Optimale Mikroarchitektursynthese mittels Ganzzahliger Programmierung

Diplomarbeit

Rainer Dömer

Betreuer
Prof. Dr. P. Marwedel
Dipl. Inform. B. Landwehr

Universität Dortmund
Fachbereich Informatik
Lehrstuhl XII

9. November 1994

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Einführung und Motivation

Die weiter ansteigende Komplexität digitaler Schaltungen erfordert rechnergestützte Entwurfsverfahren. Ohne die Unterstützung durch spezielle Werkzeuge für den Entwurf digitaler Schaltungen (*CAD*[1] *for VLSI*[2]) ist die Entwicklung hochintegrierter Schaltkreise undenkbar. Allein die hohe Integrationsdichte heutiger Mikrochips macht den Einsatz von CAD-Werkzeugen notwendig.

Hinzu kommen wachsende zeitliche Anforderungen an den Entwurf integrierter Schaltungen. Ein immer kürzerer Designzyklus wird insbesondere durch gegebene Marktanforderungen notwendig (*Time-To-Market*). Eine schnelle Schaltungsentwicklung erfordert gleichzeitig einen möglichst fehlerfreien Entwurfsprozeß, da insbesondere spät erkannte Fehler sehr zeit- und kostenaufwendig sind. Besondere Bedeutung kommt daher Verfahren zu, die bereits durch ihre Konstruktion Fehlerfreiheit garantieren (*Correctness by Construction*).

Diese Forderungen an den Entwurf elektronischer Systeme machen eine hochgradige Automatisierung notwendig.

Die Entwicklung hochintegrierter, digitaler Schaltungen ist eine sehr komplexe Aufgabe. Üblicherweise wird zwischen verschiedenen *Abstraktionsebenen* unterschieden [Ma93, Einführung]. Diese reichen von der System-Ebene (Prozessoren und Speicher) hinunter bis zur Bauelement- und Prozeß-Ebene. Gajski [GaKu83] ordnet diese Ebenen in Form eines Y-Diagramms (*Y-Chart*) an, welches den *schrittweisen* Entwurfsprozeß graphisch darstellt. Bei einem schrittweisen Entwurf wird jeweils eine *Spezifikation* auf einer höheren Ebene auf eine *Implementierung* in der darunterliegenden Ebene abgebildet. Zum Beispiel werden Boolesche Gleichungen (Logik-Ebene) i. d. R. zunächst auf Gatter abgebildet, welche wiederum aus einzelnen Schaltern zusammengesetzt werden. Ziel der *Verifikation* ist umgekehrt der Nachweis, daß ein auf einer niederen Ebene

[1] CAD, Computer Aided Design, Rechnergestützter Entwurf
[2] VLSI, Very Large Scale Integration, sehr hohe Integrationsdichte

vorliegendes System das Verhalten der Spezifikation auf der darüberliegenden Ebene aufweist (i. d. R. wird hierzu die Simulation verwendet).

Mittlerweile haben sich auf vielen Ebenen automatische Entwurfsverfahren etabliert. Insbesondere für den Bereich der Logik-Ebene und im Bereich der Steuerwerkserzeugung (*Finite State Machines*) stehen heute mächtige, weitgehend automatische Entwurfswerkzeuge zur Verfügung.

Aktuelle Forschungen beschäftigen sich mit der Automatisierung auf höheren Ebenen. Hierzu zählt insbesondere der automatische Schaltungsentwurf auf der *Register-Transfer*-Ebene aus einer algorithmischen Spezifikation heraus.

1.1.1 Die Mikroarchitektursynthese

Die Register-Transfer-Ebene (RT-Ebene) wird auch als Mikroarchitektur-Ebene bezeichnet. Die automatische Erzeugung von RT-Strukturen ist die Aufgabe der *Mikroarchitektursynthese*, welche das Rahmenthema der vorliegenden Arbeit ist.

Synthese[3] bezeichnet das Zusammensetzen von vorhandenen Komponenten zu einem Ganzen. Der Mikroarchitektursynthese stehen RT-Bausteine (Register, arithmetische und logischen Einheiten, RAMs und Busse) zur Verfügung, welche so zusammenzusetzen sind, daß ein auf der darüberliegenden, algorithmischen Ebene spezifiziertes Verhalten erreicht wird.

Die Mikroarchitektursynthese wird in der englisch-sprachigen Fachliteratur als *High-Level Synthesis* (HLS) bezeichnet. McFarland, Parker und Camposano [McPaCa88] definieren die Mikroarchitektursynthese so:

> „High-level synthesis takes an abstract behavioral specification of a digital system and finds a register-transfer level structure that realizes the given behavior."

Die Mikroarchitektursynthese bildet also eine abstrakte Verhaltensbeschreibung (die Spezifikation des Ein-Ausgabeverhaltens) eines Entwurfs auf eine Strukturbeschreibung auf RT-Ebene ab. Die Aufgabe eines Synthesesystems besteht darin, das beschriebene algorithmische Verhalten so zu implementieren, daß gegebene Randbedingungen (z. B. Zeit- und Platzbeschränkungen) eingehalten werden und der Entwurf in bezug auf die entstehenden Kosten (z. B. die Anzahl der Bausteine) optimiert wird.

Der Ablauf der Mikroarchitektursynthese läßt sich in vier Schritte unterteilen.

Der erste Schritt beinhaltet das Einlesen der Entwurfsspezifikation, welche eine algorithmische Beschreibung des Ein-Ausgabeverhaltens des zu erzeugenden Systems darstellt. Dieses Entwurfsverhalten wird i. d. R. in einer Hardware-Beschreibungssprache (z. B. in VHDL [IEEE88]) spezifiziert. Zusätzlich ist die

[3] Synthese: aus dem altgriech. $\grave{\eta}$ $\sigma\acute{\upsilon}\nu\vartheta\varepsilon\sigma\iota\varsigma$, die Zusammensetzung, Zusammenfügung

Eingabe von Syntheseparametern notwendig. Zum einen zählen hierzu Randbedingungen (*Constraints*), die bei der Synthese einzuhalten sind (z. B. Zeitvorgaben oder maximale Anzahlen funktionaler Einheiten). Zum anderen ist das Ziel (*Goal*) der Synthese zu spezifizieren (z. B. die Optimierung bezüglich minimaler Chipfläche).

Den zweiten Schritt bildet die Übersetzung der Entwurfsspezifikation in eine interne Repräsentation. In der Regel wird hierzu ein kombinierter Kontroll- und Datenflußgraph aufgestellt. Der Kontrollflußgraph beschreibt die Kontrollstrukturen des Entwurfs, d. h. er repräsentiert insbesondere Verzweigungen und Schleifenkonstrukte. Datenflußgraphen entstehen aus den in der Verhaltensbeschreibung spezifizierten Variablenzuweisungen und enthalten sämtliche arithmetischen und logischen, sowie Lese- und Schreib-Operationen. Insbesondere Datenabhängigkeiten zwischen den Operationen werden explizit in den Datenflußgraphen repräsentiert.

Der dritte und zentrale Schritt der Mikroarchitektursynthese ist die Abbildung des Entwurfsverhaltens auf RT-Strukturen, d. h. insbesondere die Zuordnung der Operationen zu ausführenden Bausteinen. (Dieser Schritt wird im folgenden Unterabschnitt gesondert vorgestellt.)

Den vierten und letzten Schritt bildet die Ausgabe der erzeugten RT-Strukturen. Diese bestehen zum einen aus einer Netzliste der verwendeten Bausteine (Register, Multiplexer, funktionale Einheiten...) und zum anderen aus der Spezifikation des benötigten Steuerwerks (*Controller*). Sowohl die Netzliste als auch das Steuerwerk können hier wieder in einer Hardware-Beschreibungssprache (z. B. VHDL) beschrieben werden.

Ziel der vorliegenden Arbeit ist es, ein solches System zur Mikroarchitektursynthese zu entwerfen und zu implementieren. Insbesondere der dritte Schritt, die Abbildung der in den Datenflußgraphen enthaltenen Operationen auf Bausteine der Register-Transfer-Ebene, ist Inhalt dieser Arbeit.

1.1.2 Scheduling, Allocation und Binding

Der Kern der Mikroarchitektursynthese ist die Zuordnung von in den Datenflußgraphen enthaltenen Operationen zu sog. Kontrollschritten und ausführenden Funktionseinheiten. Diese Abbildung läßt sich in drei Teilaufgaben unterteilen, welche mit *Scheduling*, *Allocation* und *Binding* bezeichnet werden[4].

Die Aufgabe des *Scheduling* ist die zeitliche Ablaufplanung der Operationsausführung, d. h. die Zuordnung von Operationen zu Kontrollschritten (*Control Steps*).

Allocation bezeichnet die Bereitstellung von *geeigneten* Ressourcen (RT-Bausteine) zur Ausführung der Operationen.

[4] In der vorliegenden Arbeit werden die englisch-sprachigen Fachbegriffe *Scheduling*, *Allocation* und *Binding* den deutschen Umschreibungen *Ablaufplanung*, *Bereitstellung* und *Zuordnung* vorgezogen, da sie die Teilprobleme der Mikroarchitektursynthese treffender bezeichnen.

Binding (oder auch *Assignment*) schließlich bezeichnet die feste Zuordnung von Operationen zu ausführenden Bausteinen.

Zur Behandlung dieser drei Teilprobleme sind eine Vielzahl von Algorithmen und Optimierungsverfahren entworfen worden (ein detaillierter Überblick ist in [Ma93] gegeben), auf die hier nicht weiter eingegangen werden soll.

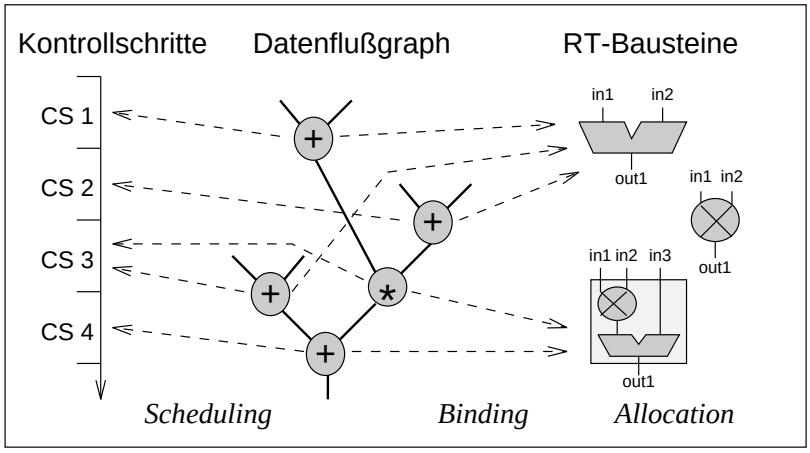

Abbildung 1.1: Die Aufgaben der Mikroarchitektursynthese

Abbildung 1.1 illustriert den Zusammenhang der drei Teilaufgaben. Das Scheduling kann mehrere Operationen nur dann gleichzeitig ausführen, wenn für diese Operationen jeweils eigene Ressourcen bereitgestellt werden (Allocation) und eine Zuordnung der Operationen zu Bausteinen gefunden werden kann (Binding), die die Ausführbarkeit der Operationen auf den Bausteinen ermöglicht.

Offensichtlich sind Scheduling, Allocation und Binding jeweils voneinander abhängig, so daß es Ziel eines Synthesesystems sein muß, die Teilaufgaben nach Möglichkeit gemeinsam anzugehen.

Zur Beurteilung eines Entwurfs wird i. d. R. eine *Kostenfunktion* definiert. Die Optimierung des Entwurfs ist somit mit der Minimierung der Kostenfunktion gleichzusetzen. Die Kosten können zum Beispiel durch den Platzbedarf der Bausteine, die Anzahl und Breite der Register oder die Anzahl der benötigten Kontrollschritte definiert werden. Auch eine Kombination der Kostenanteile ist denkbar.

In der Literatur (z. B. [HwLeHs91]) werden insbesondere zwei Optimierungsverfahren unterschieden. Ein Ressourcen-beschränktes (*resource-constraint*) Verfahren arbeitet mit fixiertem Bausteinangebot und optimiert die Ausführungszeit des Entwurfs. Umgekehrt wird ein Syntheseverfahren als zeitbeschränkt (*time-constrained*) bezeichnet, wenn die Anzahl der zur Verfügung stehenden Kontrollschritte vorgegeben ist und in Richtung minimaler Bausteinkosten optimiert wird.

In den letzten Jahren sind eine Reihe neuer Verfahren zur Mikroarchitektursynthese vorgestellt worden, die sich Mitteln des Operations Research bedienen.

Insbesondere die Integration von Scheduling, Allocation und Binding ist mit diesen Methoden formal möglich. Bevor einige dieser Ansätze vorgestellt werden, muß die sog. Ganzzahlige Programmierung definiert werden.

1.1.3 Die Ganzzahlige Programmierung

Bei der sog. *Ganzzahligen Programmierung* handelt es sich um ein Teilgebiet des *Operations Research*. Der Begriff Operations Research (OR) umfaßt insbesondere im Rahmen der Betriebswirtschaftslehre mathematische, quantitative Methoden zur Planung und Bewertung von Problemlösungen [We93].

Einen wichtigen Bereich des Operations Research stellt die *Lineare Optimierung* dar. Diese bezeichnet die Optimierung einer linearen Zielfunktion unter linearen Nebenbedingungen. In der Fachliteratur wird die Lineare Optimierung auch als *Linear Programming* (LP) bezeichnet. Der Begriff *Programming* steht in diesem Zusammenhang nicht für die Programmierung eines Rechners, sondern vielmehr für *Planung*.

Mathematisch wird das Problem der Linearen Optimierung folgendermaßen definiert: Minimiere $Z(\vec{x}) := \vec{c}\,\vec{x}$ unter der Bedingung $A\vec{x} \leq \vec{b}$ und $\vec{x} \geq 0$, wobei die Koeffizienten $A \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$ und $\vec{c} \in \mathbb{R}^n$ gegeben sind und $\vec{x} \in \mathbb{R}^n$ gesucht wird.

Anschaulicher ist die Notation in Matrix-Schreibweise:

Minimiere die Zielfunktion

$$Z(\vec{x}) := \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

unter der Nebenbedingung

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

und der weiteren Bedingung

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

mit

$$a_{i,j}, b_i, c_j \in \mathbb{R} \quad \forall i, j \in \mathbb{N}, 1 \leq i \leq m, 1 \leq j \leq n$$

wobei m die Anzahl der Nebenbedingungen (*Constraints*) und n die Anzahl der Variablen angibt.

Betrachtet man die Anzahl der linear unabhängigen Nebenbedingungen, so ist diese in aller Regel kleiner als die Anzahl der zu bestimmenden Variablen (es gilt $m < n$). Das Gleichungssystem für eine lineares Optimierungproblem ist i. d. R. also *unterbestimmt*, d. h. es existiert ein Lösungs*raum*, der die Menge zulässiger Lösungen umfaßt. Es kann nun gezeigt werden, daß dieser Lösungsraum *konvex* ist und eine in bezug auf die Zielfunktion minimale Lösung in einem *Extrempunkt* (Eckpunkt) des Raumes zu finden ist [PaSt82].

Zur Lösung des linearen Optimierungsproblems kann der *Simplex-Algorithmus* [DaOrWo55] eingesetzt werden. Die Lösungsstrategie dieses Algorithmus' besteht vereinfacht darin, einen zulässigen Extrempunkt zu bestimmen und von diesem aus weitere Extrempunkte derart aufzusuchen, daß der Wert der Zielfunktion stetig sinkt. Ist kein Extrempunkt mehr zu finden, der den Wert der Zielfunktion weiter verkleinert, so liegt eine optimale Lösung vor. Eine mathematisch basierte und detaillierte Beschreibung des Simplex-Algorithmus' ist in [We93] angegeben.

Die *Ganzzahlige Optimierung*, welche auch bezeichnet wird als *Ganzzahlige Programmierung* (*Integer Programming*, IP), stellt eine Verschärfung der Linearen Optimierung dar. Zusätzlich zur Minimierung der linearen Zielfunktion unter linearen Nebenbedingungen wird hier die Ganzzahligkeit der Variablen verlangt:

Zielfunktion (*Objective Function*):

$$Z(x) := c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \quad \rightarrow \min$$

Nebenbedingungen (*Constraints*):

$$a_{i,1} x_1 + a_{i,2} x_2 + \ldots + a_{i,n} x_n \;\leq\; b_i \quad \forall i \in I\!N, 1 \leq i \leq m$$

Ganzzahligkeits-Bedingung (*Integer Constraint*):

$$x_j \in I\!N_0 \quad \forall j \in I\!N, 1 \leq j \leq n$$

Die Bedingung der Ganzzahligkeit der Lösung schränkt den Lösungsraum offensichtlich stark ein, vereinfacht das Problem aber keineswegs. Vielmehr steigt die Komplexität hierdurch erheblich. Das Problem der Ganzzahligen Optimierung ist NP-hart [PaSt82]. Daher sind im allgemeinen extreme Rechenzeiten für diese Optimierungen zu erwarten. Dennoch stehen Heuristiken zur Verfügung, die für viele Beispiele in akzeptabler Zeit eine optimale Lösung liefern. Der Algorithmus von Gomory [Go58] stellt eine Iteration des Simplex-Algorithmus' auf Basis eines Branch-And-Bound-Verfahrens dar. Dieser Algorithmus sowie detaillierte Komplexitätsanalysen sind in [PaSt82] angegeben.

Abschließend sollen noch zwei verwandte Optimierungsprobleme erwähnt werden: Fordert man lediglich die Ganzzahligkeit eines Teils der Lösung, so wird das Problem als Gemischt-Ganzzahlige Programmierung (MILP, *Mixed Integer Linear Programming*) bezeichnet. Die Binäre Optimierung (0-1-IP, *Binary Programming*) bezeichnet den Fall, daß sämtliche Variablen nur die Werte Null oder Eins annehmen dürfen ($\vec{x} \in \{0,1\}^n$).

1.2 Literaturstudium

Im Bereich der Mikroarchitektursynthese sind in den letzten Jahren neue Optimierungsverfahren vorgestellt worden, die sich der Ganzzahligen Programmierung bedienen. Gegenüber älteren Verfahren, die die Teilprobleme Scheduling, Allocation und Binding getrennt behandeln, bietet die Modellierung des Syntheseproblems als lineares Optimierungsproblem den wesentlichen Vorteil, daß die Aufgaben der High-Level Synthese einheitlich formuliert und damit gleichzeitig gelöst werden können.

Im folgenden sollen einige grundlegende Arbeiten in diesem Bereich vorgestellt werden.

1.2.1 Die Integration von Scheduling und Allocation

Insbesondere die Aufgabe des Scheduling ist der Ansatzpunkt für die ersten Arbeiten zur Nutzung der Ganzzahligen Programmierung in der High-Level Synthese. In [HwLeHs91] stellen Hwang, Lee und Hsu ein IP-Modell vor, welches die Zuordnung von Operationen zu Kontrollschritten mittels binärer Entscheidungsvariablen vornimmt. Das Modell ist in der Lage, sowohl die Anzahl verwendeter Bausteine, als auch die Anzahl der notwendigen Kontrollschritte zu minimieren. Weiterhin werden eine Reihe von Anforderungen für allgemeines Scheduling unterstützt.

Gebotys und Elmasry beschreiben in [GeEl91] ein IP-Modell, welches eine Weiterentwicklung des obigen Ansatzes darstellt. Auch hier werden Scheduling und Allocation gleichzeitig gelöst. Insbesondere sind aber die Behandlung von Datenabhängigkeiten und die Minimierung von Registerlebenszeiten verbessert worden. Hinzu kommt die Unterstützung von Zeitvorgaben bezüglich der Ausführung einzelner Operationen.

Das IP-Modell von Achatz [Ac93] faßt die beiden Ansätze zusammen und erweitert das Modell hinsichtlich der Bausteinallokation. Zu diesen Erweiterungen gehören die Unterstützung multi-funktionaler Einheiten (ALUs) und die Behandlung von unterschiedlichen Ausführungszeiten von Operationen auf verschiedenen Bausteintypen.

Alle drei Modelle verwenden binäre Entscheidungsvariablen zur Zuordnung von Operationen zu Kontrollschritten. Formal wird jede Variable durch zwei Indizes gekennzeichnet ($x_{i,j} \in \{0,1\}$), welche jeweils für einen Kontrollschritt i bzw. eine Operation j stehen. Eine Operation j wird genau dann im Kontrollschritt i ausgeführt, wenn die entsprechende Variable auf 1 gesetzt ist ($x_{i,j} = 1$). Die Anzahl der IP-Variablen ergibt sich demnach aus dem Produkt der Anzahlen der Operationen und der zur Verfügung stehenden Kontrollschritte. Um diese Anzahl zu reduzieren, wird jeweils vor der Aufstellung der Gleichungssysteme eine ASAP-ALAP-Analyse („as soon as possible“, „as late as possible“) für die Operationen durchgeführt, welche den maximalen Bereich zur Ausführung der Operationen ermittelt und somit nicht-realisierbare Zuordnungen ausschließt.

1.2.2 Die Integration von Allocation und Binding

Auch die Integration der Aufgaben Allocation und Binding ist mittels Ganzzahliger Programmierung möglich. Das IP-Modell von Rim, Jain und De Leone [RiJaLe92] geht von einem vorgegebenen Scheduling aus und minimiert die Anzahl der Verbindungen und der benötigten Multiplexer für den Entwurf.

Auch dieser IP-Ansatz basiert auf binären Entscheidungsvariablen, die in diesem Fall die Zuordnung von Operationen zu ausführenden Funktionseinheiten repräsentieren. Zusätzlich werden Verbindungsvariablen benötigt, deren Anzahl prinzipiell quadratisch zur Anzahl der eingesetzten Bausteine ist, aber durch Berücksichtigung von Typ-Konflikten (ein Addierer kann z. B. nicht multiplizieren) reduziert werden kann.

1.2.3 Die Integration von Scheduling, Allocation und Binding

Für eine optimale Mikroarchitektursynthese ist die Integration aller drei Teilaufgaben notwendig. Diese läßt sich durch eine Kombination der obigen Ansätze erreichen. Scheduling, Allocation und Binding lassen sich in einem Modell der Ganzzahligen Programmierung vereinigen, wenn statt Variablen mit zwei Indizes dreifach indizierte IP-Variablen verwendet werden.

Gebotys und Elmasry stellen in [GeEl92] und [GeEl93] erstmals ein IP-Modell vor, welches das Binding in den Ansatz [GeEl91] integriert, und somit alle drei Aufgaben gleichzeitig löst. Dieses Verfahren zur High-Level Synthese liefert damit eine *global optimale Lösung* bezüglich der definierten Kostenfunktion, welche die Anzahl der funktionalen Bausteine und der benötigten Register berücksichtigt. Die Möglichkeit der gleichzeitigen Verbindungsoptimierung wird hier nicht beschrieben.

Das Synthesesystem MARMOR[5] [Marmor93] integriert die Verbindungsminimierung von Rim, Jain und De Leone in das obige Modell. Das System verwendet dreifach indizierte Variablen zur Zuordnung von Operationen zu Kontrollschritten und ausführenden Bausteinen und Variablen mit zwei Indizes zur Verbindungsminimierung.

1.3 Das OSCAR-System

Das Synthesesystem OSCAR[6] [LaMaDö94a], [LaMaDö94b] basiert auf den IP-Modellen [GeEl93] und [RiJaLe92] und kann als konsequente Weiterentwicklung des MARMOR-Ansatzes betrachtet werden. Ausgehend von den im MARMOR-System verwendeten Vorschriften ist das Modell der Ganzzahligen Programmierung in wesentlichen Punkten verbessert und erweitert worden.

[5] MARMOR, Microarchitektursynthese mit Hilfe von Methoden des Operations Research
[6] OSCAR, Optimum Simultaneous Scheduling, Allocation and Resource Binding

Eine zentrale Verbesserung stellt die Unterstützung komplexer Bausteinbibliotheken dar, welche sowohl Komponenten mit unterschiedlichen Ausführungszeiten und Funktionalitäten, als auch multi-funktionale (z. B. arithmetisch-logische Bausteine) und mehrstufige Einheiten (z. B. Multiplizierer-Akkumulator-Bausteine) enthalten können. Weiterhin unterstützt das Modell alternative Datenflußgraphen, den gegenseitigen Ausschluß von Operationen und allgemeines Chaining.

Das IP-Modell des OSCAR-Systems, welches den Kern der vorliegenden Arbeit bildet, ist im folgenden Kapitel ausführlich aufgeführt und beschrieben.

Kapitel 3 stellt den Ablauf der OSCAR-Synthese vor und beschreibt die Einhaltung von Randbedingungen und Spezifikationen, durch die der Anwender Einfluß auf die Synthese nehmen kann.

In Kapitel 4 werden Optimierungen vorgestellt, die den Synthesevorgang durch Elimination von Redundanzen und Einsatz von Heuristiken beschleunigen.

Kapitel 5 schließlich faßt die Möglichkeiten des OSCAR-Systems zusammen.

Kapitel 2

Das IP-Modell des OSCAR-Systems

Wie bereits in der Einleitung beschrieben, basiert der Kern des OSCAR-Synthesesystems ([LaMaDö94a], [LaMaDö94b]) auf einem Modell der Linearen Ganzzahligen Programmierung. Das Modell ist in der Lage, die drei Teilaufgaben der High-Level Synthese (Scheduling, Allocation und Binding) *gleichzeitig* und damit *global optimal* zu lösen. Die Grundlagen dieses IP-Modells sollen in diesem Kapitel im Detail vorgestellt und untersucht werden.

Zunächst werden die verwendeten mathematischen Notationen eingeführt und die im IP-Modell eingesetzten Variablen definiert. Auf diesen mathematischen Grundlagen sollen dann die notwendigen Vorschriften der Reihe nach definiert und erläutert, aber auch auf mögliche Erweiterungen und Verbesserungen hin untersucht werden.

In Anhang A dieser Arbeit wird das in diesem Kapitel verwendete mathematische Modell noch einmal in tabellarischer Form zusammengefaßt.

2.1 Mathematische Notation

Das Grundprinzip der Mikroarchitektursynthese besteht darin, Operationen an Kontrollschritte und ausführende Funktionseinheiten zu binden und dabei gegebene Randbedingungen zu beachten. Um diese Aufgabe mit Methoden der Ganzzahligen Programmierung angehen zu können, ist ein mathematisch basiertes Modell notwendig. Dieses soll im folgenden aufgestellt werden.

Zur Repräsentation der *Operationen* wird in dieser Arbeit durchgehend die Menge $J \subset I\!N$ verwendet[1]. Einzelne Operationen werden mit $j \in J$ bezeichnet. Die Menge aller Operationen eines Entwurfes ist gegeben durch $J = \{1, \ldots, j_{\max}\}$.

[1] Im Vergleich zu den Mengendefinitionen in [LaMaDö94a] werden hier die Basismengen von $I\!N$ abgeleitet (nicht von $I\!N_0$); die hier verwendete Definition stimmt mit den in der Implementierung verwendeten ID's (siehe Kapitel 3.3) überein, welche jeweils mit 1 beginnen.

Die Menge $G \subset I\!N$ steht für die *Operationstypen* (z. B. $G = \{g_+, g_-, g_*, \ldots\}$) und bildet somit die Grundmenge aller im System bekannter Funktionen. Jede Operation j ist nun von einem Typ $g \in G$, in Zeichen $f(j) \in G$, und repräsentiert eine Instanz des Operationstyps g.

Die zur Verfügung stehenden *Bausteintypen* (z. B. Addierer und Multiplizierer) werden mit $M \subset I\!N$ bezeichnet. Die *Funktionalität* eines Bausteins $m \in M$ wird durch die von ihm ausführbaren Operationstypen $G(m) \subseteq G$ bestimmt. Ein Baustein m mit $G(m) = \{g_+, g_-\}$ kann also Addition und Subtraktion ausführen und ist somit einsetzbar für alle Operationen $j \in J$ mit $f(j) = g_+$ oder $f(j) = g_-$. Äquivalent zum Begriff Bausteintyp wird in dieser Arbeit auch die Bezeichnung *Komponente* verwendet.

Da in einem Entwurf nun durchaus mehrere *Instanzen* eines Bausteintyps eingesetzt werden können (z. B. 3 Addierer und 2 Multiplizierer), wird die Menge der Bausteininstanzen $K \subset I\!N$ eingeführt. Die Anzahl $k_{\max}$ der in einem Design einsetzbaren Instanzen k muß im OSCAR-System vor der Synthese festgelegt werden. Aus dieser Menge werden im Laufe der Synthese die verwendeten Instanzen ausgewählt.

Jedes $k \in \{1, \ldots, k_{\max}\}$ stellt also eine Instanz eines Bausteins $m \in M$ dar. Der Typ m einer Instanz k wird mit $m = \text{type}(k)$ bezeichnet. Ohne Beschränkung der Allgemeinheit kann die Menge der Instanzen K nach den Bausteintypen aus M sortiert werden, so daß gilt:

$$\forall k \in K \setminus \{k_{\max}\} : \text{type}(k) \leq \text{type}(k+1)$$

Die Abbildung $m = \text{type}(k)$ wird daher im folgenden als eine monoton steigende Funktion angenommen.

Eine Operation j ist nun genau dann ausführbar auf einer Instanz k, wenn $G(j)$ ein Element der Funktionalität des Bausteins $m = \text{type}(k)$ ist. Zur Vereinfachung soll hier die *Funktionalität* einer Instanz analog zur Bausteinfunktionalität definiert werden: $F(k) := G(\text{type}(k))$. Somit kann eine Instanz k eine Operation j ausführen, wenn $f(j) \in F(k)$ ist.

Die Menge $I \subset I\!N$ wird definiert als die Menge aller für das Design erlaubter *Kontrollschritte* $i \in \{1, \ldots, i_{\max}\}$. Es stehen also genau $i_{\max}$ Schritte zur Verfügung, um das gewünschte Gesamtverhalten des Entwurfs zu realisieren.

Zur Bestimmung des Kontrollschrittbereiches, in dem eine Operation j auf einer Instanz k gestartet werden kann, müssen sowohl Datenabhängigkeiten zwischen Operationen, als auch zeitliche Randbedingungen beachtet werden. Zunächst sollen nur Datenabhängigkeiten von Operationen untereinander betrachtet werden: eine Operation j_2 ist *datenabhängig* von einer Operation j_1, in Zeichen $j_1 \prec j_2$, wenn j_2 Daten benötigt, welche von j_1 erzeugt werden. Operation j_2 kann also erst gestartet werden, wenn j_1 beendet ist.

Es ist notwendig, den maximalen Zeitrahmen (ASAP-ALAP-*Range*) für eine Operation zu definieren: mit $\text{ASAP}(j) \in I$ wird der früheste Kontrollschritt bezeichnet, in dem die Operation j gestartet werden kann. Analog dazu wird

mit ALAP$(j) \in I$ der letzte mögliche Kontrollschritt für Operation j definiert. Damit kann nun der Bereich zur Ausführung der Operation j definiert werden: $R(j) := \{ASAP(j), ASAP(j)+1, \ldots, ALAP(j)\}$.

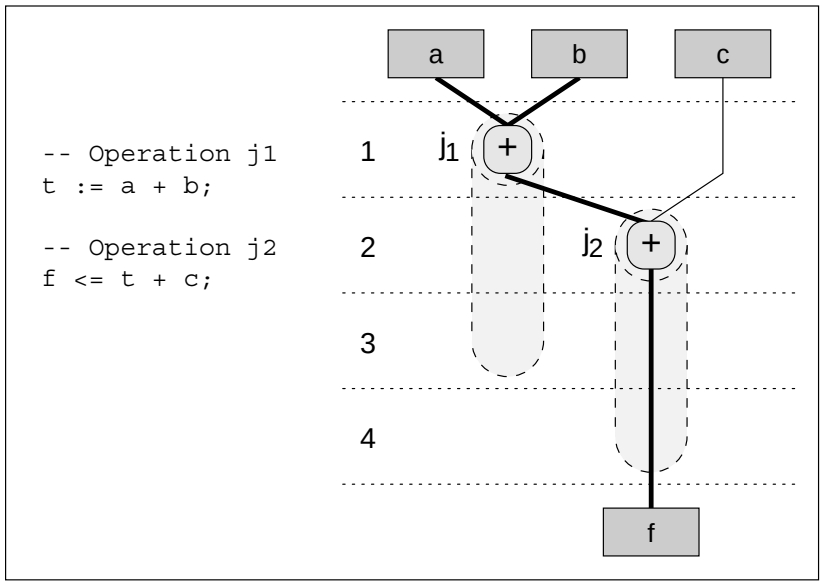

Abbildung 2.1: Datenflußbeispiel mit zwei Operationen

Abbildung 2.1 zeigt ein Beispiel mit zwei Operationen $j_1 \prec j_2$. Legt man hier $i_{\max} = 4$ Kontrollschritte zugrunde und nimmt die Ausführungszeit einer Addition mit 1 an, so ergibt sich $R(j_1) = \{1,2,3\}$ und $R(j_2) = \{2,3,4\}$.

Im allgemeinen kann man aber nicht davon ausgehen, daß jede Operation auf einem Baustein innerhalb *eines* Kontrollschrittes ausgeführt werden kann. Insbesondere sollen in diesem Modell auch *langsame* (Ausführungszeit $>$ 1 CS) und *schnelle* Komponenten (Ausführungszeit = 1 CS) unterstützt werden.

Dazu wird die *Ausführungszeit* einer Operation j auf einer Instanz k definiert: $C(j,k)$ gibt im folgenden die Zeit in Kontrollschritteinheiten (CS) an, die der Baustein $m = \text{type}(k)$ zur Ausführung der Operation j benötigt.

Betrachtet man nun noch einmal Abbildung 2.1 und nimmt zwei verschiedene Addierer als mögliche Bausteininstanzen an, z. B. ein schneller Addierer k_1 mit $C(j,k_1) = 1$ und ein langsamer Addierer k_2 mit $C(j,k_2) = 3$, dann ergeben sich folgende *von Instanzen abhängige* Kontrollschrittbereiche: $R(j_1,k_1) = \{1,2,3\}$, $R(j_1,k_2) = \{1\}$, $R(j_2,k_1) = \{2,3,4\}$ und $R(j_2,k_2) = \{2\}$.

Bei Einsatz des langsamen Bausteins k_2 verschieben sich die ALAP-Zeitpunkte jeweils um $C(j,k_2) - 1$ Kontrollschritte, d. h. die Kontrollschrittbereiche verkürzen sich um die Zeit, die der langsame Baustein zusätzlich benötigt. Eine Ausführung der Operation j_1 nach Schritt 1 oder der Start der Operation j_2 nach Schritt 2 würde auf dem langsamen Addierer k_2 unweigerlich zu einer Verletzung des Kontrollschrittmaximums $i_{\max} = 4$ führen (ein Ergebnis läge in Schritt 4 noch nicht vor).

Allgemein muß der ALAP-Zeitpunkt also abhängig von einer ausführenden Instanz bestimmt werden (ALAP(j,k)) und entsprechend auch der Kontrollschrittbereich einer Operation j von einer Instanz k abhängen ($R(j,k)$). Nur so läßt sich verhindern, daß Operationen auf langsamen Bausteinen zu spät gestartet werden.

2.2 Die Variablen des IP-Modells

Nachdem nun die grundlegenden mathematischen Notationen festgelegt sind, sollen die *Variablen* für die Ganzzahlige Programmierung definiert werden. Diese Variablen stellen die Freiheiten dar, die bei der Synthese des gegebenen Entwurfs (unter Einhaltung aller gegebenen Randbedingungen) ausgenutzt werden können.

2.2.1 Scheduling und Binding

Zuerst sollen die für die Grundaufgabe der High-Level Synthese, die Zuordnung von Operationen zu Kontrollschritten (Scheduling) und ausführenden Bausteinen (Binding), benötigten Variablen eingeführt werden. Das IP-Modell des OSCAR-Systems verwendet hierzu binäre Entscheidungsvariablen. Um die drei Teilaufgaben Scheduling, Allocation und Binding gleichzeitig zu lösen, werden dreifach indizierte Variablen benötigt.

$$\forall j \in J, \quad \forall k \in K \text{ mit } f(j) \in F(k), \quad \forall i \in R(j,k):$$

$$x_{i,j,k} \quad = \quad \begin{cases} 1, & \text{falls Operation } j \text{ auf Instanz } k \\ & \text{im Kontrollschritt } i \text{ gestartet wird} \\ 0, & \text{sonst} \end{cases} \tag{2.1}$$

Für jede Operation j wird ein Satz von Variablen benötigt, der die Wahlfreiheit dieser Operation j repräsentiert. Zum einen müssen für jeden Baustein k, auf dem j ausgeführt werden kann, Variablen existieren, zum anderen auch für jeden Kontrollschritt i aus dem für j möglichen Bereich $R(j,k)$.

Es werden also nur Variablen verwendet, die auch eine zulässige Lösung ermöglichen. Dadurch wird die Anzahl der IP-Variablen und der aufzustellenden Gleichungen klein gehalten. Insbesondere reduziert sich auch der Suchraum für den eingesetzten IP-Solver und damit die Zeit zur Berechnung einer optimalen Lösung.

2.2.2 Allocation

Das Ziel der OSCAR-Synthese ist es, die Kosten eines Entwurfs zu minimieren, d. h. insbesondere die Anzahl verwendeter Bausteine klein zu halten. Da vor der Synthese eines Entwurfs nicht bekannt ist, wieviele Instanzen von Bausteinen

benötigt werden, muß ein ausreichend großes Angebot an Bausteinen bereitgestellt werden. Ein naiver, aber korrekter Ansatz könnte für jede Operation eigene Instanzen von Bausteinen zur Verfügung stellen. In Abschnitt C.3.1 dieser Arbeit wird das im OSCAR-System verwendete Verfahren zur Berechnung einer möglichst kleinen Instanzenmenge vorgestellt.

Aus diesem Angebot von Instanzen muß während der Synthese eine Auswahl getroffen werden. Für diese Allokation von Bausteininstanzen k werden wiederum Entscheidungsvariablen verwendet:

$$\forall k \in K: \quad b_k = \begin{cases} 1, & \text{falls Bausteininstanz } k \text{ verwendet wird} \\ 0, & \text{sonst} \end{cases} \tag{2.2}$$

Jeder Instanz k wird also eine Allokationsvariable b_k zugeordnet, die nach der Berechnung angibt, ob k benötigt wird.

2.2.3 Verbindungsminimierung

Das hier vorgestellte Modell ist auch in der Lage, die Anzahl und die Breite der Verbindungen zwischen Bausteininstanzen zu minimieren. Hierzu werden Verbindungsvariablen eingeführt:

$$\forall k_1, k_2 \in K \text{ mit } \exists j_1, j_2 \in J,\ j_1 \prec j_2,\ f(j_1) \in F(k_1),\ f(j_2) \in F(k_2): \quad w_{k_1,k_2} = \begin{cases} 1, & \text{falls eine Verbindung von Instanz } k_1 \\ & \text{zu Instanz } k_2 \text{ existiert } (k_2 \text{ verarbeitet} \\ & \text{Daten, die von } k_1 \text{ erzeugt werden}) \\ 0, & \text{sonst} \end{cases} \tag{2.3}$$

Auch hier werden die Variablen auf die möglichen Lösungen beschränkt. Indexpaare k_1 und k_2 existieren nur, falls es datenabhängige Operationen j_1 und j_2 gibt, die auf den Instanzen k_1 und k_2 ausführbar sind.

2.2.4 Alternative Datenflußgraphen

Im OSCAR-System soll es auch möglich sein, erst während der Synthese zwischen alternativen Entwürfen zu entscheiden und die global kostengünstigste Version auszuwählen. Hierzu werden vor der eigentlichen Synthese algebraische Transformationen [LaMaDö94b] auf die Datenflußgraphen (DFGs) angewandt, so daß äquivalente Alternativen entstehen, welche zusätzlich zu den vom Designer spezifizierten Graphen in die Synthese einfließen.

Die Menge $D \subset I\!N$ bezeichnet im folgenden die Menge aller *Datenflüsse* des Entwurfs. Ein solcher Datenfluß $d \in D$ kann nun ein oder mehrere *Versionen* $v \in V(d) = \{1, \ldots, v_{\max}(d)\}$ enthalten, von denen während der Synthese genau eine auszuwählen ist.

Für diese Auswahl werden folgende Variablen definiert:

$$\forall d \in D, \quad \forall v \in V(d) \text{ mit } \exists v' \in V(d), v' \neq v \ : \\ u_{d,v} \quad = \quad \begin{cases} 1, & \text{falls aus den Alternativen des Datenflusses} \\ & d \text{ Version } v \text{ ausgewählt wird} \\ 0, & \text{sonst} \end{cases} \tag{2.4}$$

Auch diese Variablen existieren nur, wenn es zwei oder mehr alternative Versionen zu einem Datenfluß gibt.

2.2.5 Registeroptimierung

Zur Unterstützung von Registeroptimierung wird nur eine weitere Variable benötigt. Im Gegensatz zu den bisher definierten binären Entscheidungsvariablen wird hierzu jedoch eine uneingeschränkt ganzzahlige Variable verwendet:

$$r \quad = \quad \begin{array}{l} \text{Anzahl benötigter Register zur Aufnahme} \\ \text{von Zwischenergebnissen} \end{array} \tag{2.5}$$

2.3 Die Bewertungsfunktion

Die Bewertungsfunktion (*Objective Function*) in der Ganzzahligen Programmierung besteht aus einer Summe von IP-Variablen, welche durch beliebige Faktoren gewichtet werden können. Ziel des Verfahrens ist es, diese Summe zu optimieren, d. h. sie minimal bzw. maximal werden zu lassen.

Im Fall des OSCAR-Systems besteht die Bewertungsfunktion aus den aufsummierten *Kosten* des Entwurfs, welche zu minimieren sind.

2.3.1 Die Kostenfunktion

Die Kosten des Entwurfs setzen sich aus drei Gruppen zusammen. Jedes Element aus diesen Gruppen trägt mit unterschiedlichem Gewicht zu den Gesamtkosten bei.

Zur Skalierung der Einzelkosten werden folgende Konstanten verwendet:

c_m	*Bausteinkosten:* Kosten des Bausteins m, z. B. der Platzbedarf auf dem Chip
c_k	*Instanzkosten:* Kosten der Instanz k, i. d. R. gleich den Kosten c_m des Bausteins vom Typ $m = \text{type}(k)$
c_{k_1,k_2}	*Verbindungskosten:* Kosten der Verbindung von Instanz k_1 zu Instanz k_2, z. B. proportional zur benötigten Bitbreite
c_r	*Registerkosten:* Kosten eines Registers zur Aufnahme von Zwischenergebnissen

Die Kosten c_k einer Instanz leiten sich i. d. R. direkt aus den Kosten c_m des entsprechenden Bausteintyps ab. Durchaus sinnvoll kann es aber auch sein, die Kosten für jede Instanz individuell festzulegen. Zum Beispiel könnten die ersten zwei Addierer kostengünstiger in der Geometrie des Chip-Layouts unterzubringen sein, als alle weiteren. In der Synthese gilt es dann zwischen den teureren Addierern oder weiteren Verbindungs- und Registerkosten abzuwägen.

Die Kosten des Entwurfs ergeben sich nun wie folgt:

$$\underbrace{\sum_{k \in K} c_k * b_k}_{\text{Bausteinkosten}} + \underbrace{\sum_{k_1,k_2 \in K} c_{k_1,k_2} * w_{k_1,k_2}}_{\text{Verbindungskosten}} + \underbrace{c_r * r}_{\text{Registerkosten}} \tag{2.6}$$

Im OSCAR-Synthesesystem ist die Verbindungsoptimierung wie auch die Registeroptimierung optional, so daß sich die Kostenfunktion ggf. auf die Summe der Bausteinkosten beschränkt.

Der Übersichtlichkeit halber ist die Summe der Verbindungskosten über alle k_1, k_2 aufgeführt. Für die Indizes gelten hier dieselben Einschränkungen wie sie in Gleichung 2.3 aufgeführt sind.

2.3.2 Die Kostenbeschränkung

In vielen Fällen ist bereits im voraus bekannt, daß die Kosten eines Entwurfs ein bestimmtes Maximum nicht überschreiten. Im Fall des OSCAR-Systems ist das immer dann der Fall, wenn bereits ein Syntheselauf erfolgreich verlaufen ist und die Parameter für den neuen Lauf nur dahingehend geändert werden, daß sich der Lösungsraum vergrößert (z. B. durch Erhöhung der zulässigen Kontrollschrittanzahl).

Ist ein solches Kostenmaximum $c_{\max}$ bekannt, sollte es in das Gleichungssystem eingebracht werden. Hierzu reicht eine einzige Gleichung (*Cost Limit Constraint*) aus, die sich direkt aus der obigen Kostenfunktion ergibt.

$$\sum_{k \in K} c_k * b_k + \sum_{k_1,k_2 \in K} c_{k_1,k_2} * w_{k_1,k_2} + c_r * r \leq c_{\max} \tag{2.7}$$

Für die Synthese selbst ist diese Vorschrift nicht notwendig, sie beschleunigt aber in vielen Fällen die Berechnung der optimalen Lösung erheblich!

Als Beispiel für kürzere Berechnungszeiten soll der in Anhang D.1 vorgestellte Elliptical-Wave-Filter angeführt werden. Dieser wurde mit steigenden Kontrollschrittzahlen synthetisiert, jeweils *mit* und *ohne* Angabe einer maximalen Kostenschranke. Die angegebenen Maximalkosten ergeben sich jeweils aus den ermittelten Kosten des vorherigen Syntheselaufes, für den ersten Lauf (15 CS) werden 140 Kosteneinheiten angenommen.

Die erhaltenen Laufzeiten sind in Tabelle 2.1 aufgeführt[2]. In keinem der sechs Fälle ist eine Erhöhung der Berechnungszeiten zu verzeichnen. Die Laufzeiten

[2]Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec -a <steps> [-l <maxcosts>] elliptic oscar_behaviour`

Ohne Kostenlimit:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	2/5	2/5	2/4	2/4	2/4	1/4
Multiplizierer ($ 40)	1/3	0/3	1/2	1/2	1/2	0/2
ALUs ($ 50)	1/2	1/2	0/1	0/1	0/1	1/1
entstandene Kosten	$ 130	$ 90	$ 80	$ 80	$ 80	$ 70
Berechnungszeit	2s	1s	49s	171s	483s	315s
Mit Kostenlimit:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	2/5	2/5	2/4	2/4	2/4	1/4
Multiplizierer ($ 40)	1/3	0/3	1/2	1/2	1/2	0/2
ALUs ($ 50)	1/2	1/2	0/1	0/1	0/1	1/1
maximale Kosten	$ 140	$ 130	$ 90	$ 80	$ 80	$ 80
entstandene Kosten	$ 130	$ 90	$ 80	$ 80	$ 80	$ 70
Berechnungszeit	2s	1s	49s	97s	174s	310s

Tabelle 2.1: Vergleich der Laufzeiten *mit* und *ohne* Kostenlimit

reduzieren sich aber durch Angabe der Gleichung 2.7 in den Fällen 18 und 19 Kontrollschritte auf 57% bzw. 36%.

2.4 Die Operations-Zuordnungsvorschrift

Die Operations-Zuordnungsvorschrift (*Operation Assignment Constraint*, [GeEl92]) schreibt vor, wie die im Entwurf vorkommenden Operationen an Kontrollschritte und ausführende Instanzen zu binden sind.

2.4.1 Allgemeine Vorschrift

Im einfachsten Fall muß jede Operation j an genau einen Kontrollschritt i und genau eine Bausteininstanz k gebunden werden. Folgende Gleichung schreibt dieses vor:

$$\forall j \in J : \sum_{\substack{k \in K: \\ f(j) \in F(k)}} \sum_{i \in R(j,k)} x_{i,j,k} = 1 \tag{2.8}$$

Insbesondere impliziert diese Vorschrift, daß für jede Operation j mindestens ein Baustein k zur Verfügung stehen muß, der j ausführen kann ($f(j) \in F(k)$). Ist das nicht der Fall, so wird obige Gleichung unerfüllbar (die Summe über k ist leer und es ergibt sich $0 = 1$).

2.4.2 Erweiterung für Makrooperationen

Moderne Bausteinbibliotheken stellen für den Entwurf auf der Register-Transfer-Ebene nicht nur (*einfache*) Bausteine wie Addierer und Multiplizierer zur

Verfügung. Vergleichbar zum Entwurf auf der Logik-Ebene, wo nicht nur AND-, OR- und NOT-Gatter, sondern auch eine ganze Reihe komplexer Gatter (z. B. AND-OR-Gates) angeboten werden, bieten viele Bibliotheken zur High-Level Synthese zum Beispiel auch MAC[3]-Bausteine an.

Um nun auch solche *komplexen* Bausteine zu unterstützen, werden im OSCAR-System sog. *Makrooperationen* verwendet. Eine Makrooperation besteht aus zwei oder mehr *einfachen* Operationen und kann alternativ auf einem entsprechenden komplexen, oder einzelnen einfachen Bausteinen ausgeführt werden.

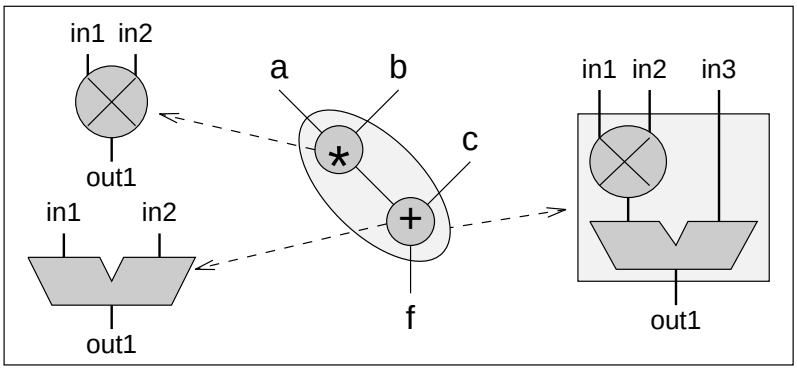

Abbildung 2.2: Alternative Zuweisung von Makrooperationen

Abbildung 2.2 zeigt die Verwendung von Makrooperationen am Beispiel einer Multiplikation mit nachfolgender Addition. Eine entsprechende Bausteinbibliothek vorausgesetzt, bestehen hier zwei Möglichkeiten, diese Operationen auszuführen. Zum einen können unabhängig voneinander die Addition und die Multiplikation auf einem Addierer bzw. einem Multiplizierer ausgeführt werden. Zum anderen ist aber auch die Bindung beider Operationen gemeinsam an einen komplexen MAC-Baustein möglich.

Ein Beispiel für das Vorkommen solcher Makrooperationen liefert der 5th-Order Elliptical-Wave-Filter (siehe Abb. D.1 im Anhang). In dessen Datenflußgraphen kommen acht Multiplikationen mit folgender Addition vor. Zusätzlich treten eine Reihe einzelner Additionen auf, so daß im vorhinein nicht zu entscheiden ist, ob der Einsatz einfacher Multiplikations- und Additions- oder komplexer MAC-Bausteine kostengünstiger ist.

Zur Notation von Makrooperationen soll die Menge $Y \subset J$ verwendet werden. Die Menge der durch eine Makrooperation $y \in Y$ abgedeckten, *einfachen* Operationen wird mit $\text{macro}(y) \subseteq J \setminus Y$ bezeichnet.

Gleichung 2.9 modelliert nun die Wahlfreiheit, geeignete Operationen an einfache Bausteine oder als Makrooperationen an komplexe Komponenten zu binden.

$$\forall j \in J \setminus Y:$$

$$\sum_{\substack{k \in K: \\ f(j) \in F(k)}} \sum_{i \in R(j,k)} x_{i,j,k} + \sum_{\substack{y \in Y: \\ j \in \text{macro}(y)}} \sum_{\substack{k \in K: \\ f(y) \in F(k)}} \sum_{i \in R(y,k)} x_{i,y,k} = 1 \qquad (2.9)$$

[3]MAC, Abk. für „Multiplier-Accumulator“, kombinierter Multiplizierer-Addierer

Falls keine Makrooperationen im Entwurf enthalten sind, ist die Summe über y leer und es ergibt sich direkt Gleichung 2.8. Sind Makrooperationen vorhanden, so können diese entweder als einfache Operationen (die linke Summe ergibt 1), oder als komplexe Operation (die rechte Summe wird zu 1) realisiert werden.

In [LaMaDö94a] wird zusätzlich folgende Gleichung angegeben:

$$\forall y \in Y : \sum_{\substack{k \in K: \\ f(y) \in F(k)}} \sum_{i \in R(y,k)} x_{i,y,k} \leq 1 \tag{2.10}$$

Diese Vorschrift stellt Gleichung 2.8 für Makrooperationen dar, berücksichtigt aber, daß Makrooperationen nicht ausschließlich als komplexe Operationen ausgeführt werden müssen (Summe ≤ 1, nicht $= 1$). Die Gleichung wird notwendig, falls die Mengen einfacher und komplexer Operationen unabhängig voneinander definiert werden. Mit der hier verwendeten Definition der Makrooperationen Y als *Untermenge* aller Operationen J wird diese Vorschrift redundant, da alle Makrooperationen bereits in Gleichung 2.9 vorkommen.

2.4.3 Erweiterung für alternative Versionen

Wie bereits in Abschnitt 2.2.4 angedeutet wurde, ist es im OSCAR-System möglich, mittels algebraischer Transformationen alternative Versionen von Datenflußgraphen in die Synthese einzubringen. Erst während der Synthese soll das System automatisch entscheiden, welche Alternative in anbetracht der Gesamtkosten am günstigsten verwirklicht werden kann.

Die Auswahl solcher Versionen kann durch eine letzte Erweiterung der Operations-Zuordnungsvorschrift modelliert werden. Hierzu wird jede Operation j einem Datenflußgraphen $d(j) \in D$ und einer Version $v(j) \in V(d)$ zugeordnet. Auf der rechten Seite der Gleichung 2.9 wird die Konstante 1 durch die Entscheidungsvariable $u_{d,v}$ ersetzt. Diese gibt nach der Synthese an, welche Version v des Datenflußgraphen d selektiert wurde. Weiterhin wird die Aufzählung aller Operationen j nach ihrer Datenfluß- $(d(j))$ und Versionszugehörigkeit $(v(j))$ geordnet.

Es ergibt sich:

$$\forall d \in D, \quad \forall v \in V(d), \quad \forall j \in J \setminus Y \text{ mit } v(j) = v, d(j) = d :$$
$$\sum_{\substack{k \in K: \\ f(j) \in F(k)}} \sum_{i \in R(j,k)} x_{i,j,k} + \sum_{\substack{y \in Y: \\ j \in \mathrm{macro}(y)}} \sum_{\substack{k \in K: \\ f(y) \in F(k)}} \sum_{i \in R(y,k)} x_{i,y,k} = u_{d,v} \tag{2.11}$$

Eine Operation j (bzw. Makrooperation y) wird also nur dann an einen Kontrollschritt i und einen Baustein k gebunden, wenn $u_{d,v} = 1$ ist, also ihre Version $v(j)$ selektiert wird.

Der gegenseitige Ausschluß der Versionen v eines Datenflußgraphen d kann modelliert werden durch:

$$\forall d \in D : \sum_{v \in V(d)} u_{d,v} = 1 \tag{2.12}$$

Es ist also genau eine Version aus den Alternativen eines Datenflußgraphen auszuwählen.

Abbildung 2.3 zeigt ein Beispiel mit drei Versionen des Ausdrucks:

$$(2+b)*a \quad \Leftrightarrow \quad 2*a+a*b \quad \Leftrightarrow \quad \mathrm{shl}(a,1)+b*a$$

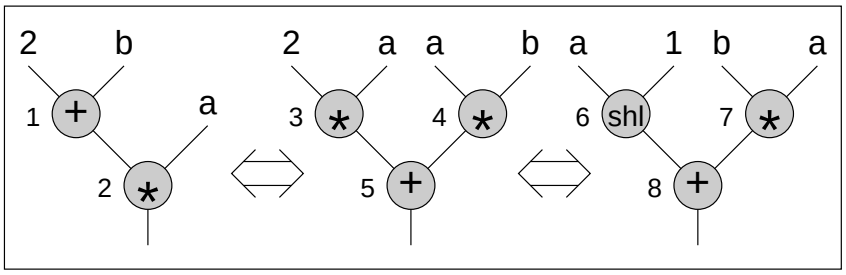

Abbildung 2.3: Alternative Versionen eines Datenflußgraphen

Alle Versionen sind funktional gleichwertig, können aber in der Synthese zu unterschiedlichen Gesamtkosten führen. Version 1 scheint günstiger als Version 2 zu sein, da sie nur zwei Operationen enthält. Diese sind aber in umgekehrter Reihenfolge auszuführen, was u. U. zu höheren Gesamtkosten führen könnte, weil z. B. kein Addierer in Schritt 1 mehr frei ist. Version 3 ersetzt eine Multiplikation durch eine Shift-Operation, wodurch ein neuer Bausteintyp notwendig wird. Ist ein solcher aber bereits vorhanden, z. B. durch Shift-Operationen an anderer Stelle des Entwurfs, so kann evtl. der Einsatz eines zweiten Multiplizierers vermieden werden.

Mit den Gleichungen 2.11 und 2.12 wird erst während der Synthese die kostengünstigste Version selektiert und so ein global optimaler Entwurf erzeugt.

2.5 Die Baustein-Zuordnungsvorschrift

Die Baustein-Zuordnungsvorschrift (*Resource Assignment Constraint*) regelt die Belegung der Instanzen. Jede Bausteininstanz k kann nur dann Operationen ausführen, wenn sie selektiert wird ($b_k = 1$). Weiterhin werden zur Berechnung einer Operation j auf Instanz k $C(j,k)$ Kontrollschritte benötigt.

Zur Unterstützung von Pipeline-Bausteinen wird die *Latenzzeit* definiert: $\ell(j,k)$ bezeichnet im folgenden den Abstand in Kontrollschritten, in dem eine Instanz k jeweils neue Daten zur Verarbeitung der Operation j übernehmen kann. Für Komponenten, die intern mit einer Pipeline arbeiten, gilt $\ell(j,k) < C(j,k)$.

Gleichung 2.13 stellt sicher, daß jeder Baustein k mit höchstens einer Operation j in $\ell(j,k)$ Kontrollschritten belegt wird.

$$\forall i \in I,\ \forall k \in K \text{ mit } \exists j \in J, f(j) \in F(k), i \in R(j,k):$$

$$\sum_{\substack{j \in J: \\ f(j) \in F(k)}} \sum_{\substack{i'=i \\ i \in R(j,k)}}^{i+\ell(j,k)-1} x_{i',j,k} \quad \leq \quad b_k \qquad (2.13)$$

Zur Synthese eines Entwurfs muß ein ausreichendes Angebot von Instanzen zur Verfügung stehen. Dieses Angebot wird im allgemeinen größer sein, als die tatsächlich benötigte Anzahl, so daß eine Auswahl getroffen werden muß. Vorschrift 2.14 (*Instance Allocation Constraint*) schränkt diese Auswahl dahingehend ein, daß die jeweils ersten Instanzen eines Bausteintyps verwendet werden.

$$\forall k \in K \setminus \{k_{\max}\}: \quad \text{type}(k) = \text{type}(k+1) \quad \Rightarrow \quad b_k \leq b_{k+1} \tag{2.14}$$

Die hierdurch erzwungene Lösung unterscheidet sich von den ohne Vorschrift 2.14 möglichen Lösungen nur durch eine Umbenennung der Bausteininstanzen. Es werden nur Lösungen mit gleichen Kosten aus dem Lösungsraum herausgenommen, es ändert also nichts an der globalen Optimalität des Modells.

Diese Einschränkung des Lösungsraumes reduziert entsprechend den Suchraum des IP-Solvers und führt schon bei kleinen Entwürfen zu erheblichen Zeitgewinnen!

Als Beispiel soll wieder der in Anhang D.1 vorgestellte Elliptical-Wave-Filter dienen. In Tabelle 2.2 sind die Ergebnisse angegeben, die sich *mit* und *ohne* Vorschrift 2.14 ergeben[4].

Ohne Einschränkung 2.14:	15 CS	16 CS	17 CS	18 CS	19 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4
Berechnungszeit	1s	17s	22s	253s	426s
Mit Einschränkung 2.14:	15 CS	16 CS	17 CS	18 CS	19 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4
Berechnungszeit	1s	2s	7s	71s	175s

Tabelle 2.2: Vergleich der Laufzeiten *mit* und *ohne* Vorschrift 2.14

Die gemessenen Geschwindigkeitssteigerungen sind deutlich zu erkennen und bedürfen keiner weiteren Interpretation.

2.6 Die Vorrangsvorschrift

Die Vorrangsvorschrift (*Precedence Constraint*, [GeEl92]) berücksichtigt im IP-Modell die in den Datenflußgraphen eines Entwurfs explizit enthaltenen Datenabhängigkeiten.

Für jede Abhängigkeit zweier Operationen $j_1 \prec j_2$ ist sicherzustellen, daß Operation j_2 nicht vor Beendigung von Operation j_1 gestartet wird.

[4] Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec -a <steps> [-De] elliptic oscar_behaviour`;
Zur Ermittlung der Rechenzeiten ohne Einschränkung 2.14 wurden die entsprechenden Gleichungen jeweils in der Datei `elliptic.eqn` manuell durch $b_k \leq 1$ ersetzt.

2.6.1 Die im OSCAR-System verwendete Vorschrift

Zur Behandlung der Datenabhängigkeiten sind verschiedene Formulierungen der Vorrangsvorschrift möglich. Folgende Vorschrift 2.15 stellt im OSCAR-Synthesesystem die Einhaltung der Datenabhängigkeiten sicher:

$$
\begin{aligned}
&\forall j_1, j_2 \in J \text{ mit } j_1 \prec j_2 \ , \\
&\forall i \in \quad \{\mathrm{ASAP}(j_2) + \mathrm{chain}(j_1, j_2), \ldots, \mathrm{ALAP}(j_2)\} \\
&\qquad \cap \quad \{\mathrm{ASAP}(j_1), \ldots, \mathrm{ALAP}(j_1) + C(j_1) - 1\} \quad : \\
&\qquad\qquad \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2,k_2): \\ i_2 \le i - \mathrm{chain}(j_1,j_2)}} x_{i_2,j_2,k_2} \quad + \\
&\qquad\qquad\qquad \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \ \sum_{\substack{i_1 \in R(j_1,k_1): \\ i - (C(j_1,k_1)-1) \le i_1}} x_{i_1,j_1,k_1} \quad \le \quad 1 \qquad (2.15)
\end{aligned}
$$

Für jeden Kontrollschritt i aus der Überschneidung der Kontrollschrittbereiche der beiden Operationen j_1 und j_2 wird eine Gleichung aufgestellt. Diese garantiert, daß Operation j_2 nicht vor Schritt i gestartet wird, falls Operation j_1 in Schritt i noch nicht beendet ist. Die Beendigung von Operation j_1 ergibt sich aus dem Startkontrollschritt und der Ausführungszeit $C(j_1, k_1)$. Eben diese Ausführungszeit muß auch bei der Überschneidung der Kontrollschrittbereiche $R(j_1)$ und $R(j_2)$ berücksichtigt werden. Im Gültigkeitsbereich der Vorschrift 2.15 ist aber nicht bekannt, auf welcher Instanz k die Operation j_1 ausgeführt wird ($C(j_1, k)$ ist nicht konstant). Es kann aber gezeigt werden, daß hier die maximale Ausführungszeit $C(j_1) := \max_k(C(j_1, k))$ eingesetzt werden kann.

In [LaMaDö94a] wird für diese Überschneidung $R(j_2) \cap (R(j_1) + C(j_1) - 1)$ angegeben. Hier ist jedoch die Addition der Ausführungszeit auf den Kontrollschrittbereich der Operation j_1 mißverständlich. $R(j_1)$ darf nicht um $C(j_1) - 1$ Schritte *verschoben*, sondern muß um diese Kontrollschritte *erweitert* werden. Die oben angegebene Notation vermeidet dieses Problem.

Zur Unterstützung von *Chaining* muß eine Ausführung der Operationen j_1 und j_2 im gleichen Kontrollschritt i gestattet sein. In Vorschrift 2.15 ist das genau dann der Fall, wenn $\mathrm{chain}(j_1, j_2) = 1$ ist. Setzt man $\mathrm{chain}(j_1, j_2) = 0$, so kann Operation j_2 frühestens einen Kontrollschritt nach Operation j_1 gestartet werden.

Im OSCAR-System wird $\mathrm{chain}(j_1, j_2)$ in einem Vorbereitungsschritt vor Aufstellen der Gleichungen für alle Kombinationen von $j_1, j_2 \in J$ ermittelt. Chaining wird erlaubt, falls die Summe der Ausführungszeiten von j_1 und j_2 kleiner als die Zykluszeit (abzüglich einer Systemverzögerung) ist. Abschnitt 2.10 behandelt detailliert die Unterstützung von allgemeinem Chaining.

Bei eingeschaltetem Chaining ($\mathrm{chain}(j_1, j_2) = 1$) entsteht für $i = \mathrm{ASAP}(j_2)$ eine leere Summe über i_2. Die übrigbleibende Gleichung enthält nur noch Terme über j_1 und ist somit redundant[5]. Diese Redundanz wird vermieden, wenn in

[5] Die Redundanz ergibt sich dadurch, daß die übrigbleibende Gleichung bereits in Vorschrift 2.8 (bzw. deren Erweiterungen) enthalten ist.

Vorschrift 2.15 die Menge $\{\text{ASAP}(j_2) + \text{chain}(j_1, j_2), \ldots, \text{ALAP}(j_2)\}$ und nicht der ganze Bereich $R(j_2)$ zur Schnittmengenbildung verwendet wird.

Bei Verwendung von *schnellen* und *langsamen* Bausteinen für dieselben Operationen tritt noch ein Problem auf: $\text{chain}(j_1, j_2)$ ist nicht mehr für alle $k_1, k_2 \in K$ konstant, sondern abhängig von den verwendeten Bausteininstanzen. Korrekterweise müßte $\text{chain}(j_1, k_1, j_2, k_2)$ verwendet werden. Nun ist aber k_1 in der Summe über i_2 nicht bekannt.

Eine Lösung dieses Problems besteht darin, die Vorrangsvorschrift mehrfach aufzustellen: Zunächst werden die Gleichungen mit $\text{chain}(j_1, j_2) = 1$ unverändert erzeugt, so daß Chaining mit sämtlichen Bausteinen möglich ist. Existieren nun langsame Instanzen $k_{slow} \in K$ für Operation j_1 (Ausführungszeit $C(k_{slow}, j_1) > 1$), so wird Vorschrift 2.15 noch einmal aufgestellt, und zwar mit eben diesen Instanzen k_{slow} (an Stelle von k_1) und abgeschaltetem Chaining.

Entsprechend werden die Gleichungen ein drittes Mal erzeugt, falls langsame Instanzen k_{slow} für Operation j_2 existieren.

2.6.2 Alternative Formulierungen der Vorrangsvorschrift

Im Vergleich zur obigen Vorrangsvorschrift 2.15 sollen noch zwei Alternativen betrachtet werden.

Die erste Alternative entspricht der oben angegebenen Vorschrift, verwendet aber *kürzere* Gleichungen. Vorschrift 2.15 wird nur dahingehend geändert, daß die Summe über i_2 beschränkt wird auf $i_2 = i - \text{chain}(j_1, j_2)$. Es bleibt also genau ein Element dieser Summe übrig, wodurch diese überflüssig wird.

Mit $i_2 = i - \text{chain}(j_1, j_2)$ ergibt sich für den Rumpf der verkürzten Vorrangsvorschrift:

$$\sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} x_{i_2, j_2, k_2} \; + \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \; \sum_{\substack{i_1 \in R(j_1, k_1): \\ i - (C(j_1, k_1) - 1) \leq i_1}} x_{i_1, j_1, k_1} \; \leq \; 1 \qquad (2.16)$$

Über alle Gleichungen betrachtet ändert sich die Lösungsmenge des Gleichungssystems hierdurch nicht. Auch diese Vorschrift schließt aus, daß Operation j_2 vor Beendigung von j_1 gestartet wird.

Der Vorteil der kürzeren Gleichungen (Speicherplatzersparnis) wirkt sich aber nachteilig auf die Berechnungszeiten aus.

Obwohl die Vorschriften 2.15 und 2.16 denselben ganzzahligen Lösungsraum zulassen, beschreibt Gleichung 2.15 diesen *enger*. Die erzeugten langen Gleichungen sind von sich aus *restriktiver* als die verkürzten Versionen.

Als Beispiel soll hier noch einmal der Elliptical-Wave-Filter (siehe Anhang D.1) angeführt werden. Tabelle 2.3 stellt die Syntheselaufzeiten mit Vorschrift 2.15 und der verkürzten Version 2.16 gegenüber[6].

Kurze Vorschriften:	15 CS	16 CS	17 CS	18 CS	19 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4
Gleichungsdateilänge	18538	30275	44266	61292	81510
Berechnungszeit	1s	5s	31s	91s	294s
Lange Vorschriften:	15 CS	16 CS	17 CS	18 CS	19 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4
Gleichungsdateilänge	19743	32474	49640	72070	100373
Berechnungszeit	1s	2s	7s	71s	175s

Tabelle 2.3: Laufzeiten mit *kurzen* und mit *langen* Vorrangsvorschriften

Die Speicherplatzersparnis durch kurze Gleichungen erweist sich als nur geringfügig und kann angesichts heutiger Speicherkapazitäten vernachlässigt werden. Andererseits sind die Geschwindigkeitsvorteile der langen Vorschriften erheblich!

In [HwLeHs91] wird eine weitere Alternative der Vorrangsvorschrift angegeben. Umgesetzt auf die in der vorliegenden Arbeit verwendete Notation lautet diese Vorschrift[7]:

$$\forall j_1, j_2 \in J \text{ mit } j_1 \prec j_2 :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \sum_{i_1 \in R(j_1,k_1)} i_1 * x_{i_1,j_1,k_1} \; - \; \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \sum_{i_2 \in R(j_2,k_2)} i_2 * x_{i_2,j_2,k_2} \quad \leq \quad -C(j_1) \tag{2.17}$$

Die linke Seite dieser Gleichung berechnet explizit den Kontrollschrittabstand der Operationen j_1 und j_2 (das linke Summenpaar gibt den Startkontrollschritt von j_1, das rechte Summenpaar den Schritt von j_2 an). Dieser Abstand kann mit Vorschrift 2.17 nun nicht kleiner als die Ausführungszeit $C(j_1)$ werden.

Vorschrift 2.17 benötigt für jede Datenabhängigkeit also nur eine einzige Gleichung!

[6] Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec -a <steps> elliptic oscar_behaviour;`
OSCAR erzeugt stets *lange* Vorrangsvorschriften; zur Ermittlung der Rechenzeiten mit *kurzen* Gleichungen wurde in der Programmdatei `ConstraintsMod.c` der Compiler-Schalter `EQ_GENERATE_SHORT_CONSTRAINTS` verwendet.

[7] Zur besseren Übersicht wird hier auf die Unterstützung von Chaining (durch chain(j_1, j_2)) und die Abhängigkeit der Ausführungszeiten von Bausteininstanzen (durch $C(j,k)$) verzichtet.

Dieses Vorgehen erscheint zwar sehr elegant, ist aber nachweislich nicht so effizient wie Vorschrift 2.15. Diese Ineffizienz beruht auf der Tatsache, daß diese Formulierung einen weniger *engen* Suchraum beschreibt, als er durch Vorschrift 2.15 gegeben ist. Ein formaler Beweis ist im Anhang von [GeEl93] angegeben.

Mit den bis hierher vorgestellten Gleichungen sind die grundlegenden Vorschriften des IP-Modells bereits vollständig: Das Basismodell besteht aus der Bewertungsfunktion 2.6 und den Vorschriften 2.8 (Zuordnung von Operationen), 2.13 (Zuordnung von Bausteinen) und 2.15 (Behandlung von Datenabhängigkeiten). In den folgenden Abschnitten sollen weitere, wichtige Erweiterungen dieses IP-Ansatzes betrachtet werden.

2.7 Die Zeitvorgabevorschriften

Zu den wichtigsten Randbedingungen, die bei der Synthese eines Entwurfs einzuhalten sind, zählen *zeitliche Bedingungen* (*Timing Constraints*, [GeEl91]). Insbesondere zur Realisierung von Protokollen sind Zeitvorgaben in Bezug auf die Ein- und Ausgabe von Daten einzuhalten.

Kontrollschrittbezogene Zeitabstände zwischen beliebigen Operationen sind in das hier beschriebene IP-Modell leicht zu integrieren. Das OSCAR-System unterscheidet drei Arten von zeitlichen Randbedingungen, welche im folgenden vorgestellt werden.

2.7.1 Konstante Zeitabstände

Sollen zwei Operationen j_1 und j_2 durch genau T Kontrollschritte voneinander getrennt werden, so kann dieses durch eine konstante Zeitvorgabe modelliert werden (*Constant Timing Constraint*, [GeEl92]):

$$\forall i_1 \in R(j_1) \; : \\ \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \; + \; \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \; \sum_{\substack{i_2 \in R(j_2): \\ i_2 \neq i_1 \pm T}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad (2.18)$$

Für jeden Startkontrollschritt i_1 der Operation j_1 verbietet diese Vorschrift quasi alle Schritte i_2 für Operation j_2, die nicht den Abstand T Kontrollschritte zu i_1 einhalten.

Auch die Ausführungszeiten der Operationen können hier einfließen, so daß Zeitbedingungen an die Beendigung von Operationen geknüpft werden können. Die jeweilige Ausführungszeit $C(j,k)$ kann einfach in der Einschränkung der Summe über i_2 auf den entsprechenden Kontrollschritt i addiert werden.

Für den Benutzer eines Synthesesystems ist es jedoch nur interessant, Zeitvorgaben für *Ein-* und *Ausgabeoperationen* (also Lese- und Schreiboperationen auf Ports) angeben zu können. Im OSCAR-System benötigen Schreib- und Leseoperationen niemals mehr als einen Kontrollschritt, so daß die Berücksichtigung

der Ausführungszeiten hier nicht notwendig ist. Aus demselben Grund werden hier auch von Instanzen unabhängige Kontrollschrittbereiche $R(j)$ verwendet.

Vorschrift 2.18 ist unabhängig von der Reihenfolge der Operationen j_1 und j_2. Die Angabe einer Zeitvorgabe für j_1 und j_2 ist äquivalent zu einem Zeitabstand für j_2 und j_1, die Vertauschung der beiden Operationen ist offensichtlich problemlos möglich. Diese Beobachtung soll im folgenden kurz daraufhin untersucht werden, ob sich diese Wahlfreiheit zugunsten der Berechnungszeit ausnutzen läßt.

Die Anzahl möglicher Kontrollschritte für Operation j_1 ($|R(j_1)|$) bestimmt die Anzahl erzeugter Gleichungen, $|R(j_2)|$ ist mitbestimmend für deren Länge. Durch entsprechendes Vertauschen der Operationsbezeichner j_1 und j_2 in einer gegebenen Zeitvorgabe kann die Art der Gleichungen bestimmt werden. Für $|R(j_1)| > |R(j_2)|$ erhält man *viele kurze*, im umgekehrten Fall *wenige lange* Gleichungen. Nun liegt die Vermutung nahe, daß auch hier lange Gleichungen zu schnelleren Berechnungen führen, wie das bei der Vorrangsvorschrift der Fall ist.

Wiederum soll der Elliptical-Wave-Filter als Beispiel dienen. Als Zeitbedingungen werden konstante Abstände von fünf Kontrollschritten zwischen den Operationen 1 und 4 und den Operationen 2 und 7 verlangt (vgl. Abb. D.1). Diese Operationen eignen sich hier besonders, da die Operationen 1 und 2 einen sehr kleinen ($R(1) = R(2) = \{1\}$), und 4 und 7 einen sehr großen Kontrollschrittbereich ($R(4) = R(7) = \{1, 2, \ldots, 11\}$) besitzen.

Kurze Vorschriften:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4	1/4
Gleichungen	245	332	415	497	584	672
entstandene Kosten	$ 140	$ 100	$ 80	$ 80	$ 80	$ 80
Berechnungszeit	1s	5s	7s	80s	152s	353s
Lange Vorschriften:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4	1/4
Gleichungen	225	312	395	477	564	652
entstandene Kosten	$ 140	$ 100	$ 80	$ 80	$ 80	$ 80
Berechnungszeit	1s	4s	8s	77s	157s	231s

Tabelle 2.4: Laufzeiten mit *kurzen* und mit *langen* Zeitabstandsvorschriften

Tabelle 2.4 bestätigt die Vermutung nur teilweise[8]. Es sind sowohl Steigerungen wie auch Einbußen der Geschwindigkeit zu verbuchen. Weitere (hier nicht

[8] Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec -a <steps> elliptic oscar_behaviour;`
OSCAR erzeugt stets *lange* Timing-Vorschriften; zur Ermittlung der Rechenzeiten mit *kurzen* Gleichungen wurde in der Datei `ConstraintsMod.c` die Vertauschung der von der Zeitvorgabe betroffenen Operationen invertiert.

aufgeführte) Beispiele lassen auch keinen klaren Vorteil langer Gleichungen erkennen. Dennoch werden im OSCAR-System zugunsten der Gleichungsanzahl Zeitvorschriften mit wenigen, langen Gleichungen erzeugt.

2.7.2 Minimale Zeitabstände

Analog zur Angabe konstanter Zeitbedingungen ist auch die Einhaltung *minimaler* Zeitabstände möglich. Die minimale Zeitvorgabevorschrift (*Minimum Timing Constraint*, [GeEl92]) leitet sich direkt aus Gleichung 2.18 ab:

$$\forall i_1 \in R(j_1) \; : \\ \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \; + \; \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \; \sum_{\substack{i_2 \in R(j_2): \\ (i_2 < i_1 + T) \\ \wedge (i_2 > i_1 - T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad (2.19)$$

Diese Gleichung erlaubt die Ausführung von Operation j_2 nur mit einem Abstand von mindestens T Kontrollschritten zu Operation j_1.

Berücksichtigt man die Lage der beiden Operationen zueinander, so läßt sich eine alternative Vorschrift verwenden, die der Vorrangsvorschrift 2.15 entspricht. Besteht zwischen den Operationen j_1 und j_2 eine Datenabhängigkeit, z. B. $j_1 \prec j_2$, oder läßt sich aus den Kontrollschrittbereichen $R(j_1)$ und $R(j_2)$ schliessen, daß j_1 stets vor j_2 ausgeführt wird, so kann folgende Vorschrift eingesetzt werden:

$$\forall i \in R(j_1) \; : \\ \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \; \sum_{\substack{i_1 \in R(j_1): \\ (i_1 \geq i)}} x_{i_1,j_1,k_1} \; + \; \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \; \sum_{\substack{i_2 \in R(j_2): \\ (i_2 < i + T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad (2.20)$$

Diese Gleichung stellt eine echte Verlängerung der Vorschrift 2.19 dar, da sie die Ausführungsreihenfolge der beiden Operationen berücksichtigt. Es ist daher zu erwarten, daß sie zu kürzeren Berechnungszeiten führt.

Verglichen mit den Vorrangsvorschriften entspricht Gleichung 2.20 der Vorschrift 2.15 und Gleichung 2.19 der verkürzten Vorrangsvorschrift 2.16. Die Verwandtschaft der Gleichungspaare ist offensichtlich.

Auch aus diesem Vergleich ergibt sich, daß Vorschrift 2.20 der verkürzten Version 2.19 nach Möglichkeit vorzuziehen ist. Auf eine Angabe von Laufzeiten soll daher verzichtet werden.

2.7.3 Maximale Zeitabstände

Analog zu den minimalen Zeitvorgaben ergeben sich die Vorschriften für maximale Zeitabstände (*Maximum Timing Constraint*, [GeEl92]). Der Vollständigkeit halber sollen auch diese Gleichungen noch angegeben werden.

Im allgemeinen Fall beschränkt folgende Vorschrift den Abstand zweier Operationen auf maximal T Kontrollschritte:

$$\forall i_1 \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 > i_1 + T) \\ \vee (i_2 < i_1 - T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad (2.21)$$

Ist die Ausführungsreihenfolge der Operationen j_1 und j_2 bekannt ($j_1 \prec j_2$ oder $\mathrm{ALAP}(j_1) \leq \mathrm{ASAP}(j_2)$), so sollte folgende Gleichung eingesetzt werden:

$$\forall i \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \ \sum_{\substack{i_1 \in R(j_1): \\ (i_1 \leq i)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 > i + T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad (2.22)$$

2.8 Die Verbindungsoptimierung

Gleichzeitig zur Minimierung der Kosten von Bausteininstanzen können mit dem hier vorgestellten Modell auch die *Verbindungskosten* minimiert werden. Damit lassen sich Leitungen (Verdrahtung) und notwendige Multiplexer im IP-Modell berücksichtigen.

Ziel der Verbindungsoptimierung ist es, die Anzahl und Bitbreite der benötigten Verbindungen minimal werden zu lassen. Insbesondere die Zuweisung von Operationen an ausführende Instanzen wird hierdurch dahingehend beeinflußt, daß benötigte Datenwege mehrfach ausgenutzt werden.

Für jede mögliche Verbindung zwischen Bausteininstanzen k_1 und k_2 wird eine Verbindungsvariable w_{k_1,k_2} benötigt. Diese Variable muß genau dann auf 1 gesetzt werden, wenn es zwei datenabhängige Operationen $j_1 \prec j_2$ gibt, die auf den Instanzen k_1 und k_2 ausgeführt werden.

Ein erster Ansatz könnte Gleichungen der Form

$$w_{k_1,k_2} := x_{i,j_1,k_1} * x_{i,j_2,k_2}$$

verwenden. Die Multiplikation zweier Variablen führt jedoch zu einem quadratischen Zuweisungsproblem, ist also mit einem Modell Linearer Programmierung unvereinbar.

Ein einfacher, aber wirkungsvoller Trick von Rim, Jain und De Leone [RiJaLe92] vermeidet mit Hilfe *zweier* Vorschriften (*Interconnection Constraints*) dieses Problem:

$$\forall j_1, j_2 \in J \text{ mit } j_1 \prec j_2,$$
$$\forall k_1, k_2 \in K \text{ mit } f(j_1) \in F(k_1), f(j_2) \in F(k_2) :$$

$$\left(\sum_{i_1 \in R(j_1,k)} x_{i_1,j_1,k_1} + \sum_{i_2 \in R(j_2,k)} x_{i_2,j_2,k_2} \right) - 1 \quad \leq \quad w_{k_1,k_2} \qquad (2.23)$$

$$0 \quad \leq \quad w_{k_1,k_2} \qquad (2.24)$$

Gleichung 2.23 setzt $w_{k_1,k_2} = 1$, wenn Operation j_1 auf Instanz k_1 und Operation j_2 auf Instanz k_2 ausgeführt wird (beide Summen liefern eine 1). Wird nur j_1 auf k_1 oder nur j_2 auf k_2 ausgeführt (nur eine Summe liefert eine 1), so ergibt sich $w_{k_1,k_2} = 0$. Gleichung 2.24 sorgt auch im Fall, daß beide Summen 0 ergeben, für $w_{k_1,k_2} = 0$.

2.9 Die Registeroptimierung

Zusätzlich zu den Kosten funktionaler Bausteine und den im vorherigen Abschnitt eingeführten Verbindungskosten tragen auch benötigte *Register* zu den Gesamtkosten des Entwurfs bei. In diesem Abschnitt soll das bisherige Modell dahingehend erweitert werden, daß eine Berücksichtigung notwendiger Register in der Kostenfunktion möglich wird[9].

Register werden eingesetzt, um berechnete Zwischenergebnisse über Kontrollschrittgrenzen hinaus zu speichern. In der Regel muß jeder von einer Bausteininstanz erzeugte Wert in einem Register gespeichert werden, so daß er in späteren Kontrollschritten weiterverarbeitet werden kann.

Innerhalb eines Datenflußgraphen werden temporäre Variablen für Zwischenergebnisse verwendet. Die *Lebenszeit* (*Lifetime*) einer solchen Variable wird bestimmt durch die Operation j_s, die die Variable beschreibt (Quelle oder *Source*), und die Operationen j_d, die den Wert der Variablen benötigen (Senke oder *Drain*). Das Ende der Lebenszeit ergibt sich also aus der Operation j_l, die als letzte von allen Operationen j_d ausgeführt wird.
Nebenstehende Abbildung 2.4 zeigt ein Beispiel für die Bestimmung der Lebenszeit eines Registers.

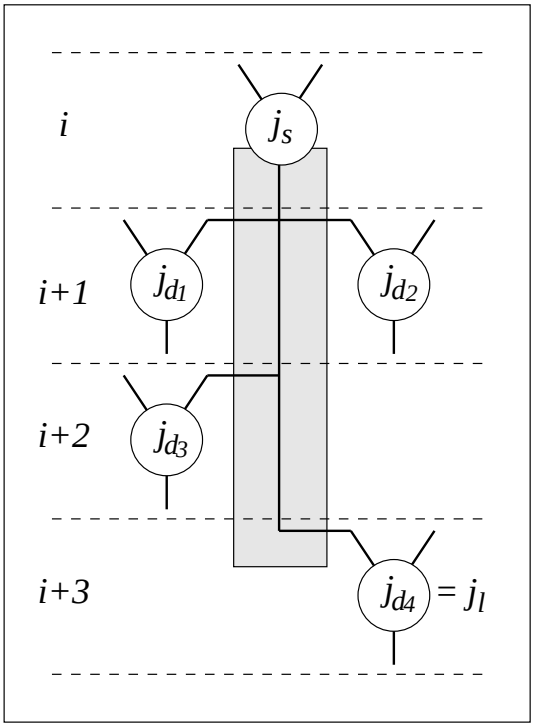

Abb. 2.4: Die Lebenszeit eines Registers

[9] Als einzige Erweiterung des IP-Modells ist die Registeroptimierung im OSCAR-System (noch) nicht implementiert. Auf Ergebnisse und Laufzeiten dieser Optimierung muß daher in dieser Arbeit verzichtet werden.

Im Abhängigkeitsgraphen des Entwurfs ist die Lebenszeit einer Variablen gegeben durch die Datenabhängigkeitskanten $j_s \prec j_d$, wobei j_d mit $d \in \{d_1, d_2 \ldots, l\}$ für alle Operationen steht, die von j_s datenabhängig sind[10].

Die Anzahl der für einen Entwurf benötigten Register ergibt sich nun aus der Summe der sich überschneidenden Lebenszeiten, d. h. gleichzeitig *lebendiger* Variablen. Ziel der *Registeroptimierung* ist es, diese Anzahl bei der Synthese minimal zu halten (z. B. dadurch, daß der Abstand der schreibenden und lesenden Operationen kurz gehalten wird).

Da das Ende einer Lebenszeit i. d. R. von mehreren Operationen j_d abhängig ist, muß die Summe der in einem Kontrollschritt i benötigten Register über jede mögliche Kombination aller Operationen j_d ermittelt werden. Im folgenden werden diese Kombinationen mit $\mathrm{ARCS}(J,i)$ bezeichnet. $\mathrm{ARCS}(J,i)$ steht für alle maximalen Mengen von Lebenszeiten $j_s \prec j_d$, die den Kontrollschritt i kreuzen, wobei jede Lebenszeit von einer anderen Quelle j_s ausgeht.

Gebotys und Elmasry [GeEl92] führen eine zusätzliche Vorschrift ein, die die Anzahl der notwendigen Register auf r Einheiten beschränkt. Die Variable r repräsentiert also die Anzahl notwendiger Register.

Umgesetzt auf die in dieser Arbeit verwendete Notation lautet die Vorschrift zur Registeroptimierung (*Register Allocation Constraint*):

$$\forall i \in I,\ \forall \{j_s \prec j_d\} \in \mathrm{ARCS}(J,i)\ :$$

$$\sum_{\{j_s \prec j_d\}} \left(\sum_{\substack{k \in K: \\ f(j_s) \in F(k)}} \sum_{\substack{i_1 \in R(j_s,k): \\ i_1 \leq i - C(j_s,k)+1}} x_{i_1,j_s,k} + \sum_{\substack{k \in K: \\ f(j_d) \in F(k)}} \sum_{\substack{i_2 \in R(j_d,k): \\ i_2 > i - \ell(j_d,k)+1}} x_{i_2,j_d,k} \right. \tag{2.25}$$

$$\left. - \sum_{\substack{k \in K: \\ f(j_d) \in F(k)}} \sum_{\substack{i_3 \in R(j_d,k): \\ i_3 \leq i - \ell(j_d,k)+1}} x_{i_3,j_d,k} - \sum_{\substack{k \in K: \\ f(j_s) \in F(k)}} \sum_{\substack{i_4 \in R(j_s,k): \\ i_4 > i - C(j_s,k)+1}} x_{i_4,j_s,k} \right) \leq 2r$$

Für jeden Kontrollschritt i summiert diese Vorschrift die doppelte Anzahl der Variablenlebenszeiten auf, die diesen Kontrollschritt i kreuzen.

Die eingeklammerten Summen ergeben den Wert 2, wenn die Operation j_s *vor* Schritt i ausgeführt und Operation j_d erst in einem Kontrollschritt *nach* Schritt i gestartet wird, also in Schritt i ein Register zur Speicherung des Ergebnisses von Operation j_s benötigt wird. In den Fällen, in denen beide Operationen j_s und j_d vor oder nach Kontrollschritt i ausgeführt werden, ergeben die geklammerten Summen den Wert 0 (das erste und dritte, bzw. das zweite und vierte Summenpaar ergeben 1 und heben sich damit auf)[11].

[10] Durch Analyse von transitiven Datenabhängigkeiten und Vergleich der Kontrollschrittbereiche der Operationen j_d läßt sich in polynomieller Zeit die Anzahl dieser Operationen reduzieren. Im allgemeinen gibt es aber auch dann noch mehrere Operationen, die als letzte lesende Operation der Variablen in Frage kommen. Die Operation, die das Ende der Lebenszeit einer Variablen bestimmt, ist also vor der Synthese nicht eindeutig bestimmbar.

[11] Dieses trifft insbesondere auch auf den Fall von *Chaining* zu. Bei Chaining wird das Ergebnis der Operation j_s im selben Kontrollschritt an Operation j_d weitergereicht, so daß kein Register zur Speicherung notwendig ist.

Die Summe $\sum_{\{j_s \prec j_d\}}$ läuft nun über alle Operationenpaare j_s und j_d, deren Variablenlebenszeit möglicherweise den Kontrollschritt i kreuzt. Das ist genau dann der Fall, wenn die beiden Schnittmengen $R(j_s) \cap \{1, \ldots, i - (C(j_s) - 1)\}$ und $R(j_d) \cap \{i + 1 - (\ell(j_d) - 1), \ldots, i_{\max}\}$ nicht leer sind.

Über alle Gleichungen betrachtet, stellt Vorschrift 2.26 somit sicher, daß in jedem Kontrollschritt i nicht mehr als r Variablen in Registern zu halten sind. In der Bewertungsfunktion kann r daher als Faktor für die Registerkosten verwendet werden.

2.10 Allgemeines Chaining

Eine wesentliche Erweiterung des OSCAR-Systems gegenüber anderen High-Level Synthesesystemen stellt die Unterstützung von allgemeinem *Chaining* (*Verkettung* von Operationen) dar. Unter Chaining versteht man die Ausführung mehrerer datenabhängiger Operationen innerhalb *eines* Kontrollschrittes. Neben der Einsparung von Kontrollschritten (Geschwindigkeitsvorteil) führt Chaining von Operationen i. d. R. auch zu geringeren Entwurfskosten, da zwischen den verketteten Funktionseinheiten kein Register zur Speicherung des Zwischenergebnisses mehr benötigt wird.

Auch andere auf IP-Modellen basierende Synthesesysteme (z. B. [GeEl93]) unterstützen Chaining von Operationen. Hier ist Chaining aber beschränkt auf Operationen, die *vor* der Synthese *manuell* zu Ketten zusammengefaßt werden. Im OSCAR-System [LaMaDö94a] hingegen wird Chaining während der Synthese *automatisch* durchgeführt, und zwar nur dann, wenn es sich in Bezug auf die Gesamtkosten des Entwurfs kostengünstig auswirkt.

In Abschnitt 2.6 ist bereits Vorsorge für Chaining getroffen worden. Für datenabhängige Operationspaare $j_1 \prec j_2$ erlaubt die Vorrangsvorschrift 2.15 die Ausführung von j_1 und j_2 im selben Kontrollschritt, wenn $\text{chain}(j_1, j_2)$ auf 1 gesetzt ist. Vor Aufstellen der Vorrangsvorschrift muß $\text{chain}(j_1, j_2)$ also für alle datenabhängigen Operationenpaare ermittelt werden.

Hierzu ist eine genauere Betrachtung des Taktmodells notwendig. Abbildung 2.5 zeigt den Taktzyklus des OSCAR-Systems und seine Unterteilung.

Die Zykluszeit $time_{cycle}$ (*Cycle Time*) teilt sich auf in die Systemverzögerung (*System Latency*) und die für Operationsausführung nutzbare Zeit. Die Systemverzögerung ergibt sich zum einen durch die Kontrollverzögerung ℓ_{ctrl} (*Control Delay*), d. h. die für das Steuerwerk selbst und die zur Ansteuerung der Bausteininstanzen benötigte Zeit. Zum anderen muß die Verbindungsverzögerung ℓ_{cnct} (*Interconnect Delay*) berücksichtigt werden. Das ist die Zeit, die beim Transport der Daten (Funktionsargumente und Ergebnisse) über Verbindungsleitungen und Multiplexer vergeht.

Die bereits eingeführte Operationsausführungszeit $C(j, k)$ in Kontrollschritteinheiten ergibt sich demnach aus der physikalischen Ausführungszeit $\text{time}(j, k)$

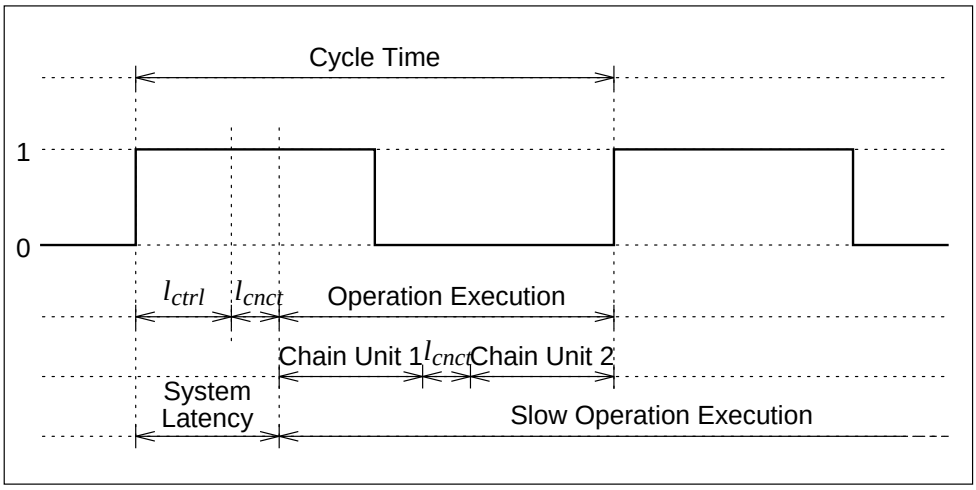

Abbildung 2.5: Der Taktzyklus im OSCAR-System

(im OSCAR-System angegeben in Nanosekunden) durch folgende Gleichung:

$$C(j,k) := \left\lceil \frac{\text{time}(j,k) + (\ell_{cnct} + \ell_{ctrl})}{time_{cycle}} \right\rceil \tag{2.26}$$

Bei der Verkettung zweier datenabhängiger Operationen muß nun die Summe der Ausführungszeiten beider Operationen kleiner sein als die gesamte Zykluszeit abzüglich der Kontroll- und der zweifachen Verbindungsverzögerung (es sind Daten von und zu *zwei* Bausteinen zu transportieren).

Die Möglichkeit, für zwei Operationen j_1 und j_2 Chaining einzusetzen, in Zeichen $j_1 \prec\!\!\prec j_2$, läßt sich folgendermaßen definieren:

$$\begin{aligned} &\forall j_1, j_2 \in J : \\ &\quad j_1 \prec\!\!\prec j_2 \iff \quad j_1 \prec j_2 \\ &\qquad\qquad \wedge \quad \exists k_1 \in K \text{ mit } f(j_1) \in F(k_1) \\ &\qquad\qquad \wedge \quad \exists k_2 \in K \text{ mit } f(j_2) \in F(k_2),\ k_2 \neq k_1 \\ &\qquad\qquad \wedge \quad \text{time}(j_1,k_1) + \text{time}(j_2,k_2) + 2\ell_{cnct} \\ &\qquad\qquad\qquad\qquad \leq \ time_{cycle} - \ell_{ctrl} \end{aligned} \tag{2.27}$$

Hiermit ist $\text{chain}(j_1, j_2)$ genau dann auf 1 zu setzen, wenn $j_1 \prec\!\!\prec j_2$ gilt.

Nun reicht es aber für allgemeines Chaining nicht aus, nur jeweils zwei Operationen zu betrachten. Es muß auch möglich sein, drei oder mehr Operationen in einem Kontrollschritt zu verketten.

Hierzu müssen die längsten Ketten von jeweils datenabhängigen Operationen betrachtet werden, für die paarweise Chaining möglich ist. Eine solche, nichtverlängerbare Kette aus einer Operationenmenge $J' \subset J$ wird im folgenden mit $\text{Chain}(J')$ bezeichnet. Weiterhin wird für die Menge aller Operationenketten eines Entwurfs die Bezeichnung $\text{CHAINS}(J)$ verwendet.

Mathematisch kann die Notation dieser Mengen folgendermaßen definiert werden:

$$\begin{aligned} \text{Chain}(J') \quad := \quad & \{\ j_1, \ldots, j_s \in J' \mid \\ & \forall j_i,\ i \in \{1, \ldots, n-1\} : j_i \prec\!\!\prec\ j_{i+1} \\ & \wedge \quad \nexists j_0 \in J' : j_0 \prec\!\!\prec\ j_1 \\ & \wedge \quad \nexists j_{n+1} \in J' : j_s \prec\!\!\prec\ j_{n+1}\} \end{aligned} \tag{2.28}$$

$$\begin{aligned} \text{CHAINS}(J) \quad := \quad & \bigcup_{J' \subseteq \mathcal{P}(J)} \text{Chain}(J') \\ & \text{mit} \quad \forall \mathit{Chain} \in \text{CHAINS}(J) : \\ & \qquad \nexists \mathit{Chain'} \in \text{CHAINS}(J) \text{ mit } \mathit{Chain} \subset \mathit{Chain'} \end{aligned} \tag{2.29}$$

Für alle diese Operationenketten muß nun sichergestellt werden, daß die pro Kontrollschritt zur Operationsausführung zur Verfügung stehende Zeit nicht überschritten wird. Folgende Verkettungsvorschrift (*Chaining Constraint*) läßt in jedem Schritt nur Verkettungen zu, die die Zykluszeit einhalten:

$$\forall \mathit{Chain} \in \text{CHAINS}(J), \quad \forall i \in I :$$

$$\sum_{\substack{j \in \mathit{Chain}: \\ i \in R(j)}} \ \sum_{\substack{k \in K: \\ f(j) \in F(k), \\ C(j,k)=1}} (\text{time}(j,k) + \ell_{cnct}) * x_{i,j,k} \quad \leq \quad \mathit{time}_{cycle} - \ell_{ctrl} \tag{2.30}$$

Diese Vorschrift betrifft nur Operationen j, die auf schnellen Bausteinen ausgeführt werden ($C(j,k) = 1$). Daher kann in der ersten Summe auch $R(j)$ statt $R(j,k)$ verwendet werden (es ist $R(j) = R(j,k)$, wenn $C(j,k) = 1$).

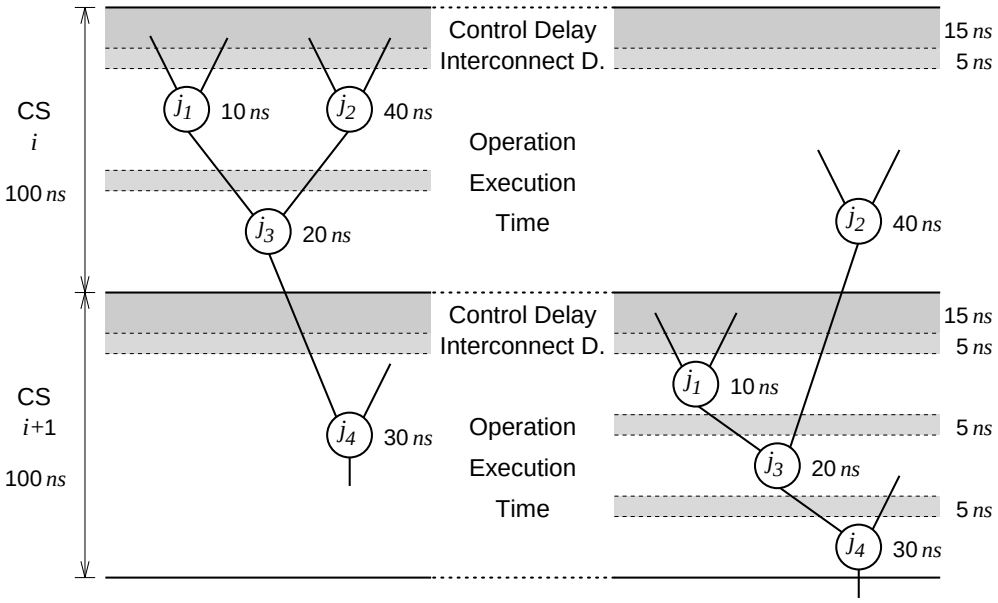

Abbildung 2.6: Allgemeines Chaining von Operationen

Abbildung 2.6 zeigt allgemeines Chaining am Beispiel von vier Operationen. Hier sind zwei maximale Operationenketten zu betrachten. CHAINS(J) besteht aus den beiden Mengen $\mathit{Chain}_1 = \{j_1, j_3, j_4\}$ und $\mathit{Chain}_2 = \{j_2, j_3, j_4\}$.

Bei einer Zykluszeit von $100ns$ und einer Kontrollverzögerung von $15ns$ verbleiben $85ns$ zur Ausführung und Verbindung der Operationen. Mit den angegebenen Ausführungszeiten und einer Verbindungsverzögerung von $5ns$ ergeben sich zwei Lösungen mit der maximal möglichen Anzahl verketteter Operationen.

In der in Abbildung 2.6 links dargestellten Lösung verbleiben im Kontrollschritt i noch $15ns$, welche zur Ausführung der Operation j_4 nicht mehr ausreichen. Operation j_4 kann daher erst in Schritt $i+1$ ausgeführt werden.

Im Fall der rechts dargestellten Lösung wird die gesamte Kette $Chain_1$ innerhalb des Kontrollschrittes $i+1$ ausgeführt. Die Verkettungsvorschrift 2.30 kann für alle Elemente aus $Chain_1$ im selben Kontrollschritt eingehalten werden. Für $Chain_2$ ist das jedoch nicht möglich. Eine Ausführung von Operation j_2 in Schritt $i+1$ würde Vorschrift 2.30 verletzen.

Die Ausführung der Operationen j_2 und j_4 im selben Kontrollschritt ist offensichtlich nicht möglich, obwohl beide Operationen in derselben Kette $Chain_2$ vorkommen. Eine solche Situation muß bei der Berechnung der Kontrollschrittbereiche aller Operationen berücksichtigt werden. Die im OSCAR-System eingesetzte ASAP-ALAP-Analyse verwendet die phys. Ausführungszeit $\mathrm{time}(j,k)$ der Operationen (nicht $C(j,k)$), so daß diese Situation korrekt behandelt wird.

2.11 Das vereinfachte Bindungsmodell

Im letzten Abschnitt dieses Kapitels soll nun noch eine Variante des bis hierher vorgestellten IP-Modells beschrieben werden. Diese Variante bietet nicht mehr die Mächtigkeit des Originalmodells, führt aber zu erheblichen Geschwindigkeitsvorteilen.

Der wesentliche Unterschied zum bisherigen Modell liegt in der Verwendung eines *vereinfachten Bindungsmodells*. Die Aufgabe des *Binding* ist die Bindung von Operationen j an ausführende Instanzen k. Verzichtet man auf die Bindung an *Instanzen* und verlangt lediglich die Zuordnung von Operationen zu *Komponenten* (Bausteintypen), so ergeben sich folgende Änderungen gegenüber dem bisherigen Modell.

Formal wird die Entscheidungsvariable b_k durch die Variable $b_m \in I\!N_0$ ersetzt. Letztere bezeichnet die Anzahl benötigter Instanzen des Bausteintyps m. Es wird also nicht mehr für jede angebotene Instanz entschieden, ob sie verwendet wird ($b_k = 1$) oder nicht ($b_k = 0$), sondern lediglich die Anzahl b_m der eingesetzten Bausteininstanzen berechnet.

Die Menge der Instanzen K enthält in diesem vereinfachten Modell jeweils genau einen Stellvertreter[12] k jeder Komponente $m \in M$, es gilt also $|K| = |M|$. An der Form und Funktion der meisten Gleichungen ändert sich hierdurch nichts,

[12] Es ist auch vorstellbar, daß in sämtlichen Vorschriften des IP-Modells die Instanzen $k \in K$ durch Komponenten $m \in M$ ersetzt werden. Das erfordert aber eine neue Definition der Relationen (z. B. $R(j,m)$, $C(j,m)\ldots$) und Variablen ($x_{i,j,m}$). Dieser Aufwand kann mit der hier abgewandelten Interpretation der Instanzenmenge vermieden werden.

lediglich die für die Zuordnung von Bausteinen zuständigen Vorschriften 2.13 und 2.14, sowie die Verbindungsoptimierung sind von Änderungen betroffen.

In der Baustein-Zuordnungsvorschrift 2.13 wird formal die Variable b_k durch b_m ersetzt. Der Rumpf dieser Vorschrift ergibt sich demnach zu:

$$\sum_{\substack{j\in J:\\ f(j)\in F(k)}} \sum_{\substack{i'=i\\ i\in R(j,k)}}^{i+\ell(j,k)-1} x_{i',j,k} \;\leq\; b_m \quad \text{mit} \quad m = \text{type}(k) \tag{2.31}$$

Die Interpretation dieser Gleichung ändert sich dahingehend, daß auf der linken Seite die *Anzahl* der in einem Kontrollschritt i benötigten Instanzen aufsummiert wird (d. h. es liefern i. a. mehrere Summen über i' eine 1).

Die Vorschrift 2.14, welche die Reihenfolge der Allokation von Instanzen gleichen Typs vorschreibt, wird überflüssig und kann entfallen.

Offensichtlich ist die Verbindungsminimierung mit diesem vereinfachten Bindungsmodell unvereinbar. Die Berechnung der Verbindungsvariablen w_{k_1,k_2} wird unsinnig, wenn jedes k einen Stellvertreter mehrerer Instanzen darstellt. Zur Minimierung von Verbindungen ist daher das Originalmodell notwendig.

Eine weitere Voraussetzung für dieses vereinfachte Modell ist, daß keine manuelle Bindung von Operationen an ausführende Instanzen vorgenommen wird. Das OSCAR-System läßt im allgemeinen die Zuordnung von Operationen zu Instanzen durch den Benutzer zu. Sind solche Instanzenbindungen für den Entwurf vorgeschrieben, kann das vereinfachte Modell nicht eingesetzt werden.

Diese beiden Einschränkungen werden aber aufgewogen durch die Reduzierung des Suchraumes für den IP-Solver. Das vereinfachte Modell benötigt wesentlich weniger Variablen und Gleichungen und erreicht damit eine erhebliche Beschleunigung der Lösungsberechnung.

Als Beispiel soll hier wieder der Elliptical-Wave-Filter (siehe Anhang D.1) herangezogen werden. Tabelle 2.5 stellt die Laufzeiten des IP-Solvers, sowie die Anzahl erzeugter Variablen und Gleichungen für das Originalmodell und die hier vorgestellte Variante gegenüber[13].

Die dargestellten Ergebnisse zeigen deutlich, daß das Originalmodell einen erheblich umfangreicheren Suchraum erzeugt. Die Bindung an Komponenten reduziert die Anzahl der Variablen, welche einen wesentlichen Anteil an der Komplexität des IP-Problems haben, auf ca. ein Drittel der Variablen des Originalmodells. Insbesondere die verringerte Variablenzahl erklärt daher die erheblich verkürzten Rechenzeiten.

Ein durch das vereinfachte Bindungsmodell entstehendes Problem ist bisher noch vernachlässigt worden: Für die Synthese ist eine Bindung an Instanzen in jedem Fall notwendig. Im OSCAR-System wird diese im Fall des vereinfachten Modells nachträglich vorgenommen. Eingesetzt wird hierzu ein modifizierter

[13] Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec -a <steps> [-Ob] elliptic oscar_behaviour;`

Instanzenbindung:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3/5	3/5	2/5	2/5	2/5	2/5
Multiplizierer ($ 40)	2/4	1/4	1/4	1/4	1/4	1/4
Anzahl Variablen	341	518	695	872	1049	1226
Anzahl Gleichungen	223	308	389	469	554	640
Berechnungszeit	1s	2s	7s	71s	175s	236s
Komponentenbindung:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3	3	2	2	2	2
Multiplizierer ($ 40)	2	1	1	1	1	1
Anzahl Variablen	113	162	211	260	309	358
Anzahl Gleichungen	147	215	283	356	434	513
Berechnungszeit	1s	1s	2s	12s	8s	40s

Tabelle 2.5: Laufzeiten mit Bindung an *Instanzen* und an *Komponenten*

Left-Edge-Algorithmus, der in polynomieller Zeit mittels einer Bindungstabelle die notwendigen Zuordnungen von Operationen zu Instanzen vornimmt. Dieser Algorithmus ist in Anhang C.3.2 angegeben.

Kapitel 3

Der Ablauf der OSCAR-Synthese

Die Aufgabe der Mikroarchitektursynthese ist es, ein auf algorithmischer Ebene beschriebenes Entwurfsverhalten unter Einhaltung aller Randbedingungen auf eine Strukturbeschreibung auf RT-Ebene abzubilden. Das OSCAR-System unterteilt diese Aufgabe in vier Schritte:

- Einlesen der Entwurfsspezifikation (*Frontend*)
- Darstellung des Entwurfs in einem kombinierten Kontroll- und Datenflußgraphen (*Datenrepräsentation*)
- Synthese des Entwurfs auf der Basis Ganzzahliger Programmierung
- Nachverarbeitung zur Erzeugung des Steuerwerks und der Netzliste von RT-Bausteinen (*Backend*)

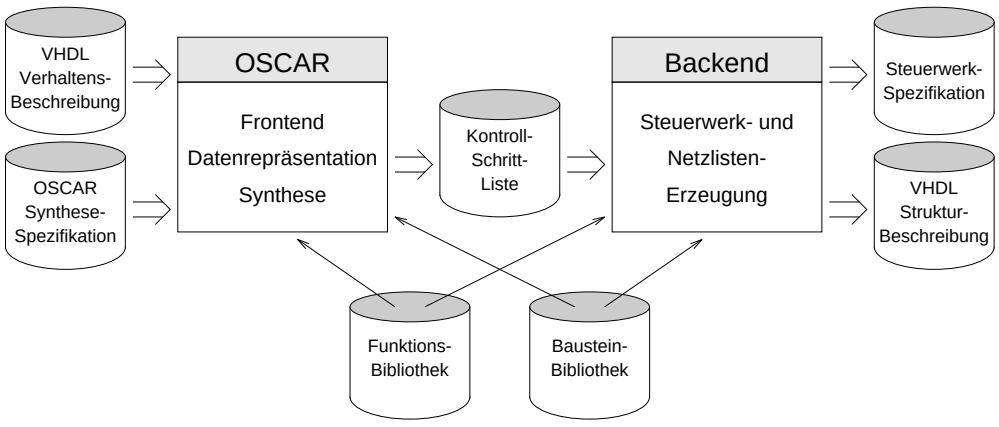

Abbildung 3.1: Das OSCAR-Synthesesystem im Überblick

Abbildung 3.1 stellt das Datenflußmodell des OSCAR-Systems dar. Die einzelnen Schritte des Syntheseablaufes werden im folgenden beschrieben.

3.1 Das Frontend

Das *Frontend* übernimmt die Vorverarbeitung der Entwurfsspezifikation. Hierzu zählt insbesondere das Einlesen und die syntaktische und semantische Überprüfung der algorithmischen Verhaltensbeschreibung.

Das OSCAR-System verwendet zur Spezifikation des Entwurfsverhaltens die Hardware-Beschreibungssprache VHDL [IEEE88]. VHDL ist eine sehr mächtige Sprache, die speziell zur Beschreibung und insbesondere zur Simulation von Hardware-Bausteinen eingesetzt wird.

Im Gegensatz zu gewöhnlichen imperativen Programmiersprachen (wie z. B. PASCAL, MODULA oder C) unterstützt VHDL spezielle Konstrukte zur Beschreibung von Signalverläufen, Nebenläufigkeiten und Bausteinstrukturen. Die Mächtigkeit der Sprache bietet zum einen den Vorteil, daß sowohl die Verhaltensbeschreibung des Entwurfs (die Eingabe) als auch die zu erzeugende Strukturbeschreibung (die Ausgabe) in VHDL formuliert werden können. Zum anderen ergibt sich aber das Problem, daß der Sprachumfang Konstrukte und Mechanismen enthält, die in der Eingabebeschreibung für ein Synthesesystem nicht unterstützt werden können.

Für die Spezifikation der Verhaltensbeschreibung muß der Sprachumfang von VHDL daher auf eine synthetisierbare Untermenge eingeschränkt werden. Das OSCAR-System unterstützt den Sprachumfang von MARMOR. Dieser ist in [Marmor93] detailliert aufgeführt und soll an dieser Stelle nicht wiederholt werden.

Weiterhin muß die Eingabebeschreibung genau *einen Prozess* enthalten, welcher das *Verhalten* des Entwurfs in Form von Kontrollstrukturen und Variablen- und Signalzuweisungen definiert. Zu den unterstützten Kontrollstrukturen gehören insbesondere Verzweigungen (IF-THEN-ELSE-Statement) und Schleifenkonstrukte (WHILE-LOOP-Statement).

Zur Spezifikation von Randbedingungen (z. B. erforderliche Zeitabstände zwischen Lese- und Schreiboperationen) ist es notwendig, insbesondere einzelne Operationen eindeutig kennzeichnen zu können. Prinzipiell bietet VHDL für solche Zwecke *Attribute* an. Diese können aber nur an Typen von Operationen gebunden werden (z. B. an Addition und Subtraktion), nicht jedoch an einzelne Operations*instanzen*. Das OSCAR-System verwendet daher *Pseudo-Kommentare*, um *Label* für Operationen und Basisblöcke vergeben zu können. Diese werden direkt hinter der zu bezeichnenden Anweisung angegeben und können so durch das Frontend mit der Operation bzw. dem Basisblock assoziiert werden.

Die so spezifizierte Verhaltensbeschreibung des Entwurfs wird durch das Frontend eingelesen und in eine interne Darstellung (*Parse Tree*) überführt. Gleichzeitig wird die Entwurfsspezifikation auf syntaktische und semantische Fehler überprüft. Hierdurch wird sichergestellt, daß z. B. alle Variablen und Signale korrekt deklariert sind, und in Zuweisungen und arithmetischen Ausdrücken die Bitbreiten von Bitvektoren übereinstimmen.

Abbildung 3.2 stellt ein Beispiel für eine korrekte VHDL-Verhaltensbeschreibung eines Entwurfs für die OSCAR-Synthese vor.

```
-------------------------------------------------------------------------
-- sample.vhdl: simple example for OSCAR-synthesis
-------------------------------------------------------------------------
ENTITY sample IS
  PORT( in1, in2, in3   : IN  bitvector(15 DOWNTO 0);
        out1, out2      : OUT bitvector(15 DOWNTO 0) );
END sample;

ARCHITECTURE behaviour OF sample IS
BEGIN PROCESS
  VARIABLE    a, b, c, x, y : bitvector(15 DOWNTO 0);
  BEGIN
        a := in1;       -- OSCAR LABEL Operation_1;  -- read the
        b := in2;       -- OSCAR LABEL Operation_2;  -- inputs
        c := in3;       -- OSCAR LABEL Operation_3;
        x := a + b * c; -- OSCAR BLOCK Block_1;      -- compute the
        y := a * b + c;                              -- functions
        out1 <= x;      -- OSCAR LABEL Operation_8;  -- output the
        out2 <= y;      -- OSCAR LABEL Operation_9;  -- results
  END PROCESS;
END behaviour;
-------------------------------------------------------------------------
```

Abbildung 3.2: Beispiel einer Verhaltensbeschreibung in VHDL

Das Beispiel beschreibt einen einfachen Baustein mit drei Ein- und zwei Ausgabeports. Dieser Baustein liest die an den Eingabeports anliegenden Daten ein, berechnet die spezifizierten Funktionen und gibt die Ergebnisse schließlich über die beiden Ausgabeports aus.

Gut zu erkennen ist die Verwendung von Pseudo-Kommentaren, welche jeweils mit dem (Pseudo-) Schlüsselwort `OSCAR` eingeleitet werden. Die Lese- und Schreiboperationen, sowie der einzige Basisblock werden mit den angegebenen Bezeichnungen gekennzeichnet und können so im Synthesesystem eindeutig referenziert werden.

3.2 Die Datenrepräsentation

Die vom Frontend erzeugte Zwischenstruktur (Parse Tree) ist als zugrundeliegende Datenstruktur für die Synthese nicht ausreichend, da insbesondere die Datenabhängigkeiten zwischen den Operationen nicht dargestellt werden. Das OSCAR-System verwendet daher einen kombinierten Kontroll- und Datenflußgraphen (CDFG, *Control Data Flow Graph*) zur internen Repräsentation des Entwurfs, welcher leicht aus den vom Frontend bereitgestellten Informationen erzeugt werden kann.

Die Verhaltensbeschreibung eines Entwurfs besteht aus einer Reihe von (evtl. verschachtelten) Kontrollstrukturen und sog. Basisblöcken. Ein Basisblock (*Basic Block*) bezeichnet eine ununterbrochene Sequenz von Variablen- und Signalzuweisungen in der Entwurfsbeschreibung.

Aus den spezifizierten Kontrollstrukturen erzeugt das OSCAR-System einen gerichteten Kontrollflußgraphen, der das Ablaufverhalten des Entwurfs widerspiegelt. Abbildung 3.3 zeigt ein Beispiel für einen solchen Kontrollfluß. Der dargestellte Graph enthält im wesentlichen eine Verzweigung, die alternativ die Basisblöcke $B2$ und $B3$ ansteuert, sowie eine Schleife um den Block $B4$.

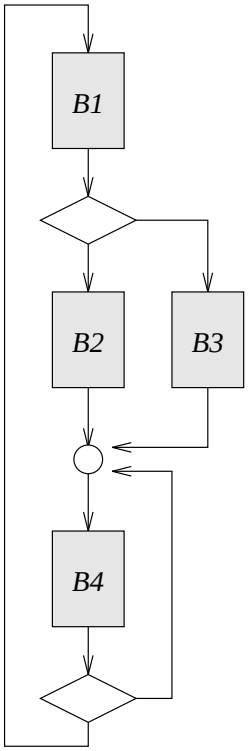

Abb. 3.3: Bsp. eines Kontrollflusses

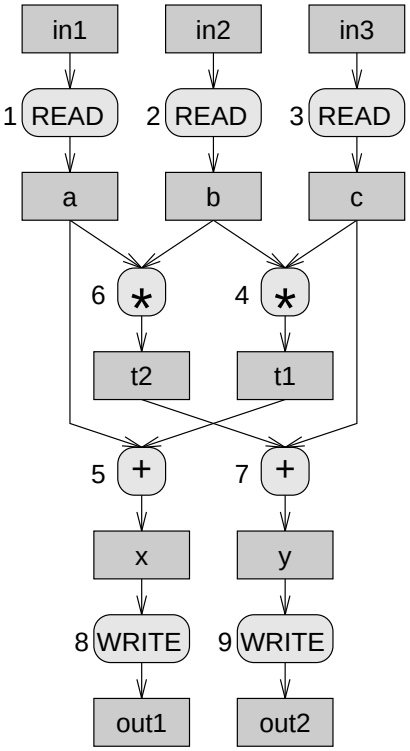

Abb. 3.4: Bsp. eines Datenflusses

Die im Kontrollflußgraphen enthaltenen Basisblock-Knoten stellen jeweils einen eigenen Datenflußgraphen dar. Ein solcher Datenflußgraph entsteht aus den im Entwurf spezifizierten Zuweisungen (*Assignments*) und Ausdrücken (*Expressions*). Im Gegensatz zum Kontrollflußgraphen enthält ein Datenflußgraph keine Zyklen, es handelt sich also um einen gerichteten, zyklenfreien Graphen (*DAG, Directed Acyclic Graph*). Die Knoten dieses Graphen stellen die spezifizierten Operationen dar. Hierzu zählen arithmetische und logische, sowie Lese- und Schreiboperationen. Die Kanten des Graphen repräsentieren den Informationsfluß zwischen den Operationen. Kehrt man die Richtung der Datenflußkanten um, so ergeben sich direkt die Datenabhängigkeiten der Operationen, welche bei der Synthese berücksichtigt werden müssen.

Die im OSCAR-System erzeugten Datenflußgraphen enthalten zusätzlich zu den Operationsknoten auch Speicherknoten, welche die verwendeten Speicherelemente (Variablen, Signale und Ports) repräsentieren. Abbildung 3.4 zeigt den

Datenflußgraphen, den das OSCAR-System für das auf Seite 41 vorgestellte Beispiel `sample.vhdl` erzeugt. Zusätzlich zu den schon in der Verhaltensbeschreibung spezifizierten Speicherelementen enthält der Graph zwei temporäre Variablen `t1` und `t2`, welche die Ergebnisse der beiden Multiplikationen aufnehmen. In der Entwurfsbeschreibung sind diese Variablen nur implizit in den Funktionsausdrücken enthalten.

Das OSCAR-System zerlegt komplexe Ausdrücke jeweils in einfache, einstufige Ausdrücke, deren Ergebnis temporären Variablen zugewiesen wird. Somit wird sichergestellt, daß in der zu erzeugenden Hardware für jedes Zwischenergebnis ein Register zur Verfügung steht. Hierdurch entsteht zunächst eine große Anzahl von Speicherelementen. Diese kann jedoch durch eine Registerfaltung nach der Synthese auf ein Minimum reduziert werden, so daß sich diese zusätzlichen Speicherelemente nicht direkt auf die Größe des Entwurfs auswirken.

In engem Zusammenhang mit den Datenflußgraphen steht die Funktionsbibliothek (*Function Attribute Library*), welche sämtliche im OSCAR-System bekannten Operationstypen (Funktionen) und deren Attribute enthält (z. B. Funktionsbezeichnung, Anzahl der Argumente, Kommutativität usw.). Diese entspricht in der in Kapitel 2 verwendeten mathematischen Notation der Menge der Operationstypen G. Jeder Operationsknoten eines Datenflußgraphen enthält insbesondere einen Verweis auf den entsprechenden Eintrag in der Funktionsbibliothek.

3.3 Die Synthese mittels Ganzzahliger Programmierung

Um die eigentliche Synthese mit Hilfe der Ganzzahligen Programmierung durchführen zu können, sind einige Vorbereitungen notwendig. Hierzu zählen das Aufstellen weiterer Datenstrukturen, die das IP-Modell geeignet repräsentieren, sowie die Integration weiterer Synthesespezifikationen, durch die der Benutzer des Systems den Syntheseablauf beeinflussen kann. Diese Schritte, sowie die Erzeugung und Lösung des IP-Gleichungssystems und die anschließende Registerfaltung sollen im folgenden beschrieben werden.

3.3.1 Das Aufstellen der IP-Datenstrukturen

Um die in dem Kontroll- und Datenflußgraphen enthaltene Entwurfsspezifikation auf das in Kapitel 2 vorgestellte IP-Modell abbilden zu können, sind zusätzliche Datenstrukturen erforderlich. Insbesondere müssen die Mengen I, J, K und M geeignet repräsentiert werden.

In Anhang C.2 sind die im OSCAR-System implementierten Datenstrukturen detailliert aufgeführt. Im folgenden werden lediglich grundlegende Eigenschaften beschrieben.

3.3.1.1 Die Operationen

Das OSCAR-System stellt zunächst die *Liste der Operationen* auf, welche die Menge J repräsentiert. Hierzu wird der Kontrollflußgraph sowie die enthaltenen Datenflußgraphen jeweils in Form eines Tiefendurchlaufs (*DFS, Depth First Search*) durchlaufen. Die besuchten Operationsknoten werden jeweils an die Liste der Operationen angehängt und mit einem eindeutigen Schlüssel (*ID*) gekennzeichnet. Dieser Schlüssel, eine natürliche Zahl, welche (mit 1 beginnend) aufsteigend vergeben wird, entspricht direkt der in Kapitel 2 verwendeten Operationsbezeichnung $j \in J$.

Eine Ausnahme stellt die Konkatenation (`&`-Operation) dar. Diese Operation benötigt zur Ausführung keinen Baustein, sondern kann vielmehr durch eine einfache Verdrahtung in der Netzliste vorgenommen werden. Sie muß daher innerhalb des IP-Modells gesondert behandelt werden. Im OSCAR-System werden die im Entwurf enthaltenen Konkatenationen nicht in die Operationenliste aufgenommen, da keine ausführende Instanz und kein Kontrollschritt benötigt werden. Es werden daher auch keine IP-Variablen für diese Operationen erzeugt. Die Konkatenationen werden stattdessen an das Backend als spezielle Netzlisten-Operationen durchgereicht.

Durch den Tiefendurchlauf durch die (bekanntlich zyklenfreien) Datenflußgraphen wird erreicht, daß die Operationenliste eine *topologische Ordnung* darstellt. Jede neue Operation kann nur von Operationen abhängen, die sich bereits in der Operationenliste befinden. Formal resultiert diese topologische Ordnung darin, daß gilt:

$$\forall j_1, j_2 \in J : \; j_1 \prec j_2 \;\Rightarrow\; j_1 < j_2 \tag{3.1}$$

Datenabhängigkeiten einer Operation können also nur zu Operationen bestehen, deren Schlüssel kleiner als der eigene ist. Daher ist es problemlos möglich, gleichzeitig mit den Operationen auch die Datenabhängigkeiten in Form von *Präzedenzkanten* in die Datenstruktur zu übernehmen, d. h. für jede Operation Verweise auf die vor ihr auszuführenden Operationen zu notieren.

Die topologische Ordnung der erzeugten Operationenliste stellt eine wichtige Eigenschaft dar, die in vielen der im OSCAR-System eingesetzten Algorithmen ausgenutzt wird. Insbesondere die ASAP-ALAP-Analyse macht hiervon intensiv Gebrauch.

Während des Tiefendurchlaufs durch den Kontrollflußgraphen wird weiterhin die *Liste der Kontrollblöcke* aufgebaut, welche die Menge der Datenflüsse $d \in D$ repräsentiert. Das OSCAR-System behandelt jeden Basisblock des Entwurfs als eigenen Datenfluß (welcher mehrere alternative Versionen enthalten kann, s. Abschnitt 2.2.4). Da die Ausführung dieser Datenflüsse von den Kontrollstrukturen des Kontrollflußgraphen abhängig ist, werden diese Basisblöcke im OSCAR-System auch als Kontrollblöcke (*Control Blocks*) bezeichnet.

3.3.1.2 Die Komponenten

Die *Selektion geeigneter Komponenten* bildet den nächsten Schritt des Syntheseablaufs. Das OSCAR-System verwendet hierzu eine Bausteinbibliothek, welche sämtliche zur Verfügung stehenden funktionalen Einheiten, sowie Register und Port-Bausteine enthält. Für alle Operationen des Entwurfs werden nun geeignete Bausteine aus der Bibliothek in eine Komponentenliste eingefügt, welche die Menge M repräsentiert. Analog zur Operationenliste werden auch hier alle Bausteine mit einem eindeutigen Schlüssel gekennzeichnet.

Gleichzeitig mit dem Aufbau der Komponentenliste notiert das OSCAR-System für jeden Operationsknoten die minimale und maximale Ausführungszeit der Operation. Diese Zeiten werden für die Bestimmung der ASAP-ALAP-Zeiten benötigt. Weiterhin muß für die Unterstützung von Chaining die Funktion $\mathrm{chain}(j_1, j_2)$ für alle datenabhängigen Operationen $j_1 \prec j_2$ bestimmt werden, so daß bekannt ist, für welche Präzedenzkanten Chaining in Frage kommt.

Mit der dann durchgeführten *ASAP-ALAP-Analyse* wird die Menge der Kontrollschritte I für jeden Kontrollblock festgelegt. Das OSCAR-System geht zunächst von der minimalen Anzahl von Kontrollschritten aus, welche durch die ASAP-Zeit der letzten Operation eines jeden Datenflusses bestimmt wird. Diese Anzahl kann vom Benutzer des Systems durch Angabe weiterer Kontrollschritte erhöht werden. Innerhalb dieser festgelegten Kontrollschrittgrenzen minimiert das System dann die Kosten des Entwurfs.

3.3.1.3 Die Instanzen

Erst nach der ASAP-ALAP-Analyse wird die *Liste der Bausteininstanzen* aufgestellt, welche die Menge K repräsentiert. Hierzu wird die Anzahl der angebotenen Instanzen eines jeden Bausteintyps benötigt. Diese Anzahl muß, sofern sie nicht durch den Benutzer angegeben wird, durch das System berechnet werden. Der hierzu eingesetzte Algorithmus ist in Anhang C.3.1 angegeben.

Schließlich wird noch die *Tabelle der Verbindungskosten* aufgestellt, welche für alle Instanzenpaare $k_1, k_2 \in K$ die Kosten einer Verbindung von k_1 zu k_2 enthält. Das OSCAR-System unterstützt sowohl konstante Kosten, welche für alle Verbindungen gleich sind, als auch relative Verbindungskosten, welche abhängig von der benötigten Bitbreite sind. Ist eine Verbindung zwischen zwei Instanzen nicht möglich, weil z. B. im gesamten Entwurf keine Addition Ergebnisse einer Multiplikation weiterverarbeitet, so entfällt der entsprechende Eintrag in der Tabelle und die betroffenen Verbindungsvariablen w_{k_1,k_2} werden nicht erzeugt.

3.3.2 Die Synthesespezifikationen

Bis hierher sind lediglich Entwurfsspezifikationen in die Datenstrukturen eingeflossen, die sich aus der VHDL-Verhaltensbeschreibung des Entwurfs ergeben.

Das OSCAR-System unterstützt zusätzlich eine Spezifikationsdatei, durch welche der Benutzer weiteren Einfluß auf die Synthese nehmen kann.

Das Format der OSCAR-Spezifikationsdatei ist detailliert in Anhang B.2.4.1 aufgeführt. Abbildung 3.5 stellt als Beispiel die Datei `sample.spec` vor, welche die Verhaltensbeschreibung des auf Seite 41 eingeführten Entwurfs `sample` um weitere Spezifikationen erweitert.

```
-------------------------------------------------------------------------
-- sample.spec: Simple example for OSCAR-Synthesis
-------------------------------------------------------------------------
SPECIFICATION FOR ARCHITECTURE behaviour OF sample IS

BEGIN
  SET RESET_NAME  Reset;          -- specifies global signals
  SET CLOCK_NAME  Clock;

  SET CONTROLLER_DELAY    50ns;   -- specifies system timing
  SET INTERCONNECT_DELAY  25ns;
  SET CYCLE_TIME          1us;    -- 1.0 MHz

  SET INTERCONNECT_COSTS  100 PER BIT;  -- specifies cost ratio

  START Operation_1 AT CS 1;      -- specifies input timing
  START Operation_2 AT CS 1;
  START Operation_3 AT CS 1;

  SEPARATE Operation_8 AND Operation_9 BY 0 CS;  -- output timing

  BIND Operation_4 TO COMPONENT Multiplier;      -- user binding
  BIND Operation_5 TO INSTANCE Adder_1;

  LIMIT Multiplier TO 1 INSTANCES;  -- specifies number of instances

  EXTEND Block_1 BY 1 CS;           -- specifies additional steps

END SPECIFICATION;
-------------------------------------------------------------------------
```

Abbildung 3.5: Beispiel einer OSCAR-Spezifikationsdatei

Die Anweisungen in der Spezifikationsdatei sind weitgehend selbsterklärend. Dennoch soll anhand des Beispiels kurz auf die möglichen Spezifikationen eingegangen werden.

Die `SET`-Anweisung definiert Konstanten, welche global im Entwurf gelten. Es können sowohl die Bezeichnungen des Reset- und des Takt-Signals, als auch der Systemtakt selbst spezifiziert werden. Weiterhin werden die Verbindungskosten hiermit definiert.

Mit der `START`-Anweisung werden absolute Zeitvorgaben für die Synthese spe-

zifiziert, welche die Kontrollschrittbereiche einzelner Operationen beschränken. Im dargestellten Beispiel werden die drei Leseoperationen von den Eingangsports auf den ersten Kontrollschritt fixiert.

Relative Zeitvorgaben lassen sich mit der `SEPARATE`-Anweisung spezifizieren. Zwischen zwei beliebigen Operationen werden hiermit konstante, maximale oder minimale Zeitabstände definiert, welche in die entsprechenden IP-Vorschriften umgesetzt werden. Im dargestellten Beispiel wird vorgeschrieben, daß die beiden Ausgabeoperationen `Operation_8` und `Operation_9` im selben Kontrollschritt auszuführen sind.

Die `BIND`-Anweisung erlaubt die Vorgabe der Bindung einzelner Operationen sowohl an Bausteintypen (Komponenten), als auch an Bausteininstanzen. Die Anzahl der Instanzen eines Bausteins kann mit der `LIMIT`-Anweisung vorgeschrieben werden.

Mit der `EXTEND`-Anweisung schließlich läßt sich für jeden Basisblock die Anzahl zusätzlich erlaubter Kontrollschritte spezifizieren. Für das Beispiel wird zusätzlich zu den drei minimal notwendigen ein weiterer Kontrollschritt zugelassen.

3.3.3 Die Lösung des Gleichungssystems

Sind alle Entwurfsspezifikationen in die internen Datenstrukturen eingetragen, so können die in Kapitel 2 beschriebenen IP-Vorschriften erzeugt werden.

Das OSCAR-System verwendet zur Lösung des Gleichungssystems den MILP-Solver `lp_solve` [Be93]. Dieser stellt ein eigenes Programm dar, das nicht fest in das Synthesesystem eingebunden ist. Die Kommunikation mit dem MILP-Solver erfolgt daher über eine Datei-Schnittstelle. Das verwendete Dateiformat ist entsprechend an `lp_solve` angepaßt.

Das OSCAR-System exportiert die zur Synthese des Entwurfs erzeugten Vorschriften in eine Gleichungsdatei und ruft den MILP-Solver auf. Dieser löst das Gleichungssystem und schreibt die berechnete Lösung wiederum in eine Datei, welche durch das Synthesesystem eingelesen und ausgewertet wird.

Das OSCAR-System verfiziert die ermittelte Lösung anhand einiger Konsistenzprüfungen. Insbesondere zählen hierzu die Einhaltung der Datenabhängigkeiten und Zeitvorgaben sowie die Überprüfung der Ausführbarkeit von Operationen auf den ihnen zugeordneten Instanzen. Hierdurch wird sichergestellt, daß die erhaltenen Variablenbelegungen eine korrekte Lösung des Syntheseproblems darstellen.

Zur Auswertung der Lösung erstellt das System eine detaillierte Aufstellung der Entwurfskosten, welche die Anzahl und Einzelkosten verwendeter Port-Bausteine, funktionaler Einheiten und benutzter Verbindungen beinhaltet.

3.3.4 Die Registerfaltung

In Abschnitt 3.2 ist beschrieben, daß in den im OSCAR-System verwendeten Datenflußgraphen für jede Operation ein eigenes Ergebnisregister vorgesehen wird. Dieses führt zu einer Vielzahl von Registern, die für den endgültigen Entwurf nicht akzeptabel ist. Das OSCAR-System führt daher nach der eigentlichen Synthese eine *Registerfaltung* durch, welche die Anzahl der internen Register des Entwurfs auf ein Minimum reduziert.

Die Lebenszeiten der Register sind nach dem Scheduling eindeutig bestimmbar (vgl. Abschnitt 2.9), daher kann die Faltung der Register mit Hilfe eines modifizierten *Left-Edge*-Algorithmus' in polynomieller Zeit vorgenommen werden [KuPa87].

Als Beispiel für die Notwendigkeit der Registerfaltung soll der in Anhang D.1 vorgestellte Elliptical-Wave-Filter angeführt werden. Dieser benötigt ohne Registerfaltung 37 interne Speichereinheiten sowie 9 Ein-Ausgabe-Ports. Die Faltung der internen Register reduziert deren Anzahl auf 10 Einheiten, was eine Ersparnis von 73% ausmacht.

Gleichzeitig mit der Registerfaltung minimiert das OSCAR-System auch die Bitbreite der internen Register. Für jedes Speicherelement werden die Bitbreiten sämtlicher Leseoperationen ausgewertet. Registerbits, welche niemals gelesen werden, sind offensichtlich redundant und werden abgeschnitten.

3.3.5 Die Erzeugung der Kontrollschrittliste

Den Abschluß der eigentlichen Synthese bildet die Erzeugung der sog. *Kontrollschrittliste*. Das OSCAR-System faßt in dieser Datei die Ergebnisse der durchgeführten Synthese, d. h. von Scheduling, Allocation und Binding, zusammen.

Das Format der Kontrollschrittliste ist in Anhang B.2.4.2 spezifiziert. Im wesentlichen unterscheidet das Format zwischen dem Deklarationsteil und dem Rumpf der Datei. Der Deklarationsteil definiert die Ein- und Ausgabe-Ports des Entwurfs einschließlich der Reset- und Clock-Eingänge, sowie sämtliche internen Register und die verwendeten arithmetischen und logischen Einheiten. Der Rumpf der Kontrollschrittliste enthält sämtliche Kontrollstrukturen des Entwurfs und die darin enthaltenen Kontrollblöcke. In diesen wird die Zuordnung der Operationen zu Kontrollschritten und ausführenden Funktionseinheiten beschrieben.

Abbildung 3.6 stellt die Kontrollschrittliste dar, welche das OSCAR-System für das auf den Seiten 41 (`sample.vhdl`) und 46 (`sample.spec`) spezifizierte Beispiel erstellt. Um den Vergleich mit der VHDL-Verhaltensbeschreibung zu erleichtern, ist auf die Registerfaltung verzichtet worden. Daher wird für die spezifizierten Variablen hier jeweils ein eigenes Register verwendet[1]. Für die

[1] Für die Variable X wird hier kein Register benötigt, da der entsprechende Wert in Kontrollschritt 4 direkt aus dem Addierer in den Ausgabeport `OUT1` geladen wird.

```
------------------------------------------------------------------------
-- Control Step List: Schedule generated by OSCAR V1.4.3
------------------------------------------------------------------------
SCHEDULE SAMPLE IS

  CLOCKNAME  CLOCK;
  RESETNAME  RESET;
  PORT IN    IN1[15:0], IN2[15:0], IN3[15:0];
  PORT OUT   OUT1[15:0], OUT2[15:0];
  REGISTER   A[15:0], B[15:0], C[15:0], T1[15:0], T2[15:0], Y[15:0];
  INSTANCE   MULTIPLIER_1  : MULTIPLIER;
  INSTANCE   ADDER_1       : ADDER;

BEGIN SCHEDULE
  BEGIN BLOCK BLOCK_1
   1 A             (LOAD (IN1[15:0]));
   1 B             (LOAD (IN2[15:0]));
   1 C             (LOAD (IN3[15:0]));
   3 MULTIPLIER_1  (*    (B[15:0]) (C[15:0]));
   3 T1            (LOAD (MULTIPLIER_1[15:0]));
   4 ADDER_1       (+    (A[15:0]) (T1[15:0]));
   2 MULTIPLIER_1  (*    (A[15:0]) (B[15:0]));
   2 T2            (LOAD (MULTIPLIER_1[15:0]));
   3 ADDER_1       (+    (T2[15:0]) (C[15:0]));
   3 Y             (LOAD (ADDER_1[15:0]));
   4 OUT1          (LOAD (ADDER_1[15:0]));
   4 OUT2          (LOAD (Y[15:0]));
  END BLOCK;
END SCHEDULE;
------------------------------------------------------------------------
```

Abbildung 3.6: Beispiel einer Kontrollschrittliste

arithmetischen Funktionen sind ein Addierer und ein Multiplizierer alloziert worden.

Der Rumpf der erzeugten Kontrollschrittliste enthält den einzigen Kontrollblock des Entwurfs, welcher die Kontrollschritte 1 bis 4 umfaßt. Für jede im Entwurf enthaltene Operation wird hier angegeben, in welchem Kontrollschritt und auf welcher Instanz die Operation auszuführen ist, sowie von welcher Quelle die benötigten Argumente bereitgestellt werden. Wie in der angegebenen Spezifikationsdatei verlangt, werden die drei Leseoperationen in Schritt 1 und die beiden Schreiboperationen gleichzeitig (in Schritt 4) ausgeführt.

3.4 Das Backend

Das *Backend* übernimmt die Nachbereitung des Syntheseergebnisses und sorgt für die Anbindung des OSCAR-Systems an weitere Entwurfswerkzeuge. Die in

der Synthese erzeugte Kontrollschrittliste ist hierzu in eine Spezifikation des benötigten Steuerwerks und eine Netzliste der verwendeten RT-Bausteine umzusetzen (siehe Abbildung 3.1, Seite 39).

Zur Erzeugung des Steuerwerks (*Controller*) müssen insbesondere die in der Kontrollschrittliste enthaltenen Kontrollstrukturen in die Spezifikation eines Endlichen Automaten (*Finite State Machine*) umgesetzt werden, welcher die Ablaufsteuerung des erzeugten Entwurfs übernimmt und in jedem Kontrollschritt so für die Ansteuerung der Bausteininstanzen sorgt, daß die in der Kontrollschrittliste angegebenen Funktionen ausgeführt werden.

Das Rechenwerk des Entwurfs, welches auch als Datenpfad (*Data Path*) bezeichnet wird, besteht aus den verwendeten Funktionseinheiten und Registern, sowie ggf. einzufügenden Multiplexern, welche vor den Eingangsports der Bausteine die jeweils benötigten Quellen selektieren (*Multiplexing*). Die in einem Kontrollschritt an einen Baustein anzulegenden Quellen sind in Form der Funktionsargumente in der Kontrollschrittliste angegeben und können anhand der Bausteinbibliothek auf die Ports des Bausteins abgebildet werden. Hierbei sind ggf. die angegebenen Argumente auf die Portbreiten des Bausteins zu erweitern, so daß z. B. ein 16-Bit-Addierer auch eine 8-Bit-Addition ausführen kann. Die Art der benötigten Erweiterung (*Extension*) ist für jedes Argument einer Funktion in der Funktionsbibliothek angegeben.

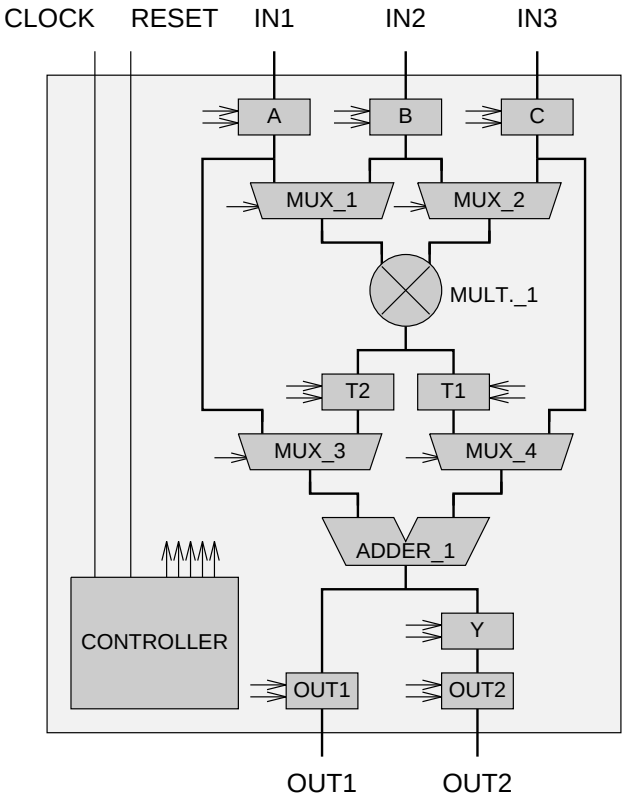

Abbildung 3.7: Beispiel einer erzeugten RT-Struktur

Abbildung 3.7 stellt die erzeugte RT-Netzliste für das Beispiel `sample` graphisch dar. Zusätzlich zu den bereits in der Kontrollschrittliste (Abbildung 3.6) definierten Bausteinen werden hier vier Multiplexer benötigt, die vor den Eingangsports der arithmetischen Einheiten (Addierer und Multiplizierer) die jeweiligen Register auswählen. Das Steuerwerk (*Controller*) ist an den spezifizierten Systemtakt und das Reset-Signal angeschlossen und erzeugt die Steuersignale für die Register und Multiplexer (der Übersicht halber sind diese Steuerleitungen nur angedeutet).

Sowohl die Spezifikation des Steuerwerks, als auch die Netzliste der RT-Bausteine werden im OSCAR-System in der Hardware-Beschreibungssprache VHDL [IEEE88] beschrieben, so daß die Anbindung des Synthesesystems an weitere Entwurfswerkzeuge gewährleistet ist. Insbesondere ist hier an das COMPASS-System [Compass91] gedacht. Mit diesem CAD-Werkzeug kann die erzeugte Steuerwerkspezifikation in einen realen Controller umgesetzt und in die RT-Netzliste des Entwurfs eingefügt werden, so daß der erzeugte Entwurf mit den angebotenen Werkzeugen weiterverarbeitet werden kann.

Zur Weiterverarbeitung zählt insbesondere die *Simulation* des Entwurfs auf RT-Ebene, so daß das Verhalten der erzeugten Implementierung mit dem spezifizierten Ein-Ausgabe-Verhalten verglichen und die Funktionalität des erzeugten Entwurfs verifiziert werden kann. Weiterhin ist mit dem COMPASS-System die Abbildung der hierarchischen Netzliste auf die Gatter-Ebene (Umsetzung in die Zieltechnologie) und schließlich auch auf ein Chip-Layout (*Floor Planning*) möglich, so daß einer Fertigung des Entwurfs in Form eines Micro-Chips nichts mehr im Wege steht.

Kapitel 4

Verbesserungen und Erweiterungen

In diesem Kapitel werden Optimierungen des OSCAR-Systems vorgestellt, welche den Synthesevorgang durch Elimination von Redundanzen und den Einsatz einer Heuristik beschleunigen.

4.1 Die Elimination von Redundanzen

Das in Kapitel 2 vorgestellte IP-Modell erzeugt im allgemeinen ein Gleichungssystem, welches auch redundante Vorschriften enthält. Eine Vorschrift ist genau dann redundant, wenn sich der Lösungsraum des Gleichungssystems durch Weglassen dieser Vorschrift nicht verändert.

Die Bestimmung redundanter Vorschriften ist im allgemeinen nicht trivial. Das OSCAR-System behandelt aber zwei Situationen, in denen in vielen Fällen Redundanzen auftreten und diese leicht erkannt werden können. Anhand zweier Beispiele soll im folgenden die Behandlung von redundanten Datenabhängigkeiten und Zeitvorgaben beschrieben werden.

4.1.1 Redundante Datenabhängigkeiten

Die Datenabhängigkeiten der in einem Entwurf enthaltenen Operationen werden in den Datenflußgraphen explizit als gerichtete Kanten repräsentiert. Im allgemeinen wird für jede dieser Kanten eine Vorrangsvorschrift aufgestellt, welche sicherstellt, daß die notwendige Reihenfolge der Operationsausführung eingehalten wird.

Abbildung 4.1 stellt zwei Datenflußgraphen mit jeweils vier datenabhängigen Operationen dar. In beiden Graphen ist die Ausführungsreihenfolge der Operationen durch die bestehenden Abhängigkeiten eindeutig festgelegt: j_1, j_2, j_3, j_4.

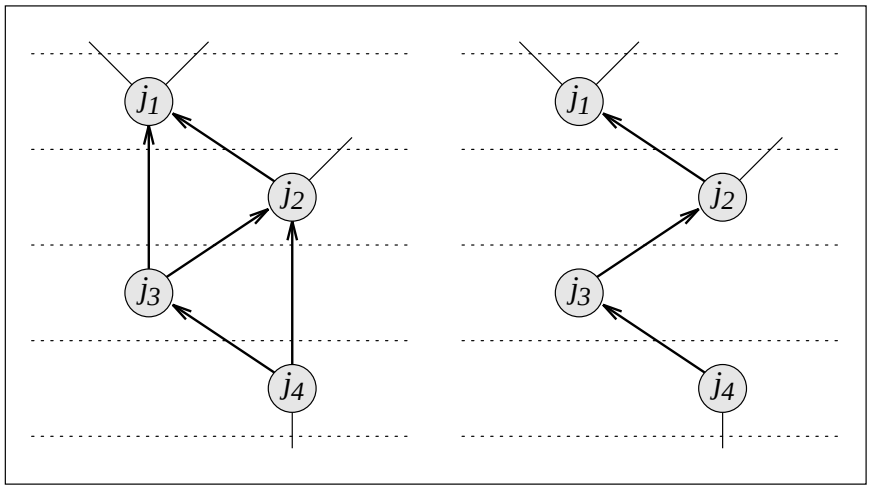

Abbildung 4.1: Elimination redundanter Datenabhängigkeiten

Der rechte Graph enthält jedoch nur drei Abhängigkeitskanten, wodurch auch nur drei (statt fünf) Vorrangsvorschriften erzeugt werden.

Offensichtlich sind die Kanten $j_1 \prec j_3$ und $j_2 \prec j_4$ des linken Graphen redundant. Diese Redundanz ergibt sich aus der in einem Abhängigkeitsgraphen geltenden *Transitivität*. Für das dargestellte Beispiel gilt:

$$j_1 \prec j_2 \ \wedge \ j_2 \prec j_3 \quad \Rightarrow \quad j_1 \prec j_3$$

und

$$j_2 \prec j_3 \ \wedge \ j_3 \prec j_4 \quad \Rightarrow \quad j_2 \prec j_4$$

Die beiden redundanten Abhängigkeiten sind implizit also auch im rechten Graphen enthalten.

Zur Erkennung redundanter Abhängigkeitskanten führt das OSCAR-System von jeder Operation aus einen Tiefendurchlauf (DFS) durch den Graphen durch, welcher die mehrfach erreichbaren Knoten markiert und somit Redundanzen aufdeckt. Die als redundant erkannten Datenabhängigkeiten werden entsprechend markiert und bei der Erzeugung der Vorrangsvorschriften übergangen.

Auch weitere im OSCAR-System verwendete Graphenalgorithmen ignorieren die redundanten Kanten. Insbesondere die Berechnung der längsten Ketten datenabhängiger Operationen, für die allgemeines Chaining möglich ist, profitiert hiervon. Im obigen Beispiel wären auf der linken Seite drei Operationenketten zu betrachten (j_4-j_3-j_1, j_4-j_3-j_2-j_1 und j_4-j_2-j_1), während auf der rechten Seite nur eine einzige Kette entsteht (j_4-j_3-j_2-j_1).

4.1.2 Redundante Zeitvorgaben

Die zweite Situation, in der das OSCAR-System redundante Vorschriften erkennt und eliminiert, betrifft die Zeitvorgaben. In vielen Fällen werden minimale, maximale oder konstante Zeitabstände zwischen Operationen bereits durch

eine geeignete Einschränkung der Kontrollschrittbereiche der betroffenen Operationen eingehalten. In diesen Fällen ist das Aufstellen einer entsprechenden Zeitvorgabevorschrift (siehe Abschnitt 2.7) daher nicht notwendig.

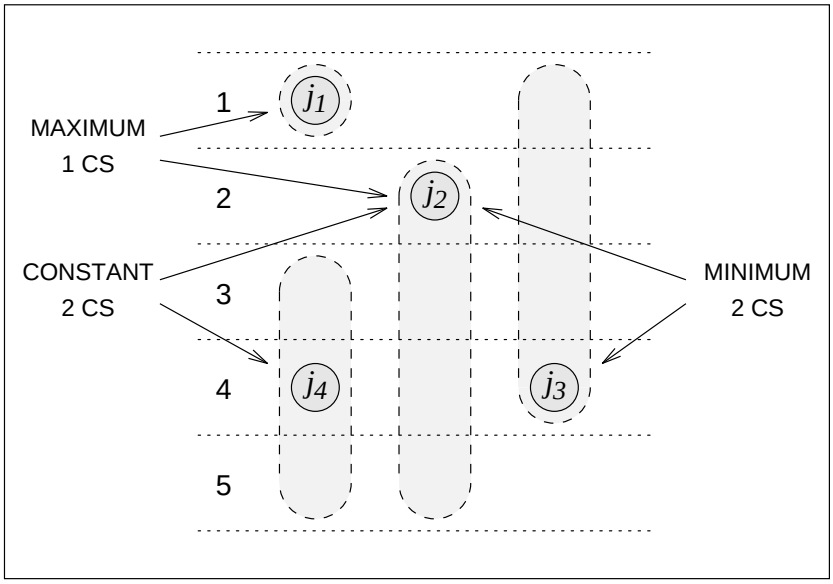

Abbildung 4.2: Elimination redundanter Zeitvorgaben

Abbildung 4.2 zeigt ein extremes, aber durchaus denkbares Beispiel. Gegeben sind vier Operationen, deren aus (nicht dargestellten) Datenabhängigkeiten resultierende Kontrollschrittbereiche angegeben sind. Weiterhin existieren drei Zeitvorgaben, welche für die Operationen eingehalten werden müssen.

Ein einfacher Ansatz erzeugt für diese Zeitvorgaben drei entsprechende Vorschriften, welche in das Gleichungssystem einfließen und somit die Einhaltung der Zeitabstände gewährleisten. Das OSCAR-System schränkt bei der Angabe von Zeitvorgaben die Kontrollschrittbereiche der betroffenen Operationen weitestgehend ein, so daß sich die geforderten Zeitabstände in vielen Fällen automatisch aus den verkürzten Kontrollschrittbereichen ergeben.

Im dargestellten Beispiel erzwingt der maximale Zeitabstand von einem Kontrollschritt für die Operationen j_1 und j_2 die Ausführung von j_2 in Kontrollschritt 2, da j_1 unbeweglich ist. Das OSCAR-System beschränkt daher den Bereich für Operation j_2 auf den einzig möglichen Schritt $R(j_2) = \{2\}$. Durch diese Einschränkung wird die maximale Zeitvorgabevorschrift redundant und kann entfallen. Analog erzwingen die beiden übrigen Zeitvorgaben die Einschränkungen $R(j_3) = \{4\}$ und $R(j_4) = \{4\}$, so daß auch diese Vorschriften implizit eingehalten werden und die entsprechenden Gleichungen entfallen können.

Da die vorgenommenen Beschränkungen der Kontrollschrittbereiche sich gegenseitig beeinflussen (im obigen Beispiel ist die Beschränkung von $R(j_3)$ und $R(j_4)$ erst durch die Einschränkung von $R(j_2)$ möglich), iteriert das OSCAR-System die beschriebene Optimierung solange, bis keine Einschränkungen mehr möglich sind. Das Verfahren terminiert spätestens dann, wenn alle Kontrollschrittberei-

che auf einen einzigen Schritt beschränkt sind, i. d. R. sind aber nur wenige Iterationen notwendig.

In die Iteration des Verfahrens ist jeweils eine erneute ASAP-ALAP-Analyse eingebunden, welche die vorgenommenen Beschränkungen auch auf die Operationen überträgt, die über Datenabhängigkeiten mit der betroffenen Operation verbunden sind.

Durch die Einschränkung der Kontrollschrittbereiche entfallen nicht nur einige Zeitvorgabevorschriften, sondern insbesondere auch die entsprechenden IP-Variablen, wodurch die Komplexität des erzeugten Gleichungssystems reduziert wird, und somit eine Beschleunigung der Synthese zu erwarten ist.

4.2 Der Verzicht auf Ganzzahligkeit

In der Einleitung dieser Arbeit wurde gezeigt, daß die *Ganzzahlige Programmierung* (IP) ein wesentlich komplexeres Problem als das der *Linearen Programmierung* (LP) darstellt, welche auf die Forderung der Ganzzahligkeit der Variablen verzichtet. Demnach sind große Geschwindigkeitssteigerungen der Synthese zu erwarten, wenn das verwendete IP-Modell durch ein LP-Modell ersetzt werden kann. Gebotys und Elmasry [GeEl93] stellen ein solches, 'gelockertes' LP-Modell (*Relaxed LP Model*) vor, welches dieselben Vorschriften wie das IP-Modell verwendet, jedoch auf die Ganzzahligkeit der Variablen verzichtet.

Nemhauser und Wolsey [NeWo88] zeigen, daß für jedes abgeschlossene, lineare System von Ungleichungen ein weiteres, lineares Ungleichungssystem existiert, welches als *Lineares Optimierungsproblem* gelöst werden kann (z. B. mit dem Simplex-Algorithmus) und dabei stets *ganzzahlige* Lösungen erzeugt. Voraussetzung für ein solches Gleichungssystem ist die Verwendung von Gleichungen eines bestimmten Typs (*Integral Facets*), auf die hier nicht tiefer eingegangen werden soll. Gebotys und Elmasry [GeEl93] zeigen, daß die von ihnen verwendeten Gleichungen einen Teil der Forderungen (und für viele Beispiele sogar sämtliche Voraussetzungen) erfüllen, die für ein solches Gleichungssystem notwendig sind. Daher ist zu erwarten, daß die für die Synthese aufgestellten Gleichungen in vielen Fällen bereits durch die Lineare Optimierung vollständig ganzzahlige Ergebnisse liefern. Weiterhin wird gezeigt, daß in dem Fall, in dem nicht alle erhaltenen Ergebnisse ganzzahlig sind, für viele Beispiele die ganzzahligen Variablen in einer optimalen Lösung unverändert erhalten bleiben [GeEl93].

Das in dieser Arbeit vorgestellte IP-Modell des OSCAR-Systems basiert in wesentlichen Vorschriften auf dem Ansatz [GeEl93]. Insbesondere die Operations- und Baustein-Zuordnungs-, sowie die Vorrangsvorschrift sind mit den im Ansatz [GeEl93] verwendeten Gleichungen praktisch identisch. Daher ist auch hier die Verwendung des LP-Modells in vielen Fällen möglich.

Das OSCAR-System verwendet *zwei* Syntheseschritte bei Einsatz des LP-Modells. Im ersten Schritt werden sämtliche Vorschriften unverändert erzeugt, aber auf die Ganzzahligkeit der Variablen wird verzichtet. Dieses Gleichungssystem

wird als Lineares Optimierungsproblem mit dem Simplex-Algorithmus gelöst. Enthält die berechnete Lösung ausschließlich ganzzahlige Ergebnisse, so ist sie mit der Lösung des Gleichungssystems als Ganzzahliges Optimierungsproblem identisch und damit optimal.

Sind nicht alle im ersten Schritt erhaltenen Variablen ganzzahlig, so muß ein zweiter Schritt durchgeführt werden. Das Gleichungssystem wird nun als IP-Modell aufgestellt, d. h. die Ganzzahligkeit der Lösung wird verlangt. Die in der ersten Lösung ganzzahlig erhaltenen Variablen werden jedoch in allen Gleichungen durch die ermittelten Belegungen ersetzt, so daß nur noch die nicht-ganzzahligen Variablen berechnet werden müssen. Diese Fixierung der Variablen führt i. d. R. zu einer erheblichen Beschleunigung der Berechnungszeit und liefert für viele Beispiele eine optimale Lösung. Im allgemeinen kann eine *optimale* Lösung jedoch nicht mehr garantiert werden. Es ist sogar möglich, daß keine ganzzahlige Lösung mehr gefunden wird.

Als Beispiel soll wieder der Elliptical-Wave-Filter (siehe Anhang D.1) angeführt werden. Tabelle 4.1 stellt die Rechenzeiten und Ergebnisse für das IP- und das LP-Modell gegenüber[1].

IP-Modell:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3/5	3/5	2/5	2/4	2/4	2/4
Multiplizierer ($ 40)	2/4	1/3	1/2	1/2	1/2	1/2
entstandene Kosten	$ 140	$ 100	$ 80	$ 80	$ 80	$ 80
Anzahl Variablen	341	498	639	674	809	944
Berechnungszeit	1s	2s	33s	24s	124s	180s
LP-Modell:	15 CS	16 CS	17 CS	18 CS	19 CS	20 CS
Addierer ($ 20)	3/5	3/5	3/5	3/5	3/5	-/5
Multiplizierer ($ 40)	2/4	1/3	1/2	1/2	1/2	-/2
entstandene Kosten	$ 140	$ 100	$ 100	$ 100	$ 100	N. S.
Anzahl Variablen	341	498	639	800	961	1122
nicht-ganzzahlig	0	62	105	162	196	165
Zeit Lauf 1	1s	1s	2s	3s	4s	4s
Zeit Lauf 2	-	1s	5s	11s	31s	(6s)
Berechnungszeit	1s	2s	7s	14s	35s	(10s)

Tabelle 4.1: Laufzeiten und Ergebnisse des IP- und des LP-Modells

Das LP-Modell erzeugt lediglich im Fall von 15 Kontrollschritten bereits im ersten Lauf eine vollständig ganzzahlige Lösung. In den weiteren Fällen ist ein zweiter Syntheseschritt notwendig. Dieser erzeugt für 16 Kontrollschritte noch dieselbe Lösung wie das IP-Modell, führt aber für 17, 18 und 19 Kontrollschritte zum Verlust der Optimalität, da 100 statt der minimalen 80 Kosteneinheiten

[1] Der Aufruf des OSCAR-Systems zur Berechnung dieser Daten lautet:
`oscar -v -p ewf.spec [-Ol] -a <steps> -l <costs> elliptic oscar_behaviour;`

benötigt werden. In diesen Fällen ist aber die Berechnungszeit für die erhaltene Lösung wesentlich kürzer als die für ein optimales Ergebnis.

Im Fall von 20 erlaubten Kontrollschritten bricht der MILP-Solver im zweiten Lauf erfolglos ab (*No Solution*), da mit den vorgenommenen Variablenbelegungen keine ganzzahlige Lösung mehr gefunden werden kann. In diesem Fall muß also zur Synthese das IP-Modell herangezogen werden.

Weitere Experimente (siehe Anhang D) zeigen, daß der LP-Ansatz auch mit dem vereinfachten Bindungsmodell (Abschnitt 2.11), sowie der Unterstützung von allgemeinem Chaining (Abschnitt 2.10) und der Verbindungsoptimierung (Abschnitt 2.8) verträglich ist.

Der im OSCAR-System optionale LP-Ansatz stellt also eine Heuristik dar, die in vielen Fällen eine erhebliche Beschleunigung der Synthese erzielt und zum Teil eine optimale Lösung liefert. Aus Zeitgründen ist in vielen Fällen auch eine suboptimale Lösung akzeptabel, wenn eine optimale Lösung durch unvertretbare Rechenzeiten nicht möglich ist.

Kapitel 5

Zusammenfassung und Bewertung

Abschließend sollen die Möglichkeiten des in dieser Arbeit vorgestellten Systems zur Mikroarchitektursynthese zusammengefaßt und bewertet, sowie mögliche Erweiterungen aufgezeigt werden.

5.1 Das IP-Modell

Das in der vorliegenden Arbeit vorgestellte OSCAR-System stellt ein System zur Mikroarchitektursynthese dar, das bei festgelegten Zeitanforderungen die Kosten eines Entwurfs minimiert (*time-constrained cost-optimized synthesis*).

Der Kern des Synthesesystems basiert auf einem Modell der Ganzzahligen Programmierung, welches sich insbesondere auf die Arbeiten von [GeEl93] und [RiJaLe92] stützt. Die in diesen Arbeiten entwickelten IP-Modelle werden in [LaMaDö94a] kombiniert und um weitere Möglichkeiten erweitert.

Im einzelnen unterstützt das im OSCAR-System verwendete IP-Modell folgende Anforderungen:

- Auf Basis des Modells der Ganzzahligen Programmierung werden die drei Teilaufgaben der Mikroarchitektursynthese Scheduling, Allocation und Binding *gleichzeitig* angegangen und gelöst. Somit werden bezüglich der definierten Kostenfunktion *global optimale* Syntheseergebnisse erzielt.
- Das IP-Modell unterstützt umfangreiche Bausteinbibliotheken, die neben Komponenten mit verschiedenen Ausführungszeiten (*mixed-speed components*) auch multi-funktionale (ALUs) und komplexe, mehrstufige Bausteine (z. B. MACs) enthalten können. Sämtliche Komponenten können auch als Pipeline-Bausteine ausgelegt sein.
- Gleichzeitig mit der Optimierung der Bausteinkosten können auch Kosten für Verbindungen und Register minimiert werden.

- Detaillierte Zeitvorgaben werden in Form von minimalen, maximalen und konstanten Zeitabständen zwischen Operationen unterstützt.

- Das IP-Modell erlaubt die Anwendung von *allgemeinem* (automatischem) *Chaining* (andere Systeme, z. B. [GeEl93], erlauben Chaining lediglich für manuell zusammengefaßte Operationen).

- Weiterhin unterstützt das IP-Modell alternative Versionen von Datenflüssen, aus denen automatisch die global kostengünstigste Version selektiert wird.

Die Modellierung des Syntheseproblems in Form eines IP-Modells auf einer mathematischen Grundlage bietet gute Ansatzpunkte für eine formale Verifikation des Synthesewerkzeugs.

5.2 Das OSCAR-System

Das OSCAR-Synthesesystem basiert auf dem oben beschriebenen Modell der Ganzzahligen Programmierung. In der vorliegenden Arbeit wurde dieses IP-Modell in das System implementiert. Weiterhin sind in dieser Arbeit Optimierungen zur Beschleunigung der Synthese und Erweiterungen des Systems vorgenommen worden, welche im folgenden zusammengestellt werden.

- Das OSCAR-System unterstützt eine detaillierte Spezifikation des Entwurfs, welche dem Benutzer weitreichende Eingriffsmöglichkeiten in den Synthesevorgang erlaubt.

- In der Eingabespezifikation des Synthesesystems sind Kontrollstrukturen in Form von Verzweigungen und Schleifen erlaubt.

- Das OSCAR-System eliminiert Redundanzen bei Datenabhängigkeiten und spezifizierten Zeitvorgaben.

- Wird keine Verbindungsoptimierung durchgeführt, so ist die Anwendung des vereinfachten Bindungsmodells möglich, welches zur Synthese wesentlich kürzere Rechenzeiten benötigt.

- Alternativ zur Lösung des Gleichungssystems als ganzzahliges Optimierungsproblem unterstützt das OSCAR-System auch ein heuristisches Verfahren (*Relaxed LP-Model*, [GeEl93]). Dieses verkürzt die Rechenzeiten des Synthesevorgangs erheblich und liefert dennoch in vielen Fällen optimale Syntheseergebnisse.

- Das System führt nach der Synthese eine Registerfaltung durch, welche die Anzahl der benötigten internen Register auf ein Minimum reduziert.

Das Synthesesystem OSCAR unterstützt somit wesentliche Anforderungen, die an ein System zur Mikroarchitektursynthese gestellt werden, und liefert global optimale Ergebnisse in akzeptablen Rechenzeiten (siehe Anhang D).

Das System unterstützt die Anbindung an weitere Entwurfswerkzeuge, insbesondere an das COMPASS-System [Compass91], und bietet somit die Möglichkeit, den erzeugten Entwurf auch auf die gewünschte Zieltechnologie abbilden zu lassen.

5.3 Ausblick

Im OSCAR-System werden bereits eine Reihe von Optimierungsverfahren zur Beschleunigung der Synthese eingesetzt. Sicherlich sind weitere Optimierungen des Systems möglich. Im folgenden sollen einige Ansatzpunkte für Verbesserungen und mögliche Erweiterungen des Synthesesystems angegeben werden.

- Die im OSCAR-System verwendete Verbindungsoptimierung stellt ein sehr einfaches Verbindungsmodell dar, welches die Bausteininstanzen des Entwurfs als Verbindungsknoten betrachtet. Die Möglichkeit, die Ports der Bausteine in der Verbindungsoptimierung zu berücksichtigen, besteht noch nicht. In [RiJaLe92] wird ein verfeinerter Ansatz beschrieben, der die Bausteinports und auch die benötigten Multiplexer in die Verbindungsminimierung einbezieht. Weiterhin ist die Unterstützung von Daten*bussen* als Alternative zur separaten Verdrahtung über Multiplexer wünschenswert.

- Das OSCAR-System unterstützt keine Arrays in der Eingabebeschreibung, welche auf entsprechende RAM-Bausteine abgebildet werden können. Für die Unterstützung von Arrays ist eine Beschränkung der Anzahl gleichzeitiger Zugriffe auf unterschiedliche Speicherzellen notwendig.

- Mit dem im OSCAR-System verwendeten IP-Modell ist eine Berücksichtigung der Anzahl der benötigten Kontrollschritte in der Bewertungsfunktion (mit wenig Mehraufwand) möglich. Hierdurch wird die Minimierung der Ausführungszeit des Entwurfs gleichzeitig zur Optimierung der Entwurfskosten möglich.

- Für komplexe Entwürfe ist eine mögliche Partitionierung des Syntheseproblems wünschenswert. Diese kann zwar im allgemeinen keinen global optimalen Entwurf garantieren, wird aber akzeptable Rechenzeiten auch für große Entwürfe ermöglichen. Im OSCAR-System kann eine Partitionierung z. B. an den Basisblockgrenzen des Entwurfs ansetzen.

- In [Ti94] wird ein polynomieller Algorithmus vorgestellt, der bestehende Ressource-Beschränkungen für einen Entwurf zu den Kontrollschrittbereichen der Operationen in Beziehung setzt und diese entsprechend einschränkt. Eine Anwendung dieses Verfahrens im OSCAR-System führt

direkt zu einer Reduzierung der Anzahl von IP-Variablen und läßt somit eine Beschleunigung des Synthesevorgangs erwarten.

- Im Bereich des Operations Research sind eine Reihe weiterer Heuristiken für die Ganzzahlige Optimierung entwickelt worden. Eine Anwendung solcher Verfahren könnte im OSCAR-System weitere Beschleunigungen des Syntheseablaufes erzielen.

Anhang A

Formelsammlung

Die insbesondere in Kapitel 2 verwendeten mathematischen Notationen, Definitionen und Gleichungen werden an dieser Stelle noch einmal übersichtlich zusammengefaßt.

In Tabelle A.1 sind die in dieser Arbeit verwendeten Grundmengen zusammengestellt. Die Tabellen A.2 und A.3 geben einen Überblick über sämtliche Relationen, die für den Aufbau des IP-Modells notwendig sind. Die in den Vorschriften verwendeten Konstanten und Variablen werden in den Tabellen A.4 und A.5 zusammengefaßt. Schließlich sind auf den Seiten 67 bis 69 alle im OSCAR-System verwendeten Vorschriften übersichtlich aufgeführt.

$I \subset \mathbb{N}$, $i \in I$	erlaubte *Kontrollschritte* (CS) $i \in \{1, \ldots, i_{\max}\}$
$J \subset \mathbb{N}$, $j \in J$	Menge aller *Operationen* $j \in \{1, \ldots, j_{\max}\}$
$K \subset \mathbb{N}$, $k \in K$	*Instanzen* von Bausteintypen, Funktionseinheiten $k \in \{1, \ldots, k_{\max}\}$
$M \subset \mathbb{N}$, $m \in M$	Baustein-*Komponenten*, Typen von Bausteinen $m \in \{1, \ldots, m_{\max}\}$
$G \subset \mathbb{N}$, $g \in G$	bekannte *Operationstypen* (i. d. R. Funktionen) $g \in \{1, \ldots, g_{\max}\}$
$D \subset \mathbb{N}$, $d \in D$	(Teil-) *Datenflüsse* des Entwurfes $d \in \{1, \ldots, d_{\max}\}$
$V \subset \mathbb{N}$, $v \in V$	alternative *Versionen* eines Datenflusses d $v \in \{1, \ldots, v_{\max}(d)\}$
$Y \subset J$, $y \in Y$	*Makrooperationen*, Untermenge aller Operationen $y \in \{1, \ldots, j_{\max}\}$, $y \in Y$

Tabelle A.1: Notation der Mengen

$j_1 \prec j_2$	*Datenabhängigkeit* der Operation j_2 von Operation j_1
$j_1 \prec\!\!\prec j_2$	*Chaining-Möglichkeit* von Operation j_1 und Operation j_2
$j_s \prec j_d$	*Lebenszeit* der Variablen, die das Ergebnis der Operation j_s zur Weiterverarbeitung durch Operation j_d speichert
$\mathrm{ASAP}(j) \in I$	*ASAP-Zeitpunkt*, erster möglicher Kontrollschritt für den Start der Operation j
$\mathrm{ALAP}(j,k) \in I$	*ALAP-Zeitpunkt*, letzter möglicher Kontrollschritt für den Start der Operation j auf Instanz k
$\mathrm{ALAP}(j) \in I$	allg. *ALAP-Zeitpunkt*, letzter möglicher Kontrollschritt für den Start der Operation j, $\mathrm{ALAP}(j) := \max_k(\mathrm{ALAP}(j,k))$
$R(j,k) \subseteq I$	*Kontrollschrittbereich* der Operation j bei Ausführung auf der Instanz k, $R(j,k) := \{\mathrm{ASAP}(j), \ldots, \mathrm{ALAP}(j,k)\}$
$R(j) \subseteq I$	maximaler *Kontrollschrittbereich* der Operation j, $R(j) := \bigcup_k R(j,k)$
$f(j) \in G$	*Operationstyp* der Operation j (i. d. R. eine Funktion)
$G(m) \subseteq G$	*Funktionalität* der Komponente m; Menge der Operationstypen, die auf dem Bausteintyp m ausführbar sind
$\mathrm{type}(k) \in M$	*Bausteintyp* der Instanz k
$F(k) \subseteq G$	Menge der ausführbaren *Operationstypen* auf Instanz k, $F(k) := G(\mathrm{type}(k))$
$\ell(j,k) \in \mathbb{N}_0$	*Latenzzeit:* Zeit in Kontrollschritten, die die Instanz k zur Übernahme der Argumente für Operation j benötigt
$C(j,k) \in \mathbb{N}_0$	*Ausführungzeit:* Zeit in Kontrollschritten, die die Instanz k zur Ausführung der Operation j benötigt
$C(j) \in \mathbb{N}_0$	maximale *Ausführungzeit* der Operation j, $C(j) := \max_k(C(j,k))$

Tabelle A.2: Notation der Relationen (Teil a)

$V(d) \subset \mathbb{N}$	alternative *Versionen* des Datenflusses d
$v(j) \in V$	*Version*, der die Operation j angehört
$d(j) \in D$	*Datenfluß*, dem die Operation j angehört
$\text{macro}(y) \subseteq J \setminus Y$	Menge der *einfachen* Operationen, die als Alternative zur Makrooperation y verwendet werden können
$\text{chain}(j_1, k_1, j_2, k_2) \in \{0, 1\}$	Möglichkeit von *Chaining* bei Ausführung der Operationen j_1 auf k_1 und j_2 auf k_2
$\text{chain}(j_1, j_2) \in \{0, 1\}$	Möglichkeit von *Chaining* der Operationen j_1 und j_2
$\text{time}(j, k) \in \mathbb{R}$	*phys. Ausführungszeit* der Operation j auf Instanz k (in ns)
$\text{ARCS}(J, i) \subset \mathcal{P}(J)$	Aufzählung aller maximalen Mengen von Operationspaaren $(j_s \prec j_d)$ aus J, deren Variablen-*Lebenszeit* den Kontrollschritt i kreuzt
$\text{CHAINS}(J') \subseteq \mathcal{P}(J')$	Menge der maximalen *Ketten* von Operationen aus J' für allgemeines Chaining
$\text{Chain}(J') \subseteq J'$	*Operationenkette*, maximale Kette von Operationen aus J', für die paarweise Chaining möglich ist

Tabelle A.3: Notation der Relationen (Teil b)

$T \in \mathbb{N}_0$	*Zeitschranke* in Kontrollschritten für Zeitbedingungen
$time_{cycle} \in \mathbb{R}$	*Zykluszeit* des Systemtaktes (in ns)
$\ell_{cnct} \in \mathbb{R}$	*Verbindungsverzögerung*, Zeitverlust (in ns), der durch Datenverbindungen im Datenfluß entsteht (z. B. durch Multiplexer und Verdrahtung)
$\ell_{ctrl} \in \mathbb{R}$	*Kontrollverzögerung*, Zeitverlust (in ns), der durch die Ansteuerung der Bausteininstanzen und durch den Kontroller selbst entsteht
$c_m \in \mathbb{R}$	*Bausteinkosten:* Kosten einer Komponente vom Typ m (i. d. R. proportional zum Platzbedarf des Bausteins)
$c_k \in \mathbb{R}$	*Instanzkosten:* Kosten der Bausteininstanz k
$c_{k_1,k_2} \in \mathbb{R}$	*Verbindungskosten:* Kosten, die eine Verbindung von Instanz k_1 zu Instanz k_2 verursacht (i. d. R. proportional zur maximal benötigten Bitbreite)
$c_r \in \mathbb{R}$	*Registerkosten:* Kosten eines Registers zur Aufnahme von Zwischenergebnissen
$c_{\max} \in \mathbb{R}$	*Kostenlimit:* maximale Kosten, die bei der Synthese des Entwurfes entstehen können (i. d. R. bekannt durch vorhergehende Syntheseläufe)

Tabelle A.4: Notation der Konstanten

$x_{i,j,k} \in \{0,1\}$	Entscheidungsvariable für *Scheduling* und *Binding*; gleich 1, falls Operation j in Kontrollschritt i auf Instanz k ausgeführt wird
$b_k \in \{0,1\}$	Entscheidungsvariable zur *Allocation*; gleich 1, falls die Bausteininstanz k verwendet wird
$w_{k_1,k_2} \in \{0,1\}$	Entscheidungsvariable für die *Verbindungsminimierung*; gleich 1, falls von Instanz k_1 zu k_2 Daten transportiert werden, also eine Verbindung besteht
$u_{d,v} \in \{0,1\}$	Entscheidungsvariable für *alternative Versionen*; gleich 1, falls die Version v aus dem Datenfluß d selektiert wird
$r \in \mathbb{N}_0$	*Registeranzahl:* die Anzahl benötigter Register bei Einsatz von Registeroptimierung
$b_m \in \mathbb{N}_0$	*Instanzenanzahl:* die Anzahl benötigter Instanzen des Bausteintyps m bei Bindung an Komponenten

Tabelle A.5: Notation der Variablen

Bewertungsfunktion (*Objective Function*):

$$\sum_{k \in K} c_k * b_k \; + \sum_{k_1,k_2 \in K} c_{k_1,k_2} * w_{k_1,k_2} \; + c_r \; * \; r \tag{A.1}$$

Kostenbeschränkung (Cost Limit Constraint):

$$\sum_{k \in K} c_k * b_k \; + \sum_{k_1,k_2 \in K} c_{k_1,k_2} * w_{k_1,k_2} \; + \; c_r * r \; \leq \; c_{\max} \tag{A.2}$$

Operations-Zuordnungsvorschrift (*Operation Assignment Constraint*):

$$\begin{aligned} &\forall d \in D, \quad \forall v \in V(d), \quad \forall j \in J \setminus Y \text{ mit } v(j) = v, d(j) = d : \\ &\sum_{\substack{k \in K: \\ f(j) \in F(k)}} \sum_{i \in R(j,k)} x_{i,j,k} + \sum_{\substack{y \in Y: \\ j \in \text{macro}(y)}} \sum_{\substack{k \in K: \\ f(y) \in F(k)}} \sum_{i \in R(y,k)} x_{i,y,k} = u_{d,v} \end{aligned} \tag{A.3}$$

Gegenseitiger Ausschluß (*Mutual Exclusion Constraint*):

$$\forall d \in D : \sum_{v \in V(d)} u_{d,v} = 1 \tag{A.4}$$

Baustein-Zuordnungsvorschrift (*Resource Assignment Constraint*):

$$\begin{aligned} &\forall i \in I, \; \forall k \in K \text{ mit } \exists j \in J, f(j) \in F(k), i \in R(j,k) : \\ &\sum_{\substack{j \in J: \\ f(j) \in F(k)}} \sum_{\substack{i' = i \\ i \in R(j,k)}}^{i+\ell(j,k)-1} x_{i',j,k} \quad \leq \quad b_k \end{aligned} \tag{A.5}$$

Instanzen-Allokationsvorschrift (*Instance Allocation Constraint*):

$$\forall k \in K \setminus \{k_{\max}\} : \quad \text{type}(k) = \text{type}(k+1) \quad \Rightarrow \quad b_k \leq b_{k+1} \tag{A.6}$$

Vorrangsvorschrift (*Precedence Constraint*):

$$\begin{aligned} &\forall j_1, j_2 \in J \text{ mit } j_1 \prec j_2 \; , \\ &\forall i \in \quad \{\text{ASAP}(j_2) + \text{chain}(j_1,j_2), \ldots, \text{ALAP}(j_2)\} \\ &\quad \cap \quad \{\text{ASAP}(j_1), \ldots, \text{ALAP}(j_1) + C(j_1) - 1\} \quad : \\ &\qquad \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \sum_{\substack{i_2 \in R(j_2,k_2): \\ i_2 \leq i - \text{chain}(j_1,j_2)}} x_{i_2,j_2,k_2} \quad + \\ &\qquad \sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \sum_{\substack{i_1 \in R(j_1,k_1): \\ i - (C(j_1,k_1)-1) \leq i_1}} x_{i_1,j_1,k_1} \quad \leq \quad 1 \end{aligned} \tag{A.7}$$

Tabelle A.6: Vorschriften des IP-Modells (Teil a)

Konstante Zeitvorgabevorschrift (*Constant Timing Constraint*):

$$\forall i_1 \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ i_2 \neq i_1 \pm T}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad \text{(A.8)}$$

Minimale Zeitvorgabevorschriften (*Minimum Timing Constraints*):

$$\forall i_1 \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 < i_1 + T) \\ \wedge (i_2 > i_1 - T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad \text{(A.9)}$$

$$\forall i \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \ \sum_{\substack{i_1 \in R(j_1): \\ (i_1 \geq i)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 < i + T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad \text{(A.10)}$$

Maximale Zeitvorgabevorschriften (*Maximum Timing Constraints*):

$$\forall i_1 \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 > i_1 + T) \\ \vee (i_2 < i_1 - T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad \text{(A.11)}$$

$$\forall i \in R(j_1) \ :$$

$$\sum_{\substack{k_1 \in K: \\ f(j_1) \in F(k_1)}} \ \sum_{\substack{i_1 \in R(j_1): \\ (i_1 \leq i)}} x_{i_1,j_1,k_1} \ + \ \sum_{\substack{k_2 \in K: \\ f(j_2) \in F(k_2)}} \ \sum_{\substack{i_2 \in R(j_2): \\ (i_2 > i + T)}} x_{i_2,j_2,k_2} \quad \leq \quad 1 \qquad \text{(A.12)}$$

Tabelle A.7: Vorschriften des IP-Modells (Teil b)

Verbindungsvorschriften (*Interconnection Constraints*):

$$\forall j_1, j_2 \in J \text{ mit } j_1 \prec j_2,$$
$$\forall k_1, k_2 \in K \text{ mit } f(j_1) \in F(k_1), f(j_2) \in F(k_2):$$

$$\left(\sum_{i_1 \in R(j_1,k)} x_{i_1,j_1,k_1} + \sum_{i_2 \in R(j_2,k)} x_{i_2,j_2,k_2} \right) - 1 \quad \leq \quad w_{k_1,k_2} \qquad \text{(A.13)}$$

$$0 \quad \leq \quad w_{k_1,k_2} \qquad \text{(A.14)}$$

Registeroptimierungsvorschrift (*Register Allocation Constraint*):

$$\forall i \in I,\ \forall \{j_s \prec j_d\} \in \mathrm{ARCS}(J,i)\ :$$

$$\sum_{\{j_s \prec j_d\}} \left(\sum_{\substack{k \in K: \\ f(j_s) \in F(k)}} \sum_{\substack{i_1 \in R(j_s,k): \\ i_1 \leq i - C(j_s,k)+1}} x_{i_1,j_s,k} + \sum_{\substack{k \in K: \\ f(j_d) \in F(k)}} \sum_{\substack{i_2 \in R(j_d,k): \\ i_2 > i - \ell(j_d,k)+1}} x_{i_2,j_d,k} \right. \qquad \text{(A.15)}$$

$$\left. - \sum_{\substack{k \in K: \\ f(j_d) \in F(k)}} \sum_{\substack{i_3 \in R(j_d,k): \\ i_3 \leq i - \ell(j_d,k)+1}} x_{i_3,j_d,k} - \sum_{\substack{k \in K: \\ f(j_s) \in F(k)}} \sum_{\substack{i_4 \in R(j_s,k): \\ i_4 > i - C(j_s,k)+1}} x_{i_4,j_s,k} \right) \quad \leq \quad 2r$$

Verkettungsvorschrift (*Chaining Constraint*):

$$\forall \mathit{Chain} \in \mathrm{CHAINS}(J), \quad \forall i \in I:$$

$$\sum_{\substack{j \in \mathit{Chain}: \\ i \in R(j)}} \sum_{\substack{k \in K: \\ f(j) \in F(k), \\ C(j,k)=1}} (\mathrm{time}(j,k) + \ell_{cnct}) * x_{i,j,k} \quad \leq \quad \mathit{time}_{cycle} - \ell_{ctrl} \qquad \text{(A.16)}$$

Tabelle A.8: Vorschriften des IP-Modells (Teil c)

Anhang B

Installations- und Benutzerhandbuch

Das im Rahmen dieser Arbeit implementierte Programm OSCAR soll an dieser Stelle dokumentiert werden.

Zunächst wird die Installation der Software beschrieben. Weiterhin werden die zum Aufruf des Programmes notwendigen Parameter und Optionen vorgestellt und die im OSCAR-System verwendeten Dateien erläutert. Eine vollständige Liste der Fehlermeldungen bildet den Abschluß dieses Kapitels.

B.1 Installation des OSCAR-Systems

B.1.1 Das Programmpaket

Das gesamte OSCAR-System ist zusammengefaßt in der Datei `oscar.tar.Z` gespeichert. Zur Installation des Systems muß diese Datei entpackt und das ausführbare Programm `oscar` erzeugt werden.

Es ist sinnvoll, ein eigenes Verzeichnis namens `oscar` anzulegen, in welches das gesamte System abgelegt wird. Dieses Verzeichnis sollte zusätzlich zum Programmpaket `oscar.tar.Z` noch mindestens 10 MB Platz bieten. Das Entpacken des Systems kann nun in zwei Schritten erfolgen:

Der Aufruf von

```
uncompress oscar.tar.Z
```

erzeugt eine ca. 3 MB große Datei `oscar.tar`, welche mit

```
tar -xvf oscar.tar
```

entpackt werden kann.

Folgende sechs Verzeichnisse enthalten nun das gesamte OSCAR-System:

`doc` enthält die Dokumentation des Systems.

`src` enthält die Quelldateien des OSCAR-Systems. Diese sind aufgeteilt in einzelne Programm-Module (siehe Anhang C), welche in getrennten Unterverzeichnissen abgelegt sind.

`obj` enthält symbolische Verweise auf die erzeugten Objekt-Dateien (notwendig zur Übersetzung des Systems).

`bin` enthält einen symbolischen Verweis auf die Programmdatei `oscar`. Dieses Verzeichnis muß daher in den Suchpfad der Shell aufgenommen werden.

`examples` enthält Eingabebeispiele für OSCAR, welche im Laufe der Entwicklung des Systems entstanden sind. Insbesondere die in dieser Arbeit vorgestellten Beispiele sind hier zu finden. Zum Testen des Systems sollten diese Dateien ins Arbeitsverzeichnis `work` kopiert werden.

`work` kann als Arbeitsverzeichnis des Benutzers verwendet werden.

Vor dem Einsatz des OSCAR-Systems muß nun noch das ausführbare Programm `oscar` erzeugt werden.

B.1.2 Die Übersetzung des Programmes

Das OSCAR-System ist nicht fest an ein bestimmtes Betriebssystem oder bestimmte Hardware-Voraussetzungen gebunden. Benötigt wird ein UNIX-ähnliches Betriebssystem und ein C++ -Compiler.

Entwickelt wurde das System unter SunOS auf einer SPARCstation. Aber auch unter den Betriebssystemen DomainOS, Linux und AmigaDOS wurde OSCAR erfolgreich übersetzt und getestet.

Konkret wird zur Übersetzung des OSCAR-Systems das GNU-Compiler-Paket verwendet. Hieraus werden der C++ -Compiler `g++` (Version $\geq$ 2.4.3), der Lexical-Analyzer `flex` (Version 2.3.8) und der Parser-Generator `bison` (Version $\geq$ 1.2.1) benötigt.

Zur Anpassung an die jeweilige Betriebssystemumgebung sind in der Datei `src/makefile.macros` i. d. R. einige Anpassungen notwendig. Insbesondere die zu verwendenden Pfade und Compiler-Aufrufe müssen hier eingestellt werden. Die Datei ist ausführlich kommentiert, so daß eine Anpassung keine Schwierigkeiten machen sollte.

Die eigentliche Übersetzung erfolgt durch Aufruf von `make` im Verzeichnis `src`. Automatisch werden hierdurch sämtliche Programm-Module übersetzt und zur ausführbaren Datei `oscar` zusammengefügt. Zusätzlich wird zu jedem Modul

ein ausführbares Programm erzeugt, das zum Testen des jeweiligen Moduls verwendet werden kann.

Der Übersetzungsvorgang sollte ohne Warnungen und Fehlermeldungen ablaufen. Andernfalls müssen die Einstellungen in der Datei `src/makefile.macros` überprüft und ggf. korrigiert werden.

Zum Betrieb des OSCAR-Systems wird außer dem Programm `oscar` auch das Programm `lp_solve` [Be93] benötigt. Beide Dateien sollten im Verzeichnis `bin` vorhanden sein, welches in den Suchpfad der verwendeten Shell aufzunehmen ist.

B.2 Die Arbeit mit dem OSCAR-System

B.2.1 Der Aufruf des Programmes

Das in dieser Arbeit entwickelte System arbeitet Shell-orientiert. Eine interaktive Bedienung des Programmes ist nicht vorgesehen. Vielmehr ist das Programm dahingehend konzipiert, daß es leicht in eine (darüberliegende) Benutzeroberfläche zur Steuerung der Synthese integriert werden kann.

Das OSCAR-System wird in der Shell-Umgebung aufgerufen und führt die gesamte Synthese automatisch durch. Die Syntax zum Aufruf des Systems lautet:

```
oscar [options] <entity> <architecture>
```

Die zu synthetisierende VHDL-Verhaltensbeschreibung wird in Form von Entwurfs- und Architekturbezeichnung spezifiziert. Zusätzlich können Optionen angegeben werden, die den Syntheseablauf beeinflussen. Diese Optionen werden im folgenden Abschnitt detailliert beschrieben.

B.2.2 Die Optionen und Parameter

Eine Kurzbeschreibung sämtlicher Optionen und Parameter, die beim Aufruf des Systems angegeben werden können, wird ausgeben, wenn das Programm allein durch Eingabe von `oscar` gestartet wird. Diese Übersicht ist in Tabelle B.1 angegeben.

Die genaue Funktion der möglichen Optionen wird im folgenden erläutert:

`-q|-v` unterdrückt die Ausgaben, die beim Syntheselauf angezeigt werden, bzw. gibt ein detailliertes Ablauf-Protokoll aus;

`-De` unterdrückt die Erzeugung der IP-Gleichungen, wodurch eine manuell erstellte oder geänderte Datei verwendet werden kann;

`-Dd` unterdrückt die Erzeugung der IP-Informationsdatei;

```
================================================================================
***   OSCAR   -   High Level Synthesis System   -   Version 1.4.3   ***
================================================================================

(written 1993,'94 by B. Landwehr and R. Doemer)

Usage: oscar [options] <entity> <architecture>
Options:              Function:                          Default:
-q                    be quiet                           off
-v                    be verbose                         off
-De                   don't generate equation file       off
-Dd                   don't generate information file    off
-Do                   don't generate step list file      off
-Dm                   don't call MILP solver             off
-Sl                   stop after reading libraries       off
-Sr                   stop after reading input files     off
-Sg                   stop after genetic algorithm       off
-Si                   stop after IP model extraction     off
-Se                   stop after equation generation     off
-Ss                   stop after solving equations       off
-Ob                   use simplified resource binding    off
-Oo                   minimize RW-operations             off
-Oi                   optimize interconnect costs        off
-Os                   minimize RW schedule ranges        off
-Oc                   optimize steps using chaining      off
-Od                   optimize operation dependencies    off
-Om                   optimize macro operations          off
-Ot                   optimize timing specifications     off
-Ol                   use relaxed LP model heuristic     off
-Og                   use genetic algorithm dummy        off
-Ow                   minimize register bit widths       off
-Of                   perform register folding           off
-a <# control steps>  specify additional control steps   0
-l <# costs limit>    specify limit to design costs      None
-i <vhdl file>        specify VHDL input file            <entity>'.vhdl'
-p <spec file>        specify specification input file   <entity>'.spec'
-e <equation file>    specify equation output file       <entity>'.eqn'
-s <solution file>    specify solution output file       <entity>'.lps'
-r <error file>       specify error output file          <entity>'.err'
-d <dump file>        specify IP model information file  <entity>'.ipm'
-o <result file>      specify control step list file     <entity>'.csl'
-f <function lib>     specify function library           'Functions.lib'
-m <ip_solver>        specify MILP solver                'lp_solve'
Ready.
```

Tabelle B.1: Optionen und Parameter des OSCAR-Systems

`-Do` unterdrückt die Ausgabe der Kontrollschrittliste;

`-Dm` schaltet den Aufruf des MILP-Solvers ab, wodurch eine manuell erstellte oder geänderte Lösungsdatei verwendet werden kann;

`-Sl|-Sr|-Sg|-Si|-Se|-Ss` stoppt die Synthese nach bestimmten Schritten (siehe Tabelle B.1), wodurch z. B. eine Syntax-Prüfung der Eingabe-Dateien ohne den sonst notwendigen Syntheselauf möglich wird;

`-Ob` verwendet das vereinfachte Bindungsmodell (Bindung an Komponenten, statt an Instanzen) (s. Abschnitt 2.11); kann nicht gleichzeitig zu Option `-Oi` verwendet werden;

`-Oo` nimmt Lese- und Schreiboperationen nach Möglichkeit aus dem IP-Modell heraus, so daß für Ein- und Ausgabeoperationen keine IP-Variablen erzeugt werden; kann nicht gleichzeitig zu den Optionen `-Oi` und `-Os` verwendet werden;

`-Oi` schaltet die Verbindungsminimierung ein (s. Abschnitt 2.8); kann nicht gleichzeitig zu den Optionen `-Ob` und `-Oo` verwendet werden;

`-Os` schränkt die Kontrollschrittbereiche von Lese- und Schreiboperationen dahingehend ein, daß diese Operationen im jeweils ersten möglichen Kontrollschritt ausgeführt werden; kann nicht gleichzeitig zu Option `-Oo` verwendet werden;

`-Oc` erlaubt die Verwendung von allgemeinem Chaining (s. Abschnitt 2.10);

`-Od` eliminiert redundante Datenabhängigkeitskanten in den Datenflußgraphen (s. Abschnitt 4.1.1);

`-Om` schaltet die Erzeugung von (einfachen) Operationen als Alternative zu (komplexen) Makrooperationen ein (s. Abschnitt 2.4.2);

`-Ot` schaltet die Optimierung von Zeitvorgaben ein (s. Abschnitt 4.1.2);

`-Ol` schaltet die Anwendung des heuristischen LP-Modells ein, durch welche die Optimierung des Gleichungssystems in *zwei* Schritten vorgenommen wird (s. Abschnitt 4.2); kann zu Verlust der Optimalität führen!

`-Og` schaltet die Erzeugung alternativer Datenflußgraphen durch algebraische Transformationen ein (s. Abschnitt 2.4.3);

`-Ow` schaltet die Bitbreitenminimierung für Register ein (s. Abschnitt 3.3.4);

`-Of` schaltet die Faltung interner Register ein (s. Abschnitt 3.3.4);

`-a <# control steps>` spezifiziert die Anzahl zusätzlicher Kontrollschritte, die zu jedem Kontrollblock hinzugefügt werden;

`-l <cost limit>` spezifiziert die Maximalkosten des Entwurfs (s. Abschnitt 2.3.2);

`-i <vhdl file>` spezifiziert die VHDL-Eingabedatei (s. Abschnitt 3.1);

`-p <spec file>` spezifiziert die Eingabedatei, welche die Synthesespezifikationen enthält (s. Abschnitt 3.3.2);

`-e <equation file>` spezifiziert die Gleichungsdatei, welche als Eingabe für den MILP-Solver generiert wird;

`-s <solution file>` spezifiziert die Datei, in welche der MILP-Solver die Lösung des Gleichungssystems ablegt;

`-r <error file>` spezifiziert die Datei, in die Fehlermeldungen des MILP-Solvers geschrieben werden;

`-d <dump file>` spezifiziert die IP-Informationsdatei, welche detaillierte Informationen zum IP-Modell enthält (s. Abschnitt C.2.1);

`-o <result file>` spezifiziert die Datei, in welche die Kontrollschrittliste geschrieben wird (s. Abschnitt 3.3.5);

`-f <function lib>` spezifiziert die verwendete Funktionsbibliothek (*Function Attribute Library*);

`-c <component lib>` spezifiziert die verwendete Bausteinbibliothek (*Component Library*);

`-m <ip_solver>` spezifiziert den aufzurufenden MILP-Solver; dieser muß Ein-Ausgabe-kompatibel mit `lp_solve` [Be93] sein;

B.2.3 Der Programmablauf

Das OSCAR-System gibt ein detailliertes Protokoll des Synthesevorgangs aus, wenn es mit der Option `-v` gestartet wird. Die einzelnen Schritte dieses Ablaufes sind in Kapitel 3 ausführlich beschrieben.

An dieser Stelle soll beispielhaft ein solches Protokoll der OSCAR-Synthese angegeben werden. Das in Kapitel 3 vorgestellte Beispiel `sample` kann synthetisiert werden mit folgendem Aufruf:

```
oscar -v -Od -Ot -Of sample behaviour
```

Das erzeugte Protokoll dieses Syntheselaufes ist auf den Seiten 77 und 78 angegeben.

```
============================================================================
***   OSCAR   -   High Level Synthesis System   -   Version 1.4.3   ***
============================================================================

(written 1993,'94 by B. Landwehr and R. Doemer)

Synthesizing architecture 'BEHAVIOUR' of entity 'SAMPLE'...

Reading function attribute library 'Functions.lib'...
        Parsing successful
Reading VHDL specification 'sample.vhdl'...
        Number of source code lines     =    33
Generating data paths...
        Number of control blocks        =     1
Reading synthesis specification 'sample.spec'...
        Global clock signal name:       'CLOCK'
        Global reset signal name:       'RESET'
        System controller delay         =    50ns
        System interconnect delay       =    25ns
        System effective cycle time     =   925ns
        System cycle time               =  1000ns
Skipping alternative dataflow generation...
        Genetic algorithm optimization is disabled
Skipping complex operation expansions...
        Macro operation optimization is disabled
Extracting integer programming model...
        Number of additional steps      =     0
        Upper limit to design costs:    (NONE)
        Simplified binding model is     OFF
        RW-operation minimization is    OFF
        Interconnect minimization is    OFF
        RW range minimization is        OFF
        Chaining optimization is        OFF
        Dependency optimization is      ON
        Timing optimization is          ON
        Relaxed LP model heuristic is   OFF
        Register width minimization is  OFF
        Register folding is             ON
Extracting operations...
        Number of operations            =     9
Extracting control blocks...
        Block   1:   1 versions, local freedom    1 control steps
Integrating specified precedences...
        Number of added precedences     =     0
Selecting usable components...
        Number of usable components     =     4
Eliminating redundant operation dependencies...
        Number of dependencies          =    10
        Number of redundancies          =     0
Ignoring possible chainings...
        Chaining is disabled
Computing ASAP- and ALAP-ranges...
        Block   1: minimum length    3 control steps
        Minimum number of control steps =     3
Adding freedom to ALAP-timings...
        Final number of control steps   =     4
Integrating scheduling specifications...
        Number of user schedules        =     3
...
```

Protokoll eines OSCAR-Syntheselaufes (Teil 1)

```
...
Integrating timing specifications...
        Number of user timings          =     1
Skipping optimizations for RW operations...
        RW range and operation minimization are disabled
Analyzing and optimizing timing specifications...
        Number of optimizer passes      =     1
        Number of redundant timings     =     0
Computing necessary instances...
        Number of function units        =     3
        Number of ports/registers       =     5
        Maximum costs for instances:            $    80000
Integrating binding specifications...
        Number of user bindings         =     0
Computing interconnection costs...
        Number of interconnections      =    13
        Maximum costs for interconnections:     $    20800
Dumping integer programming information (file 'sample.ipm')...
        Dumping successful
Generating integer programming equations (file 'sample.eqn')...
        Number of equations =      32
        Number of variables =      22
        Number of lines     =      80
        Number of bytes     =    3070
Calling MILP solver 'lp_solve'...
        Time to solve the equations:          1s
Reading linear programming solution 'sample.lps'...
        Value of objective function:        -75.000000
        Unscaled value (min. costs):      60000.000000
Verifying integer programming solution...
        Computed solution seems to be correct!
Analyzing binding of allocated instances...
        Number of used control steps    =     4
        Number of allocated instances   =     7
        Instance load factor            =    32%  (= 9 / 28)
Analyzing design costs in detail...
        IO-Registers:      5 instances          $        0
        Components:        2 functional units   $    60000
        Interconnect:      8 connections        $    12800
        ------------------------------------------------
        Total costs:                            $    72800
Computing necessary internal registers...
        Number of external ports        =     5
        Number of internal registers    =     6
Skipping register bit width optimization...
        Original register bit widths are used
Performing register folding...
        Number of eliminated registers  =     2
Dumping integer programming information (file 'sample.ipm')...
        Dumping successful
Generating control step list (file 'sample.csl')...
        Generation successful
Cleaning up...
Ready.
```

Protokoll eines OSCAR-Syntheselaufes (Teil 2)

B.2.4 Verwendete Dateien

Das OSCAR-System verwendet zur Synthese eines Entwurfs eine Reihe von Dateien, auf deren Format im folgenden kurz eingegangen werden soll:

- Zur Synthese benötigt OSCAR zwei Bibliotheken. Zum einen ist eine Bibliothek sämtlicher im Entwurf verwendeter Funktionen (`Functions.lib`) notwendig, die Angaben über die Anzahl der Funktionsargumente u. ä. enthält. Zum zweiten wird eine Bausteinbibliothek (`Component.lib`) benötigt, die alle zur Verfügung stehenden Komponenten beinhaltet. Der Aufbau beider Bibliotheken ist in [LaMa93] ausführlich beschrieben.

- Der eigentliche Entwurf wird über zwei Dateien spezifiziert. Das algorithmische Verhalten wird in VHDL [IEEE88] beschrieben (`<project>.vhdl`). Zusätzlich können über eine Spezifikationsdatei (`<project>.spec`) weitere Angaben zur Synthese des Entwurfs gemacht werden. Das Format dieser Datei ist im folgenden Unterabschnitt angegeben.

- Für den Datenaustausch mit dem MILP-Solver `lp_solve` werden drei Dateien verwendet. Die Eingabe des Programmes bildet eine Gleichungsdatei (`<project>.eqn`), die vom OSCAR-System erzeugt wird. Die Ausgabe ist unterteilt in die Lösungsdatei (`<project>.lps`) und eine Datei zur Notation von Fehlermeldungen (`<project>.err`). Zum Format dieser Dateien sei auf [Be93] verwiesen.

- Die innerhalb des IP-Modells verwendeten Datenstrukturen werden vor dem Start des MILP-Solvers und vor der Erzeugung der Kontrollschrittliste in eine Informationsdatei (`<project>.ipm`) geschrieben. Das Format dieser Datei ist nicht näher spezifiziert, sie dient dem Benutzer lediglich als detaillierte Informationsquelle über das aufgestellte IP-Modell.

- Das Ergebnis der OSCAR-Synthese wird in Form einer Kontrollschrittliste (`<project>.csl`) ausgegeben. Diese Datei enthält sämtliche Informationen, die zur Steuerwerksynthese und Netzlistenerzeugung benötigt werden. Das Format der Kontrollschrittliste ist im Unterabschnitt B.2.4.2 spezifiziert.

B.2.4.1 Format der Spezifikationsdatei

Die Syntax der Synthese-Spezifikationsdatei ist an VHDL angelehnt. Die Notation ist *case-insensitiv* (keine Unterscheidung von Groß- und Kleinschreibung), Kommentare werden durch doppelte Minuszeichen gekennzeichnet und gelten bis an das Zeilenende.

Der Aufbau der Datei ist über folgende erweiterte Backus-Naur-Form (EBNF) definiert:

```
<specfile>       ::= <spec_list>
<spec_list>      ::= <specification> {<specification>}
<specification>  ::= SPECIFICATION FOR ARCHITECTURE <identifier>
                     OF <identifier> IS
                     BEGIN <stmnt_list> END SPECIFICATION ;
<stmnt_list>     ::= <statement> {<statement>}
<statement>      ::= SET CLOCK_NAME <identifier> ;
                   | SET RESET_NAME <identifier> ;
                   | SET CONTROLLER_DELAY <integer> <unit> ;
                   | SET INTERCONNECT_DELAY <integer> <unit> ;
                   | SET CYCLE_TIME <integer> <unit> ;
                   | SET INTERCONNECT_COSTS <integer> ;
                   | SET INTERCONNECT_COSTS <integer> PER BIT ;
                   | START <identifier> AT CS <integer> ;
                   | START <identifier> AFTER CS <integer> ;
                   | START <identifier> BEFORE CS <integer> ;
                   | START <identifier> WITHIN CS <integer>
                     TO <integer> ;
                   | START <identifier> BEFORE <identifier> ;
                   | START <identifier> AFTER <identifier> ;
                   | SEPARATE <identifier> AND <identifier>
                     BY <integer> CS ;
                   | SEPARATE <identifier> AND <identifier>
                     BY MINIMUM <integer> CS ;
                   | SEPARATE <identifier> AND <identifier>
                     BY MAXIMUM <integer> CS ;
                   | SEPARATE <identifier> AND <identifier>
                     BY MINIMUM <integer> MAXIMUM <integer> CS ;
                   | BIND <identifier> TO COMPONENT <identifier> ;
                   | BIND <identifier> TO INSTANCE <identifier> ;
                   | EXTEND <identifier> BY <integer> CS ;
                   | USE C_SELECT IN <identifier> ;
                   | USE D_SELECT IN <identifier> ;
                   | LIMIT <identifier> TO <integer> INSTANCES ;
<unit>           ::= NS | US | MS
<integer>        ::= <digit> {<digit>}
<identifier>     ::= <letter> {<letter> | <digit> | _ }
<digit>          ::= 0 | 1 | 2 ... 9
<letter>         ::= A | B | C ... Z
```

B.2.4.2 Format der Kontrollschrittliste

Auch die Syntax der Kontrollschrittliste ist an VHDL angelehnt. Die Notation ist *case-insensitiv*, Kommentare werden durch doppelte Minuszeichen gekennzeichnet und gelten bis an das Zeilenende.

Folgende EBNF definiert den Aufbau der Datei:

```
<schedule>    ::= SCHEDULE <identifier> IS <decl_list> <body>
<decl_list>   ::= <declaration> {<declaration>}
<declaration> ::= CLOCKNAME <identifier> ;
                | RESETNAME <identifier> ;
                | PORT <direction> <bs_list> ;
                | REGISTER <bs_list> ;
                | INSTANCE <inst_list> ;
<direction>   ::= IN | OUT | INOUT | BUFFER
<bs_list>     ::= <bitslice> {, <bitslice>}
<inst_list>   ::= <instance> {, <instance>}
<instance>    ::= <identifier> : <identifier>
<body>        ::= BEGIN SCHEDULE <stmnt_list> END SCHEDULE ;
<stmnt_list>  ::= <statement> {<statement>}
<statement>   ::= BEGIN BLOCK <identifier> <act_list> END BLOCK ;
                | WHILE <condition> LOOP <stmnt_list> END LOOP ;
                | IF <condition> THEN <stmnt_list> END IF ;
                | IF <condition> THEN <stmnt_list>
                  ELSE <stmnt_list> END IF ;
                | WAIT <stmnt_list> UNTIL <condition> ;
                | WAIT ON <condition> ;
                | WAIT FOR <integer> CS ;
<act_list>    ::= <activation> {<activation>}
<activation>  ::= <integer> <instance> ( <operation> <arg_list> ) ;
                | <integer> <instance> ( <operation> <arg_list> )
                  <condition> ;
<arg_list>    ::= <argument> {<argument>}
<argument>    ::= ( <src_list> )
<src_list>    ::= <source> {& <source>}
<source>      ::= <bitslice> | <constant>
<bitslice>    ::= <identifier> [ <integer> : <integer> ]
<constant>    ::= <bitstring> [ <integer> : <integer> ]
<condition>   ::= ( <bool_sum> )
<bool_sum>    ::= <bool_prod> {OR <bool_prod>}
<bool_prod>   ::= <literal>
                | ( <literal> {AND <literal>} )
<literal>     ::= <bitslice> | NOT <bitslice>
<bitstring>   ::= " (C | N) (0 | 1) {0 | 1} "
<operation>   ::= + | * | - | / | > | < | = | <= | >= | <identifier>
<integer>     ::= <digit> {<digit>}
<identifier>  ::= <letter> {<letter> | <digit> | _ }
<digit>       ::= 0 | 1 | 2 ... 9
<letter>      ::= A | B | C ... Z
```

B.3 Liste der Fehlermeldungen

Das OSCAR-System bricht den Syntheselauf ab, sobald Fehler auftreten. Der Grund des Abbruchs wird angegeben in Form einer Fehlernummer und einer Fehlermeldung. Eine Liste sämtlicher möglicher Fehler ist auf den folgenden Seiten in tabellarischer Form angegeben.

Die Fehlermeldungen[1] sind weitgehend selbsterklärend, daher kann auf eine detaillierte Beschreibung der jeweiligen Ursache verzichtet werden.

Zusätzlich zu den in den Tabellen verzeichneten Fehlern kennt das System (fast ebensoviele) *interne* Fehlerzustände. Diese dienen in der Entwicklung dem Aufdecken von Programmfehlern. Aufgrund der zahlreichen erfolgreichen Testläufe des Systems besteht aber die berechtigte Hoffnung, daß diese Fehler nicht auftreten!

[1] Für `%d` und `%s` ist jeweils eine Nummer bzw. ein Bezeichner einzusetzen. Diese Notation rührt aus der Implementierung des Systems in der Programmiersprache **C** heraus.

Fehlermeldungen der Ablaufsteuerung	
1001	Bad arguments!
1002	Illegal option '%s' selected!
1003	Both quiet and verbose options selected!
1004	Unable to open file '%s' for input!
1005	Unable to open file '%s' for output!
1006	Illegal number of additional control steps (%s %d)!
1007	Illegal upper limit to the design costs (%s %d)!
1008	Option %s requires a non negative integer parameter!
1009	Too many stop options selected!
1010	Interconnect minimization impossible within the simplified binding model!
1011	Interconnect minimization requires all operations represented in the IP model!
1012	RW schedule range optimization unnecessary when these operations will be eliminated!
1013	pass 2 of relaxed LP model unnecessary!
1014	System latency (%dns) is greater than the specified cycle time (%dns)!

Fehlermeldungen des VHDL-Parsers	
1201	Parse error in line %d: %s!
1205	Unknown identifier '%s' in expression!
1206	Unknown function '%s' with %d arguments!
1207	Parse error at end of file: entity '%s' not found!
1208	Parse error at end of file: architecture '%s' not found!
1209	Bad slice [%d:%d] for variable/signal '%s'!
1210	Bad slice [%d:%d] for variable/signal '%s'!
1213	Unknown target identifier '%s'!
1214	Type mismatch with identifier '%s'!
1215	Bad bit width in assignment to variable/signal '%s'!

Fehlermeldungen des Gleichungsgenerators	
1301	Unable to write to constraint file!
1306	Out of memory!
1309	No function unit in design necessary! Sorry, the design is too simple!

Tabelle B.2: OSCAR-Fehlermeldungen (Teil a)

Fehlermeldungen der IP-Verarbeitungseinheit	
1401	Out of memory!
1410	No component for function '%s' available!
1411	No component for function '%s' available!
1412	Unable to write to dump file!
1413	Instance '%s' unknown in binding specification for '%s'!
1414	Component '%s' unknown in binding specification for '%s'!
1415	Bad binding specification for '%s': Instance/Component '%s' cannot perform operation '%s'!
1416	Bad binding specification for '%s': Instance/Component '%s' cannot perform operation '%s'!
1417	Bad timing specification: Unknown operation '%s'!
1418	Bad timing specification: Unknown operation '%s'!
1419	Unknown operation '%s' in synthesis specifications!
1420	Unknown operation '%s' in synthesis specifications!
1421	Bad precedence specification: Unknown operation '%s'!
1422	Bad precedence specification: Unknown operation '%s'!
1423	Bad timing specification: Operation '%s' and operation '%s' are not in the same version!
1424	Bad timing specification: Operation '%s' and operation '%s' cannot be separated by %d steps (maximum is %d steps)!
1425	Bad timing specification: Operation '%s' and operation '%s' cannot be separated by %d steps (minimum is %d steps)!
1427	User scheduling for operation '%s' is out of ASAP-ALAP-range %d to %d!
1428	User scheduling for operation '%s' is out of ASAP-ALAP-range %d to %d!
1433	Bad precedence specification: Operation '%s' and operation '%s' are not in the same version!
1434	Bad precedence specification: Operation '%s' before operation '%s' results in a cycle!
1436	Timing complications, no solution!
1438	Binding specifications to instances not allowed within simplified binding model!
1451	No memory unit for '%s' (%d bits) available!
1452	Illegal binding specification for '%s': RW operations cannot be bound to instances!
1453	Illegal binding specification for '%s': RW operations cannot be bound to components!

Tabelle B.3: OSCAR-Fehlermeldungen (Teil b)

Fehlermeldungen des MILP-Solver-Handlers	
1501	Parse error in line %d: %s!
1508	Removing '%s' failed because: %s!
1509	Removing '%s' failed because: %s!
1510	Calling '%s' failed because: %s (errno %d)!
1511	Calling '%s' failed (rc = 0x%x)!
1512	IP-solver returned: %s!
1513	IP-solver returned: %s!
1514	File '%s' is unreadable because: %s!
1524	Out of memory!
1526	Bad type for version %d of block %d!

Fehlermeldungen der Kontroll- und Datenflußverwaltung	
1601	Out of memory!
1602	Memory unit '%s' already exists!
1605	No identity function ':=' in attribute library!
1612	No simple function '%s' with %d arguments available!
1614	Illegal assignment target '%s'!
1615	Illegal source '%s'!
1616	No read function 'READ' in attribute library!
1617	No write function 'WRITE' in attribute library!
1622	Bad bit width in assignment to variable/signal '%s'!
1623	Argument slicing failed, because: %s!

Fehlermeldungen des Spezifikationsparsers	
1701	Parse error in line %d: %s!
1702	Out of memory!
1703	Parse error at end of file: architecture '%s' of entity '%s' not found!

Fehlermeldungen des Kontrollschrittlisten-Generators	
1801	Unable to write to step list file!
1803	System clock type mismatch: must be INPORT %s[0:0]!
1804	System reset type mismatch: must be INPORT %s[0:0]!
1805	Argument printing failed because: %s!

Tabelle B.4: OSCAR-Fehlermeldungen (Teil c)

Fehlermeldungen der Bit-Slice-Verwaltung	
1901	Out of memory!

Fehlermeldungen der Verwaltung Algebraischer Ausdrücke	
2101	out of memory!
2102	unspecified bit width!
2103	unknown expression type!
2104	Function %s with %d arguments not defined!

Fehlermeldungen des Function-Library-Parsers	
2202	parse error in line %d!
2203	Error in line %d: %d extensions instead of %d expected!
2204	Error in line %d: function '%s' with %d arguments already defined (not appended)!
2205	Unknown expression!
2206	Illegal number of expression arguments!
2207	Commutativity not defined for less than 2 arguments!
2208	Pairs of commutative arguments must be specified (%d arguments specified)!
2209	Specified argument number too large!

Tabelle B.5: OSCAR-Fehlermeldungen (Teil d)

Anhang C

Implementierung

Inhalt dieses Kapitels ist die Beschreibung der Implementierung des OSCAR-Systems. Insbesondere die Wartung und Weiterentwicklung des Systems soll mit den hier vorgestellten Informationen ermöglicht werden.

Zunächst soll die Programm-Hierarchie dokumentiert und eine Übersicht über die Aufgaben der einzelnen Programm-Module gegeben werden. Der zweite Abschnitt beschreibt die für das IP-Modell verwendeten Datenstrukturen. Im dritten Abschnitt werden einige Algorithmen aus der Implementierung des Systems vorgestellt.

C.1 Die Programm-Module

Die Implementierung des OSCAR-Systems ist in eine Menge von *Modulen* unterteilt. Diese sind als Quelldateien jeweils in eigenen Verzeichnissen gespeichert, und können auch selbst wieder in Sub-Module unterteilt sein.

Bevor die Funktion der einzelnen Programm-Module vorgestellt wird, soll eine Übersicht über die bestehenden Abhängigkeiten gegeben werden. Abbildung C.1 stellt die zugrundeliegende Hierarchie und die Beziehung der Module zu den im OSCAR-System verwendeten Ein- und Ausgabe-Dateien graphisch dar.

Die aufgeführten Abhängigkeiten ergeben sich daraus, daß ein Modul nur Funktionen und Prozeduren aus hierarchisch tiefer liegenden Modulen verwenden kann. Als einziges kann daher das Steuerungsmodul `OscarMod` auf sämtliche Funktionen des OSCAR-Systems zugreifen.

Im folgenden wird kurz auf die Aufgabe und Funktion der Module eingegangen. Für detaillierte Informationen bezüglich der jeweils exportierten Funktionen und Prozeduren ist das Studium der jeweiligen *Header*-Datei (`<ModuleName>.h`) unverzichtbar.

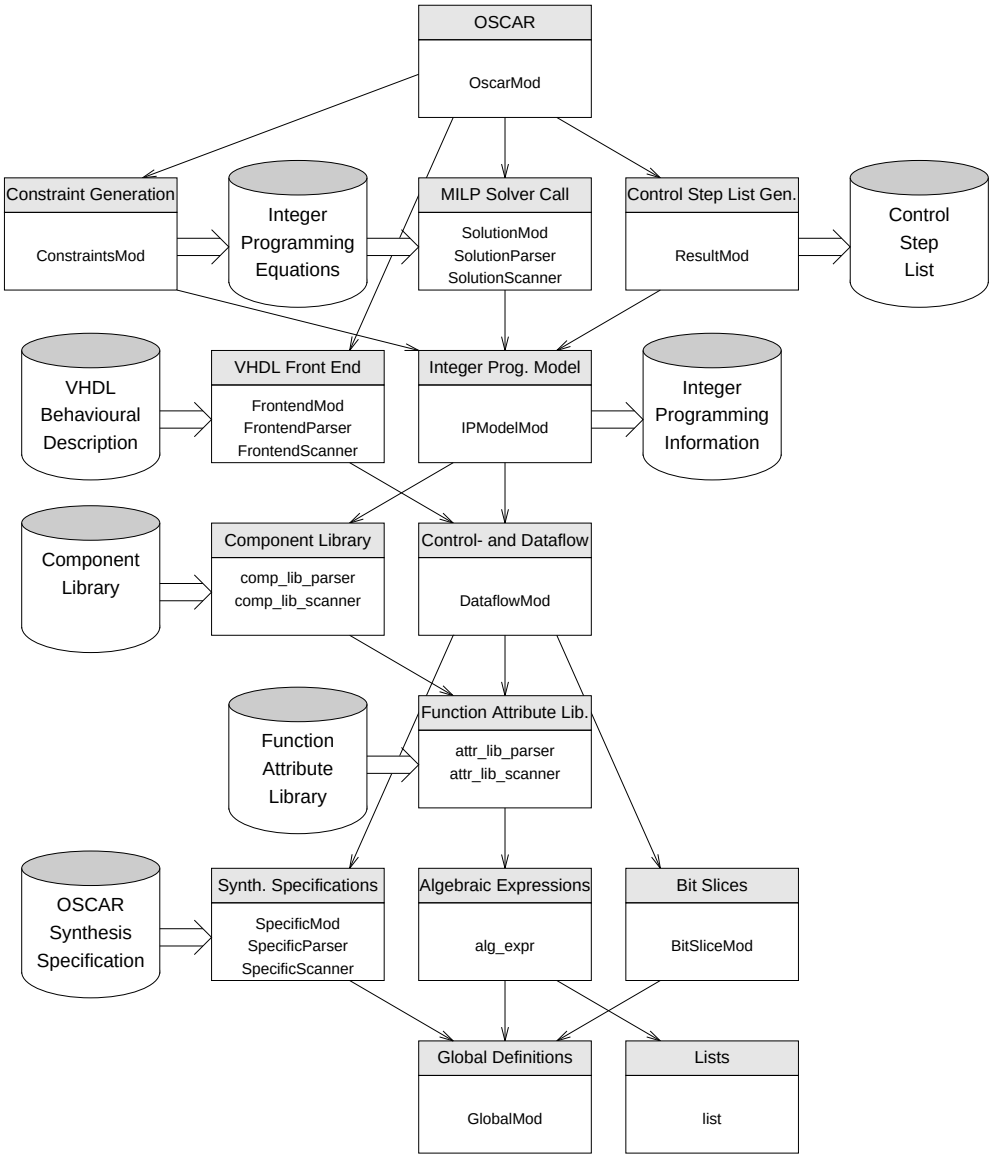

Abbildung C.1: Die Modulhierarchie des OSCAR-Systems

C.1.1 Die Basis-Module

Das Modul `GlobalMod` enthält allgemeine Definitionen und Funktionen, welche im OSCAR-System global gelten und daher von allen anderen Modulen verwendet werden können. Insbesondere die für die Behandlung von Fehlern erforderlichen Konstanten sind hier definiert.

Die Behandlung sämtlicher Operationen auf *Bit-Slices* ist Aufgabe des Moduls `BitSliceMod`. Zentrale Funktionen sind die Verwaltung, Konkatenation (`&`-Operation) und Aufsplittung von Slices.

C.1.2 Die Bibliotheken

Die Module `List`, `AlgExpr`, `AttribLib` und `CompLib` dienen zur Verwaltung der im OSCAR-System eingesetzten Bibliotheken [LaMa93].

Das Modul `List` stellt Erzeugungs-, Verwaltungs- und Zugriffsfunktionen für allgemeine Listen zur Verfügung.

Die Behandlung und Speicherung algebraischer Ausdrücke ist die Aufgabe des Moduls `AlgExpr`.

Das Datenmodul `AttribLib` stellt die Bibliothek der im System bekannten Funktionen zur Verfügung. Zentrale Funktionen sind das Einlesen der Datei `Functions.lib` und die Bereitstellung von Zugriffsfunktionen auf die Attribute der in der Bibliothek enthaltenen Funktionen.

Die Bausteinbibliothek wird durch das Datenmodul `CompLib` verwaltet. Neben dem Einlesen der Datei `Component.lib` und der Bereitstellung von entsprechenden Zugriffsfunktionen liegt die Aufgabe dieses Moduls darin, geeignete Komponenten zur Ausführung der Operationen eines Entwurfs zu finden.

C.1.3 Das Frontend

Das Einlesen und die Vorverarbeitung der Entwurfsspezifikation ist Aufgabe des Synthese-Frontends. Im OSCAR-System wird der zu synthetisierende Entwurf in zwei getrennten Dateien spezifiziert.

Das Datenmodul `SpecificMod` übernimmt das Einlesen und die Verwaltung der Syntheseparameter (Datei `<project>.spec`). Das Format dieser Spezifikationsdatei ist in Abschnitt B.2.4.1 definiert.

Die eigentliche Verhaltensbeschreibung des Entwurfs (spezifiziert durch den VHDL-Quelltext `<project>.vhdl`) wird durch das Modul `FrontendMod` eingelesen und durch das Datenmodul `DataflowMod` in einen kombinierten Kontroll- und Datenflußgraphen umgesetzt. Beide Module sind als Prototypen implementiert und werden in Zukunft (durch die Implementierungen weiterer Diplomarbeiten) ersetzt.

C.1.4 Das IP-Modell

Das Modul `IPModelMod` beinhaltet die zentrale Verwaltung der zur Repräsentation des IP-Modells notwendigen Datenstrukturen (siehe Abschnitt C.2). Insbesondere die in Kapitel 3 beschriebenen Schritte zum Aufbau der Operationen-, Komponenten- und Instanzenliste, sowie die Optimierungsverfahren aus Kapitel 4 sind in diesem Modul implementiert.

Das Funktionsmodul `ConstraintsMod` hat die Aufgabe, das Gleichungssystem aufzustellen. Auf Basis der in Anhang A zusammengefaßten Vorschriften erzeugt dieses Modul die Gleichungsdatei `<project>.eqn`, welche als Eingabe für den MILP-Solver `lp_solve` dient.

Aufgabe des Moduls `SolutionMod` ist der Aufruf des MILP-Solvers `lp_solve` [Be93], sowie das Einlesen und die Überprüfung der berechneten Lösung.

Für die Erzeugung der Kontrollschrittliste `<project>.csl` ist das Funktionsmodul `ResultMod` zuständig. Das Format der erzeugten Datei ist in Abschnitt B.2.4.2 spezifiziert.

`OscarMod` schließlich bildet das Hauptmodul des OSCAR-Systems. Hier ist das Hauptprogramm `main()` zu finden. Die Aufgaben dieses Moduls sind die Auswertung der Optionen und Parameter, die beim Systemstart angegeben werden, sowie die Ablaufsteuerung der Synthese.

C.2 Verwendete Datenstrukturen

An dieser Stelle sollen die wesentlichen Datenstrukturen dokumentiert werden, die zum Aufbau und zur Verwaltung des IP-Modells verwendet werden. Für Implementierungsdetails muß auf die entsprechenden Quelldateien (insbesondere `IPModelMod.h`) verwiesen werden. Inhalt dieses Abschnittes ist die *Struktur* der Datenhaltung.

Folgende Datenstrukturen dienen im OSCAR-System zur Repräsentation des IP-Modells:

- *Synthese-Parameter-Struktur:* eine Struktur, welche globale Parameter für den Entwurf zusammenfaßt, z. B. die Zykluszeit, die Anzahl der erlaubten Kontrollschritte und die anzuwendenden Optimierungsverfahren;

- Menge der *Operationen:* eine kombinierte Datenstruktur, welche als Basis eine einfach verkettete Liste der Operationen enthält; zusätzlich besteht eine direkte Zugriffsmöglichkeit auf jede Operation über eine vollständige Hash-Tabelle, wobei die Operations-ID als Schlüssel dient; weiterhin beinhaltet jeder Operationsknoten eine Liste der Präzedenzkanten zu den Operationen, von denen er datenabhängig ist; im Zusammenhang mit diesen Präzedenzkanten bildet die Operationenliste also den Abhängigkeitsgraphen; Abbildung C.2 zeigt den strukturellen Aufbau dieser Datenstrukturen;

- Liste der *Kontrollblöcke* und *Versionen:* eine Liste von Kontrollblockknoten, an denen jeweils eine Liste der alternativen Versionen hängt; jeder Versionsknoten beinhaltet Zeiger auf die jeweils erste und letzte Operation, die dieser Version zugeordnet sind; Abbildung C.3 zeigt diese Strukturen;

- Liste der *Komponenten:* eine einfach verkettete Liste der zur Verfügung stehenden Bausteintypen; zusätzlich zum Listendurchlauf ist jede Komponente auch direkt über eine vollständige Hash-Tabelle erreichbar;

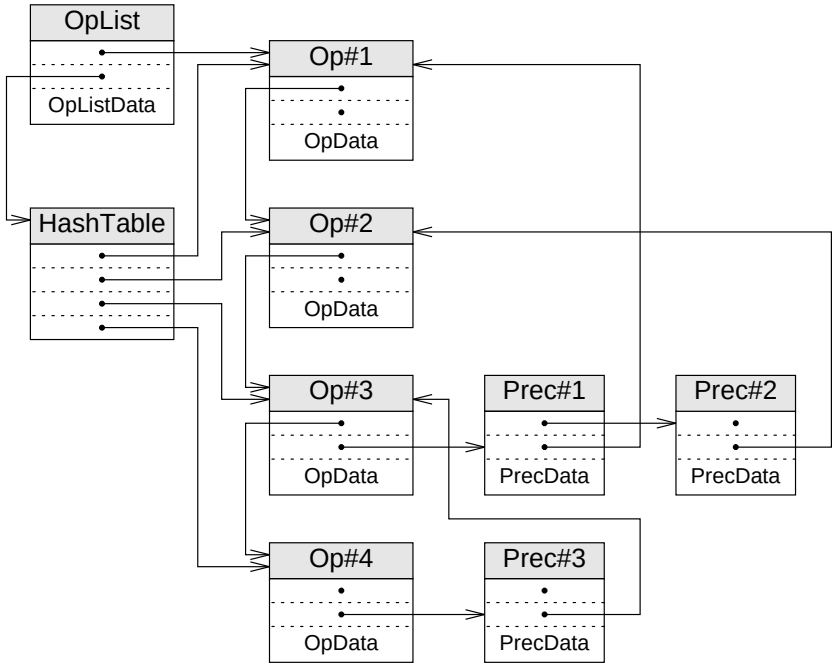

Abbildung C.2: Die Struktur der Operationenliste

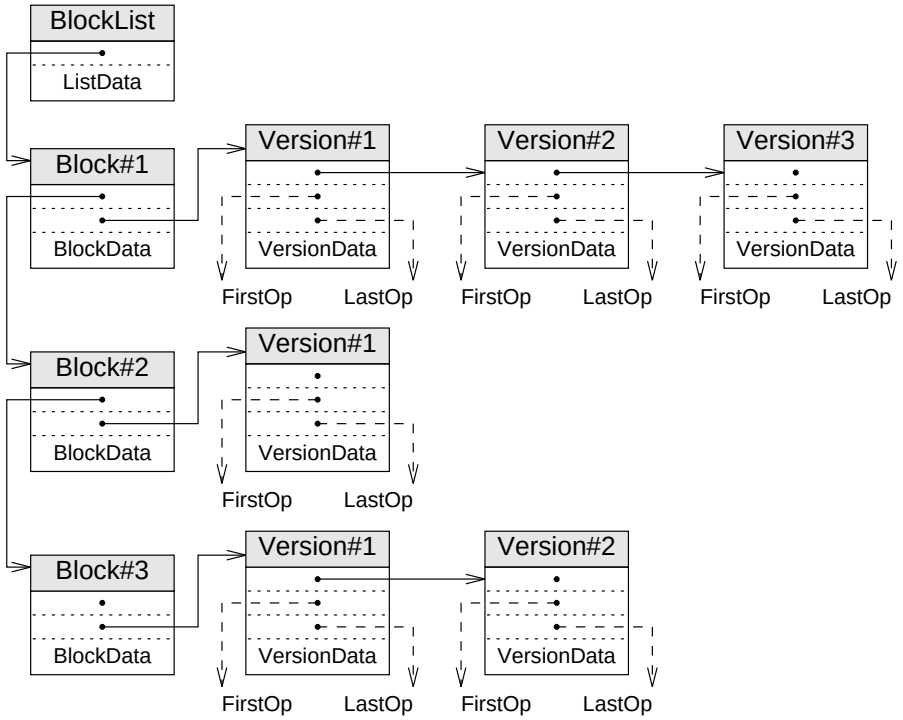

Abbildung C.3: Die Struktur der Kontrollblöcke und Versionen

- Liste der *Instanzen:* eine einfach verkettete Liste der Bausteininstanzen, kombiniert mit einer vollständigen Hash-Tabelle, so daß auf jede Instanz

in konstanter Zeit zugegriffen werden kann; jeder Instanzknoten enthält einen Verweis auf den repräsentierten Bausteintyp; Abbildung C.4 stellt die Datenstrukturen der Ressourcen graphisch dar;

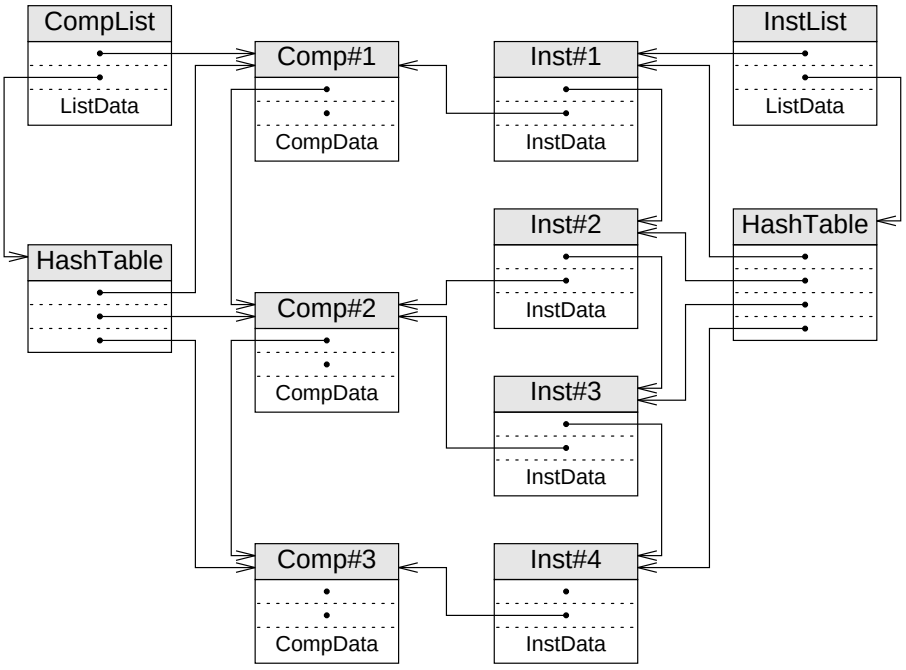

Abbildung C.4: Die Struktur der Komponenten und Instanzen

- Liste der *Zeitvorgaben:* eine einfach verkettete Liste, deren Knoten die Zeitvorgaben des Entwurfs enthalten; die beiden von der Zeitvorgabe betroffenen Operationen sind jeweils als Zeiger vermerkt;
- Liste der *Speicherbausteine:* eine einfach verkettete Liste sämtlicher Register und Ports des Entwurfs;
- *Verbindungskostentabelle:* ein zweidimensionales, quadratisches Array zur Repräsentation der Verbindungen zwischen den Instanzen des Entwurfs; die am Ende dieses Abschnittes angegebene Informationsdatei enthält eine solche Tabelle;
- *Binding-Tabelle:* ein zweidimensionales Array mit $|K|$ Spalten und $|I|$ Zeilen; dieses Array repräsentiert die Zuordnung von Operationen zu Kontrollschritten und Instanzen; auch diese Tabelle wird in der erzeugten Informationsdatei angegeben (s. u.);
- Liste der *ganzzahligen Variablen:* (nur bei Anwendung des LP-Modells) eine einfache Liste der in der ersten Lösung des MILP-Solvers enthaltenen, ganzzahligen x-Variablen (siehe hierzu Abschnitt 4.2);
- Liste der *maximalen Ketten:* eine spezielle, mehrfach verkettete Liste, welche sämtliche maximalen Ketten von Operationen enthält, für die paar-

weise Chaining möglich ist; die genaue Struktur und der Aufbau dieser Liste sind in Abschnitt C.3.3 beschrieben;

C.2.1 Die IP-Informationsdatei

Das OSCAR-System gibt nach der Synthese eines Entwurfs eine detaillierte Datei aus, welche über die verwendeten Datenstrukturen informiert[1]. Die IP-Informationsdatei `sample.ipm` für das in Kapitel 3 vorgestellte Beispiel ist im folgenden gekürzt angegeben.

```
------------------------------------------------------------------------
-- IP-Model-Information generated by OSCAR V1.4.3
-- Entity:       SAMPLE
-- Architecture: BEHAVIOUR
-- File:         sample.ipm
-- Time:         Thu Oct 20 09:20:50 1994
------------------------------------------------------------------------

1. Global Parameters:
        Cycle Time              =  1000ns (=     75ns +    925ns)
        System Latency          =    75ns (=     50ns +     25ns)
        Min. Control Steps      =     3.
        Add. Control Steps      =     0.
        Num. Control Steps      =     4.
        Limit To Design Costs   =     0.
        Optimize Connections    = FALSE.
        Allow Chaining          = FALSE.
        Optimize Datapath       = TRUE.
        Optimize Timings        = TRUE.
        Simple Allocation       = FALSE.
        Relaxed LP Model        = FALSE.
        Minimize RW Ops.        = FALSE.
        Minimize RW Ranges      = FALSE.
        Minimize Reg. Widths    = FALSE.
        Use Register Folding    = TRUE.

2. Control Block List:
  1. ID#  1, Label 'BLOCK_1',   contains 1 alternative versions
     Schedule range CS   1 to   4 (  3 +   1 CS), used upto CS   4
       1. Version#  1: selected, Op#  1 up to Op#  9, min.  3 CS

3. Operations List:
  1. ID#  1, Label 'OPERATION_1', Function 'READ', Version#1 of Block#1
     Type: Read operation
     Execution Time = (10ns - 10ns) = (1 - 1 cs)
     ASAP-ALAP-Range = (step 1 to step 1), ChainSum = (35ns, 35ns)
     Special path to Op#0, Length = 0, Color = 0
     Scheduling = 1, Binding = Inst#4 (User: Inst#4)
     Dependencies = { }
     Arguments: (IN1[15:0])
     Result:    [15:0], buffered in 'REG_1'[15:0]

[ ... ]

...
```

[1] Die Informationsdatei hat sich im Laufe der Programmierung aus erzeugten *Debug*-Ausgaben entwickelt. Die detailliert aufgeführten Daten können aber bei der Synthese mit dem OSCAR-System wertvolle Hinweise für den Benutzer enthalten. Daher wird diese Datei zusätzlich zur Kontrollschrittliste erzeugt.

```
...

  4. ID#  4, Label 'OPERATION_4', Function '*', Version#1 of Block#1
     Type: Simple operation, no alternative
     Execution Time = (800ns - 800ns) = (1 - 1 cs)
     ASAP-ALAP-Range = (step 2 to step 3), ChainSum = (825ns, 825ns)
     Special path to Op#0, Length = 0, Color = 0
     Scheduling = 3, Binding = Inst#1 (no user binding)
     Dependencies = {
        DATA: Op#2 via 'B'(16), irredundant, Chaining is OFF, unused
        DATA: Op#3 via 'C'(16), irredundant, Chaining is OFF, unused }
     Arguments: (REG_2[15:0]) (REG_3[15:0])
     Result:    [15:0], buffered in 'REG_2'[15:0]
  5. ID#  5, Label 'OPERATION_5', Function '+', Version#1 of Block#1
     Type: Simple operation, no alternative
     Execution Time = (600ns - 600ns) = (1 - 1 cs)
     ASAP-ALAP-Range = (step 3 to step 4), ChainSum = (625ns, 625ns)
     Special path to Op#0, Length = 0, Color = 0
     Scheduling = 4, Binding = Inst#2 (no user binding)
     Dependencies = {
        DATA: Op#1 via 'A'(16), irredundant, Chaining is OFF, unused
        DATA: Op#4 via 'TEMP_1'(16), irredundant, Chain. OFF, unused }
     Arguments: (REG_1[15:0]) (REG_2[15:0])
     Result:    [15:0], unbuffered

[ ... ]

  9. ID# 9, Label 'OPERATION_9', Function 'WRITE', Version#1 of Block#1
     Type: Write operation
     Execution Time = (10ns - 10ns) = (1 - 1 cs)
     ASAP-ALAP-Range = (step 3 to step 4), ChainSum = (35ns, 35ns)
     Special path to Op#0, Length = 0, Color = 0
     Scheduling = 4, Binding = Inst#8 (User: Inst#8)
     Dependencies = {
        DATA: Op#7 via 'Y'(16), irredundant, Chaining is OFF, unused }
     Arguments: (REG_3[15:0])
     Result:    [15:0], unbuffered

4. Timings:
  1. ID#  1,   0 cs CONST. timing between Op#8 and Op#9, irredundant,

5. Component List:
  1. ID#  1 'INPORT'    Costs = $ 0, 3/3 instances allocated
     Max. latency =   10ns = 1 CS, max. execution time =   10ns = 1 CS
  2. ID#  2 'MULTIPLIER'Costs = $ 40000, 1/1 instances allocated
     Max. latency =  800ns = 1 CS, max. execution time =  800ns = 1 CS
  3. ID#  3 'ADDER'     Costs = $ 20000, 1/2 instances allocated
     Max. latency =  600ns = 1 CS, max. execution time =  600ns = 1 CS
  4. ID#  4 'OUTPORT'   Costs = $ 0, 2/2 instances allocated
     Max. latency =   10ns = 1 CS, max. execution time =   10ns = 1 CS

6. Instance List:
  1. ID#  1 'MULTIPLIER_1',     Type 'MULTIPLIER'(Comp#2), allocated
  2. ID#  2 'ADDER_1',  Type 'ADDER'(Comp#3),   allocated
  3. ID#  3 'ADDER_2',  Type 'ADDER'(Comp#3),   not allocated
  4. ID#  4 'IN1',      Type 'INPORT'(Comp#1),  allocated
  5. ID#  5 'IN2',      Type 'INPORT'(Comp#1),  allocated
  6. ID#  6 'IN3',      Type 'INPORT'(Comp#1),  allocated
  7. ID#  7 'OUT1',     Type 'OUTPORT'(Comp#4), allocated
  8. ID#  8 'OUT2',     Type 'OUTPORT'(Comp#4), allocated

...
```

```
...

7. Interconnection Costs Table:
     From\To   | Inst# 1 | Inst# 2 | Inst# 3 | | Inst# 7 | Inst# 8 |
---------------|---------|---------|---------|-|---------|---------|
  1. Inst#  1  |    -    | $  1600 |($  1600)| |    -    |    -    |
  2. Inst#  2  |    -    |    -    |    -    | | $  1600 | $  1600 |
  3. Inst#  3  |    -    |    -    |    -    |.|($  1600)|($  1600)|
  4. Inst#  4  | $  1600 | $  1600 |($  1600)|.|    -    |    -    |
  5. Inst#  5  | $  1600 |    -    |    -    |.|    -    |    -    |
  6. Inst#  6  | $  1600 | $  1600 |($  1600)| |    -    |    -    |
  7. Inst#  7  |    -    |    -    |    -    | |    -    |    -    |
  8. Inst#  8  |    -    |    -    |    -    | |    -    |    -    |

8. Binding Table:
     CS\Inst.  | Inst# 1 | Inst# 2 | Inst# 4 | | Inst# 7 | Inst# 8 |
---------------|---------|---------|---------|-|---------|---------|
  1. CS    1   |    -    |    -    |  Op# 1  |.|    -    |    -    |
  2. CS    2   |  Op# 6  |    -    |    -    |.|    -    |    -    |
  3. CS    3   |  Op# 4  |  Op# 7  |    -    |.|    -    |    -    |
  4. CS    4   |    -    |  Op# 5  |    -    | |  Op# 8  |  Op# 9  |

9. Register List:
  1. ID#  1 INPORT    'IN1'[15:0], read range [15:0]
     Usage not limited by control step ranges
  2. ID#  2 INPORT    'IN2'[15:0], read range [15:0]
     Usage not limited by control step ranges
  3. ID#  3 INPORT    'IN3'[15:0], read range [15:0]
     Usage not limited by control step ranges
  4. ID#  4 OUTPORT   'OUT1'[15:0], never read!
     Usage not limited by control step ranges
  5. ID#  5 OUTPORT   'OUT2'[15:0], never read!
     Usage not limited by control step ranges
  6. ID#  6 TEMPORARY 'REG_1'[15:0], read range [15:0]
     Used in steps   2 to   4 by result of Op#  1
  7. ID#  7 TEMPORARY 'REG_2'[15:0], read range [15:0]
     Used in steps   2 to   3 by result of Op#  2
     Used in steps   4 to   4 by result of Op#  4
  8. ID#  8 TEMPORARY 'REG_3'[15:0], read range [15:0]
     Used in steps   2 to   3 by result of Op#  3
     Used in steps   4 to   4 by result of Op#  7
  9. ID#  9 TEMPORARY 'REG_4'[15:0], read range [15:0]
     Used in steps   3 to   3 by result of Op#  6

10.Integer Value List:
   (not existing)

-------------------------------------------------------------------------
```

C.3 Algorithmen

Nachdem die grundlegenden Datenstrukturen des OSCAR-Systems vorgestellt sind, sollen im folgenden einige zentrale Algorithmen aus der Implementierung des OSCAR-Systems dokumentiert werden.

Zur Notation wird hier die Form einer Pseudo-Programmiersprache gewählt, welche das Verständnis der ausgewählten Algorithmen erleichtert. Für Implementierungsdetails muß wiederum auf die Quell-Codes verwiesen werden, welche in der Programmiersprache **C** vorliegen.

C.3.1 Berechnung des Instanzenangebotes

Das OSCAR-System berechnet vor der Synthese eines Entwurfs das benötigte Instanzenangebot. Die Anzahl der Bausteininstanzen darf den Lösungsraum nicht weiter einschränken, als er durch den Benutzer vorgegeben ist, sollte aber möglichst nah an der wirklich benötigten Zahl liegen, da sich die Anzahl der Instanzen direkt auf die Zahl der Variablen im IP-Modell auswirkt.

Folgender Algorithmus bestimmt im OSCAR-System ein ausreichend großes Angebot an Bausteininstanzen:

```
INPUT:  I, J, M                    Kontrollschritte, Operationen-, Komponentenliste

OUTPUT: Number[|M|]                Anzahl der Instanzen jedes Bausteintyps

FOR ALL m ∈ M DO                                        über alle Bausteintypen
    Number[m] := 0;
    FOR ALL i ∈ I DO                                    über alle Kontrollschritte
        n = 0;
        FOR j1 = jmax DOWNTO jmin DO                    über alle Operationen
            IF i ∈ R(j)
            AND is_not_marked(j)
            AND is_executable_on(j,m) THEN
                n = n + 1;                              parallel ausführbare
                mark_ops_above(j);                      Operationen zählen
            FI;
        ROF;
        Number[m] := max(Number[m], n);                 Maximum bilden
        remove_marks_from(J);
    ROF;
ROF;
```

Die Funktion `mark_ops_above(`j`)` markiert von Operation j ausgehend die Operationen, von denen j direkt oder indirekt abhängig ist, da diese offensichtlich nicht gleichzeitig zu j ausgeführt werden können. (Im Fall von Chaining müssen hier besondere Vorkehrungen getroffen werden, auf die hier nicht weiter eingegangen werden soll.)

Die Markierungsfunktion benötigt im *worst case* $O(|J|^2)$ Schritte, daher ergibt sich die Gesamtlaufzeit dieses Algorithmus' zu $O(|I| \cdot |J^3| \cdot |M|)$.

C.3.2 Nachträgliche Instanzenbindung

Die Bindung von Operationen an Instanzen muß im Fall des vereinfachten IP-Modells (Abschnitt 2.11) nach der Lösung des Gleichungssystems gesondert berechnet werden. Hierzu wird der folgende, modifizierte Left-Edge-Algorithmus verwendet:

```
INPUT:  I, J, K                    Kontrollschritte, Operationen-, Instanzenliste

OUTPUT: Table[|K|,|I|]             Tabelle mit |K| Spalten und |i| Zeilen

new(Table[|K|,|I|]);               neue Bindungstabelle erzeugen
FOR ALL i ∈ I DO
    FOR ALL j ∈ J DO               Tabelle zeilenweise füllen
        IF start_step(j) = i THEN
            m := component_used_for(j);
            k := first_instance_of_type(m);
            WHILE is_allocated(Table[k,i]) DO        erste freie
                k := next_instance_of_type(m);       Spalte suchen
            OD;
            FOR i' := i TO i + ℓ(j,k) DO
                Table[k,i'] := j;      Operation eintragen
            ROF;
        FI;
    ROF;
ROF;
```

Die Laufzeit dieses Algorithmus' ist offensichtlich polynomiell: $O(|I|^2 \cdot |J| \cdot |K|)$.

Die berechnete Bindungstabelle des Entwurfs wird im übrigen jeweils in der IP-Informationsdatei `<project>.ipm` mit angegeben.

C.3.3 Berechnung maximaler Operationenketten

Zur Unterstützung von allgemeinem Chaining (siehe Abschnitt 2.10) müssen die längsten Ketten von jeweils datenabhängigen Operationen betrachtet werden, für die paarweise Chaining möglich ist.

Zur Berechnung dieser Operationenketten verwendet das OSCAR-System eine spezielle Liste, deren Knoten jeweils *zwei* Nachfolgerzeiger haben. Der erste Zeiger dient zur einfachen Verkettung der Liste, der zweite zeigt auf den nächsten Knoten, mit dem Chaining möglich ist. Zusätzlich erhält jeder Knoten eine Markierung, die den Beginn einer neuen Kette anzeigt. Die Struktur dieser Liste ist in Abbildung C.5 exemplarisch dargestellt.

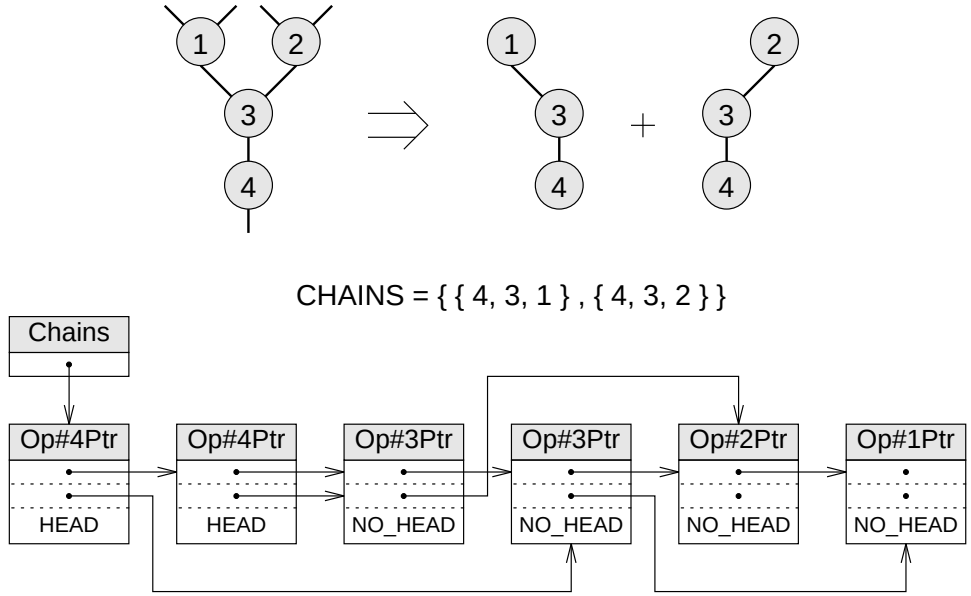

Abbildung C.5: Die Struktur der Liste maximaler Ketten

Folgender Algorithmus dient im OSCAR-System zur Erzeugung dieser Liste:

```
INPUT:  J, chain(j1,j2)                          Operationen, Chaining-Kanten

OUTPUT: CHAINS                                   Liste maximaler Operationenketten

empty(CHAINS);
FOR j1 = jmin UPTO jmax DO                       topologische Ordnung nutzen
    Inserted := FALSE;
    FOR ALL j2 ∈ J DO                            über alle Chaining-Kanten
        IF chain(j1,j2) THEN
            FOR ALL Node ∈ CHAINS DO                 Ketten um Kante
                IF operation(Node) = j2 THEN             verlängern
                    prepend(CHAINS, new_node(j1,Node,HEAD));
                    type_of(Node) := NO_HEAD;
                FI;
            ROF;
            Inserted := TRUE;
        FI;
    ROF;
    IF NOT Inserted THEN                             Kettenende erzeugen
        prepend(CHAINS, new_node(j1,NIL,NO_HEAD));
    FI;
ROF;
```

Die Anzahl der Kanten im Datenflußgraphen (ein DAG) ist bekanntlich durch $O(|J|^2)$ beschränkt, somit auch die Anzahl der Chaining-Kanten. Im *worst case*

kann jede dieser Kanten $n = |\text{CHAINS}|$ neue Knoten in der Liste erzeugen, wodurch die Anzahl der Knoten und damit auch die Laufzeit des Algorithmus' exponentiell wird. In der Praxis ist aber die Anzahl der Chaining-Kanten im Datenflußgraphen eines Entwurfs eher gering, so daß diese theoretische Laufzeitbetrachtung vernachlässigt werden kann.

Die Aufzählung aller Operationen der maximalen Ketten, so wie sie in der Chaining-Vorschrift 2.30 verlangt wird, erledigt folgender Algorithmus:

```
INPUT:  J, CHAINS                          Liste maximaler Operationenketten

OUTPUT: maximal lists of chained operations

FOR ALL Node ∈ CHAINS DO                   Liste aller Ketten durchlaufen
    IF type_of(Node) = HEAD THEN
        output("Begin Chain");
        output(operation(Node));               Kopf der Kette ausgeben
        Node2 := chain_succ(Node);
        REPEAT
            output(operation(Node2));      Elemente der Kette ausgeben
            Node2 := chain_succ(Node);
        UNTIL Node2 = NIL;
        output("End Chain");
    FI;
ROF;
```

Die Laufzeit dieser Aufzählung ist offensichtlich quadratisch zur Länge der Liste: $O(|\text{CHAINS}|^2)$.

Anhang D

Benchmarks

Zum Abschluß dieser Arbeit sollen im folgenden noch einige Ergebnisse der OSCAR-Synthese angegeben werden. Die beiden in der Fachliteratur am häufigsten zitierten Benchmarks sollen auch hier als Beispiele dienen:

- 5th-Order Elliptical-Wave-Filter (EWF) [KuWhKa85]
- Differential-Equation-Solver (DFQ) [PaKnGi86]

Für diese Entwürfe sind im folgenden die Ergebnisse verschiedener Syntheseläufe des OSCAR-Systems in tabellarischer Form angegeben.

Zur Lösung des Gleichungssystems wurde jeweils der MILP-Solver `lp_solve` [Be93] in der Version 1.0 verwendet. Die angegebenen Laufzeiten beziehen sich ausschließlich auf die Rechenzeiten des MILP-Solvers und wurden auf einer SPARCstation 20 (Taktfrequenz 60 MHz) ermittelt[1].

D.1 5th-Order Elliptical Wave Filter

Die Verhaltensbeschreibung des 5th-Order Elliptical-Wave-Filters [KuWhKa85] besteht aus einem Basisblock, der 34 arithmetische, sowie 8 Lese- und 7 Schreiboperationen enthält. Abbildung D.1 stellt den Datenflußgraphen des EWF dar.

Der Elliptical-Wave-Filter wurde bereits in Kapitel 2 als Beispiel für Optimierungen des OSCAR-Systems herangezogen (siehe Tabellen 2.1 bis 2.5). Diese Ergebnisse sollen hier nicht wiederholt werden.

Die Tabellen D.1 bis D.4 stellen weitere Syntheseergebnisse des Elliptical-Wave-Filters vor, welche unter Verwendung des vereinfachten Bindungsmodells (siehe Abschnitt 2.11) ermittelt wurden.

[1] Das Aufstellen des Gleichungssystems und die Nachverarbeitung der erhaltenen Lösung benötigen i. d. R. jeweils weniger als 1s Rechenzeit. Daher sind diese Zeiten gegenüber den Laufzeiten des MILP-Solvers vernachlässigbar.

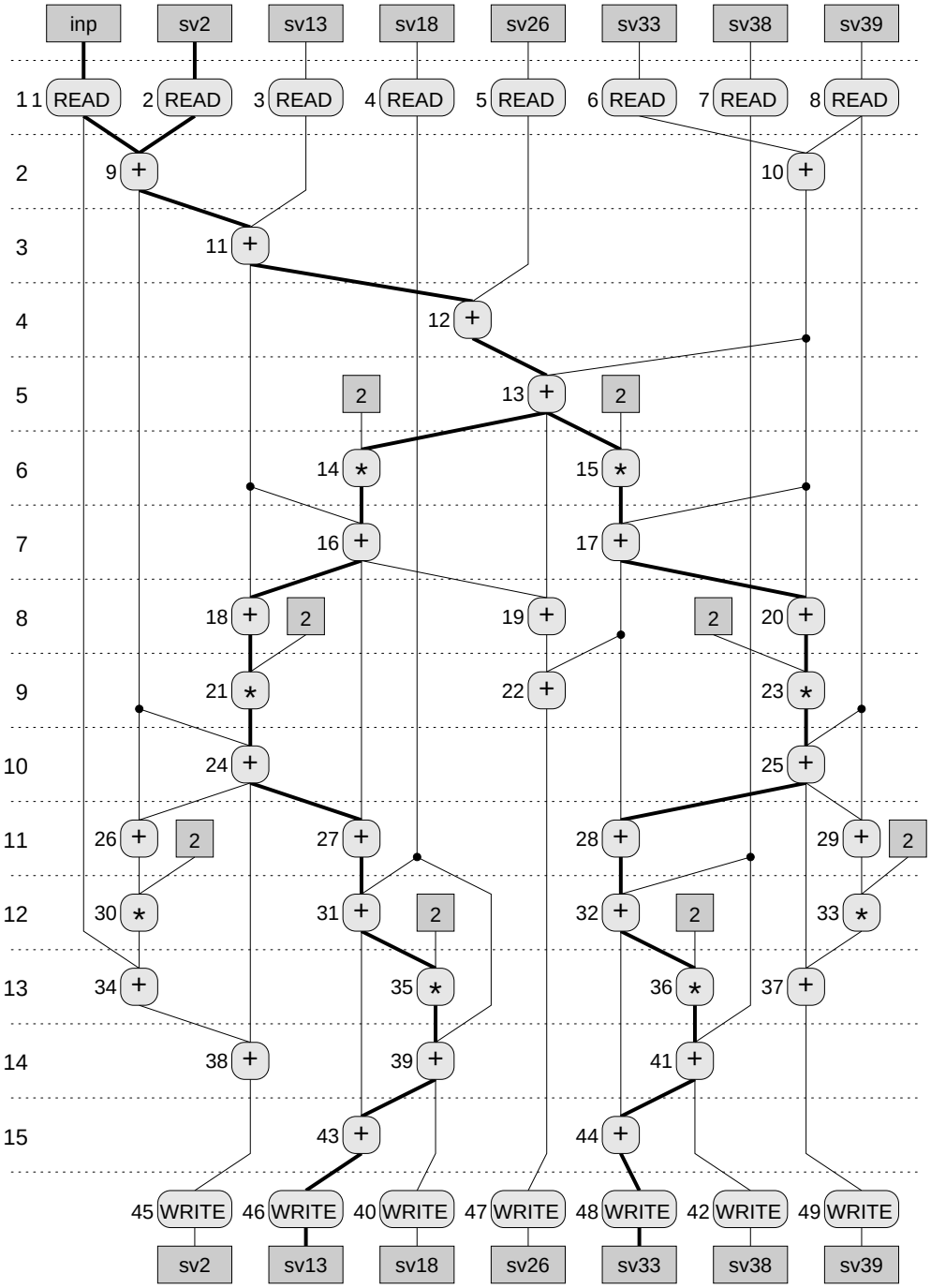

Abbildung D.1: Der Datenflußgraph des Elliptical-Wave-Filters

Die zur Verfügung stehenden arithmetischen Einheiten, ihre Kosten[2] und Ausführungszeiten sind jeweils in den Tabellen angegeben. Eine Ausführungszeit von `1:2 CS` bezeichnet einen Pipeline-Baustein, der in jedem Kontrollschritt neue Daten aufnehmen kann ($\ell(j,k) = 1$), zur Ausführung der Operation aber zwei Schritte benötigt ($C(j,k) = 2$).

Tabelle D.1 faßt die Syntheseergebnisse zusammen, die sich bei unterschiedlichem Bausteinangebot ergeben[3].

Tabelle D.2 stellt die Ergebnisse des IP- und des LP-Modells gegenüber (vgl. auch Tabelle 4.1). Es wird deutlich, daß das LP-Modell zwar erheblich kürzere Rechenzeiten erfordert, aber in einigen Fällen nur sub-optimale Lösungen liefert[4].

Tabelle D.3 zeigt die Anwendung von Chaining. Es werden zwei Fälle dargestellt: Im ersten Fall werden die Multiplikationen ($x \cdot 2$) auf einem Shifter ($\mathrm{shl}(x,1)$) ausgeführt. Bei einer Zykluszeit von 1000ns lassen sich dann eine Multiplikation (300ns) und eine vorhergehende oder nachfolgende Addition (600ns) innerhalb eines Kontrollschrittes ausführen. Der zweite Fall erlaubt eine Verkettung von jeweils zwei Additionen, welche hier mit 400ns angenommen werden[5].

Tabelle D.4 zeigt die Anwendung von alternativen Versionen (siehe Abschnitt 2.4.3). Alternativ zum Original-Datenfluß des Elliptical-Wave-Filters (Version 1) werden hier zwei weitere Versionen verwendet. Für Version 2 werden die Multiplikationen ($x \cdot 2$) durch Shift-Operationen ($\mathrm{shl}(x,1)$) und für Version 3 durch Konkatenationen (x & '0') ersetzt. Die Tabelle zeigt, daß aus den angebotenen Versionen jeweils die kostengünstigste ausgewählt wird[6].

---

[2] Das OSCAR-System verwendet als Bezeichnung für Kosteneinheiten das $-Zeichen. Auf eine Begründung dieser Wahl kann verzichtet werden.

[3] Tabelle D.1, Aufruf des OSCAR-Systems:

```
oscar -v -Od -Ob -Oo -l <limit> -a <steps> elliptic oscar_behaviour;
```

[4] Tabelle D.2, Aufruf des OSCAR-Systems:

```
oscar -v -Od -Ob -Oo [-Ol] -l <limit> -a <steps> elliptic oscar_behaviour;
```

[5] Tabelle D.3, Aufruf des OSCAR-Systems:

```
oscar -v -Od -Ob -Oo -Oc -l <limit> -a <steps> elliptic oscar_behaviour;
```

[6] Tabelle D.4, Aufruf des OSCAR-Systems:

```
oscar -v -Od -Ob -Oo -Og -l <limit> -a <steps> elliptic oscar_behaviour;
```

Kontrollschritte (CS)	15	16	17	18	19	20	21	22
Addierer (1CS, $ 20)	2	2	2	2	2	1	1	1
Multiplizierer (1CS, $ 30)	1	0	1	1	1	0	0	0
Add-Mul-ALU (1CS, $ 40)	1	1	0	0	0	1	1	1
entstandene Kosten ($)	110	80	70	70	70	60	60	60
Berechnungszeit (s)	1	1	10	16	41	96	221	501
Kontrollschritte (CS)	15	16	17	18	19	20	21	22
Addierer (1CS, $ 20)	-	-	-	3	2	2	2	2
Multiplizierer (2CS, $ 30)	-	-	-	3	2	2	2	1
Add-Mul-ALU (2CS, $ 40)	-	-	-	0	0	0	0	0
entstandene Kosten ($)	-	-	-	150	100	100	100	70
Berechnungszeit (s)	-	-	-	1	1	7	24	102
Kontrollschritte (CS)	15	16	17	18	19	20	21	22
Addierer (1CS, $ 20)	-	-	-	3	3	2	2	1
Multiplizierer (1:2CS, $ 30)	-	-	-	2	1	1	1	0
Add-Mul-ALU (1:2CS, $ 40)	-	-	-	0	0	0	0	1
entstandene Kosten ($)	-	-	-	120	90	70	70	60
Berechnungszeit (s)	-	-	-	1	2	19	23	69

Tabelle D.1: EWF, Ergebnisse bei unterschiedlichem Bausteinangebot

LP-Modell: (CS)	15	16	17	18	19	20	21	22
Addierer (1CS, $ 20)	2	3	2	2	2	2	2	2
Multiplizierer (1CS, $ 30)	1	1	1	1	1	1	1	1
Add-Mul-ALU (1CS, $ 40)	1	0	0	0	0	0	0	0
entstandene Kosten ($)	110	90	70	70	70	70	70	70
Berechnungszeit (s)	2	2	3	5	6	11	12	16
IP-Modell: (CS)	15	16	17	18	19	20	21	22
Addierer (1CS, $ 20)	2	2	2	2	2	1	1	1
Multiplizierer (1CS, $ 30)	1	0	1	1	1	0	0	0
Add-Mul-ALU (1CS, $ 40)	1	1	0	0	0	1	1	1
entstandene Kosten ($)	110	80	70	70	70	60	60	60
Berechnungszeit (s)	1	1	10	16	41	96	221	501

Tabelle D.2: EWF, Ergebnisse bei Anwendung des LP-Modells

Kontrollschritte (CS)	10	11	12	13	14
Addierer (600ns, $ 20)	-	-	4	3	3
Shifter (300ns, $ 10)	-	-	1	1	1
entstandene Kosten ($)	-	-	90	70	70
Berechnungszeit (s)	-	-	1	2	156

Kontrollschritte (CS)	10	11	12	13	14
Addierer (400ns, $ 20)	4	3	3	3	
Multiplizierer (800ns, $ 30)	2	2	1	1	
entstandene Kosten ($)	140	120	90	90	
Berechnungszeit (s)	1	3	9	2177	>5000

Tabelle D.3: EWF, Ergebnisse bei Anwendung von Chaining

Kontrollschritte (CS)	12	13	14	15	16	17	18	19
Versionsauswahl (V1)	-	-	-	-	-	-	V1	V1
Addierer (1CS, $ 20)	-	-	-	-	-	-	3	2
Multiplizierer (2CS, $ 30)	-	-	-	-	-	-	3	2
Shifter (1CS, $ 25)	-	-	-	-	-	-	0	0
Berechnungszeit (s)	-	-	-	-	-	-	1	1
entstandene Kosten ($)	-	-	-	-	-	-	150	100
Versionsauswahl (V2)	-	-	-	V2	V2	V2	V2	V2
Addierer (1CS, $ 20)	-	-	-	3	3	2	2	2
Multiplizierer (2CS, $ 30)	-	-	-	0	0	0	0	0
Shifter (1CS, $ 25)	-	-	-	2	1	1	1	1
Berechnungszeit (s)	-	-	-	1	1	2	7	8
entstandene Kosten ($)	-	-	-	110	85	65	65	65
Versionsauswahl (V3)	V3	V3	V3	V3	V3	V3	V3	V3
Addierer (1CS, $ 20)	4	3	3	3	2	2	2	2
Multiplizierer (2CS, $ 30)	0	0	0	0	0	0	0	0
Shifter (1CS, $ 25)	0	0	0	0	0	0	0	0
Berechnungszeit (s)	1	1	1	1	7	12	21	28
entstandene Kosten ($)	80	60	60	60	40	40	40	40
Versionsauswahl (V1,V2)	-	-	-	V2	V2	V2	V2	V2
Addierer (1CS, $ 20)	-	-	-	3	3	2	2	2
Multiplizierer (2CS, $ 30)	-	-	-	0	0	0	0	0
Shifter (1CS, $ 25)	-	-	-	2	1	1	1	1
Berechnungszeit (s)	-	-	-	1	1	2	18	29
entstandene Kosten ($)	-	-	-	110	85	65	65	65
Versionsauswahl (V2,V3)	V3	V3	V3	V3	V3	V3	V3	V3
Addierer (1CS, $ 20)	4	3	3	3	2	2	2	2
Multiplizierer (2CS, $ 30)	0	0	0	0	0	0	0	0
Shifter (1CS, $ 25)	0	0	0	0	0	0	0	0
Berechnungszeit (s)	1	1	1	2	3	32	28	40
entstandene Kosten ($)	80	60	60	60	40	40	40	40
Versionsauswahl (V1,V3)	V3	V3	V3	V3	V3	V3	V3	V3
Addierer (1CS, $ 20)	4	3	3	3	2	2	2	2
Multiplizierer (2CS, $ 30)	0	0	0	0	0	0	0	0
Shifter (1CS, $ 25)	0	0	0	0	0	0	0	0
Berechnungszeit (s)	1	1	1	1	3	16	31	39
entstandene Kosten ($)	80	60	60	60	40	40	40	40
Versionsauswahl (V1,V2,V3)	V3	V3	V3	V3	V3	V3	V3	V3
Addierer (1CS, $ 20)	4	3	3	3	2	2	2	2
Multiplizierer (2CS, $ 30)	0	0	0	0	0	0	0	0
Shifter (1CS, $ 25)	0	0	0	0	0	0	0	0
Berechnungszeit (s)	1	1	1	2	3	12	271	142
entstandene Kosten ($)	80	60	60	60	40	40	40	40

Tabelle D.4: EWF, Ergebnisse mit alternativen Versionen

D.2 Differential Equation Solver

Die Verhaltensbeschreibung des Differential-Equation-Solvers [PaKnGi86] beinhaltet eine Verzweigung und eine Schleife. Der Rumpf der Schleife, welcher 10 arithmetische Operationen enthält, soll hier als Entwurfsbeispiel dienen und mit dem OSCAR-System synthetisiert werden.

Tabelle D.5 faßt die Syntheseergebnisse zusammen, welche sich bei unterschiedlichem Bausteinangebot mit dem vereinfachten Bindungsmodell ergeben[7].

Tabelle D.6 stellt die Syntheseergebnisse vor, die bei Anwendung des heuristischen LP-Modells berechnet werden. Im Vergleich zu Tabelle D.5 zeigen sich wiederum erhebliche Geschwindigkeitssteigerungen, aber auch einige suboptimale Ergebnisse[8].

Tabelle D.7 stellt die Ergebnisse zusammen, die sich bei Anwendung von Chaining ergeben. Die angegebenen Ausführungszeiten erlauben Chaining für jeweils zwei aufeinanderfolgende Additions- oder Subtraktionsoperationen[9].

Tabelle D.8 stellt die Verbindungsoptimierung vor. Für den Rumpf des Differential-Equation-Solvers sind jeweils 4 arithmetische Einheiten und 4 Ein-Ausgabeports miteinander zu verbinden. Von den theoretisch möglichen 64 Kombinationen sind für dieses Beispiel maximal 24 verschiedene Verbindungen möglich. Die Tabelle stellt die Ergebnisse des IP- und des LP-Modells den Ergebnissen ohne Verbindungsoptimierung gegenüber, welche mit dem vereinfachten Bindungsmodell ermittelt wurden[10].

[7] Tabelle D.5, Aufruf des OSCAR-Systems:
`oscar -v -Od -Ob -Oo -l <limit> -a <steps> diffeq_body oscar_behaviour;`

[8] Tabelle D.6, Aufruf des OSCAR-Systems:
`oscar -v -Od -Ob -Oo -Ol -l <limit> -a <steps> diffeq_body oscar_behaviour;`

[9] Tabelle D.7, Aufruf des OSCAR-Systems:
`oscar -v -Od -Ob -Oo [-Oc] -l <limit> -a <steps> diffeq_body oscar_behaviour;`

[10] Tabelle D.8, Aufruf des OSCAR-Systems:
`oscar -v -Od -Os -Oi [-Ol] -l <limit> -a <steps> diffeq_body oscar_behaviour;`

Kontrollschritte (CS)		5	6	7	8	9
Addierer	(1CS, $ 20)	0	0	0		
Subtrahierer	(1CS, $ 20)	0	0	0		
Multiplizierer	(1CS, $ 30)	1	2	2		
Add-Sub-ALU	(1CS, $ 25)	1	1	1		
Add-Mul-ALU	(1CS, $ 40)	1	0	0		
entstandene Kosten ($)		95	85	85		
Berechnungszeit (s)		1	22	1207		

Kontrollschritte (CS)		5	6	7	8	9
Addierer	(1CS, $ 20)	-	-	0	1	0
Subtrahierer	(1CS, $ 20)	-	-	0	1	0
Multiplizierer	(2CS, $ 30)	-	-	2	2	2
Add-Sub-ALU	(1CS, $ 25)	-	-	1	0	1
Add-Mul-ALU	(2CS, $ 40)	-	-	1	0	0
entstandene Kosten ($)		-	-	125	100	85
Berechnungszeit (s)		-	-	2	22	3

Kontrollschritte (CS)		5	6	7	8	9
Addierer	(1CS, $ 20)	-	-	0	0	0
Subtrahierer	(1CS, $ 20)	-	-	0	0	0
Multiplizierer	(1:2CS, $ 30)	-	-	2	2	1
Add-Sub-ALU	(1CS, $ 25)	-	-	1	1	1
Add-Mul-ALU	(1:2CS, $ 40)	-	-	0	0	0
entstandene Kosten ($)		-	-	85	85	55
Berechnungszeit (s)		-	-	1	208	570

Tabelle D.5: DFQ, Ergebnisse bei unterschiedlichem Bausteinangebot

Kontrollschritte (CS)		5	6	7	8	9	10	11
Addierer	(1CS, $ 20)	0	0		1	1	1	
Subtrahierer	(1CS, $ 20)	0	0		1	1	1	
Multiplizierer	(1CS, $ 30)	1	2	N. S.	1	1	1	
Add-Sub-ALU	(1CS, $ 25)	1	1		0	0	0	
Add-Mul-ALU	(1CS, $ 40)	1	0		0	0	0	
entstandene Kosten ($)		95	85	N. S.	70	70	70	
Zeit Lauf 1 (s)		1	1	1	1	1	1	
Zeit Lauf 2 (s)		1	2	(4)	6	38	226	
Berechnungszeit (s)		2	3	N. S.	7	39	227	

Kontrollschritte (CS)		5	6	7	8	9	10	11
Addierer	(1CS, $ 20)	-	-	1	1	1	1	1
Subtrahierer	(1CS, $ 20)	-	-	1	1	1	1	1
Multiplizierer	(2CS, $ 30)	-	-	3	2	2	2	2
Add-Sub-ALU	(1CS, $ 25)	-	-	0	0	0	0	0
Add-Mul-ALU	(2CS, $ 40)	-	-	0	0	0	0	0
entstandene Kosten ($)		-	-	130	100	100	100	100
Zeit Lauf 1 (s)		-	-	1	1	1	1	1
Zeit Lauf 2 (s)		-	-	1	2	2	5	184
Berechnungszeit (s)		-	-	2	3	3	6	185

Kontrollschritte (CS)		5	6	7	8	9	10	11
Addierer	(1CS, $ 20)	-	-	0		1	1	1
Subtrahierer	(1CS, $ 20)	-	-	0		1	1	1
Multiplizierer	(1:2CS, $ 30)	-	-	2	N. S.	1	1	1
Add-Sub-ALU	(1CS, $ 25)	-	-	1		0	0	0
Add-Mul-ALU	(1:2CS, $ 40)	-	-	0		0	0	0
entstandene Kosten ($)		-	-	85	N. S.	70	70	70
Zeit Lauf 1 (s)		-	-	1	1	1	1	1
Zeit Lauf 2 (s)		-	-	1	(6)	19	6	16
Berechnungszeit (s)		-	-	2	N. S.	20	7	70

Tabelle D.6: DFQ, Ergebnisse bei Anwendung des LP-Modells

Mit Chaining: (CS)	4	5	6	7	8	9
Addierer (450ns, $ 20)	1	1	1	1	1	1
Subtrahierer (450ns, $ 20)	2	1	1	1	1	1
Multiplizierer (700ns, $ 30)	3	2	2	2	1	1
entstandene Kosten ($)	150	100	100	100	70	70
Berechnungszeit (s)	1	1	9	237	625	266
Ohne Chaining: (CS)	4	5	6	7	8	9
Addierer (1CS, $ 20)	-	1	1	1	1	1
Subtrahierer (1CS, $ 20)	-	1	1	1	1	1
Multiplizierer (1CS, $ 30)	-	2	2	2	1	1
entstandene Kosten ($)	-	100	100	100	70	70
Berechnungszeit (s)	-	1	5	127	360	210

Tabelle D.7: DFQ, Ergebnisse bei Anwendung von Chaining

IP-Modell: (CS)	5	6	7	8
Addierer (1CS, $ 20)	1	1	1	
Subtrahierer (1CS, $ 20)	1	1	1	
Multiplizierer (1CS, $ 30)	2	2	2	
Verbindungen ($ 1)	17	16	16	
entstandene Kosten ($)	117	116	116	
Berechnungszeit (s)	1	97	2227	
LP-Modell: (CS)	5	6	7	8
Addierer (1CS, $ 20)	1	1	1	1
Subtrahierer (1CS, $ 20)	1	1	1	1
Multiplizierer (1CS, $ 30)	2	2	2	1
Verbindungen ($ 1)	17	16	16	15
entstandene Kosten ($)	117	116	116	85
Berechnungszeit (s)	2	21	40	234
ohne Verb.opt. (CS)	5	6	7	8
Addierer (1CS, $ 20)	1	1	1	1
Subtrahierer (1CS, $ 20)	1	1	1	1
Multiplizierer (1CS, $ 30)	2	2	2	1
Verbindungen ($ 1)	17	17	18	15
entstandene Kosten ($)	117	117	118	85
Berechnungszeit (s)	1	5	125	357

Tabelle D.8: DFQ, Ergebnisse bei Anwendung von Verbindungoptimierung

Literaturverzeichnis

[Ac93] H. Achatz. *Extended 0/1 LP formulation for the scheduling problem in high-level synthesis.* Procedings EURO-DAC '93, 1993.

[Be93] M. R. C. M. Berkelaar. *UNIX^{tm} manual page of lp_solve.* Eindhoven University of Technology, Design Automation Section, 1993.

[DaOrWo55] G. B. Dantzig, A. Orden, P. Wolfe. *The Generalized Simplex Method for Minimizing a Linear Form under Linear Inequality Constraints.* Pacific J. Math., no. 2, 1955.

[GaKu83] D. D. Gajski, R. H. Kuhn. *New VLSI Tools.* IEEE Computer, Ausgabe 12, S. 11-14, 1983.

[GeEl91] C. H. Gebotys, M. I. Elmasry. *Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis.* 28th ACM/IEEE Design Automation Conference, S. 2-7, 1991.

[GeEl92] C. H. Gebotys, M. I. Elmasry. *Optimal VLSI Architectural Synthesis.* Kluwer Academic Publishers, 1992.

[GeEl93] C. H. Gebotys, M. I. Elmasry. *Global Optimization Approach for Architectural Synthesis.* IEEE Transactions on CAD, Vol. 12, No. 9, 1993.

[Go58] R. E. Gomory. *Outline of an Algorithm for Integer Solution to Linear Programs.* Bulletin American Math. Society, no. 5, 1958.

[HwLeHs91] C.-T. Hwang, J.-H. Lee, Y.-C. Hsu. *A Formal Approach to the Scheduling Problem in High Level Synthesis.* IEEE Transactions on CAD, Vol. 10, No. 4, 1991.

[IEEE88] Design Automation Standards Subcommittee of the IEEE. *IEEE standard VHDL language reference manual (IEEE Std. 1076-87).* IEEE Inc., New York, 1988.

[KuPa87] F. J. Kurdahi, A. C. Parker. *REAL: A Program for Register Allocation*. 24th ACM/IEEE Design Automation Conference, S. 210-215, 1987.

[KuWhKa85] S. Y. Kung, H. J. Whitehouse, T. Kailath. *VLSI and Modern Signal Processing*. Prentice Hall, 1985.

[LaMaDö94a] B. Landwehr, P. Marwedel, R. Dömer. *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming*. Procedings EURO-DAC '94, 1994.

[LaMaDö94b] B. Landwehr, P. Marwedel, R. Dömer. *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming*. Forschungsbericht des Fachbereichs Informatik der Universität Dortmund, 1994.

[LaMa93] B. Landwehr, P. Marwedel. *Intelligent Library Component Selection and Management in an IP-model based High Level Synthesis System*. IFIP Workshop on Logic and Architecture Synthesis, S. 381-400, 1993.

[Ma93] P. Marwedel. *Synthese und Simulation von VLSI-Systemen*. Hanser Verlag München, 1993.

[Marmor93] P. Marwedel, B. Landwehr, I. Markhof u. a. *Endbericht der Projektgruppe MARMOR*. Interne Berichte der Abteilung Informatik der Universität Dortmund, 1993.

[McPaCa88] M. C. McFarland, A. C. Parker, R. Camposano. *Tutorial on High-Level Synthesis*. 25th ACM/IEEE Design Automation Conference, S. 330-336, 1988.

[NeWo88] G. L. Nemhauser, L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience Series, New York, 1988.

[PaSt82] C. H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[PaKnGi86] P. G. Paulin, J. P. Knight, E. F. Girczyc. *HAL: A multi-paradigm approach to automatic data path synthesis*. 23rd ACM/IEEE Design Automation Conference, S. 263-270, 1986.

[RiJaLe92] M. Rim, R. Jain, R. De Leone. *Optimal Allocation and Binding in High-Level Synthesis*. 29th ACM/IEEE Design Automation Conference, S. 120-123, 1992.

[Ti94] A. H. Timmer. *Improved Execution Interval and Binding Analysis*. Proceedings of the IEEE, 1994.

[Compass91] COMPASS Design Automation. *Users Manual - V8R3*. 1991.

[We93] I. Wegener. *Spezialvorlesung Operations Research.* Skript zur Vorlesung, Fachbereich Informatik der Universität Dortmund, 1993.

Dortmund, 9. November 1994

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine weiteren als die angegebenen Hilfsmittel verwendet habe.

Rainer Dömer