Diplomarbeit

Hardware-Partitionierung für Prototypen-Boards

Heiko Falk

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

9. August 1999

Betreuer:

Dr. Ralf Niemann
Prof. Dr. Peter Marwedel

Heiko Falk, Eppenhauser Straße 16, D-58093 Hagen
Matr.-Nr. 0045614, Fachbereich Informatik, Universität Dortmund

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 – Einführende Betrachtungen

> *„Ich halte dafür, daß das einzige Ziel der Wissenschaft darin besteht, die Mühseligkeiten der menschlichen Existenz zu erleichtern."*
>
> Aus: B. Brecht, „Leben des Galilei".

Aus dem heutigen alltäglichen Leben sind digitale Schaltungen jeglicher Art nicht mehr wegzudenken. Man ist von ihnen umgeben und nutzt die Elektronik, auch wenn häufig der Eindruck entsteht, ein bestimmtes Gerät würde nicht elektronisch gesteuert. Solche Schaltungen befinden sich zum Beispiel in Waschmaschinen, Backöfen und Staubsaugern, aber auch offensichtlicher in Telefonen oder Bordcomputern von Fahrzeugen.

Derartige digitale Schaltungen sind häufig sog. *eingebettete Systeme*, worunter hier ganz allgemein ein System verstanden sei, das Daten verarbeitet und dabei mit seiner Umwelt, in die es eingebettet ist, kommuniziert. Die Funktion eines eingebetteten Systems wird in vielen Fällen ganz oder teilweise durch spezielle Hardware-Bauteile, die nur für diesen Zweck konstruiert wurden, realisiert; diese Bauteile werden *ASIC* (Application Specific Integrated Circuit) genannt.

In den letzten Jahren ist im Bereich der ASIC-Entwicklung ein zunehmendes Interesse am Einsatz von *FPGA*s (Field Programmable Gate Arrays) zu beobachten. Dabei handelt es sich um Chips, die dem Benutzer eine gewisse Anzahl von logischen Elementen (Gatter, Flip-Flops) und Input/Output-Pins zur Verfügung stellen. FPGAs zeichnen sich einerseits dadurch aus, daß sie in *VLSI*-Technik (Very Large Scale Integration) hergestellt werden und somit einen hohen Grad an Komplexität erzielen. Auf der anderen Seite wird die Funktion derartiger Chips nicht vom Hersteller bestimmt; vielmehr wird die Programmierung vom Anwender vorgenommen.

Im Gegensatz zu einem full custom-Entwurf eines ASICs, bei dem ein komplett neuer Chip entworfen und produziert werden muß, bieten FPGAs die Vorteile, daß eben auf standardisierte Chips zurückgegriffen werden kann, die quasi „von der Stange" gekauft werden können. Soll ein ASIC nur in kleinen Stückzahlen hergestellt werden, ist eine Realisierung auf der Grundlage von

FPGA-Technologien – verglichen mit einem full custom-Entwurf – äußerst kostengünstig, da in diesem Fall die Entwurfszeiten kürzer sind.

Wegen ihrer Programmierbarkeit werden FPGAs zumeist zur schnellen Erstellung von *Prototypen* verwendet. Ein Prototyp ist dabei eine Realisierung eines ASICs in Form von Hardware, die zur Simulation und zur Überprüfung der Funktionsfähigkeit des ASICs in seiner künftigen Einsatzumgebung dient. Ein Prototyp unterliegt jedoch nicht den gleichen Zeitanforderungen wie der endgültig konstruierte ASIC. Zur Validierung der Funktionsfähigkeit eines ASICs genügt es, den Prototypen mit einer niedrigen Geschwindigkeit laufen zu lassen. In der letzten Zeit ist zusätzlich die Entwicklung abzusehen, daß FPGAs nicht nur zu Testzwecken herangezogen werden, sondern daß endgültige Realisierungen von ASICs ebenfalls auf FPGA-Technologien beruhen.

Beide Einsatzgebiete von FPGAs verlangen, daß sich der ASIC-Entwickler mit den Eigenschaften von FPGAs und den damit verbundenen Problemen auseinandersetzt, bzw. daß für auftretende Probleme im Zusammenhang mit FPGAs Lösungsansätze aufgezeigt werden.

Ein immer wieder auftretendes Problem beim Entwurf von Schaltungen mit FPGAs ist, daß die Kapazität heute verfügbarer FPGAs relativ gering ist. In diesem Zusammenhang ist unter Kapazität nicht nur die Anzahl zur Verfügung gestellter logischer Elemente zu verstehen, sondern auch die Anzahl der Input/Output-Pins. Für viele konkrete Anwendungen ergibt sich somit die Situation, daß ein Entwurf nicht mit Hilfe eines einzelnen FPGAs realisiert werden kann. Daher stellt sich die Frage, wie ein derartiger Entwurf in mehrere Teile zerlegt (partitioniert) werden kann, so daß jedes einzelne Teil durch ein FPGA realisiert wird und die Funktionsfähigkeit des gesamten Systems erhalten bleibt.

Das Problem der Partitionierung eines ASIC-Entwurfs in mehrere Teile – im folgenden *Logik-Partitionierung* genannt – wird im Rahmen dieser Diplomarbeit untersucht. Die Motivation zu diesem Thema ergab sich unmittelbar aus der Arbeit an einem eingebetteten System während der Projektgruppe 293 ([PG293a], [PG293b]). Hier wurde mittels zweier FPGAs und eines *DSPs* (Digital Signal Processor) ein Fuzzy-Controller zur Steuerung einer Ampelanlage realisiert. Während dieses Projektes ergaben sich die beiden folgenden Probleme:

- Es ist ohne weiteres nicht möglich, das entworfene System über Standard-Schnittstellen mit der Umwelt zu verbinden. Damit der Fuzzy-Controller mit der Umgebung kommunizieren konnte, wurde manuell eine simple Schnittstelle konstruiert, die einige Besonderheiten des verwendeten DSPs ausnutzte. In [Schä98] wird ein Ansatz entworfen, mit dem allgemeine Schnittstellen-Protokolle spezifiziert und durch die FPGAs implementiert werden können.

- Die Partitionierung der Teile des Fuzzy-Controllers, die durch die beiden FPGAs realisiert werden sollten, mußte aufgrund des Fehlens einer leistungsfähigen Partitionierungssoftware manuell durchgeführt werden.

Durch diese Vorgehensweise ergaben sich jedoch drastische Beschränkungen für die Größen der zu partitionierenden Komponenten.

In dieser Diplomarbeit wird ein neuer Ansatz zur Lösung des Partitionierungsproblems entworfen, der aus der Kombination eines bekannten Standard-Verfahrens mit einem genetischen Algorithmus besteht. Zur Präsentation dieses neuen Ansatzes ist diese Ausarbeitung in folgender Weise strukturiert:

Kapitel 2: In diesem Kapitel soll zunächst eine Vorstellung über den Ablauf der Hardware-Entwicklung vermittelt werden. Anschließend wird das Problem der Logik-Partitionierung genau definiert und eine Übersicht über bereits bekannte Partitionierungsmethoden gegeben.

Kapitel 3: Die Beschreibung des in dieser Arbeit vorgestellten Lösungsansatzes zur Partitionierung beginnt an dieser Stelle mit der Vorstellung eines Präprozessors, der in der zu partitionierenden ASIC-Spezifikation Hierarchien einfügt, die der nachfolgenden eigentlichen Partitionierung zu besseren Ergebnissen verhelfen.

Kapitel 4: Dieses Kapitel ist dem bereits erwähnten Standard-Verfahren zur Bipartitionierung (Partitionierung in zwei Teilmengen) gewidmet. Das Konzept dieses Algorithmus sowie besondere effiziente Datenstrukturen werden hier vorgestellt.

Kapitel 5: Der Schwerpunkt dieser Diplomarbeit liegt auf diesem Kapitel. Hier wird der genetische Algorithmus beschrieben, der die in den beiden vorherigen Kapiteln beschriebenen Konzepte vereint und bedeutend erweitert, wobei vorher eine kurze Einführung in die Funktionsweise genetischer Algorithmen gegeben wird.

Kapitel 6: Das in den Kapiteln 3 bis 5 vorgestellte Partitionierungsverfahren wurde in der Software COBRA (COOL Backend for Rapid Prototyping[1]) umgesetzt. Einige Details zur Arbeitsweise und Implementierung werden in diesem Kapitel beleuchtet. Weiter werden dem Leser vergleichende Ergebnisse zwischen COBRA und anderen Verfahren aus dem Bereich der Logik-Partitionierung präsentiert.

Kapitel 7: In diesem letzten Kapitel wird zunächst eine kurze Zusammenfassung über die erarbeiteten Resultate gegeben. Anschließend werden einige Fragen angesprochen, die sich aus der Arbeit mit COBRA ergeben haben und die die Praktikabilität des entwickelten Verfahrens sowie Möglichkeiten zur Verbesserung und Erweiterung betreffen.

Anhang A: Als Abschluß der Diplomarbeit werden an dieser Stelle einige Anmerkungen zur Konfigurierung und Benutzung der Partitionierungssoftware COBRA gemacht.

[1] COBRA wurde so konzipiert, daß es zum Abschluß des Co-Design-Tools COOL ([Niem98]) aufgerufen wird – daher der Name.

2 – Das Problem der Logik-Partitionierung

„O glaube mir, der manche tausend Jahre
An dieser harten Speise kaut,
Daß von der Wiege bis zur Bahre
Kein Mensch den alten Sauerteig verdaut!“

Aus: J.W. Goethe, „Faust, der Tragödie erster Teil“.

Dieses Kapitel führt in die Problematik der Logik-Partitionierung ein. Dazu wird zunächst eine Übersicht über die bei der ASIC-Entwicklung notwendigen Entwurfsschritte gegeben und gezeigt, an welcher Stelle dieses Designflusses die Logik-Partitionierung einzuordnen ist. Im Anschluß daran wird die Problemstellung formal genau definiert, bei der Partitionierung zu beachtende Beschränkungen und Optimierungskriterien werden eingeführt. Weiterhin muß aus technischen Gründen auf einige Besonderheiten von FPGAs des Herstellers Xilinx eingegangen werden. Den Abschluß dieses Kapitels bildet eine Einordnung der hier entworfenen Partitionierungsmethode in den Rahmen anderer bereits vorgestellter Verfahren.

2.1 Vom Algorithmus zum Chip: Hardware-Synthese

Der Vorgang des Hardware-Entwurfs ist im allgemeinen überaus komplex und besteht aus vielen verschiedenen Teilaufgaben, die nacheinander auszuführen sind. Um eine Vorstellung zu vermitteln, an welcher Stelle des Hardware-Entwurfs die Logik-Partitionierung üblicherweise stattfindet, sei an dieser Stelle zunächst ein Überblick über den gesamten Designfluß gegeben (siehe hierzu Abbildung 2.1).

- Ausgangspunkt ist stets die simulierte und für korrekt befundene Spezifikation eines ASICs, die in einer Hardware-Beschreibungssprache (z. B.

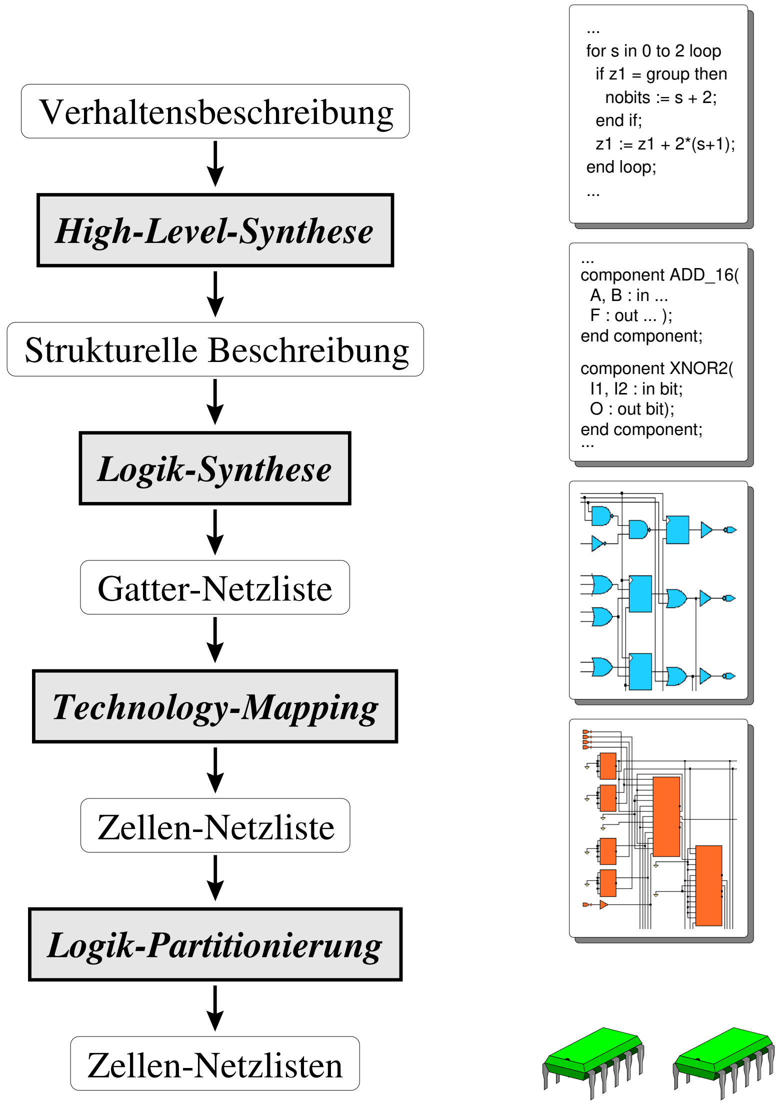


Abbildung 2.1: Designfluß des Hardware-Entwurfs mit COBRA

VHDL[1]) vorliegt. Diese Spezifikation enthält normalerweise sogenannte verhaltensbeschreibende algorithmische Elemente, wie zum Beispiel mathematische Operatoren, Schleifen, Kontrollstrukturen und Zuweisungen.

- Die erste Aufgabe, die im Hardware-Entwurf zu durchlaufen ist, ist die sogenannte *High-Level-Synthese*. Die vorliegende Beschreibung eines ASICs auf der Verhaltensebene zielt nur auf die Funktionalität des Chips ab. Damit letztlich eine Realisierung in Form von Hardware erreicht wird, müssen alle verhaltensbeschreibenden Elemente der ASIC-Spezifikation in eine Form überführt werden, die Aufschluß über die Struktur und den Aufbau des Chips gibt. Algorithmische Beschreibungen werden durch die High-Level-Synthese in einzelne Komponenten transformiert, z. B. in Ad-

[1] VHSIC Hardware Description Language

dierer, Register, Multiplexer, etc. Die Ausgabe dieses Synthese-Schrittes bildet dann eine strukturelle Beschreibung des ASICs (z. B. in strukturellem VHDL[2]), in dem obige Komponenten instantiiert und untereinander verkabelt sind. Die Software, die im Rahmen dieser Arbeit zum Zweck der High-Level-Synthese eingesetzt wird, ist das Tool OSCAR[3] ([LMD94]).

- Von der im vorherigen Schritt erreichten *Register-Transfer-Ebene* wird nun mittels der *Logik-Synthese* zur Logik-Ebene übergegangen. Die Funktionalität der instantiierten Komponenten wird durch logische Gatter und Flip-Flops ersetzt, und auf die so erzeugten booleschen Schaltwerke werden vielfältige Optimierungen angewendet, die die Logik reduzieren und redundante Teile entfernen. Das Resultat der Logik-Synthese besteht aus einer Beschreibung des ASICs, die ausschließlich logische Gatter, Flip-Flops und Input/Output-Pins enthält.

- Bis zu diesem Zeitpunkt ist für die Synthese keine Zieltechnologie festgelegt worden, d. h. daß die momentan vorliegende Gatter-Netzliste sowohl durch FPGAs als auch durch *Gate Arrays*, *PLD*s (Programmable Logic Device) oder beliebige andere Technologien realisiert werden kann. Dieser Zustand wird durch den Schritt des *Technology Mapping* beendet, in dessen Verlauf die Zellen, die von der Logik-Synthese instantiiert wurden, durch elementare Zellen ersetzt werden, die die spezifizierte Zieltechnologie zur Verfügung stellt. Das Produkt dieser Phase des Hardware-Entwurfs besteht aus einer Netzliste, die ausschließlich elementare Zellen der Zieltechnologie enthält. Somit stellt diese Netzliste eine Implementierung der ursprünglichen ASIC-Spezifikation durch eine ganz konkrete Technologie dar. Ein Programm-Paket, das Logik-Synthese und Technology Mapping in sich vereint, ist SYNOPSYS ([Syno92]).

- Allerdings wird während des Technology Mappings die Größe eines Chips der Zieltechnologie nicht in Betracht gezogen. Aus diesem Grund kann der Fall eintreten, daß die resultierende Netzliste für einen einzelnen Chip zu umfangreich ist. An dieser Stelle des Designflusses soll nun die *Logik-Partitionierung* einsetzen und die technologiespezifische Netzliste zerlegen. Ergebnis der Partitionierung sind mehrere wiederum technologieabhängige Netzlisten, wobei jede Netzliste jeweils einen Chip beschreibt. Zusätzlich ist eine Vorschrift zu erzeugen, nach der diese einzelnen Netzlisten wieder untereinander zu verdrahten sind, so daß das gesamte System funktionsfähig bleibt.

- Auf der Basis dieser verschiedenen Netzlisten sind dann in einem letzten Schritt die einzelnen Chips zu programmieren bzw. herzustellen, so daß schließlich der spezifizierte ASIC als Realisierung in Form von Hardware vorliegt.

[2] Eine Einführung in verhaltensorientierte und strukturelle Konzepte von VHDL stellt [BaMa94] dar.

[3] Optimum Simultaneous Scheduling, Allocation and Resource Binding

2.2 Eine formale Beschreibung der Logik-Partitionierung

Nachdem gezeigt wurde, welche Rolle die Logik-Partitionierung beim Entwurf digitaler Schaltungen spielt, wird nun eine formalere Darstellung des Partitionierungsproblems gegeben.

Zu diesem Zweck ist es notwendig, den Begriff des *Hypergraphen* einzuführen. Ein Hypergraph unterscheidet sich von normalen Graphen, die aus einer Knotenmenge und Kanten zwischen je zwei Knoten bestehen, dadurch, daß beliebig viele Knoten des Hypergraphen durch *Hyperkanten* miteinander verbunden werden können.

Definition 2.1 (Hypergraph)

Ein Hypergraph $H = (V, E)$ *besteht aus einer Menge von Knoten* $V = \{v_1, \ldots, v_s\}$ *und einer Menge von Hyperkanten* $E = \{e_1, \ldots, e_t\}$.

Für einen Knoten v_i *bezeichne* p_i *die Anzahl inzidenter Kanten*[4].

Eine Hyperkante $e_k = \{v_{k_1}, \ldots, v_{k_m}\}$ *wird durch die Menge aller Knoten aus* V *dargestellt, mit denen* e_k *inzidiert.*

Um einen deutlicheren Bezug zum Hardware-Entwurf herzustellen, wird im weiteren Verlauf des Textes von *Designs* anstatt von Hypergraphen gesprochen. Ein Design besteht aus einer Menge von *Zellen* ($\hat{=}$ Knoten) und *Netzen* ($\hat{=}$ Hyperkanten).

Zur Durchführung der Logik-Partitionierung stehe eine Menge $\{C_1, \ldots, C_u\}$ von Chips bereit. Jeder Chip C_l stellt dabei eine gewisse Quantität an Chip-Fläche A_l und an Pins P_l zur Verfügung. Um die Größe eines Designs zu messen, sei jeder Zelle v_i eine Fläche a_i zugeordnet.[5]

Entscheidend für die Partitionierung sind offensichtlich die Ressourcen, die von einem Design in Anspruch genommen werden, bzw. die von den einzelnen Bausteinen zur Verfügung gestellt werden:

- Das erste Optimierungsziel, das bei der Partitionierung zu berücksichtigen ist, besteht darin, die Zellen des Designs in disjunkte Mengen $V_1, \ldots, V_u$ aufzuteilen, so daß jede Zelle in genau einer Menge V_l enthalten ist. Dabei muß für alle Mengen V_l gelten, daß die Summe der Flächen der Zellen aus V_l nicht größer ist als die Fläche des FPGAs C_l:

$$\forall l \in \{1, \ldots, u\}: \quad \sum_{v_i \in V_l} a_i \leq A_l$$

- Da bis hierhin die Pins eines Chips nicht berücksichtigt werden, gilt für die Partitionierung ein zweites Optimierungskriterium. Hierbei geht es um sogenannte *geschnittene Netze*. Ein Netz e_k heißt geschnitten, wenn die

[4] Eine Kante e_k inzidiert mit einem Knoten v_i, wenn Knoten v_i mit Kante e_k verbunden ist.

[5] Im folgenden werden die Begriffe „Fläche“, „Größe“ und „Kosten“ für Zellen synonym zueinander verwendet.

Knoten, mit denen es inzidiert, in mindestens zwei verschiedenen Mengen V_{k_1} und V_{k_2} liegen. Die Menge $CS \subseteq E$ aller geschnittenen Netze wird auch *Cutset* genannt; die Größe $|CS|$ des Cutsets heißt *Cutsize*.

Ein geschnittenes Netz e_k, das Knoten aus $V_{k_1}, \ldots, V_{k_m}$ miteinander verbindet, muß in einer späteren Hardware-Realisierung auch die Chips $C_{k_1}, \ldots, C_{k_m}$ miteinander verbinden. Für diese Verbindung wird bei jedem beteiligten Chip ein Pin benötigt. Insgesamt ist also darauf zu achten, daß die Anzahl der geschnittenen Netze, die mit Knoten einer Menge V_l inzidieren, nicht die Anzahl von Pins des Chips C_l überschreitet:

$$\forall l \in \{1, \ldots, u\}: \quad \big|\big\{e_k \in CS \mid (e_k \text{ inzidiert mit } v_i) \wedge (v_i \in V_l)\big\}\big| \leq P_l$$

Formuliert man das Logik-Partitionierungsproblem als *Entscheidungsproblem*, so ist eine beliebige Partition als Lösung zu bestimmen, die beide vorstehende Bedingungen erfüllt. Eine derartige Lösung heißt *gültig*.

Ziel der Logik-Partitionierung als *Optimierungsproblem* ist es, unter allen gültigen Lösungen diejenige zu ermitteln, die eine gegebene Zielfunktion optimiert. In den meisten Fällen wird die Minimierung des Cutsize-Maßes gefordert, Beispiele für weitere Optimierungsziele werden in Abschnitt 2.4.2 gegeben.

In [GaJo79] findet sich ein Verweis auf einen Artikel, in dem gezeigt wurde, daß dieses Partitionierungsproblem NP-vollständig ist. Als Konsequenz ist davon auszugehen, daß es für dieses Problem keine Lösungsmethode gibt, die polynomielles Laufzeitverhalten aufweist. Aus diesem Grund kommen in diesem Bereich lediglich Näherungsverfahren zum Einsatz, die in akzeptabler Laufzeit gute, falls möglich optimale, Ergebnisse liefern. Verschiedene Verfahren zur Logik-Partitionierung können dann verglichen werden, indem sie auf eine Menge von *Benchmark-Designs* angewendet und die einzelnen ermittelten Cutsizes in Relation zueinander gesetzt werden. Die Ergebnisse einer derartigen Testserie sind in Abschnitt 6.4.2 zu finden.

2.3 XILINX-FPGAs als Zieltechnologie

Bevor der im Rahmen dieser Arbeit entwickelte Partitionierungsansatz anderen Konzepten gegenübergestellt werden kann, muß kurz die Struktur von XILINX-FPGAs abgehandelt werden.[6] Die spezifische Struktur dieser FPGAs wirkt sich in besonderem Maße auf das Technology Mapping aus, dessen Ergebnisse zur Partitionierung herangezogen werden.

Die grundlegenden Elemente, aus denen sich XILINX-FPGAs zusammensetzen, heißen *CLB* (<u>C</u>onfigurable <u>L</u>ogic <u>B</u>lock) und *IOB* (<u>I</u>nput/<u>O</u>utput <u>B</u>lock). Ein FPGA setzt sich aus vielen CLBs zusammen, die auf dem Chip in Form einer quadratischen Matrix angeordnet sind. An den Rändern dieser Matrix sitzen mehr oder weniger viele IOBs, deren einzige Funktion darin besteht, Signale

[6] Vergleiche hierzu [Xili91] – hier wird lediglich auf die Familie XC 4000 eingegangen.

aus dem FPGA heraus bzw. in das FPGA hinein zu leiten. Um bei der Logik-Partitionierung CLBs und IOBs korrekt zu behandeln, muß für jede Zelle eines zu partitionierenden Designs deren Typ festgehalten werden.

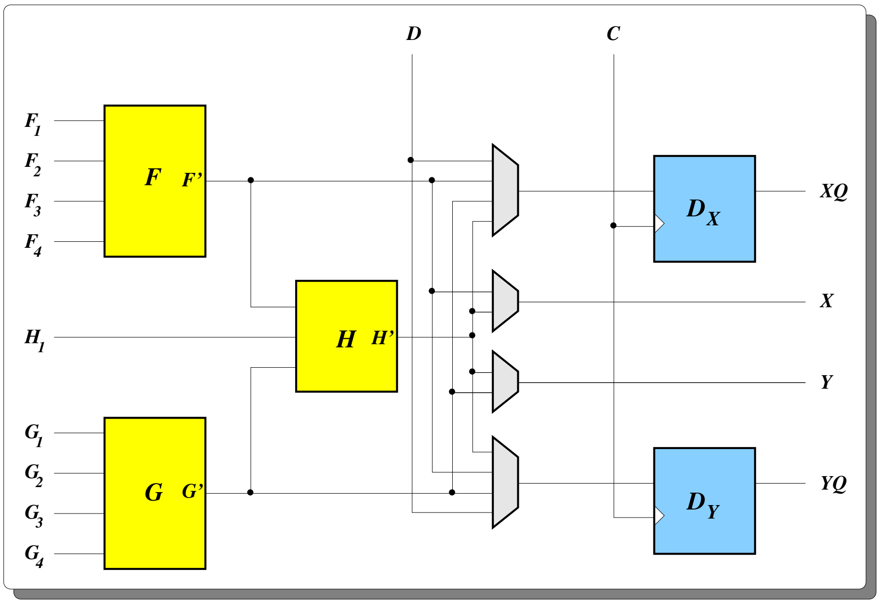


Abbildung 2.2: Struktur eines Xilinx-Configurable Logic Blocks

Die innere Struktur eines CLBs wird in Abbildung 2.2 dargestellt. Die Besonderheit eines CLBs besteht darin, daß dem Benutzer keine Gatter zur Verfügung gestellt werden, sondern sog. *Funktionsgeneratoren*[7], die beliebige boolesche Funktionen über ihre fixe Anzahl von Eingängen realisieren können. In der Abbildung sind F, G und H die Funktionsgeneratoren eines CLBs. Dabei haben F und G jeweils 4 externe Eingänge, während mittels H die Signale F' und G', die von F und G berechnet werden, miteinander und mit einem weiteren externen Signal verknüpft werden können.

Durch diese Struktur der Funktionsgeneratoren ergibt sich, daß mit Hilfe eines CLBs

- zwei beliebige vierstellige boolesche Funktionen,
- eine beliebige vierstellige und einige spezielle fünfstellige Funktionen,
- eine beliebige fünfstellige Funktion oder
- einige spezielle bis zu neunstellige Funktionen

realisiert werden können.

Weiterhin enthält jeder CLB zwei D-Flip-Flops D_X und D_Y, die die Ergebnisse von F, G bzw. H oder das externe Signal D speichern können. Die Signale F', G' oder H' können über die Pins X und Y aus dem CLB herausgeleitet werden.

[7] Auch *Lookup-Table* (LUT) genannt.

Die vordringliche Aufgabe eines XILINX *Technology Mappers* – i. e. eines Programmes zur Durchführung des Technology Mapping – besteht also darin, die Gatter der Netzliste, die aus der Logik-Synthese hervorgeht, so zu gruppieren, daß hinterher jede derartige Gruppe von Gattern durch einen einzelnen Funktionsgenerator realisiert wird. Während dieses Vorgangs wird darauf hingearbeitet, die CLBs möglichst gut zu füllen, so daß letztendlich eine möglichst geringe Anzahl von CLBs benötigt wird.

2.4 Strategien zur Logik-Partitionierung

Wurde in Abbildung 2.1 der Anschein erweckt, die Logik-Partitionierung fände stets erst nach dem Technology Mapping statt, so entspricht diese Situation nicht allen Partitionierungsansätzen.

Sehr viele Forscher vertreten den Standpunkt, daß die Partitionierung besser vor dem Technology Mapping stattfinden sollte. Die Vor- und Nachteile beider Strategien werden an dieser Stelle kurz herausgestellt.

2.4.1 Technology Mapping – Vorher oder hinterher?

Der herausragende Unterschied zwischen einer Partitionierung vor oder nach dem Technology Mapping besteht in der Bestimmung der *Kostenmaße* für die Zellen des Designs. Als Kostenmaß sei hier die Fläche a_i einer Zelle v_i verstanden.

Partitioniert man nach dem Technology Mapping, so repräsentiert jede Zelle einen CLB. Dem Partitionierer ist bekannt, wieviele CLBs ein FPGA enthält, und auf diese Weise läßt sich exakt berechnen, wie voll jedes FPGA für eine gegebene Partition ist.

Bei einer Partitionierung vor dem Technology Mapping läßt sich dagegen die belegte Chip-Fläche eines FPGAs nicht exakt berechnen. Der Grund hierfür ist in den Funktionsgeneratoren der FPGAs zu suchen. Ohne ein Technology Mapping durchzuführen läßt sich nicht bestimmen, wieviele Gatter durch einen Funktionsgenerator realisiert werden. Es kann sein, daß ein einziges Gatter in einem Funktionsgenerator gruppiert wird; es kann aber auch sein, daß fünf oder mehr Gatter zu einer solchen Einheit zusammengefaßt werden. Unter derartigen Umständen ist es schwierig, jedem Gatter eines Designs seine spezifische Fläche zuzuordnen.

Man versucht, dieses Problem mit unterschiedlichen Ansätzen zu umgehen:

- Jedes Gatter hat Kosten 1.
- Ein Gatter, dessen Anzahl von Eingabe-Signalen (*Fan-In*) p ist, erhält Kosten $p - 1$.

- Eine durchschnittliche Anzahl von Gattern, die während des Technology Mappings auf einen Funktionsgenerator abgebildet werden, kann zur Kosten-Bestimmung herangezogen werden.

Alle diese Methoden haben jedoch gemein, daß es sich um Schätzungen handelt. Die auf solchen Grundlagen berechnete Auslastung eines FPGAs ist daher immer mit einer Ungenauigkeit behaftet.

Das Technology Mapping wirkt sich auch auf die Laufzeiten eines Partitionierungsalgorithmus vorteilhaft aus, da ein Design nach dem Technology Mapping wesentlich weniger Zellen enthält als vorher (erfahrungsgemäß ergibt sich eine Reduktion auf ungefähr $\frac{1}{13}$ der ursprünglichen Größe).

Eine Partitionierung auf CLB-Ebene hat jedoch den Effekt, daß die Cutsizes derartiger Partitionen in der Regel größer sind als bei einer Partitionierung auf Gatter-Ebene. Die Ursache hierfür liegt in der Tatsache begründet, daß ein CLB insgesamt über 17 Anschlüsse verfügt (einige Pins wurden in Abbildung 2.2 nicht dargestellt), während Gatter nur selten mehr als sechs Pins haben. Die Entscheidung, einen CLB auf ein bestimmtes FPGA abzubilden, wirkt sich somit wesentlich stärker auf die Größe des Cutsets aus als bei Gattern.

Zudem tritt häufig die Situation ein, daß in den Funktionsgeneratoren F und G eines CLBs Funktionen berechnet werden, die für absolut verschiedene Zwecke in dem Design verwendet werden, so z. B. für einen Bus-Treiber und für ein Rechenwerk. Dies wird durch den Technology Mapper herbeigeführt, um den CLB möglichst gut auszulasten. Wenn nun aber die restlichen Logik-Teile, zu denen F und G gehören (sprich: restlicher Bus-Treiber und restliches Rechenwerk), auf zwei verschiedene FPGAs partitioniert wurden, so muß stets mindestens eine Verbindung zwischen den restlichen Logik-Teilen und diesem CLB über das Cutset geführt werden.[8]

Cobra wurde mit der Intention entworfen, eine Partitionierung mit exakten Kostenmaßen auf CLB-Ebene durchzuführen. Um jedoch eine Reihe von Benchmark-Designs zu partitionieren, war es notwendig, Funktionen bereitzustellen, die eine Partitionierung auch auf Gatter-Ebene ermöglichen. Somit ist Cobra also in der Lage, sowohl mit automatisch bestimmten Kostenmaßen auf CLB-Ebene als auch mit benutzerdefinierten Kostenmaßen auf Gatter-Ebene zu partitionieren.

2.4.2 Eine Übersicht über andere Verfahren

Die Entscheidungen, auf welcher Ebene innerhalb des Designflusses partitioniert wird und welche Art von Kostenmaß durch einen Partitionierungsalgorithmus verwendet wird, sind jedoch nicht die einzigen Kriterien, nach denen sich verschiedene Verfahren klassifizieren lassen. Von weiterem Interesse sind u. a.:

[8] Einige experimentelle Ergebnisse über die Auswirkungen des Technology Mappings auf das Cutsize sind in Abschnitt 6.4.1 zu finden.

Optimierungsziel: Die Zielsetzung, die im Verlauf der Partitionierung verfolgt werden soll, kann von Verfahren zu Verfahren variieren, so daß eine Methode für bestimmte Zwecke besser geeignet sein kann als andere.

Algorithmus: Der zugrunde liegende Partitionierungsalgorithmus bestimmt unmittelbar die Güte der Ergebnisse und die Laufzeiten zur Berechnung.

Zieltechnologie: Einige Verfahren sind speziell auf die Architektur von FPGAs bestimmter Hersteller ausgerichtet.

FPGA-Anzahl: Da nicht alle Algorithmen Partitionierungen für eine beliebige Anzahl von FPGAs unterstützen, muß für praktische Anwendungen der gewählte Partitionierungsalgorithmus gegebenenfalls mehrfach iteriert angewendet werden.

Board-Topologie: Häufig sind FPGAs auf Platinen angebracht, deren *Topologie* – also die physikalische Anordnung der Chips mitsamt ihren Verbindungen untereinander – fest vorgegeben ist. Partitionierungsverfahren, die diese Anordnung nicht berücksichtigen, können Zerlegungen mit kleinen Cutsets liefern, die jedoch auf einem bestimmten FPGA-Board nicht realisierbar sein können.

Bezüglich dieser Merkmale werden nun einige Arbeiten aus dem Gebiet der Logik-Partitionierung mit dem hier vorgestellten Ansatz verglichen.

Den Anfang bildet die Methode von **Weinmann** (Karlsruhe) und **Rosenstiel** (Tübingen), die in [WeRo94] vorgestellt wurde. Dieser Ansatz ist so flexibel, daß für beliebig viele FPGAs – gleich welchen Herstellers – partitioniert werden kann.

Die Eingabe bildet eine Gatter-Netzliste, die noch nicht durch einen Technology Mapper bearbeitet wurde. Die Problematik, daß für derartige Netzlisten die Kosten nicht exakt bestimmt werden können, wird dadurch gelöst, daß der Algorithmus nach jeder Iteration, in der eine neue Partition berechnet wurde, einen Technology Mapper für eine bestimmte Zieltechnologie aufruft, der dann die exakten Kosten (z. B. in CLBs) für die aktuelle Partition liefert.

Der Algorithmus berechnet für jede Zelle eine Wahrscheinlichkeit, mit der diese in eine bestimmte Partition bewegt wird. Die Zelle mit der höchsten Wahrscheinlichkeit wird letztendlich bewegt. Dieser Vorgang wird zwischen zehn- und hundertmal wiederholt.

In die Berechnung dieser Wahrscheinlichkeit können beliebig viele benutzerdefinierte Optimierungsziele einfließen. Die gebräuchlichsten sind dabei die Anzahl der Pins einer Zelle sowie deren Größe. Um auf besondere Gegebenheiten einer Zieltechnologie einzugehen, können jedoch noch weitere charakteristische Eigenschaften einer Zelle hinzugezogen werden.

Brasen, **Hiol**, **Saucier** und **Belhadj** (Grenoble) haben in [BHSB93] einen grundsätzlich anderen Ansatz gewählt. Hier wird eine zweistufige Methode vorgeschlagen: Zunächst soll auf einer hohen Ebene, in der ein ASIC noch in Form von Datenpfaden, Controllern, RAM/ROM etc. beschrieben ist, partitioniert werden. Setzt sich eine Zelle auf dieser Ebene hierarchisch aus mehreren weiteren Zellen zusammen, so werden hiernach die Zellen dieser Sub-Hierarchie rekursiv partitioniert. Danach sind die Zellen eines ASICs, die keine Sub-Hierarchien mehr besitzen und für ein einzelnes FPGA noch zu groß sind, auf Gatter-Ebene zu zerlegen.

Die durch diese Art von Partitionierung ermittelten Zerlegungen von hierarchischen Zellen haben die Eigenschaft, daß sie sowohl bezüglich der Fläche als auch bezüglich ihrer Pins auf jeweils ein FPGA passen. Diese Partitionen werden nun wieder paarweise miteinander verschmolzen, so daß größere Cluster gebildet werden, die weiterhin auch auf jeweils ein FPGA passen, aber dabei das Verhältnis zwischen beanspruchter Fläche und benötigten Pins maximieren.

Für die vorgesehene Partitionierung auf Gatter-Ebene wird ein Verfahren vorgestellt, das auf sogenannten *„Cones“* basiert. Ein Cone ist die Menge von Gattern, die zwischen einem Ausgabe-Pin und denjenigen Eingabe-Pins liegt, die zu der betrachteten Ausgabe führen. In der Gatter-Netzliste werden dann Cones bestimmt, die nicht notwendigerweise disjunkt sind. Überlappen sich zwei Cones, so werden diese entsprechend ihrer Schnittmenge aufgeteilt, so daß letztlich eine disjunkte Menge von Clustern entsteht, die dann partitioniert wird.

Aus dem Artikel geht nicht hervor, wie die Kosten für einzelne Zellen aus der Hierarchie eines ASICs bestimmt werden. Es ist davon auszugehen, daß hierzu während der Partitionierung kontinuierlich ein Technology Mapper eingesetzt wird. Da auf dieser hohen Beschreibungsebene noch keine Rücksicht auf eine konkrete Zieltechnologie genommen wird, ist dieser Partitionierungsansatz für alle FPGA-Architekturen anwendbar. Die Unterstützung von Board-Topologien ist nicht vorgesehen.

Hauck (Evanston) und **Borriello** (Seattle) stellen in [HaBo95] eine Methode vor, mit der Gatter-Netzlisten äußerst effizient in zwei gleichgroße Partitionen mit möglichst kleinem Cutset zerlegt werden können.

Die Grundlage für dieses Verfahren bildet der *Algorithmus von Fiduccia und Mattheyses* (FM), der erstmalig in [FiMa82] präsentiert wurde. Dieser Algorithmus verfolgt die Zielsetzung, einen Hypergraphen für einen gegebenen Parameter r zu bipartitionieren, so daß jede Partition mindestens r Prozent der Zellen enthält. Die Laufzeit hat lineare Komplexität $O(\sum p_i)$, gemessen an der Zahl p_i der angeschlossenen Netze aller Zellen v_i eines Hypergraphen.

Die beiden Autoren nutzen dabei eine Technik, die bei der Logik-Partitionierung zunehmend ins Blickfeld rückt: In dem Design, das zu partitionieren ist, werden Zellen miteinander verschmolzen (geclustert), so daß hinterher ein neues Design entsteht, das insgesamt weniger Zellen und Netze enthält als das ursprüngliche. Dieser Vorgang wird auf dem gerade kleinsten Design fortgeführt, bis ein Abbruchkriterium erfüllt ist. Auf diese Weise erhält man eine ganze Folge von Designs, die in ihrer Größe monoton abnehmen.

Auf das kleinste Design dieser Folge wird nun der Algorithmus von Fiduccia und Mattheyses angewendet mit der Hoffnung, daß eine gute Bipartition des kleinen Designs nicht wesentlich schlechter ist als eine Bipartition des originalen Designs. Die so gewonnene Bipartition wird auf das nächstgrößere Design übertragen und darin wiederum mittels Fiduccia & Mattheyses verfeinert. Dieser Vorgang wird iteriert, bis zuletzt das originale Design partitioniert ist. Ziel ist, das Cutsize-Maß der Bisektion zu minimieren.

Das Verfahren von Hauck und Borriello birgt allerdings einige gravierende Nachteile in sich:

- Es wird eine Partitionierung auf Gatter-Ebene für XILINX-FPGAs vorgenommen, so daß das Problem auftaucht, welche Kosten ein einzelnes Gatter verursacht. Hauck und Borriello vertreten die Meinung, daß ein Gatter, das einen Fan-In von p hat, mit Kosten $p - 1$ gewichtet werden sollte.
- Die Anzahl von FPGAs, für die partitioniert werden kann, ist auf zwei beschränkt.

  Um derartige Methoden zur Bipartitionierung auf beliebig viele FPGAs zu erweitern, ist eine rekursive Anwendung des Bipartitionierers notwendig. Meistens geschieht dies baumförmig, indem zunächst das komplette Design bipartitioniert wird, dann beide vorher erzeugten Bipartitionen, etc. Wenn allerdings bei einer anfänglichen Partitionierung ein schlechtes Ergebnis berechnet wurde, so haben die nachfolgenden Bipartitionierungen keine Chance, Korrekturen daran vorzunehmen, da ein Backtracking zwischen den einzelnen Aufrufen des Bipartitionierers nicht möglich ist.
- Topologien von FPGA-Boards werden nicht berücksichtigt.

Dutt und **Deng** (Minneapolis) versuchen in ihrem Ansatz ([DuDe96]), die Vorgehensweise des FM-Algorithmus intelligenter und weitsichtiger zu gestalten. Ursprünglich wird für jede Zelle eines Designs ein *„Gewinn“* berechnet. Die Berechnung derartiger Gewinne basiert allerdings auf sehr lokalen Informationen, die lediglich die Zelle und die daran angeschlossenen Netze betreffen. Diejenige Zelle, die den höchsten Gewinn aufweist, wird dann von einer Partition in die andere bewegt.

Dutt und Deng schlagen zur Berechnung von Zellen-Gewinnen einen probabilistischen Ansatz vor. Für jede Zelle wird eine Wahrscheinlichkeit berechnet, mit der die Zelle in die jeweils andere Partition bewegt wird. Zu Beginn des Verfahrens besitzen alle Zellen die gleiche Wahrscheinlichkeit. Auf diesen Bewegungswahrscheinlichkeiten aufbauend wird eine probabilistische Gewinn-Größe kalkuliert.

Bei der Berechnung dieser Gewinne wird vorausschauend vorgegangen, indem nicht nur überprüft wird, wie sich das Cutset bei Bewegung einer einzelnen Zelle ändert. Vielmehr wird in Betracht gezogen, wie nach einer potentiellen Zellen-Bewegung andere benachbarte Zellen bewegt werden müßten, um das Cutsize-

Maß weiter zu verkleinern. Diese „Weitsicht“ soll dafür sorgen, daß Cluster von Zellen, also Mengen von Zellen, die sehr stark untereinander vernetzt sind, nicht voneinander getrennt werden, wie dies beim Standard-FM der Fall sein kann.

Die Kritikpunkte, die gegen dieses Verfahren vorgebracht werden können, gleichen denen der vorigen Methode, da es konzeptionell keine großen Unterschiede zwischen beiden gibt. Auf die Frage des verwendeten Kostenmaßes für Zellen wird in dem Artikel nicht näher eingegangen; vermutlich werden sämtliche Zellen mit den fixen Kosten 1 gewichtet, was unter der Voraussetzung, daß die Zellen-Kosten nicht zu stark variieren, ein durchaus übliches Vorgehen ist.

Das hier vorgestellte Verfahren **Cobra** besteht aus einer Kombination von Methoden, die hier bereits vorgestellt wurden, mit einem genetischen Algorithmus. Cobra ist in der Lage, für beliebige Anzahlen von Xilinx-FPGAs unter Beachtung vorgegebener Topologien zu partitionieren.

Dabei wird als Grundlage die CLB-Ebene nach dem Technology Mapping herangezogen, so daß auch als Kostenmaß exakt berechnete CLBs verwendet werden können. Wie bereits erwähnt, kann durchaus auch auf Gatter-Ebene partitioniert werden. In diesem Fall muß der Hardware-Entwickler allerdings die Größen der einzelnen Gatter-Typen und der FPGAs in einer Bibliothek vorgeben. Die Zielsetzung, die während der Partitionierung verfolgt wird, besteht darin, das Cutsize-Maß zu minimieren und dabei die verschiedenen FPGAs möglichst gleichmäßig auszulasten.

Dieses Verfahren führt – angelehnt an [HaBo95] – ebenfalls ein iteratives Clustern des Designs durch. Die eigentliche Partitionierung des Designs wird durch den genetischen Algorithmus vorgenommen, der in Kapitel 5 vorgestellt wird. Ein Algorithmus zur Bipartitionierung basierend auf dem Verfahren von Fiduccia & Mattheyses (siehe hierzu auch Kapitel 4) wird verwendet, um für die Genetik eine Anzahl relativ guter Startlösungen zu erzeugen.

In der nachfolgenden Tabelle 2.1 sind die Merkmale der einzelnen Partitionierungsansätze kurz zusammengefaßt.

Verfahren	Ebene	Kosten	Optimierungsziel	Algorithmus	FPGA-Technologie	#FPGAs	Topologien
[WeRo94]	Gatter	CLB ($\rightsquigarrow$ TechMap)	Fläche + Pins	Moving Probability	Beliebig	Beliebig	Nein
[BHSB93]	HighLevel + Gatter	CLB ($\rightsquigarrow$ TechMap)	$\frac{\#\text{CLBs}}{\#\text{Pins}} \rightarrow$ max.	Hierarchisch + Cones	Beliebig	Beliebig	Nein
[HaBo95]	Gatter	Fan-In $-$ 1	Cutsize $\rightarrow$ min.	Clusterung + FM	Xilinx	2	Nein
[DuDe96]	Gatter	1	Cutsize $\rightarrow$ min.	FM mit Prob. Gewinnen	Beliebig	2	Nein
Cobra	CLB od. Gatter	CLB od. Konstante	Cutsize $\rightarrow$ min. + Balance	Clusterung, FM, Genetik	Xilinx	Beliebig	Ja

Tabelle 2.1: Übersicht über Partitionierungsstrategien

3 – Bilden von Hierarchien mittels Iterativen Clusterns

„Es wird zusammenwachsen, was zusammengehört.“

Willy Brandt

Im Rahmen der High-Level-Synthese werden sämtliche vom Entwickler spezifizierten verhaltensbeschreibenden VHDL-Einheiten durch eine Vielzahl kleinerer Komponenten ersetzt, die alle untereinander verbunden sind. Diese kleineren Komponenten können sich in struktureller Weise wiederum aus vielen weiteren Komponenten zusammensetzen etc., so daß auf diese Weise ein neues Design entsteht, in dem jede ursprüngliche Einheit aus ganzen *Hierarchien* kleinerer Komponenten besteht.

Alle Zellen einer einzelnen derartigen Hierarchie-Ebene haben die Eigenschaft, daß sie einerseits nur über eine relativ geringe Anzahl von Netzen mit der nächsthöheren Hierarchie-Ebene verbunden sind, daß aber andererseits intern eine sehr hohe Vernetzungsdichte erreicht wird.

Ein grundsätzliches Problem bei der Logik-Partitionierung auf Gatter- oder CLB-Ebene besteht darin, daß sämtliche Hierarchien im Verlauf der Logik-Synthese aufgelöst werden (*Flattening*). Auf dieser niedrigen Ebene ist es nicht mehr möglich, einzelne Zellen des Designs zweifelsfrei etwa einem Rechenwerk zuzuordnen, da es zwischen den Zellen keinen Unterschied gibt und alle über gleichförmige Netze miteinander verbunden sind.

Ein naiv vorgehender Partitionierungsalgorithmus wird nun häufig Partitionen berechnen, in denen Zellen voneinander getrennt werden, die eigentlich zu ein und derselben High-Level-Einheit gehören. Aus dem bereits erwähnten Grund, daß derartige Zellen über viele Netze miteinander verbunden sind, wird durch einen solchen Algorithmus nur ein relativ schlechtes Cutset berechnet.

Intelligentere Partitionierer versuchen, in dem zu partitionierenden Design wieder Hierarchien einzufügen. Diese zusätzlich berechneten Informationen werden genutzt, um Zellen, die sehr stark miteinander vernetzt sind, zu identifizieren und zu einer größeren unteilbaren Zelle (Cluster) zusammenzufassen.

In diesem Kapitel wird das Konzept des iterativen Clusterns vorgestellt, bei dem rekursiv Hierarchie-Ebenen im Design erzeugt werden. Wie diese Erzeugung von Hierarchien mit der Logik-Partitionierung einhergeht, soll schematisch in dem folgenden Abschnitt 3.1 erläutert werden. Die konkrete Strategie, die zur Berechnung von Zellen-Clustern verwendet wird, wird in Abschnitt 3.2 vorgestellt.

3.1 Das Konzept des Iterativen Clusterns

Ein Partitionierungsalgorithmus, der die Methode des iterativen Clusterns nutzt, arbeitet stets in drei Phasen, die in folgender Abbildung 3.1 dargestellt sind:[1]

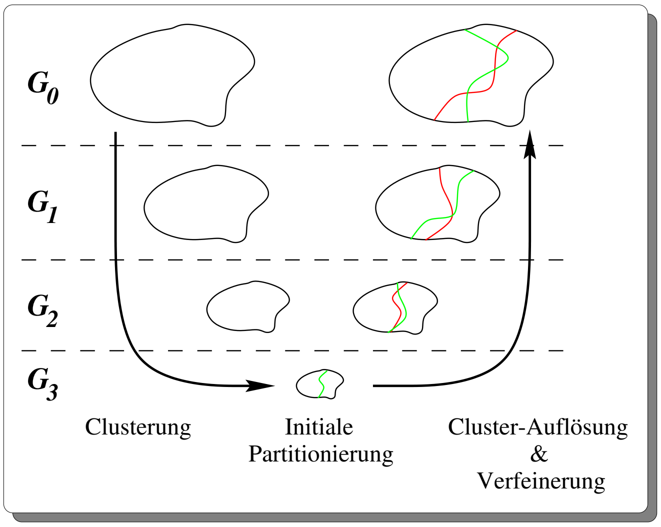


Abbildung 3.1: Partitionierung mit Iterativem Clustern

1. **Clusterung:** In der ersten sog. Clusterungsphase wird das ursprüngliche Design G_0 rekursiv geclustert, so daß eine Sequenz von Designs (in obiger Abbildung G_0 bis G_3) entsteht, die – von der Anzahl ihrer Zellen her betrachtet – kleiner werden. Die Intention dieser Phase ist, die Berechnung der Cluster möglichst entsprechend der ursprünglichen High-Level-Einheiten vorzunehmen. Grundlage dieser Phase ist eine mathematische Vorschrift, nach der einzelne Zellen zu einem neuen Cluster zusammenzufassen sind, die die Entsprechung mit ursprünglichen Hierarchien möglichst gut modelliert.[2]

[1] Entnommen aus [KAKS97].

[2] Aufgrund der bereits erwähnten Zuordungsprobleme zwischen High-Level-Hierarchien und Low-Level-Zellen kann allerdings die Güte eines Clusterungsverfahrens nicht objektiv bewertet werden.

Clusterungsverfahren treffen ihre Entscheidung auf der Basis von Netz- oder Zellen-Eigenschaften. Im ersten Fall werden alle diejenigen Zellen geclustert, die über eine Hyperkante mit bestimmten Merkmalen miteinander verbunden sind. Im letzteren Fall wird für alle Paare von Zellen ein sog. *Connectivity-Maß* berechnet, das ausdrücken soll, wie stark je zwei Zellen „zusammengehören". Es werden sukzessive die Paare mit dem höchsten Connectivity-Wert geclustert.

Das rekursive Clustern währt solange, bis ein Abbruch-Kriterium erfüllt ist. In ein derartiges Kriterium geht häufig die Größe der größten Zelle oder die Anzahl von Zellen des jeweils letzten geclusterten Designs ein.

2. **Initiale Partitionierung:** In der zweiten Phase dieses Prozesses wird mit Hilfe eines beliebigen Verfahrens eine initiale Partitionierung des zuletzt generierten Designs – in der Abbildung G_3 – berechnet. Die hier erzeugte Partition dient als Grundlage für die anschließende Phase der Cluster-Auflösung.

3. **Cluster-Auflösung:** Besonderes Kennzeichen dieser letzten Phase ist die Verbindung des zugrunde liegenden Partitionierungsalgorithmus mit der Auflösung der erzeugten Cluster. Die in der Vorphase ermittelte initiale Partition wird auf das nächstgrößere Design (G_2) projiziert, indem alle Zellen von G_2, die in G_3 zu einem Cluster zusammengefaßt wurden, in die gleiche Partition abgebildet werden. Diese Projektion (in der Abbildung stets als rote Linie dargestellt) dient als Startlösung für einen erneuten Aufruf des Partitionierungsalgorithmus, der darauf aufbauend eine verfeinerte Partition für G_2 berechnet (grüne Linien). Dieser Prozeß des Projizierens und Verfeinerns wird solange fortgeführt, bis schließlich das ursprüngliche Design G_0 partitioniert ist.

   Zu Beginn dieser Auflösungs- und Verfeinerungsphase wird aufgrund der großen Zellen-Cluster durch die Partitionierung die „grobe Richtung" vorgegeben, in der eine gute Lösung gesucht wird. Je weiter man in dieser Phase fortschreitet, desto kleiner werden die zu partitionierenden Zellen. Man spricht auch von einer feineren *Granularität* des Designs. Mit abnehmender Granularität wird die Partitionierung gleichbedeutend mit der Fein-Einstellung der bisher ermittelten Lösung. Unzulänglichkeiten in der aktuellen Partition, die durch die bis dato betrachtete grobe Struktur des Designs auftreten, können nun durch gezielte lokale Bewegungen von Zellen korrigiert werden.

3.2 Ein Connectivity-Maß zur Berechnung von Zellen-Clustern

Der in dieser Arbeit vorgestellte Clusterungsansatz beruht auf einer zellenorientierten Methode, die jeweils zwei Zellen zu einem Cluster zusammenfaßt.

Um den Berechnungsaufwand während der Clusterung nicht zu groß werden zu lassen, wird das im folgenden präsentierte Connectivity-Maß nicht für alle möglichen Paare von Zellen berechnet. Vielmehr wird eine Methode angewandt, nach der Paare, für die ein Connectivity-Wert berechnet werden soll, selektiert werden.

Dazu wird zunächst in Form einer zufällig erzeugten Permutation eine Reihenfolge auf den Zellen festgelegt. Gemäß dieser Reihenfolge wird jede einzelne Zelle untersucht und unter Umständen mit einer anderen geclustert.[3] Für das Clustern der i-ten Zelle v_i dieser Reihenfolge mit einer weiteren Zelle gelten die folgenden Regeln:

- v_i kann nur dann mit einer anderen Zelle geclustert werden, wenn v_i selbst noch kein Cluster ist.
- Für das Clustern einer Zelle v_i kommen prinzipiell nur benachbarte Zellen v_j in Frage. Zellen, die zusammengefaßt werden, müssen also über mindestens ein Netz miteinander verbunden sein.
- Eine Zelle v_i wird mit einem Nachbarn v_j nur dann geclustert, wenn der daraus resultierende Cluster eine bestimmte Maximal-Größe nicht überschreitet. Ein Cluster darf höchstens 5 Prozent der Logik des gesamten Designs in sich vereinen, und die Anzahl der angeschlossenen Netze darf die geringste Anzahl von Pins eines FPGAs, für das partitioniert werden soll, nicht überschreiten.

Entsprechend dieser elementaren Regeln und der eingangs erwähnten Zellen-Reihenfolge werden nun die Zellen-Paare (v_i, v_j) gebildet. Für jedes Paar wird ein Connectivity-Wert bestimmt, der als Bewertungsmaßstab für das Clustern herangezogen wird. In dieses Maß fließt die *Bandbreite* zwischen v_i und v_j ein, die wie folgt definiert ist:

Definition 3.1 (Bandbreite)

e bezeichne ein Netz des Designs, das die Zellen v_i und v_j miteinander verbindet.

$$Bandbreite_{i,j} = \sum_{\{v_i, v_j\} \subseteq e} \frac{1}{|e|-1}$$

Mit Hilfe der Bandbreite soll ausgedrückt werden, wie stark zwei Zellen untereinander vernetzt sind. Dazu wird jedes Netz e zwischen v_i und v_j betrachtet. Dieses Netz verbindet $|e|$ Zellen miteinander und stellt für die Zellen v_i und v_j somit einen $|e|$-ten Bruchteil seiner Kapazität zur Verfügung.

Die Bandbreite zwischen zwei Zellen nimmt zu, wenn diese über viele Netze miteinander verbunden sind und wenn an diese Netze jeweils nur eine kleine

[3]Die Zellen-Netzliste, die von SYNOPSYS geschrieben wird, und die die Eingabe für den Partitionierer bildet, enthält die Zellen des zu partitionierenden Designs in einer Reihenfolge, die sich nach den verschiedenen Zellen-Typen richtet. Um das Clustern zweier Zellen eines übermäßig großen Types zu vermeiden, wird die Clusterungsreihenfolge zufällig bestimmt.

Menge von Zellen angeschlossen ist. Intuitiv betrachtet stellt diese Definition also eine probate Umschreibung für „zusammengehörige“ Zellen dar.

Das letztlich verwendete Connectivity-Maß setzt sich neben der Bandbreite auch aus weiteren Zellen-Charakteristika zusammen, wie der folgenden Definition 3.2 zu entnehmen ist:

Definition 3.2 (Connectivity-Maß)

a_i und a_j seien die Flächen bzw. p_i und p_j die Anzahl von Netzen der Zellen v_i und v_j.

$$Connectivity_{i,j} = \frac{Bandbreite_{i,j}}{a_i * a_j * (p_i - Bandbreite_{i,j}) * (p_j - Bandbreite_{i,j})}$$

Die Tatsache, daß bei dem auf diese Weise formulierten Maß die Zellen-Flächen im Nenner auftauchen, hat zur Folge, daß sehr große Zellen v_i und v_j schlecht bewertet und somit wahrscheinlich nicht zusammengefaßt werden. So wird die Bildung von großen Clustern, die alle weiteren Nachbarn anziehen, vermieden.[4] Dies ist insbesondere am Ende der Clusterungsphase von Bedeutung, da sich große Cluster in diesem Stadium ungünstig auf die nachfolgende Partitionierung auswirken.

Die restlichen Ausdrücke im Nenner sollen das Clustern von Zellen bevorzugen, bei denen die zugrunde liegende Bandbreite über möglichst viele Pins der Zellen erreicht wird.

Eine Zelle v_i wird somit mit demjenigen Nachbarn v_j geclustert, mit dem der höchste Connectivity-Wert gemäß obiger Vorschrift erzielt wird.

Nach dieser Methode werden alle Zellen eines Designs untersucht und eventuell geclustert. Das so erzeugte kleinere Design dient in der nächsten Iteration wiederum als Eingabe für die Clusterung, wie bereits im vorigen Abschnitt beschrieben. Die Clusterungsphase wird beendet, sobald in einem Design keinerlei neue Cluster gebildet werden konnten.

Mit dem Ende des Clusterns ist eine Sequenz von Designs mit gröber werdender Granularität erzeugt worden. Häufig tritt allerdings die Situation ein, daß während einer Iteration des Clusterns nur sehr wenige neue Cluster gebildet wurden, so daß zwei in der Sequenz aufeinanderfolgende Designs G_i und G_{i+1} einander sehr ähnlich sind. Für den Partitionierungsalgorithmus hat dies zur Folge, daß für das Design G_i keine Verbesserung der bisher besten ermittelten Partition gefunden wird.

Um nun derartige zeitintensive und gleichzeitig nutzlose Durchläufe des Partitionierers zu vermeiden, wird unmittelbar nach der Clusterungsphase eine Nachbereitung durchgeführt. Aus der erzeugten Design-Sequenz werden diejenigen

[4] Dieses Kriterium ist durchaus nicht redundant, wie aufgrund der bereits vorgegebenen Maximal-Größe von Clustern angenommen werden könnte. In der Praxis hat sich herausgestellt, daß die 5-Prozent-Schranke nur in den seltensten Fällen zum Tragen kommt. Die Notwendigkeit dieser Schranke liegt allein in der technischen Anbindung des FM-Algorithmus begründet (Details hierzu in Abschnitt 5.2).

Designs entfernt, die gegenüber ihrem Vorgänger in der Sequenz eine Reduktion der Zellen-Anzahl um nur zehn Prozent oder weniger erbracht haben.

4 – Bipartitionierung nach Fiduccia & Mattheyses

„Zwei Kleine machen ein Großes, zwei Wenig machen ein Viel.“
„Wenn's ans Teilen geht, so geht's ans Raufen.“

Deutsche Sprichwörter.

Blickt man in Literaturverzeichnisse zu Artikeln aus dem Bereich der Logik-Partitionierung, so stößt man unweigerlich auf den Artikel [FiMa82] von Fiduccia & Mattheyses aus dem Jahr 1982, in dem ein Algorithmus zur Bisektion von Designs vorgestellt wird. Obwohl dieser Algorithmus nun schon 16 Jahre alt ist, dient er vielen modernen Partitionierungsansätzen nach wie vor als Grundlage. Die Ursache für die Beliebtheit dieses Verfahrens liegt hauptsächlich in seiner Effizienz begründet.

Auch im Zusammenhang mit dem genetischen Algorithmus, der in dieser Arbeit beschrieben wird, gelangt der FM-Algorithmus zur Anwendung. Er wird mit der Intention eingesetzt, die Genetik in kürzerer Zeit gute Lösungen finden zu lassen. Wie die Verbindung zwischen Genetik und FM konkret realisiert wird, wird in Abschnitt 5.2 erläutert.

In diesem Kapitel wird zunächst der Ablauf des FM-Algorithmus schematisch dargestellt. In den darauf folgenden Abschnitten 4.2 und 4.3 werden die Methoden zur Berechnung von Zellen-Gewinnen und zur Bestimmung einer zu bewegenden Zelle präsentiert. Abschließend werden in Abschnitt 4.4 einige Erfahrungen vorgestellt, die beim Arbeiten mit dem Algorithmus von Fiduccia & Mattheyses gesammelt wurden.

4.1 Der schematische Ablauf des Algorithmus

Die Vorgehensweise, nach der der FM-Algorithmus das Cutsize-Maß minimiert, wird anhand der folgenden Abbildung 4.1 erklärt.

```
procedure FM(H : Hypergraph; r : real)

   variable
      base, G, G_max : integer;
      g : array of integer;
      Blocked, A, B, A_max, B_max : set of cell;

begin
   (A,B) := RandomInit(H, r, g);

   /* Durchläufe iterieren, bis keine Verbesserung erzielt wird */
   repeat
      G := 0;
      G_max := 0;
      Blocked := ∅

      /* Beginn eines Durchlaufes */
      while (Basis-Zelle existiert) do
      begin
         base := GetBaseCell(r);
         G := G + g_base;
         CorrectGains(base, g, A, B, H);
         Blocked := Blocked ∪ {v_base};

         /* Speichern der besten gefundenen Partition */
         if (G > G_max) then
         begin
            (A_max,B_max) := (A,B);
            G_max := G;
         end;
      end;

      /* Nächsten Durchlauf auf bester Partition starten */
      (A,B) := (A_max,B_max);
   until (G ≤ 0);
end;
```

Abbildung 4.1: Algorithmus von Fiduccia & Mattheyses

Der Algorithmus erhält als Eingabe neben dem zu partitionierenden Design einen Parameter r $(0 < r < 1)$, der das sogenannte *Balance-Kriterium* darstellt. r gibt das gewünschte Verhältnis der Anzahl von Zellen in einer Partition in Relation zur Gesamtzahl der Zellen des zu partitionierenden Designs an. Soll eine der beiden Partitionen mindestens 45 Prozent der Logik enthalten, so ist r auf 0.45 zu setzen. Dieses Kriterium ist als „*Kann*-Bedingung" aufzufassen: Der Algorithmus *kann* eine Bipartition liefern, die im Verhältnis r % zu $100 - r$ % balanciert ist, *muß* es aber nicht. Ein Ergebnis mit dem Verhältnis 50 % zu 50 % ist ebenso möglich.

Der FM-Algorithmus ist ein Verfahren, bei dem das Cutsize-Maß iterativ verbessert wird. Dabei wird – ausgehend von einer zufällig ermittelten initialen Bipartition (Zeile 9) – eine *Basis-Zelle* ausgewählt (Zeile 20) und von ihrer aktuellen Partition in die jeweils andere bewegt (Zeile 22). Die Auswahl der Basis-Zelle wird so vorgenommen, daß einerseits die Balance nach der Zellen-Bewegung erhalten bleibt und daß andererseits die Bewegung die größte Verringerung bzw. die geringste Vergrößerung des Cutsets nach sich zieht.

Nach dieser Bewegung wird die Basis-Zelle blockiert (Zeile 23), was bedeutet, daß sie für weitere Bewegungen nicht mehr zur Verfügung steht. Auf diese Weise wird sichergestellt, daß der Algorithmus terminiert.

Das Bewegen der jeweils günstigsten Basis-Zelle wird iteriert (Zeile 18), bis zu einem bestimmten Zeitpunkt die Situation eintritt, daß keine Basis-Zelle selektiert werden kann, weil entweder alle Zellen blockiert sind oder das Balance-Kriterium nicht mehr erfüllt wäre. Dann spricht man davon, daß ein Durchlauf des FM-Algorithmus beendet sei. Während eines Durchlaufes wird für jede erzeugte Bipartition protokolliert, welchen Gewinn (sprich: Reduktion des Cutsize-Maßes) sie erzeugt (Zeile 21). Die Bipartition eines Durchlaufes mit dem größten Gewinn wird zwischengespeichert (Zeilen 26 - 30).

Nach Vollendung eines Durchlaufes werden sämtliche Blockierungen von Zellen aufgehoben, und ein neuer Durchlauf auf den nunmehr frei bewegbaren Zellen wird gestartet (Zeilen 13 - 15). Für diesen neuen Durchlauf wird als Startlösung die beste Bipartition aus dem vorigen Durchlauf zugrunde gelegt (Zeile 34).

Der Algorithmus von Fiduccia & Mattheyses bricht ab, sobald während eines Durchlaufes das Cutsize-Maß nicht weiter verbessert werden konnte (Zeile 35).

Die wichtigsten Teilaufgaben des Algorithmus liegen in der Berechnung und Korrektur der Zellen-Gewinne sowie in der effizienten Bestimmung der zu bewegenden Basis-Zelle. Beide Aspekte werden in den nun folgenden Abschnitten behandelt.

4.2 Die Berechnung von Zellen-Gewinnen

Nachdem die Strategie geschildert wurde, nach der der FM-Algorithmus zur Cutsize-Minimierung vorgeht, muß die Auswahl einer zu bewegenden Basis-Zelle genauer dargestellt werden.

Die Grundlage zur Auswahl der Basis-Zelle stellen sog. Zellen-Gewinne dar. Der Gewinn g_i einer Zelle v_i wird mittels der an v_i angeschlossenen *kritischen Netze* berechnet. Dies sind Netze, die ihren *Schnittzustand* bei Bewegung einer Zelle ändern:

Definition 4.1 (Kritisches Netz)
Ein Netz e heißt kritisch, wenn es an eine Zelle v_i angeschlossen ist, durch deren Bewegung e entweder aus dem Cutset wegfiele oder neu in das Cutset aufgenommen werden müßte.

Es gibt genau vier Situationen, in denen ein Netz e kritisch ist, wie in Abbildung 4.2 verdeutlicht wird:

1. Alle Zellen von Netz e sind in Partition A.
2. Zelle v_i ist als einzige Zelle des Netzes in Partition B.
3. Zelle v_i ist als einzige Zelle des Netzes in Partition A.
4. Alle Zellen von Netz e sind in Partition B.

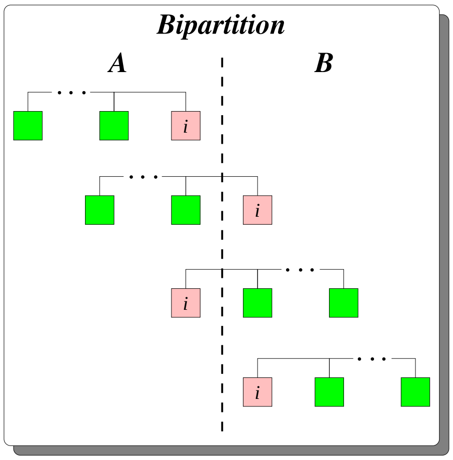


Abbildung 4.2: Auftreten kritischer Netze

Der Gewinn einer Zelle v_i, die sich im folgenden ohne Einschränkung der Allgemeinheit in Partition A befinde, soll die Reduktion des Cutsizes-Maßes bei Bewegung von v_i von A nach B widergeben. Diese Reduktion ergibt sich als Differenz der Anzahl kritischer Netze, die v_i als einzige Zelle in A haben, und der kritischen Netze, die keine Zelle in B haben:

Definition 4.2 (Zellen-Gewinn)

$A(e)$ und $B(e)$ seien die Mengen von Zellen eines Netzes e, die in Partition A bzw. B liegen.

Der Gewinn g_i der Zelle v_i aus Partition A berechnet sich durch

$$g_i = \left|\left\{e \mid \left(e \text{ kritisch}\right) \wedge \left(A(e) = \{v_i\}\right)\right\}\right| - \left|\left\{e \mid \left(e \text{ kritisch}\right) \wedge \left(B(e) = \emptyset\right)\right\}\right|$$

Definition 4.2 kann natürlich bedingen, daß Zellen einen negativen Gewinn besitzen. Ein derartiger Wert gibt dann die Vergrößerung des Cutsets bei Bewegung der betreffenden Zelle an. In einer Situation, in der sämtliche Zellen des Designs negative Gewinne aufweisen, wird diejenige Zelle mit dem betragsmäßig kleinsten Gewinn als Basis-Zelle ausgewählt. Obwohl dadurch die Menge der geschnittenen Netze vergrößert wird, wird die Bewegung ausgeführt. Dies geschieht in der Hoffnung, daß man auf diese Weise lokalen Minima ausweichen kann.

Vor Beginn eines jeden Durchlaufes des FM-Algorithmus lassen sich die Zellen-Gewinne durch einfaches Abzählen bestimmen. Die Laufzeit dieser Initialisierung beträgt $O(p)$ für $p = \sum p_i$ (p_i bezeichnet die Anzahl der an Zelle v_i angeschlossenen Netze). Innerhalb eines Durchlaufes, in dem fortwährend Zellen hin- und herbewegt werden, werden die Zellen-Gewinne nicht mittels Abzählen berechnet, da dies zu quadratischer Laufzeit führen würde. Vielmehr wird eine *Gewinn-Korrektur* nur für diejenigen Zellen durchgeführt, die unmittelbar von der Bewegung der jeweiligen Basis-Zelle betroffen sind.

Durch die Bewegung einer Basis-Zelle v_i ändert sich lediglich der Schnittzustand derjenigen Netze, die an v_i angeschlossen sind. Dementsprechend braucht eine Gewinn-Korrektur nur für die Zellen ausgeführt werden, die mit v_i über ein Netz e benachbart sind, wobei sich der Schnittzustand von e bei Bewegung von v_i ändert. Für alle Netze von v_i muß also geprüft werden, ob sie vor der Bewegung oder nach der Bewegung von v_i kritisch sind.

Ein Netz e ist *vor der Bewegung von* v_i von Partition A nach B kritisch genau dann, wenn $|A(e)| = 1$, $|B(e)| = 0$ oder $|B(e)| = 1$. Entsprechend gilt, daß e *nach der Bewegung von* v_i kritisch ist, wenn $|B(e)| = 1$, $|A(e)| = 0$ oder $|A(e)| = 1$.

Diese einzelnen Fälle lassen sich noch weitergehend vereinfachen, da gilt:
$|A(e)| = 1$ vor der Bewegung $\Leftrightarrow$ $|A(e)| = 0$ nach der Bewegung und
$|B(e)| = 0$ vor der Bewegung $\Leftrightarrow$ $|B(e)| = 1$ nach der Bewegung.

Der Ablauf der Gewinn-Korrektur bei Bewegung der Zelle v_i von Teilmenge A der Bipartition nach B folgt dann dem in Abbildung 4.3 dargestellten Schema:

```
procedure CorrectGains(i : integer; g : array of integer;
                       A, B : set of cells; H : Hypergraph)

begin
  ∀ Netze e von v_i:
    Berechne A(e) und B(e) aus A und B;

    /* e kritisch vor der Bewegung? */
    if (B(e) = ∅) then
      ∀ Zellen v_j ∈ A(e): g_j := g_j + 1;
    else if (B(e) = {v_j}) then
      g_j := g_j − 1;

    /* Bewegen von v_i */
    B(e) := B(e) ∪ {v_i};
    A(e) := A(e) \ {v_i};

    /* e kritisch nach der Bewegung? */
    if (A(e) = ∅) then
      ∀ Zellen v_j ∈ B(e): g_j := g_j − 1;
    else if (A(e) = {v_j}) then
      g_j := g_j + 1;
end;
```

Abbildung 4.3: Gewinn-Korrektur des FM-Algorithmus

Die Laufzeit dieser Gewinn-Korrektur für eine Zelle v_i ist wieder durch $O(p)$ begrenzt. Allerdings wird in [FiMa82] gezeigt, daß ein Netz während eines Durchlaufes höchstens viermal kritisch wird. Aus dieser Eigenschaft ergibt sich somit für einen einzelnen Durchlauf eine Laufzeit von $O(p)$, sofern in höchstens linearer Zeit die Zellen nach ihren Gewinnen sortiert und die Basis-Zellen bestimmt werden können. Die Datenstruktur, mit deren Hilfe diese Anforderungen erfüllt werden können, wird im folgenden Abschnitt 4.3 beschrieben.

4.3 Eine effiziente Datenstruktur zur Bestimmung der Basis-Zelle

Da die Zellen-Gewinne von der Anzahl der angeschlossenen kritischen Netze abhängen, gilt für alle Zellen v_i: $-p_i \leq g_i \leq p_i$. Wenn v_{max} diejenige Zelle bezeichnet, an die die meisten Netze angeschlossen sind, so gilt folglich, daß die Gewinne sämtlicher Zellen des Designs im Intervall $[-p_{max}, p_{max}]$ liegen.

Diese betragsmäßige Beschränkung der Gewinne legt nahe, daß zur Verwaltung der Zellen und ihrer Gewinne eine *Bucket-Datenstruktur* zum Einsatz kommt, wie sie in Abbildung 4.4 dargestellt ist. Für beide Partitionen A und B werden zwei eigenständige Mengen von Buckets eingerichtet, die die Zellen entsprechend der jeweils aktuellen Bipartition aufnehmen.

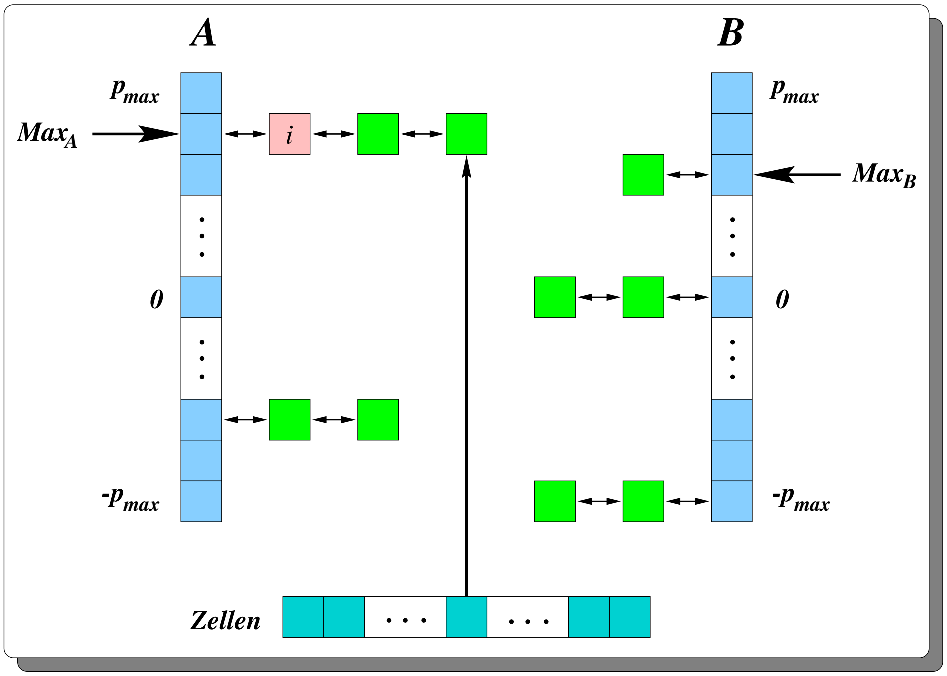


Abbildung 4.4: Bucket-Datenstruktur zur Verwaltung der Zellen-Gewinne

Die Buckets für beide Teile der Bipartition werden in Form von Feldern implementiert, die von $-p_{max}$ bis $+p_{max}$ indiziert werden können. In dem Bucket

$B[g]$ sind stets alle diejenigen nicht blockierten Zellen zu finden, die in Partition B liegen und Gewinn g haben. Diese Zellen werden innerhalb eines Buckets in einer doppelt verketteten Liste gespeichert, die in der Regel nach dem FIFO-Prinzip (First In, First Out) verwaltet wird.

Mit Hilfe dieser Datenstruktur ist es möglich, die Zellen des zu partitionierenden Designs in linearer Zeit nach ihren Gewinnen zu sortieren. Während des Sortierens werden stets zwei Zeiger Max_A und und Max_B aktualisiert, die auf die nicht leeren Buckets von A und B verweisen, die den höchsten Gewinn haben.

Für die Auswahl der Basis-Zelle werden zunächst die ersten Elemente der Buckets Max_A und Max_B betrachtet. Führt die Bewegung einer dieser Zellen zu einer unbalancierten Bipartition, so wird die Zelle verworfen und die nächste aus der Bucket-Liste herangezogen, etc. Findet sich in keinem der maximalen Buckets eine Zelle, die aufgrund des Balance-Kriteriums r bewegt werden darf, so wird der aktuelle Durchlauf des FM-Algorithmus beendet. Anderenfalls wird unter den Zellen, die bewegt werden dürfen, diejenige mit dem höchsten Gewinn als Basis-Zelle herangezogen. Zeigen sowohl Max_A als auch Max_B auf Buckets mit dem gleichen Gewinn, so wird die Zelle ausgewählt, die die bessere Balance nach ihrer Bewegung erzielt.

Im Rahmen der Gewinn-Korrektur nach Bewegung einer Zelle v_i müssen andere Zellen aus ihren bisherigen Buckets entfernt und in ihre neuen Gewinn-Buckets eingeordnet werden. Für eine effiziente Lösung dieser Aufgabe in linearer Zeit wird für jede Zelle ein Zeiger mitgeführt, der auf das betreffende Listen-Element der Zelle in der Bucket-Datenstruktur verweist.

Durch diese effiziente Datenstruktur wird gewährleistet, daß für einen Durchlauf des FM-Algorithmus lineare Laufzeit benötigt wird. Für die Anzahl der Durchläufe, die bei einem Aufruf des Algorithmus abgearbeitet werden, gibt es allerdings keine Abschätzung.

4.4 Beurteilung der Partitionierung nach Fiduccia & Mattheyses

Wie bereits im einleitenden Kapitel erwähnt, wurde während der Arbeit an dem Fuzzy-Controller zur Ampel-Steuerung die Logik-Partitionierung manuell für zwei FPGAs vom Typ XC4005 durchgeführt. Diese Partitionierung sollte nun automatisiert mit Hilfe des FM-Algorithmus vorgenommen werden.

Die Resultate, die sich bei dieser Anwendung des FM-Algorithmus ergaben, lassen sich wie folgt zusammenfassen:

- Die Qualität der berechneten Bipartitionen war ungenügend, so daß es nicht gelang, das Design für die oben beschriebene Zieltechnologie zu partitionieren.

Durch die manuelle Partitionierung wurde die Logik des Designs in zwei etwa gleichgroße Partitionen zerlegt, so daß beide FPGAs zu je 80 % ausgelastet waren. Allerdings fand der FM-Algorithmus weder mit einem Balance-Kriterium von $r = 0.5$ noch mit anderen Werten eine gültige Lösung. Selbst Modifikationen an der Zieltechnologie, bei denen FPGA-Typen eingesetzt wurden, die über mehr CLBs und/oder Pins verfügten, führten nicht zu gültigen Partitionen.

Die Gründe hierfür sind in einigen prinzipiellen Schwächen des Algorithmus zu suchen:

– Dem Algorithmus gelingt es häufig nicht, aus einem lokalen Minimum des Suchraums heraus weitere Verbesserungen zu finden.

  Dies liegt einerseits daran, daß der FM-Algorithmus ausgehend von nur einem Startpunkt, der in der Initialisierungsphase zufällig bestimmt wird, versucht, Verbesserungen vorzunehmen. Für die berechneten Ergebnisse ergibt sich daher eine große Abhängigkeit von der Wahl des Startpunktes. Andererseits reicht die Modellierung der Fähigkeit, lokalen Minima durch Auswahl einer Cutset-verschlechternden Basis-Zelle zu entgehen, in praktischen Fällen nicht aus, um den gewünschten Erfolg zu erzielen.

– Die in Abschnitt 4.2 beschriebene Formulierung von Zellen-Gewinnen basiert zu sehr auf lokalen Daten.

  Einerseits werden für die Berechnung des Zellen-Gewinns g_i lediglich die an v_i angeschlossenen Netze und somit nur die unmittelbar mit v_i benachbarten Zellen betrachtet. Dies wirkt sich insofern nachteilig aus, als die Bewegung von v_i völlig isoliert analysiert wird. Wie sich die gleichzeitige Bewegung von zwei Zellen oder von kompletten Gruppen zusammengehöriger Zellen auf das Cutsize-Maß auswirkt, bleibt unberücksichtigt.

  Andererseits stellt die ausschließliche Betrachtung kritischer Netze ebenfalls eine Einschränkung dar. Ein Netz e, für das beispielsweise $|A(e)| = 2$ gilt, hat keinen Einfluß auf die Berechnung von Gewinn-Größen. Somit bleibt auch in diesem Fall die Situation unberücksichtigt, daß die Bewegung beider Zellen aus $A(e)$ nach B eine Cutset-Reduzierung nach sich zieht.

  Der *Algorithmus von Krishnamurthy* ([Kris84]) verschafft in dieser Hinsicht Abhilfe, indem für jede Zelle nicht mehr ein einzelner Gewinn-Wert verwaltet wird, sondern vielmehr ein ganzer *Gewinn-Vektor* der Länge k.

  Der Gewinn an der ersten Stelle dieses Vektors entspricht genau den Zellen-Gewinnen nach Fiduccia & Mattheyses. Ist die l-te Stelle im Gewinn-Vektor einer Zelle v_i gleich Eins, so bedeutet dies, daß ein Netz durch l Zellen-Bewegungen (einschließlich v_i) aus dem Cutset entfernt werden kann. Enthält die $l+1$-te Stelle des Vektors den Wert -1, so lautet die Interpretation, daß nach Bewegung von v_i ein Netz nicht länger durch l nachfolgende Zellen-Bewegungen aus dem Cutset entfernt werden kann. Die Gewinn-Vektoren können lexikographisch sortiert und ebenfalls in effizienten Bucket-Datenstrukturen verwaltet werden.

- Die vorgestellten Bipartitionierungsalgorithmen von Fiduccia & Mattheyses bzw. von Krishnamurthy lassen sich nur äußerst schwer, teilweise auch gar nicht, an erweiterte Aufgabenstellungen anpassen.

  Zum einen ist eine Erweiterung auf Partitionierungen für beliebig viele FPGAs mit schlechten Resultaten verbunden, da für eine Zerlegung in bspw. 8 Partitionen 7 voneinander unabhängige Aufrufe eines Bipartitionierers notwendig sind. Wie bereits zu Beginn dieser Arbeit erläutert, sind Rückkoppelungen zwischen den einzelnen Aufrufen nicht möglich. Eine in der ersten Partitionierung getroffene – möglicherweise schlechte – Entscheidung kann später nicht mehr korrigiert werden.

  Zum anderen ist die Berücksichtigung von Board-Topologien durch diese Algorithmen nicht realisierbar. Durch eine Topologie werden obere Schranken für die Anzahlen geschnittener Netze zwischen verschiedenen FPGAs vorgegeben. Beide Algorithmen versuchen jedoch, bei der Minimierung des Cutsize-Maßes „ihr Bestes zu geben“, ohne dabei derartige absolute obere Schranken zu berücksichtigen. Insofern wäre die Angabe einer Board-Topologie sinnlos. Der Anwender kann nur hoffen, daß eine berechnete Partition in die vorhandene Topologie paßt. Zudem ist die Spezifikation einer Board-Topologie in der Praxis nur für mehr als zwei FPGAs sinnvoll, wofür Bipartitionierungsalgorithmen jedoch nicht ausgelegt sind.

Aufgrund dieser Ergebnisse stellte sich heraus, daß die Realisierung eines flexiblen und leistungsfähigen Partitionierungssystems allein auf der Grundlage des Algorithmus von Fiduccia & Mattheyses nicht möglich ist. Um dennoch zu guten Partitionierungsergebnissen zu gelangen, wurde ein genetischer Algorithmus formuliert, mit dessen Hilfe die Schwächen des FM-Algorithmus und des Algorithmus von Krishnamurthy ausgeglichen werden konnten.

Die Betrachtung genetischer Algorithmen in diesem Zusammenhang bot sich aus den folgenden Gründen an:

- Die Formulierung des Partitionierungsproblems als genetischer Algorithmus ist sehr leicht vorzunehmen.
- Eine Einschränkung der Partitionierung auf zwei FPGAs ist nicht notwendig. Es wird direkt für beliebige Anzahlen von FPGAs partitioniert.
- Genetische Algorithmen sind äußerst erweiterungsfähig. Die Anpassung an Board-Topologien sowie die Berücksichtigung weiterer benutzerdefinierter Vorgaben für die Partitionierung sind leicht zu integrieren.

- Durch die Verwendung einer frei erhältlichen Bibliothek ([Levi96]), die Routinen zur Formulierung genetischer Algorithmen bereitstellt, läßt sich der Aufwand für die Programmierung minimieren.
- Die bei einem genetischen Algorithmus notwendigen Berechnungen können parallel auf mehreren Rechnern durchgeführt werden.

In dem folgenden Kapitel 5 wird eine Lösung des Problems der Logik-Partitionierung auf der Grundlage eines genetischen Algorithmus vorgestellt.

5 – Ein genetischer Ansatz zur Logik-Partitionierung

„Instead, the circuit evolved from a "primordial soup" of silicon components guided by the principles of genetic variation and survival of the fittest.
[...]
If evolutionary design fulfils its promise, we could soon be using circuits that work in ways we don't understand.“

Aus: „New Scientist“, 15.11.1997.

Dieses Kapitel enthält eine detaillierte Beschreibung des genetischen Algorithmus, der zur Logik-Partitionierung entworfen wurde.

In Abschnitt 5.1 wird zunächst eine Einführung in das Gebiet der genetischen Algorithmen gegeben, wobei die dort üblichen Begriffe und Verfahren allgemein vorgestellt werden. Abschnitt 5.2 ist dem Thema gewidmet, wie der Algorithmus von Fiduccia & Mattheyses oder der von Krishnamurthy in den genetischen Algorithmus integriert werden kann und so zu dessen Verbesserung und Beschleunigung beiträgt. Anschließend wird in Abschnitt 5.3 die Methode vorgestellt, nach der die Güte erzeugter Partitionen bewertet wird. In Abschnitt 5.4 wird schließlich eine Erweiterung präsentiert, die die Behandlung von Topologien von FPGA-Boards erlaubt.

5.1 Zur Funktionsweise genetischer Algorithmen

Die Arbeitsweise genetischer Algorithmen ist der natürlichen *Evolution* nachempfunden, die versucht, Lebewesen einer Art immer besser an ihre Umwelt anzupassen. Die sich ständig vervollkommnende Anpassung der genetischen Informationen von Lebewesen (in der Biologie auch *Individuen* genannt) an sich verändernde Lebensräume wird in diesem Zusammenhang als Optimierungsproblem aufgefaßt. Analog dazu soll durch einen genetischen Algorithmus die Anpassung potentieller Lösungen für ein Optimierungsproblem an benutzerdefinierte Zielsetzungen verbessert werden.

Darwins Theorie der natürlichen Zuchtwahl basiert auf den folgenden Erscheinungen, die auch genetischen Algorithmen als Grundlage dienen:

- Die Individuen einer Art stehen untereinander in ständigem Wettbewerb. In diesem *Kampf ums Dasein* (struggle for life) überleben die am besten an ihre Umwelt angepaßten Individuen (survival of the fittest) und pflanzen sich fort. Die weniger Tauglichen pflanzen sich entweder gar nicht fort oder haben weniger Nachkommen.

  Auf diese Weise findet eine *Selektion* (natürliche Auslese) statt, die zu einer sich verbessernden Anpassung der Individuen führt.

- Die Nachkommen zweier Individuen sind nicht alle untereinander gleich, eine *Variation* in ihren Erbmerkmalen ist stets zu beobachten.

Diese beiden Prinzipien der Selektion und Variation von Individuen werden durch genetische Algorithmen nachgeahmt.

Bevor diese Verfahren in den Abschnitten 5.1.2 und 5.1.3 erläutert werden, werden zuvor in Abschnitt 5.1.1 einige Begriffsbildungen aus den Bereichen der Evolution und Genetik eingeführt und der Ablauf eines genetischen Algorithmus schematisch erläutert. Den Abschluß dieses einleitenden Teils bildet eine Übersicht über die gebräuchlichsten Parameter, mit denen man die Funktionsweise genetischer Algorithmen beeinflussen kann.

5.1.1 Evolution in der Informatik: Begriffe und Abläufe

Das zentrale Bestreben der natürlichen Evolution besteht in der Verbesserung des genetischen Materials von Individuen einer Art. Zu diesem Zweck müssen sich mehrere Individuen miteinander paaren. Eine derartige Gemeinschaft sich untereinander fortpflanzender Individuen heißt *Population*, häufig auch *Generation* genannt. Die Anzahl von Individuen in einer Population wird *Populationsgröße* genannt. Durch die Fortpflanzung werden die Erbinformationen zweier Individuen neu kombiniert und variiert, neue Individuen mit anderern Merkmalen werden erzeugt. Aus denjenigen Individuen der elterlichen Generation, die den Kampf ums Überleben bestehen, wird zusammen mit den neu erzeugten Individuen eine neue Population generiert, in der sich dieser Fortpflanzungskreislauf wiederholt.

Ein genetischer Algorithmus (siehe hierzu auch Abbildung 5.1) verwaltet ebenfalls Populationen von Individuen. Ein einzelnes Individuum eines solchen Algorithmus repräsentiert eine mögliche Lösung für das gestellte Optimierungsproblem. Durch Kombination von Merkmalen je zweier Individuen entstehen neue Lösungen, die in die nächste Population eingehen.

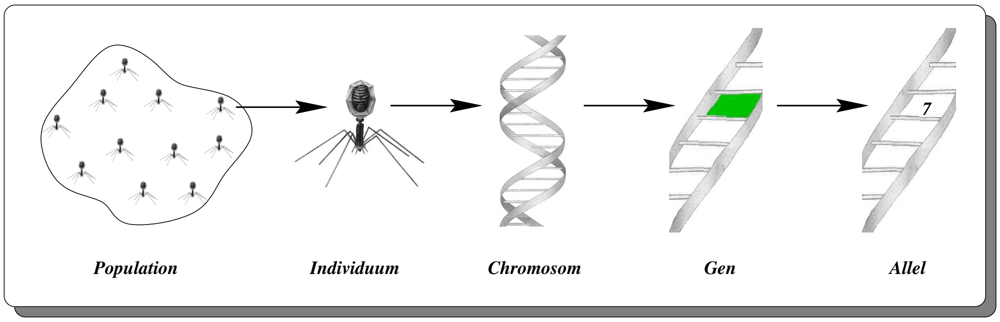


Abbildung 5.1: Wichtige Begriffe aus dem Bereich genetischer Algorithmen

Die Erbinformationen der Individuen eines genetischen Algorithmus werden durch sogenannte *Chromosomen* dargestellt, wobei ein Chromosom in viele einzelne *Gene* unterteilt ist. Ein Gen repräsentiert eine *Entscheidungsvariable* des Optimierungsproblems, kann also die verschiedensten Werte annehmen. Die Belegung eines Gens mit einem konkreten Wert heißt *Allel*. Ein Individuum wird demnach durch ein Array eines bestimmten Types (i. d. R. ganze Zahlen, Binär- oder Fließkommazahlen) repräsentiert.

Der Ablauf eines genetischen Algorithmus folgt stets dem in Abbildung 5.2 dargestellten Schema:

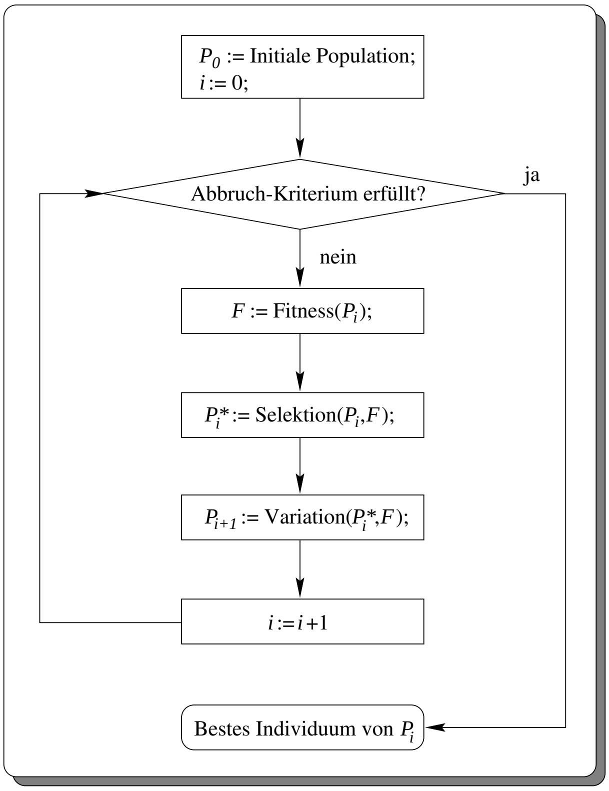


Abbildung 5.2: Ablaufdiagramm genetischer Algorithmen

- Zu Beginn des genetischen Algorithmus wird eine initiale Population P_0 per Zufall bestimmt.

- Anschließend wird für jedes einzelne Individuum der aktuellen Population i ein *Fitneß-Wert* bestimmt, der angibt, wie gut die durch das Individuum repräsentierte Lösung des Optimierungsproblems ist.

  Die Fitneß-Werte werden mit Hilfe einer *Bewertungsfunktion* berechnet, die der Anwender anzugeben hat. Diese Funktion stellt den wichtigsten Teil eines genetischen Algorithmus dar und muß daher mit entsprechender Sorgfalt formuliert werden, damit alle Zielsetzungen und Nebenbedingungen für die Optimierung korrekt modelliert werden.

- In Abhängigkeit von den berechneten Fitneß-Werten findet nun die Selektion aller Individuen statt, die in die nächste Population übernommen werden sollen.

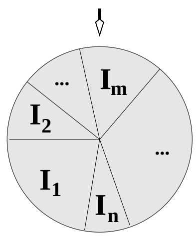
- Die Chromosomen der verbliebenen Individuen werden im anschließenden Schritt untereinander rekombiniert und variiert, so daß diejenigen Individuen, die vorher die Selektion nicht bestanden haben, durch Individuen der Folgegeneration ersetzt werden. Auf diese Weise wird die nächste Population P_{i+1} erzeugt.

- Bewertung, Selektion und Variation der Population P_i werden in einer Schleife iteriert, bis ein benutzerdefiniertes Abbruch-Kriterium erfüllt ist.

  Als berechnete Lösung gibt ein genetischer Algorithmus schließlich das beste Individuum der letzten Population zurück.

5.1.2 Survival of the fittest: Selektion von Individuen

Durch die Selektion werden diejenigen Individuen der Population P_i bestimmt, die ihre Gene in die nächste Population P_{i+1} einbringen dürfen. Die Auswahl der Individuen wird stets aufgrund der zugehörigen Fitneß-Werte getroffen. Hierbei wird zwischen Selektionsmechanismen unterschieden, bei denen generell diejenigen Individuen mit der größten Fitneß ausgewählt werden und solchen, bei denen ein großer Fitneß-Wert lediglich die Wahrscheinlichkeit eines Individuums erhöht, in die Folgepopulation übernommen zu werden.



Ein Selektionsschema, das in die zweite Kategorie fällt, ist die *fitneßproportionale Selektion*, deren Wirken gut durch ein Glücksrad wie in nebenstehender Abbildung veranschaulicht werden kann. Dieses ist in so viele Segmente unterteilt, wie Individuen in der Population P_i enthalten sind. Die Größe eines solchen Segmentes entspricht dem Anteil des Fitneß-Wertes des dazugehörigen Individuums an der Summe der Fitneß-Werte aller Individuen. Mit jeder Drehung dieses Rades wird das Individuum I_m selektiert, auf das der Pfeil zeigt, wobei ein einzelnes

Individuum mehrere Male in die Folgepopulation P_{i+1} übernommen werden kann.

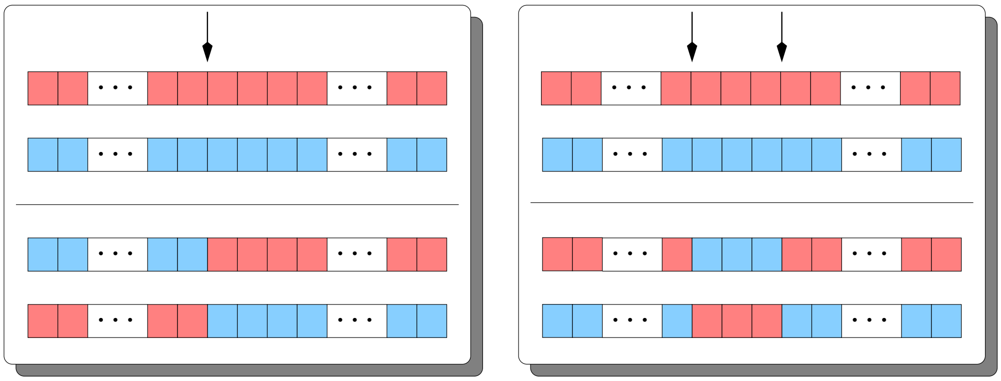
5.1.3 Die Variation von Erbmerkmalen

Die Erbinformationen der aus der Selektion hervorgegangenen Individuen werden abschließend untereinander kombiniert. Die in genetischen Algorithmen verwendeten Operatoren zur Rekombination sind – angelehnt an die Natur – das *Crossover* und die *Mutation*, auf die in den folgenden Unterabschnitten näher eingegangen wird.

Crossover

Während der Selektion werden die Individuen einer Population ausgewählt, die sich fortpflanzen sollen. Auf der Basis deren Genmaterials soll das Crossover neue Nachkommen erzeugen, die weitere Verbesserungen der Erbinformationen mit sich bringen. Zu diesem Zweck werden die Chromosomen zweier Individuen auf zufällige Weise zerteilt und wieder zusammengesetzt.

Die beiden gebräuchlichsten Varianten des Crossovers sind *1-Punkt-* und *2-Punkt-Crossover* (siehe Abbildung 5.3 a) und b)):



a) b)

Abbildung 5.3: 1-Punkt- und 2-Punkt-Crossover

1. Beim 1-Punkt-Crossover wird zufällig eine einzelne Stelle bestimmt, an der beide elterlichen Chromosomen aufgetrennt werden. Aus den resultierenden vier Chromosomen-Bruchstücken werden zwei neue Individuen zusammengesetzt, die jeweils ein Fragment beider Elterteile enthalten (Abbildung 5.3 a)).
2. Das 2-Punkt-Crossover zeichnet sich dadurch aus, daß über zwei wiederum zufällig ermittelte Crossover-Stellen ein Fragment der elterlichen Chromosomen bestimmt wird, das dann in das jeweils andere Chromosom eingesetzt wird (Abbildung 5.3 b)).

3. Eine Verallgemeinerung dieser beiden Crossover-Schemata stellt das n-Punkt-Crossover dar, bei dem über eine Auswahl von n Crossover-Stellen $n-1$ Chromosomen-Fragmente definiert werden, die zwischen den beiden zu kreuzenden Individuen ausgetauscht werden.

Eine zusätzliche weit verbreitete Variante ist das *uniforme Crossover*, bei dem für jedes einzelne Gen der elterlichen Individuen entschieden wird, ob dieses zwischen beiden Eltern vertauscht werden soll. Für alle Gene kommt hierbei eine einheitliche Austauschwahrscheinlichkeit zum Tragen, was z. B. beim 1-Punkt-Crossover nicht der Fall ist: für Gene, die sehr weit vorn auf einem Chromosom liegen, ist hier die Austauschwahrscheinlichkeit sehr hoch, während für die hinteren Gene das Gegenteil gilt.

Der Crossover-Operator läßt sich meist über die folgenden Optionen an die verschiedensten Bedürfnisse anpassen:

- Auswahl des verwendeten Crossover-Typs (n-Punkt- oder uniformes Crossover, ggfs. ein Wert für n),
- Einstellen der Crossover-Rate, die angibt, in wieviel Prozent aller Fälle zwei Chromosome mittels Crossover zu rekombinieren sind, und
- bei Nutzung des uniformen Crossovers die Angabe der Austauschwahrscheinlichkeit.

Mutation

Im Gegensatz zum Crossover stellt die Mutation einen Operator dar, der auf nur einem einzelnen durch die Selektion bestimmten Chromosom Veränderungen vornimmt, indem einzelne Allele zufällig geändert werden. Auf diese Weise werden völlig neue Merkmale in eine Population eingebracht, die allein durch Crossover nicht erzeugt werden könnten.

In Abhängigkeit vom Typ eines Gens wird die Mutation durch einen genetischen Algorithmus häufig wie folgt vorgenommen:

- Ein binäres Allel wird negiert.
- Zahlen werden um einen konstanten Betrag inkrementiert oder dekrementiert.
- Zahlen aus einem bestimmten Wertebereich werden durch eine andere Zahl aus diesem Bereich ersetzt.
- Die Allele zweier verschiedener Gene eines Chromosoms werden miteinander vertauscht.

Die wichtigsten Parameter zur Steuerung der Mutation sind einerseits die Mutationsrate, die die Wahrscheinlichkeit angibt, mit der ein Gen mutiert wird, sowie andererseits die Konstante, um die ein Allel verringert bzw. erhöht wird.

5.1.4 Parametrisierung genetischer Algorithmen

Um eine Übersicht zu vermitteln, wie flexibel genetische Algorithmen sind und wie leicht sie sich an die unterschiedlichsten Problemstellungen anpassen lassen, folgt an dieser Stelle eine Auflistung der wichtigsten Einstellungen, über die genetische Algorithmen gesteuert werden:

- der einem Chromosom zugrunde liegende Datentyp,
- die Auswahl, ob zur Optimierung eine Zielfunktion minimiert oder maximiert werden soll,
- die Populationsgröße,
- die *Ersetzungsrate*, die angibt, wie viele Individuen einer Population nicht selektiert und somit durch neu zu erzeugende Individuen ersetzt werden sollen,
- das Abbruch-Kriterium des Algorithmus, wobei häufig terminiert wird, wenn
  - eine maximale Anzahl von Generationen berechnet wurde,
  - die Individuen einer Population einander zu ähnlich sind, oder wenn
  - über eine bestimmte Anzahl von Generationen hinweg kein Individuum mit einem besseren Fitneß-Wert gefunden wurde,
- die Parameter für das Crossover,
- die Parameter für die Mutation,
- der Selektionstyp, der bestimmt, ob ausschließlich diejenigen Individuen mit den besten Fitneß-Werten selektiert werden oder ob auch schlechtere Individuen mit einer gewissen Wahrscheinlichkeit selektiert werden können.

5.2 Integration der Bipartitionierungsalgorithmen

Während der Arbeit am genetischen Partitionierungsalgorithmus hat sich herausgestellt, daß die zu Beginn generierte initiale Population P_0 einen sehr großen Einfluß auf Qualität und Laufzeit des gesamten Verfahrens hat. Diese zufällig bestimmten Startlösungen waren mit derartig großen Cutsets verbunden, daß die anschließende Optimierung dieser Population sehr lange Zeit benötigte. Terminierte der genetische Algorithmus schließlich, so war häufig die Qualität der besten berechneten Partition nicht zufriedenstellend.

Um diesen Mißstand auszugleichen, wurde die Initialisierungsphase des genetischen Algorithmus dahingehend abgeändert, daß eine Reihe qualitativ hochwertiger Startlösungen in die Population P_0 eingefügt wird. In diesem Zusammenhang werden zwei verschiedene Typen initialer Partitionen generiert:

1. Die wichtigsten anfangs berechneten Partitionen sind diejenigen, die das Bindeglied zwischen der Genetik und der iterativen Clusterung (vgl. Kapitel 3) darstellen.

   Sofern bereits in einem vorigen Durchlauf eine Partition eines Designs mit gröberer Granularität berechnet wurde, wird diese Partition in die Population P_0 des gerade gestarteten Durchlaufes übernommen, so daß der Optimierungsprozeß an genau der Stelle fortsetzen kann, an welcher vorher abgebrochen wurde.

   Zusätzlich wird die gleiche Partition als Startlösung für einen einzelnen Aufruf des Algorithmus von Fiduccia & Mattheyses verwendet. Die Partition, die nach diesem Aufruf zurückgegeben wird, wird ebenfalls in die Menge der Startlösungen für die Genetik übernommen. Während dieses Aufrufes des FM-Algorithmus muß es jedoch nicht unbedingt zu einer weiteren Verbesserung kommen, da unter Umständen schon zu Beginn das Balance-Kriterium verletzt wird.

2. Um auch für den Fall, daß das zuletzt durch die iterative Clusterung berechnete und somit zuerst zu partitionierende Design zerlegt werden soll, gute Ausgangslösungen bereitzustellen, werden die Algorithmen von Fiduccia & Mattheyses und von Krishnamurthy mehrfach hintereinander aufgerufen, wobei stets eine zufällig bestimmte Partition als Startlösung verwendet wird. Sämtliche Ergebnisse dieser Aufrufe werden in die Population P_0 eingefügt.

   Diese Aufrufe der Bipartitionierungsalgorithmen werden nicht nur für das zuletzt geclusterte Design durchgeführt (für dieses sind sie aber von besonderer Bedeutung), sondern auch für alle übrigen, so daß jedem Durchlauf des genetischen Partitionierungsverfahrens eine große Menge guten Genmaterials zugrunde liegt.

Die Struktur des in dieser Arbeit entworfenen Algorithmus zur Logik-Partitionierung wird in Abbildung 5.4 dargestellt. Neben den durch die iterative Clusterung berechneten zu partitionierenden Designs erhält der Algorithmus einen Wert N als Parameter, der die Anzahl der Aufrufe der Bipartitionierer angibt.

In einer äußeren Schleife werden nacheinander alle geclusterten Designs betrachtet und partitioniert (Zeile 13). Vor der eigentlichen Partitionierung werden bessere Startlösungen in die initiale Population P_0 eingefügt (Zeilen 18 - 32). Sobald dies geschehen ist, wird der genetische Algorithmus gestartet (Zeilen 34 - 41), dessen Struktur sich nicht von der in Abbildung 5.2 beschriebenen unterscheidet. Terminiert die Genetik, so wird die beste Partition der letzten Population bestimmt und als Ergebnis des aktuellen Partitionierungsdurchlaufes gespeichert (Zeilen 43 und 44).

Die durch die Zeilen 18 - 24 des Programmtextes generierten Partitionen, die in die Population P_0 übernommen werden, stellen die Verbindung zwischen den Partitionen vorheriger Genetik-Durchläufe und somit zur iterativen Clusterung her. Die oben unter Punkt 2 angesprochenen Partitionen werden duch die Zeilen 26 - 31 erzeugt.

```
procedure genetic_partition(designs : list of Hypergraph; N : integer)

   variable
      H, H* : Hypergraph := NULL;
      best_partition, best_partition* : Partition;
      P : list of Population;
      P* : Population;
      F : array of real;
      i : integer;

begin
   /* Für jedes von der Clusterung erzeugte Design einen Genetik-Durchlauf */
   while (is_empty(designs) = false) do
   begin
      P0 := InitPopulation;
      H := pop(designs);

      /* Beste Partition des letzten Durchlaufes auf Design H übertragen */
      if (H* ≠ NULL) then
      begin
         best_partition := project_partition(best_partition*, H*, H);
         P0 := P0 ∪ {best_partition};
         P0 := P0 ∪ {multi_FM(best_partition, H)};
      end;

      /* Mit Bipartitionierern weitere gute Startlösungen generieren */
      for i := 1 to N do
      begin
         P0 := P0 ∪ {multi_FM(NULL, H)};
         P0 := P0 ∪ {multi_Krishna(NULL, H)};
      end;
      i := 0;

      /* Normaler Ablauf des eigentlichen genetischen Algorithmus */
      while (Abbruch-Kriterium = false) do
      begin
         F := Fitness(Pi);
         P* := Selektion(Pi, F);
         Pi+1 := Variation(P*, F);
         i := i + 1;
      end;

      best_partition* := GetBestIndividual(Pi);
      H* := H;
   end;
end;
```

Abbildung 5.4: Genetischer Partitionierungsalgorithmus von Cobra

Im folgenden wird die Vorgehensweise veranschaulicht, nach der der Algorithmus von Fiduccia & Mattheyses innerhalb der Routine multi_FM aufgerufen wird, um Designs in mehr als zwei Partitionen zu zerlegen.[1] Dieses Schema wird in Abbildung 5.5 anhand einer beispielhaften Partitionierung für sieben FPGAs vorgeführt.

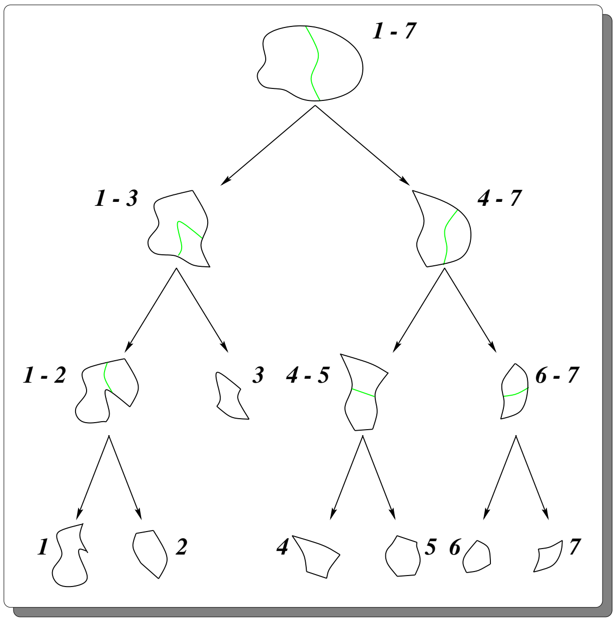


Abbildung 5.5: Partitionierung für 7 FPGAs mit dem FM-Algorithmus

Für eine Zerlegung eines Designs in sieben Partitionen sind offensichtlich sechs Bipartitionierungen notwendig. Jedes Design, das bipartitioniert werden soll oder aus einer Bipartitionierung hervorgeht, wird mit Indizes gekennzeichnet, die festlegen, auf welche FPGAs das Design abgebildet werden soll. So wird anfangs das komplette Design auf die FPGAs 1 - 7 abgebildet. Der Algorithmus von Fiduccia & Mattheyses wird stets mit einem Balance-Kriterium von 45 Prozent aufgerufen, was zur Berechnung nahezu gleichgroßer Partitionen führt. Dementsprechend werden die Indizes für eine berechnete Bipartition gebildet, indem das durch die Markierungen des partitionierten Designs festgelegte Intervall halbiert wird. Tritt die Situation auf, daß für eine ungerade Anzahl von FPGAs bipartitioniert werden soll (in der Abbildung die Designs mit den Markierungen 1 - 7 und 1 - 3), so werden die resultierenden Indizes durch Rundung ermittelt.

Die geschilderte Bestimmung der Indizes kann aus den folgenden Gründen zu Ungenauigkeiten bei den berechneten Partitionen führen:

- Die erwähnte Rundung ist offenbar nicht exakt, da z. B. in dem Fall des Designs 1 - 3 etwa 50 Prozent der Logik für das einzelne FPGA 3 bestimmt sind, während die restlichen 50 Prozent zwei FPGAs beanspruchen können.

[1] Diese Methode läßt sich ohne Änderungen auf den Algorithmus von Krishnamurthy und die Funktion multi_Krishna übertragen.

- Die rekursive Halbierung der Logik und der zugehörigen Intervalle setzt implizit voraus, daß alle verwendeten FPGAs gleich groß sind. Diese Voraussetzung ist in den meisten Fällen erfüllt, da moderne FPGA-Boards nur auf einem FPGA-Typ basieren. Der selbstverständlich mögliche Fall, daß für verschiedene FPGAs partitioniert werden soll, wird an dieser Stelle mit der folgenden Begründung nicht weiter behandelt:
  - Der Algorithmus von Fiduccia & Mattheyses versucht unabhängig vom angegebenen Balance-Kriterium, möglichst gut balancierte Bipartitionen zu berechnen, da dieses Kriterium nicht zwingend vorschreibt, wie groß einzelne Partitionen zu sein haben, sondern lediglich eine obere Schranke für die Variation der Partitionsgrößen darstellt.

    Soll ein Design für zwei verschiedene FPGAs bipartitioniert werden, deren Größenverhältnis zueinander z. B. 1 : 2 beträgt, so kann selbst ein Balance-Kriterium von $r = 0.33$ nicht verhindern, daß zwei ungefähr gleichgroße Partitionen berechnet werden, die nicht auf das kleinere FPGA passen.
  - Es sollte also stets von gut balancierten Partitionen ausgegangen werden. Entsprechend sollte die Aufteilung einer Menge von FPGAs gemäß ihrer Größe auch möglichst ausgewogen sein. Allerdings verbirgt sich hinter dieser Thematik das NP-vollständige Problem *Binpacking*. Der Aufwand zur annähernden Lösung dieses Problems würde in keinem Verhältnis zum erzielten Nutzen stehen.

Im schlimmsten Fall führt dieses Aufrufschema des Algorithmus von Fiduccia & Mattheyses dazu, daß eine hiermit berechnete Partition für mehr als zwei FPGAs nicht gültig ist, da Flächenbegrenzungen einzelner FPGAs verletzt sein können. Auf derartige Ungenauigkeiten wird jedoch nicht weiter Rücksicht genommen, da die mit Hilfe des FM-Algorithmus berechneten Partitionen lediglich das Ausgangsmaterial für die anschließende genetische Partitionierung darstellen. Durch den genetischen Algorithmus werden ungültige Partitionierungen identifiziert und im folgenden schnell verbessert bzw. nicht weiter selektiert.

Abschließend kann an dieser Stelle eine Frage beantwortet werden, die in Kapitel 3 zunächst noch offenblieb: In diesem Kapitel wurde die Tatsache erwähnt, daß Cluster von Zellen maximal fünf Prozent der Logik eines Designs enthalten dürfen. Die Begründung für diese Beschränkung ergibt sich aus dem verwendeten Balance-Kriterium für den Algorithmus von Fiduccia & Mattheyses.

Bei einer Mindestgröße für Partitionen von 45 Prozent der Logik kann eine Zelle, die z. B. 7 % der Logik enthält, nur dann von Partition A nach B bewegt werden, wenn A mindestens 52 % der Logik des gesamten Designs enthält. Dies muß jedoch in dem Moment, in dem die Entscheidung über die Bewegung dieser Zelle getroffen wird, nicht der Fall sein, da der FM-Algorithmus versucht, gleichmäßig balancierte Bipartitionen zu erzeugen. In diesem Fall kann die betreffende Zelle nicht bewegt werden, so daß unter Umständen nur eine geringere Reduzierung des Cutsize-Maßes erfolgt. Hat diese Zelle dagegen eine Größe von

5 %, so kann sie bewegt werden, wenn Partition A mindestens 50 % der Logik enthält, was der Zielsetzung des FM-ALgorithmus – der Balancierung der Bipartition – wesentlich näher kommt.

5.3 Die genetische Formulierung des Partitionierungsproblems

Der Entwurf eines genetischen Algorithmus zur Lösung eines Optimierungsproblems wird meist in zwei Phasen durchgeführt, wobei den einzelnen Phasen die folgenden Fragestellungen zugrunde liegen:

1. Welches sind die zur Optimierung zu belegenden Entscheidungsvariablen? Wie können diese möglichst einfach in Form eines Chromosoms codiert werden?
2. Auf welche Weise ist die Güte einer durch ein Chromosom repräsentierten Lösung zu bestimmen?

In diesem Teil der Arbeit werden diese Fragen – bezogen auf die Logik-Partitionierung mit COBRA – beantwortet.

5.3.1 Das zur Partitionierung verwendete Genmaterial

Die grundlegenden Entscheidungen, die zur Logik-Partitionierung getroffen werden müssen, betreffen die Auswahl eines bestimmten FPGAs für jede Zelle des zu partitionierenden Designs. Dementsprechend soll ein Chromosom für jede Zelle v_i eines Designs genau ein Gen enthalten. Das i-te Allel eines Chromosoms soll das FPGA bezeichnen, auf das die Zelle v_i abgebildet wird. Folglich wird zur Codierung des Partitionierungsproblems der Wertebereich der möglichen Allele auf die natürlichen Zahlen aus dem Intervall $[1, u]$ begrenzt, wenn zur Partitionierung u FPGAs verfügbar sind. Der Variationsoperator der Mutation ist so einzustellen, daß bei der Veränderung einzelner Allele ausschließlich Werte aus dem gleichen Intervall erzeugt werden.

Im folgenden wird nicht mehr nur von Chromosomen die Rede sein, vielmehr wird der Einfachheit halber der Begriff „Partition" verwendet. Unter einer Partition habe man sich dann ein Array vorzustellen, für das gilt:

$$partition[i] = l \Leftrightarrow \text{Zelle } v_i \text{ wird auf FPGA } C_l \text{ abgebildet}$$

5.3.2 Die Güte erzeugter Partitionen oder: Wie fit ist ein Individuum?

Die Fitneß-Werte, die die Güte genetisch erzeugter Partitionen bemessen, stellen das Optimierungsziel des genetischen Algorithmus dar. Sie werden durch eine Funktion berechnet, die beim Entwurf des genetischen Algorithmus angegeben werden muß. Diese Funktion hat die Aufgabe, ein einzelnes Chromosom einer Population zu untersuchen und dessen Fitneß in Form einer reellen Zahl zurückzugeben. Auf der Gundlage aller derartigen Zahlenwerte einer Population werden Selektion und Variation von Individuen vorgenommen. Der zur Logik-Partitionierung entworfene genetische Algorithmus verfolgt das Ziel, von Generation zu Generation die Fitneß-Werte zu minimieren.

Beim Entwurf einer adäquaten Bewertungsfunktion zur Partitionierung sollten vor diesem Hintergrund die folgenden Anforderungen berücksichtigt werden:

- Je besser eine Partition ist, desto kleiner muß der zugehörige Fitneß-Wert sein.

  Ungültige Partitionen, die gegen die in Kapitel 2.2 genannten Kriterien verstoßen, sind als besonders schlecht zu bewerten und daher mit entsprechend großen Fitneß-Werten zu versehen.

- Die Bewertungsfunktion sollte eine möglichst differenzierte Beurteilung vornehmen. Optimal wäre die (nicht realisierbare) Situation, in der für jede mögliche Partition ein individueller Fitneß-Wert berechnet wird, der von keiner weiteren Partition erzielt wird.

  Diese Differenziertheit bietet dem Selektionsmechanismus eine gute Basis zur Identifizierung der guten Individuen. Werden übermäßig viele Individuen auf den gleichen Fitneß-Wert abgebildet, obwohl sie (subjektiv betrachtet) in ihrer Güte stark variieren, so führt dies zwangsläufig zur Selektion minder guter Partitionen und somit zu schlechten Resultaten des genetischen Algorithmus.

- Da während eines Durchlaufes des genetischen Algorithmus eine große Anzahl verschiedener Partitionen generiert und somit auch bewertet wird, muß die Bewertungsfunktion effizient formuliert sein, um die Laufzeiten zur Partitionierung klein zu halten.

Die Zahlenwerte, die zur Bewertung von Partitionen berechnet werden, setzen sich aus zwei verschiedenen Elementen zusammen, die einerseits die „Bestrafung“ ungültiger Partitionen sowie andererseits eine Reihe charakteristischer Merkmale von Partitionen betreffen. Wie diese beiden Bereiche durch die Bewertungsfunktion behandelt werden, soll im folgenden dargestellt werden, wobei stets auf eine durch die Genetik generierte Partition *partition* Bezug genommen wird.

Die Bewertung ungültiger Partitionen

Eine Partition ist ungültig, wenn diese die durch die Flächen- und Pin-Ressourcen der FPGAs gegebenen Beschränkungen verletzt. Durch die Bewertungsfunktion soll das gesamte Ausmaß der Flächen- und Pin-Verletzungen aller FPGAs zahlenmäßig erfaßt werden. Zu diesem Zweck ist zunächst zu ermitteln, welche Fläche $area_l$ und wieviele Pins $pins_l$ eines jeden FPGAs C_l durch die gegebene Partition beansprucht werden.

Die belegten Flächen aller FPGAs lassen sich mit Hilfe eines Schleifen-Durchlaufes über alle Zellen $v_1, \ldots, v_s$ des zu partitionierenden Designs berechnen. Für jede betrachtete Zelle v_i muß unterschieden werden, ob es sich um einen IOB oder einen CLB handelt. Ist Ersteres der Fall, so belegt die Zelle v_i einen Pin des FPGAs $partition[i]$. Im anderen Fall ist der Wert der genutzten Fläche dieses FPGAs – $area_{partition[i]}$ – um die Fläche a_i der Zelle v_i zu erhöhen.

Zusätzlich zu den durch die IOBs belegten Pins sind weitere Pins für geschnittene Netze zu berechnen. Hierzu müssen nacheinander alle Netze $e_1, \ldots, e_t$ eines Designs betrachtet werden. Für jedes Netz e_k sind alle an dieses Netz angeschlossenen Zellen zu untersuchen und die FPGAs, auf die diese Zellen durch die gegebene Partition abgebildet werden, zu bestimmen. Sind die Zellen dieses Netzes auf mehr als zwei FPGAs verteilt, so ist e_k geschnitten, und bei allen beteiligten FPGAs ist jeweils ein Pin für das Netz zu belegen.

Nachdem auf diese Weise die exakten Werte für $area_l$ und $pins_l$ bestimmt wurden, können diese Größen mit den durch die FPGAs zur Verfügung gestellten Flächen- und Pin-Ressourcen A_l bzw. P_l verglichen werden. Gilt für ein FPGA C_l $area_l > A_l$, so ist hier eine Verletzung der Flächen-Ressourcen um $area_l - A_l$ zu beobachten (Verletzungen von Pin-Ressourcen analog). Diese Werte für die Flächen- und Pin-Verletzungen aller FPGAs werden in einer Variablen kumuliert und fließen in die Bewertung der Partition ein.

Verletzungen von Flächen- und Pin-Ressourcen werden bei diesem Verfahren gleich schwer geahndet, obwohl die Pin-Ressourcen den weitaus stärkeren Engpaß bilden. Es hat sich jedoch nicht bewährt, deshalb die Verletzungen der Pin-Ressourcen stärker zu gewichten als die der Flächen-Ressourcen, da dies zu einem einseitigen Optimierungsverhalten der Genetik führte, bei dem zunächst konsequent die Pin-Verletzungen und erst danach die Flächen-Verletzungen minimiert wurden. Durch die gleiche Gewichtung besteht für die Genetik vielmehr die Möglichkeit, Verbesserungen bezüglich der Fläche vorzunehmen, auch wenn dabei hinsichtlich der Pins kleine Verschlechterungen erzielt werden.

Das Verfahren zur Bewertung ungültiger Partitionen wird schematisch in Abbildung 5.6 dargestellt. Die Laufzeit einer Bewertung wird durch die Berechnung der kritischen Netze dominiert. Da hierbei sämtliche Netze und alle an ein Netz angeschlossenen Zellen einmal betrachtet werden, ergibt sich eine lineare Komplexität von $O(p)$ für $p = \sum\limits_{v_i \in V} p_i$.

Charakteristika zur Differenzierung von Partitionen

Neben dem Bestrafungsmaß für ungültige Partitionen – im folgenden mit V^* bezeichnet – setzt sich der letztlich von der Bewertungsfunktion gelieferte Wert aus drei weiteren Zahlen zusammen, die charakteristische Merkmale von Partitionen betragsmäßig ausdrücken. Im einzelnen sind dies:

```
function Punish(partition : Partition; H : Hypergraph;
                A, P : array of integer; u : integer) : real

   variable
      ret : real := 0.0;
      area, pins : array of integer := [0,...,0];
      S : set of integer;
      i, j, k, l : integer;

begin
   for i := 1 to H.s do
      if (Typ von v_i = IOB) then
         pins_partition[i] := pins_partition[i] + 1;
      else
         area_partition[i] := area_partition[i] + H.a_i;

   for k := 1 to H.t do
   begin
      S := ∅;
      ∀ Zellen v_j von Netz e_k : S := S ∪ {partition[j]};
      if (|S| ≥ 2) then
         ∀ FPGAs C_l, l ∈ S : pins_l := pins_l + 1;
   end;

   for l := 1 to u do
   begin
      if (area_l > A_l) then
         ret := ret + (area_l − A_l);
      if (pins_l > P_l) then
         ret := ret + (pins_l − P_l);
   end;
   return(ret);
end;
```

Abbildung 5.6: Bewertung ungültiger Partitionen

Maximale Flächenauslastung: $A^* = \max_{\text{FPGAs } C_l} \left\{ \left\lceil \frac{area_l \, * \, 100}{A_l} \right\rceil \right\}$

Für alle verfügbaren FPGAs wird berechnet, zu wieviel Prozent deren Chip-Fläche durch eine gegebene Partition ausgelastet ist. Der größte für ein FPGA ermittelte Wert wird in die Bewertung der Partition übernommen.

Mit Hilfe dieses Kriteriums soll erreicht werden, daß während der Optimierung gut balancierte Partitionen, bei denen alle FPGAs nahezu gleichmäßig ausgelastet sind, gegenüber unbalancierten bevorzugt werden. Zur Berechnung dieser Auslastungen kann auf die bereits vorher berechneten *area*-Werte zurückgegriffen werden, so daß kein zusätzlicher Rechenaufwand entsteht.

Anzahl genutzter Pins: $P^* = \sum_{\text{FPGAs } C_l} pins_l$

Durch die Summierung der *pins*-Werte über alle FPGAs wird der Umstand ausgedrückt, daß eine Partition als um so besser anzusehen ist, je weniger Pins sie beansprucht. Dieses Kriterium ist insbesondere dann von Bedeutung, wenn die untersuchte Partition gültig ist und somit keine IOB-Ressourcen verletzt. Der genetische Algorithmus fährt in dieser Situation fort, die totale Pin-Anzahl zu minimieren, wodurch sich letztlich größere Freiheiten für weitere Zellen-Bewegungen ergeben.

Anzahl genutzter FPGAs: $N^* = \sum_{\text{FPGAs } C_l} used_l$ mit

$$used_l = \begin{cases} 1, & \text{falls } area_l \neq 0 \ \vee \ pins_l \neq 0 \\ 0, & \text{sonst} \end{cases}$$

Unter Umständen kann die Situation eintreten, daß durch eine gegebene Partition keine einzige Zelle eines Designs auf ein bestimmtes FPGA abgebildet wird, dieses also ungenutzt bleibt. Vor dem Hintergrund der Kosten-Reduzierung soll durch die Genetik auch eine Minimierung der tatsächlich in Anspruch genommenen FPGAs angestrebt werden, indem diese Anzahl der genutzten FPGAs ebenfalls in die Bewertung eingeht.

In einem letzten Schritt wird der endgültig von der Bewertungsfunktion zu berechnende Fitneß-Wert aus den bis hierhin bestimmten vier Zahlenwerten zusammengesetzt. Dies geschieht mit Hilfe der folgenden Formel:

$$Fitness \ = \ c_1 * V^* \ + \ c_2 * A^* \ + \ P^* \ + \ N^*$$

Die Werte c_1 und c_2 sind Konstanten, mit deren Hilfe den Maßen V^* und A^* zu stärkerem Einfluß auf die Bewertung verholfen wird.

- c_1 wird derart in Abhängigkeit von der Größe des nicht geclusterten zu partitionierenden Designs gewählt, daß sich das Bestrafungsmaß V^* am stärksten auf die Fitneß einer Partition auswirkt und daß die Summe der drei übrigen Werte nicht größer wird als das gewichtete Bestrafungsmaß.

- c_2 wird in Abhängigkeit von der Gültigkeit der untersuchten Partition gewählt.

  Ist die betrachtete Partition ungültig, so wird der Faktor 10 verwendet. Dieser niedrige Wert wird herangezogen, da das Balance-Kriterium A^* für die Verbesserung ungültiger Partitionen und für das Finden einer gültigen Lösung nicht von entscheidendem Interesse ist.

  Die Balance wird jedoch wichtiger, wenn gültige Partitionen untersucht werden. Insbesondere für den Fall einer Partitionierung auf Gatter-Ebene sind balancierte Partitionen wichtig, da so die Wahrscheinlichkeit reduziert wird, daß aufgrund der ungenauen Kostenmaße zu viel Logik auf einzelne FPGAs abgebildet wird. Bei Betrachtung gültiger Partitionen wird daher der Faktor 100 genutzt.

5.4 Berücksichtigung von FPGA-Topologien

Zur Einführung in die Probleme, die zusätzlich durch die Spezifikation von Topologien von FPGA-Platinen entstehen, sei an dieser Stelle exemplarisch das WEAVER-Board betrachtet, das in [KKR98] vorgestellt wurde (siehe Abbildung 5.7).

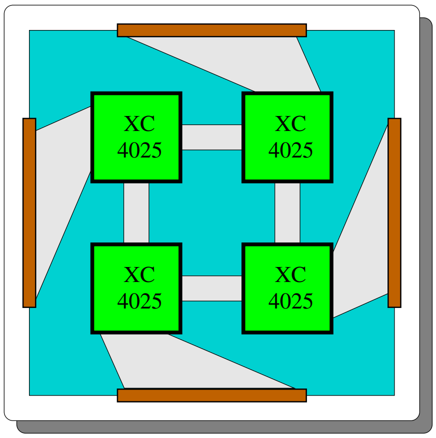


Abbildung 5.7: Die Topologie des WEAVER-Boards

Auf diesem Board befinden sich vier FPGAs vom Typ XILINX XC 4025 in quadratischer Anordnung. Jedes FPGA ist mit zwei seiner Nachbarn über jeweils 75 Pins verbunden, wobei diagonale Verbindungen in der quadratischen Anordnung nicht vorgesehen sind. 96 Pins eines jeden FPGAs sind mit einem Stecker an der Außenseite der Platine verbunden, der zum Anschluß weiterer Module dient. Mit Hilfe dieser Steckverbinder können weitere WEAVER-Boards oder zusätzliche Bus-, Interface-, Speicher- oder Prozessor-Module angeschlossen werden.

Eine Topologie läßt sich offenbar durch einen ungerichteten gewichteten Graphen modellieren, in dem die Knoten die einzelnen FPGAs der Platine darstellen. Eine Kante c in diesem Graphen repräsentiert eine Verbindung zwischen je zwei FPGAs und wird mit der entsprechenden Bitbreite max_c gewichtet. Zusätzlich erhält jeder Knoten C_l eine Zahl E_l als Markierung, die die Anzahl der möglichen externen Verbindungen eines FPGAs angibt.

Durch die Angabe einer Board-Topologie ergeben sich für einen Partitionierungsalgorithmus die beiden folgenden Aufgabenstellungen:

- Das Partitionierungverfahren hat dafür Sorge zu tragen, daß einem FPGA C_l nur solch eine Anzahl von IOBs zugewiesen wird, die die Zahl E_l der externen Anschlüsse dieses Chips in der Topologie nicht überschreitet.

Für konkrete Anwendungen eines Partitionierungsalgorithmus reicht die Überwachung der IOB-Anzahlen alleine jedoch nicht aus. Soll beispielsweise an den linken Steckverbinder eines WEAVER-Boards eine Speicherkarte angeschlossen werden, so ist unbedingt dafür zu sorgen, daß die IOBs des zu partitionierenden Designs, die die Schnittstelle zum Speicher darstellen (Daten- und Adreßbus sowie Steuerleitungen) auf das linke obere FPGA abgebildet werden. Die Partitionierung muß somit benutzerdefinierte *Plazierungsbeschränkungen* für Design-Ports berücksichtigen.

- Beim WEAVER-Board existiert z. B. keine direkte Verbindung zwischen dem linken oberen und dem rechten unteren FPGA. Ein geschnittenes Netz, das aber zwischen diesen FPGAs verläuft, muß daher über ein weiteres FPGA – z. B. das linke untere – geleitet werden.

  Der Partitionierungsalgorithmus muß also ein *Routing* durchführen, während dessen festlegt wird, mit Hilfe welcher zusätzlichen FPGAs eine Verbindung zwischen mehreren Chips aufgebaut wird, die über ein geschnittenes Netz miteinander kommunizieren. In diesem Zusammenhang ist darauf zu achten, daß die Anzahl der zusätzlich beanspruchten FPGAs minimiert wird, da anderenfalls schnell die überaus knappen Pin-Ressourcen überschritten werden.

Auf diese beiden Probleme wird in den beiden folgenden Abschnitten näher eingegangen. Für die Behandlung von Board-Topologien ist die Bewertungsfunktion (genauer: die Bewertung ungültiger Partitionen) des genetischen Algorithmus zu modifizieren. Je nachdem, ob der Benutzer eine Topologie angegeben hat, wird die ursprüngliche Funktion aus Abbildung 5.6 verwendet, oder die in Abbildung 5.8 und auf den folgenden Seiten präsentierte.

```
function Punish_Top(partition : Partition; H : Hypergraph; T : Graph;
                    A, E : array of integer; u : integer) : real

   variable
      ret : real;
      area, pins, iobs : array of integer := [0,...,0];

begin
   ret := Punish_Cells(partition, H, T, area, pins, iobs, A, E, u);
   ret := ret + Punish_Nets(partition, H, T, pins);

   return(ret);
end;
```

Abbildung 5.8: Bewertung ungültiger Partitionen für Board-Topologien

5.4.1 Externe FPGA-Anschlüsse und die IOBs eines Designs

Um sicherzustellen, daß der genetische Partitionierungsalgorithmus den Maximalwert E_l von externen Anschlüssen eines FPGAs C_l richtig in die Bewertung einer Partition einbezieht, müssen Werte $iobs_l$ berechnet werden, die angeben, wieviele IOBs durch die Partition auf FPGA C_l abgebildet werden.

Die Berechnung dieser Werte ist analog zu Abbildung 5.6 vorzunehmen, indem der Typ aller Zellen des Designs untersucht und der $iobs$-Wert des zugehörigen FPGAs entsprechend erhöht wird. Anschließend werden $iobs_l$ und E_l miteinander verglichen. Gilt für ein FPGA C_l $iobs_l > E_l$, so wird der zurückzugegebende Bestrafungswert um $iobs_l - E_l$ erhöht. Die Funktion Punish_Cells, die diese Elemente enthält, ist in Abbildung 5.9 dargestellt.

```
function Punish_Cells(partition : Partition; H : Hypergraph; T : Graph;
                      area, pins, iobs, A, E : array of integer; u : integer) : real

   variable
      ret : real := 0.0;
      i, l : integer;

begin
   for i := 1 to H.s do
      if (Typ von v_i = IOB) then
      begin
         pins_partition[i] := pins_partition[i] + 1;
         iobs_partition[i] := iobs_partition[i] + 1;
      end else
         area_partition[i] := area_partition[i] + H.a_i;

   for l := 1 to u do
   begin
      if (area_l > A_l) then
         ret := ret + (area_l − A_l);
      if (iobs_l > E_l) then
         ret := ret + (iobs_l − E_l);
   end;

   return(ret);
end;
```

Abbildung 5.9: Funktion Punish_Cells: Bewertung der IOB-Anzahlen pro FPGA

Die dauerhafte Fixierung einer Zelle v_i auf ein FPGA C_l der Topologie ist im Rahmen der Genetik ebenfalls leicht zu realisieren, indem vor der eigentlichen Bewertung einer aus Crossover und Mutation hervorgegangenen Partition $partition_i$ auf l gesetzt wird.

Diese Einschränkung der Partitionierung ist jedoch nicht nur durch den genetischen Algorithmus zu beachten, sondern auch durch die der eigentlichen

Partitionierung vorgeschalteten Stufen, nämlich der iterativen Clusterung und den Algorithmen von Fiduccia & Mattheyses und Krishnamurthy. Die Veränderungen, die an beiden Verfahren vorgenommen werden müssen, sind jedoch geringfügig, wie die folgenden Überlegungen zeigen:

- Zur Erinnerung seien hier noch einmal kurz die drei Bedingungen aufgeführt, die zur Verschmelzung zweier Zellen v_i und v_j erfüllt sein müssen:

  - v_i ist noch kein Cluster.

  - v_i und v_j sind benachbart.

  - Der durch v_i und v_j entstehende Cluster enthält höchstens fünf Prozent der Logik des Designs.

  Zur korrekten Clusterung von Zellen, die mit Plazierungsbeschränkungen belegt sind, ist lediglich die folgende vierte Bedingung hinzuzufügen:

  - v_i und v_j können geclustert werden, wenn beide Zellen auf das gleiche FPGA C_l abgebildet werden sollen bzw. wenn eine der beiden Zellen auf C_l zu plazieren ist und die andere keiner Beschränkung unterliegt.

    Der resultierende Cluster ist zwingend auf dem FPGA C_l zu plazieren.

- Um zu erzwingen, daß eine Zelle während des FM-Algorithmus (Analoges gilt für das Verfahren von Krishnamurthy) in einem bestimmten Teil der Bipartition enthalten ist und nicht aus diesem herausbewegt wird, ist zunächst die Erzeugung der zufälligen initialen Partition (vergleiche Abbildung 4.1, Zeile 9) so zu ändern, daß diese Plazierungsbeschränkung eingehalten ist.

  Im Anschluß daran ist Zeile 15 so anzupassen, daß die Menge *Blocked* aller blockierten und nicht mehr bewegbaren Zellen nicht auf die leere Menge zu setzen ist. Vielmehr ist *Blocked* mit der Menge aller Zellen mit Plazierungsbeschränkung zu initialisieren. Auf diese Weise wird eine Bewegung dieser Zellen verhindert, da die Bucket-Datenstruktur des Algorithmus, mit deren Hilfe die Basis-Zelle bestimmt wird, ausschließlich nicht-blockierte Zellen enthält.

5.4.2 Die Behandlung geschnittener Netze durch die Genetik

Die Behandlung geschnittener Netze ist im Vergleich zu der ursprünglichen Bewertungsfunktion komplexer geworden. Die Funktion Punish_Nets, die die Verdrahtung der einzelnen FPGAs untereinander bestimmt und bewertet, ist in Abbildung 5.10 dargestellt.

Zunächst ist für jedes Netz e_k zu überprüfen, ob es geschnitten ist, indem die FPGAs, auf denen sich die Zellen des Netzes befinden, in einer Menge S gespeichert werden (Zeilen 14 - 17). Enthält S mehr als zwei FPGAs (Zeile 18), so ist

```
function Punish_Nets(partition : Partition; H : Hypergraph; T : Graph;
                     pins : array of integer) : real

   variable
      ret : real := 0.0;
      j, k : integer;
      cost : array of edge := [1,...,1];
      connections : array of edge := [0,...,0];
      S : set of integer;
      B : set of edge;
      c : edge;

begin
   for k := 1 to H.t do
   begin
      S := ∅;
      ∀ Zellen v_j von Netz e_k : S := S ∪ {partition[j]};
      if (|S| ≥ 2) then
      begin
         B := Routing(S, T, cost);
         forall_edges(c, B) do
         begin
            connections_c := connections_c + 1;
            if (connections_c = T.max_c) then
               cost_c := 32767;
         end;
      end;
   end;

   forall_edges(c, T) do
   begin
      if (connections_c > T.max_c) then
         ret := ret + (connections_c − T.max_c);
      if (connections_c > 0) then
      begin
         pins_StartNode(c) := pins_StartNode(c) + connections_c;
         pins_EndNode(c) := pins_EndNode(c) + connections_c;
      end;
   end;

   return(ret);
end;
```

Abbildung 5.10: Funktion Punish_Nets: Behandlung geschnittener Netze

e_k geschnitten, und ein Routing des Netzes zwischen allen FPGAs aus S unter Berücksichtigung der spezifizierten Topologie muß durchgeführt werden.

Der Routing-Algorithmus liefert als Ergebnis eine Menge B von Kanten aus dem

Topologie-Graphen, über die Netz e_k geleitet werden muß, und die möglichst klein ist (Zeile 20). Für jede Kante aus dem Topologie-Graphen wird ein *connections*-Wert geführt, der die Anzahl der geschnittenen Netze widergibt, die über diese Kante der Topologie geroutet werden. Nach jedem Routing eines Netzes e_k müssen daher alle Kanten c aus der Menge B untersucht und die Werte $connections_c$ um 1 erhöht werden (Zeile 23).

Nachdem das Routing aller geschnittenen Netze durchgeführt wurde, sind abschließend sämtliche Kanten des Topologie-Graphen zu untersuchen (Zeilen 30-39). Wurden über eine Verbindung c zwischen zwei FPGAs C_l und C_m mehr Netze geleitet, als aufgrund der maximalen Bitbreite max_c möglich ist (also $connections_c > max_c$, Zeile 32), so wird das Bestrafungsmaß um $connections_c - max_c$ erhöht (Zeile 33). Die *pins*-Werte der beiden beteiligten FPGAs C_l und C_m werden entsprechend um $connections_c$ erhöht (Zeilen 34-38).[2]

Das Routing geschnittener Netze wird über einen Kostenvektor *cost* für die Kanten des Topologie-Graphen beeinflußt (Zeile 7). Anfangs enthält dieser Vektor für jede Kante den Wert 1, womit die Situation modelliert wird, daß keine Kante c der Topologie mit dem Maximal-Wert max_c von Netzen belegt ist. Nach jedem Routing eines geschnittenen Netzes wird überprüft, ob $connections_c = max_c$ für eine Verbindung c gilt (Zeile 24); ist dies der Fall (Zeile 25), so werden die Kosten für die Kante c auf einen sehr großen Wert (hier 32767) gesetzt. Hiermit wird erreicht, daß weitere Netze nach Möglichkeit nicht mehr über diese Verbindung c geleitet werden, da diese bereits maximal ausgelastet ist.

Der Routing-Algorithmus, der im folgenden vorgestellt wird, versucht, eine Menge S von FPGAs so miteinander zu verbinden, daß die Summe der Kosten aller Kanten, die für diese Verbindung verwendet werden, minimiert wird. Aufgrund dieser Kosten-Minimierung versucht der Algorithmus immer, zunächst noch freie Kanten mit Kosten 1 zu verwenden. Erst wenn dies nicht mehr gelingt, weil z. B. ein FPGA ausschließlich auf Wegen zu erreichen ist, die Kanten mit Kosten 32767 enthalten, werden auch solche Kanten benutzt. Dies führt allerdings zur Ungültigkeit der betrachteten Partition, was dann durch die Bestrafungsfunktion entsprechend berücksichtigt wird.

Eine effiziente Heuristik zur Verdrahtung innerhalb einer Topologie

Eine kostenminimale Verbindung einer Menge S von FPGAs innerhalb einer Topologie T hat notwendigerweise stets eine baumförmige Struktur, da eine Kante, die einen Kreis schließt, der Kostenminimalität widerspricht. Eine Methode zum Routing geschnittener Netze wird somit in irgend einer Form einen Baum berechnen und die Kantenmenge dieses Baumes an die Bewertungsfunktion des genetischen Algorithmus zurückgeben. Eine formale Modellierung des Verdrahtungsproblems wird durch die folgende Definition 5.1 von *Steiner-Bäumen* gegeben:

[2] Die *pins*-Werte fließen nicht mehr in das Bestrafungsmaß ein, da dies indirekt über die *connections*-Werte geschieht. Sie werden dennoch berechnet, um in ihrer Summe wie in Abschnitt 5.3.2 beschrieben durch die Bewertungsfunktion berücksichtigt zu werden.

Definition 5.1 (Minimaler Steiner-Baum)

$T = (V, E)$ sei ein ungerichteter Graph, $S \subseteq V$ und cost ein Kostenvektor über alle Kanten aus E.

Ein Steiner-Baum zu T und S ist ein Baum $B = (V', E')$ mit $E' \subseteq E$ und $S \subseteq V' \subseteq V$.

Ein minimaler Steiner-Baum eines Graphen T zu einer Knotenmenge S ist ein Steiner-Baum B, der unter allen möglichen Steiner-Bäumen die minimale Summe der Kanten-Kosten besitzt.

Das Problem der Bestimmung eines minimalen Steiner-Baums ist nach [GaJo79] NP-hart, so daß zur Lösung lediglich Näherungsverfahren zum Einsatz kommen. Ein effizientes Verfahren, das gute Resultate berechnet, wurde in [KMB81] vorgestellt. Dieser Algorithmus basiert auf dem *Distanzgraphen* zu T und S, der wie folgt definiert ist:

Definition 5.2 (Distanzgraph)

$T = (V, E)$ sei ein ungerichteter Graph, $S \subseteq V$ und cost ein Kostenvektor über alle Kanten aus E.

Der Distanzgraph $T_1 = (V_1, E_1)$ zu T und S ist ein vollständiger ungerichteter Graph, der die Elemente aus S als Knoten und alle Kanten zwischen je zwei verschiedenen Knoten enthält.

Eine Kante $(v, w) \in E_1$ wird mit dem kürzesten Abstand zwischen v und w in T gewichtet.

Der Basis-Algorithmus zur Bestimmung von Steiner-Bäumen mit geringen Kosten arbeitet nach dem folgenden Schema:

1. Berechne den Distanzgraphen $T_1 = (V_1, E_1)$ zu T und S.
2. Bestimme einen *minimalen Spannbaum* $T_2 = (V_2, E_2)$ von T_1.
3. Ersetze in T_2 alle Kanten $(v, w) \in E_2$ durch die kürzesten Wege zwischen v und w in T. Der resultierende Graph sei mit $T_3 = (V_3, E_3)$ bezeichnet.
4. Bestimme einen minimalen Spannbaum $T_4 = (V_4, E_4)$ von T_3.
5. Entferne rekursiv in T_4 alle Blätter, die nicht in S enthalten sind. Der resultierende Graph $T_5 = (V_5, E_5)$ ist konstruktionsbedingt ein Steiner-Baum.

In [Leng90] wurde gezeigt, daß dieses Verfahren Steiner-Bäume berechnet, deren Summe der Kantenkosten höchstens um den Faktor 2 von einem minimalen Steiner-Baum abweicht. Die Laufzeit des Algorithmus wird durch die Berechnung des Distanzgraphen dominiert, die bei Verwendung des *Algorithmus von Fredman und Tarjan* $O(|S| * (|E| + |V| \log |V|))$ beträgt.

In [Mehl88] wird eine Abänderung dieses Algorithmus vorgeschlagen, die eben die Berechnung des Distanzgraphen in Schritt 1 betrifft. Statt des Distanzgraphen wird ein neuer Hilfsgraph T_1' erzeugt, der auf einer Partition der Knoten aus T basiert. Für jeden Knoten $v \in S$ sei $N(v)$ die Menge aller Knoten aus V, die näher bei v liegen als bei irgend einem anderen Knoten aus S. Wenn $dist_{w,x}$ den kürzesten Abstand zwischen den Knoten w und x in T bezeichnet, so lassen sich die Mengen $N(v)$ wie folgt beschreiben:

$$N(v) = \{\, x \mid dist_{v,x} \leq dist_{w,x} \text{ für alle } w \in S\} \qquad \text{für } v \in S$$

Sollte ein Knoten x gleichnah an mehreren Knoten aus S liegen, so ist x in die Menge N eines einzigen dieser Knoten aus S einzufügen.

Auf der Basis der Mengen $N(v)$ wird der Hilfsgraph $T'_1 = (S, E'_1)$ mit der folgenden Kantenmenge generiert:

$$E'_1 = \Big\{\, (v,w) \;\Big|\; \big(v, w \in S\big) \wedge \big(\exists\, (x,y) \in E \text{ mit } x \in N(v),\, y \in N(w)\big)\Big\}$$

Eine Kante $(v, w) \in E'_1$ wird mit den folgenden Kosten gewichtet:

$$cost'_{1\,(v,w)} = \min\Big\{dist_{v,x} + cost_{(x,y)} + dist_{y,w} \;\Big|\; (x,y) \in E,\, x \in N(v),\, y \in N(w)\Big\}$$

In dem Artikel wird gezeigt, daß ein minimaler Spannbaum von T'_1 immer auch ein minimaler Spannbaum von T_1 ist. Daher kann der erste Schritt des Verfahrens aus [KMB81] in folgender Weise ersetzt werden:

1'. Berechne die Mengen $N(v)$ für $v \in S$, indem ein neuer Knoten v_0 und Kanten (v_0, v) für alle $v \in S$ mit Kosten 0 zu T hinzugefügt und in diesem Graphen die kürzesten Wege von v_0 zu allen übrigen Knoten bestimmt werden. Auf diese Weise erhält man für jeden Knoten $w \in V$ den dazugehörigen Knoten $v \in S$ mit $w \in N(v)$, sowie die Distanz $dist_{v,w}$. Erzeuge aus diesen Daten den Hilfsgraphen $T'_1 = (S, E'_1)$.

    Anschließend sind die Kosten für die Kanten aus E'_1 zu bestimmen. Hierzu sind zunächst für jede Kante $(x, y) \in E$ die Knoten v und w mit $x \in N(v)$ und $y \in N(w)$ zu bestimmen. Falls $v \neq w$ gilt, so wird der Wert $dist_{v,x} + cost_{(x,y)} + dist_{y,w}$ berechnet und mit einem bisherigen Minimum für die Kante $(v, w) \in E'_1$ verglichen.

Die Laufzeit dieses verbesserten Verfahrens ergibt sich wieder durch den ersten Schritt und beträgt aufgrund der Tatsache, daß nur noch ein einziger Aufruf des Algorithmus von Fredman & Tarjan notwendig ist, $O(|E| + |V| \log |V|)$.

In [Flor91] wurde eine weitere Verkürzung dieses Verfahrens vorgestellt. Es wurde gezeigt, daß bereits Schritt 3 des Algorithmus aus [Mehl88] einen Steiner-Baum zu T und S liefert, und daß man infolgedessen die nachfolgenden Schritte 4 und 5 wegfallen lassen kann. Da diese beiden letzten Schritte nicht für das Laufzeitverhalten des gesamten Algorithmus verantwortlich waren, verbleibt die totale Laufzeit bei $O(|E| + |V| \log |V|)$.

Der letztlich zum Routing geschnittener Netze verwendete Algorithmus hat somit das in Abbildung 5.11 dargestellte Aussehen:

```
function Routing(S : set of integer; T : Graph; cost : array of edge) : set of edge

    variable
      T1, T2, T3 : Graph;                /* Ti = (Vi, Ei, costi), 1 ≤ i ≤ 3 */

begin
    T1 := DistanzGraph(S, T, cost);
    T2 := MinSpannbaum(T1);
    T3 := Graph, der aus T2 hervorgeht, indem alle Kanten (v,w) ∈ E2 durch
            den kürzesten Weg zwischen v und w in T ersetzt werden;

    return(E3);
end;
```

Abbildung 5.11: Das Verfahren zum Routing geschnittener Netze

6 – Die Realisierung des Partitionierungsverfahrens COBRA

„In der indischen Mystik spielt die Kobra eine erhebliche Rolle. Noch heute ist sie das vom Publikum mit Schauder betrachtete Objekt der Schlangenbeschwörer.“

Aus: „Das neue Tierreich nach Brehm“.

Das in den vorherigen Kapiteln beschriebene Partitionierungsverfahren wurde im Rahmen dieser Diplomarbeit implementiert und ausgiebig getestet. In diesem Kapitel werden einige Details dieses praktischen Teils der Arbeit vorgestellt.

Den Anfang stellt eine Übersicht in Abschnitt 6.1 dar, in der die Einbindung verschiedener Synthese-Tools zur Automatisierung der Logik-Partitionierung sowie verschiedene Möglichkeiten der Partitionierung mit COBRA dargestellt werden. In dem folgenden Kapitel 6.2 werden kurz die wichtigsten Details zur Programmierung von COBRA hervorgehoben. Zur Validierung der Funktionsfähigkeit der implementierten Partitionierungssoftware werden in Abschnitt 6.3 Simulationsergebnisse präsentiert, die auf der Basis eines konkreten partitionierten Designs gewonnen wurden. Den Abschluß bildet Abschnitt 6.4, in dem die Qualität einer Anzahl mit COBRA berechneter Partitionen mit anderen Ergebnissen verglichen wird.

6.1 Partitionierung mit COBRA: Von VHDL zu XNF

Anhand von Abbildung 6.1 wird schematisch demonstriert, wie die Logik-Partitionierung durch COBRA vonstatten geht. Wie bereits kurz im ersten Kapitel erwähnt, wurde COBRA vor dem Hintergrund entwickelt, im Anschluß an das Codesign-Tool COOL Berechnungen durchzuführen. Zwischen COOL und COBRA liegt jedoch noch der Schnittstellen-Compiler CILC, der in [Schä98] entwickelt wurde. Dieser Compiler erhält von COOL die Beschreibung für eine

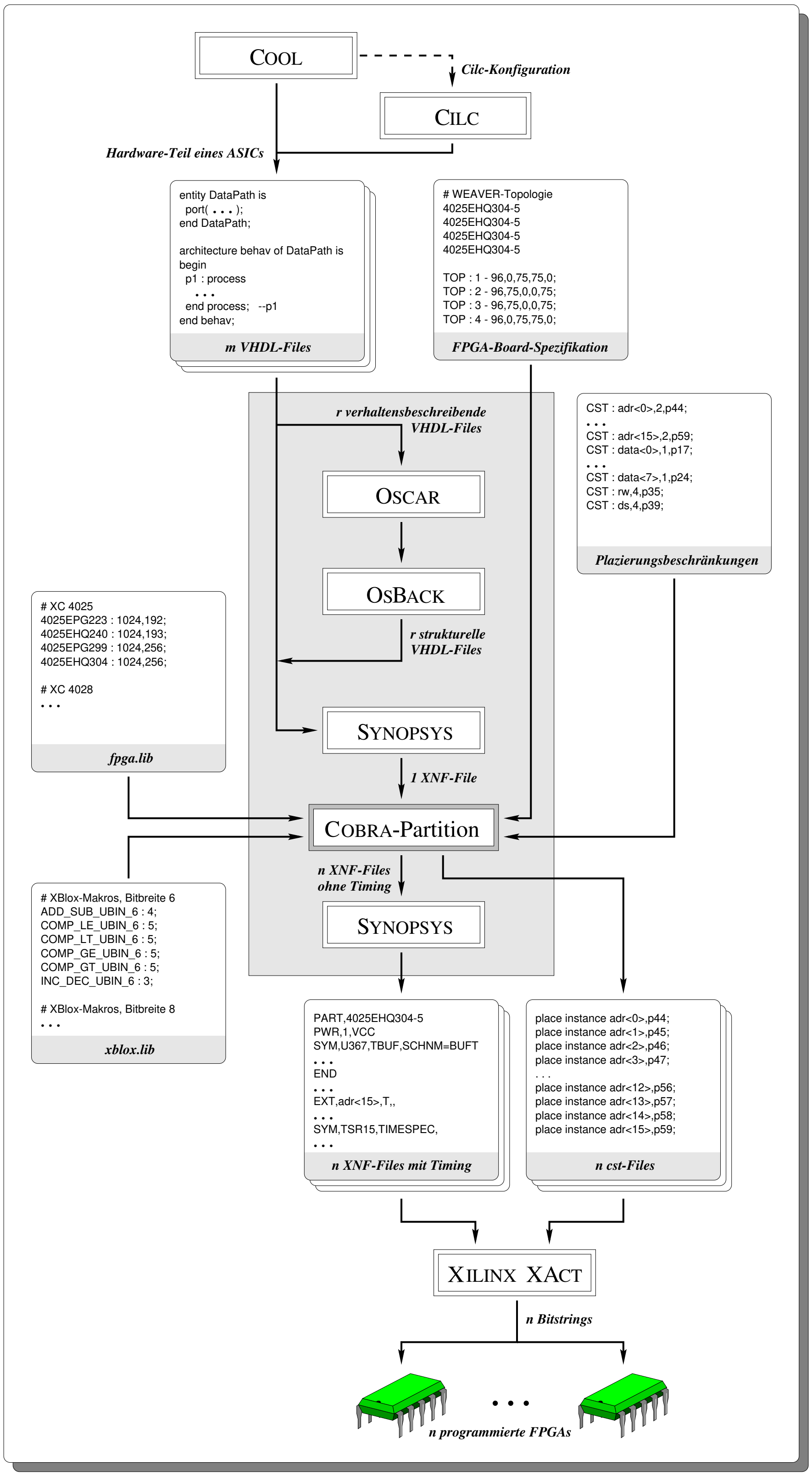


Abbildung 6.1: Ablauf der Partitionierung mit COBRA

externe Schnittstelle. CILC erzeugt aus dieser allgemeinen Spezifikation VHDL-Automaten, die dieses Interface implementieren, und fügt diese Schnittstelle in das von COOL stammende Design ein. Die Ausgaben von COOL und CILC liegen in Form von verhaltensbeschreibenden und strukturellen VHDL-Komponenten vor und sollen abschließend weitgehend automatisiert partitioniert werden.

Vor der eigentlichen Partitionierung sind die VHDL-Beschreibungen jedoch in eine spezifische Logik-Beschreibung für XILINX-FPGAs zu überführen. Dieser Synthese-Schritt wird von COBRA automatisch ausgeführt, indem zunächst die High-Level-Synthese für verhaltensbeschreibende Komponenten mit Hilfe von OSCAR und dessen Backend OSBACK durchgeführt wird. Anschließend wird eine Kommandozeilen-Version von SYNOPSYS aufgerufen, um die nunmehr in struktureller Form vorliegenden VHDL-Entities in reine Gatter-Logik zu überführen und danach in CLBs für XILINX-FPGAs zu gruppieren.

Die Ausgabe wird von SYNOPSYS in ein *XNF-File* (XILINX Netlist Format, [Xili95]) geschrieben, das eine strukturelle Beschreibung sämtlicher verwendeter CLBs und IOBs enthält. Neben diesen grundlegenden Elementen können die auf diese Weise erzeugten XNF-Dateien auch sogenannte *XBLOX-Makros* enthalten. Hierbei handelt es sich um besonders effiziente und platzsparende Beschreibungen für Addierer / Subtrahierer, Inkrementer / Dekrementer, Komparatoren, etc.

Nach dem Synthese-Schritt wird das Design von COBRA aus dem XNF-File eingelesen. Anschließend werden den Zellen in Abhängigkeit von ihrem Typ (CLB, IOB oder XBLOX-Makro) Kostenmaße zugewiesen[1], wonach das Design dann wie beschrieben unter Berücksichtigung der spezifizierten Zieltechnologie geclustert und partitioniert wird. Die Ausgabe von COBRA besteht aus einer Menge von XNF-Files, von denen jedes einzelne die Beschreibung für genau ein FPGA enthält, sowie aus der gleichen Anzahl von *Constraint-Files.* Ein Constraint-File für ein FPGA enthält die vom Benutzer angegebenen Plazierungsbeschränkungen von Design-Ports. Soll ein Port des Designs auf ein bestimmtes FPGA abgebildet werden, so enthält das Constraint-File für dieses FPGA einen Ausdruck, der diesen Port auf einem festen FPGA-Pin fixiert.

In einem letzten Schritt werden die vom Partitionierer generierten XNF-Files von SYNOPSYS eingelesen und um bestimmte Informationen zum zeitlichen Verhalten des partitionierten Designs erweitert (eine genaue Erklärung dessen, was an dieser Stelle geschieht, wird im folgenden Abschnitt 6.1.1 gegeben). Mit der Ausgabe dieser erweiterten XNF-Files ist die automatisierte Synthese und Partitionierung durch COBRA abgeschlossen.

Der Anwender muß nun lediglich die generierten XNF-Files durch das XILINX-Synthese-Werkzeug XACT bearbeiten lassen, um auf diese Weise eine Reihe von Files (in diesem Zusammenhang „Bitstring“ genannt) zu generieren, die Daten

[1] Jedes XBLOX-Makro benötigt eine konstante Anzahl von CLBs auf einem FPGA. Für die Partitionierung legt COBRA eine Bibliothek an, in der die gebräuchlichsten Makros mitsamt ihrem Platzverbrauch gespeichert sind, so daß auch für diese Arten von Zellen exakte Kostenmaße verwendet werden können.

zur Programmierung der einzelnen FPGAs enthalten. Diese Files können nun auf die FPGAs geladen werden, wonach man eine funktionsfähige Hardware-Realisierung des ursprünglichen Designs erhält.

Der bis hierhin geschilderte Ablauf der Logik-Partitionierung stellt den Regelfall dar. Abweichungen von diesem Schema können in zweierlei Hinsicht vorgenommen werden:

- Die Partitionierung soll nicht wie vorgesehen auf CLB-Ebene vorgenommen werden, sondern vielmehr auf Gatter-Ebene.
- Es soll kein konkretes aus COOL hervorgegangenes Design partitioniert werden, sondern lediglich ein Benchmark-Design, das in einer abstrakten Beschreibung auf Gatter-Ebene vorliegt.

Wie diese beiden Sonderfälle von COBRA unterstützt werden, wird in den Abschnitten 6.1.2 und 6.1.3 beschrieben.

6.1.1 Logik-Partitionierung und kritische Zeitanforderungen

Systeme, die mit Hilfe von COOL entworfen wurden, unterliegen in den meisten Fällen mehr oder weniger harten Zeitanforderungen (*Timing Constraints*). Hierbei handelt es sich in der Regel um minimale oder maximale Reaktionszeiten, also um Zeiten, die mindestens oder höchstens von dem Eintritt eines bestimmten Ereignisses im System bis zu einer Reaktion darauf an einer anderen Stelle des Systems verstreichen dürfen. Die Hardware-Teile eines derartigen COOL-Systems, die in Form von VHDL-Code ausgegeben werden, sind so konstruiert, daß diese benutzerdefinierten Zeitanforderungen inhärent erfüllt sind. Insofern schlagen sich diese Kriterien nicht in Form irgendwelcher Zeitangaben auf die generierten Programm-Texte nieder.

Das zu synthetisierende und zu partitionierende Design ist jedoch ein synchrones Schaltwerk, das somit einen *Takt* mit einer bestimmten Frequenz benötigt. Diese Takt-Frequenz geht in Form der sogenannten *Takt-Periode* in die Hardware-Synthese ein. Die Takt-Periode mißt die Zeit, die zwischen zwei Flanken auf dem Takt-Signal vergeht, und stellt somit eine während der Synthese zu berücksichtigende Zeitanforderung dar.

Diese taktbezogenen Zeitanforderungen gehen auch in das von SYNOPSYS erzeugte XNF-File ein, um während der später folgenden Bearbeitung durch XACT umgesetzt zu werden. Da zwischen diesen beiden Schritten die Logik-Partitionierung mittels COBRA liegt, ist zu klären, welchen Einfluß die Partitionierung auf das zeitliche Verhalten des Designs hat.

Die einzigen Teile eines synchronen Schaltwerkes, die getaktet sind, sind die als Speicherelemente dienenden Flip-Flops. Daraus ergibt sich, daß die für ein COOL-System generierten Timing Constraints in einem XNF-File stets von einem der drei in Abbildung 6.2 dargestellten Typen sind:

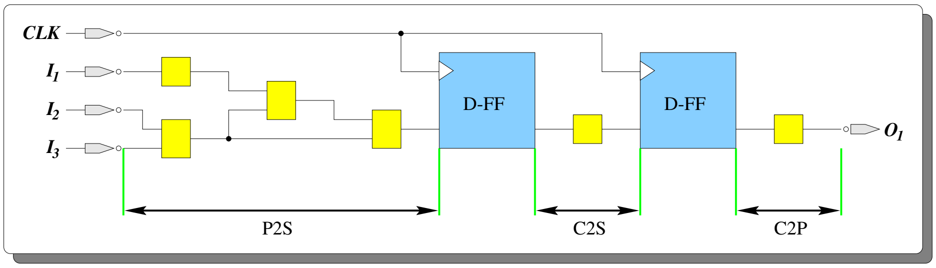


Abbildung 6.2: Taktbezogene Timing Constraints in XNF-Files

- **P2S** (Pad to Setup):
  Alle Pfade, die an einem Eingang des Schaltwerkes beginnen und an dem Eingang eines Flip-Flops enden, werden mit einem Constraint belegt.
- **C2S** (Clock to Setup):
  Alle Pfade, die an dem Ausgang eines Flip-Flops beginnen und an dem Eingang eines anderen Flip-Flops enden, werden mit einem Constraint belegt.
- **C2P** (Clock to Pad):
  Alle Pfade, die an dem Ausgang eines Flip-Flops beginnen und an einem Ausgang des Schaltwerkes enden, werden mit einem Constraint belegt.

In dieser Weise finden sich die Zeitangaben in einem XNF-File wieder: neben dem Typ des Constraints werden Start- und Endpunkt sowie eine dazugehörige Zeitspanne angegeben.

Während der Partitionierung werden nun einzelne Netze geschnitten, aus einem FPGA heraus- und in ein anderes hineingeleitet. Durch diese Umleitung eines Netzes über IOBs und aufgrund der Leitungsverzögerung, die sich durch die Verbindung zweier FPGAs über einen Draht ergibt, kann es vorkommen, daß einzelne Teile des partitionierten Designs mehr Zeit benötigen, als durch ein zugehöriges Timing Constraint erlaubt wäre. Dies kann dazu führen, daß das partitionierte Design mit der spezifizierten Takt-Periode nicht mehr korrekt arbeitet.

Ein Zerschneiden derartiger zeitkritischer Netze durch den Partitionierer kann jedoch nicht vermieden werden, da von XNF-Timing Constraints nicht auf zeitliche Vorgaben für einzelne Netze geschlossen werden kann. Ein derartiges Constraint bezieht sich stets auf mehrere zwischen Start- und Endpunkt verlaufende Pfade, wobei ein Pfad eine Aneinanderreihung von Netzen mit dazwischenliegenden logischen Zellen ist. In einem XNF-File sind jedoch keinerlei Informationen über Verzögerungszeiten einzelner Netze oder Zellen enthalten. Dies ist prinzipiell nicht möglich, da die endgültige Plazierung der logischen Zellen auf dem Chip, die unmittelbar die Signal-Laufzeiten auf dem FPGA bestimmt, von dem Synthese-Werkzeug XACT berechnet wird, welches erst nach der Partitionierung aufgerufen wird.

In einer derartigen Situation, in der ein partitioniertes Design für einen bestimmten Takt nicht mehr funktioniert, bleibt dem Benutzer nichts anderes übrig, als die Takt-Periode seiner Zieltechnologie soweit zu verlängern, bis die vormals verletzten Timing Constraints unter der neuen Takt-Periode eingehalten werden können. Dies führt natürlich zu einer Reduzierung der Geschwindigkeit des gesamten Systems. Unter dem Gesichtspunkt der Erstellung von System-Prototypen ist diese Verlangsamung jedoch durchaus hinnehmbar, da Prototypen nicht notwendigerweise Echtzeit-Anforderungen genügen müssen.

Trotz des Umstandes, daß eine Identifizierung zeitkritischer Netze nicht möglich ist, versucht COBRA, möglichst unkritische Partitionierungen zu berechnen. Durch die Tatsache, daß Partitionen gut bewertet werden, die möglichst wenige FPGA-Pins belegen, wird offensichtlich eine Minimierung des Cutsize-Maßes erstrebt. Je weniger Netze der Partitionierer zerschneidet, um so größer wird die Chance, daß sich kein zeitkritisches Netz im Cutset befindet. Zudem wird durch den beschriebenen Routing-Algorithmus versucht, geschnittene Netze über möglichst wenige FPGAs zu leiten, so daß auch die zusätzlichen Verzögerungszeiten, die sich durch die Schnitte ergeben, minimiert werden. Dieses Verfahren stellt jedoch nur eine Heuristik dar und kann gleichwohl nicht als Garantie für das Funktionieren des partitionierten Designs aufgefaßt werden.

Die einzige Möglichkeit, die COBRA bei der Handhabung von Timing-Problemen anbieten kann, besteht darin, für geschnittene Netze neue XNF-Timing Constraints zu erzeugen, die in ihrer Gesamtheit zu Constraints des nicht partitionierten Designs äquivalent sind. Dieses Vorgehen sei anhand von Abbildung 6.3 veranschaulicht.

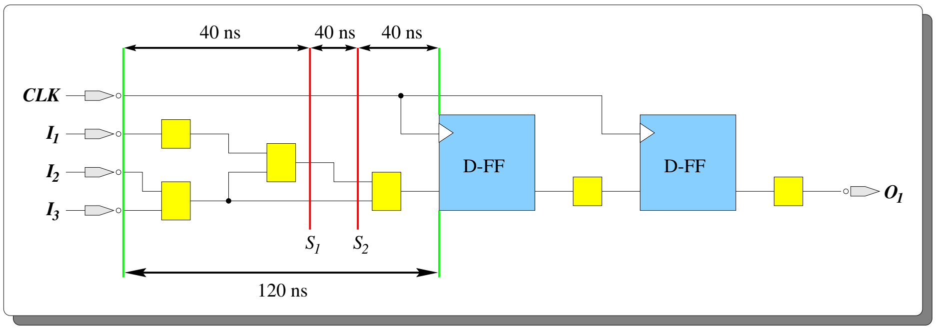


Abbildung 6.3: Timing Constraints für geschnittene Netze

Dargestellt ist eine Situation, in der von den Eingangssignalen I_1 - I_3 eines Designs bis zum Eingang des ersten Flip-Flops maximal 120 Nanosekunden vergehen sollen. Die Logik, die zwischen diesen Start- und Endpunkten liegt, ist anhand der Linien S_1 und S_2 auf drei FPGAs partitioniert worden. Von COBRA werden in diesem Fall die folgenden Timing Constraints erzeugt:

- Von den Eingangssignalen I_1 - I_3 bis zum ersten Schnitt S_1 sollen maximal 40 Nanosekunden vergehen.
- Von dem ersten Schnitt S_1 bis zum zweiten Schnitt S_2 sollen maximal 40 Nanosekunden vergehen.

- Von dem zweiten Schnitt S_2 bis zum Eingang des ersten Flip-Flops sollen maximal 40 Nanosekunden vergehen.

Da die Formulierung dieser neuen Constraints in einer XNF-gerechten Syntax aufwendig ist, wird diese Aufgabe mit Hilfe von SYNOPSYS gelöst. Nach der Partitionierung ruft COBRA SYNOPSYS auf und läßt das partitionierte Design aus den erzeugten XNF-Files einlesen. Hiernach erhält SYNOPSYS Kommandos, die zur Erzeugung der gewünschten Constraints führen, die dann schließlich in die XNF-Files geschrieben werden, die die endgültige Ausgabe von COBRA bilden.

Kann nun die Synthese dieses partitionierten Designs mit XACT erfolgreich beendet werden, so ist dieses Design auf jeden Fall mit der ursprünglich gewählten Takt-Periode lauffähig. Sollten diese von COBRA erzeugten Constraints jedoch nicht eingehalten werden können, so gibt XACT entsprechende Warnungen aus, mit deren Hilfe bestimmt werden kann, um wieviele Einheiten die Takt-Periode verlängert werden muß.

6.1.2 Partitionierung auf Gatter-Ebene

Eine Partitionierung auf Gatter-Ebene wird von COBRA zunächst in der Form unterstüzt, daß zum Abschluß der Logik-Synthese mit SYNOPSYS das synthetisierte Design auch in Form einer Gatter-Netzliste in ein XNF-File geschrieben wird. Indem man dieses Design an die Stelle der normalerweise verwendeten CLB-Netzliste kopiert, kann bei einem erneuten Aufruf von COBRA die Gatter-Struktur eingelesen und partitioniert werden.

Beim Einlesen des XNF-Files versucht COBRA, den Typ jeder Zelle zu bestimmen. Hierbei werden in jedem Fall zunächst CLBs und IOBs identifiziert. Für die Zellen, die weder CLB noch IOB sind, wird davon ausgegangen, daß es sich um ein XBLOX-Makro handelt. Die Kosten für diese Zellen werden der Bibliothek xblox.lib entnommen.

Da ein Design auf Gatter-Ebene keinen einzigen CLB und IOB enthält, wird für jede Zelle dieses Designs angenommen, daß sie ein Makro sei. Dementsprechend versucht COBRA, die Kosten der Gatter mit Hilfe der xblox-Bibliothek zu bestimmen. Der Anwender, der ein Design nun auf Gatter-Ebene partitionieren möchte, kann für jeden in einem derartigen Design vorkommenden Gatter-Typ (z. B. AND, INV, FDD) einen neuen Eintrag in dieser Bibliothek zusammen mit einem individuell wählbaren Kostenmaß einfügen. Für einen Gatter-Typ, für den kein Eintrag in dieser Bibliothek existiert, wird das einheitliche Kostenmaß 1 verwendet.

Da die Flächen, die von den einzelnen FPGA-Typen zur Verfügung gestellt werden, in der Bibliothek fpga.lib per Konvention ebenfalls in Form von CLBs angegeben sind, sind an dieser Stelle vor der Partitionierung ebenfalls manuelle Anpassungen vorzunehmen. Da die verwendeten Kostenmaße und FPGA-Größen von entscheidender Bedeutung für die Partitionierung sind, muß der

Anwender bei deren Festlegung entsprechende Sorgfalt walten lassen, um zu zufriedenstellenden Ergebnissen zu gelangen.

6.1.3 Benchmarking mit COBRA

Um die Güte des Partitionierungsverfahrens COBRA zu beurteilen, wird ein sogenanntes Benchmarking durchgeführt. Hierbei wird eine Vielzahl von Designs partitioniert. Anschließend werden die mit COBRA erzielten Cutsizes mit Werten verglichen, die mit anderen Verfahren für die gleichen Designs berechnet wurden.

Die Benchmark-Designs, die im Rahmen dieser Diplomarbeit verwendet und in Abschnitt 6.4 vorgestellt werden, liegen jedoch im allgemeinen nicht in VHDL- oder XNF-Form vor, sondern vielmehr in dem sogenannten *Net-Format* ([Alpe98]). Die Beschreibung eines Designs in diesem Format enthält neben den Zellen und deren Verbindungen untereinander als einzige zusätzliche Information den Typ von Zellen (Design-Port bzw. Gatter). Da keine weiteren Informationen über die Größe von Zellen vorliegen, werden diese zum Zweck des Benchmarking mit Kosten 1 versehen. Der Anwender hat hier keine Möglichkeit, die Bestimmung der Kostenmaße zu beeinflussen.

Über eine Kommandozeilen-Option ist es möglich, mit COBRA Benchmark-Designs in diesem Format einzulesen und zu partitionieren. In diesem Fall wird die Bearbeitung des eingelesenen Designs mit OSCAR und SYNOPSYS unmittelbar vor bzw. nach der eigentlichen Partitionierung unterdrückt, da es sich nicht um ein synthetisierbares Design handelt. Die Ausgabe des Partitionierers besteht beim Benchmarking lediglich aus einer kurzen Zusammenfassung über die verbrauchten FPGA-Ressourcen der besten gefundenen Partition sowie aus dem resultierenden Cutsize.

6.2 Zur Programmierung der Partitionierungssoftware

COBRA wurde in der Programmiersprache C^{++} implementiert. Als Rechner-Plattform sowohl für die Programmierung als auch für die zahlreichen in Abschnitt 6.4 dokumentierten Testläufe des Partitionierers diente eine Workstation vom Typ SUN ULTRASPARC unter dem Betriebssystem SUNOS 5.5.1.

Die bei der Programmierung entworfenen Datenstrukturen sowie große Teile des genetischen Algorithmus basieren auf einigen frei verfügbaren Funktionsbibliotheken für die Programmiersprache C^{++}. Die Verwendung dieser Bibliotheken bringt die Vorteile mit sich, daß auf effizient programmierte, wenig fehleranfällige und komfortabel nutzbare Routinen zurückgegriffen und somit die Zeit für eine eigenständige äquivalente Implementierung dieser Funktionen und für die einhergehende Fehlersuche eingespart werden kann. Im einzelnen handelt es sich um die folgenden Bibliotheken:

LEDA R 3.3.1 ([NäUh96]):
: LEDA stellt eine Bibliothek effizienter Datenstrukturen und Algorithmen dar.

  Mit Hilfe der zur Verfügung gestellten Datentypen für Listen, Mengen, Suchbäume etc. wurden u. a. die Strukturen, die die zu partitionierenden Designs repräsentieren, modelliert.

  Von großer Bedeutung waren außerdem die Datentypen und Algorithmen, die sich auf die Bearbeitung von Graphen bezogen. Mit deren Hilfe wurde der in Abschnitt 5.4 beschriebene Algorithmus zum Routing geschnittener Netze innerhalb einer Topologie effizient implementiert.

PGAPACK 1.0 ([Levi96]):
: Diese Bibliothek stellt dem Benutzer eine vollständige Sammlung von Funktionen zur Formulierung genetischer Algorithmen zur Verfügung.

  Sämtliche für genetische Algorithmen bedeutsame Mechanismen wie z. B. Populationsverwaltung, Selektion, Mutation und Crossover sind hier vordefiniert und über eine Vielzahl von Parametern an individuelle Bedürfnisse anpassbar.

  Der Benutzer muß der Bibliothek lediglich die gewünschten Einstellungen der Genetik-Parameter und eine Bewertungsfunktion übergeben. Der Start des genetischen Algorithmus kann dann über einen einzigen Funktionsaufruf vorgenommen werden.

  PGAPACK wurde so installiert, daß die zeitaufwendigen Bewertungen aller Individuen einer Population parallel auf mehreren SUN-Workstations, die über ein Netzwerk miteinander verbunden sind, durchgeführt werden können.

MPICH 1.1 ([GrLu97]):
: Dieses Paket stellt eine Umsetzung des *Message-Passing Interfaces* (MPI) dar, die sowohl für Parallel-Rechner als auch für vernetzte Workstations portabel ist. Die Verfügbarkeit der Funktionen dieser Bibliothek ist Voraussetzung für eine parallele Installation von PGAPACK.

  Die wichtigsten von PGAPACK benötigten MPI-Routinen betreffen die Erzeugung mehrerer parallel arbeitender Prozesse auf verschiedenen Rechnern sowie das Senden und Empfangen von (Broadcast-) Nachrichten zwischen diesen Prozessen.

Die Tatsache, daß der Kern von COBRA ein parallel ablaufender genetischer Algorithmus ist, wirkt sich auf die Struktur der implementierten Software aus. Alle Teile der Software, die nicht unmittelbar mit dem genetischen Partitionierungsalgorithmus in Zusammenhang stehen (also die Routinen zur Durchführung der Hardware-Synthese, zum Einlesen des zu partitionierenden Designs, zur iterativen Clusterung und zum Schreiben der XNF-Ausgaben), können nicht parallel zueinander ausgeführt werden. Würden der parallele genetische Algorithmus und diese sequentiellen Teile von COBRA in einem einzigen ausführbaren Programm zusammengefaßt, so würden auf allen spezifizierten Workstations parallele Prozesse erzeugt, die große Mengen sequentiellen Codes enthielten, welcher für die parallelen genetischen Berechnungen nicht benötigt würde.

Um diesen unnötigen Speicherverbrauch der parallelen Prozesse zu eliminieren, werden die oben erwähnten Software-Teile und der genetische Algorithmus voneinander getrennt und durch verschiedene ausführare Programme realisiert. Insgesamt besteht die Partitionierungssoftware von COBRA aus vier separaten ausführbaren Programmen, die über Text-Dateien miteinander kommunizieren:

cobra: Dies ist das Programm, mit dem der Anwender interagiert. Über Kommandozeilen-Optionen sind eine Reihe von Parametern für den Partitionierungsalgorithmus einstellbar.

Dieses Programm führt die automatische Synthese des VHDL-Designs und das Einlesen des synthetisierten Designs (bzw. des Benchmark-Designs im Net-Format) durch. Anschließend wird das gelesene Design iterativ geclustert, und die hierbei neu erzeugten Designs werden in eine Text-Datei geschrieben.

Sobald dies geschehen ist, wird das Partitionierungsprogramm auf den Workstations, die der Benutzer spezifiziert hat, gestartet. Sobald der parallele genetische Algorithmus terminiert, werden alle beteiligten Prozesse beendet.

cobra liest die beste gefundene Partition ein, schreibt das partitionierte Design in XNF-Files und ruft abschließend SYNOPSYS einmal auf.

partition: Der genetische Partitionierungsalgorithmus, der in Kapitel 5 beschrieben wurde, wird durch dieses Programm realisiert, das gleichzeitig auf mehreren Rechnern gestartet werden kann.

Unter allen parallel gestarteten Partitionierungsprozessen gibt es genau einen sog. Master-Prozeß, der die anfallenden Berechnungen an die übrigen sog. Slave-Prozesse delegiert.

Zunächst werden die Designs, die von der iterativen Clusterung berechnet wurden, sowie die für die Partitionierung relevante Zieltechnologie aus der Eingabe-Datei gelesen. Aus ungeklärten Gründen kann dieser Datei-Zugriff nur von dem Master-Prozeß ausgeführt werden. Da die Slave-Prozesse diese Daten jedoch ebenfalls benötigen, sind diese mittels Broadcast-Nachrichten über das Netzwerk weiterzuleiten.

Nacheinander werden diese Designs nun partitioniert, wobei die jeweils beste gefundene Partition wie in Kapitel 3 erläutert von einem Durchlauf zum nächsten übertragen wird. Während der parallelen Partitionierung hat der Master-Prozeß die Aufgabe, die Populationen zu verwalten und Selektion, Mutation und Crossover durchzuführen. Lediglich die Bewertung der neu erzeugten Individuen wird von den Slave-Prozessen vorgenommen, indem die betreffenden Chromosomen über das Netzwerk übertragen werden.

Vor jedem Durchlauf des genetischen Algorithmus wird der FM-Algorithmus mehrere Male vom Master-Prozeß aufgerufen, wobei nach dem in Abschnitt 5.2 vorgestellten Aufruf-Schema vorgegangen wird.

Der Partitionierungsalgorithmus arbeitet – sofern der Anwender über Kommandozeilen-Optionen keine anderen Werte angibt – mit den folgenden Einstellungen:

- Anzahl der Aufrufe des FM-Algorithmus und des Algorithmus von Krishnamurthy vor jedem Genetik-Durchlauf: 20
- Populationsgröße: 100 Individuen
- Ersetzungsrate von Generation zu Generation: 50 % der Populationsgröße
- Abbruch des genetischen Algorithmus, wenn
  - eine maximale Anzahl von Populationen berechnet wurde, oder
  - während einer bestimmten Anzahl von Generationen keine weitere Verbesserung der bisher besten Lösung gefunden wurde.
- Maximale Anzahl zu berechnender Populationen für das Abbruch-Kriterium: 1000
- Maximale Anzahl von Generationen ohne Verbesserung: 150

partition terminiert, indem die beste berechnete Partition in eine Datei geschrieben und somit an cobra übergeben wird, oder indem eine Fehlermeldung ausgegeben wird, wenn keine gültige Lösung gefunden wurde.

klfm: Eine Implementierung des Algorithmus von Fiduccia & Mattheyses ([FiMa82]) ist durch dieses Programm gegeben. Ausgehend von einem zu partitionierenden Design und einer evtl. vorgegebenen initialen Startlösung wird nach dem in Kapitel 4 beschriebenen Verfahren eine Bipartition berechnet und an partition übergeben.

strawman: Dieses Programm stellt die von COBRA verwendete Implementierung des Algorithmus von Krishnamurthy ([Kris84]) dar. Die Länge der Gewinn-Vektoren, die für die Zellen des Designs verwaltet werden, ist auf drei Elemente begrenzt.

6.3 Die Partitionierung eines konkreten Fallbeispiels

Um die Funktionsfähigkeit der implementierten Partitionierungssoftware zu überprüfen, wurde ein Design, das mit Hilfe von COOL entworfen wurde, synthetisiert und partitioniert. Da es aus technischen und zeitlichen Gründen nicht möglich war, das von COBRA berechnete partitionierte Design auf einem konkreten FPGA-Board zu realisieren, wurde die Korrektheit der Schaltung mittels mehrerer Simulationsdurchläufe gezeigt.

In dem folgenden Abschnitt 6.3.1 wird das Design vorgestellt, das als Fallbeispiel herangezogen wurde. Anschließend wird auf die von COBRA bestimmte Partitionierung eingegangen. Den Abschluß bilden die Vorstellung der Simulationsumgebung und der Simulationsergebnisse in Abschnitt 6.3.3, die sich für das partitionierte Design ergaben.

6.3.1 Ein Fuzzy-Controller zur Ampelsteuerung

Als Fallbeispiel zur Logik-Partitionierung wird im folgenden ein Fuzzy-Controller betrachtet, der die Steuerung der Ampelanlage an einer Kreuzung übernimmt. Die Struktur der Kreuzung ist aus Abbildung 6.4 ersichtlich. Dargestellt ist die Kreuzung einer Haupt- mit einer Nebenstraße. Die Nebenstraße hat dabei in jeder Richtung eine Fahrspur, während die Hauptstraße zusätzlich eine Linksabbiegerspur hat. Rechtsabbieger müssen auch auf der Hauptstraße die Geradeausspur benutzen. Weiterhin soll die Kreuzung auch für Fußgänger passierbar sein.

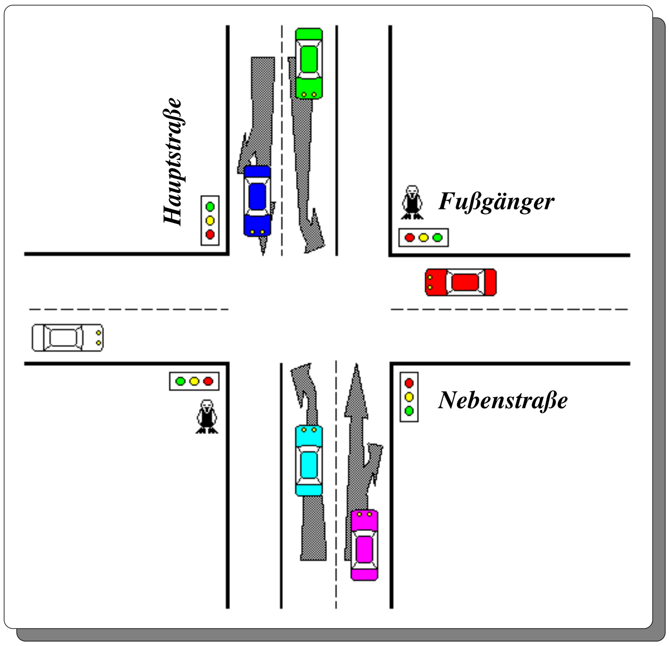


Abbildung 6.4: Kreuzung mit zu regelnder Ampelanlage

Für jede einzelne Spur (also Linksabbiegerspur Hauptstraße, Geradeausspur Hauptstraße und Nebenstraße) wird die Anzahl von Fahrzeugen bestimmt, die sich zur Zeit auf dieser Spur befinden. Zusätzlich wird die Zeit gemessen, die seit der letzten Grünphase einer Spur vergangen ist. Schließlich ist festzustellen, ob Fußgänger für die Fußgänger-Überwege eine Grünphase angefordert haben.

Anhand dieser Daten soll die Ampelsteuerung diejenige Spur ermitteln, die als nächstes Grün erhält.

Entsprechend der üblichen Funktionsweise von Fuzzy-Reglern (siehe hierzu auch [KaFr93]) wird in [PG293b] ein Controller präsentiert, bei dem die gemessenen Werte für jede einzelne Spur zunächst die Fuzzifizierung, danach die Inferenz und schließlich die Defuzzifizierung durchlaufen. Die für jede Spur berechneten Werte werden in einem letzen Defuzzifizierungsschritt zu einer Ausgabe des Fuzzy-Controllers miteinander verknüpft. Fuzzifizierung, Inferenz und Defuzzifizierung für jede einzelne Spur der Kreuzung wurden mit COOL in Form einer oder mehrerer VHDL-Komponenten spezifiziert. Der gesamte Controller – als strukturelle VHDL-Komponente betrachtet – hat somit den in Abbildung 6.5 dargestellten Aufbau.

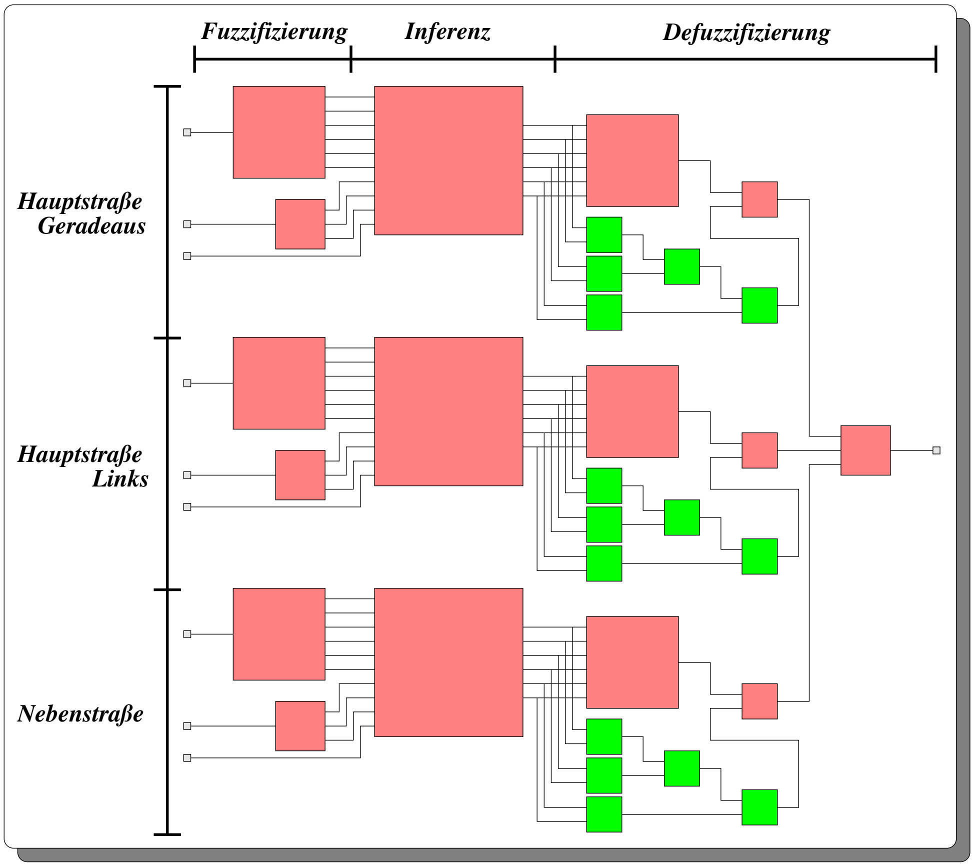


Abbildung 6.5: Struktur des Fuzzy-Controllers zur Ampelsteuerung

In [PG293b] wird beschrieben, wie dieser Fuzzy-Controller auf einer konkreten Zieltechnologie – bestehend aus zwei FPGAs, einem DSP und einer gemeinsamen Speicherkarte – als eingebettetes System realisiert werden konnte.[2] Hierzu

[2]Während der Arbeit an dem Controller wurde das Design manuell partitioniert, so daß die Zieltechnologie nicht geändert werden mußte. Für die Realisierung einer von COBRA berechneten Partition auf dieser Hardware-Plattform wären jedoch Modifikationen an der Verdrahtung der FPGAs vorzunehmen, für die während dieser Diplomarbeit die Zeit fehlte.

berechnete COOL eine Aufteilung aller VHDL-Komponenten auf den DSP und die FPGAs. Die Komponenten, die in Abbildung 6.5 rot hinterlegt sind, werden dieser Aufteilung zufolge mit Hilfe des DSPs berechnet. Alle übrigen Komponenten werden durch die Hardware der FPGAs realisiert.

Allen diesen FPGA-Komponenten ist gemein, daß sie eine Addition auf 16-Bit-Zahlen berechnen. Der Hardware-Teil dieses Fuzzy-Controllers enthält jedoch nicht nur diese Addierer, sondern noch zusätzliche Komponenten. Diese werden automatisch von COOL erzeugt und dienen im wesentlichen der Ablauf-Steuerung zwischen FPGAs und DSP. Im einzelnen sind dies:

System-Steuerung: Die System-Steuerung stellt einen sogenannten *Laufzeit-Scheduler* dar. Diese Komponente ist die Instanz, die das gesamte System kontrolliert und steuert. Ihre Aufgabe ist es, zum richtigen Zeitpunkt Rechenoperationen auf dem DSP oder auf einem FPGA entsprechend eines von COOL berechneten Schedulings zu starten. Die System-Steuerung ist in Form eines endlichen Automaten beschrieben.

I/O-Controller: Diese Komponente überwacht, ob neue Eingabe-Daten für die Ampelsteuerung (also neue Autoanzahlen, Wartezeiten oder Fußgänger-Anforderungen für alle einzelnen Spuren) vorliegen. Ist dies der Fall, so wird die System-Steuerung gestartet, woraufhin der Fuzzy-Controller einen neuen Zustand der Ampelanlage auf Basis der neuen Daten berechnet.

Bus-Scheduler: DSP, FPGAs und Speicherkarte sind über einen gemeinsamen Bus miteinander verbunden. Damit nun nicht mehrere Komponenten des Fuzzy-Controllers gleichzeitig auf diesen gemeinsamen Bus zugreifen, wird als zentrale Instanz zur Bus-Zuweisung der Bus-Scheduler erzeugt. Ehe irgend eine Komponente (also System-Steuerung, Addierer oder DSP) Signale an den Bus anlegen darf, muß diese den Bus von dem Bus-Scheduler anfordern und zugewiesen bekommen.

Datenpfad-Controller: Da die verwendeten FPGAs bloß über eine geringe Anzahl von CLBs verfügten, wurden von COOL nicht 15 verschiedene Addierer generiert sondern nur ein einzelner. Dieser Addierer (auch Datenpfad genannt) wird von einem zusätzlich erzeugten Controller gesteuert, der in Abhängigkeit vom Zustand des gesamten Systems die Operanden aus verschiedenen Speicher-Adressen lädt und an das eigentliche Rechenwerk leitet, so daß stets genau eine der insgesamt 15 Additionen ausgeführt wird.

Alle von COOL erzeugten Komponenten werden letztlich in einer sog. VHDL-Netzliste instantiiert und untereinander verbunden.[3] Diese Netzliste enthält somit die komplette strukturelle Beschreibung desjenigen Teils des Fuzzy-Controllers, der in die Hardware-Synthese und Logik-Partitionierung eingehen

[3] Die in diesem Abschnitt anzutreffenden Ausführungen können natürlich nur einen äußerst oberflächlichen Einblick in die Konzepte von COOL vermitteln. Der an HW/SW-Codesign, -Partitionierung und -Cosynthese interessierte Leser sei vielmehr auf [Niem98] verwiesen.

soll. Die Ports dieser Netzliste bestehen aus allen Signalen des gemeinsamen Busses, an den neben den FPGAs auch DSP und Speicher angeschlossen sind. Der schematische Aufbau des gesamten Hardware/Software-Systems wird in Abbildung 6.6 dargestellt.

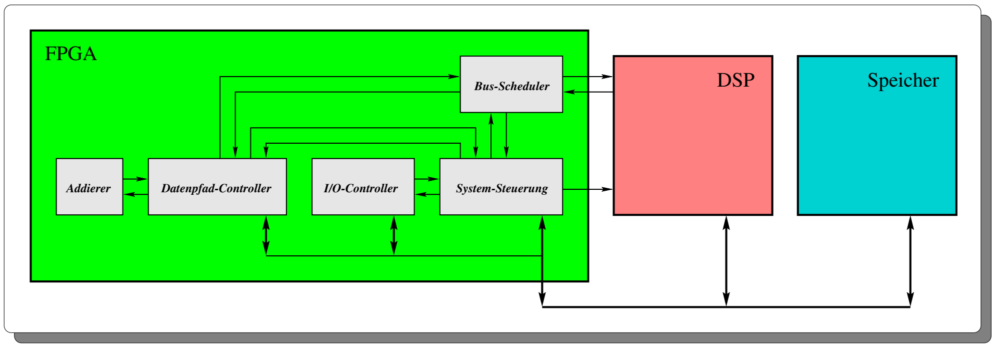


Abbildung 6.6: Der Fuzzy-Controller als Hardware/Software-System

6.3.2 Synthese und Bipartitionierung des Controllers mit Cobra

Die von Cool erzeugte VHDL-Netzliste wurde wie am Anfang dieses Kapitels beschrieben synthetisiert, wobei ausnahmsweise Oscar nicht eingesetzt wurde. Als Takt-Periode wurde eine Zyklus-Zeit von 120 ns zugrunde gelegt.

Das resultierende XNF-File wurde durch Cobra für eine Zieltechnologie partitioniert, die aus zwei FPGAs vom Typ XC 4005pc84 bestand. Diese beiden FPGAs waren so angeordnet, daß jedes einzelne über 17 externe Anschlüsse verfügte, auf die die Ports der VHDL-Netzliste abgebildet werden konnten. Untereinander waren die FPGAs über 44 Leitungen verbunden.

Nach Abschluß der Partitionierung wurden die CLBs und IOBs des partitionierten Designs mit Hilfe der XAct-Programme auf den FPGAs plaziert und unter Berücksichtigung der XNF-Timing Constraints miteinander verbunden.

Die von Cobra berechnete Partition des Hardware-Teils des Fuzzy-Controllers weist die folgenden charakteristischen Eigenschaften auf:

Pin-Belegungen der FPGAs: Von den insgesamt 32 Ports des zu partitionierenden Designs wuden 17 auf das erste FPGA und die restlichen 15 auf das zweite FPGA abgebildet.

Die 44 Verbindungen zwischen beiden FPGAs wurden in vollem Umfang mit geschnittenen Netzen belegt.

CLB-Auslastungen: Auf das erste FPGA wurden 196 CLBs des Designs abgebildet, was zur Folge hatte, daß dieser Chip zu Hundert Prozent ausgelastet war. Die zweite Partition enthielt die verbleibenden 19 CLBs der Ampelsteuerung.

Verteilung der Logik auf die Partitionen: Aufgrund der Bezeichnungen der geschnittenen Netze können Rückschlüsse darauf gezogen werden, wie die einzelnen VHDL-Komponenten ungefähr auf die beiden FPGAs verteilt worden sind.

Interessant ist in diesem Hinblick, daß das zweite FPGA keinerlei Speicherelemente enthält, die getaktet werden müssen. Sämtliche CLBs des Designs, die Flip-Flops enthalten, sind auf FPGA 1 abgebildet worden, woraus geschlossen werden kann, daß die Teile der von COOL generierten endlichen Automaten, die die Zustände der System-Steuerung, des DSPs und des Addierers überwachen, komplett in FPGA 1 enthalten sind. Der Addierer selbst, der in Form eines XBLOX-Makros realisiert ist, befindet sich ebenfalls auf diesem Chip.

Das zweite FPGA enthält lediglich kleine Teile rein sequentieller Logik, die zur Verbindung des ersten FPGAs mit denjenigen Ports des Designs dienen, für die auf FPGA 1 kein Platz mehr war.

So enthält bspw. das erste FPGA diejenigen Flip-Flops, die den Zustand des DSPs codieren. Dieser Zustandsvektor wird als Eingabe an das zweite FPGA geleitet, auf dem dann eine boolesche Funktion einen Wert für die Interrupt-Leitung des DSPs ermittelt.

6.3.3 Simulationsergebnisse für den partitionierten Controller

Nach dem in [Xili94] beschriebenen Verfahren sind die von XACT erzeugten und mit konkreten Timing-Informationen versehenen Beschreibungen für beide FPGAs wieder in strukturelles VHDL überführt worden. Dieses kann schließlich für eine Simulation mit dem SYNOPSYS-Simulator verwendet werden, wobei für die Simulation das gleiche Timing zugrunde gelegt wird, das auch auf den programmierten FPGAs auftreten würde.

Für die Simulation ist eine Simulationsumgebung erstellt worden, die die fehlenden Teile des Fuzzy-Controllers sowie weitere Hardware-Bauteile emuliert. Diese Simulationsumgebung enthält Prozesse für

- die Emulation des Teils des Fuzzy-Controllers, der auf dem DSP ausgeführt wird (wobei für die Simulation kein Wert auf korrekte Berechnungen von Fuzzifizierung, Inferenz und Defuzzifizierung gelegt wurde. Wichtig war lediglich, daß dieses DSP-Modell richtig mit der System-Steuerung auf den FPGAs interagiert),
- die Simulation der Speicherkarte,
- das Zurücksetzen und Starten des gesamten Systems,
- die Erzeugung eines Taktes mit einer Periode von 120 ns sowie für
- die Beschreibung einiger Pull-Up-Widerstände, ohne die die Simulation nicht funktionieren würde.

Die Simulation des partitionierten Designs im Rahmen der beschriebenen Simulationsumgebung lieferte die in Abbildung 6.7 dargestellten Signalverläufe.

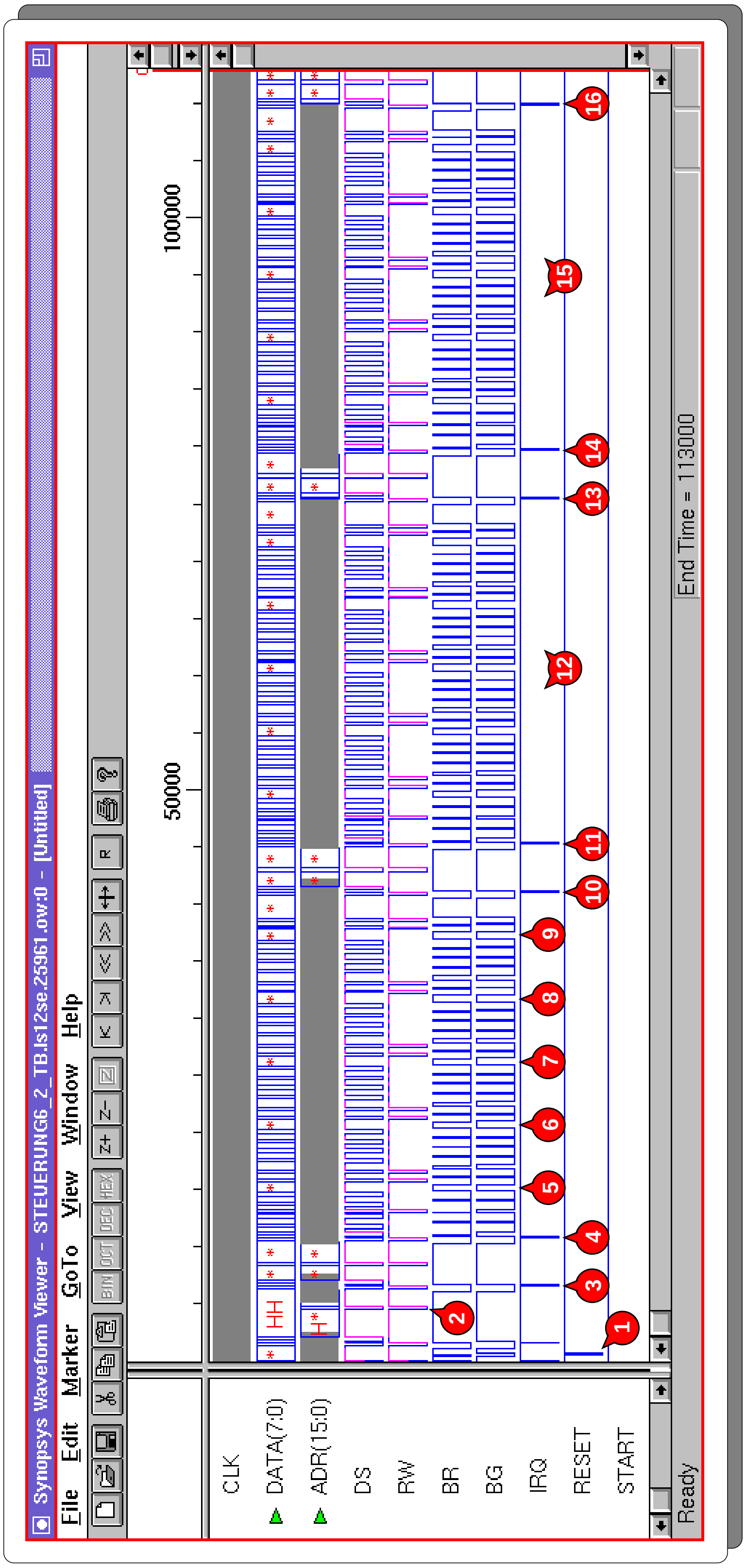


Abbildung 6.7: Simulationsergebnisse für den partitionierten Fuzzy-Controller

Anhand der in dieser Abbildung markierten Stellen ist erkennbar, daß das Design in der Simulation nach dem folgenden Schema abläuft:

1 - Über einen kurzen Impuls durch die Simulationsumgebung auf der RESET-Leitung wird der Fuzzy-Controller zurückgesetzt.

2 - Die Ampelsteuerung wird gestartet, indem durch die Simulationsumgebung schreibend auf die Speicheradresse $DFFF zugegriffen wird.

3 - Die System-Steuerung startet den DSP durch einen Interrupt, worauf dieser Fuzzifizierung und Inferenz aller Daten der Geradeausspur der Hauptstraße durchführt.

4 - DSP und der Addierer auf den FPGAs werden von der System-Steuerung gleichzeitig gestartet. So werden nebenläufig die Teile der Defuzzifizierung für die Geradeausspur ausgeführt, die parallel arbeiten können.

5 - Die erste 16-Bit-Addition wird durch die FPGAs ausgeführt.

6 - Die zweite 16-Bit-Addition wird durch die FPGAs ausgeführt.

7 - Die dritte 16-Bit-Addition wird durch die FPGAs ausgeführt.

8 - Die vierte 16-Bit-Addition wird durch die FPGAs ausgeführt.

9 - Die fünfte 16-Bit-Addition wird durch die FPGAs ausgeführt.

10 - Die zuletzt vom DSP und den Addierern berechneten Werte gehen in den letzten Defuzzifizierungsschritt für die Geradeausspur der Hauptstraße ein, der vom DSP auszuführen ist.

    Nach Abschluß der Berechnungen für die Geradeausspur wird unmittelbar mit Fuzzifizierung und Inferenz für die Linksabbiegerspur durch den DSP fortgefahren.

11 - Die parallelen Teile der Defuzzifizierung für die Linksabbiegerspur werden auf dem DSP und den FPGAs gestartet.

12 - Der Addierer auf den FPGAs führt nacheinander die fünf 16-Bit-Additionen durch.

13 - Die Teilergebnisse der Defuzzifizierung werden vom DSP zu einem Endergebnis für die Linksabbiegerspur der Hauptstraße zusammengefaßt.

    Die Fuzzifizierung und Inferenz der Eingabedaten für die Nebenstraße wird anschließend vom DSP ausgeführt.

14 - Wie für die Hauptstraße wird auch für die Nebenstraße der erste Teil der Defuzzifizierung sowohl auf dem DSP als auch auf den FPGAs gestartet.

15 - Der Addierer führt fünf Additionen auf Daten der Nebenstraße aus.

16 - Die Defuzzifizierungsergebnisse für die Nebenstraße werden vom DSP miteinander verknüpft.

Abschließend werden die Endergebnisse der Defuzzifizierung für alle drei Spuren der Kreuzung vom DSP miteinander verglichen, und ein neuer Ampel-Zustand wird berechnet.

Die Simulation endet nach 113000 ns.

Dieser Ablauf der Simulation zeigt, daß der partitionierte Hardware-Teil des Fuzzy-Controllers korrekt arbeitet. Insbesondere

- werden sowohl der DSP als auch die Addierer-Komponente auf den FPGAs zu den richtigen Zeitpunkten gestartet,
- erkennt die System-Steuerung die Beendigung einer Operation auf DSP und FPGAs richtig, und
- verarbeitet der Datenpfad-Controller stets die richtigen Operanden, wobei die Additionsergebnisse vom XBlox-Makro korrekt berechnet werden (was aus Abbildung 6.7 aufgrund des hohen Vergrößerungsfaktors nicht hervorgeht).

Fazit dieses Abschnittes ist somit, daß die von Cobra berechnete Partition und deren Repräsentation durch mit Timing Constraints versehenen XNF-Files äquivalent zum ursprünglichen unpartitionierten Design ist. Der Vollständigkeit halber soll an dieser Stelle auch das nicht weiter überraschende Ergebnis erwähnt werden, daß das partitionierte Design auch bei einer niedrigeren Takt-Frequenz als ursprünglich vorgesehen funktioniert.[4] Die Reduktion der Takt-Periode kann also als Mittel zur Umgehung von Timing-Problemen betrachtet werden, wie dies in Abschnitt 6.1.1 diskutiert wurde.

6.4 Benchmark-Partitionierungen

Zur Bestimmung der Güte des genetischen Algorithmus zur Logik-Partitionierung wird in dieser Arbeit auf die sog. MCNC-Benchmark-Serie zurückgegriffen. Eine Übersicht über die Größe dieser Designs ist in Tabelle 6.1 gegeben. Die mit * markierten Designs sind über [CBL97] in XNF-Form zu beziehen, während die restlichen über [Alpe98] im Net-Format zur Verfügung gestellt werden.

Name	# Gatter	# Netze	Name	# Gatter	# Netze
balu	801	735	s1423*	831	757
prim1	833	902	bm1	882	903
test04	1515	1658	test03	1607	1618
test02	1663	1720	test06	1752	1641
struct	1952	1920	test05	2595	2750

[4] Für die gleiche Partition des Fuzzy-Controllers wie in der ersten Simulation wurde in diesem Versuch eine Takt-Periode von 200 ns zugrunde gelegt. Die Signalverläufe änderten sich im Vergleich zu Abbildung 6.7 nicht, bis auf den Umstand, daß die Simulation nun erst nach 182000 ns beendet war.

Name	# Gatter	# Netze	Name	# Gatter	# Netze
19ks	2844	3282	prim2	3014	3029
s5378*	3225	3046	s9234*	5866	5844
biomed	6514	5742	s13207*	8772	8651
s15850*	10470	10474	industry2	12637	13419
industry3	15406	21923	s35932*	18148	17828
s38584	20995	20717	avq.small	21918	22124
s38417	23949	23843	avq.large	25178	25384

Tabelle 6.1: Größen von Benchmark-Designs der MCNC-Serie

Diese Designs sind in den Jahren von 1985 bis 1993 gesammelt worden. Sie gingen ursprünglich aus konkreten synthetisierten Schaltungen hervor und sind zum Zweck der Veröffentlichung als Benchmarks in das XNF- bzw. Net-Format transformiert worden. Die enthaltenen Zellen der Designs stellen grundsätzlich Gatter dar.

Auf Grundlage dieser Schaltkreise wurden zwei verschiedene Arten der Partitionierung mit COBRA durchgeführt, deren Ergebnisse in den Abschnitten 6.4.1 und 6.4.2 dokumentiert sind:

1. Zur Klärung der prinzipiellen Frage, inwieweit sich das Technology Mapping auf die Ergebnisse der Logik-Partitionierung auswirkt, wurden einige der Benchmarks auf CLB-Ebene überführt und anschließend partitioniert.

2. Alle Designs wurden unter Verwendung fixer Kostenmaße und Balance-Kriterien bipartitioniert, um Vergleichsmöglichkeiten mit anderen Partitionierungsverfahren zu erhalten.

6.4.1 Auswirkungen des Technology Mappings auf die Logik-Partitionierung

Da die in Tabelle 6.1 mit * markierten Designs in XNF-Format vorliegen, war es ohne weiteres möglich, für diese mit Hilfe von SYNOPSYS das Technology Mapping für XILINX-FPGAs der Serie XC 4000 durchzuführen. Für die sechs an dieser Stelle betrachteten Benchmarks ergab sich durch das Technology Mapping eine durchschnittliche Reduktion der Zellen-Anzahlen auf 7,8 % der ursprünglichen Anzahlen, wie anhand von Tabelle 6.2 nachvollzogen werden kann.

Benchmark	Original		Tech-Mapped		Reduktion
	# Gatter	# Netze	# Zellen	# Netze	
s1423	831	757	102	165	12,3 %
s5378	3225	3046	332	561	10,3 %
s9234	5866	5844	177	343	3 %
s13207	8772	8651	600	1092	6,8 %
s15850	10470	10474	623	1091	5,9 %

Benchmark	Original		Tech-Mapped		Reduktion
	# Gatter	# Netze	# Zellen	# Netze	
s35932	18148	17828	1585	2400	8,7 %
Ø					**7,8 %**

Tabelle 6.2: Größenreduktion von Benchmarks durch Technology Mapping

Zum Vergleich von Werten, die sich für Partitionierungen auf Gatter- und auf CLB-Ebene ergaben, wurde jedes Design auf beiden Ebenen jeweils zehnmal bipartitioniert. Für die Partitionierungen galt ein Balance-Kriterium, das minimale Partitionsgrößen von 45 Prozent der Logik zuließ, wobei als Kostenmaß für Gatter der konstante Wert 1 verwendet wurde. In der folgenden Tabelle 6.3 sind die Größen der Cutsets gegenübergestellt, die sich für alle Partitionierungen ergaben. In diese Tabelle wurde jeweils der beste ermittelte Wert für alle zehn Aufrufe des Partitionierers übernommen.

Benchmark	Gatter-Eb.	CLB-Eb.	Vergrößerung
s1423	14	33	236 %
s5378	58	154	266 %
s9234	40	87	217 %
s13207	53	193	364 %
s15850	42	160	381 %
s35932	43	514	1195 %
Ø			**443 %**

Tabelle 6.3: Cutsizes bei Partitionierung auf Gatter- und CLB-Ebene

An dieser Stelle bestätigt sich die Vermutung, daß bei einer Partitionierung auf CLB-Ebene die Anzahl der geschnittenen Netze drastisch ansteigt. Die Partitionierungen der Benchmarks auf CLB-Ebene lieferten Ergebnisse, deren Cutsizes durchschnittlich um das 4,4-fache größer waren als die der Partitionierungen auf Gatter-Ebene.

Nicht weiter überraschend ist der Umstand, daß die Laufzeiten für die Partitionierungen in dem Maße abnahmen, in dem durch das Technology Mapping die Zellen-Anzahlen der Designs reduziert wurden. Alle Partitionierungen auf CLB-Ebene wurden von Cobra innerhalb eines halben Tages berechnet.

Aus den in Tabelle 6.3 dokumentierten Ergebnissen läßt sich ableiten, daß sich eine Partitionierung auf Gatter-Ebene um so mehr lohnt, je größer die zu partitionierenden Designs werden. Offenbar gilt, daß sich dem Technology Mapper mit zunehmender Design-Größe wesentlich mehr Möglichkeiten bieten, einzelne Gatter zu Funktionsgeneratoren und CLBs zu gruppieren. Synopsys scheint diese Möglichkeiten zu nutzen, indem häufig Gatter aus völlig verschiedenen Bereichen des Designs in einem CLB zusammengefaßt werden, wodurch die ermittelten Cutsizes dermaßen zunehmen.

Aufgrund dieser Feststellungen kann abschließend festgehalten werden, daß für die Aufteilung konkreter Hardware-Systeme für eine FPGA-Zieltechnologie die Partitionierung auf Gatter-Ebene erfolgsversprechender ist. Die längeren Laufzeiten werden bei dieser Vorgehensweise durch die wesentlich besseren Resultate kompensiert. Der Umstand, daß COBRA eine möglichst gleichmäßige Balancierung der Partitionsgrößen anstrebt, reduziert zudem die Ungenauigkeiten, die aufgrund der inadäquaten Kostenmaße für Gatter auftreten können.

6.4.2 Partitionierungsergebnisse von COBRA und anderen Verfahren

Um den in dieser Arbeit entworfenen Partitionierungsalgorithmus mit anderen Verfahren zu vergleichen, wurden alle in Tabelle 6.1 aufgeführten Benchmarks zehnmal auf Gatter-Ebene bipartitioniert. Aus allen zehn für ein Design ermittelten Cutsizes wird der kleinste Wert als Endergebnis betrachtet. Als Balance-Kriterium wurde während aller Versuchsreihen eine minimale Partitionsgröße von 45 % der gesamten Logik verwendet. Die Input/Output-Blocks der Designs wurden mit Kosten 0, alle übrigen Zellen mit Kosten 1 bewertet.

In diesem Abschnitt wird COBRA mit den Verfahren FM ([FiMa82]), EIG 1 ([HaKa91]), Paraboli ([RDJ94]), MELO ([AlYa95]), FBB ([YaWo94]), PROP ([DuDe96]) und Hauck ([HaBo95]) zur Logik-Partitionierung verglichen, die z. T. in den Kapiteln 2 und 4 beschrieben wurden. Die nachstehende Tabelle 6.4 enthält nun die besten Cutsizes für alle Verfahren und Benchmarks (die besten bisher bekannten Cutsizes zu den Benchmarks sind kursiv hervorgehoben):

Benchmark	FM	EIG 1	Para	MELO	FBB	PROP	Hauck	COBRA
balu	30	110	41	28		*27*	*27*	**27**
s1423	19	23	16		*13*		14	**14**
prim1	56	75	53	64		*47*	49	**47**
bm1	58	75		*48*		50	49	**48**
test04	124	207		61		52	*48*	**48**
test03	87	85		60		59	*55*	**57**
test02	134	196		106		*90*	93	**87**
test06	77	295		90		76	*60*	**60**
struct	45	49	40	38		*33*	*33*	**33**
test05	124	167		102		79	*72*	**71**
19ks	130	179		119		*105*	112	**104**
prim2	249	254	146	169		*143*	*143*	**139**
s5378	102						*59*	**58**
s9234	70	166	74	79	70	*41*	42	**40**
biomed	135	286	135	115		*83*	*83*	**83**
s13207	108	110	91	104	74	75	*57*	**53**
s15850	170	125	91	52	67	65	*44*	**42**
industry2	705	525	193	319		220	*188*	**174**
industry3	377	399	267				*256*	**241**

Benchmark	FM	EIG 1	Para	MELO	FBB	PROP	Hauck	COBRA
s35932	162	105	62		49		*47*	**43**
s38584	168	76	55		*47*		49	**47**
avq.small	499	598	224				*131*	**127**
s38417	419	121	*49*		58		53	**49**
avq.large	431	571	*139*				140	**127**

Tabelle 6.4: Cutsizes von Bipartitionierungen der Benchmark-Designs

Die Zahlen dieser Tabelle zeigen, daß aus der Kombination des Algorithmus von Krishnamurthy mit einem genetischen Algorithmus ein leistungsfähiges Partitionierungsverfahren hervorgegangen ist, das die besten bisher bekannten Ergebnisse erreicht und teilweise sogar verbessert.

Dabei ist zu berücksichtigen, daß für diese Partitionierungen der Benchmarks die Leistungsfähigkeit von COBRA nur teilweise ausgeschöpft wurde. So wurden z. B. die Fähigkeiten zur Zerlegung der Designs in mehr als zwei Partitionen nicht in Anspruch genommen, da sich die Forschung bisher fast ausschließlich auf die reine Bipartitionierung konzentriert hat und somit zu diesen Benchmark-Designs kein Zahlenmaterial für Mehrfach-Partitionierungen existiert.

7 – Schließende Betrachtungen

„Mir wird von alledem so dumm,
Als ging' mir ein Mühlrad im Kopf herum."

Aus: J.W. Goethe, „Faust, der Tragödie erster Teil".

In diesem letzten Kapitel der vorliegenden Diplomarbeit soll eine Rückschau und eine Vorschau auf die Entwicklung des genetischen Partitionierungsalgorithmus von COBRA gegeben werden. Zu diesem Zweck wird zunächst in Abschnitt 7.1 kurz der entworfene Algorithmus und dessen Leistungsmerkmale skizziert sowie dessen Leistungsfähigkeit beurteilt. Im folgenden Abschnitt 7.2 werden einige Punkte angesprochen, um die die Partitionierungssoftware von COBRA erweitert werden kann, so daß sich unter Umständen bessere Ergebnisse und Laufzeiten ergeben.

7.1 Was wurde bisher erreicht?

In dieser Diplomarbeit wurde die Partitionierungssoftware COBRA entwickelt, mit deren Hilfe VHDL-Beschreibungen von Hardware-Systemen auf FPGAs des Herstellers XILINX partitioniert werden können. Die Synthese dieser hochsprachlichen Beschreibungen in Technologie-Komponenten der zugrunde liegenden FPGAs wird automatisch vorgenommen.

Der Partitionierungsalgorithmus ist im Gegensatz zu den meisten bisher präsentierten Verfahren in der Lage, Hardware-Beschreibungen für beliebig viele FPGAs zu partitionieren. Zudem können beliebige Topologien von FPGA-Boards (also die FPGAs mitsamt ihren elektrischen Verbindungen auf der Platine) spezifiziert werden, die dann entsprechend von COBRA berücksichtigt werden. Schließlich kann der Benutzer die Partitionierung dahingehend einschränken, daß Ports des zu partitionierenden Designs dauerhaft bestimmten FPGAs zugewiesen werden.

Der Partitionierer besteht im wesentlichen aus den folgenden drei Komponenten:

- Der Kern des Partitionierungsverfahrens ist ein genetischer Algorithmus, der durch Mutation und Crossover erzeugte Partitionen bewertet und diese zu optimieren trachtet. Hierin enthalten ist zudem die Funktionalität zur Partitionierung für Board-Topologien.

  Der genetische Algorithmus kann zur Laufzeit-Reduktion parallel auf mehreren Rechnern ausgeführt werden.

- Der Algorithmus von Krishnamurthy ([Kris84]) dient zur Erzeugung einer Reihe von Startlösungen für den genetischen Algorithmus. Dieses Standard-Verfahren zur Bipartitionierung (Partitionierung für zwei FPGAs) wurde um ein Schema erweitert, das die Aufteilung in beliebig viele Partitionen ermöglicht.

- Vor der eigentlichen Partitionierung wird eine Connectivity-basierte Clusterung des zu partitionierenden Designs durchgeführt. Hiermit wird erreicht, daß Zellen, die sehr stark untereinander vernetzt sind, nicht während der Partitionierung voneinander getrennnt werden.

Zu Beginn der Arbeit ist davon ausgegangen worden, daß die Logik-Partitionierung auf CLB-Ebene vorgenommen werden sollte, da auf diese Weise die Kosten der Zellen genau berechnet werden können. Rückblickend läßt sich feststellen, daß die Wahl dieses Ansatzes falsch war, da die Qualität von Partitionen auf CLB-Ebene zu wünschen übrig läßt. Dieser methodische Fehler hat jedoch keine Auswirkungen auf den Algorithmus von COBRA, da dieser auch in der Lage ist, auf Gatter-Ebene mit benutzerdefinierten Kostenmaßen zu partitionieren.

Experimentelle Partitionierungen der MCNC-Benchmark-Serie haben gezeigt, daß der in dieser Arbeit präsentierte Algorithmus mit anderen Methoden aus dem Bereich der Logik-Partitionierung durchaus ebenbürtig ist. Es stellte sich heraus, daß COBRA für dreizehn der 24 Benchmarks bessere Ergebnisse lieferte als bisher bekannt waren. In neun Fällen wurden gleich gute Resultate ermittelt, während bei nur zwei Designs ein schlechteres Ergebnis erzielt wurde.

Die große Schwäche von COBRA, die die Anwendbarkeit der Software vermindert, liegt in den Laufzeiten zur Partitionierung. So wurden bspw. für eine einzelne Partitionierung des größten Benchmark-Designs (avq.large mit 25178 Gattern) 7:25 Stunden benötigt, während z. B. in [HaBo95] die Laufzeit für die gleiche Partitionierung mit 1:50 Minuten angegeben ist.

Für den reinen genetischen Partitionierungsalgorithmus spricht andererseits, daß die Laufzeiten der Bewertungsfunktion nicht von der Anzahl von FPGAs, für die partitioniert werden soll, abhängt. Konkret bedeutet dies, daß die Genetik für die Achtfach-Partitionierung eines Designs die gleiche Rechenzeit benötigt wie für eine Zweifach-Partitionierung. Soll aber eine Achtfach-Partitionierung mit Hilfe eines reinen Bipartitionierers durchgeführt werden, so sind sieben voneinander unabhängige Aufrufe dieses Bipartitionierers notwendig, so daß sich bei dieser Methode eine um das siebenfache größere Laufzeit ergibt. Mit zunehmender Komplexität der Zieltechnologien[1] ändert sich also das Laufzeit-Verhältnis zwischen Bipartitionierungs- und genetischer Methode zugunsten der Genetik.

Trotz dieser großen Laufzeiten ist COBRA nicht dasjenige Glied in der Kette der Werkzeuge zur Hardware-Synthese, das am meisten Zeit benötigt. Erfahrungsgemäß dürfte die Logik-Synthese von SYNOPSYS die meiste Rechenzeit in Anspruch nehmen. Wenn man das im Rahmen dieser Diplomarbeit immer wieder herangezogene Beispiel des Fuzzy-Controllers zur Ampelsteuerung betrachtet (1341 Gatter), so ergibt sich eine Laufzeit von etwa $1\frac{1}{2}$ Stunden zur Logik-Synthese, der zwanzig Minuten zur Partitionierung gegenüber stehen.

7.2 Was kann verbessert und erweitert werden?

Auch wenn die Ergebnisse dieser Diplomarbeit überwiegend positiv sind, bleibt dennoch viel Raum, um die Leistungsfähigkeit von COBRA weiter zu bewerten und auszubauen. Vor diesem Hintergrund werden im folgenden einige Punkte kurz andiskutiert.

Mehrfach-Partitionierungen von Benchmarks

Zur Zeit ist es nicht möglich, Aussagen über die Qualitäten von COBRA als Mehrfach-Partitionierer zu treffen, da es für Zerlegungen der MCNC-Benchmarks in z. B. vier oder acht Partitionen kein Zahlenmaterial gibt. Um in dieser Hinsicht einen Anfang zu machen, wäre es sinnvoll, derartige Partitionierungen der erwähnten Benchmark-Designs durchzuführen, auszuwerten und zu dokumentieren.

Fein-Einstellung von Parametern für den genetischen Algorithmus

Wegen Zeitmangels wurde bisher keine Untersuchung durchgeführt, wie sich verschiedene Belegungen der genetischen Parameter (Populationsgröße, Ersetzungsrate, Selektions- und Crossover-Typ, etc.) auf Güte und Laufzeit des genetischen Partitionierungsalgorithmus auswirken. Die Werte, die momentan verwendet werden, sind entweder Standardwerte des Genetik-Paketes PGAPACK oder sind rein intuitiv gewählt worden.

Partitionierung auf LUT-Ebene

Ein Kompromiß zwischen exakten Kostenmaßen auf CLB-Ebene und kleinen Cutsizes auf Gatter-Ebene könnte darin bestehen, Designs auf der Ebene von Funktionsgeneratoren zu partitionieren. Da durch die innere Struktur eines

[1] Aufgrund der Größe heutiger Anwendungen und des modularen Aufbaus von FPGA-Boards (vergl. die Ausführungen zum WEAVER-Board in Abschnitt 5.4) sind Zieltechnologien, die 8 bis 16 FPGAs enthalten, nicht unrealistisch.

CLBs (vergl. auch Abbildung 2.2) allerdings nur wenige Möglichkeiten zur Verbindung der F-, G- und H-Funktionsgeneratoren sowie der enthaltenen Flip-Flops bestehen, muß während einer derartigen Partitionierung beachtet werden, daß Funktionsgeneratoren nicht beliebig miteinander bzw. mit sämtlichen Flip-Flops des Designs kombiniert und in einem CLB gruppiert werden können. Dem Thema, wie diese Einschränkungen für die Partitionierung effizient berücksichtigt werden können, kann zukünftige Arbeit gewidmet sein.

Design Space Exploration

Sofern COBRA eine gültige Lösung zu einem Partitionierungsproblem findet, wird als Endergebnis die beste gefundene Partition ausgegeben. Die dieser Partitionierung entsprechenden Sub-Designs werden anschließend von XACT synthetisiert. Sollte der Fall eintreten, daß XACT das partitionierte Design nicht verarbeiten kann (weil z. B. aufgrund von Kostenmaßen für Gatter Verletzungen der FPGA-Größen eintreten), ist eine Repartitionierung mit COBRA notwendig. Dieser zeitaufwendige Prozeß kann vermieden werden, indem als Partitionierungsergebnis nicht nur eine Partition ausgegeben wird, sondern ein Teil des Lösungsraumes, der aus einer festen Anzahl gültiger Partitionen besteht. Diese Methode, einen kleinen Teil des Lösungsraumes als Ergebnis zu betrachten, wird *Design Space Exploration* genannt. Im Falle eines Fehlers von XACT kann der Anwender auf eine andere schon verfügbare gültige Lösung ausweichen, die unter Umständen für die nachfolgende Synthese geeignet ist.

Reduktion der Laufzeiten

Mit einiger Anstrengung wäre es sicherlich möglich, die Laufzeiten von COBRA zu verkürzen, auch wenn sich die Komplexität der zugrunde liegenden Algorithmen nicht weiter reduzieren läßt.

Da die Bewertungsfunktion für die Genetik bereits lineare Komplexität gemessen an der Design-Größe hat, wird man am genetischen Algorithmus selbst kaum Verbesserungen vornehmen können. Reduzieren ließe sich allerdings die Zeit für die Initialisierungsphase der Genetik, während der einige Startlösungen durch die Algorithmen von Fiduccia & Mattheyses und Krishnamurthy erzeugt werden. Diese Aufrufe der Algorithmen zur Bipartitionierung ließen sich – wie schon der komplette genetische Partitionierungsalgorithmus – mit Hilfe des Message-Passing Interfaces parallel auf verschiedenen Rechnern durchführen.

Ansatzmöglichkeiten gibt es auch bei den Routinen zur iterativen Clusterung. Für den Benchmark avq.large ließ sich beobachten, daß die rekursive Clusterung eine Sequenz von 53 geclusterten Designs lieferte. Von diesen 53 Designs wurden die letzten 47 wieder entfernt, da während dieser Clusterungsdurchläufe nur sehr wenige neue Cluster gebildet wurden (vergleiche Kapitel 3). Die Rechenzeit, die für die Berechnung der letztlich wieder gelöschten Designs aufgewendet wird, ließe sich vermeiden, indem man die iterative Clusterung nicht mehr dann abbrechen läßt, wenn kein neuer Cluster mehr gebildet wurde, sondern wenn nicht mehr als ein bestimmter Prozentsatz von Zellen zu neuen Clustern vereinigt wurde. Für das Beispiel avq.large würde sich diese Modifikation spürbar in den Laufzeiten äußern, da dieses Design eine sehr hohe interne Vernetzungsdichte aufweist, was zu sehr hohen Laufzeiten zur Clusterung führt.

Intelligentere Heuristiken zur Erzeugung von XNF-Timing Constraints

Bisher werden für ein geschnittenes Netz, das zwischen mehreren FPGAs verläuft, Timing Constraints erzeugt, die alle Pfade auf den FPGAs, die an den Schnittpunkten dieses Netzes beginnen oder enden, mit der gleichen Timing-Information markieren (nämlich mit dem n-ten Bruchteil der Takt-Periode, wenn das geschnittene Netz über n FPGAs zu leiten ist, vergleiche Abschnitt 6.1.1). Diese starre Division der Takt-Periode wird unabhängig davon vorgenommen, wo sich die Schnittpunkte auf einem konkreten Pfad befinden. Eine Verbesserung könnte darin bestehen, daß für einen Pfad die Positionen von Schnittpunkten bestimmt werden. Ein derartiger Pfad könnte dann genau in die Segmente aufgeteilt werden, in die dieser durch die Partitionierung zerfällt. Die Länge dieser Segmente könnte anschließend heuristisch anhand der enthaltenen Anzahlen von Kanten und Knoten bestimmt werden. Es könnten dann Timing Constraints mit Zeitinformationen für diese Segmente erzeugt werden, die den ermittelten Längen entsprechen. Zu diesem Zweck wäre es notwendig, mit Hilfe von Graphdurchläufen Pfade und ihnen zugeordnete Timing Constraints im Design zu identifizieren.

Alternativ wäre auch die Vorgehensweise denkbar, während der Bewertung von Partitionen durch den genetischen Algorithmus ein (noch zu entwickelndes) Programm zur Plazierung und Verdrahtung für XILINX-FPGAs aufzurufen, das in kurzer Zeit konkrete Zeitdaten für die Netze des partitionierten Designs berechnet. Diese Timing-Informationen könnten von einer erweiterten Bewertungsfunktion ausgewertet werden, so daß Partitionen, die Timing Constraints verletzen, als ungültig gekennzeichnet und entsprechend bestraft werden. Hierbei stellt sich allerdings unmittelbar die Frage, wie ein derartiger Ansatz umzusetzen ist, so daß die Laufzeiten nicht ausufern.

A – Zur Benutzung von Cobra

> *„Ein Problem wird nicht im Computer gelöst, sondern in irgendeinem Kopf. Die ganze Apparatur dient nur dazu, diesen Kopf so weit zu drehen, daß er die Dinge richtig und vollständig sieht.“*
>
> C. Kettering

Dieser Anhang enthält einige technische Informationen, die die Benutzung der entwickelten Partitionierungssoftware ermöglichen. Im ersten Abschnitt A.1 wird auf die notwendigen Konfigurationsdateien für Cobra sowie deren Formate eingegangen. Abschnitt A.2 enthält eine kurze Zusammenstellung der vorgesehenen Kommandozeilen-Optionen. Cobra kann aufgerufen werden, nachdem – in dieser Reihenfolge – die Module MPICH und cobra (bzw. MPICH-Solaris und cobra-Solaris) geladen wurden.

A.1 Konfigurationsdateien

Sämtliche Dateien, die der Benutzer anzugeben hat und die von Cobra ausgewertet werden, sind in Text-Format gehalten. Die Dateien werden zeilenweise ausgewertet, so daß jeder Konfigurationsausdruck in einer eigenen Zeile stehen muß. Leerzeilen und Kommentarzeilen, die mit einem Doppelkreuz (#) beginnen, sind erlaubt. Jedes Kommando wird durch ein Schlüsselwort (Groß- und Kleinschreibung wird nicht beachtet) gefolgt von einem Doppelpunkt eingeleitet. Nach diesem Doppelpunkt folgen bis zum Ende der Zeile die zu dem Kommando gehörenden Daten.

Im allgemeinen wird Cobra in der Form

cobra -be *filename*

aufgerufen, wobei *filename* für den Namen der zu interpretierenden Haupt-Konfigurationsdatei steht.

A.1.1 Die Haupt-Konfigurationsdatei

Vor dem Hintergrund, daß COBRA als Backend von COOL anzusehen ist, wird diese Konfigurationsdatei automatisch von COOL erzeugt. Dieselbe Datei dient zugleich auch zur Konfiguration des Interface-Compilers CILC, weshalb Ausdrücke enthalten sein können, die für COBRA uninteressant sind und daher ignoriert werden. An dieser Stelle werden nur die Teile der Haupt-Konfigurationsdatei erläutert, die für COBRA von Bedeutung sind.

Netzlisten-Spezifikation

Zur Spezifikation der VHDL-Dateien, die zu synthetisieren und anschließend zu partitionieren sind, sind in der Haupt-Konfigurationsdatei drei Arten von Ausdrücken vorgesehen:

- Ein Kommando der Form
  ENTITY : *filename, entity-name*
  muß genau einmal in der Konfigurationsdatei vorhanden sein. Über diesen Ausdruck wird die VHDL-Datei festgelegt, die in der Hierarchie des zu synthetisierenden Designs am höchsten steht. Für COOL-Systeme ist hier also stets die generierte Netzlisten-Datei einzutragen. Das Feld *entity-name* muß den Namen der VHDL-Entity enthalten, die in der Hierarchie am höchsten steht.

- Werden durch die mit ENTITY spezifizierte Komponente weitere Komponenten instantiiert, die in anderen VHDL-Dateien beschrieben sind, und die unter Umgehung der High-Level-Synthese direkt in die Logik-Synthese eingehen können, so ist für jede derartige VHDL-Datei ein Ausdruck der Form
  SYNTH : *filename*
  anzugeben.

- Muß eine VHDL-Komponente durch OSCAR synthetisiert werden, so sind Ausdrücke der folgenden Form vorgesehen:
  OSCAR : *filename, entity-name, architecture-name*

Für die Konfiguration von CILC ist die separate Angabe der VHDL-Komponente, die einen von COOL generierten I/O-Controller darstellt, notwendig. Dies geschieht durch das Kommando
IOCTRL : *filename, entity-name*
Dieser Ausdruck darf höchstens einmal in der Haupt-Konfigurationsdatei vorkommen. Stößt COBRA auf dieses Kommando, so wird es unter Ignorierung des Entity-Namens behandelt wie ein SYNTH-Ausdruck.

Soll mit COBRA ein im Net-Format vorliegendes Benchmark-Design partitioniert werden, so ist lediglich ein ENTITY-Ausdruck anzugeben, wobei als Entity-Name eine beliebige Zeichenkette eingetragen werden kann.

Takt-Periode zur Hardware-Synthese

Um die spezifizierte VHDL-Netzliste erfolgreich mit OSCAR und SYNOPSYS zu synthetisieren, ist die Angabe der zu verwendenden Takt-Periode unbedingt erforderlich. Zu diesem Zweck ist das Kommando

CLK : *clock-rate, clock-signal*

vorgesehen, das genau einmal enthalten sein muß. *clock-rate* ist ein numerischer Ausdruck, der die Takt-Periode in Nanosekunden spezifiziert. Der Port des zu synthetisierenden Designs, der das Takt-Signal empfängt, wird in *clock-signal* angegeben.

Zieltechnologie für die Partitionierung

Die Zieltechnologie, die der Partitionierung zugrunde liegen soll, sowie Restriktionen des Partitionierers die Zieltechnologie betreffend, werden in zwei separaten Dateien namens *project-name*.board und *project-name*.cst beschrieben, auf die weiter unten eingegangen wird. In der Haupt-Konfigurationsdatei muß zwingend ein Verweis auf diese Dateien vorhanden sein, der durch den Ausdruck

PROJECT : *project-name*

gegeben wird.

Rechnernamen für den parallelen genetischen Algorithmus

In der Haupt-Konfigurationsdatei können beliebig viele (auch gar keine) Ausdrücke zur Angabe von Rechnern, auf denen der genetische Partitionierungsalgorithmus parallel ausgeführt werden soll, enthalten sein. Hierzu ist die Syntax

HOST : *hostname*

zu verwenden. Bei der Angabe von HOST-Kommandos ist darauf zu achten, daß die Rechner, die mit *hostname* benannt sind, tatsächlich existieren und über das Netzwerk ansprechbar sind. Der parallele genetische Algorithmus wird stets so gestartet, daß der Master-Prozeß (vgl. Abschnitt 6.2) auf der lokalen Maschine, auf der COBRA aufgerufen wurde, läuft. Falls keine HOST-Kommandos angegeben wurden, rechnet der Partitionierer ausschließlich auf dieser lokalen Maschine.

A.1.2 Die Spezifikaton von FPGAs und deren Topologie

Die Angabe der FPGAs, für die ein Design partitioniert werden soll, erfolgt in der Datei *project-name*.board. Hier kann (ohne Angabe eines Schlüsselwortes) pro Zeile ein FPGA deklariert werden, indem die *volle* Bezeichnung der FPGAs eingetragen wird – dazu gehören FPGA-Typ, Package-Typ und Speedgrade. Eine Zeile für ein FPGA könnte somit den Inhalt 4005pc84-5 haben. Es sind so viele Zeilen mit derartigen Angaben in die Konfigurationsdatei aufzunehmen, wie die Anzahl von FPGAs der Zieltechnologie beträgt.

Die dermaßen spezifizierten FPGAs werden intern in der Reihenfolge ihres Auftretens in der Konfigurationsdatei durchnumeriert. D. h., daß das FPGA, das in der ersten nicht-leeren Zeile der Datei angegeben ist, die keine Kommentar-Zeile ist, FPGA Nummer 1 ist, etc.

Topologien können mit Hilfe mehrfacher TOP-Kommandos spezifiziert werden. Die Syntax ist:

TOP : *FPGA-Nr.* – $C_0, C_1, \ldots, C_n$;

Durch eine derartige Zeile werden sämtliche Anschlüsse des FPGAs Nr. *FPGA-Nr.* spezifiziert. Die Felder C_0 bis C_n enthalten ganzzahlige Werte. C_0 enthält die Anzahl von Pins eines FPGAs, die als externe Anschlüsse des Boards dienen. Eine Zahl C_i $(1 \leq i \leq n)$ gibt die Anzahl von Verbindungen zwischen FPGA *FPGA-Nr.* und FPGA i wieder.

Bei Angabe einer Board-Topologie sind die TOP-Kommandos also so zu formulieren, daß durch sämtliche C_i-Werte eine komplette *Adjazenz-Matrix* des Graphen, der aus dem Board resultiert, mitsamt einer zusätzlichen Spalte für die externen Anschlüsse der FPGAs modelliert wird. Die korrekte Spezifikation eines WEAVER-Boards (vgl. Abschnitt 5.4) sollte folgendes Aussehen haben:

```
4025ehq304-5
4025ehq304-5
4025ehq304-5
4025ehq304-5
TOP : 1 – 96,0,75,75,0;
TOP : 2 – 96,75,0,0,75;
TOP : 3 – 96,75,0,0,75;
TOP : 4 – 96,0,75,75,0;
```

A.1.3 Plazierungsbeschränkungen für Design-Ports

Wie bereits in Abschnitt 5.4.1 vorgestellt, können die Ports des zu partitionierenden Designs vor der Partitionierung bestimmten FPGAs dauerhaft zugewiesen werden. Diese Zuweisungen werden über die Datei *project-name*.cst vorgenommen. Die Zeilen dieses Files enthalten Ausdrücke der Form

CST : *port-name*, *FPGA-Nr.*, *FPGA-pin*;

Mit einem derartigen Kommando wird der Port, der in den VHDL-Codes mit *port-name* bezeichnet wird, an Pin *FPGA-pin* des FPGAs Nr. *FPGA-Nr.* gebunden. Das i-te Signal eines Busses *bus* kann über den Port-Namen *bus*$<i>$ referenziert werden. Die Numerierung der FPGAs wird in der gleichen Weise vorgenommen wie bei der Spezifikation von Topologien. *FPGA-pin* ist eine Zeichenkette, die den Pin-Bezeichnungen von XILINX-FPGAs entspricht.

A.1.4 Bibliotheken für FPGAs und XBLOX-Makros

Um COBRA die Größen einzelner FPGA-Typen und Design-Zellen verfügbar zu machen, sind die Bibliotheken fpga.lib und xblox.lib anzulegen. Existieren diese

Files beim Start von COBRA nicht, so werden sie selbständig erzeugt und mit Daten für eine große Anzahl von FPGA-Typen und XBLOX-Makros bestückt. Der Benutzer kann diese Bibliotheken um eigene Einträge erweitern.

In die Bibliothek fpga.lib sind zeilenweise Ausdrücke der Form
FPGA-Typ : # *CLBs* , # *Pins*;
einzutragen.

FPGA-Typ ist dabei eine ähnliche Zeichenkette, wie auch für die Spezifikation der Zieltechnologie verwendet wird. Der Unterschied besteht nur darin, daß für die Angaben in dieser Bibliothek der Speedgrade der FPGAs nicht anzugeben ist. Enthält die Datei *project-name*.board also einen Eintrag 4005pc84-5, so sucht COBRA in fpga.lib nach einem Ausdruck, der mit 4005pc84 : beginnt. Nach dem Doppelpunkt sind in dem Rest der Zeile die zu dem betreffenden FPGA-Typen gehörenden Anzahlen von CLBs und Pins einzutragen.

Die Bibliothek xblox.lib enthält eine Auflistung aller zu berücksichtigenden XBLOX-Makros mitsamt ihren Größen gemessen in CLBs. Ein einzelner Eintrag dieser Auflistung wird nach der folgenden Syntax vorgenommen:
macroname : # *CLBs*;

Diese Bibliothek enthält – sofern sie von COBRA angelegt wurde – Einträge für Addierer/Subtrahierer, Inkrementer/Dekrementer und Komparatoren aller Bitbreiten. Durch Hinzufügen neuer Einträge (z. B. FDC : 3;) kann der Benutzer Kostenmaße für verschiedene Typen von Gattern angeben, falls eine Partitionierung auf Gatter-Ebene erfolgen soll.

A.1.5 Konfiguration der Synthese-Werkzeuge

Der Aufruf von OSCAR und OSBACK kann über die Datei oscar_call.spec gesteuert werden. Dieses File wird wie die beiden oben beschriebenen Bibliotheken automatisch erzeugt, falls es noch nicht existiert. Es enthält die genauen Kommandozeilen-Angaben, mit denen OSCAR und OSBACK aufgerufen werden. Standardgemäß hat diese Datei den folgenden Inhalt:

```
OSCAR : oscar -Of -m /project/oscar/packages/lp_solve201/lp_solve
OSBACK : osback -L /project/oscar/packages/lp_solve201/lp_solve -O vhdl1
```

In einem Unterverzeichnis oscar werden – falls notwendig – die OSCAR-Bibliotheken Components.lib und Functions.lib angelegt, die vom Benutzer an seine Bedürfnisse angepaßt werden können.

Die Synthese des kompletten zu partitionierenden Designs wird über den SYNOPSYS-Kommandozeilen-Interpreter dc_shell vorgenommen. Hierzu erzeugt COBRA automatisch Konfigurationsdateien und das Skript synthesis.script (wie üblich unter dem Vorbehalt, daß diese Files noch nicht existieren). Dieses Skript ist so angelegt, daß die anschließende Partitionierung auf CLB-Ebene vorgenommen wird. Änderungen an diesem Skript sollten nur von Anwendern vorgenommen werden, die sich mit SYNOPSYS auskennen.

Da High-Level- und Logik-Synthese überaus zeitintensive Prozesse sind, wird die Synthese von COBRA – in Anlehnung an das Programm make – nur dann gestartet, wenn seit der letzten Synthese Änderungen in den VHDL-Codes vorgenommen wurden.

A.2 Kommandozeilen-Optionen

COBRA läßt sich über eine Reihe von Kommandozeilen-Optionen an unterschiedliche Bedürfnisse anpassen. Im einzelnen werden die folgenden Optionen akzeptiert:

-h (help)
Hiermit wird eine Übersicht über die möglichen Kommandozeilen-Optionen von COBRA ausgegeben.

-clear
Bei Angabe dieser Option räumt COBRA das aktuelle Arbeitsverzeichnis auf, indem alle von dem Programm generierten Dateien und Verzeichnisse gelöscht werden.

-rc (recover)
Es kann (vor allem bei der SunOS 4-Version) vorkommen, daß COBRA während der Partitionierung – genauer: nach den Aufrufen der Algorithmen von Fiduccia & Mattheyses bzw. von Krishnamurthy – aus bislang ungeklärten Gründen abstürzt. Damit die bis zum Absturz berechneten Ergebnisse nicht endgültig verloren gehen, speichert COBRA diese zwischen. Beim Aufruf von COBRA im *Recover-Modus* liest die Software diese Zwischenergebnisse und fährt mit der Partitionierung an der Stelle fort, wo vorher abgebrochen wurde.

-hg (hypergraph)
Soll ein Benchmark-Design partitioniert werden, so ist diese Option anzugeben. Das Design wird dann aus dem File gelesen, das in der Netzlisten-Spezifikation mit dem ENTITY-Schlüsselwort angegeben ist. Dieses muß im Net-Format vorliegen.

-fs (force synthesis)
Wenn COBRA die Hardware-Synthese durchführen soll, obwohl die zugrunde liegenden VHDL-Codes auf dem neuesten Stand sind (z. B. wenn Änderungen an den Synthese-Skripten vorgenommen wurden), kann dies durch die force-synthesis-Option erzwungen werden.

-dc (disable clustering)
Die Durchführung der iterativen Clusterung kann mittels dieses Schalters unterdrückt werden.

-fi *N* (FM-iterations)
Die Anzahl von Aufrufen der Algorithmen von Fiduccia & Mattheyses

bzw. von Krishnamurthy kann über diese Kommandozeilen-Option auf den Wert *N* festgelegt werden (Standard: 20).

-gi *N* (genetic-iterations)
: Das Setzen der maximalen Anzahl von Generationen, die während des genetischen Algorithmus berechnet werden sollen, auf *N* kann hiermit vorgenommen werden (Standard: 1000).

-sn (stop nochange)
: Üblicherweise bricht der genetische Partitionierungsalgorithmus ab, wenn über eine bestimmte Anzahl von Generationen hinweg keine Verbesserung des besten bisher berechneten Fitneß-Wertes gefunden wurde. Dieses Verhalten kann über diese Option ausgeschaltet werden, so daß nur nach Berechnung einer bestimmten Anzahl von Populationen terminiert wird.

-nc *N* (nochange)
: Falls die Option -sn nicht verwendet wird, kann hiermit festgelegt werden, nach wievielen Generationen ohne weitere Verbesserung abgebrochen werden soll (Standard: 150).

-ps *N* (population size)
: Die Populationsgröße läßt sich über diese Option beeinflussen (Standard: 100).

-rr *N* (replacement rate)
: Die Ersetzungsrate, die angibt, welcher prozentuale Anteil der Populationsgröße von Generation zu Generation duch neue Nachkommen zu ersetzten ist, kann mit dieser Option auf *N* gesetzt werden (Standard: 50 %).

-rs *N* (random seed)
: Der Zufallszahlen-Generator, der zur Erzeugung der initialen Populationen der Genetik verwendet wird, kann auf den festen Wert *N* gesetzt werden. Normalerweise wird dieser über die System-Uhr beeinflußt.

-pf *N* (print frequency)
: Die Häufigkeit, mit der der genetische Algorithmus Ausgaben über den Fortschritt der Berechnungen macht, kann auf eine beliebige Anzahl *N* von Generationen gesetzt werden (Standard: 10).

Abkürzungsverzeichnis

„In der Kürze liegt die Würze.“
Volksmund.

ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
COOL	Co-Design Tool
COBRA	Cool Backend for Rapid Prototyping
DSP	Digital Signal Processor
FIFO	First In, First Out
FM	Algorithmus von Fiduccia & Mattheyses
FPGA	Field Programmable Gate Array
IOB	Input/Output Block
LEDA	Library of Efficient Datatypes and Algorithms
LUT	Lookup-Table
MPI	Message-Passing Interface
OSCAR	Optimum Simultaneous Scheduling, Allocation and Resource Binding
PGA	Parallel Genetic Algorithm
PLD	Programmable Logic Device
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
XNF	Xilinx Netlist Format

Literaturverzeichnis

„Habeat librarius et registrum omnium librorum ordinatum secundum facultates et auctores, reponatque eos separatim et ordinate cum signaturis per scripturam applicatis.“ [1]

Aus: U. Eco, „Der Name der Rose“.

[Alpe98] C.J. Alpert, *The Circuit Partitioning Page*, Los Angeles, 1998. `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html`

[AlYa95] C.J. Alpert und So-Zen Yao, *Spectral Partitioning: The more Eigenvectors, the better*, Design Automation Conference Proceedings, San Francisco, 1995.

[BaMa94] G. Baumann und P. Marwedel, *Einführung in VHDL*, Dortmund, 1994.

[BHSB93] D. Brasen, J. P. Hiol, G. Saucier und H. Belhadj, *Circuit Partitioning for FPGAs*, IFIP Workshop on Logic and Architecture Synthesis, Grenoble, 1993.

[CBL97] North Carolina State University, *Collaborative Benchmarking Laboratory Homepage*, Raleigh, 1997. `http://www.cbl.ncsu.edu/pub/Benchmark_dirs/Partitioning93/XILINX/XNF/`

[DuDe96] S. Dutt und W. Deng, *A Probability-Based Approach to VLSI Circuit Partitioning*, Design Automation Conference Proceedings, Las Vegas, 1996.

[FiMa82] C. M. Fiduccia und R. M. Mattheyses, *A Linear-Time Heuristic for Improving Network Partitions*, Design Automation Conference Proceedings, Las Vegas, 1982.

[1] Der Bibliothekar habe ein Verzeichnis aller Bücher, geordnet nach Themen und Autoren, und er bewahre sie einzeln auf und wohlgeordnet mit schriftlich aufgebrachten Signaturen.

[Flor91] R. Floren, *A note on "A faster approximation algorithm for the Steiner problem in graphs"*, Information Processing Letters, 1991.

[GaJo79] M. R. Garey und D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, New York, 1979.

[GrLu97] W. Gropp und E. Lusk, *User's Guide for MPIch, a Portable Implementation of MPI*, Argonne, 1997. `http://www.mcs.anl.gov/mpi/mpich/`

[HaBo95] S. Hauck und G. Borriello, *An Evaluation of Bipartitioning Techniques*, Proceedings of the 16th Conference on Advanced Research in VLSI, 1995.

[HaKa91] L. Hagen und A. Kahng, *Fast Spectral Methods for Ratio Cut Partitioning and Clustering*, IEEE International Conference on Computer-Aided Design Proceedings, Santa Clara, 1991.

[KaFr93] J. Kahlert und H. Frank, *Fuzzy-Logik und Fuzzy-Control*, Braunschweig, 1993.

[Kris84] B. Krishnamurthy, *An Improved Min-Cut Algorithm for Partitioning VLSI Networks*, IEEE Transactions on Computers Vol. C-33, New York, 1984.

[KAKS97] G. Karypis, R. Aggarwal, V. Kumar und S. Shekhar, *Multilevel Hypergraph Partitioning: Application in VLSI Domain*, Design Automation Conference Proceedings, Anaheim, 1997.

[KKR98] U. Kebschull, G. Koch und W. Rosenstiel, *The WEAVER Prototyping Environment for Hardware/Software Co-Design and Co-Debugging*, Design, Automation and Test in Europe Conference – Designer Track, Paris, 1998.

[KMB81] L. Kou, G. Markowsky und L. Berman, *A fast algorithm for Steiner trees*, Acta Informatica, 1981.

[Leng90] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*, Stuttgart, 1990.

[Levi96] D. Levine, *Users Guide to the PGAPack Parallel Genetic Algorithm Library*, Argonne, 1996. `http://www.mcs.anl.gov/pgapack.html`

[LMD94] B. Landwehr, P. Marwedel und R. Dömer, *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming*, European Design Automation Conference Proceedings, Grenoble, 1994.

[Mehl88] K. Mehlhorn, *A faster approximation algorithm for the Steiner problem in graphs*, Information Processing Letters, 1988.

[NäUh96] S. Näher und C. Uhrig, *The LEDA User Manual Version R 3.3.1*, Saarbrücken, 1996. `http://www.mpi-sb.mpg.de/LEDA/leda.html`

[Niem98] R. Niemann, *Hardware/Software Codesign for Data Flow Dominated Embedded Systems*, Dissertation, Universität Dortmund, 1998.

[PG293a] Projektgruppe 293, *Zwischenbericht der Projektgruppe 293 – Entwurf und Realisierung eines Fuzzy-Coprozessors auf der Basis von FPGAs und digitalen Signalprozessoren*, Forschungsbericht, Universität Dortmund, 1997.

[PG293b] Projektgruppe 293, *Endbericht der Projektgruppe 293 – Entwurf und Realisierung eines Fuzzy-Coprozessors auf der Basis von FPGAs und digitalen Signalprozessoren*, Forschungsbericht, Universität Dortmund, 1997.

[RDJ94] B. M. Riess, K. Doll und F. M. Johannes, *Partitioning very large circuits using analytical placement techniques*, Design Automation Conference Proceedings, San Diego, 1994.

[Schä98] C. Schäfer, *Interface-Synthese für inkompatible Protokolle*, Diplomarbeit, Universität Dortmund, 1998.

[Syno92] Synopsys Inc., *Synopsys Design Compiler Reference Manual, Version 3.0*, Mountain View, 1992.

[WeRo94] U. Weinmann und W. Rosenstiel, *Circuit clustering and partitioning for system implementations*, IFIP Workshop on Logic and Architecture Synthesis, Grenoble, 1994.

[Xili91] Xilinx Inc., *The XC4000 Data Book*, San Jose, 1991.

[Xili94] Xilinx Inc., *Xilinx Synopsys Interface FPGA User Guide*, San Jose, 1994.

[Xili95] Xilinx Inc., *Xilinx Netlist Format (XNF) Specification, Version 6.1*, San Jose, 1995.
`ftp://ftp.xilinx.com/pub/documentation/xactstep6/xnfspec.pdf`

[YaWo94] H. Yang und D. F. Wong, *Efficient Network Flow based Min-Cut Balanced Partitioning*, IEEE International Conference on Computer-Aided Design Proceedings, San Jose, 1994.

Stichwortverzeichnis

„*Der Gestank des Arrak füllt seine Nüstern. Der Brandy brennt, ist aber weniger widerlich. Den Cider hätte er fast freiwillig trinken können. Etwas ermutigt versucht er, den Rhythmus zu halten, Drambuie, Eierflip, Furlauer,* pfft...ping!, *Gin, Highland Malt, Irish Coffee, und weiter macht er,* schlürf, schlürf! *Das Trinken geht entschieden leichter, o ja, Julep, Kümmel, Lethe, umdrehn und auffangen, Malvasier, Nektar, Oggersheimer,* ping?, *Porter, Qwaß, Rum,* ping! *Sekt, Trester, Unkensaft rinnen ihm übers Kinn, egal, Veltliner, Würzbier, Xeres, Yggdrasilschnaps, zwings runter, noch einen. Er kippt den letzten mit Schwung runter, gießt ihn nach hinten auf seine gierigen Mandeln, Zythum, Pyramidensaft.* Schlürrfff.“

Ähnlich aus: L. Norfolk, „Lemprières Wörterbuch“.

Erklärung

> *„Jeder Mensch hat Anspruch auf Erholung und Freizeit sowie auf eine vernünftige Begrenzung der Arbeitszeit und auf periodischen, bezahlten Urlaub.“*
>
> Aus: „Allgemeine Erklärung der Menschenrechte“.

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit im Rahmen der gewährten wissenschaftlichen Unterstützung selbständig und ausschließlich unter Verwendung der im Literaturverzeichnis angegebenen Quellen verfaßt habe.

Dortmund, 9. August 1999