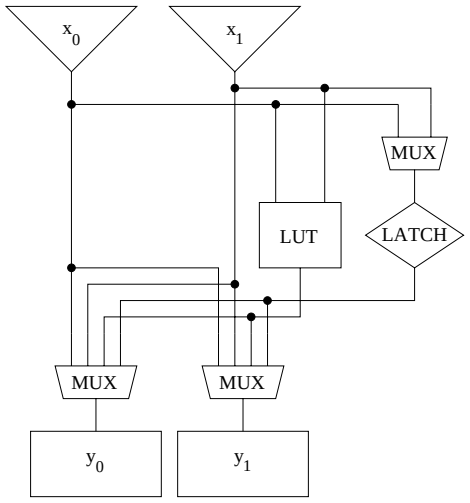


Abbildung 2.14: Spezifikation einer konfigurierbaren Zelle

Abbildung 2.15 zeigt nun rechts den statischen Graphen einer Architekturbeschreibung. Hier wurden vier Instanzen der oben vorgestellten konfigurierbaren Zelle in einer zweidimensionalen Architektur eingebettet. Links in der Abbildung ist die entsprechende syntaktische Spezifikation dargestellt. Während die Typenabbildung der Architekturbeschreibung im Abschnitt „`.progcells`“ definiert ist, sind die Kanten des Graphen zusammen mit den jeweiligen Bildern der Transitionsabbildung im Abschnitt „`.connections`“ aufgelistet. Der statische Graph besitzt offensichtlich vier Intern- und vier Externkanten.

```
.architecture SM2V001S

.dimension 2

.progcells
  #CSR0202V001 : c00 c01 c10 c11

.connections
  0@c10 : 0@c11 [0,0]
  0@c00 : 0@c01 [0,-1]
  0@c01 : 0@c00 [0,0]
  0@c11 : 0@c10 [0,1]
  1@c10 : 1@c00 [1,0]
  1@c11 : 1@c01 [0,0]
  1@c00 : 1@c10 [0,0]
  1@c01 : 1@c11 [-1,0]
```

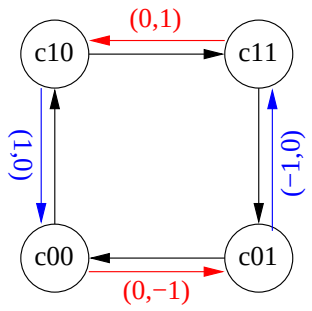


Abbildung 2.15: Beispiel einer Architekturbeschreibung

Der aus dem statischen Graphen von Abbildung 2.15 durch Abrollen erhaltene periodische Graph besitzt eine Maschenstruktur (*Single Mesh*), wie sie in Abbildung 2.16 skizziert ist. Bei der gezeigten zweidimensionalen Darstellung wurde der periodische Graph zu einer Dilatation von $(2, 2)$ abgerollt. Die Pins der zugrundegelegten konfigurierbaren Zelle wurden graphisch orthogonal zueinander angeordnet: x_0 und y_0 horizontal, sowie x_1 und y_1 vertikal.

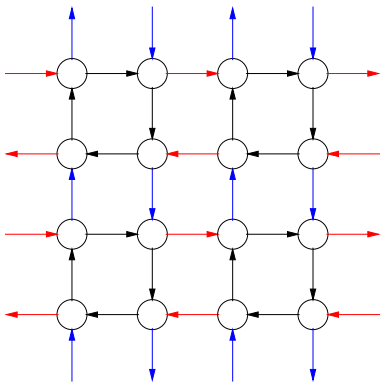


Abbildung 2.16: Struktur einer *Single Mesh* Architektur

Architekturklassen

Für solche, im obigen Beispiel gezeigten, syntaktischen Spezifikationen konfigurierbarer Zellen und Architekturbeschreibungen wurden spezielle Parser entwickelt, welche entsprechende Klassenobjekte instanziieren. Um die Ermittlung der syntaktischen Spezifikationen, insbesondere der Taskmengen zu automatisieren, wurde im Rahmen einer vom Verfasser betreuten Studienarbeit [51] ein graphisches Layoutsystem, vornehmlich für konfigurierbare Zellen und Architekturen, entwickelt (siehe Abbildung 2.17), welches die Daten, ausgehend von einem graphischen Entwurf, erzeugt.

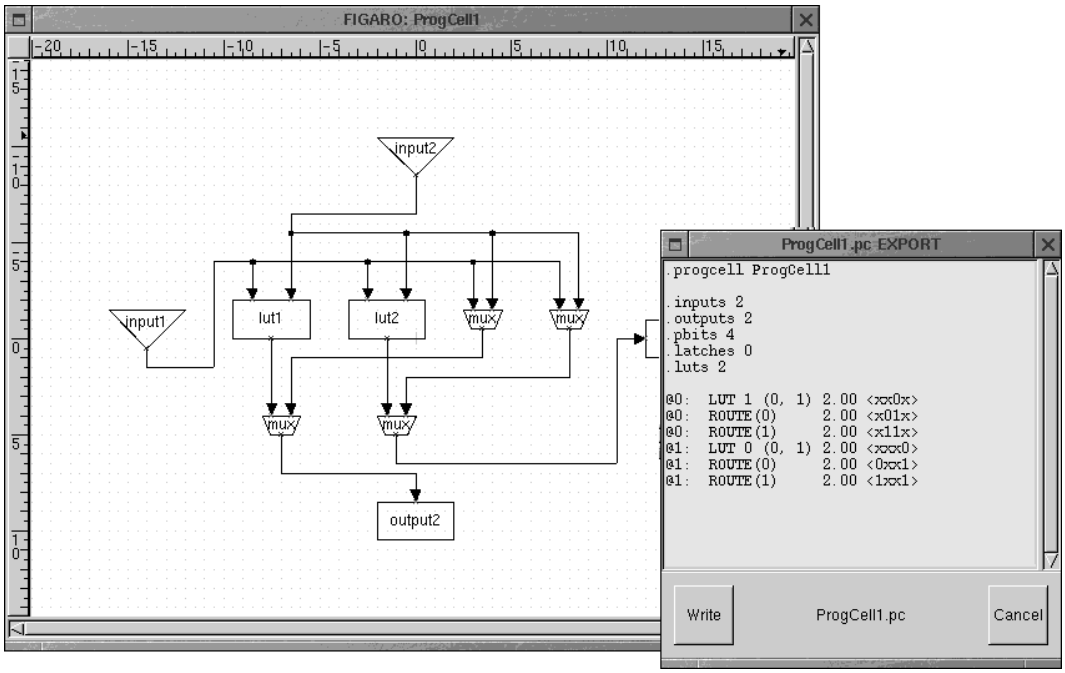


Abbildung 2.17: Ein graphisches Layoutsystem

Abbildung 2.18 gibt eine Übersicht über die verschiedenen, an der rechnerinternen Darstellung von Architekturen beteiligten Objektklassen.

Das Basisobjekt einer Architekturbeschreibung liefert die Klasse `FGArchDesc`, welche Instanzen aller in der Architektur auftretenden Zelltypen (Klasse `ProgCell`) beinhaltet. Die Struktur einer Architektur ist hier lediglich noch in abstrakter Form abgelegt. Eine besondere Aufgabe der Klasse für Architekturbeschreibungen liegt jedoch in den Methoden zur Feststellung der Kompatibilität zweier Tasks beziehungsweise der Konsistenz von Taskmengen. Auf das hierzu implementierte Verfahren wird jedoch genauer in Abschnitt 2.4.3 eingegangen. Festgehalten werden soll an dieser Stelle nur noch das Konzept der Kapselung aller statischen Eigenschaften einer Architektur und der mit ihnen zusammenhängenden Berechnungen in der genannten Beschreibungsklasse.

Mit der in einem Architekturbeschreibungsobjekt der Klasse `FGArchDesc` enthaltenen Konstruktionsvorschrift kann nun ein getyptes Objekt `PGStaticGraph` instanziiert werden, das einen statischen Graphen repräsentiert. Die ausprägenden Typen eines `PGStaticGraph`-Objektes sind die drei Klassen `AGProgCellTyp`, `AGSegmentTyp` und `AGLinkTyp`,

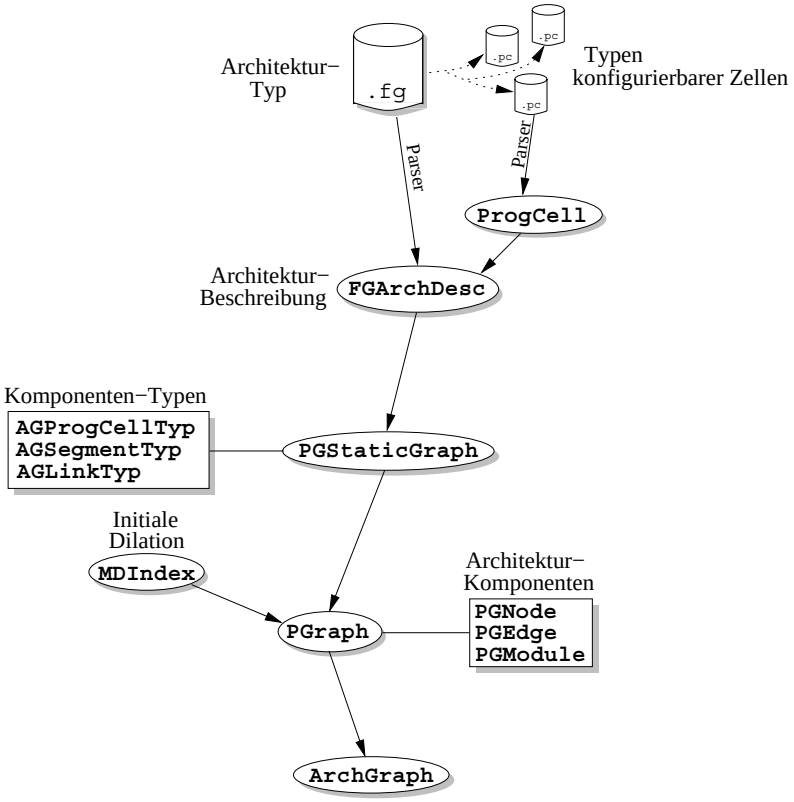


Abbildung 2.18: Klassensystem für Architekturen

welche die für spätere Implementierungen[3] von Schaltkreisen erforderlichen Datenstrukturen bereits beinhalten. Bei der Instanzen von konfigurierbaren Zellen repräsentierenden Klasse `AGProgCellTyp` beispielsweise, ist dies in erster Linie die Menge konfigurierter Tasks, sowie die an ihren Pins verfügbaren Netze. `AGSegmentTyp` repräsentiert Instanzen des statischen Graphen im periodischen Graphen, `AGLinkTyp` hingegen Kanteninstanzen des Graphen. Die Typklassen bilden beim Abrollen des statischen Graphen also Vorlagen für die Architekturkomponenten repräsentierenden Knoten- und Kantenobjekte des periodischen Graphen.

Ein solches Abrollen, also die Konstruktion eines periodischen Graphen über die Klasse `PGraph`, erfolgt mit der Instanziierung eines Architekturgraphen (Klasse `ArchGraph`), wobei neben einem getypten statischen Graphen auch ein initialer Dilatationsvektor (Klasse `MDIndex`) mitübergeben wird. Auf die Anzahl konfigurierbarer Zellen eines abgerollten Architekturgraphen kann nicht direkten Einfluß genommen werden. Genauer ausgedrückt, ist ein Hinzufügen beziehungsweise Entfernen einzelner Knoten (Zellen) oder Graphkanten nicht möglich. Eine Architektur kann nur über segmentweise und dimensionsweise Expansion beziehungsweise Schrumpfung ihrer Dilatation in ihrer Größe verändert werden. Die entsprechenden generischen Methoden liefert bereits die Klasse des periodischen Graphen selbst.

Da Instanzen der genannten Typklassen `AGProgCellTyp`, `AGLinkTyp` und `AGSegmentTyp` nur gekapselt in der `PGraph`-Klasse abgelegt sind, werden die Architektur-Komponen-

[3] Im Sinne einer Abbildung auf eine feldprogrammierbare Architektur.

ten, also Zellen, Verbindungen und Architektur-Segmente (respektive Knoten, Kanten, Module des periodischen Graphen), klassenextern lediglich über verweistragende Objekte der Klassen `PGNode`, `PGEdge` und `PGModule` angesprochen.

Die Klasse `ArchGraph`, welche den periodischen Graphen kapselt, beinhaltet schließlich noch Methoden für diverse Operationen auf konfigurierbaren Architekturen, welche vom Frontend-Zugriff auf strukturelle Informationen der Architektur und der Prüfung auf Kompatibilität von Tasks bis hin zu Methoden zur Konfiguration und Dekonfiguration von K-Pfaden und K-Bäumen reichen.

2.4.2 K-Pfade und K-Bäume

K-Pfade und K-Bäume wurden entsprechend der formalen Modellierung aus Abschnitt 2.3.4 implementiert, lediglich mit der Abweichung, daß ein bereits existierender K-Pfad bei Hinzufügung einer Verzweigung aus Effizienzgründen nicht in zwei K-Pfade unterteilt wird. Beiden Klassen liegen dynamische, rekursive Datenstrukturen zugrunde. Ein K-Baum wird konstruiert, indem er zunächst mittels eines K-Pfades initialisiert wird und dann weitere K-Pfade unter (impliziter) Angabe ihrer Verzweigungspunkte quasi „angehängt“ werden. Während Konstruktion und Erweiterung eines K-Baumes, werden intern Delayinformationen abgelegt, die später eine schnelle Distanzenbewertung ermöglichen.

Algorithmus 2.3 zeigt die Berechnung einer Aktualisierung der Delayinformationen eines K-Baumes. Die Methode *recalc_delay* wird mit dem Vater-K-Pfad des eingefügten beziehungsweise entfernten K-Pfad aufgerufen. Zu einem K-Pfad p liefere p.*delay*(n) das Delay des K-Pfades bis zum n-ten Knoten und p.*subdelay* bezeichne das Maximum über das gesamte Delay von p, sowie jenes seiner Subpfade. Da sich K-Pfad-Objekte nach der Aufnahme in einen K-Baum nicht mehr ändern, werden Delayinformationen zu jedem Knoten, über den ein K-Pfad läuft, bereits mit der Konstruktion des K-Pfades erzeugt und auch dort intern abgelegt, so daß ein Zugriff mittels der Methode *KPfad::delay(...)* in konstanter Zeit, also $\mathcal{O}(1)$, möglich ist.

Algorithmus 2.3 (Delayaktualisierung bei K-Bäumen)

```
KBaum::recalc_delay(KPfad p)
{
    DelayValue newsubdelay = p.delay(p.length);
    forall (Sub-K-Pfade q von K-Pfad p)
    {
        DelayValue d = p.delay(q.forkpos - 1) + q.subdelay;
        if (d > newsubdelay)
            newsubdelay = d;
    }
    if (newsubdelay > p.subdelay)
    {
        p.subdelay = newsubdelay;
        update_delay(p);
    }
```

```
    else
        if (newsubdelay < p.subdelay)
        {
            p.subdelay = newsubdelay;
            recalc_delay(p.father);
        }
}
KBaum::update_delay(KPfad p)
{
    if (p.is_toppath)
        return;

    DelayValue newsubdelay
        = p.subdelay + p.father.get_delay(p.forkpos - 1);
    if (newsubdelay > p.father.subdelay)
    {
        p.father.subdelay = newsubdelay;
        update_delay(p.father);
    }
}
```

Besitzt ein K-Baum nun die Tiefe t, so liegt die Laufzeit von Algorithmus 2.3 bei $\mathcal{O}(t)$. Beim Einfügen eines neuen K-Pfades gilt diese Schranke stets, da sich das Delay des K-Baumes nur erhöhen kann und somit höchstens eine Rekursion der Tiefe t über die Methode *update_delay* durchgeführt wird. Im worst-case kann die Laufzeit bei insgesamt m K-Pfaden jedoch auch bis zur Ordnung $\Theta(m)$ betragen. Letzteres tritt genau dann auf, wenn ein K-Pfad entfernt wurde, der am Gesamtdelay des K-Baumes beteiligt war, also auf dem *kritischen Pfad* lag und ferner auf diesem Pfad bis zur Wurzel die Delays *aller* K-Pfade über die Methode *recalc_delay* durchmustert werden, mit anderen Worten: wenn alle Sub-K-Pfade der entlang dieses Weges durchlaufenen K-Pfade keine weiteren Sub-K-Pfade besitzen.

Insgesamt betrachtet, liefern die beiden Klassen für K-Pfade und K-Bäume lediglich „Containerobjekte“, insofern ihre Strukturen auf Verweise auf Komponenten einer festen Architekturgraphen-Instanz basieren.

2.4.3 Kompatibilität von Tasks

Mit dem Beispiel aus Abschnitt 2.4.1 wurde bereits die Darstellung von Tasks einer konfigurierbaren Zelle in unserem Modell für Architekturbeschreibungen skizziert. Jeder Task bestimmt sich demnach durch (eine Menge von) Bitstrings über $\{0, 1, x\}$, abgeleitet aus den konkatenierten Konfigurationen der beiden KMN der Zelle, in der Literatur auch als Belegungen der *Programming Bits* bezeichnet. Für das genannte Beispiel einer konfigurierbaren Zelle zeigt die Abbildung 2.19 eine graphische Darstellung aller Tasks, wobei die Pins x_0 und y_0 der Zelle horizontal, die Pins x_1 und y_1 vertikal angeordnet wurden.

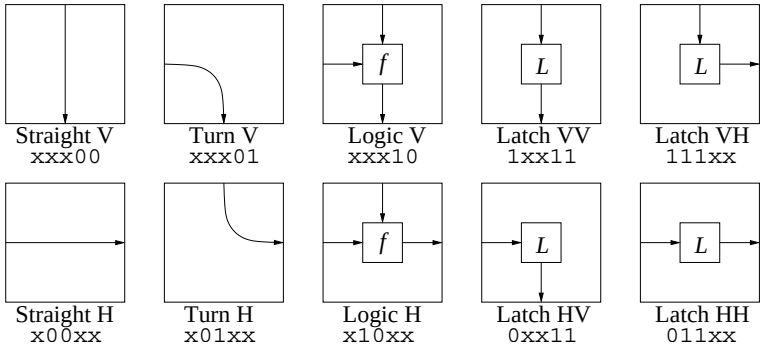


Abbildung 2.19: Tasks der Zelle CSR0202V001

Die zehn Tasks der Beispielzelle sind unterteilt in vier Routing-Tasks, zwei Logik-Tasks und vier Speichertasks. Jedem Task ist im vorliegenden Falle genau ein Don't-Care-Bitmuster zugeordnet. Die Zelle kann auf maximal zwei Tasks gleichzeitig konfiguriert werden, wobei sich die Kompatibilität der Tasks über die Schnitte der Konfig-Sets der entsprechenden Routing-Tasks in den KMN der Zelle erklärt (vgl. Definition 2.8). Von den im Beispiel insgesamt 18 möglichen Task-Kombinationen sind in Abbildung 2.20 fünf illustriert. Hier ergibt sich beispielsweise der Task *Cross* mit dem Bitmuster `x0000` durch den Schnitt der Bitmuster der beiden Tasks *Straight H* und *Straight V*.

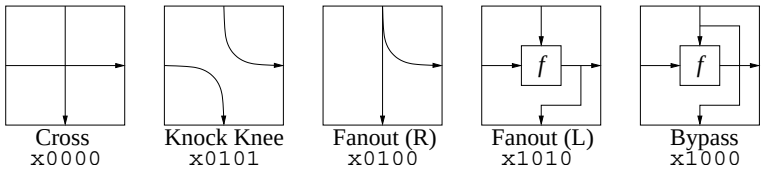


Abbildung 2.20: Kompatible Tasks

Da bei den späteren Plazierungs- und Verdrahtungs-Verfahren zur Implementierung von Schaltkreisen auf feldprogrammierbaren Architekturen eine Zelle *sukzessive* konfiguriert wird, wurde für das Prüfverfahren der Kompatibilität von Tasks ein auf dynamischem Programmieren basierter Ansatz gewählt. Beim Problem der sukzessiven Konfiguration von Tasks ist also zu prüfen, ob ein gegebener Task t einer gegebenen konsistenten Taskmenge T hinzugefügt werden kann, ohne daß diese ihre Konsistenzeigenschaft verliert. Algorithmus 2.4 skizziert ein entsprechendes Verfahren.

Algorithmus 2.4 (Konsistenzprüfung von Taskmengen)

```
bool check_consistency(TaskSet T,Task t)
{
    if (t ∈ T)
        return false;

    TaskSet T' = T ∪ { t };

    if (KonfigSetTable[T'] undefiniert)
        KonfigSetTable[T'] = new_konfigset(T,t);
```

```
    if (KonfigSetTable[T'] = ∅)
        return false
    else
        return true;
}

KonfigSet new_konfigset(TaskSet T,Task t)
{
    KonfigSet T' = ∅;

    if (KonfigSetTable[T] definiert)
        T' = KonfigSetTable[T] ∩ t.KonfigSet;
    else
    {
        T' = t.KonfigSet;

        forall (Tasks t' ∈ T)
        {
            T' = T' ∩ t'.KonfigSet;
            if (T' = ∅)
                return ∅;              // Abbruch: Taskmenge nicht konsistent
        }
    }

    return T';

}
```

Zunächst sei nochmals daran erinnert, daß eine Taskmenge gemäß Definition 2.9 *konsistent* heißt, wenn der Schnitt der Konfig-Sets aller in ihr enthaltenen Tasks nichtleer ist. Basis des Algorithmus stellt nun eine Hash-Tabelle für Konfig-Sets dar (`KonfigSetTable`), deren Schlüssel die Taskmengen-Klasse `TaskSet` liefert. Für jeden Zelltyp einer Architektur wird eine eigene Hash-Tabelle von der Architekturbeschreibungsklasse `FGArchDesc` verwaltet.

In der Methode *check_consistency* wird zunächst versucht, das Schnitt-Konfig-Set zur Taskmenge $T \cup \{t\}$ in der Tabelle zu finden. Falls dieses Konfig-Set nicht vorhanden ist, wird es mittels der Methode *new_konfigset* neu berechnet und anschließend in die Hash-Tabelle eingetragen. Die Methode *new_konfigset* prüft, ob für die Taskmenge T ein Schnitt-Konfig-Set existiert. Falls dies der Fall ist, so ist dieses lediglich mit dem Konfig-Set des Tasks t zu schneiden. Falls es nicht existiert, muß das Konfig-Set für $T \cup \{t\}$ durch sukzessives Schneiden der Konfig-Sets aller Tasks gebildet werden, wobei abgebrochen werden kann, wenn sich während der Berechnung die leere Menge ergibt.

Zur Bestimmung der Berechnungskomplexität von Algorithmus 2.4 ist zunächst die Schnitt-Operation auf Konfig-Sets genauer zu betrachten:

Lemma 2.6 (Kardinalität konsistenter Taskmengen)
Eine konfigurierbare Zelle mit p *Programming Bits (PBits)* und n Ausgängen kann höchstens auf $\min(p, n)$ Tasks gleichzeitig konfiguriert werden.

Beweis: Gemäß unseres Modells bedingt die Konfiguration einer Zelle auf einen Task die Belegung genau eines Ausgangs mit einem Signal. Also ist die Zahl gleichzeitig konfigurierbarer Tasks offensichtlich nach oben beschränkt durch die Zahl der Ausgänge der konfigurierbaren Zelle.

Sei $\mathcal{M}$ die Menge aller Multiplexer der konfigurierbaren Zelle. Seien T_1 und T_2 zwei Tasks aus einer konsistenten Taskmenge. Dann müssen verträgliche Routen $R_1 \in T_1$ und $R_2 \in T_2$ existieren. Routen sind genau dann verträglich, wenn sie Multiplexer aus $\mathcal{M}$ ausschließlich exklusiv nutzen. Ein konfigurierbarer Task nutzt mindestens einen Multiplexer, welcher durch mindestens ein PBit konfiguriert wird. Somit ist die maximale Zahl gleichzeitig konfigurierbarer Tasks auch durch die Zahl p der PBits nach oben beschränkt. □

Lemma 2.6 sagt nun zwar, daß die Zahl der Schnitte von Konfig-Sets u.a. durch die Zahl der *Programming Bits* nach oben beschränkt ist, allerdings hängt die Kardinalität eines Konfig-Sets selbst immerhin von der Zahl der Routen in den KMN ab, welche auch in scheinbar trivialen Fällen eine (asymptotisch) ungünstige Entwicklung annehmen kann. Ein kleines Beispiel mag dies illustrieren:

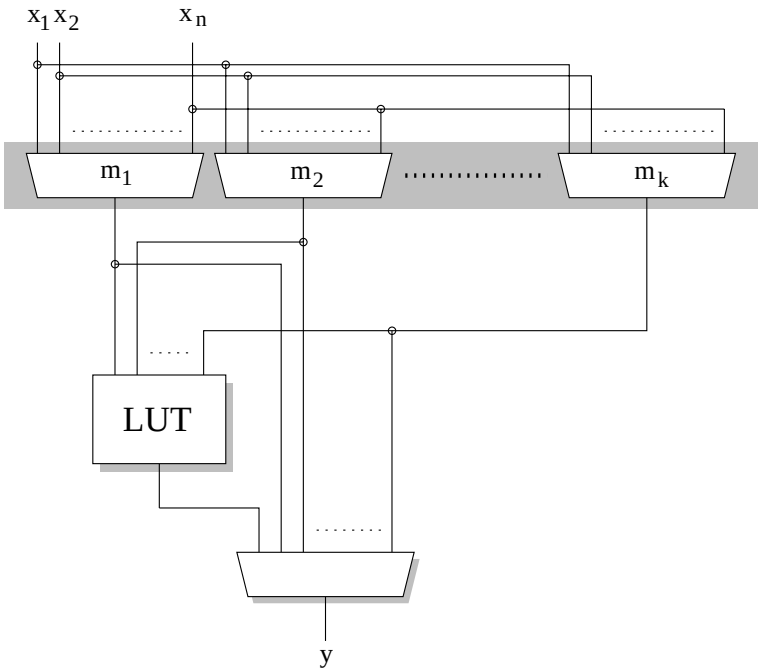


Abbildung 2.21: Beispiel zur Kardinalität von Konfig-Sets

Die Abbildung 2.21 skizziert eine konfigurierbare Zelle mit n Eingängen und einem Ausgang, sowie einer Look-Up-Table mit k Eingängen. Offenbar sind die n Eingänge der Zelle unabhängig voneinander zur Look-Up-Table verdrahtbar vermöge der Multiplexer m_1 bis m_k (*Full Crossbar Switch*). Während ein Routing-Task $y = x_i$ mit $i \in \{1, \ldots, n\}$ im vorliegenden Falle noch mittels k verschiedener Routen realisierbar ist, ergibt sich für den Logik-Task $y = f(I)$ mit $I \subset \{x_1, \ldots, x_n\}$ und $\#I = k$ aufgrund der Vertauschbarkeit der Eingangssignale bei Look-Up-Tables bereits ein Konfig-Set mit $k!$ Elementen.

In Anlehnung an die Stirlingsche Formel[4] ergibt sich für den Schnitt zweier Konfig-Sets in Algorithmus 2.4 damit sogar eine exponentielle Laufzeit in k. Da allerdings die Größe von Look-Up-Tables nicht nur hinsichtlich der Größenordnung, sondern auch absolut beschränkt[5] ist, bleibt das Verfahren dennoch für uns traktabel.

Ebenfalls darf nicht unerwähnt bleiben, daß die Zahl der möglichen Taskmengen, und damit die Größe der Hash-Tabelle, sich auch asymptotisch exponentiell in der Anzahl der *Programming Bits* verhält. Doch auch hier wird sich zum einen die Problemgröße in einem sehr beschränkten Rahmen bewegen, zum anderen wird in der Praxis, insbesondere bei komplexeren Zellen, oft nur ein kleiner Teil der Taskmengen tatsächlich konstruiert.

2.4.4 Alternativen der Implementierung

Bei dem im Rahmen der vorliegenden Arbeit implementierten C^{++}-Klassensystem für feldprogrammierbare Architekturen wurde auch intern als Darstellung von Konfigurationen der bereits beschriebene Ansatz über Don't-Care-Bitstrings gewählt. Schnitte von Konfig-Sets werden demzufolge durch *Matchings* von Strings gleicher Länge realisiert.

Daneben wurden jedoch auch weitere Alternativen hinsichtlich implementierungstechnischer Repräsentation betrachtet. Insbesondere der bereits am Ende von Abschnitt 2.2.1 vorgeschlagene Ansatz der Darstellung von Konfig-Sets durch Boolesche Funktionen, etwa in ROBDD-Darstellung, bot hier eine auf den ersten Blick vielleicht elegantere, doch hinsichtlich der Schnittoperation kaum effizientere Möglichkeit, da auch hier einerseits asymptotisch intraktable Problemgrößen auftreten können [57], jedoch anderseits eine Interpretation der in dieser Form dargestellten Konfigurationen von Architekturen zusätzlichen Berechnungsaufwand zur Folge hat.

Ein vollständig anderer modelltheoretischer Ansatz hinsichtlich der Detektion freier Ressourcen auf feldprogrammierbaren Architekturen wurde mit Überlegungen über eine Reduktion auf Erreichbarkeitsprobleme in einem *Routing-Graphen* verfolgt. Dieser Ansatz besitzt zwar den Vorteil, ohne eine direkte Speicherung von Konfigurationen auszukommen, denn in diesem Fall genügt lediglich eine Markierung bereits belegter Ressourcen. Andererseits ist dann jedoch auch eine Erkennung sich gegenseitig ausschließender Logik- und Speicher-Tasks nicht bereits beim Plazieren, sondern erst beim Verdrahten möglich. Dies macht insbesondere das Floorplanning-Problem für Makros kaum möglich beziehungsweise sehr ineffizient.

2.5 Bewertungsmetriken

Nachdem Architekturen modelltheoretisch nun ausreichend definiert sind, sollen in diesem Abschnitt Kriterien zu ihrer Charakterisierung und Bewertung motiviert werden, welche später im Rahmen von empirischen Untersuchungen auf verschiedene Architekturtypen angewandt werden.

[4] Stirlingsche Formel: $k! \sim (\frac{k}{e})^k \sqrt[k]{2k\pi}$

[5] In der Praxis kommerzieller FPGAs treten zumeist nur Look-Up-Tables mit bis zu vier Eingängen auf.

2.5.1 Allgemeine Betrachtungen

Zum Ressourcenbedarf von Schaltkreisen

Einer der Bestandteile der Implementierung eines synchronen Schaltkreises auf einer feldprogrammierbaren Architektur stellt die Abbildung seiner Funktionen auf die Logik-Ressourcen der Architektur dar. Die Logik-Ressourcen unseres Architektur-Modells bestehen in einer Menge von Look-Up-Tables mit jeweils einer festen Zahl von Eingängen, wobei eine Look-Up-Table mit k Eingängen eine beliebige Boolesche Funktion in ebensovielen Variablen berechnen kann. Betrachtet man den Spezialfall eines rein kombinatorischen Schaltkreises mit n Eingängen $x_1, \ldots, x_n$ und einem Ausgang y, so kann seine globale Funktion $y = f(x_1, \ldots, x_n)$ in einer Look-Up-Table der Breite n realisiert werden oder durch entsprechende Zerlegung (*Technology Mapping*) in mehreren Look-Up-Tables einer festen Breite k.

Interessant scheint nun im Vorfeld der Bewertung von Architekturen, wie viele Look-Up-Tables einer festen Anzahl von k Eingängen zur Berechnung einer beliebigen Booleschen Funktion in n Variablen vonnöten sind. Während hier die untere Schranke bei einer konstanten Funktion trivialerweise bei Null liegt, sind obere Schranken im nachfolgenden Satz 2.4 angegeben, welcher eine Verallgemeinerung einer Behauptung in [22][6] darstellt.

Satz 2.4 (Obere Schranken für Look-Up-Table-Realisierungen)
Zur Berechnung einer n-stelligen Booleschen Funktion werden höchstens m Look-Up-Tables mit jeweils k Eingängen benötigt, wobei gilt:

$$2^{n-k} \leq m < 2^{n-k+1} \quad \text{für } n, k \geq 3$$

Beweis: Die untere der beiden oberen Schranken ergibt sich aus folgender Überlegung: Eine k-Look-Up-Table kann alle 2^{2^k} k-stelligen Funktionen realisieren. Um die 2^{2^n} n-stelligen Funktionen zu überdecken, werden m k-Look-Up-Tables benötigt, also:

$$\begin{aligned} & \left(2^{2^k}\right)^m \geq 2^{2^n} \\ \iff \quad & m \cdot \log 2^{2^k} \geq 2^n \cdot \log 2 \\ \iff \quad & m \cdot 2^k \cdot \log 2 \geq 2^n \cdot \log 2 \\ \iff \quad & m \cdot 2^k \geq 2^n \\ \iff \quad & m \geq 2^{n-k} \end{aligned}$$

Die obere Schranke kann aus einer konkreten Realisierung abgeleitet werden. Dazu wird die n-stellige Boolesche Funktion in exakt $\frac{2^n}{2^k} = 2^{n-k}$ k-stellige Subfunktionen gesplittet. Die übrigen $n-k$ Eingänge der Funktion selektieren die Teilfunktionen über ein Multiplexer-Netzwerk mit Binärbaumstruktur der Tiefe $n-k$. Wird jeder der $2^{n-k}-1$ Multiplexer ebenfalls

[6] siehe dort S. 36

durch eine k-Look-Up-Table mit $k \geq 3$ realisiert, ergibt sich für die Gesamtzahl m der Look-Up-Tables:

$$\begin{aligned} m &= 2^{n-k} + 2^{n-k} - 1 \\ &= 2^{n-k+1} - 1 \\ &< 2^{n-k+1} \end{aligned}$$

□

Offensichtlich hängt der Platzverbrauch der Implementierung eines Schaltkreises inhärent von der „Komplexität“ dessen inneren Funktion ab – oder anders ausgedrückt, davon, inwieweit diese beim Technology-Mapping minimiert werden kann. Während nämlich für Mapping-Probleme ohne Duplikation von Logik sowohl hinsichtlich Flächenbedarf als auch Delay noch effiziente Verfahren existieren, ist das qualitativ bessere Mapping mit Duplikation im allgemeinen als NP-hart bekannt [35].

Während auf dem Gebiet des Technology-Mappings, auch für Look-Up-Table-basierte FPGAs, insbesondere in den 1990er Jahren recht intensiv geforscht und auch eine Vielzahl unterschiedlicher Ansätze gefunden wurde, wird diese Problematik kein Thema der vorliegenden Arbeit sein. Im Falle empirischer Bewertungen werden wir vielmehr von einem technologieunabhängigen Mapping-Verfahren ausgehen, welches einheitlich auf die zu implementierenden Test-Schaltkreise anzuwenden ist und das die Zielarchitektur lediglich hinsichtlich der Zahl der Eingänge der Look-Up-Tables berücksichtigt.

Als geeignetes Synthesetool wurde hier auf das an der *University of California at Berkeley* entwickelte SIS zurückgegriffen [68]. *En detail* wurde SIS für Zielarchitekturen mit k-Look-Up-Tables zunächst mit den Standardskripten *rugged* und *algebraic* gestartet, welche in mehreren Iterationen verschiedene Minimierungsverfahren, beispielsweise über Kern-Extraktion, lokale Don't Cares etc., auf den Schaltkreisknoten anwenden, bevor der maximale Ingrad der Schaltkreisknoten schließlich durch Splitting auf k begrenzt wurde.

Bewertungsziele

Ziel unserer Bewertung von feldprogrammierbaren Architekturen soll eine flächennormierte Quantifizierung der Berechnungskapazität, der -flexibilität und eine performanzorientierte Einschätzung der Leistungsfähigkeit einer Architektur sein. Wie aber definiert man nun Begriffe, wie *Kapazität*, *Flexibilität* und *Performanz*?

Während der Begriff Kapazität abstrakt als Charakterisierung des Ressourcenvorrates einer Architektur noch relativ einfach erklärbar ist, gestaltet sich die Frage nach der Flexibilität, also vielmehr der Nutzbarkeit, weitaus schwieriger. Als Basis einer Einschätzung von Architekturen diesbezüglich, kann zwar die theoretische Analyse eines Architektursegmentes herangezogen werden, wobei auch die Definition beispielsweise flächennormierter Flexibilitätsmetriken gelingt. Allerdings ist eine Fortsetzung solcher Maße auf Architekturen größerer Dilatation im allgemeinen nicht möglich [79].

Auch die Performanz einer Architektur ist lediglich anhand deren Struktur nur unbefriedigend einzuschätzen, da sie stark vom Schaltkreis und seiner Implementierung auf

der Architektur abhängt. Deshalb werden wir im folgenden nicht nur strukturbasierte Metriken für Architekturen einführen, die auf Architekturbeschreibungen angewandt werden, sondern im wesentlichen auch Metriken zum Vergleich von Architekturen im Rahmen der Implementierung konkreter Benchmark-Schaltkreise.

2.5.2 Metriken für Architekturen

Zur Bestimmung geeigneter Kenngrößen für Fläche und Delay einer Architektur wird ein Hardwaremodell benötigt. Für die vorliegende Arbeit wurde ein auf CMOS-Realisierungen basiertes Modell gewählt, welches zur formalen Definition ausgehend von Basiskomponenten über konfigurierbare Zellen und Architektursegmente bis hin zu Architekturen einer gegebenen Dilatation expandiert wird. Hinsichtlich der Flächenmetrik werden wir, wie allgemein gebräuchlich, dabei stets auf NAND2-Gatteräquivalente (zu vier CMOS-Transistoren) normieren.

Fläche

Hinsichtlich der Chipflächen-Kosten für Inverter, Transfergatter und Treiberstufen definieren wir zunächst die folgenden Werte:

$$A_{\mathrm{INV}} = A_{\mathrm{TG}} = \frac{1}{2} \quad \text{und} \quad A_{\mathrm{DRV}} = 1$$

Betrachten wir nun Basiskomponenten konfigurierbarer Zellen. Es handelt sich hier um drei Komponenten: *Multiplexer*, *D-Latches* und *Look-Up-Tables*.

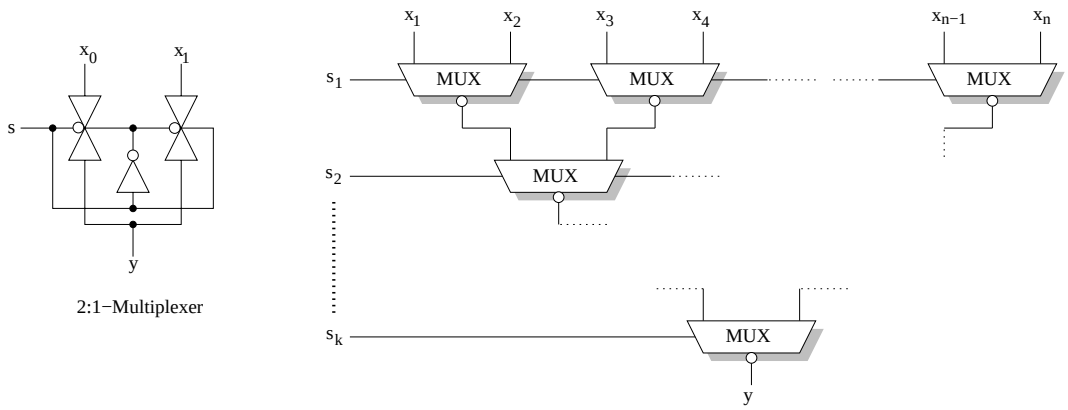


Abbildung 2.22: Konstruktion von Multiplexern

Abbildung 2.22 zeigt links die CMOS-Realisierung eines 2:1-Multiplexers und rechts das Konstruktionsprinzip von Multiplexern variabler Breite. Der 2:1-Multiplexer besteht aus zwei Transfergattern, sowie einem Inverter, also insgesamt sechs Transistoren oder

$$A_{2-\mathrm{MUX}} = \frac{3}{2}$$

Gatteräquivalenten. Ein n:1-Multiplexer mit $k = \lceil \log_2 n \rceil$ Steuereingängen besteht somit aus

$$\sum_{i=1}^{k} 2^{i-1} = \sum_{i=0}^{k-1} 2^i = \frac{1-2^k}{1-2} = 2^k - 1 = n - 1$$

2:1-Multiplexern. Da bei einer solchen Kaskadierung von Multiplexern in der Regel jedoch eine Signaldämpfung auftritt, müssen die Zwischensignale verstärkt werden. Dies wird durch Einfügen einer geraden Anzahl von Inverterstufen erreicht, so daß das Ausgangssignal wieder in positiver Phase vorliegt. Die Anzahl der Inverter in einem balancierten, binären 2:1-Multiplexerbaum läßt sich nun nach dem rekursiven Algorithmus 2.5 berechnen.

Algorithmus 2.5 (Berechnung der Zahl der Inverter)

```
int muxinv(n)
{
    if (n < 4)
        return 0;
    i = 3;                          // 3 Inverter für Vater- und beide Sohnmultiplexer
    i_LL = muxinv(⌈⌈n/2⌉/2⌉);        // Zahl der Inverter im LL-Unterbaum
    i_LR = muxinv(⌊⌈n/2⌉/2⌋);        // Zahl der Inverter im LR-Unterbaum
    i_RL = muxinv(⌈⌊n/2⌋/2⌉);        // Zahl der Inverter im RL-Unterbaum
    i_RR = muxinv(⌊⌊n/2⌋/2⌋);        // Zahl der Inverter im RR-Unterbaum
    return i + i_LL + i_LR + i_RL + i_RR;
};
```

Die Entwicklung der Inverterzahlen ist in Abbildung 2.23 graphisch dargestellt.

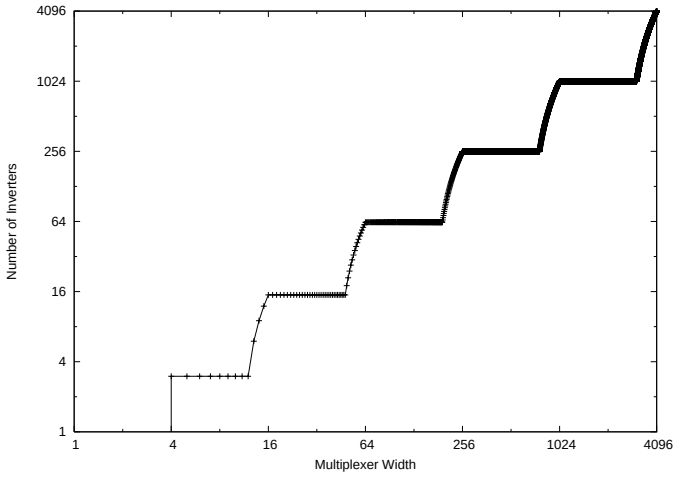


Abbildung 2.23: Zahl der Inverter bei Multiplexerkaskadierungen

Auf der Abszisse des logarithmischen Diagramms ist die Zahl n der Dateneingänge des Multiplexers, auf der Ordinate die Zahl der Stabilisierungsinverter dargestellt. Die nachfolgende Tabelle zeigt einige Werte für kleinere n.

n	2	3	4	5	6	12	13	14	15	16
muxinv(n)	0	0	3	3	3	3	6	9	12	15

Da wir jedes der k Steuersignale einer n:1-Multiplexerkaskadierung lediglich einmal invertieren, entfallen die Inverter der $n-1$ Basis-Multiplexer. Stattdessen kalkulieren wir neben den betrachteten Treiber-Invertern nur mit weiteren k Invertern und können die Fläche eines n:1-Multiplexers schließlich mit der folgenden Zahl von Gatteräquivalenten ansetzen:

$$\begin{aligned} A_{n\text{-MUX}} &= (n-1)\cdot A_{2-\text{MUX}} - (n-1)\cdot A_{\text{INV}} + \text{muxinv}(n)\cdot A_{\text{INV}} + k\cdot A_{\text{INV}} \\ &= (n-1)\frac{3}{2} - (n-1)\frac{1}{2} + \text{muxinv}(n)\frac{1}{2} + \lceil \log_2 n \rceil \frac{1}{2} \\ &= n-1+\frac{\text{muxinv}(n)}{2} + \frac{\lceil \log_2 n \rceil}{2} \end{aligned}$$

Die nachfolgende Tabelle gibt einige Werte für $A_{n\text{-MUX}}$ wieder, welche nach den vorangegangenen Überlegungen berechnet wurden und die im Rahmen der späteren empirischen Betrachtungen benötigt werden:

n	2	3	4	5	6	7	8	9	10	11	12	16	32	64
$A_{n\text{-MUX}}$	1.5	3	5.5	7	8	9	10	11.5	12.5	13.5	14.5	24.5	41	97.5

Die Konfiguration von Multiplexern in KMN wird in SRAM-Zellen gespeichert. Abbildung 2.24 zeigt die CMOS-Realisierung eines solchen 1-Bit-Speichers. Die Zelle besteht aus sechs Transistoren, also ist:

$$A_{\text{SRAM}} = \frac{3}{2}$$

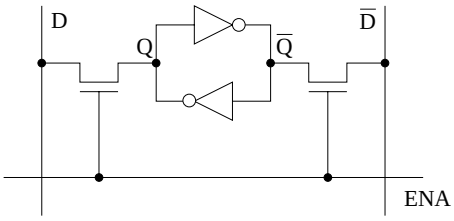


Abbildung 2.24: Realisierung einer SRAM-Zelle

Jedem der k Steuereingänge eines konfigurierbaren $n:1$-Multiplexers (CMUX) wird eine SRAM-Zelle vorgeschaltet. Somit besitzt ein konfigurierbarer $n:1$-Multiplexer die folgende Fläche:

$$\begin{aligned} A_{n\text{-CMUX}} &= A_{n\text{-MUX}} + k\cdot\frac{3}{2} \\ &= n-1+\frac{\text{muxinv}(n)}{2} + \frac{\lceil \log_2 n \rceil}{2} + \lceil \log_2 n \rceil \cdot \frac{3}{2} \\ &= n-1+\frac{\text{muxinv}(n)}{2} + 2\cdot\lceil \log_2 n \rceil \end{aligned}$$

Die nächste Tabelle gibt einige Werte für $A_{n\text{-CMUX}}$ wieder.

n	2	3	4	5	6	7	8	9	10	11	12
$A_{n\text{-CMUX}}$	3	6	8.5	11.5	12.5	13.5	14.5	17.5	18.5	19.5	20.5

Auf der Basis allgemeiner Multiplexer läßt sich auch leicht der Flächenbedarf einer typischen Look-Up-Table-Realisierung ableiten, indem die Tafel einer Booleschen Funktion wiederum mittels SRAM-Zellen gespeichert wird und die gespeicherten Funktionswerte mittels eines Multiplexers abgerufen werden.

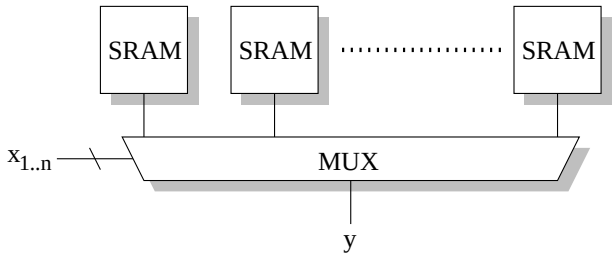


Abbildung 2.25: Realisierung einer Look-Up-Table

Die Abbildung 2.25 zeigt den Aufbau einer Look-Up-Table. Damit ergeben sich für den Platzbedarf einer Look-Up-Table mit n Eingängen die folgenden Kosten:

$$\begin{aligned} A_{n\text{-LUT}} &= 2^n \cdot A_{\text{SRAM}} + A_{2^n-\text{MUX}} \\ &= 2^n \cdot \frac{3}{2} + 2^n - 1 + \frac{\text{muxinv}(2^n)}{2} + \frac{\lceil \log_2 2^n \rceil}{2} \\ &= \left(\frac{3}{2} + 1\right) \cdot 2^n - 1 + \frac{\text{muxinv}(2^n) + n}{2} \\ &= 5 \cdot 2^{n-1} - 1 + \frac{\text{muxinv}(2^n) + n}{2} \end{aligned}$$

Die folgende Tabelle listet wiederum entsprechend berechnete Werte auf:

n	2	3	4	5	6
A_{MUX}	5.5	10	24.5	41	97.5
A_{SRAM}	6	12	24	48	96
$A_{n\text{-LUT}}$	11.5	22	48.5	89	193.5

Als dritte und letzte Komponente betrachten wir noch die CMOS-Realisierung eines flankengesteuerten Daten-Latches, wie sie in Abbildung 2.26 dargestellt ist.

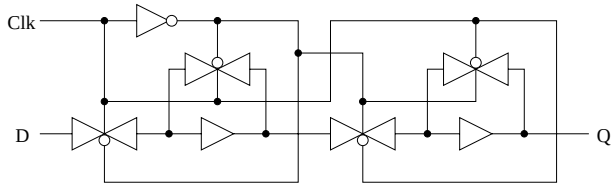


Abbildung 2.26: CMOS-Realisierung eines D-Flipflops

Die Bestandteile dieser Master/Slave-Konstruktion sind vier Transfergatter, zwei Treiber und ein Inverter. Nimmt man vereinfachend an, daß das *Clock*-Signal stets in bei-

den Phasen verfügbar ist, so beläuft sich der Platzbedarf insgesamt auf 12 Transistoren oder in Gatteräquivalenten:

$$A_{\text{LATCH}} = 6$$

Mit den vorangegangenen Kostendefinitionen der Komponenten konfigurierbarer Zellen können nun die Kosten (der *Platzverbrauch*) einer Zelle selbst bestimmt werden:

Definition 2.27 (Kosten einer konfigurierbaren Zelle)
Sei $P = (\mathcal{E}, \mathcal{T}, \delta)$ eine konfigurierbare Zelle mit KMN-Einbettung $\mathcal{E} = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{F}, \mathcal{L}, \eta_1, \eta_2)$. Dann sind die *Kosten* von P definiert wie folgt:

$$A_P = \sum_{F \in \mathcal{F}} A_{\#I_F\text{-LUT}} \; + \; \#\mathcal{L} \cdot A_{\text{LATCH}} \; + \sum_{M \in M(\mathcal{N}_1) \cup M(\mathcal{N}_2)} A_{\#I_M\text{-CMUX}}$$

Für die Beispiel-Zelle aus Abbildung 2.14 ergeben sich damit die Kosten:

$$\begin{aligned} A_{\text{CSR0202V001}} &= A_{\text{2-LUT}} + A_{\text{LATCH}} + A_{\text{2-CMUX}} + 2 \cdot A_{\text{4-CMUX}} \\ &= 11.5 + 6 + 3 + 2 \cdot 8.5 \\ &= 11.5 + 6 + 3 + 17 \\ &= 37.5 \end{aligned}$$

Da nach unserem Architekturmodell die Verdrahtungsressourcen bereits vollständig in den konfigurierbaren Zellen durch Multiplexer realisiert sind, können wir auf deren weitere Berücksichtigung verzichten und das Kostenmodell leicht auf Architektursegmente und Architekturen erweitern:

Definition 2.28 (Kosten einer Architektur)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann sind die *Kosten* von $\mathcal{A}$ definiert wie folgt:

$$A_{\mathcal{A}} = \sum_{v \in V} A_{\varphi(v)}$$

Ist zusätzlich $x \in \mathbb{N}^r$ ein Dilatationsvektor, so sind die *Kosten* der Architektur $\mathcal{A}^x$ definiert als:

$$\begin{aligned} A_{\mathcal{A}^x} &= \sum_{i=1}^{r} x_i \cdot A_{\mathcal{A}} \\ &= ||x||_1 \cdot A_{\mathcal{A}} \end{aligned}$$

Für ein Segment der Beispiel-Architektur aus Abbildung 2.15 ergeben sich die Kosten:

$$\begin{aligned} A_{\text{SM2V001S}} &= 4 \cdot A_{\text{CSR0202V001}} \\ &= 4 \cdot 37.5 \\ &= 150 \end{aligned}$$

Delay

In Abschnitt 2.3.2 wurde bereits ein taskorientiertes Delaymodell eingeführt, dem nun ein entsprechendes, konkrete Werte lieferndes Hardwaremodell zugrunde gelegt werden soll. Bei der Bestimmung eines solchen Hardwaremodells für die Signalverzögerung auf Architekturen ergeben sich, im Gegensatz etwa zu den oben betrachteten Kosten von Architekturen, im Prinzip zwei verschiedene Fragestellungen hinsichtlich der Zielsetzung einer Bewertung: Einerseits kann ein solches Modell versuchen, die Signalverzögerung anhand einer konkreten Architektur-Realisierung zu charakterisieren. Ein anderer Standpunkt betrachtet dagegen eine Bewertung der topologischen Eigenschaften von Architekturen und stellt vielmehr Fragen nach der Zahl genutzter Ressourcen, genauer: der Abbildung kritischer Pfade von Schaltkreisen. Auf dem Hintergrund dieser Überlegungen wollen wir für das Delay zwei Hardwaremodelle einführen:

Zellenmodell

Beim Zellenmodell steht die Frage im Mittelpunkt, wie sehr Pfade des Schaltkreises durch ihre Abbildung auf eine feldprogrammierbare Architektur „gedehnt“, um welchen Faktor sie quasi verlängert werden, wenn von einer Gleichgewichtung aller Netze und Funktionen im Schaltkreis ausgegangen wird. Hierzu definieren wir das Delay jedes Tasks t einer konfigurierbaren Zelle als:

$$\delta(t) = 1.0$$

Anschaulich betrachtet, zählen wir also die Anzahl konfigurierbarer Zellen, welche ein Pfad einer Implementierung eines kombinatorischen Schaltkreispfades durchläuft.

Tiefenmodell

Mit diesem Modell wird hingegen eine Abschätzung der Tiefe von Schaltkreis-Implementierungen angestrebt, wobei die Tiefe in Transistoren, wiederum normiert auf ein NAND2-Gatter[7] ausgedrückt wird. In Abschnitt 2.5.1 wurde nun bereits erläutert, daß in der vorliegenden Arbeit Bewertungen von Architekturen anhand von technologieunabhängig synthetisierten Schaltkreisen vorgenommen werden. Insofern kann also davon ausgegangen werden, daß sich insbesondere innerhalb eines Architekturtypus unterschiedliche Implementierungen eines Schaltkreises unmittelbar im Delay niederschlagen.

Für das Delay eines einfachen 2:1-Multiplexers setzen wir also an:

$$D_{\text{2:1-MUX}} = 0.5$$

Bei einer Kaskadierung von Multiplexern zu einem n:1-Multiplexer der Tiefe k, wie sie in Abbildung 2.22 dargestellt wurde, beträgt das Delay aufgrund der zusätzlichen Inverterstufen damit 1.0 pro Multiplexer und wir erhalten:

$$D_{n\text{-MUX}} = k \cdot 1.0 = \lceil \log_2 n \rceil$$

Für eine Look-Up-Table mit n Eingängen fällt hinsichtlich der SRAM-Zellen kein Delay an, womit verbleibt:

$$D_{n\text{-LUT}} = D_{2^n\text{-MUX}} = \log_2 2^n = n$$

[7] Tiefe: 2 Transistoren

Daten-Latches schließlich, bilden zugleich Quellen *und* Senken eines Schaltkreises und verursachen somit kein Delay.

Wie Definition 2.14 angibt, sind Tasks einer konfigurierbaren Zelle über Routing-Tasks der beiden KMN der Zelle erklärt. Ein solcher Routing-Task stellt gemäß Definition 2.6 jedoch nichts anderes dar, als eine Menge von Routen durch das KMN. Um nun das Delay eines Zelltasks bestimmen zu können, legen wir zunächst noch den Begriff des Delays von Routen auf KMN fest:

Definition 2.29 (Delay einer Route auf einem KMN)
Sei $\mathcal{N} = (V, E)$ ein KMN und $R = (v_0, e_0 \dots v_r, e_r, v_{r+1})$ eine Route auf $\mathcal{N}$ mit $v_i \in V$ und $e_i \in E$, wobei gilt: $v_0 \in I(\mathcal{N})$, $v_{r+1} \in O(\mathcal{N})$ und für $i = 1 \dots r$ ist $v_i \in M(\mathcal{N})$. Dann ist das *Delay* von R definiert als:

$$D_R = \sum_{i=1}^{r} D_{\#I_{v_i}\text{-MUX}}$$

Mit Hilfe des Delays von Routen lassen sich nun einfach die Delays von Tasks auf konfiguierbaren Zellen festlegen. Als Delay eines Tasks definieren wir die obere Schranke der Delays aller seiner möglichen Realisierungen:

Definition 2.30 (Delay eines Routing-Tasks)
Sei $P = (\mathcal{E}, \mathcal{T}, \delta)$ eine konfigurierbare Zelle und $T_{xy} \in \mathcal{T}$ ein R-Task auf P. Dann ist das *Delay* von T_{xy} definiert als:

$$\delta(T_{xy}) = \max_{(T_1,T_2)\in T_{xy}} \Big(\max_{R_1 \in T_1} D_{R_1} + \max_{R_2 \in T_2} D_{R_2} \Big)$$

Definition 2.31 (Delay eines Logik-Tasks)
Sei $P = (\mathcal{E}, \mathcal{T}, \delta)$ eine konfigurierbare Zelle mit $\mathcal{E} = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{F}, \mathcal{L}, \eta_1, \eta_2)$ und $T_F \in \mathcal{T}$ ein L-Task auf P für ein $F \in \mathcal{F}$. Dann ist das *Delay* von T_F definiert als:

$$\delta(T_F) = \delta(D_{\#I_F\text{-LUT}}) + \max_{(\mathcal{T}_1,\mathcal{T}_2)\in T_F} \Big(\max_{T_1 \in \mathcal{T}_1} \max_{R_1 \in T_1} D_{R_1} + \max_{R_2 \in T_2} D_{R_2} \Big)$$

Die Performanz einer Architektur zu bestimmen, gestaltet sich weitaus schwieriger, da Delays inhärent von der Implementierung eines Schaltkreises auf der Architektur abhängen. Somit können nur obere bzw. untere Schranken für die Charakterisierung der Performanz einer Architektur bestimmt werden, wie beispielsweise eine minimale bzw. maximale Zykluszeit von Implementierungen. Während die Schranke der minimalen Zykluszeit jedoch nur für bestimmte Schaltkreis-Implementierungen gelten kann, liefert hingegen die maximale Zykluszeit über alle auf der Architektur konfigurierbaren kombinatorischen Pfade eine sichere untere Performanzschranke. Andererseits kann eine solche, anhand eines Architekturausschnittes bestimmte untere Schranke nicht auf größere Dilatationen bezogen werden, eine obere Schranke hingegen schon, da die kritischen Pfade auch unter einer höheren Dilatation natürlich nicht kürzer werden können. Als Performanz einer Architektur definieren wir daher ihre anhand der Architekturbeschreibung ermittelte (theoretische) minimale Zykluszeit einer Implementierung.

Definition 2.32 (Kombinatorischer Pfad einer Architektur)
Sei $\mathcal{A}^x$ eine Architektur mit Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$. Dann ist ein *kombinatorischer Pfad* p auf $\mathcal{A}^x$ gegeben durch eine Folge von Tripeln

$$p = \Big((W_1, v_1, T_1), (W_2, v_2, T_2), \ldots, (W_n, v_n, T_n)\Big)$$

wobei $W_1 \ldots W_n$ kompatible K-Pfade und $T_1 \ldots T_n$ kompatible Logik-Tasks mit $T_i \in \mathcal{T}_{\varphi(v_i)}$ sind, die zu den W_i kompatibel sind und für alle $i = 1 \ldots n-1$ gilt: $Z^{\text{CELL}}(W_i) = v_i = Q^{\text{CELL}}(W_{i+1})$, $Z^{\text{PIN}}(W_i) \in I_{T_{i+1}}$, $Q^{\text{PIN}}(W_{i+1}) = o_{T_i}$ und $v_n \in V \cup \{v_{\text{nil}}\}$, wobei $T_n = \emptyset$, falls $v_n = v_{\text{nil}}$.

Wir schränken die Definition der kombinatorischen Pfade auf Architekturbeschreibungen $\mathcal{A}$ ein, indem wir für Beginn und Ende eines Pfades fordern, daß er entweder von einem Speicher-Task oder einer Peripherie-Kante, also $e \in E$ mit $\tau(e) \neq \vec{0}$, ausgeht bzw. dort endet.

Definition 2.33 (Performanz einer Architektur)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Sei $P_{\mathcal{A}}$ die Menge aller kombinatorischen Pfade auf $\mathcal{A}$. Dann ist das *Delay* der durch $\mathcal{A}$ beschriebenen Architektur definiert als:

$$D_{\mathcal{A}}^{\min} = \min_{\substack{\Big((W_1,v_1,T_1),\ldots,(W_n,v_n,T_n)\Big) \in P_{\mathcal{A}} \\ n \geq 2}} \quad \sum_{i=1}^{n} \delta(T_i) + \delta(W_i)$$

Die Definitionen dieses Abschnittes lieferten insbesondere Kapazität beschreibende Flächenmetriken für Architekturkomponenten, sowie Delaymetriken für Tasks. In Abschnitt 2.5.3 wird betrachtet werden, wie diese Metriken zur Bewertung von Schaltkreis-Implementierungen eingesetzt werden. Zuvor werden jedoch noch einige implementierungsunabhängige Bewertungsmaße im folgenden Abschnitt vorgestellt, welche vielmehr die *Flexibilität* von Architekturen zu charakterisieren versuchen.

Flexibilitätsmaße

Unter dem Begriff der *Flexibilität* einer feldprogrammierbaren Architektur verstehen wir im wesentlichen die Frage, wie „gut“ die Ressourcen der Architektur zur Implementierung von Schaltkreisen prinzipiell nutzbar sind. In einer früheren Arbeit des Verfassers [79] wurde zwischen Logik- und Routingflexibilität differenziert und eine Charakterisierung unter Berücksichtigung der tatsächlich realisierbaren Funktionen und Verbindungen vorgeschlagen. Insbesondere wurde hier die exakte Anzahl der in Kaskadierungen von Look-Up-Tables realisierbaren Funktionen berücksichtigt, jedoch unter Abzug der durch Kopplung und Konstantsetzung beliebiger Eingänge erhaltenen Duplikate. Mit diesen exakten Flexibilitätsmaßen wurden sodann verschiedene Architekturen bewertet, wobei jedoch der Berechnung der Maße aufgrund ihrer Komplexität Grenzen gesetzt waren.

Nachfolgend sollen stattdessen Metriken eingeführt werden, die zwar unschärfer, als die erwähnten Flexibilitätsmaße sind, jedoch ebenfalls helfen sollen, bereits anhand

der Struktur einer Architektur bestimmte Eigenschaften durch Werte zu charakterisieren. Vier der Metriken betrachten jeweils ein Segment der Architektur, das heißt die auf eine Dilatation von eins abgerollte Architekturbeschreibung, und normieren die Eigenschaften auf die Fläche des Segmentes. Wir sprechen deshalb auch von *Dichte-Metriken*.

Die erste Dichte-Metrik stellt die Frage, wieviele Ressourcen die Architektur zur Realisierung von Schaltkreisfunktionen bereitstellt. Eine alleinige Berücksichtigung der Anzahl der Look-Up-Tables würde hierbei jedoch grobgranulare Architekturen benachteiligen. Somit akkumuliert die Metrik stattdessen die Zahl K^{LUT} der Funktionsbits aller Look-Up-Tables des Architektursegmentes:

Definition 2.34 (Logikdichte)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann ist die *Logikdichte* von $\mathcal{A}$ definiert als:

$$S_{\mathcal{A}}^{\mathrm{logic}} = \frac{\sum_{v \in V} K_{\varphi(v)}^{\mathrm{LUT}}}{A_{\mathcal{A}}} = \frac{\sum_{v \in V} \sum_{f \in F_{\varphi(v)}} 2^{\#I_f}}{A_{\mathcal{A}}}$$

Mit der Zahl K_c^{KMN} der Konfigurationsbits aller KMN eines Zelltyps $c \in \mathcal{P}$ definieren wir analog:

Definition 2.35 (Verdrahtungsdichte)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann ist die *Verdrahtungsdichte* von $\mathcal{A}$ definiert als:

$$S_{\mathcal{A}}^{\mathrm{route}} = \frac{\sum_{v \in V} K_{\varphi(v)}^{\mathrm{KMN}}}{A_{\mathcal{A}}}$$

Die „Gesamtmenge“ an Information, welche zur Konfiguration einer Schaltkreis-Implementierung vorhanden ist, stellt nun die Zahl der Konfigurationsbits der Architektur dar, welche sich aus den Funktionsbits der Look-Up-Tables und den Konfigurationsbits der KMN zusammensetzen. Wir erhalten die dritte Dichte-Metrik somit durch die folgende Definition:

Definition 2.36 (Informationsdichte)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann ist die *Informationsdichte* von $\mathcal{A}$ definiert als:

$$S_{\mathcal{A}}^{\mathrm{info}} = S_{\mathcal{A}}^{\mathrm{logic}} + S_{\mathcal{A}}^{\mathrm{route}}$$

Setzt man hingegen Logik- und Verdrahtungsdichte zueinander ins Verhältnis, erhält man eine Charakterisierung der Balance zwischen den Logik- und den Verdrahtungsressourcen einer Architektur:

Definition 2.37 (Ressourcen-Balance)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann ist die *Ressourcen-Balance* von $\mathcal{A}$ definiert durch:

$$B_{\mathcal{A}}^{\mathrm{res}} = \frac{S_{\mathcal{A}}^{\mathrm{logic}}}{S_{\mathcal{A}}^{\mathrm{route}}} = \frac{\sum_{v \in V} K_{\varphi(v)}^{\mathrm{LUT}}}{\sum_{v \in V} K_{\varphi(v)}^{\mathrm{KMN}}}$$

Die vierte und letzte Dichte-Metrik, welche betrachtet werden soll, stellt die Frage, wieviele Funktionsberechnungen unter minimaler Zykluszeit (vgl. Definition 2.33) durchführbar sind, wobei wiederum auf die Fläche des betrachteten Architekturausschnittes normiert werden soll. Die Metrik, wir bezeichnen sie als *Berechnungsdichte* einer Architektur, stellt also ein Raum/Zeit-Maß dar, mit dem eine obere Schranke der Leistungsfähigkeit, wenn man Leistung als Verhältnis zwischen Arbeit und Zeit definiert, in Form einer flächennormierten Performanz wiedergegeben wird. Wie bei der Logikdichte gewichten wir die Look-Up-Tables wieder mit der Zahl ihrer Konfigurationsbits.

Definition 2.38 (Berechnungsdichte)
Sei $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ eine Architekturbeschreibung. Dann ist die *Berechnungsdichte* von $\mathcal{A}$ definiert durch:

$$S_{\mathcal{A}}^{\text{comp}} = \frac{\sum_{v \in V} K_{\varphi(v)}^{\text{LUT}}}{A_{\mathcal{A}} \cdot D_{\mathcal{A}}^{\min}}$$

2.5.3 Metriken für Schaltkreis-Implementierungen

Wir betrachten im folgenden nun einen beliebigen sequentiellen LUT-Schaltkreis $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ mit Knotenpartitionen $V_{\mathcal{C}} = I \dot\cup O \dot\cup F \dot\cup L$, sowie eine feldprogrammierbare Architektur $\mathcal{A}^x$ mit Beschreibung $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$.

Fläche

In die Berechnung der Kosten einer Schaltkreis-Implementierung $\mathcal{I} = (K, \rho^{\text{FUNC}}, \rho^{\text{NET}})$ werden jene programmierbaren Zellen einbezogen, in denen eine Funktion oder ein Latch des Schaltkreises realisiert wird oder die einen Teil der Route eines Netzes des Schaltkreises realisieren. Wir sprechen einfach von der Menge $U_{\mathcal{I}}$ der durch $\mathcal{I}$ *genutzten* programmierbaren Zellen:

$$\begin{aligned} U_{\mathcal{I}} \;=\; & \left\{ \rho^{\text{FUNC}}(f) \mid f \in F \cup L \right\} \cup \\ & \left\{ Q(e_i) \mid i \in \{2, \ldots, n\} \text{ wobei } (e_1, \ldots, e_n) \in \rho^{\text{NET}}(e) \text{ mit } e \in E_{\mathcal{C}} \right\} \end{aligned}$$

Die *(tatsächlichen)* Kosten einer Schaltkreis-Implementierung erklären sich nun einfach über die Summation der Kosten der genutzten programmierbaren Zellen:

Definition 2.39 (Kosten einer Schaltkreis-Implementierung)
Die *Kosten* einer Schaltkreis-Implementierung $\mathcal{I}$ definieren sich als:

$$A_{\mathcal{I}} = \sum_{v \in U_{\mathcal{I}}} A_{\varphi(v)}$$

Eine Betrachtung lediglich dieser Kosten gerät jedoch insofern problematisch, als in einer realen feldprogrammierbaren Architektur natürlich nicht nur die durch den Schaltkreis genutzten Zellen präsent sind, sondern auch jene Zellen, die durch gegebenenfalls ungünstige Plazierung nicht nutzbar sind. Deshalb wird ferner eine Metrik eingeführt, welche die Zellen in der Umgebung einer Implementierung mitberücksichtigt: die sogenannten *Bounding-Box-Kosten*:

Definition 2.40 (Bounding-Box-Kosten einer Implementierung)
Die *Bounding-Box-Kosten* einer Schaltkreis-Implementierung $\mathcal{I}$ definieren sich als:

$$A_{\mathcal{I}}^{\square} = \min_{\substack{y \in \mathbb{N}^r \\ U_{\mathcal{I}} \subseteq V^y}} A_{\mathcal{A}^y}$$

Die ins Verhältnis zu den tatsächlichen Kosten gesetzten Bounding-Box-Kosten einer Implementierung werden später als Charakterisierung der Ressourcenausnutzung eingesetzt.

Delay

Das Delay $D_{\mathcal{I}}$ einer Schaltkreis-Implementierung $\mathcal{I}$ orientiert sich bei feldprogrammierbaren Architekturen im Gegensatz etwa zu ASICs nicht an den längsten Pfaden des Schaltkreises in seiner jeweiligen synthetisierten Struktur, sondern an den durch Plazierung und Verdrahtung der Schaltkreis-Komponenten entstandenen längsten Wegen auf der Zielarchitektur.

Die Berechnungsmethode eines solchen maximalen Delays einer Schaltkreis-Implementierung auf einer Architektur zeigt Algorithmus 2.6. Es wird, beginnend mit den Quellen des Schaltkreises, eine an Schaltkreisknoten orientierte Breitensuche durchgeführt, wobei die Task-Delays der hierbei in der Implementierung durchlaufenen konfigurierbaren Zellen zu akkumulieren sind.

Wie aus den Definitionen 2.30 und 2.31 hervorgeht, ermittelt sich das Delay eines Tasks im Prinzip zwar aus dem maximalen Delay aller seiner möglichen Realisierungen in einer gegebenen konfigurierbaren Zelle, jedoch kommt dieser Umstand in obigem Algorithmus nicht zum tragen, da die Task-Delays in Schaltkreisknoten bei Implementierungen stets eindeutig sind. Denn ist, wie in einer Schaltkreis-Implementierung der Fall, das Ausgangsnetz eines Tasks verdrahtet, so liegt auch der entsprechende Ausgang fest, wodurch nach der eingeführten taskorientierten Spezifikation einer Zelle ein eindeutiger Delaywert existiert.

Algorithmus 2.6 (Maximales Delay einer Implementierung)

Initialisierung:
    Sequentieller LUT-Schaltkreis $\mathcal{C}$;
    ArchitekturGraph $\mathcal{G}$;
    Implementierung $(K, \rho^{\text{FUNC}}, \rho^{\text{NET}})$ von $\mathcal{C}$ auf $\mathcal{G}$;
    Liste $Q = \{\, v \in V_{\mathcal{C}} \mid v$ Quelle in $\mathcal{C} \,\}$;
    forall $(v \in V_{\mathcal{C}})$
        D[v] = 0;
    DelayValue D_{max} = 0; // Maximales Delay

while $(Q \neq \emptyset)$
{
    CircNode v = Q.pop( );
    CircNet N = $\mathcal{C}$.get_net(v);
    CTree B = $\rho^{\text{NET}}(N)$;

```
        forall ( w ∈ succ(v) );
        {
            AGCell c = ρ^FUNC(w);
            AGEdge e = c.get_inedge(N);
            Task t = G.get_task(c,w);

            DelayValue D_route = B.get_delay(e);
            DelayValue D_task = δ_c(t);
            DelayValue D_new = D[v] + D_route + D_task;

            if (D_new > D[w])
            {
                D[w] = D_new;
                if (w Senke von C)
                    if (D[w] > D_max)
                        D_max = D[w]
                else
                    Q.append(w);
            }
        }
    }
```

Die Laufzeit von Algorithmus 2.6 ist von der Ordnung $\mathcal{O}(\#V_{\mathcal{C}})$, da der durchmusterte Schaltkreis $\mathcal{C}$ durch Auftrennung seiner sequentiellen Schleifen zu einem DAG wird, was insbesondere zur Konsequenz hat, daß jeder Schaltkreisknoten lediglich einmal in die FIFO-Liste Q eingefügt wird. Ferner werden bei jeder Herausnahme eines Knotens dessen Nachfolger betrachtet, jedoch ebenfalls jeder Knoten höchstens einmal. Die übrigen Operationen sind aufgrund entsprechender Datenstrukturen in konstanter Zeit berechenbar.

Ideal-Architektur

Ein Vergleich von Architekturen durch Implementierung von Benchmark-Schaltkreisen erfolgt mittels Flächen- und Delaywerten, anhand denen gewisse *relative* Eigenschaften, also eher verhältnismäßige Tendenzen zwischen den einzelnen Architekturen erklärt werden können. Interessant scheint nun jedoch auch die Frage, inwieweit eine Architektur auf die Struktur eines Schaltkreises „einzugehen“ vermag, welche Konsequenzen im Hinblick auf den *Overhead einer Implementierung* zu ziehen sind. Beobachtungen dieser Art erfordern aber ein über die zu vergleichenden Architekturen konstantes, jedoch schaltkreisspezifisches Maß. Wie bereits erläutert, wird hinsichtlich der Benchmark-Schaltkreise eine von der Zielarchitektur unabhängige Synthese und Optimierung vorgenommen. Die aus seiner initialen Struktur im Zuge der Synthese entstandene implementierbare Form eines Schaltkreises liefert aber offenbar eine derartige „Konstante“, indem durch ein geeignetes Modell einer *Ideal-Architektur* mittels denselben, in den vorigen Abschnitten vorgestellten Bewertungsmetriken, nunmehr auch eine *absolute* Einschätzung des Overheads einer gegebenen Architektur möglich wird. Ein solches Modell soll nun im folgenden motiviert werden.

Zunächst liegt auf der Hand, daß eine Schaltkreis-Implementierung mittels Look-Up-Tables im *worst case* natürlich exponentiellen Overhead verursachen kann, da beispielsweise selbst konstante Funktionen über Look-Up-Tables realisiert werden müssen. Neben diesem, als durchaus überzogen zu bezeichnenden Implementierungsoverhead einer *einzelnen* Schaltkreisfunktion, welchen man lediglich aus Gründen der Universalität eingeht, zeigte auch bereits Satz 2.4, wie sich selbst beim Splitten Boolescher Funktionen die Anzahl benötigter Look-Up-Tables in exponentiellen Größenordnungen bewegen kann. Da jedoch alle Funktionen eines Schaltkreises zu implementieren sind, müssen die Kosten, sowie das Delay der erforderlichen Look-Up-Tables unweigerlich in die Bewertung einer Ideal-Architektur einfließen.

Ein anderes Bild zeigt sich hingegen beim Verdrahtungsaufwand. Da im allgemeinen die Zahl der Eingänge einer Schaltkreisfunktion stets durch eine Konstante beschränkt ist, kann die Entwicklung der Anzahl der Terminale eines Schaltkreises als lediglich linear in der Anzahl seiner Netze konstatiert werden. Damit wäre aber auf den ersten Blick auch der Verdrahtungsaufwand nur ein linearer Faktor und zumindest für die Flächenbewertung irrelevant. Dennoch müssen wir ihn bei der Bestimmung einer schaltkreisspezifischen Ideal-Architektur berücksichtigen – und zwar in zweifacher Hinsicht: zum einen, da wir trivialerweise stets von endlichen und auch hinsichtlich der Größenordnung beschränkten Schaltkreisen ausgehen, also kein asymptotisches Verhalten der Verdrahtungskosten erwarten können und zum anderen, da die Verdrahtung insbesondere bei Architekturen mit stark eingeschränkter Verdrahtungsflexibilität massiv die Entwicklung des Logikoverheads zu beeinflussen im Stande ist. Als Maß für den Verdrahtungsaufwand eines Schaltkreises gehen wir unter Berücksichtigung des eingeführten Architekturmodells mit routingintegrierenden Basisblöcken davon aus, daß jedes Netz mittels einfacher 2:1-Multiplexer verdrahtbar ist, wobei für jedes Zielterminal Fläche und Delay eines Multiplexers kalkuliert wird:

Definition 2.41 (Kosten einer Ideal-Architektur)
Sei $\mathcal{C} = (V, E)$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionierung $V = I \dot\cup O \dot\cup F \dot\cup L$. Dann sind die *Kosten einer Ideal-Architektur* zu $\mathcal{C}$ definiert als:

$$\hat{A}_{\mathcal{C}} = \sum_{v \in F} A_{deg^-(v)\text{-LUT}} + \#L \cdot A_{\text{LATCH}} + \sum_{(v^+, V^-) \in E} \#V^- \cdot A_{\text{2-CMUX}}$$

Definition 2.42 (Delay einer Ideal-Architektur)
Sei $\mathcal{C} = (V, E)$ ein sequentieller Look-Up-Table-Schaltkreis und $P_{\mathcal{C}}$ die Menge aller kombinatorischen Pfade in $\mathcal{C}$. Dann ist das *Delay einer Ideal-Architektur* zu $\mathcal{C}$ definiert als:

$$\hat{D}_{\mathcal{C}} = \max_{(v_0, e_1, v_1, \ldots, e_{n-1}, v_{n-1}) \in P_{\mathcal{C}}} (n-1) \cdot D_{\text{2-MUX}} + \sum_{i=1}^{n-2} D_{deg^-(v_i)\text{-LUT}}$$

Offensichtlich bilden diese Maße einer Ideal-Architektur sogar *untere Schranken* für die entsprechenden Kosten- und Delay-Metriken bezüglich *jeder* Implementierung eines Schaltkreises:

$$\forall_{\mathcal{I}_{\mathcal{C}}} \quad \hat{A}_{\mathcal{C}} \leq A_{\mathcal{I}_{\mathcal{C}}}$$

Hinsichtlich der Delaymetrik ist diese Aussage jedoch auf nichttriviale Architekturen, genauer: Architekturen mit konfigurierbaren Verdrahtungsressourcen, einzuschränken, denn bei auf leeren KMN basierenden Logiktasks könnte die Schranke noch unterschritten werden. Ansonsten gilt aber:

$$\forall_{\mathcal{I}_\mathcal{C}} \quad \hat{D}_\mathcal{C} \leq D_{\mathcal{I}_\mathcal{C}}$$

2.5.4 Relative Metriken

Während im vorangegangenen Abschnitt ausschließlich *absolute Metriken* für Architekturen und Schaltkreis-Implementierungen vorgestellt wurden, werden in diesem letzen Abschnitt des Kapitels hingegen einige *relative Metriken* betrachtet, welche später schließlich auch zur Anwendung kommen sollen. Relative Metriken erhält man einfacherweise, indem absolute Metriken zueinander ins Verhältnis gesetzt werden, um Beziehungen zwischen diesen auszudrücken.

Ein interessanter Aspekt hinsichtlich der Bewertung der Kosten und des Delays von Schaltkreis-Implementierungen wurde im vorigen Abschnitt bereits angesprochen: die Frage, wie weit eine Implementierung von einem (zumindest theoretischen) Optimum entfernt ist. Das vorgestellte Konzept der „Ideal-Architektur“ liefert jedoch gerade einen solchen notwendigen Maßstab, so daß wir den *Overhead* für Kosten und Delay wie folgt definieren können:

Definition 2.43 (Kostenoverhead)
Sei $\mathcal{I}$ die Implementierung eines sequentiellen Look-Up-Table-Schaltkreises $\mathcal{C}$ auf einer Architektur $\mathcal{A}^x$. Dann ist der *Kostenoverhead* von $\mathcal{I}$ definiert als:

$$A_\mathcal{I}^{\text{ov}} = \frac{A_\mathcal{I}}{\hat{A}_\mathcal{C}}$$

Definition 2.44 (Delayoverhead)
Sei $\mathcal{I}$ die Implementierung eines sequentiellen Look-Up-Table-Schaltkreises $\mathcal{C}$ auf einer Architektur $\mathcal{A}^x$. Dann ist der *Delayoverhead* von $\mathcal{I}$ definiert als:

$$D_\mathcal{I}^{\text{ov}} = \frac{D_\mathcal{I}}{\hat{D}_\mathcal{C}}$$

Die Kostenoverhead-Metrik ermöglicht nun zwar eine Einschätzung, inwieweit mehr Ressourcen der Architektur benötigt wurden, als dies „theoretisch nötig“ wäre, doch könnte eine Implementierung mit geringem Kostenoverhead auch eine relativ große Dilatation besitzen und lokale Ressourcen nur schlecht ausnutzen, wodurch übermäßig andere Ressourcen blockiert werden. Wir definieren deshalb das Maß der *Ausnutzung* einer Architektur:

Definition 2.45 (Architektur-Ausnutzung)
Sei $\mathcal{I}$ die Implementierung eines sequentiellen Look-Up-Table-Schaltkreises $\mathcal{C}$ auf einer Architektur $\mathcal{A}^x$. Dann ist die *Ausnutzung* von $\mathcal{A}^x$ durch $\mathcal{I}$ definiert als:

$$A_\mathcal{I}^{\text{use}} = \frac{A_\mathcal{I}}{A_\mathcal{I}^{\square}}$$

Die in diesem Kapitel vorgestellten Metriken zur Bewertung feldprogrammierbarer Architekturen, sowohl in theoretischer (architekturorientierter) Hinsicht, als auch mittels empirischer (schaltkreisorientierter) Kenngrößen, liefern ein angemessenes Instrumentarium, um Tendenzen bezüglich Kapazität, Flexibilität und Performanz von Architekturen charakterisieren zu können, sowie diese in Vergleichen einander gegenüberzustellen. Wir schließen damit also das Kapitel zur Modellierung feldprogrammierbarer Architekturen ab und wenden uns nun der Entwicklung eines generischen Layout-Werkzeuges für Schaltkreise zu, mit dessen Hilfe der empirische Anteil der Untersuchung von Architekturen zu leisten sein wird.

Kapitel 3

Makrogenerierung

In diesem Kapitel wird der erste Teil des im Rahmen dieser Arbeit entwickelten retargierbaren Layout-Werkzeugs für feldprogrammierbare Architekturen vorgestellt. Dabei werden zwei neue, interagierende Algorithmen zur Partitionierung, sowie der Plazierung und Verdrahtung von Partitionen sequentieller Look-Up-Table-Schaltkreise eingeführt, wobei versucht wird, der Retargierbarkeit durch einen neuen Ansatz, dem *Plazieren durch Verdrahten (Route-then-Place)*, Rechnung zu tragen. Zielsetzung des Verfahrens ist die Generierung von Schaltkreis-Makros für ein nachfolgendes Floorplanning und Endverdrahten.

Nach einer Einführung in die vorliegende Problemstellung wird zunächst das Schaltkreis-Partitionierungsverfahren, sowie dessen Adaption für das Gesamtverfahren beleuchtet, bevor schließlich auf die Methode zur Plazierung- und Verdrahtung einzelner Knoten und Netze eingegangen wird.

3.1 Problembetrachtung

Als *Makros* werden üblicherweise Implementierungen bestimmter, immer wieder verwendeter Subschaltkreise bezeichnet, welche zumeist in einer Bibliothek zusammengefaßt sind. Man unterscheidet prinzipiell zwischen Hardmakros und Softmakros. Bei *Softmakros* ist neben einer Liste zielarchitekturspezifisch partitionierter Funktionsblökke des Schaltkreises lediglich eine Netzliste abgelegt, welche die Verbindungsstruktur zwischen den Blöcken beschreibt. Für *Hardmakros* hingegen existieren zusätzlich relative Zuordnungen der Blöcke zu Architekturressourcen, sowie zu jedem Netz eine konkrete Routenimplementierung.

Wir wollen hingegen den Begriff des Makros auf disjunkte Zerlegungen von Schaltkreisen verallgemeinern, jedoch auch zugleich für sequentielle LUT-Schaltkreise spezialisieren und seine Bedeutung auf den Begriff des Hardmakros einschränken. Unter dem Begriff *Makrogenerierung* verstehen wir demnach die Zerlegung eines LUT-Schaltkreises, der durch eine Netzliste gegeben ist, in disjunkte Subschaltkreise, welche jeweils einzeln auf der Zielarchitektur unter Minimierung von Kostenfunktionen implementiert, also plaziert und verdrahtet werden, jedoch vermöge der repetitiven Struktur der Zielarchitektur relokierbar bleiben.

3.1.1 Zur Frage der Komplexität

Eine Generierung von Makros umfaßt die folgenden Probleme:

1. Schaltkreispartitionierung
2. Plazierung
3. Verdrahtung

Abstrahiert betrachtet, handelt es sich hinsichtlich der **Schaltkreispartitionierung** im allgemeinen um eine *Multipartitionierung* (Multiway Partition Problem), das heißt um eine Zerlegung in mehrere Subschaltkreise. In allgemeiner Form stellt sich das Problem, welches selbst unter trivialen Bedingungen NP-hart ist, wie folgt dar [50]:

> **Definition 3.1 (Multipartitionierungsproblem)**
> Sei $G = (V, E)$ ein Hypergraph mit n Knoten, Knotengewichten $w : V \to \mathbb{N}$, Kantengewichten $c : E \to \mathbb{N}$, Anzahl $r \in \mathbb{N}$ der Partitionen, obere $B(i) \in \mathbb{N}$ und untere $b(i) \in \mathbb{N}$ Schranken für die Größe jeder Partition $i = 1, \ldots, r$. Dann besteht das *Multipartitionierungsproblem* in einer Zerlegung der Knotenmenge V in r Teilmengen $P = (V_1, \ldots, V_r)$ mit $V_i \subset V$, $b(i) \leq \sum_{v \in V_i} w(v) \leq B(i)$, $V_i \neq \emptyset$, $V_i \cap V_j = \emptyset$, für $i, j \in \{1, \ldots, r\}$ und $\bigcup_{i=1}^{r} V_i = V$, wobei $c(P) = \frac{1}{2} \cdot \sum_{i=1}^{r} \sum_{e \in E_i^{\text{ext}}} c(e)$ minimal, wenn $E_i^{\text{ext}} = \{e \in E \mid \rho(e) \cap V_i \neq \emptyset \neq \rho(e) \cap \complement V_i\}$

Beim **Plazierungsproblem** geht es darum, eine gültige Implementierung der Schaltkreisknoten auf der gegebenen Zielarchitektur (vgl. Definition 2.24) zu finden, also eine Zuordnung der Look-Up-Table-Knoten auf Logik-Tasks und der Latch-Knoten auf Speicher-Tasks, welche die Delays der aus den implementierenden Tasks und den zu erwartenden Netzrouten sich zusammensetzenden kritischen Pfade minimiert. Betrachtet man den Spezialfall einer beliebigen eindimensionalen Architektur, die aus lediglich einer Kette konfigurierbarer Zellen mit Verdrahtungsressourcen zu den jeweiligen Nachbarzellen besteht, so ergibt sich beispielsweise unter der Zellenmodell-Delaymetrik, wie sie in Abschnitt 2.5.2 eingeführt wurde, unter der Kostenfunktion der gewichteten Summe der Delays, bezüglich der Block/Zellen-Zuordnung offensichtlich bereits ein lineares Anordnungsproblem, dessen NP-Härte bekannt ist [50]:

> **Definition 3.2 (Optimales Lineares Anordnungsproblem)**
> Sei $G = (V, E)$ ein Hypergraph mit Kantenkosten $c : E \to \mathbb{R}^+$ und $m \in \mathbb{N}$ mit $m \geq \#V$. Dann besteht das *Optimale Lineare Anordnungsproblem* in einer Suche nach einer injektiven Abbildung $p : V \to \{1, \ldots, m\}$, so daß die folgende Kostenfunktion minimiert wird: $c(p) = \sum_{e \in E} c(e) \cdot c(e, p)$, wobei $c(e, p) = \max_{v,w \in \rho(e)} |p(v) - p(w)|$.

Sind nun die Terminale der Netze festgelegt, wird im **Verdrahtungsschritt** eine delayminimale Implementierung der Netze (vgl. Definition 2.25) gesucht. Man unterscheidet hierbei zwischen globaler und detaillierter Verdrahtung. Eine *globale Verdrahtung* oder *Routenplanung* liefert zu jedem Netz eine Liste geometrischer Punkte der Zielarchitektur, über die eine tatsächliche Route des jeweiligen Netzes verlaufen könnte. Die *detaillierte Verdrahtung* liefert hingegen zu jedem Netz eine Sequenz von Verdrahtungsressourcen, welche bereits die konkrete Implementierung einer Route darstellt. Dabei darf

jede Verdrahtungsressource natürlich von höchstens einer Route benutzt werden. Doch selbst ohne diese Einschränkung stellt sich auch das Verdrahtungsproblem als im Prinzip nicht traktabel dar. Denn während Routen für Zweipunktnetze noch mittels relativ einfacher Kürzester-Wege-Suchen in Polynomialzeit zu berechnen sind, handelt es sich im Falle von Mehrpunktnetzen um sogenannte *Steinerbaum-Probleme* (Minimum Steiner Tree Problem), die im Falle allgemeiner Graphen ebenfalls NP-hart sind [50]:

Definition 3.3 (Minimales Steinerbaum-Problem)
Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit Kantenkosten $c : E \to \mathbb{R}^+$ und $U \subseteq V$ eine Teilmenge der Knotenmenge. Dann besteht das *Minimale Steinerbaum-Problem* in einer Suche nach einem Subgraphen H von G, der alle Knoten aus U verbindet und dessen Blätter ausschließlich Knoten aus U sind, wobei die Kostenfunktion $c(H) = \sum_{e \in E_H} c(e)$ minimiert wird.

Zu den drei genannten Problemkreisen kommt hier noch die Frage der Generizität bezüglich unterschiedlicher Architekturen hinzu. Bei der Entwicklung generischer Verfahren können somit keinerlei Annahmen über die Struktur des der Architektur zugrundeliegenden Graphen gemacht werden. Selbst die ursprüngliche Periodizität des Architekturgraphen geht mit der ersten Belegung von Architekturressourcen im Prinzip verloren und kann somit nicht weiter beim Entwurf spezieller Algorithmen berücksichtigt werden.

In den beiden folgenden Unterabschnitten werden nun die Teilprobleme für den vorliegenden Fall formal definiert, ferner werden kurzgefaßte Übersichten zu bereits in der Vergangenheit publizierten Lösungsansätzen gegeben.

3.1.2 Schaltkreispartitionierung

Die Art der Schaltkreispartitionierungsprobleme, um die es im folgenden geht, stellt genauer ein Partitionierungsproblem von Netzlisten dar. Die Netzliste eines sequentiellen LUT-Schaltkreises wird repräsentiert durch einen gerichteten Hypergraphen mit Knoten, welche Primäreingänge, -ausgänge, Look-Up-Tables und Latches des Schaltkreises darstellen. Von dieser Knotenmenge ist lediglich die Teilmenge der Look-Up-Tables und Latches repräsentierenden Knoten zu partitionieren, so daß der Gegenstand unseres Ausgangsproblems die folgende Form besitzt:

Definition 3.4 (Partitionierung eines LUT-Schaltkreises)
Sei $\mathcal{C} = (V, E)$ ein sequentieller LUT-Schaltkreis mit den Knotenpartitionen $V = I \dot\cup O \dot\cup F \dot\cup L$. Dann ist eine *Partitionierung* von $\mathcal{C}$ gegeben durch eine totale, surjektive Abbildung

$$\chi : (F \cup L) \to \{1, 2, \ldots, n_\chi\}$$

wobei $n_\chi \in \mathbb{N}$ die Anzahl der Partitionen darstellt.

Für $i \in \{1, \ldots, n_\chi\}$ bezeichnen wir mit $P_i^\chi = \{v \in F \cup L \mid \chi(v) = i\}$ die Menge der Knoten in der Partition i. Insgesamt stellt sich unser Problem nun wie folgt dar:

Definition 3.5 (Partitionierungsproblem für LUT-Schaltkreise)
Das *Partitionierungsproblem für sequentielle LUT-Schaltkreise* ist gegeben durch einen sequentiellen LUT-Schaltkreis $\mathcal{C} = (V, E)$, eine Kostenfunktion $c : \chi \mapsto \mathbb{R}$, sowie eine Kapazitätsschranke $B \in \mathbb{N}$. Gesucht ist eine Partitionierung χ von $\mathcal{C}$ mit $c(\chi)$ minimal, wobei gilt:

$$\forall_{i \in \{1,\ldots,n_\chi\}} \#P_i^\chi \leq B$$

Auf die Kostenfunktion $c(\chi)$ werden wir später noch eingehen. Zur Lösung solcher im allgemeinen Fall intraktablen Probleme wurde in der Vergangenheit eine Vielzahl unterschiedlichster Ansätze entwickelt. Eine gute Übersicht hierzu gibt beispielsweise [6]. Generell lassen sich die Lösungsmethoden in drei Kategorien unterteilen:

- **Methoden basierend auf der Bewegung von Knoten**
  Hierzu gehören Algorithmen, die im Lösungsraum die Konfigurationen durch die Bewegung von Knoten zwischen den Partitionen bewertend durchforsten. Die Problematik dabei besteht im wesentlichen im Verbleib des Verfahrens in lokalen Minima. Greedy-Methoden wie Iteratives Verbessern, Simulated Annealing, aber auch Genetische Algorithmen sind – im weiteren Sinne – Beispiele dieser Kategorie [39, 28, 40, 77].

- **Mittels geometrischen Repräsentationen arbeitende Methoden**
  Diese Verfahren konstruieren aus den Partitionierungsproblemen geometrische Probleme und versuchen dabei, auch globale strukturelle Informationen über das ursprüngliche Problem einzubringen, um schließlich effizientere Verfahren für die geometrischen Instanzen anzuwenden. Beispiele: Lineares Anordnen, Quadratic Placement [5].

- **Methoden der kombinatorischen Optimierung**
  Eine Zurückführung auf klassische Optimierungsprobleme, wie Flüsse in Netzwerken, Integer Programming oder Mengenüberdeckungen versuchen Lösungsansätze dieser Kategorie [54].

Hinsichtlich der Vorgehensweise bei der Bildung der Partitionen lassen sich ferner *Bottom-Up*- und *Top-Down*-Verfahren unterscheiden. Klassische Bottom-Up-Verfahren stellen die sogenannten *Clustering*-Methoden dar, bei denen die Partitionen durch sukzessives Kombinieren einzelner Blöcke gebildet werden.

Für das dieser Arbeit zugrundeliegende generische Layoutsystem für feldprogrammierbare Architekturen wurde ein inkrementelles Clustering-Verfahren entwickelt. Dabei wird der zu partitionierende Schaltkreis als *Energiesystem* betrachtet, welches, an Abläufen in der Natur sich orientierend, beginnend auf einer Stufe hoher Energie, sukzessive in Zustände niedriger Energie überführt wird. Anschaulich betrachtet, wird die Menge der Schaltkreisknoten, genauer: der Look-Up-Tables und Latches, als ein großes Schaumgebilde von vielen kleinen Bläschen interpretiert, welche sich im Hinblick auf das Energielevel ihrer Oberflächenspannungen nach und nach zu größeren Blasen vereinigen. Diesem Pendant in der Natur entsprechend, wurde das Verfahren von uns auch als *Bubble Partitioning Method* bezeichnet [82]. Es wird in Abschnitt 3.2 vorgestellt.

3.1.3 Plazieren und Verdrahten

Das Problem des Plazierens und Verdrahtens von Schaltkreisen im klassischen Sinne, stellt, abstrakt betrachtet, eine Einbettung eines Hypergraphen in ein orthogonales Gitter (*mesh*) dar. In der Literatur wird dafür häufig auch der Begriff des *Layouts* verwendet [50]. *Kramer* und *van Leeuwen* zeigten Anfang der 1980er Jahre, daß das Layout-Problem NP-hart ist [45], so daß in der Praxis auch hier nur der Einsatz suboptimaler Methoden zur Disposition steht. Im Falle konfigurierbarer Architekturen wurden nun im wesentlichen ursprünglich für *Full Custom* und *Semi Custom Devices* entwickelte Verfahren aufgegriffen und adaptiert, obgleich das Layout-Problem bei konfigurierbaren Architekturen viel schwerer ist, da ihre Verdrahtungsressourcen weitaus stärker eingeschränkt sind, indem sie im allgemeinen beispielsweise nicht mehr als einfacher *Mesh*-Graph darstellbar sind.

Beim Layout-Problem von Schaltkreisen können verschiedene Kostenfunktionen betrachtet werden. Während am häufigsten Platzverbrauch und maximale Verdrahtungslänge bzw. resultierendes Delay minimiert werden, stehen beim Design von *Full Custom* und *Semi Custom Devices* auch die Anzahl der Leitungsüberkreuzungen (*Crossings*), der Layerwechsel (*Vias*) oder der Leitungsbögen (*Jogs*) in Betrachtung.

Da aber bei konfigurierbaren Architekturen die Logik- und Verdrahtungsressourcen fest vorgegeben sind, der Hauptanteil des Delays einer Schaltkreis-Implementierung im allgemeinen von der Verdrahtung herrührt, wobei der Platzverbrauch auch wesentlich durch die Verdrahtung bestimmt wird [17], formulieren wir das Layout-Problem für sequentielle Look-Up-Table-Schaltkreise zwar nur unter der Minimierung des resultierenden maximalen Delays (*Timing-Driven Layout*), werden in den Untersuchungen jedoch auch stets die Ressourcenkosten beobachten.

> **Definition 3.6 (Layout-Problem für LUT-Schaltkreise)**
> Sei $\mathcal{C}$ ein sequentieller Look-Up-Table-Schaltkreis und $\mathcal{A}^x$ eine feldprogrammierbare Architektur. Dann besteht das *Layout-Problem* für $\mathcal{C}$ bezüglich $\mathcal{A}^x$ in der Bestimmung einer gültigen Implementierung $\mathcal{I}$ von $\mathcal{C}$ auf $\mathcal{A}^x$, so daß das maximale Delay $D_{\mathcal{I}}$ minimiert wird.

Zur Lösung von Layout-Problemen wurden in der Vergangenheit die unterschiedlichsten Ansätze entwickelt. Eine gerade im ASIC-Bereich häufig angewandte Heuristik besteht in der rekursiven Bipartitionierung des Schaltkreises. Hierbei werden für die Partitionen separate Layouts berechnet und durch Verdrahten der Netze zwischen den Partitionen die Subschaltkreise zu einem Gesamtlayout zusammengefügt. Ein Beispiel hierzu im FPGA-Bereich stellt das *FPR*-System von *Alexander* und *Cohoon* [3] dar, welches ein dem Standardzellen-Layout entlehntes rekursives Partitionierungsverfahren (*Thumbnail Partitioning*) mit einem globalen Router kombiniert, während detailliertes Verdrahten durch den bekannten Steinerbaum-Algorithmus von *Kou*, *Markowsky* und *Berman* [43] realisiert wird. Als Weiterentwicklungen der auf Partitionierung basierten Methoden, kann man hier die späteren *hierarchischen Layoutverfahren* bezeichnen [47, 75], welche im Gegensatz Top-Down-Ansätze verfolgen. Doch selbst die ursprünglich von Quinn [63, 70] für das Platinenlayout entwickelte Methode des *Force-Directed Arrangements* wurde für den diskreten Fall des FPGA-Plazierungsproblems adaptiert [71].

Eines der frühesten Layout-Werkzeuge für FPGAs stellt das CGE/SEGA-System der *University of Toronto* dar [17, 49]. Auch hier wird ein von *Breuer* im Jahre 1977 vorgeschlagenes, auf Partitionierung basierendes Plazierungsverfahren verwendet [15]. Ein globaler Verdrahter plant die in Zwei-Punkt-Verbindungen gesplitteten Netze als sogenannte *Coarse Graphs*, während der nachgeschaltete detaillierte Verdrahter die *Coarse Graphs* durch Allokation von Folgen entsprechender Verdrahtungskanal-Segmente expandiert. Entsprechend der Kritizität der Netze erfolgt nun eine sukzessive Auswahl expandierter *Coarse Graphs*, wobei jeweils nicht-kompatible Routen eliminiert werden.

Einen völlig anderen Ansatz zur Verdrahtung von FPGAs der bekannten Inselstruktur verfolgen *Wu* und *Marek-Sadowska* mit ihrem *GBP*-Algorithmus [86], der das Verdrahtungsproblem als zweidimensionales Intervallpackungsproblem reformuliert und mittels Greedy-Methode die Netze in die vorgegebenen Verdrahtungskanäle packt.

Modernere Ansätze stellen hingegen die Systeme *FRONTIER* und *VPR* dar. Beim *FRONTIER*-System [74] handelt es sich jedoch lediglich um ein Plazierungsverfahren, während in den publizierten Experimenten zur Verdrahtung ein kommerzielles Softwarewerkzeug herangezogen wurde. *FRONTIER* geht zunächst von einer Unterteilung der Architektur in gleichgroße Regionen (*Bins*) aus. Nach einem Bottom-Up-Clustering der Schaltkreisknoten, beziehungsweise vorhandener Makros, entsprechend der Größe der *Bins*, werden die Partitionen den *Bins* mittels *Simulated Annealing* zugeordnet. Prognostiziert eine Abschätzung der Anordnung die Nichtverdrahtbarkeit, so wird anschließend nochmals ein *Simulated Annealing* Verfahren niedrigerer Temperatur zur Perturbierung der erhaltenen Plazierung ausgeführt. *VPR* [13, 12] plaziert einen Schaltkreis durch *Simulated Annealing* auf Basis einer Kostenfunktion, welche die Ausbreitung jedes Netzes ins Verhältnis zur Anzahl der in seiner *Bounding Box* durchschnittlich verfügbaren Tracks setzt (*Linear Congestion*). Der anschließende Verdrahtungsschritt wird durch die *PathFinder*-Methode [55] von *McMurchie* und *Ebeling*, ein Maze-Router, realisiert. Wir werden später auf die *PathFinder*-Methode zurückkommen.

Zusammenfassend bleibt festzustellen, daß, neben vielen neu vorgestellten Verfahren, auch die meisten der bislang publizierten CAD-Systeme für FPGAs die bekannte, kommerzielle, zweidimensionale Architektur mit Inselstruktur voraussetzen und Layout-Verfahren beinhalten, die auf *Simulated Annealing* und klassischen *Maze-Routing*-Strategien basieren.

In dieser Arbeit wollen wir hingegen, wie bereits angekündigt, einen interaktiven Ansatz einer Partitionierungs- und einer Layout-Methode betrachten. Während sich der nachfolgende Abschnitt 3.2 zunächst mit einem neuen Partitionierungsverfahren auseinandersetzt, beschäftigt sich Abschnitt 3.3 mit einer ebenfalls neuen, generischen Layoutmethode, sowie der Verknüpfung der beiden Verfahren.

3.2 Bubble Partitioning Verfahren

3.2.1 Motivation

Neben den in Abschnitt 3.1.2 erwähnten allgemeinen Verfahren zur Partitionierung der Knotenmenge von Graphen wurden in der Vergangenheit auch spezielle Verfahren zur Partitionierung von Schaltkreisen hinsichtlich deren Implementierung auf feldpro-

grammierbaren Architekturen entwickelt [46, 52, 53]. In den meisten der Publikationen wird dabei eine Partitionierung nicht zum Zwecke der Generierung von Makros, sondern für Multi-FPGA-Implementierungen betrachtet, wobei als Zielarchitekturen in der Regel kommerzielle FPGAs zugrundegelegt werden. Es wurden jedoch auch Verfahren vorgestellt, die Partitionierungen für hierarchisch konzipierte Zielarchitekturen bestimmen.

Unabhängig des letztendlichen Zweckes, berücksichtigen die bekannten Verfahren im wesentlichen nur die Struktur des zu implementierenden Schaltkreises, lassen einfache, auf Gatter- oder Logikzellen basierte Kapazitätsschranken einfließen und schließen die Plazierung und Verdrahtung der Schaltkreiskomponenten erst nach vollständiger Lösung des Partitionierungsproblemes an. Dies hat seinen Grund darin, daß Verdrahtungsaufwand und Delay, sowie die Verdrahtbarkeit der Komponenten der resultierenden Partitionen überhaupt während des Partitionierungsverfahrens schwer zu prognostizieren sind, insbesondere dann, wenn das Architekturmodell mehrere unterschiedliche Logikblocktypen zuläßt. Diese Problematik ist jedoch vielen Optimierungsproblemen zueigen, denn häufig ist in der Tat nicht beweisbar, daß mittels einer gewählten Kostenfunktion tatsächlich das erwünschte Optimum treffbar ist. Im vorliegenden Fall kann sich dies eben beispielsweise in der Nichtimplementierbarkeit einer kostenoptimalen Lösung des Partitionierungsproblems artikulieren.

In der vorliegenden Arbeit wurde stattdessen ein anderer Ansatz versucht, der bereits in Abschnitt 1.8.2 skizziert wurde. Hiernach wird ein Schaltkreis durch Partitionierung zerlegt und aus den einzelnen Partitionen werden Makros konstruiert, welche später mittels Floorplanning auf der gegebenen Zielarchitektur angeordnet werden. Indem die für gewöhnlich getrennten Schritte Partitionierung und Plazierung sowie Verdrahtung ineinander verzahnt werden, handelt es sich also um ein inkrementelles Partitionierungsverfahren, welches die Partitionen im Wechsel mit der sukzessiven Implementierung der Schaltkreiskomponenten ermittelt. Durch diese Verzahnung wird nun ein nicht nur mehr *schaltkreissensitives*, sondern auch ein *architektursensitives Partitionieren* ermöglicht.

In diesem Abschnitt 3.2 soll jedoch das entwickelte Partitionierungsverfahren noch in weitgehend architekturunabhängiger Form vorgestellt werden, während der Abschnitt 3.3 eine modifizierte Form des Verfahrens in Kombination mit einem Plazierungs- und Verdrahtungsverfahren bringt.

3.2.2 Kostenfunktion

Zur Definition geeigneter Kostenfunktionen spielen bei Partitionierungsverfahren immer die gegenseitigen Beziehungen der aktuell konstruierten Partitionen eine Rolle. Wir betrachten dazu den *Schnitt* einer Partitionierung:

Definition 3.7 (Schnitt einer Partitionierung)
Sei $\mathcal{C} = (V, E)$ ein sequentieller LUT-Schaltkreis mit den Knotenpartitionen $V = I \dot\cup O \dot\cup F \dot\cup L$. Ist $\chi : (F \cup L) \to \mathbb{N}$ eine Partitionierung von $\mathcal{C}$, dann ist der *Schnitt* Cut_χ von χ definiert als:

$$Cut_\chi = \{e \in E \mid \exists_{\substack{i,j \in \{1,\ldots,n_\chi\} \\ i \neq j}} \rho(e) \cap P_i \neq \emptyset \text{ und } \rho(e) \cap P_j \neq \emptyset\}$$

Wir bezeichnen Netze auf dem Schnitt einer Partitionierung auch als *Intercluster-Netze* und Netze der Komplementmenge hiervon als *Intracluster-Netze*. Das grundsätzliche Verständnis beim Clustering von Schaltkreisen besteht nun darin, daß Intercluster-Netze aufwendiger zu verdrahten sind, ihre Implementierung in der Regel also neben einer erhöhten Anzahl an Ressourcen auch in einem höheren Delay resultiert, als dies bei Intracluster-Netzen der Fall ist.

Relative Delay Criticality

Wie im vorigen Unterabschnitt bereits angedeutet, wird das tatsächliche Delay eines über einen Schaltkreisknoten verlaufenden Pfades erst im Zuge dessen Implementierung auf einer feldprogrammierbaren Architektur offenbar. In streng getrennten Entwurfsphasen ist ein solches Delay damit nicht eher bestimmbar, bis die Schaltkreisknoten plaziert und verdrahtet sind. Geht man davon aus, daß das Delay einer Schaltkreis-Implementierung auf feldprogrammierbaren Architekturen im wesentlichen durch die Verdrahtung bestimmt wird, so stellt ein schlupfbasiertes Kritizitätsmaß für die Netze ein adäquates Mittel dar zur Charakterisierung der kritischen Pfade anhand der *Struktur* eines Schaltkreises. Wir bezeichnen dieses Kritizitätsmaß als *relative Delay-Kritizität (Relative Delay Criticality,* RDC*)*.

Sei $\mathcal{C} = (V, E)$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionierung $V = I \dot\cup O \dot\cup F \dot\cup L$. Zu einem Knoten $v \in F \cup L$ sei $D_i(v)$ die maximale Anzahl der durchlaufenen Netze auf einem kombinatorischen Pfad von einer beliebigen Quelle von $\mathcal{C}$ zum Knoten v. Analog sei $D_o(v)$ die maximale Anzahl der durchlaufenen Netze auf einem kombinatorischen Pfad vom Knoten v zu einer beliebigen Senke von $\mathcal{C}$. Offensichtlich sind die Werte D_i bzw. D_o durch eine Vorwärts- bzw. eine Rückwärts-Breitensuche auf dem Schaltkreis, jeweils ausgehend von den Quellen bzw. Senken des Schaltkreises berechenbar: Indem initial für alle Knoten $v \in V$ gilt $D_i(v) = D_o(v) = 0$, ist während der Schaltkreistraversen lediglich darauf zu achten, daß die Werte D_i und D_o nicht über Latch-Knoten hinweg propagiert werden. Algorithmus 3.1 skizziert das entsprechende Verfahren am Beispiel der Bestimmung der Werte D_i.

Algorithmus 3.1 (Bestimmung der Werte D_i)

```
forall v ∈ V do
    D_i(v) = 0;                                   // Initialisierung
D_max = 0;

Queue Q;
forall v ∈ I ∪ L do
    Q.append(v);

while Q ≠ ∅ do
{
    v = Q.pop();
    forall w ∈ succ(v) do                         // Betrachte alle Nachfolger von v
    {
        propagate = false;
        if v ∈ L then                             // v ist Latch-Knoten
```

```
            if D_i(w) < 1 then                    // Latch-Knoten noch nicht betrachtet
            {
                D_i(w) = 1;
                propagate = true;                 // Propagiere D_i ein Mal
            }
        else                                      // v ist kein Latch-Knoten
            if D_i(w) < D_i(v) + 1 then
            {
                D_i(w) = D_i(v) + 1;              // Update von D_i, falls erforderlich
                propagate = true;
            }
        if propagate then                         // D_i ist über w zu propagieren
        {
            if D_max < D_i(w) then                // Maximumsbildung
                D_max = D_i(w);
            if w ∉ O ∪ L then                     // w ist keine Senke von C
                Q.append(w);
        }
    }
}
```

Während der Traversen wird ferner das Maximum über die entsprechenden Werte aller Knoten gebildet:

$$D_{\max} = \max_{v \in V} \{D_i(v), D_o(v)\}$$

Unter der Annahme, daß jeder implementierte Funktionsknoten ein Delay von konstant Eins verursacht, charakterisiert nun der *Schlupf* eines Netzes, welche Verzögerung ein Signal über das Netz höchstens besitzen darf, damit der Schaltkreis noch „maximalen Durchsatz“ liefert. Wir definieren somit den Schlupf eines Netzes $e \in E$ als:

$$\text{netslack}(e) = D_{\max} - D_i(\mathit{drv}(e)) - \max_{w \in \mathit{tgts}(e)} D_o(w) - 1$$

Netze, die auf einem kritischen Pfad des Schaltkreises liegen, besitzen also den Schlupf Null. Hinsichtlich der relativen Delay-Kritizität normieren wir die Schlupfwerte auf das oben bestimmte maximale Delay des Schaltkreises und konstruieren das Maß dergestalt, daß höhere Werte eine höhere Kritizität des Netzes bedeuten:

$$\begin{aligned} \mathit{RDC}(e) &= 1 - \frac{\text{netslack}(e)}{D_{\max}} \\ &= \frac{D_i(\mathit{drv}(e)) + \max_{w \in \mathit{tgts}(e)} D_o(w) + 1}{D_{\max}} \end{aligned}$$

Es ist $\mathit{RDC}(e) \in]0;1]$ für jedes Netz $e \in E$, wobei $\mathit{RDC}(e) = 1$ genau dann, wenn ein kritischer Pfad des Schaltkreises über e verläuft. Der Fall $\mathit{RDC}(e) = 0$ kann hingegen nicht auftreten, denn es müßte $\mathit{netslack} = D_{\max}$ gelten, jedoch werden keine Schaltkreise mit direkt zwischen Quellen und Senken des Schaltkreises verlaufenden Netzen betrachtet.

Kosten einer Knotenverschiebung

Wie fließt nun die relative Delay-Kritizität in die Kostenfunktion unseres Partitionierungsverfahrens ein? Wir betrachten dazu eine Partitionierung $\chi : (F \cup L) \to \mathbb{N}$ eines Schaltkreises $\mathcal{C} = (V, E)$ mit $V = I \uplus O \uplus F \uplus L$. Für ein Netz $e \in E$ bezeichnen wir die Menge der Partitionen, zu welchen e adjazent ist durch:

$$\mathit{ADJ}_\chi(e) = \{P_i^\chi \mid \exists_{u \in P_i^\chi}\ u = \mathit{drv}(e)\ \vee\ u \in \mathit{tgts}(e)\}$$

Wir definieren die Kosten der Partitionierung χ wie folgt:

$$c(\chi) = \sum_{e \in E} \#\mathit{ADJ}_\chi(e) \cdot \mathit{RDC}(e)$$

Betrachtet man nun, ausgehend von einer Partitionierung χ, die Verschiebung eines Knotens u aus einer Partition P_i in eine Partition P_j, so muß zur Minimierung der Kosten $c(\chi)$ ein Netz e existieren, das sowohl zu u, als auch zur Partition P_j adjazent ist. Denn ansonsten würde sich bei der Verschiebung von u die Kardinalität von $\mathit{ADJ}_\chi(e)$ erhöhen, womit auch die Kosten $c(\chi')$ der resultierenden Partitionierung χ' erhöht würden. Offensichtlich reduziert sich aber die Berechnung von $c(\chi')$ auf ein Update der „alten“ Kosten $c(\chi)$, indem lediglich die Mengen ADJ_χ zu verändern sind. Insgesamt ergeben sich hier die folgenden vier Fälle, welche in Abbildung 3.1 illustriert werden:

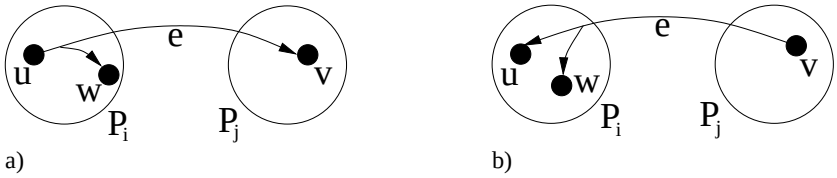


Abbildung 3.1: Szenarien bei Verschiebung des Knotens u

Sei $u = \mathit{drv}(e) \in P_i$ (vgl. Abbildung 3.1a).

1. Ist $\mathit{succ}(u) \cap P_i = \emptyset$, aber $\mathit{succ}(u) \cap P_j \neq \emptyset$, dann verringert sich die Kardinalität von $\mathit{ADJ}_\chi(e)$, da P_i aus der Menge fällt. Gilt hingegen $\mathit{succ}(u) \cap P_j = \emptyset$, so verändert sie sich nicht, da P_i zwar herausgenommen wird, aber P_j hinzukommt.

2. Ist $\mathit{succ}(u) \cap P_i \neq \emptyset$, aber $\mathit{succ}(u) \cap P_j = \emptyset$, dann erhöht sich die Kardinalität von $\mathit{ADJ}_\chi(e)$, da P_j hinzukommt. Gilt jedoch stattdessen $\mathit{succ}(u) \cap P_j \neq \emptyset$, so ändert sich die Kardinalität nicht.

Sei nun $u \in \mathit{tgts}(e) \cap P_i$ (vgl. Abbildung 3.1b).

3. Gilt $\rho(e) \cap P_i = \{u\}$ und $\rho(e) \cap P_j \neq \emptyset$, dann verringert sich die Kardinalität von $\mathit{ADJ}_\chi(e)$, da P_i aus der Menge heraus fällt. Gilt hingegen $\rho(e) \cap P_j = \emptyset$, so ändert sich die Menge nicht, da P_i aus $\mathit{ADJ}_\chi(e)$ genommen wird, aber P_j hinzukommt.

4. Gilt $\rho(e) \cap P_i = \{u, w_1, \ldots, w_k\}$ mit $k \geq 1$ und $\rho(e) \cap P_j = \emptyset$, dann erhöht sich die Kardinalität von $\mathit{ADJ}_\chi(e)$, da P_j zur Menge hinzukommt. Gilt hingegen $\rho(e) \cap P_j \neq \emptyset$, so ändert sich die Menge nicht.

Algorithmus 3.2 zeigt zusammenfassend die Berechnung des Kostenupdates bei einer Verschiebung des Schaltkreisknotens u von der Partition P_i in die Partition P_j. Der Aufwand zur Berechnung ergibt sich im wesentlichen aus Elementprüfungen der direkten Vorgänger von u und der Randknoten der in u einlaufenden Netze. Er beträgt somit $\mathcal{O}(\#V)$.

Algorithmus 3.2 (Kostenupdate beim Bubble Partitioning)

```
float calc_cost_change(u, i, j)
{
    float cost_change = 0.0;
    if (pred(u) ∪ succ(u)) ∩ P_j ≠ ∅ then                    // u adjazent zu P_j
    {
        Sei e Netz mit drv(e) = u;
        S_{u,i} = succ(u) ∩ P_i;
        S_{u,j} = succ(u) ∩ P_j;
        if S_{u,i} = ∅ und S_{u,j} ≠ ∅ then                  // Fall 1
            cost_change = cost_change - RDC(e);
        if S_{u,i} ≠ ∅ und S_{u,j} = ∅ then                  // Fall 2
            cost_change = cost_change + RDC(e);
        forall e mit u ∈ tgts(e) do
        {
            A_{u,i} = (ρ(e) \ {u}) ∩ P_i;
            A_{u,j} = ρ(e) ∩ P_j;
            if A_{u,i} ≠ ∅ und A_{u,j} ≠ ∅ then              // Fall 3
                cost_change = cost_change - RDC(e);
            if A_{u,i} = ∅ und A_{u,j} = ∅ then              // Fall 4
                cost_change = cost_change + RDC(e);
        }
    }
    return cost_change;
}
```

Verkürzung kritischer Pfade

Je nach Struktur des zu partitionierenden Schaltkreises kann es vorkommen, daß mehrere Knotenkandidaten existieren, deren Verschiebung jeweils dieselbe Kostenreduktion bewirken. Interessant scheinen jedoch insbesondere Verschiebungen von Knoten, die nicht nur die Zahl der Netze auf dem Schnitt der Partitionierung und damit die Länge der kritischen Pfade zu reduzieren suchen, sondern auch die Anzahl der kritischen Pfade auf dem Schnitt verringern.

Definition 3.8 (Kürzung eines kritischen Pfades)
Seien P_i und P_j zwei Partitionen und $u \in P_i$ ein Knoten. Dann *kürzt* die Verschiebung des Knotens u von P_i nach P_j einen kritischen Pfad genau dann, wenn mindestens eine der folgenden Bedingungen gilt:

(1) $RDC(drv^{-1}(u)) = 1$ und $\forall_{v \in succ(u)}\ v \in P_j$

(2) $\forall_{\substack{v \in pred(u),\\ RDC(drv^{-1}(v))=1}} \quad v \in P_j$

Zur Bestimmung der Anzahl der durch eine Verschiebung eines Knotens u gekürzten kritischen Pfade berechnen wir im Initialschritt des Partitionierungsverfahrens wiederum durch jeweils eine Vorwärts- bzw. Rückwärts-Breitensuche ab den Quellen bzw. Senken des Schaltkreises die Maße $icp(u)$ als die Gesamtzahl der kritischen Pfade von einer Quelle des Schaltkreises bis zum Knoten u und $ocp(u)$ die Gesamtzahl der kritischen Pfade vom Knoten u bis zu einer Senke des Schaltkreises. Algorithmus 3.3 skizziert das entsprechende Verfahren am Beispiel der Berechnung der Werte *icp*.

Algorithmus 3.3 (Berechnung der Werte icp)

```
forall v ∈ V do                                          // Initialisierung
    icp(v) = 0;

Queue Q;
forall v ∈ I ∪ L do
    Q.append(v);

while Q ≠ ∅ do
{
    v = Q.pop();
    if (RDC(drv^-1(v)) = 1) then                         // Netz auf kritischem Pfad
    {
        forall w ∈ succ(v) do                            // Betrachte alle Nachfolger von v
        {
            if v ∈ I ∪ L then                            // v ist Quelle von C
                icp(w) = icp(w) + 1;                     // Von v geht genau
                                                         // ein kritscher Pfad aus
            else
                icp(w) = icp(w) + icp(v);                // Alle kritischen Pfade durch v
                                                         // laufen über w weiter

            if w ∉ Q ∪ O ∪ L then
                Q.append(w);
        }
    }
}
```

Über die Gesamtzahl der kritischen Pfade, die über einen Schaltkreisknoten verlaufen, gibt nun das nachfolgende Lemma Auskunft:

Lemma 3.1 (Anzahl der über einen Knoten verlaufenden kritischen Pfade)
Sei $u \in V$ ein Knoten eines Schaltkreises $\mathcal{C} = (V, E)$. Dann beträgt die Zahl $cp(u)$ der über u laufenden kritischen Pfade gerade

$$cp(u) = icp(u) \cdot ocp(u)$$

Beweis: In u laufen insgesamt $icp(u)$ kritische Pfade ein, für jeden Nachfolger v von u ist die Zahl der aus v auslaufenden kritischen Pfade mit $ocp(v)$ bekannt. Jeder in u einlaufende kritische Pfad kann in u zu einem beliebigen

Nachfolger verzweigen. Über einen Nachfolger v laufen somit $\mathit{icp}(u) \cdot \mathit{ocp}(v)$ kritische Pfade. Aufsummiert über alle Nachfolger von u ergibt sich damit:

$$\begin{aligned} \mathit{cp}(u) &= \sum_{v \in \mathit{succ}(u)} \mathit{icp}(u) \cdot \mathit{ocp}(v) \\ &= \mathit{icp}(u) \cdot \sum_{v \in \mathit{succ}(u)} \mathit{ocp}(v) \\ &= \mathit{icp}(u) \cdot \mathit{ocp}(u) \end{aligned}$$

□

Vorteilhaft ist nun nicht etwa eine Verschiebung jener Knoten, über welche die *meisten* kritischen Pfade verlaufen, denn dadurch würde ungünstigerweise nur die Zahl der kritischen Netze auf dem Schnitt der Partitionierung angehoben. Vielmehr sollen solche Knoten verschoben werden, welche die meisten kritischen Pfade *verkürzen*.

Satz 3.1 (Kürzung kritischer Pfade durch Knotenverschiebung)
Eine Verschiebung des Knotens u von der Partition P_i in die Partition P_j führt zur Kürzung von $\mathit{CPS}(u)$ kritischen Pfaden *(Critical Path Shortage)*, wobei gilt:

$$\mathit{CPS}(u) = \mathit{icp}(u) \cdot \sum_{v \in \mathit{succ}(u) \cap P_j} \mathit{ocp}(v) + \mathit{ocp}(u) \cdot \sum_{v \in \mathit{pred}(u) \cap P_j} \mathit{icp}(v)$$

Beweis: Nach Lemma 3.1 beträgt die Zahl der kritischen Pfade, die über einen Knoten u verlaufen, gerade $\mathit{icp}(u) \cdot \mathit{ocp}(u)$. Bei der Summation der Werte *icp* und *ocp* über die Vorgänger und die Nachfolger von u sind jedoch lediglich jene zu berücksichtigen, die in P_j liegen, da nur die kritischen Pfade zwischen dem Knoten u und P_j durch die Verschiebung gekürzt werden. Wir betrachten unter diesem Aspekt die Fälle aus Definition 3.8:

(1) Die Zahl der gekürzten kritischen Pfade vom Knoten u zu seinen Nachfolgerknoten in P_j beträgt:

$$\mathit{CPS}_{\text{succ}}(u) = \mathit{icp}(u) \cdot \sum_{v \in \mathit{succ}(u) \cap P_j} \mathit{ocp}(v)$$

(2) Die Zahl der gekürzten kritischen Pfade von den Vorgängerknoten des Knotens u, welche in P_j liegen, zum Knoten u selbst beträgt:

$$\mathit{CPS}_{\text{pred}}(u) = \mathit{ocp}(u) \cdot \sum_{v \in \mathit{pred}(u) \cap P_j} \mathit{icp}(v)$$

Mit $\mathit{CPS}(u) = \mathit{CPS}_{\text{succ}}(u) + \mathit{CPS}_{\text{pred}}(u)$ ergibt sich die Behauptung. □

Diesem Satz beizumerken ist noch, daß mit dem Wert $\mathit{CPS}(u)$ solche kritische Pfade doppelt gezählt werden, die mit der Verschiebung des Knotens u zweimal gekürzt werden, das heißt sowohl als in u einlaufender als auch von u ausgehender Pfad. Im Hinblick auf die Verwendung von *CPS* in der Kostenfunktion des Partitionierungsverfahrens ist dies jedoch durchaus sinnvoll und wurde deshalb auch so implementiert.

Nachdem Problemstellung und Kostenfunktion motiviert und exakt definiert sind, geben wir nun im folgenden Abschnitt 3.2.3 das entwickelte Verfahren explizit an.

3.2.3 Algorithmus

Bei der *Bubble Partitionierung* handelt es sich um ein reines Greedy-Verfahren nach der Methode des „steilsten Abstiegs“. Das Verfahren ist in seiner Grundversion durch Algorithmus 3.4 dargestellt.

Algorithmus 3.4 (Bubble Partitioning Verfahren)

```
init_bubble( )
{
    forall u ∈ (F ∪ L) do
        Erzeuge eine Partition {u};
    Berechne RDC-Werte für alle Netze;
    Berechne die Werte icp und ocp für alle Knoten;
}

(u_best,P_src,P_dest) get_best_candidate( )
{
    best_candidate = (nil,nil,nil);          // (Knoten,Quell- und Zielpartition)
    best_reduction = 0.0;                    // Kostenreduktion
    best_shortage = 0;                       // CPS-Wert
    forall (P_i, P_j) mit i ≠ j, P_i ≠ ∅, P_j ≠ ∅, #P_j < B do
    {
        forall u ∈ P_i do
        {
            cost_reduction = calc_cost_change(u,i,j);
            cpath_shortage = CPS(u);
            if (cost_reduction < best_reduction)
                or ((cost_reduction = best_reduction) and
                    (cpath_shortage > best_shortage)) then
            {
                best_reduction = cost_reduction;
                best_shortage = cpath_shortage;
                best_candidate = (u,i,j);
            }
        }
    }
    return best_candidate;
}

bubble_partitioning( )
{
    init_bubble( );
    moved = true;
    while (moved) do          // Solange keine stabile Partitionierung erreicht
    {
        moved = false;
        (u, i, j) = get_best_candidate( );
        if (u, i, j) ≠ (nil,nil,nil) then          // Knoten-Kandidat gefunden
        {
            P_i = P_i \ {u};                       // Knoten u verschieben von P_i nach P_j
```

```
            P_j = P_j ∪ {u};
            moved = true;
        }
    }
}
```

Zunächst wird in einem Initialschritt **init_bubble** aus jedem Schaltkreisknoten eine Partition gebildet, sowie die für die Kostenfunktion relevanten Werte *RDC, icp* und *ocp* berechnet. Eine *while*-Schleife wird danach so oft wiederholt, bis die erreichte Partitionierung *stabil* ist. Innerhalb dieser Schleife wird der in der Methode **get_best_candidate** ermittelte Knoten u_{best} mit maximalem Verschiebungsgewinn, also einer maximalen Kostenreduktion, aus seiner Quellpartition P_{src} in die berechnete Zielpartition P_{dest} verschoben – unter Berücksichtigung der Kapazitätsschranke B. Die nach der Terminierung des Algorithmus nichtleeren Mengen P_i bilden eine Partitionierung der Menge $F \cup L$.

Definition 3.9 (Geschichte einer Partitionierung)
Sei $\mathcal{C} = (V, E)$ ein sequentieller Look-Up-Table-Schaltkreis mit Knotenpartitionen $V = I \dot\cup O \dot\cup F \dot\cup L$. Sei $F \cup L = \{v_1, \ldots, v_n\}$. Dann gilt für die *Initialpartitionierung* χ_0 zu $\mathcal{C}$ die Eigenschaft

$$\forall_{i \in \{1,\ldots,n\}}\ \chi_0(v_i) = i$$

Zu einer Partitionierung χ bezeichnen wir mit $\chi_{u,j}$ jene Folgepartitionierung, welche man aus χ durch Verschiebung des Knotens u in die Partition P_j erhält.

Wir können nun die Eigenschaft der *Stabilität* einer Partitionierung wie folgt definieren:

Definition 3.10 (Stabile Partitionierung)
Sei $\chi : (F \cup L) \to \mathbb{N}$ eine Partitionierung eines Schaltkreises $\mathcal{C} = (V, E)$ mit $V = I \dot\cup O \dot\cup F \dot\cup L$. Dann heißt χ *stabil* genau dann, wenn gilt:

$$\forall_{\substack{i,j \in \{1,\ldots,n_\chi\},\\ i \neq j}}\ \forall_{u \in P_i^\chi} \quad c(\chi_{u,j}) \geq c(\chi)$$

Das Verfahren terminiert also, wenn die Verschiebung keines Knotens mehr eine Verringerung der Kosten bringt.

Satz 3.2
Das in Algorithmus 3.4 beschriebene Verfahren terminiert nach endlich vielen Schritten.

Beweis: Für die Kosten der Initialpartitionierung χ_0 gilt:

$$c(\chi_0) = \sum_{e \in E} RDC(e)$$

Im Falle trivialer Kapazitätsschranken $B \geq \#V$ befinden sich unter einer optimalen Partitionierung χ_{opt} alle Netze innerhalb einer Partition. Somit gilt offensichtlich:

$$c(\chi_{\text{opt}}) = 0$$

Also sind die Kosten von Partitionierungen χ für jede Kapazitätsschranke B nach oben und nach unten beschränkt. Jede Reihe von Knotenverschiebungen liefert nach Konstruktion des Verfahrens aber eine monoton fallende Folge von Kosten der resultierenden Partitionierungen. Aufgrund dieser Monotonie und der Beschränktheit folgt also eine Konvergenz der Kosten. Da die Knotenmenge ferner endlich ist, existieren nur endlich viele Partitionierungen, also terminiert das Verfahren nach endlich vielen Schritten. □

Insofern das Abbruchkriterium von Algorithmus 3.4 in einer Bedingung über die Reduktion der Kosten der jeweils konstruierten Partitionierung lautet, scheint die Laufzeit des Verfahrens inhärent von den Größendifferenzen der *RDC*-Werte der einzelnen Netze abzuhängen, weshalb eine präzise Abschätzung der Laufzeit des Algorithmus nicht angegeben werden kann. Die Zahl $M_{\mathcal{C}}$ der Verschiebungen von Knoten kann jedoch sicher begrenzt werden durch die Anzahl der möglichen Verminderungen der Kosten der Initialpartitionierung unter minimalem Reduktionsbetrag:

$$M_{\mathcal{C}} \leq \left\lceil \frac{c(\chi_0)}{\min\limits_{u \in F \cup L} \min\limits_{\substack{E_1, E_2 \subseteq E_u, \\ E_1 \cap E_2 = \emptyset}} \sum\limits_{e \in E_1} RDC(e) - \sum\limits_{e \in E_2} RDC(e)} \right\rceil$$

Im Rahmen der späteren Verwendung des Bubble Partitioning Verfahrens werden wir jedoch eine Einschränkung des Problems treffen, welche die Laufzeit wieder exakt bestimmbar werden läßt.

3.2.4 Empirische Vergleiche

Im Gegensatz etwa zu einem Greedy-Verfahren, das die Partitionen sukzessive im Zuge einer Traverse durch den Schaltkreis bestimmt, konstruiert das *Bubble Partitioning* Verfahren die Partitionen simultan. In diesem Abschnitt werden einige empirische Resultate präsentiert, die einen Vergleich dreier Greedy-Methoden im Hinblick auf Laufzeit, kritische Netze auf dem Schnitt, Ausnutzung der Kapazität einer Partition, sowie der Zahl der erzeugten Partitionen veranschaulichen sollen.

Als weiteres, auf einer einfachen Schaltkreistraverse basierendes Greedy-Verfahren, welches wir Bubble Partitioning gegenüberstellen wollen, wurde das durch Algorithmus 3.5 skizzierte *Straight Partitioning* implementiert[1]. Hierbei werden die Partitionen sukzessive mit Schaltkreisknoten gefüllt bis die vorgegebene Kapazitätsgrenze B erreicht ist oder kein zur aktuell betrachteten Partition adjazenter Knoten mehr existiert oder alle Knoten in Partitionen untergebracht sind.

[1] Bekanntere Multipartitionierungsverfahren, wie beispielsweise die auf iterativem Verbessern basierende Heuristik von *Fiduccia* und *Mattheyses* [28] wurden unter Hinblick auf den Rahmen des späteren Einsatzes des Verfahrens nicht für einen Vergleich herangezogen. Eine genauere Erläuterung hierzu wird später gegeben.

Algorithmus 3.5 (*Straight Partitioning* Methode)

```
i = 1;
Queue Q = F ∪ L;
P_i = { Q.pop( ) };                                   // Beginne erste Partition
while Q ≠ ∅ do
{
    best_node = nil;
    best_gain = −∞;
    best_cps = 0;
    // Suche einen zu P_i adjazenten Knoten
    // mit höchstem Verschiebungsgewinn:
    forall v ∈ Q mit (pred(v) ∪ succ(v)) ∩ P_i ≠ ∅ do
    {
        gain = 0.0;
        forall e ∈ E_v do                  // Betrachte alle Netze, die v berühren
        {
            U_int = {w ∈ ρ(e) | w ∈ P_i};                  // Nachbarknoten in P_i
            U_ext = {w ∈ ρ(e) | w ∉ P_i};          // Nachbarknoten nicht in P_i
            gain = gain + RDC(e) · (#U_int − #U_ext);
        }
        if gain > best_gain then           // Knoten von höherem Gewinn exist.
        {
            best_node = v;
            best_gain = gain;
            best_cps = CPS(v);
        }
    }
    if best_node ≠ nil then                                // Bester Knoten gefunden
    {
        P_i = P_i ∪ {best_node};
        Q = Q \ {best_node};
    }
    if Q = ∅ then
        break;                                              // Verfahren terminiert
    // Keinen Knoten gefunden oder Kapazitätsgrenze für P_i erreicht
    if (best_node = nil) oder (#P_i = B) then
    {
        i = i + 1;                                         // Neue Partition beginnen
        P_i = { Q.pop( ) };
    }
}
```

Die Laufzeit des *Straight Partitioning* Algorithmus in seiner hier dargestellten Form ist $\mathcal{O}(\#V^2 \cdot \#E)$, allerdings aufgrund der Tatsache, daß im Verfahren lediglich unpartitio-

nierte Knoten durchmustert werden, sowie der in Schaltkreisen in der Regel stark eingeschränkten Adjazenzen, im Gegensatz zu allgemeinen Graphen, kann mit niedrigen Konstanten gerechnet werden.

Das naive, dafür jedoch relativ schnelle *Straight Partitioning* wurde auch mit dem laufzeitintensiveren *Bubble Partitioning* verknüpft, indem die mittels des *Straight*-Verfahrens erhaltenen Partitionierungen nunmehr als Initialpartitionierungen für das *Bubble*-Verfahren genutzt wurden. Wir bezeichnen diese Methode im folgenden als *Hybrid*-Verfahren.

Die Versuche wurden mit einer Auswahl von 24 verschiedenen Benchmark-Schaltkreisen aus [1] durchgeführt, wobei Schaltkreise mit bis zu 1000 Netzen betrachtet wurden. Hinsichtlich der Zahl der Schaltkreisknoten wurde jedoch auf einen repräsentativen Querschnitt geachtet. Die Schaltkreise wurden mittels des Technology Mappers von SIS in zwei Varianten bearbeitet: zum einen für Basisblöcke mit 2-Input-Look-Up-Tables (LUT2-Blöcke), zum anderen für Basisblöcke der kommerziellen XC4000-Architektur. Die Tabelle 3.1 zeigt die Liste der verwendeten Benchmark-Schaltkreise samt einigen Größendaten.

	unmapped		LUT2		XC4000	
Schaltkreis	$\#(F \cup L)$	$\#E$	$\#(F \cup L)$	$\#E$	$\#(F \cup L)$	$\#E$
5xp1	175	182	126	116	40	30
apex7	295	344	272	235	143	106
b12	778	793	97	88	47	38
c8	253	281	154	136	85	67
cm162a	87	101	48	43	31	26
comp	231	263	127	124	69	66
count	227	262	160	144	82	66
C432	362	398	209	202	100	93
C880	705	765	377	351	186	160
decod	43	48	49	33	37	21
ex6	132	138	100	92	43	35
inc	153	160	133	124	61	52
mult16a	274	292	211	210	77	76
pm1	104	120	70	57	47	34
s344	253	263	155	144	72	61
s510	202	222	264	257	118	111
s641	418	454	223	200	138	115
s820	259	278	278	259	131	112
s838	627	663	345	343	128	126
s953	690	707	424	401	207	159
styr	285	295	414	404	169	159
vg2	434	459	112	104	57	49
x1	451	502	346	311	192	157
x4	674	768	453	382	295	224
Avg.	338	364.92	213.46	198.33	106.46	90.33

Tabelle 3.1: Benchmark-Auswahl

Die Ergebnisse der empirischen Versuche sind nachfolgend in Diagrammen zusammengefaßt. Wir betrachten zunächst das Kriterium, auf welches die vorgestellte Kostenfunktion der Verfahren abzielte: die Minimierung der Länge der kritischen Pfade durch deren „Verlegung“ innerhalb der Partitionen. Bei der Längenberechnung wurden Intercluster-Netze mit dem Wert Eins, Intracluster-Netze hingegen mit dem Wert

Null berücksichtigt. Die Partitionierungsversuche wurden für LUT2- und für XC4000-Basisblöcke durchgeführt, wobei jeweils Kapazitätsschranken

$$B \in \{2, 4, 6, \ldots, 32\}$$

gewählt wurden. Die Ergebnisse der Partitionierungen der einzelnen Schaltkreise wurden für jede Kapazitätsschranke arithmetisch gemittelt.

Abbildung 3.2 zeigt die Resultate für LUT2-Blöcke, Abbildung 3.3 zeigt jene für XC4000-Blöcke. Die jeweiligen Kapazitätsschranken sind auf den Abszissen abzulesen. Insofern

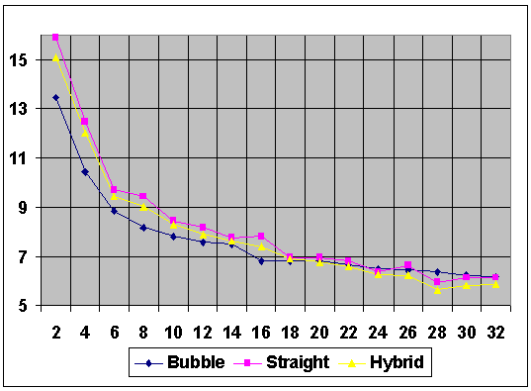


Abbildung 3.2: Länge des krit. Pfades (LUT2)

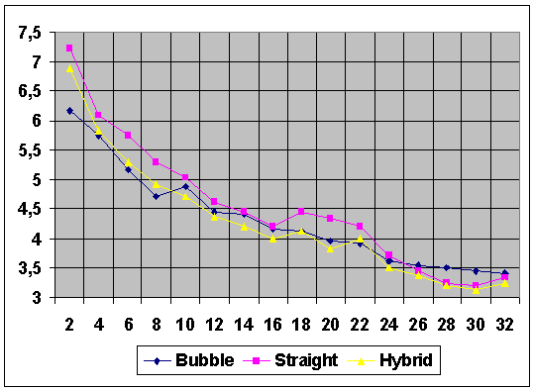


Abbildung 3.3: Länge des krit. Pfades (XC4000)

Bubble Partitioning mit einer größeren Suchfront arbeitet, als die *Straight*-Methode, sind die zumeist besseren Resultate des ersteren nicht verwunderlich. Der Verlauf der Kurven für das *Hybrid*-Verfahren zeigt aber, daß Partitionierungen des *Straight*-Verfahrens im Schnitt immer durch ein nachgeschaltetes *Bubble Partitioning* verbessert werden können. Bei größeren Partitionskapazitäten nähern sich die Ergebnisse der drei Methoden nach und nach an. Scheinbar sind hier für beide Verfahren genügend große Freiheitsgrade gegeben, so daß die Nähe zum Optimum nur noch gering ist. Allerdings zeigt sich insbesondere bei den Werten der grobgranulareren XC4000-Schaltkreise auch, daß zwischen den Suchräumen von *Straight*- und *Bubble*-Verfahren keine Inklusion besteht. Bei *Bubble Partitioning* besteht wohl unter größeren Kapazitäten eher die Gefahr eines Festlaufens in lokalen Minima. Dementsprechend arbeitet *Bubble Partitioning* auf feingranulareren Schaltkreisen, also auf Probleminstanzen mit vielen kleinen Blöcken, die stärker vernetzt sind, offensichtlich besser, wie Abbildung 3.4 zeigt.

Obwohl *Bubble Partitioning* hinsichtlich der Reduktion kritischer Pfade in der Regel bessere Resultate liefert, als die *Straight*-Methode, liegt die Anzahl der generierten Partitionen dennoch höher, wie den Abbildungen 3.5 und 3.6 zu entnehmen ist. Die Ergebnisse der *Hybrid*-Methode zeigen aber auch, daß durch eine dem *Straight*-Verfahren nachgeschaltete *Bubble*-Methode sogar noch Partitionen aufgelöst werden können.

Hinsichtlich der Ausnutzung der Kapazitätsgrenzen sehen die Resultate der drei Verfahren jedoch anders aus. Die Abbildungen 3.7 und 3.8 zeigen eine deutlich geringere Ausnutzung beim *Bubble Partitioning*, sowohl bei LUT2- als auch bei XC4000-Schaltkreisen.

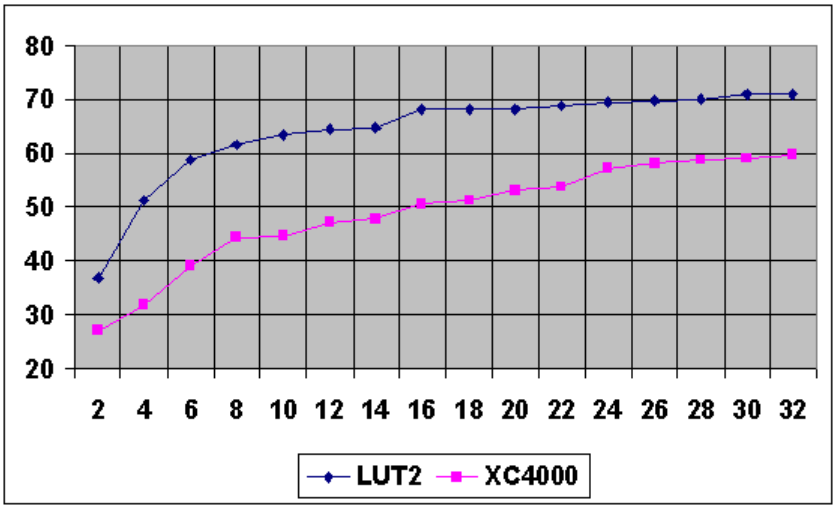


Abbildung 3.4: Reduktion des kritischen Pfades mit *Bubble Partitioning* (%)

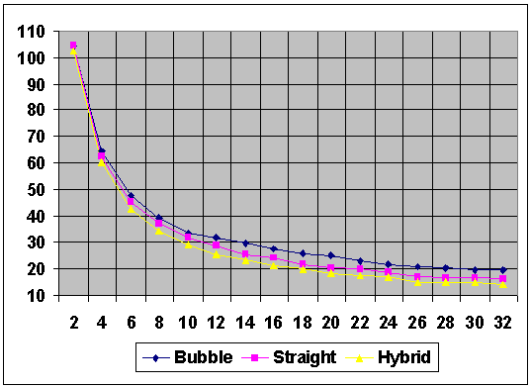


Abbildung 3.5: Anzahl der generierten Partitionen (LUT2)

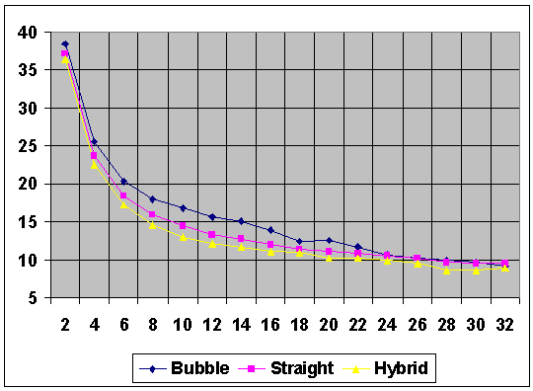


Abbildung 3.6: Anzahl der generierten Partitionen (XC4000)

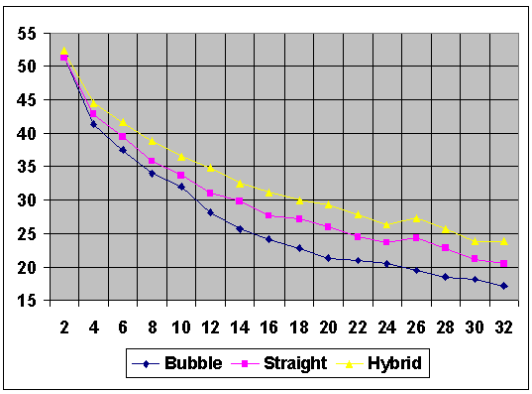


Abbildung 3.7: Prozentuale Ausnutzung der Kapazität der Partitionen (LUT2)

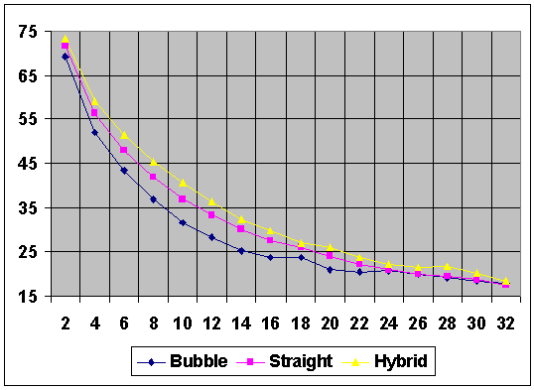


Abbildung 3.8: Prozentuale Ausnutzung der Kapazität der Partitionen (XC4000)

Anhand von Abbildung 3.9 zeigt sich jedoch, wie die höhere Anzahl und der stärkere Zusammenhang der Knoten in den LUT2-Schaltkreisen beim *Bubble*-Verfahren im Zuge höherer Partitionskapazitäten neutralisiert wird; die Ausnutzungsgrade zwischen LUT2- und XC4000-Partitionierungen gleichen sich an.

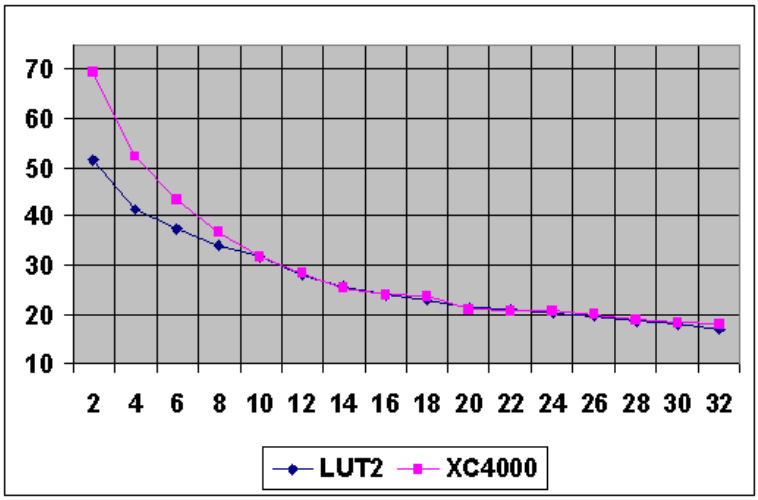


Abbildung 3.9: Ausnutzung der Kapazität mit *Bubble Partitioning* (%)

Die im Rahmen der Versuche beobachteten niedrigen Ausnutzungsgrade erhoben auch die Vermutung, daß durch Verschmelzen nicht-voller Partitionen eine weitere Reduktion der kritischen Pfadlänge möglich sei. Dementsprechend wurden auch Versuche mittels einer *MaxClique*-Heuristik unternommen, bei denen adjazente Partitionen derart verschmolzen werden sollten, daß möglichst kritische Netze vom Schnitt der Partitionierung entfernt werden, soweit die Kapazitätsgrenzen dadurch nicht überschritten werden. Diese Versuche brachten jedoch keine signifikanten Verbesserungen und sollen deshalb an dieser Stelle auch nicht weiter dokumentiert, sondern nur am Rande erwähnt bleiben.

Der eingangs erwähnte, größere Suchraum des *Bubble Partitioning* wirkt sich allerdings auch inhärent auf die Laufzeit des Verfahrens aus. Abbildung 3.10 zeigt den Verlauf des Zeitaufwandes in CPU-Sekunden für LUT2-Partitionierungen nach dem *Bubble*-Verfahren. Während einerseits die Laufzeiten des *Straight*-Verfahrens auf den vorliegenden Probleminstanzen kaum meßbar sind, wie in Abbildung 3.11 zu sehen ist, steigt die Laufzeit von *Bubble Partitioning* mit wachsender Kapazität streng monoton. Wie angesichts der bereits gesehenen Resultate über die Länge der kritischen Pfade erwartet, zeigt die Laufzeit-Kurve der *Bubble*-Methode jedoch einen degressiven Verlauf. Bei einer genügend großen Kapazitätsschranke tritt also offensichtlich nach einer stets etwa gleich großen Laufzeit eine Sättigung ein, so daß Verschiebungen von Knoten kaum noch Gewinne bringen. Die Kombination beider Methoden im *Hybrid*-Verfahren läßt *Bubble Partitioning* hinsichtlich der Laufzeit auch für sehr große Probleminstanzen wieder interessant werden.

Zusammenfassend kann man feststellen, daß *Bubble Partitioning* bessere Resultate liefert, was die Längenreduktion kritischer Pfade betrifft, dafür jedoch extensivere Laufzeiten besitzt, wohingegen das *Straight*-Verfahren zu einer besseren Ausnutzung der

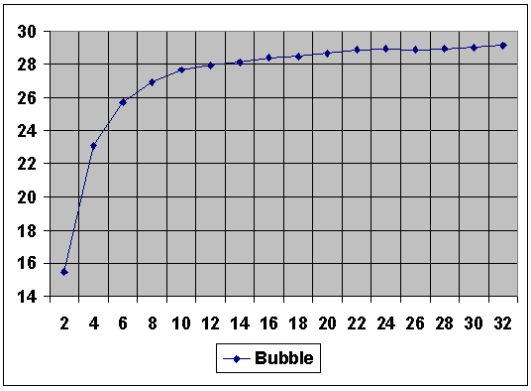


Abbildung 3.10: Laufzeit von *Bubble Partitioning* (CPUsec.)

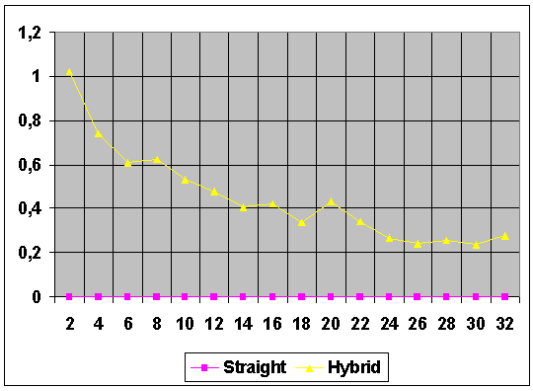


Abbildung 3.11: Laufzeiten von *Straight*- und *Hybrid*-Methode (CPUsec.)

Partitionen führt. Das *Hybrid*-Verfahren liefert zwar einen guten Kompromiß zwischen Laufzeit und Qualität der Ergebisse, ist aber für unser Konzept interagierender Algorithmen nicht brauchbar, da alle Partitionen vollständig konstruiert sein müssen, bevor die *Bubble*-Methode zur Verbesserung angewandt wird.

3.3 Ein interaktives Konzept

In diesem Abschnitt wird das eigentliche Verfahren zur Generierung von Schaltkreismakros vorgestellt, welches, wie bereits eingangs erwähnt, aus zwei interagierenden Algorithmen besteht: zum einen das in Abschnitt 3.2 dargestellte *Bubble Partitioning* Verfahren, zum anderen eine Methode zur inkrementellen Plazierung und Verdrahtung von Schaltkreisknoten, auch als *Layouter* bezeichnet, auf die genauer in Abschnitt 3.3.3 eingegangen wird.

3.3.1 Adaption von *Bubble Partitioning*

Das *Bubble Partitioning* Verfahren wurde bislang nur als in sich abgeschlossene Partitionierungsmethode vorgestellt, welche die einzelnen Partitionen quasi *simultan* konstruiert. Für die Interaktion mit einem kombinierten Plazierungs- und Verdrahtungsverfahren wird jedoch eine schrittweise Partitionierungsmethode benötigt, die zunächst lediglich einzelne Schaltkreisknoten als Kandidaten für eine Bewegung in andere Partitionen vorschlägt, so daß erst nach erfolgreicher Plazierung und Verdrahtung eines der vorgeschlagenen Kandidaten die Übernahme in dessen zugehörige Zielpartition tatsächlich vorgenommen wird.

Algorithmus 3.6 stellt eine entsprechende Methode für das *Bubble*-Verfahren dar zur Ermittlung einer nach Kosten sortierten Kandidatenliste. Die Methode erhält eine Referenz auf die Datenstruktur eines größenbeschränkten Heaps *(bounded heap, BHeap)*, welche nach Ausführung der Methode eine begrenzte Anzahl nach dem *Bubble*-Verfahren ermittelter Bewegungskandidaten mit Kostenreduktion enthält. Ordnungsgrößen

in der *Heap*-Struktur stellen dabei die größte Kostenreduktion als Erst- und die Anzahl der gekürzten kritischen Pfade (*CPS*-Wert) als Zweitkriterium dar.

Algorithmus 3.6 (Schrittweises Bubble Partitioning)

```
get_best_candidates( BHeap H(max_heapsize) )
{
    forall (P_i, P_j) mit i ≠ j, P_i ≠ ∅, P_j ≠ ∅, #P_j < B do
    {
        forall v ∈ P_i mit locked(v,j) = false do
        {
            cost_change = calc_cost_change(v,j);
            if cost_change < 0 then
                H.push(v,j,cost_change,CPS(v));
        }
    }
}
```

In der Methode **get_best_candidates** ist in Abweichung zur ursprünglichen Version des *Bubble*-Verfahrens auch die Übergehung gesperrter *(locked)* Knoten/Partitionen-Paare vorgesehen - dazu in Kürze mehr.

3.3.2 Rahmenalgorithmus

Die Interaktion zwischen *Partitioner* und *Layouter* wird nun mit dem Rahmenalgorithmus (Algorithmus 3.7) des Makrogenerierungsverfahrens deutlich.

Algorithmus 3.7 (Rahmenalgorithmus Makrogenerierung)

```
Sei F ∪ L = {v_1, ..., v_n};
forall j ∈ {1...n} do
    M_j = Layouter::create_macro(v_j);
do
{
    BHeap H(max_heapsize);
    BubblePartitioner::get_best_candidates(H);
                                    // Ermittle Liste bester Kandidaten
    implemented = false;
    while(H ≠ ∅ and not implemented)
    {
        (v, j) = H.top( );          // Bester Kandidat (Knoten, Zielpartition)
        implemented = Layouter::place_and_route(v, M_j);
                    // Versuche Implementierung von Knoten v in Makro M_j
```

```
        if implemented then
        {
            BubblePartitioner::move_node(v, j);
                                    // Knoten v tatsächlich nach P_j verschieben
            forall i ∈ {1 ... n} do
                locked(v, i) = true;            // Knoten v nicht mehr bewegen
        }
        else
        {
            locked(v, j) = true;              // Sperre Partition P_j für Knoten v
            H.pop( );                          // Kandidat von BHeap entfernen
        }
    }
}
while (H ≠ ∅);
```

Der Initialpartitionierung entsprechend, wird zunächst zu jedem Schaltkreisknoten ein Makro erzeugt, indem der Knoten in einem Segment der gegebenen Architektur implementiert wird. Solange der *Partitioner* Kandidaten zur Verschiebung vorschlägt, wird versucht, eine Implementierung des besten Kandidaten in dem zur Zielpartition gehörenden Makro zu berechnen. Gelingt dies, wird die vorgeschlagene Knotenbewegung vom *Partitioner* auch tatsächlich durchgeführt. Gelingt dies nicht, wird die Zielpartition für den Kandidaten künftig gesperrt und eine Implementierung des vorgeschlagenen zweitbesten Kandidaten versucht. Wurde ein Kandidat implementiert, also die Verschiebung vorgenommen, müssen die Verschiebungskosten neu berechnet werden, das heißt der *Partitioner* muß neu angestoßen werden. Mißlang hingegen ein Implementierungsversuch, so kann mit den übrigen vorgeschlagenen Kandidaten ohne eine Neuberechnung fortgefahren werden.

Die erwähnte Sperrung von Knoten/Partitions-Paaren erfolgt dauerhaft bis zum Ende des gesamten Makrogenerierungsverfahrens. Mit der Sperrung wird ein doppelter Zweck verfolgt: zum einen wird dadurch die Neuberechnung der Verschiebungskosten vermieden, falls die Implementierung eines Knotens in einem Makro bereits scheiterte. Zum anderen werden wir im Rahmen der Verwendung des *Bubble Partitioning* Verfahrens bei der Makrogenerierung aus noch zu erläuternden Gründen die Schaltkreisknoten nach einer Verschiebung sperren, so daß jeder Knoten höchstens einmal verschoben wird.

Daß diese Einschränkung des Suchraumes des *Bubble Partitioning* Verfahrens nicht notwendigerweise auch in Qualitätseinbußen der Partitionierungen resultiert, zeigten Versuche mit dem bereits genannten Benchmark-Set. Das Diagramm in Abbildung 3.12 illustriert die prozentuale Veränderung der Länge der kritischen Pfade, sowie der Kosten der Partitionierungen bei LUT2-Schaltkreisen, wenn zusätzlich die Sperrung bereits verschobener Knoten aktiviert wurde. Die Werte sind wieder über die Schaltkreise jeweils für verschiedene Partitionskapazitäten arithmetisch gemittelt.

Der Verlauf der Kurven scheint zunächst zwar etwas merkwürdig, doch ist in beiden Kurven eine gewisse synchrone Wellenstruktur ersichtlich, die auf die Bildung von Kno-

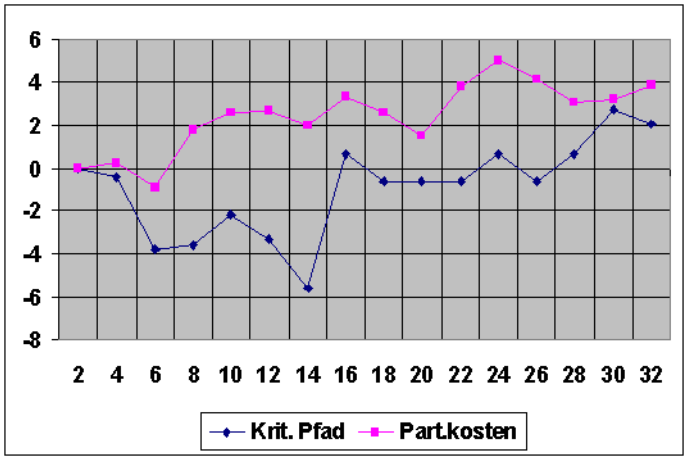


Abbildung 3.12: Prozentuale Veränderung der Länge der kritischen Pfade, sowie der Kosten der Partitionierung unter Knotensperrung

tengruppen in den Partitionen schließen lassen, welche über ihre gemeinsamen Adjazenzen eine gewisse konzentrierte Gravitation ausüben, die sich jedoch durch die Begrenzung der Partitionskapazitäten etappenweise erschöpft.

Insgesamt ist mit zunehmender Partitionskapazität bei beiden Kurven zwar die erwartete Zunahme von Partitionskosten und Pfadlänge zu erkennen, doch verschlechtert sich die Pfadlänge erstaunlicherweise nur bei fünf von 16 Werten der Kapazität, wobei die Verschlechterung zumeist unter einem Prozent liegt.

Indem jeder Knoten aus seiner Zielpartition, in die er einmal bewegt wurde, nach einer Sperrung nicht mehr entfernt werden kann, besitzt die aufgrund eines gescheiterten Implementierungsversuches erfolgte Ablehnung eines Kandidaten bis zum Ende des Verfahrens hin Gültigkeit, das heißt der Knotenkandidat kann im entsprechenden Makro auch zu einem beliebigen späteren Zeitpunkt nicht implementiert werden. Somit kann die zugehörige Partition für diesen Knoten dauerhaft gesperrt werden.

Mit dem durch den obigen Rahmenalgorithmus dargestellten Konzept der Makrogenerierung kann nunmehr auch besser erläutert werden, weshalb eine Kostenprofil konstruierende Heuristik wie jene von *Fiduccia* und *Mattheyses* [28] hier nur relativ schlecht einsetzbar ist. Denn steht nach Konstruktion des Kostenprofils eine Bewegungsreihenfolge einzelner Knoten über den Schnitt der Partitionierung fest, so kann diese hernach dann nicht vollständig durchgeführt werden, wenn die einhergehende sukzessive Plazierung und Verdrahten der Knoten fehlschlägt. Die entstehende Implementierungsfolge der Knoten könnte somit auf einem Kostenmaximum enden, wodurch Freiheitsgrade bei der Partitionierung verloren gingen.

Satz 3.3
Das *Bubble Partitioning* Verfahren mit Knotensperrung terminiert nach höchstens $\mathcal{O}(\#(F \cup L)^4)$ Schritten.

Beweis: Wir betrachten zunächst die Methode **get_best_candidates** aus Algorithmus 3.6. Sei $n = \#(F \cup L)$. Dann iteriert die äußere Schleife der Methode auf der Initialpartitionierung χ_0 über alle n^2 Paare von Partitionen und

die innere Schleife wird jeweils genau einmal ausgeführt. Doch auch im Zuge weiterer Aufrufe der Methode gilt unter jeder Partitionierung χ stets:

$$\sum_{i=1}^{n} \#P_i^{\chi} = n$$

Somit wird der Rumpf der inneren Schleife auch bei weiteren Aufrufen von **get_best_candidates** höchstens n^2-mal ausgeführt. Für die Methode **calc_cost_change** aus Algorithmus 3.2 wurde die Laufzeit aber bereits zu $\mathcal{O}(n)$ festgestellt. Somit kann die Laufzeit der Methode **get_best_candidates** abgeschätzt werden durch $\mathcal{O}(n^3)$.

Das gesamte *Bubble Partitioning* Verfahren terminiert, wenn die Methode **get_best_candidates** einen leeren *BHeap* liefert, also wenn kein Kandidat mehr verschoben werden kann. Da jeder Knoten nur einmal verschoben werden kann, ist dies spätestens nach n Iterationen der Fall, womit sich die Behauptung des Satzes ergibt. □

Aus dem Beweis des vorangegangenen Satzes 3.3 ist insbesondere ersichtlich, daß jeder *Bubble*-Schritt, also die Berechnung der Liste der Verschiebungskandidaten, von kubischer Laufzeit ist. Bemerkt sei jedoch, daß sich die angegebene Laufzeit nur auf den Partitionierungsalgorithmus als abgeschlossenes Verfahren bezieht, nicht aber auf die gesamte Makrogenerierungsmethode.

Ein vom *Partitioner* vorgeschlagener Knotenkandidat wird, wie bereits erläutert, erst dann in seine Zielpartition verschoben, wenn eine Implementierung des Knotens und seiner adjazenten Netze in dem zur Zielpartition gehörenden Makro gefunden werden konnte. Das Verfahren zur Berechnung einer solchen gültigen Implementierung, der *Layouter* wird im folgenden Unterabschnitt betrachtet.

3.3.3 Placing-by-Routing Verfahren

Das Layoutproblem besteht für einen einzelnen Schaltkreisknoten, vereinfacht ausgedrückt, darin, im Architektursegment des Zielmakros unter Berücksichtigung dessen bisheriger Konfiguration eine freie Taskressource zu finden, die den Knotenkandidaten zu implementieren vermag und freie Verdrahtungsressourcen zu lokalisieren, so daß K-Bäume konstruierbar sind, welche eine Implementierung der adjazenten Netze darstellen. Die allgemein übliche Reihenfolge in der knotensequentiellen Lösung von Layoutproblemen geht so vor, daß zunächst, unterstützt durch Abschätzungsheuristiken, eine Plazierung des Schaltkreisknotens gewählt wird, danach versucht wird, die zugehörigen Netze zu verdrahten, um beide Schritte wieder zu verwerfen, wenn der letztere fehlschlug.

Indem es bei feldprogrammierbaren Architekturen jedoch nicht nur um die Auswahl einer „Plazierung“ für einen Schaltkreisknoten, wie etwa bei ASICs, geht, sondern vielmehr um die Zuordnung eines Tasks einer konfigurierbaren Zelle, ferner die Verdrahtungsressourcen nicht etwa durch ein zweidimensionales orthogonales Gitter darstellbar sind, sondern durch die gegebenenfalls sogar multidimensionale Struktur eines periodischen Graphen und durch Routing-Tasks, welche mit Logik- und Speichertasks

hinsichtlich Kompatibilität konkurrieren, wurde ein anderer Ansatz gewählt, der zuerst eine „günstige“ Implementierung der Netze sucht und anhand dieser Netzrealisierung eine Implementierung des Schaltkreisknotens ermittelt *(Route-then-Place, Placing-by-Routing)*.

Die insbesondere bei feingranularen feldprogrammierbaren Architekturen anzutreffende, starke Einschränkung der Verdrahtungsressourcen führen bei konventionellen Ansätzen mit Verdrahtung anhand festgelegter Terminallokationen relativ schnell zu Blokkaden *(Routing Congestions)*, durch welche leicht sehr große Teillösungen im Rahmen des *Backtrackings* verworfen werden müssen. Beim Entwurf eines generischen Layoutverfahrens, das für die verschiedensten Architekturen einzusetzen ist, sollte hingegen eine gewisse „Robustheit“ der Verfahren hinsichtlich der Implementierbarkeit, insbesondere auch der Verdrahtbarkeit von Schaltkreisen einen Schwerpunkt darstellen.

Unser neuer Ansatz verfolgt somit eine umgekehrte Strategie, wobei im Gegensatz zur konventionellen Netz-für-Netz-Verdrahtung alle, den aktuell betrachteten Schaltkreisknoten berührenden Netze *simultan* verdrahtet werden. Wir bezeichnen daher unsere im folgenden genauer dargestellte generische Layout-Methode als *Simultane Pfad Suche (Simultaneous Path Search, SPS)*. Sie basiert auf mehreren, im wesentlichen voneinander unabhängigen, jedoch quasi-gleichzeitig ablaufenden, klassischen Kürzeste-Wege-Suchen, wobei im Gegensatz zur sonst üblichen Betrachtungsweise die Rollen von Knoten und Kanten vertauscht sind; eine Wege-Suche verläuft demnach von Kante zu Kante eines periodischen Graphen, während die Menge der freien Routing-Tasks in den Knoten die Adjazenzrelation bestimmen [84].

Temporäre Routen

Ein Problem, welches sich bei einer gleichzeitigen Implementierung aller adjazenten Netze eines Knotenkandidaten ergibt, ist die mögliche Unbestimmtheit der Lokationen von Terminalen. Dies tritt genau dann auf, wenn es sich bei einem Terminal um einen primären Eingang oder primären Ausgang des Schaltkreises handelt oder das Netzterminal gerade ein Pin eines noch nicht implementierten, benachbarten Schaltkreisknotens ist.

Wir führen deshalb den Begriff der *temporären Route* ein, die eine Netzimplementierung darstellt, auf die das Zielmakro zwar konfiguriert wird, welche jedoch zu einem späteren Zeitpunkt wieder entfernt *(dekonfiguriert)* werden kann. Eine temporäre Route verläuft stets zwischen einem Terminal eines implementierten Schaltkreisknotens und der *Peripherie* des Makros, wobei wir als Peripherie die Menge aller Nullkanten des dem Makro zugrundeliegenden periodischen Graphen bezeichnen.

Definition 3.11 (Temporäre Route)
Ein K-Pfad W wird als *temporärer K-Pfad* oder auch *temporäre Route* bezüglich eines Makros M bezeichnet, falls eine der beiden folgenden Bedingungen gilt:

(1) $Q^{\text{CELL}}(W) \in M$ und $Z^{\text{CELL}}(W) = v_{\text{nil}}$

(2) $Z^{\text{CELL}}(W) \in M$ und
$\left(Q^{\text{CELL}}(W) = v_{\text{nil}} \text{ oder } \exists_{W' \text{ temporärer K-Pfad}}\ Q^{\text{CELL}}(W) = Z^{\text{CELL}}(W')\right)$

Die Strategie der Bildung temporärer Routen wendet sich gleich drei Problematiken zu: zum ersten wird damit eine Reservierung von Verdrahtungsressourcen für spätere, jedoch in jedem Falle[2] folgende Netzimplementierungen getroffen, zum zweiten wird, im selben Sinne, auch „Raum“ für einen noch zu plazierenden Nachbarknoten geschaffen und zum dritten wird durch eine temporäre Route die Verfügbarkeit des Netzes auch außerhalb des Makros garantiert. Indem aufgrund der Periodizitätseigenschaft des Architekturmodells die Dilatation eines Makros in alle Dimensionen angepaßt werden kann, ergibt sich folgender zwar günstiger, jedoch zugegebenermaßen eher theoretischer Zusammenhang:

Satz 3.4 (Unblockierbarkeit von Implementierungen)
In jeder *vernünftigen* Architektur[3], die keiner Beschränkung der Dilatation unterliegt, existiert unter jeder beliebigen Implementierungsreihenfolge der Knoten eines Schaltkreises stets eine gültige Implementierung des Schaltkreises.

Beweis: Aufgrund der unbeschränkten Dilatation ist zu jedem Schaltkreisknoten jederzeit eine gültige Implementierung zu finden. Da in einer vernünftigen Architektur ferner zu jedem beliebigen Paar von Pins ein K-Pfad existiert, der die beiden Pins verbindet und jedes durch eine temporäre Route implementiertes Netz vermöge eines Terminals an der Peripherie verfügbar ist, kann die Dilatation der Architektur so erhöht werden, daß stets freie Routingressourcen zur Verfügung stehen, um Terminale der Peripherie mit den Terminalen eines plazierten Schaltkreisknotens zu verbinden. □

Zur genauen Unterscheidung, welche Teilimplementierungen eines Netzes als temporäre Routen zu konstruieren sind, führen wir die folgende Klassifikation von Netzen bezüglich Partitionen ein (vgl. Abbildung 3.13):

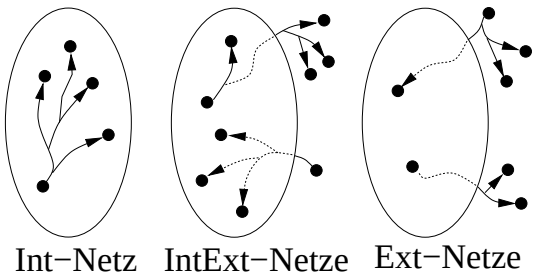


Abbildung 3.13: Netz-Klassifikation

[2] spätestens im Zuge der Endverdrahtung der Makros
[3] vgl. Definition 2.23 in Abschnitt 2.3.4

Definition 3.12 (Int-, IntExt- und Ext-Netze)
Sei χ eine Partitionierung eines LUT-Schaltkreises $\mathcal{C} = (V, E)$. Sei P_i^χ eine Partition für $i \in \{1, \ldots, n_\chi\}$ und $e \in E$ ein Netz mit $\rho(e) \cap P_i^\chi \neq \emptyset$. Wir bezeichnen ein Netz e als:

Int-Netz, falls gilt:	$\rho(e) \subseteq P_i^\chi$
IntExt-Netz, falls gilt:	$\#(\rho(e) \cap P_i^\chi) > 1$ und $\#(\rho(e) \setminus (\rho(e) \cap P_i^\chi)) \geq 1$
Ext-Netz, falls gilt:	$\#(\rho(e) \cap P_i^\chi) = 1$ und $\#(\rho(e) \setminus (\rho(e) \cap P_i^\chi)) \geq 1$

Die Generierung eines Makros M wird dann unter der Aufrechterhaltung der folgenden Invarianten über K-Pfade und K-Bäume durchgeführt:

1. Der K-Baum eines IntExt- oder eines Ext-Netzes in M enthält einen temporären K-Pfad (gegebenenfalls mit ebenfalls temporären Subpfaden).
2. Der in M verlaufende K-Pfad eines Ext-Netzes ist temporär.
3. Der von oder zur Peripherie von M verlaufende K-Pfad eines IntExt-Netzes ist temporär.

Der Status der zu einem Schaltkreisknoten adjazenten Netze wird entsprechend obiger Klassifikation also bei jedem Implementierungsversuch aktualisiert, die berechneten Routen gegebenenfalls als *temporär* markiert.

Generierung von Suchen

Wir betrachten im folgenden ein Makro M der Dilatation x mit zugrundeliegender Architektur A^x, deren Knotenmenge V^x und Kantenmenge E^x sei. Der zu implementierende Schaltkreis sei $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$.

Zu einem K-Baum T bezeichne $E_{\text{FAN}}(T) \subset E^x$ die Menge der Kanten e, deren Quellknoten $Q^{\text{CELL}}(e)$ in T liegt, wobei in diesem Quellknoten ein konfigurierbarer Verzweigungs-Brückentask von T nach e existiert.

Sei $E_{\text{USE}} \subseteq E^x$ die Menge der Kanten, welche in der auf $\mathcal{A}^x$ bereits konfigurierten Implementierung irgendeines Netzes vorkommen:

$$E_{\text{USE}} = \{e \in E^x \mid \exists_{N \in E_\mathcal{C}}\ e \in \rho^{\text{NET}}(N)\}$$

Sei B ein im Makro M zu implementierender Schaltkreisknoten und $\mathcal{N}$ die Menge der zu B adjazenten Netze:

$$\mathcal{N} = E_{\mathcal{C}B}^- \cup E_{\mathcal{C}B}^+$$

Die Menge $\mathcal{N}$ setzt sich also zusammen aus den in B einlaufenden Netzen und dem von B ausgehenden Netz. Zu jedem einlaufenden Netz generieren wir eine Vorwärtssuche, zum ausgehenden Netz eine Rückwärtssuche. Der Initialisierungsschritt ist in Algorithmus 3.8 dargestellt. Er liefert eine Menge $\mathcal{S}$ von Kürzeste-Wege-Suchen mit jeweils initialisierter Suchfront zurück.

Algorithmus 3.8 (SPS: Generierung von Suchen)

Dekonfiguriere auf M die temporären Routen aller Netze aus $\mathcal{N}$.

forall $B' \in \mathit{pred}(B)$ do
{
  Sei $N = \mathit{drv}^{-1}(B')$.
  if N implementiert in M then
    $F = E_{\text{FAN}}(\rho^{\text{NET}}(N)) \setminus E_{\text{USE}}$;
  else
    if (B' implementiert in M) then
      $F = \{e \in E^x \setminus (N^x \cup E_{\text{USE}}) \mid Q^{\text{PIN}}(e) = o_{T_{B'}}\}$;
    else
      $F = \{e \in N^x \setminus E_{\text{USE}} \mid Z(e) \neq v_{\text{nil}}\}$;
  if $F \neq \emptyset$
  {
    Generiere Vorwärtssuche S mit Front F;
    $\mathcal{S} = \mathcal{S} \cup \{S\}$;
  }
}
if kein $B' \in \mathit{succ}(B)$ implementiert in M then
  $F = \{e \in N^x \setminus E_{\text{USE}} \mid Q(e) \neq v_{\text{nil}}\}$;
else
{
  Sei $N = \mathit{drv}^{-1}(B)$.
  forall $B' \in \mathit{succ}(B)$ mit B' implementiert in M do
    $F = \{e \in E^-_{\rho^{\text{FUNC}}(B')} \setminus E_{\text{USE}} \mid Z^{\text{PIN}}(e) \in I_{T_{B'}}\}$;
}
if $F \neq \emptyset$ then
{
  Generiere Rückwärtssuche S mit Suchfront F.
  $\mathcal{S} = \mathcal{S} \cup \{S\}$;
}

Besitzt ein Netz aus $\mathcal{N}$ bereits eine nicht-temporäre Implementierung, also einen K-Baum T, so wird die Front der für dieses Netz zu erzeugenden Suche mit allen ungenutzten Kanten aus $E_{\text{FAN}}(T)$ initialisiert. Existiert hingegen noch keine Netzimplementierung, sondern lediglich die Implementierung eines Vorgänger- oder Nachfolgerknotens von B, so wird die Suchfront mit den entsprechenden Terminalen dieser Implementierung initialisiert. Ist ein Vorgängerknoten von B nicht implementiert bzw. keiner der Nachfolgerknoten, so werden die Fronten der entsprechenden Suchen mit allen ungenutzen einlaufenden bzw. auslaufenden Nullkanten initialisiert.

Vor der Initialisierung der Suchen werden die temporären Routen aller zu B adjazenten Netze entfernt. Im Falle von IntExt-Netzen bleiben also (nicht-temporäre) Routen auf dem Architektursegment konfiguriert. Die damit verfolgte Strategie orientiert sich

an der Steinerbaum-Heuristik von *Wu, Widmayer* und *Wong*, auch bekannt als *WWW-Algorithmus* [85], welche auf Kruskal's Verfahren für minimale aufspannende Bäume basiert. Im Unterschied zum WWW-Algorithmus findet hier die sukzessive Expansion einer bisher gefundenen Teillösung jedoch nicht notwendigerweise in unmittelbarer Folge statt, sondern kann, entsprechend der Implementierungsreihenfolge der Schaltkreisknoten, mit Routingversuchen anderer Netze abwechseln.

Im Gegensatz hierzu könnten natürlich auch alle Routen eines zu verdrahtenden Netzes, also nicht nur die temporären Routen, dekonfiguriert werden. Der Vorteil hiervon wären höhere Freiheitsgrade in der Plazierung des Knotenkandidaten B, andererseits würde jedoch auch das Verdrahten schwieriger, da mehr Engpässe *(Routing Congestions)* entstehen können. Im schlimmsten Fall könnte dabei die Anzahl der Re-Iterationen *(Backtracks)* nicht mehr ausreichen, wodurch Schaltkreisknoten trotz der Existenz einer Implementierung abgelehnt würden. Ferner stiege zumindest auch der bisherige Verdrahtungsaufwand bis um den Faktor der Knotenzahl des Schaltkreises an. Aus diesen Gründen, verbunden mit empirisch gewonnenen Beobachtungen, beschränkt sich unser Verfahren auf die Dekonfiguration lediglich der temporären Routen.

Implementierbarkeit von Schaltkreisknoten

Wie wird nun die beste Implementierung für den Knoten B in Makro M ermittelt? Während SPS werden die Netze der einzelnen Pfadsuchen an jeder konfigurierbaren Zelle, welche die Suchen erreichen, als „verfügbar" registriert. Sind nun alle Netze der Simultansuchen an einer Zelle verfügbar, so ist zu ermitteln, ob die erreichten Pins der Zelle eine Teilmenge der Eingangsmenge eines Tasks sind, der B implementieren kann. Die Frage lautet also: existiert ein ein Task T, vermöge dem B implementierbar ist und eine injektive Abbildung λ aus der Menge der von B benötigten Netze in die Menge der zu T gehörenden Eingänge?

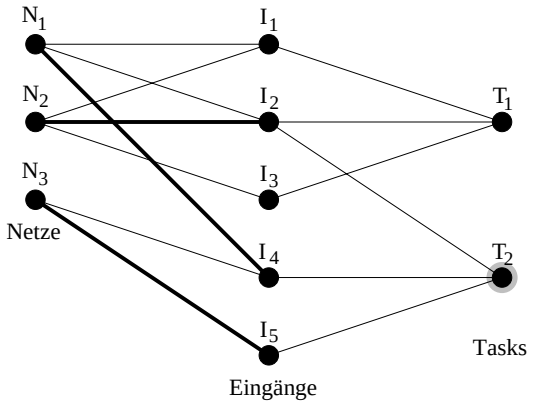


Abbildung 3.14: Beispiel eines TN-Matchings

Diese Fragestellung entspricht jedoch einem *Maximum Matching Problem* auf einem bipartiten Graph. Wir bezeichnen daher eine Lösung (T, λ) als *Task/Netz-Matching (TN-Matching)* für die konfigurierbare Zelle. Zur Lösung eines TN-Matching-Problems existieren effiziente Algorithmen, beispielsweise über Netzwerkflußmethoden. Für unse-

ren generischen Layouter wurde ein entsprechendes Verfahren über augmentierende Pfade mit Laufzeit $\mathcal{O}(r_B \cdot s_B)$ adaptiert [58], wobei r_B die Anzahl der Tasks ist, die den Schaltkreisknoten B zu implementieren vermögen und s_B die Summe der Anzahl jener Eingänge der konfigurierbaren Zelle darstellt, welche von diesen Tasks genutzt werden.

Abbildung 3.14 illustriert das TN-Matching-Problem nochmals anhand eines Beispiels. Die drei Netze N_1, N_2 und N_3 sind jeweils an mehreren verschiedenen Eingangspins $I_1, \ldots, I_5$ einer konfigurierbaren Zelle verfügbar. Jeder der beiden Tasks T_1 und T_2 der Zelle bezieht seine Eingangssignale aus einer dreielementigen Teilmenge der Pins. Im Beispiel der Abbildung existiert kein TN-Matching für Task T_1, da das Netz N_3 an keinem der Eingangspins des Tasks verfügbar ist. Es existiert jedoch ein TN-Matching für T_2 vermöge $\lambda(N_1) = I_4$, $\lambda(N_2) = I_2$ und $\lambda(N_3) = I_5$.

Kostenfunktion

Die wichtigsten Zielsetzungen des vorgestellten Makrogenerierungsverfahrens stellen die Implementierbarkeit des vorgegebenen Schaltkreises, sowie die Minimierung des aus der Implementierung resultierenden Verdrahtungsdelays dar. Während das erste der beiden Ziele durch das Konzept der temporären Routen angegangen wird, zielen die Kostenfunktionen von Partitionierungs- und Layoutverfahren auf die zweite Anforderung.

Wie bereits erwähnt, geht der Partitionierungsalgorithmus im wesentlichen vom anhand der Struktur des Schaltkreises bestimmten, schlupfbasierten, *RDC*-Maß aus, welches die Kritizität eines Netzes unter der Vorstellung bewertet, daß Intracluster-Netze grundsätzlich hinsichtlich des Delays besser zu implementieren sind, als Intercluster-Netze. Das Layoutverfahren greift nun ebenfalls dieses strukturelle Maß auf und integriert es in eine Kostenfunktion zusammen mit konkreten Delaywerten, wie sie mit der Beschreibung der konfigurierbaren Zellen spezifiziert wurden.

Die Pfadsuchen zur Implementierung der zu einem Knotenkandidaten adjazenten Netze werden quasi-simultan durchgeführt, doch der definitive Verlauf der hierbei berechneten Routen steht erst nach Terminierung der Suchen fest. Somit können auch von mehreren Routen gemeinsam genutzte Verdrahtungsressourcen erst nach Terminierung der Suchen detektiert werden. Wir bezeichnen dabei sowohl mehrfach genutzte Kanten als *Konfliktkanten*, als auch – allgemeiner – Kanten, deren R-Tasks im gemeinsamen Quellknoten nicht kompatibel sind. Durch einen erneuten Start aller Suchen wird nun versucht, diese Art von Routenüberschneidungen aufzulösen, wobei auch nicht konfliktierende Netze neu verdrahtet werden, da diese gegebenenfalls Platz bei Engpässen schaffen können. Die Nutzung der Konfliktkanten wird nun durch einen *Straf faktor* verteuert, der linear von der Anzahl der bislang durchgeführten Re-Iterationen abhängt. Konfliktierende Pfadsuchen „verhandeln“ also praktisch um die Nutzung von Ressourcen, wobei die Suchen kritischer Netze vermöge ihres höheren *RDC*-Wertes einen größeren Einfluß geltend machen können. Die Strategie eines solchen iterativen Verdrahtungsalgorithmus wurde für FPGAs bereits mit dem sogenannten *PathFinder*-Verfahren vorgestellt [55]. Jedoch sind dort, im Gegensatz zum hiesigen Verfahren, die Lokationen aller Netzterminale bereits vorgegeben; durch unser Konzept der temporären Routen mit freier Terminalpositionierung auf der Peripherie ergeben sich hinsicht-

lich der Entschärfung konfliktierender Routen jedoch wesentlich größere Freiheitsgrade, als bei der *PathFinder*-Methode.

Entsprechend motiviert, soll die in unserem Verfahren angewandte Kostenfunktion nun konkret angegeben werden. Betrachtet sei eine Pfadsuche des Netzes N, welche unmittelbar nacheinander die Kanten u und v erreicht, wobei in der verdrahtenden konfigurierbaren Zelle der R-Task t benutzt wird. Dann ergeben sich die Verdrahtungskosten zur Kante v auf diesem Pfad wie folgt:

$$d_v = d_u + (c_{u,N} + \delta_t) \cdot \mu + \gamma_u$$

Die Kosten des bislang ermittelten Pfades von der Startkante der Suche bis zur Kante v ergeben sich also durch Akkumulation der Kosten bis zur Kante u, den skalierten Kosten $c_{u,N}$ zur Verdrahtung des Netzes N über die Kante u und den skalierten Kosten des Routing-Tasks t, sowie einem randomisierten Anteil γ_u. Die Kantenkosten $c_{u,N}$ definieren sich genauer durch:

$$c_{u,N} = p_u \cdot \big(2 - \mathit{RDC}(N)\big)$$

Die Netzkritizität $\mathit{RDC}(N)$ wird hier skaliert mit einem Straffaktor p_u, der initial den Wert Eins besitzt. War die Kante u in einer früheren Iteration hingegen eine Konfliktkante, so bestimmt sich der Faktor durch:

$$p_u = r_u \cdot n_i$$

Dabei bezeichne r_u die Anzahl der Routen, welche in der letzten konfliktierenden Iteration die Kante u benutzten und n_i die Anzahl der bis dorthin durchgeführten Re-Iterationen plus Eins. Die Bestrafung einer Kante wird also im Zuge der Re-Iterationen langsam erhöht, damit nicht sofort übermäßig teure Routen genommen werden.

Mittels des Skalierungsfaktors μ können die Verdrahtungskosten im Falle des Suchens nach temporären Routen reduziert werden, da temporäre Routen später ohnehin wieder entfernt werden. Der neutrale Wert für μ beträgt 1. Im Rahmen empirischer Versuche wurden für kleinere Werte (optisch) kompaktere Layouts beobachtet.

Der Zufallswert γ_u erfüllt den Zweck einer Arbitrierung zwischen den einzelnen Suchen hinsichtlich der Nutzung der Kante u, indem er für geringfügige Abweichungen zwischen den jeweils berechneten Kostenwerten sorgt. Anschaulich interpretiert, sorgt der Wert γ_u für ein geringfügiges „Rauschen“ auf den Kantenkosten, so daß eine gewisse „Loslösung vom algorithmischen Schematismus“ propagiert wird, der in Konfliktfällen zur Wahl unterschiedlicher Routen führen soll. Während der Wert im Normalfall Null beträgt, hat er sich in der Praxis bei konfliktierenden Kanten als Bruchteil der Grundkosten $c_{u,N}$ als effektiv erwiesen.

Simultane Pfad Suche (SPS)

Die Durchführung der *Simultanen Pfad Suche* auf einer generierten Menge $\mathcal{S}$ von Suchen für einen Schaltkreisknoten B ist durch Algorithmus 3.9 dargestellt.

Algorithmus 3.9 (Simultane Pfad Suche)

```
(v, T) layout_sps(S, B)
{
    d_best = ∞;
    ∀_{S∈S} ∀_{e∈E^x} δ_S(e) = ∞;                    // Reset der Distanzen
    ∀_{S∈S} ∀_{v∈V^x} A_S(v) = false;                // Reset der Netzverfügbarkeiten
    while ∃_{S∈S} S.front ≠ ∅ do
    {
        e = S.front.pop( );
        if S Vorwärtssuche then
            v = Z^CELL(e);
        else
            v = Q^CELL(e);
        A_S(v) = true;                               // Netz nun an v verfügbar
        if δ_S(Q^PIN(e)) < d_best und ∀_{S'∈S} A_{S'}(v) then
            if TN-Matching (T, λ) auf v existiert
                mit max_{S''∈S} δ_{S''}(Q^PIN^-1(λ(S''.netz))) + δ(T) < d_best
                und T nicht gesperrt für B then
            {
                ρ^FUNC_best = (v, T);                // Beste Plazierung
                d_best = max_{S''∈S} δ_{S''}(Q^PIN^-1(λ(S''.netz))) + δ(T);
            }
        Sei K_v die Menge der in v konfigurierten Tasks.
        // Prüfe alle verfügbaren R-Tasks in v über e:
        forall R-Tasks T ∈ T_v mit K_v ∪ {T} konsistent
            und Z^PIN(e) = i_T, falls S Vorwärtssuche
            oder Q^PIN(e) = o_T, falls S Rückwärtssuche do
        {
            if S Vorwärtssuche then
                e' = Q^PIN^-1(o_T);
            else
                e' = Z^PIN^-1(i_T);
            if e' ∈ N^x then                         // Keine Nullkanten der Suchfront hinzufügen
                continue;
            if δ_S(e) + (c_S(e) + δ(T)) · μ + γ(e) ≤ δ_S(e') then
            {
                δ_S(e') = δ_S(e) + (c_S(e) + δ(T)) · μ + γ(e);
                R_S(e') = (e, T);
                S.front.append(e');
            }
        }
    }
    return ρ^FUNC_best;
}
```

In der Initialisierungsphase von Algorithmus 3.9 werden die maximalen Verdrahtungskosten d_{best} über alle ermittelten Pfade von bzw. zur Implementierung, die Distanzenwerte aller Suchen δ_S, sowie die Markierungen A_S für Knoten hinsichtlich dort bereits verfügbarer Netze zurückgesetzt. In der *while*-Schleife werden alle Suchen reihum propagiert, wobei zwischen Vorwärts- und Rückwärtssuche unterschieden wird. Nachdem die Verfügbarkeit des Netzes der aktuell betrachteten Suche S an der erreichten konfigurierbaren Zelle v registriert wurde, wird v auf die Möglichkeit einer Implementierung von B hin untersucht, jedoch nur dann, wenn alle erforderlichen Netze an v verfügbar sind und die Verdrahtungskosten des im Rahmen der aktuellen Suche bisher gefundenen Pfades kleiner als die maximalen Verdrahtungskosten d_{best} zur bisher gefundenen, besten Implementierung sind. Falls ein TN-Matching existiert, so daß die zugehörige Implementierung kleinere maximale Verdrahtungskosten besitzt und der entsprechende Task nicht gesperrt ist, so wird die Implementierung $\rho_{\text{best}}^{\text{FUNC}}$ gespeichert. Die Pfadpropagation erfolgt schließlich über die Durchmusterung aller in v konfigurierbaren R-Tasks entsprechend der bereits vorgestellten Kostenfunktion. Der SPS-Algorithmus terminiert, wenn die Fronten aller Suchen leer sind. Abbildung 3.15 skizziert die simultane Propagation der Suchfronten, wobei am Treffpunkt der Fronten, einer konfigurierbaren Zelle, eine Implementierung des Schaltkreisknotens versucht wird.

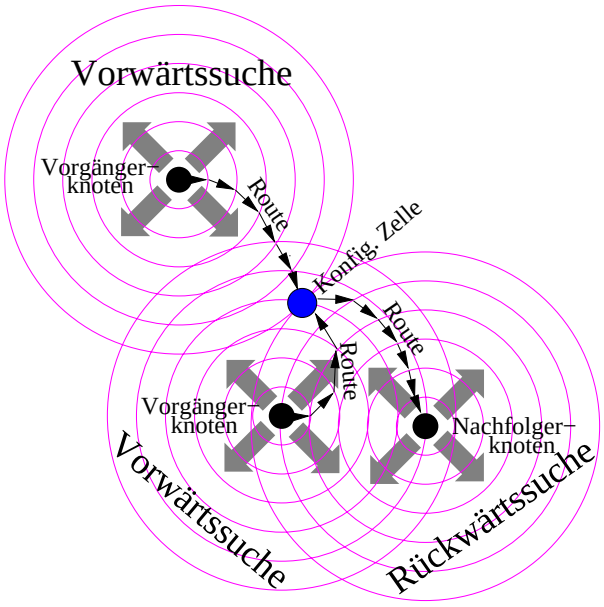


Abbildung 3.15: Prinzip der *Simultanen Pfad Suche*

Die Laufzeit eines Layoutversuches nach Algorithmus 3.9 kann zu einem gegebenen Schaltkreisknoten und einem gegebenen Architekturgraphen nur sehr ungenau abgeschätzt werden, da die Menge K_v der in einer Zelle v vorkonfigurierten Tasks naturgemäß die Adjazenzen im Architekturgraphen bestimmt. Ferner wurden in der praktischen Implementierung Laufzeitkonstanten durch Abfangen von Spezialfällen, wie beispielsweise die Berücksichtigung durch Konfiguration früherer Routen bereits verfügbarer Signale an Zellpins, stark gesenkt. Andererseits muß angemerkt werden, daß eine asymptotische Abschätzung der Laufzeit im Hinblick auf Parameter, wie die Zahl der Eingänge einer konfigurierbaren Zelle oder die Zahl ihrer Tasks insofern wenig Sinn

macht, als, wie bereits früher erwähnt, sich diese Problemgrößen in der Praxis nur in relativ geringen Größenordnungen bewegen.

Das dargestellte Verfahren ist dennoch traktabel, als es sich aufgrund stets nichtnegativer Kosten im Prinzip um nichts anderes, als ein *multiples Dijkstra-Verfahren* [23] zur Bestimmung kürzester Wege handelt, wobei die Suchfronten durch Prioritäts-Warteschlangen realisiert sind. Die Laufzeit zur Bestimmung aller konfigurierbaren R-Tasks einer Zelle ist linear beschränkt durch deren Anzahl, wobei auch hier das *average case* Verhalten durch die Behandlung von Spezialfällen in der Praxis verbessert werden konnte.

Gleichzeitig zur nach dem SPS-Verfahren folgenden Rekonstruktion der tatsächlichen Routen, erfolgt deren Prüfung auf Disjunktheit, also auf Konsistenz der Menge der resultierenden Konfigurationen. Mittels im Zuge des SPS-Verfahrens entsprechend gesetzter Kantensignaturen laufen die beiden Vorgänge jedoch in Linearzeit der Pfadlänge ab. An dieser Stelle soll auf diese, hinsichtlich des Prinzips recht einfache Technik jedoch nicht weiter eingegangen werden.

Konfliktierende Routen können auftreten, wenn es aufgrund der berechneten Knoten- oder Netzimplementierungen zu Taskinkonsistenzen in einer konfigurierbaren Zelle kommt oder, als Spezialfall hiervon, Kanten des Architekturgraphen von mehr als einer Route angefordert werden. In diesem Falle wird der bereits im Rahmen der Kostenfunktion erwähnte Straffaktor dieser Kante (bzw. der Nachfolgerkante der konfliktierenden Zelle) erhöht und eine Re-Iteration des SPS-Verfahrens angestoßen. Hierzu werden alle Suchen re-initialisiert und die Methode aus Algorithmus 3.9 erneut aufgerufen.

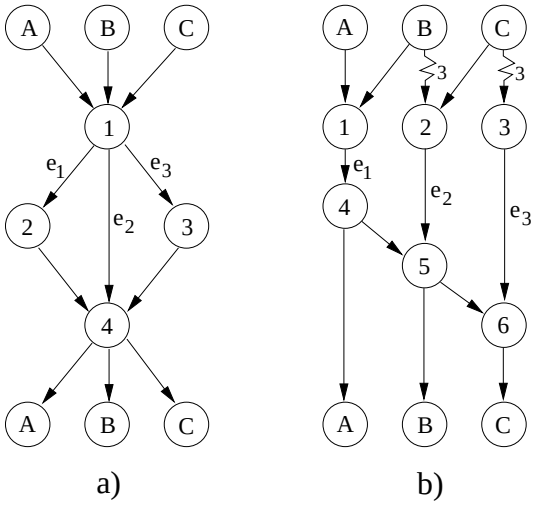


Abbildung 3.16: Beispiele für Konflikte ersten und zweiten Grades

Hinsichtlich der Art der auftretenden Routingkonflikte kann man unterschiedliche Grade unterscheiden. Abbildung 3.16 zeigt hierzu beispielhaft zwei Architekturszenarien. Es seien Routen für die drei kritischen Netze A, B und C zu ermitteln, d.h. es sei: $RDC(A) = RDC(B) = RDC(C) = 1$. Jede konfigurierbare Zelle könne jedes einlaufende Signal zu jedem Ausgang routen, wobei für jeden R-Task t gelte: $\delta_t = 1$. Die Kantenkosten sind initial gleich Null.

- **Konflikt ersten Grades** (Abbildung 3.16a)
  Im Beispiel der Abbildung berechnet SPS für alle drei Netze kürzeste Routen über die Kante e_2. Vor der ersten Re-Iteration wird die Kante e_2 bestraft und die Netzrouten weichen auf die Kante e_1 oder e_3 aus. Unter weiteren Konflikten werden auch diese bestraft, wobei mit jeder Re-Iteration der Straffaktor p_u erhöht wird. Durch den Zufallswert γ der Kostenfunktion, welcher für jede Kante und für jedes Netz unterschiedlich ausfällt, divergieren jedoch die Kosten der drei Kanten e_1 bis e_3 für die einzelnen Suchen mit jeder Re-Iteration immer stärker, so daß Routen über die drei Kanten für jedes Netz jeweils unterschiedliche Kosten verursachen und schließlich disjunkt gewählt werden müssen.

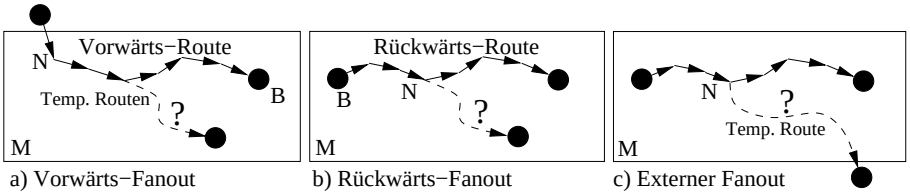
- **Konflikt zweiten Grades** (Abbildung 3.16b)
  Besitzen im Beispiel der Abbildung die bislang gefundenen kürzesten Wege der Netze B bzw. C zu den Knoten 2 bzw. 3 bereits die Kosten 3, so berechnet SPS für die Netze A und B Pfade über die Kante e_1, für das Netz C einen Pfad über die Kante e_2. Durch Bestrafung der Kante e_1 weicht Netz B in der Re-Iteration auf einen Pfad über Kante e_2 aus, konfliktiert aber dort mit Netz C. Erst durch eine Bestrafung der Kante e_2 wird Netz C in einer weiteren Re-Iteration über Kante e_3 geführt, so daß die nun resultierenden Pfade aller drei Netze disjunkt sind.

Daneben können natürlich auch Konstellationen von Konflikten auftreten, die durch eine Bestrafung von Kanten nicht auflösbar sind. Dies ist insbesondere dann der Fall, wenn zwei Terminale unterschiedlicher Netze lediglich über eine Route erreichbar sind, welche vom SPS-Verfahren dann aber für beide Netze angefordert wird. Aus diesem Grunde ist die Zahl der Re-Iterationen wegen Verdrahtungskonflikten beschränkt. Bei Erreichen der Schranke wird die ermittelte Implementierung des Knotenkandidaten B gesperrt und ein neuer Implementierungsversuch gestartet.

Während sich durch unterschiedliche Suchen ermittelte Routen bei SPS also durchaus blockieren können, wurde mit Satz 2.2 jedoch bereits bewiesen, daß sich ein im Rahmen einer Suche berechneter Pfad hingegen nicht selbst blockieren kann.

Multiterminalnetze

Da es sich bei Netzen eines Schaltkreises formal um Hyperkanten handelt und der SPS-Algorithmus eine Quelle/Ziel-Pfadsuche für jedes Netz durchführt, können nach der Terminierung von SPS im Falle multiterminaler Int-Netze und IntExt-Netze noch zu verdrahtende Terminale verbleiben. Solche noch zu berechnenden *Fanout-Routen* können nur in exakt drei Fällen auftreten, die in Abbildung 3.17 illustriert sind:



a) Vorwärts−Fanout b) Rückwärts−Fanout c) Externer Fanout

Abbildung 3.17: Problem von *Fanout-Routen*

a) Netz N wurde in Makro M durch eine Vorwärtssuche ausgehend von der Peripherie zur Implementierung des Schaltkreisblockes B verdrahtet. Es existiert jedoch noch mindestens ein weiteres Zielterminal von N.

b) Netz N wurde durch eine Rückwärtssuche ausgehend von einem seiner Zielterminale an die Implementierung von B verdrahtet. Es existiert jedoch noch ein weiteres Zielterminal von N.

c) Netz N wurde durch eine Vorwärts- oder Rückwärtssuche verdrahtet. Jedoch besitzt N noch ein nicht in M plaziertes Zielterminal. Somit ist eine temporäre Route zu berechnen.

Zur Lösung des Problems der Fanout-Routen wurde ein Ansatz zur Expansion konfigurierter Pfade entwickelt, welcher auf der Idee von Prim's Algorithmus zur Bestimmung minimaler Spannbäume basiert [62].

In allen dreien der obigen Fälle, wird dazu die Suchfront einer *Pfadexpansions-Suche* mit allen jenen Kanten des Architekturgraphen initialisiert, für die ein R-Task existiert, der von der bereits berechneten Route des Netzes N auf diese Kante verzweigt. In den Fällen a) und b) wird eine Pfadsuche zu den noch nicht verdrahteten Zielterminalen von N durchgeführt, im Fall c) wird nach einem kostengünstigsten Weg zur Peripherie gesucht. Ein gefundener Weg wird konfiguriert, das Verfahren unter Einbeziehung der gefundenen Fanout-Route für eventuelle weitere unverdrahtete Terminale wiederholt.

Konnte aber eine Fanout-Route mangels freier Verdrahtungsressourcen nicht konstruiert werden, so ist die durch SPS ermittelte Implementierung für B ungültig. In diesem Falle wird die Implementierung für B gesperrt, die in der letzten Iteration ermittelten Routen werden dekonfiguriert und SPS wird neu gestartet. Die maximale Anzahl von Re-Iterationen ist durch eine festgelegten Schranke nach oben begrenzt, die als Bruchteil der Größe des Architektursegments empirisch gewählt wurde.

Eine Sperrung von Logik- oder Speichertasks hat sich auch im Falle einer Nichtkonvergenz von SPS aufgrund nichtdisjunkter Routen als vorteilhaft erwiesen. Damit wird also die Anzahl der Re-Iterationen in erster Instanz aufgrund konfliktierender Routen und in zweiter Instanz aufgrund der Sperrung von Ressourcen beschränkt. Erst wenn die zweite Schranke erreicht wurde, wird der Schaltkreisknoten als „nicht implementierbar“ betrachtet und abgelehnt.

3.3.4 Layout-Beispiele

Um die Funktionsweise des vorgestellten Makrogenerierungsverfahrens zu illustrieren, ohne die in Kapitel 5 präsentierten Ergebnisse vorwegzunehmen, werden in diesem letzten Unterabschnitt lediglich die Implementierungen einiger Beispiel-Schaltkreise mittels eines universellen *LayoutViewers* visualisiert. Als zugrundeliegende Architektur dient dabei jeweils die bereits in Abschnitt 2.4.1 vorgestellte *einfache Maschenstruktur* mit der konfigurierbaren Zelle *CSR0202V001* (siehe Abbildung 2.14).

Die Schaltkreise wurden jeweils vollständig innerhalb eines Architektursegments implementiert, wobei hinsichtlich der Reihenfolge der Implementierung der Schaltkreisknoten eine Heuristik angewandt wurde: Für jeden Primärausgang des Schaltkreises

wurde eine (Rückwärts-)Breitensuche gestartet; die Reihenfolge der Implementierung bestimmte sich sodann durch die Folge des erstmaligen Besuchs der Schaltkreisknoten.

Die Implementierung eines Schaltkreises wird eingeleitet, indem der erste Schaltkreisknoten der berechneten Implementierungsreihenfolge in einem zentralen Modul eines Architektursegments vorgegebener Dilatation plaziert und mittels temporärer Routen verdrahtet wird. Dabei genügt es, eine gültige Implementierung der Knotenfunktion durch Iterieren über alle Logikressourcen lediglich dieses einen Moduls zu suchen, denn ist in einem unkonfigurierten Modul keine gültige Implementierung für jeden beliebigen Knoten des Schaltkreises zu finden, so ist auch der Schaltkreis auf der gegebenen Architektur offensichtlich nicht implementierbar.

Beispiel 1

Die Abbildung 3.18 zeigt das erste Beispiel: ein Schaltkreis mit drei Eingängen und zwei Ausgängen, sowie fünf Look-Up-Tables. Als Implementierungsreihenfolge wurde mittels genannter Heuristik die Knotensequenz $(7, 8, 11, 12, 13)$ ermittelt.

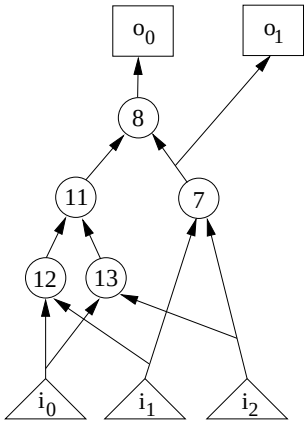


Abbildung 3.18: Schaltkreis *b1*

Die Folge der Abbildungen 3.19a) bis 3.19e) zeigt die sukzessive Implementierung dieses Schaltkreises in einem Architektursegment von 3×3 Modulen[4], wobei die Skalierung der Kosten temporärer Routen den Wert $\mu = 0.1$ erhielt.

Die Anzahl der konfiguierbaren Zellen im gewählten Architektursegment beträgt 3 · 3 · 4 = 36. Tatsächlich genutzt werden vom resultierenden Layout nur 24 konfigurierbare Zellen. Mit dem in Abschnitt 2.5.2 vorgestellten Kostenmodell ergeben sich damit für die Implementierung $\mathcal{I}$ aus Abbildung 3.19e):

$$\begin{aligned} A_{\mathcal{I}}^{\square} &= 36 \cdot A_{\text{CSR0202V001}} = 1350 \\ A_{\mathcal{I}} &= 24 \cdot A_{\text{CSR0202V001}} = 900 \\ A_{\mathcal{I}}^{\text{use}} &= \frac{900}{1350} = \frac{2}{3} \\ D_{\mathcal{I}} &= 11 \end{aligned}$$

[4]Die Abbildungen 3.19a) bis 3.19d) zeigen aus Platzgründen hier nur die *genutzten* Module des Architektursegments.

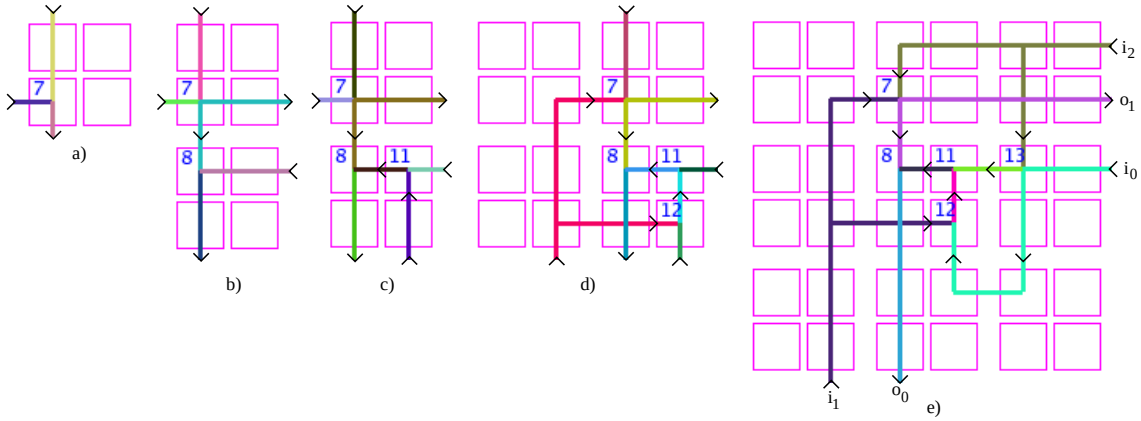


Abbildung 3.19: 3×3-Layout des Schaltkreises *b1*

Der Delaywert $D_{\mathcal{I}}$ wurde hierbei nach dem *Zellenmodell* ermittelt und ergibt sich aus dem kombinatorischen Pfad zwischen Eingang i_0 und Ausgang o_0 über die Knotenfolge $(12, 11, 8)$ des Layouts[5]. Für die *Ideal-Architektur* zum Schaltkreis aus Abbildung 3.18 berechnen sich die entsprechenden Werte wie folgt:

$$\begin{aligned} \hat{A}_{\mathcal{C}} &= 5 \cdot A_{\text{2-LUT}} + 12 \cdot A_{\text{2-CMUX}} = 93.5 \\ \hat{D}_{\mathcal{C}} &= 4 \cdot D_{\text{2-MUX}} + 3 \cdot D_{\text{2-LUT}} = 8 \end{aligned}$$

Wir erhalten für diese Implementierung $\mathcal{I}$ somit Kosten- und Delayoverhead-Werte von:

$$\begin{aligned} A_{\mathcal{I}}^{\text{ov}} &= \frac{900}{93.5} = 9.62567 \\ D_{\mathcal{I}}^{\text{ov}} &= \frac{11}{8} = 1.375 \end{aligned}$$

Der Anteil der für die Implementierung von Schaltkreisknoten genutzten Zellen in der *Bounding Box* beträgt 13%, jener der für Verdrahtung genutzten Zellen 53%.

Beispiel 2

Nach dem Benchmark-Schaltkreis des vorigen Beispiels betrachten wir nun einen *Modell-Schaltkreis*, also einen Schaltkreis von manuell gestalteter Struktur. Der in Abbildung 3.20 dargestellte Schaltkreis *bigfan32* besitzt vier Eingänge, 32 Ausgänge und 34 Look-Up-Tables, wobei zwei Look-Up-Tables den relativ hohen Fanout von 32 aufweisen. Im Zuge der sukzessiven Implementierung der Schaltkreisknoten werden in jedem Schritt für die beiden Fanout-Netze jeweils eine definitive Route berechnet und für die noch nicht implementierten Knoten pro Netz ferner eine temporäre Route.

Die unter der Dilatation $(18, 2)$ ermittelte Implementierung von *bigfan32* ist in Abbildung 3.21 visualisiert. Sie zeigt eine relativ hohe Ausnutzung der *Bounding Box*, was auch durch die nachfolgenden Werte bestätigt wird.

[5]Man beachte, daß in der Knoten 13 implementierenden Zelle das Netz vom Eingang i_0 über einen Fanout-Task weiter zu Knoten 12 verdrahtet wird.

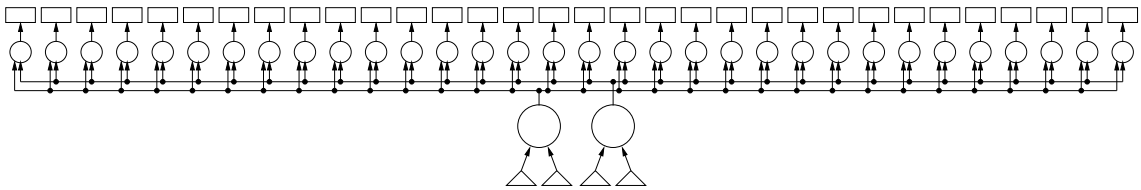


Abbildung 3.20: Modell-Schaltkreis *bigfan32*

Abbildung 3.21: 18×2-Layout des Modell-Schaltkreises *bigfan32*

$$\begin{aligned} A_{\mathcal{I}}^{\square} &= 18 \cdot 2 \cdot 4 \cdot A_{\text{CSR0202V001}} = 5400 \\ A_{\mathcal{I}} &= 136 \cdot A_{\text{CSR0202V001}} = 5100 \\ A_{\mathcal{I}}^{\text{use}} &= \frac{5100}{5400} = \frac{17}{18} \\ D_{\mathcal{I}} &= 74 \end{aligned}$$

Die hohe Ausnutzung resultiert in diesem Falle offensichtlich auch aus einem relativ hohen Nutzungsanteil für Logik von 23%. Die Werte der Ideal-Architektur skizzieren jedoch insbesondere hinsichtlich des Delays die „Kehrseite der Medaille“:

$$\begin{aligned} \hat{A}_{\mathcal{C}} &= 34 \cdot A_{\text{2-LUT}} + 100 \cdot A_{\text{2-CMUX}} = 691 \\ \hat{D}_{\mathcal{C}} &= 3 \cdot D_{\text{2-MUX}} + 2 \cdot D_{\text{2-LUT}} = 5.5 \end{aligned}$$

In der Tat zeigt damit das Beispiel 2, wie durch den hohen Fanout zweier Netze auch der *Delay-Overhead* der Schaltkreis-Implementierung im Layout stark anwachsen kann, denn die beiden entsprechenden Netze müssen an alle ihre Zielknoten herangeführt werden, welche aufgrund der ungünstigen schlauchförmigen Dilatation des Architektursegments jedoch sehr weit auseinander liegen.

$$\begin{aligned} A_{\mathcal{I}}^{\text{ov}} &= \frac{5100}{691} = 7.38061 \\ D_{\mathcal{I}}^{\text{ov}} &= \frac{74}{5.5} = 13.4545 \end{aligned}$$

Beispiel 3

Die Struktur eines Binärbaums der Tiefe 5 besitzt der Modell-Schaltkreis *bintree32* unseres letzten Beispiels in Abbildung 3.22. Der Schaltkreis besitzt 32 Eingänge, einen Ausgang und 31 Look-Up-Tables.

Die in Abbildung 3.23 dargestellte Implementierung von *bintree32* auf einem Architektursegment der Dilatation $(9, 8)$ zeigt unter anderem, daß – bis auf zwei Ausnahmen – alle Eingangspins der Peripherie durch Netze belegt sind. Andererseits bleiben relativ viele konfigurierbare Zellen ungenutzt, da aufgrund der zahlreichen Eingänge häufig nur „in das Architektursegment hineinführende“ Pfade benötigt werden. Zur Bereitstellung des hierbei benötigten Platzes, lieferte unser Plazierungs- und Verdrahtungsver-

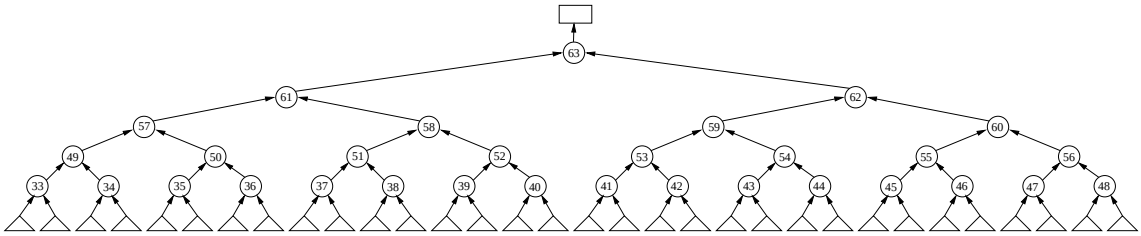


Abbildung 3.22: Modell-Schaltkreis *bintree32*

fahren ein Layout mit einer, schematisch betrachtet, für Binärbäume typischen Anordnung der Schaltkreisknoten: das Layout ähnelt einem sogenannten *H-Baum*.

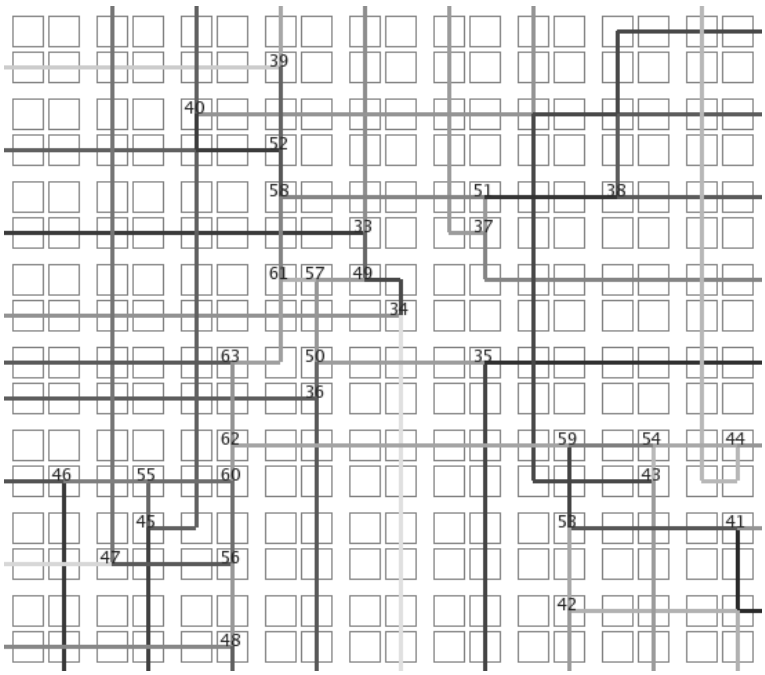


Abbildung 3.23: 9×8-Layout des Modell-Schaltkreises *bintree32*

Die numerischen Resultate der Implementierung verhalten sich entsprechend obiger Beobachtungen:

$$\begin{aligned} A_{\mathcal{I}}^{\square} &= 9 \cdot 8 \cdot 4 \cdot A_{\text{CSR0202V001}} = 10800 \\ A_{\mathcal{I}} &= 209 \cdot A_{\text{CSR0202V001}} = 7837.5 \\ A_{\mathcal{I}}^{\text{use}} &= \frac{7837.5}{10800} = 0.725694 \\ D_{\mathcal{I}} &= 38 \end{aligned}$$

Die Ausnutzung der *Bounding Box* ist mit rund 73% wieder etwas geringer, doch aufgrund der zahlreichen Routen zur Peripherie besser, als die Anordnung der Schaltkreisknoten suggeriert. Der eigentliche „Effekt“ dieses Beispiels wird jedoch erst beim Be-

trachten der Werte des Ideal-Schaltkreises bzw. beim Berechnen des Kosten- und Delay-Overheads sichtbar:

$$\begin{aligned}
\hat{A}_{\mathcal{C}} &= 31 \cdot A_{\text{2-LUT}} + 63 \cdot A_{\text{2-CMUX}} = 545.5 \\
\hat{D}_{\mathcal{C}} &= 6 \cdot D_{\text{2-MUX}} + 5 \cdot D_{\text{2-LUT}} = 13 \\
A_{\mathcal{I}}^{\text{ov}} &= \frac{7837.5}{545.5} = 14.3676 \\
D_{\mathcal{I}}^{\text{ov}} &= \frac{38}{13} = 2.92308
\end{aligned}$$

Scheinbar resultiert aus der weiten Verzweigung des Schaltkreises *bintree32* ein starker Bedarf an Verdrahtungsressourcen, wodurch der *Kosten-Overhead* auch vergleichsweise hoch ausfällt.

Mit den vorangegangenen drei Layout-Beispielen schließen wird dieses Kapitel ab und behalten uns weitere Versuche mit den vorgestellten Verfahren für das Kapitel 5 vor.

Kapitel 4

Floorplanning und Verdrahtung von Schaltkreismakros

Zwei weitere, interagierende Verfahren unseres generischen Layoutsystems für konfigurierbare Architekturen zur Implementierung von Schaltkreismakros werden im vorliegenden Kapitel dieser Arbeit betrachtet.

Nach wiederum einer einführenden Problembetrachtung wird dazu ein neues Verfahren zur sukzessiven Anordnung von Makros vorgestellt, das gekoppelt ist mit einem Verdrahtungssystem, wobei jeweils eine vollständige und delayoptimierte Implementierung einer gegebenen Menge von Makros auf einer Zielarchitektur angestrebt wird.

4.1 Problembetrachtung

Die beiden, der Implementierung von Makros zugrundeliegenden Problemkreise, bestehen nach Gesagtem also in den folgenden Punkten:

1. Ermittlung einer „günstigen“ Anordnung aller Makros auf einem Segment geeigneter Größe der Zielarchitektur *(Floorplanning)*.
2. Berechnung gültiger Implementierungen für alle zwischen den angeordneten Makros verlaufenden Netze.

Da das in unserem Falle vorliegende Anordnungsproblem in gewisser Hinsicht von der Definition eines allgemeinen Floorplanning-Problemes abweicht, wie wir noch sehen werden, soll im folgenden Unterabschnitt zunächst die gewählte Begriffsfassung motiviert werden.

4.1.1 Einordnung

Hinsichtlich der Anordnung blockförmiger Subschaltkreisimplementierungen unterscheidet man prinzipiell zwischen Problemen, bei denen Blockgröße und -form *fixiert* sind, sowie Problemen, bei denen sie *variabel* sind. Während die Probleme des ersten

Falles als allgemeine *Plazierungs-* oder *Packungsprobleme* bezeichnet werden, spricht man im zweiten Fall auch von *Floorplanning-Problemen.*

Floorplanning-Probleme können auch als hierarchische Plazierungsprobleme interpretiert werden, indem mit jeder Stufe der Hierarchie – durch die Freiheitsgrade in der Konstruktion einer Teillösung – kleinere Blöcke zu größeren Blöcken variabler Abmessungen kombiniert werden können. Formal stellt sich das Problem wie folgt dar [70]:

Definition 4.1 (Floorplanning-Problem)
Sei $B = \{B_1, \ldots, B_n\}$ eine Menge von Blöcken und sei jeweils $h_i \in \mathbb{R}$ die Breite und $v_i \in \mathbb{R}$ die Höhe von Block B_i. Sei $N = \{N_1, \ldots, N_m\}$ die Menge der Netze, welche die Blöcke aus B verbinden. Dann besteht das *Floorplanning-Problem* in der Bestimmung einer Menge von Rechtecken $R = \{R_1, \ldots, R_n\}$ in der Plazierungsebene so, daß gilt:

1. Rechteck R_i hat mindestens die Breite h_i und mindestens die Höhe v_i, das heißt, Block B_i kann in R_i plaziert werden.
2. Für $i \neq j$ ist $R_i \cap R_j = \emptyset$, das heißt, die Rechtecke überlappen sich nicht.

Optimierungskriterien sind die Minimierung der Summe der Längen der Realisierungen der Netze aus N, sowie die Fläche des R umgebenden kleinsten Rechteckes.

Der Floorplanning-Fall für Makros auf zweidimensionalen feldprogrammierbaren Architekturen mit Insel-Struktur, beispielsweise, definiert sich sehr ähnlich – lediglich die Plazierungsebene stellt hier ein diskretes Gitter von Logikblöcken dar, wobei die Breiten- und Höhenwerte der Makroblöcke Vielfache von Logikblöcken sind (vgl. Abbildung 4.1).

Für diese Art von Anordnungsproblemen wurde inzwischen eine Vielzahl von Algorithmen entwickelt, über die in Unterabschnitt 4.1.2 eine kurze Übersicht gegeben wird. Durch die Diskretisierung nunmehr zu einem kombinatorischen Optimierungsproblem geworden, sind auch die Nebenbedingungen des 2D-FPGA-Floorplanning-Problems dabei effizient abzuprüfen.

Im vorliegenden Falle gestaltet sich das Anordnungsproblem jedoch in wesentlichen Punkten abweichend. Die Freiheitsgrade unseres Architekturmodells lassen innerhalb eines Moduls auch die Konstruktion inhomogener Subarchitekturen zu, weshalb wir Floorplanning primär nur auf der Ebene von Modulen betreiben können. Somit stellt unsere „Plazierungsebene“ vielmehr ein mehrdimensionales Gitter dar, wobei sich auf den Gitterpunkten nicht Logikblöcke, sondern Architekturmodule befinden, die ihrerseits mehrere konfigurierbare Zellen enthalten können. Die Dilatation eines entsprechend des vorangegangenen Kapitels generierten Subschaltkreismakros impliziert nun zwar im Prinzip eine Quaderform des Makros, allerdings müssen nicht notwendigerweise alle Zellen und innerhalb diesen wiederum nicht alle Logik- und Verdrahtungsressourcen vollständig ausgenutzt sein. Aus diesem Grunde sollen im Rahmen des hier vorliegenden Anordnungsproblemes auch *Überlappungen* von Makros betrachtet werden, indem der Kompatibilitätsbegriff für Taskmengen als Kriterium für gültige Überlappungen herangezogen wird. Eine formale Definition des Problems wird in Abschnitt 4.1.3 gegeben.

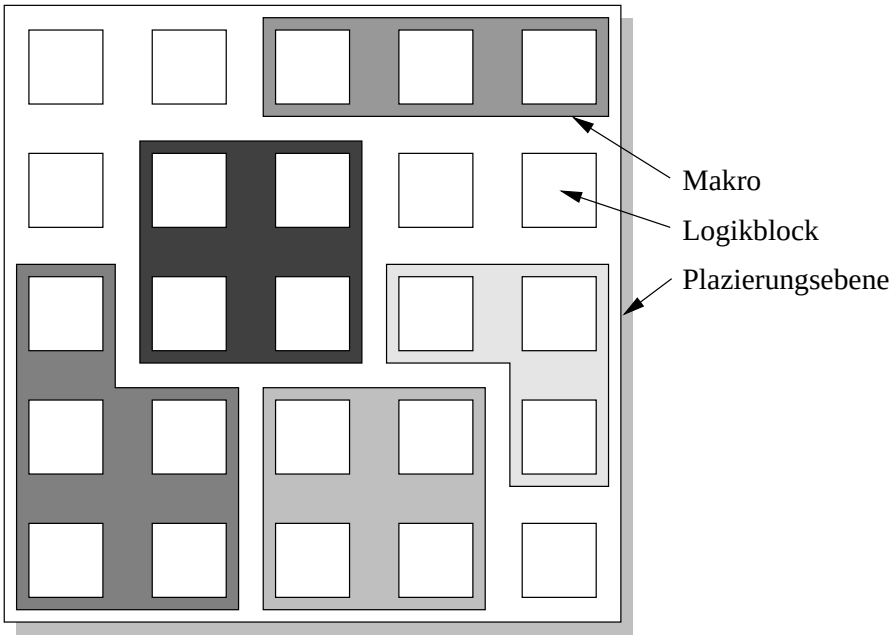


Abbildung 4.1: Beispiel eines allgemeinen 2D-FPGA-Floorplans

4.1.2 Frühere Ansätze

Eine Klassifikation allgemeiner Floorplanning-Verfahren ist beispielsweise in [70] zu finden. Die Vielfalt bislang entwickelter Methoden reicht von auf Pfadeliminationen in Bedingungsgraphen basierenden Verfahren über Methoden des Integer Programmings, Verfahren der Dualisierung bis hin zu genetischen Algorithmen. Häufig angewandt werden auch hierarchische Methoden, die eine sukzessive Partitionierung der Blockmenge durchführen. Während die genannten Methoden im wesentlichen die Fläche des den resultierenden Floorplan umgebenden kleinsten Rechtecks zu minimieren suchen, sind für den Anwendungsbereich des performanzoptimierten Schaltungsentwurfs vor allem die sogenannten *Timing Driven Floorplanning* Verfahren interessant [90]. Prinzipiell ebenfalls auf den genannten Verfahren basierend, gehen hier zusätzlich Abschätzungen über die zu erwartenden Leitungsverzögerungen in die Kostenfunktion ein.

Bei feldprogrammierbaren Architekturen treten Floorplanning-Verfahren auf, wenn es um die Anordnung von Subschaltkreismakros geht. Aus den bislang publizierten Verfahren, welche beinahe ausnahmslos auf der bekannten kommerziellen zweidimensionalen Architektur mit Insel-Struktur arbeiten, sollen im folgenden vier Repräsentanten vorgestellt werden.

Ein Floorplanning-Ansatz über das aus dem allgemeinen VLSI Design bekannten kräfteorienterte Plazierungsverfahren *(Force-directed placement)* wird in [71] präsentiert, wobei in das Kräftemaß neben der Anzahl der Inter-Makro-Netze auch eine schlupfbasierte Kritizität der Netze eingeht.

Für hierarchisch spezifizierte Schaltkreise und Zielarchitekturen versucht das in [47] vorgestellte Verfahren, durch rekursive Zerlegung und durch Kombination von Blöcken

entlang des kritischen Pfades die Hierarchie des Schaltkreises auf die Hierarchie der Zielarchitektur abzubilden.

Der Ansatz von [27] beginnt mit einer am Vernetzungsgrad orientierten Kombination der Schaltkreismakros zu *Clustern*, verwaltet freie Plazierungsregionen der Zielarchitektur als sogenannte *Buckets* und ermittelt mittels Tabu Suche eine Abbildung der *Cluster* auf *Buckets*.

Auch das bereits in Kapitel 3 vorgestellte *FRONTIER-System* [74] kann, neben ursprünglich einzelnen Logikblöcken, auch Schaltkreismakros in seinem Plazierungsschritt berücksichtigen. Es wendet zunächst Clustering- und Bipartitionierungstechniken auf der Makro-Netzliste an zur Bildung eines Floorplanbaumes *(slicing tree)*, anschließend wird der Baum traversiert und anhand von Abschätzungen über die zu erwartende Verdrahtungslänge, sowie der Blockgröße ein Floorplan im *Bottom-Up*-Verfahren konstruiert. Stellt der nachfolgende Verdrahtungsalgorithmus eine Nichtverdrahtbarkeit fest, wird der ermittelte Floorplan durch *Simulated Annealing* perturbiert, wobei auch eine Relaxation der Anordnung oder die Aufspaltung von Makros in ihre Logikblöcke möglich ist.

Wie bereits motiviert, liegt aufgrund des retargierbaren Architekturmodells dieser Arbeit ein anderer Typus von Floorplanningproblemen zugrunde, weshalb wir im nachfolgenden Unterabschnitt eine Neufassung des Problemes geben.

4.1.3 Reformulierung

Ziel des im vorangegangenen Kapitel 3 betrachteten Verfahrens zur Generierung von Schaltkreismakros war die Partitionierung eines sequentiellen LUT-Schaltkreises, sowie die Berechnung gültiger Implementierungen der entsprechenden Subschaltkreise. Dabei wurden Inter-Makro-Netze in Form *temporärer Routen* realisiert. Abbildung 4.2 zeigt die zweidimensionale, schematische Skizze eines generierten Makros.

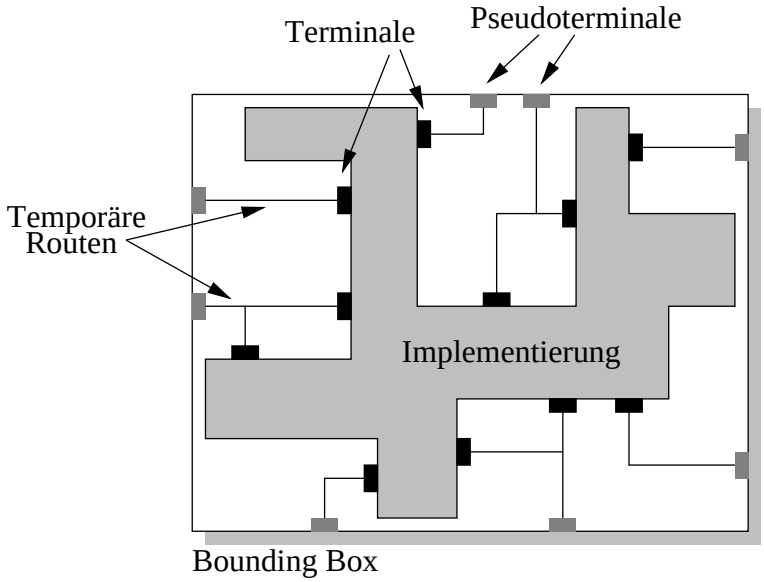


Abbildung 4.2: Skizzierung eines generierten Makros

Die eigentliche Makroimplementierung stellt hierbei einen konfigurierten Bereich innerhalb eines Segments der Zielarchitektur dar, welches auch die *Bounding Box* des Makros markiert. Das Segment besteht wiederum aus einem oder mehreren Modulen des Architekturgraphen. Während die eigentlichen Netzterminale des Makros an beliebiger Stelle innerhalb der *Bounding Box* liegen können, wurde im Zuge der Makrogenerierung die Verfügbarkeit der Netze außerhalb durch temporäre Routen zur Peripherie des Makros garantiert. Da die Inter-Makro-Netze jedoch anhand der ermittelten Anordnung der Makros zu verdrahten sind, werden die temporären Routen vor dem Floorplanning wieder dekonfiguriert.

Definition 4.2 (Terminal/Pseudoterminal)
Sei B ein K-Baum von temporären Routen eines Netzes N auf einem Architektursegment $\mathcal{A}^x$. Dann heißt $e_N^{\text{PTerm}} = (e, N)$ das *Pseudoterminal* des Netzes N, wobei e die eindeutige, zur Peripherie von $\mathcal{A}^x$ gehörende Kante von B darstellt. Die Menge E_N^{Term} der *Terminale* des Netzes N sei dann gegeben durch:

$$E_N^{\text{Term}} = \left\{ (e, N) \mid e \in \big(Q(B) \cup Z(B)\big) \setminus \{e_N^{\text{PTerm}}\} \right\}$$

Für ein Terminal oder Pseudoterminal t gebe ferner $o(t) \in \{\text{Q}, \text{Z}\}$ dessen Orientierung bezüglich des Netzes N an.

Für das weitere Vorgehen fassen wir ein Makro eines sequentiellen LUT-Schaltkreises formal wie folgt auf:

Definition 4.3 (Makro/Reduziertes Makro)
Sei $\mathcal{C} = (V, E)$ ein sequentieller LUT-(Sub-)Schaltkreis, $\mathcal{A}$ die Beschreibung einer r-dimensionalen Architektur und $\tilde{\mathcal{I}} = (K, \rho^{\text{FUNC}}, \rho^{\text{NET}})$ eine Implementierung von $\mathcal{C}$ auf $\mathcal{A}^x$ für eine Dilatation $x \in \mathbb{N}^r$. Dann bezeichnet $\tilde{M} = (\mathcal{A}^x, \tilde{\mathcal{I}}, \tilde{T})$ ein *Makro* von $\mathcal{C}$ bezüglich der Architektur $\mathcal{A}$, wobei $\tilde{T}$ die Menge der Pseudoterminale aller K-Bäume aus K darstellt. Ist $\mathcal{I}$ die aus $\tilde{\mathcal{I}}$ erhaltene Implementierung, wenn alle temporären Routen entfernt werden, dann bezeichnet $M = (\mathcal{A}^x, \mathcal{I}, T)$ ein *reduziertes Makro,* wobei T die Menge der Terminale aller jener Netze ist, deren temporäre Routen entfernt wurden.

Die Abbildung 4.3 skizziert das zum Makro aus Abbildung 4.2 gehörende reduzierte Makro. Für den Rest dieses Kapitels gehen wir beim Begriff des Schaltkreismakros stets von seiner reduzierten Implementierung aus.

Zunächst setzen wir den aus Abschnitt 2.3.5 bereits bekannten Begriff der Kompatibilität von Implementierungen auf konfigurierbaren Zellen für Module eines Architekturgraphen fort:

Definition 4.4 (Modulkompatible Implementierungen)
Sei I eine Menge von Schaltkreis-Implementierungen auf einem Modul der durch $\mathcal{A} = (\mathcal{P}, V, E, \varphi, Q, Z, r, \tau)$ beschriebenen Architektur. Dann bezeichnen wir die Menge I als *modulkompatibel* bezüglich $\mathcal{A}$, wenn für alle konfigurierbaren Zellen $v \in V$ gilt: die Einschränkungen aller Implementierungen aus I auf v sind kompatibel.

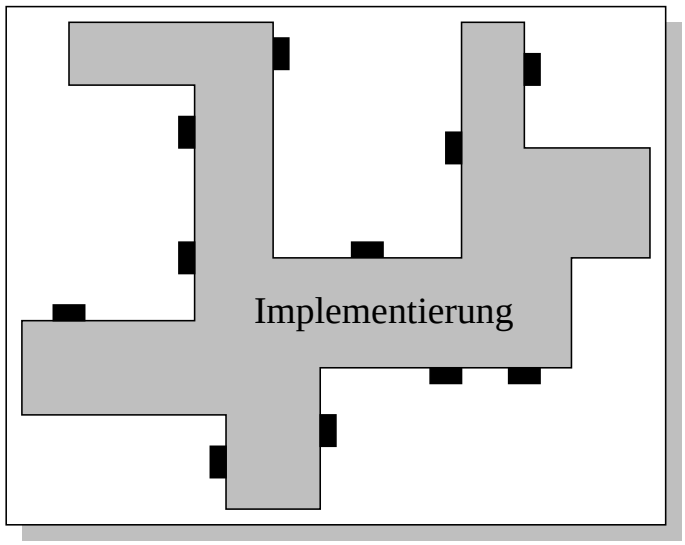


Abbildung 4.3: Reduziertes Makro

Bei einem Floorplan wird nun hinsichtlich der Überlappung von Makros gerade diese Modul-Kompatibilität gefordert. Auffallend hierbei ist, daß diese Eigenschaft aufgrund der Isomorphie der Module bereits anhand der Architekturbeschreibung feststellbar ist. Dies wird im später vorgestellten Floorplanning-Verfahren auch Berücksichtigung finden.

Definition 4.5 (Einschränkung einer Implementierung auf Module)
Sei $M = (\mathcal{A}^x, \mathcal{I}, T)$ ein Makro auf einer r-dimensionalen Architektur $\mathcal{A}^x$. Für ein $z \in \mathbb{N}^r$ sei $\mathcal{I}|_z$ die Einschränkung der Implementierung $\mathcal{I}$ auf das Modul an Position z der Architektur $\mathcal{A}^x$. Falls jedoch $z \notin \mathit{span}(x)$, sei $\mathcal{I}|_z$ die leere Implementierung.

Als Gegenstand der folgenden Betrachtungen definieren wir damit einen Floorplan für LUT-Schaltkreis-Makros wie folgt:

Definition 4.6 (Floorplan für LUT-Schaltkreis-Makros)
Sei $\mathcal{M} = \{M_1, \ldots, M_k\}$ eine Menge von Makros $M_i = (\mathcal{A}^{x_i}, \mathcal{I}_i, T_i)$ auf einer r-dimensionalen Architektur mit Beschreibung $\mathcal{A}$. Dann ist ein *Floorplan* $\mathcal{F}$ der Makros aus $\mathcal{M}$ auf der durch $\mathcal{A}$ beschriebenen Architektur gegeben durch $\mathcal{F} = (\mathcal{M}, \psi, y)$, wobei $\psi : \mathcal{M} \to \mathbb{N}^r$ eine totale Abbildung und $y \in \mathbb{N}^r$ die Dilatation des Floorplans ist, so daß gilt:

1. $\forall_{i \in \{1,\ldots,k\}}\ \psi(M_i) + \mathit{span}(x_i) \subseteq \mathit{span}(y)$
2. $\forall_{z \in \mathit{span}(y)}\ \bigcup_{j \in \{1,\ldots,k\}}\ \mathcal{I}_j|_{z-\psi(M_j)}$ modulkompatibel bezüglich $\mathcal{A}$

Ein Floorplan beschreibt also lediglich eine kompatible Anordnung von Makros. Zur Implementierung des aus den Makros zusammengesetzten Schaltkreises sind jedoch auch Implementierungen der Inter-Makro-Netze zu berücksichtigen, welche anhand einer gegebenen Anordnung erst zu bestimmen sind.

Definition 4.7 (Makro-Verdrahtungsproblem)
Sei $\mathcal{M} = \{M_1, \ldots, M_k\}$ eine Menge von Makros $M_i = (\mathcal{A}^{x_i}, \mathcal{I}_i, T_i)$ auf einer Architektur mit Beschreibung $\mathcal{A}$. Sei $\mathcal{N} = \{N_1, \ldots, N_m\}$ die Menge der Inter-Makro-Netze. Zu einem gegebenen Floorplan $\mathcal{F} = (\mathcal{M}, \psi, y)$ besteht dann das *Makro-Verdrahtungsproblem* in der Bestimmung einer Implementierung $\mathcal{I}_N$ jedes Netzes $N \in \mathcal{N}$ auf $\mathcal{A}^y$, so daß für alle $z \in \mathit{span}(y)$ gilt:

$$\bigcup_{j \in \{1,\ldots,k\}} \mathcal{I}_j|_{z-\psi(M_j)} \quad \cup \quad \bigcup_{N \in \mathcal{N}} \mathcal{I}_N|_z \text{ ist modulkompatibel bezüglich } \mathcal{A}$$

Wir bezeichnen die Gesamtheit aller zum Floorplan $\mathcal{F}$ und der Lösung seines Makro-Verdrahtungsproblems gehörenden Implementierungen auf $\mathcal{A}^y$ durch $\mathcal{I}_\mathcal{F}$. Das *Floorplanning-Problem für LUT-Schaltkreis-Makros* kann nun wie folgt formuliert werden:

Definition 4.8 (Floorplanning-Problem für LUT-Schaltkreis-Makros)
Sei $\mathcal{M}$ eine Menge von Makros. Dann besteht das *Floorplanning-Problem* für $\mathcal{M}$ in der Bestimmung eines Floorplans $\mathcal{F}$, sowie der Lösung des zugehörigen Makro-Verdrahtungsproblems, so daß das Delay $D_{\mathcal{I}_\mathcal{F}}$ der resultierenden Implementierung $\mathcal{I}_\mathcal{F}$ minimiert wird.

Das Konzept der im Rahmen der vorliegenden Arbeit entwickelten Verfahren zur Implementierung von Makros wird nun im folgenden Abschnitt vorgestellt, während in Abschnitt 4.3 genauer auf die Methode des Floorplannings und in Abschnitt 4.4 auf den Löser der Makro-Verdrahtungsprobleme eingegangen wird.

4.2 Implementierung von Makros

4.2.1 Rahmenalgorithmus

Algorithmus 4.1 gibt einen Überblick über das im Rahmen der vorliegenden Arbeit entwickelte Verfahren zur Bestimmung einer gültigen Implementierung einer Menge von Makros. Die Methode **implement_macros** erhält dabei eine Menge $\{M_1, \ldots, M_k\}$ von Schaltkreismakros und liefert eine Referenz I auf eine gültige Implementierung zurück, falls das Ergebnis des Aufrufes den Wert *true* besitzt.

Algorithmus 4.1 (Rahmenalgorithmus)

```
bool implement_macros(I, M_1, ..., M_k)
{
    Berechne Sortierung π der Menge {M_1, ..., M_k};
    I = M_π(1);
    for i = 2, 3, ..., k do                    // Implementiere alle Makros
    {
        forall Terminalnetze N von M_π(i) do
            Dekonfiguriere temporäre Route von N auf I, falls existent;
        I.expand_area(M_π(i),δ);
```

```
        P = PropertyContainmentSearch(I,M_π(i));
        if P = ∅ then
            return false;
        Berechne Sortierung π' der Menge P = {p_1, ..., p_r};
        j = 1;
        routed = false;
        while not routed und j < r do
        {
            M = create_placement(I,M_π(i),p_π'(j));
            Sei R die Menge der Routingprobleme auf M;
            if R = ∅ then                    // Kein Verdrahtungsproblem zu lösen
                routed = true;
            else                             // Löse Verdrahtungsprobleme
            {
                routed = MacroRouter(M,R);
                if not routed then           // Zweiter Versuch
                {
                    M.create_hull();
                    routed = MacroRouter(M,R);
                }
            }
            j = j + 1;
        }
        if not routed then
            return false;
        else
            I = M;
    }
    return true;
}
```

Zunächst wird, einer Heuristik folgend, welche in Abschnitt 4.2.3 vorgestellt wird, eine Reihenfolge bestimmt, in der die Makros $M_1, \ldots, M_k$ sukzessive implementiert werden. Die Implementierung des Makros $M_{\pi(1)}$ selbst bildet die erste Teillösung I. Mittels des auf der Basis mehrdimensionaler Mustererkennung entwickelten Verfahrens der *Property Containment Search (PCS)*, das in Abschnitt 4.3 erläutert wird, werden danach alle, hinsichtlich Konfigurierbarkeit gültigen Plazierungen eines aktuell betrachteten Makros $M_{\pi(i)}$ bestimmt. Die jeweils erhaltene Menge P potentieller Plazierungen wird mittels der in Abschnitt 4.2.2 vorgestellten Heuristik bewertet, wobei in der aus absteigender Bewertung resultierenden Reihenfolge eine Verdrahtung des entsprechenden Makros versucht wird.

Zur Verdrahtung werden alle aus der gewählten Plazierung $p_{\pi'(j)}$ des Makros $M_{\pi(i)}$ sich ergebenden Verdrahtungsprobleme, also die Menge R der zu verdrahtenden Netze, sowie deren Terminale bezüglich der soeben konstruierten Anordnung M ermittelt. Der

Fall $R = \emptyset$ tritt hierbei genau dann auf, wenn das Makro so plaziert wurde, daß alle zu verbindenden Terminale bereits aufeinander liegen. Ansonsten wird der in Abschnitt 4.4 beschriebene Makroverdrahter aufgerufen. Scheitert dieser, wird ein weiterer Versuch nach einer Vergrößerung des Architektursegmentes gestartet. Auf die beiden entsprechenden Methoden *create_hull* und *expand_area* in Algorithmus 4.1 wird jedoch später, in Abschnitt 4.3.2 genauer eingegangen.

Konnte das Makro $M_{\pi(i)}$ für keine Plazierung $p_{\pi'(j)}$ vollständig verdrahtet werden, so bricht das Verfahren sofort ab. Ansonsten wird die bis dahin berechnete gültige Implementierung der Makros $M_{\pi(1)}$ bis $M_{\pi(i)}$ als neue Teillösung I des Gesamtproblems interpretiert.

4.2.2 Heuristik zur Bewertung von Plazierungen

Gegenstand der Bewertung stellt eine Teillösung I des Floorplanning-Problems und ein zu plazierendes Makro M_i im Vorfeld der Verdrahtung der zwischen diesen verlaufenden Netze dar. Das *PCS*-Verfahren liefert in Algorithmus 4.1 eine Liste P legaler Plazierungspositionen für das Makro M_i auf der Teillösung I. Da die Verdrahtung der Netze einen zeitintensiven Faktor darstellt, scheint es ziemlich ungünstig, zu jeder Plazierung $p \in P$ eine gültige Implementierung zu berechnen und sie zu bewerten. Somit liegt es nahe, bestimmte Plazierungen aus der Liste P bevorzugt auszuwählen.

Im Hinblick auf den nachfolgenden Verdrahtungsschritt wäre eine Plazierung eines Makros als günstig zu erachten, welche die Länge des längsten zu erwartenden Pfades aller zu verdrahtender Netze zwischen Vorplazierung I und Makro M_i minimiert. Eine exakte Bestimmung der Netzlängen ist jedoch andererseits ohne die Bestimmung konkreter Netzrouten und damit auch die Wahl einer Plazierung für M_i nicht möglich.

Eine Möglichkeit zur Abschätzung des Abstandes von Netzterminalen stellt die maximale Manhattan-Distanz zwischen den Modulen, in denen sich Terminale eines Netzes befinden, dar. Andererseits kann jedoch die tatsächliche Lage der Terminale in den Modulen, je nach Konstruktion des statischen Graphen, in sehr unterschiedlichen Distanzen resultieren; der Fehler der Abschätzung steigt also mit der Länge der Wege im statischen Graphen.

Eine exaktere Abschätzung liefert hingegen die Verdrahtungsdistanz eines Netzes in Gestalt einer unteren Schranke für die minimale Anzahl zu erwartender Routing-Tasks. Mit dem Maximum dieser Werte über alle zu verdrahtenden Netze erhält man somit eine untere Schranke für den Verdrahtungsaufwand der Netze, welche nach dem Zellendelaymodell sogar das minimale resultierende Delay der Verdrahtung eines Makros zu charakterisieren vermag.

Die Bestimmung einer minimalen Anzahl zu erwartender R-Tasks einer Netzroute entspricht jedoch gerade der minimalen Anzahl durchlaufener Kanten im Architekturgraphen und dieses Problem ist für zusammenhängende periodische Graphen bekannt unter der Bezeichnung *minimales m-Pfad-Problem* (vgl. Definition 1.22 in Abschnitt 1.3.2). Wir geben hier nun die adaptierte Definition für den Anwendungsfall konfigurierbarer Architekturen an:

Definition 4.9 (Minimales m-Pfad-Problem)
Sei $\mathcal{A}^x$ eine Architektur zur Beschreibung $\mathcal{A} = (\mathcal{C}, V, E, \varphi, Q, Z, r, \tau)$ und $x \in \mathbb{N}^r$. Sind $(u, y), (v, z) \in V^x$ zwei konfigurierbare Zellen der Architektur und $c : E^x \to \mathbb{R}$ eine Kostenfunktion, dann ist mit $m = z - y \in \mathbb{Z}^r$ das *minimale m-Pfad-Problem* zwischen den Knoten u und v gegeben durch die Bestimmung eines Pfades $p = (w_0, e_0, w_1, e_1, \ldots, e_{s-1}, w_{s-1})$ in $\mathcal{A}^x$ mit $w_0 = (u, y)$ und $w_{s-1} = (v, z)$, so daß $\sum_{i=0}^{s-1} c(e_i)$ minimal.

In Anlehnung an Definition 1.22 läßt sich das *minimale m-Pfad-Problem* auch lediglich anhand des der Architektur zugrundeliegenden statischen Graphen $G = (V, E, \tau)$ mathematisch formulieren, nämlich durch das folgende *Integer Programm* für $n = \#V, k = \#E$:

$$Ax = b \text{ mit } x \geq 0, \tag{4.1}$$

$$\text{so daß } c^T x \text{ minimal} \tag{4.2}$$

$$\{e_i \in E^x \mid x_i > 0\} \text{ spannt einen zusammenhängenden Subgraphen auf} \tag{4.3}$$

wobei $c \in \mathbb{R}^r$ und

$$A = \left(\frac{I_G}{T}\right) = \left(\begin{array}{ccc} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \\ \hline \tau(e_1)_1 & \cdots & \tau(e_k)_1 \\ \vdots & \ddots & \vdots \\ \tau(e_1)_r & \cdots & \tau(e_k)_r \end{array}\right) \text{ mit } a_{ij} = \begin{cases} +1, & \text{falls } e_j \in E_{v_i}^+ \setminus E_{v_i}^- \\ -1, & \text{falls } e_j \in E_{v_i}^- \setminus E_{v_i}^+ \\ 0, & \text{sonst} \end{cases}$$

und

$$b = \left(\begin{array}{c} b_1 \\ \vdots \\ b_n \\ \hline m_1 \\ \vdots \\ m_r \end{array}\right) \text{ mit } b_i = \begin{cases} +1, & \text{falls } v_i = u \\ -1, & \text{falls } v_i = v \\ 0, & \text{sonst} \end{cases}$$

Man beachte, daß die obersten n Zeilen der Matrix A gerade der Inzidenzmatrix I_G des statischen Graphen entsprechen, wobei für Schlingen $e_j = (v_i, v_i)$ der Matrixeintrag $a_{ij} = 0$ gesetzt wird. Der Wert einer Komponente x_i des Lösungsvektors entspricht der Zahl des Auftretens der zugehörigen Kante e_i in einem m-Pfad. Mittels der Einträge $+1$ bzw. -1 der Komponenten b_i des Vektors b wird sichergestellt, daß der Startknoten u einmal mehr verlassen, als besucht wird und der Zielknoten v einmal mehr besucht, als verlassen wird. Die unteren r Zeilen der Matrix A enthalten die Bilder der Transitionsabbildung des statischen Graphen und stellen als Nebenbedingung zusammen mit den unteren r Einträgen des Vektors b sicher, daß zwischen Start- und Zielmodul die Gitterdistanz $m = z - y$ liegt. Mit dem geforderten Zusammenhang des durch die Kanten e_i mit $x_i > 0$ aufgespannten Subgraphen folgt, daß das obige *Integer Programm* einen minimalen m-Pfad auf einer Architektur ausreichender Dilatation liefert [32].

Da wir jedoch nicht an der Berechnung eines konkreten minimalen m-Pfades interessiert sind, sondern lediglich nach einer unteren Schranke für die Anzahl der auf einem solchen Pfad durchlaufenen Kanten suchen, kann auf die Nebenbedingung 4.3 des *Integer Programms* über die Menge zusammenhängender Kanten verzichtet werden. Aus demselben Grund können wir ferner von einer Architektur mit einer zur Bildung eines solchen minimalen m-Pfades ausreichenden Dilatation ausgehen. Definieren wir nun die Kostenfunktion c als Einsvektor $c = (1)^k$, so erhalten wir die gesuchte Schranke $u(u,v,m)$ aus einer Optimallösung $x^{\text{opt}} \in \mathbb{Z}^k, x^{\text{opt}} \geq 0$ des *Integer Programms* gerade durch:

$$u(u,v,m) = \sum_{i=1}^{k} x_i^{\text{opt}}$$

Im allgemeinen kann dieses *Integer Programm* durch einen *Branch & Bound*-Ansatz über seine lineare Relaxation gelöst werden, wobei es auch Spezialfälle gibt, in denen *Branch & Bound* bereits nach der ersten Stufe terminiert. Ein solcher Fall liegt gemäß des Satzes von Hoffmann/Kruskal (Satz 1.1) genau dann vor, wenn die Matrix A vollständig unimodular ist, es sich hinsichtlich des Optimierungsproblemes also um ein *einfaches Integer Programm* handelt.

Bei der Relaxation handelt sich genauer um ein beschränktes lineares Optimierungsproblem *(BLOP)*, wobei die untere Schranke der Lösung durch den Nullvektor, die obere Schranke durch einen Vektor mit der Dilatation des periodischen Graphen als Komponenten gegeben ist. Die Lösung der Relaxation wird mittels eines *revidierten Simplexverfahrens* berechnet, bei welchem sich in praktischen Versuchen eine recht schnelle Konvergenz beobachten ließ.

Im Falle einer nicht-ganzzahligen Lösung der linearen Relaxation erhält man andererseits eine untere Schranke zur Abschätzung der Distanz von Terminalen auch durch Abrunden der Komponenten des minimalen Lösungsvektors. Der hierbei jedoch entstehende, maximale Fehler der linearen Relaxation des Integer Programms ist bezüglich jeder Vektorkomponente echt kleiner als Eins. Entsprechend der Kostenfunktion muß also der maximale Gesamtfehler echt kleiner als $\#E$ sein, das heißt er ist durch die Anzahl der Kanten des statischen Graphen beschränkt.

Lemma 4.1 beschreibt nun ein erstes, einfaches Charakteristikum eines Spezialfalles, für den das vorliegende *Integer Programm* einfach ist.

Lemma 4.1
Existiert zu jeder Dimension $i \in \{1,\ldots,r\}$ höchstens eine Kante $e_i \in E$ mit $\tau(e_i)_i \in \{-1,+1\}$ und gilt für alle übrigen Kanten e_j die Bedingung $\tau(e_j)_i = 0$, so ist die Matrix A des obigen *Integer Programms* vollständig unimodular.

Beweis: Allgemein bekannt ist, daß die Inzidenzmatrix I_G jedes gerichteten Graphen $G = (V,E)$ vollständig unimodular ist [50]. Man zeigt dies leicht durch Induktion über die Anzahl der Spalten einer beliebigen quadratischen Submatrix. Wir betrachten deshalb nur den Fall quadratischer Submatrizen mit Zeilen aus der unteren Matrix T der Matrix A.

Sei U eine beliebige $(n \times n)$-Submatrix von A mit $k \leq n$ Zeilen aus der Matrix T der Transitionsabbildung τ. Induktion über k:

Sei $k = 1$. Es enthalte die i.-te Zeile von U lediglich Einträge aus T. Nach Voraussetzung handelt es sich hierbei um höchstens ein Nichtnullelement $u_{ij} \in \{-1, +1\}$ mit $j \in \{1, \ldots, n\}$. Nach dem *Laplace'schen Entwicklungssatz* gilt dann aber für die Determinante von U:

$$\begin{aligned} \mathit{det}(U) &= \sum_{s=1}^{n} u_{is} \cdot (-1)^{i+s} \cdot \mathit{det}(U_{is}) \\ &= u_{ij} \cdot (-1)^{i+j} \cdot \mathit{det}(U_{ij}) \end{aligned}$$

wobei U_{ij} die um die i-te Zeile und die j-te Spalte reduzierte $(n-1) \times (n-1)$-Submatrix von U darstellt. Da U_{ij} aber Submatrix einer Inzidenzmatrix ist, muß $\mathit{det}(U_{ij}) \in \{0, -1, +1\}$ sein.

Im Induktionsschritt folgt die Beweisführung demselben Prinzip: auch bei Hinzunahme weiterer Zeilen aus T, entwickelt sich die Determinante zu Werten aus $\{0, -1, +1\}$, da die hinzugenommenen Zeilen jeweils höchstens ein Nichtnullelement besitzen und die um eine Zeile und Spalte verringerte Submatrix nach Induktionsvoraussetzung stets eine Determinante aus $\{0, -1, +1\}$ besitzt. □

Einige Kriterien an den statischen Graphen einer Architektur, welche notwendige Voraussetzungen für eine Unimodularität der dem *Integer Programm* zugrundeliegenden Matrix A darstellen, gibt auch das nachfolgende Lemma an:

Lemma 4.2
Sind alle Komponenten der Bilder der Transitionsabbildung τ aus der Menge $\{0, -1, +1\}$, so darf zur Gewährung der Unimodularität der Matrix A keines der nachfolgenden Kriterien verletzt sein:

1. Zwei aus demselben Knoten auslaufende Kanten bzw. zwei in denselben Knoten einlaufende Kanten des statischen Graphen dürfen sich nicht innerhalb einer Dimension entgegengesetzt fortbewegen.
2. Eine in einen Knoten einlaufende und eine aus demselben Knoten auslaufende Kante dürfen sich nicht bezüglich einer Dimension in gleicher Richtung fortbewegen.
3. Es dürfen keine Kanten existieren, die sich in einer Dimension gleichgesetzt und in einer anderen Dimension entgegengesetzt fortbewegen.

Beweis: Wir betrachten die Determinanten der sich in jedem der Fälle ergebenden 2×2-Submatrizen. Sei dazu $u_1 \in \{-1, +1\}$ und $u_2 \in \{0, -1, +1\}$.

1. Interpretation der ersten Zeile aus I_G, der zweiten Zeile aus T:

$$\mathit{det}\begin{pmatrix} u_1 & u_1 \\ u_2 & -u_2 \end{pmatrix} = -u_1u_2 - u_1u_2 = -2u_1u_2 \in \{0, -1, +1\} \iff u_2 = 0$$

2. Interpretation der ersten Zeile aus I_G, der zweiten Zeile aus T:

$$\mathit{det}\begin{pmatrix} u_1 & -u_1 \\ u_2 & u_2 \end{pmatrix} = u_1u_2 + u_1u_2 = 2u_1u_2 \in \{0, -1, +1\} \iff u_2 = 0$$

3. Hier sind zwei Fälle zu unterscheiden, bei denen sich jeweils die Submatrizen aus den Beweisen zu 1. und 2 ergeben, jedoch mit der Interpretation beider Matrixzeilen aus T.

□

Insofern die definitionsgemäße Prüfung einer Matrix auf vollständige Unimodularität, selbst in verallgemeinerter Charakterisierung [42], aufgrund der Betrachtung aller quadratischen Submatrizen als intraktabel gelten muß, scheint auch im vorliegenden Fall ein effizient nachprüfbares hinreichendes Kriterium für statische Graphen und ihre Transitionsvektoren nicht ermittelbar. Insbesondere führten Versuche konstruktiver Beweise mittels *Seymour's* Zerlegungstheorem [69] auf der Bipartition der Zeilen der Matrix A ins Leere, da Inzidenz- und Transitionsmatrix offenbar zu stark hinsichtlich der Unimodularitätseigenschaft zusammenhängen.

Interessanterweise liefert die bereits in Abbildung 2.15 des Abschnitts 2.4.1 dargestellte Maschenarchitektur ein Beispiel, unter welchem die vollständige Unimodularität der Matrix A gegeben ist.

4.2.3 Heuristik zur Bestimmung einer Implementierungsabfolge

Einer Heuristik zur Berechnung einer möglichst „günstigen" Implementierungsreihenfolge der gegebenen Schaltkreismakros legen wir die folgenden Kriterien zugrunde:

1. Um eine Distanzenbewertung, wie sie im vorigen Unterabschnitt vorgestellt wurde, zwischen den in einer Reihenfolge I bereits implementierten Makros und einem zu implementierenden Makro m überhaupt zu ermöglichen, muß mindestens ein Makro in I existieren, das zu m adjazent ist. Wir bezeichnen nur dann die Implementierungsfolge $I' = (I, m)$ als *legal*.

2. Makros von hoher Dilatation sollten möglichst früh implementiert werden, so daß die durch sie zwangsläufig entstehende Fragmentierung des Architekturraumes noch durch kleinere, später implementierte Makros ausgenutzt werden kann.

3. Zur Minimierung der durchschnittlichen Anzahl während der sukzessiven Implementierung vorhandener temporärer Routen, sollten Makros mit starker Vernetzung unmittelbar nacheinander plaziert werden.

Sei $G = (M, N)$ ein ungerichteter Graph, wobei die Knotenmenge M die Menge der Schaltkreismakros repräsentiert und die Kantenmenge N definiert ist durch:

$$e = \{m_1, m_2\} \in N \quad \Longleftrightarrow \quad \text{Makro } m_1 \text{ und } m_2 \text{ haben gemeinsame Netze}$$

Offensichtlich bildet dann jede Traverse durch G eine legale Implementierungsfolge, womit dem ersten der obigen Kriterien genügt ist.

Sei ferner $\nu(e)$ für ein $e = \{m_1, m_2\} \in N$ die Anzahl der gemeinsamen Netze der Makros m_1 und m_2 und $\sigma(m)$ das Volumen der *bounding box* eines Makros m. Ist I eine Folge

bereits implementierter Makros, so definieren wir für ein Makro $m \in M \setminus I$ und für ein $\alpha \in [0,1]$ die Kostenfunktion

$$c_I(m) = \frac{1-\alpha}{\nu_{\max}} \cdot \left(\sum_{\substack{e=\{m,m'\}\in N \\ m'\in I}} \nu(e) - \sum_{\substack{e=\{m,m'\}\in N \\ m'\notin I}} \nu(e) \right) + \frac{\alpha \cdot \sigma(m)}{\sigma_{\max}}$$

Diese Kostenfunktion berücksichtigt also die Optimierungskriterien 2 und 3, wobei mittels des Faktors α zwischen den Kriterien skaliert werden kann. Die Werte σ und ν gehen dabei normiert auf ihr jeweiliges Maximum ein, damit sie sich hinsichtlich ihrer jeweiligen Größenordnung nicht beeinflussen.

Die Heuristik zur Bestimmung einer Implementierungsfolge beginnt nun mit einem Makro m_1, für welches die Initialkosten

$$c_{\text{init}}(m_1) = \frac{1-\alpha}{\nu_{\max}} \cdot \sum_{\substack{e\in N \\ m_1\in\rho(e)}} \nu(e) - \frac{\alpha \cdot \sigma(m_1)}{\sigma_{\max}}$$

minimal sind und wir setzen: $I_1 = (m_1)$. Die Initialimplementierung enthält somit ein „möglichst" großes Makro m_1 mit „möglichst" wenigen externen Netzen, also möglichst wenigen temporären Routen.

Im weiteren Verlauf des Verfahrens werden zu einer gegebenen Implementierungsfolge I_j für $j = 2,3,\ldots,\#M$ für alle, zu Makros aus I_j adjazenten Makros m die Kosten $c_{I_j}(m)$ berechnet. Ist m' ein Makro *maximaler* Kosten, so setze $I_{j+1} = (I_j, m')$. Das Anordnungsverfahren terminiert also, sobald G vollständig traversiert ist.

In jedem Schritt werden nur Makros betrachtet, die adjazent zur aktuellen Implementierungsfolge sind. Zur Berechnung der Kostenfunktion werden jedoch alle vom entsprechenden Knoten ausgehenden Kanten des Graphen betrachtet. Das Verfahren benötigt exakt $\#M$ Schritte, somit ist die Laufzeit der Heuristik von der Größenordnung $\mathcal{O}(\#M^2 \cdot \#N)$.

4.3 Ein Floorplanning-Konzept für Makros

Das im Rahmen der vorliegenden Arbeit entwickelte Floorplanning-Verfahren basiert auf einem Algorithmus zur Erkennung von Mustern in einem mehrdimensionalen Zeichenfeld. Um die Adaption dieses Algorithmus für den Einsatz beim Floorplanning konfigurierbarer Architekturen später besser erläutern zu können, wird im nachfolgenden Unterabschnitt zunächst eine kurze Einführung in das ursprüngliche Verfahren gegeben.

4.3.1 Multidimensionale Mustererkennung

Das Problem mehrdimensionaler Mustererkennung stellt sich formal wie folgt dar:

Definition 4.10 (Multidimensionale Mustererkennung)
Sei Σ eine Zeichenmenge, $d \in \mathbb{N}$, $t \in \mathbb{N}^d$ und $p \in \mathbb{N}^d$ mit $p \in$ *span*(t). Seien $T \in \Sigma^t$ und $P \in \Sigma^p$ zwei d-dimensionale Zeichenfelder. Dann besteht das *Problem der Erkennung des Musters P im Text T* in der Bestimmung aller Vektoren $x \in$ *span*$(t-p)$, so daß gilt:

$$\forall_{q \in span(p)} \quad P_q = T_{x+q}$$

Baker [9, 38] gibt dazu einen rekursiven Algorithmus an, der das Problem über die Dimension hinweg bis zum eindimensionalen Fall zerlegt und dort dann direkt löst. Wir verwenden am Rekursionsende den bekannten Mustererkennungsalgorithmus von *Knuth*, *Morris* und *Pratt* [41], der lineare Zeitkomplexität in der Länge des zu durchsuchenden Zeichenfeldes besitzt (Algorithmus 4.2).

Algorithmus 4.2 (Knuth-Morris-Pratt-Algorithmus)

```
IndexList calc_contpos(P)
{
    IndexList F;
    F_0 = 0;
    k = 0;
    for i = 1, 2, ..., m-1 do
    {
        while k > 0 und P_k ≠ P_i do
            k = F_{k-1};
        if P_k = P_i then
            k = k + 1;
        F_i = k;
    }
    return F;
}
```

```
IndexList KMP(T,P)
{
    IndexList L;
    IndexList F = calc_contpos(P);
    k = 0;
    for i = 0, 1, ..., n-1 do
    {
        while k > 0 und P_k ≠ T_i do
            k = F_{k-1};
        if P_k = T_i then
            k = k + 1;
        if k = m then
        {
            L.append(i - m + 1);
            k = F_{k-1};
        }
    }
}
```

Algorithmus 4.3 zeigt nun die Zerlegung des Gesamtproblems entlang seiner Dimension auf.

Algorithmus 4.3 (Mehrdimensionale Mustererkennung)

```
CharFunc calc_charfunc(P,d)
{
    M = {P_r ∈ Σ^{p_{d-1}} | r ∈ span(p_0, ..., p_{d-2})};      // Betrachte 1D-Zeichenketten
    Sortiere Menge M: M' = (P_{π(0)}, ..., P_{π(n-1)}) mit n = #M;

    CharFunc χ = (0)^{(p_0,...,p_{d-2})};
    k = 1;
    χ_{π(0)} = k;
    for i = 1, ..., n-1 do
    {
```

```
        if P_π(i) ≠ P_π(i−1) then          // Folgende Zeichenkette unterschiedlich?
            k = k + 1;                                          // Neue Signatur
        χ_π(i) = k;                                         // Signatur zuweisen
    }

    return χ;
}


L_d BAKER(T,P,d)
{
    if d = 1 then                                  // 1D-Fall: Löse direkt mittels KMP
        return KMP(T,P);

    χ = calc_charfunc(P,d);                  // Berechne charakteristische Funktion

    L_d = ∅;
    for j = 0, 1, ..., t_{d−1} − p_{d−1} − 2 do   // Betrachte 1D-Subkette ab Pos. j in T
    {
        Γ = (0)^{(t_0,...,t_{d−2})};

        forall r ∈ span(t_0, ..., t_{d−2}) do                    // Alle 1D-Ketten in T
            forall s ∈ span(p_0, ..., p_{d−2}) do                // Alle 1D-Ketten in P
                if (T_(r,j), ..., T_(r,j+p_{d−1}+1)) = P_s then
                {                                          // Subkette in Muster enthalten
                    Γ_r = χ_s;                                      // Signaturzuweisung
                    break;
                }

        L_{d−1} = BAKER(Γ,χ,d − 1);                                    // Rekursion

        while L_{d−1} ≠ ∅ do          // Liste der Matchpositionen zusammensetzen
        {
            q = L_{d−1}.pop( );
            L_d.append( (q, j) );
        }

    }

    return L_d;

}
```

In der Methode *calc_charfunc* werden alle, bezüglich der Dimension d eindimensionalen Subketten des Musters durchnumeriert, wobei die ermittelte *charakteristische Funktion* $\chi : \Sigma^{p_{d-1}} \to \mathbb{N}_0$ gleiche Subketten auf den gleichen Wert abbildet. Die Bildung von χ erfolgt hier durch lexikographische Sortierung aller Subketten und anschließende Durchmusterung in der Zeit $\mathcal{O}(n \cdot \log n)$.

Nach der Berechnung von χ werden alle, bezüglich der Dimension d eindimensionalen Subketten der Länge $p_{d-1} - 1$ des Feldes T durchmustert. Falls die Subkette an Position $r \in \mathbb{N}^{d-1}$ mit einer bezüglich der Dimension d eindimensionalen Subkette des Musters P an Position $s \in \mathbb{N}^{d-1}$ übereinstimmt, so wird in einem $d-1$-dimensionalen Feld Γ an Position r der Wert von χ_s geschrieben. Die Rekursion erfolgt mit Γ als Textfeld und χ als Musterfeld unter der Dimension $d-1$.

Nach Rückkehr aus der Rekursion kann der d-dimensionale Lösungsindex wie im Algorithmus 4.3 dargestellt, zusammengesetzt werden. Die Laufzeit des Verfahrens bestimmt sich in jeder der insgesamt d Rekursionsstufen im wesentlichen aus der Durchmusterung des Zeichenfeldes T nach in P enthaltenen 1D-Subketten. Besitzt jede 1D-Zeichenkette in T die Länge n und jede 1D-Zeichenkette in P die Länge m, so ergibt sich die Laufzeit des Verfahrens zu

$$\begin{aligned} & d \cdot n \cdot n^{d-1} \cdot m^{d-1} \cdot m \\ = \; & d \cdot n^d \cdot m^d \\ \leq \; & d \cdot n^d \cdot n^d \\ = \; & d \cdot n^{2d} \end{aligned}$$

und damit von der Größenordnung $\mathcal{O}(d \cdot n^{2d})$. Abbildung 4.4 visualisiert das Verfahren nochmals anhand eines einfachen 3D-Beispiels. Dabei wird das 2×2-Muster `P[]` im Textfeld `T[]` nach der Rückkehr aus der zweistufigen Rekursion bei Position $(2, 2, 2)$ gefunden.

4.3.2 Floorplanning durch Mustererkennung

Die Integration eines Floorplanning-Verfahrens für konfigurierbare Architekturen in einem Verfahren zur multidimensionalen Mustererkennung, läßt sich in zwei Schritten charakterisieren [80, 81]. Der Übergang zwischen den beiden, auf den ersten Blick scheinbar unverwandten Problemkreisen, vollzieht sich primär sowohl in Gegenstand als auch Kriterium des ursprünglichen Verfahrens. Zunächst ist von Zeichenketten auf Eigenschaften (*properties*) konfigurierbarer Architekturen und Makro-Implementierungen zu abstrahieren. Zum zweiten stehen beim Floorplanning im Gegensatz zur Mustererkennung keine Muster/Text-Vergleiche im Mittelpunkt, sondern vielmehr Aspekte der Implementierbarkeit, die sich durch Relationen zwischen Eigenschaften formal ausdrücken lassen.

Abstraktion auf Eigenschaften

Der Begriff einer Relation der Implementierbarkeit definiert sich dabei zwischen den Eigenschaften der Ressourcen, also den Logik- und Verdrahtungsmöglichkeiten einer konfigurierbaren Architektur (*resource properties*), und den Eigenschaften der Makro-Implementierungen (*requirement properties*).

Mit der Definition der Modulkompatibilität von Implementierungen (Definition 4.4) haben wir jedoch bereits eine entsprechende Relation eingeführt, wobei die Konsistenz der zu den Implementierungen gehörenden Taskmengen jeder konfigurierbaren Zelle die atomare Bedingung der Relation darstellt. Ferner ist zu überprüfen, daß jede Kante des Architekturgraphen durch maximal ein Netz genutzt wird. Für alle *Intern-* und *Externkanten* des Architekturgraphen wird dies zwar implizit durch die Konsistenz ihrer zugehörigen Routing-Tasks garantiert, einen Ausnahmefall bilden jedoch die Nullkanten. Denn während bei Intern- und Externkanten entsprechend der gegebenen Anordnung der Makros stets die *korrespondierenden Kanten* der Module beim Floorplanning aufeinandertreffen, sind es im Falle von Nullkanten vielmehr die *dualen Kanten* (vgl.

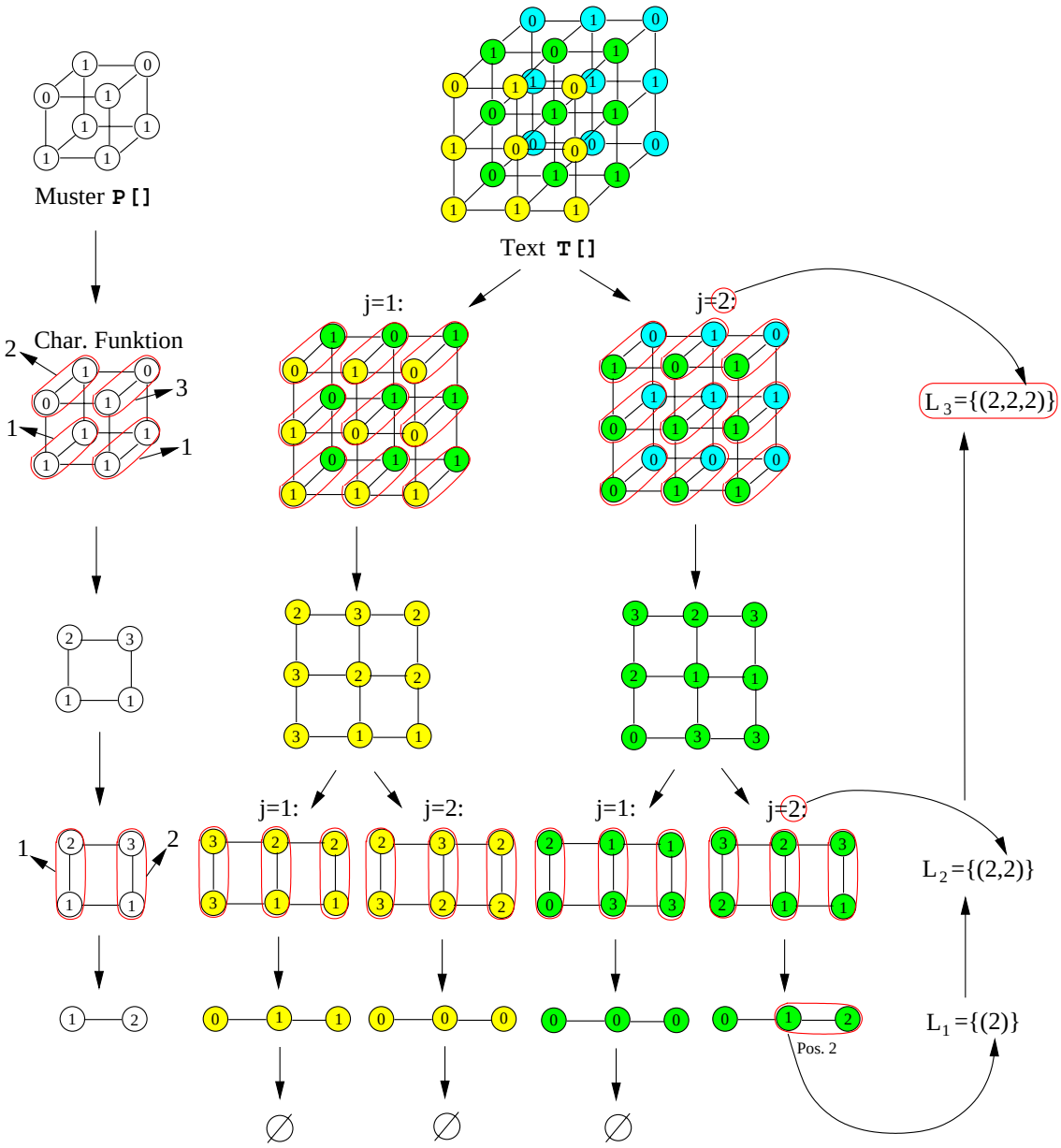


Abbildung 4.4: Beispiel zu Algorithmus 4.3

Definition 1.21), welche auf illegale Belegung zu überprüfen sind. Abbildung 4.5 illustriert beide Fälle in einem eindimensionalen Beispiel.

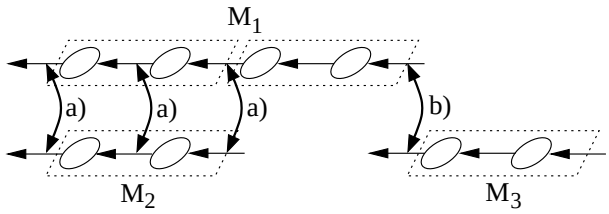


Abbildung 4.5: Beispiel zur Modulkompatibilität

Die drei Makros M_1, M_2 und M_3 seien bezüglich der in der Abbildung dargestellten Anordnung auf Kompatibilität zu überprüfen. Neben den jeweils *korrespondierenden Knoten* der Makros sind zwischen M_1 und M_2 auch drei *korrespondierende Kanten* zu überprüfen (Fall a der Abbildung). Die Makros M_1 und M_3 hingegen überlappen sich lediglich in einer ihrer Nullkanten, welche bezüglich ihrer zugehörigen Module *duale Kanten* zueinander darstellen (Fall b der Abbildung).

Entsprechend des bereits vorgestellen Rahmenalgorithmus 4.1 werden die Schaltkreis-Makros sukzessive auf einem gegebenen Architektursegment angeordnet. Ein einzelnes Makro kann dabei auch als einelementiger Floorplan interpretiert werden; für jedes Makro liegt also bereits eine legale Anordnung vor. Ist nun hingegen ein Floorplan einer Menge modulkompatibler Implementierungen auf einer Architektur gegeben, so ist ein weiteres Makro an jeder Position der Architektur implementierbar, für die dessen Anforderungs-Eigenschaften in den noch verbliebenen Ressourcen-Eigenschaften der Architektur „enthalten“ sind. Wir sprechen deshalb im folgenden nicht mehr von einer *Pattern-Matching-*, sondern vielmehr von einer *Property-Containment*-Suche (*PCS*).

Property Containment Search

Die *Property-Containment*-Suche basiert nun zwar auf dem mit Algorithmus 4.3 vorgestellten rekursiven Verfahren, allerdings braucht die *Containment*-Relation der Ressourcen- und Anforderungs-Eigenschaften nur in der ersten Rekursionsstufe tatsächlich überprüft zu werden. Für die verbleibenden Stufen transformiert sich das Problem auf ein einfaches *Bitstring-Containment*-Problem:

Definition 4.11 (Bitstring Containment)
Sind $S = (s_1, \ldots, s_n)$ und $R = (r_1, \ldots, r_m)$ mit $m \leq n$ zwei Zeichenketten über der Menge $\{0, 1\}$, so heißt R in S *enthalten* (in Zeichen: $R \sqsubseteq S$), genau dann, wenn ein $k \in \{1, \ldots, n-m\}$ existiert, so daß gilt:

$$\forall_{i \in \{1,\ldots,m\}} \quad r_i = 1 \implies s_{k+i} = 1$$

Sei $\mathcal{F} = (\mathcal{M}, \psi, y)$ ein Floorplan auf einer d-dimensionalen Architektur $\mathcal{A}$ und $M \notin \mathcal{M}$ ein $\mathcal{F}$ hinzuzufügendes Makro. Die Adaption des Verfahrens von *Baker* besteht nun in den folgenden vier Punkten:

1. Sei p die Anzahl der bezüglich der d-ten Dimension unterschiedlichen eindimensionalen Ketten von Anforderungs-Eigenschaften des Makros M. Dann ist das Bild der zu berechnenden charakteristischen Funktion χ von M ein Bitstring der Länge p, so daß $\chi(x)$ für ein $x \in \mathbb{Z}^{d-1}$ jenen Bitstring liefert, der an $p-1$ Positionen Null ist, jedoch genau an der i-ten Position Eins ist, wenn an Position x des Makros M die i-te Eigenschaftskette vorliegt. Man beachte, daß p in jedem Falle durch das Volumen der Implementierung von M nach oben beschränkt ist.

2. Das im Zuge der Rekursion quasi die Bedeutung des „Textfeldes“ erhaltende $(d-1)$-dimensionale Feld Γ wird, statt ursprünglich durch Nullen, nun durch Null-Bitstrings der Länge p initialisiert.

3. Anstelle der Identitätsvergleiche der Sub-Zeichenketten $T_{(r,j)}$ und P_s wird eine Prüfung der *Containment*-Relation zwischen Anforderungs- und Ressourceneigenschaften in Rekursionstiefe $q \in \{0, \ldots, d-1\}$ wie folgt durchgeführt:

   - $q = 0$:

     Die Kette P_s von Anforderungs-Eigenschaften ist in der Kette $T_{(r,j)}$ gleicher Länge von Ressourcen-Eigenschaften enthalten genau dann, wenn die Vereinigung der entsprechenden Subimplementierungen des hinzuzufügenden Makros M und des Floorplans $\mathcal{F}$ eine modulkompatible Implementierung liefert.

   - $q > 0$:

     Die Kette P_s von Bitstrings ist in der Kette $T_{(r,j)}$ gleicher Länge von Bitstrings enthalten genau dann, wenn gilt: $P_s \sqsubseteq T_{(r,j)}$.

4. Wurde die *Containment*-Relation positiv getestet, so wird $\Gamma(r)$ nicht das Bild $\chi(s)$ zugewiesen, sondern es wird stattdessen auf den Bitstrings komponentenweise das logische Oder über der booleschen Interpretation der Werte Null und Eins gebildet:

$$\Gamma(r) := \Gamma(r) \ \vee \chi(s)$$

Die Laufzeit der *Property-Containment*-Suche verändert sich im Hinblick auf die ursprüngliche *Pattern-Matching*-Version im wesentlichen nur in der Berechnung der *Containment*- bzw. *Matching*-Relationen. Während beim *Matching*-Verfahren zwei Zeichen in konstanter Zeit miteinander verglichen werden, handelt es sich bei der *Containment*-Bedingung genauer um den Schnitt zweier Konfig-Sets, nämlich der Anforderungs- und der Ressourceneigenschaften. Die Komplexität eines Schnittes von Konfig-Sets wurde andererseits bereits in Abschnitt 2.4.3 betrachtet und hängt nur von der Spezifikation der konfigurierbaren Zellen, also nicht von der Dilatation der Floorplan-Instanz, ab. Deshalb können wir die *Containment*-Relation als ebenfalls in konstanter Zeit berechenbar interpretieren.

Anpassung der Dilatation

Der Plazierungsraum eines Floorplans wird initial durch die Dilatation der Implementierung des ersten angeordneten Makros determiniert. Das sukzessive Hinzufügen von

Makros zum Floorplan erfordert in der Regel jedoch auch eine Erweiterung des Plazierungsraumes, wenn eine vollständig überlappende Implementierung aufgrund der noch zur Verfügung stehenden Ressourcen nicht mehr möglich ist. Andererseits steigt mit zunehmender Dilatation des Plazierungsraumes auch der Berechnungsaufwand des Floorplanning-Verfahrens, weshalb lediglich eine sukzessive Expansion sinnvoll erscheint.

In Rahmenalgorithmus 4.1 wird die Dilatation des Plazierungsraumes eines Floorplans zu zwei Zeitpunkten verändert. Zum einen mittels der Methode *expand_area()* vor dem Starten der *Property-Containment*-Suche jedes hinzuzufügenden Makros, zum anderen mittels der Methode *create_hull()* vor dem Beginn eines zweiten Verdrahtungsversuches des Makros.

Der ersten Expansion liegt die bereits genannte Motivation der Bereitstellung freier Ressourcen zugrunde. Die Erweiterung des aktuellen Plazierungsraumes wird hier durch einen Parameter $\delta \in [0;1]$ bestimmt, welcher ein Maß für die Überlappung gefundener Lösungen der *Property-Containment*-Suche mit dem bisherigen Floorplan darstellt. Abbildung 4.6 skizziert die Expansion des Plazierungsraumes einer Floorplan-Implementierung der Makros $M_{\pi(1)}, \ldots, M_{\pi(i-1)}$.

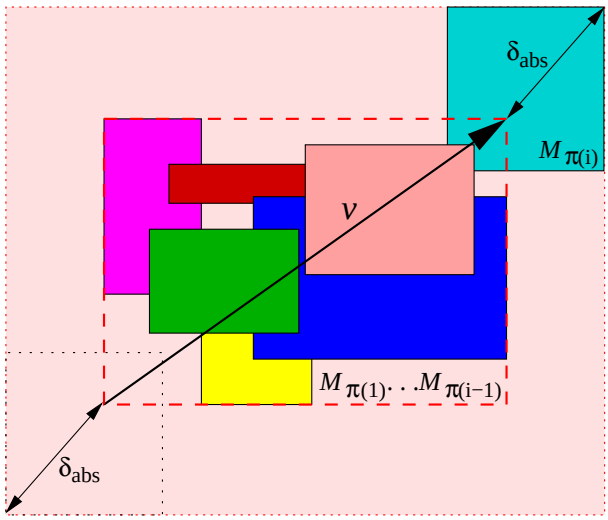


Abbildung 4.6: Expansion der Plazierungsebene

Sei $v \in \mathbb{N}^d$ die Dilatation der *bounding box* des Floorplans. Die Expansion erfolgt nun über alle d Dimensionen der Architektur, jeweils sowohl an der oberen als auch an der unteren Dilatationsgrenze der Floorplan-Implementierung, wobei sich die Vergrößerung δ_{abs} des Plazierungsraumes aus der Skalierung der Dilatation des zu plazierenden Makros $M_{\pi(i)}$ mit dem Faktor δ ergibt: Anschaulich betrachtet, wird eine „Hülle“ freier Ressourcen um den bisherigen Floorplan gelegt. Offensichtlich garantiert eine Skalierung von $\delta = 1$ stets die Existenz einer legalen Plazierung des Makros $M_{\pi(i)}$.

Konnte eine legale Plazierung für ein Makro ermittelt werden, so besteht noch immer das Problem der Nichtverdrahtbarkeit seiner Terminale. In empirischen Versuchen wurde hier bei verschiedenen Architekturen beobachtet, wie insbesondere eine Verdrahtung von der Peripherie zugewandten Terminalen gelegentlich aufgrund dort man-

gelnder Verdrahtungsressourcen scheitert. Eine Expansion der *bounding box* des ermittelten gültigen Floorplans vor einem zweiten Verdrahtungsversuch verfolgt nun die Methode *create_hull()*. Die Expansion wird dimensionsweise, unter der Berücksichtigung bereits vorhandener freier Module jeweils an der oberen sowie an der unteren Dilatationsgrenze des Architekturgraphen durchgeführt, wobei die Schrittweite der Expansion s_d in der Dimension d sich durch die Anzahl u der zuletzt nicht-verdrahtbaren Netze und die Anzahl e_d der Externkanten des Moduls bezüglich Dimension d bestimmt als:

$$s_d = \left\lceil \frac{u}{e_d} \right\rceil$$

Erst nach einem Fehlschlagen des zweiten Verdrahtungsversuches fährt das Floorplanning-Verfahren mit einer anderen Plazierung fort.

4.4 Verdrahtung von Makros

Das zur Verdrahtung von Schaltkreismakros angewandte Verfahren orientiert sich hinsichtlich der Implementierung von Multiterminalnetzen an der bereits in Abschnitt 3.3.3 motivierten Strategie. Demnach werden in Analogie zur Steinerbaum-Heuristik von *Wu, Widmayer* und *Wong* [85] auch hier Multiterminalnetze durch sukzessive Erweiterung einer bisher gefundenen Teillösung auf der Basis eines minimalen Spannbaumes implementiert, doch alternieren im Gegensatz zur *WWW-Heuristik* die Netze bezüglich ihrer schrittweisen Expansion entsprechend der Anordnungsreihenfolge der Schaltkreismakros.

Das sogleich genauer vorgestellte Verdrahtungsverfahren verfolgt ebenfalls das Konzept temporärer Routen. Es löst Verdrahtungskonflikte *(Routing Congestions)* auf durch Re-Iterationen nach einer partiellen Manipulation der Kostenfunktion unter Abwägung einer alternativen Anordnung der Schaltkreismakros.

4.4.1 Struktur des Verdrahters

Die Implementierung eines Multiterminalnetzes stellt einen K-Baum dar, der sich wiederum aus eine Menge von Routen zusammensetzt, welche sukzessive berechnet werden. Die Menge der Terminale eines Netzes kann dabei hinsichtlich eines aktuellen Floorplans partitioniert werden:

> **Definition 4.12 (Implementierte/Nicht-implementierte Terminale)**
> Sei $\mathcal{M}$ die Menge der zu einem gegebenen Schaltkreis $\mathcal{C} = (V, E)$ generierten Makros. Sei $N \in E$ ein Netz und $\mathcal{F} = (\mathcal{M}', \psi, y)$ ein Floorplan mit $\mathcal{M}' \subseteq \mathcal{M}$. Dann bezeichnet $T_{\mathcal{F},N}$ jene Menge von Terminalen aller Makros aus $\mathcal{M}'$, die zu N gehören (Menge der *implementierten Terminale*). Ferner bezeichne $\bar{T}_{\mathcal{F},N}$ die Menge von Terminalen aller Makros aus $\mathcal{M} \setminus \mathcal{M}'$, die zu N gehören (Menge der *nicht-implementierten Terminale*).

In einem gegebenen Floorplan $\mathcal{F}$ ist nun ein Netz N genau dann zu implementieren, wenn $T_{\mathcal{F},N} \neq \emptyset$. Ferner ist auch exakt eine Route zur Peripherie des Floorplans genau dann zu berechnen (*temporäre Route*), wenn $\bar{T}_{\mathcal{F},N} \neq \emptyset$ oder wenn N Netz eines

primären Eingangs oder Ausgangs des Schaltkreises ist. Aus Gründen erhöhter Freiheitsgrade beim Verdrahten sukzessiv dem Floorplan hinzugefügter Makros, werden also auch Netze primärer Eingänge und Ausgänge des Schaltkreises durch temporäre Routen realisiert.

In der Folge aller im aktuellen Floorplan zu berechnenden Routen eines Netzes wird die Ermittlung temporärer Routen als letztes vorgenommen, um insbesondere floorplaninternen Netzen hinsichtlich freier Ressourcen den Vorrang zu geben, da temporäre Routen später ohnehin wieder entfernt werden. Weitere Aspekte für die hier angewandte Methodik der Berechnung von Routen eines Netzes stellen schließlich noch die Fragen dar, ob das Netz bereits eine Teilimplementierung besitzt und ob das Quellterminal des Netzes bereits implementiert ist. Auf den genannten Fallunterscheidungen basiert nun die Grobstruktur des im folgenden vorgestellten Makroverdrahters, welche in Abbildung 4.7 illustriert ist.

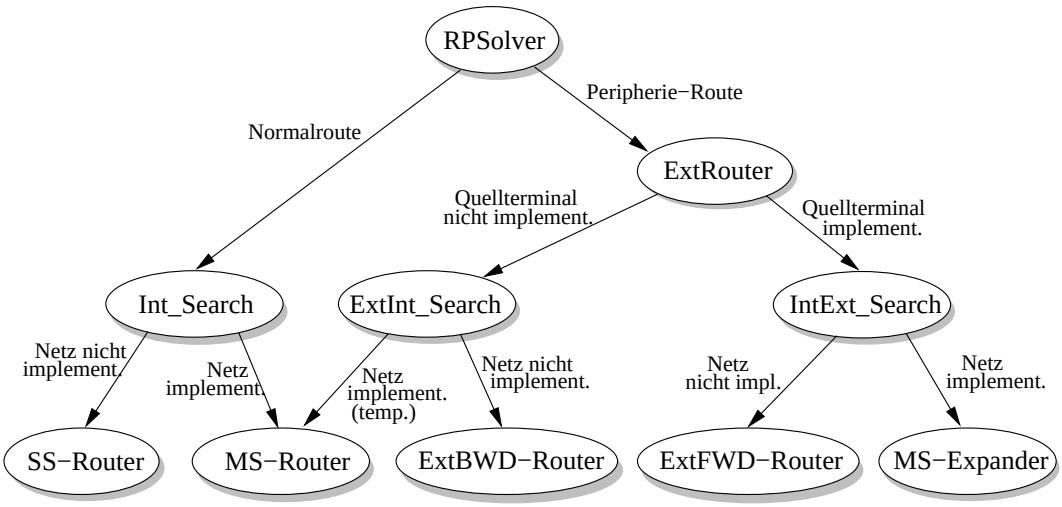


Abbildung 4.7: Struktureller Aufbau des Makroverdrahters

In der untersten Hierarchiestufe der Abbildung sind die eigentlichen, routenberechnenden Verfahrenskomponenten dargestellt. Da es sich hierbei jedoch im wesentlichen um Adaptierungen prioritätsgesteuerter Standardverfahren zur Bestimmung kürzester Wege in gerichteten Graphen unter nicht-negativer Kostenfunktion handelt, soll im folgenden auf deren explizite Ausbreitung verzichtet und stattdessen lediglich auf deren Einordnung in das Gesamtkonzept des Verdrahters eingegangen werden.

Normale Routen

Sei wieder N ein in einem Floorplan $\mathcal{F}$ zu verdrahtendes Netz. Der Löser für Verdrahtungsprobleme *(RPSolver)* hat zunächst zu unterscheiden, ob eine Peripherie-Route zu berechnen ist. Ist dies *nicht* der Fall, handelt es sich bei der zu berechnenden Route um eine „normale“ Route, das heißt sowohl Quell- als auch Zielterminal sind implementiert und eine floorplaninterne Suche wird durchgeführt. Falls N bereits eine Teilimplementierung besitzt, wird diese durch eine Suche ab der Menge ihrer Fanoutpunkte expandiert *(Multiple-Source-Router)*. Falls hingegen noch keine Implementierung von

N existiert, wird die erste Route für N durch eine Kürzeste-Wege-Suche in Vorwärtsrichtung ab dem implementierten Quellterminal bestimmt *(Single-Source-Router)*.

Peripherie-Routen

Bei der Verdrahtung von Peripherie-Routen *(ExtRouter)* ist zunächst zu unterscheiden, ob eine Route von oder zu der Peripherie des Floorplans zu bestimmen ist *(ExtInt-* oder *IntExt-Search)*. Ist das Quellterminal von N implementiert, so wird eine Vorwärtssuche bis zur Peripherie durchgeführt: im Falle einer existierenden Teilimplementierung ab der Menge ihrer Fanoutpunkte *(Multiple-Source-Expander)*, im Falle eines nichtimplementierten Netzes ab dessen implementierten Quellterminal *(ExtFWD-Router)*.

Eine Route von der Peripherie zu einem Zielterminal von N wird durch eine Rückwärtssuche ab dem Terminal berechnet, falls keine Teilimplementierung des Netzes bislang existiert *(ExtBWD-Router)*. Bei einer existierenden Netzimplementierung hingegen, welche im übrigen dann bereits eine periphere Route enthält, braucht nur expandiert zu werden, indem man eine Vorwärtssuche ab ihren Fanoutpunkten bis zu den jeweiligen Zielterminalen durchführt *(Multiple-Source-Router)*.

4.4.2 Verfahren

Zu einem gegebenen Floorplan definieren wir das Verdrahtungsproblem eines Netzes wie folgt:

Definition 4.13 (Verdrahtungsproblem eines Netzes)
Sei $\mathcal{F}$ ein Floorplan und N ein Netz. Dann heißt $P_{\mathcal{F},N} = (q, Z, \xi)$ das *Verdrahtungsproblem* von N auf $\mathcal{F}$, wenn gilt: q ist das Quellterminal von N, falls $q \in T_{\mathcal{F},N}$ und undefiniert, sonst. $Z = T_{\mathcal{F},N} \setminus \{q\}$ ist die Menge der Zielterminale von N. Ferner ist $\xi \in \mathbb{B}$ mit $\xi = 1$ genau dann, wenn q undefiniert oder $Z = \emptyset$ oder N Netz eines primären Eingangs oder Ausganges des zugrundeliegenden Schaltkreises ist.

Falls $\#Z > 1$ bezeichnen wir $P_{\mathcal{F},N}$ als *Multiterminalproblem*. Die Lösung eines Verdrahtungsproblems stellt einen K-Baum auf dem Floorplan dar, welcher genau dann eine temporäre Route enthält, wenn $\xi = 1$. Algorithmus 4.4 zeigt die eingebettete Hauptschleife des Verdrahtungsverfahrens. Die Verdrahtungprobleme werden hier Netz für Netz behandelt, wobei jedes Multiterminalproblem implizit in Zweiterminalprobleme zerlegt wird, welche sukzessive gelöst werden.

Algorithmus 4.4 (Makroverdrahter)

```
bool MacroRouter(Floorplan F, Menge P_F von Verdrahtungsproblemen)
{
    routed = false;
    while not routed do
    {
```

```
    R = ∅;                                   // Menge der berechneten Routen
    P'_F = ∅;                  // Menge der vorgemerkten Verdrahtungsprobleme

    forall P ∈ P_F do                        // Betrachte alle Verdrahtungsprobleme
    {
        if q_P definiert und Z_P ≠ ∅ then                          // Normalroute
            R = Int_Search(F, q_P, Z_P);
        else                                                   // Peripherie-Route
            if q_P definiert then
                R = IntExt_Search(F, q_P);
            else
                R = ExtInt_Search(F, Z_P);

        if R = ∅ then                                        // Keine Route gefunden
            return false;

        R.insert(R);

        if Z_P ≠ ∅ then
        {
            P' = (q_P, Z_P \ {Z(R)}, ξ_P);                       // Terminal entfernen

            if #Z_P' ≥ 1 oder ξ_P then
                P'_F = P'_F ∪ {P'};                  // Rest-Problem P' vormerken
        }

    }
    if alle Routen R ∈ R disjunkt then
    {
        forall R ∈ R do
            Konfiguriere R auf F;

        if P'_F = ∅ then
            routed = true;                    // Alle Verdrahtungsprobleme gelöst
        else
            P_F = P'_F;
    }
    else
        if iterations < max_iterations then
        {
            iterations = iterations + 1;
            manage_conflicts(F,R);
        }
        else                                  // Max. Anzahl Re-Iterationen erreicht
            return false;

}

return true;
}
```

Das Verfahren ist in eine *while*-Schleife eingebettet, welche zwei Funktionen erfüllt: zum einen die Iteration über im Falle von Multiterminalnetzen entstehende Subprobleme in der Verdrahtung, zum anderen eine Re-Iteration bei konfliktierenden Routen.

Die Aufteilung eines Multiterminalproblems im Sinne der Festlegung einer Reihenfolge der Lösung seiner Subprobleme erfolgt hierbei *dynamisch*, indem die aufgerufenen Algorithmen zur Verdrahtung von Zweiterminalnetzen (*Int_Search*, *IntExt_Search* und *ExtInt_Search*) zwar stets mit einer *Menge* von Zielterminalen des Multiterminalnetzes arbeiten, allerdings lediglich für höchstens ein Zielterminal eine Route R berechnen, wobei dieses Zielterminal $Z(R)$ jeweils aus der Menge entfernt wird. Durch diese Freiheitsgrade im Zuge der Expansion einer Netzimplementierung gewinnt das Gesamtverfahren an Flexibilität. Trotz der Berechnung lediglich einer Route, terminiert eine Pfadsuche jedoch erst, wenn alle Terminale in die Suchfront aufgenommen, das heißt „gesehen“ wurden. Damit wird die Nicht-Erreichbarkeit von Terminalen zu frühestmöglichem Zeitpunkt festgestellt. Terminiert nun die Pfadsuche eines Multiterminalnetzes, so wird aus der Menge der Terminale jenes zur Konstruktion einer Route ausgewählt, für welches sich der *längste* kürzeste Pfad ergibt. Anschaulich betrachtet, wird das „am aufwendigsten“ zu erreichende Terminal somit stets zuerst verdrahtet.

Entsprechend dieser Vorgehensweise wird zu jedem Netz eine Route berechnet, welche anschließend auf Disjunktheit (im Sinne einer Kompatibilität ihrer berechneten Implementierungen auf dem gegebenen Floorplan) überprüft werden. Sind die Routen disjunkt, wird mit dem Lösen der Subprobleme fortgefahren, ansonsten werden die Verdrahtungskonflikte untersucht und gegebenenfalls in einer Re-Iteration derselben Problemmenge, jedoch unter manipulierter Kostenfunktion, erneut behandelt. Im folgenden wird auf diesen Aspekt noch genauer eingegangen.

Kostenfunktion

Wie bei der Generierung von Makros stehen als Zielsetzungen auch bei deren Verdrahtung die Implementierbarkeit und die Minimierung des resultierenden Verdrahtungsdelays im Zentrum der Betrachtung. Zur Vermeidung lokaler Minima unter einer festen Reihenfolge in der Implementierung der Netze, sowie aufgrund des Berechnungsaufwandes wurde auch hier, wie in Abschnitt 3.3.3, ein Ansatz gewählt, der eine quasiparallele Berechnung von Netzimplementierungen durch in Re-Iterationen um Ressourcen konkurrierende Suchen verfolgt.

In die Kostenfunktion der Pfadsuchen geht deshalb neben dem Verdrahtungsdelay der Routing-Tasks auch ein skalierter Straffaktor ein, welcher im Konfliktfall angehoben wird:

$$d_v = d_u + (c_u + \delta_t) \cdot \mu$$

Seien u und v zwei unmittelbar aufeinanderfolgende Kanten in einem Pfad, wobei in der zwischen u und v liegenden konfigurierbaren Zelle der Routing-Task t benutzt werde. Dann ergeben sich die Kosten d_v der bislang ermittelten Route vom Startterminal bis zur Kante v durch Akkumulation der Kosten bis zur Kante u, den skalierten Kosten zur Verdrahtung über die Kante u und den Kosten des Routing-Tasks t. Die Kantenkosten c_u definieren sich genauer durch:

$$c_u = p_u \cdot n_i + k_u$$

Hierbei stellt p_u einen nullinitialisierten Straffaktor der Kante u dar, n_i entspricht der Anzahl der bisherigen Re-Iterationen aufgrund von Verdrahtungskonflikten und k_u bezeichnen wir als die *Congestion*-Kosten der Kante u, welche sich bestimmen durch das Verhältnis der im Quell- und Zielknoten von u *bezüglich* u konfigurierbaren Routing-Tasks zu den dort insgesamt existierenden Routing-Tasks bezüglich u. Die *Congestion*-Kosten drücken somit etwa den Grad der Nutzbarkeit einer Kante aus und erhöhen entsprechend bereits zur ersten Verdrahtungsiteration die Kosten der Kante mit dem Ziel, daß die Kante möglichst nur von Netzrouten benutzt wird, für die sie essentiell ist. Schließlich führen wir wieder einen Skalierungsfaktor μ für temporäre Routen ein.

Konfliktbewältigung

Wurden im Rahmen der Pfadsuchen der einzelnen Netze nicht-disjunkte Routen berechnet, so werden die entsprechenden Kanten der Routen, im folgenden wieder als *Konfliktkanten* bezeichnet, vor der Durchführung von Re-Iterationen bestraft (Methode *manage_conflicts()* in Algorithmus 4.4), um hierbei das Ausweichen der Suchen auf alternative Routen zu begünstigen. Eine alternative Route kann im besten Falle dieselben Kosten besitzen, wie die ursprünglich gefundene Route, jedoch kann auch die Nutzung einer teureren Variante erforderlich werden.

Genauer handelt es sich bei Konfliktkanten nun um alle Kanten, deren R-Tasks im gemeinsamen Quellknoten unter Zugrundelegung der bereits konfigurierten Taskmenge nicht kompatibel sind oder, als Spezialfall hiervon, von Routen unterschiedlicher Netze genutzt werden. Unter Motivation des bereits in Abschnitt 3.3.3 Gesagten, definieren wir den Straffaktor p_u einer Konfliktkante u wieder über die Anzahl der u gemeinsam nutzenden Netze, welche mit der Anzahl n_i der Re-Iterationen multipliziert wird.

Im Hinblick auf diese doch relativ „unscharfe“ Kostenmanipulation im Konfliktfall, stellt sich auch die Frage, ob die Höhe einer tatsächlich notwendigen Bestrafung von Konfliktkanten nicht exakter feststellbar ist, so daß einerseits weniger Re-Iterationen nötig und andererseits nicht überteuerte Routen genommen werden. Eine solche Abschätzung gelingt jedoch durchaus auf der Basis der im Zuge der Pfadsuche ohnehin bereits ermittelten Daten. Abbildung 4.8 zeigt die Skizze eines entsprechenden Szenarios.

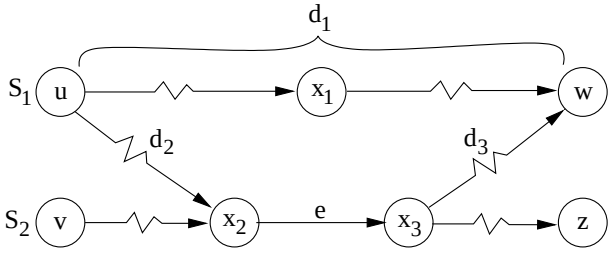


Abbildung 4.8: Strategie der alternativen Pfade

Zu verdrahten seien die beiden Signale S_1 und S_2, jeweils von Ausgängen der konfigurierbaren Zellen u bzw. v zu Eingängen der Zellen w bzw. z. Wird für beide Signale eine Route über die Zelle x_2 ermittelt, so ergibt sich e als Konfliktkante. Zur Auflösung des

Konfliktes müßte e so bestraft werden, daß Signal S_1 über x_1 verdrahtet wird, daß also gilt:

$$\begin{aligned} & d_1 < d_e + d_2 + d_3 \\ \Longleftrightarrow \quad & d_e > d_1 - d_2 - d_3 \end{aligned}$$

Während die Kosten d_2 und d_3 anhand der für S_1 ermittelten Route bekannt sind, ist dies bei den Kosten d_1 im allgemeinen nicht der Fall. Eine Route für S_1 über den Knoten x_1 würde aber eine alternative Route darstellen, deren Nutzung den Verdrahtungskonflikt direkt auflöste. Man erhält die Kosten einer solchen alternativen Route jedoch leicht, indem man das Verfahren zur Pfadsuche so erweitert, daß neben der üblichen Speicherung der Kosten eines günstigsten auch die Kosten eines nächstteuren Alternativ-Pfades gehalten werden. Sofern nun im Rahmen der Suche einmal ein Alternativ-Pfad – im Beispiel gerade der Pfad über x_1 – gefunden wurde, wird die Kostenfunktion der Konfliktkante e entsprechend obiger Ungleichung so angehoben, daß die Route für S_1 nicht mehr über x_2 verläuft. Die Route für S_2 muß jedoch nach wie vor über den, jetzt allerdings teureren Weg, über x_2 laufen, wenn dies die einzige Möglichkeit, wie beispielsweise in der Abbildung skizziert, darstellt. Doch selbst die Existenz von Alternativen für Routen von Signal S_2 bleibt in diesem Falle unkritisch, da S_2 nur Alternativ-Routen nehmen wird, deren Gesamtkosten unterhalb von d_1 bleiben, das heißt das durch S_1 ohnehin bestimmte resultierende Delay kann sich nicht verschlechtern.

Im Zuge der Re-Iterationen können auch Zyklen in der Berechnung auftreten. Illustrativ betrachtet, „springen“ dann Routen von Netzen zwischen mehreren jeweils konfliktierenden Varianten hin und her. Im Verdrahtungsverfahren aus Abschnitt 3.3.3 wird versucht, solche Zyklen durch einen netzspezifischen, randomisierten Anteil der Kostenfunktion zu vermeiden. Eine andere Möglichkeit stellt das Aufbrechen von Berechnungszyklen durch konkretes Begünstigen einer Route für ein Verdrahtungsproblem dar. Im vorliegenden Falle wurde dies realisiert durch eine Reduktion der Bestrafung der Konfliktkante, allerdings nur für jenes Netz, zu welchem zuletzt die *längste* Routenimplementierung berechnet wurde. Diese Strategie konnte für den zuvor genannten Fall aus Abschnitt 3.3.3 hingegen nicht angewandt werden, da dort die Plazierung eines Terminals erst durch Verdrahtung ermittelt wird und eine eventuell günstigere Routen ermöglichende, alternative Plazierung durch eine Begünstigung der längsten Netzrealisierung verhindert werden kann.

Da das Verdrahten in der Praxis einen nicht unwesentlichen Zeitaufwand in Anspruch nimmt, wurden auch die Kostenabschätzungen der durch den Floorplanner gelieferten Makroanordnungen einbezogen. Demnach wird die maximale Anzahl von Re-Iterationen dynamisch dem Kostenprofil der vom Floorplanner berechneten Lösungen angepaßt. Mit anderen Worten, wird der Verdrahtungsvorgang vorzeitig abgebrochen und stattdessen die nächste Makroanordnung betrachtet, wenn deren geschätzte Kosten nicht „übermäßig“ über jener der aktuell betrachteten Lösung liegen. Genauer hat sich in empirischen Versuchen ein Kriterium als vorteilhaft erwiesen, das bei mehr als drei Re-Iterationen und mehr als der Hälfte konfliktierender Netzrouten abbricht, wenn die Bewertung der nächsten Makroanordnung nicht über 50% schlechter liegt als die der aktuell betrachteten Anordnung.

Aufgrund der Re-Iterationen durch Verdrahtungskonflikte, welche von den unterschiedlichsten Faktoren beeinflußt werden, ist eine Laufzeit des Makroverdrahters im allge-

meinen nicht konkret abschätzbar. Für den Fall nicht aufretender Konflikte lassen sich hingegen Aussagen treffen. Die Gesamtzahl während eines Aufrufs von Algorithmus 4.4 auftretender Verdrahtungsprobleme entspricht hierbei der Anzahl t der Terminale des zu verdrahtenden Makros zuzüglich höchstens einer temporären Route pro Terminalnetz, deren Anzahl wiederum durch t nach oben beschränkt ist. Jedes dieser Verdrahtungsprobleme wird genau einmal gelöst durch ein Kürzeste-Wege-Verfahren auf einem gerichteten Graphen mit nichtnegativen Kantenkosten und Prioritätswarteschlange. Die Prüfung der berechneten Routen auf Disjunktheit erfolgt linear in deren Länge, wird also durch die Suchverfahren subsummiert.

Damit schließen wir die Vorstellung des im Rahmen der vorliegenden Arbeit entwickelten generischen Layoutsystems für feldprogrammierbare Architekturen ab und wenden uns im noch verbleibenden, letzten Kapitel der Untersuchung von Architekturen zu.

Kapitel 5

Bewertung

In diesem Kapitel werden Beispiele verschiedener feldprogrammierbarer Architekturen vorgestellt und mit den in Kapitel 2 eingeführten Bewertungsmaßen untersucht. Insbesondere werden unter Zuhilfenahme des entwickelten generischen Layoutsystems Implementierungen verschiedener Schaltkreise auf den Architekturen berechnet, um eine Einschätzung ihres Verhaltens hinsichtlich Kosten- und Delay zu erhalten.

In Abschnitt 5.1 werden zunächst die betrachteten Architekturen beschrieben, während Abschnitt 5.2 auf die durchgeführten Experimente und ihre jeweiligen Resultate eingeht, wobei eine Charakterisierung der Eigenschaften der Architekturen versucht wird. Der letzte Abschnitt 5.3 faßt die Ergebnisse der vorliegenden Arbeit zusammen und verweist zugleich auf Ansatzpunkte für weitergehende Untersuchungen, welche im Rahmen dieser Arbeit zwar nicht mehr betrachtet werden konnten, jedoch mit Hilfe des vorgestellten Bewertungskonzeptes, sowie des nun zur Verfügung stehenden universellen Layout-Werkzeuges weiterverfolgt werden können.

5.1 Architekturen

Als Gegenstand der Untersuchungen der vorliegenden Arbeit wurden drei beispielhafte Architekturtypen ausgewählt, welche jeweils wiederum in drei unterschiedlichen Ausprägungen betrachtet wurden. Im Zentrum des Interesses stand dabei zum einen die genauere Betrachtung der in einer früheren Arbeit [79] des Verfassers entwickelten *Maschenarchitekturen*, welche dort als Grundstruktur eines minimalistischen Architekturkonzeptes entwickelt wurden. Zum anderen wurden Architekturen des im kommerziellen Bereich häufig auftretenden *Inseltypus* definiert, um sie den Maschenarchitekturen gegenüber zu stellen. In beiden Typklassen wurden ferner auch dreidimensionale Architekturvarianten betrachtet.

5.1.1 Maschenarchitekturen

SM2V001S

Abbildung 5.1 zeigt den statischen Graphen der Variante 1 der zweidimensionalen *einfachen Maschenarchitektur (single mesh)*, wie sie bereits in Abschnitt 2.4.1 als Beispiel

eingeführt wurde. Die Struktur der vier konfigurierbaren Zellen ist nochmals rechts daneben in Abbildung 5.2 dargestellt.

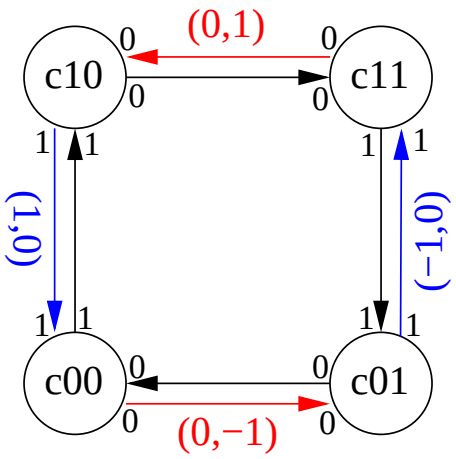


Abbildung 5.1:
Architektur *SM2V001S*

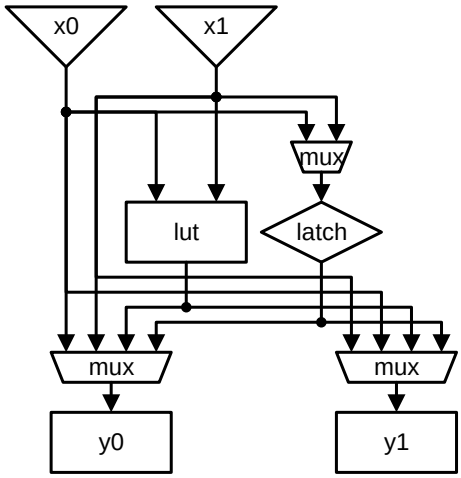


Abbildung 5.2:Zelle *CSR0202V001*

SM2V002S

Mit der Variante 2 der einfachen Maschenarchitektur (Abbildung 5.3) wurde versucht, die Ausnutzung der einzelnen Zellen bei der Implementierung von Schaltkreisen durch Hinzufügen weiterer Verdrahtungsressourcen in Form von Rückkopplungsleitungen zu verbessern. Das Ausgangssignal der Look-Up-Table und des Latches kann hier über die Pins x_0 und y_0 bzw. x_1 und y_1 in die Zelle zurück geleitet werden (siehe Abbildung 5.4).

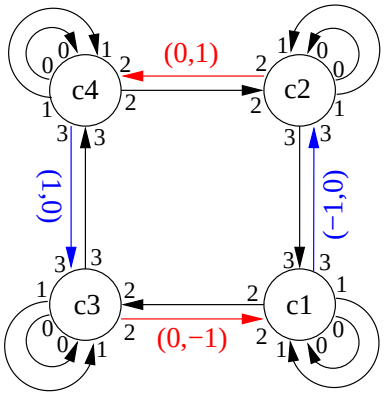


Abbildung 5.3:
Architektur *SM2V002S*

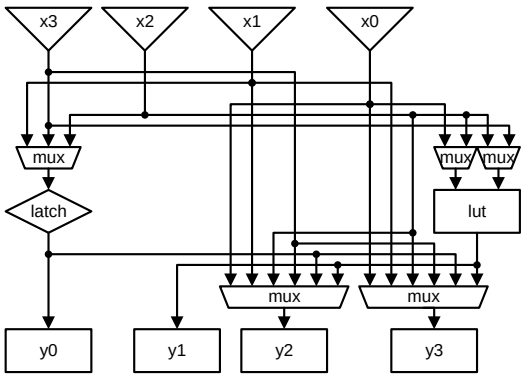


Abbildung 5.4:
Zelle *CSR0404V002*

SM3V001S

Abbildung 5.5 zeigt eine dreidimensionale Version der Variante 1 der einfachen Maschenarchitektur. Die Erweiterung der Dimension wurde durch eine Verbesserung der Verzweigungsmöglichkeiten von Signalen motiviert. Demnach wird ein über die Pins x_2 bzw. y_2 die Dimension wechselndes Signal direkt wieder an den Eingang einer Look-Up-Table gelegt (Abbildung 5.6).

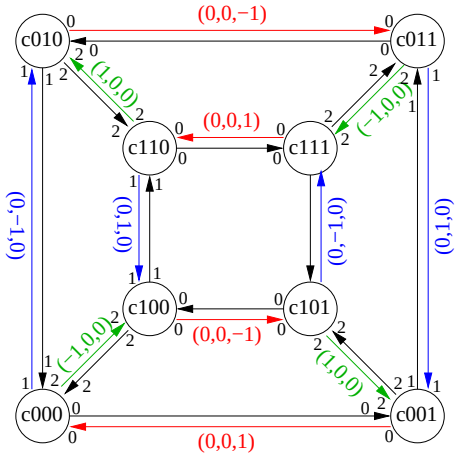


Abbildung 5.5:
Architektur *SM3V001S*

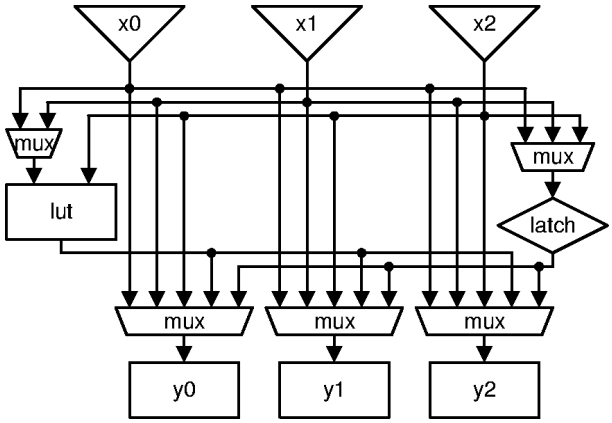


Abbildung 5.6: Zelle *CSR0303V001*

DM2V001S

In [79] wurde anschaulich gezeigt, wie die Verdrahtungsflexibilität der einfachen Maschenarchitektur durch eine Kombination von vier Modulen in jeweils unterschiedlich gespiegelter 2×2-Anordung angehoben werden kann. Wir bezeichnen eine solche Architektur auch als *Doppelmasche* (siehe Abbildung 5.7). Ihre konfigurierbaren Zellen sind identisch zu jenen der Einmaschenversion (Abbildung 5.2).

DM2V002S

Bisher wurden lediglich Architekturen mit Verbindungen zwischen direkten Nachbarzellen betrachtet. Häufig werden Signale jedoch auch an entfernteren Look-Up-Tables benötigt. Um die Zahl der Routing-Tasks entlang eines Pfades solcher Signale und damit auch das resultierende Delay zu reduzieren, wurden mit der Variante 2 der Doppelmaschenarchitektur zusätzliche Leitungen eingeführt, mittels derer ein Signal die benachbarte Zelle überspringen kann (*double span wires*, siehe Abbildung 5.8). Die hierbei zusätzlich eingefügten Eingänge und Ausgänge der konfigurierbaren Zelle wurden bezüglich Verwendbarkeit den bisherigen gleichwertig gestellt (Abbildung 5.9).

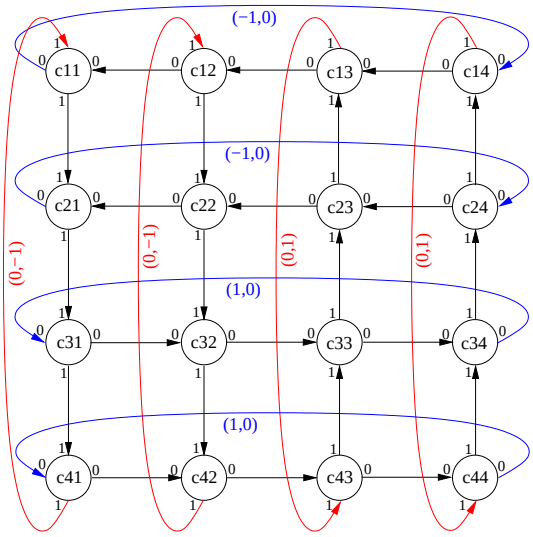


Abbildung 5.7: Architektur *DM2V001S*

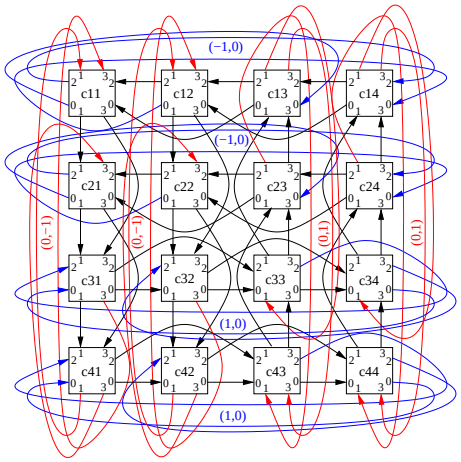


Abbildung 5.8:
Architektur *DM2V002S*

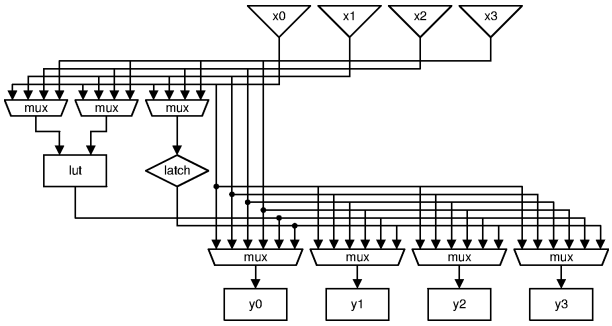


Abbildung 5.9: Zelle *CSR0404V001*

DM2V003S

Mit der dritten Variante der Doppelmaschenarchitektur wurde versucht, die Kosten der Architektur zu senken, indem im Gegensatz zur Variante 1 nun zwölf der bisher 16 Latches und acht der bisher 16 Look-Up-Tables pro Modul eingespart wurden. Abbildung 5.10 zeigt, daß der statische Graph jenem der ersten Architekturvariante der Doppelmasche entspricht, jedoch wurden zwölf konfigurierbare Zellen in ihrer Funktionalität reduziert: weiße Knoten in der Abbildung stellen Instanzen der ursprünglichen Zellen dar (Abbildung 5.2), hellgraue Knoten repräsentieren Zellen, die um die Speichertasks reduziert wurden (Abbildung 5.11) und dunkelgraue Knoten stehen für reine Routing-Zellen (Abbildung 5.12).

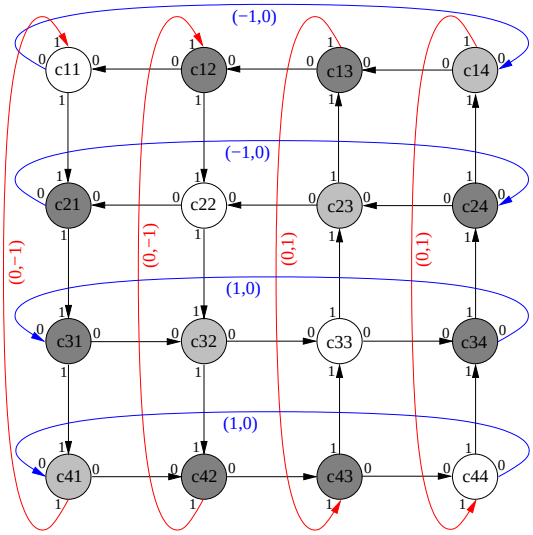


Abbildung 5.10:Architektur *DM2V003S*

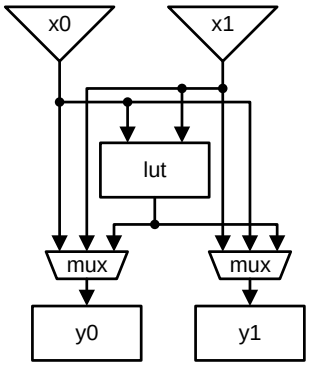


Abbildung 5.11:
Zelle *CR0202V001*

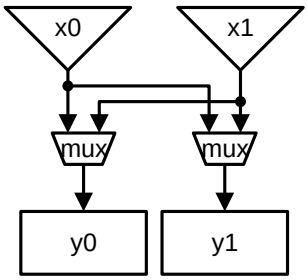


Abbildung 5.12:
Zelle *R0202V001*

5.1.2 Inselarchitekturen

MX2V001S

Die Grundform der betrachteten Inselarchitekturen ist in Abbildung 5.13 illustriert. Jedes Modul der Architektur enthält einen Logikblock (LB) mit jeweils einer Look-Up-Table und einem Latch (Abbildung 5.14). Die Eingangs- und Ausgangssignale der Logikblöcke laufen ausschließlich in Connector-Blöcke (CB) ein, welche die in Abbildung 5.15 dargestellte Struktur besitzen. Ein Connector-Block ist demnach in der Lage, jeden seiner Eingänge x_i für $i \in I = \{0,\ldots,5\}$ auf jeden Ausgang y_j mit $j \in I \setminus \{i\}$ zu schalten, was in diesem Falle der Vermeidung direkter Rückkopplungen zum Logikblock entspricht.

Die Switch-Blöcke (SB) besitzen die Aufgabe, einlaufende Signale in horizontaler und vertikaler Richtung an ihre benachbarten Connector-Blöcke zu routen. Ihrer in Abbildung 5.16 illustrierten Struktur ist in Verbindung mit der in Abbildung 5.13 dargestellten Anordnung ihrer Pins zu entnehmen, daß jedes einlaufende Signal auf jeder der drei gegenüberliegenden Seiten zu genau einem Ausgangssignal verdrahtbar ist.

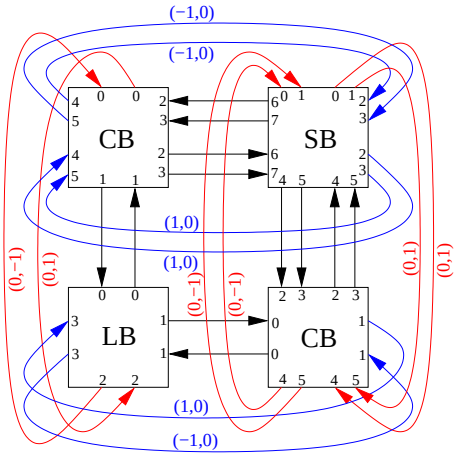


Abbildung 5.13:
Architektur *MX2V001S*

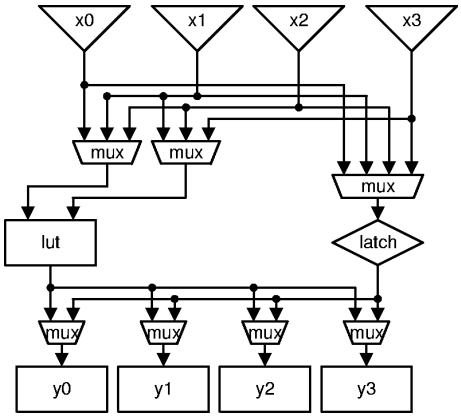


Abbildung 5.14: Zelle *CS0404V001*

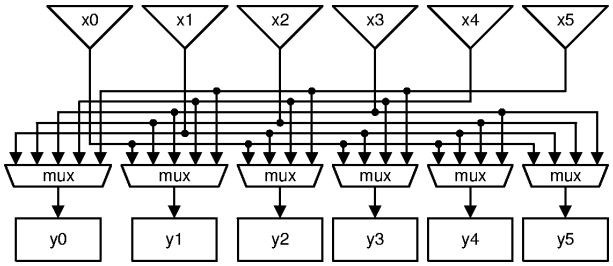


Abbildung 5.15: Zelle *R0606V001*

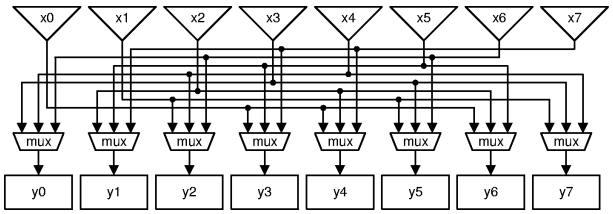


Abbildung 5.16: Zelle *R0808V001*

MX2V002S

Eine Erweiterung der soeben vorgestellten Inselarchitektur um *Double-Span*-Leitungen stellt die Variante 2 in Abbildung 5.17 dar. Während der Logikblock unverändert blieb, erforderte die Einführung der zusätzlichen Verdrahtungsressourcen eine Erweiterung der Connector-Blöcke (Abbildung 5.18) und der Switch-Blöcke (Abbildung 5.19).

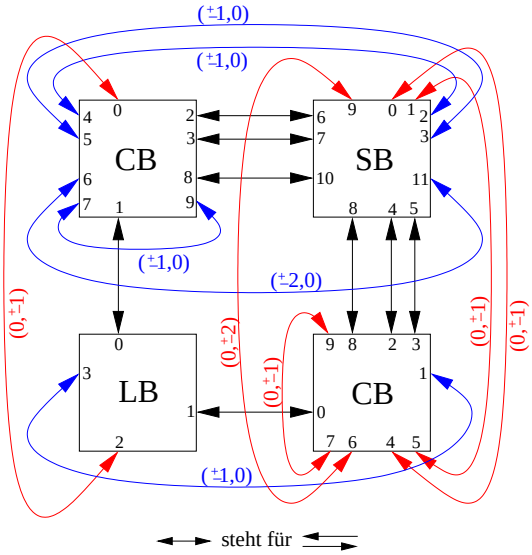


Abbildung 5.17:Architektur *MX2V002S*

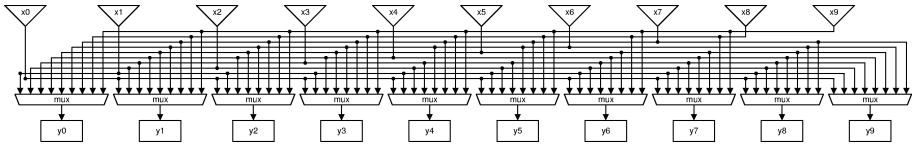


Abbildung 5.18: Zelle *R1010V001*

MX3V001S

Schließlich wurde auch eine dreidimensionale Inselarchitektur definiert, wobei sich das Problem stellte, hinsichtlich der Anordnung der konfigurierbaren Zellen zwischen den in Abbildung 5.20 dargestellten möglichen Varianten zu wählen.

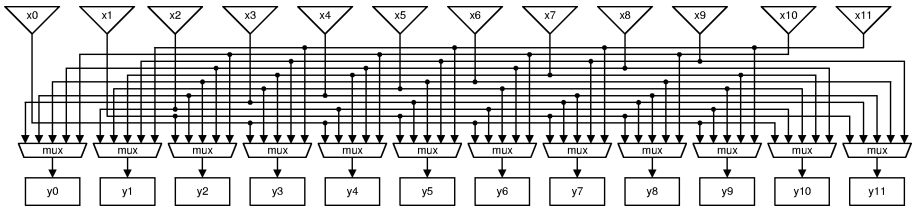


Abbildung 5.19: Zelle *R1212V001*

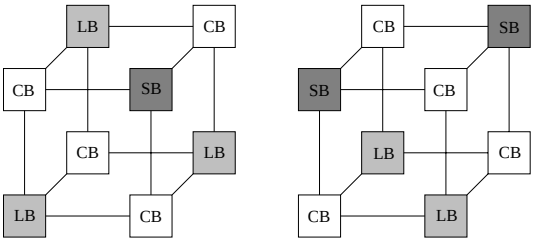


Abbildung 5.20: Anordnungsmöglichkeiten für 3D-Inselarchitektur

Da wir von einem größeren Bedarf an Verdrahtungsressourcen, als an Logikressourcen ausgehen, wählen wir die zweite Variante mit zwei Logikblöcken und zwei Switch-Blöcken (Abbildung 5.21), wodurch bei der dreidimensionalen Entfaltung der Architektur reine „Verdrahtungslagen“ entstehen. Die Logikblöcke wurden um zwei gleichberechtigte Eingänge und zwei Ausgänge erweitert (Abbildung 5.22), die Connector-Blöcke in Bezug auf Variante 1 ebenso (Abbildung 5.23). Die Switch-Blöcke wurden von Variante 2 beibehalten (Abbildung 5.19).

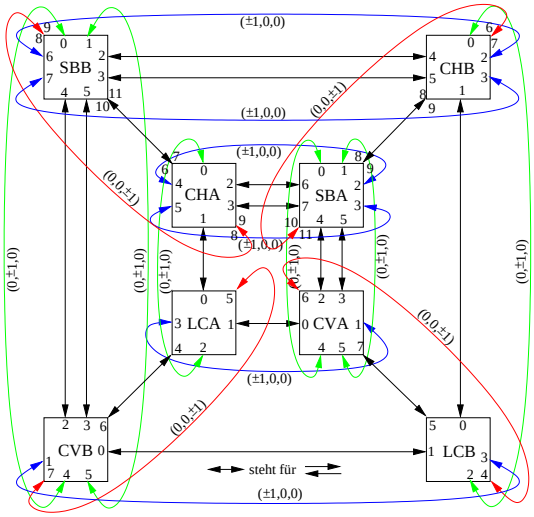


Abbildung 5.21:Architektur *MX3V001S*

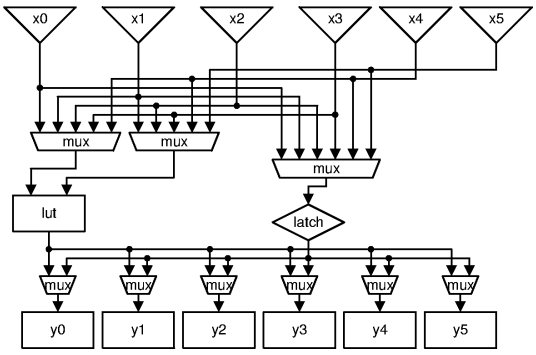


Abbildung 5.22: Zelle *CS0606V001*

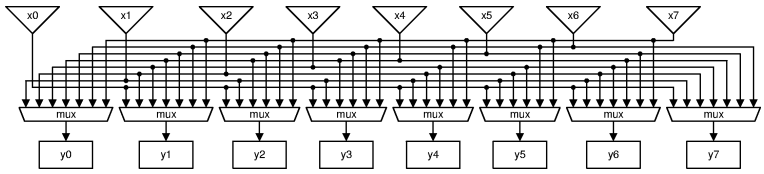


Abbildung 5.23: Zelle *R0808V002*

5.2 Ergebnisse

5.2.1 Architekturmetriken

In diesem Abschnitt sollen die zuvor eingeführten Architekturen anhand der der in Abschnitt 2.5.2 definierten Architekturmetriken charakterisiert werden. Die Betrachtung konzentriert sich primär also eher auf die im Zuge der Spezifikation verfolgten *Eigenschaften*, denn auf die durch diese erzielten *Auswirkungen* bei einer Implementierung von Schaltkreisen. So steht zunächst vielmehr eine quantitative Betrachtung der Architekturen im Vordergrund – qualitative Untersuchungen werden hingegen mit den danach folgenden Unterabschnitten begonnen.

Die Tabellen 5.1 und 5.2 zeigen die zur Berechnung der in Abschnitt 2.5.2 eingeführten Architekturmetriken notwendigen Werte für Typen konfigurierbarer Zellen und Architekturen: die Flächenkosten A, die Anzahl K^{LUT} bzw. K^{KMN} der Konfigurationsbits der Look-Up-Tables bzw. Multiplexer, sowie die minimale Zykluszeit $D^{\min}$ einer Architektur.

Zelltyp c	A_c	K_c^{LUT}	K_c^{KMN}
CSR0202V001	37,5	4	5
CSR0303V001	61	4	12
CSR0404V001	93	4	18
CSR0404V002	54,5	4	10
CR0202V001	23,5	4	4
CS0404V001	56	4	10
CS0606V001	71	4	15
R0202V001	6	0	2
R0606V001	57	0	14
R0808V001	48	0	16
R0808V002	104	0	24
R1010V001	151	0	32
R1212V001	138	0	36

Tabelle 5.1:
Kosten für konfigurierbare Zellen

Architektur $\mathcal{A}$	$A_{\mathcal{A}}$	ΣK^{LUT}	ΣK^{KMN}	$D_{\mathcal{A}}^{\min}$
SM2V001S	150	16	20	4
SM2V002S	218	16	40	6
SM3V001S	488	32	96	5
DM2V001S	600	64	80	4
DM2V002S	1488	64	288	7
DM2V003S	292	32	52	4
MX2V001S	218	4	54	5
MX2V002S	496	4	118	5
MX3V001S	928	8	214	6

Tabelle 5.2: Kosten für Architekturen

Für die verschiedenen Dichtemetriken ergeben sich damit die in Tabelle 5.3 aufgelisteten (gerundeten) Werte, welche anhand graphischer Diagramme nachfolgend noch genauer betrachtet werden sollen.

Abbildung 5.24 zeigt graphisch die Informationsdichte $S_{\mathcal{A}}^{\mathrm{info}}$ jeder untersuchten Architektur $\mathcal{A}$. Der untere Teil der gesplitteten Diagrammbalken stellt dabei jeweils die Logikdichte $S_{\mathcal{A}}^{\mathrm{logic}}$, der obere Teil die Verdrahtungsdichte $S_{\mathcal{A}}^{\mathrm{route}}$ dar.

Architektur $\mathcal{A}$	$S_{\mathcal{A}}^{\text{logic}}$	$S_{\mathcal{A}}^{\text{route}}$	$S_{\mathcal{A}}^{\text{info}}$	$B_{\mathcal{A}}^{\text{res}}$	$S_{\mathcal{A}}^{\text{comp}}$
SM2V001S	0.107	0.133	0.240	0.800	0.026666
SM2V002S	0.073	0.183	0.256	0.400	0.012232
SM3V001S	0.066	0.197	0.263	0.333	0.013115
DM2V001S	0.107	0.133	0.240	0.800	0.026666
DM2V002S	0.043	0.194	0.237	0.222	0.006144
DM2V003S	0.110	0.178	0.288	0.615	0.027397
MX2V001S	0.018	0.248	0.266	0.074	0.003670
MX2V002S	0.008	0.238	0.246	0.034	0.001613
MX3V001S	0.009	0.231	0.240	0.037	0.001437

Tabelle 5.3: Werte der Architekturmetriken

Auffallend ist, daß sich die Informationsdichten aller Architekturen, trotz deren Unterschiedlichkeit in der Quantität ihrer Ressourcen, mehr oder weniger um den konstanten Wert von 0.25 bewegen. Dies folgt als Konsequenz daraus, daß die betrachteten Architekturen fast ausschließlich aus konfigurierbaren Look-Up-Tables und konfigurierbaren Multiplexern bestehen. Die flächennormierten Summen der Anzahl der Programmierbits von Architekturmodulen bewegen sich nun um den vierten Teil eines Gatteräquivalents, der quantitativ jedoch gerade einem Transistor entspricht, welcher wiederum durch ein Programmierbit leitend oder sperrend konfigurierbar ist. Abweichungen von der Konstanten resultieren insbesondere aus der Anzahl der in einem Modul enthaltenen, selbst nicht-konfigurierbaren, Latches.

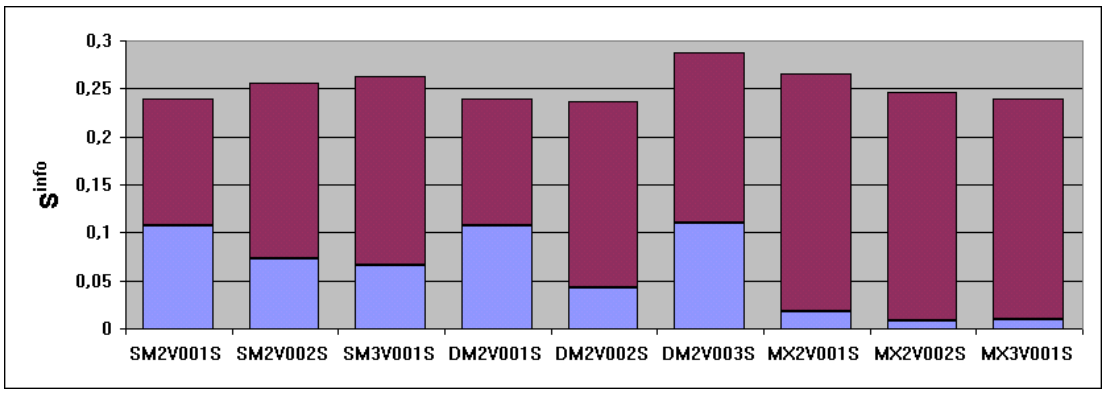


Abbildung 5.24: Informationsdichte $S_{\mathcal{A}}^{\text{info}}$

Die höchsten Logikdichten weisen die Maschenarchitekturen auf, welche – mit Ausnahme der Architektur *DM2V003S* – in jeder konfigurierbaren Zelle eine Look-Up-Table enthalten. Dennoch besitzt auch diese genannte Architektur ebenfalls keine geringe Logikdichte, was auf den Einsatz kostengünstiger Verdrahtungszellen (vgl. Abbildung 5.12) zurückzuführen ist. In den übrigen Fällen ist auch quantitativ zu beobachten, wie unterschiedlich bei vermehrter Investition in die Verdrahtungsressourcen auch die Logikdichte sinkt und die Verdrahtungsdichte steigt. So resultiert auch der Einsatz relativ aufwendiger Connector- und Switch-Blöcke in den Inselarchitekturen letztlich in einer sehr niedrigen Logikdichte.

Das Verhältnis zwischen Logik- und Verdrahtungsressourcen gibt jedoch die in Abbildung 5.25 dargestellte Ressourcen-Balance $B_{\mathcal{A}}^{\text{res}}$ an.

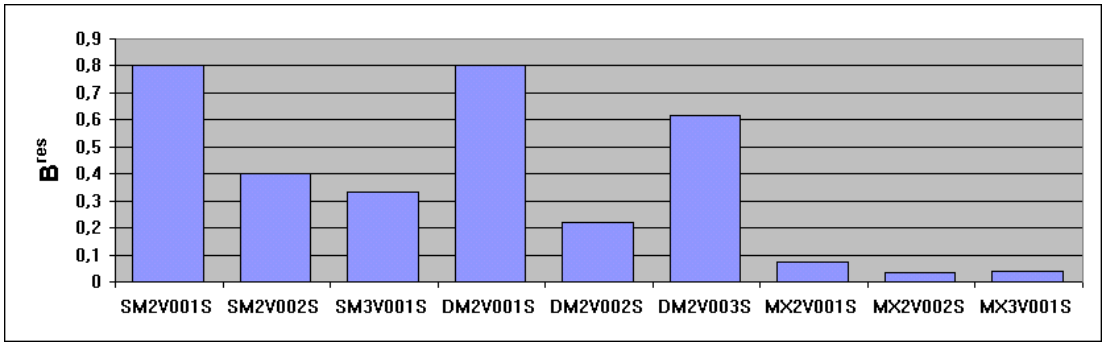


Abbildung 5.25: Ressourcen-Balance $B_{\mathcal{A}}^{\text{res}}$

Die identischen Werte der jeweils ersten Varianten von Einfach- und Doppelmaschenarchitektur (*SM2V001S* bzw. *DM2V001S*) stehen außer Frage – mit einem Verhältnis von vier zu fünf (0.8) sind hier Logik- und Verdrahtungsressourcen beinahe gleichgewichtet. Bereits durch Erweiterung der Zellen um einen Eingang (vgl. *SM3V001S*) oder durch Hinzufügen von *Double-Span*-Leitungen (vgl. *DM2V002S* bzw. *MX2V002S*) verschiebt der Zusatzaufwand an konfigurierbaren Multiplexern die Ressourcen-Balance zugunsten der Verdrahtung um das Doppelte bis 3.6-fache.

Bei Architektur *DM2V003S* ändert sich die Balance zugunsten der Verdrahtungsressourcen – trotz Reduktion der Anzahl der Look-Up-Tables auf die Hälfte gegenüber *DM2V001S* – nur um etwa ein Viertel, da gleichzeitig auch die Größe der Verdrahtungsmultiplexer verringert wurde.

Die Berechnungsdichte $S_{\mathcal{A}}^{\text{comp}}$ charakterisiert als Raum/Zeit-Maß eine obere Schranke für die Leistungsfähigkeit einer Architektur, indem die im *best-case*, also unter optimaler Schaltkreisimplementierung mögliche Anzahl der Funktionsauswertungen unter minimaler Zykluszeit betrachtet wird. Als Zahl der Funktionsauswertungen dient hierbei die flächennormierte Logikdichte.

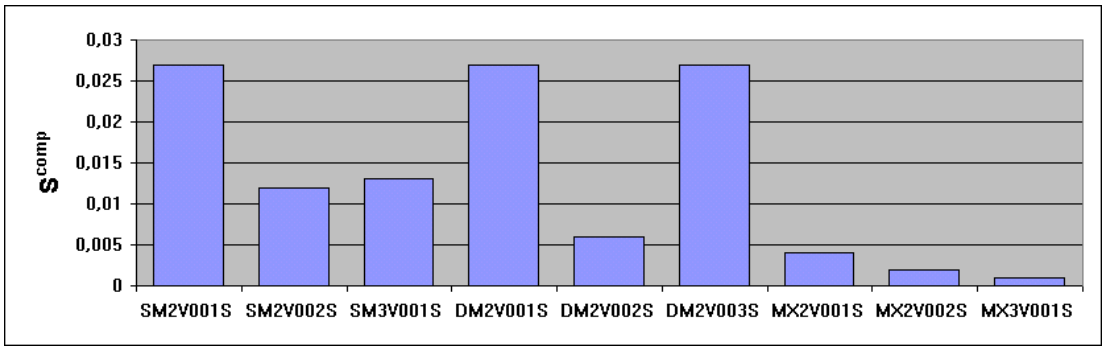


Abbildung 5.26: Berechnungsdichte $S_{\mathcal{A}}^{\text{comp}}$

Die in Abbildung 5.26 graphisch dargestellten Werte der Berechnungsdichte $S_{\mathcal{A}}^{\text{comp}}$ verhalten sich, abgesehen von der definitionsbedingten Größenordnung, im Vergleich zwi-

schen den einzelnen Architekturklassen primär analog zu den Werten der Logikdichte. Betrachtet man aber die Verhältnisse der Werte zueinander, ergeben sich dennoch einige Auffälligkeiten, in denen sich offenbar eine unter einer „günstigen“ Implementierung gegebene, geringere Zykluszeit auswirken kann.

Zunächst zeigen die zweite und dritte Variante der einfachen Maschenarchitektur im Gegensatz zur ersten Variante eine weitaus geringere Berechnungsdichte, als dies deren Logikdichten suggerieren. Andererseist weist die 3D-Architektur *SM3V001S* gegenüber der 2D-Architektur *SM2V002S* eine geringfügig höhere Berechnungsdichte auf, obwohl letztere sogar eine um über 10% höhere Logikdichte besitzt.

Eine starke Diskrepanz zwischen Logik- und Berechnungsdichte zeigt sich auch im Verhältnis der Werte von Doppelmasche *DM2V001S* und ihrer um *Double-Span*-Leitungen erweiterten Variante *DM2V002S*. Während die Logikdichte bei letzterer um rund 60% niedriger liegt, beträgt der Unterschied in der Berechnungsdichte knapp 78%. Mit anderen Worten würde dies zwar bedeuten, daß sich die *Double-Span*-Leitungen erst ab einem ebenso hohen Performanz-Gewinn lohnen würden. Dies ist jedoch insofern hypothetisch, als nicht für jeden beliebigen Schaltkreis eine Implementierung unter der minimalen Zykluszeit $D_{\mathcal{A}}^{\min}$ der gegebenen Architektur $\mathcal{A}$ existiert.

Doch sind es nicht zuletzt Zusammenhänge wie dieser, die neben einer Bewertung mittels theoretischer Architekturmetriken auch auf die Notwendigkeit der Untersuchung mittels Implementierungen von Schaltkreisen hinweisen. Diesem Punkt wenden sich nun die beiden nächsten Unterabschnitte zu.

5.2.2 Implementierung von Modellschaltkreisen

Programmparameter

Die Flexibilität des im Rahmen dieser Arbeit entwickelten generischen Layout-Systems für Schaltkreise und feldprogrammierbare Architekturen wird stark beeinflußt durch eine Reihe von Programmparametern, auf deren Wahl zuvor kurz eingegangen werden soll und die im Laufe aller durchgeführten Versuche aufgrund der Vergleichbarkeit nicht mehr manipuliert wurden.

Als einen der einflußreichsten Parameter ist dem System die *Maximaldilatation* eines Schaltkreismakros vorzugeben. Die Anzahl der in einem Architektursegment maximaler Dilatation vorhandener Look-Up-Tables und Latches diente nun in den Versuchen als maximale Kapazität einer Schaltkreispartition. Die Größe der Prioritätsliste für Verschiebungskandidaten des *Bubble*-Partitionierers wurde hierbei auf zehn Knoten beschränkt. Die Option der Duplikation von Schaltkreisknoten wurde deaktiviert. Die Kosten für temporäre Routen wurden auf zehn Prozent ihres Nominalwertes gesetzt. Die maximale Anzahl der Re-Iterationen betrug bei konfliktierenden Pfaden zehn und bei Ressourcensperrungen fünf.

Der Faktor δ zur Anpassung der Dilatation beim Floorplanning wurde auf den Wert 2.0 fixiert. Die Kosten für temporäre Routen wurden bei der Makroverdrahtung jedoch nur auf 50 Prozent herabskaliert, da diese im Gegensatz zu den bei der Makrogenerierung erzeugten temporären Routen im allgemeinen nicht wieder entfernt werden und sich somit auf das resultierende Delay der Implementierung auswirken können.

Versuchsaufbau

Die dem Layout-System übergebene Maximaldilatation für zu generierende Makros hat natürlich große Auswirkung auf die Anzahl der Makros, auf die Anzahl der zwischen diesen verlaufenden Netzen und über die implizit resultierende Granularität der Makromenge schließlich auch auf deren Verdrahtungsaufwand, also auf Kosten und Delay einer Implementierung.

Die Untersuchung der Auswirkungen verschiedener Makrodilatationen steht nun primär im Mittelpunkt der Betrachtungen zunächst dieses Unterabschnitts. Dazu wurden verschiedene Implementierungen berechnet von geeigneten Modellschaltkreisen, welche ein einfaches, regelmäßiges Berechnungsschema repräsentieren, wobei Implementierungen auch visuell verifizierbar bleiben sollten. Deshalb fiel die Entscheidung auf Schaltkreise von der Struktur binärer Bäume, für welche mit der H-Baum-Struktur auch bereits eine delayeffiziente zweidimensionale Implementierung bekannt ist.

Diese *bintree*-Schaltkreise wurden als vollständige Binärbäume mit $n = 3, 7, 15$ und 31 Knoten konstruiert. Ein solcher Schaltkreis besitzt die Tiefe $\log_2 n+1$ und hat n Eingänge sowie einen Ausgang. Da jeder Knoten zwei Eingangsnetze besitzt, beträgt die Zahl der Netze $2n$. Jeder Schaltkreis wurde separat auf allen vorgestellten Architekturen für quadratische bzw. kubische Dilatationen gleicher Seitenlängen k mit $k = 1, 2, 3, \ldots$ implementiert, wobei die Länge k solange erhöht wurde, bis die Anzahl der generierten Makros schließlich eins betrug, d.h. der Schaltkreis vollständig innerhalb eines Makros implementiert werden konnte.

Ergebnisse

Von den berechneten Implementierungen betrachten wir an dieser Stelle exemplarisch jene der beiden strukturell recht unterschiedlichen Architekturen *SM2V001S* und *MX2-V001S*. Die Abbildungen 5.27 und 5.28 zeigen die Kosten- und Delaywerte der verschiedenen *bintree*-Implementierungen auf der einfachen Maschenarchitektur *SM2V001S*. Die Werte $1, 2, \ldots, 5$ auf der Abszisse stehen hierbei als Indizes der Folgen der für die einzelnen Schaltkreise unter ansteigender Maximaldilatation berechneten Implementierungen.

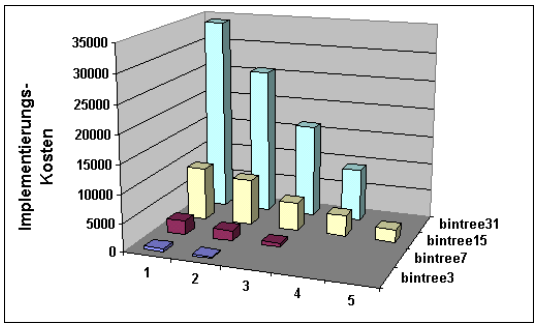


Abbildung 5.27: Implementierungskosten für SM2V001S

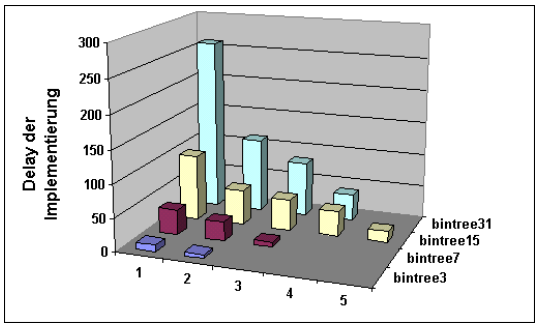


Abbildung 5.28: Delay der Implementierung für SM2V001S

Sowohl bei den Kosten- als auch den Delaywerten scheint offensichtlich, daß die Begrenzung der Dilatation der Makros wesentlichen Einfluß auf die Qualität des berechneten Schaltkreislayouts besitzt. So liefert beispielsweise die Ein-Makro-Implementierung des Schaltkreises *bintree31* um fast 73% geringere Kosten als die Implementierung unter 1×1-Dilatation mit 25 generierten Makros. Die Delaywerte dieser Implementierungen unterscheiden sich sogar um knapp 85%. Die prozentualen Verhältnisse zwischen diesen „extremalen Implementierungen“ ändern sich interessanterweise auch bei kleineren Schaltkreisen nur geringfügig, wie Tabelle 5.4 zeigt.

Schaltkreis	Kostendifferenz	Delaydifferenz
bintree31	72.9%	84.6%
bintree15	76.6%	84.0%
bintree7	74.2%	81.6%
bintree3	60.0%	50.0%

Tabelle 5.4: Verhältnisse extremaler Implementierungen (*SM2V001S*)

Die Abbildungen 5.29 und 5.30 zeigen die Kosten- und Delaywerte der Implementierungen auf der Inselarchitektur *MX2V001S*.

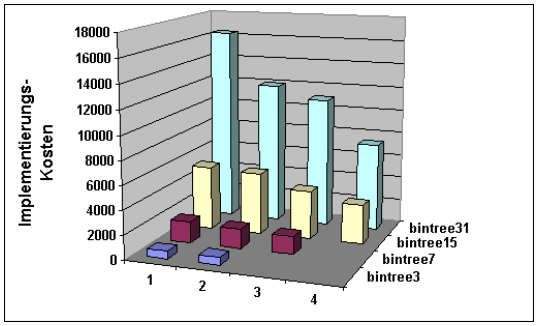


Abbildung 5.29: Implementierungskosten für MX2V001S

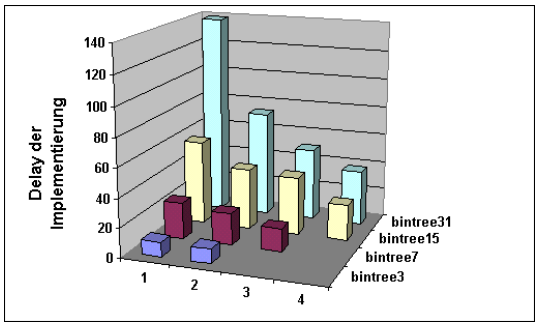


Abbildung 5.30: Delay der Implementierung für MX2V001S

Auch bei dieser Architektur, deren Schwerpunkt – wie bereits gesehen – offensichtlich auf den Verdrahtungsressourcen liegt, zeigt sich der deutliche Einfluß der Makrodilatation. Sieht man sich allerdings hier die Verhältnisse der Werte der extremalen Implementierungen genauer an, bietet sich ein etwas anderes Bild (siehe Tabelle 5.5).

Schaltkreis	Kostendifferenz	Delaydifferenz
bintree31	55.0%	72.7%
bintree15	39.0%	56.9%
bintree7	18.1%	36.0%
bintree3	0.0%	0.0%

Tabelle 5.5: Verhältnisse extremaler Implementierungen (*MX2V001S*)

Zunächst sind hier neben den Werten der Kosten- und Delaymaße auch deren maximale Differenzen innerhalb der berechneten Implementierungen weitaus geringer, als dies bei der Maschenarchitektur zu beobachten war. Während das Phänomen der generell niedrigeren Kosten- und Delaywerte im nächsten Unterabschnitt noch genauer betrachtet werden wird, zeigt sich in den ebenfalls geringeren Differenzen der extre-

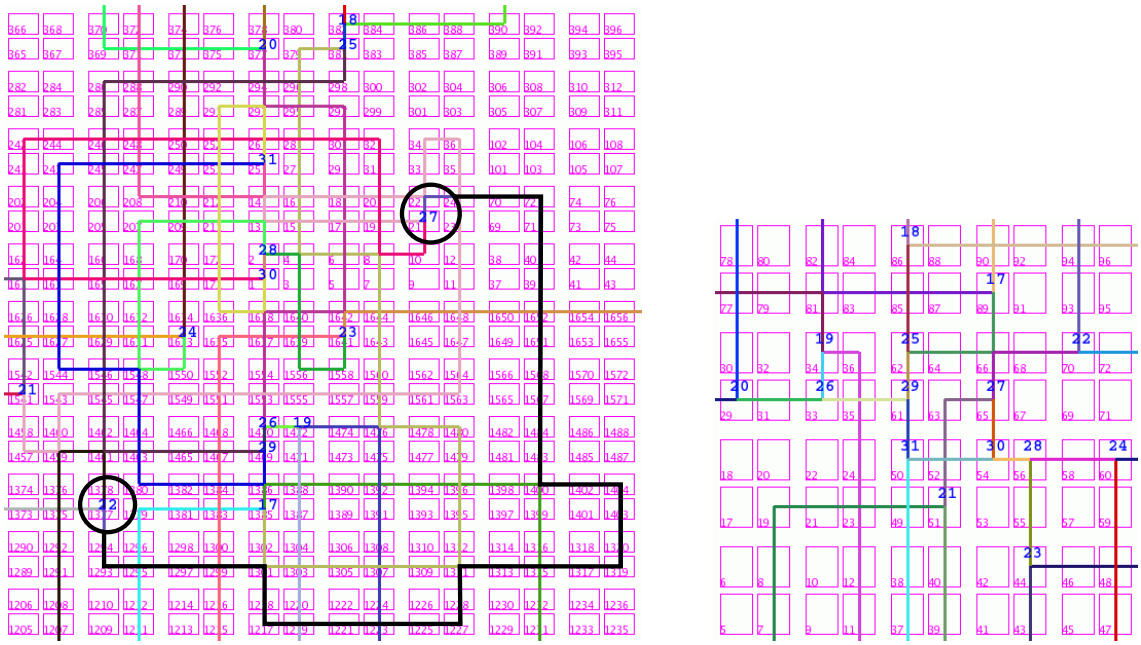
malen Implementierungen, daß ein Floorplanning von Makros auf einer Inselarchitektur vorteilhafter erscheint, als auf einer Maschenarchitektur, da aufgrund der höheren Verdrahtungsflexibilität der Inselarchitektur häufig auch zwischen den angeordneten Makros hindurch verdrahtet werden kann, wohingegen bei der in ihren Verdrahtungsmöglichkeiten stark eingeschränkten Maschenarchitektur oft Umwege um plazierte Makrogruppen genommen werden müssen.

Erstaunlicherweise scheint ferner bei der Inselarchitektur im Gegensatz zur Maschenarchitektur nun auch die Größe des Schaltkreises eine Rolle zu spielen, insofern, als sich vollständig mit dem Makro-Generator implementierte und durch Makro-Floorplanning berechnete Schaltkreis-Layouts unter abnehmender Schaltkreisgröße offenbar qualitativ annähern. Auch diese Beobachtung läßt sich auf die stärker eingeschränkte Verdrahtungsflexibilität der einfachen Maschenarchitektur gegenüber der betrachteten Inselarchitektur zurückführen. Denn während bei der Maschenarchitektur ein konfigurierter Logiktask mit seinen drei beteiligten Netzen eine konfigurierbare Zelle für andere Netze komplett blockiert, werden an einen Logikblock der Inselarchitektur lediglich die an der dort zu berechnenden Funktion beteiligten Netze heranverdrahtet. Bis zu drei weitere, also fremde Netze können in den benachbarten *Connector*- und *Switch*-Blöcken das Architekturmodul dennoch passieren. Je weniger Netze nun der Schaltkreis besitzt, desto besser können in der Inselarchitektur diese Ressourcen lokal auch bei der Makroverdrahtung genutzt werden. Beim Floorplanning auf der Maschenarchitektur entstehen hingegen oft undurchquerbare Regionen, die aufgrund fester Makroterminal-Lokationen zu aufwendiger Verdrahtung führen. Die direkte Implementierung eines Schaltkreises mittels der *Route-then-Place*-Methode des Makro-Generators umgeht jedoch genau dieses Problem, indem hier die Lage der Terminale erst durch die erforderlichen Routen festgelegt wird. Dadurch entstehen Kostenunterschiede, die im wesentlichen unabhängig von der Größe des zu implementierenden Schaltkreises sind.

Am Beispiel der extremalen Implementierungen des Schaltkreises *bintree15* kann dies auch visuell beobachtet werden (siehe Abbildung 5.31). In Implementierung a) der Abbildung ist deutlich zu erkennen, wie beispielsweise das zwischen den Schaltkreisknoten 22 und 27 verlaufende Netz mangels Ressourcen ungünstigerweise um das „Zentrum“ des Layouts herumgeführt werden muß. Beim direkten Layout durch den Makrogenerator (Implementierung b) mußten hingegen praktisch keine Umwege in Kauf genommen werden.

Die vorangegangenen Beobachtungen zeigen somit, daß eine Implementierung von Schaltkreisen insbesondere auf feingranularen, in ihren Verdrahtungsressourcen relativ eingeschränkten Architekturen mittels Makro-Floorplanning und Makro-Verdrahtung in der Regel zu keinen guten Ergebnissen führt. Hinsichtlich der Bewertung entsteht dabei die Gefahr, daß manche Architekturen schlechter dargestellt werden könnten, als sie tatsächlich sind. Zur Sicherstellung einer fairen Bewertung der Architekturen, werden wir Schaltkreise im Rahmen unserer nun folgenden Architekturuntersuchungen nicht zerlegen, sondern direkt durch das entwickelte *Route-then-Place*-Verfahren zu implementieren versuchen, wobei hinsichtlich der Größe der Schaltkreise in Anbetracht von Rechenzeit und Speicherbedarf ein Kompromiß zu treffen ist.

Makro-Floorplanning-Verfahren spielen dennoch eine wichtige Rolle bei der Implementierung größerer Schaltkreise, sowie bei Implementierungsproblemen, die neben Funktionsknoten auch in, im Zuge von Spezifikation oder Synthese erhaltenen Makro-



a) b)

Abbildung 5.31: Extremale Implementierungen des Schaltkreises *bintree15*

blöcken aus Komponenten-Bibliotheken bestehen, welche beispielsweise bereits feste funktionale Einheiten, wie etwa Addierer, realisieren.

5.2.3 Implementierung von Benchmark-Schaltkreisen

Schaltkreise

In diesem Unterabschnitt sollen nun die vorgestellten neun Architekturen mittels der Implementierung verschiedener Schaltkreise genauer betrachtet und bewertet werden. Tabelle 5.6 zeigt die Liste der aus der MCNC-Suite [1] ausgewählten Schaltkreise.

Schaltkreis	PIs	POs	Knoten	Netze
bbara	5	2	62	67
bbtas	3	2	26	29
cm150a	21	1	42	63
cm151a	12	2	23	35
cm152a	11	1	21	32
cm162a	14	5	29	43
cm42a	4	10	21	25
cm82a	5	3	10	15
decod	5	16	28	33
lion	3	1	11	14
majority	5	1	9	14
mc	4	5	17	21
parity	16	1	15	31
rd53	5	3	23	28
s27	5	1	16	21
xor5	5	1	4	9
z4ml	7	4	23	30

Tabelle 5.6: Auswahl von Benchmark-Schaltkreisen

Die Auswahl der Schaltkreise orientierte sich in erster Linie an der Rechenzeit und dem Arbeitsspeicher der für die Versuche zur Verfügung stehenden Rechnern. Hierbei handelte es sich um einen Pool von 13 PCs unter dem Betriebssystem Linux mit jeweils bis zu 450MHz Prozessortakt und 256MB Hauptspeicher, sowie einem PC mit 2GHz Prozessortakt und 1GB Hauptspeicher. Die in Diagrammen nachfolgend dargestellten Ergebnisse zeigen für jede der betrachteten Architekturen das auf die entsprechende Bewertungsmetrik angewandte arithmetische Mittel über alle implementierten Schaltkreise.

Architektur-Ausnutzung

Abbildung 5.32 skizziert das Verhalten der Architekturen hinsichtlich der Ausnutzung

$$A_{\mathcal{I}}^{\text{use}} = \frac{A_{\mathcal{I}}}{A_{\mathcal{I}}^{\square}}$$

der Plazierungsebene, wobei der untere Teil der gesplitteten Balken des Diagramms für den Logikanteil, genauer, für den Anteil der durch die Implementierung von Schaltkreisknoten genutzten konfigurierbaren Zellen, während der obere Teil der Balken für den Verdrahtungsanteil steht. Die niedrigste Ausnutzung weisen hierbei die dreidi-

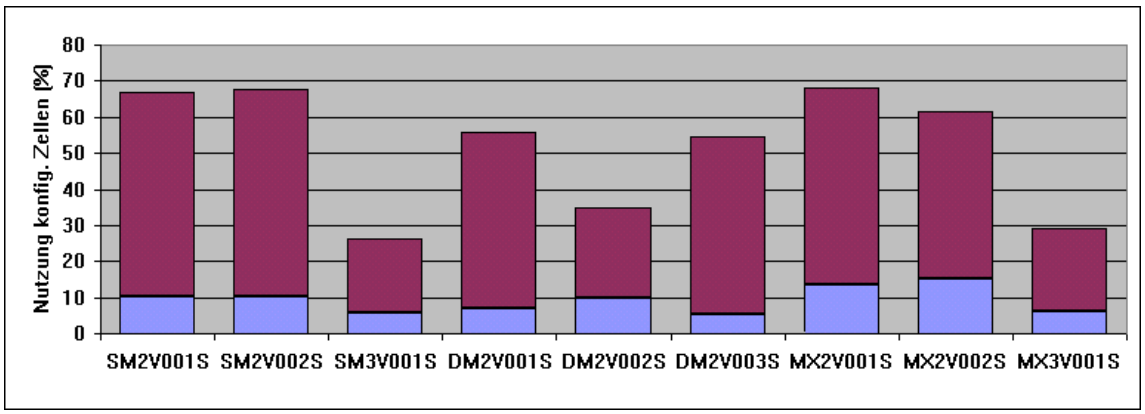


Abbildung 5.32: Architektur-Ausnutzung $A_{\mathcal{I}}^{\text{use}}$

mensionalen Architekturen *SM3V001S* und *MX3V001S* auf, was in Anbetracht der im 3D-Fall größeren *Bounding Box* auch zu erwarten war. Der Grund für die ebenfalls weit unter 50% liegende Ausnutzung der *DM2V002S*-Architektur liegt in deren lokal recht geringen Verdrahtungsflexibilität, wodurch sich Implementierungen sehr weit über die *Double-Span*-Leitungen zu erstrecken vermögen. Während die zweidimensionalen einfachen Maschenarchitekturen *SM2* zusammen mit den 2D-Inselarchitekturen *MX2* mit Werten zwischen 60% und 70% die höchsten Ausnutzungen unter den betrachteten Architekturen zeigen, sind die Verhältnisse zwischen Logik- und Verdrahtungsanteil bei den letztgenannten Architekturen sogar noch zu relativieren, da lediglich eine von vier konfigurierbaren Zellen eines *MX2*-Architekturmoduls überhaupt Logikressourcen bietet. Insofern ermöglichen diese Architekturen hinsichtlich der Implementierung der Schaltkreisknoten offenbar doch sehr kompakte Anordnungen. Die *SM2*-Architekturen wenden im Gegensatz tatsächlich viele ihrer Zellen für Verdrahtungstasks auf.

Eine hohe Architektur-Ausnutzung für Verdrahtungsaufgaben muß andererseits jedoch nicht zwangsläufig eine „Verschwendung“ von Ressourcen bedeuten. Abbildung 5.33 zeigt die Werte des über die Schaltkreise gemittelten Kostenoverheads

$$A_{\mathcal{I}}^{\mathrm{ov}} = \frac{A_{\mathcal{I}}}{\hat{A}_{\mathcal{C}}}$$

(helle Balken), der die tatsächlichen Kosten der Implementierung ins Verhältnis setzt zu den Kosten der schaltkreisspezifischen Ideal-Architektur. Entsprechend geben die dunklen Balken graphisch die Werte des Delayoverheads

$$D_{\mathcal{I}}^{\mathrm{ov}} = \frac{D_{\mathcal{I}}}{\hat{D}_{\mathcal{C}}}$$

wieder. Wir betrachten zunächst nur die Werte des Kostenoverheads.

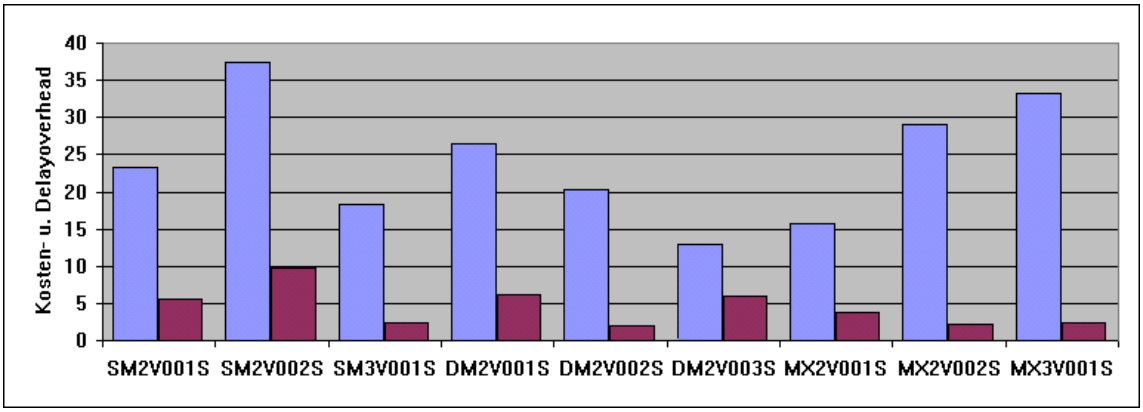


Abbildung 5.33: Kosten- und Delayoverhead $A_{\mathcal{I}}^{\mathrm{ov}}$ bzw. $D_{\mathcal{I}}^{\mathrm{ov}}$

Kostenoverhead

Wie im vorangegangenen Unterabschnitt bereits beobachtet, opfern Maschenarchitekturen wie *SM2V001S* und *DM2V001S* relativ viele Look-Up-Table- und Latch-Ressourcen für die Verdrahtung und produzieren damit bereits einen nicht geringen Kostenoverhead.

Auf der um Rückkopplungen erweiterten *SM2V002S*-Architektur wurden nun ähnliche Implementierungen, wie für die Architektur *SM2V001S* berechnet, da offensichtlich eine durch die Rückkopplungsleitungen angestrebte Mehrfachnutzung der Zellen nicht erreicht wurde. Die höheren Modulkosten (vgl. Tabelle 5.2) der *SM2V002S*-Architektur resultierten schließlich sogar im höchsten Kostenoverhead aller betrachteten Architekturen.

Auch die Architekturen *MX2V002S* und *MX3V003S* weisen einen recht hohen Kostenoverhead auf. Scheinbar kann hier selbst die höhere Verdrahtungsflexibilität die ebenfalls sehr hohen Architekturkosten durch ein kompakteres Layout nicht kompensieren.

Andererseits besitzt die Architektur *DM2V002S* mit Abstand die höchsten Architekturkosten, doch erstaunlicherweise den viertbesten Kostenoverhead unter den neun be-

trachteten Architekturen. Aufgrund der durch die *Double-Span*-Leitungen höheren Verdrahtungsflexibilität kann die im Vergleich zu den Inselarchitekturen weitaus höhere Logikdichte hier wohl besser genutzt werden.

Auch die dreidimensionale Architektur *SM3V001S* profitiert hinsichtlich der Implementierungskosten scheinbar von einer erhöhten Verdrahtungsflexibilität bei, unter der einfachen Maschenstruktur noch moderaten Architekturkosten.

Den niedrigsten Kostenoverhead unter den betrachteten Architekturen liefert die um Look-Up-Table- und Latch-Ressourcen reduzierte Architektur *DM2V003*. Offensichtlich liegt hier trotz der in ausschließlich direkten Nachbarschaftsverbindungen bestehenden Verdrahtungsressourcen eine günstigere Ressourcenbalance vor.

Delayoverhead

Während unter den Maschenarchitekturen, bis auf die Ausnahme *DM2V003S*, wohl eine gewisse Korrelation zwischen Kosten und Delay zu bestehen scheint, die jedoch aus dem Vorhandensein jeweils einer Look-Up-Table und eines Latches in jeder konfigurierbaren Zelle und deren Verlust im Verdrahtungsfalle erklärbar wird, kann dies von den Inselarchitekturen nicht gesagt werden.

Zunächst ist jedoch, wie auch erwartet, ein signifikanter *Speedup* bei den um *Double-Span*-Leitungen erweiterten Architekturen *DM2V002S* und *MX2V002S* gegenüber den Versionen mit lediglich direkten Nachbarschaftsverbindungen zu beobachten.

Die Reduktion der *DM2V003S*-Architektur um Logikressourcen senkte zwar die Kosten, hatte aber im Vergleich zur ursprünglichen Variante *DM2V001S* offensichtlich kaum eine Veränderung des Delays zur Folge. Möglicherweise wurden hier ähnliche Implementierungen berechnet.

Die im Mittel ungünstigsten Delays liefert die *SM2V002S*-Architektur. Trotz der hinzugefügten Rückkopplungen an den konfigurierbaren Zellen konnte offenbar ein kompakteres Layout durch Mehrfachnutzung nicht erreicht werden, da jeweils nur eine Look-Up-Table und ein Latch pro Zelle vorhanden sind, aber die auf der Architektur implementierten Schaltkreise nur wenige LUT/Latch-Kombinationen aufweisen.

Erstaunlich gute Delaywerte zeigen hingegen die 3D-Architekturen. Quantitativ genauer betrachtet, bewegen sich die Werte sogar um jene der Architekturen mit *Double-Span*-Leitungen. Allerdings fordert die 3D-Konnektivität beim Inseltyp *MX3V001S* jedoch sehr hohe Kosten. Beim einfachen Maschentyp *SM3V001S* sind die Architekturkosten dagegen etwas niedriger.

Überblick

Abbildung 5.34 zeigt alle neun betrachteten Architekturen nochmals hinsichtlich ihrer Kosten- und Delaywerte in einer Übersicht zur Bestimmung eines Kosten/Delay-Tradeoffs. Aus der Graphik lassen sich nun vier pareto-optimale Architekturen ablesen: während hinsichtlich des Kostenoverheads die logikreduzierte *DM2V003S*-Doppelmasche am besten abschneidet, liegt beim Delayoverhead ihre zweidimensionale Variante *DM2V002S* mit *Double-Span*-Leitungen vorn. Desweiteren dominieren auch die erste Variante der zweidimensionalen Inselarchitektur *MX2V001S*, sowie die einfache 3D-

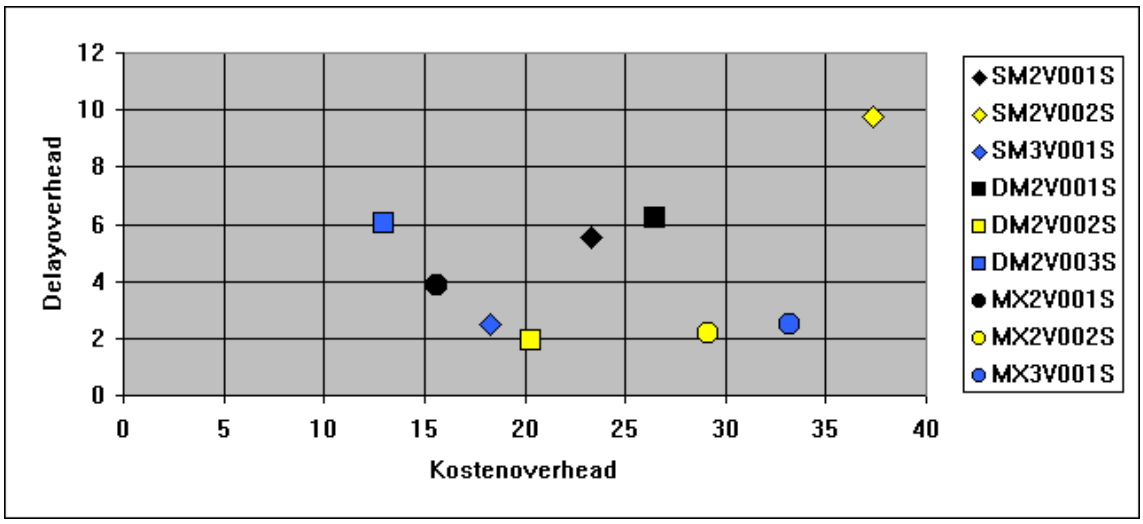


Abbildung 5.34: Kosten/Delay–Tradeoff

Maschenarchitektur *SM3V001S*. Festgehalten werden soll an dieser Stelle noch, daß das auf einer kommerziellen Architektur basierende *MX2V001S*-Modell weder hinsichtlich Kosten, noch hinsichtlich Delay extremal erscheint.

5.3 Zusammenfassung und Ausblick

Zielsetzung der vorliegenden Arbeit bildete die Konzipierung eines Modells für feldprogrammierbare Architekturen, ferner die Entwicklung eines entsprechenden generischen Systems zur Berechnung von Schaltkreis-Implementierungen auf den Architekturen und schließlich die Definition geeigneter Metriken für eine Bewertung von Architekturen, sowie von Schaltkreis-Implementierungen.

Im Zuge der Verfolgung dieser eigentlichen Zielsetzung evolvierten auch eine Reihe, zugegebenermaßen zum Teil etwas unkonventioneller, dafür jedoch neuer Techniken, welche durchaus auch Ansätze für ähnliche Problemstellungen anderer Anwendungsbereiche liefern können, wenn die Verfahren problemspezifisch verfeinert werden.

Modellierung

Nach einer kurzen Einführung, sowie der Motivation eines neuen Ansatzes zur Modellierung feldprogrammierbarer Architekturen, wurde ein generisches Konzept zur Modellierung mittels *routing-integrierender Architekturzellen* auf der Basis konfigurierbarer Multiplexer-Netzwerke vorgestellt. Nachdem, ausgehend von einer exakten funktionalen Einführung der Multiplexer-Netzwerke, schließlich auf Routing-Tasks bzw. Mengen von Routing-Tasks abstrahiert wurde, rückten auch Eigenschaften solcher Netzwerke, insbesondere im Hinblick auf deren Optimierung, ins Zentrum der Betrachtung. An dieser Stelle sei auch erwähnt, daß eine vergleichbare Studie solcher Netzwerke vom Verfasser bis zum gegenwärtigen Zeitpunkt in der Literatur nicht ausfindig gemacht werden konnte.

Zur Charakterisierung der Eigenschaften feldprogrammierbarer Architekturen wurden schließlich Bewertungsmaße eingeführt und motiviert: zum einen handelte es sich hierbei um theoretische Flexibilitätsmetriken, wie beispielsweise Logik-, Verdrahtungs- und Berechnungsdichte, aber auch Metriken zur Bewertung von Schaltkreis-Implementierungen hinsichtlich Architektur-Ausnutzung, Ressourcenkosten und Performanz wurden definiert. Aus diesen *absoluten* Kennziffern wurden sodann anhand einer fiktiven, schaltkreisspezifischen Idealarchitektur auch *relative Metriken* hergeleitet, welche die Einschätzung einer Art „theoretischen *Overheads*“ ermöglichten.

Makrogenerierung

Den ersten Teil des entwickelten generischen Layout-Systems bildete ein integratives Verfahren zur Generierung von Hardmakros zu einem gegebenen Schaltkreis und einer gegebenen Architektur. Hier wurde mit der *Bubble-Partitioning*-Methode ein neues Partitionierungsverfahren vorgestellt, das sich gegenüber einer *Straight-Partitioning*-Greedy-Heuristik als qualitativ besser erwies und in Kombination zu einem Hybrid-Verfahren auch hinsichtlich der Laufzeit überzeugen konnte.

Das simultan die einzelnen Partitionen konstruierende *Bubble Partitioning* wurde sodann verknüpft mit einem *Placing-by-Routing*-Verfahren, das für Schaltkreisknoten, welche vom Partitionierer zur Aufnahme in eine bestimmte Partition vorgeschlagen wurden, anhand der berechneten Netzrouten eine Implementierung findet. Die Netzrouten wurden hierbei durch eine simultane Pfadsuche berechnet, ein an der Steinerbaum-Heuristik von *Wu, Widmayer* und *Wong* orientierter iterativer Verdrahtungsalgorithmus, der konfliktierende Routen, in Anlehnung an die auf *Nair's* Algorithmus basierende *Pathfinder*-Methode, im Zuge von Re-Iterationen durch eine geeignete Kostenfunktion bestraft. Die aus der *Pathfinder*-Methode adaptierte Kostenfunktion wurde für die simultane Pfadsuche jedoch erweitert. Um auch die strukturellen Eigenschaften im Hinblick auf kritische Pfade des zu implementierenden Schaltkreises einfließen zu lassen, wurde ein aus dem Partitionierer gewonnenes Kritizitätsmaß für Netze integriert.

Eine weitere Besonderheit des entwickelten *Placing-by-Routing*-Verfahrens stellte das Konzept der *temporären Routen* dar, welche eingeführt wurden einerseits zur Reservierung von Verdrahtungsressourcen für spätere Netzimplementierungen, zur Reservierung von Plazierungsraum für später implementierte, benachbarte Schaltkreisknoten und andererseits zur Garantierung der Verfügbarkeit von Netzen außerhalb des Makros. Mit einer parametrischen Reduktion der Verdrahtungskosten temporärer Routen ließ sich ferner auch die Kompaktheit der Makroimplementierungen kontrollieren. Für die Klasse der *vernünftigen Architekturen* unbeschränkter Dilatation konnte mit dem Konzept der temporären Routen schließlich auch die „Robustheit“ des Makrogenerierungsverfahrens gezeigt werden insofern, als für jeden Schaltkreis stets eine gültige Implementierung berechnet wird.

Makro-Floorplanning

Im zweiten Abschnitt des generischen Layout-Systems wurden ein Floorplanning- und ein Verdrahtungsverfahren für Makros kombiniert. Hierbei wurde ein neues Floorplanning-Verfahren auf der Basis mehrdimensionaler Mustererkennung vorgestellt, indem

eine Abstraktion von Zeichenketten auf Eigenschaften von Schaltkreis-Implementierungen durchgeführt wurde: das ursprüngliche *Pattern-Matching*-Verfahren wurde zu einem *Property-Containment*-Verfahren adaptiert.

Da dieses Floorplanning-Verfahren alle gültigen Anordnungen eines Makros bezüglich einer Menge in einem beschränkten Plazierungsraum bereits implementierter Makros berechnet, wurde eine Heuristik zur Bewertung der ermittelten Plazierungen durch eine Abschätzung der resultierenden Längen der Netzrouten mittels beschränkter linearer Optimierung entwickelt. In einem Exkurs wurden ferner Untersuchungen für eine Unimodularitätseigenschaft der Bedingungsmatrix des Optimierungsproblems durchgeführt, wobei einige Bedingungen an Architekturbeschreibungen formuliert werden konnten, in denen das lineare Optimierungsproblem als Relaxation des eigentlich zugrundeliegenden *Integer Programms* eine ganzzahlige und damit exakte Lösung liefert.

Da der Verdrahtungsvorgang für Makros im Hinblick auf die Rechenzeiten sehr teuer ist, wurde auf eine Backtracking-Technik bezüglich der Anordnungsreihenfolge verzichtet und stattdessen eine Greedy-Heuristik auf einem, die Konnektivität der Makros repräsentierenden Graphen eingesetzt zur Bestimmung einer möglichst „günstigen“ Anordnungsreihenfolge der Makros. Die Kostenfunktion der Heuristik balancierte dabei Kriterien, wie die Dilatation der Makros, sowie die Minimierung der aus der Anordnungsreihenfolge resultierenden Anzahl temporärer Routen.

Das hinsichtlich erforderlicher Routentypen differenzierte Makro-Verdrahtungsverfahren orientierte sich wieder an der bereits oben erwähnten Steinerbaum-Heuristik. Zur Auflösung von Routenkonflikten wurde hierbei jedoch eine „Strategie der alternativen Pfade“ eingesetzt, welche die Kosten konfliktierender Ressourcen quasi auf die Kosten ebenfalls gefundener Alternativ-Routen anhebt und damit sensitiver bestraft, als die ansonsten häufig angewandte Bestrafungsfunktion des Produktes aus der Anzahl konfliktierender Netze und der Anzahl der Re-Iterationen.

Bewertung

Als Beispiel des Einsatzes der in der vorliegenden Arbeit entwickelten Konzeption wurden auf der Basis des eingeführten Architekturmodells schließlich neun feldprogrammierbare Architekturen spezifiziert, welche strukturell in drei verschiedene Klassen gliederbar sind. Mittels Verfahren des generischen Layoutsytems wurden Implementierungen von Modell- und Benchmark-Schaltkreisen auf den Architekturen berechnet und diese hinsichtlich Architektur-Ausnutzung, Kosten- und Delayoverhead einander gegenübergestellt.

Nach einer Charakterisierung der Architekturen mittels der theoretischen Architekturmetriken, wurde anhand von Modellschaltkreisen unterschiedlicher Größe die Auswirkung der maximalen Makrodilatation auf die vom Layout-System berechneten Implementierungen untersucht. Wie zu erwarten, ließ sich dabei beobachten, daß insbesondere bei Architekturen stark eingeschränkter Verdrahtungsmöglichkeiten eine sukzessive Anordnung von Makros schnell zu dichten Regionen im Plazierungsraum führt, die spätere Netze nicht mehr durchqueren können, was mitunter zu teuren Routen führt. Somit scheint Makro-Floorplanning eher vorteilhafter für Architekturen mit stärker ausgeprägten Verdrahtungsstrukturen, sowie Schaltkreise, die bereits aus größeren funk-

tionalen Einheiten, wie beispielsweise Addierer oder Komparatoren bestehen, und für welche bereits Implementierungen aus sogenannten Makro-Bibliotheken vorliegen.

Um jedoch unter den neun betrachteten Architekturen auch die hinsichtlich ihrer Verdrahtungsressourcen minimalistischer Geprägten fair bewerten zu können, wurde die Dilatation der Makros unbeschränkt gewählt, so daß die Schaltkreise vollständig mittels des *Route-then-Place*-Verfahren direkt implementiert wurden. Bei der Gegenüberstellung des gemittelten Kosten- und Delayoverheads der berechneten Implementierungen für die einzelnen Architekturen konnten denn auch einige im Vorfeld durchaus nicht zu erwartende Beobachtungen gemacht werden.

So markieren interessanterweise sogar zwei, hinsichtlich ihrer Verdrahtungsmöglichkeiten eigentlich minimalistisch konstruierte Doppelmaschen-Varianten die Optima unter den neun betrachteten Architekturen – eine davon hinsichtlich Ressourcenkosten, die andere hinsichtlich Performanz. Auch die bisher kaum beachteten 3D-Architekturen zeigten ein erstaunlich gutes Performanzverhalten, das mit den Resultaten der ebenfalls untersuchten *Double-Span*-Architekturen durchaus zu konkurrieren vermag. Das Beispiel einer anderen, ebenfalls um Verdrahtungsressourcen erweiterten Architektur, zeigte jedoch andererseits auch, wie ungünstig gewählte, da wenig nutzbare Strukturen zu Lasten der Qualität von Implementierungen gehen können.

Ausblick

Das in der vorliegenden Arbeit entwickelte, generische Konzept zur Modellierung und Bewertung feldprogrammierbarer Architekturen liefert über die oben dargestellten Untersuchungen hinaus noch zahlreiche Ansätze für weitergehende Betrachtungen. Nachfolgend sollen dazu einige Aspekte noch kurz angerissen werden.

Eine grundsätzliche Möglichkeit der Vorgehensweise stellt zunächst die Spezifikation von Grundformen neuer Architekturklassen dar, welche im Zuge ihrer Bewertung variiert und verfeinert werden können. Dies wurde beispielsweise bei der untersuchten Doppelmasche *DM2V001S* so angewandt, die entsprechend des verursachten, hohen Kostenoverheads, sowie in Anbetracht der Ressourcenbalance um Logik reduziert wurde und sodann als Architekturvariante *DM2V003S* hinsichtlich der Implementierungskosten sogar am besten unter den betrachteten Architekturen abschnitt. Weitere Fragestellungen könnten sich hier noch etwa mit der Anordnung der Logikzellen, mit einer weiteren Reduzierung oder auch einer möglichen Verbesserung des Delays durch Hinzufügen beispielsweise von *Double-Span*-Leitungen befassen. Ein anderes Beispiel stellen in diesem Zusammenhang auch die 3D-Architekturen dar, welche hinsichtlich des Delays ein gutes Verhalten zeigten. So wäre zu untersuchen, ob und inwieweit höhere Dimensionen diesbezüglich noch weitere Vorteile bringen.

Im Rahmen eines solchen „Architektur-Profilings“ könnten auch Untersuchungen über die Häufigkeit der Nutzung von Tasks in konfigurierbaren Zellen durchaus Hinweise zu einer geschickteren Wahl der Taskmengen liefern, was letztlich auch Optimierungen hinsichtlich der technischen Konstruktion der Zellen eröffnen würde.

Ein anderer Aspekt, der in dieser Arbeit nicht mehr angegangen werden konnte, betrifft die Betrachtung grobgranularerer Architekturen, also insbesondere Untersuchungen des Verhaltens von Architekturen mit größeren Look-Up-Tables, wobei die Frage in-

teressant scheint, inwieweit die in [65, 72] für Inselarchitekturen beobachteten Trends hinsichtlich der Auswirkung der Look-Up-Table-Größe auf Platzeffizienz und Performanz auch auf andere Architekturen übertragen werden können.

Der Fokus weiterer Betrachtungen kann schließlich auch auf die Suche gänzlich neuer Architekturen gelenkt werden. Hier gehen Ideen des Verfassers in Richtung einer Loslösung von der bisher betrachteten, streng orthogonalen Gitterstruktur. Architekturen mit Wabenstruktur oder triangulierten Strukturen, auch in höherdimensionalen Varianten bilden hier beispielsweise alternative Ansätze.

Abstrahiert man jedoch über dem generischen Konzept dieser Arbeit, indem man den Schritt von den signalbasierten Schaltkreisen beispielsweise zu Tasksystemen geht, so zeigt sich schließlich, daß die vorliegende Arbeit durchaus auch Bedeutung über die in ihr konkretisierte Modellumgebung hinaus besitzt. So kann eine andere Interpretation des vorgeschlagenen Konzeptes beispielsweise im Übergang von konfigurierbaren Zellen auf Prozessorelemente, Speicher und Verbindungszellen bestehen. Man betrachtet dann anstelle sequentieller Schaltkreise vielmehr Datenflußgraphen, deren Knoten Berechnungstasks und deren Kanten Datenabhängigkeiten repräsentieren mit dem Ziel, flexibel spezifizierbare Multiprozessor-Architekturen hinsichtlich der aus Einbettungen des Graphen resultierenden Kosten und Performanz bewerten zu können. Geeignete Datenstrukturen und Verfahrenskonzepte hierzu wurden im Rahmen dieser Arbeit bereitgestellt.

⋆ ⋆ ⋆

Literaturverzeichnis

[1] MCNC Benchmark Examples for the 1991 International Workshop on Logic Synthesis. http://www.cbl.ncsu.edu/pub/, 1991.

[2] Martin Aigner. *Diskrete Mathematik*. Vieweg, 1996.

[3] Michael J. Alexander et al. Performance-oriented placement and routing for field-programmable gate arrays. *Proceedings of the European Design Automation Conference*, pages 80–85, 1995.

[4] Michael J. Alexander et al. Three-dimensional field programmable gate arrays. In *Proc. IEEE International ASIC Conference*, pages 253–256, September 1995.

[5] C. J. Alpert and S. Z. Yao. Spectral partitioning: the more eigenvectors, the better. *Proceedings of the 32nd Design Automation Conference*, pages 195–200, 1995.

[6] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: A survey. UCLA Computer Science Department, Los Angeles, 1995.

[7] Altera Inc. *Apex II Programmable Logic Device Family Data Sheet*, 2002. http://www.altera.com.

[8] Wolfgang Backes. *The Structure of Longest Paths in Periodic Graphs*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1994.

[9] T. P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal of Computing*, (7):533–541, 1978.

[10] J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory I*. Springer, 1995.

[11] John F. Beetem. *Simultaneous Placement and Routing of the LABYRINTH Reconfigurable Logic Array*, chapter 4.8, pages 232–243. Abingdon EE&CS Books, Abingdon (UK), 1991.

[12] Vaughn Betz. *VPR and VPack User's Manual*. University of Toronto, February 1999.

[13] Vaughn Betz and Jonathan Rose. *VPR: A New Packing, Placement and Routing Tool for FPGA Research*, pages 213–222. Springer, 1997.

[14] Gaetano Boriello et al. The triptych FPGA architecture. *IEEE Transactions on VLSI Systems*, 3(4):491–501, December 1995.

[15] M. A. Breuer. Min-cut placement. *Design Automation and Fault Tolerant Computing*, 1(4):343–362, 1977.

[16] Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design & Test of Computers*, pages 42–57, 1996.

[17] Stephen D. Brown et al. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[18] Stephen D. Brown et al. A stochastic model to predict the routability of field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1827–1838, December 1993.

[19] Pak. K. Chan et al. On routability prediction for field-programmable gate arrays. *IEEE Design Automation Conference*, pages 326–330, 1993.

[20] Don Cherepacha and David Lewis. DP-FPGA: An FPGA architecture optimized for datapaths. *VLSI Design*, 4(4):329–343, 1996.

[21] Edith Cohen and Nimrod Megiddo. Recognizing properties of periodic graphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 4:135–146, 1991.

[22] André DeHon. Reconfigurable architectures for general-purpose computing. Technical Report 1586, Massachusetts Institute of Technology, October 1996.

[23] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[24] Carl Ebeling et al. *TRIPTYCH: a New FPGA Architecture*, chapter 3.1, pages 75–90. Abingdon EE&CS Books, Abingdon (UK), 1991.

[25] Abbas El Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Transactions on Circuits and Systems*, CAS-28(2):127–133, February 1981.

[26] Abbas El Gamal et al. An architecture for electrically configurable gate arrays. *IEEE Journal of Solid-State Circuits*, 24(2):394–398, April 1989.

[27] John M. Emmert and Dinesh Bhatia. A methodology for fast FPGA floorplanning. *Int'l Symposium for Field-Programmable Gate Arrays*, pages 47–56, 1999.

[28] C. M. Fiduccia and Mattheyses R. M. A linear time heuristic for improving network partitions. *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[29] D. Gajski and R. Kuhn. Guest editors' introduction: New VLSI tools. *IEEE Transactions on Computers*, 16(12):11–14, December 1983.

[30] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[31] Franz Höfting and Egon Wanke. Polynomial algorithms for minimum cost paths in periodic graphs. *Proceedings of SODA, SIAM, Philadelphia*, pages 493–499, 1993.

[32] Franz Höfting and Egon Wanke. Minimum cost paths in periodic graphs. *SIAM Journal on Computing*, 24(5):1051–1067, 1995.

[33] Franz Höfting and Egon Wanke. Polynomial-time analysis of toroidal periodic graphs. *Journal of Algorithms*, 34:14–39, 2000.

[34] Dwight Hill and Nam-Sung Woo. The benefits of flexibility in lookup table-based FPGA's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):349–353, February 1993.

[35] Uwe Hinsberger and Reiner Kolla. TEMPLATE: A generic technology mapping platform. Technical Report 186, Universität Würzburg, Institut für Informatik, 1997.

[36] William H. Kautz. Cellular logic-in-memory arrays. *IEEE Transactions on Computers*, C-18(8):719–727, August 1969.

[37] William H. Kautz, Karl N. Levitt, and Abraham Waksman. Cellular interconnection arrays. *IEEE Transactions on Computers*, C-17(5):443–451, May 1968.

[38] Zvi M. Kedem et al. Parallel suffix-prefix-matching algorithm and applications. *SIAM Journal on Computing*, 25(5):998–1023, October 1996.

[39] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[40] S. Kirkpatrick et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[41] D. E. Knuth et al. Fast pattern matching in strings. *SIAM Journal on Computing*, (6):323–350, 1977.

[42] Balázs Kotnyek. *A generalisation of totally unimodular and network matrices*. PhD thesis, London School of Economics, 2002.

[43] L. T. Kou et al. A fast algorithm for steiner trees. *Acta Informatica*, 15:141–145, 1981.

[44] Jack L. Kouloheris and Abbas El Gamal. FPGA performance versus cell granularity. *IEEE Custom Integrated Circuits Conference*, pages 6.2.1–6.2.4, 1991.

[45] M. R. Kramer and J. van Leeuwen. The complexity of wire routing and finding minimum area layouts for arbitrary vlsi circuits. *Advances in Computing Research*, pages 129–146, 1984.

[46] Helena Krupnova et al. A hierarchy-driven FPGA partitioning method. *Proceedings of the 34th Design Automation Conference*, pages 522–525, 1997.

[47] Helena Krupnova et al. Synthesis and floorplanning for large hierarchical FPGAs. *ACM/SIGDA Int'l Symposium on Field-Programmable Gate Arrays*, pages 105–111, 1997.

[48] Miriam Leeser et al. *Rothko: A Three Dimensional FPGA Architecture, Its Fabrication, and Design Tools*, pages 21–40. Springer, 1997.

[49] Guy G. Lemieux and Stephen D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. *Proceedings of the ACM Physical Design Workshop*, April 1993.

[50] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, B.G. Teubner, 1990.

[51] Andreas Lenz and Frank Wolz. Eine graphische Entwicklungsumgebung für Schaltkreise und feldprogrammierbare Architekturen. Studienarbeit, Lehrstuhl für Technische Informatik, Universität Würzburg, 2002.

[52] Huiqun Liu et al. Circuit partitioning with complex resource constraints in FPGAs. *Int'l Symposium on Field-Programmable Gate Arrays*, pages 77–84, 1998.

[53] Alexander Marquardt et al. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. *Int'l Symposium on Field-Programmable Gate Arrays*, pages 37–46, 1999.

[54] D. W. Matula and F. Shahrokhi. Graph partitioning by sparse cuts and maximum concurrent flow. *Technical Report, Southern Methodist University, Dallas, TX*, 1986.

[55] Larry McMurchie and Carl Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. *Int'l Symposium on Field-Programmable Gate Arrays*, pages 111–117, 1995.

[56] R. C. Minnick. Cutpoint cellular logic. *IEEE Transactions on Electronic Computers*, pages 685–698, December 1964.

[57] Paul Molitor and Christoph Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. B.G. Teubner Stuttgart-Leipzig, 1999.

[58] Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[59] Katta G. Murty. *Linear and Combinatorial Programming*. John Wiley & Sons, 1976.

[60] James B. Orlin. Some problems on dynamic/periodic graphs. *Progress in Combinatorial Optimization*, pages 273–292, 1984.

[61] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[62] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[63] N. R. Quinn and M. A. Breuer. A force directed component placement procedure for printed circuit boards. *IEEE Transactions on Circuits and Systems*, pages 377–388, June 1979.

[64] Jonathan Rose and Stephen Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE Journal of Solid-State Circuits*, 26(3):277–281, March 1991.

[65] Jonathan Rose et al. Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency. *IEEE Journal on Solid-State Circuits*, 25(5):1217–1225, October 1990.

[66] Jonathan Rose et al. The effect of switch box flexibility on routability of field-programmable gate arrays. *IEEE Custom Integrated Circuits Conference*, pages 27.5.1–27.5.4, 1990.

[67] Tsutomu Sasao, editor. *Logic Synthesis and Optimization*. Kluwer Academic Publishers, 1993.

[68] Ellen E. Sentovich et al. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California at Berkeley, Department of Electrical Engineering and Computer Science, 1992.

[69] P. D. Seymour. Decomposition of regular matroids. *Journal of Combinatorial Theory*, pages 305–359, 1980.

[70] Naveed A. Sherwani. *Algorithms for VLSI Physical Design*. Kluwer Academic Publishers, 1999.

[71] Jianzhong Shi and Dinesh Bhatia. Performance driven floorplanning for FPGA based designs. *Int'l Symposium for Field-Programmable Gate Arrays*, pages 112–118, 1997.

[72] Satwant Singh et al. The effect of logic block architecture on FPGA performance. *IEEE Journal of Solid-State Circuits*, 27(3):281–287, March 1992.

[73] Russel Tessier. Programmable cellular logic: Past, present, and future. http://www.ecs.umass.edu/ece/tessier/, September 1994.

[74] Russell Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, Massachusetts Institute of Technology, 1999.

[75] Nozomu Togawa et al. A simultaneous placement and global routing algorithm with path length constraints for transport-processing FPGAs. *Proceedings of the ASP-DAC*, 1997.

[76] Egon Wanke. Paths and cycles in finite periodic graphs. *Lecture Notes in Computer Science*, 711:751–759, 1993.

[77] Y.-C. A. Wei and C.-K. Cheng. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer-Aided Design*, 10(12):1502–1511, 1991.

[78] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design - A Systems Perspective*. Addison-Wesley, 1992.

[79] Frank Wolz. Entwicklung, Modellierung und Bewertung von Field-Programmable Gate Arrays. Master's thesis, Universität Würzburg, 1997.

[80] Frank Wolz and Reiner Kolla. Discrete floorplanning by multidimensional pattern matching. Technical Report 249, Universtität Würzburg, Institut für Informatik, 2000.

[81] Frank Wolz and Reiner Kolla. A new floorplanning method for FPGA architectural research. *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, pages 432–442, 2000.

[82] Frank Wolz and Reiner Kolla. Bubble partitioning for LUT-based sequential circuits. *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 336–345, 2001.

[83] Frank Wolz and Reiner Kolla. Zu Routingflexibilität und Optimierung konfigurierbarer Multiplexer-Netzwerke. Technical Report 289, Universität Würzburg, Institut für Informatik, 2001.

[84] Frank Wolz and Reiner Kolla. A retargetable macro generation method for the evaluation of repetitive configurable architectures. *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 997–1006, 2002.

[85] Y.-F. Wu, P. Widmayer, and C. K. Wong. A faster approximation algorithm for the Steiner problem in graphs. *Acta Informatica*, 23:223–229, 1986.

[86] Y. L. Wu and M. Marek-Sadowska. An efficient router for 2-d field-programmable gate arrays. *Proceedings of the European Design Automation Conference*, pages 412–416, 1994.

[87] Yu-Liang Wu et al. Graph based analysis of 2-d FPGA routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):33–44, January 1996.

[88] Xilinx Inc. *Virtex-E 1.8V Field-Programmable Gate Arrays, Preliminary Product Specification*, 2001. http://www.xilinx.com.

[89] Xilinx Inc. *Virtex-II Pro Advance Product Specification*, 2002. http://www.xilinx.com.

[90] Takayuki Yamanouchi et al. Hybrid floorplanning based on partial clustering and module restructuring. *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 478–483, 1996.

[91] Andy Yan et al. On the sensitivity of FPGA architectural conclusions to experimental assumptions, tools, and techniques. *Int'l Symposium on Field-Programmable Gate Arrays*, pages 147–156, 2002.

Notation

$\mathbb{B}$	Boolesche Menge $\{0,1\}$
$\mathbb{N}$	Menge der natürlichen Zahlen $1,2,\ldots$
$\mathbb{N}_0$	Menge der natürlichen Zahlen inklusive Null
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\#M$	Kardinalität der Menge M
$\complement M$	Komplement der Menge M
$u(b)$	Interpretation des Bitstrings b als vorzeichenlose Binärzahl
$\mathit{span}(x)$	$\{y \in \mathbb{N}^d \mid \forall_{i=0\ldots d-1}\ 0 \leq y_i < x_i\}$ für $x \in \mathbb{N}^d$
$\|\|x\|\|_1$	für $x \in \mathbb{R}^n$ die *Manhattan-Norm* (oder auch: *1-Norm*): $\sum_{i=0}^{n-1} \|x_i\|$
$(k)^x$	für $k \in \mathbb{R}$, $x \in \mathbb{N}^d$ die d-dim. Matrix mit x_i Einträgen k in der i-ten Dimension ($i \in \{1,\ldots,d\}$)
$\rho(e)$	Rand der Kante e, Menge der adjazenten Knoten
$\mathit{deg}^-(v)$	Ingrad des Knotens v
$\mathit{deg}^+(v)$	Ausgrad des Knotens v
E_v^-	Menge der in v einlaufenden Kanten
E_v^+	Menge der von v ausgehenden Kanten
E_v	Menge der v berührenden Kanten
$\vee$	logisches Oder
$\wedge$	logisches Und
$f : A \to B$	Abbildung einer Menge A in eine Menge B
$f : a \mapsto B$	Abbildung eines Elementes a auf ein Element aus der Menge B
$\dot\cup$	Disjunkte Vereinigung von Mengen
$y = (f \circ g)(x)$	$y = f(g(x))$ „f nach g“
$\exists^1$	es existiert *genau* ein Element