Ein Software-System

zur Synthese von Rechnerstrukturen

und zur Erzeugung von Mikrocode

Peter Marwedel
Universität Dortmund
Informatik Lehrstuhl XII
Forschungsbericht Nr. 356, Juli 1990

Ein Software - System
zur Synthese von Rechnerstrukturen
und zur Erzeugung von Mikrocode

von

Peter Marwedel
Universität Dortmund
Informatik Lehrstuhl XII

Dortmund, im Juli 1990

unveränderter Nachdruck der Habilitationsschrift
Original:
Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität Kiel,
September 1985

Inhaltsverzeichnis

1. Einleitung

Aufgrund der Fortschritte im Bereich der Halbleitertechnologie werden immer komplexere integrierte Schaltkreise hergestellt. Als Folge davon muß der Entwurf dieser Schaltkreise von immer höheren Spezifikationsebenen ausgehen. Während die Spezifikation der ersten gefertigten Halbleiter, nämlich der Einzeltransistoren, nur elektrische Eigenschaften beinhaltete, so umfaßt diese heute beim Entwurf von Mikroprozessoren ganze Maschinenbefehlssätze.

Damit wird der Bereich der Hardware verlassen und Anforderungen aus dem Bereich der Software haben wesentlichen Einfluß auf den Entwurf von Schaltkreisen. Hierzu folgendes Zitat [Mye83] (VLSI=very large scale integration; Höchstintegration von Schaltkreisen)

"One of the effects of VLSI is to give designers flexibility as to whether to put a particular function in hardware or software. However, there is no way, according to Mayo, to know which is best except through simulation. Consequently integrated design aids are needed for the joint design of hardware and software. The two sides must be developed in parallel and hardware / software tradeoffs made before the product is actually built."

Hilfsmittel zum integrierten Entwurf von Hardware und Software sind aber bislang kaum entwickelt worden. Werkzeuge zum computerunterstützten Entwurf (CAD) von Digitalrechnern kann man anhand von sogenannten Schichtenmodellen klassifizieren. Solche Modelle stellen unterschiedlich abstrakte Sichtweisen von Rechnern dar. Ein für die Realisierung in VLSI mögliches Schichtenmodell zeigt Abbildung 1.1. Für jede der Ebenen enthält diese Abbildung typische Komponenten, d.h. Teile, aus denen Beschreibungen auf der jeweiligen Ebene zusammengesetzt sind.

Name der Schicht bzw. Ebene	Komponenten der Beschreibung - (Beispiele)
funktionale Ebene	Rekursionsformeln, prädikatenlogische Axiome
algorithmische Ebene	PASCAL-Anweisungen
Maschinenbefehlsebene	Semantikspezifikation einzelner Maschinenbefehle, z.B. in ISPS [BarBarCatSie77]
Register-Transfer (RT)-Verhaltensebene ("register transfer behaviour" [Par841)	Zuweisungen an Speicher und Register
RT-Strukturebene ("register transfer structure" [Par841)	Speicher, Register, Busse, arithm./log. Bausteine
Logik-Ebene ("logic level")	Boolsche Ausdrücke
Gatter-Ebene	Gatter, Leitungen
Switch-Ebene	Transistoren, Widerstände
symbolische Layout-Ebene	Sticks [MeaCon80]
Layout - Ebene	Rechtecke (vgl. CIF [MeaCon801)

Abb. 1.1 Schichtenmodell

Auf der funktionalen Ebene wird die von einem Programm auszuführende Funktion z.B. durch Rekursionsformeln dargestellt. Eine Anwendung findet diese Technik u.a. in der `Software-Spezifikation. Mit Rücksicht auf klassische Implementierungen von Funktionsaufrufen werden diese Formeln dann meist in iterative Formen überführt [BauWös81].

Charakteristisch für die algorithmische Ebene sind höhere Kontrollstrukturen (wie z.B. Prozeduraufrufe und FOR-Schleifen) sowie höhere Datenstrukturen (wie z.B. Arrays). Die RT-Verhaltensebene unterscheidet sich von der algorithmischen Ebene u.a. durch die Abwesenheit impliziter Operationen (wie z.B. impliziter Index- und Adreßrechnungen) und die Abwesenheit höherer Kontrollstrukturen mit Ausnahme von bedingten Anweisungen.

Im Falle von Rechnern mit zwei Programmebenen (d.h. Ebenen, auf denen der Rechner "programmierbar" ist) bezeichnet man die RT-Verhaltensebene auch als Mikroprogrammebene. Bei Rechnern mit einer Programmebene fallen die Maschinenbefehlsebene und die RT-Verhaltensebene zusammen. In diesem Fall macht man die Bezeichnung meist davon abhängig, ob Zuweisungen parallel ausgeführt werden können oder nicht. Ist die parallele Ausführung mehrerer Zuweisungen möglich, so spricht man von der Mikrobefehlsebene, sonst von der Maschinenbefehlsebene.

Zwischen den beiden RT-Ebenen wird in manchen Fällen nicht deutlich unterschieden. Auf der anderen Seite wird zum Teil auf mehreren Ebenen zwischen Verhaltens- und Strukturbeschreibung differenziert [GajKuh83, Sah85]. In dieser Arbeit scheint es angemessen, RT - Verhaltensund Strukturbeschreibungen als Darstellungen auf zwei verschiedenen Ebenen zu betrachten, ansonsten aber nicht weiter zu differenzieren.

In der Vergangenheit entwickelte CAD-Systeme haben sich überwiegend mit der RT-Verhaltensebene und darunter liegenden Ebenen beschäftigt. Die RT-Verhaltensebene selbst wurde überwiegend für Simulationen genutzt.

Aufgrund der Erhöhung der Komplexität digitaler Systeme ist es erforderlich, höhere Beschreibungsebenen zu verwenden. Das in Kiel entwickelte **MIMOLA-Software-System** (MSS) soll ein Beitrag sein, die eingeschränkte Verwendung der höheren Ebenen zu überwinden. Das **MSS ist ein System zum computerunterstützten Entwurf von Digitalrechnern.**

Die **Hardware-Synthese**, die **Codeerzeugung**, die **Testerzeugung** und der **Simulator** bilden die vier Kernstücke des MSS. Im Rahmen dieser Arbeit wurden die beiden ersten Kernstücke konzipiert und implementiert sowie die Gesamtkonzeption des MSS betreut.

Der **erste Hauptteil** dieser Arbeit stellt die Hardware-Synthese des MSS vor. Als **Synthese bezeichnet man das Zusammensetzen von Komponenten einer niedrigeren Ebene zu einem System, welches ein auf einer höheren Ebene beschriebenes Verhalten zeigt.**

Ausgangspunkt der Synthese im MSS sind Überlegungen darüber, auf welcher Ebene die Verhaltensbeschreibung des zu entwerfenden Rechners (d.h. die Entwurfsspezifikation) erfolgen sollte. Diese Spezifikation sollte angeben, was der Rechner leisten soll und nicht wie er es leisten soll. Die Aufgabe von Digitalrechnern besteht nun im Lösen von Anwenderproblemen. Es liegt daher nahe, diese Anwenderprobleme selbst als wichtigsten Teil der Spezifikation zu betrachten. Die vielfach benutzte Spezifikation in Form eines auszuführenden Maschinenbefehlssatzes oder von Beschreibungen auf darunter liegenden Ebenen schränkt den möglichen Entwurfsspielraum unnötig ein.

Die Anwenderprobleme werden im MSS repräsentiert durch **Anwenderprogramme.** Diese Anwenderprogramme kennzeichnen den Typ des zu entwerfenden Prozessors. So hängen beispielsweise die Zahl und Art der Arithmetikbausteine von den im Programm vorkommenden Operationen ab. Soll ausnahmsweise eine Maschine mit einem vorgegebenen Maschinenbefehlssatz entworfen werden, so kann ein Interpreter für diesen Befehlssatz als Entwurfsspezifikation benutzt werden.

Aufgabe des in dieser Arbeit beschriebenen Entwurfsverfahrens ist die Ableitung einer Hardwarestruktur mit einem für diese Programme

günstigen Kosten / Leistungsverhältnis. Diese Struktur wird vom MSS maschinell aus dem Programm erzeugt. Ein großer Vorteil der maschinellen Erzeugung liegt darin, daß die Rechnerstruktur bei vorausgesetzter Korrektheit des MSS korrekt ist in dem Sinne, daß sie die vorgegebenen Programme gemäß deren Semantik richtig ausführt ("correctness by construction"). Auch wenn die formale Korrektheit des MSS nicht bewiesen ist, ergibt sich gegenüber dem Handentwurf immer noch eine relativ hohe Wahrscheinlichkeit, einen korrekten Entwurf zu erhalten. Diese Wahrscheinlichkeit wird noch dadurch erhöht, daß mehrere unabhängige Teile des MSS zu einer gegenseitigen Überprüfung eingesetzt werden können. Auf diese Weise wird vermieden, daß die Übereinstimmung eines manuell entworfenen Rechners mit seiner Spezifikation verifiziert werden muß. Derartige Verifikationen sind in der Regel schwierig durchzuführen.

Ein weiterer Vorteil liegt in der **drastischen Verkürzung der Entwurfszeit.** Damit wird es möglich, mehrere alternative Entwürfe untereinander zu vergleichen und den besten auszuwählen. Dies scheiterte bislang meist aus Zeitgründen.

Als Ebene, auf der die Struktur des Rechners dargestellt wird, benutzt das MSS die **RT-Strukturebene.** Unter "Rechnerstruktur"-Beschreibung wird im folgenden stets eine aus den Komponenten der RT-Strukturebene zusammengesetzte Beschreibung verstanden. Diese Ebene besitzt den Vorteil, weitgehend von Schaltkreis-Technologien unabhängig zu sein. Auf diese Weise wird ein schnelles Veralten des MSS vermieden. Der Entwurfsspielraum umfaßt damit die Freiheit der Auswahl von RT-Modulen und deren Verbindungen untereinander sowie die Wahl des Befehlsformates der ersten Befehlsebene oberhalb der Hardware.

Überwiegend benutzen bisherige CAD-Systeme eine Spezifikation auf niedrigeren Ebenen. So auch die sogenannten **Silicon-Compiler**, die dadurch charakterisiert sind, daß sie Beschreibungen auf der Layout-Ebene erzeugen. Die Entwurfsspezifikation erfolgt bei diesen Compilern auf unterschiedlichen Niveaus,so beim Macpitts-Compiler [Sou83] und beim DSL-System [Ros82] auf der RT-Verhaltensebene und in anderen Fällen [Joh79, GraBUCROb82] auf der RT-Strukturebene.

Unter den Systemen, die wie das MSS eine Beschreibung auf der RTStrukturebene generieren, geht das CMU - DA - System (siehe u.a. [HitTho83, KowTho83]) von einer Spezifikation der Maschinenbefehle und das von Huang [Hua81] entwickelte Verfahren von einer Spezifikation auf der algorithmischen Ebene aus.

Diesen CAD-Systemen ist gemeinsam, daß sie zum Abspeichern von **Zwischenergebnissen** einzelne Register erzeugen. Einschließlich der zu diesen Registern führenden und der von ihnen abgehenden Leitungen, der zugehörigen Multiplexer und Testeinrichtungen (z.B. "scan path") kann sich dadurch ein hoher Hardwarebedarf ergeben [GirKni84]. Im MSS werden statt dessen die **Zellen eines adressierbaren Speichers** benutzt. Dadurch ergibt sich in der Regel eine übersichtlichere Struktur und eine verbesserte Testbarkeit. Auf einen solchen Speicher ist nur eine beschränkte Zahl von parallelen Zugriffen möglich. Das MSS enthält ein Verfahren zur **Zerlegung komplexer arithmetischer Ausdrücke**, welches diese Einschränkung berücksichtigt.

Mit Ausnahme des Ansatzes von Hafer [HafPar83] benutzen bekannte CAD-Systeme lokale Optimierungen. Deren Auswirkung auf die globale Struktur bleibt meist unklar. Hafers Ansatz einer globalen Optimierung führt zu einem Gleichungssystem nicht handhabbarer Komplexität. Im MSS für Teilaufgaben benutzte globale Optimierungen bereiten dagegen keine Komplexitätsprobleme.

Immer wichtiger wird in der VLSI-Technologie die **Minimierung der für Leitungen benötigten Fläche.** Der einzige bekannte Ansatz der Minimierung bereits auf einer hohen Ebene, die Weiterentwicklung von Hafers Arbeit durch Parker et al. [ParKurMli84], besitzt eine noch größere Komplexität als Hafers Ansatz. Die für das MSS entwickelte Methode der Optimierung der Zahl der Leitungen führt in der Praxis zu vertretbaren Rechenzeiten und guten Ergebnissen.

Als Erweiterung gegenüber klassischen Maschinen erlaubt das MSS die Steuerung der **parallelen Ausführung mehrerer Zuweisungen** durch einen einzigen Befehl. Dadurch soll die Ausführungsgeschwindigkeit im Ver

gleich zu üblichen sequentiellen Maschinen gesteigert werden. Die Suche nach geeigneten parallel ausführbaren Zuweisungen heißt **Kompaktierung.** Im MSS wird eine neues Kompaktierungsverfahren benutzt, welches im Gegensatz zu üblichen Verfahren keine partielle Ordnung der zu kompaktierenden Zuweisungen voraussetzt und durch eine verzögerte Vergabe der für die Speicherung von Zwischenergebnissen benutzten Hilfszellen ("delayed binding") stärker kompaktieren kann.

In gewissem Umfang läßt sich der Entwurfsspielraum der Synthese durch Variation von Ressource-Restriktionen beeinflussen. Nach Durchführung einer Reihe von Syntheseläufen wird der Designer jedoch mehr und mehr eine konkrete Vorstellung von der Hardwarestruktur entwickeln. Weitgehend wird diese in der Regel einer der automatisch generierten Strukturen entsprechen, jedoch wird er häufig in einigen kleineren Details davon abweichen wollen.

Diese Abweichungen sind das Ergebnis eines kreativen Denkprozesses des Designers, der vom MSS durch die Bereitstellung von statistischen Daten über Hardware-Auslastung und Laufzeit-Abschätzungen unterstützt wird. Es kommt darauf an, das Ergebnis dieses Denkprozesses zur Verbesserung des Entwurfes nutzen zu können.

Detailänderungen des Entwurfes können relativ einfach durch Editieren der vom MSS erstellten Rechnerstrukturbeschreibung ausgedrückt werden. Dabei wird davon Gebrauch gemacht,daß auch das Entwurfsergebnis mit der beim MSS benutzten Sprache MIMOLA (maehine independent microprogramming language) formal beschrieben werden kann. Hier bewährt es sich, daß MIMOLA sowohl zur Darstellung von Algorithmen als auch zur vollständigen Wiedergabe von Rechnerstrukturen (einschließlich aller Verbindungen) geeignet ist.

Ein Vorteil der automatischen Synthese ist die relativ große Wahrscheinlichkeit, einen korrekten Entwurf zu erhalten. Erlaubt man nun aber manuelle Modifikationen der Rechnerstruktur, so können sich wieder Entwurfsfehler einschleichen. Einer der Grundgedanken des MIMOLA-Entwurfsverfahrens ist, daß diese Entwurfsfehler vom MIMOLA-System erkannt werden sollen. Wenn man akzeptiert, daß manuelle Eingriffe

erforderlich sind (und alle bisherigen Erfahrungen sprechen dafür), dann muß man diese auch vom Entwurfssystem her unterstützen!

Eine Rechnerstruktur ist im Sinne des MIMOLA-Systems korrekt entworfen, wenn sie in der Lage ist, die vorgegebenen Programme semantisch richtig zu bearbeiten. Unter der Voraussetzung, daß die MIMOLA-Strukturbeschreibung die echte Hardware richtig wiedergibt, reduziert sich das Problem des Korrektheitsbeweises auf den Nachweis, daß die Programme korrekt in den Maschinencode des Rechners zu übersetzen sind.

Diese Aufgabe wird im MSS gelöst, indem versucht wird, diesen Maschinencode mit Hilfe eines Übersetzers ([Mar84]) zu erzeugen. Dieser Übersetzer ist Gegenstand des **zweiten Hauptteils** dieser Arbeit.

Gelingt die Übersetzung und ist der Übersetzer selbst korrekt, so ist die Maschine korrekt. Mißlingt sie, so ist entweder die Maschine inkorrekt oder der Übersetzer für die Maschine nicht geeignet.

Das führt zu der Frage, wie man überhaupt zu einem Übersetzer für eine in MIMOLA beschriebene Maschine gelangt. Sicherlich kann man einem Hardware-Designer nicht zumuten, für jede in Betracht kommende Maschine selbst einen Compiler zu schreiben.

Zur Lösung dieses Problems wurde als Teil des MSS ein maschinenunabhängiger Übersetzer entwickelt. Dieser benutzt MIMOLA-Programme und MIMOLA-Hardwarebeschreibungen als Eingabe und versucht, die Programme in Maschinenprogramme zu übersetzen. Da sich der "Zielrechner", d.h. der Rechner für den Code erzeugt wird, bei einem solchen Ansatz leicht austauschen läßt, wird ein solcher Übersetzer im Englischen als **"retargetable code generator"** bezeichnet.

Um den Entwerfer nicht auf einen bestimmten Satz von Modifikationen einzuschränken, soll der Codegenerator in der Lage sein, für alle in MIMOLA darstellbaren Strukturen Code erzeugen zu können. Das bedeutet aber, daß man ihn nicht nur für Strukturen benutzen kann, die durch die Synthese erzeugt wurden.

Wegen der notwendigen Kompatibilität zum Synthesesystem ist der Übersetzer des MSS so beschaffen, daß er Code für Maschinen erzeugen kann, deren Befehle die parallele Ausführung mehrerer Zuweisungen bewirken. Diese Eigenschaft erlaubt aber gerade die Erzeugung von Mikrocode durch das MSS und der Codegenerator des MSS besitzt mit der Erzeugung von Mikrocode eine wichtige. zweite Aufgabe. Es ist ein Ziel dieser Arbeit, der **Mikroprogrammierung mit Hilfe des Codegenerators weitere Anwendungen zu erschließen.**

Generell ist der Codegenerator in der Lage, Code für die **erste Programmebene oberhalb** der **echten Hardware** zu erzeugen, da seine Maschinenbeschreibung eine reine Hardwarebeschreibung ist. Je nachdem, ob der Rechner auf einer oder zwei Ebenen programmierbar ist, wird diese Ebene üblicherweise als Maschinenbefehlsebene oder als Mikroprogrammebene bezeichnet. Konzipiert ist das MSS in erster Linie für Maschinen mit **einer** Programmebene. Für solche Maschinen wird die erste Ebene oberhalb der Hardware üblicherweise als Maschinenbefehlsebene bezeichnet. Da auf dieser Ebene aber die parallele Ausführung von Anweisungen erlaubt ist, besitzt sie gleichzeitig typische Merkmale der Mikroprogrammebene.

Der verstärkte Einsatz der Mikroprogrammierung wird z.Zt. durch einen Mangel an Programmierwerkzeugen behindert. Meist werden Mikroprogramme immer noch mit Hilfe von wenig komfortablen Assemblern erstellt. Mehrere Gründe verhindern bislang den verstärkten Einsatz von Mikroprogramm-Compilern. Einige dieser Gründe sind:

- Die mögliche Parallelität erschwert die Codeerzeugung.
- Viele der verfügbaren Mikroarchitekturen sind nicht leicht zu programmieren, da die Einfachheit der Programmierung kein Entwurfsziel bei deren Konstruktion war.
- In vielen Fällen müssen vom Compiler Zeitbedingungen beachtet werden.
- Die Datentypen der Programmiersprache und der Mikroarchitektur können stark divergieren.
- Fast immer wird extrem schneller Code benötigt.
- Während der Entwicklung eines Rechners wird der Mikrocode bereits

in einer sehr frühen Phase benötigt und auf die Entwicklung eines Compilers kann nicht gewartet werden. In späteren Phasen werden neue Mikroprogramme meist kaum noch erstellt.
Da bereits für eine bestimmte Maschinenarchitektur verschiedene Mikroarchitekturen existieren, ist die Zahl verschiedener Zielrechner besonders groß.

Besonders die beiden letzten Gründe zeigen, daß Mikroprogramm-Compiler nur erfolgreich sein können, wenn sie leicht auf andere Maschinen angepaßt werden können. Daher sind von der Zielmaschine unabhängige Compiler in der Mikroprogrammierung noch wichtiger als in der üblichen Programmierung. Folglich wurde auf die Verwendbarkeit des MSS-Codegenerators unabhängig vom Synthesesystem besonderer Wert gelegt.

Eine gute Übersicht über maschinenunabhängige Codegeneratoren bietet Ganapathi [GanFisHen83]. Nur wenige Arbeiten gibt es im Bereich maschinenunabhängiger Mikrocode-Compiler. Für diese Arbeit relevant sind im wesentlichen Ergebnisse dreier Arbeitsgruppen: das MPG - System [BabHag81], die Arbeiten von Vegdahl [Veg82, Veg82a, Veg83] und die Veröffentlichungen von Mueller und Varghese [MueVar83, MueVarAll84].

Die Autoren stellen Verfahren zur Erzeugung von Mikrocode anhand einer formalen Spezifikation der Zielmaschine vor. Diese Beschreibungen enthalten jedoch nicht nur die Darstellung der reinen Hardware in Form von Moduln und Verbindungen. Vielmehr muß eine manuelle Vorverarbeitung erfolgen. So muß bei Mueller beispielsweise eine von mehreren Möglichkeiten zum Transport von Daten zwischen zwei Speichern manuell ausgewählt werden. Dadurch entsteht zusätzlicher Aufwand für den Benutzer und durch die vorzeitige Auswahl einer der Möglichkeiten kann die Optimalität verlorengehen.

Bei dem für das MSS entwickelten Codegenerator entfällt diese Vorverarbeitung. Besondere Kenntnisse im Bereich des Compilerbaus sind zum Erstellen einer Maschinenbeschreibung nicht erforderlich.

Der Codegenerator basiert auf einer Analyse der Verbindungen innerhalb

einer Rechnerstruktur. Fehlende Verbindungen führen zu Fehlermeldungen des Codegenerators. Daher kann mit ,dem Codegenerator geprüft werden, ob die Verbindungen ausreichen, um das durch Programme beschriebene Verhalten erzielen zu können. In einer Anwendung bei der Firma Honeywell hat sich die Benutzung des MIMOLA-Systems allein schon durch die Möglichkeit, die Vollständigkeit einer Verbindungsliste zu prüfen, rentiert [Zim85].

Der **Simulator** des MSS kann u.a. die erzeugten Maschinenprogramme simulieren. Damit ergibt sich die Möglichkeit einer vom Codegenerator und vom Synthesesystem **unabhängigen Uberprüfung der vom MSS generierten Maschinenprogramme** und so eine größere Sicherheit bei der Benutzung des MIMOLA-Systems.

Voraussetzung für die Entwicklung der Hardware-Synthese und der Codeerzeugung ist die Definition einer Sprache mit den Ausdrucksmöglichkeiten von MIMOLA. Mit dieser Sprache **kann sowohl Software als auch Hardware beschrieben werden.** Wesentlich ist dabei, daß MIMOLA **zwischen der Darstellung von Hardware und Software unterscheidet** und die **Beschreibung von Leitungen ermöglicht.** Zur Simulation entwickelte Sprachen erlauben zwar die Darstellung der Funktion von Hardware und Software. Sie unterscheiden aber nicht zwischen Hardware-Bausteinen und Software-Funktionen oder -Prozeduren und sind daher für die hier vorgestellten Methoden der Synthese und Codeerzeugung nicht geeignet.

MIMOLA kann allgemein als Ausgangspunkt für weitere Untersuchungen der Wechselwirkung zwischen Hardware und Software angesehen werden.

Kapitel 2 dieser Arbeit enthält eine Einführung in die Sprache MIMOLA. Kapitel 3 behandelt Software-Komponenten, die der Rechnersynthese und der Codeerzeugung vorgeschaltet sind. Anschließend wird in Kapitel 4 die Rechnersynthese und in Kapitel 5 die Codeerzeugung vorgestellt. Das Kapitel 6 gibt einen Überblick über die sog. Systemunterteile. Diese können für die Komponenten der Kapitel 4 und 5 eine Nachverarbeitung vornehmen. In den Abschnitten 4.13 und 5.4 wird exemplarisch auf einige Anwendungen des MSS eingegangen.

z. Die Sprache MIMOLA

2.1 Hardware-Beschreibung

2.1.1 Syntax

Das in dieser Arbeit beschriebene MSS wird zur Unterscheidung von seinem Vorläufer, dem MSS1 [Zim80, Mar79, Mar80a], auch MSS2 genannt. Beide Systeme unterscheiden sich in ihrem Funktionsumfang, ihrem inneren Aufbau und ihrer Eingabesprache voneinander ganz erheblich. Das MSS1 besteht im wesentlichen aus einem Syntheseprogramm mit relativ lokalen Optimierungen (dieses Programm wird z.Zt. bei der Firma Honeywell industriell eingesetzt). Die übrigen Komponenten des hier vorgestellten Systems, wie z.B. die Codeerzeugung für vorgegebene Rechnerstrukturen, die Simulation und die Testerzeugung sind im MSS1 nicht vorhanden. Aufgrund von Speicherplatzbeschränkungen ist weder eine Funktionserweiterung des MSS1 noch eine Erweiterung der Eingabesprache des MSS1

Das MSS2 besteht im Gegensatz zum MSS1 aus einer Reihe untereinander kommunizierender Programme. Aufgrund dieses Ansatzes ist es möglich, für das MSS2 eine wesentlich erweiterte und benutzerfreundlichere Eingabesprache zu verwenden. Die vom MSS1 akzeptierte Form von MIMOLA [Zim77, MarZim79] ist syntaktisch von anderen Sprachen weitgehend unabhängig. Die jetzige Form von MIMOLA [JöhMar] basiert (soweit möglich) auf der Syntax anderer bekannter Sprachen, wie z.B. PASCAL.

In der Einleitung wurde bereits erwähnt, daß MIMOLA zur Beschreibung von Hardware und Software (Programmen) geeignet ist. Dementsprechend lautet die Syntax des Axioms der Sprache:

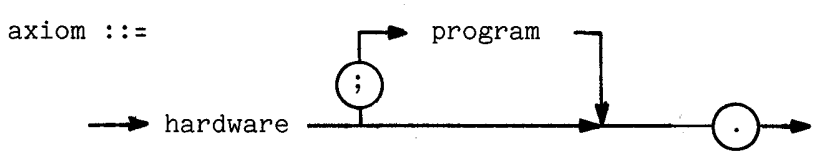


Abb. 2.1 Syntax des Axioms von MIMOLA

In dieser und den folgenden Abbildungen dieses Kapitels bezeichnen groß geschriebene Worte und einzelne Zeichen terminale Symbole und klein geschriebene Worte nichtterminale Symbole. Die Bedeutung hier nicht definierter nichtterminaler Symbole ist im wesentlichen selbsterklärend. Details entnehme man dem MIMOLASprachreport [JöhMar].

Hardware wird in MIMOLA repräsentiert als Menge von Bausteinen und deren Verbindungen untereinander.

Die Syntax der Darstellung von Bausteinen orientiert sich aus Akzeptanzgründen an der Syntax von Prozeduren. Das folgende Beispiel beschreibt einen Bausteintyp, der bis auf die Behandlung von Überträgen und die Länge der Bitstrings dem kommerziellen Typ SN74S381 [Tex77] entspricht:

```
MODULE B74S381(IN a,b:(15:0); FCT s:(2:0); OUT f : (15:0));
 BEGIN
  f:= CASE s OF
       0 : 0;
       1 : b  "-"  a;
       2 : a  "-"  b;
       3 : a  "+"  b;
       4 : a "XOR" b;
       5 : a "OR"  b;
       6 : a "AND" b;
       7 : #FFFF            (*# = hexadezimal*)
      END
 END;
```

In der ersten Zeile wird die Hardware-Schnittstelle (die nach außen geführten Leitungen) beschrieben. Neben den Richtungsqualifikatoren IN, OUT und INOUT (bidirektional) sind im Sinne eines strukturierten Hardware-Entwurfs als spezielle Eingangsqualifikatoren FCT (Funktionsauswahl), ADR (Adresse) und CLK (Clock) zugelassen. Die Ein/Ausgänge können neben den einfachen, unstrukturierten Bitbereichen wie (15:0) auch strukturierte Bereiche umfassen. Damit können z.B. spezielle Übertragsausgänge beschrieben werden.

Def.:
Die durch die Hardware-Schnittstelle eingeführten Identifikatoren heißen S u b p o r t - Identifikatoren.

Der CASE-Ausdruck spezifiziert das **Verhalten** des Bausteins. Alle angegebenen Operationen können von diesem Baustein nach außen exportiert werden.

Zur Vereinfachung der Angabe der Länge von Bitstrings können den Moduldeklarationen globale Typvereinbarungen vorangestellt werden.

Beispiel:

```
TYPE
 word   = (15:0);
 nibble = (3:0);
```

Das folgende Beispiel behandelt einen 2-Port Speicher, d.h. einen Speicher, der unter maximal 2 Adressen einen unabhängigen Zugriff ermöglicht. Diese logische Unabhängigkeit kann in MIMOLA durch die Aufteilung der Beschreibung auf die einzelnen Ports ausgedrückt werden.

```
MODULE Storage
 PORT P1(OUT f : (15:0); ADR ad : (2:0); FCT c : (0));
  BEGIN
   f:= CASE c OF
        0 : Storage(ad);
        1 : TRISTATE;                    (*hochohmig*)
       END
  END,
 PORT P2(IN e:(15:0);OUT f:(15:0);ADR ad:(2:0);FCT c:(1:0));
  BEGIN
   CASE c.(0) OF
    0 : Storage(ad):=e;
    1 : ;
   END,
```

```
 f:= CASE c.(1) OF
      0 : Storage(ad);
      1 : TRISTATE;
     END
END;
```

In diesem Beispiel wurde der Takteingang zur Vereinfachung ausgelassen.

Das Port P2 ermöglicht sowohl das Lesen als auch das Beschreiben einer Zelle in einem Programmschritt. Dabei wird stets der Inhalt der Zelle vor Ausführung des Programmschrittes gelesen (Flankentriggerung). Ports dieser Art werden als **Schreib/Leseports** bezeichnet. Wie noch zu sehen sein wird, bedürfen diese Ports einer besonderen Berücksichtigung. Diese entspricht der Behandlung der sog. Überdeckung von Ziel- und Quelladressen klassischer Maschinenbefehle (vgl. [Jes75]). Insbesondere werden Kopieroperationen zwischen Zellen des Registerspeichers erforderlich.

Beim Anlegen einer 0 an Bit 0 des Kontrolleingangs erfolgt über P2 ein Schreiben in den Speicher. Beim Anlegen einer 1 am Bit 0 wird der Speicherinhalt über P2 nicht verändert. In dieser Arbeit werden die zum Schreiben, bzw. zum Unterdrücken des Schreibens benötigten Codes als "LORD-code" bzw. als "INHIBIT-code" bezeichnet. Bezogen auf die gesamte Breite des Kontrolleingangs können diese Codes im Beispiel als %XO bzw. %X1 dargestellt werden. Dabei kennzeichnet Binärzahlen und X ein "don't Gare" Bit.

Das obige Beispiel behandelt einen wahlfrei adressierbaren Speicher (random access memory, RAM). Generell bezeichnet das Wort "Speicher" in dieser Arbeit einen solchen Speicher.

In beiden vorangegangenen Beispielen werden **Typen** von Hardware-Bausteinen deklariert. Von diesen unterscheiden muß man **Exemplare** eines Typs. Exemplare entsprechen den Variablen im Sinne von PASCAL. Sie werden in der PARTS Deklaration vereinbart. Bei der Vereinbarung von PASCAL-Variablen kann man entweder auf definierte Typen (wie z.B. "INTEGER") Bezug nehmen oder implizit neue Typen (wie z.B.

"SET OF 0..7") definieren. Das gleiche gilt auch für Exemplare von MIMOLA-Hardwarebausteinen.

Beispiel:

```
MODULE Storage ..
 BEGIN
   ....
 END;
PARTS
 SR       : Storage;              (*vordefinierter Modultyp*)
 ALU0,ALU1: MODULE B74S381 ..     (*implizite Definition   *)
            BEGIN
             ....
            END;
```

Die implizite Definition läßt leichter erkennen, welche Eigenschaften ein bestimmtes Exemplar besitzt. Die Möglichkeit der getrennten Definition von Modultypen wird benötigt, da nicht von jedem Modultyp ein Exemplar in einer Hardware vorhanden sein muß. Insbesondere enthalten Modulbibliotheken nur Modultyp-Deklarationen.

Aus Gründen der Kompatibilität mit älteren Teilen des MSS müssen die Bezeichner für Modultypen mit bestimmten Anfangsbuchstaben beginnen. Diese Anfangsbuchstaben sind:

S R für adressierbare Speicher, für Register
A,B,C (speziell: RP = Programmzähler),
oder N
für Netzwerke.

Das MSS geht stets von einem speicherprogrammierbaren Rechner aus. Ein solcher Rechner wird von Befehlen gesteuert, welche in einem **Befehlsspeicher** hinterlegt sind. Der Befehlsspeicher muß nicht notwendig gleichzeitig auch Datenspeicher sein. Befehle werden in der Regel in einzelne Felder aufgeteilt, die weitgehend unabhängig voneinander bestimmte Steuerungsaufgaben übernehmen. Beispielsweise kann ein solches Feld eine Zelle eines Registerspeichers auswählen. Es wird stets angenommen, daß die Befehlsfelder direkt die Hardware steuern und

nicht durch ein weiteres Programm interpretiert werden.

Die Aufteilung von Befehlen in Befehlsfelder wird ähnlich wie ein PASCAL-Record erklärt.

Beispiel:

```
INSTRUCTION
 I : (opr    (10:8),
      enable  (7:6),
      reg2    (5:3),
      reg1    (2:0));
```

In diesem Beispiel werden I.opr, I.enable usw. als Felder des jeweils aktuellen Befehlswortes deklariert. Die Zahlen geben die absoluten Nummern der Bits innerhalb des Befehlsworts an.

Als weitere Sprachelemente enthält MIMOLA Beschreibungsmöglichkeiten für festverdrahtete Konstanten und Busse.

Die Notation für festverdrahtete Konstanten basiert auf der Konstantendeklaration in PASCAL. Als Erweiterung ist zusätzlich die Angabe eines Bitbereichs möglich.

```
CONST
 ZERO = 0.(0);   (*Bereich (0:0), d.h. 1 Bit*)
```

Die Notation für Busse ist selbsterklärend.

Beispiel:

```
BUS
 BBUS : (15:0);
```

Bei MIMOLA wird ein besonderes Gewicht auf die vollständige Darstellung der Verbindungen der Module untereinander gelegt. Die Kenntnis der Verbindungen ist z.B. notwendig, wenn ein synthetisierter Rechner realisiert werden soll.

Die Liste der Verbindungen wird eingeleitet durch das Schlüsselwort CONNECTIONS. In der Liste werden Port-Identifikatoren durch Bindestriche und Subport-Identifikatoren durch Punkte von den unter PARTS deklarierten Bezeichnern getrennt.

Beispiel:

```
CONNECTIONS
  I.reg1      -> SR-P1.ad;
  I.reg2      -> SR-P2.ad;
  SR-P2.f     -> BBUS  ;
  BBUS        -> ALU0.b;
  SR-P1.f     -> ALU0.a;
  I.opr       -> ALU0.s;
  I.enable    -> SR-P2.c;
  ZERO        -> SR-P1.c;
  ALU0.f      -> SR-P2.e;
```

Diese Verbindungen sind enthalten in der Abbildung 2.2. Diese Abbildung könnte Teil des Blockdiagramms eines Rechners sein, der durch die bisherigen Beispiele auszugsweise beschrieben wird.

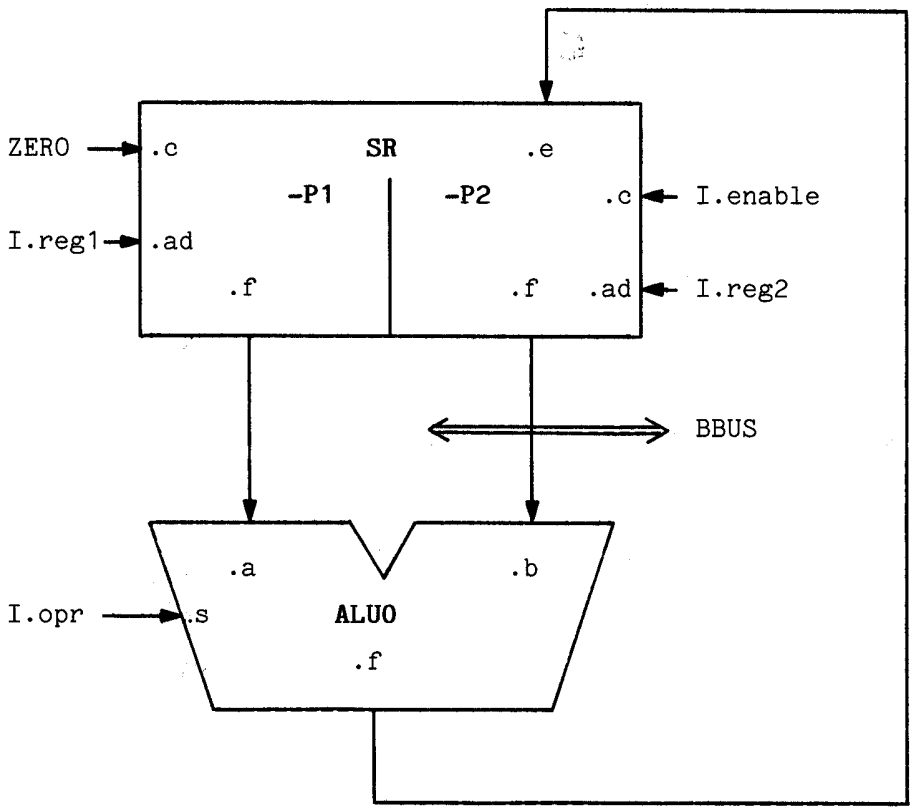


Abb. 2.2 Ausschnitt eines Blockdiagramms (vereinfacht)

Im Prinzip kann die Hardwarebeschreibung beliebig viele Speicher enthalten. In den Beispielen dieser Arbeit wird jedoch vorausgesetzt, daß ein großer, als **Hauptspeicher** bezeichneter Speicher mit dem Namen SH und ein kleinerer, als **Registerspeicher** bezeichneter Speicher mit dem Namen SR als Parts deklariert sind. SR entspricht dem sogenannten "Registersatz" einiger Rechner.

In der Regel dient SH der Aufnahme von Variablen (im Sinne von PASCAL) und SR u.a. der kurzfristigen Abspeicherung von Zwischenergebnissen. Für bestimmte Aufgaben reservierte Speicherbereiche können dem MSS mittels LOCATIONS-Definitionen bekannt gemacht werden.

Beispiele:

```
LOCATIONS_FOR_VARIABLES   SH(12000:0);
LOCATIONS_FOR_TEMPORARIES SR(5:0);
LOCATIONS_FOR_IO          SH(#FFFF:#F000);
```

Außer den bislang erwähnten Sprachelementen können noch die in Abschnitt 2.2.2 besprochenen Programmtransformationsregeln Teil der Hardwarebeschreibung sein. Nachdem bislang die Komponenten der Hardwarebeschreibung behandelt wurden, folgt nun die Syntax im Zusammenhang:

hardware ::=

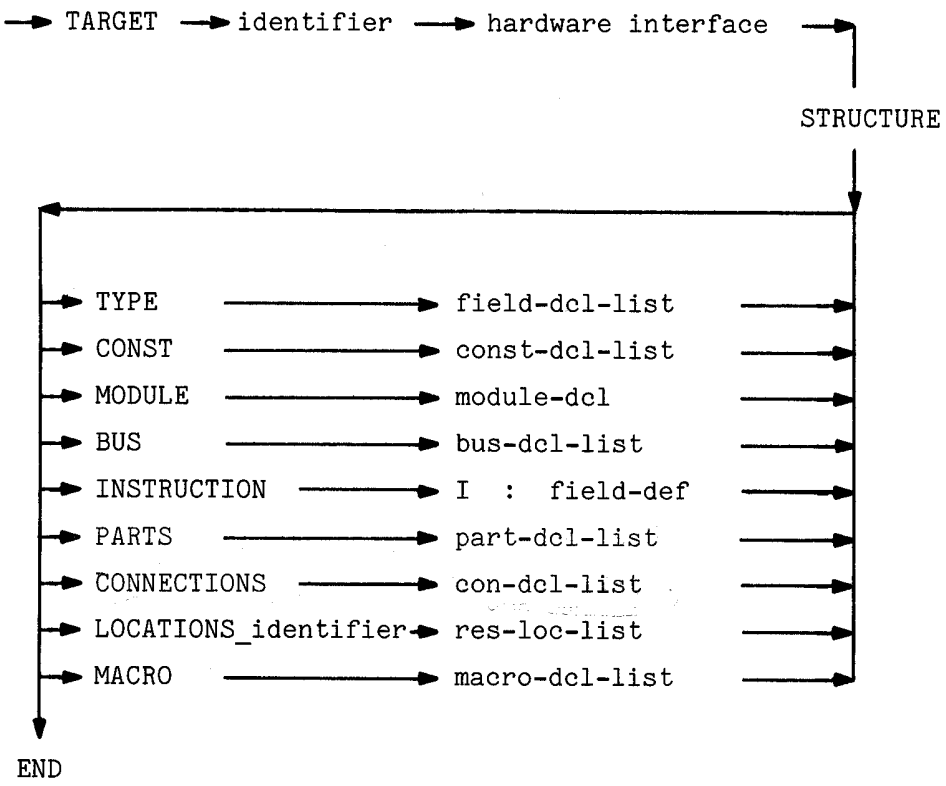


Abb. 2.3 Syntax der Hardwarebeschreibung

Ähnlich wie MIMOLA basieren die Hardware-Beschreibungssprachen CAP [Ram80], LALSD II[Hua81], KARL III [HarLem84], Zeus[Lie84] und (indirekt über ADA) auch VHDL[Sha85] auf PASCAL. Jede dieser Sprachen besitzt ihre eigenen Vorzüge. Es dürfte jedoch kaum gelingen, alle Vorzüge in einer Sprache zu vereinen. Über die im CONLAN-Projekt entwickelte Zwischensprache IREEN [PilBor85] sollte es dennoch möglich sein, für unterschiedliche Quellsprachen dieselben Entwurfswerkzeuge zu benutzen.

Im Sinne des hierarchischen Entwurfs von VLSI-Schaltungen ist es vorteilhaft, Module hierarchisch schachteln zu können. Sowohl die MIMOLA-Syntax als auch die interne Datenstruktur des MSS sind auf eine solche Schachtelung hin ausgelegt. Da die Synthese und die Codeerzeugung davon jedoch noch nicht Gebrauch machen, wurde auf die Darstellung geschachtelter Moduln hier verzichtet.

Generell sind in dieser Arbeit nur diejenigen Teile der Sprache MIMOLA angeführt, die für das Verständnis der Methode wichtig sind. In einzelnen Bereichen geht MIMOLA über die hier behandelte Syntax hinaus (siehe [JöhMar]).

2.1.2 Formales Maschinenmodell

Als Abstraktion von der Syntax der Hardwarebeschreibung dienen die im folgenden eingeführten Begriffe. Darin sei N_0 der Bereich der natürlichen Zahlen unter Einschluß der Null.

Def.:

1. Ein O p e r a t i o n s b e z e i c h n e r ist ein Tupel (op_symbol,arity,length).
   o p _ s y m b o l ist das in der MIMOLA-Beschreibung vorkommende, in " " eingeschlossene Operations - Symbol. op_symbol ist eine Zeichenkette, z.B. +, - oder AND. Für 0-stellige Operationen (Konstanten) ist op_symbol gleich der (Binär-) Zeichenketten-Darstellung des abgelieferten Wertes.
   a r i t y $\in N_0$ ist die Stelligkeit der Operation und dient z.B. der Unterscheidung zwischen 1-stelligem "-" (2-er Komplement) und 2-stelligem "-" (Subtraktion).
   l e n g t h $\in N_0$ bezeichnet die Länge der von dieser Operation bearbeiteten Bitstrings. Mit Hilfe von length wird u.a. zwischen der Addition von 1-, 2- und 4-Byte Integern unterschieden.

2. Sei f ein Operationsbezeichner. Die Funktionen **op_symbol**(f), **arity**(f) und **length**(f) sind definiert als die Projektionen von f auf die jeweiligen Komponenten des Tupels.

Operationsbezeichner charakterisieren die Operation eindeutig. Überdefinitionen ("overloading") des Operationssymbols über unterschiedliche Werte von arity und length hinaus sind nicht zulässig. So wird neben dem Symbol "<" für den Vergleich vorzeichenbehafteter Zahlen das Symbol "'.<!" (vgl. [Ram80]) für den Vergleich vorzeichenloser Zahlen benutzt.

Von üblichen Compilern müssen häufig **Anpassungen zwischen den von der Hardware ausführbaren und den in Programmen vorkommenden Operationen** vorgenommen werden. Es wird angenommen, daß diese im Rahmen des MIMOLA-Systems **in der Regel nicht erforderlich** sind. In Ausnahmefällen können die in Abschnitt 2.2.2 beschriebenen Transformationsregeln zur Anpassung benutzt werden. In der Hardwarebeschreibung und im Programm vorkommende, gleich benannte Operationsbezeichner besitzen immer die gleiche Bedeutung.

Die Definition neuer Operationen durch den Benutzer ist in MIMOLA möglich. Um die Definition neuer Operationssymbole zu vereinfachen, werden sie in MIMOLA in Anführungszeichen eingeschlossen. Die Semantik neuer Operationen muß in Form einer (hier nicht dargestellten) Operationsdeklaration auf die Semantik vordefinierter Operationen zurückgeführt werden.

Zu jeder Operation gehört eine Menge von Eigenschaften wie z.B. die Kommutativität und der Typ des Ergebnisses. Diese Eigenschaften sind im MSS in einer Tabelle gesammelt, die durch den Benutzer erweitert werden kann.

Für vordefinierte Operationssymbole besitzen die Bitstringdarstellungen aller Argumente die gleiche Länge und für length reicht die Angabe einer einzigen Zahl. Eine Ausnahme davon bilden Schiebeoperationen. Diese werden besonders behandelt.

Def.:

3. Die Menge der in einer MIMOLA-Beschreibung enthaltenen Identifikatoren von Modultypen wird mit **modules** bezeichnet.

4. Für m ∈ **modules** ist **oplist**(m) die Menge der von einem Modultyp bereitgestellten Operationen.

5. Die Elemente r ∈ **oplist**(m) heißen O p e r a t i o n s-
b e s c h r e i b u n g e n. Sie werden dargestellt durch ein 7-Tupel (op_
symbol, arity, length, Code, Inputs, Output, timing).
o p - s y m b o 1 , a r i t y und 1 eng t h sind wie oben definiert.

c o d e ist der am Kontrolleingang eines Modultyps zur Ausführung der Operation benötigte Wert. Kann die gleiche Operation unter mehreren Codes ausgeführt werden, so wird dies über getrennte Tupel modelliert. Code ist aus dem Bereich der Zeichenketten Darstellungen von Binärkonstanten (Präfix %) unter Einschluß von don't Gares (X).

n p u t s spezifiziert, wie die Argumente der Operation an die Eingänge des Moduls anzuschließen sind, damit die Operation semantisch korrekt ausgeführt wird. Inputs besteht aus einer geordneten Menge von Tupeln. Für jedes Argument der Operation gibt es genau ein Tupel. Die Komponenten des i-ten Tupels sind die Namen des Ports und des Subports sowie der Bitbereich, an dem das i-te Argument anzuschließen ist. Das Beispiel am Ende dieses Abschnitts zeigt, wie die Tupel konkret dargestellt werden.

O u t p u t gibt an, an welchen Ausgangsleitungen das Ergebnis abgeliefert wird. Zur Darstellung vergleiche man ebenfalls das Beispiel am Ende dieses

t i m i n g enthält Informationen über das zeitliche Verhalten der Operation. Das zeitliche Verhalten kann in MIMOLA über sog. Properties beschrieben werden. Z.B. drückt <TIME=10> eine mittlere Verzögerungszeit von 10 Zeiteinheiten aus. Da für Properties außer den spitzen Klammern keine Syntax fest vorgegeben ist, können Verfeinerungen der Darstellung des zeitlichen Verhaltens (z.B. die Einführung von Unsicherheitsintervallen) nach Bedarf erfolgen.

6. Für alle Operationsbeschreibungen r bezeichnen die Funktionen
op_ symbol (r), arity(r), length(r), code(r), inputs(r), output(r),

timing(r) die Projektionen auf die jeweiligen Tupel-Komponenten.

7. Eine B a u s t e i n b e s c h r e i b u n g ist ein Tupel d=(part_id, module_type) einschließlich der zugehörigen Liste **oplist**(module_type).
   p a r t _ i d ist der unter PARTS definierte Identifikator eines Hardware-Bausteins.
   m o d u l _ t y p e ist der zugehörige Name des Modul - Typs.

8. Die Menge aller Bausteinbeschreibungen heißt **parts**.

9. Für alle Bausteinbeschreibungen d ∈ **parts** sind die Funktionen **part_id**(d) und **module_type**(d) definiert als Projektionen auf die jeweiligen Komponenten.

10. Für d ∈ **parts** ist **oplist**(d) definiert durch **oplist**(**module_type**(d)). (Die Funktion **oplist** ist -abhängig vom Argumenttyp- überdefiniert).

11. Ein B i t t u p e l ist ein Tupel s=(part_id, port_id, subport_id, bits).
    p a r t _ i d ist ein Identifikator eines Bausteins.
    p o r t _ i d ist der Identifikator eines für **module_type**(part_id) vereinbarten Ports. Ist kein Port vereinbart, so ist diese Komponente leer (Ø).
    s u b p o r t _ i d ist der Name eines für das Port vereinbarten Subport-Identifikators. Ist kein Subport vereinbart, so ist diese Komponente leer (Ø).
    b i t s ist ein Tupel natürlicher Zahlen, den sog. B i t n u m m e r n.

Beispiel :

(SR,P2,f,(5,3,1)) ist ein Bittupel. Dieses Tupel entspricht der MIMOLA-Schreibweise SR-P2.f.(5,3,1).

12. Ein B i t b e r e i c h ist ein Bittupel, dessen Bitnummern von links nach rechts gelesen im Bereich der natürlichen Zahlen eine lückenlose absteigende Folge bilden.

Beispiel :

SR-P2.f.(2,1,0) ist ein Bitbereich in MIMOLA-Notation.
SR-P2.f.(2:0) ist eine dazu äquivalente Schreibweise.

Zu jedem Subport eines Bausteins gehört eine Menge in der Deklaration vorkommender Bitnummern.

13. Ein Bitbereich heißt m a x i m a l , falls man diesen Bereich um keine für das betreffende Subport deklarierte Bitnummer erweitern kann.

Beispiel :

Sei SR beschrieben wie in Abschnitt 2.1.1. Dann ist SR-P2.f.(15:0) die MIMOLA-Schreibweise eines maximalen Bitbereichs.

14. Die Menge der im INSTRUCTION-Teil erklärten Befehlsfelder heißt **ifields**. Elemente i $\in$ **ifields** können als Bittupel dargestellt werden. Dabei wird der Feldname für subport_id benutzt.

15. Für alle i $\in$ **ifields** ist **length**(i) die Länge des Feldes in Bits.

16. Die Menge aller Bitbereiche heißt S.

Weitere Beispiele für Bitbereiche:

In MIMOLA	T u p e l - N o t a t i o n			
	part_id	port_id	subport_id	range_id
SR-P2.f.(1:0)	SR	P2	f	1:0
ALU.a.(15:0)	ALU	Ø	a	15:0
I.regb	I	Ø	regb	2:0
ZERO.(0)	ZERO	Ø	Ø	0

Def.:

17. S_a ist die Menge aller Bitbereiche, deren Subport mit OUT oder INOUT qualifiziert ist. S_a ist die Menge der A u s g a n g s - b i t b e r e i c h e.

18. S_e ist die Menge aller Bitbereiche, deren Subport mit IN, INOUT, ADR, FCT oder CLK qualifiziert ist. S_e ist die Menge der E i n g a n g s b i t b e r e i c h e.

19. Elemente des Kreuzproduktes S_a X S_e heißen Verbindungen. Die Menge der in einer bestimmten Hardware h enthaltenen Verbindungen wird mit **connections**(h) oder kurz **connections** bezeichnet.

20. Für alle c ∈ **connections** ist **source**(c) und **sink**(c) definiert als Projektion auf die erste bzw. zweite Komponente von c.

21. **templocs** ist die Menge der als LOCATIONS_FOR_TEMPORARIES erklärten Zellen. Elemente aus **templocs** sind Tupel (part_id, address) mit
p a r t _ i d ist ein Element aus **parts**,
a d d r e s s ist eine natürliche Zahl.

22. Für l ∈ **templocs** ist **part_id**(l) die Projektion auf die erste Komponente.

23. Eine H a r d w a r e b e s c h r e i b u n g ist ein Tupel (**modules**, **parts**, **ifields**, **connections**, **templocs**) einschließlich der zugehörigen Listen von Operationsbeschreibungen.

Für die Hardware nach Abb. 2.2 gilt folgende Tupel-Darstellung ("F%0" bzw. "W%15" sind die intern generierten Modultyp-Namen der Konstanten "ZERO" und des Busses "BBUS"):

```
h =
 (
   {Storage, B74S381, F%0, W%15},          "modules"
   {(SR,Storage),                          "parts"
    (ALU0,B74S381),
```

```
    (ZERO,F%0),
    (BBUS,W%15)},
   {(I,Ø,opr  ,10:8),                          "ifields"
    (I,Ø,enable,7:6),
    (I,Ø,reg2  ,5:3),
    (I,Ø,reg1  ,2:0)},
   {((I,Ø,reg1,2:0)), (SR,P1,ad,2:0)),         "connections"
    ((I,Ø,reg2,5:3)), (SR,P2,ad,2:0)),
     ..............................    (weitere Verbindungen)
    ((ALUØ,Ø,f,15:0)),(SR,P2,e,15:0))},
    Ø}                                         "templocs"
 )

oplist(ALUØ) = oplist(B74S381) = oplist((ALUØ,B74S381)) =
 (
  (%0..0,0,16,%000,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (-    ,2,16,%001,{(Ø,b,15:0),(Ø,a,15:0)},(Ø,f,15:0),Ø),
  (-    ,2,16,%010,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (+    ,2,16,%011,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (XOR  ,2,16,%100,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (OR   ,2,16,%101,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (AND  ,2,16,%110,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø),
  (%1..1,0,16,%111,{(Ø,a,15:0),(Ø,b,15:0)},(Ø,f,15:0),Ø)
 )

oplist(SR) = oplist(Storage) = oplist((SR,Storage)) =
 (
  (LOAD    ,2,16,%X0,{(P2,e,15:0),(P2,ad,2:0)},     Ø,Ø),
  (NOLOAD  ,0,16,%X1,Ø                    ,          Ø,Ø),
  (READ    ,1,16,%0X,{(P2,ad,2:0)}       ,(P2,f,15:0),Ø),
  (TRISTATE,0,16,%1X,Ø                    ,(P2,f,15:0),Ø),
  (READ    ,1,16,%0 ,{(P2,ad,2:0)}       ,(P1,f,15:0),Ø),
  (TRISTATE,0,16,%1 ,Ø                    ,(P1,f,15:0),Ø)
 )
```

```
oplist(ZERO) = oplist(F%0) = oplist((ZERO,F%0)) =
 (
  (%0       ,0,1 ,%X ,Ø                    ,(Ø ,Ø, 0:0),Ø)
 )

oplist(BBUS) = oplist(W%15) = oplist((BBUS,W%15)) =
 (
  (DAT      ,1,16,%X ,(Ø,Ø,15:0)           ,(Ø ,Ø,15:0),Ø)
 )
```

DAT ist das Operationssymbol für die identische Abbildung eines Eingangs auf den Ausgang.

Abschließend sei noch bemerkt, daß im Rahmen dieser Arbeit nur streng synchron arbeitende Hardwarestrukturen betrachtet werden. Das bedeutet, daß alle Zustandsübergänge durch einen (und nur einen) Takt ausgelöst werden. An der Erweiterung auf gekoppelte Systeme wird z.Zt. gearbeitet. Einen möglichen Weg zur hardwaremäßigen Realisierung der Systemsynchronisation beschreiben Anantharaman et al. [AnaClaFosMis85].

2.2 Programme

Neben der Hardware enthält eine MIMOLA-Beschreibung optional auch ein Programm. Aus Gründen der Akzeptanz ist es vorteilhaft, zur Darstellung von Programmen eine einfach zu lernende, weit verbreitete Programmiersprache zu benutzen.

Folglich wurden zwei ältere Definitionen von MIMOLA (vgl. [Zim77, MarZim79]) stark an PASCAL angeglichen. Wesentliche Unterschiede zwischen MIMOLA und PASCAL im Bereich der Syntax von Programmen betreffen vor allem Möglichkeiten zur Spezifikation paralleler Blöcke und von maschinennahen Programmen.

2.2.1 Blöcke

Die folgenden Diagramme geben den für Blöcke relevanten Teil der Programmsyntax wieder.

program :.=

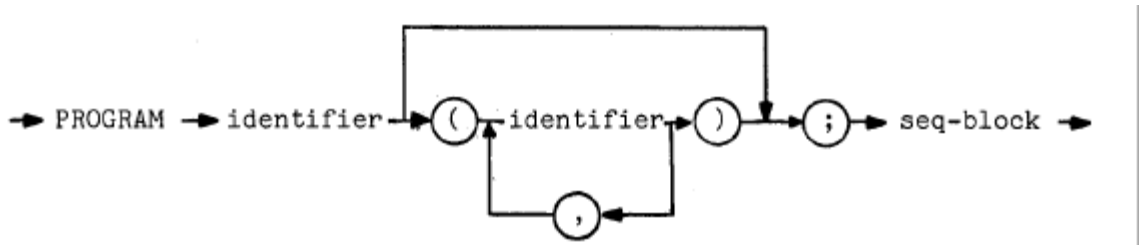


Abb. 2.4 Syntax von Programmen

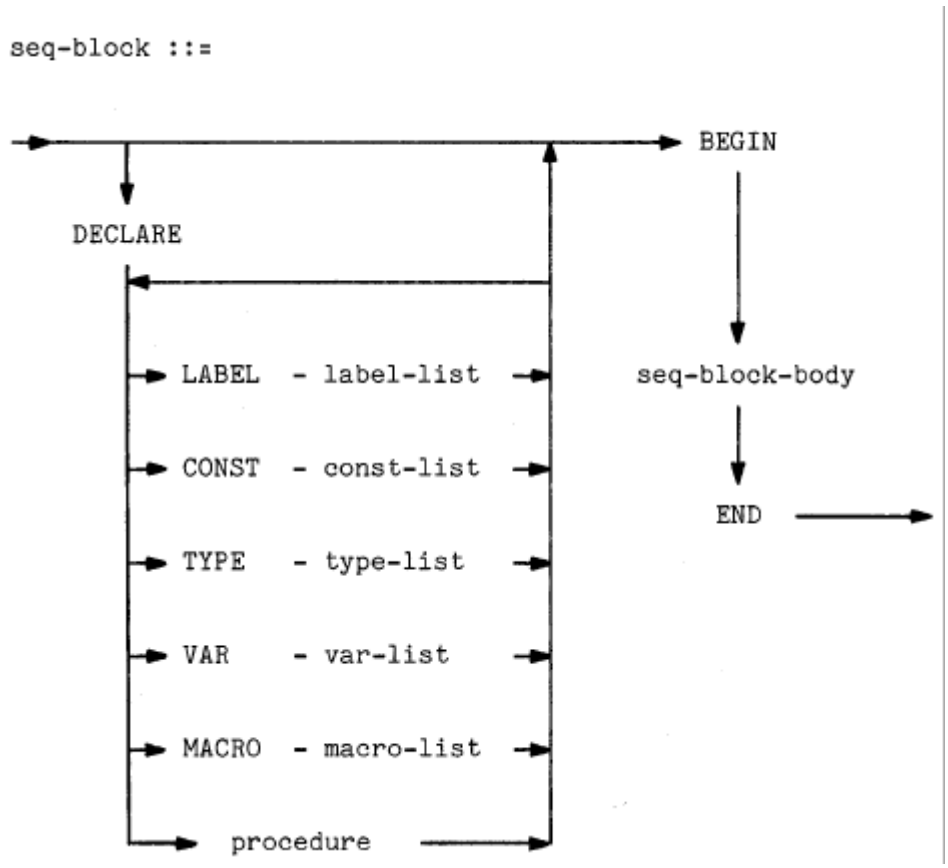


Abb. 2.5 Syntax von sequentiellen Blöcken

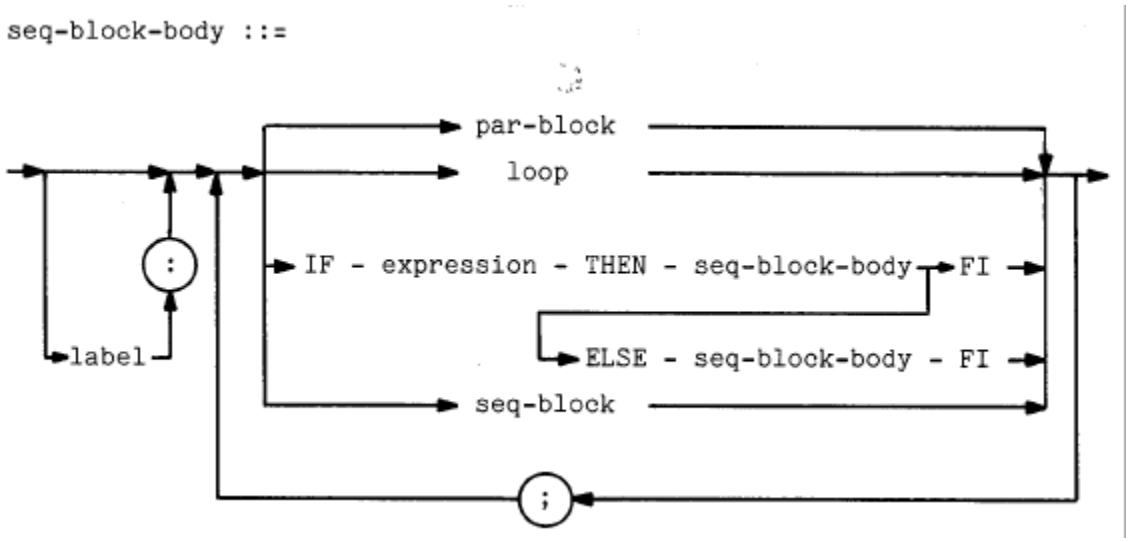


Abb. 2.6 Syntax des Rumpfes sequentieller Blöcke

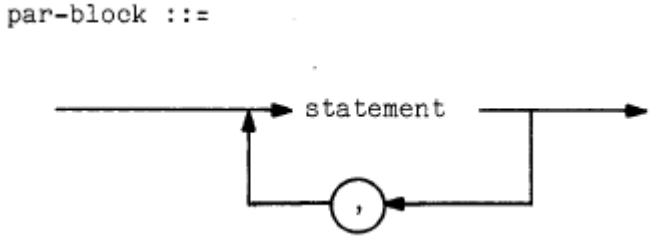


Abb. 2.7 Syntax von parallelen Blöcken

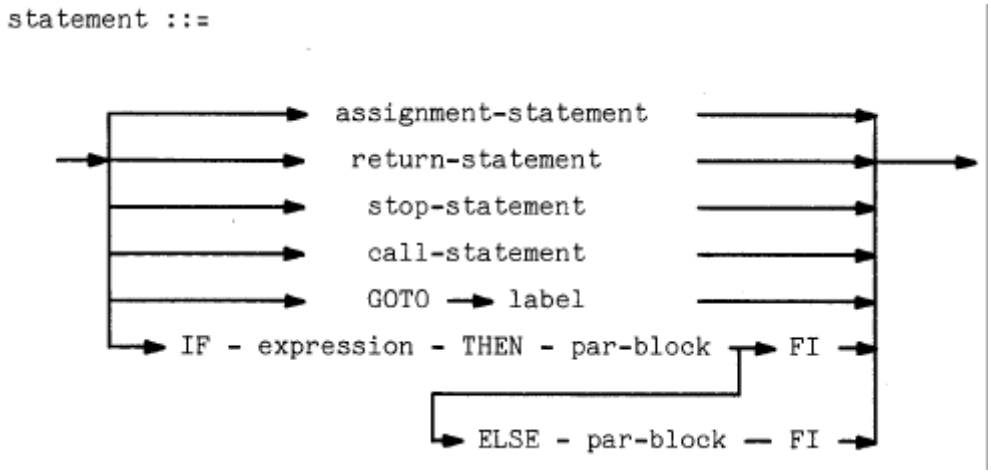


Abb. 2.8 Syntax von Anweisungen

In parallelen Blöcken können Anweisungen zusammengefaßt werden, die potentiell parallel ausgeführt werden können (die tatsächliche Art der Ausführung hängt von der zur Verfügung stehenden Hardware ab).

Beispiel:

```
PROGRAM p;
DECLARE
 a,b,c,d : integer;
 BEGIN
  a:=7,       b:=3;          (*1. paralleler Block*)
  c:=a "*" b, d:=a "/" b;    (*2. paralleler Block*)
 END.
```

Operatoren werden in MIMOLA in " " eingeschlossen, um die Definition neuer Operatoren durch den Benutzer zu vereinfachen.

Ergebnisse der Mikroprogrammierung belegen, daß durch parallele Ausführung mehrerer Anweisungen Rechenleistungen gegenüber vergleichbar komplexen Rechnern gesteigert werden können (vgl. z.B. [Mar80, NicFis84]). Daher soll das Entwurfssystem auch die Behandlung von Rechnern mit paralleler Ausführung von Anweisungen erlauben. Um diese Parallelität explizit ausdrücken zu können, erlaubt MIMOLA die Benutzung paralleler Blöcke. Die Einführung von parallelen Blöcken ist auch eine der Voraussetzungen dafür, daß MIMOLA-Programme nach einer Parallelisierung wieder als MIMOLA-Programme darstellbar sind. Diese Abgeschlossenheits-Eigenschaft vereinfacht die Konstruktion des MSS. Die automatische Erkennung von parallel ausführbaren Anweisungen einer PASCAL ähnlichen sequentiellen Sprache würde nicht ausreichen.

Falls innerhalb eines parallelen Blockes die gleichen Speicherzellen sowohl auf der linken als auch auf der rechten Seite von Zuweisungen vorkommen, erhebt sich die Frage nach der Semantik der Zuweisungen. In ISPS [BarBarCatSie77] ist eine beliebige Ausführungsreihenfolge der Zuweisungen erlaubt und die Semantik folglich undefiniert. Dies führt zu einer Reihe von Problemen bei der Darstellung streng synchron arbeitender Maschinen [DamBarHalJoo81, S. 235]. Für MIMOLA wird daher eine diesen Maschinen angemessene Bedeutung vereinbart: Konzeptionell

werden zunächst alle rechten Seiten der in einem parallelen Block enthaltenen Zuweisungen berechnet, bevor die erste Zuweisung erfolgt. Mit dieser Festlegung spezifiziert

a: =b, b: =a

stets eine Vertauschung. In Sprachen wie ISPS muß zur Spezifikation dieser Semantik eine in der Hardware nicht vorkommende Hilfszelle eingeführt werden.

Diese Festlegung der Semantik erleichtert die Beschreibung streng synchron arbeitender Hardware. Diesem Vorteil für den Benutzer steht allerdings ein nicht unerheblicher Aufwand in der Implementierung gegenüber.

Wegen der entstehenden semantischen Probleme sind Blöcke, die mehrere Zuweisungen zur gleichen Speicherzelle enthalten, nicht erlaubt. Insbesondere darf ein paralleler Block nur eine Zuweisung an den Programmzähler (einschl. GOTO-,CALL-und RE TURN-Statements) enthalten.

2.2.2 Maschinennahe Programme

Das Studium von Hardware/Software-Tradeoffs ist nur möglich, wenn die als Entwurfsspezifikation benutzten Programme einen engen Bezug zur Hardware haben. Insbesondere müssen **alle von der Hardware durchzuführenden Operationen explizit in den Programmen vorkommen**. Zu diesen Operationen gehören u.a. sämtliche Adreßrechnungen und alle Leseund Schreiboperationen. Das ursprüngliche MIMOLA-Programm, das in der Regel auf einem PASCAL-artigen Niveau geschrieben ist, erfüllt diese Anforderungen nicht. Es muß vielmehr auf die RT-Verhaltensebene abgebildet werden (diese Abbildung entspricht weitgehend der Abbildung von CAT nach CAL bei Schmidt [Sch84]). Es muß noch unterschieden werden zwischen verschiedenen Arten von RT-Programmen:

Def.:

Die Eingabeprogramme der Synthese werden als **ungebundene RT Programme** bezeichnet, da die in diesen Programmen vorkommenden arithmetisch/logischen steine gebunden sind. In ungebundenen Programmen sind Lese- und Schreiboperationen bestimmten Speichern, nicht jedoch bestimmten

z. **Gebundene RT-Programme** dagegen enthalten für jede arithmetisch/ logische Operation den Baustein, der diese Operation ausführt und für jede Lese- oder Schreiboperation das zugehörige Port.

3. Werden in gebundenen Programmen auch solche Bausteine aufgeführt, die identische Abbildungen ausführen (Multiplexer, Bus-Treiber, Busse), so heißen diese Programme **vollständig gebundene Programme.**

Man könnte jetzt für jede Form der Programme eigene Sprachen definieren. Ähnlich wie bei Bauer [Bau78] wird im MSS aber aus Flexibilitätsgründen eine Sprache benutzt, die alle Ebenen abdeckt, eine "Breitbandsprache". Dadurch sind die Programme der Sprache gegenüber den in Frage kommenden Transformationen abgeschlossen.

In den Beispielen werden folgende Bezeichnungen benutzt:

```
SR(..)          : Zugriff auf Registerspeicher  \ ≙ cont(..)
SH(..)          : Zugriff auf Hauptspeicher     / in ALGOL68
SR-P1,SR-P2     : Bezeichnung für Ports des Speichers SR
SH-P1,SH-P2     : Bezeichnung für Ports des Speichers SH
I(wert).(x:y)   : In den Bits y bis x des gegenwärtigen
                  Befehls enthaltener Wert
```

Das nächste Beispiel veranschaulicht die unterschiedlichen Programmierebenen anhand der eingeführten Bezeichnungen:

algorithmische Ebene

```
q := p
```

ungebundenes RT-Programm (SR(2), SR(3) sind Basisregister):

```
SH(SR(2) "+" 2) := SH(SR(3) "+" 7)
```

gebundenes RT-Programm:

(I(3): Im Befehlsfeld enthaltener Auswahlcode für "+";
ALU : arithmetisch/log. Einheit, siehe Abschn. 2.1.1;
Kontrolleingänge der Speicher ausgelassen):

```
SH-P1(ALU0(SR-P1(I(2).(2:0)), I(2).(31:16), I(3).(10:8))):=
 SH-P2(ALU1(SR-P2(I(3).(5:3)), I(7).(47:32), I(3).(13:11)))
```

vollständig gebundenes RT-Programm:

(zusätzlich alle Multiplexer, Busse und den Kontrolleingängen der Speicher zugeführte Ausdrücke)

Die Abbildung von der algorithmischen Ebene auf ungebundene RT-Programme im MSS schließt die Ersetzung von Variablen ein. Hierfür gibt es u.a. folgende Möglichkeiten:

1.Explizite Zuordnung

Es ist möglich, den Variablen in der Variablendeklaration Speicherzellen zuzuordnen.

Beispiel:

```
VAR q : SH(5).integer;
    t : SH(SR(1) "+" 2).integer;
```

2.Implizite Zuordnung

Wird in der Variablendeklaration lediglich der Typ der Variablen angegeben, so müssen in der Hardwarebeschreibung für die Zuordnung von Variablen geeignete Speicherbereiche durch LOCATIONS-Definitionen erklärt werden. Die Zuordnung wird dann vom MSS vorgenommen.

Neben den Variablen sind auch die in MIMOLA erlaubten höheren Kontrollstrukturen (wie z.B. Unterprogramm-Aufrufe und FOR-Schleifen) vor der eigentlichen Synthese durch einen Satz von Operationen auf der RT-Ebene zu ersetzen. Diese Ersetzungen oder **Programmtransformationen** können ebenfalls in MIMOLA beschrieben werden.

Beispiele:

```
REPLACE
  GOTO &lab
WITH
  RP:= &lab
END,
REPLACE
  &lab : WHILE &expr DO &block
WITH
  &lab : IF &expr THEN BEGIN &block; GOTO &lab END FI
END,
```

In diesem Beispiel stellen Bezeichner, welche &-Zeichen enthalten, formale Parameter dar. Sie passen auf beliebige, in dem betreffenden Zusammenhang mögliche MIMOLA-Sprachkonstrukte. Im Falle von &lab sind dies gerade die Label. Die erste Regel führt also zum Ersetzen aller Vorkommen der Sequenz bestehend aus "GOTO" und irgendeinem Label durch die Zuweisung des Labels an den Programmzähler.

In der zweiten Regel paßt &expr auf beliebige Ausdrücke und &block auf beliebige Blöcke. Diese Regel führt zum Ersetzen aller WHILE-Blöcke durch bedingte Anweisungen.

Im speziellen Fall des Befehlswortes I wurde bereits von der Möglichkeit zur Angabe von Bereichen von Bits Gebrauch gemacht. Dies ist häufig notwendig, um Details auf der RT-Ebene auszudrücken. Beispiele dafür sind:

```
a.(7:0) := b.(15:8);
IF a.(0) THEN b:=0 FI;
b.(15:0):=b.(7:0) ! b.(15:8)
```

Im letzten Beispiel bezeichnet ! die Aneinanderreihung (Konkatenation) der einzelnen Bitstrings. Folglich werden in diesem Beispiel gerade zwei Bytes vertauscht.

3. Systemoberteile

Der Codegenerator, der Testgenerator und das Synthesesystem des MSS verarbeiten MIMOLA - Beschreibungen nicht direkt. Vielmehr erfolgt eine Vorverarbeitung durch Komponenten des MSS. Diese Komponenten werden hier unter dem Begriff Systemoberteile zusammengefaßt.

Außerdem kann noch eine Nachverarbeitung durch die in Kap.6 beschriebenen sog. Systemunterteile erfolgen.

Abb.3.1 gibt eine grobe Ubersicht über die Struktur des MSS. Anhang D enthält ein detaillierteres Bild der einzelnen Systemkomponenten.

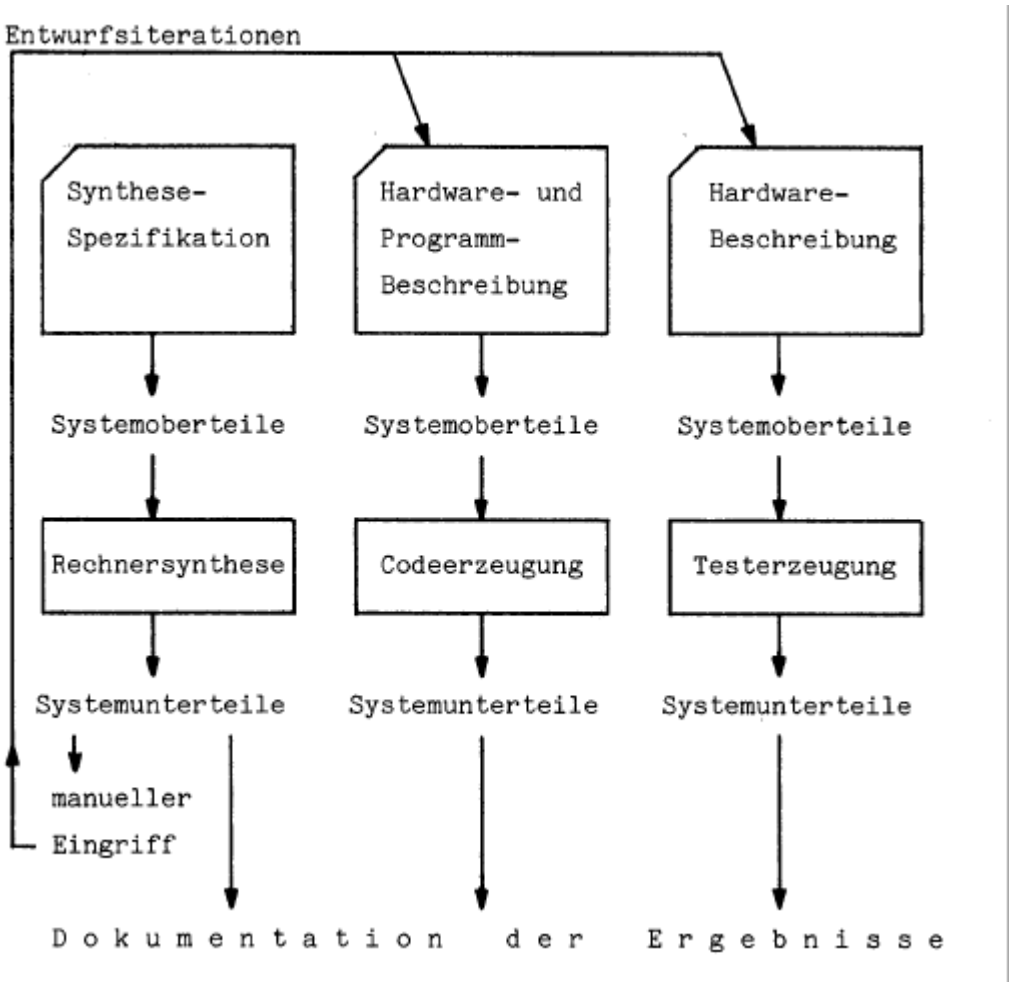


Abb. 3.1 Grobe Übersicht über das MIMOLA-Software-System

Alle Teile des Systems sind in PASCAL geschrieben. Zwecks Reduktion softwaretechnologischer Probleme wir„d neben dem PASCAL-Compiler ein PASCAL-Vorcompiler benutzt. Dadurch wird u.a. eine maschinenunabhängige modulare Programmierung möglich. Einige der benutzten Moduln realisieren abstrakte Datentypen wie z.B. variabel lange Strings und beliebig große Mengen natürlicher Zahlen.

Der Vorcompiler generiert Standard-PASCAL Programme gemäß [ANSI/IEEE83]. Dadurch ist das System hochgradig portabel. Probleme bei der Übertragung auf andere Wirtsrechner wurden so weit eliminiert, daß inzwischen die reinen Übersetzungszeiten bei einer Portierung dominieren. Es existieren Installationen auf Rechenanlagen der Firmen Siemens, DEC, Apollo und Data General.

Abb. 3.2 gibt einen Überblick über die Systemoberteile:

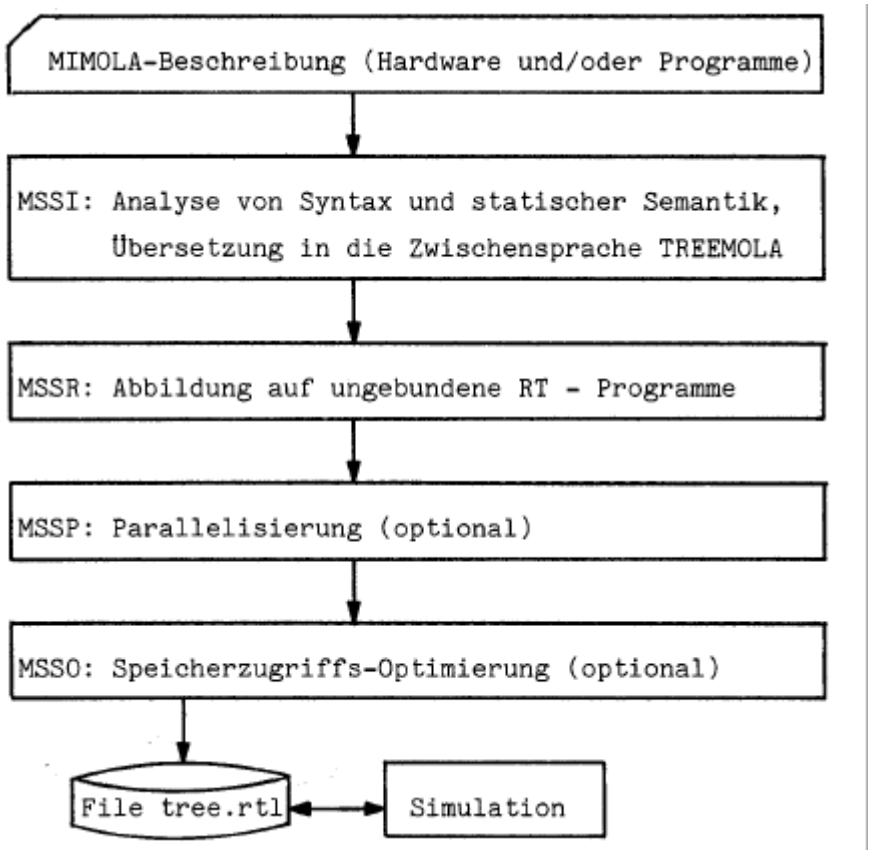


Abb. 3.2 Übersicht über Systemoberteile

3.1 Übersetzung in die Zwischensprache TREEMOLA

Die Komponente MSSI des MSS prüft Syntax und statische Semantik der eingegebenen MIMOLA-Beschreibung und übersetzt sie in die interne Zwischensprache TREEMOLA (**tree** microoperation language, [Jöh81]). Zur syntaktischen Analyse wird in neueren Versionen des MSS das recursivedescent-Verfahren (vgl. z.B. [Wir77]) benutzt. Die statische Semantik, d.h. die Menge der vom Compiler zu prüfenden Eigenschaften einer MIMOLA-Beschreibung, wird durch den MIMOLA-Sprachreport [JöhMar] definiert.

Die Zwischensprache TREEMOLA ist eine baumartige Sprache. Im Bereich von Ausdrücken entspricht diese Sprache abstrakten Syntaxbäumen. Diese Bäume kennzeichnen den Fluß der Daten innerhalb einer Anweisung und werden daher im folgenden als D a t e n f l u ß b ä u m e bezeichnet. Die Bäume könnten auch als Kantorovic-Bäume [BauWös81] bezeichnet werden. Der verwandte Begriff "Strukturbaum" wird u.a. deswegen hier nicht benutzt, weil man ihn mit Hardwarestrukturen verwechseln könnte.

Zu ihrer externen Darstellung auf Textfiles wird eine Klammernotation benutzt. Das jeweils erste Zeichen eines Knotens bezeichnet den Knotentyp.

Beispiel:

Abb. 3.3 zeigt die interne und externe Darstellung der Zuweisung SH(0):=a:

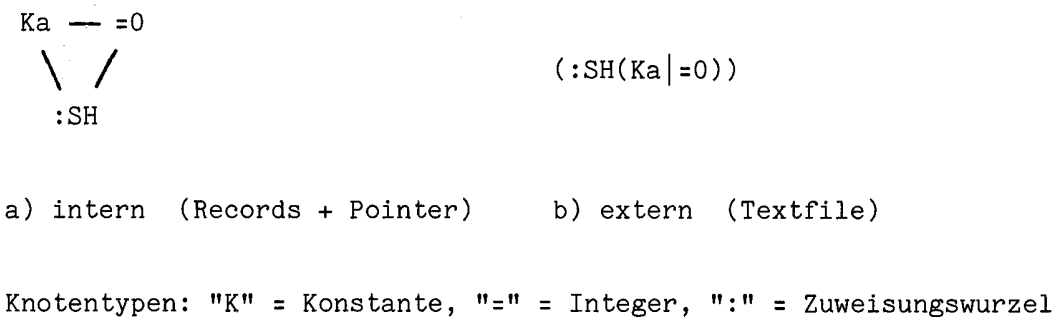


Abb. 3.3 TREEMOLA-Darstellungen

Parallele und sequentielle Blöcke werden durch die Knotentypen
und „„charakterisiert.,

Beispiel:

Label: SH(0):=a, SH(1):=b

wird auf Textfiles dargestellt als

(,Label(:SH(Ka|=0)|:SH(Kb|=1)))

Konstanten-, Variablen-, Typ- und Programmtransformationsvereinbarungen werden in TREEMOLA einheitlich als Programmtransformationsregeln dargestellt.

Beispiel:

CONST a = 7;

wird dargestellt als

(> (Ka|=7)), d.h ersetze "Ka" durch "=7".

Um Operationen auf TREEMOLA-Bäumen auszudrücken, werden die folgenden Definitionen benötigt:

Def.:

1. Ein Baum ist ein endlicher, schleifenfreier, zusammenhängender Graph b=(Q,E,w) mit:
   Q : Menge der Knoten,
   E $\subseteq$ Q X Q : Menge der Kanten,
   w $\in$ Q : die Wurzel (ein ausgezeichneter Knoten).

2. Für alle Bäume b bezeichnen
   nodes(b) : die Menge der Knoten,
   vert (b) : die Menge der Kanten,
   root (b) : die Wurzel von b.

3. Sei b ein Baum und n ϵ **nodes**(b).
   Dann bezeichnet **tree**(n,b) den Teilbaum von b mit der Wurzel n. Da alle im folgenden betrachteten Knoten die Knoten eines aus dem Zusammenhang bekannten Baumes sind, werden wir die Angabe des umgebenden Baumes b von nun an auslassen.

4. Sei b ein Baum. Für alle n ϵ **nodes**(b) ist **sons**(n) definiert durch
   sons(n):= {n' | $\exists$ (n',n) ϵ **vert**(**tree**(n)) }

3.2 Erzeugung von RTL-TREEMOLA

Von MSSI erzeugte TREEMOLA-Files enthalten noch Hochsprachenelemente wie z.B. Unterprogramm-Aufrufe. Die in Abschnitt 2.2.2 erwähnte Ersetzung von Hochsprachen-Elementen durch RT-Sprachelemente erfolgt durch die Komponente MSSR.

Von MSSR erzeugte TREEMOLA-Files enthalten (abgesehen von STOP und CALL PASCAL, s.u.) nur noch RT-Elemente. Die entsprechend eingeschränkte Form von TREEMOLA heißt RTL - TREEMOLA. RTL-TREEMOLA-Programme sind Programme auf der RT-Verhaltensebene. Alle Komponenten des MSS mit Ausnahme von MSSI, MSSR und MSSM akzeptieren ausschließlich RTL-TREEMOLA als Eingabe. Die folgende Liste zeigt Beispiele typischer RTL-TREEMOLA Knoten und mögliche Inhalte:

Knotentyp	Möglicher Inhalt
integertype	ganze Zahl, **range_id**
operationtype	**op_symbol**, **range_id**
sourcetype	**part_id**, **port_id**, **range_id**
assignmenttype	**part_id**, **port_id**, **range_id**
partype	label (*Label gemäß MIMOLA Syntax*)

Anhang A enthält eine vollständige Syntax der RTL-TREEMOLA-Bäume.

Def.:

1. Die Menge aller RTL-TREEMOLA Knoten gemäß Anhang A heißt **nodeclass**.

2. Die Menge aller gemäß Anhang A möglichen RTL-TREEMOLA Teilbäume heißt **treeclass**.

3. Für n ∈ **nodeclass** bezeichnet **nodetype**(n) den Typ des Knotens n.

4. Für n ∈ **nodeclass** bezeichnen **part_id**(n), **port_id**(n), **var_id**(n) und **range_id**(n) die entsprechenden Knoteninhalte, sofern diese gemäß Anhang A für den jeweiligen Knotentyp definiert sind; sonst ist das Ergebnis undefiniert.

5. Ein paralleler Block pb ist ein RTL-TREEMOLA Baum mit **nodetype**(**root**(pb))=partype

6. Eine Zuweisung s ist ein RTL-TREEMOLA Baum mit **nodetype**(**root**(s))=assignmenttype

7. Ein Program ist eine dem syntaktischen Symbol program des Anhangs A entsprechende Menge von RTL-TREEMOLA Bäumen.

Die Abbildung auf die RT-Ebene erfolgt durch Konstanten-, Typen-, Variablen- und REPLACE- Transformationsregeln. REPLACE-Transformationsregeln werden im MSS für eine Reihe unterschiedlicher Aufgaben eingesetzt:

a. Ersetzung von höheren Sprachelementen

Beispiele:

```
REPLACE   goto &next   WITH   RP:=&next  END,
```

```
REPLACE
 call &id
WITH
 SH(stackp):= "INCR" RP,
 RP:=&id,
 stackp:=stackp "+" 1
END
```

Eine Reihe vordefinierter Prädikate und anderer Funktionen erlaubt es, Ersetzungen von bestimmten Eigenschaften der Parameter abhängig zu machen. So kann beispielsweise getestet werden, ob ein Parameter eine Konstante als aktuellen Wert zugewiesen bekommen hat und ob für einen Prozedurparameter CALL-BY-VALUE oder CALL-BY-REFERENCE Übergabe verlangt wurde.

Beispiel:

```
REPLACE
 call &id (&par)
WITH
 if Misrefvar (&par) then SH(stackp):=REF(&par)
                     else SH(stackp):=&par
 fi,
 SH(stackp "+" 1) := "INCR" RP, RP := &id,
 stackp := stackp "+" 2
END
```

Durch rekursive Anwendung von Transformationsregeln ist es möglich, Prozeduraufrufe mit beliebig vielen Parametern und Zugriffe auf Arrays mit beliebig vielen Indices zu ersetzen.

b. Ersetzung nicht implementierter Operationen durch implementierte:

Beispiel:

```
REPLACE &a "=" &b WITH "=0" ( &a "-" &b) END
```

"=0" ist eine 1-stellige Operation in MIMOLA

c. Optimierungen

Beispiel:

```
REPLACE   (&a "+" 0)  WITH   &a   END
```

Eine Menge solcher Optimierungsregeln ist standardmäßig vordefiniert.

Aufgrund der gewonnenen Erfahrungen hat sich die Möglichkeit, Trans formationsregeln explizit zu definieren, bewährt. Gegenüber einer festen Programmierung von Transformationsregeln ist dieser Ansatz flexibler. Auf diese Weise war es möglich, in den Anwendungen des MSS eine Reihe von Spezialfällen zu behandeln, die nicht vorhergesehen waren.

Als weitere Optimierung enthält die Komponente MSSR des MSS die Berechnung der Werte von arithmetischen Ausdrücken mit konstanten Argumenten ("constant-folding").

Auf der RT-Verhaltensebene muß für alle in Zuweisungen vorkommenden Unterausdrücke die zur Darstellung benötigte Zahl von Bits bekannt sein. Zwecks Erzeugung von Fehlermeldungen ist es weiterhin wichtig

zu wissen, welchen Wertebereich Unterausdrücke annehmen können. Die Kombination dieser Informationen wird im folgenden als **Datentyp** bezeichnet. Die Berechnung des Datentyps erfolgt in zwei Durchläufen durch die Datenflußbäume. Im ersten Durchlauf von den Blättern zur Wurzel erfolgt die Berechnung des Wertebereichs sowie der minimalen Zahl benötigter Bits. Im zweiten Durchlauf von der Wurzel zu den Blättern erfolgt eine Typanpassung der Argumente von Operationen untereinander und an den Typ des Ergebnisses. Sind Anpassungen trotz vom Benutzer angegebener Längen erforderlich, so werden Warnungen generiert. Anpassungen können entweder mit Nullen oder dem Vorzeichenbit erfolgen. Die Zahl der benötigten Bits wird den Knoten der Bäume als Komponente **range_ id** (vgl. Anhang A) zugefügt. Sie steht damit allen folgenden Teilen des MSS zur Verfügung.

Im Interesse u.a. der Synthese möglichst einfacher Hardware-Bausteine wird stets die kleinste mögliche Länge von Bitstrings gewählt. Auf diese Weise können z.B. 24-Bit Adreßaddierer generiert werden, obwohl die für Integer benutzte Wortbreite 32 Bit beträgt. Dies wäre mit dem bei PASCAL verwendeten Ansatz, alle Zwischenrechnungen stets mit der vollen Wortbreite durchzuführen, nicht möglich.

3.3 Parallelisierung

Als Zielarchitekturen sollen Rechner erzeugt werden können, deren Befehlsstruktur dem horizontal mikroprogrammierter Rechner entspricht. Die Erzeugung solcher Rechner aus sequentiellen Programmen wird vereinfacht, wenn die Programme vor der eigentlichen Synthese parallelisiert werden. Für das MSS wurden Parallelisierungsverfahren von Konow [Kon83] und von Berger entwickelt.

Das folgende Beispiel gibt die Wirkung der Parallelisierung auf algorithmischer Ebene wieder:

Aus

```
L0: s:=s "+" a[i]; i:=i "+" 1; IF   i  "<"  max   THEN GOTO L0 FI
```

wird

```
L0: s:=s "+" a[i], i:=i "+" 1, IF (i"+"1) "<" max THEN GOTO L0 FI
```

Die beiden Semikolons der ersten Form bewirken, daß mindestens drei Befehlsschritte zur Ausführung der gesamten Zeile benötigt werden. Dagegen kann das MSS für die zweite Form Maschinencode für die Ausführung innerhalb eines Befehlsschrittes erzeugen, da die Kommata eine mögliche Parallelausführung angeben. Die Ausführung wird aber nur bei ausreichenden Hardware-Ressourcen tatsächlich einschrittig erfolgen.

Durch den Übergang auf einen parallelen Block besitzt die Referenz der Variablen i im dritten Statement den gleichen Wert wie die lesende Referenz im zweiten Statement. Zur Wahrung der Semantik müssen Referenzen auf i im dritten Statement daher während der Parallelisierung durch die rechte Seite des zweiten Statements ersetzt werden ("statement substitution" [Kuc78]). Die Zahl der Operationen, die die Hardware auszuführen hat, wird dadurch nicht erhöht, wenn bei der Erzeugung von Maschinencode gemeinsame Teilausdrücke erkannt werden.

Ähnliche Transformationen zwischen und sequentiellen parallelen

3.4 Optimierung der Speicherzugriffe

Die Laufzeit von Programmen läßt sich im allgemeinen beträchtlich verkürzen, wenn einige der Variablen in schnellen Registern statt im langsamen Hauptspeicher gehalten werden. Da die Zahl der schnellen Register meist nicht zur Unterbringung aller Variablen ausreicht, entsteht das Problem der optimalen Auswahl. Dieses Problem führt auf eine detaillierte Analyse des Daten- und des Kontrollflusses sowie das anschließende Lösen eines Cliquen- oder Färbeproblems für Graphen. Für das MSS wurden entsprechende Programmtransformationen von Berger [Ber85] entwickelt. Der Ansatz korrespondiert zu einem ähnlichen Ansatz von Kim und Tan [KimTan79].

3.5 Simulation

Eine Vielzahl von Anwendungen hat gezeigt, daß auf die Simulation von MIMOLA-Programmen nicht verzichtet werden kann, wenn man sich von deren einwandfreier Funktion überzeugen will. Daneben kann die Simulation bei der Messung von dynamischen Ausführungshäufigkeiten und bei Laufzeitabschätzungen einen wichtigen Zweck erfüllen.

Um den Simulator unabhängig von der Maschine zu halten, auf der das MIMOLA-System läuft (dem sog. Wirtsrechner), muß der Simulator wie das übrige MSS in PASCAL geschrieben sein. Hierfür gibt es prinzipiell zwei Möglichkeiten, nämlich einen in PASCAL geschriebenen Interpreter von TREEMOLA und die Übersetzung von TREEMOLA nach PASCAL. Im MSS wurde aus Gründen der Simulationsgeschwindigkeit und der günstigeren Speicherplatz-Dimensionierung letzterer Ansatz gewählt.

Zur Programmierung von Simulator-Ein/Ausgaben ist es möglich, Ein/ Ausgabe-Leitungen der Rechnerstruktur explizit anzusprechen.

Um die Simulation zu erleichtern, sind Aufrufe fast aller PASCALStandardprozeduren und -funktionen erlaubt. Dadurch ist insbesondere PASCAL-Ein/Ausgabe möglich und die Simulation kann mit echten Daten

files durchgeführt werden. Da allerdings für Aufrufe von PASCALRoutinen keine Hardware erzeugt werden kann, sind diese Aufrufe im Zuge der Entwurfsverfeinerung zu entfernen.

Während der Simulation können die dynamischen Ausführungshäufigkeiten von Anweisungen mitgezählt werden. Das Programm MSSW dient dem Einkopieren dieser Häufigkeiten in TREEMOLA-Files.

Abb. 3.4 gibt einen Überblick über die Programme des Simulations Subsystems. Details entnehme man dem User's Guide [KrüJöhMar].

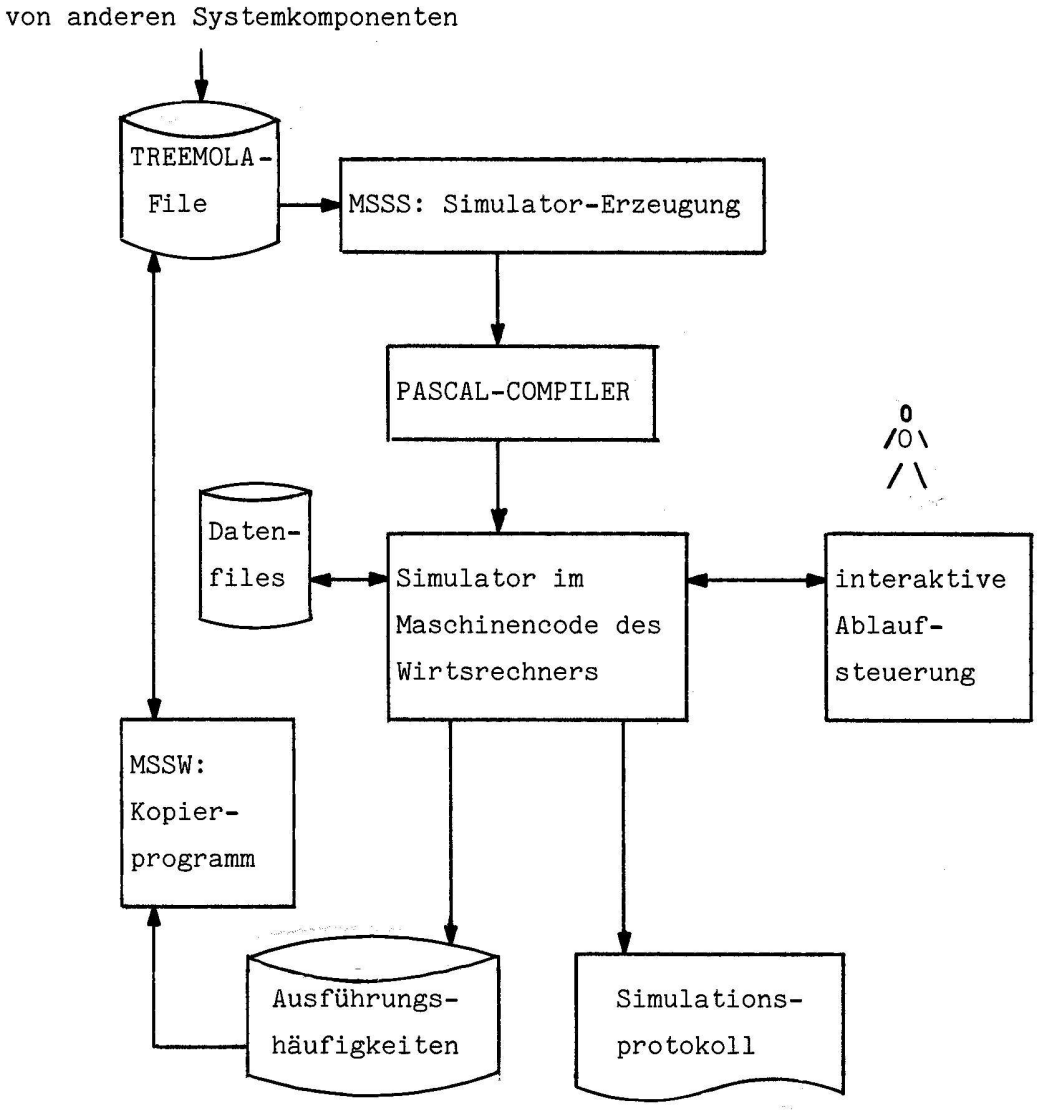


Abb. 3.4 Übersicht über das Simulations - Subsystem

4. Synthese von Rechnerstrukturen

4.1 Stand der Technik

Als Synthese bezeichnet man das Zusammensetzen von Komponenten oder Teilen zu einem Ganzen. Im naturwissenschaftlichen Bereich erfolgt die Synthese meist mit dem Ziel, ein vorgegebenes gewünschtes Verhalten zu erreichen. Besonders vorteilhaft sind dabei Methoden, mit denen konstruktiv aus einer Spezifikation des Verhaltens abgeleitet werden kann, welche Teile wie zusammenzusetzen sind. Der Vorteil solcher Methoden liegt darin, daß das zusammengesetzte Ganze bei vorausgesetzter Korrektheit des Verfahrens automatisch das vorgegebene Verhalten zeigt. Verfügbar sind derartige Verfahren z.B. für die Synthese von Filtern aus passiven und aktiven Bauelementen [Lan75]. Im ProgrammiersprachenBereich verfolgt man mit VDM (vgl. [Sch84]) das gleiche Ziel.

Syntheseverfahren für Digitalrechner kann man anhand von Schichtenmodellen wie das der Abb. 1.1 klassifizieren.

Eine Entwurfs-Spezifikation auf der **RT-Strukturebene** benutzen die Silicon-Compiler Bristle Blocks [Joh79] und MODEL von Lattice Logic [GraBucRob83]. Ein Silicon-Compiler ist ein Synthesesystem, welches eine Beschreibung auf Layout-Ebene erzeugt. Da der vollständig automatische Entwurf der Geometrie integrierter Schaltkreise z.Zt. nicht realisierbar erscheint, benutzen Silicon-Compiler eine Zwischenebene. Die Compiler erzeugen zunächst eine Beschreibung auf dieser Zwischenebene. Für die Umsetzung der Primitive dieser Ebene (z.B. Register, Addierer u.s.w.) existiert dann eine Bibliothek manuell entworfener Layouts.

Ein Ansatz von Darringer [Darjoy80] beschäftigt sich mit dem Übergang von einer Spezifikation in einer CDL ähnlichen Sprache zu einer Implementierung auf der Gatter-Ebene. CDL [Chu72] enthält Sprachelemente **beider RT-Ebenen.**

Das MacPitts-System [Sou83] ist ebenfalls ein Silicon-Compiler. Die Spezifikation besteht bei diesem System aus einer Beschreibung der

gewünschten arithmetisch/logischen Operationen und der Zuweisungen. Als "Variable" werden direkt die Namen von Hardware-Registern benutzt. Es handelt sich folglich um eine Spezifikation auf der **RT-Verhaltensebene.**

Das DSL-System [Ros82] verwendet ebenfalls eine Spezifikation auf der RT-Verhaltensebene. In den älteren Versionen dieses Systems werden relativ einfache Abbildungen zwischen RT-Verhaltens- und RT-Strukturebene benutzt [Cam85]. Dadurch reduzieren sich die Unterschiede gegenüber einer Spezifikation auf RT-Strukturebene. Ziel des DSL-Projekts scheint vorwiegend die Unterstützung von Gate-Array-Entwürfen zu sein [RosMarSch83].

Auch das japanische "Prototype Design Expert System" [Tak84] benutzt eine Spezifikation auf der RT-Verhaltensebene, synthetisiert aber eine Schaltung auf der RT-Strukturebene.

Einige Synthesesysteme, die nicht primär der Synthese von Digitalrechnern dienen, fügen sich nicht problemlos in das Schema der Abbildung 1.1 ein. Hierzu gehört insbesondere eine ganze Reihe von Systemen zur Synthese (meist bitserieller) digitaler Filter (siehe z.B. [Den82]) und zur Synthese von Hardwarestrukturen, für die das Modell der endlichen Automaten eine angemessene Beschreibung darstellt (siehe z.B. [KarTriU1183]). Da Automatentafeln u.a. "Transfers" in die Zustandsregister spezifizieren, sollen diese Tafeln hier als Spezifikation auf der RT-Verhaltensebene bezeichnet werden.

Eines der bekanntesten Systeme zum Entwurf endlicher Automaten ist das LOGE-System (siehe u.a. [GraBieHa180]). Der Schwerpunkt bei LOGE liegt in der Erzeugung von Automaten zur Steuerung von industriellen Abläufen. Als Ergebnis des Entwurfs sind unterschiedliche Realisierungen möglich, z.B. als Rechnerprogramm oder als Mikroprogramm-Steuerwerk.

Das an der Carnegie-Mellon Universität entwickelte CMU-DA-System zielt auf eine Synthese aus einer **Spezifikation der Maschinenbefehle.** Hierfür wurden unterschiedliche Synthesemethoden entwickelt (siehe z.B. [KowTho83, HitTho83]).

Im Zusammenhang mit den Arbeiten an der Carnegie-Mellon Universität entstand die Dissertation von Hafer (siehe [HafPar83]). Die Dissertation enthält eine sehr präzise Modellierung des zeitlichen Verhaltens und erlaubt neben der synchronen, parallelen Ausführung auch die asynchrone, nebenläufige Ausführung. Die Lösung der Entwurfsaufgabe erfolgt mittels gemischt-ganzzahliger linearer Optimierung.

Leider ist das Modell so komplex, daß es nur auf kleine Probleme anwendbar ist. So ergeben sich für die Modellierung von vier Zuweisungen und drei Typen verfügbarer arithmetisch/logischer Bausteine 124 Ungleichungen mit 87 Variablen. Nur für eine der möglichen Rechnerstrukturen konnte die Optimalität innerhalb weniger als 1 Stunde Rechenzeit (vermutlich VAX/780 oder PDP-20) nachgewiesen werden.

Für größere Entwurfsaufgaben dürften kaum noch Lösungen gefunden werden, da die Zahl der Gleichungen mit 0((Zahl der Operationen im Programm) 2 Zahl verfügbarer Ressourcen) wächst.

Andererseits werden Leitungskosten in diesem Modell noch nicht berücksichtigt und für die Speicherung von Variablen werden Register (und keine adressierbaren Speicher) benutzt. Das Steuerwerk wird wegen eines komplizierten Timings sehr aufwendig. In einer Arbeit von Parker [ParKurMli84] wird das Modell auf die Beschreibung von Verbindungen erweitert. Die Zahl der hierzu erforderlichen Ungleichungen wächst mit 0(Zahl der Operationen im Programm *(Zahl verfügbarer Ressourcen) 2). Um Komplexitätsprobleme bei der linearen Optimierung zu umgehen, wird von Parker vorgeschlagen, nur das Timing mittels linearer Optimierung zu behandeln und die übrige Synthese interaktiv und mittels heuristischer Methoden durchzuführen. Die Optimierung des Timings ist Gegenstand eines kürzlich erschienenen Artikels [ParPar85].

Das CMU-System wurde für die Vorgabe von Maschinenbefehlssätzen konzipiert. G. Zimmermann schlug 1976 vor ([Zim76]) von einer Spezifikation in Form von Anwenderprogrammen auf der **algorithmischen Ebene auszugehen.** Da die von Rechnern zu lösenden Probleme heutzutage auf höheren

Ebenen als der Maschinenbefehlsebene formuliert werden, entspricht diese Spezifikation stärker den Anforderungen der Anwender. Durch diese Form der Spezifikation werden nicht bereits unnötigerweise Implementierungsdetails festgelegt, und es entsteht ein größerer Entwurfsspielraum. Anwenderprogramme beschreiben weder Details der Hardwarestruktur noch des Maschinenbefehlssatzes. Sie kennzeichnen statt dessen den Typ des zu entwerfenden Prozessors. So hängen beispielsweise Zahl und Art der Arithmetikbausteine wesentlich von den im Programm vorkommenden arithmetischen Operationen ab.

Neuerdings wird dieser Ansatz häufiger verfolgt:

In einigen Fällen ([DeuNew83, Ke1851) befinden sich die Arbeiten noch in einem Anfangsstadium, sodaß über die vorgesehenen Methoden wenig bekannt ist.

Girczyc und Knight [GirKni84] verwenden eine Untermenge von ADA als Spezifikationssprache. Die Synthese basiert auf einer Ersetzung von Teilgraphen eines zum Programm äquivalenten Datenflußgraphen durch Hardwarestrukturen. Den theoretischen Hintergrund dieser Ersetzung bilden Graph-Grammatiken und Scheduling-Algorithmen. Die Auswahl der anzuwendenden Ersetzungsregeln erfolgt u.a. in Abhängigkeit von Abschätzungen für die resultierenden Hardwarekosten.

Das PLEX-System [BurChrMat83] benutzt in der Programmiersprache C beschriebene Spezifikationen. Es erzeugt Layouts von Mikroprozessoren und ist somit ein Silicon-Compiler.

Im LALSD II-System verwertet Huang [Hua81] Methoden des MSS1. Huang geht aus von einer Zerlegung des Anwenderprogramms in Sequenzen von n-Adreßbefehlen, die höchstens an ihrem Ende einen Sprung enthalten (sog. "Basisblöcken"). In einem einfachen Scheduling-Modell werden den n-Adreßbefehlen Hardware-Ressourcen zugeteilt und notwendige Verbindungen erzeugt.

Das MSS unterscheidet sich von den erwähnten Systemen auf verschiedene Weise:
Das PLEX-System überdeckt eine große Zahl von Ebenen. Möglich ist dies durch die Beschränkung auf die Generierung einer relativ festen Hardwarestruktur. Lediglich gewisse Parameter (Wortbreite, Zahl der Interruptebenen u.s.w.) können geändert werden. Das MSS soll dagegen die Erzeugung einer großen Zahl von Strukturen erlauben, auch wenn dadurch z. Zt. (noch?) keine Layouts generiert werden können.
Zum Abspeichern von Zwischenergebnissen werden bislang einzelne Register erzeugt. Das MSS erlaubt statt dessen das Halten von Zwischenergebnissen in adressierbaren Speichern. Der Vorteil dieser Speicher liegt u.a. in einer Vereinfachung des Context-Switching und dem Erzeugen einer übersichtlicheren Struktur. Eine größere Zahl in der Struktur "verteilter", schwer zugänglicher Register macht außerdem meist zusätzliche Maßnahmen zur Verbesserung der Testbarkeit notwendig (Stichwort "scan Fazit'')

Ebenfalls eine andere Methode benutzt das MSS zur Auswahl arithmetisch/ logischer Bausteine (ALUS). Bei Huang werden diese bereits in der Spezifikation festgelegt. In einer Version des CMU-DA-Systems [TseSie83] werden ALUS durch Lösen eines Cliquenproblems für Graphen konstruiert. Im MSS erfolgt eine kostenoptimierte Auswahl von ALUS aus einer Bibliothek mittels linearer Programmierung.

Ältere Versionen des CMU-DA-Systems (und vermutlich auch das LALSD II-System) wurden zum Entwurf von diskret (z.B. mit TTL-Chips) aufgebauten Schaltungen konzipiert und enthalten daher keine Optimierung der Verbindungen. Wegen der zunehmenden Bedeutung von Leitungen in der VLSI-Technologie ist diese Optimierung nunmehr notwendig. Der Ansatz von Parker [ParKurMli84] kann aufgrund seiner Komplexität nicht befriedigen. Im MSS wird ein neues Verfahren zur Verbindungsminimierung eingesetzt.

In gewissem Umfang implizieren auch Anwenderprogramme bereits die spätere Rechnerstruktur. Beispielsweise wurden im Rahmen des MIMOLAProjekts Teile des in Assembler geschriebenen Betriebssystems eines Prozeßrechners nach MIMOLA übertragen und als Eingabe für die Synthese benutzt. Die synthetisierten Architekturen enthielten in der Regel einen Registerspeicher der gleichen Größe, wie er auch im Prozeßrechner vorhanden war. Obwohl die Programme nicht explizit auf den Registerspeicher Bezug nahmen, wurde er doch durch sie impliziert. Der Grund lag darin, daß alle Datenstrukturen des Betriebssystems auf den vorhandenen Befehlssatz hin optimiert waren.
Effekte dieser Art würden vermieden und der Entwurfsspielraum würde weiter vergrößert werden, wenn die Spezifikation auf der funktionalen statt auf der algorithmischen Ebene erfolgte. Dadurch könnte u.a. die Entwicklung eines geeigneten Algorithmus dem Entwurfsverfahren überlassen werden. Es erscheint jedoch unwahrscheinlich, daß ein Syntheseverfahren automatisch z.B. einen "optimalen" Sortieralgorithmus entwickelt. Diese Annahme wird gestützt durch die Komplexität von Programmtransformations-Systemen (vgl. [Bau78]).

Neben dem klassischen von - Neumannschen Rechnerkonzept werden seit einiger Zeit alternative Rechnerkonzepte untersucht. Dazu gehören beispielsweise die Konzepte von LISP-Maschinen, Datenflußmaschinen, Reduktionsmaschinen, objektorientierten Systemen und PROLOG - Maschinen (siehe z.B. [Tre82]). Auch zum Entwurf dieser Rechner ist der Ansatz der Vorgabe eines Programms geeignet, da der größte Teil von ihnen durch Mikroprogrammierung eines geeigneten Interpreters auf einer relativ konventionellen Hardware realisiert wird([DobPatDes84,Tic84,Kra81]). Zum Entwurf eines solchen Rechners muß der Interpreter in einer Programmiersprache formuliert und als "Anwenderprogramm" benutzt werden. Interpreter werden häufig in einer sog. "systems implementation language" (SIL) auf relativ niedrigem Niveau programmiert. Da MIMOLA zum Programmieren auf niedrigem Niveau geeignet ist und auch als SIL bezeichnet werden kann, können Interpreter in dieser Sprache gut beschrieben werden.

4.2 Funktion des Synthesesystems

Unter Benutzung der Definitionen aus Kap.2 kann man die Synthese im MSS betrachten als Ausführen einer Abbildung

```
    ( p, (modules , parts , ifields , connections ,  templocs ) )
 -> ( p',(modules', parts', ifields', connections',  templocs') )
```

mit:

p	: Eingabeprogramm
modules	: Menge von Modultypen (Bibliothek)
parts	: enthält den Programmzähler sowie ggf. Haupt- und Registerspeicher
ifields	: Befehlsformat-Deklaration; in der Regel leer; kann aber als Ressource-Restriktion dienen
connections	: Verbindungsliste; wird, sofern vorhanden, ignoriert
templocs	: Liste erlaubter Hilfszellen eines adressierbaren Speichers; kann leer sein
p'	: In ein Maschinenprogramm für die erzeugte Hardware transformiertes Eingabeprogramm (d.h. die Synthese enthält einen kleinen "Übersetzer")
modules'	: Menge benutzter Modultypen; in der synthesierten Hardware nicht benutzte Bibliotheks-Module sind in **modules'** nicht enthalten; da die Synthese einfache Module (z.B. Multiplexer) selbst generiert, ist **modules'** nicht notwendig eine Teilmenge von **modules**
parts'	: Liste benutzter Bausteine
ifields'	: Generiertes Befehlsformat
connections'	: Generierte Verbindungen
templocs'	: Gleich **templocs** falls **templocs** nicht leer; wird sonst aus den Angaben für den kleinsten Speicher erzeugt

Der Umfang und die Art des Programms p muß so gewählt werden, daß der Rechner nicht für zu spezielle Aufgaben entworfen wird. Müssen gewisse Operationen unabhängig von ihrem Vorkommen im Programm ausführ

bar sein, so sollten diese Operationen dem Programm zugesetzt werden. Auf diese Weise kann z.B. sichergestellt werden, daß für alle arithmetischen Operationen einer Quellsprache Hardware generiert wird.

Übliche Programme enthalten einige für den Hardware-Entwurf benötigte Informationen nur implizit, so z.B. Informationen über die maximale Größe vorkommender Zahlen, gewünschte Laufzeit-Fehlertests u.s.w.. Diese impliziten Informationen müssen explizit gemacht werden. Die Sprache MIMOLA bietet hierzu Mittel an.

Bei der Synthese von Rechnerstrukturen aus Programmen handelt es sich im Prinzip um ein Top-Down-Entwurfsverfahren. Da in der Regel aber nur ein bestimmter Satz von Hardware-Komponententypen (wie z.B. Standardzellen oder TTL-Schaltkreise) zur Verfügung steht, müssen Elemente des Bottom-Up-Designs berücksichtigt werden. Konkret bedeutet dies, daß vorhandene Hardware-Komponententypen **(modules)** bereits in der Entwurfsspezifikation enthalten sein sollen. Da in der Regel keine allzu großen Wahlmöglichkeiten bezüglich der Speicher bestehen, geht das in dieser Arbeit beschriebene MSS2 i. Ggs. zum älteren MSS1 ([Zim80, Mar79, Mar80a]) davon aus, daß diese bereits in der Spezifikation als **parts** erklärt werden. Ggf. muß für eine geringe Zahl von Alternativen ein getrennter Syntheselauf durchgeführt werden.

Mit Hilfe von Befehlsfeld-Deklarationen **(ifields)** kann die maximale Zahl **von Befehlsfeldern** einer bestimmten Breite vorgegeben werden. Werden z.B. zwei 32-Bit Felder deklariert, so enthält das erzeugte Befehlsformat maximal zwei solcher Felder für Adreß- und ImmediateKonstanten.

Sofern Zwischenergebnisse vorübergehend abgespeichert werden müssen, benutzt das MSS hierfür einen adressierbaren Speicher. Zusätzlich wird meist ein einzelnes 1-Bit Condition-Code-Register erzeugt. Hierdurch wird das Problem der Vielzahl von Registern vermieden. Der Speicher selbst muß unter **parts** beschrieben werden. Damit wird auch die Zahl der möglichen gleichzeitigen Zugriffe auf den Speicher defi

niert. Steht nur ein Teil der Zellen dieses Speichers für Zwischenergebnisse zur Verfügung, so müssen 'diese unter templocs aufgeführt werden.

Das folgende Beispiel zeigt die verschiedenen Teile einer Synthesespezifikation in MIMOLA im Zusammenhang. Aus Platzgründen ist sowohl die Liste der Modultypen wie auch das Programm untypisch kurz. Man beachte, daß die Spezifikation weder Verbindungen noch Befehlsfelder enthält.

```
(*----------- 'Hardware' ---------------------------*)
TARGET rechner;
 STRUCTURE
 TYPE
  word    = (15:0);

  MODULE B7483 <COST=3> (in l,r : word; out f : word);
   BEGIN
    f:=l "+" r
   END;

  MODULE B74xy <COST=10>
     (IN a,b:word;FCT sel:(1:0);OUT f:(eq(16),res(word)));
   BEGIN
    f.res:= CASE sel OF
            0 : a  "-"  b;
            1 : a  "+"  b;
            2 : a  "*"  b;
            3 : a  "/"  b;
           END,
    f.eq := CASE sel OF
            0 : a "<="  b;
            1 : a  "<"  b;
            2 : a  "="  b;
            3 : a "<>"  b;
           END
   END;
```

```
PARTS
 SR: MODULE S8 <SIZE=8>                    (*Registerspeicher*)
      PORT P1(IN e : word; ADR ad:(2:0) ; FCT c : (0));
       BEGIN
        CASE c OF
         0 : S8(ad):=e;
         1 : ;
        END
       END;
      PORT P2,P3(OUT f : word; ADR ad:(2:0) ; FCT c : (0));
       BEGIN
        f:= CASE c OF
             0 : S8(ad);
             1 : TRISTATE;              (*hochohmiger Ausgang*)
            END
       END;

 SH: MODULE S4k   <SIZE=4096>                (*Hauptspeicher*)
      (inout ea:word; adr address:word; fct sel:(0) );
      BEGIN
       CASE sel OF
        0 : S4k(address):=ea;
        1 : ea:=S4k(address);
       END
      END;

 PC: MODULE RP (in e : word; out a : word); (*Programmzähler*)
     BEGIN
      RP:=e, a:=RP
     END;

 LOCATIONS_FOR_VARIABLES   SH (4095:0);
 LOCATIONS_FOR_TEMPORARIES SR ( 5:0);

END;
```

```
PROGRAM synthesis_example;
DECLARE
 (*----- Programmtransformationen --------------------*)
 MACRO
  REPLACE
   &a: WHILE &expr DO &block
  WITH
   &a: IF &expr THEN BEGIN &block; RP:=&a END FI
  END,
  REPLACE
   GOTO &a
  WITH
   RP:=&a
  END;
 (*--------- 'Software' ----------------------------------*)
 CONST
  n = 7;
 TYPE
  integer = (15:0);
 VAR  i,m : integer;
  BEGIN
   i:=1, m:=1;
   WHILE i "<=" n DO
    BEGIN m:=m "*" i, i:=i "+" 1 END
  END.
```

Als Ebene, auf der der fertige Entwurf beschrieben wird, verwendet das MSS die RT-Strukturebene. Diese Ebene besitzt den Vorteil, weitgehend von Schaltkreis-Technologien unabhängig zu sein. Damit wird der "software life cycle" des Systems verlängert. Außerdem existieren zahlreiche CAD-Werkzeuge für den Hardwareentwurf auf den unteren Ebenen. Die Umsetzung in TTL-Schaltkreise kann in den meisten Fällen sogar manuell erfolgen. Sofern der Wunsch nach Erweiterung des MSS nach unten entsteht, kann dies ähnlich wie beim MacPitts-Compiler durch Bibliotheken von Bausteinen mit bekannten Layouts erfolgen. Das ent-

wickelte Syntheseverfahren wurde speziell im Hinblick auf BausteinBibliotheken konzipiert (Vorgabe von **modules).**

Um den Aufwand für die Kontrollogik klein zu halten, erzeugt die gegenwärtige Version des MSS ausschließlich Hardware, bei der **ein Programmschritt genau einem internen Zustandsübergang** entspricht. Es gibt also während der Ausführung eines Befehls keine Sequenzen von internen Zuweisungen. Als Folge davon kann jeder Eingang eines HardwareBausteins während der Ausführung eines Befehls nur mit einer Quelle verbunden werden und ein Multiplexen findet nicht statt.

Die vom Synthesesystem vorzunehmende Abbildung ist durch die Aufgabenstellung nicht eindeutig beschrieben. Vielmehr können vom MSS noch zahlreiche Optimierungen durchgeführt werden. Damit stellt sich die Frage nach der Zielfunktion, d.h. nach dem Ziel der Optimierungen. Mögliche Zielfunktionen sind u.a. die Minimierung der Programmlaufzeit bei vorgegebenen Ressource-Restriktionen und die Minimierung der Kosten bei vorgegebener Programmlaufzeit. Die Genauigkeit,mit der Programmlaufzeiten angegeben werden können, wenn das Layout des Rechners nicht bekannt ist, ist relativ gering. Der Grund dafür liegt u.a. in der wachsenden Bedeutung der Laufzeiten auf Leitungen in VLSI-Schaltungen.

Daher versucht das MSS standardmäßig, Größen zu minimieren, die genauer bekannt sind : die Zahl der dynamisch durchlaufenen Instruktionen oder wahlweise die Zahl der erzeugten (statischen) Instruktionen. Denkbar wäre, alternativ die Minimierung der Summe der Speicherzugriffszeiten zu gestatten. Die Zahl der dynamisch durchlaufenen Instruktionen kann das MSS aus der Zahl der Ausführungen der Basisblöcke berechnen. Diese wiederum kann der Benutzer entweder selbst im Quellprogramm p angeben oder durch das Simulations-Subsystem einfügen lassen.

Unter den Rechnern mit minimaler Zahl von statischen bzw. dynamischen Instruktionen versucht das MSS, den billigsten zu erzeugen.

Für Rechnerstrukturen (wie auch für andere Systeme) ist die Bestimmung einer Kostenfunktionen problematisch;. Alle durch den Vertrieb des Rechners verursachten und Stückzahl - abhängige Kosten lassen sich praktisch nicht berücksichtigen. Bei hochintegrierten Schaltungen können ferner die Testkosten die Herstellungskosten übersteigen. Folglich wird die Komplexität von Schaltkreistests auch im Rahmen des MIMOLASystems betrachtet [Krü85, KelWos85]. Da bislang vorliegende Ergebisse aber noch keine Berücksichtigung während der Synthese zulassen, müssen auch die Testkosten in dieser Arbeit außer acht gelassen werden.

Im folgenden sollen daher nur die Herstellungskosten betrachtet werden. Für Implementierungen in TTL-Technologie kann man diese Kosten einigermaßen abschätzen [Zim79]. Das gleiche dürfte auch für die Gate-Array Technologie gelten. Für die Realisierung als 'full custom VLSI Chip' kann man die Fläche und damit die Kosten eigentlich erst angeben, nachdem ein Layout des Chips erstellt worden ist. Die Schätzung des Flächenbedarfs eines Chips aus einer Rechnerstrukturbeschreibung ist Gegenstand laufender Arbeiten [Zim85] und konnte noch nicht berücksichtigt werden.

Das MSS soll Hardware für Programme erzeugen, die potentiell wesentlich länger sein können, als die Spezifikation von Maschinenbefehlen, wie sie im CMU-DA-System benutzt wird.

Daher ist für das MSS höchstens eine bezüglich der Länge des **Programms lineare Komplexität** zulässig. Unter dieser Prämisse erscheint der Versuch, eine einschrittige, global optimierende Synthese zu erzeugen, aussichtslos. Die gilt insbesondere, wenn man die Komplexität des Verfahrens von Hafer [HafPar83] analysiert.

Im MSS wird daher das Entwurfsproblem in eine Reihe von Teilproblemen zerlegt. Als Folge davon kann eine Optimierung im strengen, mathematischen Sinn nicht mehr erreicht werden. Damit trotzdem eine möglichst gute Gesamtlösung erhalten wird, wird das Problem in möglichst wenige Teilprobleme gerade noch handhabbarer Komplexität zerlegt.

Abb. 4.1 gibt einen Überblick über die im MIMOLA-System vorgenommene Zerlegung. Die einzelnen Syntheseschritte sind realisiert in den Programmen MSH1, MSH2 und MSH3.

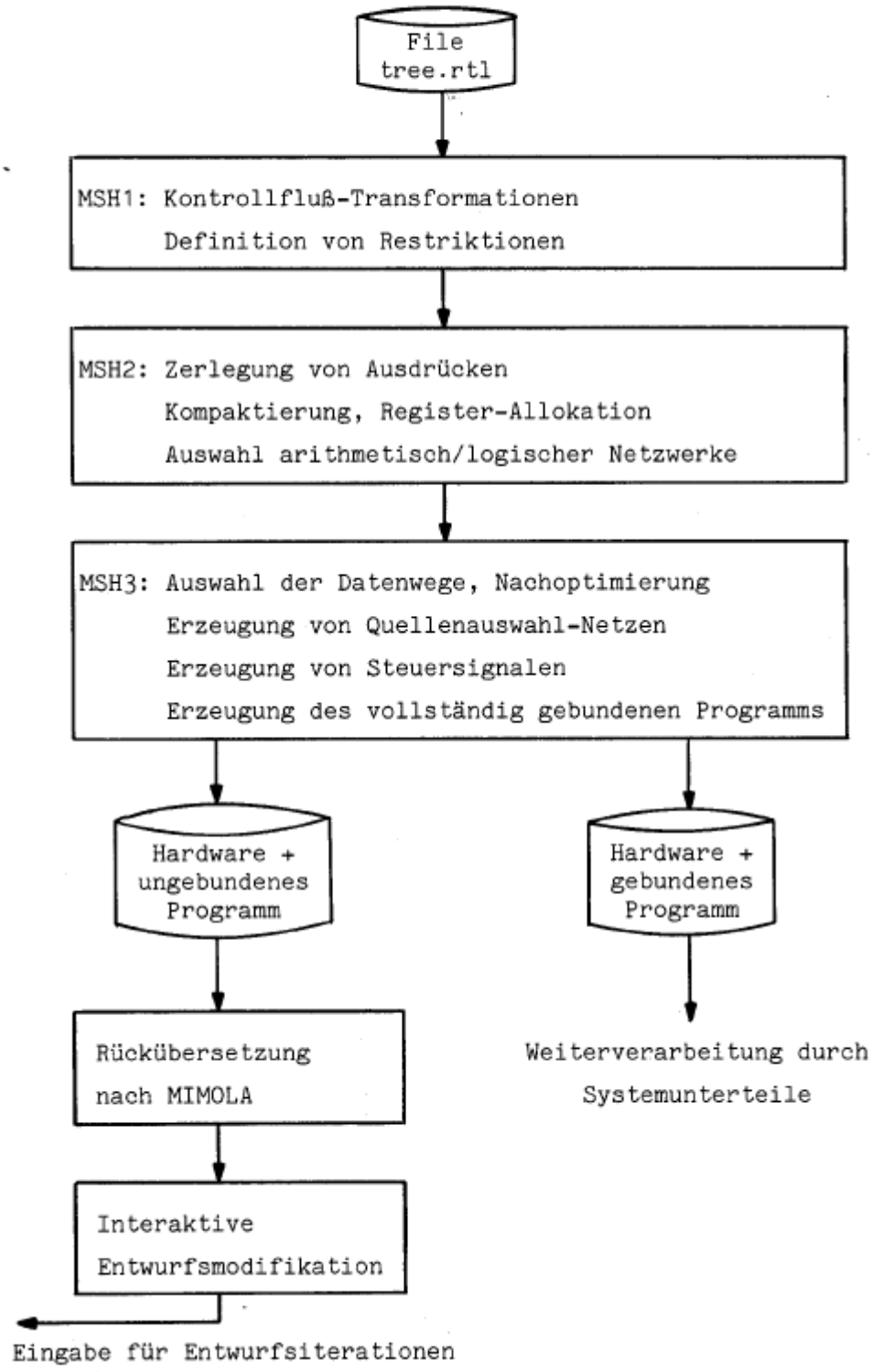


Abb. 4.1 Übersieht über das Synthese-Subsystem

Die wichtigsten Entwurfsentscheidungen fallen bei dem gewählten Verfahren an drei Stellen

1. Bei der Zerlegung von Ausdrücken,
z. bei der Auswahl e

Die übrigen Entwurfsschritte sind im wesentlichen durch diese drei bestimmt.

4.3 Kontrollfluß-Transformationen

Die Eingabeprogramme des Synthese - Systems sind RTL-TREEMOLA-Programme, d.h. Programme auf der RT-Verhaltensebene (vgl. Abb. 1.1 und Anhang A). Sie enthalten neben unbedingten Zuweisungen noch bedingte Anweisungen wie z.B.:

```
IF SM(2) ">" 0 THEN SM(0):=SM(1) ELSE SM(0):=0 FI
```

Vor der eigentlichen Synthese müssen diese Anweisungen in eine Darstellung transformiert werden,die direkt in Hardware realisiert werden kann.

Hardwaremäßig realisierbar ist die bedingte Auswahl entweder von Daten, von Adressen oder von Kontrollcodes. Die bedingte Auswahl von Daten oder Adressen zeigt Abb. 4.2:

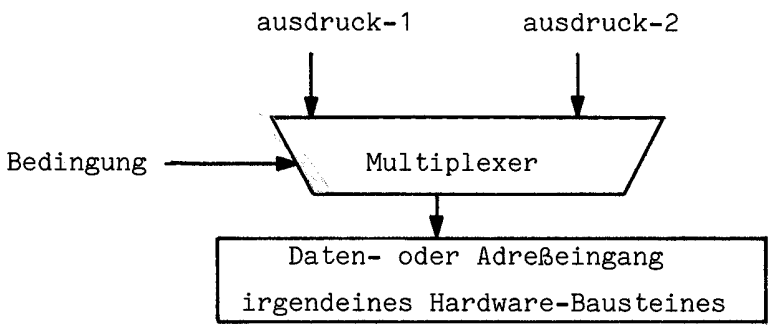


Abb. 4.2 Bedingte Auswahl von Daten oder Adressen

Die bedingte Auswahl von Kontrollcodes führt zu **bedingten Operationen.** So kann z.B. eine Operation der Art

```
(IF condition THEN "+" ELSE "-" FI)
```

durch Multiplexen der Kontrollcodes für "+" und "-" realisiert werden. Z. Zt. wird im MSS nur eine spezielle bedingte Operation unterstützt, nämlich das bedingte Laden eines Speichers oder Registers.

Die entsprechende Hardware zeigt Abb. 4.3:

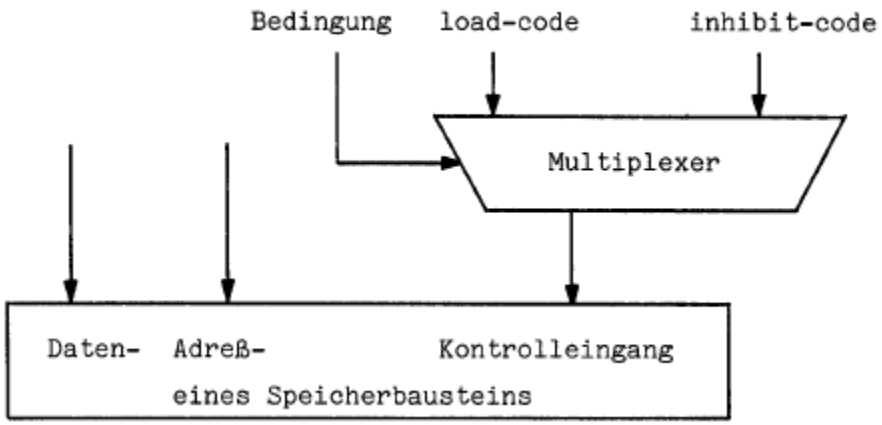


Abb. 4.3 Bedingtes Schreiben in Speicher (load-code führt zum Schreiben, inhibit-Code läßt Speicherinhalt unverändert)

Je nachdem, ob die Hardware nach Abb. 4.2 bzw. 4.3 für eine Zuweisung zum Programmzähler oder für eine Zuweisung zu einem Datenspeicher benutzt wird, lassen sich vier Fälle unterscheiden:

a) Bedingte Auswahl entsprechend Abb. 4.2 und Zuweisung zu einem Datenspeicher (im folgenden bedingte Ausdrücke genannt)
Auf diese Weise läßt sich das obige Beispiel darstellen als:

```
SM(0):=(IF SM(2) ">" 0 THEN SM(1) ELSE 0 FI)
```

b) Bedingte Auswahl entsprechend Abb. 4.2 und Zuweisung zum Programmzähler (im weiteren bedingte Sprünge genannt)

Damit läßt sich obiges Beispiel in die folgende Form transformieren (PC = Programmzähler):

```
        PC:=(IF SM(2) ">" 0 THEN PC "+" 1 ELSE Lelse FI);
        SM(0):=SM(1), PC:=Lnext                          ;
Lelse:  SM(0):=0                                         ;
Lnext:  ......
```

c) Bedingtes Schreiben in einen Datenspeicher entsprechend Abb. 4.3 (im weiteren als bedingte Zuweisung bezeichnet). Das obige Beispiel läßt sich damit wie folgt darstellen (der mit "IF" eingeleitete Term stellt den Ausdruck am Kontrolleingang dar):

```
SM(0/(IF SM(1) ">" 0 THEN <load-code>
                     ELSE <inhibit-code> FI)) := SM(1);
SM(0/(IF SM(1) ">" 0 THEN <inhibit-code>
                     ELSE <load-code> FI)   ) := 0
```

d) Bedingtes Beschreiben des Programmzählers

Beispiel:

```
PC(IF SM(1) ">" 0 THEN <load-code> ELSE <inhibit-code> FI)
  := PC "+" 1
```

Diese Form kann im wesentlichen nur zur Realisierung von Schleifen benutzt werden, die genau einen Programmschritt enthalten (Warteschleifen, Schleifen bei der Implementation der Multiplikation per Mikroprogramm). Diese Möglichkeit wird im MSS nicht genutzt.

Üblicherweise erzeugen Compiler ausschließlich bedingte Sprünge. Nachteilig ist dabei die stark sequentielle Verarbeitung (die bei schnellen Rechnern durch Pipelining mit hohem Aufwand zum Teil wieder aufgehoben

wird [HolIbb80]). Für die schnelle Verarbeitung besser geeignet sind die Darstellungen a) und c). Sie benötigen eine geringere Zahl von Programmschritten und erlauben zudem noch die parallele Ausführung weiterer Anweisungen (vgl. Abschn.. 4.13).

Im Falle geschachtelter bedingter Anweisungen und großer Blöcke benötigt man für die Darstellungen a) und c) allerdings einen hohen HardwareAufwand, da die bei sukzessiven bedingten Sprüngen implizite UNDVerknüpfung hier explizit erfolgen muß. Daher werden die Darstellungen a) und c) im MSS nur für die innerste IF-Schachtelung benutzt.

Ein weiterer Nachteil der Darstellungen nach a) und c) ist der mögliche Zugriff auf undefinierte Daten.

Es ist kaum vorherzusehen, welche der Möglichkeiten zur Abarbeitung von IF-statements entsprechend der jeweiligen Zielfunktion (minimale Zahl statischer bzw. dynamischer Instruktionen) zur besten Lösung führt. Daher erzeugt der erste Syntheseschritt drei Varianten der Abarbeitung:

1. Ausschließliche Benutzung bedingter Sprünge (Fall b)).

z. Erzeugung bedingter Ausdrücke für Zuweisungen, die in THEN- und ELSE-Teil mit gleichen linken Seiten vorkommen und Erzeugung von bedingten Sprüngen für Zuweisungen mit unterschiedlichen linken Seiten. (nur für die innerste IF-Schachtelung)

3. Erzeugung bedingter Ausdrücke für Zuweisungen, die in THEN- und ELSE-Teil mit gleichen linken Seiten vorkommen und Erzeugung von bedingten Zuweisungen für Zuweisungen mit unterschiedlichen linken Seiten. (nur für die innerste IF-Schachtelung)

Die drei Varianten werden den folgenden Syntheseschritten übergeben und von diesen unabhängig voneinander bearbeitet. Erst wenn die Länge des erzeugten Maschinenprogramms bekannt ist, erfolgt die Entscheidung zugunsten einer Variante (vgl. Abschn. 4.6 und 4.13).

Häufig zerlegt man Rechner gedanklich in ein Rechenwerk und ein Steuerwerk. Dabei wird der Programmzähler zum Zustandsspeicher und im Rechenwerk getestete Bedingungen zur Eingabe des Steuerwerks. Die Beschränkung auf bedingte Sprünge entspricht dann der Einschränkung auf MOORE-Automaten als Steuerwerk. Hängt die Funktion des Rechenwerkes wie bei bedingten Zuweisungen von der Eingabe an das Steuerwerk und nicht nur von dessen innerem Zustand ab, so kann man das Steuerwerk als MEALY-Automat bezeichnen. Letztlich ist die Bezeichnung aber davon abhängig, ob man die getesteten Bedingungen gedanklich über das Steuerwerk zum Rechenwerk führt oder ob man sich eine direkte Verbindung innerhalb des Rechenwerkes vorstellt.

4.4 Interaktive Festlegung von Entwurfsrestriktionen

Zur Analyse des Entwurfsspielraumes ist es notwendig, gewisse Entwurfsrestriktionen bei der Synthese zu berücksichtigen. Es ist nun nicht sinnvoll, alle diese Restriktionen bereits in der ursprünglichen Entwurfsspezifikation anzuführen, da dort die Zahl der vom Programm her möglichen parallelen Operationen noch unbekannt ist. Im zweiten Syntheseschritt wird diese daher berechnet und dem Benutzer auf dem Terminal angezeigt. Er hat dann die Möglichkeit, die erlaubten Hardware-Ressourcen, wie z.B. die Zahl der Speicherports, die Zahl der Konstantenfelder im Befehlswort und die obere Grenze für die Breite des Befehlsworts interaktiv zu definieren.

4.5 Zerlegung von Ausdrücken

Im nächsten Syntheseschritt werden die parallelen Blöcke daraufhin analysiert, ob sie unter Einhaltung der Ressource - Restriktionen einschrittig ausführbar sind. Ist dies nicht der Fall, so werden die parallelen Blöcke in eine Sequenz einschrittig ausführbarer Blöcke zerlegt. Das Verfahren entspricht im Prinzip dem Aufbrechen komplexer Ausdrücke in eine Reihe von Maschinenbefehlen, wie es von üblichen Compilern vorgenommen wird. Das MIMOLA-System ist aber im Gegensatz zu üblichen Compilern nicht für eine spezielle Zielmaschine ausgelegt

und erzeugt statt einzelner Maschinenbefehle Blöcke parallel ausführbarer Zuweisungen. Um für unterschädliche Restriktionen generierte Rechner untereinander in bezug auf ihre Rechenleistung vergleichen zu können, muß das MSS für alle Restriktionen eine möglichst gute (am besten: die optimale) Zerlegung liefern. Es ist daher notwendig, gemeinsame Teilausdrücke bei der Zerlegung auszunutzen.

Eine besondere Bedeutung gemeinsamer Teilausdrücke im MSS ergibt sich auch dadurch, daß während der Parallelisierung und bei der Erzeugung bedingter Ausdrücke und bedingter Zuweisungen zusätzliche gemeinsame Teilausdrücke entstehen.

Während schnelle Algorithmen für eine optimale Zerlegung unter Vernachlässigung gleicher Teilausdrücke bekannt sind (vgl. [AhoJoh76, SetU1170]), liegen entsprechende Verfahren für den allgemeinen Fall nicht vor. Der Grund dafür liegt in der NP - Vollständigkeit des Problems. Lediglich für spezielle Fälle sind polynomielle Algorithmen bekannt. So betrachten Prabhala und Sethi [PraSet80] Zerlegungen für eine kellerartige Verwaltung der Hilfszellen, spezielle Flußgraphen und einfache Maschinenbefehle.

Im MSS wird daher eine heuristische Methode zur Zerlegung der Ausdrücke benutzt.

Zunächst werden in den Datenflußbäumen gemeinsame Teilausdrücke bestimmt, explizit von links nach rechts verkettet und mit einem Index versehen ("value numbering"). Dieses Vorgehen bewirkt, daß die Bäume als gerichtete, azyklische Graphen ("DAG") behandelt werden können, in denen keine gemeinsamen Teilausdrücke vorkommen. Bei einem Durchlaufen der Bäume werden dabei jene Teilbäume ausgelassen, deren Index gleich dem Index eines bereits betrachteten Teilbaumes ist.

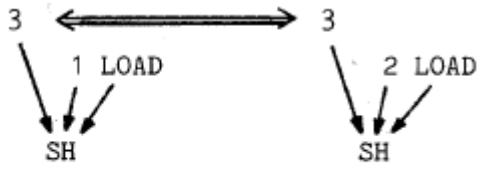


Abb.4.4 Explizite Verkettung (⟺) von gemeinsamen Teilausdrücken

Als wesentliches Kriterium bei der Zerlegung wird im MSS die Zahl der gleichzeitig möglichen Speichelzugriffe betrachtet. Es sei hier erwähnt, daß Beschränkungen dieser Zahl in den meisten anderen CADSystemen umgangen werden, indem für jede Variable ein Register eingebaut wird (siehe z.B. [Sou83]). Dadurch ergibt sich aber ein unbefriedigend hoher Hardwarebedarf.

Die Zerlegung der Ausdrücke basiert auf der Definition
des Prädikats **treetoobig : treeclass** -> {true,false}
der Funktion **mostcomplex: treeclass -> nodeclass.**

treetoobig und **mostcomplex** enthalten die Zahl der Speicherzugriffe als wichtige Kriterien.

treetoobig(t) mit t ∈ **treeclass** liefert den Wert true, falls der Baum t nicht unter Einhaltung der Entwurfsrestriktionen einschrittig ausgeführt werden kann und false, sonst.

Im MIMOLA-System ist **treetoobig** definiert durch:

treetoobig(t):=
    true, falls die in t enthaltene Zahl von Speicherzugriffen
                oder
            die Zahl der Direktoperanden bestimmter Länge
                oder
            die Summe der für Direktoperanden benötigten Bits
                oder
            die Zahl der Operationen
        das festgelegte Maximum überschreiten
    false, sonst

Schreib/Leseports (vgl. Abschn. 2.1.1) werden bei der Berechnung der Zahl möglicher Speicherzugriffe besonders berücksichtigt. Hierbei bewährt es sich, daß gleiche Speicheradressen als spezielle gemeinsame Teilausdrücke explizit verkettet sind.

Das im folgenden beschriebene Verfahren wäre auch anwendbar, falls **treetoobig(t)** genau dann den Wert false liefern würde, falls t mit Hilfe eines einzigen Befehls eines vorgegebenen Maschinenbefehlssatzes implementiert werden könnte. Diese softwarenahe Definition wird hier nicht benutzt, da kein Maschinenbefehlssatz vorgegeben werden soll.

Das Aufbrechen der Ausdrücke erfolgt nun, indem bei einem Durchlauf durch den parallelen Block von den Blättern zur Wurzel jeweils das Prädikat **treetoobig** berechnet wird. Ergibt dieses den Wert true, so wird einer der im gegenwärtigen Ausdruck enthaltenen Teilausdrücke einer Hilfsvariablen zugewiesen, und der Teilausdruck wird durch diese Hilfsvariable ersetzt. Ist der Teilausdruck ein gemeinsamer Teilausdruck, so wird dieser überall durch diese Hilfsvariable ersetzt (hiervon ausgenommen sind Fälle, in denen der gemeinsame Teilausdruck bereits als Teil eines umfassenderen Ausdrucks einer anderen Hilfsvariablen zugewiesen wurde).

Die Auswahl des abzuspaltenden Teilausdrucks übernimmt die Funktion **mostcomplex**. In Übereinstimmung mit der Spezifikation des Entwurfszieles in Abschnitt 4.1 erfolgt diese Auswahl derart, daß eine möglichst hohe Verarbeitungsgeschwindigkeit erzielt wird. Zu diesem Zweck werden alle Speicherleseoperationen so früh wie möglich angestoßen, indem der Teilausdruck mit der größtmöglichen Zahl von Speicherzugriffen ausgewählt wird. Bei gleicher Zahl von Zugriffen erhalten die Zugriffe auf den (langsamen) Hauptspeicher ein höheres Gewicht. Ergibt sich weiterhin eine gleiche Zahl von gewichteten Zugriffen, so werden gemeinsame Teilausdrücke bevorzugt. Ist auch dieses Kriterium nicht eindeutig, so erfolgt die Auswahl von links nach rechts. Als Hauptspeicherzugriffe gelten dabei alle Zugriffe auf Speicher, die nicht der Aufnahme von Zwischenergebnissen dienen.

In der nachfolgenden genaueren Beschreibung der Funktion **mostcomplex** sei:

as ∈ **treeclass**, **nodetype**(**root**(as))=assignmenttype
: der Datenflußbaum einer Zuweisung,

t,n,n' ∈ **nodes**(as) : beliebige Knoten der Zuweisung
mem_ref(n') : die Gesamtzahl der Speicherzugriffe im Teilbaum mit Wurzel n'
main_ref(n') : die Gesamtzahl der Hauptspeicherzugriffe im Teilbaum mit Wurzel n'
commonsub : die Menge der Wurzeln gemeinsamer Teilausdrücke

Dann ist **mostcomplex**(n) eine Funktion, die einen Knoten x mit folgenden Eigenschaften liefert:

a. x ∈ **nodes**(**tree**(n))- {n}
b. ∀ x' ∈ **nodes**(**tree**(n)) - {n}, x' ≠ x :
(**mem_ref**(x) > **mem_ref**(x'))
OR (((**mem_ref**(x) = **mem_ref**(x'))
AND (((**main_ref**(x) > **main_ref**(x'))
OR (((**main_ref**(x) = **main_ref**(x'))
AND ((x ∈commonsub)
OR NOT(x' ∈commonsub)))))
c. Für die Zuweisung von **tree**(x) zu einer Hilfsvariablen ist **treetoobig** = false
d. x bezeichnet nicht das einzige lesende Vorkommen einer Hilfsvariablen, die Adresse des für Hilfsvariablen benutzten Speichers oder ein Condition-Code-Register.
e. Es existiert kein den Knoten x enthaltender Teilbaum mit den Eigenschaften a. bis d.
f. Existieren mehrere Knoten mit den Eigenschaften a. bis e., so erfolgt die Auswahl von links nach rechts.
g. Existiert kein Knoten mit den Eigenschaften a. bis e., so liefert **mostcomplex** den speziellen Wert **nil**.

Das Verfahren zur Aufspaltung der Ausdrücke arbeitet nun wie folgt (die Prozedur push hinterlegt die erzeugten Zuweisungen zur Weiterverarbeitung in einem Stack):

```
PROCEDURE chopping(t : treeclass);
FOR ALL n ∈ nodes(t)
 SEQUENCE: depth-first,left-to-right DO
 WHILE treetoobig(n) DO
  x:=mostcomplex(n);
  IF x=nil THEN Fehler('keine Lösung') FI;
  Weise den durch x repräsentierten Ausdruck
   einer Hilfsvariablen zu; Sei x' der erzeugte Baum.
  push(tree(x'));
  Ersetze x durch Lesen der Hilfsvariablen
  Ersetze alle Teilausdrücke, die gleich dem durch x
   repräsentierten sind, ebenfalls durch Lesen der
   Hilfsvariablen, sofern diese nicht bereits Teil eines
   push übergebenen Ausdrucks waren
 OD;
 IF nodetype(n)=assignmenttype THEN push(tree(n)) FI;
OD;
```

Alg. 4.1 Zerlegung von Ausdrücken

Wesentlich für das Terminieren des Algorithmus 4.1 ist Punkt d. der Definition von **mostcomplex**. Algorithmus 4.1 reduziert die Zahl der Speicherzugriffe in **tree(n)** bis entweder **treetoobig(n)** false ist oder alle Knoten q E **sons(n)** Referenzen auf nur einmal lesend vorkommende Hilfsvariable sind. Im zweiten Fall ist **mostcomplex(n)=nil**. Ist trotzdem **treetoobig(n)=true** , so findet sich keine Lösung. Dieser Fall tritt ein, wenn die Zahl der Ports des Registerspeichers oder dessen Adressenfelder zu stark eingeschränkt sind.

Liefert **treetoobig(n)** im Algorithmus 4.1 den Wert true, dann gilt für alle Knoten q E **sons(n) treetoobig(q)=false**, da das Abspalten von den Blättern zur Wurzel hin erfolgt. **mostcomplex(n)** liefert folglich nur dann keinen Knoten E **sons(n)**, falls für die Zuweisung zur Hilfsvariablen bereits belegte Ressourcen benötigt werden. Dieser Fall entsteht insbesondere, wenn für alle Knoten aus **sons(n)** alle möglichen Adreßeingänge des Registerspeichers belegt sind und zum Abspeichern in einer Hilfszelle ein weiterer Adreßeingang benötigt wird.

Beispiel:
Gegeben sei ein Registerspeicher SR mit einem Lese- und einem Schreib/ Leseport. Zu zerlegen sei der Ausdruck

```
SR(0):= (SR(1)"+"SR(2)) "*" (SR(3)"+"SR(4))
```

Für keine der Zellen SR(1) bis SR(4) sei ein Uberschreiben erlaubt. Gelangt man nun mit Algorithmus 4.1 zum "*", so kann keines der Argumente einer Hilfszelle zugewiesen werden und es muß eine Register/ Register-Zuweisung erfolgen, wie z.B. in der Sequenz

```
SR(0):=SR(1);
SR(0):=SR(0) "+" SR(2);
SR(5):=SR(3);
SR(5):=SR(5) "+" SR(4);
SR(0):=SR(0) "*" SR(5);
```

Im maschinenunabhängigen Codegenerator von Cattell [Cat78] werden wie bei der hier beschriebenen Methode möglichst große Teilausdrücke abgespalten und Hilfsvariablen zugewiesen. Cattell gibt an, daß seine Methode sich in Beispielen "nearly as well as the (optimal) brute force method" verhielt. Bei Zerlegung mittels **mostcomplex und chopping konnten** in einer ganzen Reihe von Beispielen manuell keine besseren Zerlegungen gefunden werden.

Trotz der Ähnlichkeiten zwischen Cattells "maximum munching method" und dem hier benutzten Verfahren gibt es einen wesentlichen Unterschied: Während bei Cattell die Bäume mit Mustern verglichen werden, die durch die vorhandenen Maschinenbefehle definiert sind, ist dem Syntheseverfahren noch kein Maschinenbefehlssatz bekannt. Dieser wird vielmehr erst entworfen.

Anhand der Zuweisung

```
h := ((a "+" b) "+" c) "+" ((d "+" e) "+" (f "+" g))
```

soll die Zerlegung nach Algorithmus 4.1 mit der Zerlegung nach der Kellermethode (siehe z.B. [KanLan731] verglichen werden.

Es wird angenommen, daß die Variablen einem Hauptspeicher SH zugeordnet sind und über eine in einem Registerspeicher SR enthaltene Basis adressiert werden. Abb. 4.5 zeigt einen entsprechenden Datenflußbaum. a' bis h' stellen die Adressen der entsprechenden ungestrichenen Variablen dar und enthalten eine Referenz auf den Registerspeicher.

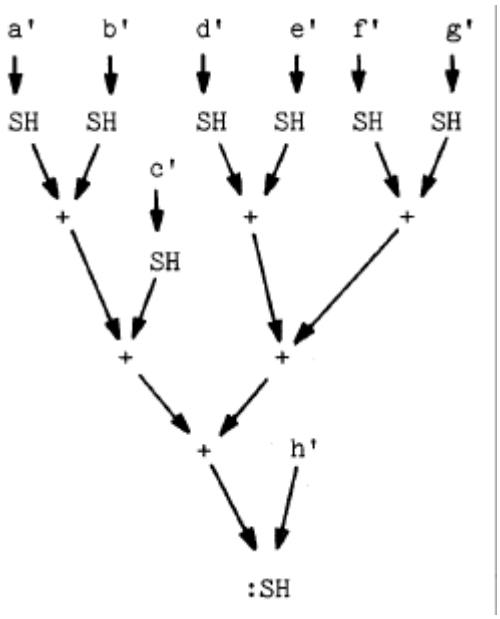


Abb.4.5 Datenflußbaum

Im MSS wird den Hilfsvariablen zunächst nur ein Name und ein Speichermodul zugewiesen. Die Zelle innerhalb dieses Moduls und damit seine Adresse wird erst durch Algorithmus 4.4 bestimmt. In der Tabelle 4.1 sind daher die Hilfsvariablen des MSS als v_i , $1 \leq i \leq$ vmax, dargestellt. Für die Kellermethode sind dagegen bereits Adressen angegeben.

Es wird angenommen, daß die Hilfsvariablen ebenfalls dem Registerspeicher zugeordnet werden sollen und daß der Hauptspeicher lediglich 1 Port besitze. Die Zahl der Ports des Registerspeichers sei alternativ entweder unbeschränkt oder 3 (1 Eingang, 2 Ausgänge).

	MIMOLA - SH : 1 Port SR : unbeschränkt	System SH : 1 Port SR : 1+2 Ports	Kellermethode (für 1 1/2 - Adreßmaschine)
1	v_1 :=SH(a')	v_1 :=SH(a')	SR(1):=SH(a')
2	v_2 :=v_1 +SH(b')	v_2 :=v_1 +SH(b')	SR(1):=SR(1)+SH(b')
3	v_3 :=SH(d')	v_3 :=SH(d')	SR(1):=SR(1)+SH(c')
4	v_4 :=SH(f')	v_4 :=SH(f')	SR(2):=SH(d')
5	v_5 :=v_3 +SH(e')	v_5 :=v_3 +SH(e')	SR(2):=SR(2)+SH(e')
6	v_6 :=v_5 +v_4 +SH(g')	v_6 :=v_4 +SH(g')	SR(3):=SH(f')
7	v_7 :=v_2 +SH(c')+v_6	v_7 :=v_2 +SH(c')	SR(3):=SR(3)+SH(g')
8	SH(h'):=v_7	v_8 :=v_5 +v_6	SR(2):=SR(2)+SR(3)
9		v_9 :=v_7 +v_8	SR(1):=SR(1)+SR(2)
10		SH(h'):=v_9	SH(h'):=SR(1)

Tab.4.1 Vergleich verschiedener Zerlegungsstrategien

Anhand der linken Spalte erkennt man, daß bei Abwesenheit von Restriktionen für die Zugriffe auf den Registerspeicher eine 100%ige Auslastung der Zugriffsbandbreite des Hauptspeichers resultiert. Hierzu sind Befehle mit zweifacher Addition notwendig. Die Anzahl der Befehle ist kleiner als die von Compilern für eine 1 $\frac{1}{2}$ Adreßmaschine erzeugte Zahl.

Für das MIMOLA-System gibt die Tabelle die Sequenz für die Aufrufe der Prozedur **push** wieder. Die im nächsten Abschnitt beschriebene Prozedur **packtemp** versucht, die Zahl benötigter Hilfszellen klein zu halten und sortiert daher die **push** übergebenen Zuweisungen so um, daß Zuweisungen zu Hilfsvariablen möglichst dicht vor den lesenden Referenzen plaziert werden. In Kombination mit Algorithmus 4.4, der den Hilfsvariablen v_i Adressen zuordnet, ergibt sich dadurch die in Tabelle 4.2 wiedergegebene Reihenfolge.

Ports des Registerspeichers :	
unbeschränkt	1 Eingang + 2 Ausgänge
SR(1) :=SH(d') SR(1) :=SR(1)+SH(e') SR(2) :=SH(f') SR(2) :=SR(1)+SR(2)+SH(g') SR(1) :=SH(a') SR(1) :=SR(1)+SH(b') SR(1) :=SR(1)+SH(c')+SR(2) SH(h'):=SR(1)	SR(1):=SH(f') SR(1):=SR(1)+SH(g') SR(2):=SH(d') SR(2):=SR(2)+SH(e') SR(2):=SR(2)+SR(1) SR(1):=SH(a') SR(1):=SR(1)+SH(b') SR(1):=SR(1)+SH(c') SR(1):=SR(1)+SR(2) SH(h'):=SR(1)

Tab. 4.2 Hilfszellen-Belegung beim MIMOLA-System

Das MSS benötigt eine Hilfszelle weniger als die Kellermethode und erreicht damit die minimale Zahl möglicher Hilfszellen, wie man anhand des Verfahrens von Sethi und Ullman [SetU1170] nachweisen kann. Der Grund für das gute Abschneiden des MSS liegt darin, daß die Algorithmen 4.2 und 4.3 in der Tendenz dafür sorgen, daß die Zweige des Datenflußbaumes mit der größten Zahl von Speicherzugriffen als erste ausgeführt werden. Damit wird das bei Abwesenheit gleicher Teilausdrücke optimale Verfahren von Sethi und Ullman approximiert.

Der wesentliche Vorteil der beschriebenen Zerlegung von Ausdrücken liegt in der Ausnutzung gleicher Teilausdrücke und der Flexibilität hinsichtlich möglicher Entwurfsvorgaben. Hinsichtlich der Zahl benötigter Hilfszellen hält das Verfahren den Vergleich mit Standardverfahren aus.

4.6 Kompaktierung

Als Erweiterung gegenüber klassischen Compilern soll erlaubt werden, daß mehrere unabhängige Zuweisungen durch einen einzigen Befehl gesteuert werden. Hierdurch kann die Rechengeschwindigkeit meist einfacher als durch Pipelining-Techniken gesteigert werden. Die parallele Abarbeitung mehrerer Zuweisungen ist aus der horizontalen Mikroprogrammierung bekannt. Aus diesem Bereich stammen daher auch die im folgenden definierten Begriffe (vgl. z.B. Mallett [Mal78]).

Sei p ein Programm.

Def.:

1. Eine M i k r o o p e r a t i o n (MO) ist ein Teilbaum des Programms p mit einer Wurzel vom Typ assignmenttype, d.h.
   MO=**tree**(s) mit s ∈ **nodes**(p), **nodetype**(s)=assignmentype.

Eine Mikrooperation beschreibt die Berechnung eines Wertes und dessen Zuweisung an eine (Daten-) Speicherzelle.

z. Mengen von MO, deren Wirkung durch Inhalte der Zellen des Mikrobefehlsspeichers dargestellt werden können, heißen M i k r o i n s t r u k t i o n e n (MI) oder M i k r o b e f e h l e.

3. Unter (Mikrocode-) K o m p a k t i e r u n g versteht man die Zuordnung von Mikrooperationen zu Mikroinstruktionen, d.h. die Partitionierung der Menge der MO bezüglich der MI, die ihre Ausführung bewirken.

Den Zusammenhang des im MSS verwendeten Algorithmus mit üblichen Kompaktierungsalgorithmen erhält man, indem man die **an push übergebenen** Zuweisungen mit Mikrooperationen identifiziert. Im Gegensatz zu üblichen mikroprogrammierten Maschinen gibt es aber bei den vom MSS erzeugten Rechnern oberhalb der "Mikrobefehlsebene" z.Zt. keine weitere von der Maschine interpretierte Befehlsebene. Deshalb werden im folgenden die Begriffe "Mikroinstruktionen" und "Instruktionen" synonym benutzt.

Die Kompaktierung der im vorigen Abschnitt erzeugten Zuweisungen ist möglich, sofern einzelne Zuweisungen so viele Ressourcen nicht belegen, daß weitere Zuweisungen parallel ausgeführt werden können. Gibt es beispielsweise ein Eingangsund ein Ausgangs-Port des Hauptspeichers, so können das Laden eines Registers aus dem Hauptspeicher und das Kopieren eines zweiten Registers in den Hauptspeicher in einem Befehl untergebracht werden.

Es sei an dieser Stelle vorweggenommen, daß als Ergebnis der Kompaktierung zwei Arten von Befehlen, die auch in klassischen Maschinen vorkommen, relativ häufig beobachtet wurden. Es sind dies der Zugriff auf ein Arrayelement mit gleichzeitiger Erhöhung des Index ("autoincrement") und der bedingte Sprung mit paralleler Erhöhung des getesteten Wertes ("increment and branch"). Diese Kombinationen sind auch bei relativ wenigen parallelen Speicherzugriffen möglich, da sie gleiche Teilausdrücke enthalten. Es ist interessant, daß sich diese Befehle, die zur Steigerung der Rechenleistung eingeführt wurden, in das Schema des MSS besser einfügen als in das Schema normaler Compiler.

Zur Darstellung von Kompaktierungsverfahren müssen zunächst weitere Definitionen und Begriffe eingeführt werden:

Def.:

4. Z ist die Menge aller Teilbäume von p, deren Wurzel eine Speicherreferenz ist:
   $Z := \{\mathbf{tree}(n) \mid n \in \mathbf{nodes}(p), \mathbf{part\_id}(n) \text{ ist Speichername}\}$

5. $\forall\ z_i, z_j \in Z$ wird die Funktion
   $\mathbf{samelocation}(z_i, z_j)$ beschrieben durch:

   $\mathbf{samelocation}(z_i, z_j) :=$
   {true, falls z_i und z_j entscheidbar gleich sind
   {false, falls z_i und z_j -"- -"- ungleich sind
   {unknown, sonst

"Entscheidbar" in diesem Sinne meint nicht die theoretische Entscheid-

barkeit,

Die Funktion **samelocation** ist einfach zu berechnen, falls als Speicherzellen lediglich Register zu betrachten sind.

Diese Voraussetzung ist im MSS im Gegensatz zu vielen Mikroprogrammiersystemen nicht erfüllt. Im MSS müssen auch Speicherzellen mit berechneten Adressen betrachtet werden. Um die Frage nach der Gleichheit zweier Speicherzellen trotzdem möglichst häufig entscheiden zu können, wird ausgenutzt, daß je zwei verschiedene Variablenidentifikatoren, von denen keiner ein formaler CALL-BY-REFERENCE Parameter ist, disjunkte Speicherbereiche bezeichnen (Ausschluß des sog. "aliasing" (vgl. Cooper [Coo85])). Zu diesem Zweck werden Variablenidentifikatoren bei der Abbildung auf die RT-Verhaltensebene nicht eliminiert, sondern den Bäumen als Attribut zugewiesen. Die Komponente "var_id" der Knoten dient der Aufnahme dieses Attributes (vgl. Anhang A).

Dies erlaubt es, **samelocation** im MSS zu definieren durch:

samelocation(z_i, z_j) :=
  true, falls die Datenflußbäume z_i und z_j gleich sind
  false,falls
    part_id(**root**(z_i))≠**part_id**(**root**(z_j))
      oder
    range_id(**root**(z_i)) ∩ **range_id**(**root**(z_j))=∅
      oder
    var_id(**root**(z_i))≠**var_id**(**root**(z_j))
    (d.h. die Wurzeln bezeichnen unterschiedliche
    Variable, von denen keine eine CALL-BY-REFERENCE
    Variable ist)
      oder
    root(z_i) und **root**(z_j)
    unterscheiden sich in ihren Adressen um eine von Null
    verschiedene additive Konstante
  unknown, sonst

Die folgenden Beispiele zeigen exemplarisch, in welchen Fällen mit der so definierten Funktion eine Entscheidung möglich ist:

z_i	z_j	**samelocation**(z_i, z_j)
i	j	false
i.(7:0)	i.(15:8)	false
a [1]	a [2]	false
a [i]	a [2]	unknown
a [i]	b [2]	false
a [i+1]	a [i]	false
c [i,2]	c [i,3]	false

Tab. 4.3 Ergebnisse von **samelocation**

Dabei wurde vorausgesetzt, daß keine der Variablen eine CALL-BY-REFERENCE Variable ist. Das Ergebnis gilt sowohl für die absolute Adressierung als auch für die Adressierung über (möglicherweise verschiedene) Basisregister.

Seien nun s_i und s_j zwei Zuweisungen eines parallelen Blockes des Eingabeprogramms.

Def.:

6. $RD(s_i)$ bzw. $RD(s_j) \in Z$ seien die lesenden Speicherreferenzen von s_i bzw. s_j, d.h. für s_i:
   $RD(s_i) := \{$ **tree**(n) $|\ n \in$ **nodes**(s_i); **part_id**(n) ist Speichername, **nodetype**(n)=sourcetype$\}$

7. $WR(s_i)$ bzw. $WR(s_j) \in Z$ seien die Mengen der von s_i bzw. s_j beschriebenen Zellen, also z.B. für s_i:
   $WR(s_i) := \{$ **tree**(n) $|\ n \in$ **nodes**(s_i); **part_id**(n) ist Speichername, **nodetype**(n)=assignmenttype$\}$

Ähnlich wie bei den meisten anderen Programmiersprachen für von-Neumann Rechner ist es auch bei MIMOLA möglich, Variablen mehrfach einen Wert zuzuweisen, sie also als wiederverwendbare "Behälter" zu benutzen. Um zwischen verschiedenen Benutzungen desselben Behälters zu unterscheiden, sei ein Prädikat **sameuse**(z_i, z_j) definiert. **sameuse** kann in der Regel nur in einem lokalen Bereich eines Programmes erklärt werden. Die übliche Einschränkung besteht in der Definition jeweils nur für einen sog. Basisblock. Da Basisblöcke keine Verzweigungen enthalten, kann aus der statischen Reihenfolge auf die Lebensbereiche der einzelnen Benutzungen einer Zelle geschlossen werden (siehe [Hec77]).

Programmiersprachen für Datenflußmaschinen folgen in der Regel dem "single assignment"-Prinzip, d.h. jeder Variablen darf nur einmal ein Wert zugewiesen werden. Folglich gilt bei Anwendung dieses Prinzips **sameuse**(z_i,z_j)=true für z_i=z_j. Das "single assignment"-Prinzip gilt, wie im nächsten Abschnitt begründet wird, für im MSS benutzte Hilfsvariablen.

Die Kompaktierung beschränkt sich auf die Betrachtung eines einzelnen parallelen Blockes zur Zeit. Daher sind alle lesenden Zugriffe auf Zellen, die nicht Hilfszellen sind, Zugriffe auf beim Eintritt in den Block gültige Zelleninhalte. Alle schreibenden Zugriffe definieren Inhalte für nachfolgende Blöcke. Damit konsistent ist die Definition von **sameuse** im MSS als:

Def.:

8. **sameuse**(z_i, z_j) :=
   IF z_i und z_j sind Hilfsvariable THEN true
   ELSE IF z_i oder z_j sind Hilfsvariable THEN false
   ELSE IF ($z_i \in RD(s_i)$) = ($z_j \in RD(s_j)$) THEN true
   ELSE false;

Will man **sameuse** auch für sequentielle Blöcke definieren, so muß man die Lebensbereiche der Benutzungen von Variablen mit Hilfe einer umfangreicheren Datenflußanalyse studieren [Hec77].

Mit **sameuse** lassen sich jetzt folgende Relationen definieren:

Def.:

9. s_i heißt datenabhängig von s_j
(in Zeichen: s_i **da** s_j) g.d.w.
$\exists\ z_i \in RD(s_i),\ z_j \in WR(s_j)$:
sameuse(z_i, z_j) = true und
samelocation(z_i, z_j)=true

10. s_i heißt antidatenabhängig von s_j
(in Zeichen: s_i **ada** s_j) g.d.w.
$\exists\ z_i \in WR(s_i),\ z_j \in RD(s_j)$:
sameuse(z_i, z_j) = false und
samelocation(z_i, z_j)= true

11. s_i heißt ausgabeabhängig von s_j
(in Zeichen: s_i **aa** s_j) g.d.w.
$\exists\ z_i \in WR(s_i),\ z_j \in WR(s_j)$:
samelocation(z_i, z_j)=true

12. s_i heißt potentiell datenabhängig von s_j
(in Zeichen: s_i **da*** s_j) g.d.w.
$\exists\ z_i \in RD(s_i),\ z_j \in WR(s_j)$:
sameuse(z_i, z_j) = true und
samelocation$(z_i, z_j) \neq$ false

Analog zu 12. werden die Relationen ada* und aa* definiert.

Diese Definitionen stellen eine Erweiterung der Definitionen in [PadKucLaw80] und [Ban79] dar. Insbesondere macht die Einführung des Prädikats **sameuse** den Bezug auf eine Ordnung der Zuweisungen, die bei parallelen Blöcken nicht definiert werden kann, überflüssig. Auch kommen die Probleme, die durch die Wiederbenutzung der Speicherzellen entstehen, dadurch deutlicher zum Vorschein. Die Definitionen 9. bis 12. können unverändert erhalten bleiben, wenn man **sameuse** auch für sequentielle Blöcke definiert.

Das folgende Beispiel veranschaulicht die Relation ada:
Seien a und b Bezeichnungen verschiedener Speicherzellen. Dann sind in dem parallelen Block

a:=b, b:=a

die beiden Zuweisungen gegenseitig antidatenabhängig, da die schreibende Referenz der Zellen die nächste Benutzung des "Behälters" darstellt, **sameuse** für lesende und schreibende Referenzen derselben Zelle also = false ist. Eine solche gegenseitige Anti - Datenabhängigkeit heißt z y k l i s c h e Abhängigkeit.

Die Semantik mehrerer, in einem parallelen Block enthaltener, ausgabeabhängiger Statements ist nicht definiert. Da die Kompaktierung im MSS jeweils nur parallele Blöcke betrachtet, brauchen Ausgabeabhängigkeiten nicht berücksichtigt zu werden.

In der Mikroprogrammierung wird zwischen den Relationen da und ada meist nicht unterschieden. Ausnahmen bilden Vegdahl [Veg82] und Mueller [MueVar83]. Wie später zu sehen sein wird, ist der Unterschied in der vorliegenden Arbeit wesentlich.

Bei Anwendung des "single assignment"-Prinzips entfällt gerade die Notwendigkeit, die Relationen ada und aa zu betrachten.

Algorithmen zur Kompaktierung sind seit längerem untersucht worden. Übersichten findet man etwa bei Bode [Bod84] oder bei Mallett [Mal78, DaVLanShrMal81]. Für garantiert optimale Lösungen, d.h. Lösungen mit minimalem Programmspeicherbedarf oder minimaler Programmlaufzeit sind große Rechenzeiten erforderlich. Der Grund liegt in der NP-Vollständigkeit des Problems, die u.a. in [DeWi76, Das80, GarJoh79] bewiesen wird.

Andererseits zeigen aber die Experimente von Mallett, daß einfache Verfahren, deren Rechenzeiten quadratisch von der Zahl der Mikroinstruk

tionen abhängen, in der Regel eine optimale Lösung finden. Eines dieser Verfahren ist das zuerst von Dasgupta und Tartar [DasTar76] angegebene Verfahren des paarweisen Vergleichs.

Das Verfahren des paarweisen Vergleichs (gelegentlich auch first-come, first-served (FCFS) genannt) kompaktiert MO innerhalb von sequentiellen Mikroprogrammblöcken. Die zu einem solchen Block gehörenden MO werden in der Reihenfolge, in der sie von einem Compileroberteil angeliefert werden, zu einer ursprünglich leeren Liste von MI zugefügt. Beim Einfügen einer neuen MO wird angenommen, daß sich alle MO_j, von denen MO_i datenabhängig ist, bereits in der Liste $MI_1,..,MI_m$ der Mikroinstruktionen befinden (die Reihenfolge, in der MO vom Compileroberteil angeliefert werden, muß also eine gemäß der durch Datenabhängigkeiten definierten partiellen Ordnung zulässige totale Ordnung sein). Sei nun t_i der Index der letzten MI, die MO enthält, von denen MO_i datenabhängig ist (t_i=0 falls eine datenabhängige MO nicht existiert). Dann wird die Liste der MI bei t_i+1 beginnend nach MI durchsucht, zu denen sich MO_i ohne Ressource - Konflikte zufügen läßt (zur Definition von Ressource - Konflikten siehe z.B. Mallett [Mal78]). Existiert eine solche nicht, so wird die Liste an ihrem Ende um eine neue MI erweitert und MO_i dieser MI zugewiesen.

Das Verfahren von Dasgupta und Tartar geht, wie die anderen bekannten Kompaktierungsverfahren auch, von einer durch Datenabhängigkeiten definierten partiellen Ordnung der Mikrooperationen aus. Im Falle von MIMOLA ist diese Voraussetzung nicht erfüllt, da aufgrund von Anti-Datenabhängigkeiten zyklische Abhängigkeiten vorliegen können. Daher wird das Verfahren des paarweisen Vergleichs in der vorliegenden Arbeit zwecks Behandlung zyklischer Anti-Datenabhängigkeiten modifiziert. Zyklen sind bisher scheinbar nur im Zusammenhang mit der Programmierung von Parallelrechnern untersucht worden ([PadKucLaw80, Ban79]).

Wie oben wird im folgenden der Fall betrachtet, daß MO_i einer Liste von Mikroinstruktionen MI_1, .., MI_m hinzugefügt werden soll.

Es erweist sich dabei als vorteilhaft, die Reihenfolge der MO i gegenüber dem FCFS-Verfahren gerade umzukehren:: alle M0, die lesende Referenzen auf eine Hilfsvariable enthalten, werden gepackt, bevor die Zuweisung zu dieser Hilfsvariablen gepackt wird.

Auf diese Weise muß man sich beim Lösen von Datenabhängigkeits-Zyklen nicht um bereits gepackte Zuweisungen zu Hilfsvariablen kümmern. Würde man die Zuweisungen zu Hilfsvariablen wie beim Verfahren von Dasgupta früher als die lesenden Referenzen packen, so kann ein Backtracking notwendig werden, wie das folgende Beispiel zeigt:

Sei a ein Array, t0 eine Hilfsvariable und p, q und r seien Variable, von denen unbekannt ist, ob sie gleiche oder ungleiche Werte liefern. s(t0) sei ein t0 enthaltender Ausdruck. Durch das bisherige Packen entstanden sei die Sequenz

```
a[p] := a[q];
t0   := ..  ;
```

Als neue MO sei a[r]:=a[s(t0)] zu packen. Die zyklische Abhängigkeit kann nur durch Zuweisung der rechten Seite zu einer weiteren Hilfsvariablen gelöst werden. Diese neue MO t1:=a[s(t0)] muß spätestens parallel mit der Zuweisung zu a[p] ausgeführt werden, andererseits aber nach der Zuweisung zu t0. Folglich muß die Sequenz geändert werden. Dieser Fall tritt nicht ein, falls die Zuweisung zu a[r] früher als die Zuweisung zu t0 gepackt wird. q.e.d.

Ein weiterer Vorteil dieser Reihenfolge des Packens liegt in der sich dadurch ergebenden Möglichkeit, Zuweisungen zu Hilfsvariablen möglichst dicht vor die lesenden Referenzen zu packen und so die Zahl benötigter Hilfszellen zu reduzieren.

Aus diesen Gründen werden die in **chopping** entstehenden Zuweisungen nicht direkt gepackt, sondern zunächst auf einem Stack abgelegt. Werden die sich in diesem Stack befindenden Anweisungen von oben nach unten dem Packen übergeben, so werden lesende Hilfszellen-Refe

renzen gerade früher als die schreibenden gepackt. Aus diesem Grunz sei dieses Verfahren als LIFO- (last-in, first-out) Verfahren bezeich ne t.

Beim Packen sind die Daten- und Antidaten-Abhängigkeiten zu beachten

Zunächst sollen die Antidaten-Abhängigkeiten betrachtet werden. Diese entstehen dadurch, daß MO_i möglicherweise Zellen beschreibt, deren beim Eintritt in den gegenwärtigen parallelen Block gültiger Inhalt von gewissen anderen MO benötigt wird bzw. dadurch, daß andere MO Zellen beschreiben, deren alter Inhalt von MO_i benötigt wird. Die Berechnung der für MO_i möglichen Plazierungen in der Liste der MI, welche jeweils den Zugriff auf benötigte alte Zelleninhalte sicherstellen, basiert auf den im folgenden definierten Größen $bottom_i$ und top_i :

Def.:

13. $bottom_i$ ist der Index der ersten MI, deren MO (evtl. auch nur potentiell) eine Zelle beschreiben, deren alter Inhalt von MO_i benötigt wird:

$$Nb_i := \{ n \mid \exists\, MO_j \in MI_n : MO_j \ \textbf{ada*}\ MO_i \}$$

$$bottom_i := \begin{cases} \infty & , \text{falls } Nb_i = \emptyset \\ \min\limits_{k \in Nb_i}(k), & \text{falls } Nb_i \neq \emptyset \end{cases}$$

14. top_i ist der Index der letzten MI, deren Mikrooperationen (evtl. auch nur potentiell) den alten Inhalt einer Zelle benötigen, die MO_i beschreibt:

$$Nt_i := \{ n \mid \exists\, MO_j \in MI_n : MO_i \ \textbf{ada*}\ MO_j \}$$

$$top_i := \begin{cases} 0 & , \text{falls } Nt_i = \emptyset \\ \max\limits_{k \in Nt_i}(k), & \text{falls } Nt_i \neq \emptyset \end{cases}$$

Aufgrund der Anti-Datenabhängigkeiten müßte MO_i im Intervall

$$[top_i, bottom_i]$$

plaziert werden. Der mögliche Fall

$$top_i > bottom_i$$

wird später gesondert behandelt.

Als nächstes wird die Relation **da*** betrachtet.

Beschränkt man sich -wie die gegenwärtige Implementierung des MSSauf die Kompaktierung von einzelnen parallelen Blöcken, so kann eine Datenabhängigkeit nur durch Hilfsvariable (temporaries) verursacht werden. Da ferner die schreibenden Referenzen auf Hilfsvariable zuletzt gepackt werden, muß lediglich dafür gesorgt werden, daß diese vor den lesenden Referenzen ausgeführt werden.

Def.:

15. $firstref_i$ ist der Index der MI, welche die erste lesende Referenz der durch MO_i beschriebenen Hilfsvariablen enthält:

$$Nl_i := \{ n \mid \exists MO_j \in MI_n : MO_j \ \mathbf{da^*} \ MO_i \}$$

$$firstref_i \begin{cases} \infty & , \text{ falls } Nl_i = \emptyset \\ \min\limits_{k \in Nl_i}(k) & , \text{ falls } Nl_i \neq \emptyset \end{cases}$$

Damit ist $Nd[i] := [top_i, \min(bottom_i, firstref_i - 1)]$ das Intervall, in das MO_i bei Abwesenheit von Ressource - Konflikten gepackt werden kann. Unter Berücksichtigung dieser Konflikte ist

$$Nf_i := \{n \in Nd \mid 1 \leq n \leq m, \ \mathbf{treetoobig}(MI_n \cup \{MO_i\}) = false \}$$

die Menge der Indizes jener MI aus der Liste $MI_1, .., MI_m$, zu denen MO_i gepackt werden kann.

Falls $Nf_i \neq \emptyset$, so seien $firstfree_i$ und $lastfree_i$ die kleinsten und größten dieser Indices:

$$firstfree_i := \min_{n \in Nf}(n)$$

$$lastfree_i := \max_{n \in Nf}(n)$$

Wegen der anders gearteten Datenabhängigkeiten ist es sinnvoll, zwischen dem Packen von Zuweisungen zu Hilfsvariablen und dem Packen von bereits im Quellprogramm enthaltenen Zuweisungen zu unterscheiden.

Im zweiten Fall müssen Anti-Daten-, aber keine Datenabhängigkeiten beachtet werden. Dies leistet packnontemps:

```
PROCEDURE packnontemps(i);
 BEGIN
    IF Nf_i ≠ ∅
     THEN
       MI_firstfree_i := MI_firstfree_i U {MO_i}
     ELSE
      IF top_i < bottom_i
       THEN
        IF (top_i=0)
         THEN
           Einfügen einer neuen MI={MO_i} vor allen anderen MI
         ELSE
           Einfügen einer neuen MI={MO_i} hinter top_i
        FI
       ELSE packcycles(i);
      FI
    FI
 END;
```

```
PROCEDURE packcycles(i);
 BEGIN
  Spalte MO_i auf in MO_i' und MO_i'', mit
  MO_i' :=Berechnung der rechten Seite der Anweisung und Zuweisung
          des resultierenden Wertes an eine Hilfsvariable
  MO_i'':=Kopieren der Hilfsvariablen in die durch
          MO_i zu beschreibende Zelle
  chopping(i'');
  chopping(i')
 END;
```

Alg. 4.2 Packen von Zuweisungen des Quellprogramms

Zuweisungen zu Hilfsvariablen beschreiben keine Zellen des Quellprogramms. topf ist daher stets 0. Daher enthält die Prozedur packtemps keine Behandlung von Zyklen:

```
PROCEDURE packtemps(i);
 BEGIN
    IF Nf_i≠Ø
     THEN MI_lastfree_i:=MI_lastfree_i U {MO_i}
     ELSE  Einfügen einer neuen MI={MO_i} vor
           min(bottom_i,firstref_i)
    FI
 END
```

Alg. 4.3 Packen von Zuweisungen zu Hilfsvariablen

Exemplarisch sei das Packen an von MOen mit zyklischen Abhängigkeiten folgendem Beispiel gezeigt:

```
a:=b, b:=c, c:=a
```

Dabei bezeichen a,b und c beliebige, aber verschiedene Speicherzellen. Tabelle 4.4 verdeutlicht den Ablauf des Verfahrens unter der Annahme, daß **treetoobig** aufgrund von Ressource-Restriktionen stets true ist:

Inhalt der MI vor dem Packen				zu packen		top	first-ref	bottom
MI_1	MI_2	MI_3	MI_4	i	MO_i			
-	-	-	-	1	a:=b	0	∞	∞
MO_1	-	-	-	2	b:=c	1	∞	∞
MO_1	MO_2	-	-	3	c:=a	2	∞	1
(rekursiver Aufruf)								
MO_1	MO_2	-	-	3''	c:=h	2	∞	∞
MO_1	MO_2	$MO_{3'}$	-	3'	h:=a	0	3	1
$MO_{3'}$	MO_1	MO_2	$MO_{3''}$					

Tab. 4.4 Packen bei zyklischen Abhängigkeiten

Die Reihenfolge der erzeugten Zuweisungen ist also (h Hilfszelle):

```
h:=a;
a:=b;
b:=c;
c:=h
```

Das Verfahren besitzt den Vorteil, daß bei zyklischen Abhängigkeiten nicht in jedem Fall Hilfsvariable zum Aufbrechen der Zyklen benötigt werden. Eigens zum Aufbrechen der Zyklen eingeführte Hilfsvariable können entfallen, falls sich der gesamte Zyklus in eine MI packen läßt oder falls zum Zerlegen der Anweisungen ohnehin geeignete Hilfsvariable eingeführt werden.

Für die drei Varianten der Realisierung von bedingten Anweisungen erfolgen die Syntheseschritte der Abschnitte 4.4 bis 4.6 unabhängig voneinander. Am Ende der Kompaktierung ist die Länge der jeweils benötigten Befehlssequenz bekannt. Gemäß der aktuellen Zielfunktion (minimaler statischer bzw. dynamischer Code, vgl. Absch. 4.3) erfolgt nun die Auswahl der günstigsten Variante.

4.7 Register - Allokation

Auf die Verwaltung der Hilfszellen wurde in den Abschnitten 4.5 und 4.6 nicht eingegangen. Das Synthesesystem unterscheidet hier zwischen (physikalischen) Hilfszellen und (logischen) Hilfsvariablen. Bei der Aufspaltung komplexer Ausdrücke werden zunächst nur Hilfsvariable benutzt. Für diese gilt das "single assignment"-Prinzip, d.h. jeder Variablen wird nur einmal ein Wert zugewiesen. Erst nach der Kompaktierung werden den Variablen Speicherzellen zugeordnet. Die Entscheidung über zu benutzende Hilfszellen wird also so lange wie möglich verzögert, und während der Kompaktierung braucht nicht über freie Hilfszellen Buch geführt zu werden.

Die Zuordnung der Hilfsvariablen zu Hilfszellen soll so erfolgen, daß die Zahl der benötigten Hilfszellen minimal wird. Dieses Zuordnungsproblem wird meist mit Register - Allokation bezeichnet. Allgemein läßt sich das Problem auf das Färbungsproblem, oder -äquivalentauf das Cliquenproblem in Graphen zurückführen (vgl. z.B. [Ber85, MueDud0Ha84]). In Spezialfällen läßt sich eine kellerartige Verwaltung erreichen ([KanLan73]). Dieses ist aber wegen der Ausnutzung gemeinsamer Teilausdrücke im MSS nicht möglich. In [Sch80] wird die RegisterAllokation unter Ausnutzung gemeinsamer Teilausdrücke theoretisch behandelt.

Infolge der Gültigkeit des"single assignment"-Prinzips und der Beschränkung auf die Betrachtung einzelner paralleler Blöcke kann im MSS ein relativ einfaches Verfahren Anwendung finden. Das Verfahren setzt eine Numerierung der Mikroinstruktionen, der Hilfsvariablen und der Hilfszellen voraus. Sei:

V = { 1:vmax } die Menge der Indices der Hilfsvariablen,
Z = { 1:zmax } die Menge der Hilfszellen,
N = { 1:m } die Menge der Indices der MI.

Für $k \in V$ seien die Funktionen **def**(k) und **lastuse**(k) definiert:
def(k) : der Index der MI, die die Zuweisung an die Hilfsvariable v_k enthält,

lastuse(k) : das Maximum der Indices der MI, die ein Lesen der Hilfsvariablen v_k enthalten.

Gesucht ist eine Abbildung location: V -> Z derart, daß eine minimale Zahl von Hilfszellen benutzt wird und es gilt:

Für alle j,k ∈ V, mit j ≠ k :
(location[j] ≠ location[k]) oder
def(j)≥ **lastuse**(k) oder
def(k)≥ **lastuse**(j)

Das folgende Programm gibt das Verfahren zur Bestimmung der Abbildung location wieder:

```
VAR
   location : array[1..vmax] of 1..zmax;      (*Abb. V -> Z*)
   inuse    : array[1..zmax] of boolean;
 FOR z:=1 TO zmax DO inuse[z]:=false OD;
 FOR i:=1 to m DO                          (*Schleife über MI   *)
  FOR k:=1 to vmax DO                      (*Zellen-Freigabe    *)
   IF lastuse(k)=i THEN inuse[location[k]]:=false FI;
  OD;
  FOR k:=1 to vmax DO                      (*Zellen-Reservierung*)
   IF def(k)=i THEN
    z:=1;
    WHILE inuse[z] DO z:=z+1 OD;
    location[k]:=z; inuse[z]:=true;
   FI;
  OD;
 OD;
```

Alg. 4.4 Register - Allokation

In einer Schleife über die generierten Instruktionen (FOR i:=..) werden zunächst die Zellen jener Hilfsvariablen freigegeben,die in der Instruktion MI_i zuletzt gelesen werden. Anschließend werden Zellen für die Hilfsvariablen blockiert, die in Instruktion MI_i definiert werden.

Die Komplexität des Verfahrens ist $O(m * vmax * zmax)$, bzw., da vmax in der Regel proportional zu m und zmax in etwa konstant sein dürfte, $O(m^2)$.

Das Verfahren liefert bei gegebener Liste von Mikroinstruktionen stets eine Lösung mit der minimalen Zahl benötigter Hilfszellen, da die Freigabe von Zellen stets vor der Blockierung erfolgt. Um insgesamt zu einer kleinen Zahl benötigter Hilfszellen zu kommen, plaziert die Kompaktierung schreibende Zugriffe auf Hilfsvariable möglichst dicht vor den lesenden.

Die Darstellung der Algorithmen 4.1 bis 4.4 berücksichtigt den Fall einer unzureichenden Anzahl von Hilfszellen nicht. Dieser Fall kann in den Algorithmen auf folgende Weise behandelt werden: **chopping wird** derart modifiziert, daß alle generierten MO höchstens soviele lesende Referenzen auf Hilfsvariable enthalten, wie Hilfszellen vorhanden sind. Ggf. müssen dazu Hilfsvariable in einen größeren Speicher kopiert werden ("register spilling"). Die Kompaktierung ist so zu erweitern, daß die Zahl der belegten Hilfsvariablen in die Berechnung zulässiger Plazierungen der MO eingeht. Ist keine Plazierung zulässig, so muß ebenfalls eine Zuweisung von Zwischenergebnissen an einen größeren Speicher eingefügt werden. Da das Ziel in der Synthese einer möglichst schnellen Architektur besteht, sollte dem Designer aber in diesem Fall vorgeschlagen werden, eine größere Zahl von Hilfszellen einzubauen.

Es kann vorkommen, daß die Zahl der Ports eines Registerspeichers nicht für die benötigten gleichzeitigen Zugriffe auf Hilfszellen ausreicht. Ein typisches Beispiel ist hierfür ein Registerspeicher mit 2 Leseports und einem Schreibport. Mit einem solchen Speicher allein ist die Synthese für 3-stellige Operationen, wie z.B. für die bedingten Ausdrücke aus Abschnitt 4.3, nicht möglich. Andererseits kann gerade zur Zwischenspeicherung von Bedingungen vorteilhaft von zusätzlichen 1-Bit Registern Gebrauch gemacht werden. Solche Register entsprechen dann dem Condition-Code-Register klassischer Maschinen. Im Gegensatz zu üblichen Condition-Code-Registern erfolgt das Beschreiben aber nicht

als "Seiteneffekt" und kann daher vom Compiler besser genutzt werden (vgl. Myers [Mye78], S.521).

Das MSS unterscheidet daher zwischen 2 Klassen von Hilfszellen, nämlich zwischen Registerspeicher - Zellen und Condition-Code-Registern. Beim Zerlegen der Ausdrücke wird den Hilfsvariablen eine dieser Klassen zugeordnet. Die Register-Allokation ordnet dann eine spezielle Zelle einer Klasse zu.

Häufig werden die Adressen des Registerspeichers bereits vor der Kompaktierung fest vergeben. Durch Mehrfachbenutzung von Adressen entstehen dann aber zusätzliche Anti-Datenabhängigkeiten, die die Möglichkeiten der Kompaktierung unnötig einschränken. Diese Einschränkungen entfallen bei der hier dargestellten "verzögerten" Vergabe von Adressen. Der Grund für den üblichen Ansatz scheint vor allem in der Verwendung von Oberteilen von Maschinencode-Compilern als Oberteile von MikrocodeCompilern zu liegen. Eine Diskussion möglicher Registervergabe-Strategien findet sich bei Mueller et al. [MueDudOHa84].

4.8 Auswahl arithmetisch/logischer Netzwerke

Aufgabe des nächsten Syntheseschrittes ist die Auswahl arithmetisch/ logischer Einheiten (ALUS). Dafür seien vorgegeben:

M ⊂ modules: Die Menge arithmetisch/logischer Modultypen,
k_m : die Kosten des Typs m, m ∈ M,
oplist(m) : die Menge der vom Typ m ∈ M bereitgestellten Operationen.

Im allgemeinen enthält **oplist(m)** die Angaben für mehrere unterschiedliche Operationen. Deshalb heißen diese Bausteine auch M u l t i f unk t i o n s b a u s t e i n e. Es wird im folgenden vorausgesetzt, daß von einem Baustein pro **Instruktion** jeweils nur eine Operation ausgeführt werden soll.

Zu bestimmen ist für jeden Modultyp m die Zahl x_m ($x_m \geq 0$) der zu implementierenden Exemplare derart, daß die Bausteinkosten minimal werden:

$$\sum_{m \in M} x_m * k_m \quad \overset{!}{\longrightarrow} \text{Min.}$$

Diese Minimierung erfolgt unter der Randbedingung, daß für alle MI eine ausreichende Zahl von Bausteinen zur Verfügung steht.

Seien $f_{i,j}$ die Häufigkeiten der Benutzung der Operation j in Instruktion i. Hierbei zählen gemeinsame Teilausdrücke jeweils nur einfach. Ferner sei F_i die Menge der in Instruktion i benutzten Operationen und F_i^* deren Potenzmenge:

$$F_i := \{ j \mid f_{i,j} > 0 \}$$
$$F_i^* := P(F_i)$$

Das Prädikat j **supplied** by m liefere im folgenden den Wert **true** falls die Operation j vom Modul m ausgeführt werden kann und sonst **false**. **supplied** by kann über oplist(m) definiert werden (vgl. Abschn. 5.2.2).
Ein offensichtlich hinreichendes Kriterium für eine ausreichende Zahl von Bausteinen ist das folgende:

$$\forall i, \forall g \in F_i^* : \sum_{g \text{ supplied\_by } m} x_m \geq \sum_{j \in g} f_{i,j} \qquad (4.1)$$

D.h. für alle Instruktionen i und alle Elemente g der Potenzmenge der in MI_i benutzten Operationen ist die Summe der Exemplare jener Modultypen, die mindestens eine benötigte Operation ausführen können, mindestens gleich der Gesamtzahl der ausgeführten Operationen.

Andererseits ist das Kriterium notwendig, denn: gäbe es i und $g \in F_i^*$ derart, daß:

$$\sum_{m} x_m \quad < \quad \sum_{j \in g} f_{i,j}$$

g **supplied_by** m

dann wäre eine Zuordnung von Bausteinen nicht möglich.

Mit $a_{g,m}$:= 0 , falls (g **supplied_by** m) =**false**
1 , falls (g **supplied_by** m) =**true**
folgt aus (4.1) :

$\forall$ i, $\forall$ g $\in F_i^*$:

$$\sum_{m \in M} a_{g,m} * x_m \geq \sum_{j \in g} f_{i,j}$$

Für irgendein festes $g \in F^* = \bigcup_i F_i^*$ können diese

Ungleichungen nur erfüllt sein, wenn

$$\sum_{m \in M} a_{g,m} * x_m \geq \max_i \left(\sum_{j \in g} f_{i,j} \right)$$

Mit $b_g := \max_i \left(\sum_{j \in g} f_{i,j} \right)$ folgt :

$$\forall\, g \in F^* : \sum_{m \in M} a_{g,m} * x_m \geq b_g \qquad (4.2)$$

(4.2) ist nicht nur notwendig, sondern auch hinreichend.

Beweis:

Wegen $F_i^* \subseteq F^*$ für alle i folgt aus (4.2) unmittelbar:

$$\forall\, i,\ \forall g \in F_i^* : \sum_{m \in M} a_{g,m} * x_m \geq \max_i \left(\sum_{j \in g} f_{i,j} \right) \geq \sum_{j \in g} f_{i,j}$$

und damit (4.1).

(4.2) zeigt, daß die Randbedingungen in Form von linearen Ungleichungen darstellbar sind. Damit ist die optimale Wahl der x_m auf eine lineare Optimierung zurückgeführt. In der Implementierung wurde der Gomory I - Algorithmus [Lan78] benutzt.

Beispiel

Ein Programm enthalte die Operationen "+","-" sowie "'*" und bestehe aus zwei Instruktionen mit den Häufigkeiten:

	"+"	"-"	"*"
1.Instruktion:	1	1	0
2.Instruktion:	2	0	1

Es gebe 6 Modultypen mit den folgenden Operationen:

Typ	Kosten	Operationen
A	k_A	"+", "-"
B	k_B	"+", "*"
C	k_C	"+", "-", "*"
D	k_D	"*"
E	k_E	"+"
F	k_F	"-"

Für die erste Instruktion gelten die Ungleichungen:

$$\{"+"\} x_A + x_B + x_C + x_E \geq 1$$
$$\{"-"\} x_A + x_C + x_F \geq 1$$
$$\{"+","-"\} x_A + x_B + x_C + x_E + x_F \geq 2$$

Für die zweite Instruktion gelten die Ungleichungen:

$$
\begin{array}{lllllll}
\{"+" & \} x_A + x_B + x_C + & & x_E & \geq 2 \\
\{"*" & \} & x_B + x_C + x_D & & \geq 1 \\
\{"+","*"\} & x_A + x_B + x_C + x_D + x_E & & & \geq 3
\end{array}
$$

Zusammengefaßt ergibt sich ein System von 5 Ungleichungen:

$$
\begin{array}{lllllll}
\{"-" & \} x_A & + x_C + & & x_F & \geq 1 \\
\{"+","-"\} & x_A + x_B + x_C + & & x_E + x_F & & \geq 2 \\
\{"+" & \} x_A + x_B + x_C + & & x_E & & \geq 2 \\
\{"*" & \} & x_B + x_C + x_D & & & \geq 1 \\
\{"+","*"\} & x_A + x_B + x_C + x_D + x_E & & & & \geq 3
\end{array}
$$

Da sich bei längeren Programmen ein großer Teil der Ungleichungen zusammenfassen läßt, wächst ihre Zahl nicht proportional zur Länge der Programme. Bei Bedarf ließen sich auch noch einzelne redundante Ungleichungen (wie hier z.B. die zweite) eliminieren.

Die arithmetisch/logischen Bausteine werden in den verschiedenen CADSystemen sehr unterschiedlich ausgewählt.

Bei Huang [Hua81] erfolgt die Auswahl bereits durch den Benutzer beim Erstellen der Spezifikation. Dadurch muß der Benutzer aber sehr frühzeitig eine Entscheidung treffen, für die ihm eigentlich die Basis fehlt. Andererseits kann der Benutzer auf diese Weise aber die Kosten für arithmetisch/logische Bausteine reduzieren. Huangs CADSystem zerlegt nämlich komplexe Anweisungen in Teilschritte derart, daß die vorgegebenen ALUs für die Bearbeitung jedes Teilschritts ausreichen.

In einer Version des CMU-DA-Systems [TseSie83] werden zunächst überhaupt keine vordefinierten Modultypen berücksichtigt. Vielmehr wird analysiert, welche Operationen in Programmen nie gleichzeitig vorkommen. Durch die anschließende Lösung eines Cliquenproblems für Graphen erhält man eine Liste zu benutzender Mehrfunktionsbausteine. Die Liste der

von diesen Bausteinen ausführbaren Operationen ist das Ergebnis der Lösung des Cliquenproblems. Es wird also nicht berücksichtigt, daß gewisse Modultypen möglicherweise in einem Katalog fertiger Bausteine enthalten sind. Dadurch fehlt auch die Möglichkeit, die unterschiedlichen Kosten von vorhandenen Bausteinen bei Optimierungen zu beachten. Außerdem kann die Liste ausführbarer Operationen unsinnige Kombinationen enthalten (z.B. Operationen auf unterschiedlichen Datentypen bzw. Wortbreiten innerhalb desselben Bausteins). Treten sehr komplexe Anweisungen auf, so erzeugt das System eine große Zahl von ALUS, da eine Zerlegung von Anweisungen in eine Reihe von Teilschritten aufgrund einer Kostengrenze für ALUS nicht möglich ist.

Der "Module binden'' [Leise] des CMU-DA-Systems erlaubt die Benutzung vordefinierter Modultypen, enthält aber keine globale Optimierung.

Die lineare Programmierung wurde bereits von anderen Autoren zur global optimierenden Lösung des Modul-Auswahlproblems vorgeschlagen.

Bei Hafer [HafPar83] sind die entsprechenden Ungleichungen Teil eines größeren Ungleichungssystems. In diesem wird für jedes **Exemplar** eines Modultyps eine binäre Entscheidungsvariable benötigt. Die dadurch entstehende große Zahl von Variablen ist ein Grund für die unakzeptablen Rechenzeiten des Verfahrens. Bei der hier vorgestellten Methode wird dagegen nur eine Variable pro Modultyp benötigt.

Für die Auswahl von Bausteinen auf der Gatterebene wurde ebenfalls die lineare Programmierung vorgeschlagen (siehe [SahBha83, Kod72]). Diese Methode scheint aber nicht benutzt worden zu sein, da auf der Gatterebene zusätzliche Probleme auftreten. So muß auf dieser Ebene im Gegensatz zur RT-Ebene unbedingt berücksichtigt werden, daß die Operationen NAND, AND, NOR u.s.w. auf mehrere Arten durcheinander ersetzbar sind. Dadurch wird je Operation wieder eine binäre Variable benötigt. Wegen der großen Zahl von Gatter-Operationen wurde die Methode dadurch nicht praktikabel.

Das in dieser Arbeit vorgestellte Verfahren ist global optimierend und bereitet trotzdem keine Komplexitätsprobleme. Es erlaubt die Benutzung von Bibliotheken vordefinierter Bausteine, berücksichtigt deren Kosten bei der Optimierung, und kann komplexe Ausdrücke in eine Reihe von Teilschritten zerlegen, falls dies aufgrund der für arithmetische Operationen geltenden Kostenrestriktionen erforderlich ist. Mit der Auswahl der Bausteine ist ferner nicht gleichzeitig auch entschieden, welcher Baustein eine im Programm vorkommende Operation ausführen wird. Dadurch bleibt ein Freiheitsgrad, der zur Minimierung von Verbindungen genutzt werden kann. Neu an diesem Verfahren ist u.a. auch die Behandlung von Multifunktionsbausteinen über Potenzmengen.

4.9 Zuordnung von Ressourcen

In diesem Syntheseschritt werden den Operationen im Programm HardwareRessourcen zugeordnet. Lese- und Schreiboperationen werden Speicherports, arithmetisch/logischen Operationen werden ALUS und Konstanten werden Befehlfelder zugeordnet.

Als Beispiel sei eine aus folgendem Statement bestehende MI gegeben (dabei seien SR und SH Speicherbausteine):

```
SR(0):= SH(abs)
```

Durch die Zuordnung von Ressourcen werden die benötigten Verbindungen zwischen den Ressourcen bestimmt. Wählt man z.B. ein bestimmtes Speicherport und ein bestimmtes Registerfeld des Befehlsformats als Ressourcen für den Ausdruck SR(0), so impliziert dies eine Verbindung von diesem Befehlsfeld zum Adreßeingang des gewählten Ports.

Bei der Beurteilung der Qualität der erzeugten Rechnerstruktur spielen nun die Verbindungskosten eine entscheidende Rolle. Bei der Realisierung als integrierter Schaltkreis kann die für Verbindungen benötigte Fläche wesentlich größer sein als die für Transistoren und Widerstände. Ebenso können Laufzeiten auf den Leitungen die Verzögerungen innerhalb

der aktiven Bauelemente bei weitem überwiegen. Wünschenswert wäre also ein Verfahren, welches die Zuordnung der Ressourcen derart vornimmt, daß der Bedarf an Chipfläche und die Laufzeit auf den Leitungen minimal werden. Hierin ist als Teilproblem das Problem der automatischen Erzeugung von Schaltkreis-Layouts enthalten. Da bereits für dieses Teilproblem keine befriedigenden Lösungen existieren, ist eine optimale Lösung des Gesamtproblems in absehbarer Zeit nicht zu erwarten.

Es wird daher eine Aufspaltung des Problems vorgenommen. Vereinfachend wird angenommen, daß zunächst nur die **Zahl** der Verbindungen minimiert werden soll. Die Ergebnisse laufender Arbeiten zur Schätzung des Flächenbedarfs aufgrund von Rechnerstruktur - Beschreibungen [Zim85] sollen später zur Verbesserung dieses Optimierungskriteriums genutzt werden.

Spezielle Fälle des Problems der Minimierung der Verbindungen lassen sich auf Standardprobleme des Operations - Research abbilden. Entarten die Datenflußbäume z.B. zu einer linearen Kette, so enthält das Problem noch das Handlungsreisenden-Problem [Neu75] als Sonderfall. Damit ist für Lösungsverfahren eine exponentielle Komplexität zu erwarten.

Aufgrund von Vergleichen verschiedener Ansätze stellt sich heraus, daß ein sehr einfaches Backtracking-Verfahren in der Regel die geringste Rechenzeit benötigt. Dieses Verfahren basiert auf der Idee, den Operationen des Programms probehalber Ressourcen zuzuordnen und jeweils die Zahl, fehlender Verbindungen zu zählen. Liegt diese Zahl über dem gegenwärtig erlaubten Limit, so wird diese Zuordnung wieder verworfen und die nächste mögliche Zuordnung getestet. Das Limit wird mit einer getrennt berechneten unteren Schranke der Zahl fehlender Verbindungen initialisiert und jeweils um eins erhöht, falls keine weitere Zuordnung möglich ist.

Das Verfahren ist im Prinzip in der Lage, die Verbindungen über mehrere Instruktionen hinweg zu optimieren. Implementiert wurde das Verfahren jedoch zunächst nur für die Optimierung innerhalb einzelner

MI. Um dennoch zu einer möglichst guten Lösung zu kommen, werden die MI nach ihrer Komplexität (d.h. im wesentlichen nach der Zahl der Operationen) sortiert. Komplexe Instruktionen erzeugen meist den größten Teil der von einfachen Instruktionen benötigten Verbindungen.

Die Vorteile dieses Verfahrens sind:

a. Gleichzeitig mit den zu realisierenden Verbindungen wird die Zuordnung der Ressourcen zum Programm berechnet und das Programm kann einfach an die Hardware gebunden werden.
b. Die erste gefundene Lösung ist immer eine (für die einzelne MI) optimale Lösung.

Nachteilig ist die mit der Menge gleichartiger Ressourcen (z.B. mit der Zahl der Speicherports) exponentiell wachsende Komplexität. Auf sehr parallele Architekturen läßt es sich daher nicht anwenden.

In einer Veröffentlichung von Parker [ParKurMli84] wird die Existenz von Verbindungen mittels 0/1-Entscheidungsvariablen modelliert. Auf diese Weise lassen sich Verbindungen mittels linearer Programmierung minimieren. Das Modell von Parker ist durch die Einbeziehung weiterer Entwurfsentscheidungen leider so komplex, daß eine Lösung in vertretbarer Zeit nicht erwartet werden kann. Es ist geplant, zu untersuchen, welche Rechenzeiten sich beim Abspalten der Verbindungsoptimierung in Parkers Modell ergeben.

Sowohl beim letztgenannten Ansatz als auch bei dem oben beschriebenen Backtracking-Verfahren muß geprüft werden, ob eine Verbindung zwischen zwei Ressourcen existiert. Da in MIMOLA die Angabe von Bitbereichen möglich ist, müßte dieser Vergleich eigentlich auch diese Angabe testen. Auf diesen Vergleich wird jedoch aus zwei Gründen bewußt verzichtet, nämlich erstens, um den Algorithmus zu vereinfachen und zu beschleunigen und zweitens, um die Lokalität der Verbindungen zu erhöhen.

Würde der Vergleich Bitbereiche einschließen, so wären die folgenden Verbindungslisten gleich teuer:

```
a. quelle.(15:0)  -> senkeA.(15:0);
   quelle.(16)    -> senkeB.(17);
b. quelle.(15:0)  -> senkeA.(15:0);
   quelle.(16)    -> senkeA.(17);
```

Für die Liste b dürfte sich jedoch in der Regel einfacher ein Layout erzeugen lassen.

Die aktuelle Implementation des Verfahrens enthält noch folgende Erweiterungen:

1. Gemeinsamen Teilausdrücken können gleiche Ressourcen zugeordnet werden. Die Zuordnung von gleichen Ressourcen kann aufgrund der Zahl verfügbarer Hardwarebausteine notwendig sein. Andererseits führt eine solche Zuordnung häufig nicht zu einer minimalen Zahl von Verbindungen. Der implementierte Algorithmus ordnet daher gemeinsamen Teilausdrücken zunächst gleiche Ressourcen zu. Falls es aber zur Minimierung der Verbindungen erforderlich ist, wird diese Zuordnung wieder verworfen.
2. Die Kommutativität von Operatoren wird ausgenutzt, um die Zahl der Verbindungen zu minimieren. Durch diese zusätzliche Optimierung **sinkt** interessanterweise die mittlere Komplexität des Backtracking-Verfahrens, da das oben erwähnte Limit seltener erhöht werden muß.

Eine vergleichbare Optimierung enthält das Programm SUGAR des CMU-DA-Systems [RajTho85]. In SUGAR wird die Kommutativität bereits vor der Zuordnung von Hardware-Ressourcen ausgenutzt ("operator canonization"). Dieser Ansatz führt zu einer etwas globaleren Optimierung als im MSS. Auf der anderen Seite kann nicht berücksichtigt werden, welcher Baustein oder welches Port einer Operation zugeordnet wird. Beide Methoden können aber kombiniert werden.

Das Backtracking-Verfahren stellt gegenüber dem älteren MIMOLA-Software-System MSS1 ([Zim80, Mar79, Mar80a]) eine bedeutende Verbesserung dar. So hat Nowak mit Hilfe des MSS1 einen Prozessor synthetisiert und anschließend in einem zeitraubenden interaktiven Prozeß die Zahl

der Verbindungen um ca. 50% auf 50 Leitungsbündel (je 16 Bit) reduziert. Das neue System liefert ohne manuellen Eingriff einen Rechner, der bei 48 Leitungsbündeln praktisch die gleiche Leistung bietet [Now84].

4.10 Nachoptimierung

Nach der Zuordnung von Ressourcen ist eine globale Verbindungsmatrix bekannt.

Mit Hilfe dieser Matrix lassen sich verschiedene Nachoptimierungen entwickeln. Sie haben das Ziel, Nachteile durch die Verwendung nicht global optimaler Verfahren wieder auszugleichen. Diese Optimierungen arbeiten auf der erzeugten Struktur und entsprechen damit etwa der "Peephole"-Optimierung aus dem Compilerbau.

Als Beispiel sei etwa das Duplizieren von billigen Moduln mit dem Ziel der Verbindungsoptimierung genannt.

Gegenwärtig implementiert ist eine Optimierung, die davon Gebrauch macht, daß die unter 4.8 erzeugten Verbindungen einzelne don't care Bits enthalten können. Gedacht ist an eine Erweiterung dieses Syntheseschrittes um ein Mini-Expertensystem, mit dem aufgrund von Entwurfsregeln Transformationen auf der erzeugten Struktur durchgeführt werden können.

4.11 Erzeugung von Quellenauswahl-Netzen

Im Schritt 4.9 bzw. 4.10 werden noch keine Quellenauswahl - Netze (Multiplexer, Bus-Treiber) erzeugt. Zur Vermeidung von Leitungskonflikten müssen diese nachgetragen werden.

Sofern keine Busse generiert werden sollen, reicht es aus, Multiplexer zu erzeugen.

Werden vom Benutzer Busse gewünscht, so muß zunächst deren Zahl minimiert werden. Ein mögliches Verfahre, dazu beschreiben Torng und Wilhelm [TorWi177]. Entsprechende Arbeiten für das MSS werden vorbereitet.

Weiterhin müssen die Steuereingänge der übrigen Bausteine noch geeignet verschaltet werden. Standardmäßig werden hierfür eigene Mikroprogrammfelder vorgesehen (sog. "direct encoding" oder "single level encoding" [AgrRau76]).

Schon ohne Anwendung von Techniken zur Wortbreitenreduktion (vgl. z.B. [Bod84]) liegt die erzeugte Codemenge in der gleichen Größenordnung wie die für klassische Maschinen erzeugte. So generiert der PASCALCompiler für die Siemens 7.760 für das Programm "mergesort" nach Wirth [Wir75, S. 113] 936 Bytes Code (Ein/Ausgabe nicht gerechnet). Das Synthesesystem erzeugte in einem Experiment trotz fester Befehlslänge je nach Entwurfsvorgaben zwischen 755 und 1143 Bytes Code.

4.12 Binden der Programme an die Hardware

Zum Zwecke der Simulation, der Erzeugung binären Codes und der Beurteilung der Rechnerstruktur werden die eingegebenen Programme noch vollständig an die Hardware gebunden. Das bedeutet, daß jeder benutzte Multiplexer, jede benutzte ALU u.s.w. explizit im Programm vorkommen. U.a. werden die in Schritt 4.9 zugeordneten Ressourcen und die in Schritt 4.11 erzeugten Quellenauswahl-Netze in das Programm einkopiert.

Die vollständig gebundenen Programme enthalten in Form der den Befehlsfeldern zugewiesenen Werte den gesamten Programmcode. Dieser Code wird mit Hilfe der in Absenitt 6.4 vorgestellten Komponente MSSB aus den gebundenen Programmen extrahiert. Damit ist als Ergebnis eines Syntheselaufes auch gleich das Maschinenprogramm für die generierte Hardware bekannt. Anhang B enthält den Programmcode und die resultierende Rechnerstruktur für das Beispiel des Abschnitts 4.2.

4.13 Anwendungen

Mit dem Synthesesystem des MSS1 wurden in einer Reihe von Diplomarbeiten und im Rahmen einer Dissertation Rechnerentwürfe durchgeführt. Die Anwendungen wurden dabei u.a. charakterisiert durch Routinen der numerischen Mathematik [Mar80], eine Mischung aus Betriebssystems- und typischen PASCAL-Routinen [Now84], Probleme aus der kommerziellen Bildverarbeitung [Sch83] und einen Interpreter für den Befehlssatz der Serie IBM /370 [Krü80].

In den ersten drei Fällen konnte die Rechenleistung gegenüber vergleichbar komplexen Rechnern deutlich gesteigert werden. Im ersten Fall lag die Steigerung etwa bei einem Faktor 20 im Vergleich zum Prozeßrechner MODCOMP II. Im zweiten Fall wurde ein Faktor zwischen 2 und 3 im Vergleich zu Rechnern des Typs Siemens 7.760 erreicht. Im dritten Fall würde die Realisierung des entworfenen Rechners den Übergang von der bisherigen Batch-Verarbeitung zum Dialogbetrieb erlauben. In allen drei Anwendungen war die Dichte des Codes derjenigen auf konventionellen Anlagen vergleichbar.

Im Falle des Entwurfs für den IBM-Befehlssatz schließlich wurde in etwa die Geschwindigkeit des Rechners 7.755 der Fa. Siemens erreicht. Die Entwurfszeit und die Länge des Mikroprogramms konnten im Vergleich dazu aber erheblich gesenkt werden.

Aus diesen Beispielen erkennt man, daß deutliche Steigerungen der Rechenleistungen bei Beibehaltung der Technologie nur durch eine Abkehr von den bisherigen Befehlssätzen möglich sind.

Anwendungsmöglichkeiten des in dieser Arbeit beschriebenen Synthesesystems sollen im folgenden exemplarisch vorgestellt werden. Als Beispiel dient dazu das bei Wirth [Wir75] angegebene Programm **mergesort** zum Sortieren ganzer Zahlen (bei einem echten Entwurf würde man ein umfangreicheres Programm verwenden).

Mergesort wurde manuell von PASCAL nach MIMOLA übertragen und -soweit offensichtlich möglich- parallelisiert. Mittels der Komponente MSSR des

MSS wurde das Programm auf die ungebundene RT-Ebene abgebildet. Unter Benutzung des Simulators des MSS wurden die Transformationen überprüft und für einen Satz von 20 Zufallszahlen als Daten die Ausführungshäufigkeiten bestimmt. Diese wurden in das Programm **mergesort** einkopiert.

Die Komponente MSSR erlaubt es, zwischen der absoluten und der relativen Adressierung von Variablen zu wählen. Im ersten Fall werden Variable durch Ausdrücke der Form SH(const), im zweiten Fall durch Ausdrücke der Form SH(SR(const) "+" const') ersetzt. const und const, sind dabei natürliche Zahlen. Da **mergesort** keine rekursiven Prozeduren enthält, sind beide Adressierungsarten anwendbar. Abbildung 4.6 zeigt für beide Adressierungsarten die generierte Zahl von Befehlen in Abhängigkeit von unterschiedlichen Ressource-Restriktionen. Als Restriktionen dienen dabei die Zahl der Hauptspeicherports, die Zahl der arithmetisch/ logischen Einheiten (ALUS) und die Zahl der für Daten- und Adreßkonstanten benutzten (16-Bit-) Befehlsfelder. Aufgeführt sind nur solche Kombinationen dieser drei Zahlen, die zu einer einigermaßen gleichmäßig ausgelasteten Hardware führen. In der Abbildung dargestellt ist die Zahl der dynamisch durchlaufenen Befehle, errechnet aus den einkopierten Häufigkeiten.

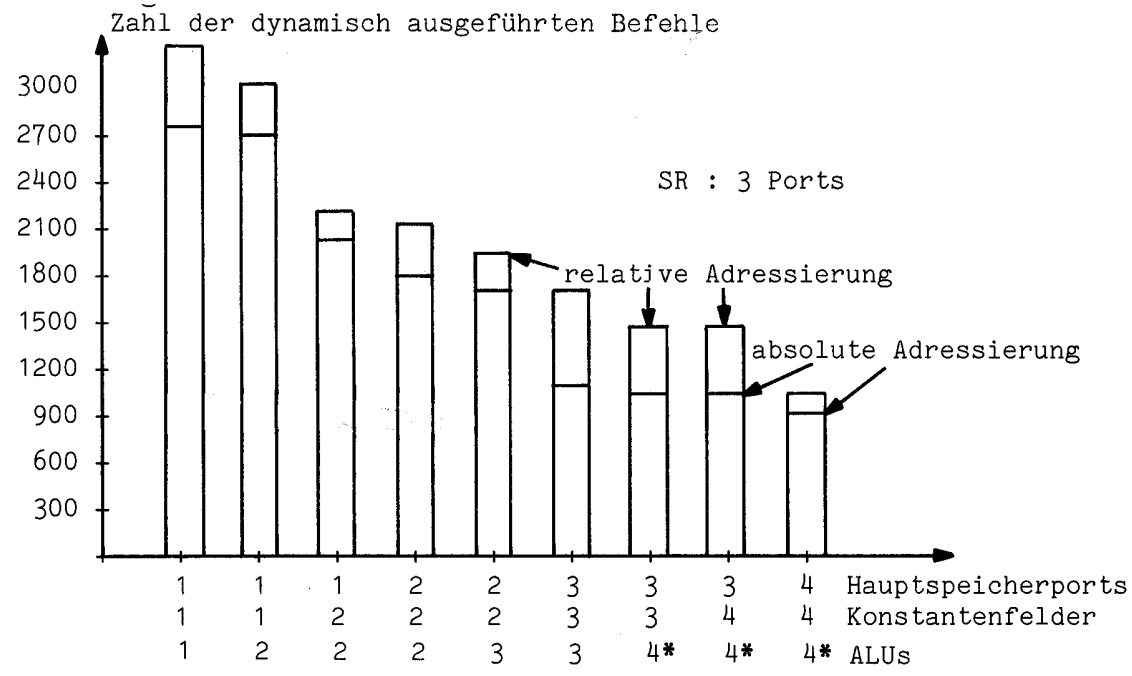


Abb. 4.6 Einfluß von Adressierungsarten für Variable

(*: Die vierte ALU wird nur bei relativer Adressierung erzeugt)

Auch bei sehr parallelen Hardwarestrukturen ergeben sich noch große Unterschiede. Im Extremfall verkürzt:, die absolute Adressierung die Zahl der ausgeführten Befehle gegenüber der relativen Adressierung auf 65%.

Anschließend an die Abbildung auf das RT-Niveau kann optional eine Ersetzung von Zugriffen auf den Hauptspeicher durch Zugriffe auf den Registerspeicher erfolgen. Für diesen Zweck können Zellen des Registerspeichers mittels einer LOCATIONS FOR OPTIMIZATION-Vereinbarung freigegeben werden. Die Zahl der freigegebenen Zellen bestimmt wesentlich die Zahl der ersetzten Speicherzugriffe. Die Zahl der Ersetzungen hat nun ihrerseits Einfluß auf die Länge des Maschinenprogramms. Die Zusammenhänge für **mergesort** zeigt Abb. 4.7:

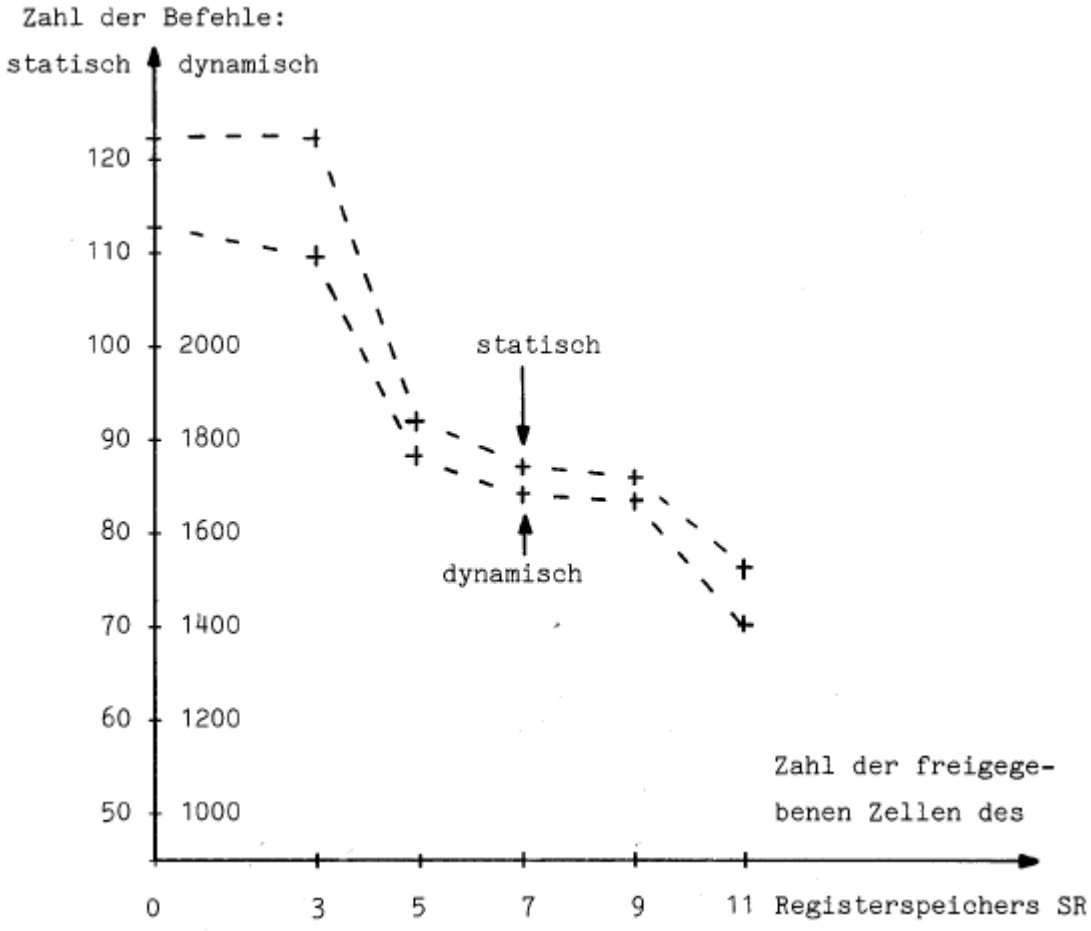


Abb. 4.7 Einfluß der Zahl der zur Ersetzung von Haupt speicherzugriffen freigegebenen Zellen von SR

Die "statische Zahl der Befehle" ist die Zahl der vom Übersetzer generierten Befehle.

Als Parameter liegen der Abbildung 4.7 zugrunde : 1 Hauptspeicherport, 3 Registerspeicherports, 2 Konstantenfelder für Adressen oder Direktoperanden, 2 ALUS, relative Adressierung der Variablen.

Bei mehr als 5 freigegebenen Zellen reduziert sich die Zahl der Befehle nur noch wenig. Der Grund liegt darin, daß dann die ALUS zu einem Engpaß werden. Die Zahl der Ports des Registerspeichers dagegen bildet keinen wesentlichen Engpaß, da sich für einen Registerspeicher mit 5 Ports weitgehend die gleiche Abbildung ergibt.

Dargestellt wurde hier lediglich die Abhängigkeit der Zahl der Befehle. Der Effekt auf die Laufzeit dürfte größer sein, sofern der Hauptspeicher langsamer ist als der Registerspeicher.

Dieses Beispiel zeigt, wie das MSS benutzt werden kann, um eine für eine bestimmte Anwendung sinnvolle Größe des Registerspeichers zu bestimmen, wenn ein Teil der Variablen in Registern gehalten werden soll.

Neben der Realisierung von bedingten Anweisungen durch bedingte Sprünge erlaubt das MSS auch bedingte Ausdrücke und bedingtes Laden (vgl. Abschn. 4.3). Optional lassen sich die beiden letzten Realisierungsmöglichkeiten unterdrücken. Abb. 4.8 zeigt, welchen Einfluß dies auf die Zahl der Befehle hat. Als einzige Ressource-Beschränkung wird dabei die Zahl der Hauptspeicherports benutzt. Die Daten basieren auf einer unbeschränkten Zahl von Ports des Registerspeichers und einer absoluten Adressierung der Variablen.

Um den **relativen** Einfluß besser vergleichen zu können, ist die Ordinatenachse logarithmisch geteilt. Es ist deutlich erkennbar,wie der Einfluß mit zunehmender Parallelität wächst.

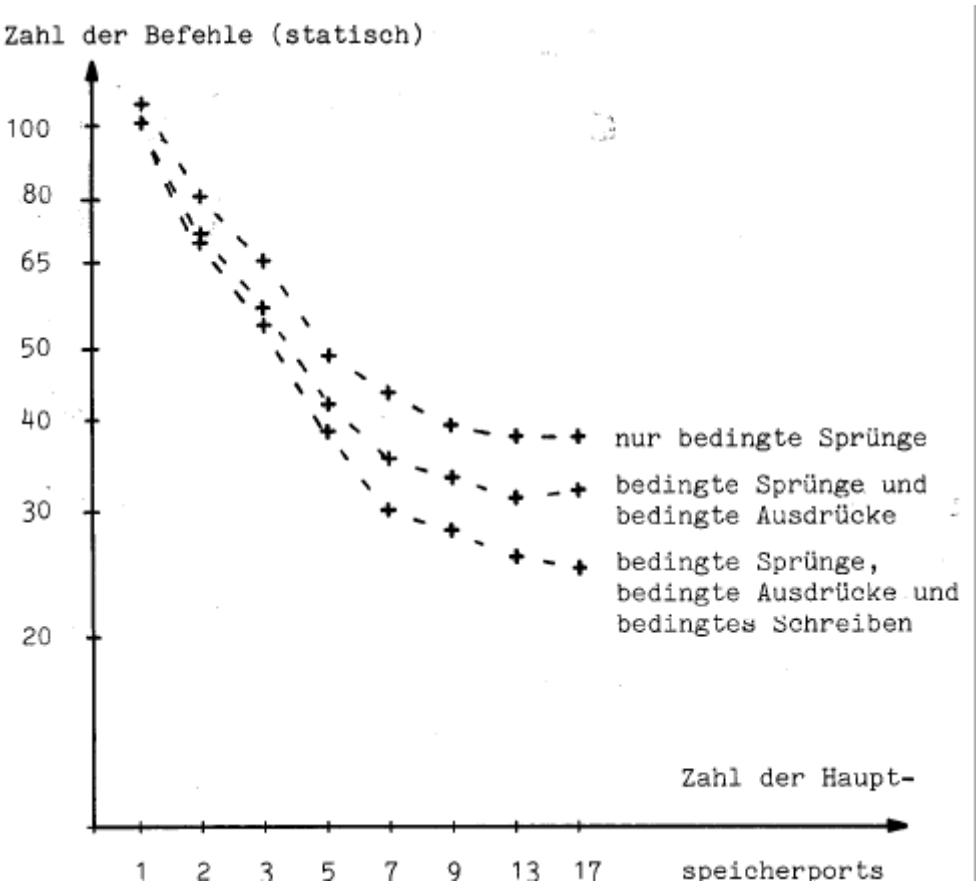
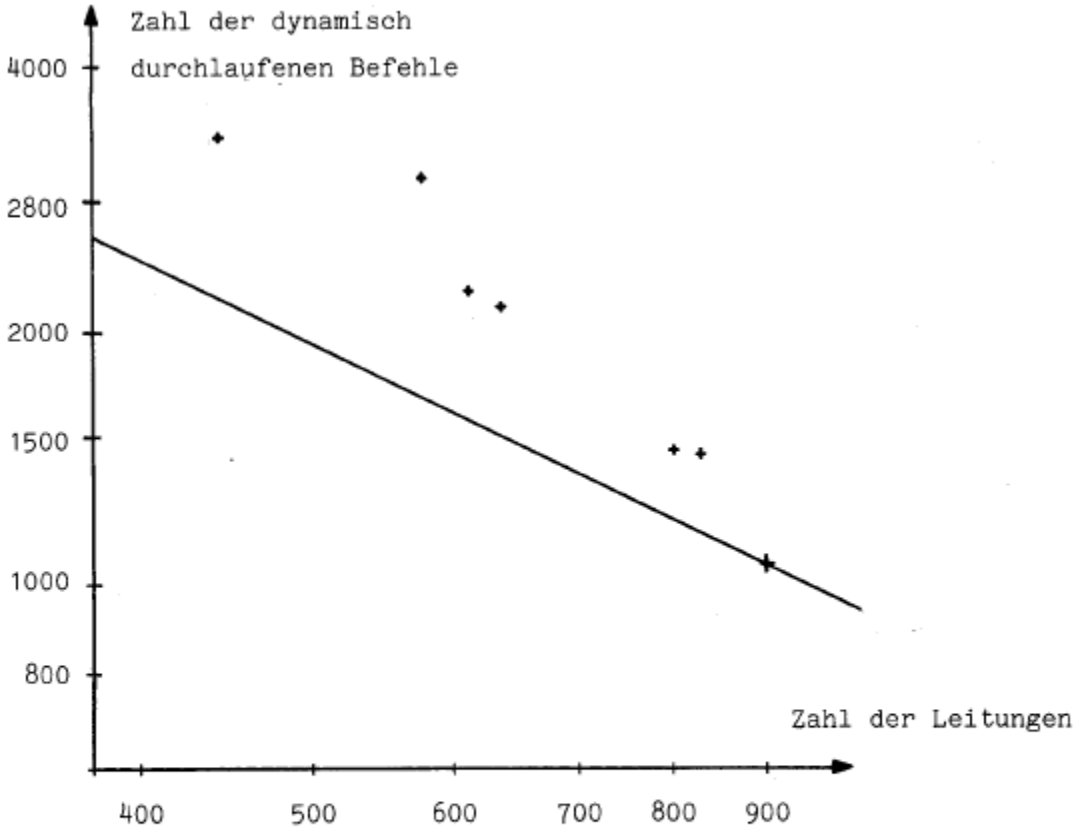


Abb. 4.8 Einfluß der Realisierung bedingter Anweisungen

Schaltungstechnisch relativ einfach zu realisieren sind Registerspeicher mit einem Lese- und einem Schreib/Leseport. Bei Verwendung eines solchen Speichers wird bei Register/Register - Operationen einer der Operanden überschrieben und es werden Kopieroperationen zwischen Registern erforderlich. Die Zahl der Befehle, die im Vergleich zu einem Speicher mit einem Schreib- und zwei Leseports zusätzlich erforderlich sind, kann durch eine Vorgabe entsprechender Speicher untersucht werden.

Bei **mergesort** ergeben derartige Studien einen Zuwachs der ausgeführten Befehle um höchstens 7 % und im Mittel um etwa 4%. Diese Werte dürften sich durch Optimierungen innerhalb des MSS noch reduzieren lassen. Da Register/Register-Befehle auch nur eine kurze Ausführungszeit benötigen, ist der Effekt für die Gesamt-Laufzeit ohnehin geringer. Die Werte deuten daher an, daß sich der Einsatz von 3-Port Registerspeichern in der Tat selten lohnt.

Bislang wurde der Hardware-Bedarf lediglich durch Hardware - Module und Befehlsfelder charakterisiert. Abb. 4.9 zeigt Zusammenhänge zwischen der Zahl der benötigten Hardware - Verbindungen und der Zahl der Befehle. Die Zahl der Verbindungen schließt alle Steuerleitungen und alle an Multiplexer angeschlossenen Leitungen ein.



Parameter:

Ports SH :	1		1	1	2		3	3	4
I-Felder :	1		1	2	2		4	3	4
ALUs :	1		2	2	2		4	4	4

Registerspeicherports : 3; Variablenadressierung : relativ;

Abb. 4.9 Einfluß der Zahl der Leitungen

Für die Punkte auf der Geraden ist das Produkt aus der Zahl der Befehle und der Zahl der Leitungen konstant. Benutzt man die Zahl der Leitungen als alleiniges Kostenkriterium, können also parallele Strukturen ein optimales Kosten/Leistungsverhältnis besitzen.

Im folgenden sollen nun die Programmlaufzeit und die Codemenge der mit dem MSS erzeugten Entwürfe mit einem konventionellen Rechner verglichen werden.

Die Ausführungszeit eines Befehls der mit dem MSS erzeugten Rechner sei 1 Mikrosekunde. Dieser Wert läßt sich leicht erreichen. Mit den Ausführungshäufigkeiten für 20 zu sortierende Zahlen ergeben sich so die Zeiten der Tabelle 4.5. Als Vergleich wird die Ausführungszeit des entsprechenden PASCAL-Programms auf einer Rechenanlage des Typs Siemens 7.760 benutzt. In beiden Fällen ist die Ein/Ausgabe nicht berücksichtigt.

	MSS - Entwurf		SIEMENS 7.760
Hauptspeicherports	1	4	PASCAL V 3.1A10
Konstantenfelder	1	4	(keine
ALUS	1	4	Laufzeittests)
Laufzeit [Millisek.]	3,3	1,0	6,2
Programmcode [Bytes]	1143	755	936

Tab. 4.5 Vergleich von Laufzeiten und Codemengen

Es könnte nun der Eindruck entstehen, daß dieses günstige Ergebnis durch die Konstruktion von Spezialrechnern für die jeweils eingegebenen Programme entsteht. Dies ist aber nicht der Fall, da voll programmierbare Rechner erzeugt werden und z.B. spezielle Konstanten (außer der Null) nicht hartverdrahtet werden.

Um zu verdeutlichen, daß keine Spezialrechner entstehen, soll untersucht werden, wie sich die Hardware ändert, wenn die Spezifikation um Programme erweitert wird. Speziell soll geprüft werden, ob sich die Zahl der Verbindungen erhöht.

Zu diesem Zweck werden zunächst die Syntheseschritte bis zur Auswahl von ALUS für die folgenden drei Programme bei gleichen RessourceBeschränkungen getrennt durchgeführt:

1. Das oben benutzte Programm **mergesort**,
2. das Progamm **gomoryI** [Lan78] zur linearen Programmierung und
3. einen ursprünglich in BASIC geschriebenen Logiksimulator [Der83].

Durch Zusammenfügen von TREEMOLA-Files kann man erreichen, daß zunächst die Liste der Verbindungen für eines der Programme generiert wird und anschließend alle für die weiteren Programme zusätzlich erforderlichen Verbindungen aufgelistet werden. Als erstes Programm sollte dabei kein sehr kleines Programm benutzt werden. Verwendet man dafür **gomoryI**, so wird für die beiden übrigen Programme lediglich eine 1 -Bit -Leitung zusätzlich angelegt. **gomoryI** erzeugt also bis auf diese Ausnahme alle benötigten Leitungen (ohne etwa alle Bausteine untereinander zu verbinden).

Voraussetzung dafür ist allerdings, daß für alle Programme die gleiche Adressierungsart für Variable angewandt wird. Benutzt man für das aus BASIC entstandene Programm eine absolute Adressierung und für die aus ALGOL und PASCAL entstandenen Programme eine relative Adressierung, so werden insgesamt 4 neue Leitungsbündel angelegt. U.a. werden die Befehlsfelder direkt mit dem Adresseneingang des Hauptspeichers verbunden.

Das Beispiel zeigt, daß bei ausreichend großen Eingabeprogrammen keine Spezialrechner entstehen. Lediglich gewisse Programmeigenschaften, wie z.B. die Art der Variablenadressierung oder das Vorkommen arithmetischer Operationen werden ausgenutzt. Um zu erreichen, daß gewisse Operationen auf jeden Fall (unabhängig von ihrem Vorkommen in Programmen) ausführbar sind, kann man diese Operationen ggf. den Progammen der Spezifikation zusetzen.

5. Codeerzeugung für eine vorgegebene Rechnerstruktur

5.1 Ansatz, Einordnung des Verfahrens

In der Einleitung wurde bereits erwähnt, daß der Codegenerator im MSS zwei wichtige Aufgaben erfüllt, nämlich erstens die Unterstützung von Entwurfsiterationen beim Rechnerentwurf und zweitens die Erzeugung von Code für vollständig entworfene Rechner. Insbesondere die erste Aufgabe erfordert eine extrem kurze Umrüstzeit des Codegenerators auf andere Zielmaschinen. Außerdem macht sie es notwendig, eine Form der Zielmaschinenbeschreibung zu benutzen, die dem Hardware-orientierten Designer verständlich und mit dem Synthesesystem kompatibel ist. Daher geht auch der Codegenerator aus von einer Beschreibung der Hardware durch Bausteine und deren Verbindungen untereinander.

Eine gute Übersicht über existierende, von der Zielmaschine unabhängige Codegeneratoren bietet Ganapathi [GanFisHen83]. Darüber hinaus ist aus Kiel insbesondere die Arbeit von Schmidt [Sch84] zu erwähnen. Die angegebenen Verfahren sind aber bislang nicht zur Erzeugung von Mikrocode benutzt worden.

Übersichten über ältere Verfahren im Bereich maschinenunabhängiger Mikrocode-Generatoren bieten Bushell [Bus78] und Sint [Sin80]. Neuere Arbeiten liegen von Baba [BabHag81], von Vegdahl [Veg82,Veg82a,Veg83] und von Mueller et al. [MueVar83, MueVarAl184] vor.

Babas MPG-System enthält ein interessantes Verfahren zur Anordnung von Mikrobefehlen im Mikroprogrammspeicher, welches insbesondere für Rechner mit komplizierter Adressierung des Folgebefehls gedacht ist. Im übrigen läßt sich Babas Methode aber nicht für das MSS verwenden, da sie nicht ausschließlich die Komponenten der RT-Verhaltensebene benutzt.

Im Zusammenhang mit dem MSS besonders interessant ist die Dissertation von Vegdahl, da sie ähnlich wie das MSS auf Programmtransformationen basiert. Sie stellt eine Erweiterung der Arbeiten von Cattell [Cat78]

dar. Vegdahls Codegenerator ist nicht vollständig maschinenunabhängig, da zur Erzeugung von Konstanten noch Routinen jeweils von Hand geschrieben werden müssen ([Veg82],S.71). Für die Mikrocode-Erzeugung selbst ist das akzeptabel. Einem Hardware-Entwickler, der den Codegenerator in Entwurfsiterationen benutzt, ist das nicht zuzumuten. Generell dürfen im MSS beim Übergang auf neue Maschinen keine Änderungen am Compilercode nötig sein. Damit sind die Anforderungen an die Maschinenunabhängigkeit größer als allgemein üblich. Andererseits kann gerade die Erzeugung von Konstanten im MSS einen erheblichen Teil der Übersetzungszeit ausmachen.

Bei Mueller wird wie beim MSS ein Maschinenmodell benutzt, das die Bedeutung der Hardware-Module und deren Verbindungen untereinander hervorhebt. Die Liste der Verbindungen muß bei dem Verfahren von Mueller manuell vorverarbeitet werden, bevor sie in den Übersetzer eingegeben wird. Diese Vorverarbeitung dient beispielweise der Auswahl eines von mehreren möglichen Datenwegen zum Transport von Daten zwischen zwei Hardware-Bausteinen. Dadurch geht evtl. bereits bei der Maschinenbeschreibung die Optimalität verloren.

Beim Vergleich der Arbeiten von Vegdahl und Mueller mit dem MSS wird ein wesentlicher Unterschied im Ansatz deutlich: Bei beiden Autoren ist eine gewisse manuelle Aufbereitung der Hardwarebeschreibung notwendig, um zu einem Codegenerator zu gelangen. Ein Vorteil dieser Aufbereitung liegt in einer möglichen Beschleunigung der Übersetzung. Aufgrund des Anwendungsbereiches ist die Codeerzeugung beim MSS so konzipiert, daß diese Aufbereitung entfallen kann.

Takagi [Tak84] beschreibt ein Verfahren, welches speziell der Überprüfung manueller Eingriffe bei Entwurfsiterationen dient. Dieses Verfahren kontrolliert, ob RT-Programme auf einer RT-Hardwarestruktur ablaufen können. Das Verfahren wird daher als "Verifier" bezeichnet. An eine Codeerzeugung mit diesem Verfahren ist aber offenbar nicht gedacht.

Zu Beginn der Arbeiten am Codegenerator wurde erkannt, daß sich die Codeerzeugung als Parsen eines Eingabeprogramms bezüglich einer

der Hardwarebeschreibung ableitbaren Grammatik darstellen läßt. Dieser Ansatz wurde unabhängig voneinander mehrfach entdeckt [EvaGoe0fe79, AncLidMerPay69]. Der Ansatz wurde im MSS aber nicht benutzt, da die Grammatik hochgradig mehrdeutig und schwer zu parsen ist und außerdem noch Ressource-Konflikte ignoriert. Letztlich stellt die Arbeit von Vegdahl eine Möglichkeit zur Behandlung dieser Probleme dar.
Maschinenunabhängige Codegeneratoren benötigen eine modellhafte Beschreibung der Zielmaschine. Unter den denkbaren Modellen befinden sich u.a. die Modelle von Mallett [Mal78], Sint [Sin81] und Dasgupta [Das84].
Für das MSS liegt das Modell durch die Notwendigkeit der Kompatibilität mit dem Synthesesystem fest. Die Darstellung einer Maschine durch Hardwarebausteine und deren Verbindungen hat auch den Vorteil, daß Maschinenbeschreibungen von hardwareorientierten Mikroprogrammierern selbst erstellt werden können. In einem Anwendungsfall dauerte die Einarbeitung dazu'ca. 1 Woche.

Die meisten Compiler erzeugen direkt binären Code. Im Gegensatz dazu ist es im MSS vorteilhaft, zunächst vollständig gebundene Programme zu erzeugen. Aus diesen kann der binäre Code anschließend abgeleitet werden. Vollständig gebundene Programme werden im MIMOLA-System u.a. zur Analyse der Hardwareauslastung und zur Simulation von Maschinenprogrammen benötigt. Darüberhinaus sind sie (im Vergleich zu Bitstrings) besser lesbar.

Unter Benutzung der Definitionen aus Kap.2 läßt sich die Übersetzung eines Programms p in den Code einer Maschine

h=(modules, parts, ifields, connections, templocs)

kennzeichnen als Durchführung einer Abbildung:

(p, h) -> p,

wobei alle im Programm p' enthaltenen Zuweisungen vollständig gebunden sind und innerhalb der parallelen Blöcke aus p' keinerlei Konflikte bezüglich Hardware-Ressourcen auftreten (vgl. Abschn. 5.2.6).

In einer Hardwarestruktur, die eine große Zahl von arithmetisch/ logischen Einheiten und Datenwegen enthält, gibt es in der Regel mehrere Möglichkeiten, eine bestimmte Anweisung oder einen Teil einer Anweisung in ein Maschinenprogramm zu übersetzen.

Def.:

1. Jedes zu einem Ausdruck oder einer Anweisung äquivalente Maschinenprogramm heißt V e r s i o n des Ausdrucks oder der Anweisung.

z. Sei n ein Knoten eines Datenflußgraphen. Dann bezeichnet **versions(n)** die Menge der diesem Knoten zugeordneten Versionen.

Versionen sind stets vollständig gebunden. Die zugehörigen Ausdrücke oder Anweisungen können ungebunden, partiell gebunden oder vollständig gebunden sein. Partiell gebundene Programme dienen vor allem dem bewußt maschinenabhängigen Programmieren, sowie in Einzelfällen zur Beschleunigung der Ubersetzung und zur Reduktion des Speicherbedarfs. Die Eingabe vollständig gebundener Programme ist dagegen nur in Ausnahmefällen sinnvoll.

Vor der eigentlichen Codeerzeugung durchläuft die Eingabe die Systemoberteile wie in Kap. 3 beschrieben.

Der Codegenerator muß neben der Versionserzeugung auch eine Mikroprogramm-Kompaktierung enthalten. Aufgrund der Komplexität beider Teilprobleme sind die Versionserzeugung und die Versionsauswahl und Kompaktierung im MSS in zwei getrennten Programmen MSSV und MSSC realisiert. Damit ergibt sich für den Codegenerator eine Struktur nach Abb. 5.1 .

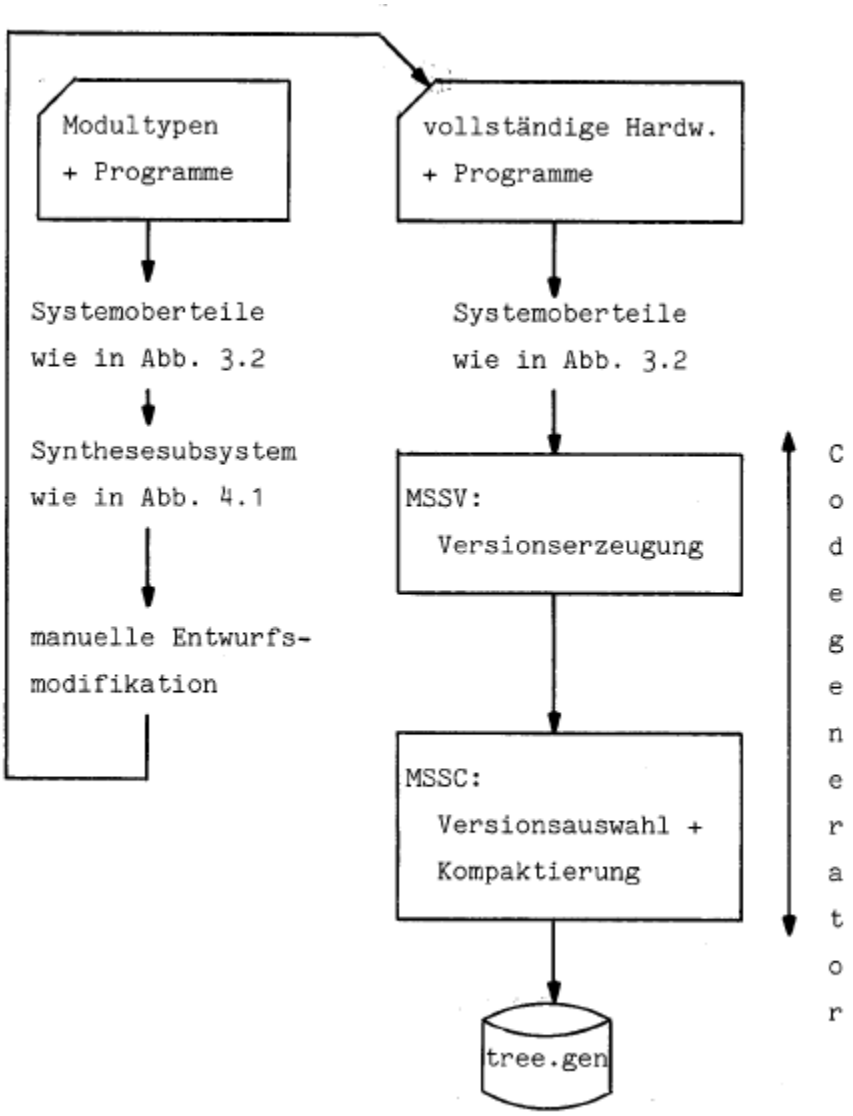


Abb. 5.1 Codegenerator-Subsystem und seine Umgebung

Während der Versionserzeugung läßt sich leider nicht vorhersehen, welche Version ein optimal kompaktiertes Programm ergibt. Dazu folgendes Beispiel:

Gegeben sei ein Rechner mit zwei bidirektionalen Hauptspeicherports und ein aus folgenden zwei Anweisungen bestehendes Programm:

```
SH(O):=SH(1) "+" SH(2), SH(3):=SH(4) "+" SH(5);
```

Prinzipiell sind für jede der beiden Zuweisungen zwei Versionen denkbar nämlich eine Version der Art

```
SR(0):= SH(y) "+" SH(z);
SH(x):= SR(0)
```

sowie eine Version der Art

```
SR(0):= SH(y);
SH(x):= SR(0) "+" SH(z)
```

mit x=0,3; y=1,4; z=3,5.

Welche der beiden Versionen günstiger ist, kann erst in der Kompaktierung entschieden werden. Dies zeigt das zu den beiden Anweisungen gehörige optimale Mikroprogramm, bei dem für die beiden Zuweisungen unterschiedliche Arten von Versionen benutzt werden müssen:

```
L1: SR(0):=SH(1) "+" SH(2);
L2: SH(0):=SR(0),    SR(0):=SH(4);
L3:                  SH(3):=SR(0) "+" SH(5);
```

Folglich muß MSSV im Prinzip alle möglichen Versionen abliefern und die Auswahl von Versionen muß MSSC überlassen bleiben. Auf vorgenommene Einschränkungen hinsichtlich der Zahl generierter Versionen wird im nächsten Abschnitt eingegangen.

5.2 Versionserzeugung

5.2.1 Blöcke, Zuweisungen

Die Versionserzeugung basiert auf einem Vergleich der Programmbäume mit der Hardwarebeschreibung.

Zur Veranschaulichung soll in diesem Abschnitt das folgende RT-Programm als Beispiel benutzt werden

```
BEGIN
 SH(0).(15:0) := SH(1).(15:0) "+" 8.(15:0)
END
```

Um die Beschaltung der Kontrolleingänge einfacher erzeugen zu können, benutzt MSSV eine gegenüber der Synthese modifizierte Form der Datenflußbäume. In dieser Form kommen die Operationen als Blätter statt als innere Knoten vor, und innere Knoten stellen "Platzhalter" für die Bausteine dar, welche die Operation ausführen sollen. Innere Knoten enthalten bei READ- und LOAD-Operationen den Speichernamen.

In gebundenen Programmen können sie zusätzlich noch vom Benutzer definierte Portnamen enthalten.

Im Falle arithmetisch/logischer Operationen enthalten innere Knoten bei ungebundenen Programmen hier als "?" dargestellte "Joker", die auf jene Bausteine $d \in$ **parts** "passen", die die jeweilige Operation ausführen können. Im gebundenen Programm wird "?" durch einen der "passenden" Bausteine ersetzt. Werden als Eingabe gebundene Programme benutzt, so enthalten diese Knoten bereits den vom Benutzer vorgegebenen Bausteinnamen.

Auf diese Weise ergibt sich für das obige Programm beispielsweise der Datenflußbaum nach Abb. 5.2.:

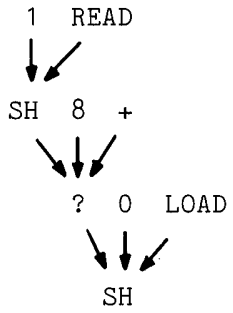


Abb. 5.2 Datenflußbaum des Beispielprogramms

Die Erzeugung von Versionen soll anhand der Hardware nach Abb. 5.3 demonstriert werden. Die vollständige:- MIMOLA-Beschreibung dieser Hardware ist in Anhang C enthalten.

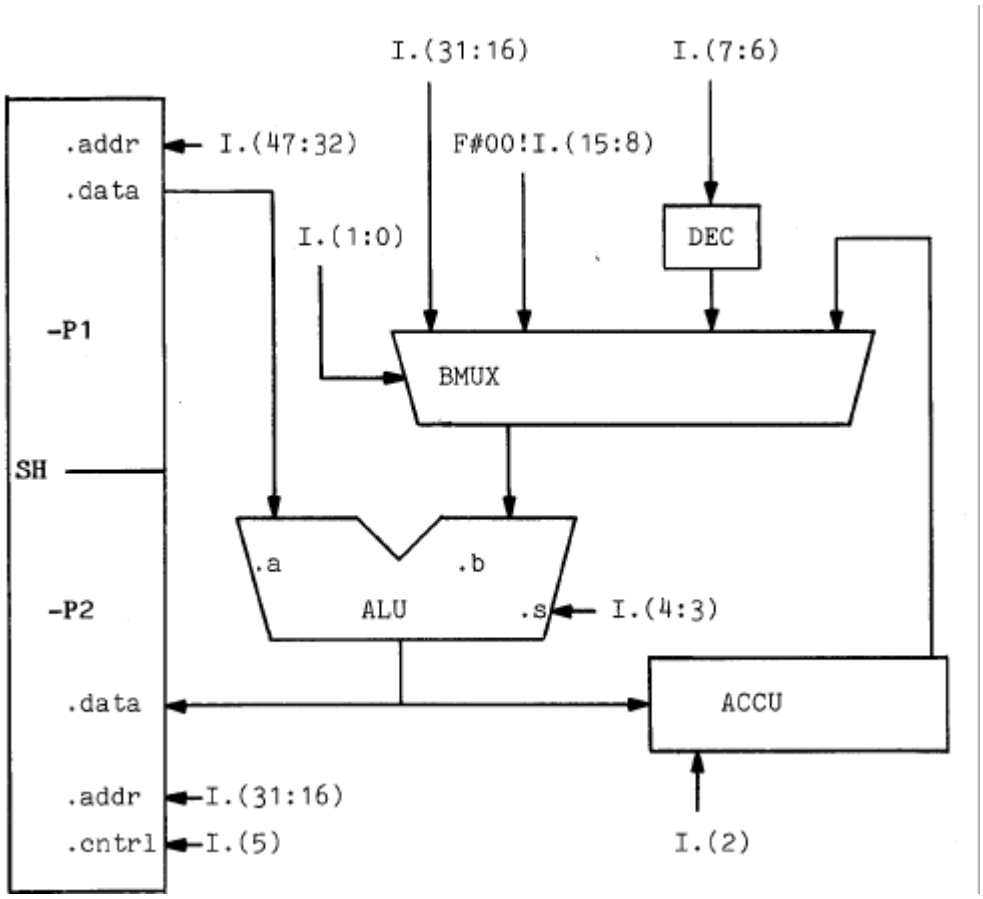


Abb. 5.3 Beispielhardware

"I" steht als Synonym für den Ausgang des Bausteins, der den jeweils aktuellen Befehl enthält, also in der Regel für den Ausgang des Befehlsregisters (vgl. Kap. 2).

Aus Platzgründen kann die nachfolgende Behandlung der Versionserzeugung nur einen groben Überblick über die Grundideen des Verfahrens geben. Auf Vollständigkeit muß verzichtet werden.

Der Kontrollfluß der Versionserzeugung entspricht dem Kontrollfluß eines recursive-descent Compilers. Die Prozeduren **parblock und assign**

ment werden pro parallelem Block und Zuweisung je einmal gerufen (zur Definition von **treeclass** vgl. Abschn. 3.2, Def. 2)

```
PROCEDURE parblock (pb : treeclass);
 BEGIN
  FOR ALL n ∈ sons (root(pb)) DO
   IF nodetype(n)=ifstatementtype THEN ifstatement(tree(n))
   ELSE
    IF nodetype(n)=assignmenttype THEN assignment (tree(n))
    FI
   FI
  OD
 END;

PROCEDURE assignment (s : treeclass);
 BEGIN
  expression(s);
  (*weitere Aktionen betreffen Fehlermeldungen, die Ausgabe
  fertiger Versionen und die Verwaltung der Hilfszellen*)
 END;
```

Alg. 5.1 Behandlung von parallelen Blöcken und Zuweisungen

Im folgenden bezeichnen pgm_source und pgm sink stets Knoten eines Datenflußbaumes. In einer bestimmten Prozedur repräsentiert pgm source eine "Datenquelle" und pgm sink eine "Datensenke". Vgl. dazu das Beispiel in Abb. 5.4:

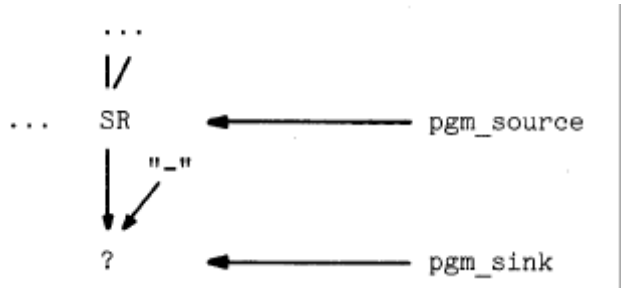


Abb. 5.4 Erläuterung von pgm sink und pgm source

pgm sink und pgm source verbindet eine Kante des Datenflußbaumes; es sei denn, der Knoten "unterhalb" von pgm source repräsentiert eine Konkatenation. In diesem Fall befindet sich pgm sink zwei Stufen "unterhalb" von pgm source (vgl. Alg. 5.2).

Die Prozedur **expression** erzeugt die Aufrufe an Code generierende Prozeduren. Die-Aufrufe erfolgen bei einem Durchlauf durch den Datenflußbaum von den Blättern zur Wurzel und von rechts nach links.

Blätter von Datenflußbäumen sind entweder Konstanten (Integer, Label, Strings), Operationen oder Referenzen auf Befehlsfelder. Da Referenzen auf Befehlsfelder bereits ein vollständig gebundenes Programmstück darstellen, brauchen für sie keine Versionen erzeugt zu werden.

```
PROCEDURE expression (e : treeclass);
 BEGIN
  FOR ALL pgm_source ∈ nodes(e), SEQUENCE: depth-first,right-to-left DO
   IF nodetype(pgm_source) ∈ {integertype,labeltype,stringtype}
    THEN constant(pgm_source)  FI;
   IF nodetype(pgm_source)=operationtype
    THEN operation(e,pgm_source)
   FI;
   IF nodetype(pgm_source)=catenationtype THEN catbundling(pgm_source)
    ELSE
     bundling(pgm_source);
     IF pgm_source ≠ root(e) THEN
      Definiere help durch (pgm_source,help) ∈ vert(e);
      IF nodetype(help)=catenationtype
       THEN Definiere pgm_sink durch (help,pgm_sink) ∈ vert(e);
       ELSE Definiere pgm_sink durch (pgm_source,pgm_sink)∈ vert(e);
      FI;
      path(e,pgm_source,pgm_sink)
   FI FI
  OD
 END;
```

Alg. 5.2 Behandlung von Ausdrücken

5.2.2 Operationen

Sei nodetype(pgm_source)=operationtype und die durch pgm source beschriebene Operation. sei f=(fid,farity,flength)

Die Prozedur **operation** erzeugt für alle Möglichkeiten, die Operation f durch einen der Bausteine der Hardware auszuführen, Versionen für die Beschaltung des Kontrolleingangs. Die Versionen **versions(pgm** sink) werden um die erzeugten Versionen erweitert.

```
PROCEDURE operation(e:treeclass; pgm_source : nodeclass);
 BEGIN
  Sei f=(fid,farity,flength) die durch pgm_source
  beschriebene Operation;
  Definiere pgm_sink durch (pgm_source,pgm_sink) evert(e);
  FOR ALL d ∈ parts
   DO
    IF (part_id(pgm_sink)=?) OR (part_id(pgm_sink)=part_id(d))
    THEN
     FOR ALL r ∈ oplist(d) DO
      IF (op_symbol(r)=fid)  und  (arity(r)=farity)  und
         ((length(r)=flength) oder
          ( (length(r)>flength) und f kann durch
            Operationen auf längeren Daten ersetzt werden
          )
         )
        THEN
         erzeuge Baum mit Wurzel w und darüber liegendem
         Knoten c;
         c repräsentiert die Konstante code(r);
         w repräsentiert den Baustein mit Namen part_id(d), d.h.
           part_id (w) = part_id(d),
           port_id (w) = port_id(output(r)),
           range_id(w) = range_id(output(r)),
           var_id  (w) = var_id(pgm_sink).
```

```
        IF code(r)≠don't care (%XX..X) THEN
         rufe expression(tree(w)), um die Konstante
         c am Kontrolleingang von part_id(d) zu erzeugen.
        FI;
        erweitere die Versionen von pgm_sink um alle
        Versionen, die den benötigten Code erzeugen.
      FI;
     FI;
    OD;
   FI;
  OD;
 END;
```

Alg. 5.3 Versionserzeugung für Operationen

Es sei hier noch einmal darauf hingewiesen, daß in der Hardwarebeschreibung und im Programm vorkommende, gleich benannte Operationen auf jeden Fall die gleiche Bedeutung besitzen. Zur Definition der in **operation** enthaltenen Aufrufe von **op symbol**, **length**, **arity und code** siehe Abschnitt 2.1.2, Def. 5 und 6.

Für den Aufruf mit dem "+" enthaltenden Knoten des Programms nach Abb. 5.2 erzeugt **operation** beipielsweise die Version:

```
ALU.s (I(2).(1:0))
```

ALU.s bedeutet dabei, daß es sich um eine Beschaltung des Kontroll-eingangs handelt.

Die durch die Bausteine ausgeführten Operationen sind jeweils für eine bestimmte Länge der Bitstrings definiert. Manche dieser Operationen können auch für weniger lange Bitstrings genutzt werden. Hierzu zählen insbesondere logische Operationen und Operationen auf vorzeichenlosen Zahlen. So kann beispielsweise die "AND"-Verknüpfung zweier 16 Bit langer Daten durch einen für 32 Bit lange Daten vorgesehenen Baustein erfolgen. Hiervon wird in der Prozedur **operation** Gebrauch gemacht.

5.2.3 Erzeugung von Konstanten

Für jede als Blatt vorkommende ganze Zahl wird die Hardware-Beschreibung nach Möglichkeiten zur Erzeugung dieser Zahl durchsucht. Außer durch 0-stellige Operationen können Konstanten noch durch spezielle Belegungen von Befehlsfeldern erzeugt werden. Dabei können Konstanten sowohl vollständig von einem einzelnen Hardwarebaustein oder Befehlsfeld als auch von Konkatenationen von diesen gebildet werden.

Werden Konstanten durch Dekoderoperationen erzeugt, d.h. durch Bausteine, deren Steuereingang selbst wieder mit einer Konstanten beschaltet werden muß, so sorgt ein rekursiver Aufruf von **expression** für deren Generierung.

```
PROCEDURE constant(pgm_source : treeclass);
 BEGIN
  clength:=Länge der Konstanten in Bits, berechnet aus
           range_id(pgm_source) (vgl. Abschn.3.2, Def.4);
  FOR ALL i ∈ ifields DO
   IF length(i)=clength THEN
      erweitere die Versionen von pgm_source um einen Knoten
      w. w entspricht der TREEMOLA-Darstellung des mit dem
      Wert von pgm_source belegten Feldes i:
      nodetype(w)=instrtype,
      value(w)=Wert der Konstanten pgm_source,
      range_id(w)=Bitnummern des Befehlsfeldes,
        (im Falle nicht zusammenhängender Nummern wird statt
         eines einzelnen Knotens eine Konkatenation erzeugt)
      var_id(w)=var_id(pgm_source)
   ELSE IF length(i)<clength THEN
      verfahre wie im Fall passender Längen für
      die niederwertigsten length(i) Bits; schiebe die
      Konstante um length(i) Stellen logisch nach rechts;
      rufe constant für die neue Konstante und
      konkateniere die Teilversionen
   FI FI;
  OD;
```

```
FOR ALL d ∈ parts DO
 FOR ALL r ∈ oplist(d) DO
  IF arity(r)=0 und
   der Wert von pgm_source paßt zu dem durch die
   0-stellige Operation erzeugten Wert THEN
    IF length(r)=clength THEN
      IF code(r) ≠ don't care THEN
        rufe expression rekursiv, um code(r)
        zu erzeugen
      FI;
      erweitere die Versionen von pgm_source um einen Baum
      mit Wurzel w und ggf. Knoten zur Erzeugung von
      code(r);
      w repräsentiert den Baustein part_id(d).
     ELSE IF length(r)<Länge der Konstanten THEN
     verfahre analog zum Fall i ∈ ifields
    FI FI;
  FI
 OD
OD
END;
```

Alg.5.4 Versionserzeugung für Konstanten

Als Versionen der Konstanten 8.(15:0) aus obigem Beispiel ergeben sich für die Hardware nach Abb. 5.3 u.a.:

1. ein mit dem Wert 8 belegtes 16 Bit langes Befehlsfeld:

```
I(8).(31:16),
```

z. ein ebenfalls mit dem Wert 8 belegtes 8 Bit langes Befehlsfeld, konkateniert mit 8 fest verdrahteten 0-Bits (F#00):

```
F#00.(7:0)!I(8).(15:8),
```

3. ein Dekoder DEC, gesteuert durch die Bits (7:6) des Befehls:

```
DEC(I(3).(7:6)).(15:0)
```

Durch die Möglichkeit des Konkatenierens von Teilversionen ist die Komplexität der Konstanten-Erzeugung zunächst einmal exponentiell bezüglich der Länge der Konstanten in Bits. Zur Reduktion der Komplexität sind im MSS mehrere Maßnahmen vorgesehen. So kann die Zahl der konkatenierten Teilversionen nach oben und die Zahl der Bits in einer Teilversion nach unten beschränkt werden. Weitere Maßnahmen werden im Abschnitt 5.2.10 beschrieben.

Die Menge der erhaltenen Versionen wird der jeweiligen Konstanten zugewiesen.

5.2.4 Verbindungen

Für jedes der Abb. 5.4 entsprechende Paar (pgm source, pgm sink) von Knoten des Datenflußbaumes ruft **expression** die Prozedur **path**. Aufgabe von **path** ist es, Wege von den Wurzeln der Versionen von pgm source zu den Eingängen jener Bausteine zu finden, die aufgrund der auszuführenden Operation auf pgm sink "passen".

Die auszuführende Operation sei dargestellt durch den Operationsbezeichner f = (fid, farity, flength).

farity wird bestimmt durch die Zahl der Söhne des Knotens pgm sink. Einer dieser Söhne enthält das Operationssymbol fid ("-" in Abb.5.4). flength kann anhand der Komponenten **range id** der Söhne bestimmt werden.

Die Menge aller Bausteine, die die Operation f ausführen können, sei im folgenden mit **matching parts(pgm** sink) bezeichnet:

matching_parts(pgm_sink) :=
{d | d ∈ **parts**,
(**part_id**(pgm_sink)=?) OR (**part_id**(pgm_sink)=**part_id**(d)),
∃ r ∈ **oplist**(d) mit **op_symbol**(r)=fid, **arity**(r) = farity,
length(r) = flength oder (**length**(r)>flength und
f kann durch Operationen auf längeren Daten ersetzt werden) }

Es hängt von der Operation f und dem Knoten pgm source ab, zu welchen Eingängen der Bausteine aus **matching parts** (pgm sink) Wege gesucht werden müssen. Handelt es sich bei pgm source um die Adresse einer Lese- oder Schreiboperation, um zu schreibende Daten oder um einen Kontrollcode, so sind selbstverständlich nur Wege zu Adressen-, Datenoder Kontrolleingängen der Module aus **matching parts** (pgm sink) zu suchen. Bei arithmetisch / logischen Operationen müssen Wege zu jenen Eingängen gesucht werden, an denen das betreffende Modul die Argumente der Operation erwartet. Zusätzlich lassen sich Aufrufe von **expression** so einschränken, daß nur Wege zu bestimmten Eingängen gesucht werden. Die Menge der Eingänge, zu denen Wege gesucht werden, sei im folgenden mit **matching Inputs** (pgm sink) bezeichnet.

matching_inputs(pgm_sink) :=
{ (**part_id**(d), s) | ∃ d, ∃ r∈**oplist**(d) :
r "paßt" gemäß der Definition von **matching_parts** zu f,
s ∈ **inputs**(r),
s bezeichnet den Eingang für das n-te Argument von r
und pgm_source bezeichnet das n-te Argument von f}

Um zu berücksichtigen, daß die Eingänge bei kommutativen Operationen vertauscht werden dürfen, werden die Operationslisten **oplist(d)** vor Beginn der Übersetzung automatisch erweitert. So werden diese Listen im Falle einer Deklaration a "+" b um einen Eintrag b "+" a ergänzt.
Allgemeiner : Zu jeder in der Hardwarebeschreibung aufgeführten Operation wird die zugehörige "konverse" Operation nachdeklariert. Zueinander konvers sind Operationen wie "<" und ">", für die gilt:

a "<" b g.d.w. b ">" a.

Die Kommutativität ergibt sich gerade als Spezialfall der zu sich selbst konversen Operationen.

Zunächst versucht path, durch Aufruf von **dirpath** einen direkten Weg von den Wurzeln der Versionen von pgm-source zu den Eingängen aus **matching inputs(pgm** sink) zu finden.

```
PROCEDURE path(t : treeclass; pgm_source,pgm_sink : nodeclass);
 BEGIN
  dirpath (pgm_source,pgm_sink);
  IF weitere Wege sind möglich
   THEN
    viapath (t, pgm_source, pgm_sink);
    temppath (t, pgm_source, pgm_sink);
  FI
 END;
```

Alg. 5.5 Ablauf der Suche nach Verbindungen

```
PROCEDURE dirpath(pgm_source, pgm_sink : nodeclass);
  BEGIN
   FOR ALL vtree ∈ versions(pgm_source) DO
    vroot:=root(vtree);
    FOR ALL c ∈ connections DO
     IF part_id(source(c))=part_id(vroot)                    und
        part_id(sink(c)) ∈ matching_parts(pgm_sink)          und
        c führt zu einem Eing.∈ matching_inputs(pgm_sink)    und
        die Bitnummern "passen" (s. u.)
      THEN
       ergänze die Versionen von pgm_sink um einen Baum mit
       Wurzel sink(c) und darüber befindlicher Kopie von vtree
     FI
    OD
   OD
  END;
```

Alg. 5.6 Versionserzeugung für direkte Verbindungen

Wege werden von **dirpath** nur dann akzeptiert, wenn auch die Bitnummern im Programm und in der Verbindungsli$te zueinander passen. Generell gilt, daß vorhandene Datenwege nicht immer in der vollen Breite genutzt werden müssen. Auch Unterbereiche davon sind zulässig.

Die erzeugte neue Version enthält **sink(c)** als Wurzel und darüber vtree, d.h. den "alten" Versionsbaum.

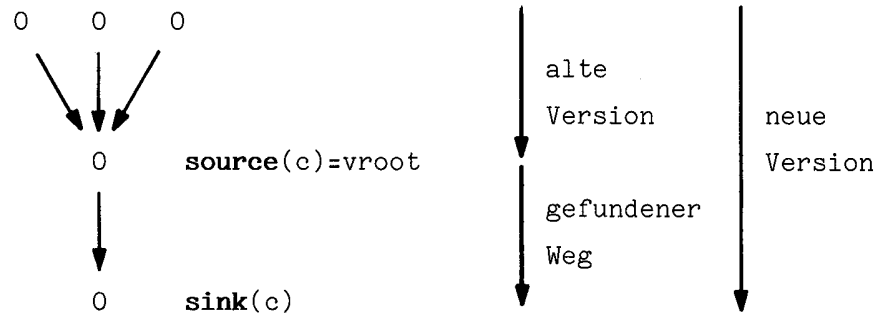


Abb. 5.5 Zur Erweiterung von Versionen in **dirpath**

Beispiel:

pgm_source zeige auf den oberen SH-Knoten der Abb. 5.2. Zu diesem Knoten gehört die Version

SH-P1(I(1).(47:32),X)

dirpath erzeugt daraus eine Version

ALU.a(SH-P1(I(1).(47:32),X)),

die dem in Abb. 5.2 mit ? bezeichneten Knoten zugeordnet wird.

Sofern weitere Wege nicht durch die gefundenen direkten Wege ausgeschlossen sind, versucht **path** anschließend, über Umwege zu Eingängen aus **matching-inputs(pgm** sink) zu gelangen. Die Umwege werden unterteilt in nicht-speichernde (sog. "vias") und speichernde (sog. "temporaries").

Als "vias" werden standardmäßig betrachtet: Multiplexer, Busse, BusTreiber und Bausteine, die eine Verknüpfung mit einem neutralen Element ausführen. Die Prozedur **viapath** sucht Umwege über solche Bausteine:

```
PROCEDURE viapath(pgm_source, pgm_sink : nodeclass);
 BEGIN
  FOR ALL vtree ∈ versions(pgm_source) DO
   vroot:=root(vtree);
   FOR ALL c ∈ connections DO
    IF part_id(source(c))=part_id(vroot)       und
       sink(c) ist zum Durchschleifen geeignet
     THEN füge sink(c) zwischen vroot und pgm_sink ein;
          rufe expression(tree(pgm_sink)) rekursiv mit dem
          Ziel, Wege von sink(c) zu Eingängen aus matching_
          inputs(pgm_sink) zu finden. Ergänze die Versionen
          von pgm_sink um die gefundenen Versionen.
    FI
   OD
  OD
 END;
```

Alg. 5.7 Suche nach nicht-speichernden Umwegen

Wird ein Umweg über einen nicht-speichernden Baustein gefunden, so wird dieser Baustein vorübergehend zwischen pgm_source und pgm sink eingefügt und **expression** rekursiv gerufen, um Wege vom Umwegbaustein zu den Eingängen aus **matching inputs(pgm_sink)** zu finden.

Im Falle der Konstanten 8 im obigen Beispiel erfolgt z.B. für die erste Version ein rekursiver Aufruf für den Ausdruck

```
? (BMUX (I(8).(31:16),...),"+")
```

Die drei Punkte stehen dabei hier und im folgenden für der Übersichtlichkeit halber ausgelassene Beschaltungen der Kontrolleingänge.

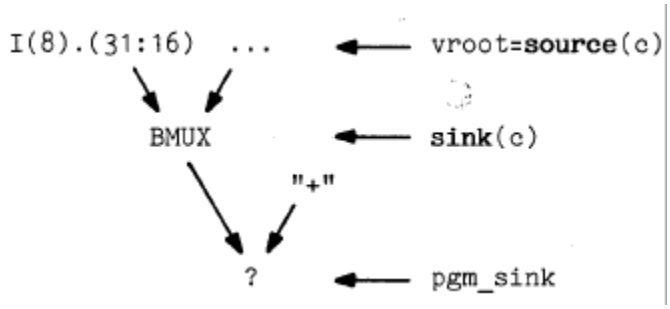


Abb. 5.6 Zur Baumtransformation in viapath

Während des rekursiven Aufrufs von expression erzeugt dirpath eine Version

```
ALU.b (BMUX (I(8).(31:16),...)
```

Die Prozedur **temppath** sucht nach speichernden Umwegen:

```
PROCEDURE temppath(t : treeclass; pgm_source,pgm_sink : nodeclass);
 BEGIN
  FOR ALL vtree ∈ versions(pgm_source) DO
   vroot:=root(vtree);
   FOR ALL c ∈ connections DO
    IF part_id(source(c))=part_id(vroot) und
      ∃ l ∈ templocs mit part_id(l)=part_id(sink(c))
     THEN erzeuge eine Zuweisung von vroot zu einer
          Hilfszelle im Speicher part_id(sink(c));
          ersetze pgm_source durch eine Leseoperation der Hilfszelle
          und rufe assignment für den entstandenen Baum rekursiv;
          erweitere die Versionen von root(t) um die
          gefundene sequentielle Version;
          restauriere die modifizierten Bäume.
    FI;
   OD;
  OD;
 END;
```

Alg. 5.8 Suche nach speichernden Umwegen

Wird ein Weg zu einem speichernden Baustein gefunden, so wird eine sogenannte sequentielle Version erzeugt. Diese besteht aus zwei nacheinander auszuführenden Anweisungen. Die erste stellt eine Zuweisung des durch den Knoten pgm source repräsentierten Ausdrucks an eine Hilfszelle dar. Die zweite Anweisung geht aus der ursprünglichen Zuweisung durch Ersetzen von tree(pgm source) durch eine Leseoperation der Hilfszelle hervor. Für diese zweite Anweisung wird **assignment** wiederum rekursiv gerufen. Während dieses Aufrufs können für die zweite Anweisung mehrere Versionen erzeugt werden.
Falls die arithmetisch/logische Einheit in obigem Beispiel die Daten des rechten Eingangs unverändert zum Ausgang weiterleiten kann, so wird z.B. die Sequenz

```
ACCU := ALU (X, BMUX (I(8).(31:16),...),...);
SH(0):= SH(1) "+" ACCU
```

erzeugt (X = unbenutzter Eingang). Für die zweite Zuweisung erfolgt ein erneuter Aufruf von **assignment**.

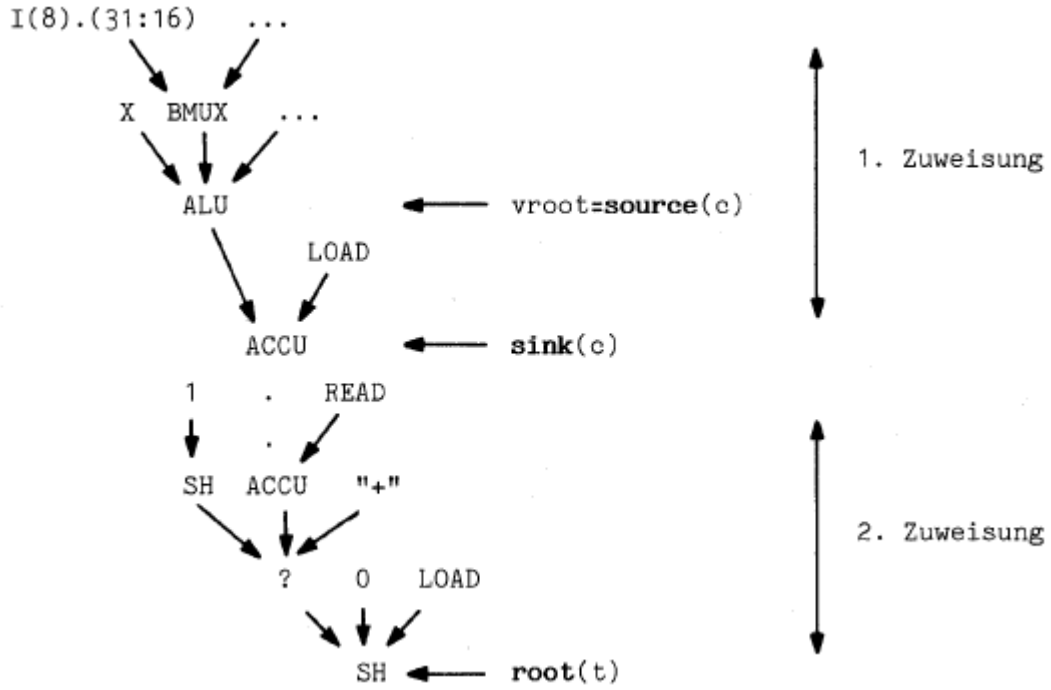


Abb.5.7 Zur Baumtransformation in **temppath**

Die Sequenz erscheint zunächst wenig sinnvoll. Sie wäre aber bei Abwesenheit anderer Versionen für die Konstante 8 erforderlich, da das Befehlsfeld 1.(31:16) gleichzeitig der Adressierung des Hauptspeichers dient. Dieses Beispiel zeigt die starke Kontextabhängigkeit der durchzuführenden Baumtransformationen. Obwohl die ALU ALU die einzige Einheit ist, die die benötigte Operation "+" ausführen kann, muß sie eventuell zum Durchschalten der Daten zu einer Hilfszelle verwendet werden.

5.2.5 Bundling

Die als Ergebnis der Suche nach Verbindungen von dirpath gefundenen Versionen werden dem Knoten pgm sink zugeordnet. Die so gefundenen Versionen heißen **partielle** Versionen, da sie den Datenfluß lediglich zu einem der Eingänge aus **matching inputs(pgm** sink) repräsentieren.

Sind für einen Knoten q die Versionen aller Knoten aus **sons(q)** bekannt, so müssen daraus Versionen für den Datenfluß zum **Ausgang** der Bausteine bestimmt werden. Dieser Prozeß wird im MSS als **Bundling** bezeichnet.

Beispiel:

Aus den partiellen Versionen

```
ALU.a (SH-P1 (I(1).(47:32)) ),
ALU.b (BMUX (F#00 ! (I(8).(15:8)),...) ) und
ALU.s ( I(2).(1:0) )                          (*siehe 5.2.2*)
```

resultiert die Version

```
ALU( SH-P1(I(1).(47:32)),
     BMUX(F#00!I(8).(15:8),...),
     I(2).(1:0)
   )
```

Beim Bundling werden alle möglichen Kombinationen von partiellen Versionen zu nicht-partiellen Versioneh zusammengefügt. Die erzeugten Versionen werden wieder dem gleichen Knoten zugeordnet und die partiellen Versionen werden gelöscht:

```
PROCEDURE bundling(n : nodeclass);
 BEGIN
  erzeuge alle gültigen Kombinationen von partiellen Versionen
  und ordne sie n zu;
  lösche alle partiellen Versionen von n.
 END;
```

Alg. 5.9 Erzeugung nicht-partieller Versionen

Wie die Prozedur **expression** zeigt, erfolgt das Bundling jeweils bevor Pfade mit den Wurzeln vroot der neu erzeugten Versionen als Quelle gesucht werden.

Eine nicht-partielle Version für einen Knoten pgm_sink kann nur dann erzeugt werden, wenn für alle Knoten pgm_source $\in$ **sons** (pgm_sink) partielle Versionen mit gleichem Inhalt des Wurzelknotens existieren. Z.B. dürfen zwei partielle Versionen, deren Wurzeln unterschiedliche arithmetisch/ logische Bausteine bezeichnen, niemals zu einer nicht-partiellen Version zusammengefaßt werden.

Diese Tatsache wird zu einem sogenannten "Prebundling" ausgenutzt. Für einen Knoten pgm_source $\in$ **sons**(pgm_sink) wird in den Algorithmen 5.4, 5.6, 5.7 und 5.8 eine partielle Version nur dann erzeugt, wenn für die übrigen, bereits betrachteten pgm_source' $\in$ **sons**(pgm_sink) eine partielle Version mit dem gleichen Baustein als Wurzelknoten existiert.

Da die Datenflußbäume von rechts nach links traversiert werden, wird unter den Söhnen eines Knotens pgm-sink zuerst der Knoten betrachtet, der die Operation beschreibt. Für diesen werden die partiellen Versionen zuerst erzeugt. Dadurch werden partielle Versionen für Adreß- und Daten-

eingänge von vornherein verworfen, wenn der betreffende Baustein oder das betreffende Port die Operation nicht ausführen kann.

Im Falle von Konkatenationen werden zunächst die Versionen für die zu konkatenierenden Teilausdrücke bestimmt. Die Prozedur **catbundiing** erzeugt daraus die Versionen für die Konkatenation als Ganzes. **catbundling** arbeitet ähnlich wie **bundling**, jedoch gibt es im Detail Unterschiede bei der Behandlung der Bitnummern.

5.2.6 Ressource-Konflikte

Während des Bundlings dürfen partielle Versionen nur dann kombiniert werden, wenn sie gegeneinander keine Konflikte um Hardware-Ressourcen erzeugen.

Beispiel:

Für die Wurzel des Ausdrucks nach Abb. 5.2 ergeben sich u.a. die partiellen Versionen

```
SH-P2.addr( I(0).(31:16) )              und
SH-P2.data(ALU(SH-P1(..),BMUX(I(8).(31:16),...),...)
```

Diese partiellen Versionen sind bezüglich Bit 19 der Instruktion nicht konfliktfrei, da die Binärdarstellung der 8 in Bit 3 - (19-16) im Gegensatz zur 0 eine Eins enthält. Ihre Kombination zu nicht partiellen Versionen ist daher nicht möglich. Es stellt sich die generelle Frage, welche Konflikte während des Bundlings zu beachten sind.

Wesentlich für die Realisierung der Codeerzeugung ist die Erkenntnis:

> Die einzigen hardwaremäßig relevanten Ressourcekonflikte sind Versuche, Leitungen in einer Rechnerstruktur gleichzeitig mit unterschiedlichen Werten zu belegen. Konflikte bezüglich der Benutzung von Bausteinen können stets als Konflikt auf einer der Eingangsleitungen betrachtet werden.

In Hardware realisiert und mit MIMOLA beschrieben werden können Bausteine, die mehr als eine Operation zur Zeit ausführen. Beispielsweise stellen manche arithmetischen Bausteine während der Operation "-" an einem getrennten Ausgang das Ergebnis der Operation "=" zur Verfügung. Dies zeigt, daß beliebig viele Operationen eines Bausteins gleichzeitig ohne Hardwarekonflikt genutzt werden können, solange keine Leitungskonflikte an den Eingängen auftreten. Dies ist mit dem Maschinenmodell nach Mallett z.B. nicht möglich, da dort ein Baustein z.Zt. nur von jeweils einer Operation benutzt werden kann. Durch die gedankliche Auftrennung eines physikalischen Bausteins in mehrere logische Bausteine würde man diese Schwierigkeiten mit dem Malletschen Modell überwinden. Dieser Weg soll im MSS aber nicht gegangen werden, um eine möglichst enge Beziehung zwischen den physikalischen Bausteinen und ihrer Beschreibung in MIMOLA zu erhalten.

Aus dieser Erkenntnis ergibt sich die Definition:

Def. .
Zwei Versionen v1 und v2 erzeugeneinen Hardwarekonflikt (in Zeichen: v1 **confl** v2) genau dann, wenn sie potentiell auf mindestens einer der Leitungen der Rechnerstruktur unterschiedliche Werte benötigen.

Im letzten Beispiel liegt ein Konflikt bezüglich der Benutzung der Instruktionsbits vor. Gemäß der folgenden Definition heißen solche Konflikte Befehlskonflikte:

Def. .
Zwei Versionen v1 und v2 erzeugen einen Befehlskonflikt (in Zeichen: v1 **instrconfl** v2) genau dann, wenn für mindestens ein Befehlsbit in v1 ein anderer Wert als in v2 erforderlich ist.

In der Hardware vermieden werden müssen Konflikte auf den Bussen. Diese sind erklärt durch die folgende Definition:

Def. .
Zwei Versionen v1 und v2 erzeugen einen Buskonflikt (in Zeichen:

v1 **busconfl** v2) genau dann, wenn sie potentiell für mindestens ein Bit eines Busses einen unterschiedlichen Wert benötigen.

Für das im MSS benutzte Hardware-Modell gilt nun der folgende Satz:

Satz: Zwei Versionen erzeugen Hardwarekonflikte genau dann, wenn sie Befehlskonflikte oder Buskonflikte erzeugen, d.h. (v1 **confl** v2) g.d.w. (v1 **instrconfl** v2) V (v1 **busconfl** v2

Beweis: Zunächst ist klar, daß Befehlskonflikte und Buskonflikte Hardwarekonflikte implizieren, da Steuerleitungen und Busse spezielle Leitungen sind.

Seien andererseits die Versionen v1 und v2 frei von Befehls - und Buskonflikten. Es ist zu zeigen, daß folglich keine Konflikte bezüglich der Verwendung von Leitungen existieren. Zunächst wird gezeigt, daß keine Konflikte bezüglich der Eingänge existieren.

Angenommen, es gäbe einen Konflikt an einem der Eingänge von Moduln. Sei d $\in$ **parts** eines dieser Moduln. Seien n1 $\in$ **nodes**(v1) und n2 $\in$ **nodes**(v2) Vorkommen von d in v1 bzw. v2. Damit ein Konflikt erzeugt werden kann, muß es unter den Argumenten **sons**(n1) und **sons**(n2) gleiche Eingänge geben. Seien e1 und e2 die Knoten, die diese Eingänge bezeichnen. Da Buskonflikte nach Voraussetzung ausgeschlossen sind, kann d kein Bus sein und jeder Eingang von d ist nur an einem Ausgang angeschlossen und e1 und e2 bezeichnen das gleiche Modul. Da die Werte der Ausgänge verschieden sein sollen, muß es ein Modul geben, welches ohne einen Instruktionskonflikt zu erzeugen an demselben Ausgang unterschiedliche Werte generiert. Das ist aber nicht möglich, da der Kontrollcode für einen bestimmten Ausgang eindeutig die ausgeführte Operation spezifiziert und verschiedene Kontrollcodes ohne Instruktionskonflikte ausgeschlossen sind. Damit sind nur nur gleiche Operationen möglich, die Werte am Ausgang sind notwendigerweise gleich und ein Leitungskonflikt existiert nicht.

Konflikte kann es damit nur noch an Ausgängen geben, die nicht an Eingänge angeschlossen sind. Da dazu aber mindestens zwei verbundene Ausgänge erforderlich sind und Ausgänge in MIMOLA nicht miteinander verbunden werden dürfen (es sei denn über Busse), ist dies nicht möglich. q.e.d.

Damit brauchen Konflikte bezüglich anderer Bausteine als Bussen nicht betrachtet zu werden. Dieser Vorteil gegenüber den meisten Modellen ergibt sich, da durch den expliziten Einschluß von Multiplexern in die Beschreibung an jedem Eingang nur ein Ausgang angeschlossen ist. Beim Modell von Mallett z.B. müssen dagegen auch Konflikte bezüglich arithmetisch/logischer Einheiten beachtet werden.

Wenn bei jeder Erzeugung eines Versionsknotens mit mehr als einem Sohn **bundling** gerufen wird, muß nur in dieser Prozedur geprüft werden, ob Ressourcekonflikte vorliegen.

5.2.7 Ablauf für das Beispielprogramm

Die folgenden Diagramme skizzieren den Ablauf der Versionserzeugung für das Beispielprogramm nach Abb. 5.2. Versionen sind jeweils durch - -gekennzeichnet.

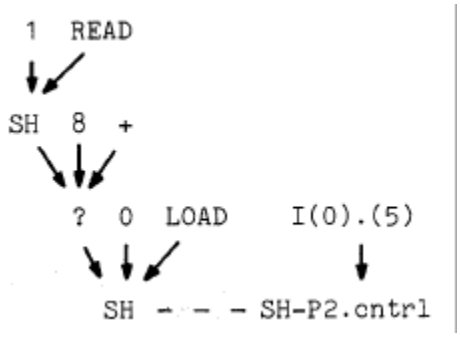


Abb.5.8 Aufruf von **operation** für LOAD

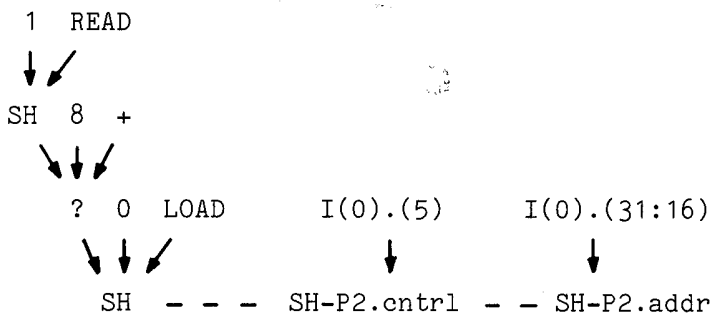


Abb.5.9 Aufruf von constant und path für 0

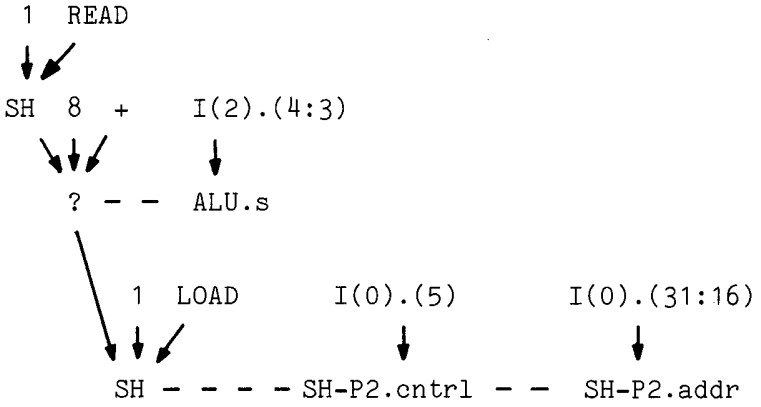


Abb.5.10 Aufruf von operation für +

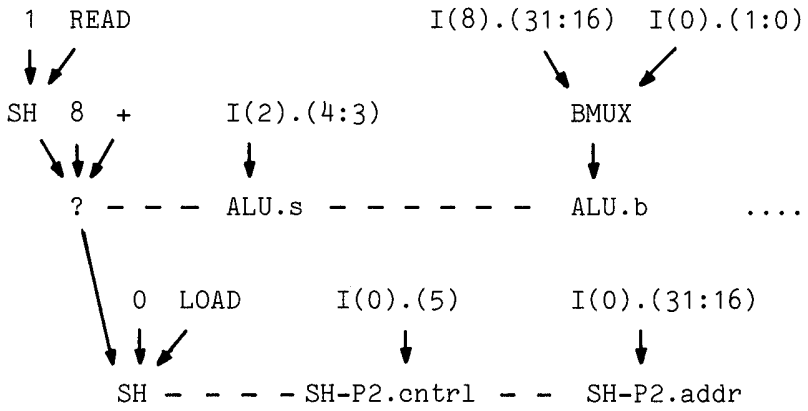


Abb.5.11 Aufruf von constant, viapath, dirpath für 8

(F#00-und DEC-Versionen ausgelassen (....))

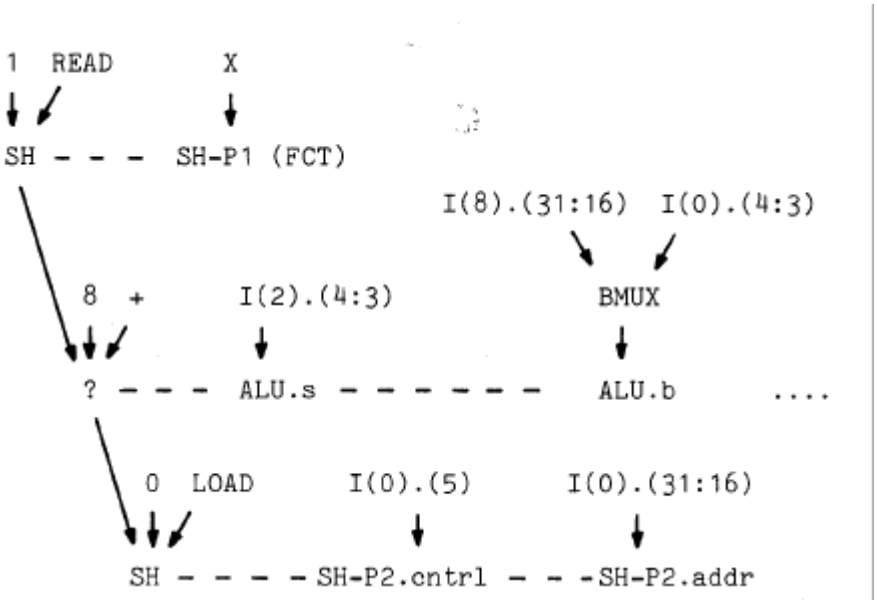


Abb.5.12 Aufruf von operation für READ

(F#00-und DEC-Versionen ausgelassen (....))

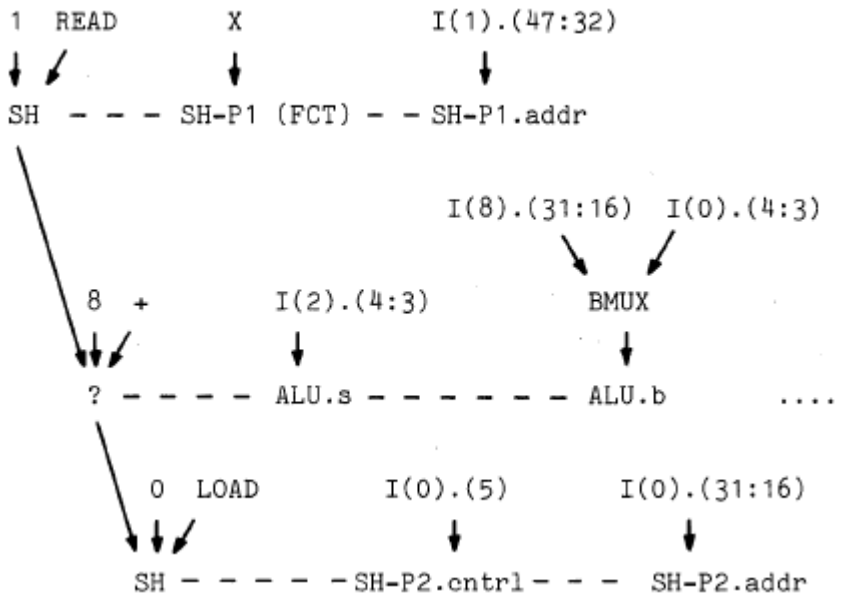


Abb.5.13 Aufruf von constant für 1

(F#00-und DEC-Versionen ausgelassen (....))

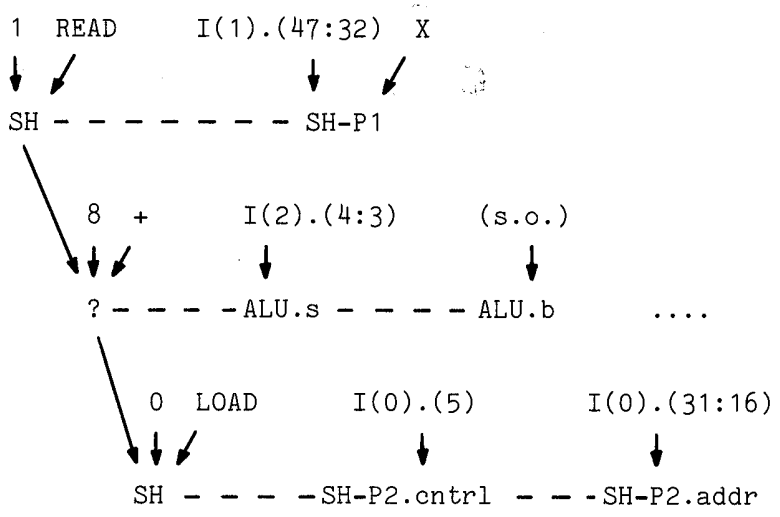


Abb.5.14 Aufruf von bundling für SH

(F#00-und DEC-Versionen ausgelassen (....))

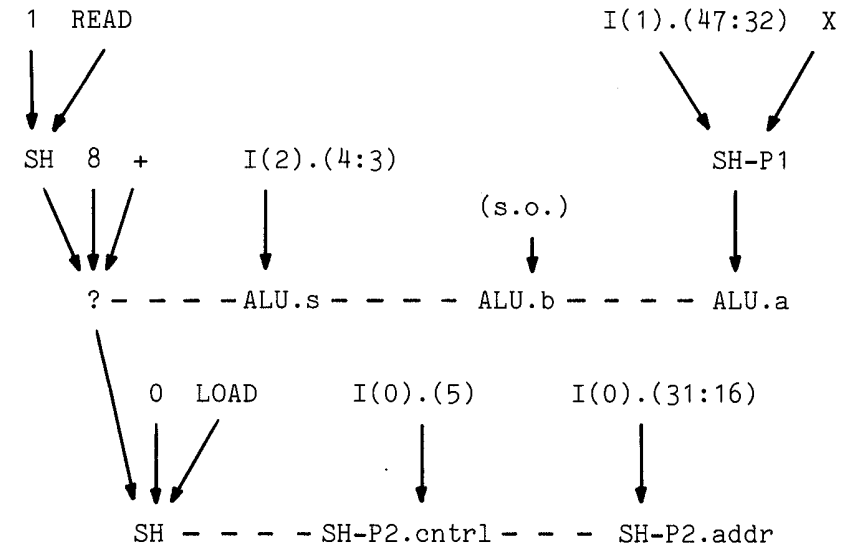


Abb.5.15 Aufruf von path für SH

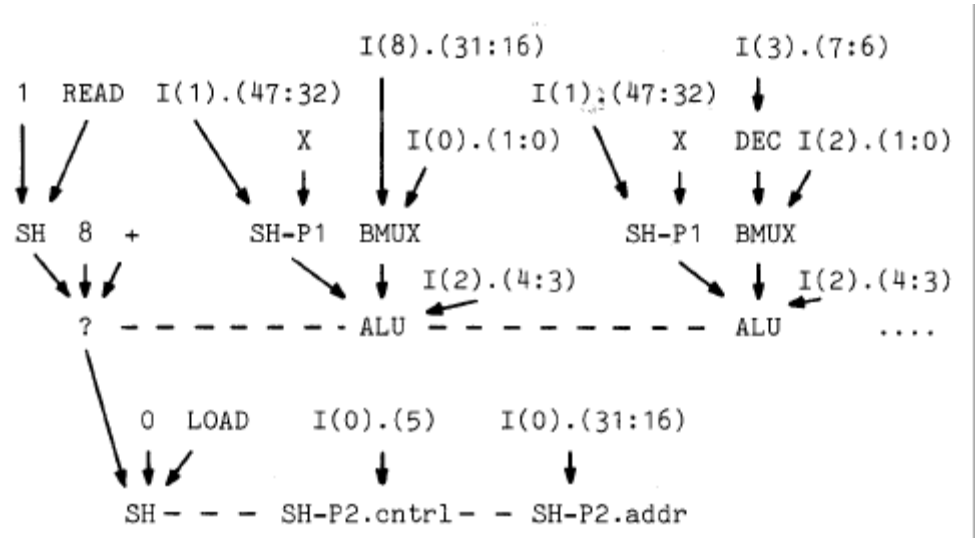


Abb.5.16 Bundling am Knoten ? (F#00-Version ausgelassen (....))

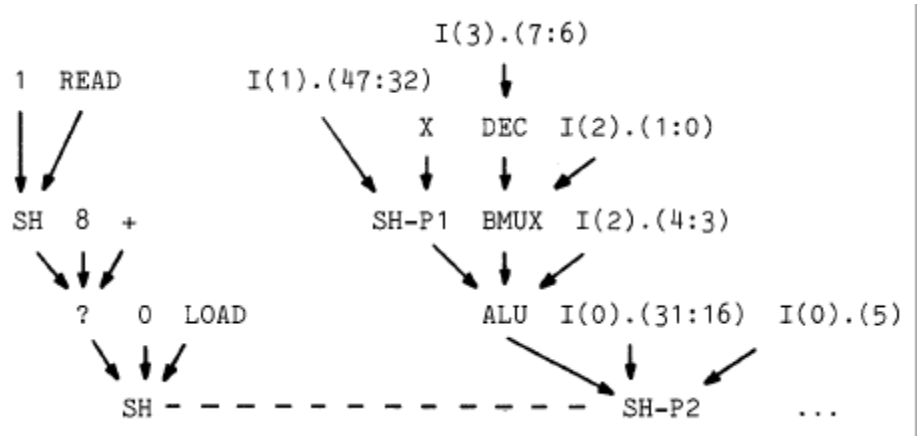


Abb.5.17 Bundling am Knoten SH (F#00-Version ausgelassen)

Der Befehlscode kann den gezeigten Bäumen leicht entnommen werden. So ergibt sich z.B. für die Version der Abb. 5.17:

Befehlsbits	47:32	31:16	15:8	7:6	.5	4:3	2	1:0
Werte	1	0	X	3	0	2	X	2

Abb.5.18 Befehlscode

5.2.8 Programmtransformationen

Codegeneratoren für feste Zielmaschinen führen häufig implizit bestimmte Programmtransformationen durch, um Code für die Maschine erzeugen zu können. Bei einem maschinenunabhängigen Codegenerator kann man solche Transformationen nun nicht fest einprogrammieren. Vielmehr muß man diese Transformationen zusammen mit der Beschreibung der Zielmaschine eingeben können. Dies ist insbesondere deshalb erforderlich, weil bei einem Wechsel auf eine andere Zielmaschine **keinerlei Änderung** am Codegenerator erfolgen soll.

In den folgenden Beispielen werden der Übersichtlichkeit halber die in der Praxis benötigten Bitbereichs-Angaben ausgelassen:

```
REPLACE   0   WITH   (0 "AND" RQ)    END
```

Diese Regel wird z.B. bei AMD-Bitslice Prozessoren benötigt, um eine am Eingang der ALU hartverdrahtete 0 an deren Ausgang weiterleiten zu können (Vegdahl bezeichnet diese Transformationen als "constantunfolding").

```
REPLACE (&a "+" &b) "+" &c WITH &a "+" (&b "+" &c) END
```

Dies ist die MIMOLA-Schreibweise für eine Implikationsrichtung des Assoziativgesetzes. In bestimmten Fällen wird es zur Codeerzeugung benötigt.

```
REPLACE (&a "=" &b) WITH "=0" (&a "-" &b) END
```

Die Vergleichsoperation wird auf eine Subtraktion und einen Vergleich auf Null zurückgeführt.

```
REPLACE SH( &a<constant>.(15:0) ) WITH
        SH( (&a.(15:8)!0.(7:0)) "+" (0.(7:0)!&a.(7:0)) ) END
```

Diese Regel kann bei einer seitenähnlichen Adressierung des Speichers erforderlich werden.

Die Beispiele geben einen kleinen Eindruck von der Vielfalt der Anwendungsmöglichkeiten für Transformationsregeln.

Insbesondere Regeln wie die beiden ersten können nicht bedingungslos angewandt werden. Sie sollen nur dann benutzt werden, wenn dies zu gutem Code führt. Daher werden diese Transformationen nicht alle bereits in MSSR (vgl. Abschnitt 3.2) durchgeführt. Vielmehr kann der Benutzer angeben, welche Regeln bedingt angewandt werden sollen. MSSV vergleicht während der Codeerzeugung, ob eine dieser Regeln paßt. Ist dies der Fall, so wird ein der rechten Seite der Regel entsprechender Baum erzeugt. Für diesen Baum wird wiederum **expression** gerufen und die gefundenen Versionen werden den Versionen des aktuellen Knotens zugefügt. Da während der Bearbeitung des rekursiven Aufrufs von **expression** weitere Regeln passen können, kann man die Behandlung der Ersetzungsregeln als kleines **Expertensystem** bezeichnen. Aus Laufzeitgründen ist allerdings die Schachtelungstiefe der Aufrufe von **expression** und damit die Schachtelungstiefe der Anwendung von Ersetzungsregeln beschränkt. Die Schranke kann von außen durch eine Compilersteuerung gesetzt werden. Üblicherweise werden 4 - 5 rekursive Aufrufe von **expression** erlaubt. Im Falle einer konkreten Hardwarestruktur mit sehr langen kombinatorischen Pfaden wurden allerdings bis zu 13 rekursive Aufrufe notwendig.

Eine große Zahl von Ersetzungsregeln reduziert deutlich die Übersetzungsgeschwindigkeit. Aus diesem Grunde wurde von dem ursprünglichen Plan, auch Verbindungen und vorhandene Bausteine über solche Regeln zu modellieren, Abstand genommen. Auch wurde die Ausnutzung der Kommutativität und der neutralen Elemente aus diesem Grunde fest einprogrammiert. Andererseits zeigt die Erfahrung, daß nicht alle Regeln einprogrammiert werden können, da immer wieder neue, unvorhergesehene Regeln notwendig werden.

Es wird als Vorteil des Verfahrens betrachtet, daß meist implizit vorhandenes Wissen in Form dieser Ersetzungsregeln explizit repräsentiert wird. Die Notwendigkeit der Benutzung eines kleinen Expertensystems bildet **den Grund für die durchgängige Verwendung von Programmoder Baum-Transformationen als Mittel der Codeerzeugung im MSS.**

5 2.9 Bedingte Anweisungen

Im Gegensatz zur üblichen Praxis werden die Programme im MSS nicht bereits vor der eigentlichen Codeerzeugung in sog. Basisblöcke (d.h. Blöcke, die höchstens an ihrem Ende einen Sprung enthalten) zerlegt. Damit soll erreicht werden, daß MSSV sowohl Versionen für bedingtes Laden und bedingte Ausdrücke als auch Versionen mit bedingten Sprüngen erzeugen kann. Die Auswahl aufgrund des Zusammenhangs obliegt MSSC.

Die Behandlung bedingter Anweisungen übernimmt die Prozedur **ifstatement:**

```
PROCEDURE ifstatement(i : treeclass);
 BEGIN
  cond_branch:
   Versuche, den Baum i durch Anwenden von Programmtransforma-
   tionen in eine Sequenz s eines bedingten Sprunges und einer
   Menge unbedingter Zuweisungen zu zerlegen.
   Rufe seqblock(s), falls erfolgreich
  cond_load:
   IF die bedingte Anweis. ist innerste bedingte Anweis. THEN
    1 : Versuche, je eine Zuweisung in THEN- und ELSE- Teil zu
        einem bedingten Ausdruck zusammenzufassen. (optional)
        Rufe assignment für die erzeugte Zuweisung
    2 : Erzeuge bedingte Zuweisungen für unter 1: nicht zusammen-
        gefaßte Zuweisungen falls die linke Seite der Zuweisung
        einen bedingt ladbaren Speicher bezeichnet.
        Rufe assignment für die erzeugte Zuweisung
    3 : Falls das ifstatement Zuweisungen enthält, die weder
        unter 1: noch unter 2: behandelt wurden, verfahre
        für diese Zuweisungen wie unter cond_branch.
   FI;
 Definiere versions(root(i)) als Vereinigung der in den Abschnitten
 "cond_load" und "cond_branch" generierten Versionen.
END
```

Alg. 5.10 Behandlung bedingter Anweisungen

Zur Erzeugung bedingter Sprünge obliegt es dem Benutzer, für seine Hardware passende Programmtransformationen aus einer Bibliothek auszuwählen oder ggf. neue zu schreiben. Tut er dies nicht, so werden keine bedingten Sprünge erzeugt. Die Bibliothek enthält standardmäßig Transformationsregeln für Hardware, die 1-oder 2-Wegesprünge durchführen kann.

Beispiel: Regel für 2-Wegesprünge (der Übersichtlichkeit halber werden Bitbereichsangaben wieder ausgelassen)

```
REPLACE
 if &cond then &then else &else fi
WITH
 begin
  LIF&:   RP:=(if &cond then LTHEN& else LELSE& fi);
  LTHEN&: &then;
  LELSE&: &else
 end
END
```

LIF&, LTHEN& und LELSE& sind spezielle Label, die in der Kompaktierung umgesetzt werden. Außerdem wird &then in der Kompaktierung automatisch um einen Sprung über LELSE& hinweg erweitert, falls &then nicht bereits einen Sprung enthält.

Auch mit "AND" und "OR" verknüpfte Bedingungen können mit Hilfe von Regeln sukzessive in eine Sequenz aufgelöst werden.

Bedingte Zuweisungen werden nur erzeugt, falls in einem Vorlauf erkannt wurde, daß der betreffende Speicher bedingt ladbar ist. Weiterhin werden bedingte Zuweisungen nur für die innerste geschachtelte Zuweisung erzeugt. Weiter außen liegende Bedingungen werden stets in bedingte Sprünge umgesetzt.

Die in **ifstatement** vorkommende Prozedur **seqblock** ruft direkt **parblock:**

```
PROCEDURE seqblock(sb :nodeclass);
 BEGIN
  FOR ALL pb ∈ sons(sb) DO
   if nodetype(pb)=partyp then parblock(pb)
  OD
 END
```

Alg. 5.11 Behandlung sequentieller Blöcke

5.2.10 Beschleunigung des Verfahrens

Das in den Abschnitten 5.2.2 bis 5.2.4 dargestellte Verfahren läßt sich so, wie es beschrieben wurde, nur auf Rechnerstrukturen anwenden, die eine kleine Zahl von Bausteinen und Verbindungen enthalten. Bestehen Rechner aus vielen Komponenten,so werden die Laufzeiten unakzeptabel. Um das Verfahren zu beschleunigen, findet vor der Übersetzung des Programms ein Vorlauf statt (dieser Vorlauf entspricht in etwa einer "compiler-compile"-Phase).

Wesentliche Aufgabe des Vorlaufes ist es, den Einbau eines zusätzlichen Tests in die obigen Algorithmen zu ermöglichen, mit dessen Hilfe möglichst viele der Sackgassen der Versionssuche vermieden werden. Gesucht ist also eine notwendige Bedingung dafür, daß eine partielle Version zu einer Version für die gesamte Zuweisung führt. Eine solche Bedingung ermöglicht die im folgenden definierte Relation **tempcontext**, die eine globale Information über die Verbindungen innerhalb der Rechnerstruktur darstellt.

Jedem Subport (vgl. Abschn. 2.1.1) ist aufgrund der Hardware-Beschreibung eine endliche, nicht-leere Menge von maximalen Bitbereichen (vgl. Abschn. 2.1.2, Def. 12 und 13) zugeordnet.

Beispiel:

Aufgrund einer Deklaration I : (ab((7:6), (4:3))) gehören zu dem "Subport" I.ab die maximalen Bitbereiche 1.(7:6) und 1.(4:3).

Gehören zu einem Subport mehrere maximale Bitbereiche, so soll jeder einzelne Bereich in diesem Abschnitt ebenfalls als ein Subport betrachtet werden. Das heißt, 1.(7:6) und 1.(4:3) gelten als Subports (obwohl sie nicht mit einem eigenen Namen bezeichnet werden). Damit ist umgekehrt auch jedem maximalen Bereich ein Subport zugeordnet.

Zunächst sei eine globale Numerierung der Subports vorausgesetzt, d.h. es existiert eine eineindeutige Abbildung der Subports auf die natürlichen Zahlen.

Def.:

1. SN sei die Menge der sich bei einer globalen Numerierung der Subports ergebenden Zahlen. Diese Zahlen heißen
   S u b p o r t n u m m e r n.

2. SN_a sei die Menge der Subportnummern, welche Ausgänge bezeichnen.

3. SN_e sei die Menge der Subportnummern, welche Eingänge bezeichnen.

4. Für $s \in SN$ sind **part_id**(s) und **subport_id**(s) die zu s gehörigen Parts und Subportidentifier.

5. Für $s \in SN$ ist **ranges**(s) die zu s gehörige Menge von maximalen Bitbereichen.

6. Für alle maximalen Bereiche b sei **subp**(b) die zu b gehörende Subportnummer $s \in SN$.

7. Seien b,b' (nicht notwendigerweise maximale) Bitbereiche oder nichtleere Mengen von solchen. Dann bedeutet die Schreibweise $b \cap b' \neq \emptyset$, daß b und b' mindestens ein gemeinsames Bit enthalten.

Beispiel:

Unter den Voraussetzungen des letzten Beispiels gilt
$I.(7:6) \cap I.(6) \neq \emptyset$

Def.:

8. Die Relation **dircontext** wird definiert durch:
   s1 **dircontext** s2 mit s1,s2 $\in$ SN g.d.w. eine der folgenden Bedingungen gilt:
   a. Für alle zu s1 gehörenden Bereiche existiert mindestens eine Verbindung nach s2:
      $\forall$ q $\in$ **ranges**(s1) : $\exists$ c $\in$ **connections**
      mit **source**(c) $\cap$ q $\neq \emptyset$ und **sink** (c) $\cap$ **ranges**(s2) $\neq \emptyset$
      (zur Definition von **connections**, **source** und **sink** siehe Abschn. 2.1.2, Def. 19 und 20)
   b. s1 ist ein Eingang, der, ohne Bedingungen an andere als den Kontrolleingang zu stellen, identisch auf den Ausgang s2 desselben Bausteins abgebildet werden kann:
      $\exists$ r $\in$ **oplist**(**part_id**(s1)) mit **op_symbol**(r)=DAT
      (DAT = Identische Abbildung eines Eingangs auf den Ausgang)
      und **inputs**(r) enthält s1 als einzigen Eingang
      und **output**(r) bezeichnet s2.
   c. die am Eingang s1 anliegenden Daten können mit Hilfe eines neutralen Elementes an einem weiteren Eingang unverändert zum Ausgang s2 weitergeleitet werden.
      $\exists$ r $\in$ **oplist**(**part_id**(s1)), wobei r ein neutrales Element besitzt und der für das neutrale Element zu benutzende Eingang nicht s1 ist.
   d. s1 ist Dateneingang und s2 ist Datenausgang eines Hilfszellen enthaltenden Speichers:
      s1 $\in SN_e$, s1 ist **Datene**ingang, s2 $\in SN_a$,
      $\exists$ l $\in$ **templocs**: **part_id**(l)=**part_id**(s1)=**part_id**(s2)

Die Relation dircontext modelliert direkte Verbindungen der Rechnerstruktur. dirset(s) ist definiert als Menge aller vom Subport s aus direkt erreichbaren Subports:

Def.:

9. $\forall$ s $\in$ SN sei:
   dirset(s) := { s' | s' $\in$ SN, s **dircontext** s' }

Eine notwendige Bedingung für einen indirekten Informationstransport von einem nur einen Bereich enthaltenden s1 nach s2 ist die Existenz einer Kette t_0, t_1, .., t_n von Subports mit t_0=s1, t_n=s2 und $\forall\, j \in [1:n]$ gilt t_{j-1} **dircontext** t_j. Eine notwendige Bedingung ist also, daß (s1,s2) in der transitiven Hülle **dircontext+** von **dircontext** liegt.

Enthält s1 mehrere Bitbereiche, so bestimmt der Schnitt der transitiven Hüllen der einzelnen Bereiche die Erreichbarkeit, wie Abb. 5.19 verdeutlicht: von I.ab aus läßt sich s2 indirekt erreichen, da s2 im Schnitt der von 1.(7:6) und 1.(4:3) aus indirekt erreichbaren Suports liegt. s2 liegt also im Schnitt der transitiven Hüllen für die zu I.ab gehörenden Bereiche. Die transitive Hülle des Schnittes der direkt erreichbaren Subports ist dagegen leer, da bereits der Schnitt der von 1.(7:6) und 1.(4:3) aus direkt erreichbaren Subports leer ist.

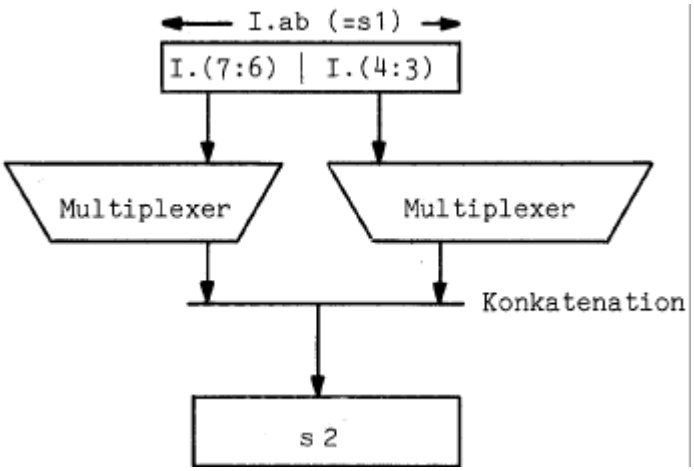


Abb. 5.19 Zur Erreichbarkeit bei Konkatenationen

Die indirekte Erreichbarkeit wird dementsprechend durch die Relation **tempcontext** beschrieben:

Def.:

10. Für alle s1, s2 $\in$ SN ist die Relation **tempcontext** definiert durch:
    s1 **tempcontext** s2 g.d.w.
    $\forall$ q $\in$ranges(s1) : **subp**(q) **dircontext+** s2

s1 **tempcontext** s2 ist eine notwendige Bedingung dafür, daß ein Daten
transport von s1 nach s2 möglich ist, wenn nur die Operationen gemäß Definition 6 b.-d. benutzt werden, um Eingänge identisch auf Ausgänge abzubilden. Da Hardware-Konflikte dabei aber nicht berücksichtigt werden, ist diese Bedingung nichthinreichend und in jedem einzelnen Fall muß mit Hilfe von **expression** nachgeprüft werden, ob

Die Algorithmen 5.3, 5.4, 5.7 und 5.8 enthalten nun folgende wichtige Erweiterung (vgl. Abb. 5.20):

f sei die Operation, die von den auf pgm sink passenden Bausteinen auszuführen ist. Dann sei **matching inputs** (pgm-sink) wie in Abschn. 5.2.4 die Menge jener Bausteineingänge, an die pgm source zwecks Durchführung der Operation f angelegt werden kann.

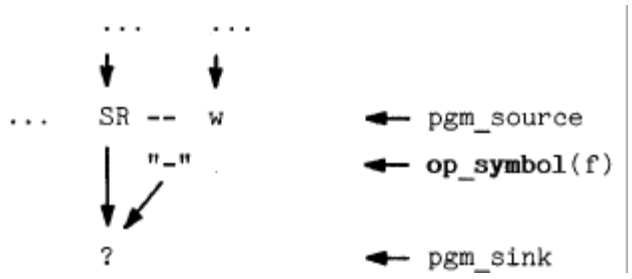


Abb. 5.20 Erläuterung von pgm sink, pgm source, w und f

Dann wird für pgm-source eine Version mit Wurzel w nur dann erzeugt, wenn von w aus gemäß der Relation **tempcontext** mindestens ein Element aus **matching inputs(pgm** sink) erreichbar ist.

Für dieses Verfahren gibt es allerdings eine Einschränkung: Sofern ein Baustein nur aufgrund einer REPLACE-Transformationsregel geeignet ist, Daten an einem Eingang unverändert an den Ausgang weiterzuleiten, so wird dieser Baustein bei der Berechnung von **tempcontext** nicht als Umweg in Betracht gezogen.

Die Definition der Context-Relationen zwischen Subports statt zwischen

Bits stellt einen Kompromiß zwischen Speicherbedarf und Laufzeit dar. Context-Relationen zwischen Bits würden zwar Übersetzungszeit sparen, dafür aber unvertretbar viel Speicherplatz erfordern. (In der Praxis muß mit bis zu ca. 800 Subports gerechnet werden.)

Die zur Berechnung der Context-Relationen ausgeführten Schritte sind in der Beschreibung der Prozedur **prepareresources** enthalten:

```
PROCEDURE prepareresources;
 BEGIN
  IF der Zielrechner ist gegenüber der letzten Übersetzung verändert
   THEN
    Markiere alle Eingänge, die mittels einer identischen Abbildung
      ("DAT") auf den Ausgang abgebildet werden können sowie alle
      Eingänge, die als Argument einer Operation mit einem neutralen
      Element benutzt werden können.
    Berechne dircontext und tempcontext.
    Rufe expression für alle benötigten neutralen Elemente und hinter-
      lege die gefundenen Versionen.
    Lösche die Markierung an Eingängen, für die keine neutralen
      Elemente gefunden wurden.
    Berechne dircontext und tempcontext.
    Für alle deklarierten Operationen rufe expression, um die Versionen
      für den Kontrollcode zu erzeugen.
    Berechne die Versionen für INHIBIT-Operationen.
    Warne den Benutzer,falls Kontrollcodes nicht erzeugt werden können.
    Rufe statement, um zu testen, welche Art von bedingten Sprüngen
      möglich ist.
    Rufe statement, um zu testen, welche Speicher bedingt geladen
      werden können.
    Rufe statement, um die Versionen zum Fortschalten des Programm-
      zählers zu bestimmen und teile diese MSSC mit.
    Rette die berechnete Information in Dateien.
   ELSE lese die sichergestellten Dateien
 END;
```

Alg. 5.11 Vorlauf der Versionserzeugung

Zunächst werden die Relationen berechnet unter der Annahme, daß an allen Eingängen die benötigten neutralen Elemente erzeugt werden können. Die so berechneten Relationen werden ausgenutzt während des Versuchs, Versionen für die neutralen Elemente mittels Aufrufen von **expression** zu erzeugen. Da nicht an allen Eingängen neutrale Elemente erzeugt werden können, werden die Relationen anschließend korrigiert.

Um weitere Übersetzungszeit zu sparen, werden auch die Versionen zur Beschaltung der Kontrolleingänge bereits im Vorlauf ermittelt. Dadurch entfallen die rekursiven Aufrufe von **expression** in den Prozeduren **operation** und **constant** während der Programmübersetzungszeit.

Versionen zur Erzeugung von INHIBIT-Codes (vgl. Abschn. 2.1.1) werden ebenfalls im Vorlauf berechnet und der Kompaktierung mitgeteilt. Das gleiche gilt für Versionen zum Fortschalten des Programmzählers und für unbedingte Sprünge. Ebenfalls wird berechnet, welche Art von bedingten Sprüngen möglich ist und welche Speicher bedingt ladbar sind.

Die Ergebnisse dieses Vorlaufs werden in Dateien sichergestellt. Erfolgt eine weitere Übersetzung für die gleiche Zielmaschine, so kann die Zeit für den Vorlauf fast vollständig durch ein Lesen dieser Dateien eingespart werden.

5.3 Kompaktierung

Die Komponente MSSV erzeugt für jede Zuweisung eine Liste von Versionen, d.h. eine Liste von äquivalenten Maschinenprogrammen. Aufgabe der folgenden Komponente MSSC ist es, für jede Zuweisung genau eine Version auszuwählen und daraus wie in Abschnitt 4.6 eine Sequenz von Befehlen zu erzeugen. Konzeption und Implementierung von MSSC erfolgten durch R. Jöhnk [Jöh85]. Wesentliche Unterschiede zu Abschnitt 4.6 ergeben sich durch eine möglicherweise große Zahl von Versionen.

Die Kompaktierung erfolgt jeweils lokal für die Zuweisungen eines parallelen Blocks. Dieser parallele Block kann im Gegensatz zu den sog. Basisblöcken, die höchstens an ihrem Ende einen Sprung enthalten dürfen, aber noch bedingte Anweisungen einschließen. Die Zuweisungen innerhalb eines solches Blockes werden (Abwesenheit von Anti-Datenabhängigkeitszyklen vorausgesetzt) derart sortiert, daß für eine Liste $s_1, \ldots, s_n$ von Zuweisungen stets gilt:

$$\forall\ i,j \in [1..n], i \neq j : s_i \ \mathbf{ada}^* \ s_j \ \rightarrow i > j$$

Für $j < i$ beschreibt also s_j nie eine Zelle, die s_i liest.

Um möglichst schnellen Code erzeugen zu können, analysiert MSSC die Ausführungszeiten der einzelnen Versionen. Soweit im Rahmen der obigen Sortierung noch die Möglichkeit dazu besteht, werden die Zuweisungen derart angeordnet, daß für $k < l$ stets die schnellste Version von s_k nicht schneller ist als die schnellste Version von s_l (den langsamsten Zuweisungen werden also die kleinsten Indices zugeordnet).

Die Kompaktierung wählt nun stets die ungepackte Zuweisung mit dem kleinsten Index und versucht, (mit der schnellsten Version beginnend) eine ihrer Versionen einer Mikroinstruktion (MI) zuzufügen. Gelingt das nicht, so wird das gleiche mit der nächsten Zuweisung versucht. Ist die Liste der Zuweisungen abgearbeitet, so wird eine neue (leere) MI erzeugt und der Prozeß beginnt von vorn.

In der Liste der Zuweisungen werden jeweils nur solche Zuweisungen betrachtet, die nicht von ungepackten Zuweisungen anti-datenabhängig sind (Zykel werden getrennt behandelt).

Besonders betrachtet werden muß der Fall einer Version, die nicht innerhalb einer Mikroinstruktion ausführbar ist. Solche Versionen heißen sequentielle Versionen. Algorithmus 5.8 erzeugt in diesem Fall eine Sequenz, bestehend aus einer Zuweisung zu einer Hilfszelle und einer zweiten Zuweisung, die ein Lesen der Hilfszelle enthält. Für beide Zuweisungen generiert MSSV Versionen und zwar für die erste nur nicht

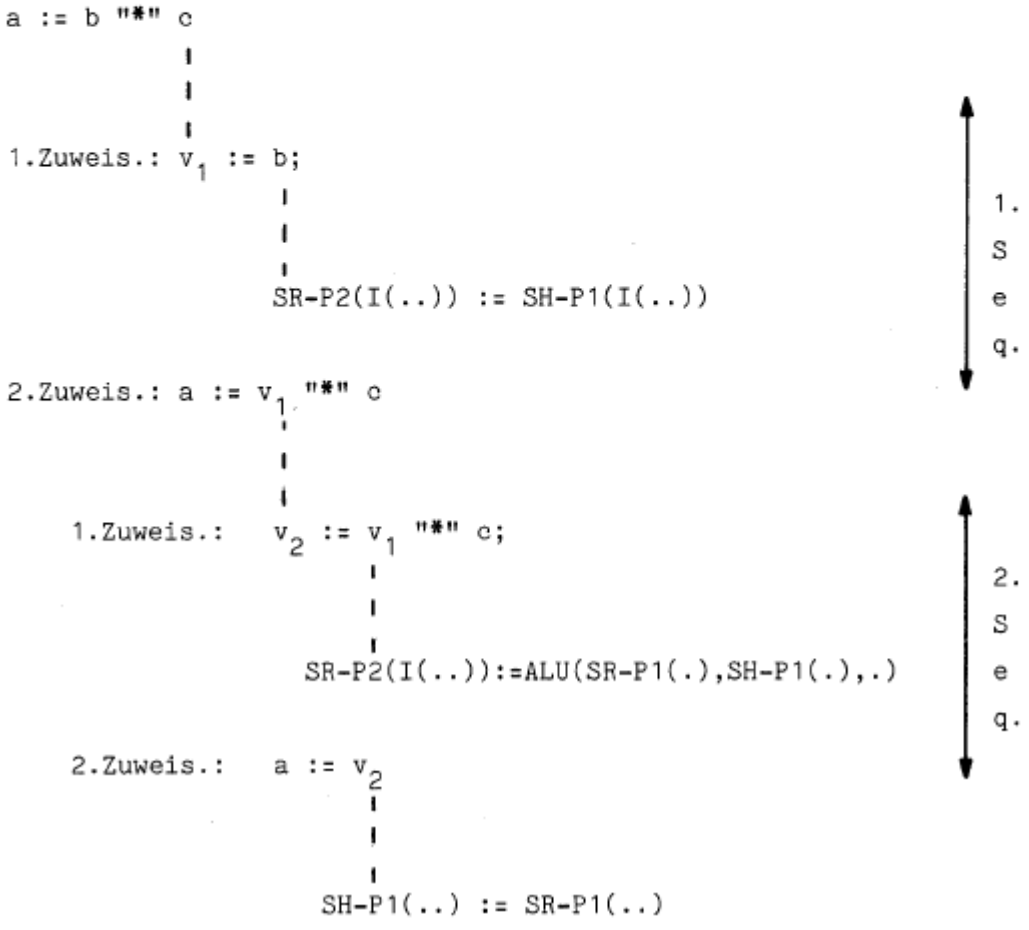
sequentielle und für die zweite beliebige. Im Falle sequentieller Versionen für die zweite Zuweisung wiederholt sich die Struktur rekursiv.

Beispiel:

Zu erzeugen sei Code für die Anweisung

a:=b "*" c

Jede Variable sei dem Hauptspeicher zugeordnet und dieser habe lediglich ein Port. Die Hilfsvariablen v_i seien dem Registerspeicher zugeordnet. MSSV könnte folgende Versionen generieren (-- zeigt auf Versionen) :



Das oben erklärte Verfahren wird zunächst für die erste Anweisung einer sequentiellen Version angewandt. Ist diese gepackt, so wird die Menge der zu packenden Anweisungen um die zweite Zuweisung erweitert.

Anschließend an das Packen werden die benötigten Zuweisungen an den Programmzähler nachgetragen und es wird dafür gesorgt, daß alle in einer MI nicht vorkommenden Speichereingänge und Bus-Treiber keine Operation ausführen (vgl. "INHIBIT-Codes", Abschn. 2.1.1).

Vorteile der Kompaktierung liegen u.a. darin, daß sie die Bearbeitung einer großen Zahl von Versionen erlaubt, daß die Versionsauswahl von der Ausführungszeit abhängig ist, daß sie keine Standardannahmen über INHIBIT-Codes benötigt und daß sie durch die Behandlung der Versionen von bedingten Anweisungen über eine Basisblock-lokale Kompaktierung hinausgeht.

5.4 Anwendungen und Erfahrungen

Der Codegenerator wurde zur Erzeugung von Code für eine Vielzahl unterschiedlicher Rechnerstrukturen benutzt. Die Zahl der Strukturen dürfte 20 übersteigen.

Eingesetzt wurde der Codegenerator u.a. im Rahmen einer Übung zur Vorlesung "Rechnerorganisation", um den Studenten die Wechselwirkungen zwischen Maschinenbefehlssatz, mikroprogrammiertem Befehlsinterpreter und interner Rechnerstruktur zu verdeutlichen. Der Ablauf der Interpretation klassischer Maschinenbefehle wurde vielen Studenten dabei erstmalig klar.

Immer wieder überrascht waren die Anwender, wenn ein Programm wegen fehlender Verbindungen nicht übersetzt werden konnte. Gerade hier passieren bei einem manuellen Entwurf der Hardware offenbar viele Fehler. Die Möglichkeit, fehlende Datenwege zu entdecken, sollte den Einsatz des MSS selbst bei handentworfenen Rechnern lohnend machen.

Beim Entwurf des Rechners SAMP [Now84] wurde der Codegenerator während der manuellen Entwurfsiterationen eingesetzt und dient z.Zt. der Erzeugung von Code für die realisierte Hardware. Auf diese Weise wird für eine Maschine mit einem 142 Bit breiten Befehlswort Code erzeugt. Als Quellprogramme werden dabei u.a. Programme zum Sortieren von Zahlen, zur Lösung des Knappsack-Problems und zur rekursiven Berechnung der Ackermann-Funktion benutzt. Damit ist es möglich, maschinenunabhängig geschriebene Anwenderprogramme in horizontalen (Mikro-) Code zu übersetzen. Eine manuelle Erzeugung des Codes wäre bei der vorliegenden parallelen Struktur so gut wie ausgeschlossen.

In einem weiteren Anwendungsfall wurde ein industriell genutzter Prozessor in MIMOLA beschrieben. Die Bausteine dieses Prozessors werden meist nicht direkt von einem eigenen Befehlsfeld gesteuert. Statt dessen befinden sich in der Regel Dekoder zwischen Befehlsfeldern und Kontrolleingängen und jedes Befehlsfeld steuert mehrere Bausteine. Dadurch entsteht bei der Übersetzung eine große Zahl von partiellen Versionen, die später beim Bundling infolge von Konflikten wieder verworfen werden müssen. Die entstehenden Laufzeiten liegen daher etwa an der Grenze dessen, was von einem Benutzer akzeptiert werden kann. Sofern die auftretenden Ausdrücke aber nicht zu komplex sind, kann auch in diesem Fall Code erzeugt werden.

Der Prozessor ist mit AMD-Bitslice-Bausteinen [AMD83] aufgebaut. Die Beschreibung dieser Bausteine würde übersichtlicher, wenn man eine hierarchische Beschreibung der Bausteine benutzen könnte. Entsprechende Arbeiten wurden bereits begonnen und lassen keine weiteren Probleme erwarten.

Das Beispiel dieses Prozessor zeigt, wo die Grenzen des jetzigen Codegenerators liegen. Diese sollen im folgenden umrissen werden.

Der Entwurf des Codegenerators basiert auf einer Reihe von gesetzten Schwerpunkten:

- möglichst enge Übereinstimmung zwischen der internen Rechnerstruktur und der Hardwarebeschreibung
- Vorrang der raschen Änderbarkeit vor der Übersetzungsgeschwindigkeit
- Betonung von Rechnern, die
  - mehrere Zuweisungen pro Befehl ausführen können
  - ein einfaches Timing und wenige "Tricks" verwenden
  - eine schwache Kodierung der Befehlsfelder benutzen
    (D.h. die Mehrzahl der Hardwarebausteine wird direkt und möglichst von einem eigenen Befehlsfeld gesteuert (sog. "direct encoding" [AgrRau76]))

In gewissem Umfang ist auch für Rechner, die nicht diser Schwerpunktsetzung entsprechen, eine Codeerzeugung sinnvoll und möglich. Eine starke Kodierung läßt sich z.B. durch Hardware-Dekoder darstellen. Tricks und Sonderfälle können mittels Programmtransformationsregeln erfaßt werden. Umso weiter man aber von der Schwerpunktsetzung abweicht, umso eher kann die Übersetzungsgeschwindigkeit durch allzu viele Transformationsregeln und rekursive Aufrufe von **expression** auf ein unbequemes Maß sinken. Es ist geplant, Erfahrungen, die beim Entwurf des zuletzt entstandenen Synthesesystem gemacht wurden, auf den Codegenerator zu übertragen und damit dessen Anwendungsbereich zu erweitern.

Als Erfahrung aus den Anwendungen kann festgehalten werden, daß es gelungen ist, einen maschinenunabhängigen Übersetzer zur Erzeugung parallelen Codes zu erstellen. Die Form der Maschinenbeschreibung war in den betrachteten Fällen anwendbar. Eine detailliertere Behandlung des Zeitverhaltens im Codegenerator war entbehrlich. Allerdings wäre die Einbeziehung von Speichern, die durch Schreiboder Leseoperationen während mehrerer Befehlszyklen beschäftigt sind, nützlich und auch in das jetzige Verfahren einzubeziehen.

Die Erfahrung bestätigt, daß der vorgestellte Ansatz einen gangbaren Weg zur Erzeugung von Mikrocode darstellt. Verfeinerungen mit dem Ziel einer Erhöhung der Übersetzungsgeschwindigkeit und einer Erweiterung des Anwendungsbereichs tangieren nicht das Grundkonzept einer Maschinenbefehlsbeschreibung in Form der tatsächlich vorhandenen Hardware.

6. Systemunterteile

6.1 Ubersicht

Die drei Hauptteile des MIMOLA-Systems sind das Synthese-Subsystem, das Codeerzeugungs-Subsystem und die Testerzeugung. Die Ausgaben dieser drei Hauptteile können gemäß Abb. 3.1 von Komponenten des MSS weiterverarbeitet werden, die unter dem Begriff "Systemunterteile" zusammengefaßt werden.

Abb. 6.1 zeigt einen Überblick über diese Komponenten.

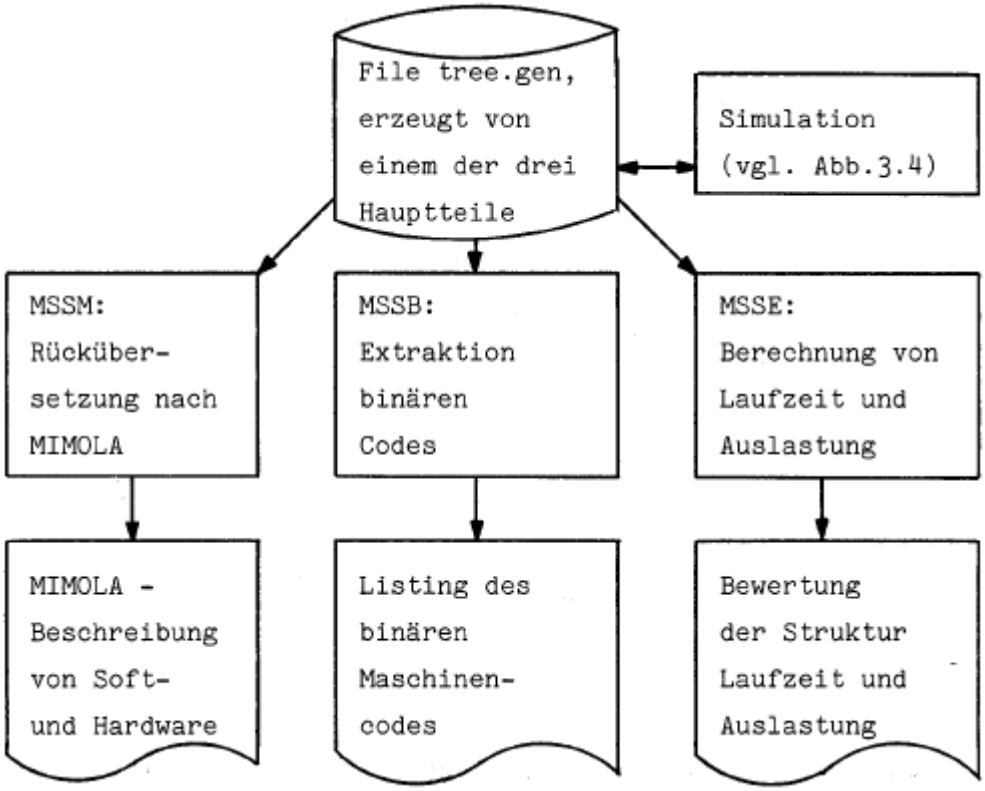


Abb. 6.1 Systemunterteile

Gearbeitet wird z.Zt. an einer weiteren Komponente MSSG, die Blockdiagramme der Rechnerstruktur erzeugen soll. Ein wesentliches Problem liegt dabei in der intelligenten Anordnung der Bausteine. Lösungsverfahren besitzen Ähnlichkeiten mit Verfahren zur Erzeugung von Layouts. Ein kürzlich publiziertes Verfahren [AryKumSwaMis85] scheint im Gegensatz zu einem 1984 publizierten Ansatz [Ah1HadStr84] zufriedenstellende Ergebnisse zu liefern.

6.2 Simulation

Der Simulator des MSS ist in der Lage, sowohl gebundene als auch ungebundene RT-Programme zu simulieren. Damit kann er als Systemoberteil und als Systemunterteil benutzt werden.

Die Simulation gebundener RT-Programme erfüllt folgende Aufgaben:

- Bestimmung von Ausführungshäufigkeiten mit dem Ziel der Benutzung in MSSE (s.u.)
- Überprüfung, ob Ausgabeabhängigkeiten (vgl. Abschnitt 4.6) vorliegen (der Benutzer könnte sich über Warnungen des MSS hinweggesetzt haben)
- Überprüfung der korrekten Funktion der übrigen Komponenten des MSS

Die Simulation gebundener Programme könnte im Prinzip von der Struktur abstrahieren und nur die Funktion simulieren. Damit könnte aber die letzte der genannten Aufgaben nicht mehr erfüllt werden. Daher berücksichtigt die Simulation gebundener Programme die Struktur des Rechners. So werden z.B. alle Modultypen durch Prozeduren dargestellt, in deren Rumpf eine Operation durch einen Kontrollcode ausgewählt wird. Sollte der Codegenerator einen für diesen Baustein nicht definierten Kontrollcode erzeugen, so würde dies bei der Simulation erkannt. Eine rein funktionelle Simulation dagegen würde die Operation ausführen, ohne auf ein Modul oder einen Kontrollcode Bezug zu nehmen. Möglich wäre dies, da die benutzten TREEMOLA-Files die ausgeführten Operationen als Pseudo-Kommentar enthalten.

Durch die Simulation der generierten Maschinenprogramme auf der Struktur ergibt sich die Möglichkeit einer **Überprüfung der Arbeitsweise des Synthesesystems** und **des Codegenerators.** Solange das MSS selbst nicht formal verifiziert ist, ergibt sich dadurch eine zusätzliche Sicherheit im Entwurfsprozess.

6.3 Rückübersetzung nach MIMOLA

Eines der wesentlichen Prinzipien beim Entwurf des MSS war es, einzelne Teilaufgaben des MSS möglichst als Programmtransformationen darzustellen und so eine Zerlegung des Systems in Komponenten zu erleichtern.

Um die Ergebnisse aller beabsichtigten Transformationen wieder in der gleichen Sprache ausdrücken zu können, sind sowohl MIMOLA als auch TREEMOLA möglichst allgemein definiert. Dadurch können sowohl PASCAL-ähnliche Programme wie auch Maschinenprogramme und Hardware in MIMOLA beschrieben werden (in diesem Sinne ist MIMOLA eine "Breitbandsprache").

Infolgedessen ist es möglich, TREEMOLA-Files gebundener Programme wieder nach MIMOLA zu übersetzen. Diese Rückübersetzung wird benötigt, um Ergebnisse in lesbarer Form auszudrucken und um Entwurfsiterationen zu unterstützen (vgl. Abb. 4.2).

Da Maschinenprogramme sehr viele Details enthalten, können einige dieser Details optional während der Rückübersetzung unterdrückt werden. So ist es möglich, z.B. Multiplexer und Busse auszulassen oder statt der arithmetisch/logischen Bausteine die ausgeführten Operationen einzusetzen.

6.4 Extraktion binären Codes

In den vollständig gebundenen Programmen, wie sie von den Hauptteilen des MSS erzeugt werden, ist bereits der gesamte Programmcode enthalten. Die Software-Komponente MSSB übernimmt die Aufgabe, aus vollständig gebundenen Programmen in TREEMOLA-Darstellung lesbare Listen binären Codes zu erzeugen.

In der Eingabe von MSSB können noch symbolische Marken und Unterprogrammadressen sowie Character-Strings vorkommen. Diese werden von MSSB durch die binären Werte ersetzt. Damit kann MSSB im Prinzip

als Binder benutzt werden. Die Umsetzung von Charactern in binäre Werte kann unabhängig von der Character-darstellung auf dem Wirtsrechner erfolgen, da eine entsprechende Umsetztabelle durch den Benutzer angegeben werden kann.

6.5 Bewertung

Insbesondere die Vorbereitung von Entwurfsiterationen bedarf einer Bewertung der bislang erzeugten Rechnerstruktur. Diesen Zweck erfüllt die Komponente MSSE. MSSE berechnet:

Die Laufzeit der Programme auf der Hardware Die Auslastung der Bausteine Die Häufigkeiten der gleichzeitigen Benutzung von Bausteinen und Operationen Die kritischen Wege innerhalb der Rechnerstruktur

Die Berechnung der Programmlaufzeit basiert auf der Formel

$$\text{Laufzeit} = \sum_{\text{alle Befehle}} \text{Dauer des Befehls} * \text{Ausführungshäufigkeit}$$

Die Ausführungshäufigkeiten können entweder vom Benutzer als sog. Properties in den Quelltext eingefügt oder durch den Simulator berechnet werden. Die Komponente MSSW dient dem Kopieren der durch Simulation gewonnenen Werte in TREEMOLA-Files (vgl. Abb. 3.4).

Die Dauer der Befehle kann aus den Verzögerungszeiten abgeleitet werden, die Teil der Beschreibung von Modultypen sind. Problematisch bleibt allerdings die Berechnung von Verzögerungszeiten auf Leitungen, sofern eine Realisierung als ULSI-Schaltkreis vorgesehen ist und kein Layout vorliegt.

Weitere Details entnehme man einer Publikation [Mar81].

7. Zusammenfassung

Aufgrund der wachsenden Komplexität digitaler Schaltkreise lassen sich diese in der Zukunft nicht länger auf dem Gatter-Niveau entwerfen. Die vorliegende Arbeit beschreibt ein **Software-System zum computerunterstützten Entwurf digitaler Rechner**, in dem diese Komplexitätsprobleme durch Verwendung höherer Entwurfsebenen zumindest reduziert werden. Auf diese Weise wird der Bereich des reinen Hardwareentwurfs zugunsten eines **integrierten Entwurfs von Hardware und Software** verlassen. Folglich können die Konzepte und Methoden dieser Arbeit traditionell getrennten Forschungsgebieten zugerechnet werden, da sowohl Aspekte des Compilerbaus als auch des Hardwareentwurfs betrachtet werden.

Voraussetzung für die beschriebenen Verfahren war die Entwicklung der Sprache **MIMOLA**. Mit dieser Sprache **können sowohl Hardware als auch Software beschrieben werden**. Wesentlich ist dabei die Trennung zwischen der Darstellung von Hardware und Software. Erst diese (für reine Simulationssysteme unnötige) Trennung ermöglichte die Entwicklung der vorgestellten Algorithmen.

Weitere wesentliche Eigenschaften der Sprache MIMOLA sind die Möglichkeiten zur Definition **paralleler Blöcke** und der **Beschreibung von maschinennahen Programmen.**

Die gewählte Definition der Semantik paralleler Blöcke spiegelt Mengen in der Hardware streng synchron ausgeführter Zuweisungen wider. Gegenüber quasi-sequentiellen Zuweisungen wie in ISPS ergibt sich hierdurch ein Vorteil für den Benutzer. Es zeigt sich, daß dieser zu einer aufwendigeren Betrachtung der Datenabhängigkeiten führt.

Von den vier Kernstücken des MIMOLA Entwurfssystems, nämlich der Hardware-Synthese, der Codeerzeugung, der Testerzeugung und der Simulation wurden die beiden ersten im Rahmen dieser Arbeit erstellt und und hier vorgestellt.

Bei der **Hardware-Synthese** wird zu einem vorgegebenen Programm und einer Menge von Entwurfsrestriktionen eine dem Programm angepaßte

Hardware generiert und so das Studium von Software/Hardware-Tradeoffs ermöglicht. Der große Vorteil der Hardware-Synthese liegt im Ausschluß manueller Entwurfsfehler. Sofern das Synthesesystem selbst korrekt ist, sind die erzeugten Hardwarestrukturen automatisch korrekt.

In dieser Arbeit wird ein neues Verfahren zur Zerlegung des Syntheseproblems in eine Reihe von Teilproblemen handhabbarer Komplexität angegeben.

Um im Vergleich zu anderen Synthesesystemen Hardware einzusparen, werden Variable in adressierbaren Speichern und nicht (wie üblich) in eigenen Registern untergebracht. Wegen der beschränkten parallelen Zugriffsmöglichkeiten auf Speicher entsteht dadurch als erstes wesentliches Teilproblem das Problem der **Zerlegung komplexer Ausdrücke** in eine Menge einfacherer Ausdrücke. Diese Arbeit stellt ein neues Verfahren zur Zerlegung komplexer Ausdrücke vor, welches vorhandene gemeinsame Teilausdrücke zur Optimierung nutzt.

Soweit eine ausreichende Zahl von Hardware-Ressourcen gemäß der Entwurfsrestriktionen zulässig ist, generiert die Synthese Befehle, die eine **parallele Ausführung mehrerer Zuweisungen** bewirken. Dadurch soll die Ausführungsgeschwindigkeit im Vergleich zu üblichen sequentiellen Maschinen gesteigert werden. Die Suche nach geeigneten parallel ausführbaren Zuweisungen wird als **Kompaktierung** bezeichnet. Es wird ein neues Kompaktierungsverfahren vorgestellt, welches insbesondere die Erfordernisse paralleler Blöcke berücksichtigt. Um bei der Kompaktierung möglichst viele Freiheiten zu haben, werden die Register zur Aufnahme von Zwischenergebnissen im Gegensatz zur gängigen Praxis **nach** statt **vor** der Kompaktierung vergeben.

Für unterschiedliche Technologien liegen meist vorentworfene HardwareBausteine wie z.B. arithmetisch/logische Netzwerke vor. Solche Bausteine können pro Programmschritt jeweils eine Operation aus einer Liste von Operationen oder Funktionen ausführen. Sie heißen daher Multifunktionsbausteine. Die kostengünstige **Auswahl von Multifunktionsbausteinen** wird in dieser Arbeit auf die lineare Programmierung zurückgeführt. Die Zahl der benötigten Variablen wird im Vergleich zu für einfache

Bausteine existierenden Verfahren so weit reduziert, daß eine Lösung der linearen Programme in vertretbarer :Zeit möglich wird.

Aufgrund des Flächenbedarfs von Leitungen in der VLSI - Technologie wichtig ist ferner die **Minimierung der Zahl benötigter Verbindungen** durch das MIMOLA-System.

Die als Resultat der Synthese erhaltenen Rechnerstrukturen sind sowohl in bezug auf die Laufzeiten der Programme wie auch in bezug auf die Dichte des Codes zumindest nicht schlechter als konventionelle Rechner, häufig sogar schneller. Entgegen einer verbreiteten Auffassung ergibt sich sogar bei einer direkten Ansteuerung der Hardware-Bausteine durch Felder des aktuellen Befehls (sog. "direct encoding" [AgrRau76]) ein kompakter Programmcode.

Der zweite Hauptteil dieser Arbeit geht auf den **Codegenerator** des MIMOLA-Systems ein. Dieser transformiert vorgegebene PASCAL-artige Programme in Maschinenprogramme.

Die Besonderheit liegt erstens darin, daß der Codegenerator die **Erzeugung parallelen Mikrocodes** erlaubt. Bislang wird Mikrocode noch überwiegend mit Hilfe von Assemblern erzeugt. Es ist damit möglich, Geschwindigkeitsvorteile durch Mikroprogrammierung nicht nur für wenige Systemprogramme, sondern auch für Anwenderprogramme zu nutzen.

Zweitens ist der Codegenerator **nicht** für eine **feste Zielmaschine** geschrieben. Statt dessen kann diese durch eine MIMOLA-Hardwarebeschreibung definiert werden. Diese Beschreibung kann durch den HardwareDesigner selbst erfolgen und erfordert keine besonderen Kenntnisse im Bereich des Compilerbaus. Hardware-Änderungen sind in der Codeerzeugung einfach zu berücksichtigen. Durch die Kompatibilität der Hardwarebeschreibung mit den übrigen Systemkomponenten ergibt sich ein 11synenergetischer Effekt".

Die **Steuerung** des Codegenerators erfolgt ganz wesentlich **in Abhängigkeit der** in der Hardware vorhandenen **Leitungen**. Dabei werden auch Leitungen unterschiedlicher Breite, verschränkte Leitungen, Einzelbits und "Um

wege" über arithmetisch/logische Einheiten berücksichtigt.

Ein kleines, in den Codegenerator eingebautes "Mini"-Expertensystem erlaubt die Eingabe von **Programmtransformationsregeln**. Auf diese Weise können Transformationen, die zur Codeerzeugung für eine bestimmte Hardware nützlich sind, dem System mitgeteilt werden. Dadurch gelingt es, die von Mikroprogrammierern benutzten "Tricks" auch bei einer automatischen Übersetzung auszunutzen.

Der Codegenerator wurde u.a. im Rahmen einer Übung eingesetzt, um Studenten die Bedeutung der internen Register-Transfers mikroprogrammierter Rechner klarzumachen. Durch die Benutzung des Codegenerators wurde erreicht, daß die Studenten nicht durch Schwierigkeiten bei der manuellen Erzeugung von Mikrocode von dem eigentlichen Ziel der Übung abgelenkt wurden.

Gegenwärtig wird der Codegenerator eingesetzt, um Code für einen Rechner zu erzeugen, der in einem Parallelprojekt entwickelt und aufgebaut wurde. Eine manuelle Erzeugung von Code für diesen Rechner wäre bei der vorliegenden Struktur so gut wie ausgeschlossen.

Insgesamt stehen mit den hier beschriebenen Komponenten des MIMOLASoftware-Systems Werkzeuge zur Verfügung, welche die Erstellung von Mikroprogrammen in einer höheren Programmiersprache und das Studium der Wechselwirkungen zwischen Hardware und Software ermöglichen. Alle beschriebenen Komponenten sind implementiert und auf Rechnern mehrerer Typen verfügbar.

<u>Schlußbemerkung</u>

Der Autor dieser Arbeit dankt allen, die die Entwicklung des MIMOLASoftware-Systems durch ihre Mitarbeit oder ihre Förderung ermöglicht haben. Ganz besonders hervorzuheben ist die ideelle Unterstützung durch Herrn Prof. Zimmermann, Herrn T. Berger, Herrn R. Jöhnk, Herrn G. Krüger und Herrn L. Nowak sowie die finanzielle Unterstützung seitens des Bundesministeriums für Forschung und Technologie.

Stichwortverzeichnis

Literaturverzeichnis

[AgrRau76] A.K. Agrawala und T.G. Rauseher: Foundations of Microprogramming, Acadamic Press, New York, 1976

[Ah1HadStr84] M.L. Ahlstrom, G.D. Hadden und G.R. Stroick An Expert System for the Generation of Schematics, IEEE Int. Conf. an Computer Design: VLSI in Computers (ICCD), 1984, S. 720-725

[AhoJoh76] A.V. Aho und S.C. Sethi : Optimal Code Generation for Expression Trees, Journal of the ACM, Vol. 23, 30976), S. 488-501

[AMD83] Advanced Micro Devices : Bipolar Microprocessor Logic and Interface Data Book, Sunnyvale, 1983

[AnaClaFosMis85] T.S. Anantharaman, E.M. Clarke, M.J. Foster und B. Mishra : Compiling Path Expressions into VLSI Circuits, 12th ACM Annual Symp. an Principles of Programming Languages, 1985, S. 191-204

[AneLidMerPay69] F. Anceau,P. Liddell,J. Mermet und C. Payan: CASSANDRE : A Language to Describe Digital Systems, Software Engineering, COINS III, Proc. 3rd Symp. an Computer and Information Sciences, 1969, S. 179-204

[ANSI/IEEE83] IEEE Pascal Standards Committee und American National Standards Committee X3 : IEEE Standard Pascal Computer Programming Language, IEEE, 1983

[AryKumSwaMis85] A. Arya, A. Kumar, V.V. Swaminathan und A. Misra : Automatic Generation of Digital System Schematics, 22nd Design Automation Conf., 1985, S. 388-395

[BabHag81] T. Baba und H. Hagiwara : The MPG System : A Machine-Independent Efficient Microprogram Generator, IEEE Trans. Comp., Vol. C-30, 6(1981) S. 373-395

[Ban79] U. Banerjee : Speedup of Ordinary Programs, Bericht No. UIUCDCS-R-79-989, Dpt. of Computer Science, Universität Illinois, Urbana - Champaign, 1979

[BarBarCatSie77] M.R. Barbacci, G.E. Barnes, R.C. Cattell und D.P. Siewiorek : The ISPS Computer Description Language, Dpt. of Computer Science, Carnegie-Mellon Universität, Pittsburgh, 1977

[Bau78] F.L. Bauer : Design of a programming language for a program transformation system, Informatik-Fachberichte, Bd.16, Springer, Berlin, 1980, S.1-27

[BauW6s81] F.L. Bauer und H. W6ssner : Algorithmische Sprache und Programmentwicklung, Springer, Berlin, 1981

[Ber85] T. Berger : Ein Programm zur Speicheroptimierung im MIMOLA-Software-System (Diplomarbeit), Institut für Informatik und Prakt. Mathematik der Universität Kiel, 1985

[Bod84] A. Bode : Mikroarchitekturen und Mikroprogrammierung: Formale Beschreibung und Optimierung, Informatik-Fachberichte, Bd. 82, Springer, Berlin, 1984

[BurChrMat83] M.R. Buric, C. Christensen und T.G. Matheson: PLEX : Automatically Generated Microcomputer Layouts, IEEE Int. Conf. an Computer Design: VLSI in Computers (ICCD), 1983, S. 181-184

[Bus78] R.G. Bushell : Higher level language for microprogramming, Euromicro Journal, Vol.4, 2(1978), S. 67-75

[Cat78] R.G.G. Cattell, Formalization and Automatic Derivation of Code Generators, Dissertation, Carnegie-Mellon Universität, Pittsburgh, 1978

[Cam85] R. Camposano : Synthesis Techniques for Digital Systems Design, 22nd Design Automation Conf., 1985, S. 475-481

[Chu72] Y. Chu : Computer Organization and Microprogramming, Prentice-Hall, Englewood Cliffs, 1972

[Coo85] K. Cooper : Analyzing Aliases of Reference Formal Parameters, 12th Annual ACM Symp. an Principles of Programming Languages, 1985, 5.281-290

[DamBarHaljoo81l A. van Dam, M.R. Barbacci, C. Halatsis und J. Joosten : Simulation of a Horizontal Bit Sliced Processor : The MICE Experience, 5th Int. Symp. an Computer Hardware Description Languages, 1981, S. 281-291

[Darjoy80] J.A. Darringer und W.H. Joyner : A New Look at Logic Synthesis, 17th Design Automation Conf., 1980, S. 543-549

[Das80] S. Dasgupta : Some Aspects of high-level micro-programming, Computing Surveys, Vol. 11, 3(1980), S. 295-323

[Das84] S. Dasgupta : A Model of Clocked Micro -Architectures for Firmware Engineering and Design Automation Applications, 17th Annual Microprogramming Workshop (MICRO-17), 1984, S. 298-308

[DasTar76] S. Dasgupta und J. Tartar : The identification of maximal parallelism in straight-line microprograms, IEEE Trans. Comp., Vol. C-25, 10(1976), S. 986-992

[DavLanShrMal81] S. Davidson, D. Landskov, B.D. Shriver und P.W. Mallett : Some Experiments in Local Microcode Compaction for Horizontal Machines, IEEE Trans. Comp., Vol. C-30, 7(1981), S. 460-477

[Den82] P.B. Denyer : An Introduction to Bit-Serial Architectures for VLSI Signal Processing, Arbeitsunterlagen des Advanced Course an VLSI Architecture, Bristol, 1982

[Der83] R. McDermott : Simulation of Simple Digital Logic through a Computer-Aided Design System, BYTE, 1983, S. 396-414

[DeuNew83] J.T. Deutsch und A.R. Newton : Data-Flow Based Behavioral-Level Simulation and Synthesis, IEEE Int. Conf. an Computer-Aided Design (ICCAD), 1983, S. 63-64

[DeWi76] D.J. DeWitt : A machine independent approach to the production of optimal horizontal microcode (Dissertation), Bericht 76 DT 4, University of Michigan at Ann Arbor, 1976

[DObPatDeS84] T.P. Dobry, Y.N. Patt und A.M. Despain Design Decisions Influencing the Microarchitecture for a PROLOG Machine, 17th Annual Microprogramming Workshop (MICRO-17), 1984, S. 217-231

[EvaGoe0fe79] C.J. Evangelisti, G. Goertzel und H. Ofek Using the Dataflow Analyzer an LCD Descriptions of Machines to Generate Control, 4th Int. Symp. an Computer Hardware Description Languages 1979, S. 109-115

[GajKuh83] D.D. Gajski und R.H. Kuhn, Guest Editors' Introduction : New VLSI Tools, Computer, Vol. 16, 12(1983), S. 11-14

[GanFisHen82] M. Ganapathi, C.N. Fisher und J.L. Hennessy: Retargetable Compiler Code Generation, Computing Surveys, Vol. 14, 4(1982), 5.573-593

[GarJoh79] M.R. Garey und D.S.IJohnson : Computers and Intractability : A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979

[GirKni84] E.F. Girczyc und J.P. Knight : An ADA to Standard Cell Hardware Compiler Based an Graph Grammars and Scheduling,IEEE Int. Conf. an Computer Design: VLSI in Computers (ICCD), 1984, S. 727-731

[GraBieHal80] W. Grass, G. Biehl und S. Hall : LOGE-MAT, A Program for the Synthesis of Microprogrammed Controllers, CAD 80, Brighton, S. 543-558

[GraBucRob83] J.P. Gray, I. Buchanan und P.S. Robertson Controling VLSI Complexity Using a High-Level Language for Design Description, IEEE Int. Conf. an Computer Design: VLSI in Computers (ICCD), 1983, S. 523-526

[HafPar83] L. Hafer und A. Parker : A Formal Method for the Specification, Analysis and Design of Register - Transfer Level Digital Logic, IEEE Trans. an Computer-Aided Design, Vol. CAD-2, 1(1983), S.4-18

[HarLem84] R. Hartenstein und K. Lemmert : KARL-III Language Reference Manual, Fachbereich Informatik, Universität Kaiserslautern, 1984

[Hec77] M.S. Hecht : Flow Analysis of Computer Programs, North Holland, Amsterdam, 1977

[HitTho83] C.Y. Hitchcock und D.E. Thomas : A Method of Automatic Data Path Synthesis, 20th Design Automation Conf., 1983, S. 484-489

[Hua81] C.-L. Huang : Computer-Aided Logic Synthesis Based an a New Multi-Level Hardware Design Language -- LALSD II (Dissertation), State University of New York at Binghamton, 1981

[HolIbb80] R.W. Holgate und R.N. Ibbett : An Analysis of Instruction Fetching Strategies in Pipelined Computers, IEEE Trans. Comp., Vol. C-29, 40980), 5.325-329

[Joh79] D. Johannsen : Bristle Blocks : A Silicon Compiler, 16th Design Automation Conf., 1979, S_~~n-~~~

[Jöh81] R. J6hnk : Programm- und Datenstrukturen für das MIMOLA-Software-System (Diplomarbeit), Institut für Informatik und Prakt. Mathematik der Universität Kiel, 1981

[J6h85] R. J6hnk : Mikrocode-Kompaktierung in der Komponente MSSC des MIMOLA-Software-Systems (Arbeitstitel), Interner Bericht (in Vorbereitung)

[J6hMar] R. Jöhnk und P. Marwedel : MIMOLA Language Reference Manual, Revision 3, (in Vorbereitung)

[Jes75] E.Jessen : Architektur digitaler Rechenanlagen, Springer, Berlin, 1975

[KanLan73] P. Kandzia und H. Langmaack : Informatik Programmierung, Teubner, Stuttgart, 1973

[KarTriU1183] A.R. Karlin, H.W. Trickey und J.D. Ullman Experience with a Regular Expression Compiler, IEEE Int. Conf. an Computer Design: VLSI in Computers (ICCD), 1983, S. 656-665

[Ke185] S.H. Kelem : A Method for the Automatic Translation of Algorithms from a High-Level Language into Self-Timed Integrated Circuits, IEEE Circuits and Devices Magazine, 1985, S.17-19,44

[KelWos85] K. Kelle und F. Wosnitza : Zwischenbericht des Projektes NT 2816/9 des Bundesministeriums für Forschung und Technologie, Kap. 3, 1985

[KimTan79] J. Kim und C.J. Tan : Register Assignment for Optimizing Microcode Compilers -- Part I, Research Report RC 7639 (#33035), IBM T.J. Watson Research Center, Yorktown Heights, 1979

[Kod72] U.R. Kodres : Partioning and Card Selection, in: M.A. Breuer (Hrg.) : Design Automation of Digital Systems,Vo1.1, Prentice Hall, Englewood Cliffs, 1972

[Kon83] T. Konow : Parallelisierung von MIMOLA-Programmen (Diplomarbeit), Institut für Informatik und Prakt. Mathematik der Universitdt Kiel, 1983

[KowTho83] T.J. Kowalski und D.E. Thomas : The VLSI Design Automation Assistant : Prototype System, 20th Design Automation Conf., 1983, S. 479-483

[Kra81] G. Krasner : The Smalltalk-80 Virtual Machine, BYTE, 1981, S. 300-319

[Krü80] G. Krüger : Entwurf einer Rechnerzentraleinheit für den Maschinenbefehlssatz des Siemens Systems 7.000 mit dem MIMOLA-Rechnerentwurfssystem (Diplomarbeit), Institut für Informatik und Prakt. Mathematik `der Universität Kiel, 1980

[Krü85] G. Krüger : Automatische Erzeugung von Test programmen, Zwischenbericht des Projektes NT 2816/9 des Bundesministeriums für Forschung und Technologie, Kap. 2, 1985

[KrüJöhMar] G. Krüger, R. J6hnk und P. Marwedel MIMOLA Software System : User's Guide, on-line Dokumentation, laufend aktualisiert

[Kuc78] D.J. Kuck : The Structure of Computers and Compu tations, Vol. I, Wiley, New York, 1978

[Lan75] D. Lancaster : Active Filter Cookbook, Prentice Hall, Englewood Cliffs, 1975

[Lan78] H. Langmaack : Gomory I, Collected Algorithms of the ACM, Algorithm 263A, 1978

[Lei81l G.W. Leive : The Design, Implementation and Analysis of an Automated Logic Synthesis and Module Selection System (Dissertation), Carnegie Mellon Universität, Pittsburgh, 1981

[Lie84] K.J. Lieberherr : Towards a Standard Hardware Description Language, 21st Design Automation Conf., 1984, S. 265-272

[Ma178] P.W. Mallett : Methods of compacting micropro grams (Dissertation), University of Southwestern Louisiana, Lafayette, 1978

[Mar79] P. Marwedel : The MIMOLA Design System : Detailed Description of the Software System, 16th Design Automation Conf., 1979, S. 59-63

[Mar80] P. Marwedel : The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA, in: G. Zimmermann : GI-NTG Fachtagung Struktur und Betrieb von Rechensystemen, Springer Informatik Fachberichte, Vol. 27, 1980

[Mar80a] P. Marwedel : Hardware Allocation for Horizon tal Microinstructions in the MIMOLA Software System, Bericht 5/80, Institut für Informatik und Prakt. Mathematik, Universität Kiel, 1980

[Mar81] P. Marwedel : Statistical Studies of Horizontal Microprograms, 5th Int. Symp. an Computer Hardware Description Languages, 1981, S. 281-291

[Mar84] P. Marwedel : A Retargetable Compiler for a High - Level Microprogramming Language,17th Annual Microprogramming Workshop (MICRO-17), 1984, S. 267-276,

[Mar85] P. Marwedel : The MIMOLA Design System : A Design System Which Spans Several Levels, in W.K. Giloi und B.D. Shriver (Hrg.) Methodologies for Computer System Design, North Holland, 1985, S. 223-237

[MarZim79] P. Marwedel und G. Zimmermann : MIMOLA-Report Revision 1 and MIMOLA Software System User Manual, Bericht 2/79, Institut für Informatik und Prakt. Mathematik der Universität Kiel, 1979

[MeaCon801 C. Mead und L. Conway : Introduction to VLSI Systems, Addison Wesley, London, 1980

[Mir79] G.S. Miranker : The Use of Conflict in the Translation and Optimization of Hardware Description Languages (Dissertation), Massachusetts Institute of Technology, Cambridge, 1979

[MueDudOHa84] R.A. Mueller, M.R. Duda und S.M. 0'Haire A Survey of Resource Allocation Methods in Optimizing Microcode Compilers, 17th Annual Microprogramming Workshop, 1984, S. 285-295

[MueVar83] R.A. Mueller und J. Varghese : Flow Graph Machine Models in Microcode Synthesis, 16th Annual Microprogramming Workshop (MICRO-16), 1983, S. 159-167

[MueVarAl184] R.A. Mueller, J. Varghese und V.H. Allan: Global Methods in the Flow Graph Approach to Retargetable Microcode Generation, 17 th Annual microprogramming Workshop (MICRO-17), 1984, S. 275-284

[Mye78] G.J. Myers : Advances in Computer Architecture, Wiley, New York, 1978

[Mye83] W. Myers : Extend Design Automation Systems, Computer, Vol. 16, 8(1983), S. 100-103

[Neu75] K. Neumann : Operations Research Verfahren, Bd. 1-3, Carl Hanser, München, 1975

[NicFis84] A. Nicolau und J.A. Fisher : Measuring the Parallelism Available for Very Long Word Architectures, IEEE Trans. Comp., Vol. C-33, 11(1984) S. 968-976

[Now84] L. Nowak : Entwurf eines hochgradig parallelen Rechners : ein Anwendungsbeispiel für MIMOLA, Vortrag auf einem Statusseminar am 4.9. 1984 im Institut für Theoretische Elektrotechnik der RWTH Aachen

[PadKucLaw80] D.A. Padua, D.J. Kuck und D.H. Lawrie : High - Speed Multiprocessors and Compilation Techniques, IEEE Trans. Comp., Vol. C-29, 9(1980), S. 763-776

[Par84] A.C. Parker : Automated Synthesis of Digital Systems, IEEE Design and Test of Computers, Vol.1, 4(1984), S. 75-81

[ParKurMli84] A.C. Parker, F. Kurdahl und M. Mlinar A General Methodology for Synthesis and Verification of Register-Transfer Designs, 21th Design Automation Conf., 1984, S. 329-335

[ParPar85] A.C. Parker und N. Park: Synthesis of Optimal Clocking Schemes, 22nd Design Automation Conf., 1985, S. 489-495

[PilBor85] R. Piloty und D. Borrione : The Conlan Project: Concepts, Implementations, and Applications, Computer, Vol. 18, 2(1985), S.81-92

[PraSet80] B. Prabhala und R. Sethi : Efficient Computation of Expressions with Common Subexpressions, Journal of the ACM, Vol. 27, 1(1980), S. 146-163

[Ram80] F.J. Rammig, CAP/DSDL : Preliminary Language Reference Manual, Bericht 129 der Abteilung Informatik der Universität Dortmund, 1980

[RajTho85] J. V. Rajan und D. E. Thomas : Synthesis by Delayed Binding of Decisions, 22nd Design Automation Conf., 1985, S. 367-373

[Ros82] W. Rosenstiel : DSL-Eine Sprache zur Spezifikation der Funktion digitaler Systeme, Bericht 9/82, Institut für Informatik IV, Universität Karlsruhe, 1982

[RosMarSch83] W. Rosenstiel, M. Marhöfer und D. Schmid: Das erste Gate-Array-Labor 1982/83 Entworfene Schaltungen, Werkzeuge, Testüberlegungen, Ergebnisse, Bericht 5/83, Inst. für Informatik IV, Universität Karlsruhe, 1983

[SahBha83] S. Sahni und A. Bhatt : The Complexity of Design Automation Problems, 20th Design Automation Conf., 1983, S. 402-411

[Sch80] F. Scholz : Registerzuteilung für Ausdrücke mit gemeinsamen Unterausdrücken auf dem Adressierungsniveau (Dissertation), Universität Karlsruhe, 1981

[Sch83] T. Schulz : Entwicklung schneller Prozessoren zur Bildbearbeitung (Diplomarbeit), Institut für Informatik und Prakt. Mathematik der Universität Kiel, 1983

[Sch84] U. Schmidt : Ein neuartiger, auf VDM basierender Codegenerator-Generator, Bericht 4/84, Institut für Informatik und Prakt. Mathematik, Universität Kiel, 1984

[SetU1170] R. Sethi und J.D. Ullman : The Generation of Optimal Code for Arithmetic Expressions, Journal of the ACM, Vol. 17, 4(1970), S. 715-728

[Sha85] M. Shadad, R. Lipsett, E. Marschner, K. Sheean, H. Cohen, R. Waxman und D. Ackley : VHSIC Hardware Description Language, Computer, Vol. 18, 2(1985), S. 94-103

[Sin80] M. Sint : A Survey of High Level Microprogramming Languages, 13th Annual Microprogramming Workshop (MICRO-13), 1980, S. 141-153

[Sin81] M. Sint : MIDL-A Microinstruction Description Language, 14th Annual Microprogramming Workshop (MICRO-14), 1981, S. 95-106

[Sou83] J.R. Southard : MacPitts : An Approach to Silicon Compilation, Computer, Vol. 16, 120983), S. 74-82

[Tak84] S. Takagi : Rule Based Synthesis, Verification and Compensation of Data Paths, Int. Conf. an Computer Aided Design (ICCAD) 1984, S. 133-138

[Tex77] The Engineering Staff of Texas Instruments The TTL Data Book for Design Engineers, Texas Instruments, 1977

[Tic84] E. Tick : Sequential PROLOG Machine : Image and Host Architectures, 17th Annual Microprogramming Workshop (MICRO-17), 1984, S. 204-216

[TorWil77] H.C. Torng und N.C. Wilhelm : The Optimal Interconnection of Circuit Modules in Microprocessor and Digital System Design, IEEE Trans. an Comp. Vol. C-26, 5(1977); S. 450-457

[Tre82] P. Treleaven : Data-Driven and Demand - Driven Computer Architecture, ACM Computing Surveys, Vol. 14, 1(1982), S. 93-143

[TseSie83] C.-J. Tseng und D.P. Siewiorek : Facet : A Procedure for the Automated Synthesis of Digital Systems, 20th Design Automation Conf., 1983 S. 490-496

[Veg82] S.R. Vegdahl : Local Code Generation and Compaction in Optimizing Microcode Compilers (Dissertation), Bericht CMU-CS-82-153, Carnegie-Mellon Universität, Pittsburgh, 1982

[Veg82a] S.R. Vegdahl : Phase Coupling and Constant Generation in an Optimizing Microcode Compiler, 15th Annual Microprogramming Workshop (MICRO-15), 1982, S. 125-133

[Veg83] S.R. Vegdahl : A New Perspective an the Classical Microcode Compaction Problem, SIGMICRO Newsletter, Vol. 14, 1(1983), S.11-14

[Wir75] N. Wirth : Algorithmen und Datenstrukturen, Teubner, Stuttgart, 1975

[Wir77] N. Wirth : Compilerbau, Teubner, Stuttgart, 197 7

[Zim76] G. Zimmermann : Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA, Springer Informatik Fachberichte, Vo1.6 1976, S. 465-478

[Zim77] G. Zimmermann : Report an the Computer Architecture Design Language MIMOLA, Bericht 4/77, Institut für Informatik und Prakt. Mathematik, Universität Kiel, 1977

[Zim79] G. Zimmermann : Cost Performance Analysis and Optimization of Highly Parallel Computer Structures : First Results of a Structured Top - Down Design Method, 4th Int. Conf. an Computer Hardware Description Languages, 1979, S. 33-39

[Zim80] G. Zimmermann : MDS-The MIMOLA Design Method, Journal of Digital Systems, Vol.4, 3(1980), S. 337-369

[Zim85] G. Zimmermann, mündliche Mitteilung, 1985

Anhang A : Syntax von RTL-TREEMOLA

Die folgenden Regeln geben die Syntax von RTL-TREEMOLA gemäß der externen Darstellung auf Files wieder. Die Darstellung erfolgt in einer modifizierten BNF-Form. Dabei bedeuten:

```
{..}    : mindestens ein Vorkommen des Ausdrucks in Klammern
[..]    : höchstens ein Vorkommen des Ausdrucks in Klammern
[{..}] : beliebig viele Vorkommen des Ausdrucks in Klammern
(..)    : textlich in TREEMOLA vorkommende Klammern
|       : Trennzeichen zwischen TREEMOLA Knoten (!)
```

Die folgenden Regeln beschreiben die Syntax der Kombinationsmöglichkeiten von Knoten:

```
axiom      ::= ( hardware [| program] )
program    ::= proghead | seqblock | progend
proghead   ::= progbegin [ ( filenode {|filenode} ) ]
seqblock   ::= dcl | seqbegin | seqbody | seqend | dclend
seqblock   ::=       seqbegin | seqbody | seqend
seqbody    ::= parblock
seqbody    ::= seqblock
seqbody    ::= seqnode  ( if-block )
seqbody    ::= seqbody  [ | parblock ]
seqbody    ::= seqbody  [ | seqblock ]
seqbody    ::= seqbody  [ | seqnode ( if-block ) ]
dcl        ::= declbegin
dcl        ::= dcl   | constdcl
dcl        ::= dcl   | typedcl
dcl        ::= dcl   | vardcl
dcl        ::= dcl   | procdcl
dcl        ::= dcl   | macrodcl
parblock   ::= parnode (statement {|statement})
statement  ::= ifstatement
statement  ::= assignment
statement  ::= stopnode
statement  ::= callpascal
```

```
assignment ::= destnode (data [|address] [|oprnode])
assignment ::= destnode (data [|address] [|control])
data       ::= expression
address    ::= expression
control    ::= expression
expression ::= catnode (expression' [{|expression'}])
expression ::= expression'
expression'::= constant
expression'::= sourcenode(expression[{|expression}]|oprnode)
expression'::= oprnode   (expression [{|expression}])    (*)
constant   ::= integnode
constant   ::= strngnode
constant   ::= labelnode
constant   ::= instrnode
ifstatement::= ifnode   (expression | thenpart | elsepart)
thenpart   ::= thennode (statement [{|statement}])
elsepart   ::= elsenode (statement [{|statement}])
if-block   ::= ifnode   (expression | seqblock | seqblock)
callpascal ::= callnode (cid | filenode [{|expression}] )
cid        ::= dummynode(idnode)
```

Der Aufbau der Symbole hardware, constdcl, typedcl, vardcl, macrodcl und procdcl ist für diese Arbeit ohne Belang und wird daher hier nicht näher dargestellt. Die mit (*) gekennzeichnete Regel wird nur lokal innerhalb des Synthesesystems verwendet.

Die folgenden Regeln drücken den Aufbau der Knoten aus. Fett gedruckte Identifikatoren sind Namen für durch jeweils ein Zeichen dargestellte Knotentypen. Groß geschriebene Zeichenketten sind terminale Symbole. integer, string, identifier und op-symbol sind wie in MIMOLA definiert. prop dient der Aufnahme von Zusatzinformation wie z.B. den im MIMOLA-Programm enthaltenen Properties, Kommentaren und Optionen.

```
progbegin ::= progbegintype  progid                    [prop]
progend   ::= endtype        PROGRAM                   [prop]
seqbegin  ::= begintype      BEGIN                     [prop]
seqend    ::= endtype        END                       [prop]
```

```
declbegin ::= begintype      DECLARE                          [prop]
declend   ::= endtype        DECLARE                          [prop]
parnode   ::= partype        label                            [prop]
seqnode   ::= seqtype        label                            [prop]
destnode  ::= assignmenttype module   var_id range_id         [prop]
sourcenode::= sourcetype     module   var_id range_id         [prop]
catnode   ::= catenationtype          var_id range_id         [prop]
oprnode   ::= operationtype op_symbol var_id range_id         [prop]
integnode ::= integertype    integer  var_id range_id         [prop]
strngnode ::= stringtype     string   var_id range_id         [prop]
labelnode ::= labeltype      label    var_id range_id         [prop]
instrnode ::= instrtype      value    var_id range_id         [prop]
filenode  ::= filetype       fileid                           [prop]
idnode    ::= cidtype        identifier                       [prop]
thennode  ::= thentype                                        [prop]
elsenode  ::= elsetype                                        [prop]
ifnode    ::= ifstatementtype                                 [prop]
dummynode ::= dummytype                                       [prop]
callnode  ::= hlltype        CALL[ _ identifier]              [prop]
stopnode  ::= hlltype        STOP                             [prop]
module    ::= typeid[ _ partid[-portid]]
typeid    ::= identifier
partid    ::= identifier
portid    ::= identifier
prop      ::= {in '$' eingeschl. Folge v. Zeichen außer '$'}
label     ::= identifier
var_id    ::= . [^identifier]
fileid    ::= identifier
value     ::= integer
value     ::= string
value     ::= label
progid    ::= identifier
range_id  ::= "["integer : integer"]"
```

Für **nodetype**(n)=instrtype ist **part_id**(n)=I.

In der letzten Regel bedeuten "[" und "]" in TREEMOLA vorkommende eckige Klammern.

Anhang B: Synthetisierte Rechnerstruktur

Im folgenden sind Resultate aufgeführt, die sich bei Benutzung des Beispiels aus Abschnitt 4.2 als Eingabe für das Synthese- System ergeben. Als Ressource-Beschränkungen sind dabei maximal zwei ImmediateFelder und 12$ als Kosten für ALUS erlaubt. (Die Kostengrenze für ALUS geht als weiteres Kriterium in die Berechnung von **treetoobig** ein.)

Als erstes sei das resultierende RT-Programm nach Abschluß der RegisterAllokation betrachtet. Sprünge an die Marke Line0091 sind in einer konkreten Realisierung durch Maßnahmen zur Beendigung des Programms zu ersetzen.

```
Line0086      :
  SH(1.^m) := 1,
  PC := "INCR" PC;
Line0086_1    :
  SH(0.^i) := 1,
  PC := "INCR" PC;
Line0087      :
  CC0 := SH(0.^i)"<="7;
  PC := "INCR" PC;
Line0087_1    :
  PC := IF CC0 THEN "INCR" (PC) ELSE Line0091 FI;
Line0087_t0   :
  SR(0) := SH(0.^i),
  PC := "INCR" PC;
Line0087_t0_1 :
  SH(0.^i) := SR(0)"+"1,
  PC := "INCR" PC;
Line0087_t0_2 :
  SR(0) := SH(1.^m)"*"SR(0),
  PC := "INCR" PC;
Line0087_t0_3 :
  PC := Line0087,
  SH(1.^m) := SR(0);
```

Als nächstes sei die resultierende Rechnerstruktur wiedergegeben:

```
TARGET rechner;
STRUCTURE
 PARTS
  MbP0,
  MaP0,
  MDSRP1:
   MODULE Nmux02x16                                    (*Multiplexer*)
    (OUT f:(15:0);IN inp00,inp01:(15:0);FCT sel:(0));
    BEGIN
     f :=
      CASE sel OF
       0          : inp00 ;
       1          : inp01
      END
    END;
```

```
MDSH,
MDPC:
 MODULE Nmux03x16
  (OUT f:(15:0);IN inp00,inp01,inp02:(15:0);FCT sel:(1:0));
  BEGIN
   f :=
    CASE sel OF
     0          : inp00 ;
     1          : inp01 ;
     2          : inp02
    END
  END;
CC0:
 MODULE RCC
  (OUT f:(0);IN e:(0);FCT s:(0));
  BEGIN
   CASE s OF
    %1         : RCC := e ;
    %0         :
   END,
   f := RCC
  END;
U1:
 MODULE NU1
  (OUT f:(15:0);IN inp00:(0);IN inp01,inp02:(15:0));
  BEGIN
   f := IF inp00 THEN inp02 ELSE inp01 FI
  END;
U0:
 MODULE AU0
  (OUT f:(15:0);IN inp00:(15:0));
  BEGIN
   f := "INCR" inp00
  END;
P0:
 MODULE B74xy
  (OUT f:(16:0);IN a,b:(15:0);FCT sel:(1:0));
  BEGIN
   f.(15:0) :=
    CASE sel OF
     %00        : a "-" b;
     %01        : a "+" b;
     %10        : a "*" b;
     %11        : a "/" b
    END,
   f.(16) :=
    CASE sel OF
     %00        : a "<=" b;
     %01        : a "<" b;
     %10        : a "=" b;
     %11        : a "<>" b
    END
  END;
```

```
SR:
 MODULE S8
  PORT P1
   (IN e:(15:0);ADR ad:(2:0);FCT c:(0));
   BEGIN
    CASE c OF
     %0         : S8(ad) := e;
     %1         :
    END
   END;
  PORT P2,P3
   (OUT f:(15:0);ADR ad:(2:0);FCT c:(0));
   BEGIN
    f :=
     CASE c OF
      %0         : S8(ad);
      %1         : TRISTATE
     END
   END;
SH:
 MODULE S4k
  (INOUT ea:(15:0);ADR address:(15:0);FCT sel:(0));
  BEGIN
   CASE sel OF
    %0         : S4k(address) := ea;
    %1         :
   END,
   ea :=
    CASE sel OF
     %1         : S4k(address);
     %0         : TRISTATE
    END
  END;
PC:
 MODULE RP
  (OUT a:(15:0);IN e:(15:0));
  BEGIN
   RP := e,
   a := RP
  END;
INSTRUCTION
 I          :(MbP0       (48),
              MaP0       (47),
              MDSRP1     (46),
              MDSH       (45:44),
              MDPC       (43:42),
              FSRP3      (41),
              FCC0       (40),
              FSRP1      (39),
              FSRP2      (38),
              FSH        (37),
              FP0        (36:35),
              D3         (34:32),
              D2         (31:16),
              D1         (15:0));
```

```
CONNECTIONS
 MbP0          .(15:0)          -> P0          .b.(15:0);
 I             .MbP0            -> MbP0        .sel.(0);
 SR-P2         .(15:0)          -> MbP0        .inp01.(15:0);
 I             .D2              -> MbP0        .inp00.(15:0);
 MaP0          .(15:0)          -> P0          .a.(15:0);
 I             .MaP0            -> MaP0        .sel.(0);
 SH            .(15:0)          -> MaP0        .inp01.(15:0);
 I             .D2              -> MaP0        .inp00.(15:0);
 I             .FP0             -> P0          .sel.(1:0);
 I             .FSRP3           -> SR-P3       .c.(0);
 I             .D3              -> SR-P2       .ad.(2:0);
 I             .FSRP2           -> SR-P2       .c.(0);
 I             .D3              -> SR-P1       .ad.(2:0);
 MDSRP1        .(15:0)          -> SR-P1       .e.(15:0);
 I             .MDSRP1          -> MDSRP1      .sel.(0);
 P0            .(15:0)          -> MDSRP1      .inp01.(15:0);
 SH            .(15:0)          -> MDSRP1      .inp00.(15:0);
 I             .FSRP1           -> SR-P1       .c.(0);
 I             .D1              -> SH          .address.(15:0);
 MDSH          .(15:0)          -> SH          .ea.(15:0);
 I             .MDSH            -> MDSH        .sel.(1:0);
 P0            .(15:0)          -> MDSH        .inp02.(15:0);
 SR-P2         .(15:0)          -> MDSH        .inp01.(15:0);
 I             .D2              -> MDSH        .inp00.(15:0);
 I             .FSH             -> SH          .sel.(0);
 MDPC          .(15:0)          -> PC          .e.(15:0);
 I             .MDPC            -> MDPC        .sel.(1:0);
 U0            .(15:0)          -> MDPC        .inp02.(15:0);
 I             .D2              -> MDPC        .inp01.(15:0);
 U1            .(15:0)          -> MDPC        .inp00.(15:0);
 PC            .(15:0)          -> U0          .inp00.(15:0);
 U0            .(15:0)          -> U1          .inp02.(15:0);
 I             .D1              -> U1          .inp01.(15:0);
 CC0           .(0)             -> U1          .inp00.(0);
 P0            .(16)            -> CC0         .e.(0);
 I             .FCC0            -> CC0         .s.(0);
 LOCATIONS_FOR_TEMPOR   CC0, SR(5:0);
 LOCATIONS_FOR_VARIAB   SH(4095:0);
END.
```

Die Steuerung des Rechners übernimmt der folgende binäre Code:

Adresse des Befehlsspeichers	Inhalt Bit: 48 44 40 36 32 28 24 20 16 12 8 4 0
#0000	XXX00101011OXXXXX0000000000000001000000000000001
#0001	XXX00101011OXXXXX0000000000000001000000000000000
#0002	01XXX10111110OXXX0000000000000111000000000000000
#0003	XXXXX0010111XXXXXXXXXXXXXXXXXXX0000000000001000
#0004	XXOXX1010011XX000XXXXXXXXXXXXXXX0000000000000000
#0005	10X10101010001000000000000000000010000000000000000
#0006	111XX10100011000OXXXXXXXXXXXXXXX0000000000000001
#0007	XXX01011010OXX0000000000000000010000000000000001

Anhang C: Eingabebeispiel für die Codeerzeugung

```
TARGET  simplecpu;
 STRUCTURE
  TYPE   word = (15:0);
  CONST  NULL = 0.(7:0);
  PARTS
   SH : MODULE S64k
        PORT P2 (in data:word; adr addr:word;fct cntrl:(0));
          BEGIN
           CASE cntrl OF
            0 : S64k(addr):=data;
            1 : ;
           END
          END;
        PORT P1 (out data: word; adr addr : word);
         BEGIN
         data:= S64k(addr)
        END;

   ALU : MODULE Balu (in a,b:word;fct s:(1:0); out f:word);
         BEGIN
          f:= CASE s OF
               0 : a ;
               1 : b ;
               2 : a "+" b;
               3 : a "-" b;
              END
         END;

   PC : MODULE RP (in e : word; out a : word);
        BEGIN
         RP:=e, a:=RP
        END;

   ACCU : MODULE REG (in e:word; fct s:(0); out a:word);
          BEGIN
           CASE s OF
            0 : REG:=e;
            1 : ;
           END,
           a:=REG
          END;

   DEC : MODULE ADEC (fct s:(1:0); out a : word);
         BEGIN
          a:= CASE s OF
               0 : 0;
               1 : 2;
               2 : 4;
               3 : 8
              END;
         END;
```

```
   INCR : MODULE Ai  (in e : word; out a : word);
          BEGIN
           a:= "INCR" e;
          END;

   BMUX : MODULE NMUX (in a,b,c,d : word;
                       fct s : (1:0); out f : word);
          BEGIN
           f:= CASE s OF
                0 : a;
                1 : b;
                2 : c;
                3 : d
               END;
           END;

  INSTRUCTION
   I : (imm2  (47:32),
        imm1  (31:16),
        imm0  (15: 8),
        cdec  ( 7: 6),
        cshp2 (    5),
        calu  ( 4: 3),
        caccu (    2),
        cbmux ( 1: 0));
  CONNECTIONS
   I.imm2          ->   SH-P1.addr;
   ALU             ->   SH-P2.data;
   I.imm1          ->   SH-P2.addr;
   I.cshp2         ->   SH-P2.cntrl;
   SH-P1           ->   ALU.a;
   BMUX            ->   ALU.b;
   I.calu          ->   ALU.s;
   ALU             ->   ACCU.e;
   I.caccu         ->   ACCU.s;
   I.imm1          ->   BMUX.a;
   NULL            ->   BMUX.b.(15:8);
   I.imm0          ->   BMUX.b.(7:0);
   DEC             ->   BMUX.c;
   ACCU            ->   BMUX.d;
   I.cbmux         ->   BMUX.s;
   I.cdec          ->   DEC.s;
   INCR            ->   PC.e;
   PC              ->   INCR.e;
 LOCATIONS_FOR_TEMPORARIES
                  SH(5:0), ACCU;
END_target;

PROGRAM beispiel;
 BEGIN
  SH(0):=SH(1) "+" 8
 END.
```

Anhang D : Komponenten des MIMOLA-Software-Systems