# Konzeption, Entwurf und Realisierung eines Prozessors für die funktionale Programmiersprache Scheme als VLSI-Chip in Standard- und Makrozellentechnik

Dissertation
zur Erlangung des Doktorgrades
am Fachbereich Informatik der Universität Hamburg

vorgelegt von
Jörg Lohse
aus Hamburg

Hamburg, 1990

**Genehmigt vom Fachbereich Informatik der Universität Hamburg**

**auf Antrag von Prof. Dr.-Ing. Klaus Lagemann (Referent)**

**und Prof. Dr. Günther Görz (Korreferent)**

**und Prof. Dr. Peter Marwedel (Korreferent)**

**Hamburg, den 29. 10. 1990**
**(Datum der Disputation)**

**Prof. Dr. Christopher Habel**
**Der Sprecher des Fachbereichs**

# Inhaltsverzeichnis

**Zusammenfassung** x

**Summary** xi

**Danksagungen** xii

**Kurzfassung** xiii

**1 Einleitung** 1

**2 Die Programmiersprache Scheme** 3
2.1 Aufbau von Scheme . . . 3
2.2 Vergleich von Scheme mit Pascal . . . 4
2.2.1 Daten . . . 4
2.2.2 Prozeduren . . . 5
2.2.3 Gültigkeitsbereiche von Variablen (Scoping) . . . 5
2.2.3.1 Die dynamische Variablenbindung . . . 6
2.2.3.2 Die statische Variablenbindung . . . 6
2.2.3.3 Closures . . . 7
2.2.4 Kontrollkonstrukte . . . 8

**3 Die Implementierung von Scheme** 11
3.1 Repräsention der Objekte . . . 11
3.1.1 Repräsentation von Daten . . . 14
3.1.1.1 Symbole . . . 14
3.1.1.2 Zahlen . . . 14
3.1.1.3 Zeichen (Chars) . . . 15
3.1.1.4 Zeichenketten (Strings) . . . 15
3.1.1.5 Die leere Liste . . . 16
3.1.1.6 Paare und Listen . . . 16
3.1.1.7 Vektoren . . . 17
3.1.2 Interne Objekte . . . 17
3.1.2.1 Closures . . . 17
3.1.2.2 Bindungsumgebungen (Environments) . . . 18
3.1.2.3 Controlframes . . . 20
3.1.2.4 Interne Darstellung von Programmen . . . 23
3.2 Die Speicherverwaltung . . . 24
3.2.1 Referenzenzählung . . . 25

3.2.2 Mark-and-Sweep Algorithmus . . . 25
3.2.3 Stop-and-Copy Algorithmus . . . 26
3.2.4 Inkrementelle Garbage Collection . . . 26

**4 Wichtige Implementationen von Scheme 28**
4.1 Interpretation der Quelldarstellung . . . 28
4.2 Interpretation von S-Code . . . 29
4.2.1 MIT C-Scheme . . . 29
4.2.2 Scheme-79 . . . 29
4.2.3 Scheme-81 . . . 31
4.3 Übersetzung in virtuellen Maschinencode . . . 32
4.3.1 PC Scheme . . . 33
4.3.2 MacScheme . . . 33
4.4 Übersetzung in realen Maschinencode . . . 34

**5 Konzeption einer neuen Implementation 35**
5.1 Folgerungen aus den bestehenden Implementationen . . . 35
5.2 Hardware-unterstützte Implementationen von LISP . . . 35
5.3 Eine neue Scheme-Implementation . . . 36
5.3.1 Verwandte Arbeiten . . . 40
5.4 Repräsentation der Objekte . . . 40
5.4.1 Das Speicherwort . . . 41
5.4.2 Die Typen der Objekte . . . 42
5.4.3 Darstellung von Bindungsumgebungen . . . 43
5.4.4 Darstellung von Controlframes und Continuations . . . 43
5.4.5 Darstellung von Codevektoren . . . 44
5.5 Diskussion der Randbedingungen für den Befehlssatz . . . 44

**6 Entwurf integrierter Schaltungen 46**
6.1 Integrierte Schaltungen . . . 46
6.2 Entwurfsverfahren für integrierte Schaltungen . . . 47
6.2.1 Der Full-Custom Entwurf . . . 47
6.2.2 Der Semi-Custom Entwurf . . . 48
6.2.2.1 Die Gate-Array-Technik . . . 48
6.2.2.2 Die Standardzell-Technik . . . 49
6.2.2.3 Die Makrozell-Technik . . . 50
6.3 Der Ablauf eines Semi-Custom Entwurfs . . . 51
6.3.1 Schaltplaneingabe . . . 51
6.3.2 Netzlistenkompilation . . . 53
6.3.3 Logiksimulation . . . 53
6.3.4 Testmustergenerierung oder Fehlersimulation . . . 53
6.3.5 Chipkonstruktion . . . 54
6.3.6 Resimulation mit Leitungslaufzeiten . . . 54
6.4 Fertigungsmöglichkeiten für Hochschulen . . . 55
6.5 Folgerungen für den Entwurf des Scheme-Prozessors . . . 56

**7 Festlegung von Architektur und Befehlssatz 57**
7.1 Register- oder Stackarchitektur . . . 57
7.1.1 Register mit VENUS-S . . . 57
7.1.2 Wahl einer Stackarchitektur . . . 58
7.2 Definition und Implementation eines Befehlssatzes . . . 58
7.2.1 Die Prozessorregister . . . 59
7.2.1.1 Der Stack . . . 59
7.2.1.2 Die Environmentzeiger . . . 59
7.2.1.3 Der Controlframezeiger . . . 59
7.2.1.4 Der Funktionszeiger . . . 59
7.2.1.5 Der Programmzähler . . . 59
7.2.1.6 Die Register der Speicherverwaltung . . . 60
7.2.1.7 Die Register für Speicherzugriffe . . . 60
7.2.1.8 Der Instructioncache . . . 60
7.2.1.9 Weitere Register . . . 61
7.2.1.10 Verwendung der Register . . . 61
7.2.2 Wahl der Befehle . . . 62
7.3 Literatur über Befehlssätze für LISP-Dialekte . . . 65
7.3.1 Der Befehlsatz von Schooler und Stamos . . . 65
7.4 Der Befehlssatz des Scheme-Prozessors . . . 66
7.4.1 Zugriff auf Variable und Konstante . . . 66
7.4.2 Sprungbefehle und Verzweigungen . . . 66
7.4.3 Aufruf und Rückkehr von Prozeduren . . . 66
7.4.4 Typprüfungen und Vergleiche . . . 69
7.4.5 Befehle für Listen- und Vektoroperationen . . . 69
7.4.6 Numerische Befehle . . . 69
7.4.7 Zeichen- und zeichenkettenorientierte Befehle . . . 70
7.5 Repräsentation der Befehle . . . 70
7.6 Beispiele für Programme in Bytecode . . . 70

**8 Entwurf und Realisierung des Prozessors 73**
8.1 Entwurfsebenen . . . 73
8.2 Verhaltensbeschreibung des Scheme-Prozessors . . . 76
8.2.1 Analyse der Verhaltensbeschreibung . . . 78
8.3 Strukturbeschreibung des Scheme-Prozessors auf RT-Ebene . . . 79
8.3.1 Das Operationswerk . . . 80
8.3.1.1 Die Registerbank . . . 81
8.3.1.1.1 Auswahl einzelner Felder der Register . . . 81
8.3.1.1.2 Signale der Registerbank . . . 81
8.3.1.1.3 Ausgabe von Konstanten . . . 83
8.3.1.1.4 Einführung von Pseudoregistern . . . 83
8.3.1.2 Die ALU . . . 84
8.3.1.3 Der Buszugriffsmodul . . . 84
8.3.1.4 Das gesamte Operationswerk . . . 86
8.3.1.4.1 Verbindungen zum Steuerwerk . . . 86
8.3.1.4.2 Testbarkeit des Operationswerks . . . 89
8.3.2 Das Steuerwerk . . . 89
8.3.2.1 Der Befehlsdekoder . . . 91
8.3.2.2 Die Folgeadreßsteuerung . . . 91

8.3.2.3 Gewährleistung der Testbarkeit . . . 92
8.3.2.4 Zeitbedingungen zwischen Operationswerk und Steuerwerk . . . 93
8.4 Erstellung des Mikroprogramms . . . 94
8.4.1 Veröffentlichungen über automatische Mikroprogrammgenerierung . . . 94
8.4.2 Mikroprogrammgenerierung für den Scheme-Prozessor . . . 95
8.5 Validierung von RT-Struktur und Mikroprogramm . . . 98
8.6 Realisierung des Scheme-Prozessors mit VENUS-S . . . 102
8.6.1 Realisierung des Buszugriffsmoduls . . . 102
8.6.2 Realisierung der ALU . . . 102
8.6.3 Realisierung der Registerbank . . . 102
8.6.3.1 Funktionseinheiten der Registerbank . . . 105
8.6.3.2 Realisierung der Spezialregister . . . 105
8.6.4 Realisierung des Operationswerks . . . 107
8.6.4.1 Validierung des Operationswerks . . . 109
8.6.5 Realisierung des Steuerwerks . . . 109
8.6.5.1 Validierung des Steuerwerks . . . 111
8.6.6 Realisierung des Prozessors . . . 111
8.6.6.1 Validierung des Prozessors . . . 111
8.6.6.2 Gewährleistung der Testbarkeit des Prozessors . . . 114

**9 Die Ergebnisse 115**
9.1 Das Chip . . . 115
9.2 Die Implementation . . . 118
9.3 Vergleich mit Scheme-81 . . . 119
9.4 Neuere Veröffentlichungen über Prozessorentwürfe für Scheme und LISP . . . 120
9.4.1 Scheme-86 . . . 120
9.4.2 Texas Instruments' LISP-Chip . . . 121
9.5 Bewertung der Ergebnisse . . . 123

**10 Schlussbetrachtungen 124**
10.1 Ausblick . . . 124
10.1.1 Vervollständigung der Implementation . . . 124
10.1.2 Verbesserungsmöglichkeiten des Prozessors . . . 125
10.1.3 Entwurfsmöglichkeiten bei anderen Randbedingungen . . . 126
10.1.3.1 Einfluß eines anderen Entwurfssystems . . . 126
10.1.3.2 Einfluß eines anderen Zielrechners . . . 126
10.2 Anwendung auf neuartige Entwurfssoftware . . . 127

**A Der Befehlssatz des Scheme-Prozessors 129**
A.1 Zugriffe auf Variablen . . . 129
A.2 Zugriffe auf Konstanten . . . 130
A.3 Sprung- und Verzweigungsbefehle . . . 130
A.4 Aufruf und Rückkehr von Prozeduren . . . 130
A.5 Typprüfungen und Vergleiche . . . 130
A.6 Listen- und Vektorbefehle . . . 131
A.7 Numerische Befehle . . . 131
A.8 Zeichen- und zeichenkettenorientierte Befehle . . . 131
A.9 Sonstige Befehle . . . 132

**B Gehäuse und Anschlussbelegung des Scheme-Prozessors** **133**

**C Externe Beschaltung des Scheme-Prozessors** **135**
C.1 Der Taktgenerator . . . . . 135
C.2 Die Chipselectlogik . . . . . 137

**D Test der gefertigten Prozessoren** **138**

**Literaturverzeichnis** **140**

**Eidesstattliche Erklärung** **148**

**Lebenslauf** **149**

# Abbildungsverzeichnis


2.1 Konstanten verschiedener Objekttypen in Scheme. . . . . 5
2.2 Übergabe der Identitätsfunktion in Pascal und Scheme. . . . . 6
2.3 Beispiel für dynamische Variablenbindung. . . . . 6
2.4 Verwendungsmöglichkeit für eine Closure. . . . . 7
2.5 Veranschaulichung der Closures. . . . . 8
2.6 Beispiel für `call-with-current-continuation`. . . . . 9
2.7 Beispiel für die unbegrenzte Lebensdauer einer Continuation. . . . . 10

3.1 Beispielprogramm für die Objektrepräsentation. . . . . 11
3.2 Veranschaulichung der Zeiger auf Objekte. . . . . 12
3.3 Das Paar `(6 . ())` bei Typrepräsentation im Zeiger. . . . . 13
3.4 Das Paar `(6 . ())` bei Typrepräsentation im Objekt. . . . . 13
3.5 Über Zeiger zu referenzierende Fließkommazahl (boxed `FLONUM`). . . . . 15
3.6 Repräsentation eines Strings als kontinuierlicher Speicherbereich. . . . . 15
3.7 Aufbau einer CONS-Zelle. . . . . 16
3.8 Repräsentation der Liste `(1 2 3)` mit CONS-Zellen. . . . . 16
3.9 Repräsentation der Liste `(1 2 3)` in CDR-Kodierung. . . . . 17
3.10 Repräsentation des Vektors `#(1 2 3)`. . . . . 17
3.11 Beispielprogramm für einen Prozeduraufruf. . . . . 18
3.12 Die Bindungsumgebung von `f` nach dem Funktionsaufruf. . . . . 18
3.13 Repräsentation der Bindungsumgebung von `f` durch Vektoren. . . . . 19
3.14 Repräsentation der Bindungsumgebung von `f` durch Listen. . . . . 20
3.15 Beispiel für die unbegrenzte Lebensdauer einer Continuation. . . . . 21
3.16 Implementierung einer Continuation. . . . . 22

4.1 Das Wordformat von Scheme-79. . . . . 29
4.2 Mit Scheme-79 gerechnetes Testprogramm. . . . . 30
4.3 Das Wordformat von Scheme-81. . . . . 31

5.1 Der Scheme-Prozessor in einem 68000-Rechner. . . . . 38
5.2 Der Scheme-Prozessor am 68000-Bus. . . . . 39
5.3 Struktur der Speicherworte des Scheme-Prozessors. . . . . 41
5.4 Bindungsumgebung nach Prozeduraufruf `(f 'A 2 "abc")`. . . . . 43
5.5 Aufbau eines Controlframes. . . . . 43
5.6 Aufbau eines Codevektors. . . . . 44
5.7 Codevektor der Prozedur `list`. . . . . 44

6.1 Schematischer Aufbau eines Gate-Arrays. . . . . 48
6.2 Schematischer Aufbau eines Standardzell-ICs. . . . . 49

6.3 Ausschnitt aus einer Standardzellzeile. . . . . 50
6.4 Schematischer Aufbau eines Makrozell-ICs. . . . . 51
6.5 Schematischer Ablauf eines Entwurfs mit VENUS. . . . . 52

7.1 Die Register nach Aufruf einer Funktion mit drei Parametern. . . . . 61
7.2 Format der Verzweigungsbefehle. . . . . 66
7.3 Der Stack nach Anlegen eines Controlframes. . . . . 67
7.4 Der Stack nach Evaluieren von Parametern und Prozedur g. . . . . 68
7.5 Der Stack nach Aufruf der Prozedur. . . . . 69
7.6 Die Befehlsformate des Scheme-Prozessors. . . . . 70

8.1 Sichten und Abstraktionsebenen bei Spezifikation und Entwurf (Y-Diagramm). . . 74
8.2 Beitrag von VENUS-S zum Entwurfsprozeß als Y-Diagramm. . . . . 75
8.3 Die Entwurfsebenen des Prozessors. . . . . 76
8.4 Operationswerk und Steuerwerk des Scheme-Prozessors. . . . . 79
8.5 Schematische Struktur des Operationswerks. . . . . 80
8.6 Die Signale der Registerbank. . . . . 82
8.7 Struktur der ALU. . . . . 86
8.8 Buszugriffsmodul und Registerbank sowie die Verbindungen des Scheme-Prozessors zur Außenwelt. . . . . 87
8.9 Das Operationswerk des Scheme-Prozessors. . . . . 88
8.10 Das Steuerwerk des Scheme-Prozessors. . . . . 90
8.11 RT-Spezifikation des Befehlsdekoders in *Karl II*. . . . . 91
8.12 Generierung des Mikroprogramms. . . . . 95
8.13 Protokoll eines Mikroprogrammcompilerlaufes. . . . . 99
8.14 Ausschnitt aus dem erzeugten binären Mikroprogramm. . . . . 99
8.15 Einbindung des RT-Simulators in die Entwurfsprogramme. . . . . 101
8.16 Realisierung der Registerbank mit Multiplexern. . . . . 103
8.17 Realisierung der Registerbank mit Tri-State-Bussen. . . . . 103
8.18 Plazierung der Standardzellen in der Registerbank. . . . . 104
8.19 Die Struktur der Registerbank. . . . . 106
8.20 Plazierung des Operationswerks. . . . . 108
8.21 Simulation des Operationswerks auf Gatterebene. . . . . 110
8.22 Simulation des Steuerwerks auf Gatterebene. . . . . 112
8.23 Plazierung des Scheme-Prozessors. . . . . 113

9.1 Das Layout des Scheme-Prozessors. . . . . 116
9.2 Photo des Scheme-Prozessors im geöffneten Gehäuse. . . . . 117
9.3 Photo des Chips im Gehäuse. . . . . 117
9.4 Operationswerk des LISP-Prozessors von TI. . . . . 122

B.1 Das Gehäuse PGA-88 der Scheme-Prozessors. . . . . 133

C.1 Die Takte des Scheme-Prozessors. . . . . 136
C.2 Der Taktgenerator für den Scheme-Prozessor im Atari ST. . . . . 136
C.3 Die Chipselectlogik für den Scheme-Prozessor im Atari ST. . . . . 137

# Tabellenverzeichnis


2.1 Schlüsselworte in Scheme. . . . 3
2.2 Objekte in Scheme. . . . 4

3.1 CDR-Codes. . . . 16

4.1 Durch Scheme-79 unterstützte Standardprozeduren. . . . 30

5.1 Verteilung der CPU-Zeit in LISP-Programmen nach [StHe88]. . . . 37
5.2 Belegungen der CDR-Codes. . . . 42
5.3 Vom Scheme-Prozessor unterstützte Objekte. . . . 42

6.1 Integrationsgrade integrierter Schaltungen. . . . 46
6.2 Mögliche Chipgrößen für Hochschulentwürfe. . . . 55

7.1 Die Register des Scheme-Prozessors. . . . 60
7.2 Im Mikroprogramm realisierte Standardprozeduren. . . . 64

8.1 Tabellarische Form des Y-Diagramms. . . . 74
8.2 Tabellarische Form des Y-Diagramms beim Entwurf des Scheme-Prozessors. . . . 76
8.3 Bestandteile des Verhaltenssimulators. . . . 77
8.4 Grammatik in der Verhaltensbeschreibung vorkommender Ausdrücke. . . . 78
8.5 Operationen der Feldauswahl. . . . 82
8.6 Register und Pseudoregister in der Registerbank. . . . 85
8.7 Operationen von Addierer und Komplementer. . . . 85
8.8 Operationen des Shifters. . . . 85
8.9 Mikrobefehle für die Folgeadreßbestimmung. . . . 92
8.10 Mikroinstruktionsabarbeitung ohne Pipelining. . . . 93
8.11 Mikroinstruktionsabarbeitung mit Pipelining. . . . 93
8.12 Mikrobefehle einer Mikroinstruktion. . . . 94
8.13 Statistik des Operationswerks. . . . 109
8.14 Statistik des Steuerwerks (ohne ROM). . . . 111
8.15 Statistik des Scheme-Prozessors. . . . 113

9.1 Vergleich des Scheme-Prozessors mit anderen 68000-Implementationen. . . . 118
9.2 Vergleich des Scheme-Prozessors mit Scheme-81. . . . 119
9.3 Statistik des LISP-Chips von TI. . . . 122

B.1 Die Anschlußbelegung der Scheme-Prozessors. . . . 134

D.1 Ergebnisse des Tests der Prozessoren. . . . . 139

# Konzeption, Entwurf und Realisierung eines Prozessors für die funktionale Programmiersprache Scheme als VLSI-Chip in Standard- und Makrozellentechnik

**Zusammenfassung** Das Ziel dieser Arbeit war eine neue Hardwareimplementation der Programmiersprache Scheme. Ein als integrierter Schaltkreis realisierter Prozessor wird in vorhandene 68000-Mikrorechner eingesetzt und arbeitet einen Scheme-spezifischen Befehlssatz ab. Der Prozessor ermöglicht eine beschleunigte und speicherplatzeffiziente Implementation von Scheme.

Die bisherigen Pozessorentwürfe zielten darauf ab, einige Grundeinsichten zu gewinnen, und sie begnügten sich deshalb mit dem unumgänglichen Minimum an Objekttypen: Nur Zahlen und CONS-Zellen wurden repräsentiert. Dagegen stellt sich diese Arbeit zusätzlich den Kriterien praktischer Nutzbarkeit: dabei waren vor allem schwierige Anforderungen zur Objektrepräsentation — effizienter Zugriff, kompakte Repräsentation, große Vielfalt der Objekte — unter restriktiven Randbedingungen, wie z.B. begrenzte Chipfläche, zu erfüllen. Der Repräsentation der Programme, der Schnittstelle zwischen Hardware und Software, wird dabei eine besondere Bedeutung zugemessen. Ausgehend von dem Stand der Technik in Compilerbau einerseits und den Entwurfs- und Fertigungsmöglichkeiten andererseits wurde der Befehlssatz festgelegt. Die entwickelte Architektur umfaßt eine 24-Bit ALU, eine Tag-orientierte 3-Port Registerbank, einen parallel arbeitenden Buszugriffsmodul und ein mikroprogrammiertes Steuerwerk mit 2-stufiger Pipeline. Die Mikroprogramme wurden durch einen dafür erstellten Compiler automatisch aus der Verhaltensbeschreibung generiert.

Das Entwurfsvorgehen berücksichtigte durch Entwurfssoftware und Fertigungsmöglichkeiten bestehende Randbedingungen und erlaubte den Entwurf des Prozessors in kurzer Zeit. Insbesondere wurden Simulationsstimuli und erwartete Simulationsergebnisse automatisch generiert und die erzielten Simulationsergebnisse mit den erwarteten automatisiert verglichen. Die Struktur des Prozessors auf Gatterebene erwies sich als sehr komplex. Sie enthält mehr als die doppelte vom Hersteller angegebene maximal erreichbare Transistorzahl und bewirkte extrem lange Laufzeiten und das Überschreiten der Kapazität der Entwurfsprogramme.

Die als integrierte Schaltungen gefertigten Prototypen wurden in einen Atari ST eingebaut. Damit gerechnete Benchmarks weisen eine Geschwindigkeitssteigerung um einen Faktor von über zehn gegenüber anderen 68000-Implementationen aus, und selbst gegenüber in 68000-Maschinencode übersetzten Programmen erzielen diese Chips eine Verdopplung bis Versiebenfachung der Geschwindigkeit.

# The conceptual and structural design of a processor for the functional programming language Scheme in VLSI standard- and macro-cell technique.

**Summary** We are aiming to provide a new hardware implementation of the programming language Scheme [Rees86]. The hardware implementation is realized as an integrated circuit which can be added to available 68000-systems. The processor which runs Scheme-specific machine code is fast and makes efficient use of memory. Previous hardware implementations of Scheme [Suss81, Bata82] were based on the (theoretical) minimum of number and CONS-cell object types. In this project this overt theoretical view is rejected for a practical approach: In the practice of programming Scheme the language implementation has to provide efficient access to a large number of diverse object types. These requirements are our main design criteria. The representation of programs — interface between hardware and software — is considered especially important. The state of the art in compiler construction on the one hand and the design software and fabrication facility on the other hand lead to the instruction set.

The developed architecure consists of a 24-bit ALU, a tag-oriented 3-port register bank, an autonomous bus controler, and a micro-programed control unit with 2-stage pipeline. The neccessary micro-programs were generated automatically from the behavioural description.

The design process was restricted by the production facility's time table and the available design software. (For example the whole design had to be completed within a time-span of 11 months). Our approach was to write programs which generated simulation data and expected results. These expected simulation results were automatically compared with the results of the simulation.

The gate-level complexity of the processor is very high. It has twice the transistor count as provided by the available design software. The capacity of the design software was greatly overstretched and the great number of components led to extremely long program runs.

Prototypes of the chip were tested with an Atari ST. Benchmark results show a speed gain of a factor of ten compared to other 68000 Scheme software implementations. It is even two to seven times faster than compiled 68000 machine code.

**Danksagungen** Ich danke Prof. Dr.-Ing. K. Lagemann für die freundliche Unterstützung dieses Projekts. Ferner danke ich Prof. Dr. Günther Görz für das Interesse an diesen Arbeiten, für das Überlassen der Benchmarkprogramme und für die fruchtbaren Diskussionen.

An der Ausführung der Arbeiten waren Kollegen und Studenten beteiligt: Mein besonderer Dank gilt den Kollegen Dr. Reinhard Rauscher und Bernd Schütz für ihre Unterstützung bis hin zur Beteiligung an der Schaltplaneingabe und Simulation der Registerbank. Norman Hendrich danke ich für die zuverlässige Realisierung des Buszugriffsmoduls mit VENUS-S. Lars Hagge, Lars Larsson und Dirk Michelsen gebührt Dank für die Realisierung von großen Teilen des Steuerwerks. Andreas Großmann und Marcus Schorn haben fast alle der nicht als Prozessorbefehle vorhandenen Standardprozeduren implementiert. Ingo Marks hat einen bildschirmorientierten Editor erstellt, und Thomas Scharler hat die gefertigten Chips getestet.

Auch dem Rechenzentrum des Fachbereichs gebührt Dank für die Duldung der vielen Batchjobs mit unbegrenzter Laufzeit auf der VAX-8550. Nur dadurch war es möglich, den Befehlssatz und das Mikroprogramm so intensiv zu simulieren, daß sie unabänderlich in einem integrierten Schaltkreis untergebracht werden konnten.

# Konzeption, Entwurf und Realisierung eines Prozessors für die funktionale Programmiersprache Scheme als VLSI-Chip in Standard- und Makrozellentechnik (Kurzfassung)

Das Ziel dieser Arbeit war eine neue Hardwareimplementation der Programmiersprache Scheme [Rees86]. In dieser Arbeit werden die Programmiersprache Scheme [Rees86] und die Engpässe bei der Implementierung dieser Sprache beschrieben. Als Engpässe wurden Typenprüfungen zur Laufzeit, häufige und aufwendige Prozeduraufrufe, die Speicherverwaltung sowie die Notwendigkeit, lokale Variablen und Controlframes im allgemeinen Fall nicht in einem Stack allozieren zu können, identifiziert. Zur Entschärfung dieser Engpässe wurde eine Hardwareimplementation konzipiert, die speziell auf Scheme zugeschnitten ist. Für die Konzeption dieser Implementation wurden bestehende Hardware- und Software-Implementationen von Scheme analysiert und Voraussetzungen für eine effiziente Implementation erarbeitet. Diese Voraussetzungen bestehen in einer effizienten Repräsentation der Objekte, im Schonen der rechenzeitintensiven Speicherverwaltung durch die Allokation von lokalen Variablen und Controlframes — soweit das möglich ist — im Stack und in der Verlagerung der Typprüfungen und der Speicherverwaltung in Hardware. Als Ziel einer solchen neuartigen Hardwareimplementation wurde ein Scheme-Prozessor, der als integrierter Schaltkreis in einen vorhandenen 68000-Rechner [68000] eingesetzt werden soll, gewählt. Diese Wahl führte zur zusätzlichen Randbedingung, daß der Hauptspeicher eine knappe Resource ist, mit der sparsam umzugehen ist. Als Entwufssystem für den integrierten Schaltkreis stand die Software VENUS-S der Siemens AG zur Verfügung [Hörb86] mit Fertigungsmöglichkeiten in einem $2\,\mu$ CMOS-Prozeß [Stei88]. Die zeitlich beschränkte Nutzungsmöglichkeit von VENUS-S führte zu der Forderung, den Prototyp des Scheme-Prozessors in nur 11 Monaten bis zur Fertigungsreife zu entwerfen. Auch dieser Zeitplan hatte Einfluß auf das Entwurfsvorgehen.

Der Entwurf mit dem beschriebenen Ziel und unter den dargestellten Randbedingungen begann mit der Bestimmung der Repräsentation der Objekte, insbesondere mit der Repräsentation der Programme. In diesem Ansatz überwindet dieser Entwurf die festgefahrene Konzeption der beste-

henden, am M.I.T. entworfenen Scheme-Prozessoren [Suss81, Bata82]. Die bestehenden Entwürfe gingen von einer *theoretisch* ausreichenden, minimal nötigen Vielfalt von Objekten aus: Nur Zahlen und CONS-Zellen wurden implementiert. In dieser Arbeit wird bei der Objektrepräsentation *praktischen* Anforderungen — effizienter Zugriff, kompakte Repräsentation, große Vielfalt der Objekte — gefolgt. Die hier gewählte Objektrepräsentation wurde in einer frühen Phase, vor Erstellung der Verhaltensbeschreibung, vorgegeben und damit sichergestellt, daß sie bei Spezifikation weiterer Parameter und beim Hardwareentwurf nicht mehr zur Disposition gestellt werden konnte. Programme werden in dieser Implementation durch Bytecode repräsentiert. Bytecode ist sehr kompakt (Faktor 3 kompakter als der Code der bestehenden Scheme-Prozessoren [ScSt84] und Faktor 5 kompakter als 68000-Maschinencode [Sem87]). Die effiziente Übersetzung von Scheme in Bytecode ist bereits Stand der Technik [BaJe86, Sem87, ScSt84, Kran88]. Listen werden durch CDR-Kodierung [Hans69] repräsentiert, ein zuvor nur bei kommerziellen LISP-Maschinen eingesetztes Verfahren.

Am Anfang des Entwurfsprozesses stand der Entwurf eines Befehlssatzes und die Festlegung der benötigten Prozessorregister. Ein Befehlssatzinterpreter wurde erstellt und um einen Scheme-Compiler, die Speicherverwaltung und diverse Scheme-Standardprozeduren erweitert. Entstanden war damit eine Verhaltensbeschreibung des Prozessors in Form einer Software-Implementation von Scheme. Diese Verhaltensbeschreibung wurde genutzt, um den Befehlssatz zu erproben und zu optimieren und um noch vor der Fertigstellung des Prozessors Software für den Prozessor zu erstellen.

Nachdem das Verhalten des zu entwerfenden Prozessors in Form einer Verhaltensbeschreibung spezifiziert war, begann der Hardwareentwurf einer Struktur auf Registertransferebene. Es stand nur eine sehr kurze Zeit bis zum Abgabetermin des fertigen Chips zur Verfügung, und dieser restriktive Zeitplan erforderte die Entwicklung eines systematischen Vorgehens beim Entwurf: Um systematisch eine Prozessorarchitektur entwerfen zu können, wurde die als Pascalprogramm vorliegende Verhaltensbeschreibung analysiert. Die erhaltenen Analyseergebnisse umfassen häufige auftretende Befehlsfolgen, die Verteilung der Register und Registerfelder in Ausdrücken und Zuweisungen, die verwendete Konstrollstrukturen und eine Grammatik der enthaltenen Ausdrücke. Diese Analyseergebnisse beschreiben Anforderungen an die Hardwarearchitektur des entworfenen Prozessors. Zusätzliche die Vielzahl der möglichen Architekturen einschränkende Randbedingungen kommen aus dem IC-Entwurfssystem mit Standard- und Makrozellentechnik und den Fertigungsmöglichkeiten hinzu. Die Anforderungen und Randbedingungen bestimmen die entworfene Architektur: Die Häufigkeit der Ausdrücke mit zwei Operanden erforderte eine Registerbank mit mindestens zwei Ausgangsports, das Entwurfssystem erlaubt nur die Realisierung einer solchen Registerbank aus Standardzellen, eine platzaufwendige Lösung, die nur eine geringe Anzahl von Registern zuläßt und damit zur Entscheidung für einen stackorientierten Befehlssatz führte. Die bei der Analyse der Verhaltensbeschreibung ermittelten häufigen Schreib- und Lesezugriffe auf einzelne Felder der Register führte zur Entscheidung für eine Registerbank mit drei Ports, die neben den Portadressen auch die einzelnen Felder direkt adressieren kann. Die Spezifikation der ALU wurde direkt aus der Grammatik der in der Analyse ermittelten Ausdrücke entwickelt. Die ALU kann alle diese Ausdrücke in 20 ns berechnen. Ein Hauptspeicherzugriff des Prozessors dauert, durch die Verwendung des vorhandenen Hauptspeichers im PC bedingt, bis zu zehn Zyklen. Hauptspeicherzugriffe waren deshalb zu reduzieren. Ferner entstand ein Buszugriffsmodul, der parallel zur Abarbeitung des Mikroprogramms im Scheme-Prozessor diese Zugriffe abwickelt. Das Mikroprogramm verzögert daher die in anderen LISP-Prozessoren parallel ausführbaren Typenprüfungen bis zum nächsten Hauptspeicherzugriff und kann sie daher fast immer ohne zusätzlichen Zeitbedarf ausführen.

Die Struktur des Steuerwerks wurde so entworfen, daß sie sowohl den Anforderungen aus der Analyse der Verhaltensbeschreibung dient als auch den Randbedingungen des Entwurfssystems

genügt: Ein mikroprogrammiertes Steuerwerk mit einem Mikroprogramm-ROM mit 39 Bit Breite und 1024 Worten, der mit dem Entwurfssystem maximal möglichen ROM-Größe, ermöglicht eine Vielzahl von aus den Kontrollstrukturen der Verhaltensbeschreibung entwickelten Folgeadressierungen. Das Steuerwerk beinhaltet eine zweistufige Pipeline und gewährleistet auch die bei IC-Entwürfen sehr wichtige Testbarkeit des Prozessors. Das Mikroprogramm wurde automatisch durch einen für diesen Zweck erstellten optimierenden Compiler aus der Verhaltensbeschreibung generiert. Ein neuartiger RT-Simulator simulierte RT-Struktur und Mikroprogramm und verglich seine Ergebnisse parallel mit einer Simulation der Verhaltensbeschreibung.

Nach simulativer Validierung der Verhaltensbeschreibung erfolgte die Realisierung der RT-Struktur mit dem auf Gatterebene aufsetzenden Chip-Entwurfssystem VENUS-S. Die Struktur wurde manuell auf Gatterebene transformiert und in VENUS-S eingegeben und simuliert. Die automatische Generierung von Simulationsstimuli und einer Spezifikation des erwarteten Verhaltens durch den RT-Simulator wurde entwickelt, um die Struktur auf Gatterebene systematisch validieren zu können. Die Simulationsergebnisse wurden durch (neu erstellte und vorhandene) Programme mit der Spezifikation des erwarteten Verhaltens verglichen und erlaubten so eine effiziente Validierung der Struktur auf Gatterebene.

Die Struktur des Prozessors aus Gatterebene erwies sich als sehr komplex. Sie enthält mehr als die doppelte vom Hersteller für die zur Verfügung stehende Chipfläche angegebene maximal erreichbare Transistorzahl und bewirkte extrem lange Laufzeiten und das Überschreiten der Kapazität der Entwurfsprogramme. Um diese Schaltung auf der Chipfläche unterbringen zu können, waren detaillierte Überlegungen über eine optimale Plazierung der Zellen auf dem Chip und das manuelle Plazieren von über 90 % aller Standardzellen nötig.

Der Prozessor wurde bei der Siemens AG gefertigt. Die gefertigten Prototypen wurden auf Herstellungfehler getestet und die korrekt gefertigten Exemplare in einen Atari ST eingebaut. Gerechnete Benchmarks weisen eine Geschwindigkeitssteigerung mit einem Faktor von über zehn gegenüber anderen 68000-Implementationen aus, und selbst gegenüber in 68000-Maschinencode übersetzten Programmen bewirken die Chips eine Verdopplung bis Versiebenfachung der Geschwindigkeit. Damit ist das Ziel einer effizienten, speicherplatzsparenden Implementation für 68000-Rechner erreicht. Ein Vergleich des Prozessors mit den Entwürfen des MITs zeigt die Richtigkeit der hier zugrunde gelegten Implementationsentscheidungen: Von praktischen Anforderungen ausgehend konnte ein Prozessor entworfen werden, der die gesamte Sprache unterstützt und ein Vielfaches der in den bisherigen Prozessoren unterstützten Standardprozeduren in die Hardware verlagert. Das Resultat ist der erste wirklich einsetzbare Scheme-Prozessor.

Abschließend werden weitere Arbeiten beschrieben. Diese umfassen eine kurze Beschreibung der bestehenden Implementation und geplante Erweiterungen. Der Einfluß der Randbedingungen auf den Prozessor-Entwurf wird untersucht und dabei diskutiert, welche alternative Möglichkeiten die IC-Entwurfssoftware anderer Hersteller geboten hätte. Auch wird beschrieben, wie der Scheme-Prozessor für andere Zielrechner aussehen müßte. Den Abschluß bildet eine Darstellung der Vorteile, die ein zukünftiges Expertensystem aus dem hier entwickelten Entwurfsverfahren gegenüber traditionellen Synthesewerkzeugen ziehen kann.

# Kapitel 1

# Einleitung

Programmiersprachen sind so alt wie die elektronische Datenverarbeitung. Am Anfang der Entwicklung erlaubten Assembler die symbolische Benennung von Maschinenbefehlen und Hauptspeicheradressen. Später etablierten sich problemorientierte Programmiersprachen, die Programme auch zwischen unterschiedlichen Rechnern portabel gestalten sollten. Heute gibt es eine Vielzahl von Programmiersprachen für die unterschiedlichsten Anwendungsgebiete.

LISP ist die nach FORTRAN zweitälteste heute noch eingesetzte höhere Programmiersprache. LISP wurde bereits 1960 von John McCarty beschrieben [McCa60]. Der Begriff „funktionale Sprache“ wird von den Verfechtern seiteneffektfreier Sprachen, die mit dem 1971 in Wadsworths Dissertation erstmals beschrieben [Wads71] Verfahren der Graphenreduktion implementiert werden, mit einem Anspruch auf Ausschließlichkeit für sich reserviert. Diese Sprachen und ihre Implementierung ist bei Peyton Jones [Peyt87] dargestellt und in deutscher Sprache zum Beispiel von Maurer und Wilhelm [MaWi89] zusammengefaßt.

In dieser Arbeit erstreckt sich der Begriff funktionale Sprachen auch auf LISP und seine Dialekte. LISP ist mit seinen Eigenschaften, der gleichen Darstellung für Programme und Daten sowie der dynamischen Typisierung im Gegensatz zu den reinen funktionalen Sprachen wie etwa ML, Ponder, Lazy ML und Miranda [Peyt87] eine sehr weit verbreitete Programmiersprache, die insbesondere im Bereich der künstlichen Intelligenz immer mehr Verwendung findet.

Scheme, ein Dialekt von LISP, entstand 1975 am Massachusetts Institute of Technology (MIT) [SuSt75]. Die statischen Gültigkeitsbereiche für Variablen erlaubten die Aufhebung einer Einschränkung traditioneller LISP-Dialekte bei der Verwendung von Funktionen: Funktionen können in Scheme ohne Komplikationen auch als Funktionswerte zurückgegeben und als Parameter an Funktionen übergeben werden. Damit war Scheme eine orthogonal aufgebaute Sprache, in der Funktionen gegenüber allen anderen Objekten nicht eingeschränkt wurden, und Scheme erlangte an amerikanischen Universitäten Bedeutung, wo Yale die ähnliche Sprache T [Rees84] entwickelte und Scheme auch an der Indiana University implementiert wurde [Fell83, Frie85]. Sogar in der Grundausbildung der Informatik- und Elektrotechnikstudenten wurde Scheme an diesen Hochschulen eingesetzt. Der Kurs am MIT wurde besonders durch das Lehrbuch von Abelson und Sussman „Structure and Interpretation of Computer Programs“ [AbSu85] verbreitet. Nach diesem Lehrbuch wird inzwischen an vielen Hochschulen auch in Europa die Informatikgrundausbildung gegeben und damit auch Scheme verwendet. Scheme ist eine Stufe eines europäischen LISP-Standards [Padg86] und wird auch von der IEEE genormt. Die wachsende Anzahl von Lehrbüchern für Scheme [Dybv87, Eise88, Smit88] zeigt die steigende Akzeptanz der Sprache.

Der Preis, den der Anwender bei der Nutzung einer Sprache wie Scheme zahlen muß, besteht zum einen in zusätzlichem Aufwand zur Laufzeit (Typprüfungen, aufwendige Prozeduraufrufe, Un-

terstützung von Closures[1] und Continuations[2] ohne Restriktionen) und zum anderen in durch die Datenstrukturen (Objekte mit Garbage Collection) verursachtem Rechenzeit- und Speichermehrbedarf. Von vielen Anwendern wird eine Scheme-Implementation auf herkömmlichen Rechnern (Universalrechnern) aufgrund der Laufzeiten und des Speicherbedarfs als nicht akzeptabel empfunden. Gerade dies war die Intention für eine Hardwareimplementation, die einen preiswerten Mikrorechner durch einen zusätzlichen Scheme-Prozessor gegenüber Softwareimplementationen um den Faktor 10 beschleunigt und auch den Speicherplatzbedarf verringert. Diese Arbeit stellt nicht nur die Hardwareimplementation vor; auch die dafür entwickelten neuen Techniken, die die gesamten Entwicklung in elf Monaten erlaubten, werden ebenso wie die gewählten Entwurfsentscheidungen, die es ermöglichten, mehr als die doppelte Transistorzahl, als der Hersteller angibt, auf dem entworfenen integrierten Schaltkreis unterzubringen, beschrieben. Ferner wird der entstandene Chip mit den Entwürfen des MITs verglichen und gezeigt, daß dieser Chip als einziger die Sprache Scheme vollständig unterstützt und auch im Vergleich mit den bisherigen Prozessoren ein Vielfaches der Anzahl von Standardprozeduren als Prozessorbefehle bietet.

[1] Folge: Die lokalen Variablen einer Prozedur bleiben auch nach Rückkehr von der Prozedur noch zugreifbar.
[2] Folge: Eine Prozedur kann nach einmaligem Aufruf beliebig häufig zurückkehren.

# Kapitel 2

# Die Programmiersprache Scheme

Die Programmiersprache Scheme ist ein Dialekt der Sprache LISP, die 1960 zuerst beschrieben wurde [McCa60]. Der derzeit aktuelle Report der Sprache Scheme ist der „Revised$^3$ Report on the Algorithmic Language Scheme“[1] [Rees86]. Da Scheme ausschließlich in Forschung und Lehre genutzt wird, unterliegt es laufenden Veränderungen, denen im Abstand von einigen Jahren jeweils ein neuer Report Rechnung trägt [StSu78, Clin85, Rees86]. Der „Revised$^4$ Report“ soll 1990 erscheinen [WG89] und den aktuellen Report ablösen. Ferner wird Scheme international standardisiert (IEEE P1178, ISO/IEC JTC1/SC22/WG16), wobei der „Revised$^4$ Report“ auch als Vorschlag für diese Standards dienen soll [WG89].

In diesem Kapitel soll ein Überblick über Prinzipien und Eigenschaften der Sprache Scheme gegeben werden. Auf Unterschiede zu traditionellen LISP-Dialekten wird hingewiesen. Für eine vollständige, systematische Darstellung wird auf den Sprachreport [Rees86] verwiesen. Dort wird die Sprache erklärt, und es werden dort Syntax und Semantik formal spezifiziert.

## 2.1 Aufbau von Scheme

Scheme besteht aus Ausdrücken, die entweder primitive Ausdrücke (primitive expressions) oder abgeleitete Ausdrücke (derived expressions) sind. Die abgeleiteten Ausdrücke dienen der bequemen und besonders lesbaren Notation, sie werden vor der Auswertung der Ausdrücke in Kombinationen primitiver Ausdrücke übersetzt.

```
=>       do       or
and      else     quasiquote
begin    if       quote
case     lambda   set!
cond     let      unquote
define   let*     unquote-splicing
delay    letrec
```

Tabelle 2.1: Schlüsselworte in Scheme.

Scheme kennt Schlüsselworte (Tabelle 2.1), die nicht als Bezeichner Verwendung finden dürfen.

[1] Dieser Titel ist an die Reports von Algol-60 [Naur63] angelehnt.

Primitive und abgeleitete Ausdrücke werden syntaktisch durch die Schlüsselworte von Prozeduraufrufen unterschieden.

Scheme ist eine in einer interaktiven Umgebung verwendete Programmiersprache. Der Programmierer gibt Definitionen und Ausdrücke ein, die sofort ausgewertet werden. Programme und Daten werden (extern) gleich dargestellt. Ausdrücke und Definitionen sind als Programme zu interpretierende Daten.

## 2.2 Vergleich von Scheme mit Pascal

Hier sollen die Unterschiede zu Pascal [JeWi78] als Vertreter einer verbreiteten, modernen Universalsprache dargestellt werden.

### 2.2.1 Daten

In Pascal haben Variablen, Parameter von Prozeduren und Funktionen sowie die Funktionswerte einen festen vom Programmierer festzulegenden Typ. Diese Typen werden aus einer Menge skalarer Typen gewählt (Aufzählungen, Boolean, Char, Integer, Real) und können zu strukturierten Typen (Array, File, Pointer, Recond, Set) kombiniert werden. Die Implementation von Pascal erfolgt durch einen Compiler, der die Typinformationen verwendet, um Speicher zu allozieren (z.B. belegt ein Datum vom Typ Boolean in der Regel weniger Speicherplatz als eines vom Typ Real), Typwandlungen vorzunehmen (z.B. bei Zuweisung eines Integer-Ausdrucks an eine Realvariable), Operationen auszuwählen (z.B. bei +: Integeraddition, Realaddition oder Mengenvereinigung) und Überprüfungen zur Übersetzungszeit vorzunehmen (z.B. führt ein Vergleich zwischen Daten der Typen Char und Integer bei der Übersetzung zu einer Fehlermeldung). Eine Reihe von Einschränkungen der Typen möglicher Daten besagen, daß Funktionen nur Daten skalarer Typen und Pointer als Funktionswert haben dürfen (also in einem Maschinenwort unterzubringende Daten), während Variablen und Parameter Daten aller Typen aufnehmen dürfen. Auch können Konstante von strukurierten Typen (außer von Strings und Sets) nicht notiert werden.

Die Programme in Scheme arbeiten auf Objekten. Objekte bestehen aus einem Typ und einem Datum. Diese Objekte können als Parameter an Prozeduren übergeben werden, als Funktionswert von Prozeduren dienen und an Variablen gebunden werden. Die Tabelle 2.2 zeigt eine Liste von Objekttypen in Scheme, in der neben Objekten wie Zahlen auch Paare und Vektoren aufgeführt

| | |
|---|---|
| Boole'sche Konstante | Paar |
| Zahl | Vektor |
| String | Prozedur |
| Symbol | |
| Character | |
| leere Liste | |

Tabelle 2.2: Objekte in Scheme.

sind. Vektoren entsprechen den Feldern (Arrays) in Pascal: Sie beeinhalten Verweise auf eine beim Erzeugen des Vektors festgelegte Anzahl von Objekten, auf welche mit einem berechneten Index zugegriffen werden kann. Paare sind Objekte, die Verweise auf zwei Objekte, den CAR und den CDR enthalten. Paare dienen zum Aufbau verketteter Datenstrukturen, von denen Listen die

häufigste Verwendung sind, bei denen der CAR auf ein Objekt in der Liste verweist und der CDR auf den Rest der Liste (ein weiteres Paar oder die leere Liste) zeigt.

Konstanten aller in der Tabelle aufgeführten Objektypen können in Scheme notiert werden (Abbildung 2.1).

```
#t #f          ; Boole'sche Werte
7              ; Zahl
"String"       ; Zeichenkette
xyz            ; Symbol
#\A            ; Zeichen 'A'
()             ; leere Liste
(a . b)        ; Punktepaar
(a b c)        ; Listenschreibweise für  (a . (b . (c . ())))
#(1 2 3)       ; Vektor
(lambda (a)    ; Prozedur für Quadrierung
  (* a a))
```

Abbildung 2.1: Konstanten verschiedener Objekttypen in Scheme.

### 2.2.2 Prozeduren

Pascal unterscheidet Prozeduren und Funktionen. Prozeduren sind Unterprogramme, die für Seiteneffekte aufgerufen werden. Funktionen können in Ausdrücken verwendet werden; sie liefern einen Funktionswert. Prozeduren und Funktionen können zwar als Prozedurparameter übergeben werden. Sie können jedoch nicht an Variablen zugewiesen oder in Datenstrukturen aufgenommen werden.

In Scheme sind Prozeduren nichts grundsätzlich anderes als Daten; Prozeduren zählen, wie in Tabelle 2.2 gezeigt, zu den Objekten. Prozeduren sind Objekte, die als Operation den Aufruf erlauben. Alle Prozeduraufrufe liefern einen Funktionswert, der bei manchen Prozeduren, die wegen ihrer Seiteneffekte aufgerufen werden, nicht definiert ist. Da Prozeduren Objekte sind, unterliegen sie, verglichen mit den anderen Objekten, keinerlei Einschränkungen: Prozeduren können wie alle Objekte als Parameter an Prozeduren übergeben werden, als Funktionswert von Prozeduren dienen, an Variablen gebunden und in Datenstrukturen aufgenommen werden. Auch können, wie in Abbildung 2.1 gezeigt, Prozeduren in Scheme als Konstanten notiert werden, d.h. ohne einen Namen zu erhalten, z.B. als Parameter bei einem Funktionsaufruf dienen. Die Abbildung 2.2 zeigt eine Anwendung dieser Möglichkeit in einem Beipielprogramm, das die Identitätsfunktion als einzigen Parameter an eine Prozedur `f` übergibt. In der Pascalversion muß nicht nur die Identätsfunktion benannt (mit `id`) werden, darüberhinaus sind auch Typen für Parameter und Funktionswert zu spezifizieren, wodurch diese Identitätsfunktion nur für Fließkommazahlen nutzbar ist. Für andere Typen müssen gesonderte Identitätsfunktionen programmiert werden. Die Identitätsfunktion in Scheme erlaubt beliebige Argumente und benötigt auch keinen Namen.

### 2.2.3 Gültigkeitsbereiche von Variablen (Scoping)

Die dynamische Variablenbindung wurde in LISP-Dialekten seit 1960 bis zur Entwicklung von Scheme [SuSt75] ausschließlich verwendet. Erst Common LISP [Stee84] führte — aufgrund der Erfahrungen mit Scheme — die statische Variablenbindung ein.

```
PROCEDURE call_f_with_id;     { Beispiel in Pascal }
 FUNCTION id(r: real): real; { Identitätsfunktion }
  BEGIN
   id := r
  END;
 BEGIN { call_f_with_id }
  f(id)
 END;

(define (call_f_with_id)      ; Beispiel in Scheme
  (f (lambda (r) r)))         ; Identitätsfunktion: (lambda (r) r)
```

Abbildung 2.2: Übergabe der Identitätsfunktion in Pascal und Scheme.

#### 2.2.3.1 Die dynamische Variablenbindung

Bei der dynamischen Variablenbindung werden die Bindungen freier Variablen einer Prozedur bei den Aufrufern der Prozedur gesucht. Die Abbildung 2.3 zeigt ein Beispiel für die traditionell

```
(define (f x)
  (* x y))                    ⟹ f
(define (g y)                 ⟹ g
  (f 3))
(f 7)                         ⟹ Fehler: y nicht gebunden
(g 2)                         ⟹ 6
```

Abbildung 2.3: Beispiel für dynamische Variablenbindung.

in Lisp verwendete dynamische Variablenbindung, die in Scheme *nicht* verwendet wird. Wie im Report [Rees86] steht nach „$\Longrightarrow$“ das Ergebnis der Auswertung eines Ausdrucks. Zuerst werden die Prozeduren `f` und `g` definiert. Die Prozedur `f` multipliziert ihren Parameter mit dem Ergebnis der Evaluation von `y`, und `g` ruft `f` mit dem Argument 3 auf. Wenn `f` mit einem beliebigen Argument aufgerufen wird, erfolgt eine Fehlermeldung, weil `y` nicht an einen Wert gebunden ist. Wenn hingegen `g` mit einem Argument aufgerufen wird, so wird das Argument an `y` gebunden. `g` ruft `f` auf, und `f` evaluiert `y` zu dem in `g` gebundenen Wert. Die dynamische Variablenbindung bewirkt, daß die Bindungen der Aufrufer einer Prozedur nach der Bindung einer nichtlokal gebundenen Variablen durchsucht werden.

#### 2.2.3.2 Die statische Variablenbindung

In Scheme haben Variablen einen begrenzten Gültigkeitsbereich, der dem blockorientierter Programmiersprachen wie Algol, PL/1 und Pascal entspricht. Scheme weicht damit von traditionellen LISP-Dialekten ab.

Bei statischer Variablenbindung ist eine Variable nur in einem statischen Bereich sichtbar. Mit statischem Bereich ist gemeint, daß der Bereich aus dem Programmtext erkennbar ist, nicht von der Ausführung, wie etwa der Aufruffolge bei der dynamischen Variablenbindung, abhängt. Lambda-Ausdrücke können geschachtelt werden; nur innerhalb der Lambda-Ausdrücke sind die Parameter sichtbar.

Der Aufruf der Prozedur `g` in Abbildung 2.3 würde in Scheme genau wie der Aufruf von `f` mit der Fehlermeldung abbrechen, weil die Variable `y` in der Prozedur `f` nicht in der statischen Umgebung gefunden werden kann.

Zusätzlich zu den Parametern der Lambda-Ausdrücke gibt es in Scheme eine globale Bindungsumgebung, in der die Bindungen an Variablen gesucht werden, wenn sie nicht lokal oder in einer statisch umschließenden Umgebung gebunden sind. In diese globale Bindungsumgebung kann mit `define` eine Bindung geändert oder neu eingerichtet werden.

#### 2.2.3.3 Closures

Ein Lambda-Ausdruck ergibt bei der Auswertung ein Prozedurobjekt. Dieses Prozedurobjekt enthält in Scheme die bei der Auswertung des Lambda-Ausdrucks bestehende Bindungsumgebung. Wenn die Prozedur aufgerufen wird, wird die im Prozedurobjekt gespeicherte Bindungsumgebung um die Bindungen der aktuellen Parameter an die formalen Parameter erweitert. Ein Prozedurobjekt, das die Bindungsumgebung bei seiner Entstehung speichert, wird Closure genannt. Als deutscher Begriff wird, zumindest in [MaWi89], „Abschluß“ verwendet.

Die Kombination statischer Variablenbindung mit Closures erlaubt die Verwendung von Prozeduren als Parameter und Funktionswerte von Prozeduren. Die Abbildung 2.4 zeigt eine Prozedur `make-proc`, die als Funktionswert eine Prozedur (eine Closure) liefert. Ein Aufruf der Proze-

```
(define (make-proc op)
  (lambda (a)
    (op a a)))                ⟹ make-proc
(make-proc *)                 ⟹ Prozedurobjekt
(define sqr (make-proc *))    ⟹ sqr
(sqr 16)                      ⟹ 256
(define string-twice
  (make-proc string-append))  ⟹ string-twice
(string-twice "abc")          ⟹ "abcabc"
(sqr 4)                       ⟹ 16
```

Abbildung 2.4: Verwendungsmöglichkeit für eine Closure.

dur `make-proc` liefert ein Prozedurobjekt, eine Closure als Funktionswert. Es handelt sich bei dem Funktionswert um eine Prozedur, die ein Argument erwartet und es als beide Parameter beim Aufruf einer zweistelligen Funktion übergibt. Die zweistellige Funktion ist das Argument von `make-proc`. Die Definition von `sqr` definiert eine Prozedur für die Quadrierung von Zahlen, weil `make-proc` als Argument die Multiplikationsprozedur übergeben wird. Die Prozedur `string-twice` bewirkt eine Verdopplung von Strings, weil `string-append` seine Argumente konkateniert. Die Aufrufe arbeiten deshalb wie beschrieben, weil das Prozedurobjekt eine Closure ist, die die jeweilige Bindung an `op` gespeichert hat (Abbildung 2.5).

Die Modellierung von Prozedurparametern durch Closures ist nicht auf Scheme beschränkt. Jede Programmiersprache mit statischer Variablenbindung, die geschachtelte Prozeduren und Prozedurparameter unterstützt, muß als Prozedurparameter eine Closure übergeben. Eine solche Programmiersprache ist Pascal. Ein in Pascal arbeitender Programmierer wird wohl nie in die Lage kommen, in Begriffen wie Closures sein Programm beschreiben zu müssen. Wenn in Pascal aber eine Prozedur als Parameter übergeben wird, wird tatsächlich die Anfangsadresse des Codes der Prozedur und ein Verweis auf die statisch umgebende Bindungsumgebung (static link) übergeben

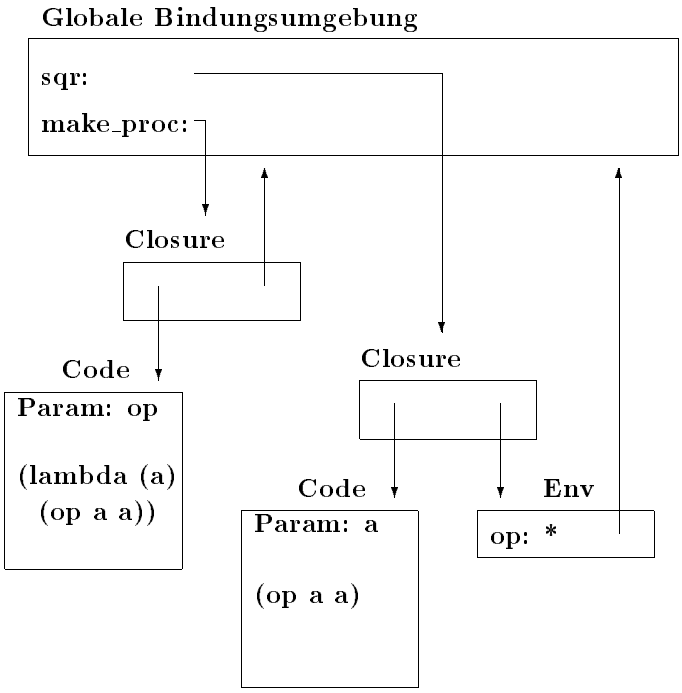



Abbildung 2.5: Veranschaulichung der Closures.

(vergl. S. 6-90 in [Lohs86]). Diese beiden Daten bilden eine Closure. Weil in Pascal Prozedurparameter (Closures) nicht an Variablen zugewiesen und nicht als Funktionswert zurückgegeben werden können, kann weder die Closure noch die von ihr referenzierte Bindungsumgebung eine längere Lebensdauer als die sie einrichtende Prozedur haben. Die Möglichkeiten wie lokaler Zustand, die Closures in Scheme dem Programmierer bieten, sind in Pascal nicht möglich. Diese Einschränkungen ermöglichen, daß in Pascal alle Bindungsumgebungen im Stack angelegt werden können.

### 2.2.4 Kontrollkonstrukte

Die wichtigsten Kontrollstrukturen in Pascal sind Schleifenkonstrukte (`while`, `repeat` und `for`) und Bedingungen (`if` und `case`). Zudem können Prozeduren (auch rekursiv) aufgerufen werden und Sprünge (`goto`) lokal innerhalb einer Prozedur und in statisch übergeordnete Prozeduren erfolgen.

Scheme kennt als primitive Ausdrücke die bedingte Auswertung (`if`) und den Prozeduraufruf als Kontrollstrukturen. Unter den abgeleiteten Ausdrücken sind `cond`, `case` und `do` Kontrollstrukturen, die in Kombinationen der primitiven Ausdrücke `if` und Prozeduraufruf übersetzt werden. Iterationen können durch endrekursive (tail recursive) Prozeduraufrufe modelliert werden, da festgelegt ist, daß endrekursive Aufrufe bei konstantem Speicherbedarf zu implementieren sind[2].

Eine weitere mächtige Kontrollstruktur wird durch die Standardprozedur `call-with-current-continuation` bereitgestellt, die die Programmfortsetzung (Continuation) als Prozedur (escape

[2]Die Optimierung der Endrekursion ist nur bei statischer Variablenbindung möglich, da bei dynamischer Variablenbindung die Bindungen des Aufrufers erhalten bleiben müssen.

procedure) zugänglich macht. `Call-with-current-continuation` erwartet eine Prozedur mit einem Parameter als Argument. Wenn `call-with-current-continuation` aufgerufen wird, bindet es die Fortsetzung des gesamten Programms als einstellige Prozedur an diesen Parameter und ruft seinen Prozedurparameter anschließend auf. Die Fortsetzung des Programms ist ein Prozedurobjekt ohne jegliche Beschränkungen; es kann wie alle Objekte als Parameter an Prozeduren übergeben werden, als Funktionswert von Prozeduren dienen, an Variablen gebunden und in Datenstrukturen aufgenommen werden. Insbesondere ist die Lebensdauer der Continuation unbegrenzt. Sie kann mehrfach aufgerufen und auch noch nach der Rückkehr von `call-with-current-continuation` aufgerufen werden. Continuations bieten über die Modellierung Pascals nicht-lokaler `goto`s hinaus die Möglichkeiten zum Programmieren von Koroutinen [Hayn84, Hayn86] und Backtracking. Die Fortsetzung ist eine Prozedur mit einem Parameter, welcher den Funktionswert von `call-with-current-continuation` bildet. `Call-with-current-continuation` kann damit nach einem einzigen Aufruf beliebig häufig und mit unterschiedlichen Funktionswerten zurückkehren. Im Beispielprogramm in Abbildung 2.6 wird `call-with-current-continuation` als Fehlerausgang eingesetzt. Die Prozedur `div-list` erwartet zwei Listen als Parameter und liefert eine Liste mit den Quotienten der Elemente der Parameter. Eine Division durch null wird vermieden, und führt zum Aufruf der Continuation mit Rückgabe eines Strings mit einer Fehlermeldung. Ein Beispiel, das die unbegrenzte Lebensdauer einer Continuation demonstriert, zeigt Abbildung 2.7. Die globale Variable `g` wird definiert, damit sie bei der Ausführung von `f` zur Verfügung steht. Die Prozedur `f` hat eine lokale Variable `counter`, die am Anfang auf null initialisiert wird, anschließend wird der Funktionswert des Aufrufs `call-with-current-continuation` mit drei multipliziert ausgegeben, eine neue Ausgabezeile begonnen, die Variable `counter` um eins erhöht und als Funktionswert zurückgegeben. Nach diesem ersten Aufruf ist der Funktionswert von `call-with-current-continuation` 1 und damit der verdreifachte Wert 3. Der Funktionswert von `f` ist 1, da `counter` auf 1 gesetzt wurde. Gleichzeitig wurde die Continuation an die globale Variable `g` gebunden. Nach dem ersten Aufruf der Continuation ist der Funktionswert von `call-with-current-continuation` 7 und damit der verdreifachte Wert 21. Der Funktionswert von `f` ist 2, da `counter` auf 2 erhöht wurde. Der zweite Aufruf von `g` zeigt, daß die Continuation beliebig oft aufgerufen werden kann.

```
(define (div-list l1 l2)
  (call-with-current-continuation
    (lambda (exit)
      (define (div a1 a2)
        (if (null? a1)
            '()
            (if (zero? (car a2))
                (exit "Fehler: Division durch 0")
                (cons (/ (car a1) (car a2))
                      (div (cdr a1) (cdr a2))))))
      (div l1 l2))))           ⟹ div-list
(div-list '(4 6 8) '(1 2 4))   ⟹ (4 3 2)
(div-list '(4 6 8) '(1 0 4))   ⟹ "Fehler: Division durch 0"
```

Abbildung 2.6: Beispiel für `call-with-current-continuation`.

```
(define g '())                  ⟹ g
(define (f)
  (let ((counter 0))
    (write (* 3
              (call-with-current-continuation
                (lambda (conti)
                  (set! g conti)
                  1)))
     (newline)
     (set! counter (+ counter 1))
     counter))                  ⟹ f
  (f)                           ⟹ 1  ;  Ausgabe: 3
  (g 7)                         ⟹ 2  ;  Ausgabe: 21
  (g -2)                        ⟹ 3  ;  Ausgabe: -6
```

Abbildung 2.7: Beispiel für die unbegrenzte Lebensdauer einer Continuation.

# Kapitel 3

# Die Implementierung von Scheme

Hier sollen die Probleme und Lösungsmöglichkeiten bei der Implementierung von Scheme beschrieben werden. Unter *Implementierung* wird hier immer die allgemeine Aufgabe der Modellierung einer Scheme-Umgebung auf einem realen Rechner verstanden. Ein konkretes Programm oder eine Hardware, die das leistet, wird hier als *Implementation* bezeichnet.

Beschrieben werden Repräsentationsmöglichkeiten der Objekte. Objekte sind sowohl die extern repräsentierbaren Daten als auch interne Strukturen wie Bindungsumgebungen, Closures, Controlframes und Continuations, die bei der Abarbeitung von Scheme-Ausdrücken entstehen. In der Unterstützung einer speicherplatz- und rechenzeiteffizienten Repräsentation der Objekte unterscheidet sich der hier zu beschreibende Scheme-Prozessor grundlegend von seinen Vorgängern.

Objekte haben eine unbegrenzte Lebensdauer. Objekte müssen grundsätzlich in einem Heap alloziert werden. Der Implementierung fällt die Aufgabe zu, nicht mehr referenzierte Objekte zu identifizieren und deren Speicher wiederzuverwenden.

## 3.1 Repräsention der Objekte

Objekte werden in Scheme an Bezeichner gebunden und als Funktionswert von Prozeduren zurückgeliefert. Da diese Objekte sehr häufig umfangreiche Datenstrukturen sind, werden die Objekte tatsächlich nur einmal im Speicher repräsentiert. Bestimmte Standardprozeduren können Objekte erzeugen. Bei der Übergabe von Parametern und der Rückgabe von Funktionswerten werden keine Objekte kopiert oder erzeugt, sondern Zeiger verwendet. Als Beispiel dient das Programm in Abbildung 3.1.
In diesem Programm wird eine Liste an die globale Variable `a` gebunden. Beim Einlesen dieser

```
(define a '(1 2 3))
(define b a)
(define c (cdr a))
```

Abbildung 3.1: Beispielprogramm für die Objektrepräsentation.

Liste, der Transformation von der externen in die interne Darstellung durch die Standardprozedur `read`, wird diese Liste im Speicher abgelegt. Die Abbildung 3.2 zeigt die globale Bindungsumgebung und das genau einmal repräsentierte Objekt. An die globale Variable `b` wird ein weiterer Verweis an das nur einmal repräsentierte Objekt gebunden. Die Bindung an `c` zeigt, daß auch

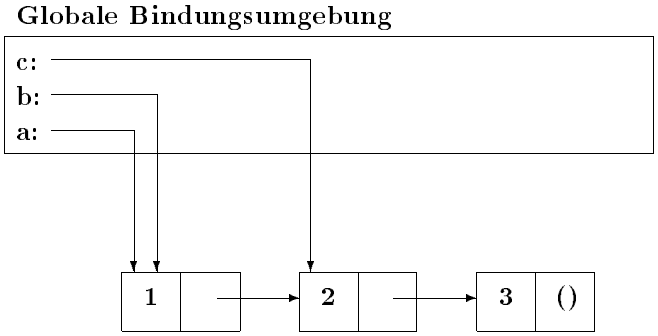



Abbildung 3.2: Veranschaulichung der Zeiger auf Objekte.

Standardprozeduren (wie `cdr`) einen Zeiger zurückliefern; kein Objekt wird kopiert. Wegen der Verwendung dieser Zeiger ist es auch nicht ganz korrekt, die Parameterübergabe in Scheme als Werteparameterübergabe zu bezeichnen. Zwar werden Ausdrücke erst ausgewertet und dann das Ergebnis an Parameter gebunden, jedoch werden Zeiger auf die Ergebnisse übergeben. So ist die Parameterübergabe von Vektoren in Scheme eine Variablenparameterübergabe.

An Bezeichner können in Scheme Objekte beliebiger Typen gebunden werden. An Prozeduren können Objekte beliebiger Typen übergeben und auch von diesen zurückgegeben werden. Standardprozeduren sind nur für Objekte bestimmter Typen definiert. Beispielsweise ist die Prozedur `*` für die Multiplikation nur für numerische Objekte als Argumente definiert. Zur Laufzeit müssen Standardprozeduren die Typen ihrer Argumente überprüfen und eventuell einen Fehler melden. Objekte müssen einschließlich ihrer Typen bei der Implementierung repräsentiert werden.

Bei der Implementierung von LISP-Dialekten ist die Entscheidung zu fällen, wie die Typen der Objekte repräsentiert werden können: Da Zeiger auf die Objekte verwendet werden, kann einerseits der Typ im Objekt kodiert sein (tagged objects), andererseits können die Zeiger eine Kennzeichnung erhalten, auf Objekte welchen Typs sie zeigen (tagged pointers). Die Kennzeichnung der Zeiger ist selbst dann möglich, wenn neben der Adresse kein weiterer Platz für einen Typ reserviert wird: Dann wird der Hauptspeicher in Seiten eingeteilt, und in jeder Seite werden nur Objekte eines bestimmten Typs untergebracht (BIBOP[1]-Verfahren); der Typ ist dann Teil der Adresse. Bei durch Typen gekennzeichneten Zeigern kann der Typ auch angeben, daß anstelle des Zeigers schon das Datum steht. Insbesondere Integerzahlen, Boole'sche Werte und Zeichen (Chars) werden auf diese Weise platzsparend und effizient repräsentiert. Die Abbildung 3.3 zeigt die Typrepräsentation im Zeiger, während Abbildung 3.4 die Typrepräsentation im Objekt demonstriert. Die Typrepräsentation im Zeiger weist einen geringeren Speicherbedarf auf, weil für die Zahlenkonstante und die leere Liste kein Zeiger auf ein Objekt gespeichert werden muß, sondern die Zahl selber und bei der leeren Liste außer dem Typ kein weiteres Datum zu repräsentieren ist. Ein weiterer Vorteil der Typrepräsentation im Zeiger ist bei den relativ häufigen Typprüfungen (`null?`, `pair?`, `symbol?`, ...) gegeben: Es wird ein Speicherzugriff gespart, weil der Typ schon zusammen mit dem Zeiger zur Verfügung steht, während bei der Typrepräsentation im Objekt erst noch über den Zeiger der Typ einzulesen ist.

Eine weitere Anforderung an Repräsentationen der Objekte stammt von der Speicherverwaltung. Alle verbreiteten Algorithmen zur Speicherverwaltung in LISP-Dialekten müssen Referenzen auf noch benötigte Objekte folgen. Sie müssen dazu Informationen vorfinden, um Zeiger von Daten wie Zahlen, Zeichen und Maschinencodebefehlen unterscheiden zu können. Zeigern wird gefolgt,

[1] BIg Bag Of Pages. Ein Extremfall dieses Verfahrens ist je ein Speicherbereich pro Objekttyp.

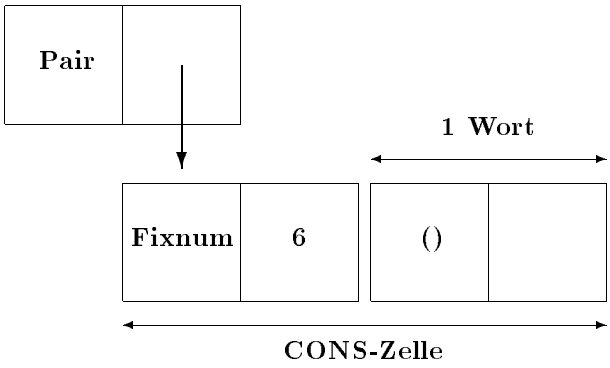



Abbildung 3.3: Das Paar `(6 . ())` bei Typrepräsentation im Zeiger.

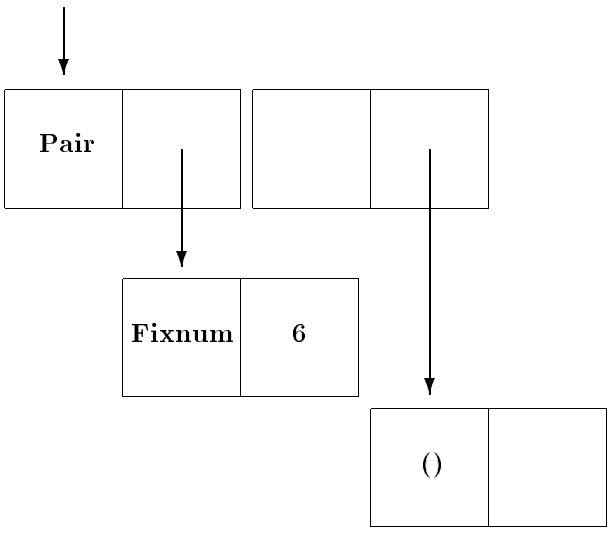



Abbildung 3.4: Das Paar `(6 . ())` bei Typrepräsentation im Objekt.

während die anderen Daten nicht als Zeiger interpretiert werden dürfen.

### 3.1.1 Repräsentation von Daten

Daten werden im Report [Rees86] die extern repräsentierbaren Objekte genannt. Daten können vom Programmierer notiert werden. Auch Programme gehören zu den Daten.

#### 3.1.1.1 Symbole

Symbole sind in traditionellen LISP-Dialekten als Zeiger auf Propertylisten implementiert und zu einer Liste, der `oblist`, zusammengefaßt. Die Zusammenfassung in der `oblist` dient dazu, daß alle Symbole (insbesondere für die Garbage Collection und den Reader) erreichbar sind. Die Auswertung von Ausdrücken ist in den traditionellen LISP-Dialekten mit Inspektionen der Propertylisten verbunden.

In Scheme wird bei der Auswertung keine Propertyliste benötigt. Propertylisten sind in Scheme nicht einmal vorgesehen, obwohl einige Implementationen sie als Erweiterung bereitstellen. Eine `oblist` wird in Scheme auch nicht gefordert.

In Scheme ist die einzige Operation auf Symbolen der Vergleich. Auch wenn Symbole als Bezeichner in Bindungsumgebungen Verwendung finden, bleibt der Vergleich beim Suchen nach einem Bezeichner die einzige Operation[2]. Beim Einlesen und Ausgeben wird zwischen der externen Darstellung der Symbole und der internen gewandelt. Die interne braucht nur den Vergleich zuzulassen, wodurch effiziente Repräsentationen Zahlen oder Zeiger (z. B. auf die Lokation einer globalen Bindung) sind.

Die Speicherverwaltung einer Implementation kann nicht mehr verwendete Symbole ermitteln und den durch sie belegten Speicher freigeben. Viele LISP-Implementationen verzichten auf das Ermitteln von durch nicht mehr benötigte Symbole belegten Speichers.

#### 3.1.1.2 Zahlen

Scheme kennt ganze Zahlen, rationale Zahlen, reelle Zahlen und komplexe Zahlen. Neben dem Typ und dem Wert der Zahlen ist als Attribut die Exaktheit zu repräsentieren und beim Einlesen und bei Operationen zu berücksichtigen.

Für die Repräsentation der Objekte bei der Implementierung der Sprache ist weniger diese Hierarchie der Zahlenmengen von Interesse als die Unterscheidung nach der Größe des für die Darstellung der Zahlen benötigten Speichers. Bei den ganzen Zahlen wird zwischen solchen, die in einem für den Anteil des Datums in einem Speicherwort zur Verfügung stehenden Bereichs Platz finden (`FIXNUM`s) und solchen, die sich über mehrere Speicherworte erstrecken (`BIGNUM`s), unterschieden. Reelle Zahlen werden als Fließkommazahlen (`FLONUM`s) dargestellt.

`FIXNUM`s werden, wie in Abbildung 3.3 gezeigt, in einem Wort untergebracht. Für `BIGNUM`s und `FLONUM`s wird Speicher alloziert und ein Zeiger auf die Zahl verwendet (boxed numbers). Dieses Verfahren bewirkt, daß der Zugriff langsamer erfolgt, weil zusätzliche Speicherzugriffe über einen Zeiger erfolgen müssen, und daß Speicher verbraucht wird, wodurch zusätzlicher Aufwand durch die Speicherverwaltung entsteht. Implementationen versuchen daher, Zahlen möglichst direkt und nicht über Zeiger repräsentieren zu können: So verwendet beispielsweise die Lispmaschine Symbolics 3600 Worte von 34-Bit Breite, um zusammen mit einem 2-Bit Typfeld noch 32-Bit ganze und Fließkommazahlen in einem Wort repräsentieren zu können [Moon85]. Bei der Verwendung von Zeigern auf Zahlen, ist zu berücksichtigen, daß neben den Zahlen noch die Information für die

[2] Tatsächlich braucht eine Implementation zur Laufzeit in Scheme nicht nach einer Bindung zu suchen: Ein Compiler kann wie in herkömmlichen Sprachen mit statischer Variablenbindung die Positionen innerhalb der Bindungsumgebungen ermitteln und braucht die Bezeicher zur Laufzeit nicht zu repräsentieren.

Speicherverwaltung abzulegen ist, daß es sich nicht um Zeiger handelt. In der Abbildung 3.5 wird das durch Aufteilung in Worte des Typs `FIXNUM` erreicht. Bei `BIGNUM`s ist auch noch die Länge der Zahl im Speicher abzulegen.

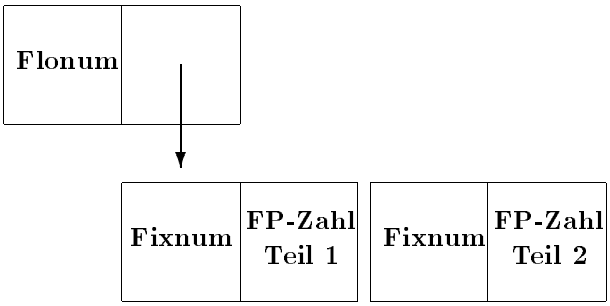



Abbildung 3.5: Über Zeiger zu referenzierende Fließkommazahl (boxed `FLONUM`).

#### 3.1.1.3 Zeichen (Chars)

Zeichen unterscheiden sich nur im Typ von `FIXNUM`s. Der Report [Rees86] erlaubt sogar eine Identität von Zeichen mit Zahlen. Beim Einlesen dürfen Zeichen in Zahlen gewandelt werden und sind dann durch Scheme-Programme nicht mehr von Zahlen zu unterscheiden.

#### 3.1.1.4 Zeichenketten (Strings)

Zeichenketten können als Listen von Zeichen dargestellt werden oder als Länge gefolgt von einzelnen Zeichen im Speicher stehen. Die Wahl der Repräsentation hängt von den Fähigkeiten der Speicherverwaltung, nämlich inwieweit Speicherbereiche unterschiedlicher Länge verwaltet werden können, ab. Die in Abbildung 3.6 dargestellte Repräsentation speichert jeweils drei Zeichen des Strings in einem Wort, das durch den Typ `FIXNUM` vor Veränderungen bei der Speicherverwaltung geschützt wird.

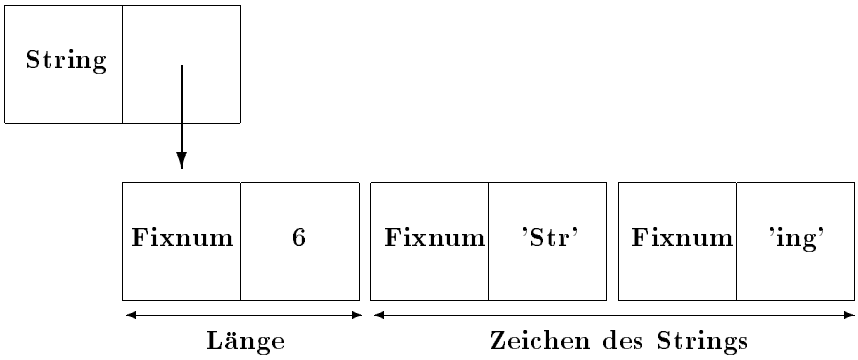



Abbildung 3.6: Repräsentation eines Strings als kontinuierlicher Speicherbereich.

#### 3.1.1.5 Die leere Liste

Die leere Liste ist durch ihren Typ bestimmt. Dieses Objekt wurde in Abbildung 3.3 gezeigt. Die Repräsentation durch einen eigenen Typ anstelle eines Paares mit einer für die leere Liste reservierten Adresse berücksichtigt, daß sich die leere Liste und Paare bezüglich der Prädikate `pair?` und `null?` unterscheiden.

#### 3.1.1.6 Paare und Listen

Paare und die aus ihnen konstruierten Listen werden in LISP-Dialekten praktisch immer als CONS-Zellen repräsentiert: Eine CONS-Zelle wird aus zwei direkt im Speicher aufeinander folgenden Maschinenworten[3] gebildet, dem CAR und dem CDR der in Abbildung 3.7 gezeigten CONS-Zelle. Eine Liste `(1 2 3)` wird durch CONS-Zellen, wie in Abbildung 3.8 gezeigt, repräsentiert.

Abbildung 3.7: Aufbau einer CONS-Zelle.

Abbildung 3.8: Repräsentation der Liste `(1 2 3)` mit CONS-Zellen.

Auffällig ist, daß bei durch CONS-Zellen dargestellten Listen die überwiegende Anzahl der CDR der CONS-Zellen einen Zeiger auf eine weitere CONS-Zelle oder die leere Liste enthält. Da in LISP und Scheme fast alle Datenstrukturen Listen sind, kann durch Berücksichtigung dieser Eigenschaft eine Kompression der Listen bis auf halbierten Speicherbedarf erzielt werden: Bei der CDR-Kodierung [Hans69] werden nicht mehr CONS-Zellen aus zwei Maschinenwörtern verwaltet, sondern einzelne Maschinenwörter, die CAR-Felder repräsentieren und um einen 2 Bit breiten CDR-Code erweitert sind. Der CDR-Code kann die in Tabelle 3.1 gezeigten Werte annehmen.

| | |
|---|---|
| **Normal** | Das folgende Speicherwort enthält den CDR. Dieser Fall ist identisch zu CONS-Zellen. |
| **Next** | Der CDR ist eine Liste, dessen erstes Element an der nächsten Adresse liegt. Ein Zeiger auf den CDR wird daher nicht mehr repräsentiert. |
| **Nil** | Letztes Element der Liste, der CDR ist die leere Liste. |
| **Indirekt** | Zeiger auf Zelle mit CDR-Code **Normal**, in der CAR und CDR stehen. Diese Verweise entstehen nur, wenn mit der destruktiven Operation `set-cdr!` bei CDR-Code **Next** oder **Nil** gearbeitet wird. |

Tabelle 3.1: CDR-Codes.

Die Liste `(1 2 3)` wird mit CDR-Kodierung, wie in Abbildung 3.9 gezeigt, repräsentiert.

[3] Hier wird von modernen Architekturen mit großem Adreßraum ausgegangen. Viele LISPs wurden ursprünglich auf dem DECsystem10 implementiert. Dort ist ein Maschinenwort 36 Bit breit, und Adressen sind 18 Bit breit, so daß eine CONS-Zelle ein Maschinenwort belegt [Gabr85]. Auch eine Implementation von MacLISP auf einem TR440 allozierte CONS-Zellen in nur einem 48-Bit Wort als zwei Zeiger von jeweils 24-Bit [Laub76].

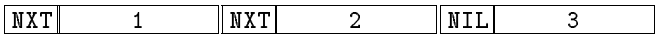



Abbildung 3.9: Repräsentation der Liste `(1 2 3)` in CDR-Kodierung.

In diesem Beipiel hat sich der Speicherbedarf für die Liste durch die CDR-Kodierung von 6 Speicherwörtern auf 3 Speicherwörter halbiert. Der Preis für die CDR-Kodierung ist, daß bei jeder der Listenoperationen `car` und `cdr` immer anhand des CDR-Codes verzweigt werden muß: Bei `car` muß **Indirekt** anders behandelt werden, bei `cdr` müssen sogar alle vier Fälle unterschieden werden. Wegen der damit verbundenen zusätzlichen Abfragen und Verzweigungen wird die CDR-Kodierung nur in Hardwareimplementationen eingesetzt.

#### 3.1.1.7 Vektoren

Vektoren in Scheme sind die aus anderen Programmiersprachen bekannten Felder. Ein zur Laufzeit errechneter Index wählt ein Element aus. Auch in Scheme können Vektoren als Block von im Speicher aufeinander folgenden Elementen repräsentiert werden. Zusätzlich muß auch die Länge des Vektors für die Standardprozedur `vector-length` und die Speicherverwaltung repräsentiert sein. Die Abbildung 3.10 zeigt die Darstellung eines Vektors als kontinuierlicher Speicherbereich.

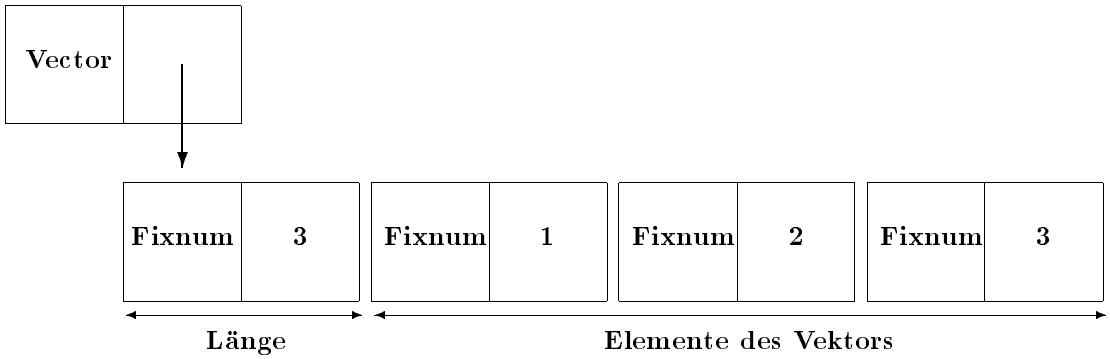



Abbildung 3.10: Repräsentation des Vektors `#(1 2 3)`.

### 3.1.2 Interne Objekte

Unter internen Objekten sind jene Objekte zu verstehen, die bei der Abarbeitung von Scheme-Programmen entstehen. Zu diesen Objekten gehören neben Interndarstellungen von Programmen insbesondere Closures, Bindungsumgebungen und Controlframes.

#### 3.1.2.1 Closures

Die Repräsentation einer Closure entspricht der einer CONS-Zelle, da auch eine Closure ein Objekt ist, das auf genau zwei weitere Objekte — die interne Repräsentation ihrer Prozedur und die beim Erzeugen der Closure aktuelle Bindungsumgebung — verweist (vergl. Abbildung 2.5 auf Seite 8).

#### 3.1.2.2 Bindungsumgebungen (Environments)

Bindungsumgebungen werden beim Aufruf einer Prozedur eingerichtet. Die Ergebnisse der Auswertung der aktuellen Parameter werden an die formalen Parameter aus dem $\lambda$-Ausdruck der aufgerufenen Prozedur gebunden und bilden zusammen mit einem Zeiger auf die statisch übergeordnete Bindungsumgebung eine Bindungsumgebung. Ein Beispielprogramm für einen Prozeduraufruf steht in Abbildung 3.11. Eine Prozedur `f` mit den beiden formalen Parametern `a` und `b` wird definiert und anschließend mit den aktuellen Parametern `1` und `2` aufgerufen. Die Abbildung 3.12

```
(define f (lambda (a b) (cons a b)))
(f 1 2)
```

Abbildung 3.11: Beispielprogramm für einen Prozeduraufruf.

zeigt die Bindungsumgebung nach Aufruf der Prozedur `f`: Die aktuellen Parameter, in diesem Fall die Zahlen `1` und `2`, sind an die formalen Parameter `a` und `b` gebunden, und die Bindungsumgebung enthält einen Verweis auf die statisch übergeordnete Bindungsumgebung, in diesem Fall die globale Bindungsumgebung. Bei der Evaluierung der Symbole werden die Bindungen entlang

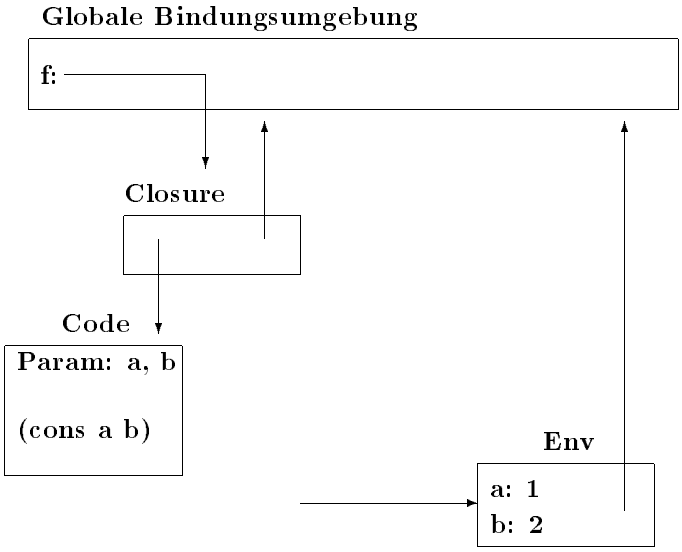



Abbildung 3.12: Die Bindungsumgebung von `f` nach dem Funktionsaufruf.

der verketteten Bindungsumgebungen gesucht. Die Bindungen von `a` und `b` werden in der rechts unten in der Abbildung gezeigten Bindungsumgebung gefunden. Die Bindung von `cons` wird in der übergeordneten Bindungsumgebung, der globalen Bindungsumgebung gefunden.

Da Scheme eine Sprache mit statischer Variablenbindung ist, in der die lokalen Variablen nur in einem aus dem Programmtext ermittelbaren Bereich sichtbar sind, brauchen die Bindungen der Variablen zur Laufzeit nicht wirklich gesucht zu werden. In diesem Beispiel heißt das: Die Variablen `a` und `b` sind nur innerhalb der Prozedur `f` sichtbar. Bei der Ausführung der Prozedur muß nur geprüft werden, daß exakt zwei Parameter übergeben wurden. Die Zugriffe auf die Bindungen der Parameter können unabhängig von deren Namen erfolgen. Der Ausdruck

```
(cons a b)
```

im $\lambda$-Ausdruck kann durch

```
(cons [Arg 1] [Arg 2])
```

ersetzt werden, so daß auf einen Parameter mit einer bestimmten Nummer zugegriffen wird, ohne daß dessen Name zu repräsentieren ist. Im allgemeinen Fall ist diese Nummer ein Paar, bestehend aus der Nummer der Variablen und der Schachtelungstiefe der Bindungsumgebung. Wenn die Bindung an `cons` an dritter Stelle in der globalen Bindungsumgebung steht, kann der Ausdruck weiter zu

```
([Arg 3,1] [Arg 1,0] [Arg 2,0])
```

vereinfacht werden. In diesem resultierenden Ausdruck sind keine Variablennamen mehr enthalten. Es wird lediglich auf Positionen von Lokationen innerhalb der Bindungsumgebungen zugegriffen.

Auf die Bindungsumgebung der Prozedur `f` kann nur innerhalb der Prozedur zugegriffen werden. Mit dem Verlassen der Prozedur bleibt kein Zeiger auf die Bindungsumgebung zurück. Der durch die Bindungsumgebung belegte Speicher wird von der Speicherverwaltung zu einem späteren Zeitpunkt als nicht mehr belegt identifiziert und einer erneuten Verwendung zugeführt. Der zusätzliche Aufwand durch Funktionsaufrufe für die Speicherverwaltung kann entfallen, wenn die Bindungsumgebungen für Prozeduren wie `f`, deren Bindungsumgebung nach Ende des Prozeduraufrufs nicht mehr zugreifbar ist, auf einem Stack angelegt werden. Bei der Definition einer Prozedur kann ermittelt werden, ob die Bindungsumgebung der Prozedur nach Rückkehr von der Prozedur noch zugreifbar bleiben muß [BaJe86, McDe80, Stee78]. Ein Allozieren von Bindungsumgebungen auf einem Stack verringert den Rechenzeitaufwand für die Speicherverwaltung. Nicht alle Bindungsumgebungen können auf dem Stack alloziert werden: Die Bindungsumgebung der Prozedur `make_proc` in Abbildung 2.4 auf Seite 7 bleibt, wie in Abbildung 2.5 gezeigt, nach Rückkehr von der Prozedur zugreifbar.

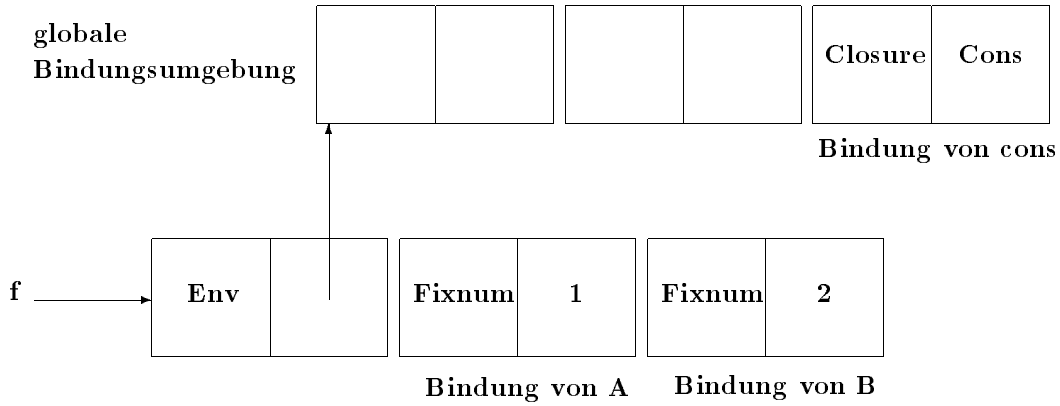



Abbildung 3.13: Repräsentation der Bindungsumgebung von `f` durch Vektoren.

Die vorausgegangenen Absätze haben Möglichkeiten aufgezeigt, inwieweit Zugriffe auf die Bindungen effizient gestaltet werden können und daß die Speicherverwaltung von den Bindungsumgebungen der Mehrzahl der Prozeduren entlastet werden kann. Die Repräsentation der Bindungsumgebungen bleibt noch zu zeigen: Die zu wählende Repräsentation ist von den Möglichkeiten der Speicherverwaltung abhängig. Wenn die Speicherverwaltung Blöcke unterschiedlicher Länge

bereitstellen kann, können Bindungsumgebungen als Vektoren der Bindungen repräsentiert werden (Abbildung 3.13). Die Bindungen können damit durch indizierten Zugriff gelesen und zugewiesen werden. Implementationen mit Speicherverwaltungen, die nur Blöcke fester Länge (nämlich CONS-Zellen) bereitstellen können, müssen Bindungsumgebungen durch Listen repräsentieren (Abbildung 3.14). Der Zugriff auf die Bindungen erfolgt dann durch Schachtelungen der Prozeduren `car` und `cdr`, wobei rechenzeitintensiv den verketteten CONS-Zellen gefolgt wird.

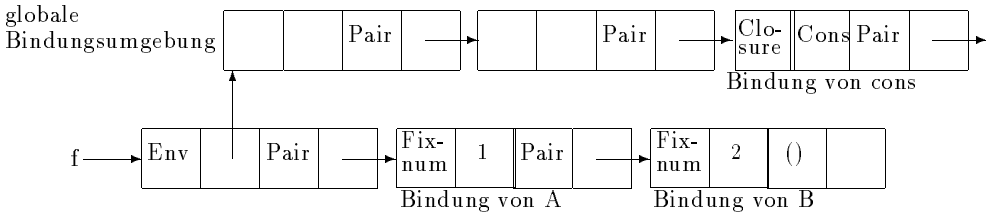



Abbildung 3.14: Repräsentation der Bindungsumgebung von `f` durch Listen.

Wenn man ein Register `env` annimmt, das einen Zeiger auf die Bindungsumgebung enthält, kann der Ausdruck

```
(cons a b)
```

bei Repräsentation der Bindungsumgebungen durch Listen durch

```
((caddar env)  (cadr env) (caddr env))
```

ersetzt werden. Bei Repräsentationen der Bindungsumgebungen durch Vektoren wird der Ausdruck durch

```
((vector-ref (vector-ref env 0)  2) (vector-ref env 1) (vector-ref env 2))
```

ersetzt. Die Repräsentation als Vektor benötigt nicht nur weniger Speicherplatz, sie ist auch effizienter im Zugriff: Die Anzahl der Speicherzugriffe ist bei `vector-ref` konstant, während sie bei `c...r` proportional zur Schachtelungstiefe ist.

#### 3.1.2.3 Controlframes

Bei der Abarbeitung eines Programms in Scheme rufen Prozeduren sich selbst oder andere Prozeduren auf. Dabei sind zwei Fälle zu unterschieden: Ein Prozeduraufruf als letzte Aktion in der Abarbeitung einer Prozedur und ein Prozeduraufruf, an den anschließend die aufrufende Prozedur fortzusetzen ist. Im folgenden Programm wird `iteriere` immer endrekursiv, und `+`, `*` und `>` werden nicht endrekursiv aufgerufen:

```
(define (fakultaet n)
   (define (iteriere produkt zaehler)
      (if (> zaehler n)
          produkt
          (iteriere (* zaehler produkt) (+ zaehler 1))))
   (iteriere 1 1))
```

Bei nicht endrekursiven Aufrufen muß der Zustand bei der Ausführung der aufrufenden Prozedur gesichert und nach der Rückkehr von der aufgerufenen Prozedur wiederhergestellt werden. Dieser Zustand wird in Controlframes abgelegt. Der Zustand besteht aus Registern, wie einem Zeiger auf den noch abzuarbeitenden Rest der aufrufenden Prozedur (Programmzähler bei Maschinencode)

und den Zeiger auf die Bindungsumgebung der aufrufenden Prozedur. Auch Zwischenergebnisse werden im Controlframe gespeichert. Im obigen Beispiel werden vor dem Aufruf von * die Prozeduren + und - aufgerufen. Die Reihenfolge dieser beiden Aufrufe ist nicht spezifiziert. Jedoch ist beim zweiten Aufruf das Ergebnis des ersten schon vorhanden, und dieses Ergebnis wird als Zwischenergebnis im Controlframe gespeichert.

Die Repräsentation der Controlframes kann, abhängig von den Möglichkeiten der Speicherverwaltung, als Vektoren oder Listen erfolgen. Der Zugriff auf die Daten in den Controlframes erfordert bei Vektoren weniger Speicherzugriffe, und Vektoren belegen weniger Speicherplatz.

Controlframes können, wie bei nahezu allen anderen Programmiersprachen, auf einem Stack angelegt werden, wenn der Ablauf eines Scheme-Programms ausschließlich aus Prozeduraufrufen besteht. Die Verwendung des Stacks entlastet, wie bei den Bindungsumgebungen beschrieben, die Speicherverwaltung.

```
(define g '())                    ⟹ g
(define (f counter)
    (write (* 3
              (call-with-current-continuation
                (lambda (conti)
                  (set! g conti)
                  1)))
     (newline)
     (set! counter (+ counter 1))
     counter)                     ⟹ f
  (f 0)                           ⟹ 1  ;  Ausgabe: 3
  (g 7)                           ⟹ 2  ;  Ausgabe: 21
  (g -2)                          ⟹ 3  ;  Ausgabe: -6
```

Abbildung 3.15: Beispiel für die unbegrenzte Lebensdauer einer Continuation.

Die Standardprozedur `call-with-current-continuation` kann unter Rückgiff auf die Controlframes implementiert werden: `Call-with-current-continuation` ruft seinen Parameter, eine einstellige Closure, auf. Bei diesem Aufruf wird, wie bei jedem nicht endrekursiven Aufruf, ein Controlframe angelegt, in den der Zustand für die Fortsetzung der Ausführung des Ausrufers nach `call-with-current-continuation` eingetragen wird. An die einstellige Closure, den Parameter von `call-with-current-continuation`, soll eine Continuation übergeben werden, die die Fortsetzung der Programmausführung nach `call-with-current-continuation` ermöglicht. Diese Continuation wird durch den Controlframe repräsentiert, in den zuvor der Zustand für diese Fortsetzung eingetragen wurde. Der Aufruf der Continuation wird durch Restaurieren des Zustandes aus diesem zuvor gesicherten Controlframe erreicht. Die Abbildung 3.16 zeigt die Ausführung des Beispielprogramms nach Ausführen von

```
(set! g conti).
```

Die Abbildung zeigt, daß die Continuation als Zeiger auf einen Controlframe implementiert wird.

In diesem Beispiel wird die Continuation an eine globale Variable gebunden, und die Continuation wird mehrfach aufgerufen. Daher können die zu einer Continuation gehörenden Controlframes nicht im Stack alloziert werden; sie sind in Scheme ein Objekt mit unbegrenzter Lebensdauer. Im Gegensatz zu Bindungsumgebungen kann aus dem Programmtext nicht ermittelt werden, welche Controlframes im Heap zu allozieren sind. Das Anlegen aller Controlframes im Heap verlangsamt die Programmausführung durch Belastung der Speicherverwaltung. Das grundsätzliche Anlegen von Controlframes in einem Stack und Kopieren aller Controlframes in den Heap, sobald

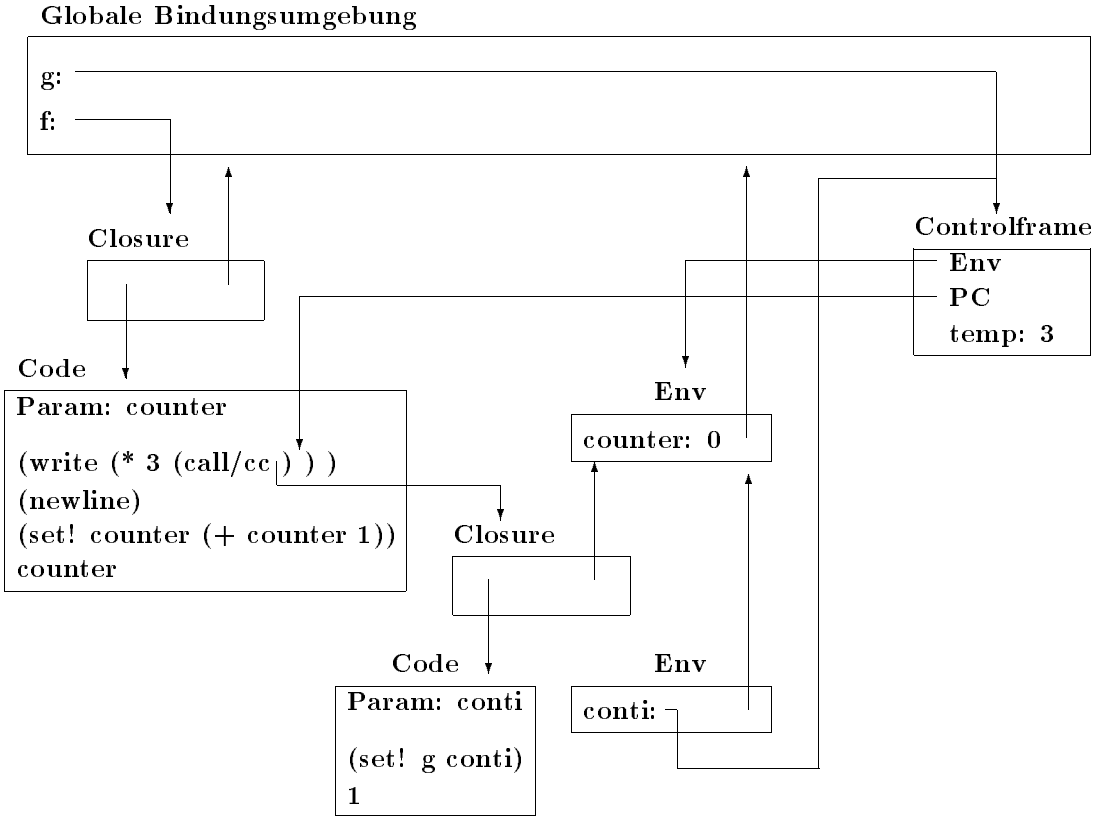



Abbildung 3.16: Implementierung einer Continuation.

`call-with-current-continuation` aufgerufen wird, erlaubt den effizienten Ablauf von Programmen, die keine Continuations verwenden. Eine Vielzahl verschiedener Stategien zur effizienten Allozierung von Controlframes werden von Clinger et al. [Clin88] beschrieben.

#### 3.1.2.4 Interne Darstellung von Programmen

Unter Programmen sind im folgenden die Ausdrücke innerhalb eines $\lambda$-Ausdrucks zu verstehen. Scheme ist eine interaktive Sprache, in der der Programmierer Ausdrücke eingibt, die sofort ausgewertet werden und Definitionen eingibt, die die globale Bindungsumgebung erweitern und insbesondere Prozeduren definieren. In Scheme werden diese Prozedurdefinitionen als Einheiten wie Programme in anderen Programmiersprachen behandelt, das heißt, sie werden zum Beispiel in Maschinencode übersetzt.

Die Auswertung eines $\lambda$-Ausdrucks ergibt eine Closure. Neben der schon beschriebenen Bindungsumgebung enthält die Closure auch einen Verweis auf beim Aufruf der Closure auszuwertende Ausdrücke. Die Darstellung der Ausdrücke eines $\lambda$-Ausdrucks ist eine wichtige Entscheidung bei der Implementierung von Scheme. Möglich ist eine große Zahl von Varianten von der Übernahme der Quelldarstellung bis hin zur Übersetzung in realen Maschinencode.

Die Übernahme der Quelldarstellung als interne Repräsentation für Programme wird bei interpretierenden Implementationen gewählt. Eine solche Implementation ist insbesondere dann sehr einfach zu erstellen, wenn sie in einem LISP-Dialekt erfolgt. In diesem Fall stehen Standardfunktionen, Daten und die Speicherverwaltung schon zur Verfügung. Es bleiben nur Bindungsumgebungen und Controlframes durch vorhandene Datenstrukturen zu repräsentieren und ein Interpreter zu erstellen. Diese Implementationen für (große Teile von) Scheme sind von so geringem Umfang, daß sie in diversen Veröffentlichungen komplett abgedruckt sind [AbSu85, Dybv87, SuSt75]. Ein gravierender Nachteil dieser besonders einfachen Implementationsart besteht darin, daß — wie allgemein bei Interpretation gegenüber Kompilation — bestimmte Aufgaben immer wieder zur Laufzeit wiederholt werden: Neben dem Parsen des Programms müssen die Bindungen von Variablen zur Laufzeit gesucht werden, und es muß vor jedem Prozeduraufruf festgestellt werden, ob Endrekursion vorliegt.

Eine weitere Klasse von Implementationen stellt Programme in Listen dar, transformiert aber die Quelldarstellung. Insbesondere werden die Variablennamen, wie bei den Bindungsumgebungen beschrieben, durch Adressen (Paare aus Level und Position) ersetzt und endrekusive durch Kennzeichnung von anderen Prozeduraufrufen unterschieden. Diese Implementationen sind weniger rechenzeitintensiv als reine Interpreter.

Die nächste Klasse von Implementationen übersetzt Programme in einen virtuellen Maschinencode. Dieser Maschinencode ist speziell auf Scheme zugeschnitten und beinhaltet Befehle für Prozeduraufrufe, Variablenzugriffe und Standardfunktionen. Im Gegensatz zur Listendarstellung stehen wie bei realem Maschinencode aufeinander auszuführende Befehle an aufeinander folgenden Speicherplätzen. Der in der Listendarstellung nötige Zeiger auf den nächsten „Befehl“ entfällt, so daß der Maschinencode kompakter als Listen ist und mit weniger Speicherzugriffen abgearbeitet werden kann. Der virtuelle Maschinencode ist häufig ein Bytecode: Ein Code, in dem jedes Byte einen Befehl oder einen Operanden enthält. Bytecode kann sehr effizient auf einer realen Maschine interpretiert werden, weil das ganze Byte als Index in eine Sprungtabelle dienen kann (sog. Threaded-Code-Technik [BaJe86]).

Ein Extremfall bei der Kompilation von Scheme ist die Übersetzung in realen Maschinecode. Die Übersetzung ermöglicht eine maximale Ablaufgeschwindigkeit der Programme. Die Übersetzung in realen Maschinencode ist zeitintensiver als die in einen Scheme-spezifischen virtuellen Code. Ferner benötigt realer Maschinencode mehr Speicherplatz als ein virtueller Code wie z.B. Bytecode.

## 3.2 Die Speicherverwaltung

Scheme vermittelt seinem Programmierer die Illusion eines unbegrenzt großen Speichers. Objekte werden mit Konstruktoren (`cons`, `make-vector`, `make-string`), mit Prozeduren, die letztlich Konstruktoren aufrufen (z. B. `read`), bei Rechnungen mit `BIGNUM`s und `FLONUM`s und auch beim Aufruf von Prozeduren für das Anlegen von Bindungsumgebungen und Controlframes erzeugt. Der Report über Scheme [Rees86] schreibt vor, daß alle Objekte eine unbegrenzte Lebensdauer haben (unlimited extent). Da alle realen Maschinen nur über einen begrenzt großen Hauptspeicher verfügen, ist eine Speicherverwaltung ein integraler Teil jeder Implementation der Sprache. Die Speicherverwaltung hat die Aufgabe, nicht mehr referenzierbare Objekte zu identifizieren und den durch diese belegten Speicherplatz für neu anzulegende Objekte wiederzuverwenden.
Speicherverwaltungen unterscheiden sich hinsichtlich mehrerer Anforderungen:

- Speicherverwaltungen können für die Verwaltung von Speicherbereichen fester oder variabler Länge ausgelegt sein.
  Speicherverwaltungen von Blöcken fester Länge können nicht mehr referenzierte Objekte einfach zu einer Freispeicherliste zusammenfassen, von der dann weitere Anforderungen von Speicher befriedigt werden. Speicherverwaltungen von Blöcken variabler Länge sind mit der Fragmentierung des Speichers konfrontiert. Um Anforderungen nach beliebig großen Blöcken befriedigen zu können, müssen sie entweder die Fragmentierung beseitigen, indem noch referenzierte Blöcke verschoben werden (Kompaktifizierung), oder sie versuchen, aneinander angrenzende, nicht mehr referenzierte Blöcke zusammenzufassen.

- Speicherverwaltungen können auf Maschinen mit realem und solche mit virtuellem Speicher zugeschnitten sein.
  Die Annahme lokaler Speicherzugriffe, für die nur ein Teil des Datenbereichs eines Prozesses jeweils im realen Hauptspeicher sein muß, führt zum Konzept eines virtuellen Speichers, in den auf Anforderung (bei Seitenfehlern) Seiten aus dem Hintergrundspeicher in den realen Hauptspeicher verlagert werden. Speicherverwaltungen müssen bei virtuellem Speicher einerseits selbst überwiegend lokale Speicherzugriffe ausführen und andererseits die noch referenzierbaren Objekte zur Ermöglichung lokaler Zugriffe umgruppieren.

- Speicherverwaltungen können unterbrechend arbeiten: Sie werden nur dann aktiviert, wenn der gesamte Speicher belegt ist. Sie unterbrechen dann das Ausführen des Scheme-Programms für eine spürbare Zeit (im Bereich von Sekunden bis Minuten) für die Ausführung einer Garbage Collection. Die Garbage Collection ist das Ermitteln aller noch zugreifbaren Objekte im Speicher, mit dem Ziel, den Rest, den durch nicht mehr zugreifbare Objekte belegten Speicher, für weitere Speicheranforderungen zur Verfügung zu stellen zu können.
  Speicherverwaltungen können auch inkrementell arbeiten: Sie verschränken dabei die Ausführung des Scheme-Programms mit der Garbage Collection. Das Scheme-Programm läuft zwar langsamer ab, es wird aber nie für längere Wartezeiten unterbrochen. Implementationen, an die Echtzeitanforderungen für bestimmte Anwendungen (z. B. Steuerungsaufgaben) oder aus ergonomischen Gründen gestellt werden, müssen eine inkrementelle Garbage Collection verwenden.

Die folgenden Abschnitte stellen drei grundsätzlich unterschiedliche Algorithmen für unterbrechende Garbage Collection vor: Referenzenzählung, Mark-and-Sweep und Stop-and-Copy. Anschließend erfolgt die Beschreibung inkrementeller Algorithmen. Dazu gehören sowohl inkrementell in die Programmausführung eingearbeitete Varianten der unterbrechenden Algorithmen als auch einige Erweiterungen, die nur bei inkrementeller Garbage Collection verwendet werden.

Der Stellenwert der Garbage Collection Algorithmen kann an der Vielzahl der vorliegenen Algorithmen gemessen werden. Die meisten Veröffentlichungen stellen ein in einer bestimmten Implementation genutztes Verfahren vor [AbSu85, BaJe86, Bake78, Broo84, Chen70, Hend80, LiHe83, Moon84, ScWa67]. Ein Überblick über verschiedene Garbage Collection Algorithmen findet sich in [Knut68], [Alle78] und [Gabr87].

### 3.2.1 Referenzenzählung

Die Referenzenzählung ist ein verbreitetes Verfahren bei der Speicherverwaltung von Objekten [Knut68]. Jedes Objekte enthält dabei ein zusätzliches Feld, das die Anzahl der Referenzen auf das Objekt angibt. Beim Einrichten und Löschen von Referenzen wird die Anzahl im Objekte jeweils angepaßt. Sobald diese Anzahl gleich *null* wird, ist bekannt, daß das Objekt nicht mehr benötigt wird und der durch dieses Objekt belegte Speicherplatz neu vergeben werden kann. Bevor dieser Speicherplatz neu vergeben werden kann, müssen alle im nicht mehr benötigten Objekt bestehenden Referenzen gelöscht werden, wobei möglicherweise weitere Objekte als nicht mehr benötigt identifiziert werden und rekursiv weitere Referenzen gelöscht werden müssen.

Neben zirkulären Datenstrukturen, bei denen auch nach Aufheben der letzten Referenz auf die Gesamtstruktur die Anzahlen in den Objekten nicht *null* werden, werden in LISP-Dialekten überwiegend Objekte geringer Größe (CONS-Zellen) verwaltet, bei denen ein zusätzliches Feld einen großen Speichermehrbedarf bedeutet. Zudem ändern sich in LISP Referenzen auf Objekte sehr häufig (bei jeder Parameterübergabe werden neue Referenzen übergeben, bei Rückkehr von einer Prozedur diese Referenzen gelöscht), so daß durch Aktualisierung der Anzahl-Felder ein bedeutender Mehrbedarf an Rechenzeit entsteht. Daher wird die Referenzenzählung bei LISP, außer bei einigen Interlisp-Dialekten [Deut80], sehr selten verwendet [Gabr87].

### 3.2.2 Mark-and-Sweep Algorithmus

Der Mark-and-Sweep Algorithmus ist der älteste Algorithmus für LISP, der schon bei der ersten LISP-Implementation von McCarthy beschrieben wurde [McCa60]. Der Mark-and-Sweep Algorithmus ist nach seinen beiden Phasen benannt.

Die erste Phase markiert alle erreichbaren Objekte. Alle Objekte enthalten ein zusätzliches Markierungsbit, das zu Beginn der Garbage Collection zurückgesetzt ist. Bei erreichten Objekten wird es gesetzt. Die Markierungsphase besteht im Durchlaufen der Baumstrukturen. Wenn dabei ein noch nicht markiertes Objekt angetroffen wird, wird es markiert und seine Unterobjekte durchlaufen. Da das Durchlaufen von Bäumen bei Programmierung in üblicher Weise einen Stack benötigt und die Garbage Collection hingegen einsetzt, wenn kein Speicher mehr frei ist, wird häufig der Algorithmus nach Schorr und Waite [ScWa67] eingesetzt, der ohne einen Stack arbeitet, dafür die Zeiger verändert und dazu wesentlich mehr Speicherzugriffe ausführt.

Die zweite Phase geht sukzessive den gesamten Speicher sequentiell durch, setzt die Markierungen markierter Objekte zurück und faßt nichtmarkierte Objekte zu einer Freispeicherliste zusammen.

Der Mark-and-Sweep Algorithmus kann, insbesondere in der Erweiterung nach Schorr und Waite, den gesamten Speicher für Objekte nutzen. Er ist besonders für Implementationen geeignet, die Blöcke fester Länge verwalten, da er nicht kompaktifiziert. Bei Systemen mit virtuellem Speicher ist neben der fortbestehenden Fragmentierung auch die in der zweiten Phase (Sweep-Phase) erfolgende Inspektion des gesamten Speichers nachteilig: Dabei werden alle Seiten vom Hintergrundspeicher sequentiell in den Hauptspeicher gelesen.

### 3.2.3 Stop-and-Copy Algorithmus

Ein in LISP-Implementationen weit verbreiteter Algorithmus ist der Stop-and-Copy Algorithmus [Chen70]. Der Stop-and-Copy Algorithmus nutzt jeweils nur eine Hälfte des zur Verfügung stehenden Speichers. Wenn in der einen Hälfte kein Speicher mehr alloziert werden kann, werden alle noch referenzierbaren Objekte an den Anfang der zweiten Hälfte kopiert und anschließend die zweite Hälfte zum Allozieren von Speicher genutzt, bis die beiden Hälften bei der nächsten Garbage Collection wieder ihre Rollen vertauschen.

Auch der in dieser Arbeit beschriebene Prozessor verwendet eine Variante des Stop-and-Copy Verfahrens; daher soll es hier genauer beschrieben werden: Für das Stop-and-Copy-Verfahren wird der Speicher in zwei Hälften aufgeteilt und jeweils immer nur in einer der Hälften Speicher vergeben. Die Speicherverwaltung verwaltet dazu einen Zeiger **free** auf den Anfang des freien Bereichs. Wenn Speicher angefordert wird, wird **free** um die benötigte Blockgröße erhöht und **free** vor der Erhöhung als Anfangsadresse des zugeteilten Speichers zugewiesen. Der Stop-and-Copy Algorithmus kann daher Blöcke unterschiedlicher Länge liefern.

Wenn beim Stop-and-Copy Algorithmus mehr Speicher angefordert wird, als noch in der zuzuteilenden Hälfte des Speichers frei ist, erfolgt die Garbage Collection: Das neue Register **scan** und **free** werden mit der Anfangsadresse der nicht genutzten Speicherhälfte geladen. Die von Registern und Stack zugegriffenen Objekte werden reloziert: Wenn Register oder Verweise im Stack auf einen Vorwärtsverweis zeigen, wird die neue Adresse eingetragen, sonst werden die Objekte nach **free** kopiert, die Adresse von **free** in Register und Stack eingetragen, an die alte Adresse des Objekts ein Vorwärtsverweis auf die neue Adresse geschrieben und **free** um die Größe des Objekts erhöht. Anschließend wird **scan** solange reloziert und erhöht, bis **scan** und **free** auf die gleiche Adresse weisen.

Der Stop-and-Copy Algorithmus kompaktifiziert den benutzten Speicher am Anfang der Hälfte. Deswegen wird er besonders häufig für Rechner mit virtuellem Speicher eingesetzt. Ferner inspiziert der Algorithmus nur alle kopierten Objekte und nicht den gesamten Speicher, eine deutliche Verbesserung gegenüber dem Mark-and-Sweep Algorithmus.

Ein Nachteil des Stop-and-Copy Algorithmus gegenüber anderen Verfahren ist die jeweilige Nutzung nur der Hälfte des Speichers. Daher werden Varianten eingesetzt, bei denen die benutzten Objekte auf einen Hintergrundspeicher kopiert werden, (**free** und **scan** sind dann während der Garbage Collection Positionen in einer Plattendatei) dort reloziert werden und anschließend in den Hauptspeicher zurückgeschieben werden.

### 3.2.4 Inkrementelle Garbage Collection

Die Referenzenzählung ist ein inkrementelles Verfahren, weil es die Operationen für die Speicherverwaltung auf den normalen Programmablauf, nämlich beim Erzeugen und Löschen von Referenzen, verteilt. Es wurde zwar bei einer experimentellen LISP-Maschine von Xerox implementiert [Deut80], ist bei LISP aus den bei der Beschreibung der Referenzenzählung erwähnten Gründen jedoch nicht verbreitet.

Ein verbreiteter inkrementeller Algorithmus ist eine von Baker [Bake78] beschriebene und in der M.I.T. LISP-Maschine [Gree77] eingesetzte inkrementelle Version des Stop-and-Copy Algorithmus: Während aller Speicheranforderungen werden immer einige Objekte reloziert. Während des normalen Abarbeitens von Programmen muß dann nach Zugriff auf ein Objekt immer auf eine Vorwärtsreferenz abgefragt werden und dieser gegebenenfalls gefolgt werden. Wegen dieser zahlreichen Abfragen wird inkrementelle Garbage Collection bei LISP nur in Hardwareimplementationen eingesetzt.

Neuere inkrementelle Garbage Collection Algorithmen machen sich Untersuchungsergebnisse

über die Lebensdauer der Objekte in typischen LISP-Programmen zu nutze: Neu allozierte Objekte sind mit hoher Wahrscheinlichkeit schon bei der nächsten Garbage Collection nicht mehr zugreifbar. Objekte, die schon mehrere Garbage Collections überlebt haben, werden mit hoher Wahrscheinlichkeit auch noch weitere Garbage Collections überleben. Als Konsequenz aus diesen Untersuchungen wird der Speicher in verschiedene Blöcke eingeteilt, in die Objekte unterschiedlicher Generationen eingetragen werden. Ein Objekt gelangt in die nächste Generation, wenn es eine generationsspezifische Anzahl von Garbage Collections überlebt hat. Die LISP-Implementation führt Garbage Collections für Blöcke mit kurzlebigen Objekten wesentlich häufiger aus als für Blöcke mit längerlebigen Objekten und vermeidet nicht nur die zeitintensive Garbage Collection für Speicherbereiche, in denen kaum freier Speicher zu erlangen ist. Vielmehr wird die Garbage Collection auf einen lokalen Speicherbereich beschränkt und damit bei virtuellem Speicher eine wesentliche Beschleunigung erzielt. Dieses Verfahren wird von Symbolics verwendet [Moon84]. Es nutzt für die Garbage Collection in den einzelnen Blöcken den inkrementellen Stop-and-Copy Algorithmus. Zusätzlicher Aufwand entsteht dabei durch Referenzen von Objekten aus einem Block in einen anderen. Solche Referenzen erfolgen über Tabellen. Zudem müssen Objekte ein Feld für die Anzahl der Garbage Collections, die sie schon überlebt haben, enthalten, damit feststellbar ist, wann sie in die nächste Generation übergehen.

# Kapitel 4

# Wichtige Implementationen von Scheme

Dieses Kapitel soll typische Implementationen von Scheme mit ihren Vor- und Nachteilen vorgestellen. Im nächsten Kapitel wird dann eine neue Implementation konzipiert, die Schwächen der bestehenden Implementationen durch ein neues Konzept überwindet.

Eine Einteilung der existierenen Implementationen kann nach unterschiedlichen Kriterien erfolgen: nach den Rechnerklassen auf denen sie laufen (Mikrorechner, Workstations, Großrechner, Spezialhardware), nach dem Einsatzbereich (Ausbildung, Programmentwicklung, Einsatz beim Anwender) und nach bei der Implementierung gewählte Alternativen (interpretierende oder compilierende Implementation, mit oder ohne Unterstützung virtuellen Speichers, unterbrechende oder inkrementelle Garbage Collection).

Hier sollen die Implementationen nach den bei der Implementierung gewählten Alternativen für die Repräsentation der Programme, entspechend der Einteilung in Kapitel 3.1.2.4, dargestellt werden. Diese Alternativen beeinflussen insbesondere den Rechenzeit- und Speicherbedarf, zwei aussagekräftige Vergleichskategorien für Algorithmen und Programme.

Allen hier vorgestellten Implementationen gemeinsam ist die Objektrepräsentation durch Typen im Zeiger (tagged pointer). Auch werden Listen immer aus CONS-Zellen gebildet; keine Scheme-Implementation verwendet CDR-Kodierung. Die Speicherverwaltungen arbeiten alle unterbrechend. Sie unterscheiden sich in den Algorithmen (Stop-and-Copy oder Mark-and-Sweep) und in der Möglichkeit, Objekte variabler Länge zu unterstützen.

## 4.1 Interpretation der Quelldarstellung

Die erste Implementation von Scheme war ein in LISP erstellter Interpreter. Der Quellcode in MacLISP ist im Anhang von [SuSt75], der ersten Veröffentlichung über Scheme, enthalten. Da LISP eine Programmiersprache ist, die „rapid prototyping“ und das Parsen von in Listenstruktur dargestellten Ausdrücken besonders unterstützt, ist diese Art der Implementation war sehr einfach zu bewerkstelligen. Das darunterliegende LISP stellt die Daten, die Standardprozeduren und die Speicherverwaltung bereit. Ein Nachteil besteht in dem großen Speicherbedarf, weil das LISP-System, das nur zum Teil genutzt wurde, immer vorhanden war. Ferner ist die Ablaufgeschwindigkeit der Programme gering.

## 4.2 Interpretation von S-Code

Interpretierende Implementationen stellen das abzuarbeitende Programm als Listenstruktur dar. Transformationen des Programms in eine Interndarstellung des Interpreters ermöglichen ein effizienteres Abarbeiten als die Übernahme der Quelldarstellung es erlaubt. Diese Darstellung wird — zumindest beim MIT — *S-Code* genannt. In S-Code werden Variablennamen durch lexikalische Adressen ersetzt, werden endrekursive Prozeduraufrufe gekennzeichnet, und es werden weitere Typen für die Bestandteile der Kernsprache eingeführt, damit der Interpreter nicht das erste Symbol einer Liste suchen und anhand dessen verzweigen muß. Bei S-Code besteht die Grundroutine des Interpreters aus einer Tabelle, über die abhängig von dem Typ der S-Code-Listen verzweigt wird. Diese Typen spielen also die Rolle von Opcodes in Maschinensprachen. Im Gegensatz zu Maschinensprachen bleibt das S-Code Programm eine listenförmige Darstellung des Parsebaumes. S-Code wurde am MIT bei Softwareimplementationen (Chipmunk Scheme, C-Scheme) und Hardwareimplementationen (Scheme-79, Scheme-81) eingesetzt. Die Hardwareimplementation Scheme-86 [BeWu88a], die nur simuliert, aber nicht fertiggestellt wurde, wird in Kapitel 9.4.1 beschrieben.

### 4.2.1 MIT C-Scheme

MIT C-Scheme [CSch84] ist eine vollständige, portable Implementation von Scheme, die im Gegensatz zu allen anderen hier beschriebenen Implementationen auch komplexe und rationale Zahlen bereitstellt. Sie löste beim MIT das Chipmunk Scheme, eine auf inzwischen veralteten Rechnern 9836 von Hewlett Packard (68000 mit 12,5 MHz, 4 MByte Hauptspeicher, kein virtueller Speicher) in 68000-Assembler erstellte Implementation, ab.

C-Scheme ist ein sehr umfangreiches Programm mit sehr großem Speicherbedarf, das nur auf 32 Bit Rechnern mit virtuellem Speicher unter Unix und VMS eingesetzt wird. C-Scheme unterstützt Objekte mit variabler Länge und verwendet den Stop-and-Copy Algorithmus für die Garbage Collection.

### 4.2.2 Scheme-79

Scheme-79 ist der erste Schemeprozessor, der gefertigt und getestet wurde.

Scheme-79 besteht aus zehn Registern, Inkrementer, Dekrementer und einem Komparator, der nur auf Gleichheit prüfen kann sowie aus je einem Mikroprogramm- und Nanoprogramm-PLA. Über diese Vergleiche, das Inkrementieren und Dekrementieren hinausgehende Operationen sind nicht möglich. Insbesondere beherrscht der Prozessor keine weitere Arithmetik.

Die einzige Speicherstruktur sind aus 2 Maschinenworten bestehende CONS-Zellen. Die Programme sind als Listenstruktur in S-Code mit Typen im Pointer dargestellt, und auch der Stack wird aus CONS-Zellen im Heap alloziert.

Die Maschinenworte sind 32 Bit breit, wovon 24 Bit für die Pointer auf CONS-Zellen oder für Integer, 7 Bit für den Typ und 1 Bit für die Speicherverwaltung genutzt werden (Abbildung 4.1).

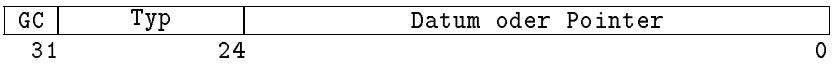



Abbildung 4.1: Das Wordformat von Scheme-79.

Das Chip implementiert die in Tabelle 4.1 angegebenen Standardprozeduren.

| | | |
|---|---|---|
| cons | −1+ | null? |
| car | 1+ | pair? |
| cdr | null? | number? |
| set-car! | zero? | |
| set-cdr! | eq? | |

Tabelle 4.1: Durch Scheme-79 unterstützte Standardprozeduren.

Scheme-79 kann Interrupts bearbeiten und selbst auslösen. Die Speicherverwaltung, die Abarbeitung von S-Code und die Bearbeitung von Interrupts sind im Mikroprogramm untergebracht. Unterprogramme im Mikroprogramm sind nicht vorgesehen; daher realisiert ein Nanoprogramm häufig wiederkehrende Abläufe und steuert Hauptspeicherzugriffe. Die Garbage Collection erfolgt nach dem Mark-and-Sweep Algorithmus modifiziert nach Schorr und Waite [ScWa67], um Listen durchlaufen zu können, ohne dazu einen Stack zu benötigen.

Scheme-79 wurde in kurzer Zeit – [Holl80] berichtet von fünf Wochen – entworfen. Die Register und Komparatoren wurden dabei von Hand entworfen und die PLAs von einem PLA-Generator erzeugt. In weiteren 5 Monaten wurden noch Fehler berichtigt, bis die Fertigung 1979 / 1980 mit einem NMOS Prozeß unter Verwendung der Entwurfsregeln von Mead und Conway [MeCo80] mit $\lambda$[1]$= 1,25\mu$ erfolgte. Bis auf zwei Entwurfsfehler – einer davon in der Garbage Collection – funktionierte einer von vier gefertigten Chips und hat Testprogramme ausführen können [Holl80]. In [Holl80] wird die Taktfrequenz mit 625 KHz angegeben, dabei aber erwähnt, daß dies noch nicht die maximale Taktfrequenz des 45 mm$^2$ großen Chips ist.

```
(define (+ x y)              ; + muß auf -1+ und 1+ zurückgeführt werden
  (if (zero? x)
      y
      (+ (-1+ x) (1+ y))))
(define (fib x)
  (cond ((zero? x) 0)        ; fib(0) = 0
        ((zero (-1+ x)) 1)   ; fib(1) = 1
        (else (+ (fib (-1+ x)) (fib (-1+ (-1+ x)))))))
(fib 20)
```

Abbildung 4.2: Mit Scheme-79 gerechnetes Testprogramm.

In Abbildung 4.2 ist das Testprogramm gezeigt, mit dem die Entwerfer Scheme-79 und einen MacLISP-Interpreter auf einem DEC KA-10 Prozessor verglichen haben. Scheme-79 erhielt dazu 256 KByte Hauptspeicher und benötigte bei leerem Speicher 1 Minute und bei halber Füllung 3 Minuten. Der MacLISP Interpreter rechnete 3,6 Minuten, und ein vom MacLISP Interpreter ausgeführter Scheme Interpreter benötigte 9 Minuten [Suss81]. Resultate für compiliertes MacLISP und ohne Definition für +, da der KA-10 Prozessor im Gegensatz zu Scheme-79 Additionen beherrscht, sind nicht veröffentlicht.

[1] Der Parameter $\lambda$ ist ein Maß für die Güte bei der IC-Fertigung: Die nacheinander auf die Siliziumscheibe gebrachten Strukturen sind um weniger als $\lambda$ gegeneinander versetzt. Abhängig von diesem Parameter $\lambda$ werden die Größen und Abstände der Strukturen gewählt. Beispielsweise werden Metalleitungen minimal $3\lambda$ breit und mit mindestens $3\lambda$ Abstand gegenüber anderen Metalleitungen verlegt [MeCo80]. Die Länge eines Gatters wird meistens als Prozeßmaß angegeben; sie beträgt $2\lambda$. Scheme-79 wurde daher mit einem $2{,}5\mu$-Prozeß gefertigt.

Ein Einsatz von Scheme-79 ist kaum sinnvoll, da bis auf Inkrementieren und Dekrementieren keine Arithmetik möglich ist. Addition und Subtraktion können zwar prinzipiell auf die Addition und Subtraktion von 1 zurückgeführt werden, aber ohne ein Prädikat, das negative Zahlen von positiven unterscheidet, kann Addition und Subtraktion nur für positive Zahlen realisiert werden. Zudem ist natürlich eine Addition, die auf die Addition von 1 zurückgeführt wird, für praktisch alle Anwendungen zu ineffizient.

Ferner belegt der Stack, der als Liste aus CONS-Zellen implementiert ist, doppelt so viel Speicher wie er benötigte, wenn als kontinuierlicher Speicherbereich realisiert worden wäre. Noch gravierender ist vermutlich der zusätzliche Aufwand für die Garbage Collection, der durch diese Realisierung des Stacks entsteht. Zwar erlaubt diese Implementation des Stacks eine problemlose Realisierung von Continuations, aber die meisten Programme werden gerade dadurch und aufgrund der daraus folgenden häufigen Garbage Collections wesentlich langsamer. Tatsächlich ergeben Untersuchungen der Entwerfer, daß Scheme-79 80% seiner Zeit mit Garbage Collection verbringt [Holl80].

Außerdem sind Bindungsumgebungen — wie alle Daten — als Listen dargestellt. Daher muß zum Auffinden oder Ändern der Bindung einer Variable immer mit den Listenoperationen zur gesuchten Stelle vorgedrungen werden. Das effizientere Indizieren einer als kontinuierlicher Speicherbereich repräsentierten Bindungsumgebung ist nicht möglich.

### 4.2.3 Scheme-81

Scheme-81 [Bata82] ist die zweite bekannte Hardwareimplementation von Scheme. Auch Scheme-81 interpretiert S-Code und kann Interrupts bearbeiten. Scheme-81 implementiert ein Busprotokoll, das es verschiedenen Prozessoren erlaubt, unterschiedliche Standardprozeduren zu implementieren und diese allen beteiligten Prozessoren zur Verfügung zu stellen. Scheme-81 hat in diesem Konzept speziell die Aufgabe, den S-Code zu interpretieren und den anderen Prozessoren auf Anforderung Speicher zuzuweisen. Jeder dieser maximal 31 Prozessoren verfügt über maximal 16 Millionen CONS-Zellen Hauptspeicher, und Objekte im Speicher eines Prozessors können auch in den Speicher eines anderen Prozessors verweisen.

Die einzige implementierte Speicherstruktur sind aus 2 Maschinenworten bestehende CONS-Zellen. Die Programme sind — wie bei Scheme-79 — als Listenstruktur in S-Code mit Typen im Pointer dargestellt, und auch die Stacks werden aus CONS-Zellen im Heap alloziert. Zusätzlich waren Vektoren als Speicherstruktur zwar geplant, konnten aber aus Platzgründen nicht realisiert werden [Bata82].

Die Maschinenworte sind 36 Bit breit, wovon 29 Bit für die Pointer auf CONS-Zellen oder für Integer, 7 Bit für den Typ genutzt werden (Abbildung 4.3).

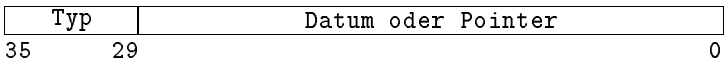



Abbildung 4.3: Das Wordformat von Scheme-81.

Von 29 Bit breiten Adressen werden die obersten 5 Bit zur Adressierung des Prozessors und die unteren 24 Bit zur Adressierung einer CONS-Zelle im Speicher dieses Prozessors verwendet.

Die Struktur von Scheme-81 ist in [Bata82] leider nicht sehr genau beschrieben: Daten wie Anzahl der Register, welche Operationen ausgeführt werden können sowie wieviele und welche Standardprozeduren des S-Codes interpretiert werden, sind nicht veröffentlicht.

Die Register von Scheme-81 sind in Teilfelder aufgespalten und mit Operationseinheiten wie Shifter und Inkrementer vereinigt. Daher können viele Operationen gleichzeitig auf unterschiedlichen Registern und Teilfeldern der Register ausfgeführt werden. Diese Parallelität bedingte ein horizontales Mikroprogrammwort von über 200 Bit Breite, das dann durch Einteilung in Felder und Hinzutreten von Dekoder-PLAs bei Beibehaltung der Parallelität schmaler wurde. Die Register sind als „Racks“ [StSu80b], ein Kunstwort aus „Register“ und „Stack“, ausgeführt: Jedem Register wird ein eigener Stack und ein Zustandsbit zugeordnet. Das Zustandsbit speichert, ob ein Register gesichert oder restauriert werden soll und führt diese Operation nur dann aus, wenn der Registerinhalt wirklich verändert oder gelesen wird. Dadurch werden überflüssige Stackoperationen zur Laufzeit optimiert.

Das Mikroprogramm realisiert den S-Code Interpreter und die Speicherverwaltung nach dem Stop-and-Copy Algorithmus sowie den Bus-Controller für die Vernetzung mehrerer Prozessoren und wurde durch Einführung von Unterprogrammen auf 374 Mikroinstruktionen reduziert, die in einem PLA untergebracht sind. Als Größe des Chips wird bei einem $2,0\,\mu$-Prozeß eine Fläche von 85 mm$^2$ angegeben.

Stacks und Bindungsumgebungen werden wie in Scheme-79 aus CONS-Zellen aufgebaut. Die Garbage-Collection verwendet den Stop-and-Copy-Algorithmus, wobei alle vernetzten Prozessoren eine Garbage-Collection durchführen müssen, sobald einem der Prozessoren der Speicherplatz ausgeht. Ob Scheme-81 eine verbesserte Integerarithmetik aufweist, geht aus [Bata82] leider nicht hervor. Ferner sind keine Berichte vom Test oder Einsatz von Scheme-81 veröffentlicht.

Eine Anfrage des Verfassers bei Gerald J. Sussman ergab, daß Scheme-81 nach einer Fertigung nicht weiterverfolgt wurde. Nach der Fertigung des Chips wurden Fehler gefunden. Der gefertigte Scheme-81 hätte einfache Programme etwa so schnell wie compiliertes MacLISP auf dem Prozessor KL-10 ausgeführt [Suss88]. Die Frage nach einer Aufzählung der implementierten Standardprozeduren wurde nicht beantwortet.
Ein Fazit der Scheme-Prozessoren-Entwürfe am MIT veröffentlicht Wu in einem neueren MIT-Memo [Wu88]:

> „In 1979 and 1982 two attempts were made to implement an SCode interpreter in microcode on a microprocessor.
> ...
> A complete Scheme system never ran on any of them and extensive performance testing was not completed.“ [Wu88]

## 4.3 Übersetzung in virtuellen Maschinencode

Die Übersetzung in virtuellen Maschinencode erlaubt Implementationen auf Mikrorechnern. Der virtuelle Maschinencode bewirkt eine sehr kompakte Repräsentation von Programmen und kommt damit dem begrenzten Hauptspeicher von Mikrorechnern entgegen. Die Implementation umfaßt einen Compiler, der Ausdrücke von Scheme in den virtuellen Maschinencode übersetzt. Als virtueller Maschinencode wird ein Bytecode gewählt, in dem Befehle ein Byte und mögliche Argumente auch Bytes umfassen. Diese Darstellung ist sowohl kompakt als auch effizient durch Programme auf Mikrorechnern interpretierbar. Ferner werden die Bytecodebefehle auf Scheme zugeschnitten. Aufrufe von Standardprozeduren können in einen einzigen Befehl übersetzt werden, der dann Typprüfungen und generische Operationen umfaßt. Andere Befehle erlauben den Zugriff auf Argumente und den Aufruf von Prozeduren. Die Orientierung des Befehlssatzes an der Sprache trägt auch wesentlich zur Kompaktheit der repräsentierten Programme bei. Schooler und Stamos berichten von Untersuchungen von Chris Hanson am MIT, nach denen Programme in Bytecode um den Faktor drei weniger Speicherplatz als in S-Code belegen [ScSt84].

Implementationen, denen ein Bytecode zugrundeliegt, sind PC Scheme [BaJe86], MacScheme [Sem87], XScheme [Betz89] und eine in [ScSt84] vorgeschlagene Macintosh-Implementation, die allerdings nie realisiert wurde [BaJe86]. Daß alle diese Implementationen für Mikrorechner konzipiert sind, unterstreicht die Bedeutung der Programmrepräsentation durch Bytecode für diese Rechnerklasse. XScheme ist eine frei kopierbare, vollständig in C erstellte Implementation, die noch sehr fehlerbehaftet ist und besonders auf Rechnern betrieben wird, für die keine anderen Implementationen erhältlich sind. Die beiden anderen Implementationen sollen genauer vorgestellt werden.

### 4.3.1 PC Scheme

Die bekannteste — und nach Meinung Vieler die beste (z. B. [Shep87]) — Implementation für Mikrorechner ist PC Scheme der Firma Texas Instruments. Es ist eine auf IBM PCs und ATs unter MS-DOS ablauffähige Implementation, bei der Schemeprogramme in einen registerorientierten Bytecode übersetzt werden. Der registerorientierte Bytecode wird von einem in Assembler und in C geschriebenen Programm interpretiert.

Neben dem Compiler, der mit Closure- und Environmentanalysen und weiteren Optimierungen einen gelungenen Kompromiß zwischen Übersetzungszeit und Ausführungszeit des erzeugten Codes darstellt, tragen der enthaltene Editor `Edwin` und die Erweiterung für objektorientierte Programmierung `SCOOPS` viel zur Beliebtheit von PC Scheme bei.

PC Scheme verwendet Typrepräsentation im Pointer mit dem BIBOP-Verfahren. Es unterstützt Stack und Heap, wobei im Heap eine Garbage Collection nach dem Mark-and-Sweep Algorithmus mit anschließender Kompaktifizierungsphase erfolgt. Continuations werden nur bei Bedarf und bei Stacküberlauf in den Heap kopiert [BaJe86].

Die Grenzen von PC Scheme sind durch den begrenzten Hauptspeicher gegeben. PC Scheme ist besonders für das Entwickeln kleinerer Programme geeignet. Die vom 80286-Prozessor adressierbaren 640 KByte sind sehr schnell erreicht. Ab Version 3.00 werden zwar bis zu 2,5 MByte unterstützt, aber nur um den Preis einer deutlich geringeren Ablaufgeschwindigkeit (Faktor 2 bis 3). Eine Version, die bis zu 4 MByte Speicher in Protected Mode unterstützt, wird in Produkten von TI eingesetzt, aber nicht separat vertrieben.

### 4.3.2 MacScheme

MacScheme ist eine Implementation auf dem Apple Macintosh. Sie ist in die Bedienungsoberfläche des Macintosh integriert und verwendet zum Beispiel deren Editor. Die ursprüngliche Implementation von 1985 übersetzt Programme in einen stackorientierten Bytecode. Benchmarks laufen um den Faktor 2 bis 3 langsamer auf einem Macintosh mit 512 KByte Hauptspeicher als in PC Scheme auf einem IBM AT [BaJe86]. Die Autoren von PC Scheme führen diesen Unterschied auf die Überlegenheit ihres registerorientierten Bytecodes zurück.

Über Interna von MacScheme ist sehr wenig veröffentlicht. Aus [Clin88] geht jedoch hervor, daß die Version von 1985 die Verwaltung von Speicherbereichen unterschiedlicher Länge unterstützt und nur einen Heap verwendet. Durch das Anlegen aller Bindungsumgebungen und Controlframes im Heap und die dadurch bewirkte Belastung für die Speicherverwaltung sind die Benchmarkergebnisse wahrscheinlich besser erklärbar als durch Unterschiede im Modell der Bytecodemaschine.

Die aktuelle Version von MacScheme [Sem87] unterstützt sowohl Stack als auch Heap und erlaubt die Übersetzung von Prozeduren wahlweise in Bytecode und in realen Maschinencode. Sie beeinhaltet umfangreiche Unterstützungen für die Nutzung der Macintosh Benutzeroberfläche durch Scheme-Programme. Die Benchmarks laufen etwa Faktor zwei schneller als in PC Scheme, wenn sie in realen Maschinencode übersetzt werden [Clin88].

## 4.4 Übersetzung in realen Maschinencode

Eine Geschwindigkeitssteigerung der Programme gegenüber Bytecode-basierten Implementationen kann durch Übersetzung in realen Maschinencode erzielt werden. Die Übersetzung in realen Maschinencode ist mit zwei Problemen verbunden: Realer Maschinencode benötigt wesentlich mehr Speicherplatz als Bytecode. MacScheme unterstützt in der Version 1.5 sowohl Bytecode als auch 68000-Code. Das Handbuch [Sem87] gibt an, daß realer Maschinencode um den Faktor fünf platzaufwendiger ist (Seite I-30 in [Sem87]). Ferner sind Programme in realem Maschinencode schwerer auszutesten als Programme in Bytecode oder Listendarstellung [Sem87].

Compiler, die echten Maschinencode erzeugen, sind Liar [Roza84] und Orbit [Kran86, Kran88]. Während Liar erst 1989 in die $\beta$-Testphase kam, ist Orbit seit 1986 im T-System, einem Scheme-ähnlichen LISP-Dialekt der Yale-Universität, im Einsatz.

Alle drei Compiler erzeugen, ebenso wie ein in [Feel86] beschriebener Compiler, 68000-Maschinencode. Wegen des Platzbedarfs des erzeugten Codes werden sie auf Workstations eingesetzt, die gegenüber Mikrorechnern über einen großen Hauptspeicher und fast immer sogar virtuellen Speicher verfügen. Beispielsweise wird für MacScheme bei der Übersetzung in realen Maschinencode ein Macintosh II mit 5 MByte Hauptspeicher empfohlen [Sem87].

Im erzeugten Maschinencode wird auf Typenprüfungen verzichtet (vergl. Beispiele für den erzeugten Code in [Kran86]), da nur so ein großer Geschwindigkeitsgewinn gegenüber interpretiertem virtuellen Code erreichbar ist. Wegen des Verzichts auf Typenprüfungen sind solche Implementationen für Lehre und Programmentwicklung weniger geeignet als für das Erzielen einer maximalen Ablaufgeschwindigkeit von ausgetesteten Programmen.

# Kapitel 5

# Konzeption einer neuen Implementation

In diesem Kapitel soll ein Konzept für eine neue Implementation der Sprache Scheme dargelegt werden, dem in den folgenden Kapiteln die Darstellung der Realisierung des Konzeptes folgt. Das Konzept berücksichtigt die Erfahrungen mit den bestehenden Implementationen und gewährleistet auch die Realisierbarkeit unter restriktiven personellen und finanziellen Randbedingungen.

## 5.1 Folgerungen aus den bestehenden Implementationen

Das vorangegangene Kapitel bot einen Überblick über bestehende Implementationen von Scheme. Es existieren sowohl interpretierende als auch compilierende Implementationen. Die interpretierenden Implementationen sind in der Ausführungsgeschwindigkeit der Programme erwartungsgemäß den compilierenden Implementationen unterlegen. Als Vorteil der interpretierenden Implementationen wird die Nähe des repräsentierten Programms zur Quelldarstellung angeführt, die beim Austesten der Programme besonders vorteilhaft sei. Tatsächlich setzten sich compilierende Implementationen durch, und diese beinhalten das Debuggen unterstützende Übersetzungsmodi [BaJe86, Sem87]. Die Entwicklung von Compiler für Scheme ist vom Forschungsstadium (etwa Rabbit [Stee78]) zu praktisch eingesetzten Compilern auf Mikrorechnern [BaJe86, Sem87] und auf Workstations [Kran86, Kran88] gediehen.

Die Darstellung der Objekte ist ein zentrales Kriterium für die Effizienz einer Implementation. Die Darstellung der Prozeduren, die in Scheme auch Objekte sind, ist zentral für Ausführungsgeschwindigkeit und Speicherbedarf der Programme.

Auch die Speicherverwaltung beeinflußt die Effizienz einer Implementation. Eine Speicherverwaltung, die Objekte variabler Länge unterstützt, ist zwar aufwendiger als eine, die nur Objekte fester Länge verwalten kann. Die Unterstützung Objekte variabler Länge ist aber für eine effiziente Objektrepräsentation unverzichtbar.

## 5.2 Hardware-unterstützte Implementationen von LISP

Neben den Scheme-Chips (Scheme-79 und Scheme-81) vom MIT bestehen Erfahrungen mit LISP-Maschinen als Forschungsgegenstand und kommerzielles Produkt. Ein Überblick über LISP-Maschinen findet sich in [PlTh87]. Pleszkun und Thazhuthaveetil sehen dort drei Aufgaben bei der Implementierung von LISP-Dialekten, die durch Hardware zu beschleunigen sind:

1. Effizienter Prozeduraufruf.
2. Manipulation von Bindungsumgebungen und Controlframes.
3. Zugriff auf Listen und Repräsentation derselben.
4. Speicherverwaltung.

Die Entwicklung der heutigen LISP-Maschinen begann 1977 mit dem LISP-Maschinen-Projekt am MIT [Gree77], das durch Berichte über mit mikroprogrammierten Rechnern [Deut80] bei Xerox PARC gesammelte Erfahrungen initiiert wurde [PlTh87]. Die MIT-LISP-Maschine ist ein mikroprogrammierter 32-Rechner, dessen Architektur und Befehlssatz auf LISP zugeschnitten sind. LISP-Funktionen konnten interpretiert, in die Maschinensprache der LISP-Maschine compiliert und sogar in Mikroprogramme der LISP-Maschine übersetzt werden. Die MIT-LISP-Maschine ist als Arbeitsplatzrechner (Workstation) konzipiert; nur ein Benutzer kann sie zu einer Zeit verwenden. Sie implementiert auch erstmals die CDR-Kodierung [Hans69] zur Reduktion des Speicherbedarfs von Listen. Die MIT-LISP-Maschine war der Ausgangspunkt für kommerzielle Entwicklungen: Die Maschinen Symbolics 3600, LMI Lambda und TI Explorer sind alle als Arbeitsplatzrechner konzipiert und verfolgen unterschiedliche Aspekte der MIT-LISP-Maschine weiter. Symbolics hat das Wortformat auf 36 Bit Breite erweitert und kann damit 32-Bit Zahlen effizient repräsentieren und bietet virtuellen Speicher [Moon84, Moon85]. Die Maschine LMI Lambda ermöglicht den Anwendern das Übersetzen von LISP-Funktionen in Mikroprogramme [Gabr87].

Die Klasse der Arbeitsplatzrechner ist nur eine der untersuchten Möglichkeiten für eine hardware-unterstützte LISP-Implementierung. Andere Verfahren nutzen Multiprozessorarchitekturen, die entweder unterschiedliche Prozessoren für verschiedene Aufgaben enthalten oder eine Vielzahl gleicher Prozessoren für eine parallele Abarbeitung von LISP-Programmen einsetzen. Diese Systeme befinden sich, im Gegensatz zu LISP-Maschinen, noch im Forschungsstadium und werden nicht kommerziell eingesetzt [PlTh87].

Die verbreiteten LISP-Maschinen sind mikroprogrammiert und haben einen LISP-spezifischen Maschinenbefehlssatz. Die stürmische Entwicklung der RISC-Prozessoren hat in neuester Zeit auch Prozessoren für LISP hervorgebracht. Basierend auf der Entwicklung des RISC-Prozessors SOAR (Smalltalk on a RISC) für Smalltalk in Berkeley durch Ungar [Pend86] wurde in Berkeley von Kong der LISP-Prozessor SPUR (Symbolic Processing using RISC) entwickelt [Kong89]. Ferner entstand mit MIPS-X an der Stanford University ein LISP-Prozessor in der RISC-Philosophie [StHe88]. Beide Prozessoren, SPUR und MIPS-X, sind noch Prototypen. Berichte vom praktischen Einsatz oder von der Fertigstellung einer kompletten Workstation mit LISP-Umgebung liegen bis dato nicht vor.

Für eine Unterstützung der Implementierung durch Hardware müssen besonders kritische Operationen identifiziert werden. Tabelle 5.1 zeigt die Verteilung der CPU-Zeit bei der Abarbeitung von LISP-Programmen nach in [StHe88] veröffentlichten Untersuchungen der Stanford University. Diese Werte berücksichtigen den CPU-Zeitbedarf für die Speicherverwaltung nicht mit. Sie basieren auf zehn Testprogrammen mit zusammen ca. 11500 Zeilen Quellcode [StHe88].

## 5.3 Eine neue Scheme-Implementation

Insbesondere da Scheme immer stärker in der Lehre eingesetzt wird, ergibt sich der Wunsch nach einer neuen Implementation, die auf preiswerten Rechnern abläuft. Ein Einsatz von Scheme auf herkömmlichen Timesharing-Rechnern ist aufgrund des ihres großen benötigten Prozeßadreßraums und der geringen Lokalität der Zugriffe im virtuellen Speicher zu verwerfen. Das Ziel ist daher, die verbliebenen Nachteile der schon sehr guten Implementationen auf Mikrorechnern (PC Scheme

| | |
|---|---|
| Funktionsaufrufe | 20 % |
| CAR / CDR | 18 % |
| Zugriffe auf lokale Variablen | 17 % |
| CONS | 11 % |
| Typprädikate | 11 % |
| Arithmetik | 6 % |
| Vektoren | 5 % |
| Zugriff auf leere Liste | 3 % |

Tabelle 5.1: Verteilung der CPU-Zeit in LISP-Programmen nach [StHe88].

und MacScheme) durch neue Konzepte zu überwinden. Zusammenfassend sind Anforderungen an die neue Implementation:

1. Hohe Ablaufgeschwindigkeit. Die Ablaufgeschwindigkeit soll sich der in imperativen Sprachen erstellter Programme weiter nähern.

2. Geringer Hauptspeicherbedarf. Eine gute Nutzung der bei Mikrorechnern knappen Resource Hauptspeicher bewirkt eine Beschleunigung der Ausführungsgeschwindigkeit durch die Reduktion von Garbage Collections und ermöglicht umfangreichere Systeme (komfortable Editoren, Debugger und Benutzeroberflächen, komplexere Compiler, mehr Standardprozeduren).

3. Schnelle oder inkrementelle Garbage Collection. Die Garbage Collection ist ein ständiges Ärgernis für die Programmierer in LISP-Dialekten. Die Relevanz dieses Punktes zeigt ein Bericht David Moons von Symbolics über die Akzeptanz der Garbage Collection durch die Kunden der LISP-Maschinen seiner Firma:

   > „Users of the 3600 and its predecessors have avoided the garbage collector whenever possible. They found that garbage collection made interactive response so poor and unpredictable, and its overhead was so high, that it was preferable to turn off garbage collection and simply wait for the unevitable exhaustion of virtual address space (at which point the system crashes and must be re-booted).“ [Moon84]

Diesen drei Anforderungen kann durch folgende Spezifikationen entsprochen werden:

1. Eine hohe Ablaufgeschwindigkeit ist durch einen optimierenden Compiler erreichbar. Durch die zusätzliche Forderung nach geringem Hauptspeicherbedarf ist nur durch Interpretation von Bytecode — der etwa Faktor fünf weniger Speicherplatz als realer Maschinencode beansprucht [Sem87] und etwa Faktor drei weniger Platz als S-Code belegt [ScSt84] — und einen optimierenden Compiler, der Scheme in Bytecode übersetzt, erzielbar.

   Eine Bytecodeimplementation war bisher immer mit einem Interpreter in Maschincode verbunden, der den virtuellen Bytecoderechner auf dem realen Rechner implementierte. Eine Hardwareimplementation des Bytecodeinterpreters ist ein neuer Ansatz, der gegenüber den bestehenden Scheme-Prozessoren eine effizientere Implementation durch Nutzung des erlangten Stands der Compiler-Technik für Scheme verspricht. Der hardwareimplementierte Bytecodeinterpreter hat Typprüfungen, Laufzeitprüfungen bei Prozeduraufrufen und Standardprozeduren zu implementieren.

2. Geringer Hauptspeicherbedarf der Daten läßt sich durch CDR-Kodierung [Hans69] erreichen. Dieses Verfahren wird nur in Hardwareimplementationen eingesetzt, da es bei jeder primitiven Listenoperation eine Verzweigung anhand eines CDR-Kodes bedingt. Ferner ist der Hauptspeicherbedarf durch Repräsentation der Programme in Bytecode zu senken.

3. Eine schnelle Garbage Collection kann durch Verlagerung in den Hardware-Bytecodeinterpreter erreicht werden. Ferner ist die Belastung der Speicherverwaltung durch das Allozieren von Controlframes und Bindungsumgebungen auf einem Stack gering zu halten.

Die Hardwareimplementation eines Bytecodeinterpreters verspricht aus den oben genannten Gründen also eine Scheme-Implementation mit hoher Ablaufgeschwindigkeit bei geringem Hauptspeicherbedarf.

Bei einer solchen Hardwareimplementation wird ein Prozessor, der zentrale Teil eines Rechners, neu entworfen. Ein neuer Prozessor bedingt einen sehr großen personellen Aufwand für das Erstellen von Hardware und Software, die den Prozessor zu einem einsatzfähigen Rechner erweitern. Der Prozessor ist um Speicher und Ein-/Ausgabesysteme zu erweitern. Programme müssen neben dem Scheme-Compiler ein Betriebssystem, eine Benutzeroberfläche und Bibliotheken umfassen. Das Ergebnis wäre eine komplette Scheme-Workstation. Der Entwurf und Bau einer Scheme-Workstation erfordert zu große personelle und finanziellen Mittel. Daher mußte ein Konzept entwickelt werden, das so weit wie möglich vorhandene preiswerte Hardware und Software (Betriebssystem) nutzt. Es fiel die Entscheidung zugunsten eines Scheme-Prozessors, der in einen vorhandenen Mikrorechner eingesetzt wird. Der Mikrorechner stellt seinen Hauptspeicher, sein Betriebssystem und Bibliotheken mit Routinen für Arithmetik, das Wandeln von Zahlen in Strings, u.s.w. zur Verfügung. Der Scheme-Prozessor arbeitet einen eigenen, Scheme-spezifischen Maschinencode ab und übernimmt auch die Speicherverwaltung.

Der Scheme-Prozessor wird in einem vorhandenen Mikrorechner eingesetzt, welcher die Ein- und Ausgabe und Rechnungen mit Fließkommazahlen übernimmt, sowie seinen Hauptspeicher mit dem Koprozessor teilt (Abbildung 5.1). In diversen Veröffentlichungen wurde der Prozessor als Ko-

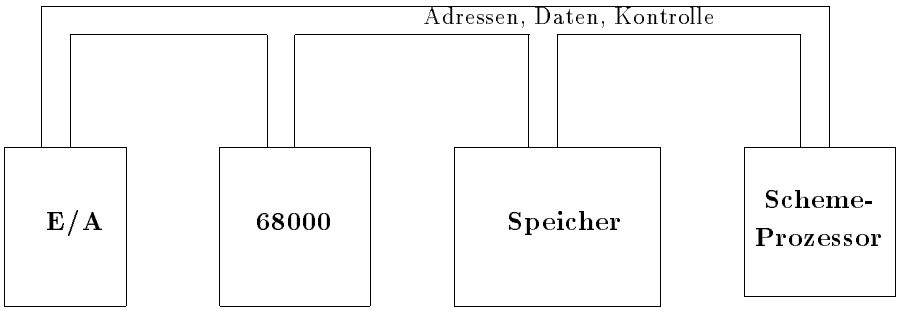



Abbildung 5.1: Der Scheme-Prozessor in einem 68000-Rechner.

prozessor bezeichnet [LoRa89a, LoRa89b, Lohs89], weil er nur zusammen mit einem Wirtsprozessor, der die Ein- und Ausgabe sowie Rechnungen mit Fließkommazahlen übernimmt, einsatzfähig ist. Der Name Koprozessor ist jedoch im wahrsten Sinne des Wortes zu verstehen, da der Scheme-Prozessor nicht wie etwa ein Arithmetik-Koprozessor, nur für bestimmte Operationen von einem Wirtsprozessor hinzugezogen wird. Tatsächlich holt der Scheme-Prozessor aber selbsttätig seine Maschinenbefehle und führt diese selbständig aus, wobei er nur für Ein- und Ausgabeoperationen den Wirtsprozessor reaktiviert. Der Wirtsprozessor wird vom Scheme-Prozessor also zu einem I/O-Prozessor degradiert. Nur bei der Ausführung von Scheme-Programmen, die überwiegend

Fließkomma-Arithmetik treiben, wird die Aktivität des Wirtsprozessors die des Scheme-Prozessors erreichen oder übertreffen.

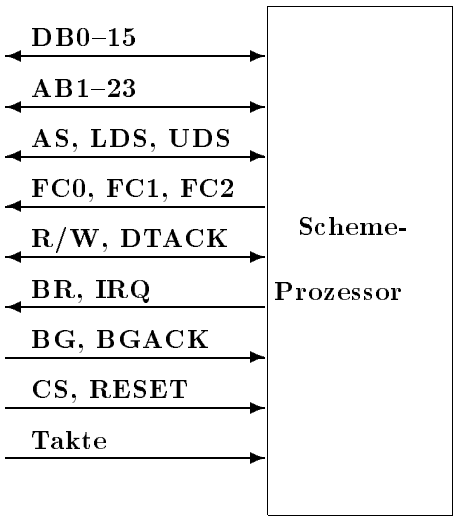



Abbildung 5.2: Der Scheme-Prozessor am 68000-Bus.

Die Wahl eines Rechners, der mit dem Scheme-Prozessor versehen werden soll, fiel auf Rechner mit Motorola 68000 Prozessor [68000]. Diese Prozessoren unterstützen ein Protokoll zur Busvergabe, welches auch im Scheme-Prozessor implementiert werden kann. Der Scheme-Prozessor mit den 68000-Bussignalen ist in Abbildung 5.2 gezeigt. Als erster Rechner wurde der Atari ST mit dem Scheme-Prozessor ausgerüstet. Gerade dieser Rechner bot sich aufgrund der restriktiven Zeitplanung an, da der Verfasser aus früheren Tätigkeiten als Autor des Pascalcompilers (`ST Pascal Plus` in Deutschland [Lohs85], `Personal Pascal` in den USA [Lohs86]) und Implementator eines Betriebssystems auf dem Atari ST (OS-9/68000) über die nötigen Detailkenntnisse der Hardware und Software dieses Rechners verfügt.

Eine Alternative wäre der Einsatz in Rechnern des Typs IBM AT, da diese in noch größerer Zahl als der Atari ST am Fachbereich Informatik der Universität Hamburg vorhanden sind. Leider ist der Bus in IBM ATs nicht dafür ausgelegt, daß Hauptspeicherzugriffe ausführende Scheme-Prozessoren hinzugefügt werden können. In diesen Rechnern können nur die CPU und die DMA-Einheit von Festplatte und Diskettenlaufwerk Adressen an den Hauptspeicher anlegen.

Diese Aufgabe, einen preiswerten Rechner durch eine preiswerte Zusatzhardware zu beschleunigen, ist vermutlich nicht nur an Universitäten interessant. Auch KI-Anwendungen, die in LISP erstellt werden, können häufig nur dann außerhalb der Labore ihrer Entwickler eingesetzt werden, wenn sie auf preiswertere Hardware portiert wurden. Beispielsweise schreiben Thuy und Schnupp [ThSc89] in einer Einführung in Expertensysteme:

> „Expertensysteme sollen Wissen verbreiten und nutzbar machen. Also gehören sie nicht in den Elfenbeinturm.
> Doch da befanden sie sich Anfang bis Mitte der 80er Jahre fast ausschließlich. Der Grund war eine akademisch orientierte Forschungspolitik, die den Hochschulen zusammen mit Forschungsmitteln auch gleich die *Lisp-Maschinen* zur Verfügung stellte, auf denen sie ihre Systeme entwicklen sollten. Die dann natürlich niemand einsetzen konnte und wollte. Nicht nur, daß diese Maschinen und die darauf laufenden Softwaresysteme

> eine Zehnerpotenz teuer waren als in der Leistung vergleichbare, eingeführte Computer. Sie ließen sich auch in die vorhandene DV-Umgebung nicht sinnvoll einpassen. Wissensbasierte Software gehört auf Alltagsmaschinen." [ThSc89]

Thuys und Schnupps Folgerung, daß die Software auf Alltagsmaschinen gehöre, ist dadurch zu erreichen, daß eine Alltagsmaschine um einen preiswerten Zusatzprozessor zu erweitert wird, der wissensbasierte Software optimal unterstützt. Der eingeführte, in seine DV-Umgebung eingepaßte Rechner wird durch den Zusatzprozessor für KI-Anwendungen tauglich.

### 5.3.1 Verwandte Arbeiten

In der Literatur sind Zusatz- oder Koprozessoren für LISP zur Ergänzung vorhandener Rechner kaum vertreten. Zwei ähnliche Arbeiten sollen zum Vergleich erwähnt werden. Sie zeigen insbesondere, wie verschieden die Randbedingungen ausfallen können.

Eine ähnlich gelagerte Arbeit ist die Dissertation von Werner Eicher an der Universität Kaiserslautern [Eich85]. Eicher beschreibt in seiner 1985 erschienenen Arbeit das Erweitern eines Rechners Mincal 621 der Firma Dietz (0,2 MIPS) um ein LISP-Werk. Der Universalrechner führt Listenprimitive und die Garbage Collection aus. Das LISP-Werk wertet Ausdrücke aus, enthält dazu vier Stacks und speichert die Propertylisten der Atome sowie die Bindungsumgebungen. Der verwendete LISP-Dialekt ist eine Teilmenge von LISP 1.5 mit dynamischer Variablenbindung. Funktionen als Parameter werden ausgeschlossen, um mit Stacks arbeiten zu können [Eich85]. Zur Messung der Performanz rechnet Eicher kurze, eigene Programme. Er vergleicht mit Implementationen auf anderen Rechnern und kommt zum Schluß, daß seine Hardwarelösung schneller als eine nicht näher bezeichnete Symbolics mit 5 MIPS ist, wozu er die Ergebnisse aller Rechner auf eine Rechnerleistung von 1 MIPS skaliert. Die Verwendung des betagten Mincal-Rechners erklärt sich vermutlich durch die aus seinem Lebenslauf ersichtliche elfjährige Bearbeitung der Aufgabe.

Eine aktuellere Aufgabe wird bei der Siemens AG gelöst. Der Prozessor Colibri (Coprozessor für LISP auf der Basis von RISC) [Hafe89] basiert auf den Erfahrungen mit SPUR und MIPS-X und ist als Mikroprozessor konzipiert, der eine LOAD/STORE-Architektur implementiert. Eine 4-stufige Pipeline wird verwendet, und 250 Register sind vorgesehen. Colibri soll zusammen mit einem Instruktionscache, einem Datencache, einer Fließkommaeinheit, einer MMU und 16 MByte Hauptspeicher einen Rechner bilden, der über ein Businterface an einen Wirtsrechner angeschlossen wird [Hafe89]. Das Colibrisystem stellt damit einen vollständigen Rechner dar und unterliegt damit nicht den restriktiven Randbedingungen (Vermeidung von Hardware- und Softwareaufwand), denen der in dieser Arbeit beschriebene Scheme-Prozessor genügen muß.

Auch Symbolics MacIvory [West89] und TI MacExplorer [Boss87, Matt87] sind an den Apple Macintosh angepaßte Versionen von als eigenständige LISP-Maschinen entwickelten Rechnern, deren Peripherie durch den Macintosh ersetzt wurde. Sie unterliegen daher anderen Randbedingungen als der hier beschriebene Prozessor.

## 5.4 Repräsentation der Objekte

Die Repräsentation der Objekte wurde als zentrale Implementierungsentscheidung betont. Die Repräsentation der Objekte soll daher vor der Entwicklung von Architektur und Befehlssatz vorgegeben werden. Damit wird gewährleistet, daß der zu entwerfende Prozessor alle hier vorgegebenen Objekte in der hier vorgegebenen effizienten Weise unterstützt. Architekturen wie jene bei Scheme-79 und Scheme-81 sind damit in dieser frühen Phase schon als Lösungen ausgeschlossen.

Die Objekte sollen kompakt repräsentiert und effizient zugegriffen werden können. Unter den Objekten sind hier die Daten, die Scheme dem Benutzer zur Verfügung stellt, die Bindungsum-

gebungen, die Controlframes und die Interndarstellung der Programme, der Maschinencode des Prozessors (Bytecode), zu verstehen.
Es liegen folgende Implementationsentscheidungen zugrunde:

- Der Hauptspeicher wird als Feld von 32 Bit Worten angesehen. Kleinere Einheiten als 32 Bit Worte werden nie gelesen oder geschrieben und größere Einheiten vom Interpreter in Zugriffe von 32 Bit Worten zerlegt.

- Die Speicherverwaltung kann Blöcke beliebig vieler 32 Bit Worte liefern. Sie arbeitet als unterbrechendes Verfahren nach dem Stop-and-Copy Algorithmus [Chen70]. Der Algorithmus arbeitet mit einer geringen Zahl von Hauptspeicherzugriffen und kompaktifiziert den Speicher, sodaß Blöcke beliebiger Länge zu jeder Zeit angefordert werden können. Eine Modifikation zu einem inkrementellen Verfahren ist wünschenswert, aber aufgrund der restriktiven Zeitplanung im ersten Prototyp nicht realisiert.

- Listen werden CDR-kodiert [Hans69] dargestellt.

- Die Typen der Objekte werden im Pointer dargestellt und damit FIXNUMs und Zeichen als Datum anstelle des Pointers repräsentiert.

- Vektoren, Strings und der Maschinencode des Scheme-Prozessors werden als Folgen von Speicherworten repräsentiert.

- Symbole werden durch Zahlen dargestellt, die gleichzeitig als Index in der globalen Bindungsumgebung verwendet werden.

- Controlframes und Bindungsumgebungen werden soweit wie möglich im Stack alloziert. Nur wenn diese eine längere Lebensdauer als die sie erzeugende Prozedur haben, werden sie in den Heap kopiert.

### 5.4.1 Das Speicherwort

Die Maschinenworte sind 32 Bit breit, wovon 24 Bit für die Pointer oder für Zeichen und ganze Zahlen, 6 Bit für den Typ und 2 Bit für den CDR-Code genutzt werden. Abbildung 5.3 zeigt den Aufbau eines Maschinenwortes.

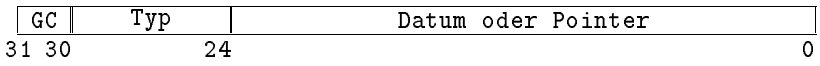



Abbildung 5.3: Struktur der Speicherworte des Scheme-Prozessors.

Der CDR-Code nimmt die in Tabelle 5.2 angegebenen Werte an. Der CDR-Kode hat nur in Listen die in der Tabelle beschriebene Bedeutung, bei allen anderen Objekten ist dieses Feld undefiniert. In der globalen Bindungsumgebung wird der CDR-Code von der Speicherverwaltung verwendet, um nicht mehr referenzierte Symbole zu erkennen und ihre Nummer — falls sie keine globale Bindung haben — wiederzuverwenden.

| | |
|---|---|
| **Normal** | Das folgende Speicherwort enthält den CDR. Dieser Fall ist identisch zu CONS-Zellen. |
| **Next** | Der CDR ist eine Liste, dessen erstes Element an der nächsten Adresse liegt. Ein Pointer auf den CDR wird daher nicht mehr repräsentiert. |
| **Nil** | Letztes Element der Liste, der CDR ist NIL. |
| **Indirekt** | Pointer auf Zelle mit CDR-Code **Normal**, in der CAR und CDR stehen. |

Tabelle 5.2: Belegungen der CDR-Codes.

| Objekttyp | Datum / Pointer | Bemerkung |
|---|---|---|
| NULLT | 0 | repräsentiert () und #F |
| FIXNUM | Wert | |
| EOFT | 0 | End-Of-File |
| CHART | Wert | |
| UNBOUND | undefiniert | |
| PAIR | Pointer | |
| SYMBOL | Zahl | |
| BIGNUM | Pointer | |
| FLONUM | Pointer | |
| STRINGT | Pointer | |
| VECTOR | Pointer | |
| CODE_VECTOR | Pointer | |
| CLOSURE | Pointer | |
| ENVT | Pointer | Bindungsumgebung |
| CONTROL | Pointer | Controlframe, Continuation |
| INPUT_PORT | hostspezifisch | |
| OUTPUT_PORT | hostspezifisch | |
| GC_FORWARD | Pointer | während der GC verwendet |
| GC_PROTECT | Zahl | Worte bleiben bei der GC unangetastet |

Tabelle 5.3: Vom Scheme-Prozessor unterstützte Objekte.

### 5.4.2 Die Typen der Objekte

Der Typ der Objekte wird in 6 Bit kodiert. Damit können maximal 64 unterschiedliche Objekttypen implementiert werden. Vom Prototyp des Scheme-Prozessors werden die in Tabelle 5.3 angegebenen Typen unterstützt.

Im Gegensatz zu praktisch allen anderen Implementationen von LISP-Dialekten gibt es für die leere Liste einen eigenen Typ anstelle einer Darstellung durch der Typ `PAIR` mit einer für die leere Liste reservierten Adresse (meistens 0). Dieser eigene Typ für die leere Liste und die Konstante #F berücksichtigt, daß das Prädikat `pair?` auch Listen von der leeren Liste unterscheiden muß. `pair?` prüft in dieser Implementation wie auch `null?` einfach den Typ. Die Identität der leeren Liste mit der booleschen Konstante #F erleichtert die Implementierung bedingter Verzweigungen, bei der die leere Liste und #F gleich behandelt werden müssen.

### 5.4.3 Darstellung von Bindungsumgebungen

Bindungsumgebungen werden in einem kontinuierlichen Speicherbereich dargestellt. Im Gegensatz zur Darstellung als Liste, die in den vorhandenen Scheme-Prozessoren Scheme-79 und Scheme-81 gewählt wurde, können damit alle Argumente einer Prozedur in gleicher Zeit zugegriffen werden. Eine Bindungsumgebung in Scheme besteht aus einem Verweis auf die umgebende Bindungsumgebung und den an die Variablen gebundenen Objekten. Im Gegensatz zu LISP mit seiner dynamischen Bindung sind die Namen der Variablen nicht mit zu repräsentieren, da sie in Scheme nicht zur Laufzeit durchsucht werden müssen[1]. Abbildung 5.4 zeigt den Aufbau einer Bindungsumgebung für den Aufruf einer Prozedur von drei Argumenten.

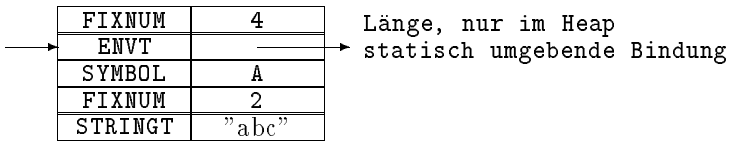



Abbildung 5.4: Bindungsumgebung nach Prozeduraufruf `(f 'A 2 "abc")`.

Auf den $i$-ten Parameter kann durch Indizieren des Pointers auf die Bindungsumgebung mit $-i-1$ zugegriffen werden. Bindungsumgebungen sind in Stack und Heap fast identisch dargestellt. Im Heap haben sie zusätzlich ein Feld, das ihre Länge angibt und von der Speicherverwaltung verwendet wird.

### 5.4.4 Darstellung von Controlframes und Continuations

Controlframes speichern den Status einer Prozedur, während eine andere Prozedur aufgerufen wird. Controlframes werden immer im Stack angelegt. Nur bei der Erzeugung einer Continuation werden sie in den Heap kopiert. Während eine Funktion ausgeführt wird, befindet sich ihr Controlframe immer auf dem Stack. Controlframes werden, wie in Abbildung 5.5 gezeigt, dargestellt. Unterhalb

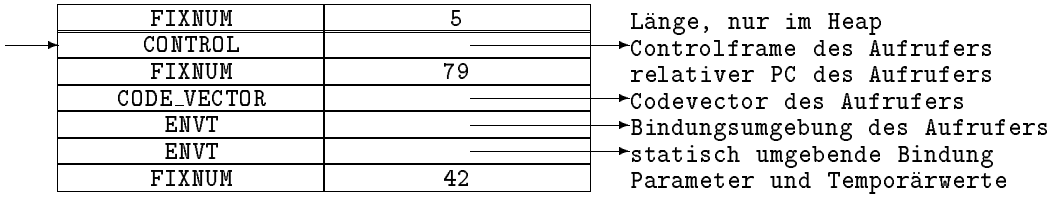



Abbildung 5.5: Aufbau eines Controlframes.

des Zeigers auf die Bindungsumgebung befinden sich die Parameter bei einem Controlframe im Stack oder deren Continuation und darunter bei allen Controlframes temporäre Werte, die auf dem Stack lagen. Bei Controlframes im Heap wird deren Länge für die Speicherverwaltung abgelegt.

[1] Für Debugzwecke, Fehlermeldungen und für die (optionalen) Standardprozeduren `make-environment` und `access` werden Variablennamen auch in Scheme benötigt. Sie werden in dieser Implementation jedoch nicht in den Bindungsumgebungen, sondern im Codevektor der zugehörigen Prozedur abgelegt.

### 5.4.5 Darstellung von Codevektoren

Der Maschinencode des Scheme-Prozessors wird in Codevektoren abgelegt. Der Code ist als Bytecode repräsentiert: Jeder Befehl belegt ein Byte und hat eventuell noch weitere Bytes als Argumente. Vier Bytes liegen in ein Maschinenwort gepackt im Speicher. Da damit auch das Typfeld belegt ist, darf dieser Maschinencode bei der Garbage Collection nicht interpretiert werden. Er wird daher durch ein Wort mit dem Typ `GC_PROTECT` und der Anzahl der mit Bytecode belegten Worte gekennzeichnet. Neben dem Maschinencode enthält ein Codevektor noch die Anzahl der Parameter und den Namen der Prozedur, dessen Code er darstellt, falls die Prozedur ein `NAMED-LAMBDA` [Rees86] ist. Dieser Name wird für Fehlermeldungen und für Debugzwecke verwendet. Für Debugzwecke kann auch ein Vektor mit den Namen der Parameter und mit dem Quellcode der Prozedur vorhanden sein. Ferner enthält der Codevektor Literale, die im Code abgelegt sind und der Garbage Collection unterliegen. Abbildung 5.6 zeigt des genauen Aufbau eines Codevektors. Wenn der Eintrag für die Anzahl der Parameter im Bereich von $0 - 255$ liegt, wird

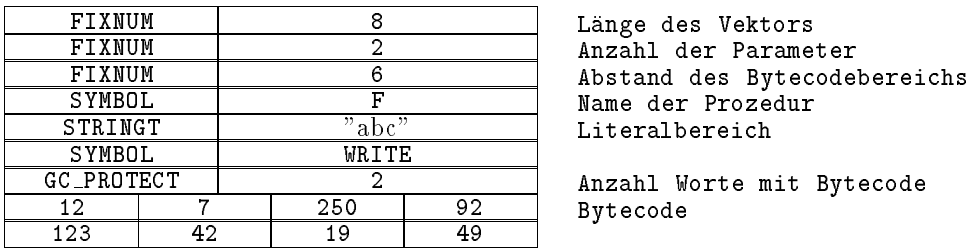



Abbildung 5.6: Aufbau eines Codevektors.

genau diese Parameteranzahl erwartet. Ist die Zahl größer als 255, so hatte die Parameterliste ein sogenanntes Restargument, an die der Rest der Parameter als Liste gebunden wird. Der in Abbildung 5.7 gezeigte Codevektor folgender Prozedur enthält damit die Parameterzahl 256:

```
(define (list . a)
  a)
```

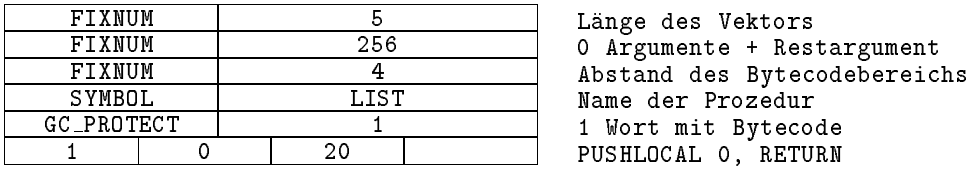



Abbildung 5.7: Codevektor der Prozedur `list`.

## 5.5 Diskussion der Randbedingungen für den Befehlssatz

Für den Entwurf eines Scheme-Prozessors wie jedes anderen Prozessors ist die Festlegung des Befehlssatzes die zentrale Aufgabe. Einerseits muß jedes Scheme-Programm in Folgen von Befehlen aus diesem Befehlssatz übersetzt werden können, wobei diese Übersetzung schnell durchführbar

sein muß, weil sie in einer interaktiven Programmierumgebung stattfinden soll. Andererseits muß der Befehlssatz von einem Prozessor effizient interpretiert werden können, und dieser Prozessor muß zudem als integrierte Schaltung mit den zur Verfügung stehenden Entwurfshilfsmitteln und Fertigungsmöglichkeiten realisierbar sein.

Zur Festlegung des Befehlssatzes des Scheme-Prozessors muß seine Architektur bestimmt sein. Damit sind Entscheidungen verbunden wie die zwischen einer register- und einer stackorientieren Architektur. Ferner ist über die Anzahl der Register im Prozessor zu entscheiden.

Da die Festlegung von Architektur und Befehlssatz für den Scheme-Prozessor und Angabe der Hardwarestruktur des Scheme-Prozessors Randbedingungen, die durch die Realisierung des Scheme-Prozessors als Chip bestehen, genügen muß, sollen diese Randbedingungen im folgenden Kapitel angegeben werden. Architektur und Befehlssatz werden daran anschließend in Kapitel 7 beschrieben.

# Kapitel 6

# Entwurf integrierter Schaltungen

Dieses Kapitel führt in die Problematik der integrierten Schaltungen ein, stellt allgemein übliche Entwurfs- und Fertigungsverfahren für kundenspezifische Schaltungen vor, beschreibt den Ablauf eines Entwurfs und beschreibt schließlich die Einschränkungen, unter denen Entwürfe von Hochschulen gefertigt werden können. Dieses Kapitel hat in dieser Arbeit die Aufgabe, die Randbedingungen beim Entwurf integrierter Schaltungen, diejenigen durch die hier zu verwendende Standard- und Makrozellentechnik und die Randbedingungen durch die Fertigung von Hochschulchips anzugeben, um die Entwurfsentscheidungen bei Architektur und Realisierung des Scheme-Prozessors, die zum Einhalten dieser Randbedingungen getroffen wurden, (Bottom-Up-Einflüsse) begründen zu können.

## 6.1 Integrierte Schaltungen

Die Entwicklung der integrierten Schaltungen nahm eine sehr stürmische Entwicklung. Zu Beginn dieser Entwicklung, Anfang der 60er Jahre, wurden Schaltungen mit wenigen Transistorfunktionen gefertigt. Ende 1989 gehen ICs (Speicherchips) mit über 4 Millionen Transistorfunktionen in die Serienproduktion. Anfang 1989 wurden schon erste Labormuster von ICs mit über 16 Millionen Transistorfunktionen vorgestellt. Nach den bisherigen Erfahrungen vervierfacht sich die Anzahl der auf einem Chip fertigbaren Transistorfunktionen alle drei Jahre [Hörb86].

Unter integrierten Schaltungen (Integrated Circuits, ICs) werden hier monolithisch integrierte Schaltungen verstanden, bei denen eine Vielzahl von elektronischen Bauteilen gleichzeitig auf einer Siliziumscheibe hergestellt sind. Diese integrierten Schaltungen werden nach der Anzahl der in enthaltenen Transistorfunktionen in die Gruppen SSI, MSI, LSI und VLSI eingeteilt (Tabelle 6.1). Die Tabelle folgt der Einteilung nach [Lage87]. In der Literatur werden auch andere Zuordnungen der Transistorzahlen zu den Begriffen vorgenommen.

| Abkürzung | Name | Transistorfunktionen |
|---|---|---|
| SSI | Small Scale Integration | $< 100$ |
| MSI | Medium Scale Integration | 100 – 1000 |
| LSI | Large Scale Integration | 1000 – 10000 |
| VLSI | Very Large Scale Integration | $> 10000$ |

Tabelle 6.1: Integrationsgrade integrierter Schaltungen.

Als LSI- und MSI-Bausteine wurden Standard-ICs wie Gatter, Flipflops, Register und ALU-Bausteine gefertigt, die für digitale Steuerungen und für die Computertechnik in großen Stückzahlen verkauft werden konnten.

Ab Anfang der 70er Jahre war es möglich und kosteneffizient, ICs mit einigen tausend Transistorfunktionen zu fertigen, die einem Funktionsumfang von mehreren Platinen mit SSI und MSI ICs entsprechen. Schaltungen dieser Komplexität wurden kaum, von Ausnahmen wie etwa Speicher-ICs und Taschenrechner-ICs abgesehen, in Stückzahlen benötigt, die die Entwurfskosten für ein LSI IC rechtfertigten. Die Halbleiterindustrie löste dieses Stückzahlproblem durch die Erfindung des Mikroprozessors, eines LSI oder VLSI ICs, das in großen Stückzahlen produziert werden kann, weil es erst beim Kunden durch ein anwendungsspezifisches Programm auf eine spezielle Aufgabe zugeschnitten wird.

In den 80er Jahren wurde der Entwurf integrierter Schaltungen durch Rechnerunterstützung soweit erleichtert, daß auch für die Anwender der Entwurf integrierter Schaltungen möglich wurde. Sehr viel Aufwand wurde in die Entwicklung von Design-Software investiert. Beispielsweise stellen die Halbleiterhersteller fertige Funktionsblöcke zur Verfügung, die den bisher verwendeten ICs entsprechen, und die Design-Software unterstützt die graphische Eingabe von Schaltplänen. Da die Anwender durch die Verwendung selbstentworfener ICs die Abmessungen und Verlustleistungen verringern sowie die Zuverlässigkeit und den Schutz gegen Nachbau durch die Konkurrenz steigern konnte, verbreiteten sich die ASICs (Application Specific Integrated Circuits). Es gibt verschiedene Entwurfs- und Fertigungsverfahren, die sich in den mit ihnen realisierbaren Schaltungen und dem Fertigungsaufwand unterscheiden.

## 6.2 Entwurfsverfahren für integrierte Schaltungen

In den folgenden Abschnitten werden die wichtigsten Entwurfstechniken dargestellt.

### 6.2.1 Der Full-Custom Entwurf

Im Full-Custom Entwurf erstellen die Entwerfer alle Bausteine ihrer Chips, z. B. Gatter, Flipflops, Speicher selbst und verbinden sie miteinander. Es werden alle Transistoren manuell erstellt, die daher optimal für die Gesamtschaltung ausgelegt werden können, um Treiberleistung und Platzbedarf zu optimieren. Die Entwerfer können die Fläche des Chips völlig frei in Transistoren und Verdrahtung einteilen.

Da sich der Full-Custom Entwerfer mit Transistoren als unterster Betrachtungsebene beschäftigt, von denen VLSI-Chips über 10000 beinhalten, ist diese Entwurfstechnik sehr personalaufwendig und damit teuer. Sie wird nur dort eingesetzt, wo sehr enge Restriktionen für Chipfläche oder Verarbeitungsgeschwindigkeit bestehen und die ICs in sehr großen Stückzahlen gefertigt werden sollen.

Die Verifikation eines Full-Custom-Chips gestaltet sich schwierig, weil die Primitivelemente dieses Chips Transistoren sind. Es werden Schaltkreisextraktoren eingesetzt, das sind Rechnerprogramme, die aus dem entworfenen Layout eine Transistorschaltung einschließlich der Parameter jedes Transistors extrahieren. Diese Transistorschaltung kann der Entwerfer dann simulieren. Der Simulator muß Differentialgleichungssysteme lösen, um diese Simulation durchzuführen. Diese extrem rechenzeitintensive Aufgabe kann nur für Schaltungen bis zu einigen hundert Transistoren gelöst werden, wodurch immer nur kleine Teile des entworfenen Chips simulativ verifiziert werden können. Es sind immer mehrere Fertigungsvorgänge einzuplanen, da viele Fehler erst in den Prototypen entdeckt werden.

### 6.2.2 Der Semi-Custom Entwurf

Die Idee beim Semi-Custom Entwurf ist, dem Entwerfer fertige Bausteine, sog. Zellen, zu liefern, für die der Hersteller Parameter wie die logische Funktion, das Zeitverhalten und die Treiberfähigkeit der Ausgänge angibt. Solche Zellen enthalten von 2 bis zu einigen hundert Transistorfunktionen, und der Hersteller garantiert für die korrekte Funktion der Zellen. Zellen bei digitalen ICs entsprechen Gattern, Flipflops und Registern. Der Entwerfer baut seinen Entwurf aus diesen Zellen auf, wodurch er wesentlich weniger Bausteine verwenden muß. Insbesondere gestaltet sich die Simulation einer Schaltung aus diesen Zellen sehr viel praktikabler als eine Transistorschaltung, weil sie als digitale Schaltung simuliert wird, wodurch problemlos auch komplette VLSI-Chips simulierbar bleiben.

Als typische Produktivität eines in Full-Custom-Technik arbeitenen Entwerfers werden in [Hörb86] 300 Gatterfunktionen pro Mannjahr angegeben. Bei Semi-Custom-Entwürfen sei dieser Wert 10 bis 20 mal so hoch.

#### 6.2.2.1 Die Gate-Array-Technik

Beim Gate-Array Entwurf werden vorgefertigte integrierte Schaltungen eingesetzt, die schon alle Transistoren enthalten und nur noch durch unterschiedliche Verdrahtung der Transistoren spezialisiert werden. Ein Gate-Array ist fest in Bereiche mit Transistoren und Kanäle fester Breite für Verdrahtung eingeteilt. Die Abbildung 6.1 zeigt ein Gate-Array. Eine Zellbibliothek enthält die Verdrahtungen für Transistorbereiche. Auch die Peripheriezellen werden durch Verdrahtung zum Beispiel entweder zu Ein- oder zu Ausgabetreibern spezialisiert.

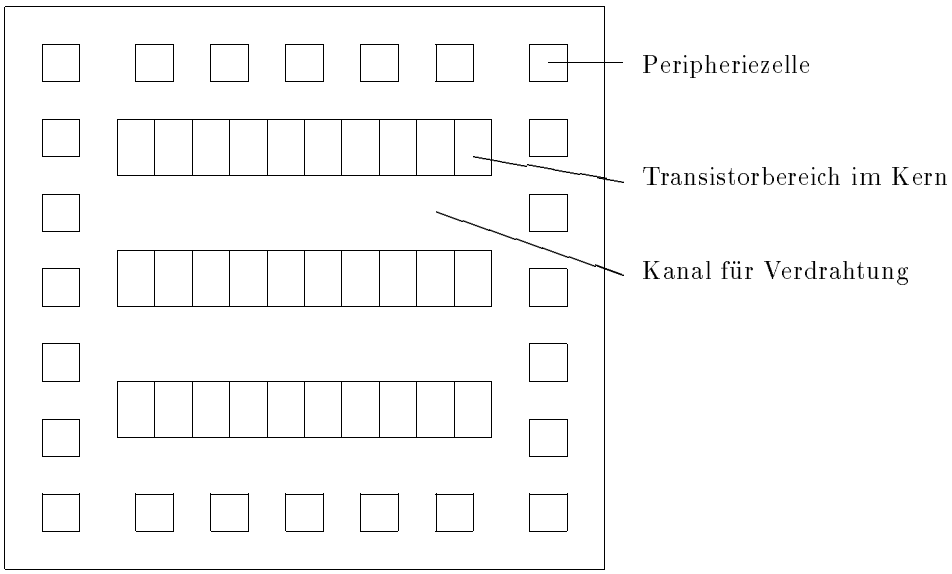



Abbildung 6.1: Schematischer Aufbau eines Gate-Arrays.

Die Gate-Array-Technik ist durch die Verwendung vorgefertigter sogenannter Master besonders wirtschaftlich, weil die Master in großen Stückzahlen identisch gefertigt werden. Nur die Verdrahtung ist für den jeweiligen Entwurf unterschiedlich aufzudampfen. Damit kann sich ein Großteil der Herstellungskosten auf die Masterfertigung mit großen Stückzahlen verteilen, und es

ist im günstigsten Fall nur eine Maske kundenspezifisch zu erstellen. Zudem haben Gate-Arrays wegen des geringen kundenspezifischen Aufwands auch die kürzesten Fertigungszeiten von allen hier vorgestellten Entwurfsverfahren.

Die Nachteile der Gate-Arrays sind in dem Master mit seinen in Anzahl, Parametern und Anordnung festen Transistoren und den festen Verdrahtungskanälen begründet: Die Fläche kann nicht optimal genutzt werden, und Schaltungen, die als reguläre Transistoranordnungen in Full-Custom Technik sehr platzsparend realisiert werden können, wie PLAs oder Speicher, sind so gut wie unmöglich mit Gate-Arrays zu realisieren.

#### 6.2.2.2 Die Standardzell-Technik

Beim Standardzell-Entwurf wird ein Chip aus vom Halbleiterhersteller vorentworfenen Standardzellen zusammengesetzt. Diese Standardzellen haben alle die gleiche Höhe und eine unterschiedliche Breite. Abbildung 6.2 zeigt ein Standardzell-Chip. Die Abbildung zeigt deutlich, daß der

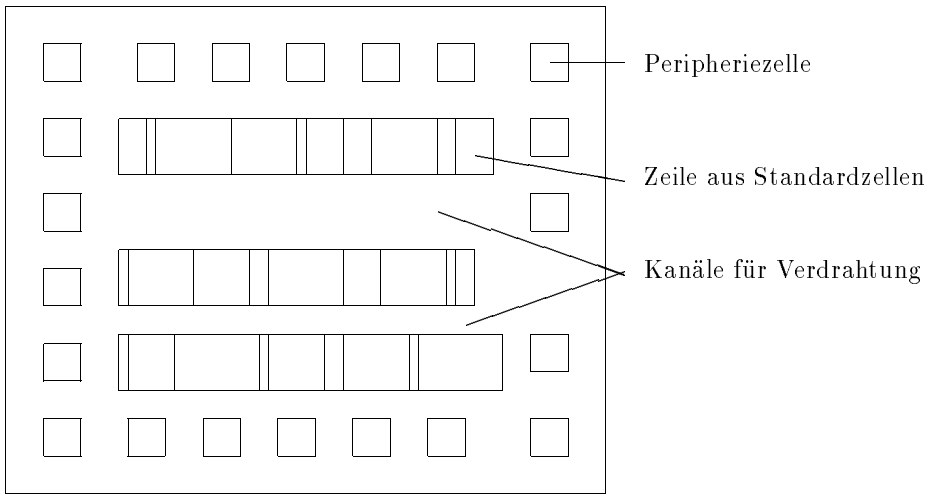



Abbildung 6.2: Schematischer Aufbau eines Standardzell-ICs.

Entwurf gegenüber einem Gate-Array deutlich flexibler ist: Die drei gezeigten Zeilen aus Standardzellen haben alle eine unterschiedliche Breite. Die Breite der Kanäle ist variabel und kann dem tatsächlichen Bedarf angepaßt werden. Ferner sind die Anzahl und Lage der Peripheriezellen auf dem Rand durch den Entwerfer bestimmt.

Im Detail wird ein Ausschnitt eines Standardzell-Chips in Abbildung 6.3 gezeigt: Die drei dort gezeigten Standardzellen sind alle gleich hoch und verschieden breit. Die Stromversorgung (Vss und Vdd) wird innerhalb einer Standardzellzeile parallel über die ganze Zeile geführt. Die Signalanschlüsse der Standardzellen befinden sich am oberen und am unteren Rand, also am Rand der Verdrahtungskanäle. Lücken in den Standardzellen ermöglichen es, Signale durch Standardzellzeilen zu führen und damit nicht aneinander angrenzende Kanäle zu verbinden.

Der Preis für diese erhöhte Flexibilität des Entwurfs gegenüber dem Gate-Array besteht darin, daß keine vorgefertigten Master verwendet werden können. Die Fertigung eines Standardzell-Entwurfs ist genauso teuer wie die eines Full-Custom-Entwurfs. Das Standardzellen-Verfahren hat seine Grenzen, wenn reguläre, großflächige Transistoranordnungen wie PLAs oder Speicher benötigt werden, da diese nicht in das Konzept von Zellen fester Höhe passen.

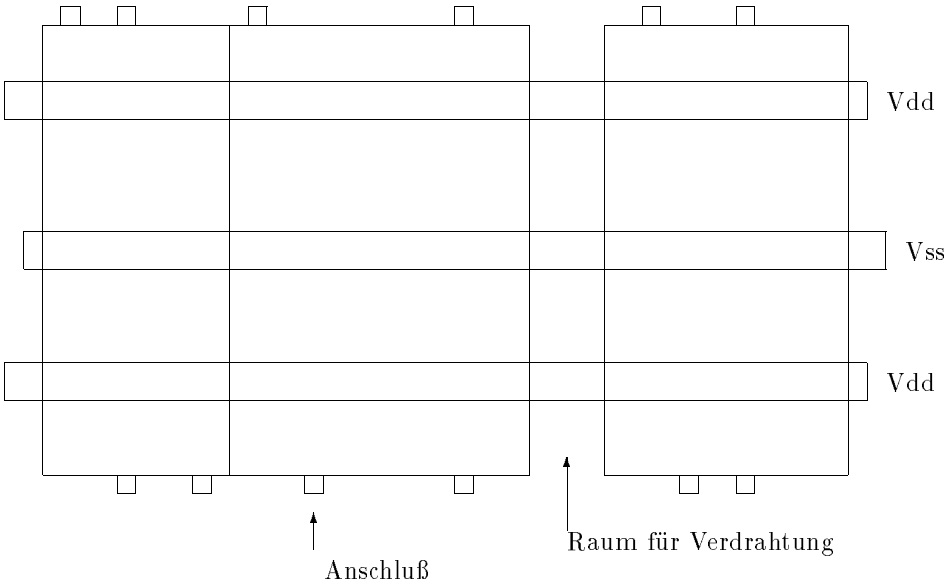



Abbildung 6.3: Ausschnitt aus einer Standardzellzeile.

#### 6.2.2.3 Die Makrozell-Technik

In Makrozell-Entwürfen werden Makros, das sind rechteckige Zellen beliebiger Größe, auf einem Chip verdrahtet. Solche Makros sind entweder fest, z. B. komplette Mikroprozessoren, oder sie werden von Rechnerprogrammen nach Vorgaben des Entwerfers generiert, z. B.:

- RAMs, nach Vorgabe von Wortbreite und Adreßraum.
- ROMs, nach Vorgabe von Wortbreite, Adreßraum und Belegung.
- PLAs, nach Vorgabe der Schaltfunktionen.

Makros können zudem in Full-Custom oder Standardzellentechnik entworfen sein. Solche entworfenen Makros werden wie Chips erstellt, mit dem einzigen Unterschied, daß die Ein- und Ausgänge nicht an Peripheriezellen geführt, sondern auf den Rand eines Rechtecks gelegt werden. Abbildung 6.4 zeigt ein Makrochip, das aus zwei generierten Makros und einem mit Standardzellen entworfenen Makro aufgebaut ist.

Der Makrozellenentwurf ist die mächtigste Semi-Custom Entwurfsart, da beliebige Rechteckstrukturen möglich sind. Zudem kann eine Mischform zwischen Semi-Custom und Full-Custom gewählt werden, indem ein besonders zeitkritisches oder platzintensives Schaltungsteil als Makro in Full-Custom entworfen wird. Gegen einen Makroentwurf sprechen, wenn auch ein Gate-Array verwendet werden kann, die Fertigungskosten, die so hoch wie bei Full-Custom Chips sind. Ferner ist der Flächenbedarf von Makrochips gegenüber reinen Standardzellenchips, die auf generierte Makros verzichten können, größer, weil durch das Verdrahten der Rechtecke variabler Größe mehr Verdrahtungsfläche benötigt wird.

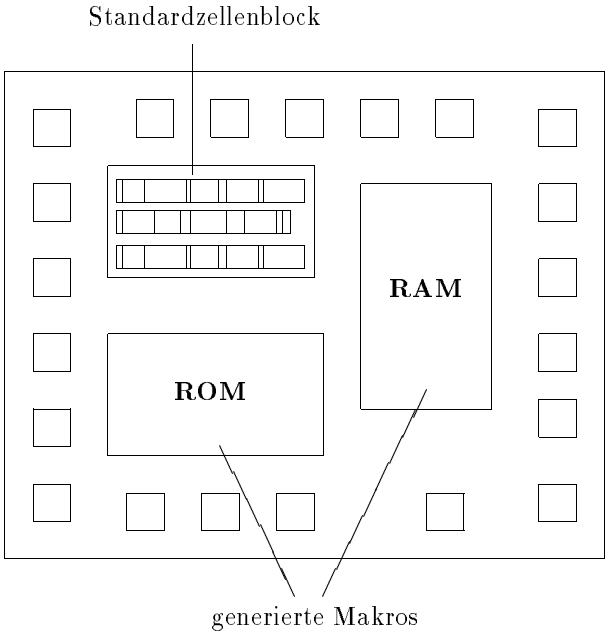



Abbildung 6.4: Schematischer Aufbau eines Makrozell-ICs.

## 6.3 Der Ablauf eines Semi-Custom Entwurfs

Hier soll der Ablauf eines Semi-Custom Entwurfs dargestellt werden, wie er mit einem zellorienterten Entwurfssystem wie VENUS-S der Siemens AG [Hörb86] möglich ist. In Abbildung 6.5 ist dieser Ablauf schematisch dargestellt. In den folgenden Absätzen sollen die Teilschritte in diesem Ablauf beschrieben werden.

### 6.3.1 Schaltplaneingabe

Bei der Schaltplaneingabe plaziert der Entwerfer Schaltsymbole, die Standardzellen entsprechen, am Bildschirm und verdrahtet diese untereinander. Die Schaltplaneingabe dient im Entwurfssystem dazu, die verwendeten Standardzellen und ihre Verbindungen ermitteln zu können. Eine textuelle Beschreibung der Schaltung wäre alternativ möglich. Die rechnergestützte graphische Eingabe entspricht jedoch eher den Präferenzen der Zielgruppe dieser Entwurfssysteme. Die graphischen Schaltbilder dienen auch der Dokumentation der entworfenen Schaltung, sofern sie auch geplottet werden können[1]. Schaltplaneditoren unterstützen die Partitionierung einer Schaltung. Sie erlauben auch die Benennung von Standardzellen und Leitungen, so daß der Entwerfer die Teile der Schaltung in den folgenden Entwurfsschritten wie Simulation und Chipkonstruktion unter den von ihm gewählten Bezeichnungen ansprechen kann.

[1] Für das Entwurfssystem VENUS-S ermöglicht ein vom Verfasser an der Universität Hamburg entwickeltes Programm das Plotten der Schaltbilder [Lohs87].

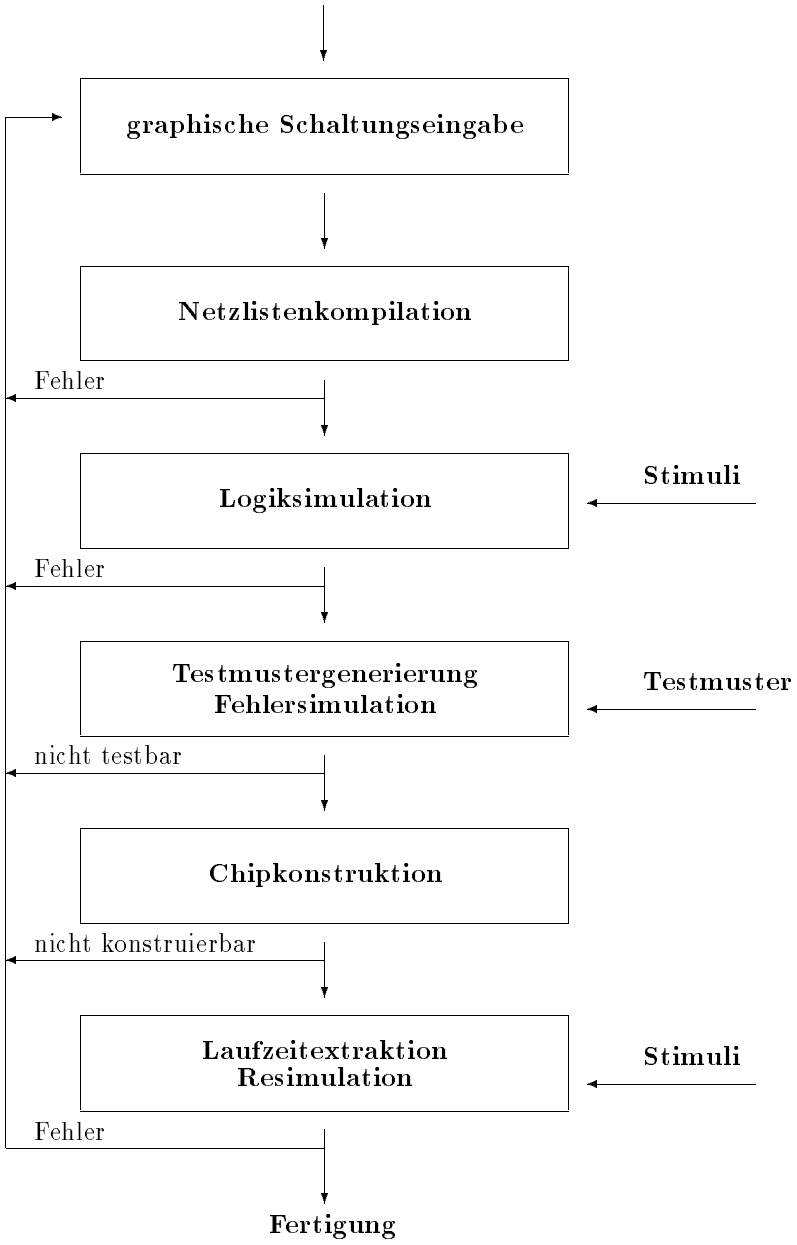



Abbildung 6.5: Schematischer Ablauf eines Entwurfs mit VENUS.

### 6.3.2 Netzlistenkompilation

Nach der Eingabe einer Schaltung erfolgt eine Netzlistenkompilation, in der die zuvor eingegebene Schaltung eingelesen und auf einfache Entwurfsfehler wie offene Eingänge und zusammengeschaltete Ausgänge geprüft wird. Wenn keine Fehler festgestellt werden, wird als Ergebnis der Netzlistenkompilation eine Netzliste[2] geschrieben. Diese Netzliste beschreibt die Standardzellen und ihre Verbindungen untereinander. Sie enthält keinerlei Informationen mehr wie die Lage der Bauteile im Schaltplan oder die Aufteilung in Teilschaltpläne. Diese Netzliste ist die Schaltungsbeschreibung, auf der alle folgenden Entwurfsprogramme aufsetzen. Zusätzlich zu dieser Netzliste werden bei der Netzlistenkompilation in VENUS-S noch Statistiken über die verwendeten Standardzellen sowie Abschätzungen der Signallaufzeiten aufgrund der Belastungen der Ausgänge durch die Kapazitäten der Eingänge erstellt.

### 6.3.3 Logiksimulation

In der Logiksimulation wird die entworfene Schaltung simulativ verifiziert. Der Entwerfer erstellt Stimuli, die die an die Schaltungseingänge anzulegenden Daten als Funktion der Zeit darstellen, und er kann als Ergebnis seiner Simulation die logischen Pegel an den Ausgängen und beliebigen internen Schaltungspunkten betrachten. Die Logiksimulation berücksichtigt die Verzögerungszeiten der Standardzellen und die vom Netzlistencompiler ermittelten Verzögerungszeiten durch die Belastung der Standardzellausgänge. Die durch die Leitungskapazitäten auf dem Chip entstehenden Verzögerungszeiten sind erst nach der Plazierung und Verdrahtung der Standardzellen bekannt; sie können daher hier noch nicht berücksichtigt werden. Diese erste Simulation dient dem frühzeitigen Finden von Entwurfsfehlern. Sie soll verhindern, daß die weiteren sehr rechenzeit- und arbeitsintensiven Entwurfsschritte für eine fehlerhafte Schaltung gemacht werden. Die Logiksimulation kann nicht nur für ein Gesamtchip, sondern auch für Teilschaltungen durchgeführt werden. Diese Möglichkeit ist besonders wichtig, weil das Aufspüren von Fehlern eine mit der Größe der Schaltung überproportional steigende Arbeitszeit beansprucht.

### 6.3.4 Testmustergenerierung oder Fehlersimulation

Bei der Fertigung integrierter Schaltungen wird ein bestimmter Prozentsatz der Schaltungen fehlerhaft gefertigt. Um nach der Fertigung die defekten Chips von den korrekten zu unterscheiden, werden die entstandenen Chips einzeln getestet und nur die korrekt arbeitenden Chips in (teure) Gehäuse eingebaut und dem Kunden geliefert. Der Kunde, der eine integrierte Schaltung entwirft, muß dem Halbleiterhersteller also neben den eigentlichen Fertigungsdaten Testmuster liefern, mit denen nach der Produktion defekte von gelungenen Chips getrennt werden können. Fertigungsfehler werden durch angenommene Kurzschlüsse eines jeden Knoten in der Schaltung mit logisch 0 und logisch 1 modelliert (stuck-at-Fehlermodell). Dieses ursprünglich für TTL-Schaltungen entwickelte Fehlermodell ist für CMOS-Schaltungen, in denen Leitungsunterbrechungen (stuck-open-Fehler) vorkommen, kombinatorische zu sequentiellen Schaltungen machen können [Abra86, Will86], zwar nicht optimal; es wird aber von den meisten Designsystemen, wie auch VENUS-S, verwendet. Algorithmen für die automatische Generierung von Testmustern existieren nur für kombinatorische Schaltungen. Der Test sequentieller Schaltungen kann auf den kombinatorischer Schaltungen zurückgeführt werden, wenn die Schaltung in einen Testmode gebracht werden kann, in dem alle speichernden Elemente geladen und ausgelesen werden können (scan-path-Verfahren). Wenn keine

[2] Eine Besonderheit von VENUS-S gegenüber anderen Entwurfssystemen ist, daß zwei unterschiedliche Netzlisten für Simulator (SBS-Datei) und Chipkonstruktion (CALBUS-Datei) generiert werden. Der Grund dafür liegt im Entstehen dieser Teilprogramme in verschiedenen Abteilungen des Konzerns.

rein kombinatorische oder Scan-path-Schaltung entworfen wurde, muß der Entwerfer seine Testmuster manuell erstellen. Ein Fehlersimulator, ein Programm, das die Schaltung mit den Testmustern simuliert, ermittelt die durch die Testmuster noch nicht gefundenen Stuck-at-Fehler sowie die Fehlerabdeckung, das Verhältnis der mit den Testmuster entdeckten zu allen angenommenen Fehlern. Der Fehlersimulator ist das rechenzeitintensivste Rechnerprogramm innerhalb eines Entwurfsablaufs, und das Erstellen von Testmustern, die eine ausreichende Fehlerabdeckung (>95%) erreichen, erweist sich für VLSI-Schaltungen sehr häufig als unmöglich, wenn die Testbarkeit nicht schon beim Entwurf der Schaltung als Ziel berücksichtigt wurde.

### 6.3.5 Chipkonstruktion

Der Chipkonstruktion fällt die Aufgabe zu, die Netzliste einer Schaltung in eine geometrische Darstellung eines Chips zu transformieren, die als Eingabe für die Fertigung dient.

Die Chipkonstruktion, die versucht, Positionen für alle Zellen und Leitungen zu finden, die eine minimale Gesamtleitungslänge und minimale Chipfläche ergeben, wird in die Teilschritte Plazierung und Verdrahtung zerlegt. Auch diese beiden Teilprobleme sind ebenso wie das Gesamtproblem NP-vollständig [Ullm84].

Bei der Plazierung der Standardzellen eines Standardzellchips oder Standardzellblocks innerhalb eines Makrochips werden den Standardzellen auf Zeilen und Positionen innerhalb der Zeilen verteilt, wobei versucht wird, die Gesamtlänge der Verdrahtung zu minimieren. Die Plazierungsprogramme enthalten Heuristiken für die Plazierung und erlauben Plazierungsvorgaben durch den Entwerfer. Die Peripheriezellen werden ebenfalls durch Heuristiken oder manuelle Plazierung auf Positionen am Rand verteilt. In VENUS-S werden die Standardzellen durch eine Heuristik einmal auf die Zeilen verteilt, und der Entwerfer kann die Zellen innerhalb dieser Zeilen permutieren und manuell vertauschen. Für Makrozellen erfolgt auch eine Plazierung durch eine Heuristik, die dann manuell verbessert werden kann.

Die Verdrahtung plazierter Standard- oder Makrozellen wird in zwei Teilschritte zerlegt: Die Verdrahtung beginnt mit der globalen Verdrahtung, bei der für jede Leitung eine stückweise Aufteilung in Verdrahtungskanäle durch eine Heuristik ermittelt wird, die die Leitungslänge kurz zu halten versucht. Anschließend werden die Leitungsstücke innerhalb der Kanäle verdrahtet, wozu der schnelle Kanalverdrahtungsalgorithmus [Ullm84] eingesetzt wird. VENUS-S kann nicht immer alle Leitungen legen. Der Entwerfer muß von den Programmen nicht verdrahtete Leitungen manuell vervollständigen. Bei dieser interaktiven Nacharbeit ist es auch möglich, automatisch gelegte Leitungen manuell zu verlegen, wenn besonders zeitkritische Signale zu lang geraten sind.

Nach Plazierung und Verdrahtung steht eine geometrische Beschreibung des Chips zur Verfügung, aus der zusammen mit der Netzliste die Verzögerungszeiten der Signale durch die kapazitive Belastung durch die Verdrahtung errechnet wird (Laufzeitextraktion).

Die Chipkonstruktion wird bei kommerziellen Entwürfen in den meisten Fällen dem Hersteller überlassen. Die Chipkonstruktion erfordert sehr viel Erfahrung, wenn ein gutes Ergebnis erzielt werden soll. Ein üblicher Entwurfsablauf ist daher, daß der Anwender seine Netzliste beim Hersteller abliefert, der Hersteller die Chipkonstruktion durchführt und dem Anwender das Ergebnis der Laufzeitextraktion übergibt.

### 6.3.6 Resimulation mit Leitungslaufzeiten

Aus der im Rahmen der Chipkonstruktion erfolgen Laufzeitextraktion liegen Verzögerungszeiten für das Chip vor, die das nach der Fertigung zu erwartende Verhalten gut approximiert. Eine erneute Simulation der Schaltung unter Berücksichtigung dieser Leitungslaufzeiten entscheidet

darüber, ob das Chip den Spezifikationen genügt. Wenn die Simulationsergebnisse den Spezifikationen des Entwerfers entsprechen, kann das Chip gefertigt werden. Sollten die Spezifikationen nicht eingehalten werden, ist zu entscheiden, ob eine erneute Chipkonstruktion unter Berücksichtigung der in dieser Resimulation aufgedeckten kritischen Pfade erfolgen soll, oder ob die Schaltung zum Beispiel durch Einbau von Treibern für lange Leitungen modifiziert werden soll.

## 6.4 Fertigungsmöglichkeiten für Hochschulen

Das Entwurfssystem VENUS-S wurde von der Siemens AG an 20 bundesdeutschen Hochschulen installiert. Dazu erhielten die Hochschulen einen Minirechner mit dem Betriebssystem BS2000, der etwa 0,7 MIPS leistet. Auf diesem Rechner laufen die Entwurfsphasen Logiksimulation, Testmustergenerierung, Fehlersimulation, Chipkonstruktion, Laufzeitextraktion und Resimulation ab. Die Schaltplaneingabe und die Netzlistenkompilation erfolgen auf Workstations der Firma Apollo. Die Siemens AG übernahm die Kosten für den BS2000-Rechner und die System- und Entwurfssoftware auf diesem Rechner und ermöglicht den Hochschulen den Betrieb des Rechners durch Einräumen reduzierter Wartungskosten für die Dauer von drei Jahren.

Die Hochschulen erhielten von Siemens die Zellbibliotheken für je zwei Gate-Array- und Standardzell-Prozesse. Fertigungsmöglichkeiten bestehen für die Hochschulen jedoch nur für Standardzell- und Makrozellentwürfe mit dem Prozeß ACMOS3 von Siemens, einem 2 $\mu$ CMOS-Prozeß mit zwei Metallebenen für Verdrahtung. Wenn mit VENUS-S entworfene Chips von Hochschulen für Forschungszwecke gefertigt werden sollen, übernimmt das Bundesministerium für Forschung und Technologie (BMFT) die Fertigungskosten. Diese Finanzierung besteht bis 1992.

Zur Verringerung der Fertigungskosten werden je nach Größe zwei bis neun Hochschulchips zusammengefaßt und als ein großes Chip (Reticle) in die Fertigung gegeben (Shared-Reticle-Verfahren). Nach der Fertigung werden diese Reticles zersägt und die Chips in Fassungen montiert. Jede Hochschule erhält 20 ICs, die sie selbst auf Fertigungsfehler untersuchen muß. Dieses Verfahren führt dazu, daß ein von einer Hochschule abgegebenes Chip erst dann gefertigt werden kann, wenn sich alle (bis zu neun!) Entwürfe für ein Reticle angesammelt haben. Ferner müssen die Entwürfe der Hochschulen ganz bestimmte Abmessungen haben, um zu diesen Reticles kombiniert werden zu können. Es sind daher nur Chips mit den vier in Tabelle 6.2 gezeigten Größen erlaubt. Die dort angegebenen maximalen Transistorzahlen sind [Stei88] entnommen.

| Chipgröße | Chips pro Reticle | max. Transistorzahl |
|---|---|---|
| 4,538 mm × 4,538 mm | 9 | 8000 |
| 5,388 mm × 5,368 mm | 6 | 19200 |
| 6,898 mm × 6,898 mm | 4 | 24000 |
| 8,828 mm × 8,758 mm | 2 | 36000 |

Tabelle 6.2: Mögliche Chipgrößen für Hochschulentwürfe.

Die Zellbibliothek zum Prozeß ACMOS3 enthält Standardzellen, die alle eine Höhe von 134 $\mu$m haben. Die Breite der Zellen variiert zwischen 24,6 $\mu$m für AND-, NAND-, NOR- und OR-Gatter mit zwei Eingängen und 926,6 $\mu$m für acht 4:1-Multiplexer in einer einzigen Standardzelle [Siem88]. Die Zellbibliothek bietet Gatter mit 2, 4, 6 und 8 Eingängen, Multiplexer 2:1, 4:1 und 8:1, Inverter und Treiber in drei Treiberstärken sowie Encoder und Decoder. Auch tri-state-Busse sind möglich. Als speichernde Elemente sind Latches, Flipflops, Schieberegister und Zähler enthalten. An arithmetischen Elementen werden Komparatoren und Addierer geboten.

Makrozellen können, wie in Abbildung 6.4 dargestellt, aus Standardzellenblöcken und aus generierten Makros aufgebaut werden. VENUS-S enthält Makrogeneratoren für RAMs, ROMs und PLAs. Die generierbaren Makros sind wie folgt eingeschränkt:

**ROM** ROMs können mit einer Wortbreite zwischen 1 und 64 sowie mit einer Anzahl von Worten, die eine Zweierpotenz sein muß, zwischen 32 und 16384 generiert werden. Das Produkt aus der Wortbreite und der Anzahl der Worte muß ferner zwischen 128 und 65536 liegen. Das ROM ist intern dynamisch aufgebaut: Es benötigt einen Takt, nach dessen negativer Flanke die Daten am Ausgang anliegen. Wenn der Takteingang logisch 1 ist, ist der Ausgang des ROMs hochohmig.

**RAM** Bei RAMs darf die Wortbreite zwischen 1 und 32 liegen und die Anzahl der Worte zwischen 16 und 16384 betragen. Das Produkt aus Wortbreite und Anzahl der Worte muß zudem zwischen 64 und 16384 liegen. Die generierten RAMs haben genau einen bidirektionalen Port. Der RAM-Generator enthält in der Version, die an die Hochschulen abgegeben wurde, einen Programmfehler, der ihn immer mit einer Fehlermeldung abbrechen läßt. Dieser Fehler kann durch Eingriff in Kommandodateien umgangen werden.

**PLA** PLAs können für 2 bis 50 Eingänge, 3 bis 50 Ausgänge und 2 bis 75 Terme generiert werden. Die an die Hochschulen abgegebene Version des PLA-Generators enthält jedoch einen Programmfehler, der dazu führt, daß mit einer geraden Anzahl von Termen generierte PLAs einen Kurzschluß enthalten. Es können daher nur PLAs mit ungerader Termanzahl generiert werden.

## 6.5 Folgerungen für den Entwurf des Scheme-Prozessors

Die zur Verfügung stehende Technologie von 2 $\mu$m CMOS mit der maximalen Chipgröße von 77,3 mm$^2$ erlaubt verglichen mit dem Full-Custom-Entwurf Scheme-81, dessen Fläche 85 mm$^2$ bei 2 $\mu$m NMOS beträgt, nur ein weniger komplexes Chip, da CMOS für die gleiche Funktion mehr Transistoren als NMOS erfordert, die zur Verfügung stehende Fläche kleiner ist und durch die Beschränkung auf Standardzelltechnik diese Fläche auch nicht optimal genutzt werden kann.

Im Jahre 1990 läuft der Wartungsvertrag zu erschwinglichen Bedingungen für den BS2000-Rechner an der Universität Hamburg aus. Danach müßten jährlich 120000 DM dafür aufgebracht werden.

Es kann nicht davon ausgegangen werden, daß das erste gefertigte Chip voll funktionsfähig ist, weil die Funktion der Schaltung innerhalb des Designprozesses nur simulativ überprüft werden kann. Mit Simulation kann man nur die Anwesenheit von Fehlern zeigen, nicht aber deren Abwesenheit. Nach aller Erfahrung, auch der mit Scheme-79 und Scheme-81, werden beim Entwurf Fehler gemacht, die erst nach Prototypfertigungen aufgedeckt werden. Die Dauer von der Abgabe des letzten Entwurfs für ein Reticle bis zum Erhalt der gefertigten Chips beträgt nach den bisherigen Erfahrungen 6 bis 8 Monate.

Für den Scheme-Prozessor folgt, daß er schnell bis zum ersten Prototyp zu entwerfen war, damit der Fertigungstermin im April 1989 wahrgenommen werden konnte. Wenn man von einem fehlerhaften Prototypen ausgeht, ist dann noch eine Modifikation und eine erneute Fertigung Ende 1989 oder Anfang 1990 möglich. Die Arbeiten an diesem Projekt wurden im Mai 1989 aufgenommen. Daher standen von der Erstellung einer Scheme-Implementation bis zur Abgabe eines fertig realisierten Chips nur elf Monate zur Verfügung.

# Kapitel 7

# Festlegung von Architektur und Befehlssatz

In den Kapiteln zuvor wurde beschrieben, daß ein Prozessor zu entwerfen war, der einen Scheme-spezifischen Bytecode ausführt. Die begrenzte Chipgröße, die Möglichkeiten und Grenzen eines Standardzellentwurfs und die restriktive Zeitplanung wurden erläutert. In diesen Kapitel sollen die gefundenen Lösungen und die Gründe für die Wahl dieser Lösungen dargelegt werden.

Die Repräsentationen der Objekte und die Speicherverwaltung werden hauptsächlich von der Programmiersprache Scheme bestimmt, sie werden unabhängig vom Befehlssatzes festgelegt und so gewählt, daß alle Objekte (Daten, Bindungsumgebungen und Controlframes) kompakt repräsentiert und effizient zugegriffen werden können. Die Repräsentation der Objekte wurde in Kapitel 5.4 beschrieben.

## 7.1 Register- oder Stackarchitektur

Eine Registerarchitektur verwendet einen Maschinenbefehlssatz, in dem die Befehle Operationen mit Quellen und Resultaten in Registern bezeichnen. Bei einer Stackarchitektur sind die Quellen der Operationen immer auf dem Stack, und auch das Resultat wird auf dem Stack abgelegt.

Die Wahl einer registerorientierten Architektur für Scheme wird von den Autoren von PC Scheme favorisiert, die einen registerorientierten Befehlssatz für 64 Universalregister ihrer virtuellen Maschine zugrunde gelegt haben [BaJe86]. PC Scheme übergibt alle Parameter bei Aufruf einer Funktion in diesen Registern. Die relativ große Geschwindigkeit des Compilers verdankt PC Scheme nach [BaJe86] besonders der Tatsache, daß wegen dieser großen Zahl von Registern keine rechenzeitintensive Berechnung der optimalen Nutzung einer kleinen Zahl von Registern erfolgen muß. Auch ein RISC-Prozessor für Smalltalk wurde von Ungar [Pend86] in Berkeley als Full-Custom-Chip mit 72 Universalregistern entworfen.

Der in dieser Arbeit beschriebene Scheme-Prozessor implementiert eine Stackarchitektur. Die Gründe für diese Entscheidung liegen in Beschränkungen des Entwurfssystems VENUS-S, das nur eine sehr geringe Anzahl von Registern zuläßt. Diese Problematik wird im folgenden erläutert.

### 7.1.1 Register mit VENUS-S

Die Entscheidung, ob ein mit VENUS-S entworfener Prozessor viele Register enthalten kann, muß berücksichtigen, daß es zwei Möglichkeiten zur Realisierung von Registern gibt: Es kann mit

dem RAM-Generator eine Registerbank als RAM erzeugt werden, und es lassen sich Register aus Standardzellen aufbauen.

Ein mit VENUS-S generiertes RAM hat den gravierenden Nachteil, daß es nur einen Port hat: Es kann zum gleichen Zeitpunkt nur unter genau einer Adresse entweder gelesen oder geschrieben werden. Ein Rechner mit Universalregistern hingegen muß aus zwei beliebigen Registern Operanden holen, diese verknüpfen und in ein drittes beliebiges Register schreiben. Für eine solche Operation werden Registerbänke mit zwei oder drei Ports eingesetzt, damit mindestens die beiden Operanden gleichzeitig zur Verknüpfung zur Verfügung stehen können. Mit dem Ein-Port-RAM von VENUS-S müßte eine solche Operation sequentiell mit drei RAM-Zugriffen erfolgen. Ob eine solche Lösung gewählt werden kann, hängt von der Zykluszeit des RAMs und seinem Platzbedarf ab. Die von VENUS-S generierten RAMs erwiesen sich als zu langsam und auch zu groß.

Die Verwendung mehrerer RAMs, in denen unterschiedliche Register, abhängig von den Operationen zwischen den Registern, allokiert werden, erwies sich aufgrund der restriktiven Chipgröße nicht als gangbarer Lösungsweg. Jedes RAM ist ein Makro, das bei der Chipkonstruktion durch zusätzliche Kanäle zu viel Fläche belegt.

Aus Standardzellen zusammengesetzte Register haben den Nachteil, daß sie wesentlich mehr Fläche belegen als ein generiertes RAM, sie haben aber den Vorteil, daß sich wegen des manuellen Erstellens beliebige Register erstellen lassen. Abschätzungen aus der Fläche der Standardzellen und Verdrahtungsfläche ergeben, daß eine Registerbank mit zwei Leseports und einem Schreibport aus Standardzellen bis zu etwa 12 bis 15 Registern realisierbar ist und auch die Geschwindigkeit in der Größenordnung eines einzigen RAM-Zugriffs liegt.

### 7.1.2 Wahl einer Stackarchitektur

Da die Registerbank aus Standardzellen zusammengesetzt sein muß und nur maximal 15 Register zur Verfügung stehen werden, gilt es, zu entscheiden, ob Universalregistern oder Stack der Vorzug gegeben werden soll. Es ist zu beachten, daß nicht alle 12 bis 15 Register als Universalregister verwendbar sind, weil neben den Universalregistern auch noch Register unabhängig von der gewählten Architektur für Programmzähler, Stackzeiger, Zeiger auf Ende des Stacks, Environmentzeiger, Zeiger auf den Codevektor, Zeiger auf globale Bindungen, Register für Adresse und Datum bei Hauptspeicherzugriffen sowie Register für die Speicherverwaltung vorzusehen sind. Alle diese Register werden so häufig verwendet, daß eine Auslagerung in den Hauptspeicher nicht sinnvoll ist. Wegen der viel zu geringen Zahl von verbleibenden möglichen Universalregistern wurde eine Stackarchitektur gewählt.

Eine Stackarchitektur kommt mit einer relativ geringen Anzahl von Registern aus. Ein weiterer Vorteil einer Stackarchitektur ist, daß ein Compiler, der stackorientierten Code erzeugt, einfacher zu erstellen und zu warten ist als einer, der registerorientierten Code erzeugt. Diese Vereinfachung des Compilers war bei der LISP-Maschine Symbolics 3600 das entscheidende Argument für die Wahl einer Stackarchitektur [Moon85].

## 7.2 Definition und Implementation eines Befehlssatzes

Ein stackorientierter Befehlssatz muß aus Gründen der Realisierbarkeit mit VENUS-S gewählt werden. Es verbleibt noch die Festlegung aller Befehle, die der Befehlssatz enthalten soll. Zuvor sollen aber die Register für den stackorientierten Befehlssatz angegeben werden.

### 7.2.1 Die Prozessorregister

Die Register des Scheme-Prozessors sind in der Tabelle 7.1 zusammengestellt. Im folgenden werden sie genau einzeln erklärt.

Die Breite eines Registers (1, 4, 24 oder 32 Bit) hängt von der Verwendung des jeweiligen Registers ab. Das Flag `tos_valid` ist nur ein Bit breit, die Zeiger sind 24 Bit breit, und Register, die wie `mdr` oder `tos` ein ganzes Wort aufnehmen, belegen die volle Breite von 32 Bit.

#### 7.2.1.1 Der Stack

Der Stack wird über einen Stackzeiger `sp` zugegriffen. Der Stackzeiger ist ein Register, das auf das erste belegte Wort im Stack zeigt. Der Stack ist ein kontinuierlicher Speicherbereich, in dem alle Parameter bei Funktionsaufrufen übergeben sowie alle Controlframes und Bindungsumgebungen angelegt werden. Controlframes und Bindungsumgebungen werden bei Bedarf in den Heap kopiert.

Der Stackbereich unterliegt nicht der Speicherverwaltung. Bei der Garbage Collection werden lediglich Referenzen aus dem Stack in den Heap auf neue Adressen abgeändert. Für die Garbage Collection und zum Kopieren einer Continuation in den Heap muß auch die Anfangsadresse des Stacks bekannt sein, sie ist wegen des relativ seltenen Zugriffs kein Prozessorregister, sondern liegt im Hauptspeicher.

Der Bytecode des Scheme-Prozessors ist stackorientiert; alle Befehle beziehen sich auf das oberste Objekt auf dem Stack. Um Hauptspeicherzugriffe zu sparen, wird das oberste Objekt auf dem Stack daher in einem Register, dem Register `tos`, gespeichert. Ein weiteres Register — `tos_valid` — gibt an, ob das oberste Stackobjekt im Register `tos` steht oder ob es sich im Hauptspeicher befindet.

#### 7.2.1.2 Die Environmentzeiger

Der Prozessor hält die Adressen der globalen Bindungsumgebung im Register `glo`. Sein Register `env` enthält immer die Adresse der Bindungsumgebung der gerade ausgeführten Funktion.

#### 7.2.1.3 Der Controlframezeiger

Der Controlframezeiger `ctrl` zeigt immer auf den aktuellen Controlframe. Der aktuelle Controlframe befindet sind immer im Stack und ist der Controlframe, der bei dem nächsten Funktionsaufruf oder der nächsten Rückkehr von einer Funktion verwendet wird.

#### 7.2.1.4 Der Funktionszeiger

Der Funktionszeiger `func` zeigt immer auf den Codevektor der Funktion, die gerade ausgeführt wird. Er wird indiziert, um Literale aus dem Codevektor auf den Stack zu laden. `Func` wird bei Aufrufen anderer Funktionen im Controlframe gespeichert, da Codevektoren bei der Garbage Collection verschoben werden.

#### 7.2.1.5 Der Programmzähler

Der Programmzähler `pc` zeigt auf den nächsten auszuführenden Bytecodebefehl in den Codevektor der ausgeführten Funktion. Die beiden niedrigwertigsten Bits geben das Byte an, die restlichen Bits die Adresse des Wortes, das den Bytebefehl, und noch drei weitere, enthält. Wenn der `pc` in einem Controlframe gesichert wird, steht in den höchstwertigsten 22 Bits nicht die absolute Adresse, sondern die Adresse relativ zum Codevektor. Nur diese relative Adresse bleibt bei Garbage Collections unverändert.

| Register | Breite | Erläuterung |
|---|---|---|
| `mdr` | 32 bit | Memory Data Register |
| `mar` | 24 bit | Memory Address Register |
| `tos` | 32 bit | Top Of Stack |
| `tos_valid` | 1 bit | Top Of Stack Flag |
| `sp` | 24 bit | Stackzeiger |
| `env` | 24 bit | Environmentzeiger |
| `glo` | 24 bit | Zeiger auf globales Environment |
| `ctrl` | 24 bit | Zeiger auf Controlframe |
| `func` | 24 bit | Zeiger auf Codevektor |
| `pc` | 24 bit | Zeiger auf Maschinenbefehl |
| `instr_cache` | 32 bit | 4 Bytes Code |
| `instr_addr` | 24 bit | Adresse des Codes in `instr_cache` |
| `free` | 24 bit | Zeiger auf Freispeicherbereich |
| `mem_end` | 24 bit | zur Prüfung auf Freispeicherüberlauf |
| `csr` | 4 bit | Control and Status Register |
| — | 32 bit | 2 Temporärregister |
| — | 24 bit | 3 Temporärregister |

Tabelle 7.1: Die Register des Scheme-Prozessors.

#### 7.2.1.6 Die Register der Speicherverwaltung

Die Speicherverwaltung verwendet die Register `free` und `mem_end`. Das Register `free` zeigt auf den Anfang des zu vergebenden Speicherbereichs. Bei einer Speicheranforderung wird `free` als Adresse des vergebenen Speichers zurückgegeben und anschließend um die Anzahl der angeforderten Speicherworte erhöht. Wenn nicht genug freier Speicher vorhanden ist — das ist daran erkennbar, daß `free` nach dieser Erhöhung größer als `mem_end` ist — erfolgt eine Garbage Collection. Für die Garbage Collection werden Daten wie Größe und Adressen der Speicherbereiche benötigt, die im Hauptspeicher abgelegt sind.

#### 7.2.1.7 Die Register für Speicherzugriffe

Ein Speicherzugriff erfolgt immer zwischen dem Hauptspeicher und dem Register `mdr`. Das Register `mar` enthält dabei die Hauptspeicheradresse.

#### 7.2.1.8 Der Instructioncache

Der Prozessor liest seinen Maschinencode in Worten von 32 Bits. Da der Maschinencode byteorientiert ist, werden mit jedem Hauptspeicherzugriff vier Befehle und Operandenbytes gelesen. Um die Anzahl der Hauptspeicherzugriffe möglichst gering zu halten, wird ein Wort im Register `instr_cache` gespeichert. Das Register `instr_addr` enthält die Hauptspeicheradresse, von der `instr_cache` gelesen wurde. Vor einem erneuten Lesen eines Wortes mit Bytecode wird `instr_addr` mit der Adresse, unter der gelesen werden soll, verglichen und das Lesen bei Gleichheit der Adressen nicht ausgeführt.

#### 7.2.1.9 Weitere Register

Neben den oben beschriebenen Registern enthält der Prozessor noch temporäre Register, die während der Garbage Collection und während des Abarbeitens eines Maschinenbefehls benötigt werden. Temporäre Register sind eingeführt, um während des Abarbeitens eines Maschinenbefehls auf das Auslagern und spätere Restaurieren von Registerinhalten in den Hauptspeicher verzichten zu können. Ferner enthält der Prozessor das Register `csr`, das zum Auslösen eines Hauptspeicherzugriffs oder Hostprozessorinterrupts sowie zur Abfrage, ob ein Hauptspeicherzugriff beendet wurde, eingesetzt wird.

#### 7.2.1.10 Verwendung der Register

Die Abbildung 7.1 zeigt die Verwendung der wichtigsten Register nach einem Aufruf einer Funktion `f` mit drei Parametern, die ihre Bindungsumgebung auf dem Stack hält. Ferner wird die Verschachtelung der Bindungsumgebung in den Controlframe deutlich. Es ist auch erkennbar, daß nur das Register `pc` auf ein Byte zeigt, während alle anderen Register auf ein Wort zeigen.

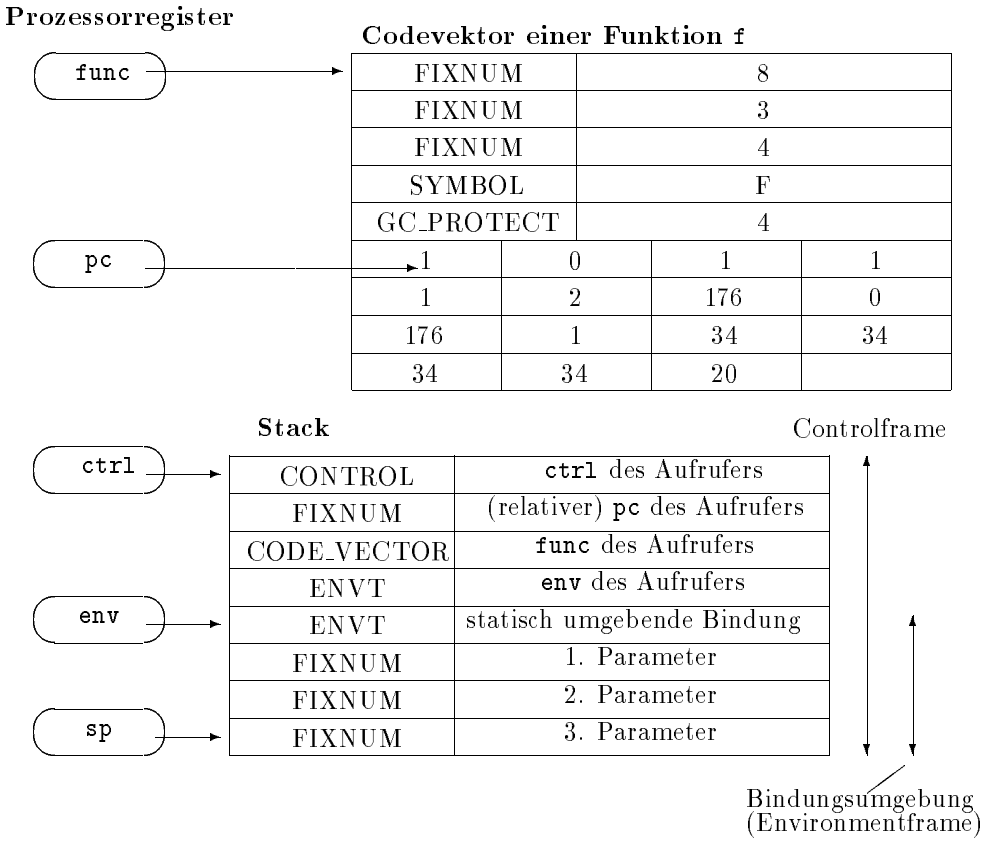



Abbildung 7.1: Die Register nach Aufruf einer Funktion mit drei Parametern.

### 7.2.2 Wahl der Befehle

Der stackorienterte Befehlssatz für den Scheme-Prozessor muß bestimmte Befehle aufweisen, die für das Abarbeiten von Scheme-Programmen unverzichtbar sind. Dazu zählen folgende Befehlsklassen:

- Befehle, die die Bindungen von Parametern auf den Stack laden und Befehle, die die Bindung von Parametern durch ein Datum vom Stack ersetzen. Diese Befehle entsprechen dem Lesen und Schreiben von Variableninhalten in imperativen Programmiersprachen.
- Befehle zum Aufrufen von Closures. Der Aufruf bei Endrekursion muß dabei anders als ein Aufruf innerhalb einer Sequenz von Ausdrücken behandelt werden.
- Bedingte Sprungbefehle, um die Sprachkonstrukte `if` und `cond` übersetzen zu können.
- Befehle, um eine Closure zu erzeugen. Ein solcher Befehl muß einen Codevektor mit der Bindungsumgebung der gerade ausgeführten Funktion zu einem neuen Objekt vom Typ `Closure` zusammenfassen. Der Befehl wird zur Implementierung eines $\lambda$-Ausdrucks benötigt.
- Befehle, um Standardprozeduren zu implementieren. Dazu gehören beispielsweise die Prozeduren `cons`, `car` und `cdr` ebenso wie Vektorprozeduren, arithmetische Prozeduren, String-, Char- sowie Ein-/Ausgabeprozeduren.

Eine wichtige Entscheidung war, wie komplex die einzelnen Befehle des Scheme-Prozessors sein sollen. Einerseits könnte für jede Standardprozedur ein Befehl vorgesehen sein, ein anderes Extrem wären einige wenige, relativ primitive Befehle. Aus Folgen dieser primitiven Befehle wären alle benötigten Standardprozeduren zusammenzusetzen.

Eine Entscheidung zwischen diesen Möglichkeiten ist die Entscheidung zwischen CISC (Complex Instruction Set Computer) und RISC (Reduced Instruction Set Computer) [Patt85].

In Pattersons Beitrag über RISC in den „Communications of the ACM“ [Patt85] werden folgende Gründe für RISC-Prozessoren genannt:

- Von den Befehlen und Adressierungsarten eines CISC-Prozessors werden relativ wenige sehr häufig genutzt. Eine Optimierung des Prozessors auf diese wenigen Befehle erlaubt eine wesentlich schnellere Ausführung dieser Befehle. Die entfallenen Befehle sind dann als Unterprogramme zu realisieren.

  Die schnellere Ausführung einiger weniger Befehle wird durch ein festverdrahteten Steuerwerks anstelle eines mikroprogrammierten, durch ein festes Instruktionsformat sowie durch die Verringerung der Speicherzugriffszeiten durch einen Cache erreicht.

- Eine Befehlspipeline bewirkt in RISC- und CISC-Prozessoren eine parallele Abarbeitung der Phasen Befehl holen, Befehl dekodieren, Operanden holen, Operation ausführen und Ergebnis speichern. Mehrere Befehle werden gleichzeitig in dieser Pipeline abgearbeitet, so daß jede dieser Phasen gleichzeitig für unterschiedliche Befehle erfolgt. Pipelining erlaubt eine Performanzsteigerung durch die bestmögliche Ausnutzung der vorhandenen Hardwareresourcen. Jedoch muß ein Prozessor verschiedene Konfiktfälle erkennen und behandeln können: Nach einem Sprungbefehl dürfen keine Befehle in der Pipeline ausgeführt werden, bevor die Sprungadresse errechnet und damit bekannt ist, woher weitere Befehle zu holen sind. Ferner müssen Datenabhängigkeiten zwischen den Befehlen in der Pipeline berücksichtigt werden. In dem Beispiel

  ```
  R0 := R1 + R2;
  R4 := R0 - R7;
  ```

darf die Rechnung R0 − R7 entweder erst nach Abschluß der Berechnung von R0 begonnen werden, oder der Prozessor erkennt den Konflikt und holt den Operanden statt aus R0 dorther (aus einem Temporärregister, ALU-Ausgangsregister oder Pipelineregister), wo das Ergebnis der Addition R1 + R2 steht.

In RISC-Prozessoren wird die Erkennung und Behandlung von diesen Konfliktfällen den Hochsprachencompilern überlassen[1]. Der Befehl nach einem Sprungbefehl wird immer aufgeführt, und der Compiler hat die Aufgabe, dort notfalls einen wirkungslosen Befehl (NO-OP) einzusetzen. In der überwiegenden Anzahl der Fälle kann der Compiler durch Umgruppierung von Instruktionen auch die Befehle nach Sprüngen und bei Datenabhängigkeiten nutzen. Der Prozessor kann für eine nie zu unterbrechenden Pipeline ausgelegt und dabei eine kürzere Taktzykluszeit erzielt werden.

- RISC-Prozessoren gehen davon aus, daß die Maschinenprogramme heutiger Rechner in virtuellem Speicher liegen. Die Länge der Programme ist daher kein zu minimierender Parameter mehr.

- RISC-Prozessoren verwenden einen registerorientierten Befehlssatz, bei dem alle Operationen Register als Operanden haben und es LOAD- und STORE-Befehle zum Transfer der Registerinhalte vom und zum Hauptspeicher gibt. Die Registertransfers werden vom Compiler minimiert. Der Nachteil dieser stark genutzten Register besteht darin, daß die Registerinhalte bei Prozeduraufrufen zu retten sind. Dazu gibt es Hardware- und Softwarelösungen: Die RISC-Prozessoren aus Berkeley verwenden einen Fenstermechanismus, bei dem der Prozessor mehrere Registersätze enthält, von denen bei Prozeduraufrufen ein Fenster auf einen neuen Registersatz verschoben wird [ChHe87, Milu87, SiMi87]. Diese Fenster überlappen sich, damit Parameter in Registern übergeben werden können. Solange die Verschachtelungstiefe der Prozeduren die Anzahl der Registerfenster nicht überschreitet, sind daher keine Registerinhalte in den Hauptspeicher zu retten. Softwarelösungen reduzieren Prozeduraufrufe, indem der Compiler globale Registeroptimierungen vornimmt oder (wo das sinnvoll ist) aufzurufende Prozeduren an der Aufrufstelle in den Code einkopiert (Inline-Expansion).

- Eine Verlagerung von häufig vorkommenden Maschinenbefehlsfolgen in ein Mikroprogramm wird von RISC-Befürwortern abgelehnt, weil fast jeder Befehl eines RISC-Rechners in einem Taktzyklus und damit genauso schnell wie eine Mikroinstruktion eines mikroprogrammierten Rechners ausgeführt wird. Ein Zeitunterschied des Faktors 10 zwischen einer Mikroinstruktion und einem Hauptspeicherzugriff besteht bei RISC-Prozessoren nicht mehr, da der Hauptspeicher aus Halbleiterspeichern statt Kernspeicher aufgebaut ist, die weitaus meisten Operanden in Registern stehen und die meisten Befehle im Cache angetroffen werden.

Für den Scheme-Prozessor wurde eine Realisierung als CISC-Prozessor gewählt. Viele der für RISC sprechenden Gründe sind hier nicht stichhaltig. Die Hauptgründe dafür liegen im geforderten Einsatz des Prozessors in einen 68000-Rechner und in den Randbedingungen durch das Entwurfssystem. Daß ein RISC-Befehlssatz für Scheme allgemein nicht sinnvoll ist, sollte daraus nicht geschlossen werden.

Ein Hauptspeicherzugriff des Scheme-Prozessors besteht im Lesen oder Schreiben eines Wortes von 32 Bit. Der Hostrechner kann nur 16 Bit breite Daten vom oder zum Hauptspeicher transferieren. Daher muß der Scheme-Prozessor für jeden Speicherzugriff zwei Hostspeicherzugriffe ausführen. Ferner muß er noch die Berechtigung anfordern, auf den Speicher zugreifen zu

[1] Praktisch alle RISC-Prozessoren führen noch Befehle nach Branchinstruktionen aus. Es gibt aber sowohl RISC-Prozessoren, die Datenabhängigkeiten in der Pipeline behandeln (IBM 801, Berkeley RISC I und II) als auch solche, die diese Aufgabe dem Compiler überlassen (Stanford MIPS) [ChHe87, Milu87, SiMi87].

dürfen. Insgesamt dauert ein Hauptspeicherzugriff des Koprozessors etwa eine Mikrosekunde. Eine Taktfrequenz von etwa 10 MHz wird beim Prozessorentwurf angestrebt, so daß ein Hauptspeicherzugriff mindestens 10 Taktzyklen beansprucht. Die in RISC-Prozessoren angewandten Techniken zur Reduktion von Hauptspeicherzugriffen — große Registerbänke für die Daten und Cache für die Instruktionen — sind beim Scheme-Prozessor nicht anwendbar. Die Anzahl der möglichen Register ist durch die Möglichkeiten des Entwurfssystems sehr beschränkt, wie am Anfang dieses Kapitels dargestellt ist.

Zudem ist die Länge eines Bytecodeprogramms im Gegensatz zu den Programmen von RISC-Prozessoren sehr kritisch. Es muß kompakt sein, weil der Mikrorechner keinen virtuellen und nur einen sehr begrenzten Speicher hat. Auch bei der Garbage Collection sind jeweils alle Codevektoren umzukopieren. Kompakter Bytecode beschleunigt also auch die Garbage Collection.

Daher wird ein Befehlssatz gewählt, der aus vielen, sehr mächtigen Befehlen besteht. Da die sehr langsamen Speicherzugriffe bevorzugt zu minimieren sind, werden damit nur relativ selten Befehle geholt. Der Prozessor wird mikroprogrammiert realisiert, und es werden so viele Primitivfunktionen wie möglich in das Mikroprogramm verlagert. Die beim Scheme-Prozessor im Mikroprogramm realisierten Standardprozeduren sind in Tabelle 7.2 gezeigt.

| Listenfunktionen | Numerische Funktionen | Typprädikate |
|---|---|---|
| cons | −1+ | null? |
| car | 1+ | pair? |
| cdr | + | number? |
| set-car! | − | symbol? |
| set-cdr! | * | char? |
| list | / | vector? |
| | quotient | string? |
| Vektorfunktionen | remainder | input-port? |
| make-vector | = | output-port? |
| vector-set! | < | eof-object? |
| vector-ref | <= | procedure? |
| vector-length | > | continuation? |
| vector | >= | |
| make-codevector | negative? | Charfunktionen |
| codevector-set! | zero? | char=? |
| | | char<? |
| Stringfunktionen | Sonstige Funktionen | char<=? |
| make-string | eq? | char>? |
| string-length? | not | char>=? |
| string-ref | unbound? | char→integer |
| string-set! | call-with-current-continuation | integer→char |

Tabelle 7.2: Im Mikroprogramm realisierte Standardprozeduren.

## 7.3 Literatur über Befehlssätze für LISP-Dialekte

Einige Befehlssätze für LISP-Dialekte sind veröffentlicht: Deutsch [Deut80] implementierte einen stackorientierten Befehlssatz für Interlisp [Teit74] auf einem mikroprogrammierbaren Minirechner. Deutsch beschreibt sowohl seinen Befehlssatz in [Deut80] als auch die statische und relative Häufigkeit der ausgeführen Befehle. Zudem haben Masinter und Deutsch lokale Optimierungen ihres Compilers, der den Bytecode erzeugt, beschrieben [MaDe80]. Wholey und Fahlman implementierten einen stackorientierten Befehlssatz für Common Lisp [Stee84] auf einem mikroprogrammierbaren Mikrorechner [WhFa84]. Sie erweiterten ihren Befehlssatz später allerdings um registerorientierte Befehle [WhFa84].

### 7.3.1 Der Befehlsatz von Schooler und Stamos

Die einzige detaillierte Beschreibung eines stackorientierten Bytebefehlssatzes für Scheme stammt von Schooler und Stamos [ScSt84]. Ihr „Vorschlag für eine Implementierung“ hat eine Implementierung mit einem Bytecodeinterpreter auf einem Apple Macintosh zum Ziel, die allerdings nie vollendet wurde [BaJe86]. Schooler und Stamos legen eine Maschine zugrunde, die Register für den Programmzähler, einen Zeiger auf die Bindungsumgebung, einen Zeiger auf den Literalbereich der gerade ausgeführten Prozedur und einen Stackzeiger hat. Sie legen alle Controlframes im Stack an und unterstützen Bindungsumgebungen in Stack und Heap.
Die Übernahme der Implementation von Schooler und Stamos kommt aus den folgenden Gründen nicht in Betracht:

- Continuations können damit nicht implementiert werden, da alle Controlframes auf dem Stack alloziert werden. Continuations können über die Rückkehr einer Prozedur hinaus lebendig sein und müssen deshalb im Heap alloziert werden.
- Um alle noch zugreifbaren Objekte bei der Garbage Collection ermitteln zu können, müssen weitere Register vorgesehen sein. Zum Beispiel ist nur aus dem PC, der in einen Bereich von Bytecode zeigt, nicht der Anfang und die Länge des Bytecodebereichs zu ermitteln.
- Die dort vorgeschlagenen Befehle `GetWord` und `SetWord` sollen `car` und `cdr` sowie Referenzen auf Vektorelemente implementieren. Dieses Vorgehen läßt die erforderlichen Typprüfungen außer acht.

Die hier beschriebene Implementation unterstützt Controlframes im Heap, wodurch die Befehle zum Aufruf und zur Rückkehr von Prozeduren komplizierter werden: Eine Continuation ist wie eine Prozedur ein Objekt erster Klasse, erst zur Laufzeit müssen die Objekte unterschiedlich behandelt werden. Denn der Compiler kann im allgemeinen Fall nicht entscheiden, ob eine Continuation oder eine Closure an einen bestimmten Bezeichner zur Laufzeit gebunden sein wird.

Das Register `func` erlaubt der Garbage Collection, in der Ausführung oder in Continuations befindliche Codevektoren zu ermitteln. Ferner enthalten alle Objekte variabler Länge (Bindungsumgebungen, Controlframes, Vektoren, Strings, ...) eine Kennzeichnung ihrer Länge, wenn sie im Heap alloziert sind. Es sind alle noch zugreifbaren Objekte ermittelbar.

`Car` und `cdr` werden wie Vektorreferenzen durch mikroprogrammierte Standardprozeduren implementiert, die alle Typprüfungen vornehmen.

Das Register `ctrl` vereinfacht den Aufruf und das Verlassen von Prozeduren. Beispielsweise brauchen hier die Befehle zum Aufruf die Anzahl der Parameter der aufgerufenen Prozedur im Gegensatz zu Schooler und Stamos [ScSt84] nicht mehr als Argument übergeben, die Anzahl der Parameter ergibt sich aus `ctrl` und `sp`. Es sind auch keine unterschiedlichen Befehle zur Rückkehr

von einer Prozedur nötig, die sich darin unterscheiden, ob die Prozedur ihre Bindungsumgebung auf dem Stack oder im Heap alloziert hatte.

## 7.4 Der Befehlssatz des Scheme-Prozessors

Die grundlegenden Überlegungen bei der Festlegung des Befehlssatzes für den Scheme-Prozessor werden auf den folgenden Seiten angegeben. Der vollständige Befehlssatz ist tabellarisch im Anhang A dargestellt.

### 7.4.1 Zugriff auf Variable und Konstante

Der Zugriff auf Variablen erfolgt für lokale Variablen, globale Variablen und Variablen auf dazwischenliegenden Ebenen durch unterschiedliche Befehle. Der Befehlssatz unterstützt 16 Ebenen zwischen den lokalen und globalen Variablen.

Jede Stackarchitektur benötigt Befehle, die Konstante auf den Stack bringen. Umfangreiche Untersuchungen ergaben, daß die in Scheme am häufigsten benötigten Konstanten die leere Liste, kleine Integerzahlen und die Boole'schen Konstanten sind. Daher werden in dem hier implementierten Befehlssatz diese häufig benötigten Konstanten nicht im Literalbereich des Codevektors abgelegt, sondern durch eigene Befehle auf den Stack gebracht.

### 7.4.2 Sprungbefehle und Verzweigungen

Die Sprungbefehle enthalten eine 12 Bit breite relative Adresse als Operanden. Sie können daher im Bereich −2048 bis 2047 springen. Die höchstwertigsten vier Bits der Adresse stehen im Bytecodebefehl, das restliche Byte folgt nach dem Befehl (Abbildung 7.2).

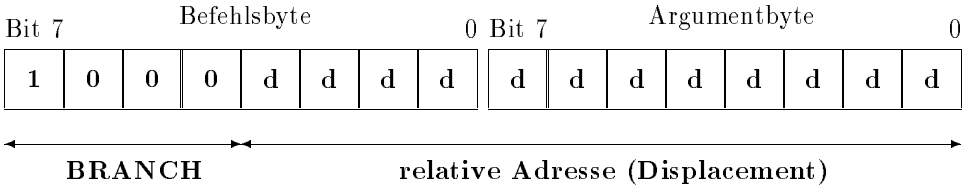



Abbildung 7.2: Format der Verzweigungsbefehle.

### 7.4.3 Aufruf und Rückkehr von Prozeduren

Der Aufruf von Prozeduren spielt eine zentrale Rolle in funktionalen Sprachen, weil der Funktionsaufruf nicht nur konzeptionell eine grundlegende Kontrollstruktur ist, sondern auch bei der Ausführung von Programmen besonders häufig erfolgt.

In Scheme muß der endrekursive Aufruf bei der Implementierung bei konstantem Speicherbedarf abgearbeitet werden [Rees86].

Ein Aufruf einer Prozedur erfolgt in dieser Implementation in vier Schritten:

1. Anlegen eines Controlframes mit dem Befehl `PUSHCONTROL`. Die Abbildung 7.3 zeigt den Stack, der in der Abbildung 7.1 auf Seite 61 angegeben wurde, nach Anlegen eines Controlframes.

2. Evaluieren der Argumente. Alle Argumente werden auf dem Stack übergeben, daher entsteht nach Evaluierung der Parameter in der Reihenfolge von links nach rechts eine Bindungsumgebung im Stack.

3. Evaluieren der Prozedur. Die Prozedur muß zu einer Closure oder zu einer Continuation evaluieren. In Abbildung 7.4 wurden zwei Parameter und die Closure einer Prozedur `g` evaluiert.

4. Sicherung des alten Zustandes im in Punkt 1 angelegten Controlframe und Einsprung in den Codevektor der aufzurufenden Prozedur. Die aufgerufene Prozedur überprüft die Anzahl der übergebenen Parameter und kopiert ihre Bindungsumgebung bei Bedarf in den Heap. In der Abbildung 7.5 wurde der alte Zustand im Controlframe gesichert, und die Prozedur `g` wird ausgeführt.

Wenn der auszuführende Prozeduraufruf unmittelbar vor der Rückkehr von einer Prozedur erfolgt (Endrekursion), entfällt das Anlegen eines Controlframes und das Sichern des alten Zustandes.

Das Anlegen eines Controlframes *vor* der Ablage der Parameter steht im Gegensatz den den bei der Implementierung von imperativen Programmiersprachen üblichen Gepflogenheiten. Diese Verfahrensweise erlaubt in Scheme einen endrekursiven Aufruf, bei dem die Anzahl der Parameter des Aufrufers und des Aufgerufenen unterschiedlich sein kann, ohne daß der Controlframe im Speicher verschoben werden muß.

Der Befehlssatz unterstützt den Prozeduraufruf durch Befehle zum Anlegen eines Controlframes und durch verschiedene Befehle (endrekursiv, nicht endrekursiv, Bindungsumgebung in Stack oder Heap) zum Aufruf einer Prozedur (vergl. Anhang A).

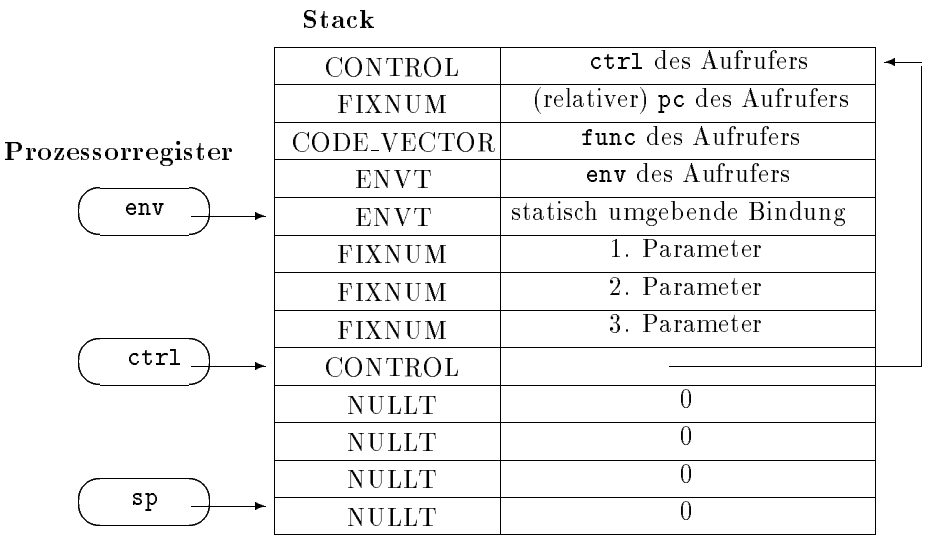



Abbildung 7.3: Der Stack nach Anlegen eines Controlframes.

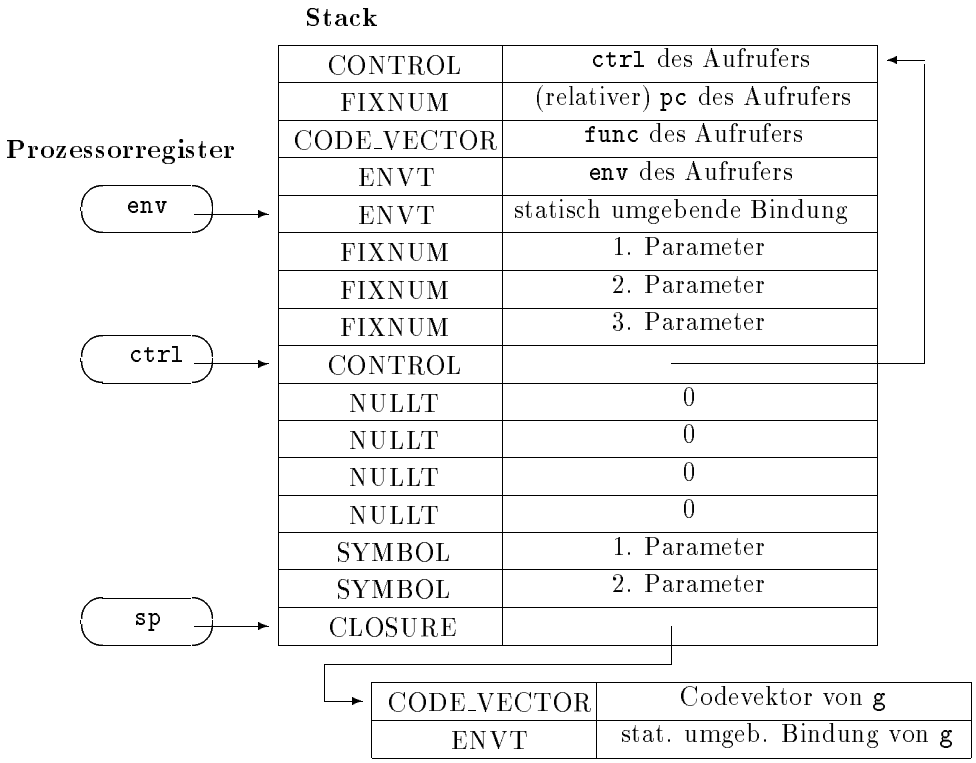



Abbildung 7.4: Der Stack nach Evaluieren von Parametern und Prozedur g.

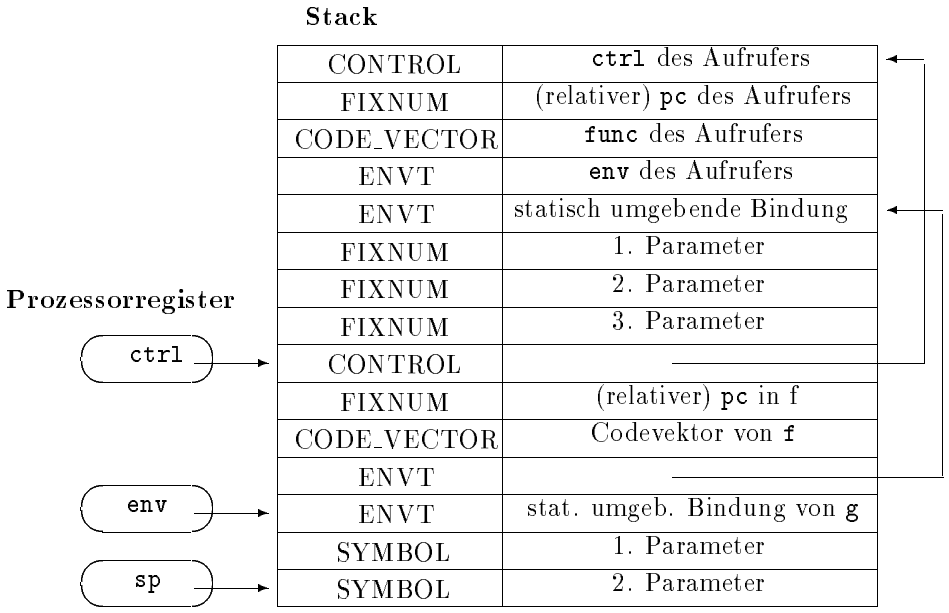



Abbildung 7.5: Der Stack nach Aufruf der Prozedur.

### 7.4.4 Typprüfungen und Vergleiche

Für Typprüfungen und Vergleiche wurde je ein Befehl gewählt, der den Typ und die Vergleichsoperation im Befehlsbyte enthält. Der Vergleichsbefehl bestimmt ferner, ob er die Typen der Operanden auf Zahlen oder Zeichen prüfen soll. Wenn zu vergleichende Zahlen nicht vom Typ `FIXNUM` sind, wird der Wirtsprozessor aufgerufen.

### 7.4.5 Befehle für Listen- und Vektoroperationen

Der Prozessor stellt Befehle für Listen- und Vektoroperationen bereit, die alle Typenprüfungen beinhalten und die CDR-Kodierung von Listen unterstützen. Bei allen Zugriffen auf Vektorelemente wird zudem geprüft, ob der Indexausdruck im Bereich des Vektors liegt. Von den Standardprozeduren `list` und `vector` sind je zwei Varianten implementiert: Eine Variante erlaubt bis zu 255 Argumente. Die anderen Variante erlaubt beliebig viele Argumente und auch eine erst zur Laufzeit errechnete Anzahl von Argumenten. Der Reader nutzt diese zweite Variante. Zusätzlich zu den Befehlen für Vektoren sind auch Befehle für die Konstruktion und Modifikation von Codevektoren enthalten. Diese Befehle werden vom Bytecode-Compiler genutzt.

### 7.4.6 Numerische Befehle

Die numerischen Befehle arbeiten derzeit nur mit Zahlen des Typs `FIXNUM`. Alle anderen Typen führen zu einer Fehlerbehandlung durch den Hostprozessor. Dieser kann daher Rechnungen mit Zahlen der Typen `REAL` und `BIGNUM` übernehmen, ohne daß dazu eine Erweiterung des Scheme-Prozessors nötig ist.

### 7.4.7 Zeichen- und zeichenkettenorientierte Befehle

Die zeichenorientierten Befehle erlauben die Wandlung zwischen den Typen `FIXNUM` und `CHART`. Alle Vergleiche werden mit `COMPARE` vorgenommen. Die zeichenkettenorientierten Befehle arbeiten mit Objekten des des Typs `STRINGT`. Alle anderen Typen führen zu einer Fehlerbehandlung durch den Hostprozessor. Auch beim Zugriff auf Elemente in Zeichenketten wird der Wert des Indexausdrucks geprüft.

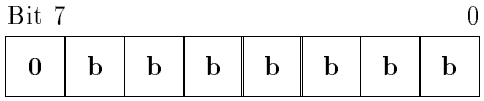
## 7.5 Repräsentation der Befehle

Die zuvor beschriebenen Bytecodebefehle sind wie folgt aufgebaut: Ein Byte enthält den Befehl. Optional ist in diesem Befehlsbyte ein Argument enthalten. Weitere Argumente stehen in den folgenden Bytes.
Es sind zwei Formate vorgesehen:

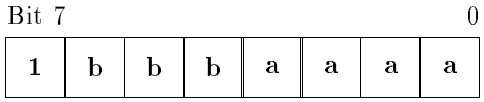
**Format A** Das Format A ist dadurch gekennzeichnet, daß das höchstwertigste Bit eines Befehlsbytes gelöscht ist. Die restlichen 7 Bits wählen einen aus 128 möglichen Befehlen aus.

**Format B** Im Format B ist das das höchstwertigste Bit eines Befehlsbytes gesetzt. Die Bits 0 bis 3 enthalten ein vier Bit breites Argument, und mit den Bits 4 bis 6 wird einer aus acht möglichen Befehlen ausgewählt.

Diese Befehlsformate werden in der Abbildung 7.6 dargestellt. Befehle im Format B sind `BRANCH`, `BRANCHF`, `BRANCHT`, `PUSHVAR`, `STOREVAR`, `COMPARE` und `TYPECHK`. Alle anderen Befehle sind im Format A repräsentiert.




**Format A: Nur Befehl im Befehlsbyte**




**Format B: Befehl und Argument im Befehlsbyte**

Abbildung 7.6: Die Befehlsformate des Scheme-Prozessors.

## 7.6 Beispiele für Programme in Bytecode

Der Befehlssatz muß sich einerseits für die Realisierung in einem Prozessor eignen. Andererseits müssen alle Scheme-Programme in Folgen von Bytecodebefehlen übersetzen lassen. Hier soll die Bytecoderepräsentation für einige Scheme-Programme beispielhaft angegeben werden.

Eine einfache Prozedur, die selbst nur die Standardprozedur für die Multiplikation aufruft, ist die Prozedur `square`:

```
(define (square x)
  (* x x))
```

Der Bytecode für `square` berücksichtigt, daß der Befehl `duplicate` einen Hauptspeicherzugriff weniger ausführt als der Befehl `pushlocal`, mit dem der gleiche Effekt zu erzielen wäre:

```
        pushlocal 0   ; Parameter x
        duplicate     ; verdopple Parameter
        mul           ; (* x x)
        return        ; zurück zum Aufrufer
```

Die folgende Prozedur zeigt eine bedingte Anweisung und die Darstellung in Bytecode:

```
  (define (nonneg x)
    (if (minus? x)
        0
        x))

        pushlocal 0   ; Parameter x
        minusp        ; (minus? x)
        branchf lab1  ; nein, ->
        pushpint 0    ; Konstante 0
        return        ; Funktionswert ist 0
lab1:   pushlocal 0   ; x
        return        ; Funktionswert ist x
```

Einen Aufruf einer Prozedur zeigt das nächste Beispiel:

```
  (define (test)
    (+ (f 1 2 3) 7))

        pushcontrol   ; Anlegen eines Controlframes
        pushpint 1    ; Konstante 1
        pushpint 2    ; Konstante 2
        pushpint 3    ; Konstante 3
        pushglobal f  ; f
        call          ; Aufruf der Prozedur
        pushpint 7    ; Konstante 7
        add           ; addiere 7
        return        ; fertig
```

Der Aufruf könnte auch endrekursiv sein. Es entfällt das Anlegen des Controlframes:

```
(define (test)
  (f 1 2 3))

      pushpint 1    ; Konstante 1
      pushpint 2    ; Konstante 2
      pushpint 3    ; Konstante 3
      pushglobal f  ; f
      stacktrcall   ; endrekursiver Aufruf der Prozedur
```

Ein letztes Beispiel zeigt eine Prozedur, die ihre Bindungsumgebung in den Heap kopiert:

```
(define (test x)
  (lambda (y) (* x y)))

      heapcontext 1 ; kopiere Bindungsumgebung in den Heap
      pushclosure 0 ; (lambda (y) (* x y)) ist Codevektor im Literalbereich
      return
```

# Kapitel 8

# Entwurf und Realisierung des Prozessors

Im vorherigen Kapitel wurde der Befehlssatz des Scheme-Prozessors und seine Register angegeben. In diesem Kapitel soll der Entwurf des Prozessors beschrieben werden.

Dazu werden die verschiedenen Entwurfs- und Beschreibungsebenen, auf denen der Prozessor entworfen wird, dargestellt. Ferner wird für jede Ebene auf die gewählte Beschreibung und Simulation der beschriebenen Schaltung eingegangen.

## 8.1 Entwurfsebenen

Ein Hardwarenentwurf beinhaltet verschiedene Sichten von Spezifikation und Entwurfsergebnissen. Diese Sichten umfaßen:

- Verhalten
- Struktur
- physische Schaltung (Layout bei ICs)

Die Struktur ist dabei eine Modellierung der physischen Schaltung. Die Sichten werden in verschiedene Abstraktionsebenen unterteilt:

- Architekturebene
- Algorithmenebene
- Funktionsblockebene (Registertransfer-Ebene)
- Logikebene
- Schaltkreisebene
- Geometrieebene

Diese Gliederung wird im Y-Diagramm von Gajski und Kuhn übersichtlich dargestellt [GeKu83, WaTh85]. Die Abbildung 8.1 zeigt ein Y-Diagramm. Im Y-Diagramm wird ein Entwurfsprozeß durch Pfeile symbolisiert, welche sich dem Mittelpunkt der Abbildung nähern. Die Abbildung 8.2

zeigt die mit VENUS-S erzielbare Entwurfsunterstützung im Y-Diagramm. VENUS-S geht von einer Strukturbeschreibung auf Logikebene aus. Durch die für die Standardzellen enthaltenen Simulationsmodelle steht über eine Simulation eine Verhaltensbeschreibung auf Logikebene zur Verfügung. Ferner wird durch die Chipkonstruktion eine Transformation der Struktur auf der Logikebene in die Geometrieebene einer physischen Schaltung erreicht.

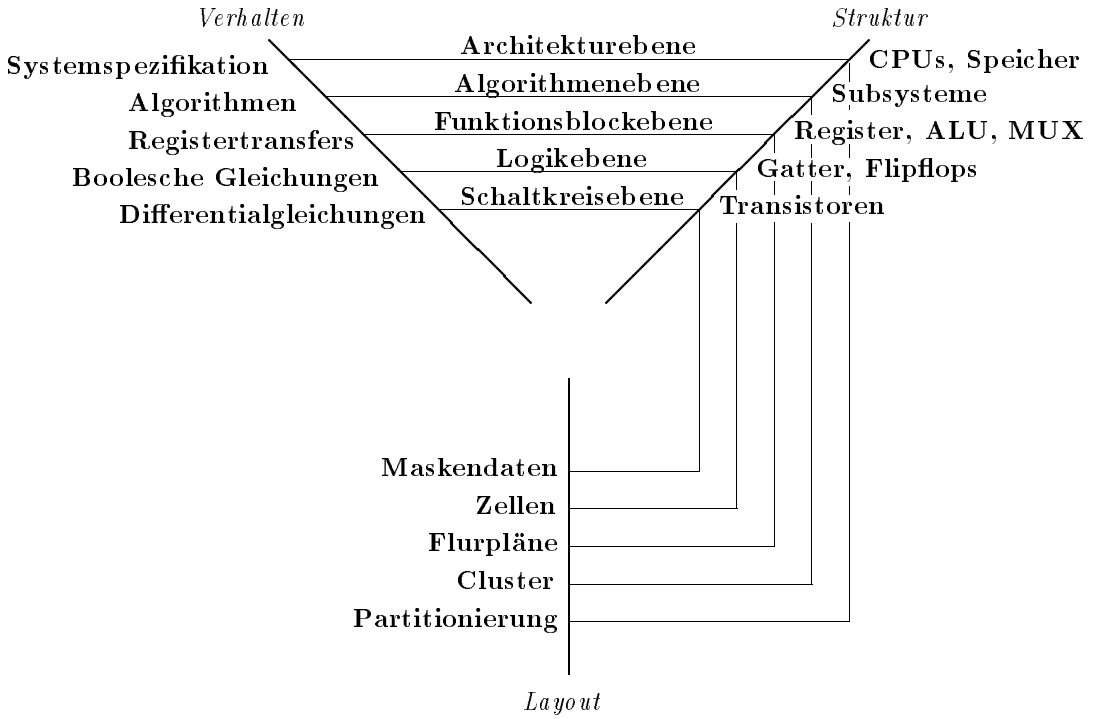



Abbildung 8.1: Sichten und Abstraktionsebenen bei Spezifikation und Entwurf (Y-Diagramm).

| | Verhalten | Struktur | Physische Realisierung |
|---|---|---|---|
| Architekturebene | Systemspezifikation | CPUs und Speicher | Paritionierung |
| Algorithmenebene | Algorithmen | Subsysteme | Cluster |
| Funktionsblockebene | Registertransfers | Register, ALUs, Mux | Flurpläne |
| Logikebene | Boolesche Gleichungen | Gatter, Flipflops | Zellen |
| Schaltkreisebene | Differentialgleichungen | Transistoren | Geometrie |

Tabelle 8.1: Tabellarische Form des Y-Diagramms.

Der durch VENUS-S im Entwurfsprozeß abgedeckte Anteil besteht in einer Transformation von der Strukturbeschreibung auf Logikebene in geometrische Maskendaten. Oberhalb der Logikebene erforderliche Entwurfsschritte werden nicht unterstützt.

Tabelle 8.1 stellt das allgemeine Y-Diagramm in tabellarischer Form dar. In Tabelle 8.2 wird diese Tabelle auf den speziellen Fall — den Entwurf des Scheme-Prozessors — angepaßt. Die

Schaltkreisebene ist dabei entfallen, da diese bei einem Semicustom-Entwurfssystem nur beim Entwurf der Zellen durch den Hersteller, nicht aber beim Entwurf durch den Anwender zu berücksichtigen ist. Das Gesamtsystem ist hier der 68000-Rechner mit eingesetztem Scheme-Prozessor. Der 68000-Rechner mit seinem Speicher und seiner Ein-/Ausgabeeinheit ist bereits fertig; zu entwerfen bleibt der Scheme-Prozessor, dessen Verhalten auf der Algorithmenebene durch den Befehlssatz und die Speicherverwaltung spezifiziert ist. Die nächst niedrigere Ebene, die Registertransferebene, geht von einer Struktur aus Multiplexern, ALUs und Registern aus und beschreibt das Verhalten durch die Registerinhalte und ALU-Operationen und durch Multiplexer ausgewählte Datenpfade. Das Verhalten ist bei einer mikroprogrammierten Struktur durch ein Mikroprogramm, das die Datenpfade auswählt, gegeben. Die unterste Entwurfsebene bei einem Semicustomentwurf ist die Logikebene. Die Struktur wird aus Standardzellen und Makrozellen gebildet, und das Verhalten ist durch Simulation der eingegebenen Struktur erkennbar. Diese Struktur auf Logikebene wird mit dem Entwurfssystem in eine physische Struktur, ein gefertigtes Chip, transformiert.

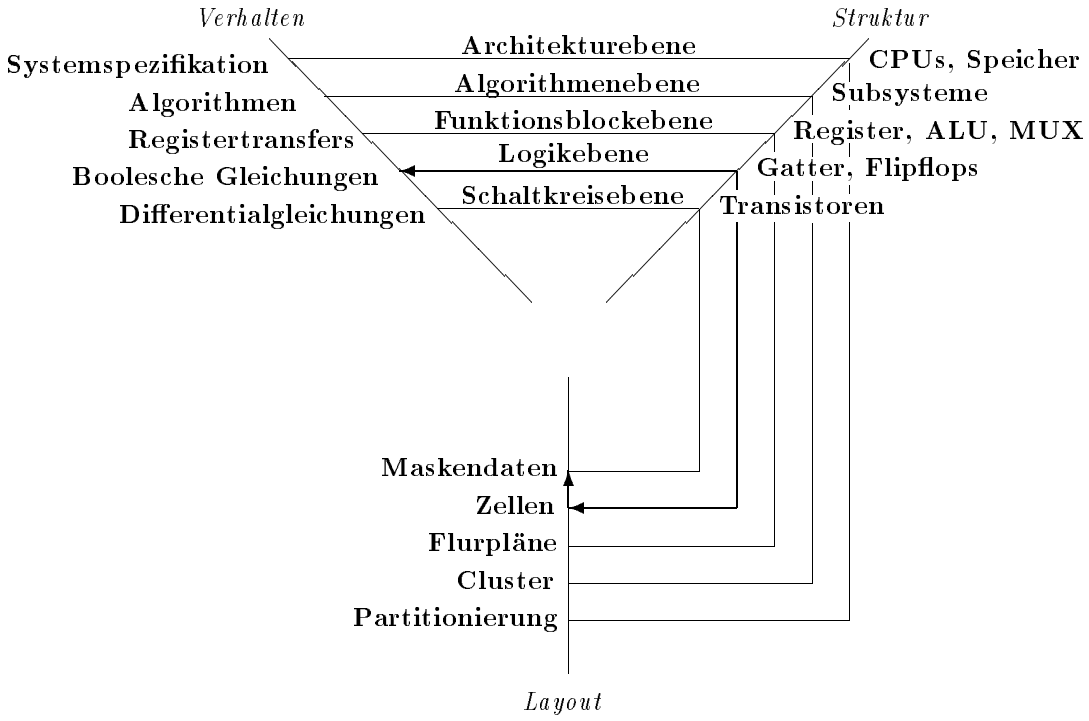



Abbildung 8.2: Beitrag von VENUS-S zum Entwurfsprozeß als Y-Diagramm.

Mehr um diesen Entwurf zentriert zeigt die Abbildung 8.3 das Vorgehen beim Entwurf des Scheme-Prozessors: Das gewünschte Verhalten wird durch den Befehlssatz spezifiziert. Dieser Befehlssatz muß nicht nur vom zu entwerfenden Chip auszuführen sein, er muß auch die Implementation von Scheme effizient und vollständig ermöglichen. Daher ist das Verhalten des Prozessors im Rahmen einer Scheme-Implementation simulativ auf seine Eignung zu untersuchen. Dieser Simulator ist auch ein Hilfsmittel, um Software für den Scheme-Prozessor vor Ende seiner Fertigung erstellen zu können.

Auf der nächsten Entwurfsebene wird eine Registertransfer-Struktur für den Scheme-Prozessor entwickelt und das Mikroprogramm, das das gewünschte Verhalten mit dieser Struktur ermöglicht,

erstellt. Simulativ wird das Verhalten von Mikroprogramm und RT-Struktur mit dem spezifizierten Verhalten verglichen. Diese Simulation gestattet auch eine erste Geschwindigkeitsabschätzung des Scheme-Prozessors, da die Laufzeit von Programmen als Vielfache von Taktzyklen ermittelt werden kann.

| | Verhalten | Struktur | Physische Realisierung |
|---|---|---|---|
| Architekturebene | 68000-Rechner mit Scheme-Prozessor | 68000, Speicher, Scheme-Prozessor | Paritionierung auf Platinen u. ICs |
| Algorithmenebene | Befehlssatz u. Speicherverwaltung | Steuerwerk, Operationswerk | Plazierung der Makros |
| Funktionsblockebene | Registertransfers, Mikroprogramm | Register, ALUs, MUX | Flurpläne |
| Logikebene | VENUS-Simulation | Standardzellen, ROM | Standardzellen, generiertes ROM |

Tabelle 8.2: Tabellarische Form des Y-Diagramms beim Entwurf des Scheme-Prozessors.

Auf der Logikebene entsteht eine Strukturbeschreibung durch die Eingabe der Schaltung in ein Semicustom-Entwurfssystem. Das Verhalten der Schaltung ist durch den Simulator des Entwurfssystems zugänglich. Wieder ist das Verhalten der Struktur auf dieser Ebene mit dem zuvor spezifizierten Verhalten zu vergleichen. Ferner ist das Zeitverhalten zu ermitteln, und es können Aussagen über kritische Pfade und die maximale Taktfrequenz getroffen werden. Die simulierte Schaltung dient auch der Fehlersimulation und Chipkonstruktion als Eingabe.

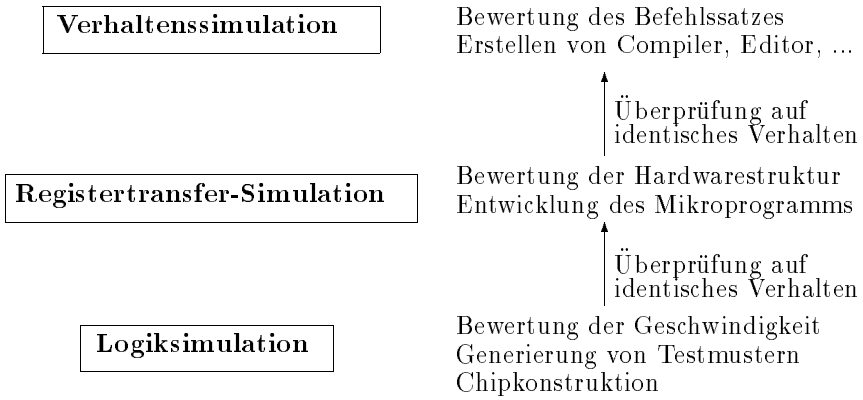



Abbildung 8.3: Die Entwurfsebenen des Prozessors.

## 8.2 Verhaltensbeschreibung des Scheme-Prozessors

Eine Verhaltensbeschreibung eines Prozessors besteht hauptsächlich in der Beschreibung der Wirkung seiner Befehle. Die Wirkung der Befehle ist die der Veränderung der Inhalte von Prozessorregistern und Hauptspeicher. Neben der Wirkung der Befehle werden das Holen der Befehle und die Interruptbearbeitung beschrieben.

Die Beschreibung des Verhaltens kann in dafür entworfenen Beschreibungssprachen erfolgen, für die es Compiler gibt, welche aus der Verhaltensbeschreibung einen Simulator erzeugen. Als Beschreibungssprachen für das Verhalten einer Hardware stehen an der Universität Hamburg ISPS, „Instruction Set Processor Specification“ von der Carnegie-Mellon University [Barb81] und Mimola von der Universität Kiel [Marw86] zur Verfügung.

Eine Beschreibung des Prozessors in Mimola erschien sehr attraktiv, da Mimola nicht nur einen Simulator für das beschriebene Verhalten bereitstellt, sondern auch automatisch eine mikroprogrammierte Hardwarestruktur generiert. Versuche mit Mimola ergaben, daß die zur Verfügung stehende Version von Mimola noch sehr unzulänglich war. Zudem wird das Mikroprogramm nur für eine Hardware ohne Pipelineing und Residuumkontrolle erstellt. Da Pipelining eine größere Geschwindigkeit und Residuumkontrolle eine Verkleinerung des Mikroprogramms bewirken, sollte auf diese beiden Verfahren nicht grundsätzlich verzichtet werden, gerade weil die resultierenden Vorteile gravierend waren.

Eine Beschreibung des Verhaltens wurde mit ISPS erstellt. Damit war der Befehlssatz beschrieben und konnte simuliert werden. Es war ein vorrangiges Ziel, einen Compiler von Scheme in diesen Befehlssatz zu erstellen, um sicherstellen zu können, daß jedes Scheme-Programm in diesem Befehlssatz repräsentierbar ist. Der durch ISPS generierte Simulator erwies sich allerdings als sehr ineffizient und unbequem. Insbesondere erlaubt der Simulator das Laden des Hauptspeichers und der Register nur mit als Zahlen darzustellenden Werten. Ein Scheme-System arbeitet mit Zeigern auf Objekte, wobei diese bei jeder Garbage Collection ihre Position im Speicher ändern. Es wurde ein Simulator benötigt, der mehr auf Scheme zugeschnitten war.

Als praktikablere Beschreibungssprache auf Befehlssatzebene hat sich Pascal erwiesen. Der Befehlssatz wurde durch ein Pascal-Programm beschrieben. Ferner enthält dieses Programm Funktionen wie die Ein- und Ausgabe, die nach Fertigstellung des Prozessors vom Wirtsprozessor bereitgestellt werden. Der in Pascal erstellte Simulator lädt eine Datei, die den in Bytecode übersetzten Scheme-Compiler sowie einige häufig benötigste Standardfunktionen enthält und arbeitet dann als compilierende Scheme-Implementation wie z. B. PC Scheme, indem ein Ausdruck vom Benutzer eingelesen wird, dieser erst in Bytecode übersetzt und dann ausgeführt wird. Der Simulator dient dazu, den im vorherigen Kapitel angegebenen Befehlssatz nebst dem zugehörigen Compiler zu entwickeln. Ferner wird er in einem vom Verfasser veranstalteten Programmierprojekt, in dem Studenten Software wie einen bildschirmorientierten Editor, einen Debugger und alle im Sprachstandard geforderten Funktionen für den Scheme-Prozessor erstellen, eingesetzt. Der Simulator ist ein Programm von etwa 4200 Zeilen Länge, das ursprünglich vom Verfasser auf einem Atari ST erstellt wurde, später für den Einsatz in der Lehre aber auch nach VAX/VMS portiert wurde. Die Verteilung der Zeilen auf die Bestandteile des Simulators zeigt Tabelle 8.3. Die Summe der Prozentzahlen ist kleiner als 100 %, weil Teile des Programms, insbesondere Deklarationen und Kommentare, nicht eindeutig einem der Bestandteile zugeordnet werden konnten. Der Compiler und die Standardfunktionen wurden vom Verfasser in Scheme implementiert und haben eine Länge von 1100 Zeilen. Der Bootstrap des Schemeanteils erfolgte mit PC Scheme.

| Bestandteil | Quellzeilen |
|---|---|
| Bytecodeinterpreter | 47 % |
| Initialisierung mit Urlader | 11 % |
| Hostfunktionen | 8 % |
| Debugger | 7 % |

Tabelle 8.3: Bestandteile des Verhaltenssimulators.

### 8.2.1 Analyse der Verhaltensbeschreibung

Für die Angabe der Struktur des Scheme-Prozessors wird die Verhaltensbeschreibung analysiert. Dazu werden die in dieser Beschreibung vorkommenden Kontrollstrukturen und Ausdrücke ermittelt. Die vorkommenden Ausdrücke bestimmen Anforderungen an die Hardwarestruktur, nämlich von einer ALU oder mehreren ALUs bereitzustellende Operationen. Ferner ist bei Operationen, die nur selten vorkommen, ein Ersetzen durch eine Folge von anderen, häufiger vorkommenenden Operationen vorzusehen. Beispielsweise enthält die Verhaltensbeschreibung nur einmal eine Division durch vier. Diese Division erfolgt bei der Implementierung der Funktion `make-string`, wenn aus der Länge des Strings die Anzahl der zu allozierenden Speicherworte errechnet wird. Da diese Funktion nach fundierten Abschätzungen einerseits nicht sehr häufig ausgeführt wird und andererseits auch nicht besonders zeitkritisch ist, wird diese einzige Division durch vier durch zwei Divisionen durch zwei ersetzt. Weitere Analysen betreffen die Anzahl der Operanden der vorkommenden Ausdrücke und ob diese Operanden Variablen oder Konstanten sind. Ferner sind die Variablen zusammengesetzte Daten (Records) mit den Feldern CDR-Code, Typ und Datum. Daher wird auch analysiert, ob Ausdrücke fast ausschließlich das gleiche Feld zugreifen oder ob sie unterschiedliche Teilfelder beinhalten. Auch bei diesen Analysen der Register und Felder werden auch Ausnahmen wie im obigen Beispiel mit der Division durch Folgen häufig vorkommenender Ausdrücke ersetzt. Ferner sind häufig ausgeführte Folgen von Anweisungen zu ermitteln, um diese besonders effizient in Hardware zu implementieren.
Die Analysen haben folgende Ergebnisse:

1. Kontrollstrukturen sind `IF-THEN-ELSE`, `IF-THEN` und `CASE`. Bei dem `CASE`-Statement wird nach 4 und 72 Zielen verzweigt. Diese Mehrfachverzweigungen erfolgen anhand des Bytebefehls und bei den Listenoperation und in der Garbage Collection anhand des CDR-Codes. Sie sind daher zeitkritisch und sollten nicht serialisiert werden. Zudem sind Unterprogrammtechniken in der Verhaltensbeschreibung stark genutzt, Rekursion tritt nicht auf, die größte Verschachtelungstiefe beträgt 5. Bedingten Sprüngen folgt häufig der Aufruf einer Prozedur und die Rückkehr von einer Prozedur.

2. Ausdrücke beinhalten häufig zwei Variablenreferenzen. Auch Konstanten sind häufig in Ausdrücken vertreten. Alle Ausdrücke $A$ lassen sich in der EBNF nach Wirth [Wirt81] durch in in Tabelle 8.4 gezeigte Grammatik beschreiben, in der Schlüsselworte durch Fettdruck hervorgehoben sind. Auch alle Vergleiche können durch diese Ausdrücke und Vergleich mit null realisiert werden. Die Konstanten liegen im Bereich $0 - 31$.

$$
\begin{aligned}
A &= L \mid L + R\,[+1] \mid L - R\,[-1] \\
L &= \mathit{Variable} \mid 0 \\
R &= R1 \mid 256 * R1 \mid R1\ \mathbf{MOD}\ 16 \mid R1\ \mathbf{DIV}\ 2 \\
R1 &= \mathit{Variable} \mid \mathit{Konstante}
\end{aligned}
$$

Tabelle 8.4: Grammatik in der Verhaltensbeschreibung vorkommender Ausdrücke $A$.

3. Verschiedene Felder der Variablen treten im gleichen Ausdruck auf. Das Ergebnis einer Zuweisung verändert sehr häufig nur ein Feld einer Variablen.

4. Besonders häufig auftretende Folgen von Anweisungen sind Abfragen der Variable `tos_valid` mit anschließendem Löschen oder Setzen von `tos_valid`, da sehr viele Befehle das oberste

Stackobjekt laden, wenn es nicht in der Variablen `tos` steht oder es in den Speicher transferieren, falls `tos` belegt ist. Zudem sind Zugriffe auf die obersten 22 Bit von `pc` sehr häufig, da das die Wortadresse des Bytecodebefehls bildet. Insbesondere finden häufig Vergleiche mit `instr_addr` statt. Ferner ist auch ein Zugriff auf ein Byte aus `instr_cache` häufig, da es sich dabei um Befehle und Operanden handelt.

Diese Ergebnisse der Analyse der Verhaltensbeschreibung sollen dem Entwurf der Struktur als Grundlage dienen.

## 8.3 Strukturbeschreibung des Scheme-Prozessors auf RT-Ebene

Auf der Registertransfer-Ebene (RT-Ebene) wird die Struktur des Prozessors, die das in der Verhaltensbeschreibung spezifizierte Verhalten realisieren soll, auf einer relativ hohen Ebene beschrieben. Die Struktur wird aus Speichern, Registern, Multiplexern und ALUs erstellt, wobei die Realisierung dieser Strukturelemente erst auf einer niedrigeren Ebene, der Logikebene, erfolgt. Die RT-Ebene erlaubt das Beschreiben verschiedener Hardwarestrukturen für ein gefordertes Verhalten, da die Anzahl der vom Entwerfer und Simulator zu betrachtenden Strukturelemente wesentlich kleiner als auf der Logikebene ist.

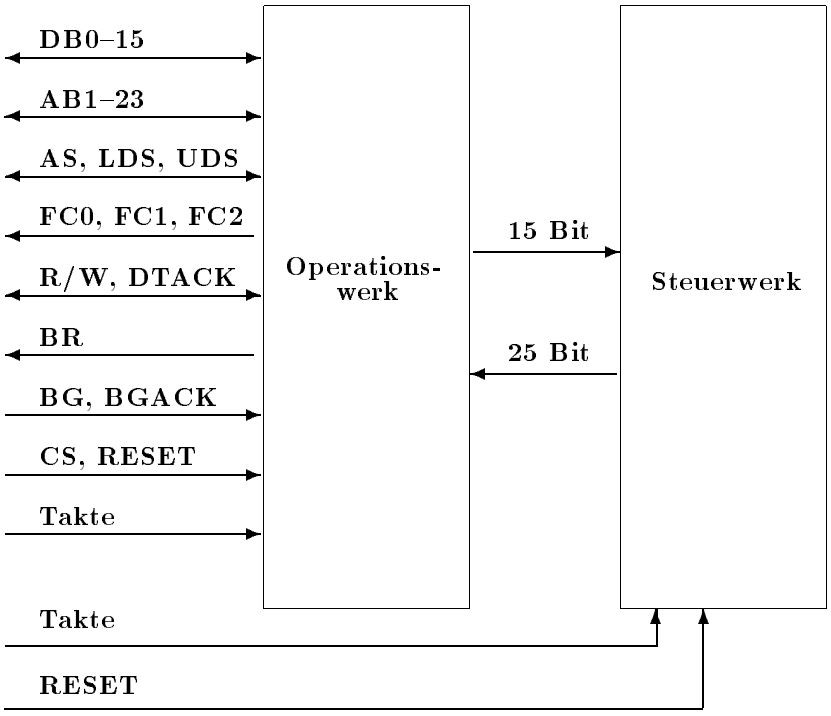



Abbildung 8.4: Operationswerk und Steuerwerk des Scheme-Prozessors.

Eine Struktur wird für ein vorgegebenes Verhalten weitgehend intuitiv, unter Rückgriff auf den Entwerfern bekannte Strukturen, gefunden. Van Cleemput [Clee79] schreibt dazu:

„...many designers base their designs on an library of examples. These examples may originate from previous design experience, from the literature, or from classroom exposure."

Dieses Zitat von 1979 trifft auch auf die heutige Situation zu. Darum soll das Problem hier systematischer angegangen werden. Zusätzlich muß bei der Angabe einer Struktur auf hoher Ebene auch berücksichtigt werden, daß die Strukturelemente durch die niedrigeren Ebenen bereitgestellt werden können.

Die Analyseergebnisse der Verhaltensbeschreibung sollen als Entscheidungskriterien für die Wahl zwischen unterschiedlichen Entwurfsmöglichkeiten beim Entwurf einer RT-Struktur dienen. Zusätzlich ist auch die Realisierbarkeit auf der Logikebene durch das Entwurfssystem zu berücksichtigen.

Ein komplexes Schaltwerk, wie es ein Prozessor darstellt, wird klassisch in Steuerwerk und Operationswerk aufgeteilt. Dieser in Abbildung 8.4 gezeigten Aufteilung folgend soll die Hardwarestruktur des Scheme-Prozessors in den folgenden Abschnitten entwickelt werden.

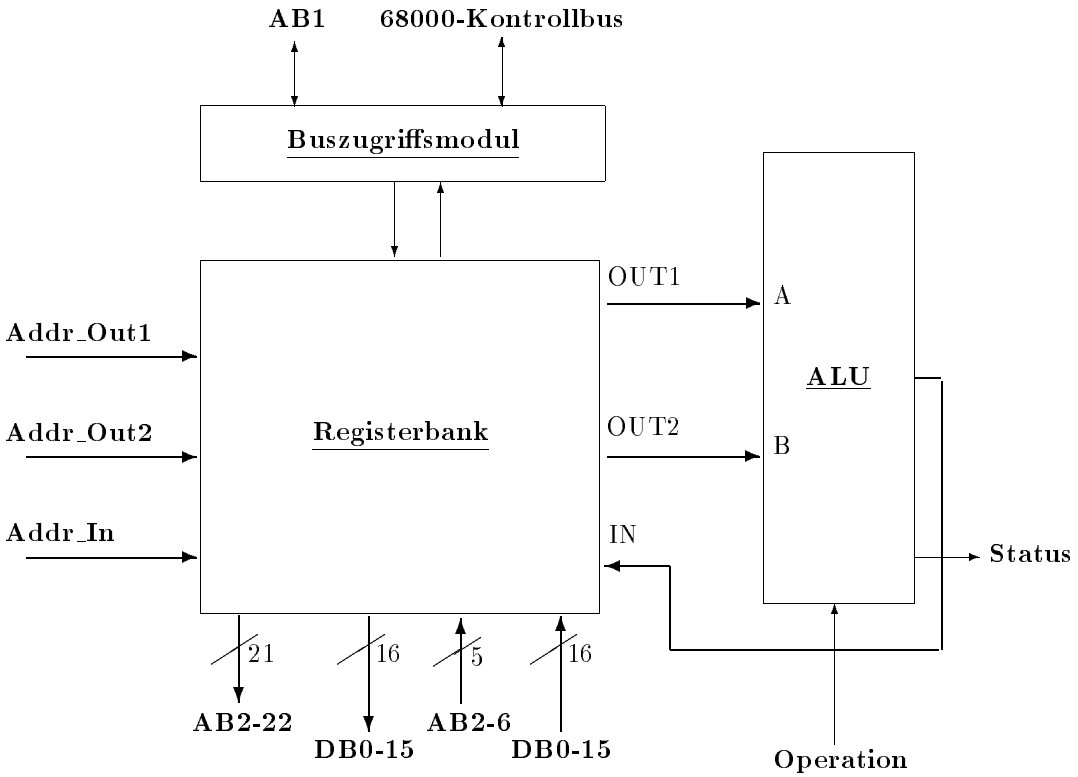



Abbildung 8.5: Schematische Struktur des Operationswerks.

### 8.3.1 Das Operationswerk

Die Struktur des Operationswerks wird schematisch in der Abbildung 8.5 dargestellt. Das Operationswerk besteht aus folgenden Subsystemen:

- Eine Registerbank enthält alle Register des Prozessors. Diese Register unfassen auch solche, die für Hauptspeicherzugriffe eingesetzt werden (`mar` und `mdr`). Daher ist die Registerbank auch mit dem Adreß- und Datenbus des 68000-Rechners verbunden (Anschlüsse unten links in der Abbildung 8.5).

- Eine ALU nimmt die eigentliche Verarbeitung aller Daten einschließlich der Übertragung der Daten zwischen verschiedenen Registern in der Registerbank vor.

- Ein Buszugriffsmodul wickelt Hauptspeicherzugriffe des Scheme-Prozessors ab und ermöglicht dem Hostprozessor Zugriffe auf die Register des Scheme-Prozessors. Der Buszugriffsmodul ist daher mit dem 68000-Kontrollbus und der Registerbank verbunden. In der Registerbank steuert er beispielsweise Übernahmen vom 68000-Datenbus in das Register `mdr`. Der Buszugriffsmodul arbeitet parallel zu Registerbank und ALU. Damit kann die für Hauptspeicherzugriffe benötigte Zeit genutzt werden, um Operationen und Typprüfungen vorzunehmen.

Die Strukturen dieser drei Subsysteme des Operationswerks werden im folgenden dargestellt. Mit der genauen Kenntnis dieser Subsysteme wird anschließend die Operationswerksstruktur detaillierter entwickelt und dabei auch auf die Verbindungen zum Steuerwerk und die Testbarkeit eingegangen.

#### 8.3.1.1 Die Registerbank

Die Analyse der Verhaltensbeschreibung ergibt, daß Ausdrücke sehr häufig zwei Variablen enthalten. In der Hardware befinden sich die Variablen in einer Registerbank. Wenn die Registerbank nur einen Ausgang hat, sind jeweils zwei Auslesevorgänge notwendig, bevor beide Operanden für einen Ausdruck vorliegen. Um Leseoperationen aus der Registerbank nicht zum Engpaß des Prozessors werden zu lassen, muß daher eine Registerbank mit zwei Ausgängen verwendet werden.

##### 8.3.1.1.1 Auswahl einzelner Felder der Register

Die Verhaltensbeschreibung verlangt, daß die Felder der Register ausgewählt werden können. Das Auswählen der Felder für eine Operation soll in der Registerbank erfolgen, womit die Adressierung der Register durch geordnete Paare bestehend aus der Adresse des Registers und dem Feld innerhalb des Registers erfolgt. Zusätzlich zu den drei Feldern CDR-Code, Typ und Datum kann auch das gesamte Register ausgewählt werden. Das gesamte Register wird für Zuweisungen zum Beispiel zwischen `mdr` und `tos` gebraucht.

Beim Schreiben in die Registerbank wird häufig nur ein Feld des gesamten Registers verändert. Es soll daher eine Registerbank entworfen werden, die auch Schreiboperationen durch ein Paar (Registeradresse, Feld) implementiert. Ohne einen solchen Mechanismus müßte zur Modifikation eines Feldes das gesamte Register gelesen, dann das Feld modifiziert und anschließend der neue gesamte Registerinhalt zurückgeschrieben werden. Dieser Zeitaufwand wird durch die hier angegebene spezielle Registerbank vermieden.

##### 8.3.1.1.2 Signale der Registerbank

Abbildung 8.6 zeigt alle Signale der Registerbank. Links sind die Adressen und Feldauswahlsignale für die drei Ports dargestellt. Jede Adresse eines Registers ist 5 Bit breit, da Tabelle 7.1 auf Seite 60 insgesamt 20 Register beinhaltet, zu denen unten noch Pseudoregister hinzukommen werden. Tabelle 8.5 zeigt die Belegung der Feldauswahlsignale.

Rechts werden der Eingangsport `IN` und die beiden Ausgängsports `OUT1` und `OUT2` gezeigt. Der Port `OUT2` hat nur eine Breite von 24 Bit. Die anderen beiden Ports sind 32 Bit breit.

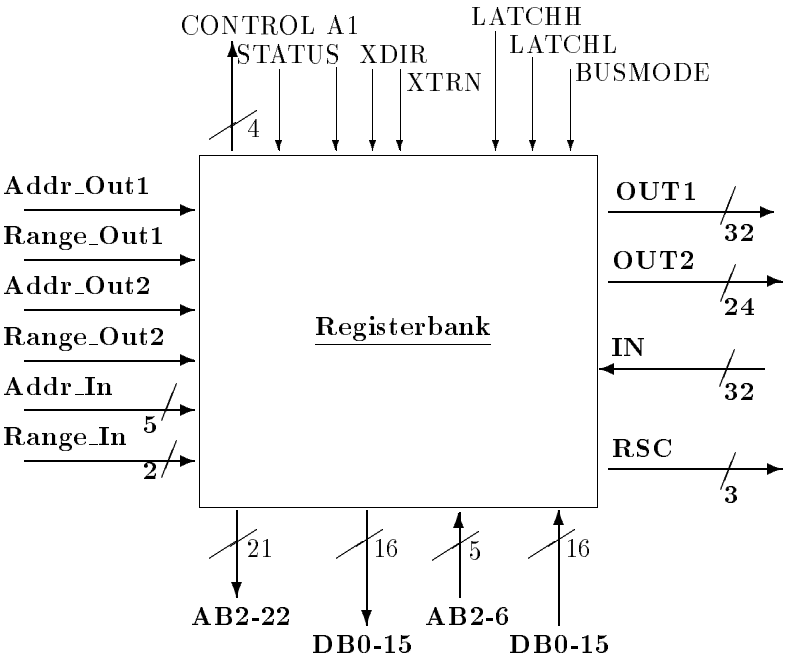



Abbildung 8.6: Die Signale der Registerbank.

| Belegung | Bitbereich | Feld |
|---|---|---|
| 00 | 31–0 | ganzes Wort |
| 01 | 23–0 | Datum, Pointer |
| 10 | 29–24 | Typ |
| 11 | 31–30 | CDR-Kode |

Tabelle 8.5: Operationen der Feldauswahl.

Die unten an der Registerbank dargestellen Signale dienen der Kommunikation des Scheme-Prozessors mit dem Wirtsrechner. Da die Registerbank die Register `mar` und `mdr` beinhaltet, die Adresse und Datum für Hauptspeicherzugriffe enthalten, müssen die Inhalte dieser Register an die Anschlüsse des Prozessorchips gelangen: Der Inhalt des Registers `mar` wird als `AB2-22` herausgeführt und bildet die Adresse bei Hauptspeicherzugriffen des aktiven Scheme-Prozessors. Der Inhalt des Registers `mdr` kann von den Anschlüssen `DB0-15` geladen und nach `DB0-15` ausgegeben werden. Zusätzlich werden die Signale `AB2-6` in die Registerbank hinein geführt. Diese Signale bilden eine Registeradresse, um die Register des inaktiven Scheme-Prozessors durch den Hostprozessor laden und auslesen zu können.

Oben an der Registerbank sind Steuerleitungen dargestellt. `A1` steuert, ob die oberen oder die unteren 16 Bits des Registers `mdr` von `DB0-15` übernommen oder dorthin ausgegeben werden. Für einen Zugriff des Hostprozessors auf die Register in der Registerbank werden die Leitungen `XDIR` und `XTRN` benötigt. Ein solcher Zugriff berücksichtigt, daß durch das Register `mdr` direkt vom Hostprozessor gelesen und geschrieben werden kann. Es muß zusätzlich ein Datentransfer zwischen `mdr` und dem Register, das der Hostprozessor zugreifen will, erfolgen. `XTRN` bewirkt einen solchen Datentransfer, und `XDIR` gibt an, ob der Inhalt von `mdr` in das vom Hostprozessor durch `AB2-6` ausgewählte Register geschrieben werden soll oder ob `mdr` aus diesem Register geladen werden soll. Die verbleibenen Register steuern die Übernahme in das Register `mdr`: `LATCHH` und `LATCHL` bewirken, daß die obere bwz. untere Hälfte von `mdr` aus `DB0-15` geladen wird. `BUSMODE` schaltet die Quelle von `mdr` zwischen `IN` mit der Adresse von `mdr` auf `DB0-15` um.

Einige Registerinhalte sind für später eräuterte interne Zwecke des Prozessors aus der Registerbank herauszuführen: `CONTROL` ist mit dem zuletzt in das Register `csr` geschriebenen Datum belegt. Die Belegung `STATUS` wird beim Lesen von `csr` als Ergebnis geliefert. `RSC` sind die untersten drei Bits des gerade ausgeführten Befehls.

#### 8.3.1.1.3 Ausgabe von Konstanten

Der Ausgang `OUT1` soll der Produktion für *L* und der Ausgang `OUT2` der Produktion für *R1* der Tabelle 8.4 entsprechen. In dieser Tabelle auf Seite 78 sind alle in der Verhaltensbeschreibung vorkommenden Ausdrücke dargestellt. Es fehlt noch eine Lösung, um die in diesen Produktionen angegebenen Konstanten an die ALU ausgeben zu können. Für die Null der Produktion *L* muß die Registerbank eine Null am Ausgang `OUT1` liefern können. Da die 20 Register nicht alle 32 möglichen Adressen belegen, kann eine Registeradresse spezifiziert werden, die die Ausgabe einer Null an `OUT1` bewirkt. Die Produktion *R1* verlangt die Ausgabe einer Konstanten zwischen 0 und 31 am Ausgang `OUT2` der Registerbank. Eine zusätzliche Spezifikation für die Registerbank ermöglicht auch dies ohne die Einführung zusätzlicher Steuerleitungen: Da der Ausgang `OUT2` nur 24 Bit breit ist, kann er kein volles 32 Bit Wort ausgeben. Der Feldauswahlcode für ein volles Wort soll an `Range_Out2` bewirken, daß die Adresse `Addr_Out2` als Konstante auf 24 Bit erweitert an `OUT2` ausgegeben wird. Damit kann die Registerbank auch alle benötigten Konstanten bereitstellen.

#### 8.3.1.1.4 Einführung von Pseudoregistern

Die Analyse der Verhaltensbeschreibung zeigt, daß einige besonders häufig auftretende Folgen von Anweisungen auftreten, die durch die Hardware besonders beschleunigt werden sollen:

- Abfragen der Variablen `tos_valid` mit anschließendem Löschen oder Setzen von `tos_valid`.
- Lesen und Schreiben der obersten 22 Bits von `pc`. Das sind Zugriffe auf die Wortadresse des Bytecodebefehls und deren Operanden.
- Zugriff auf ein Byte aus `instr_cache`. Lesender Zugriff für Bytecodebefehls und deren Operanden, schreibender Zugriff bei der Funktion `string-set!`.

Auch diese Operationen können mit geringem Implementierungsaufwand in die Registerbank verlagert werden: Es werden Pseudoregister eingeführt, die keine zusätzlichen speichernden Elemente der Registerbank darstellen, sondern Teile schon vorhandener Register auslesen oder modifizieren. Folgende Pseudoregister werden den schon vorhandenen Registern hinzugefügt:

`test_and_set` Dieses Pseudoregister kann nur gelesen werden. Es gibt den Inhalt von `tos_valid` aus und überschreibt ihn mit logisch eins.

`test_and_clear` Dieses Pseudoregister kann nur gelesen werden. Es gibt den Inhalt von `tos_valid` aus und überschreibt ihn mit logisch null.

`pc4` Dieses Pseudoregister kann gelesen und geschrieben werden. Beim Lesen werden die obersten 22 Bits von `pc` ausgegeben. Beim Schreiben werden nur die obersten 22 Bits von `pc` modifiziert; die untereren beiden Bits bleiben erhalten.

`cur_byte` Dieses Pseudoregister kann gelesen und geschrieben werden. Beim Lesen wird eines der vier Bytes aus `instr_cache` ausgegeben. Das auszugebende Byte wird durch die untersten beiden Bits von `pc` ausgewählt. Beim Schreiben wird das durch die untersten beiden Bits aus `pc` ausgewählte Byte aus `instr_cache` überschrieben.

Damit ist die Registerbank vollständig spezifiziert. Tabelle 8.6 führt alle Register und Pseudoregister mit ihren Adressen auf.

#### 8.3.1.2 Die ALU

Die ALU nimmt die eigentliche Datenmaniputation vor. Die ALU muß alle Operationen bereitstellen, deren Grammatik in Tabelle 8.4 auf Seite 78 angegeben ist. Die Operationen sind Addition und Subtraktion mit einem optionalen Übertrag, wobei der zweite Operand zuvor den dort in der Produktion für $R$ angegebenen Schiebe- und Maskierungsoperationen zu unterziehen ist.

Eine Hardwarerealisierung besteht daher aus einem Shifter, der die Schiebe- und Maskierungsoperationen ausführt, einem Komplementer, der zusammen mit dem Addierer eine Subtraktion ermöglicht, und einem Addierer. Abbildung 8.7 zeigt den Aufbau dieser ALU. Die steuerbaren Operationen von Addierer und Komplementer sind in Tabelle 8.7, die des Shifters in Tabelle 8.8 angegeben. Die Statusleitungen `zero`, `negative` und `overflow` ermöglichen Vergleiche und die Überlauferkennung.

#### 8.3.1.3 Der Buszugriffsmodul

Der Buszugriffsmodul wickelt die Buszugriffe des Scheme-Prozessors (bzw. der Registerbank) auf den 68000-Bus ab. Es ist ein autonomes Schaltwerk, das parallel zum dem Steuerwerk des Scheme-Prozessors arbeitet. Der Buszugriffsmodul wickelt einerseits von Steuerwerk des Scheme-Prozessors angestoßene Buszugriffe ab; andererseits ist der Scheme-Prozessor aus der Sicht des Hostprozessors ein Peripheriebaustein, dessen Register vom Hostprozessor wie Speicher gelesen und geschrieben werden können. Auch diese Lese- und Schreibzugriffe des Hostprozessors werden vom Buszugriffsmodul abgewickelt.

Der Buszugriffsmodul ist mit den Steuerleitungen des 68000-Busses [68000], die an den Scheme-Prozessor geführt werden müssen (in Abbildung 5.2 auf Seite 39 gezeigt) und mit Steuerleitungen der Registerbank (in Abbildung 8.6 gezeigt) verbunden.

Der Buszugriffsmodul wird mit seinen Verbindungen mit der Registerbank und mit der Außenwelt in Abbildung 8.8 gezeigt. Auch die Verbindungen der Registerbank mit der Außenwelt sind dort dargestellt.

| Adresse | Register | Breite | Erläuterung |
|---|---|---|---|
| 0 | Konstante 0 | 24 bit | für Ausgang `OUT1` |
| 1 | `tos` | 32 bit | Top Of Stack |
| 2 | — | 32 bit | Temporärregister |
| 3 | — | 32 bit | Temporärregister |
| 4 | `mdr` | 32 bit | Memory Data Register |
| 5 | `instr_cache` | 32 bit | 4 Bytes Code |
| 6 | `cur_byte` | 8 bit | Pseudoregister |
| 7 | `pc4` | 22 bit | Pseudoregister |
| 8 | `pc` | 24 bit | Pointer auf Maschinenbefehl |
| 9 | `tos_valid` | 1 bit | Top Of Stack Flag |
| 10 | `mar` | 24 bit | Memory Address Register |
| 11 | `test_and_set` | 1 bit | Pseudoregister |
| 12 | `test_and_clear` | 1 bit | Pseudoregister |
| 13 | `csr` | 4 bit | Control and Status Register |
| 14 | `mem_end` | 24 bit | zur Prüfung auf Freispeicherüberlauf |
| 15 | `instr_addr` | 24 bit | Adresse des Codes in `instr_cache` |
| 16 | `free` | 24 bit | Pointer auf Freispeicherbereich |
| 17 | `env` | 24 bit | Environmentpointer |
| 18 | `glo` | 24 bit | Pointer auf globales Environment |
| 19 | — | 24 bit | Temporärregister |
| 20 | — | 24 bit | Temporärregister |
| 21 | — | 24 bit | Temporärregister |
| 22 | `sp` | 24 bit | Stackpointer |
| 23 | `ctrl` | 24 bit | Pointer auf Controlframe |
| 24 | `func` | 24 bit | Pointer auf Codevektor |

Tabelle 8.6: Register und Pseudoregister in der Registerbank.

| Negate | Carry | Operation |
|---|---|---|
| 0 | 0 | $a + b$ |
| 0 | 1 | $a + b + 1$ |
| 1 | 0 | $a - b - 1$ |
| 1 | 1 | $a - b$ |

Tabelle 8.7: Operationen von Addierer und Komplementer.

| Belegung | Operation |
|---|---|
| 00 | identische Op. |
| 01 | DIV 2 |
| 10 | MOD 16 |
| 11 | $\times$ 256 |

Tabelle 8.8: Operationen des Shifters.

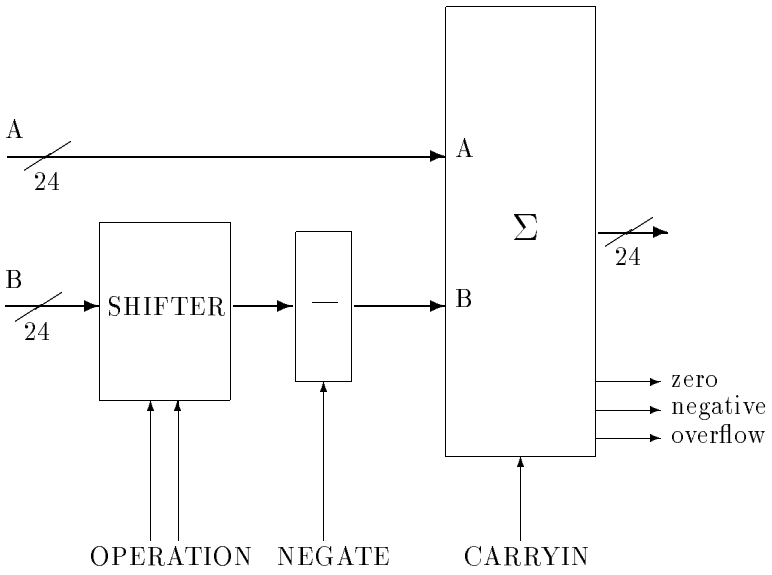



Abbildung 8.7: Struktur der ALU.

Die dargestellten Signale des 68000-Busprotokolls werden in dieser Arbeit nicht weiter erläutert. Diese sind in der Literatur über den 68000-Prozessor detailliert beschrieben [68000]. Hier noch zu erläutern sind die Signale `CONTROL` und `STATUS` zwischen Buszugriffsmodul und Registerbank: `CONTROL` ist mit dem Inhalt des Registers `csr` belegt, von dessen 4 Bits eines den Interruptpin des Scheme-Prozessors, mit dem er den Hostprozessor unterbrechen kann, steuert. Zwei Bits steuern, wie in Abbildung 8.8 gezeigt, den Buszugriffsmodul. Sie geben Start (S) und Richtung (D) für einen Buszugriff an. Das vierte Bit bewirkt ein Anhalten des Steuerwerks, es hat daher in dieser Abbildung keine Funktion. `STATUS` gibt an, ob der Buszugriffsmodul beschäftigt ist, oder ob er mit einem neuen Buszugriff beauftragt werden kann.

#### 8.3.1.4 Das gesamte Operationswerk

Das Operationswerk besteht aus den zuvor vorgestellten Moduln Registerbank, ALU und Buszugriffsmodul. Abbildung 8.9 zeigt alle diese Moduln und ihre Verbindungen untereinander. In dieser Abbildung sind gegenüber Abbildung 8.6 die Bitbreiten der Busse `OUT1` und `IN` von 32 auf 24 verringert. Die höchstwertigsten 8 Bits dieser beiden Signale sind miteinander zu verbinden. Diese Verbindung ist in dieser Abbildung in die Registerbank verlagert worden, wo sie auch bei der Realisierung als Chip erfolgen wird. Eine Zuweisung eines 32 Bit Registers in ein anderes ist weiterhin in einer Operation möglich: Die ALU addiert eine Null auf die unteren 24 Bits, und die oberen 8 Bits sind direkt verbunden.

##### 8.3.1.4.1 Verbindungen zum Steuerwerk

Das Operationswerk ist bereits diskutiert. Zusammen mit dem Steuerwerk ist es in Abbildung 8.4 auf der Seite 79 dargestellt. Das Steuerwerk erhält 15 Statusleitungen vom Operationswerk, und es steuert 25 Leitungen des Operationswerks. Die 25 Leitungen, die das Operationswerk steuern, sind die in Abbildung 8.9 links an der Registerbank dargestellen 21 Eingänge sowie die vier Steuerleitungen der ALU, die in der gleichen Abbildung unterhalb der ALU gezeigt werden. Das

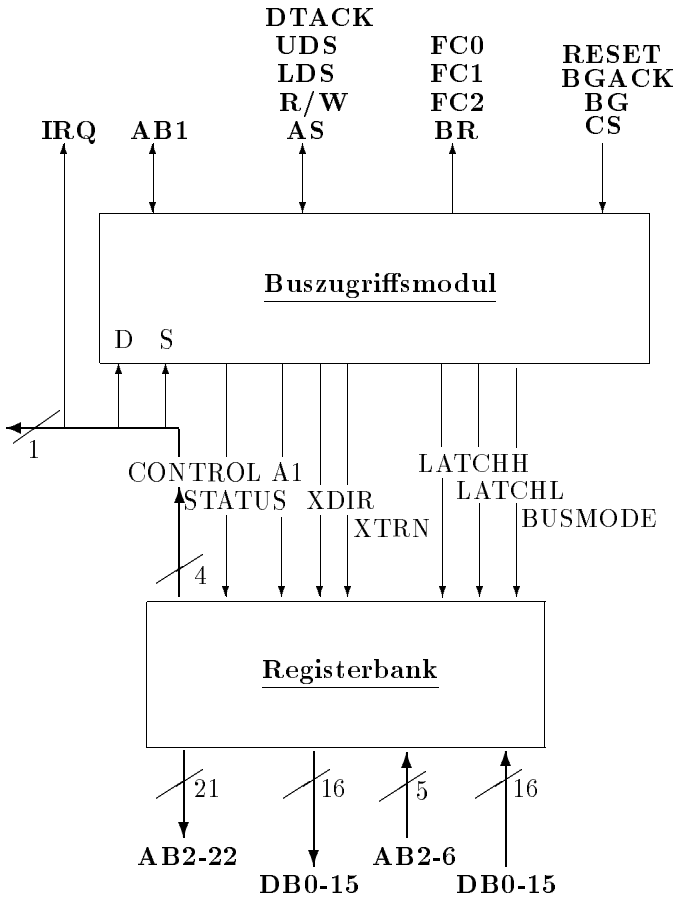



Abbildung 8.8: Buszugriffsmodul und Registerbank sowie die Verbindungen des Scheme-Prozessors zur Außenwelt.

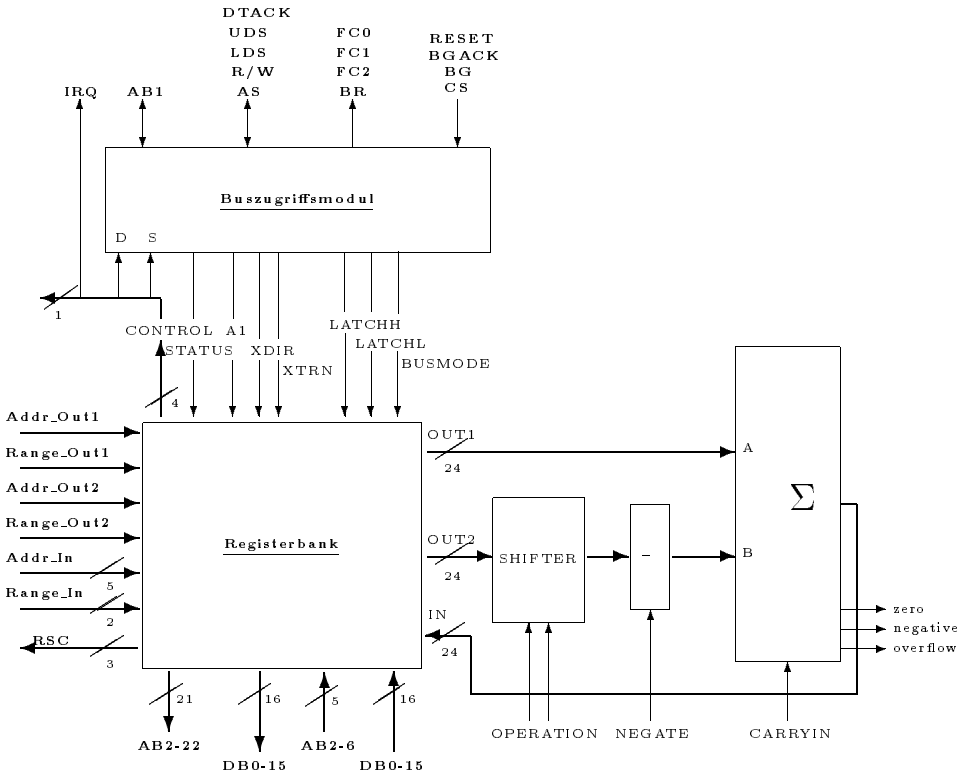



Abbildung 8.9: Das Operationswerk des Scheme-Prozessors.

Steuerwerk erhält vom Operationswerk die 3 Statusleitungen der ALU (zero, negative und overflow), den 3 Bit breiten Bus `RSC` aus der Registerbank, die Leitung `HALT` aus dem Bus `CONTROL` sowie die untersten 8 Bits des Busses `OUT2` für Befehle und Mehrfachverzeigungen.

#### 8.3.1.4.2 Testbarkeit des Operationswerks

Die Testbarkeit des Operationswerks läßt sich ermitteln, indem die Testbarkeit seiner Teilstrukturen gezeigt wird und überprüft wird, ob Testmuster an diese Teilstrukturen angelegt werden können und die Ausgaben dieser Teilstrukturen an die Ausgänge des Operationswerk propagiert werden können.

Die Registerbank kann vollständig getestet werden: Unter Nutzung des Buszugriffsmoduls ist das Register `mdr` steuerbar und beobachtbar im Sinne der Testbarkeit. Ferner kann der Inhalt des Registers `mdr` in jedes andere Register geschrieben und `mdr` von jedem anderen Register geladen werden. Die Feldauswahllogik kann für beide Ports beim Auslesen und Leiten des Ergebnisses über die ALU getestet werden. Beim Einschreiben in ein beliebiges Register ist sie ohnehin testbar, weil die Registerinhalte beobachtbar sind. Die ALU ist testbar, da beliebige Belegungen an die Eingänge angelegt werden können und das Ergebnis einer ALU-Operation in ein Register eingeschrieben werden kann und damit beobachtbar ist. Es ist auf dieser hohen Entwurfsebene noch nicht zu entscheiden, ob der Buszugriffsmodul testbar ist. Der Buszugriffsmodul ist ein Schaltwerk, das autonom arbeitet. Der Zustand ist anhand der erzeugten Signale erkennbar und Zustandswechsel erfolgen abhängig von Eingangssignalen des 68000-Kontrollbusses. Inwieweit es durch Steuerung und Beobachtung dieser Signale vollständig testbar ist, kann erst nach Entwurf auf Logikebene eine Fehlersimulation erweisen. Wenn die Testbarkeit nicht ausreichend ist, können die speichernden Elemente des Buszugriffsmoduls in einen Scanpath aufgenommen und damit steuerbar und beobachtbar gemacht werden.

Diese Ermitlung der Testbarkeit des Operationswerks basiert darauf, daß die vom Steuerwerk kontrollierten Leitungen steuerbar und die Statusleitungen, die vom Operationswerk zum Steuerwerk führen, beobachtbar sind. Nochmals: Das Steuerwerk erfüllt diese Voraussetzung.

### 8.3.2 Das Steuerwerk

Eine mikroprogrammiertes Steuerwerk war schon aufgrund des komplexen Befehlssatzes zu wählen. Aus der Analyse kommt die Forderung von Verzeigungen nach Vergleichen (in der Hardware nach Statussignalen einer ALU) und Mehrfachverzweigungen zu 4 bis 72 Zielen hinzu. Auch sind Unterprogrammtechniken in der Verhaltensbeschreibung reichlich genutzt; sie sollen auch im Mikroprogramm bis zur ermittelten Verschachtelungstiefe von fünf möglich sein.

Die RT-Struktur des Steuerwerks zeigt Abbildung 8.10. Das Steuerwerk enthält ein Mikroprogramm-ROM mit 1024 Worten von 39 Bit Breite. Eine solches mit dem ROM-Generator aus VENUS-S generiertes ROM ergibt eine Größe von 7,77 $mm^2$. Da insgesamt eine Chipfläche von 77,3 $mm^2$ zur Verfügung steht, belegt es nur etwa 10% der gesamten Chipfläche und ist damit für eine CISC-CPU recht klein. Das liegt aber auch daran, daß diverse Techniken zur Minimierung verwendet werden.

Die Ausgänge des ROMs werden alle auf Register geführt, weil die in VENUS-S generierten ROMs getaktet sind, und die Daten in den Registern zwischengespeichert werden sollen, sobald die ROM-Ausgänge mit der steigenden Taktflanke hochohmig werden.

Von den Ausgängen des ROMs führen 25 Leitungen zum Operationswerk, 10 Leitungen stellen eine mögliche Sprungadresse dar, und 4 Leitungen steuern die Folgeadresse des Mikroprogramms. Für die Folgeadresse gibt es 4 Quellen: Sie kann die inkrementierte aktuelle Mikroprogrammadresse sein, vom Mikroprogrammstack geladen werden, aus dem Sprungadreßfeld der aktuellen Mikroinstruktion kommen und für Mehrfachverzweigungen das Sprungadreßfeld mit einem 7-Bit

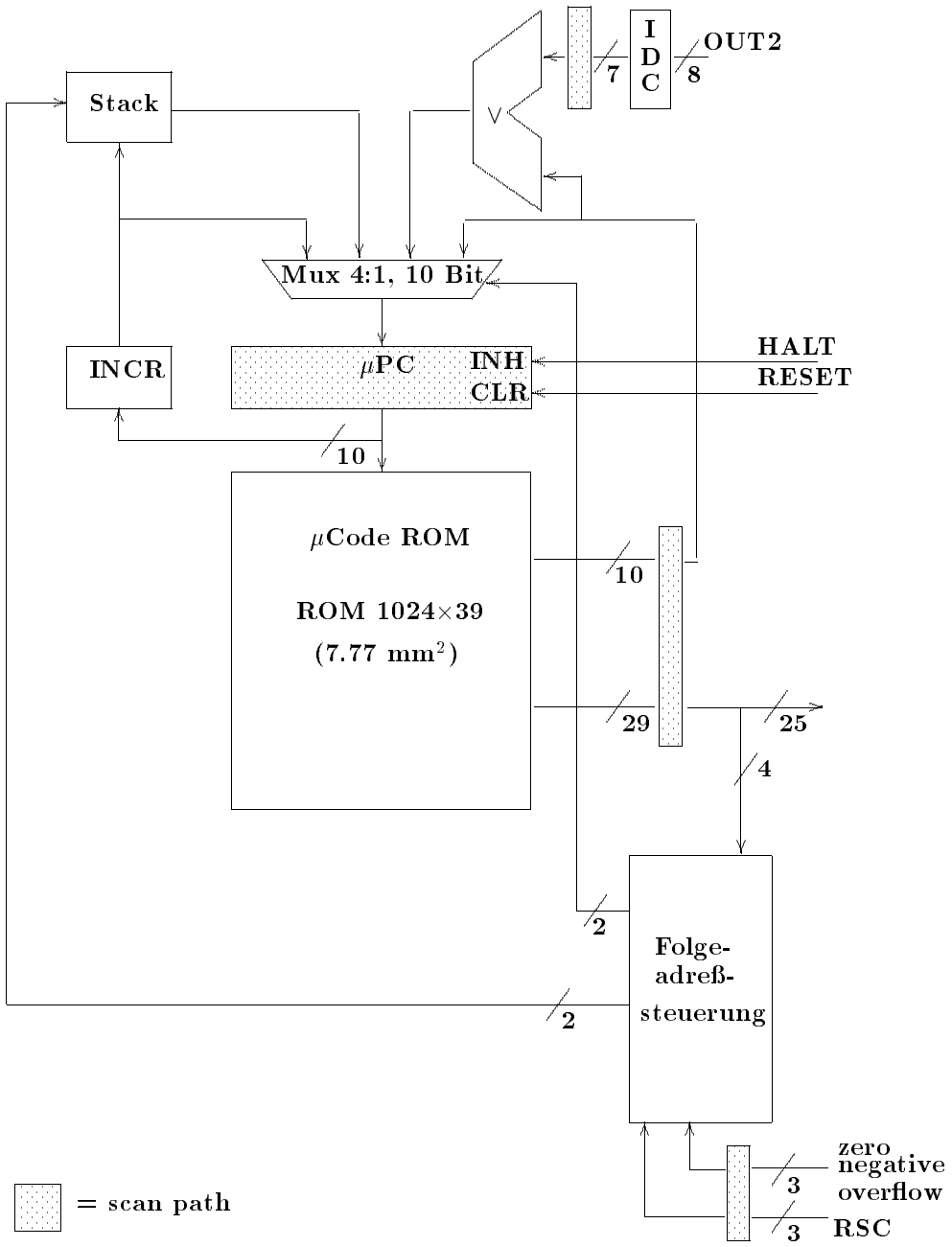



Abbildung 8.10: Das Steuerwerk des Scheme-Prozessors.

```
FUNC idc [6..0] (regbank_out2[7..0]);
  BEGIN
    idc[6..0] .= IF regbank_out2[7]              (* >= 128 ?    *)
                 THEN '1000 . regbank_out2[6..4] (* 64 + opcode *)
                 ELSE regbank_out2[6..0]
  END;
```

Abbildung 8.11: RT-Spezifikation des Befehlsdekoders in *Karl II*.

Wort bitvektoriell disjunktiv verküpft sein. Dieses 7-Bit Wort kommt vom Ausgang `OUT2` der Registerbank und wird im Steuerwerk durch den Befehlsdekoder `IDC` auf 7-Bit Worte abgebildet.

Der Mikroprogrammzähler $\mu$`PC` ist nicht als Zähler ausgeführt, sondern entgegen seinem Namen nur ein Register, dessen inkrementierter Inhalt allerdings als Folgeadresse eingeschrieben werden kann. Diese Ausführung wird zur Umgehung eines Fehlers in den Zählerzellen des VENUS-Systems gewählt: Die Simulationsmodelle dieser Zellen sind fehlerhaft: sie melden ungerechtfertigterweise Haltezeitverletzungen, sobald man die Zellen kaskadiert. Schon die Struktur auf RT-Ebene muß auf Gatterebene bestehende Einschränkungen berücksichtigen.

Ein Eingang `RESET` dient dazu, eine Anfangsbedingung einzustellen: An der Adresse null beginnt das Mikroprogramm. Durch `RESET` wird der Mikroprogrammzähler mit null initialisiert.

#### 8.3.2.1 Der Befehlsdekoder

Die Bytecodebefehle sind, wie in Kapitel 7.5 beschrieben, durch je ein Byte codiert, das bei einigen Befehlen ein optionales Argument von 4 Bit enthält. Befehle ohne dieses Argument werden durch Zahlen zwischen 0 und 127 codiert, Befehle mit Argument durch Zahlen zwischen 128 und 255.

Beim Dekodieren der Befehle verzweigt das Steuerwerk über eine Sprungtabelle. Diese Sprungtabelle könnte einen Eintrag für jedes Byte enthalten, sie belegte dann 256 Worte im Mikroprogramm. Da ein Mikroprogrammspeicher eine knappe Ressource ist — er ist platzintensiv, und seine Zugriffszeit wächst mit der Kapazität — muß diese Sprungtabelle klein gehalten werden.

Es sind Befehle im Bereich von 0 bis 54, sowie 8 Befehle mit einem Argument in den unteren 4 Bits zu implementieren. Es soll daher eine Lösung gewählt werden, die die 8 möglichen Befehle mit einem Argument im Opcode auf die Zahlen von 64 bis 71 abbildet. Die Anfangsadresse 64 der Befehle mit Argument wird gewählt, weil diese Zahl eine Zweierpotenz ist, die eine Realisierung des Befehlsdekoders sehr einfach und damit platzsparend und schnell macht. Ferner können noch bis zu neun weitere Befehle im Bereich von 55 bis 63 definiert werden, ohne daß eine Modifikation des Befehlsdekoders erfolgen muß. Zudem ermöglicht dieser Befehlsdekoder, der alle Zahlen unter 64 unverändert weiterleitet, daß auch anhand der Typen und CDR-Codes über Sprungtabellen verzweigt werden kann.

Ein Befehlsdekoder, der die beschriebene Abbildung vornimmt, kann sehr einfach aufgebaut werden und verringert die Größe der benötigten Sprungtabelle von 256 auf 72 Worte. Die RT-Spezifikation des Befehlsdekoders IDC, in der RT-Sprache *Karl II* [LiSc82] angegeben, wird in Abbildung 8.11 gezeigt.

#### 8.3.2.2 Die Folgeadreßsteuerung

Die Folgeadreßsteuerung im Steuerwerk ist ein Schaltnetz, das einen Folgeadreßbefehl aus dem Mikroprogramm ausführt, indem es über den Multiplexer eine der vier möglichen Folgeadressen auswählt. Ferner steuert es den Stack.

| Mikrobefehl | Belegung | Erklärung |
|---|---|---|
| `inc_pc` | 0000 | inkrementiere Mikroprogrammzähler |
| `jump` | 0001 | unbedingter Sprung |
| `jump_case` | 0010 | Mehrfachverzweigung |
| `return_eq` | 0011 | bedingte Unterprogrammrückkehr |
| `call` | 0100 | unbedingter Unterprogrammsprung |
| `return` | 0101 | unbedingte Unterprogrammrückkehr |
| `call_eq` | 0110 | bedingter Unterprogrammsprung |
| `call_ne` | 0111 | bedingter Unterprogrammsprung |
| `jump_eq` | 1000 | bedingter Sprung |
| `jump_neq` | 1001 | bedingter Sprung |
| `jump_lss` | 1010 | bedingter Sprung |
| `jump_leq` | 1011 | bedingter Sprung |
| `jump_gtr` | 1100 | bedingter Sprung |
| `jump_geq` | 1101 | bedingter Sprung |
| `jump_ov` | 1110 | bedingter Sprung bei Überlauf |
| `jump_res` | 1111 | residuumgesteuerer Sprung |

Tabelle 8.9: Mikrobefehle für die Folgeadreßbestimmung.

Es sind 16 Folgeadreßmikrobefehle vorgesehen, die auch abhängig von den 6 Statussignalen vom Operationswerk verschiedene Folgeadreßoperationen bewirken. Tabelle 8.9 zeigt alle diese Mikrobefehle. Es sind die Mikrobefehle enthalten, die nach der Analyse der Verhaltensbeschreibung tatsächlich benötigt werden (z.B. ist `return_eq` enthalten, nicht aber `return_ne`). Die Bedingungen für die bedingten Sprünge gehen aus dem Namen des Mikobefehls hervor. Der Mikrobefehl `jump_res` ist ein bedingter Sprung, bei dem der Folgeadreßmikrobefehl in einem Residuumregister in der Registerbank steht. Die Befehl wird durch die 3-Bit Leitung `RSC` an das Steuerwerk übergeben; das vierte Bit des Folgeadreßmikrobefehls hat bei allen bedingten Sprüngen einen festen Wert.

#### 8.3.2.3 Gewährleistung der Testbarkeit

Dieses Steuerwerk ist ohne zusätzliche schaltungstechnische Maßnahmen nicht testbar. Testmuster können an die Strukturelemente des Steuerwerks weder angelegt werden noch sind die Ausgänge dieser Stukturelemente beobachtbar.

Um eine Testbarkeit des Steuerwerks und des gesamten Prozessors zu erreichen, sind die in Abbildung 8.10 gefüllt dargestellten Register als Scan-Path-Register ausgeführt: In dem durch einen Eingang am Prozessor wählbaren Testmodus arbeiten sie als ein Schieberegister, in das von außen Daten hineingeschoben werden und aus dem die in den Registern enthaltenen Daten herausgeschoben werden. Im Betriebsmodus arbeiten sie als normale (Pipeline-)Register.

Während des Testens wird über diesen Scanpath das Mikroprogramm-ROM ausgelesen, sowie alle Strukturelemente des Steuerwerks wie Inkrementer, Stack, Befehlsdekoder, ODER-Logik und (indirekt) die Folgeadreßsteuerung getestet. Ferner dient der Scanpath auch dem Test des Operationswerks, weil unter Verwendung dieses Scanpathes die Statussignale des Operationswerks beobachtet und seine Steuerleitungen stimuliert werden können.

#### 8.3.2.4 Zeitbedingungen zwischen Operationswerk und Steuerwerk

Das Steuerwerk belegt die Steuerleitungen des Operationswerks mit Mikrobefehlen, die das Operationswerk ausführt. Das Operationswerk belegt nach der Ausführung der Mikrobefehle seine Statusleitungen mit Werten, die dem Steuerwerk für Verzweigungen dienen. Die einfachste Form der Zusammenarbeit zwischen Steuerwerk und Operationswerk besteht darin, daß das Steuerwerk jeweils wartet, bis die Statussignale vom Operationswerk vorliegen und dann eine neue Mikroinstruktion holt. Die erreichbare Zykluszeit für das Ausführen der Mikrobefehle ist dann die Summe der von Steuerwerk und Operationswerk beanspruchten Ausführungszeiten. In Tabelle 8.10 wird

| | 1. Periode | 2. Periode |
|---|---|---|
| Steuerwerk | Mikroinstruktion $n$ holen | |
| Operationswerk | | Mikroinstruktion $n$ ausführen |

Tabelle 8.10: Mikroinstruktionsabarbeitung ohne Pipelining.

diese Form der Mikroprogrammausführung gezeigt. Dabei wird davon ausgegangen, daß Operationswerk und Steuerwerk jeweils eine Periode eines Taktes benötigen. Es fällt auf, daß jeweils nur entweder das Operationswerk oder das Steuerwerk aktiv ist, das jeweils andere muß warten.

Zur Steigerung des Durchsatzes wird eine 2-stufige Pipeline eingeführt: Steuerwerk und Operationswerk arbeiten parallel. Während das Operationswerk eine Mikroinstruktion ausführt, holt das Steuerwerk schon die nächste Mikroinstruktion. Die erreichbare Zykluszeit sinkt gegenüber der Version ohne Pipeline von der Summe der Ausführungszeiten von Steuerwerk und Operationswerk auf das Maximum dieser Zeiten. Tabelle 8.11 zeigt die Mikroinstruktionsabarbeitung mit der

| | 1. Periode | 2. Periode |
|---|---|---|
| Steuerwerk | Mikroinstruktion $n$ holen | Mikroinstruktion $n+1$ holen |
| Operationswerk | Mikroinstruktion $n-1$ ausführen | Mikroinstruktion $n$ ausführen |

Tabelle 8.11: Mikroinstruktionsabarbeitung mit Pipelining.

Pipeline. Da davon ausgegangen wird, daß Operationswerk und Steuerwerk jeweils eine Periode eines Taktes benötigen, verdoppelt sich der Durchsatz. Der Preis für die Geschwindigkeitssteigerung ergibt sich daraus, daß die Statussignale für die vom Steuerwerk ausgeführten Verzeigungen eine Periode zuvor errechnet werden müssen. Eine Lösung wäre das Anhalten des Steuerwerks bei bedingten Sprüngen für eine Periode, um das Statussignal abzuwarten. Auf ein Anhalten wird hier aber verzichtet und nach Errechnung der Verzweigungsbedingung noch eine Mikroinstruktion ausgeführt (delayed branch). Dadurch wird zwar die Mikroprogrammierung erschwert, aber für die Fälle, in denen eine Instruktion gefunden werden kann, die gleichzeitig mit der Verzweigung ausführbar ist, der Durchsatz gegenüber einem Anhalten des Steuerwerks erhöht. Wenn keine solche Instruktion gefunden werden kann, wird eine wirkungslose Instruktion (NOP) ausgeführt. Die 2-stufige Pipeline wird in der Hardware durch die Register unterstützt, die die Statussignale vom Operationswerk und die aktuelle Mikroinstruktion zwischenspeichern. Diese Register dienen beim Test als Scanpath und im normalen Betrieb als Pipelineregister.

## 8.4 Erstellung des Mikroprogramms

Die RT-Struktur des Prozessors beinhaltet alle benötigten Informationen, um das Mikroprogramm zu erstellen. Eine Mikroinstruktion besteht aus den in Tabelle 8.12 aufgeführten Mikrobefehlen (Feldern). Das Mikroprogramm des Scheme-Prozessors soll zusammen mit der RT-Struktur das gleiche Verhalten wie in der Verhaltensbeschreibung dargestellt aufweisen. Ein platz- und ausführungszeitoptimales Mikroprogramm kann nur manuell erstellt werden. Der enge Zeitrahmen für den Entwurf des Scheme-Prozessors läßt jedoch keinen Raum für diese zeitintensive manuelle Mikroprogrammierung. Das Mikroprogramm mußte daher automatisch aus der Verhaltensbeschreibung generiert werden.

| Mikrobefehl (Feld) | Bitbreite | Wertebereich |
|---|---|---|
| Komplementer- und Addierer-Operation | 2 | siehe Tabelle 8.7 |
| Shifter-Operation | 2 | siehe Tabelle 8.8 |
| Registeradresse `IN` | 5 | siehe Tabelle 8.6 |
| Feldadresse `IN` | 2 | siehe Tabelle 8.5 |
| Registeradresse `OUT1` | 5 | siehe Tabelle 8.6 |
| Feldadresse `OUT1` | 2 | siehe Tabelle 8.5 |
| Registeradresse `OUT2`, Konstante | 5 | siehe Tabelle 8.6, 0 bis 31 |
| Feldadresse `OUT2` | 2 | siehe Tabelle 8.5 |
| Folgeadreßmikrobefehl | 4 | siehe Tabelle 8.9 |
| Folgeadresse | 10 | 0 bis 1023 |

Tabelle 8.12: Mikrobefehle einer Mikroinstruktion.

### 8.4.1 Veröffentlichungen über automatische Mikroprogrammgenerierung

Nach der Vorstellung des Konzeptes der Mikroprogrammierung durch Wilkes [Wilk51] 1951 fand der erste kommerzielle Einsatz in den 60er Jahren in Rechnern der Serie 360 von IBM statt. Ende der 70er, Anfang der 80er Jahre war die Technologie so weit fortgeschritten, daß die Mikroprogrammierung nicht mehr nur durch den Rechnerhersteller, sondern für den Anwender propagiert wurde. Die Technologie erlaubte große und auch schreibbare Mikroprogrammspeicher, und der Anwender sollte seine Programme durch Verlagerung ins Mikroprogramm beschleunigen [Davi78, Davi83]. Es wurden höhere Programmiersprachen, Übersetzer und Verifikationswerkzeuge für die Mikroprogrammierung erforscht [Muel84, Raus87]. Die Erfindung des RISC-Prozessors, der die derzeitige VLSI-Technologie besser zu nutzen scheint als mikroprogrammierte Systeme, hat sich inzwischen weitgehend durchgesetzt. Die Mikroprogrammierung durch den Anwender bleibt weiterhin Spezialaufgaben vorbehalten.

Mikroprogramme sind überwiegend fest und bestimmen den Befehlssatz eines Prozessors. Um eine schnelle Abarbeitung der Maschinenbefehle zu erreichen, werden Mikroprogramme manuell erstellt. Mikroprogrammcompiler, die aus einer höheren Beschreibung ein Mikroprogramm erzeugen, sind selten und auf Spezialrechner beschränkt. In der VLSI-Technologie werden Mikroprogramme in PLAs und ROMs gespeichert, da diese Speicher wesentlich weniger Transistoren enthalten und Fläche belegen als die für ladbare Mikroprogramme benötigten RAMs[1].

[1] Mit VENUS-S generierte ROMs enthalten einen Transistor pro Bit, generierte RAMs sechs.

Der Scheme-Prozessor soll ein Mikroprogramm erhalten, das automatisch aus der in Pascal abgefaßten Verhaltensbeschreibung generiert ist. Eine verwandte Aufgabe ist in der Dissertation von Nowak [Nowa86] 1986 in Kiel bearbeitet worden: Darin ist der mikroprogrammierbare Rechner SAMP entworfen und realisiert worden, dessen Mikroprogramme durch Mimola [Marw86] aus einer Pascal-ähnlichen Verhaltensbeschreibung übersetzt werden. SAMP ist ein Rechner, der mehrere Zuweisungen an Variablen im Hauptspeicher parallel ausführen soll, was durch einen Hauptspeicher mit fünf bidirektionalen Ports erreicht wird. Die Mikroprogramme werden in einen Speicher mit 4096 Worten von 142 Bit Breite geladen. SAMP wurde in TTL-Technologie aus 1463 ICs aufgebaut.

Wegen der schon erwähnten Einschränkungen in Mimola — keine Residuumkontrolle und kein Pipelining — kann Mimola zur Generierung der Mikroprogramme für den Scheme-Prozessor nicht eingesetzt werden.

### 8.4.2 Mikroprogrammgenerierung für den Scheme-Prozessor

Das Mikroprogramm für den Scheme-Prozessor soll automatisch generiert werden. Da kein geeigneter Compiler zur Verfügung steht, muß ein Mikroprogrammcompiler, der die in der Verhaltensbeschreigung genutzten Pascal-Konstrukte in ein binäres Mikroprogramm übersetzt, erstellt werden (Abbildung 8.12). Der zu erstellende Compiler braucht nicht retargierbar zu sein, d.h. er muß nicht für unterschiedliche Zielhardware Mikroprogramme generieren können, er soll nur ein Mikroprogramm für die zuvor vorgestellte RT-Struktur generieren. An das erzeugte Mikroprogramm ist als Hauptanforderung zu stellen, daß es maximal 1024 Worte belegt, da der ROM-Generator in VENUS-S weder ein ROM mit mehr als 1024 Worten bei einer Breite von 39-Bit erzeugen kann noch mehrere ROMs auf dem Chip untergebracht werden können. Zudem soll das Mikroprogramm die Möglichkeiten der Hardware vollständig nutzen und verzögerte Sprünge (delayed branches) sowie Residuumkontrolle verwenden.

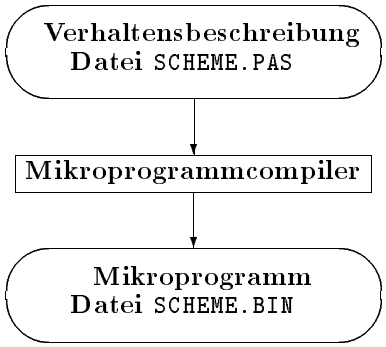



Abbildung 8.12: Generierung des Mikroprogramms.

Der Mikroprogrammcompiler liest ein Pascalprogramm ein und übersetzt es in ein Mikroprogramm. Die Hardware, für die das Mikroprogramm bestimmt ist, ist fest im Mikroprogrammcompiler vorgegeben. Ferner sind im Mikroprogrammcompiler alle Variablen mit ihrer Bitbreite und Adresse in der Registerbank vordefiniert. Der Mikroprogrammcompiler prüft die Realisierbarkeit der Quellstatements durch die vorgegenene Hardware. Auch bei Konstanten wird überprüft, ob die Hardware sie bereitstellen kann.

Der Mikroprogrammcompiler ist als Single-Pass-Pascalcompiler ausgeführt, der seinen Code für das ganze Programm in einem Feld ablegt. Der Fixup — das nachträgliche Eintragen der

Argumente von Vorwärtssprüngen in den erzeugten Code — wird durch dieses Feld vereinfacht. Der Mikroprogrammcompiler erzeugt ein Protokoll, in dem neben der Pascalquelle und eventuellen Fehlermeldungen auch die erzeugten Mikroinstruktionen mit ihren Adressen aufgeführt sind. Dieses Protokoll dient der Überprüfung den Compilers und erlaubt eine Rückverfolgung von Mikroprogrammadressen in die Pascalquelle, die erfolgen muß, wenn bei der späteren Simulation des Mikrocodes Fehler festgestellt werden.

Der Übersetzung des gesamten Mikroprogramms folgt eine Optimierungsphase, für die das erzeugte Mikroprogramm in eine bidirektional verkettete Liste von Labels und Mikroinstruktionen übersetzt wird. Auf dieser neuen Datenstruktur arbeitet dann der Mikroprogrammoptimierer regelgesteuert: Es wurde eine Reihe von Regeln entwickelt, die kompaktifizierende und beschleunigende Modifikationen am erzeugten Code beschreiben. Der Mikroprogrammoptimierer prüft der Reihe nach die Anwendbarkeit dieser Regeln und wendet sie auf den erzeugten Code an. Die Reihenfolge der Anwendung der Regeln wurde experimentell so ermittelt, daß das optimierte Mikroprogramm kurz und schnell wird. Die Optimierung terminiert, wenn keine der Regeln mehr anwendbar ist. Zusätzlich zur Anwendung der Regeln berücksichtigt der Optimierer die geforderte Einhaltung der Zeitbedingungen für das Pipelining.
Es sind folgende Regeln enthalten:

- Zusammenfassen von Mikroinstruktionen. Da in den Mikroinstruktionen beliebige Belegungen der Steuerleitungen mit beliebigen Folgeadreßbefehlen kombiniert werden können, können viele Sprungbefehle und Unterprogrammaufrufe mit ihrer Vorgängerinstruktion zusammengefaßt werden.
  Beispiel:

```
mdr := tos              ; sequentiell
JUMP L2
```

  wird ersetzt durch

```
mdr := tos, JUMP L2     ; parallel
```

- Nutzung der verzögerten Sprünge (delayed branches). In der Übersetzungsphase sind wirkungslose Operationen (NOPs) eingeführt worden. In dieser Optimierung wird versucht, diese NOPs durch sinnvolle Mikrobefehle zu ersetzen.
  Beispiel:

```
tos := mdr              ; Zuweisung
tos.type = 7            ; Vergleich
JUMP_EQ L2              ; bedingter Sprung
```

  wird ersetzt durch

```
mdr.type = 7            ; Vergleich
tos := mdr, JUMP_EQ L2  ; Zuweisung parallel mit bedingtem Sprung
```

- Löschen nicht erreichbarer Mikroinstruktionen. Nach RETURN und unbedingten Sprüngen wird (außer in Sprungtabellen) der Code bis zum nächsten Label gelöscht.

Beispiel:

```
        RETURN
        pc := mdr.cdr_code + 4
        tos := mdr
L1:     JUMP L2
```

wird ersetzt durch

```
        RETURN
L1:     JUMP L2
```

- Auflösung von Sprungbefehlen, die auf Sprungbefehle führen (branch chains).
  Beispiel:

```
        JUMP L1
...
L1:     JUMP L2
```

  wird ersetzt durch

```
        JUMP L2
...
L1:     JUMP L2
```

- Sprünge zu RETURN werden durch RETURN ersetzt.
  Beispiel:

```
        JUMP L1
...
L1:     RETURN
```

  wird ersetzt durch

```
        RETURN
...
L1:     RETURN
```

- Optimierung von Endrekursion im Mikroprogramm.
  Beispiel:

```
        CALL L1
        RETURN
```

  wird ersetzt durch

```
        JUMP L1
```

- Bedingter Unterprogrammaufruf und bedingte Unterprogrammrückkehr wird ermittelt und durch die dafür vorgesehenen Folgeadreßmikrobefehle (`call_ne`, `call_eq`, `return_eq`) realisiert.

Beispiel:

```
        JUMP_EQ L2
        CALL L1
L2:
```

wird ersetzt durch

```
        CALL_NE L1
```

- Sprungbefehle werden zur Vermeidung gleicher Folgen von Mikroinstruktionen eingeführt (cross-jumping).
  Beispiel:

```
        mdr := tos
        mar := pc4
        func := ctrl
        JUMP L1
...
        mdr := tos
        mar := pc4
        func := ctrl
        JUMP L1
```

  wird ersetzt durch

```
        JUMP L2
...
L2:     mdr := tos
        mar := pc4
        func := ctrl
        JUMP L1
```

Neben der für die Mikroprogrammierung möglichen Zusammenfassung von Mikroinstruktionen und der von RISC-Architekturen her bekannten Optimierungen verzögerter Sprünge [GrHe82] sind diese Regeln Peephole-Optimierungen, wie sie bei Wulf [Wulf75] beschrieben sind.

Der Mikroprogrammcompiler ist ein vom Verfasser auf dem Atari ST erstelltes Pascal-Programm von 2900 Zeilen. Abbildung 8.13 zeigt das Protokoll eines Compilerlaufes: Bei der Übersetzung entstanden 1554 Mikroinstruktionen, die durch die Optimierungen auf 1024 Mikroinstruktionen reduziert wurden. Daß genau die maximal mögliche Zahl von Mikroinstruktionen generiert wird, ist darauf zurückzuführen, daß soviele Standardprozeduren wie möglich in in den Mikrocode verlagert wurden, bis auch das letzte Wort im Mikroprogrammspeicher ausgenutzt wurde. In Abbildung 8.14 wird ein Ausschnitt aus dem erzeugten binären Mikroprogramm gezeigt. Die Mikrobefehle sind dort in der gleichen Reihenfolge wie in Tabelle 8.12 angegeben.

Ein wichtiger Vorteil eines automatisch erzeugten Mikroprogramms ist, daß es weniger fehlerbehaftet ist als ein manuell erstelltes. Da aber die Korrektheit des Mikroprogrammcompilers nicht vorausgesetzt werden kann, muß das generierte Mikroprogramm zusammen mit der Hardware simulativ auf gleiches Verhalten überprüft werden.

## 8.5 Validierung von RT-Struktur und Mikroprogramm

Es liegt eine Verhaltensbeschreibung in Pascal, eine Hardwarestruktur auf hoher Ebene und ein Mikroprogramm vor. Es muß jetzt überprüft werden, ob die Hardwarestruktur zusammen mit dem

```
Mikroprogrammcompiler V3.1 fuer Pipeline-Hardware, JL 15-Mar-89.

1554 Mikroinstruktionen
 717 address references
 292 different labels
   1 calls expanded

  10 passes
   5 branches to RETURN replaced
 105 branch chains resolved
 116 unreachable instructions deleted
 439 control instructions merged
   7 tail recursion optimizations
  61 conditional CALLs
   4 conditional RETURNs
  46 pipeline merges

1024 Mikroinstruktionen geschrieben
```

Abbildung 8.13: Protokoll eines Mikroprogrammcompilerlaufes.

```
     AU SH  AIN   RI AOUT1 R1 AOUT2 R2 CONT   ADDRESS
  0: 11 00 00000 XX 00100 10 00110 00 1010 00011110100   ( 124)
  1: 00 00 00011 00 00100 00 00000 00 1001 00001111001   ( 127)
  2: 00 00 01010 01 00100 01 00000 00 0100 00001010100   ( 128)
  3: 11 00 00000 XX 00100 10 00100 00 0000 XXXXXXXXXXX   ( 130)
  4: 00 00 00000 XX 00100 11 00000 00 1001 00001111000   ( 131)
  5: XX XX 00000 XX XXXXX XX XXXXX XX 1001 00011110100   ( 133)
  6: 00 00 00100 11 00000 01 00001 00 0001 00001011011   ( 134)
  7: XX XX 00000 XX XXXXX XX XXXXX XX 0101 XXXXXXXXXXX   ( 136)
  8: 11 00 00000 XX 00100 10 01110 00 0000 XXXXXXXXXXX   ( 137)
  9: XX XX 00000 XX XXXXX XX XXXXX XX 1011 00001111100   ( 138)
 10: 00 00 10011 01 00000 01 11110 00 0001 00001101101   ( 139)
 11: 11 00 00000 XX 00100 10 01101 00 0000 XXXXXXXXXXX   ( 142)
```

Abbildung 8.14: Ausschnitt aus dem erzeugten binären Mikroprogramm.

Mikroprogramm das gleiche Verhalten realisiert, das auch die Verhaltensbeschreibung beschreibt. Ein solcher Vergleich zweier Verhalten erfolgt im Entwurfsprozeß simulativ: Wenn beide Modelle für sinnvoll ausgewählte Folgen von Eingaben (Simulationsstimuli) die gleichen Reaktionen aufweisen, wird das Verhalten der Modelle als gleich angesehen.

Für eine Simulation des Scheme-Prozessors sind daher geeignete Stimuli zu benennen. Es ist ein Simulator für Hardwarestruktur und Mikroprogramm zu erstellen und zu entscheiden, wie die Reaktionen zu vergleichen sind.

Es sollten auf dieser Simulationsebene umfangreiche Simulationen vorgenommen werden können. Es sollten insbesondere Compilerläufe simuliert werden. Das Übersetzen des Scheme-Compilers durch sich selbst (Compiler-Bootstrap) wurde zuvor bei der Überprüfung der Verhaltensbeschreibung als eine Anwendung, bei der viele Fehler, auch solche in der Garbage Collection, aufgedeckt wurden, identifiziert. Ein Compiler-Bootstrap hat mit der Verhaltensbeschreibung eine Laufzeit von 1 3/4 Stunden auf dem Atari ST. Damit eine Simulation dieses komplexen Verhaltens auch von Mikroprogramm und Hardwarestruktur möglich ist, wurde der Simulator für die RT-Struktur und das Mikroprogramm auch als Pascalprogramm realisiert.

Der RT-Simulator beinhaltet das Verhalten der Strukturelemente in Pascal programmiert. Er liest das vom Mikroprogrammcompiler erzeugte Mikroprogramm ein und interpretiert es. Die Verhaltensbeschreibung wurde so erweitert, daß sie bei jedem Hauptspeicherzugriff und vor der Ausführung jeder Bytecodeinstruktion den Inhalt aller Scheme-Prozessorregister ausgibt. Bei Lesezugriffen auf den Hauptspeicher wird auch das gelesene Datum mit ausgegeben.

Dem RT-Simulator dienen diese Ausgaben der Verhaltensbeschreibung sowohl als Stimuli als auch als Beschreibung der korrekten Reaktionen. Der RT-Simulator führt die Mikroinstruktionen aus. Bei Hauptspeicherzugriffen und vor der Ausführung jeder Bytecodeinstruktion vergleicht er die Inhalte seiner Register mit den Registerinhalten, die die Verhaltensbeschreibung ausgegeben hatte. Bei Abweichungen meldet der Simulator einen Fehler und gibt seine sowie die erwarteten Registerinhalte zusammen mit den Adressen der letzten 16 ausgeführten Mikroinstruktionen aus. Wenn keine Abweichung vorliegt, lädt der Simulator bei Lesezugriffen sein Register `mdr` mit der Ausgabe der Verhaltensbeschreibung.

Der RT-Simulator ist als Koroutine in der Verhaltensbeschreibung realisiert. Frühere Versionen kommunizierten über Dateien und unter VMS als parallele Prozesse über VMS-Mailboxen. Verhaltensbeschreibung und Simulator wurden vom Verfasser auf dem Atari ST entwickelt und für umfangreichere Simulationen nach VAX/VMS portiert. Die Verhaltensbeschreibung mit dem Simulator als Koroutine wurde erstellt, um auch sehr große Simulationen zu ermöglichen. Beispielsweise wurden Compiler-Bootstraps simuliert. Derartig umfangreiche Simulationen fanden unter VAX/VMS auf einer VAX-8550 mit 7 MIPS statt und beanspruchten bis zu 8 CPU-Stunden.

Gegenüber vorhandenen Simulatoren ermöglicht das gewählte Vorgehen mit einem neu erstellten Simulator nicht nur die Simulation, sondern auch den Vergleich der Reaktionen gleichzeitig mit der Verhaltenssimulation. Damit ist auch die Auswertung der Simulationsergebnisse durch den erstellten Simulator automatisiert worden. Die Verwendung der Koroutinen vermeidet das Speichern der sehr großen Datenmengen für Stimuli und Simulationsergebnisse, die besonders bei der Simulationsauswertung das ungelöste Hauptproblem darstellen [BuLa87]. Abbildung 8.15 zeigt den Simulator und seine Einbindung in die zuvor vorgestellten Programme. Insbesondere die Verhaltensbeschreibung wird dort in ihren beiden Eigenschaften dargestellt: Sie ist sowohl Datum (Eingabe für den Mikroprogrammcompiler) als auch Operator (Verhaltenssimulator und Generator von Stimuli und Erwartungswerten).

Über die Überprüfung von RT-Struktur und Mikroprogramm hinaus liefert der Simulator statistische Informationen. Es wird die Anzahl der ausgeführten Mikroinstruktionen, die Anzahl der Lese- und Schreibzugriffe auf den Hauptspeicher und auf jedes Prozessorregister sowie die Anzahl der Ausführungen jeder Mikroinstruktion ermittelt. Diese Werte dienen der Optimierung der Hard-

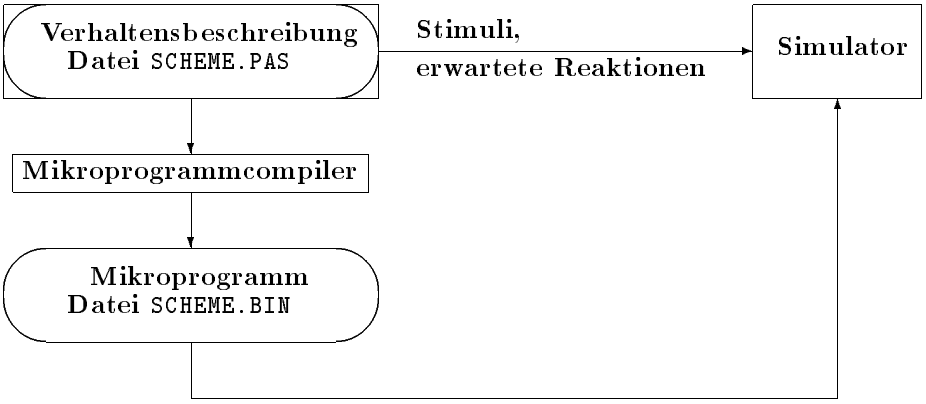



Abbildung 8.15: Einbindung des RT-Simulators in die Entwurfsprogramme.

ware. Wenig genutzte Register können in den Hauptspeicher verlagert werden, nicht ausgeführte Mikroinstruktionen durch Erweiterung der Stimuli mitsimuliert werden. Die Anzahl der ausgeführten Mikroinstruktionen entspricht der Anzahl der Taktzyklen und erlaubt eine Abschätzung der Geschwindigkeit des Scheme-Prozessors.

## 8.6 Realisierung des Scheme-Prozessors mit VENUS-S

Es liegt sowohl eine RT-Struktur als auch das dazugehörige Mikroprogramm vor. Beide sind bereits simulativ validiert. Die sich anschließende Phase ist die Verfeinerung der RT-Struktur auf Logikebene. Dabei werden die Elemente der RT-Ebene aus Standardzellen zusammengesetzt und miteinander verbunden, bis der Scheme-Prozessor fertig ist. Bei dieser Verfeinerung auf eine niedrigere Entwurfsebene sind Entscheidungen zu fällen, die die folgende Chipkonstruktion berücksichtigen: Es sind bei mehreren Realisierungsmöglichkeiten nicht nur die weniger platzaufwendigen zu bevorzugen, vielmehr wird die Konstruierbarkeit des Chips durch den für Verdrahtung benötigten Platz bestimmt. Es sind Realisierungsmöglichkeiten zu finden, die wenig Verdrahtungsfläche benötigen.

Auch die aus Standardzellen gebildeten Schaltungen (Elemente auf RT-Ebene, Steuerwerk, Operationswerk, ganzes Chip) sind simulativ zu validieren, wobei der Generierung der Stimuli und dem Vergleich mit den erwarteten Reaktionen Aufmerksamkeit zu schenken ist.

Die Realisierung ist ein Bottom-Up-Vorgang: Eine Struktur auf höherer Hierarchieebene wird aus ihren Bestandteilen (bereits realisierten Subsystemen) zusammengesetzt. Die Beschreibung der Realiserungen erfolgt daher von kleinsten zu den größeren Einheiten, von Buszugriffsmodul und ALU über Operationswerk und Steuerwerk bis hin zum gesamten Prozessor.

### 8.6.1 Realisierung des Buszugriffsmoduls

Der Buszugriffsmodul wird durch die Beschreibung als RT-Element und durch die Busspezifikation des 68000 in [68000] spezifiziert. Ausgehend von diesen Spezifikationen ist der Buszugriffsmodul als synchrones Schaltwerk mit JK-Flipflops im August 1988 mit VENUS-S realisiert und simuliert worden.

Da der Buszugriffsmodul im gesamten Prozessor der Modul ist, der unbedingt funktionieren muß, um nach der Fertigung des Prozessors alle anderen Moduln auf Entwurfsfehler hin überprüfen zu können, wurde der Buszugriffsmodul vorab gefertigt. Er sollte vor der Fertigstellung des Scheme-Prozessors getestet werden. Das Layout des Buszugriffsmoduls ist im Oktober 1988 zur Fertigung eingereicht worden. Leider lag er bis zur Abgabe des kompletten Scheme-Prozessors im April 1989 noch nicht vor.

### 8.6.2 Realisierung der ALU

Vordringliches Ziel bei der Realisierung der ALU ist das Erzielen einer kurzen Verarbeitungszeit. Alle Operationen verwenden die ALU, auch ein Register-Register-Transfer erfolgt immer über die ALU.

Die ALU besteht aus Shifter, Negierer und Addierer. Der Shifter wird als Schaltnetz aus Multiplexern und Gatter gebildet. Der Komplementer entsteht aus EXOR-Gattern, und der Addierer ist in Carry-Look-Ahead-Technik realisiert. Der Shifter benötigt mehr Fläche, als wenn er ein Schieberegister enthielte, und der Addierer ist auch flächenintensiver als ein Ripple-Carry-Addierer. Die gewählten Realisierungen erzielen aber eine höhere Verarbeitungsgeschwindigkeit der ALU.

### 8.6.3 Realisierung der Registerbank

Bei der Realisierung der Registerbank mit einem Eingangsport und zwei Ausgangsports gibt es verschiedene Möglichkeiten. Es ist eine Realisierung mit Multiplexern (Abbildung 8.16) und eine mit Tri-State-Bussen (Abbildung 8.17) möglich. Die Abbildungen zeigen schematisch eine Registerbank mit vier als Flipflops (FF1 bis FF4) dargestellten Registern.

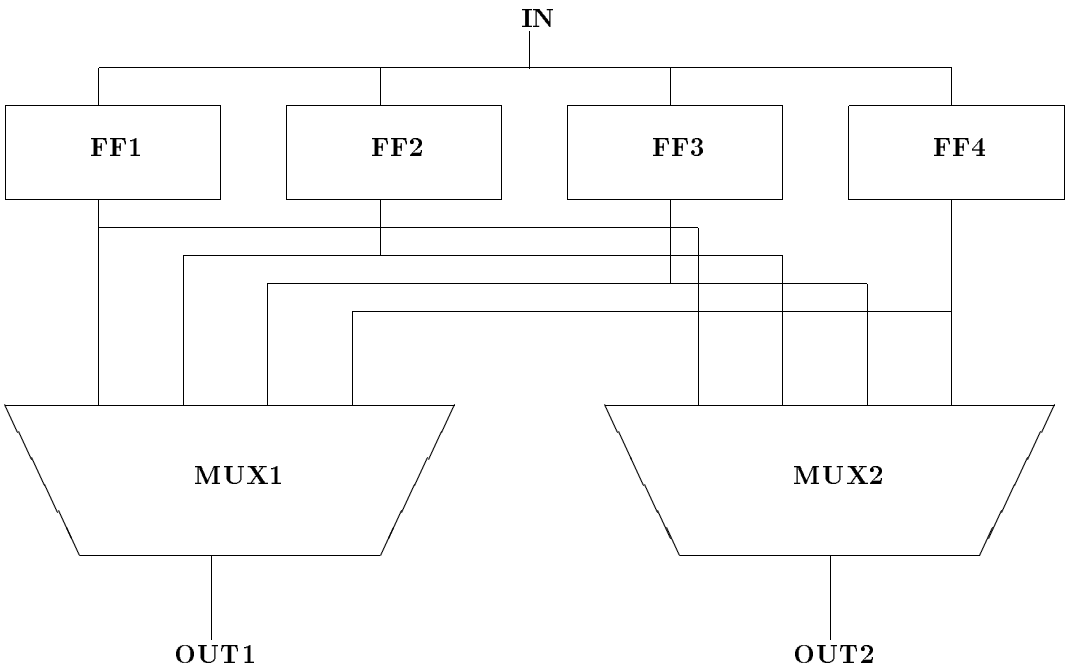



Abbildung 8.16: Realisierung der Registerbank mit Multiplexern.

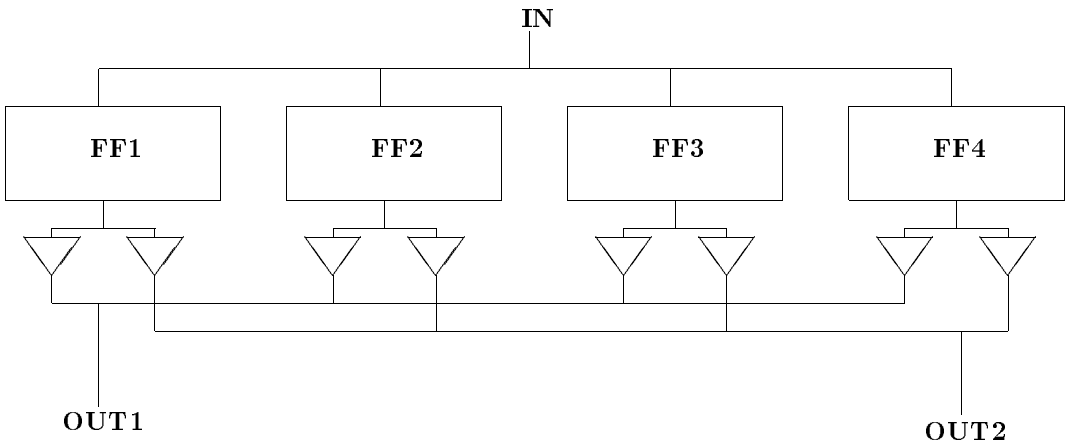



Abbildung 8.17: Realisierung der Registerbank mit Tri-State-Bussen.

Eine Abwägung zwischen den beiden Realisierungsmöglichkeiten erfolgt durch Vergleich der durch die Standardzellen belegten Fläche und durch den Verdrahtungsaufwand. Ein 4:1-Multiplexer hat eine Breite von 82 $\mu$m, ein Tri-State-Treiber ist 41 $\mu$m breit. Die Breiten sind proportional zur Fläche, weil alle Standardzellen die gleiche Höhe (hier 134 $\mu$m) aufweisen. Eine Abschätzung der Fläche spricht für eine Realisierung mit Multiplexern, da vier Tri-State-Treiber die doppelte Fläche eines 4:1-Multiplexers belegen. Die Realisierung mit Multiplexern führt dazu, daß sehr viele Leitungen zu den Multiplexern zu führen sind. Ein interessantes und bemerkenswertes Ergebnis des Entwurfs war, daß Leitungen in VLSI-Schaltungen eine größere Fläche belegen als die Zellen selber. Der Verfasser hat sich für die Realisierung mit Tri-State-Bussen entschieden und eine Plazierung der Standardzellen entwickelt, in der diese Busse nur Kanäle überqueren und somit nicht zur Verbreiterung der Kanäle beitragen. Die Registerbank stellt das umfangreichste Schaltungsteil des gesamten Prozessors dar. Ein geschickter Entwurf der Registerbank ist der Schlüssel zu einem Prozessor, der auf der sehr begrenzten Chipfläche Platz findet. Abbildung 8.18 zeigt die durch die Tri-State-Busse mögliche Plazierung der Registerbank: Jede Standardzellzeile enthält ein Register zusammen mit den Tri-State-Treibern für beide Ausgänge. Wie in der Abbildung gezeigt, können damit alle Busse senkrecht durch die Standardzellzeilen verlaufen, ohne die Kanalbreite zu beeinflußen. Die drei Busse umfassen insgesamt 96 Leitungen, die damit sehr platzsparend verlegbar sind. Die Plazierung nutzt die Tatsache, daß die Standardzellen oben und unten elektrisch äquivalente Anschlüsse haben oder eine Möglichkeit aufweisen, Leitungen senkrecht über die Zellen zu führen. Die Breite der Zellen ist in der Abbildung maßstabsgerecht gezeigt, die Latches weisen die achtfache Breite der Tri-State-Treiber auf. Latches (pegelgesteuerte Flipflops) werden verwendet, weil sie die kleinsten speichernden Zellen sind. Die platzsparendste Ausführung enthält 8 Latches in einer Standardzelle; diese Ausführung wird in der Abbildung gezeigt.

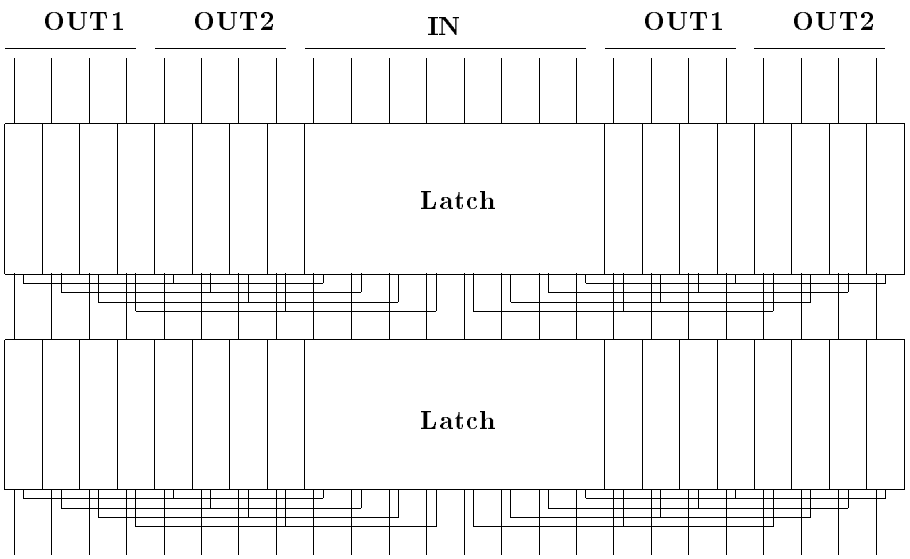



Abbildung 8.18: Plazierung der Standardzellen in der Registerbank.

#### 8.6.3.1 Funktionseinheiten der Registerbank

In die Registerbank sind beim Entwurf der RT-Struktur viele Aufgaben verlagert worden: Eine Feldauswahl erlaubt das Lesen und Schreiben einzelner Felder, Konstanten können generiert werden, und diverse Register haben Spezialfunktionen. Da die Registerbank aus Standardzellen manuell aufzubauen ist, können solche Spezifikationen erfüllt werden. Die Registerbank umfaßt insgesamt 1600 Standardzellen und kann daher nur im Überblick beschrieben werden. Der Überblick soll die Einteilung der Registerbank in verschiedene Funktionseinheiten und deren Realisierung schildern.

Die Registerbank enthält als Kern die Register und ihre drei Busse. Zusätzlich müssen Adreßdekoder die Adressierung der Register ermöglichen, muß die Feldauswahl erfolgen und müssen die Register mit Spezialfunktionen wie Ein- und Ausgabe (`mdr`, `mar`, `csr`), Seiteneffekte (`test_and_set`, `test_and_clear`) sowie die Pseudoregister (`cur_byte`, `pc4`) entworfen werden. Abgesehen von den Spezialregistern mit Breiten von 1, 4, 8 und 22 Bit sind Register mit 24 und mit 32 Bit Breite vorhanden.

Abbildung 8.19 zeigt die Struktur der Registerbank, bestehend aus den oben erwähnten Funktionseinheiten. Den Kern bilden die eigentlichen Register, die aus Latches und Tri-State-Treibern bestehen. Ferner führen zu diesem Kern auch die in dieser Abbildung nicht gezeigten Verbindungen mit der Außenwelt und dem Buszugriffsmodul. Die Latches haben Takteingänge, die mit negativem Pegel das Datum von dem Eingangsbus übernehmen. Da keine Latches mit Takt- und Enableeingang zur Verfügung stehen, muß der Takt durch ein Schaltnetz ausgeblendet werden: Der in Abbildung 8.19 rechts gezeigte Block „Schreibtakt“, erhält den Takt „ClockWrite“ sowie die Register- und Feldadresse des zu schreibenden Registers. Jedes Register erhält separate Takte für seine drei Felder. Der Block „Schreibtakt“ schaltet den Takt nur auf das durch die Registeradresse ausgewählte Register und auch nur auf das zu schreibende Feld dieses Registers. Nur wenn in das gesamte Register geschrieben werden soll, werden alle Takte geschaltet. Zusätzlich müssen, abhängig von den zu schreibenden Feldern, die Daten vom Eingang „IN“ auf die Eingänge der Latches geschaltet werden. Der Block „Feldauswahl IN“ übernimmt diese Aufgabe: Wenn das ganze Wort oder nur das Datenfeld geschrieben werden soll, läßt er die Daten unmodifiziert durch. Beim Schreiben in das Typfeld bzw. CDR-Codefeld werden die unteren sechs bzw. zwei Bits auf die Leitungen dieser Felder geschaltet. Der Block „Schreibtakt“ ist aus Dekodern ausgebaut, „Feldauswahl IN“ besteht aus zwei Multiplexern.

Das Auslesen der Registerbank erfolgt folgendermaßen: Die Tri-State-Treiber genau eines Registers pro Ausgangsbus werden durchgeschaltet. Die in der Abbildung links gezeigten Adreßdekoder steuern diese Treiber an. Der ausgelesene Registerinhalt wird in den Blöcken „Feldauswahl OUTn“ zu einer mit „ReadClock“ bestimmten Zeit in Latches übernommen. Das Zwischenspeichern der ausgelesenen Werte berücksichtigt, daß die Ausgänge der Registerbank über die ALU wieder auf den Eingang geführt werden. Mit nur einem Latch in dieser Schleife würde das Operationswerk zu einem asynchronen Schaltwerk. Die Hauptaufgabe der Blöcke „Feldauswahl OUTn“ besteht darin, aus dem ausgelesenen Register die auszugebenden Felder auszuwählen. „Feldauswahl OUT2“ kann zudem Konstanten generieren: „Addr_Out2“ kann als Konstante, um führende Nullen erweitert, ausgegeben werden. Die Adreßdekoder bestehen aus Dekodern, und die Feldauswahlbloöcke sind aus Multiplexern, Gattern und Pullup-Zellen aufgebaut. Die Pullup-Zellen sorgen für ein definiertes Potential auf den Tri-State-Bussen, wenn keiner der Treiber auf den Bus geschaltet ist. Diese Maßnahme ist bei MOS-Technologien erforderlich.

#### 8.6.3.2 Realisierung der Spezialregister

Die Register mit 24-Bit und 32-Bit Breite in der Registerbank sind wie in Abbildung 8.18 gezeigt realisiert. Über diese Register hinaus enthält die Registerbank noch Register mit Spezialfunktio-

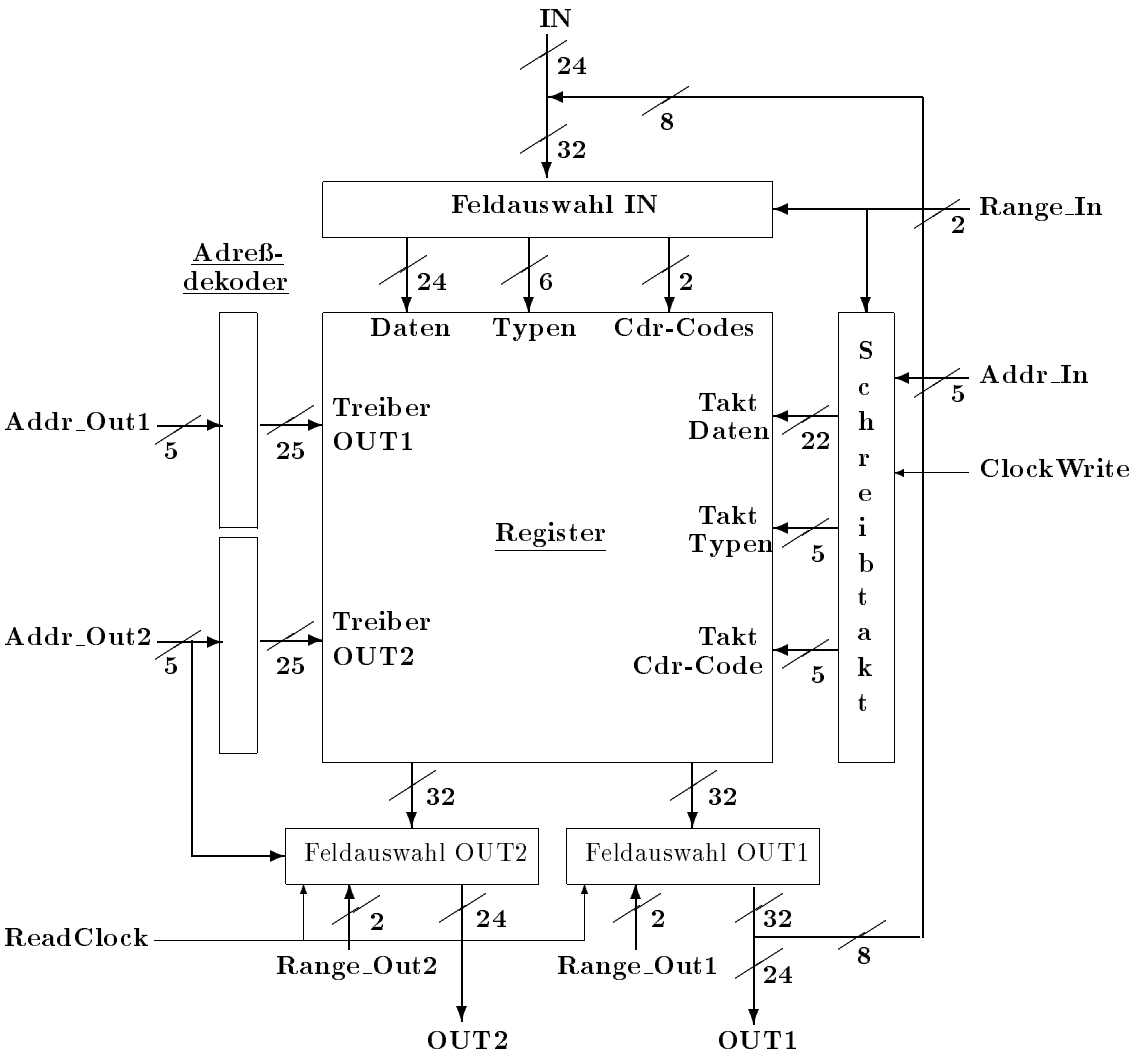



Abbildung 8.19: Die Struktur der Registerbank.

nen, die separat zu entwerfen und in drei Gruppen einteilbar sind:

- Pseudoregister, die Teile anderer Register repräsentieren. Die Register `pc`, `pc4`, `cur_byte` und `instr_cache` fallen in diese Gruppe. `pc4` repräsentiert die höchstwertigen 24 Bits von `pc`, und `cur_byte` repräsentiert eines von vier Bytes aus `instr_cache`.
- Register mit Seiteneffekten. Die Register `tos_valid`, `test_and_set` und `test_and_clear` bilden diese Gruppe. Das Auslesen der Pseudoregister `test_and_set` und `test_and_clear` liest tatsächlich `tos_valid` aus und modifiziert es anschließend.
- Register mit Ein- und Ausgabefunktionen. Die Register `mar`, `mdr` und `csr` haben diese Funktionen. Das Register `mar` hält die Hauptspeicheradressen. `mdr` ist das Ziel bzw. die Quelle beim Lesen bzw. Schreiben des Hauptspeichers. `csr` kann zum Abfragen des Status des Buszugriffsmoduls gelesen werden. Durch Schreiben in `csr` werden Buszugriffe, Interrupts des Hostprozessors und Start sowie Beendigung der Aktivität des Scheme-Prozessors initiiert.

Die Realisierung dieser Register erfordert Multiplexer, die einzuschreibende Daten aus mehreren Quellen auswählen und Taktausblendungen, die ein Einschreiben in Teile eines Registers ermöglichen (`mdr`, `pc4`, `cur_byte`). Beim Auslesen werden Registerinhalte um vorstehende Nullen erweitert (`csr`, `tos_valid`, `pc4`, `cur_byte`), und es werden Teile der Register durch Multiplexer ausgewählt (`pc4`, `cur_byte`, `mdr`).

Der Entwurf einer Struktur auf Gatterebene für diese Spezialregister war sehr zeitintensiv. Die Strukturen sind sehr umfangreich, weshalb sie hier auch nicht schematisch durch eine Abbildung angegeben werden.

Ferner durchbrechen diese Spezialregister das Plazierungsschema aus Abbildung 8.18, da sie außer Latches und Tri-State-Treibern auch Gatter und Multiplexer enthalten. Eine Plazierung unter Beibehaltung der Leitungsführung für die Busse wurde durch Aufteilung der Spezialregister auf mehrere Standardzellzeilen erreicht.

### 8.6.4 Realisierung des Operationswerks

Das Operationswerk, in Abbildung 8.9 auf Seite 88 gezeigt, besteht aus Registerbank, ALU und Buszugriffsmodul. Das Operationswerk wurde als ein Makro aus Standardzellen aufgebaut. Ein 32-Bit Register bestimmt die Breite einer Standardzellenzeile. Da die Mehrzahl der Register nur 24 Bit breit sind, werden Buszugriffsmodul, Adreßdekoder und Generator für die Schreibtakte auf den durch die fehlenden oberen 8 Bits freien Platz aufgeteilt. Abbildung 8.20 zeigt die Plazierung des Operationswerks.

Auch bei der Plazierung des Operationswerks führten Überlegungen über die Leitungsführung zu einer platzsparenden Realisierung. Der Kern wird von den 24-Bit und 32-Bit Registern gebildet. Zwischen diesen beiden Bereichen sind die Spezialregister angeordnet, da diese sowohl 24-Bit als auch 32-Bit Register beinhalten. Die Feldauswahllogiken benötigen alle drei 32-Bit Busse und sind daher unterhalb der 32-Bit Register angeordnet. An die Feldauswahllogiken ist die ALU angeschlossen, die sich auch in der Plazierung eine Zeile darunter anschließt. Neben den 24-Bit Registern wird der nicht benötigte Platz für die obersten 8 Bits durch den Buszugriffsmodul, die Adreßdekoder und den Schreibtaktgenerator eingenommen.

Insgesamt belegt das Operationswerk 25 Standardzellenzeilen, von denen die ALU und die Feldauswahllogik je zwei Zeilen und jedes 24-Bit und 32-Bit Register eine Zeile einnimmt. Der Platzbedarf der Spezialregister fällt sehr unterschiedlich aus: Das Register `mdr` belegt beispielsweise zwei Zeilen, während sich die Register `mar` und `tos_valid` eine Zeile teilen.

Das Operationswerk besteht, wie in Tabelle 8.13 angegeben, aus genau 1796 Standardzellen, die mit 1360 Leitungen untereinander verbunden sind. Ursprünglich war geplant, das Operationswerk

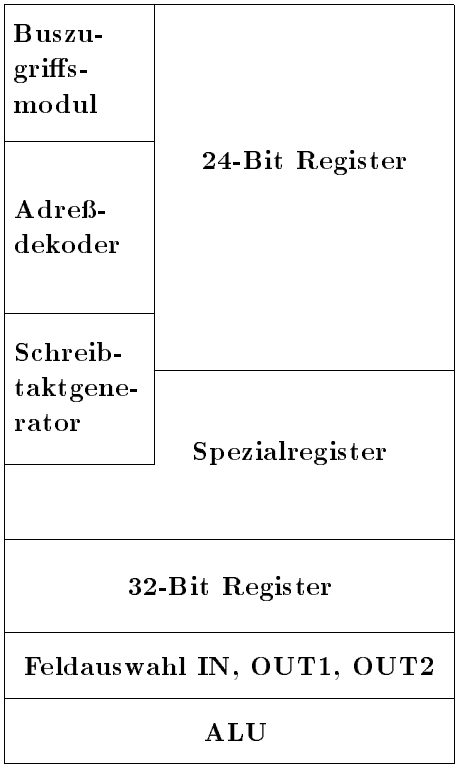



Abbildung 8.20: Plazierung des Operationswerks.

auf mehrere Standardzellenblöcke aufzuteilen. Aber durch den Platzbedarf für die Verdrahtung zwischen diesen Blöcken waren die geforderte Chipabmessungen nicht einzuhalten. Nicht einmal der Buszugriffsmodul als separater Standardzellenblock erwies sich als vertretbar. Das Entwufssystem VENUS-S ist nicht für Schaltungen dieses Umfangs ausgelegt. Es entstanden abnorm lange Programmlaufzeiten bei der Bearbeitung des Operationswerks: Die Netzlistenkompilation an der Workstation benötigte 8 Stunden. Die Chipkonstruktion erfolgte als ein Batchjob auf dem BS2000-Rechner mit mehreren Tagen Laufzeit. In beiden Fällen waren dies jeweils die einzigen Aktivitäten auf den Rechnern.

| | |
|---|---|
| Anzahl der Standardzellen | 1796 |
| Anzahl der Signale | 1360 |
| Anzahl der Transistoren | 26908 |
| Anzahl der Gatteräquivalente | 6727 |
| Fläche in $mm^2$ | 36,15 |

Tabelle 8.13: Statistik des Operationswerks.

Für die Chipkonstruktion wurden alle Standardzellen mit Positionen im Layout versehen. Das wird durch sogenannte Vorplazierungsproperties ermöglicht, die bei der Schaltplaneingabe an die Zellen vergeben werden können. Die meisten Zellen erhielten eine feste Position, einige Zellen wurden nur auf eine Standardzellzeile fixiert, um später bei der Chipkonstruktion eine optimale Position innerhalb der Zeile zu ermitteln. Auch die Positionen der Ein- und Ausgänge des Operationswerks wurden in der Chipkonstruktion vorgegeben.

#### 8.6.4.1 Validierung des Operationswerks

Beim Entwurf der Funktionselemente Registerbank, ALU und Buszugriffsmodul mit VENUS-S wurden diese Funktionselemente und auch die noch kleineren Einheiten jeweils mit dem in VENUS-S enthaltenen Simulator validiert. Auf diese Weise wurde eine Vielzahl von Fehlern vor der Fertigstellung des gesamten Operationswerks gefunden und korrigiert. Auch das Operationswerk wurde automatisch mit VENUS-S simuliert, wozu ein Stimulicompiler erstellt wurde, der Mikrobefehle für das Operationswerk als Eingabe erhält und sowohl die Stimuli für den Simulator SMILE in VENUS-S als auch die erwartete Reaktion in der Beschreibungssprache EXPRESS für das Programmpaket SIMUEVA [BuLa87] erstellt. Durch dieses Vorgehen können die Stimuli auf hoher Ebene erstellt werden, die Simulation findet auf Gatterebene statt und die Simulationsergebnisse werden automatisch mit den erwarteten Reaktionen verglichen. Die Beschreibung auf hoher Ebene ist insbesondere notwendig, weil alle Daten durch den Buszugriffsmodul eingelesen und ausgegeben werden müssen, wofür ein Protokoll und Zeitbedingungen einzuhalten sind. Ferner erlaubt dieses Vorgehen auch das Erstellen der Testmuster für den Test der Chips nach der Fertigung auf hoher Ebene. Auch diese sehr umfangreichen Muster werden durch den Stimulicompiler in Stimuli für den Fehlersimulator übersetzt und dann ihre Fehlerabdeckung ermittelt.

Abbildung 8.21 zeigt den beschriebenen Ablauf bei der Simulation. Der Stimulicompiler ist ein vom Verfasser erstelltes Pascalprogramm von etwa 1800 Zeilen.

### 8.6.5 Realisierung des Steuerwerks

Die Realisierung des Steuerwerks mit VENUS-S erfolgt durch zwei Makros: Das Mikroprogramm-ROM ist ein generiertes Makro, und das Folgeadreßrechenwerk ist als Standardzellblock realisiert. Beide Makros bilden zusammen das in Abbildung 8.10 auf Seite 90 dargestellte Steuerwerk.

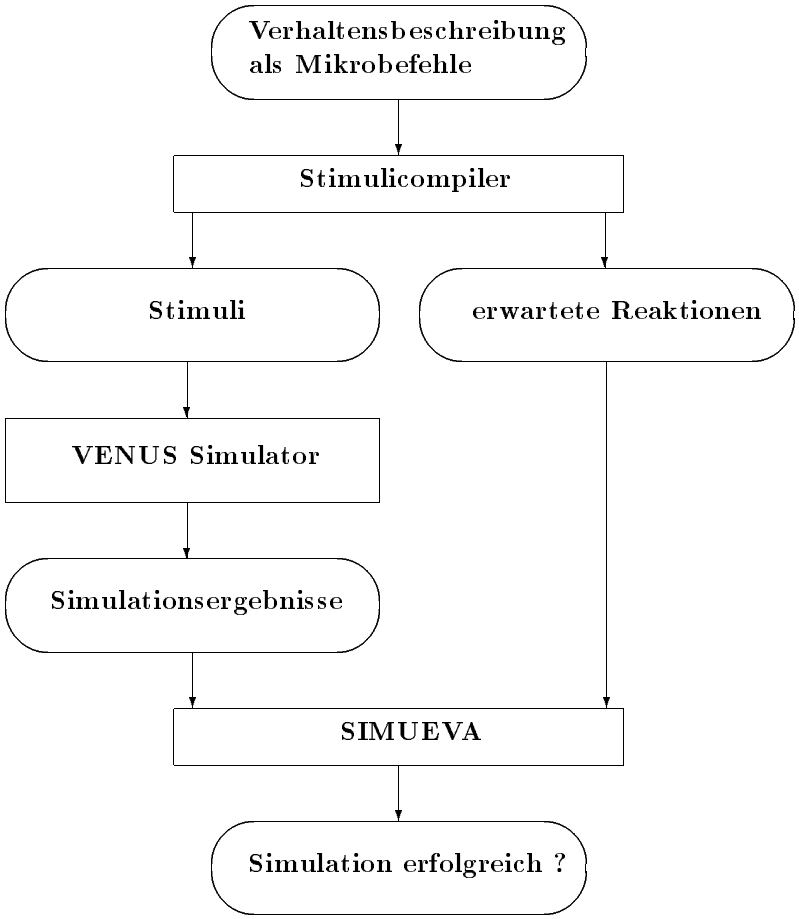



Abbildung 8.21: Simulation des Operationswerks auf Gatterebene.

Um ein ROM zu erhalten, muß eine Steuerdatei erstellt werden, die das ROM einschließlich seines Inhaltes spezifiziert. Das vom Mikroprogrammcompiler erstellte binäre Mikroprogramm wurde durch ein vom Verfasser erstelltes Programm in eine solche ROM-Steuerdatei für VENUS umgesetzt. Das generierte ROM enthält 43202 Transistoren, es belegt eine Fläche von 7,77 $mm^2$ und weist eine Zykluszeit von 54 ns auf.

Das Folgeadreßrechenwerk ist auf Standardzellen aufgebaut. Es hat eine mit VENUS-S gut beherrschbare Komplexität. Tabelle 8.14 zeigt eine Statistik des Folgeadreßrechenwerks.

| | |
|---|---|
| Anzahl der Standardzellen | 166 |
| Anzahl der Signale | 251 |
| Anzahl der Transistoren | 5472 |
| Anzahl der Gatteräquivalente | 1368 |
| Fläche in $mm^2$ | 6,30 |

Tabelle 8.14: Statistik des Steuerwerks (ohne ROM).

#### 8.6.5.1 Validierung des Steuerwerks

Die Validierung des mit VENUS-S realisierten Steuerwerks durch Simulation wurde automatisiert. Der RT-Simulator wurde um eine Ausgabe von Stimuli für den Simulator SMILE in VENUS-S sowie um die Ausgabe der erwarteten Reaktionen erweitert. Da das Steuerwerk eine wenig umfangreiche Schaltung ist, konnten, verglichen mit dem Operationswerk und gesamten Chip, sehr umfangreiche Simulationen erfolgen: Simulationsläufe von zu 40000 Mikroinstruktionen erfolgten in nur 2 CPU-Stunden.

Für den Vergleich der erwarteten Reaktionen mit den Simulationsergebnissen wurde vom Verfasser ein Vergleichsprogramm im C erstellt. Das bei der Simulation des Operationswerks eingesetzte Programmpaket SIMUEVA konnte hier nicht verwendet werden, da es einerseits viel zu rechenzeitintensiv bei diesen großen Datenmengen ist und es andererseits keine Busse in den Simulationsergebnissen verarbeiten kann.

Abbildung 8.22 zeigt die bei der Validierung des Steuerwerks beteiligten Programme und den Fluß der Daten.

### 8.6.6 Realisierung des Prozessors

Der Prozessor besteht aus dem Steuerwerk und dem Operationswerk. Diese beiden Bestandteile sind als Makros realisiert. Der Prozessor ist aus diesen Makros und den Padzellen, die die Verbindungen des Prozessors mit der Außenwelt ermöglichen, aufgebaut. Die in Abbildung 8.23 gezeigte Plazierung der Komponenten ermöglicht das Einhalten der Chipabmessungen. Eine Statistik des gesamten Prozessors ist in Tabelle 8.15 gezeigt.

#### 8.6.6.1 Validierung des Prozessors

Auch der gesamte Prozessor mußte, wie zuvor seine Bestandteile, simulativ validiert werden. Die Erstellung der Stimuli wurde automatisiert. Der RT-Simulator wurde wiederum um eine Ausgabe von Stimuli für den Simulator SMILE in VENUS-S erweitert. Eine Ausgabe der erwarteten Reaktionen der Simulation erfolgte nicht, da wegen des Umfangs der Schaltung keine sehr umfangreichen Simulationen möglich waren. Es galt aber die korrekte Verschaltung der Makrozellen

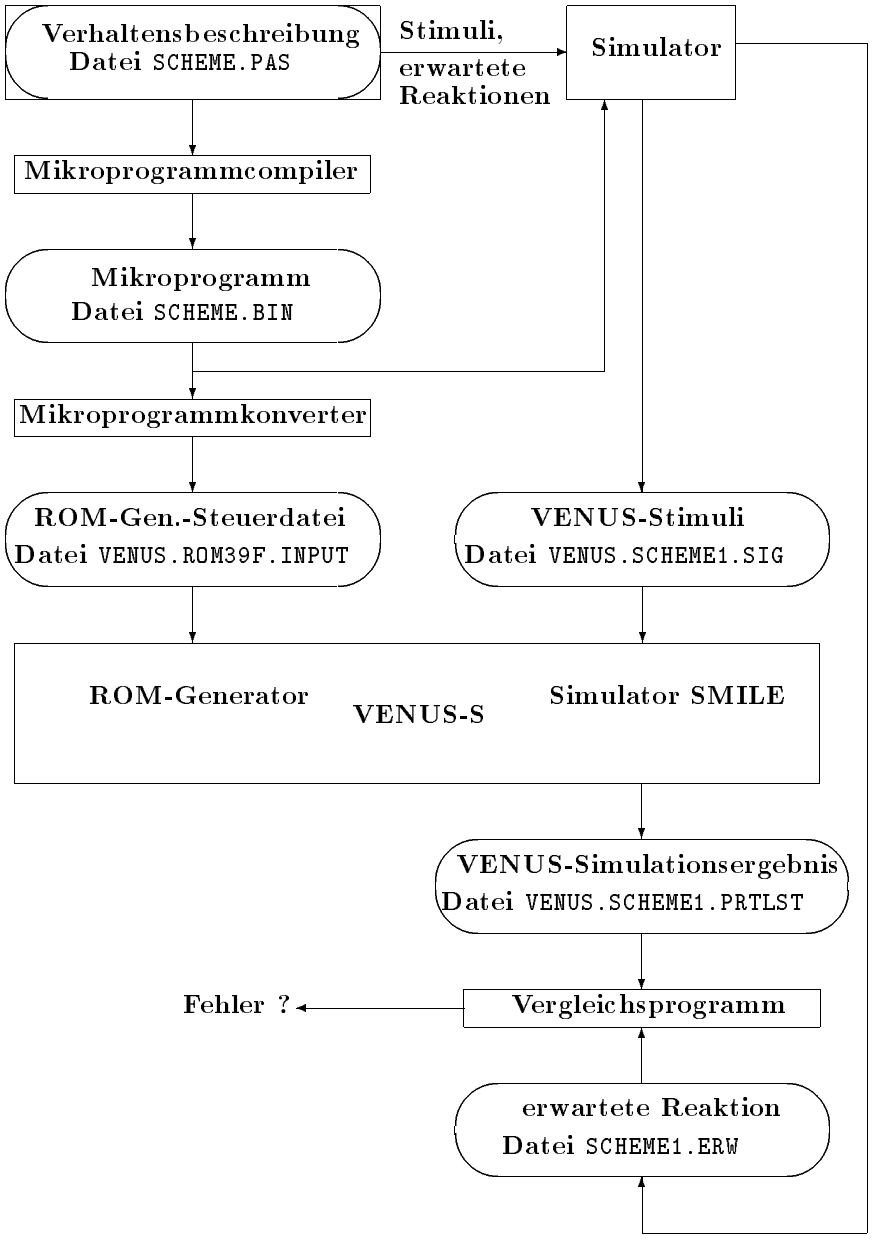



Abbildung 8.22: Simulation des Steuerwerks auf Gatterebene.

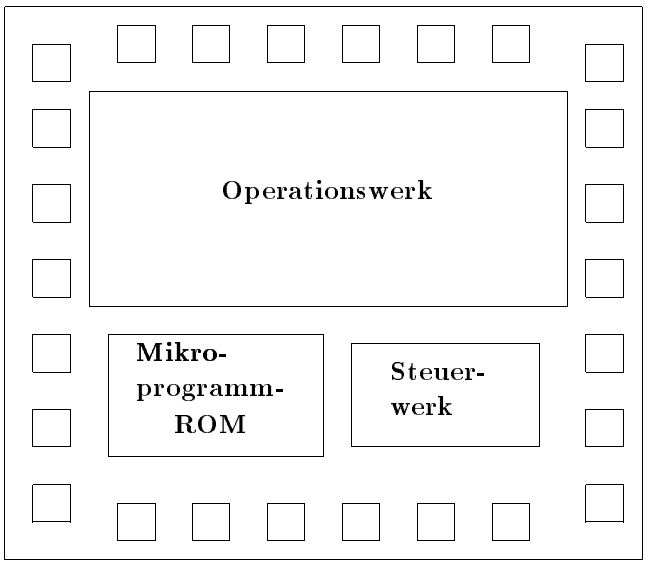



Abbildung 8.23: Plazierung des Scheme-Prozessors.

| | |
|---|---|
| Anzahl der Standardzellen | 1962 |
| Anzahl der Transistoren | $> 76000$ |
| Anzahl der Padzellen | 82 |
| Anzahl der Gatteräquivalente (ohne ROM) | 8095 |
| Fläche in $mm^2$ | 77,3 |

Tabelle 8.15: Statistik des Scheme-Prozessors.

und Padzellen und besonders ihr zeitlich korrektes Zusammenarbeiten zu verifizieren. Die Simulationen umfaßten wenig mehr als einhundert Taktzyklen. Der diese Simulationen begrenzende Faktor war der Plattenplatz, der belegt wurde, weil der Simulator jeden Signalwechsel eines jeden inneren Schaltungsknotens in eine Plattendatei protokolliert.

#### 8.6.6.2 Gewährleistung der Testbarkeit des Prozessors

Zur Fertigung eines integrierten Schaltkreises sind als Entwurfsergebnisse das Layout der Schaltung und die Testmuster zu liefern. Die Testmuster erlauben dem Hersteller die Überprüfung jedes gefertigten ICs auf Herstellungsfehler.

Beim Entwurf des Scheme-Prozessors wurde die Testbarkeit berücksichtigt. Jeder interne Schaltungsknoten kann auf einen Kurzschluß mit logisch "0" und logisch "1" überprüft werden. Die Testbarkeit wird dadurch erreicht, daß sowohl das Steuerwerk als auch das Operationswerk für sich testbar sind und durch den in Abbildung 8.10 auf Seite 90 gezeigten Trenn-Scanpath (boundary-scan-path) zwischen Steuerwerk und Operationswerk auch die Verbindungen zwischen Steuerwerk und Operationswerk steuerbar und beobachtbar gemacht wurden.

Nachdem die Testbarkeit auf höherer Entwurfsebene durch den Scanpath prizipiell gewährleistet wurde, verbleibt mit der Fertigstellung des Prozessors die Aufgabe, die Testmuster für die gesamte Schaltung zu erstellen. Für die Erstellung der Testmuster galt es, folgende Testmuster über den Scanpath an ihr Zielmakro zu bringen und die Reaktionen an den Ausgängen dieses Zielmakros zu beobachten:

- Testmuster für das Operationswerk. Diese Testmuster wurden auf der Ebene von Mikrobefehlen manuell erstellt und durch den Stimulicompiler in Stimuli für den Logiksimulator umgesetzt.
- Testmuster für das ROM. Diese Testmuster beinhalten das Auslesen des ROMs unter allen Adressen. Sie wurden durch ein Programm erstellt.
- Testmuster für die Kombinatorik des Steuerwerks. Da das Steuerwerk neben $\mu$PC, Mikroprogrammstack und Pipelineregistern aus umfangreichen kombinatorischen Schaltungsteilen besteht, wurden die speichernden Elemente aus einer Kopie des Steuerwerks entfernt, Testmuster für die verbliebenen kombinatorischen Schaltungsteile automatisch generiert und diese Testmuster durch ein dafür erstelltes Programm um das Ein- und Ausschieben über den Scanpath erweitert. Diese Muster testen das Steuerwerk mit Ausnahme von ROM und Stack.
- Testmuster für den Mikroprogrammstack. Diese Testmuster wurden manuell erstellt.

Ein in Pascal erstelltes Programm von etwa 450 Zeilen kombiniert die aufgezählten Testmuster zu Testmustern für den gesamten Scheme-Prozessor. Die dabei entstandenen Muster haben eine Länge von fast 600 KByte. Eine Fehlersimulation mit diesen Mustern konnte nicht erfolgen, da schon das Modell des ROMs für den Fehlersimulator zu umfangreich war. Für die Fehlersimulation erzeugt der ROM-Generator eine Ersatzschaltung aus Gattern. Aufgrund einer Unzulänglichkeit in VENUS-S stürzt der Fehlersimulator beim Einlesen des ROMs ab. Daher mußte auf die Ermittlung der Testüberdeckung durch eine Fehlersimulation verzichtet werden.

# Kapitel 9

# Die Ergebnisse

In diesem Kapitel sollen die Ergebnisse des Prozessorentwurfs dargestellt werden.

Ergebnisse sind auf den Chipentwurf bezogen Layout und Testmuster sowie elektrische Parameter wie die erreichbare Taktfrequenz. Auf die Implementierung der Sprache Scheme bezogen werden unter Ergebnissen Vergleiche des Scheme-Prozessors mit anderen Implementationen verstanden. Ergebnisse in Relation zu den zuvor entworfenen Scheme-Prozessoren können im Vergleich mit dem auf den Chips implementierten Sprachumfang, der Geschwindigkeit und der genutzten Fläche erfolgen.

Der Entwurf des letzten vorhergehenden Scheme-Prozessors stammt aus dem Jahre 1981. Abschließend sollen daher neuere Architekturen für LISP vorgestellt werden und ihre Realisierbarkeit mit den hier zur Verfügung stehenden Mitteln verglichen werden.

## 9.1 Das Chip

Das im April 1989 neben den Testmustern zur Fertigung eingereichte Layout ist in Abbildung 9.1 gezeigt. Weil der Prozessor sogar weniger Platz als die zur Fertigung einzureichende Chipgröße belegt, wurde die freie Fläche am unteren Rand und an der rechten Seite hinzugefügt. Entsprechend der schematischen Angabe in Abbildung 8.23 sind im Layout oben das Operationswerk, unten das Steuerwerk und zwar links das Mikroprogramm-ROM und rechts das Folgeadreßrechenwerk plaziert.

Das Layout wurde im April 1989 bei der Gesellschaft für Mathematik und Datentechnik (GMD) in Birlinghoven eingereicht. Dort werden die Chip-Entwürfe der Hochschulen Prüfungen unterzogen, zu Reticles zusammengefaßt und zur Fertigung an Siemens weitergeleitet.

Bei diesen Prüfungen an der GMD stellte sich heraus, daß der Scheme-Prozessor das komplexeste jemals von einer bundesdeutschen Hochschule zur Fertigung eingereichte Chip ist [Stum89]. Für eine dieser Prüfungen, den Design-Rule-Check (DRC), eine Prüfung auf das Einhalten geometrischer Regeln, mußte der Prozessor sogar zerlegt werden, da das Prüfprogramm von Siemens nur für kleinere Chips konzipiert war [Stum89]. Der Scheme-Prozessor enthält über 76000 Transitoren, obwohl für die verwendete Chipgröße laut Herstellerangaben nur maximal 36000 (vgl. Tabelle 6.2 auf Seite 55) erreichbar sind.

Aus dem zur Fertigung eingereichten Layout wurden Leitungslaufzeiten extrahiert. Mit diesen Leitungslaufzeiten wurde eine maximale Taktfrequenz von 10,3 MHz simulativ ermittelt.

Im November 1989 wurden zwanzig ungetestete Exemplare der gefertigten Scheme-Prozessoren geliefert. Die Prozessoren wurden auf Fertigungsfehler getestet und in einem Atari ST eingebaut. Die Tests ergaben elf bei 8 MHz Taktfrequenz im Atari ST funktionsfähige Prozessoren.

Abbildung 9.2 zeigt einen der gefertigten Prozessoren mit geöffnetem Gehäuse, so daß das Chip

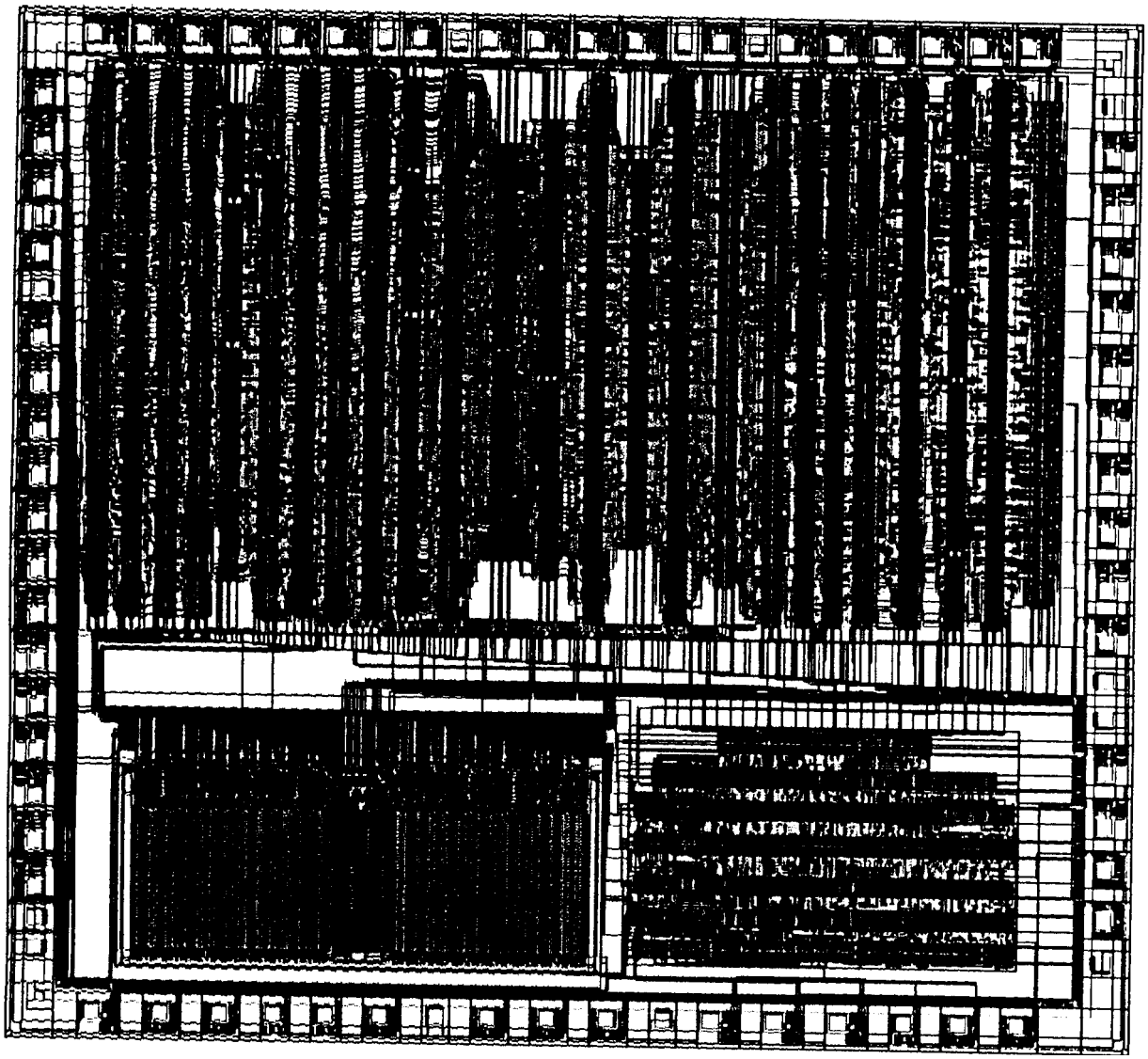


Abbildung 9.1: Das Layout des Scheme-Prozessors.

sichtbar ist. In Abbildung 9.3 ist eine vergrößerte Photographie des gefertigten Chips gezeigt.

Die Softwareumgebung für die Prozessoren wurde aus dem Verhaltenssimulator entwickelt. Im Verhaltenssimulator wurde die Hauptschleife, die den Bytecode interpretiert, durch einen Aufruf des Prozessors ersetzt. Ferner wurden Zugriffe auf die Variablen, die im Simulator die Prozessorregister repräsentieren, durch Zugriffe auf die Register ersetzt. Diese Softwareumgebung für den Scheme-Prozessor verweilt die weitaus meiste Zeit in der `STOP`-Instruktion [68000] des 68000 und wartet auf einen Interrupt des Scheme-Prozessors, wodurch der Hauptspeicher dem Scheme-Prozessor fast immer exklusiv zur Verfügung steht. Die Softwareumgebung ist ein vom Verfasser erstelltes Pascalprogramm von 2200 Zeilen Länge und umfaßt 120 Zeilen in Assembler für pri-

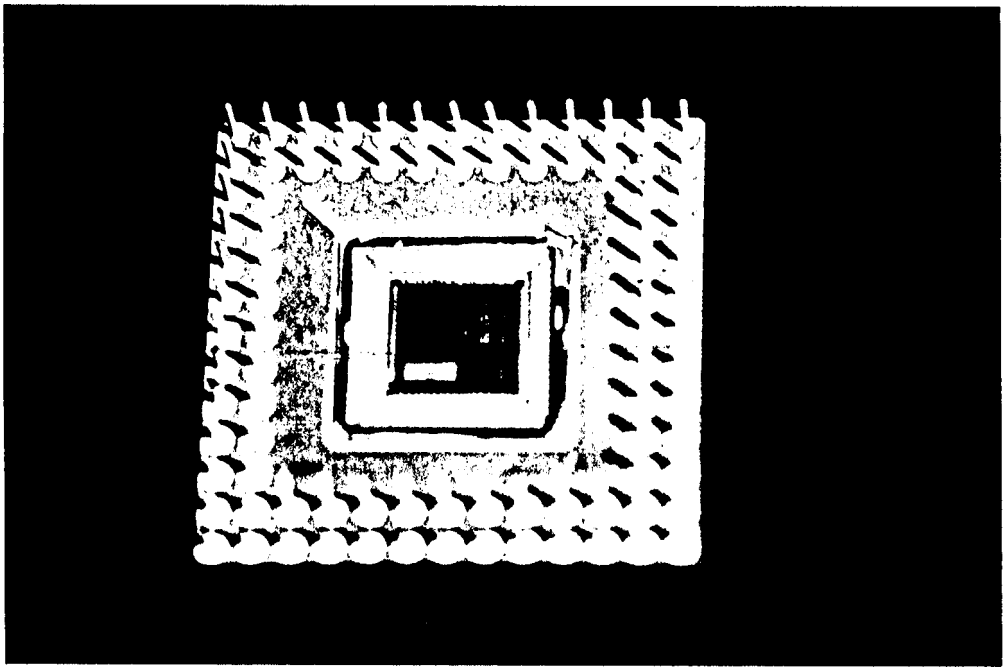


Abbildung 9.2: Photo des Scheme-Prozessors im geöffneten Gehäuse.

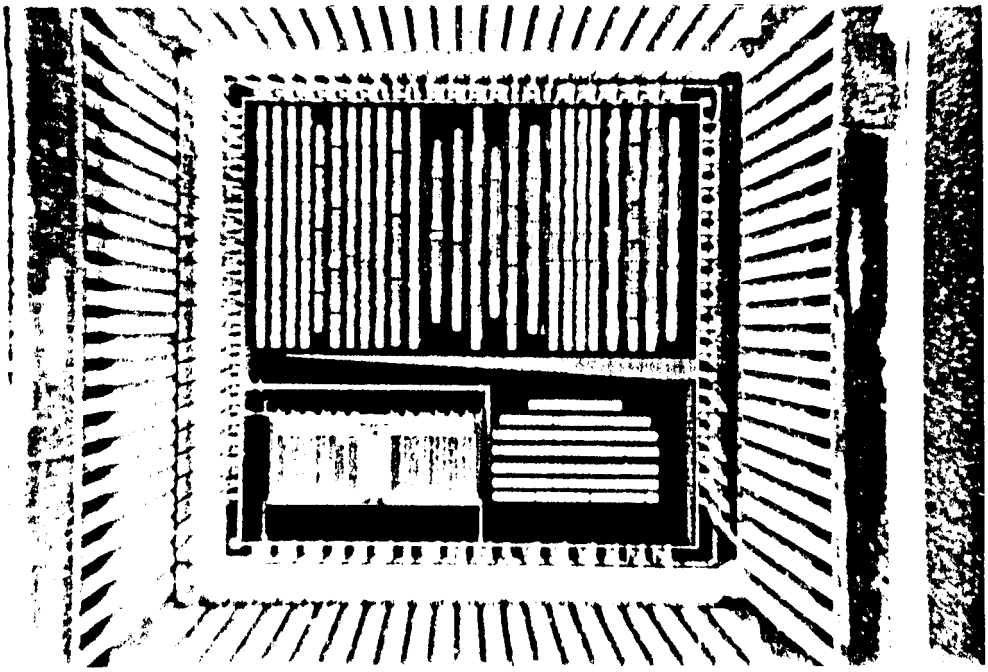


Abbildung 9.3: Photo des Chips im Gehäuse.

vilegierte Befehle und Interruptbearbeitung. Die Softwareumgebung wurde bisher hauptsächlich genutzt, um den Scheme-Prozessor zu erproben und Benchmarks zu rechnen.

## 9.2 Die Implementation

Um die Implementation im Vergleich zu anderen 68000-basierten Implementationen zu bewerten, wurden Vergleichsprogramme (Benchmarks) gerechnet. Gabriel hat in [Gabr85] eine Serie von Benchmarkprogrammen veröffentlicht, die zum Vergleich von LISP-Programmen herangezogen werden. Diese Programme wurden auch mit dem gefertigten Scheme-Prozessor gerechnet und die Resultate zusammen mit Vergleichsdaten aus der Literatur in Tabelle 9.1 aufgeführt. Die Daten für XScheme auf einem Atari ST wurden vom Verfasser ermittelt, MacScheme in Bytecode auf einem Macintosh Plus mit 512 KByte Hauptspeicher wurden [BaJe86] entnommen, und die Daten für MacScheme in Maschinencode auf einem Macintosh SE mit 1 MByte Hauptspeicher stammen aus [Clin88]. Der angegebene Wert ist die Laufzeit in Sekunden als Mittelwert von 3 – 10 Programmläufen direkt nach einer Garbage Collection [Gabr85]. Das Programm `FIB` (rekursive Fibonacci-Funktion) ist in [BaJe86] zusammen mit dem angegebenen Wert für MacScheme veröffentlicht. Die Werte für den Scheme-Prozessor wurden auf einem Atari MegaST mit 8 MHz Taktfrequenz gemessen. Dem Scheme-Prozessor wurden dabei 250 KByte Speicher zugeordnet. Im Anhang ist die dabei verwendete externe Beschaltung für den Scheme-Prozessor im Atari ST angegeben.

| Benchmark | XScheme auf Atari ST | MacScheme Bytecode | MacScheme 68000-Code | Scheme-Prozessor im Atari ST |
|---|---|---|---|---|
| `TAK` | 169,9 | 72,24 | 10,5 | 4,86 |
| `FIB 20` | 36,7 | 22,50 | 8,28[1] | 1,76 |
| `DIV2` rekursiv | 258,1 | 143,69 | 52,16[1] | 10,70 |
| `DIV2` iterativ | 253,8 | 60,17 | 36,15[1] | 7,66 |
| `DERIV` | 244,3 | 243,36 | 112,4 | 16,71 |

Tabelle 9.1: Vergleich des Scheme-Prozessors mit anderen 68000-Implementationen.

Der Vergleich mit den 68000-Implementationen zeigt eine erhebliche Steigerung der Geschwindigkeit gegenüber den anderen Bytecode-Implementationen. Die gemessenen Werte übersteigen sogar den ursprünglich angestrebten Faktor 10.

Auch gegenüber realem 68000-Code ist der Scheme-Prozessor um etwa den Faktor 2 – 7 schneller. Besonders bedeutend ist der Geschwindigkeitsvorteil des Scheme-Prozessors im Benchmark `DERIV`, der symbolischen Ableitung eines Polynoms. `DERIV` ist — im Gegensatz zu `TAK` — eine Aufgabe, die sehr an LISP orientiert ist und die nicht so effizient wie `TAK`, das nur Prozeduraufrufe und Integer-Arithmetik beinhaltet, in Maschinencode übersetzbar ist. Ferner ist zu berücksichtigen, daß die Prozessor-Implementation gegenüber einer Übersetzung in Maschinencode mit einem Bruchteil des Hauptspeichers für Programme und Listen auskommt und damit bei gleicher Speichergröße wesentlich umfangreichere Systeme zuläßt.

Speziell für den Atari ST als Zielrechner ist die Implementation mit dem Scheme-Prozessor ein gewaltiger Fortschritt, da der Geschwindigkeitsgewinn gegenüber XScheme, der einzigen Scheme-Implementation auf dem Atari ST, besonders hoch ist.

[1] Dieses Ergebnis ist nicht in [Clin88] veröffentlicht. Es wurde an der Universität Oldenburg gerechnet.

## 9.3 Vergleich mit Scheme-81

Ein Vergleich mit den beiden beim M.I.T. vorgenommenen Entwürfen braucht dieser Scheme-Prozessor nicht zu scheuen. Tabelle 9.2 zeigt einen Vergleich einiger Parameter der beiden Chips und der auf ihnen untergebrachten Funktionalität.

Der Vergleich zeigt, daß beide Prozessoren mit einem $2\,\mu$-Prozeß gefertigt wurden. Dieser Scheme-Prozessor entstand gegenüber Scheme-81 unter diversen erschwerenden Bedingungen:

- Er entstand in CMOS, das gegenüber NMOS für die gleichen Funktionen mehr Transistoren benötigt.
- Er entstand in Standardzelltechnik, wodurch die Chipfläche weniger gut nutzbar war als in Full Custom.
- Es stand mit $77{,}3\,\mathrm{mm}^2$ weniger Chipfläche als die von Scheme-81 belegten $85\,\mathrm{mm}^2$ zur Verfügung.
- Es muß den vorhandenen, nur 16 Bit breiten Hauptspeicher eines existierenden Rechners nutzen.

Selbst unter diesen durch das Entwurfssystem, die Fertigung und den Einsatzbereich gegebenen Randbedingungen entstand ein Scheme-Prozessor, der einen wesentlich größeren Teil der Sprache als Scheme-81 implementiert, der Stack und Heap unterstützt und eine Vielzahl von Standardprozeduren und Objekte implementiert.

| | Scheme-81 | Scheme-Prozessor |
|---|---|---|
| Technologie | $2\,\mu$ NMOS | $2\,\mu$ CMOS |
| Design | Full Custom | Standardzell |
| Fläche | $85\,\mathrm{mm}^2$ | $77\,\mathrm{mm}^2$ |
| implementierte Standardprozeduren | Listenprozeduren<br>Garbage Collection | Listenprozeduren<br>Garbage Collection<br>Vektorprozeduren<br>Stringprozeduren<br>numerische Prozeduren<br>Charprozeduren |
| Speicher | Heap | Stack u. Heap |
| unterstützte Objekte | FIXNUM<br>CONS-Zellen<br>leere Liste | FIXNUM<br>CONS-Zellen<br>leere Liste<br>CDR-kodierte Listen<br>Vektoren<br>Strings<br>Char<br>Ports<br>Controlframes<br>Bindungsumgebungen |

Tabelle 9.2: Vergleich des Scheme-Prozessors mit Scheme-81.

## 9.4 Neuere Veröffentlichungen über Prozessorentwürfe für Scheme und LISP

An dieser Stelle soll ein Überblick über neuere Prozessorentwürfe in der Literatur geboten werden. Unter neueren Entwürfen sind hier solche zu verstehen, über die bei Einreichung des Scheme-Prozessors zur Fertigung noch keine Literatur vorlag.

Inzwischen ist Scheme-86, ein weiterer Scheme-Prozessor vom M.I.T. [BeWu88a], beschrieben worden. Ferner wird ein LISP-Prozessor der Firma Texas Instruments stellvertretend für den Stand der Technik bei industriellen Prozessoren vorgestellt.

### 9.4.1 Scheme-86

Scheme-86 [BeWu88a, BeWu88b, Wu88] ist ein weiterer Scheme-Prozessor, der am M.I.T. entworfen wurde. Scheme-86 arbeitet, wie seine Vorgänger Scheme-79 [Suss81] und Scheme-81 [Bata82], einen S-Code ab. Auch bei Scheme-86 wird das Abarbeiten des S-Codes gegenüber Maschinencode als Vorteil dargestellt und der Prozessor als interpretierende Implementation beschrieben.

Das Operationswerk von Scheme-86 beinhaltet eine Registerbank von 64 Registern von 32-Bit Breite mit drei Schreibports und acht Leseports. Es enthält zwei ALUs und drei Komparatoren. Das Steuerwerk ist mikroprogrammiert. Das manuell erstellte Mikroprogramm umfaßt 4096 Worte von 160 Bit Breite und ist in einem RAM mit 35 ns Zugriffszeit untergebracht [Wu88].

Scheme-86 wurde nie aufgebaut. Simulationen erfolgten unter Annahme der Verzögerungszeiten bei Verwendung der Advanced Schottky TTL-Technik, der schnellsten auf dem Markt erhältlichen TTL-ICs. Eine Taktfrequenz von 6,1 MHz wurde simulativ erreicht.

Mit Scheme-86 ergab der `TAK`-Benchmark eine Ausführungszeit von 0,580 Sekunden. Das ist ein Geschwindigkeitsfaktor von 7,8 gegenüber dem in dieser Arbeit beschriebenen Prozessor.

Im Vergleich mit dem in dieser Arbeit beschriebenen Scheme-Prozessor, muß berücksichtigt werden, daß Scheme-86 unter anderen Randbedingungen als eine integrierte Schaltung entworfen wurde. Allein das Mikroprogramm-RAM von Scheme-86 hat die 16,4fache Kapazität des ROMs des in dieser Arbeit beschriebenen Scheme-Prozessors. Ein ROM der zehnfachen Fläche würde schon die hier zur Verfügung stehende Chipfläche übersteigen. Ein RAM weist noch einmal mindestens die sechsfache Fläche eines ROMs auf. Auch eine Registerbank mit einer derartig großen Anzahl von Registern und Ports ist, wegen der benötigten Fläche für Leitungen, kaum VLSI-gerecht.

Der direkte Vergleich der beiden Prozessoren ist nicht möglich, weil die Beschränkung des langsamen Hauptspeicherzugriffs bei Scheme-86 nicht vorliegt. Scheme-86 führt in jedem Taktzyklus einen Hauptspeicherzugriff aus. Er werden statische RAMs mit 45 ns Zugriffszeit für den Hauptspeicher eingesetzt.

Scheme-86 ist damit ein diskret auf mehreren Platinen aufgebauter Prozessor aus den teuersten marktgängigen Standard-ICs. Ein solcher Prozessor ist nach Ansicht des Verfassers im Gegensatz zu IC-Entwürfen seit Jahren veraltet.

Verglichen mit dem in dieser Arbeit beschriebenen Scheme-Prozessor ist Scheme-86 nicht VLSI-gerecht entworfen. Ferner beinhaltete die Arbeit die Simulation von Scheme-86 auf allen Entwurfsebenen in Scheme und das Erstellen von Hilfssoftware in Scheme für den Entwurf der (nicht fertiggestellten) Platinen für Scheme-86 [Wu88].
Am Ende seiner Arbeit zieht Wu folgendes Fazit:

> „As a result of a lack of time, the SCode abstraction was taken wholesale from previous implementations of Scheme as instruction set for this machine.
> ...
> Future work on this type of architecture must focus on designing an instruction set

> that fully exploits the architecture and many known techniques of compiling Scheme.“
> (Seite 34 in [Wu88])

Gerade diese Überlegungen über eine effiziente und (speziell für unsere Randbedingungen) kompakte Repräsentation der Programme und die Entscheidung für eine compilierende Implementation wurde in dieser Arbeit geleistet. Hier wurde keine Programmdarstellung gewählt, nur weil sie vorhanden war, sondern die Repräsentation der Programme als Schlüsselproblem erkannt und entsprechend ihrer Bedeutung berücksichtigt. Dafür wurde bestehende Entwurfssoftware genutzt, soweit sie vorhanden war und zusätzliche Hilfssoftware auch in Pascal und C erstellt. Es mußte nicht jedes Programm in Scheme erstellt werden. Scheme behält auch dann seine Berechtigung als Programmiersprache, wenn man es nicht für *jedes* Programm einsetzt.

### 9.4.2 Texas Instruments' LISP-Chip

Ein LISP-Prozessor von Texas Instruments soll hier zum Vergleich vorgestellt werden. Der Prozessor enthält das Operationswerk und das Steuerwerk ohne Mikroprogrammspeicher auf einem Chip [Boss87, Matt87]. Das Chip soll eine bestehende Implementation — den TI Explorer —, der aus SSI und MSI ICs aufgebaut ist, so kompakt gestalten, daß er für militärische Anwendungen einsetzbar wird [Matt87], und mit dem LISP-Prozessor eine fünffache Geschwindigkeitssteigerung gegenüber dem TI Explorer erreichen [Boss87].

Abbildung 9.4 zeigt einen Teil der Struktur des LISP-Chips von TI. Die beiden Eingänge (`A` und `M`) der ALU werden aus verschieden Quellen gespeist: Das `A-Memory` ist die einzige Quelle für den Operanden `A`. Es ist ein Temporärspeicher von 1024 Worten, der dem Mikroprogrammierer zur Verfügung steht. Der ALU-Eingang `M` hat als eine Quelle auch einen solchen Temporärspeicher `M-Memory` mit 64 Worten. Eine andere Datenquelle von `M` ist das `PDL-Memory`, ein Speicher von 1024 Worten, der als Speicher für den Stack genutzt wird. Das LISP-Chip von TI implementiert einen stackorientierten Befehlssatz, und im `PDL-Memory` stehen somit die Operanden der Makrobefehle. Weitere Funktionseinheiten — wie Stackpointer, Speicherdatenregister und Makrobefehlsqueue — sind in Abbildung 9.4 nicht dargestellt. Sie liefern auch Werte an den ALU-Eingang `M`.

Den Speichern `A-Memory`, `M-Memory` und `PDL-Memory` ist je ein Scheibpuffer von 4 Worten zugeordnet. Schreiboperationen schreiben Adresse und Datum in diesen Schreibpuffer. In Taktzyklen, in denen der Speicher nicht ausgelesen wird, wird das Datum dann in den Speicher geschrieben. Wenn ein Schreibpuffer überläuft, wird der Prozessor für eine Taktperiode angehalten und ein Datum aus dem Schreibpuffer in den Speicher geschrieben. Ferner sind diese Speicher dynamisch realisiert, sie enthalten Refreshlogik und haben die Möglichkeit, den Prozessor für einen Refresh anzuhalten.

Die ALU beinhaltet einen Datenpfad über einen Shifter und einen Masker. Shifter und Masker dienen unter anderem dem Extrahieren und Modifizieren von einzelnen Feldern der Worte. Sie unterstützen so die Einteilung der Worte in Felder.

Das in der Abbildung nicht gezeigte mikroprogrammierbare Steuerwerk wird durch einen Speicher mit 2,5K Worten von 18 Bit Breite auf dem Chip unterstützt. Dieser Speicher enthält Sprungtabellen. Ferner sind auf dem Chip der Befehlsdekoder und der Mikroprogrammstack mit 64 Worten von 32 Bit Breite. Das Mikroprogramm ist außerhalb des Chips in einem 16K $\times$ 64 Bit organisierten RAM abgelegt. Ein 16Kbit ROM auf dem Chip enthält Programme für den Selbsttest des Prozessors und zum Urladen seiner Mikroprogramme.

Tabelle 9.3 zeigt einige statistische Daten des TI-LISP-Chips. Neben der Größe des Chips und der Transistorzahl fällt der sehr moderne Fertigungsprozeß auf. Zudem geht aus den Daten hervor, daß das Chip überwiegend aus Speicher besteht. Über 88 % aller Transistoren sind in RAM oder ROM eingesetzt.

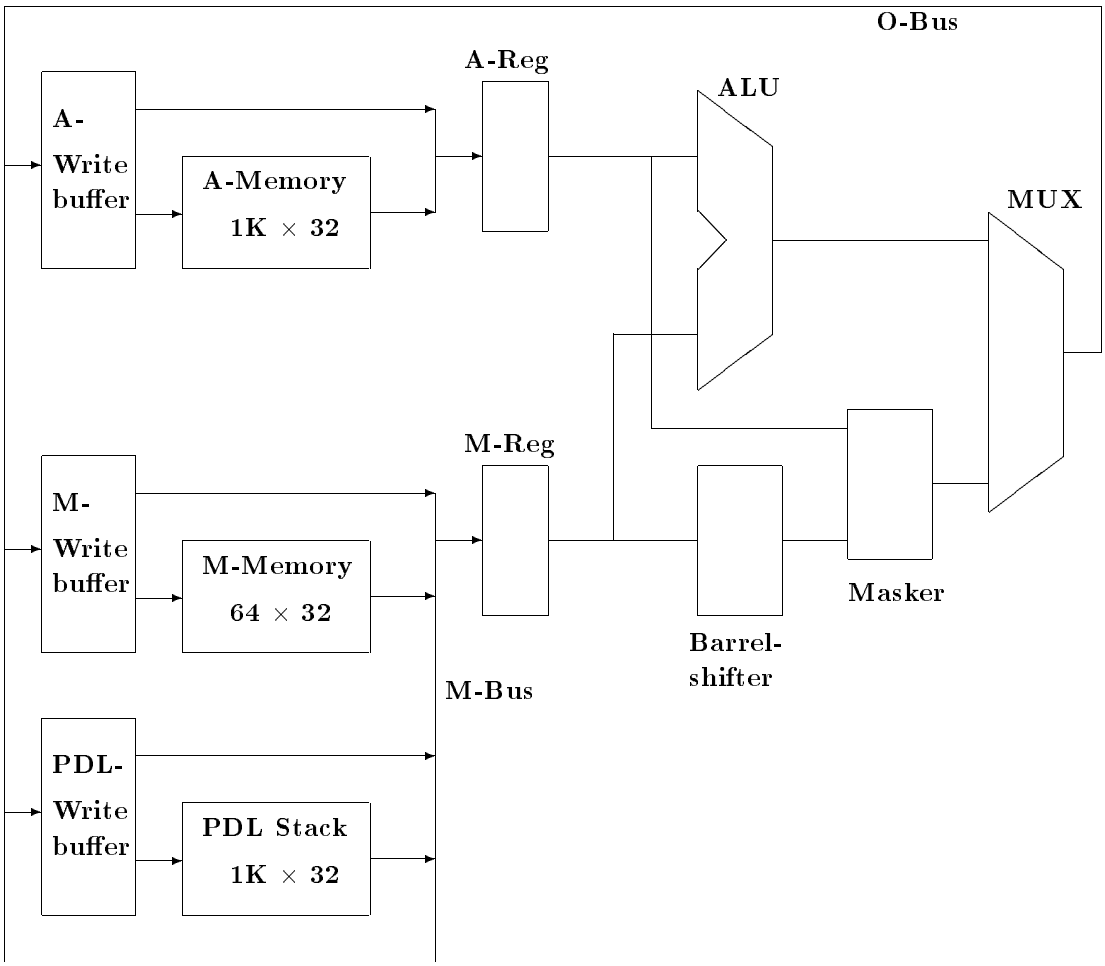



Abbildung 9.4: Operationswerk des LISP-Prozessors von TI.

| | |
|---|---|
| Technologie | 1,25 $\mu$ CMOS |
| Entwurfsart | Full-Custom |
| Fläche | 100 $mm^2$ |
| Transistoren | 553687 |
| RAM Bits | 114K |
| ROM Bits | 16K |
| Anschlüsse | 224 |
| Taktfrequenz | 33 MHz |

Tabelle 9.3: Statistik des LISP-Chips von TI.

Das TI-Chip wurde so entworfen, daß er sowohl auf der Ebene seiner Maschinenbefehle als auch der seiner Makrobefehle kompatibel zum TI Explorer ist. Damit kann die vorhandene Software und Firmware weiterhin genutzt werden. Unter dieser Vorgabe wurde damit eine Architektur auf ein Chip gebracht, die als Neuentwurf sicherlich VLSI-gerechter erfolgt wäre. Interessante Daten über die Firmware, beispielsweise Angaben über die durchschnittliche Anzahl der in `A-Memory` und `M-Memory` tatsächlich genutzten Speicherplätze, sind leider nicht veröffentlicht.

## 9.5 Bewertung der Ergebnisse

Es wurde systematisch ein Prozessor entworfen, der auf einem Chip einen wesentlich größeren Teil der Sprache implementiert als die bisher entworfenen Scheme-Prozessoren. Alle nicht im Mikroprogramm implementierten Standardfunktionen werden durch den Wirtsprozessor oder durch Folgen von Maschinenbefehlen realisiert. Damit ist es auch der erste Scheme-Prozessor, der als Mittelpunkt einer vollständigen Implementation der Programmiersprache Scheme dienen kann.

Der Prozessor ist als Koprozessor für vorhandene 68000-Rechner konzipiert. Dieses Konzept erlaubt eine preiswerte Beschleunigung dieser Rechner. Der Prozessor erlaubt im Vergleich mit bestehenden Softwareimplementationen eine sehr große Geschwindigkeitssteigerung.

Im Vergleich mit den vorhandenen Scheme-Prozessoren enthält der hier beschriebene Prozessor mehr Funktionalität auf kleinerer Fläche und ist für einen praktischen Einsatz konzipiert.

Verglichen mit neueren Arbeiten (Scheme-86 und TIs LISP-Chip) ist dieser Prozessor VLSI-gerecht entworfen und trotz der beschränkten Fertigungsmöglichkeiten der Hochschulen als Ein-Chip-Prozessor realisiert.

# Kapitel 10

# Schlussbetrachtungen

Diese Schlußbetrachtungen sollen einen Ausblick auf weitere Arbeiten bieten und berücksichtigungswürdige alternative Vorgehensweisen sowie die Abhängigkeit der gewählten Lösung von den Randbedingungen darstellen.

## 10.1 Ausblick

Diese Arbeit beschreibt den Entwurf eines Prozessors, die dabei getroffenen Implementierungsentscheidungen, die Randbedingungen und das erzielte Ergebnis. Im diesem Ausblick wird auf noch nötige Arbeiten eingegangen, die auf dem entstandenen Prozessor aufbauend die Scheme-Implementation vervollständigen. Ferner werden auch Variationen der Randbedingungen und die daraus resultierenden Varianten im Entwurfsprozeß und Prozessor diskutiert.

### 10.1.1 Vervollständigung der Implementation

Der bis zu dieser Stelle beschriebene Prozessorentwurf resultierte in als Chip gefertigten Scheme-Prozessoren. Die Prozessoren wurden in einen Atari ST eingesetzt und mit einer Softwareumgebung versehen, die die Ausführung von Scheme-Programmen ermöglicht. Die wichtigsten bisher ausgeführten Programme sind die Benchmarks (vergl. Tabelle 9.1 auf Seite 118), die den Vergleich mit anderen Scheme-Implementationen erlauben.

Für eine Implementation einer Programmiersprache ist ein schnelles und speichersparendes Ablaufen der Zielprogramme zwar ein sehr wichtiges, aber nicht alleiniges Kriterium für einen Einsatz. Wichtig sind auch die Vollständigkeit der Implementation, die über den Sprachstandard hinausgehenden bereitgestellten Erweiterungen und eine benutzerfreundliche Gestaltung.

Der Scheme-Prozessor und seine Softwareumgebung bieten zur Zeit:

- Zahlreiche Standardprozeduren nach [Rees86].
- Einen Compiler, der jeden eingelesenen Ausdruck vor der Auswertung übersetzt.
- Einen Debugger, der bei Aufruf und Verlassen einer Prozedur aufrufbar ist und bei Fehlern automatisch aufgerufen wird. Der Debugger erlaubt die Anzeige von Variableninhalten und eines Tracebacks sowie das Disassemblieren von Prozeduren.
- Einen einfachen bildschirmorientierten Editor.

- Eine Prozedur zum Auslesen des vom Betriebssystem des Atari ST bereitgestellten 200 Hz Zählers. Diese Prozedur wurde zur Unterstützung der Benchmarkrechnungen hinzugefügt.

Wünschenswerte Erweiterungen fallen in mehrere Klassen. Eine Klasse besteht in durch den Prozessor unterstützten aber bisher nicht in der Softwareumgebung realisierten Erweiterungen. Solche Erweiterungen sind Zahlen der Typen `FLONUM` und `BIGNUM`. Ferner könnte die Softwareumgebung bei einem Überlauf des Stacks den Stackinhalt in den Heap kopieren, so wie es auch der Befehl `CALLCC` macht und den Scheme-Prozessor anschließend weiterarbeiten lassen (vergl. [BaJe86]). Bisher wird bei Stacküberlauf mit einer Fehlermeldung abgebrochen.

Eine andere Klasse von Erweiterungen besteht im Hinzufügen von in der Softwareumgebung realisierten Prozeduren. In diese Gruppe fallen Anbindungen an die Graphiksystemkomponenten (AES, VDI) des Atari ST und an weitere Betriebssystemaufrufe (GEMDOS, BIOS, XBIOS). Insbesondere bei den Graphikaufrufen ist die Möglichkeit, den Scheme-Prozessor *gleichzeitig* mit der Softwareumgebung arbeiten zu lassen, eine bisher im Atari ST nicht mögliche Neuerung.

Die letzte Klasse von Erweitungen ersetzt bestehende Komponenten durch verbesserte. Beispielsweise ist der Prozessor so konzipiert, daß jeder Befehl und auch die Garbage Collection durch die Softwareumgebung ersetzt werden kann. Daher läßt sich etwa mit einer inkrementellen Garbage Collection experimentieren und diese später bei einer erneuten Fertigung des Prozessors in das Mikroprogramm verlagern. Ferner sollte insbesondere der Compiler durch einen neuen ersetzt werden, welcher Closure- und verbesserte Environmentanalysen vornimmt. Der bestehende Compiler übersetzt jeden $\lambda$-Ausdruck in eine Closure, und er macht eine sehr pessimistische Environmentanalyse: Jede Prozedur, die einen $\lambda$-Ausdruck enthält, legt ihre Bindungsumgebung im Heap an. Trotz dieses bestehenden Compilers wurden die beschriebenen Ergebnisse bei den Benchmarks erzielt. Es wird aber beispielsweise ein `let`-Ausdruck viel zu ineffizient übersetzt, da dieser bei der Makroexpansion in den Aufruf eines $\lambda$-Ausdrucks überführt wird. Im Fall eines `let`-Ausdrucks muß weder eine neue Closure erzeugt noch die Bindungsumgebung grundsätzlich im Heap alloziert werden. Der neue Compiler wird bereits im Rahmen einer Diplomarbeit entwickelt.

### 10.1.2 Verbesserungsmöglichkeiten des Prozessors

Bei jedem Programm und jeder Hardware entstehen nach der Fertigstellung Wünsche nach Erweiterungen und Verbesserungen. Der Scheme-Prozessor bildet dabei keine Ausnahme. Auch durch Erfahrungen mit dem Prozessor werden beispielsweise bestimmte Befehle als verzichtbar und andere, nicht enthaltene als wünschenswert angesehen. Verbesserungsideen des Verfassers sind:

- Eine inkrementelle Garbage Collection. Durch die unterbrechende Garbage Collection unterscheidet sich der Scheme-Prozessor im Atari ST von kommerziellen LISP-Maschinen. Die Erstellung einer inkrementellen Garbage Collection wurde noch vor Einreichung des Prozessors zur Fertigung in Angriff genommen. Allerdings gelang es in der zur Verfügung stehenden Zeit nicht, diese soweit simulativ zu validieren, daß sie in das Mikroprogramm des gefertigten Prozessors aufgenommen werden konnte.

- Bedingte Sprungbefehle, die das oberste Stackobjekt zwar auf die Sprungbedingung hin untersuchen, es aber auf dem Stack belassen. Solche Befehle, wie sie von Masinter und Deutsch vorgeschlagen werden [MaDe80], erlauben insbesondere eine effizientere Übersetzung von `and` und `or`.

- Die Prozeduren `make-code-vector` und `code-vector-set` könnten vom 68000-Rechner übernommen und dafür beispielsweise `assq` durch einen Prozessorbefehl unterstützt werden.

- Ein Einschrittbetrieb (single-instruction mode) des Prozessors hätte beim Erproben der gefertigten Prozessoren gute Dienste geleistet.

Ob und welche dieser Änderungsideen jemals realisiert werden, hängt davon ab, ob ein anderer Prozessor gefertigt wird. Die Fertigung des Scheme-Prozessors hat etwa DM 150.000–200.000 gekostet. Für relativ kleine Verbesserungen ist ein erneuter Einsatz solcher Mittel kaum zu rechtfertigen.

### 10.1.3 Entwurfsmöglichkeiten bei anderen Randbedingungen

Der Entwurf des Scheme-Prozessors erfolgte unter vorgegebenen Randbedingungen. Zu den Randbedingungen gehörte das Entwurfssystem VENUS-S, da nur dieses zur Verfügung stand und das Bundesministerium für Forschung und Technologie die immensen Kosten für die Fertigung der mit VENUS-S entstandenen Chips übernahm.

Andere Randbedingungen ergeben sich auch, wenn der Scheme-Prozessor für einen anderen Zielrechner (nicht mit 68000-Prozessor) oder als eigenständiges System spezifiziert wäre.

#### 10.1.3.1 Einfluß eines anderen Entwurfssystems

Das Entwurfssystem VENUS-S hatte insbesondere die folgenden einschränkenden Einflüsse auf den Prozessorentwurf: Eine Registerbank mußte aus Standardzellen aufgebaut werden, und das Mikroprogramm-ROM konnte keine größere Kapazität haben.

Beide Einflüsse bestehen in anderen Entwurfssystemen nicht, wobei die anderen Entwurfssysteme auch einen 2 $\mu$m CMOS-Prozeß und die Standard- und Makrozelltechnik unterstützen.

Die Standardzell-Bibliothek der Firma Texas Instruments [TI] ist nicht nur wesentlich umfangreicher als die von VENUS-S [Siem88]. Sie enthält insbesondere Zellen, die den Entwurf des Scheme-Prozessors wesentlich erleichtert hätten: Für eine Registerbank steht die Grundzelle `RF408LH`, die bei 8 Bit Breite 3 Ports und 16 Adressen bietet, zur Verfügung [TI]. Sie hätte eine wesentlich umfangreichere, schnellere und kompaktere Registerbank ermöglicht, wodurch wesentlich mehr Register zur Verfügung gestanden hätten, die einen schnelleren Prozessor und einen registerorientierten Befehlssatz ermöglichten. Ferner sind in der Zellbibliothek auch die Zellen `2901`, `2902`, `2904` und `2910` enthalten. Dabei handelt es sich um zu Prozessoren kaskadierbare sogenannte Slices. Auch damit sind Prozessoren sehr einfach entwerfbar. Zusätzlich erlaubt TI den Einsatz von ROMs, deren Adreßzahl keine Zweierpotenz ist. Diese ROMs können dann umfangreichere Mikroprogramme enthalten, deren Umfang nur durch die Chipfläche begrenzt ist.

Die Entwurfssoftware Solo 2000 [ES2b] der Firma European Silicon Structures (ES2) ermöglicht den Entwurf einzelner Makros in Full-Custom Technik. Damit wäre ein manueller Entwurf der Registerbank in Full-Custom Technik und damit auf minimale Fläche optimiert möglich gewesen. Auch das hätte einen schnelleren, registerorientierten Prozessor ermöglicht. Auch bei ES2 muß das ROM keine Zweierpotenz von Adressen enthalten [ES2a]. Eine weitere Leistungssteigerung des Scheme-Prozessors wäre durch einen Cache für den Stack erreichbar. Ein solcher Cache ist in Full-Custom-Technik mit Solo 2000 realisierbar. Ferner fertigt ES2 sehr preiswert und schnell, da es mit Elektronenstrahldirektbelichtung arbeitet und somit auf die zeit- und kostenintensive Erstellung von Masken verzichten kann.

#### 10.1.3.2 Einfluß eines anderen Zielrechners

Der hier beschriebene Scheme-Prozessor wurde für den Einsatz in einem 68000-Rechner konzipiert. Der Scheme-Prozessor mußte den vorhandenen Hauptspeicher des 68000-Rechners nutzen. Dieses Konzept wurde im Jahre 1988 festgelegt, als Speicherchips nicht nur extrem teuer waren, sondern auch nur Kunden mit langfristigen Verträgen überhaupt beliefert wurden. An dieser Stelle soll

untersucht werden, wie sich Konzept und Architektur unterscheiden, wenn ein anderer Zielrechner als ein 68000-System mit einem solchen Scheme-Prozessor ausgestattet werden soll.

Bei einem Verzicht auf die Nutzung des vorhandenen Hauptspeichers des Zielrechners könnte ein anderes Wortformat gewählt werden. Das beim realisierten Scheme-Prozessor gewählte Wortformat mußte die Wortbreite des vorhandenen Hauptspeichers berücksichtigen. Bei einem Scheme-Prozessor mit eigenem Hauptspeicher könnte das Feld für Datum und Pointer von 24-Bit auf 32-Bit erhöht werden. Damit wären dann 32-Bit `FIXMUN`s und Fließkommazahlen in einem Wort darstellbar (vgl. Symbolics 3600 [Moon85] und SPUR [Kong89]), und die Wortbreite erhöhte sich auf 40 Bit.

Der 68000-Prozessor im Zielrechner konnte bei der Abarbeitung von Scheme-Programmen vom Scheme-Prozessor übertroffen werden, weil der Scheme-Prozessor Typenprüfungen, Prozeduraufrufe und die Speicherverwaltung besser unterstützt, aber sonst ähnliche Merkmale wie der 68000-Prozessor aufweist: beide sind mikroprogrammiert, haben keinen Befehls- und Datencache, arbeiten bei gleicher Taktfrequenz, und sie enthalten eine ähnliche Zahl von Transistoren (etwa 68000 im 68000-Prozessor [DaBr85], über 76000 im Scheme-Prozessor).
Einen leistungsähigeren Prozessor wie den 68020 mit 200.000 Transistoren [DaBr85], Befehlscache und wesentlich höherer Taktfrequenz kann dieser Scheme-Prozessor in seiner Leistung nicht übertreffen. Ein Scheme-Prozessor für Zielrechner mit solchen leistungsfähigen Prozessoren muß auch eine entsprechend höhere Taktfrequenz zulassen und über einen Cache verfügen. Um einen solchen leistungsfähigeren Scheme-Prozessor zu erstellen, werden auch leistungsfähigere Entwurfssysteme und Fertigungsprozesse benötigt.

## 10.2 Anwendung auf neuartige Entwurfssoftware

Für des Entwurf des Scheme-Prozessors wurde ein systematisches Vorgehen eingeführt, das dem restriktiven Zeitplan Rechnung tragen sollte. Um systematisch eine Prozessorarchitektur entwerfen zu können, wurde die als Pascalprogramm vorliegende Verhaltensbeschreibung analysiert. Die erhaltenen Analyseergebnisse umfaßten häufig auftretende Befehlsfolgen, die Verteilung der Register und Registerfelder in Ausdrücken und Zuweisungen, die verwendete Kontrollstrukturen und eine Grammatik der enthaltenen Ausdrücke. Diese Analyseergebnisse beschreiben Anforderungen an die Hardwarearchitektur des entworfenen Prozessors: Die Häufigkeit der Ausdrücke mit zwei Operanden erforderte eine Registerbank mit mindestens zwei Ausgangsports. Die bei der Analyse der Verhaltensbeschreibung ermittelten häufigen Schreib- und Lesezugriffe auf einzelne Felder der Register führte zur Entscheidung für eine Registerbank mit drei Ports, die neben den Portadressen auch die einzelnen Felder direkt adressieren kann. Die Spezifikation der ALU wurde direkt aus der Grammatik der in der Analyse ermittelten Ausdrücke entwickelt. Die ALU kann alle diese Ausdrücke in 20 ns berechnen. Auch die Struktur des Steuerwerks wurde so entworfen, daß sie sowohl den Anforderungen aus der Analyse der Verhaltensbeschreibung dient als auch den Randbedingungen des Entwurfssystems genügt: Ein mikroprogrammiertes Steuerwerk mit einem Mikroprogramm-ROM mit 39 Bit Breite und 1024 Worten, der mit dem Entwurfssystem maximal möglichen ROM-Größe, ermöglicht eine Vielzahl von aus den Kontrollstrukturen der Verhaltensbeschreibung entwickelten Folgeadressierungen.

Diese Analyse und die daraus folgenden Entwurfsentscheidungen sind eine interessante Alternative zu existierenden Synthesewerkzeugen, die automatisch aus einer Verhaltensbeschreibung eine Struktur entwerfen. Solche Synthesewerkzeuge, wie Mimola [Marw86], enthalten eine Bibliothek von verwendbaren Bausteinen und deren Kosten und synthetisieren eine Struktur aus diesen Bausteinen. Diese Synthese aus den festen Bausteinen basiert auf dem traditionellen Ansatz einer diskreten Realisierung. Für einen VLSI-Entwurf scheint der Ansatz beim Entwurf des Scheme-

Prozessors eine berücksichtigenswerte Alternative zu sein, da aus der Verhaltensbeschreibung Daten wie eine Grammatik der verwendeten Ausdrücke oder Häufigkeiten von Ausdrücken mit zwei Registeroperanden ermittelt wurden, die in Hinblick auf zukünftige, den Entwurf unterstützende Expertensysteme auf einem Abstraktionsniveau liegen, auf dem Erklärungen und Fragen an den Entwerfer gerichtet werden können. Auch die beim Entwurf des Scheme-Prozessors erarbeiteten Kriterien für Stack- oder Registerarchitektur und für die Anzahl der Ports der Registerbank können als Regeln in ein solches Expertensystem eingebracht werden.

# Anhang A

# Der Befehlssatz des Scheme-Prozessors

In diesem Anhang wird der vollständige Befehlssatz angegeben, der vom Prozessor durch das Mikroprogramm ausgeführt wird. Zusätzliche Befehle, die bei der Assemblerprogrammierung des Prozessors zwar wie Bytecodebefehle gehandhabt werden, tatsächlich aber als Aufrufe des Wirtsprozessors implementiert sind, werden hier nicht zum Befehlssatz gerechnet.

Die Befehle werden tabellarisch aufgezählt. Die Anzahl der Argumente beschreibt die Anzahl der Bytecodeargumente; auf dem Stack übergebene Objekte sind damit nicht gemeint. Das angegebene Befehlsformat bezieht sich auf die Darstellung in Kapitel 7.5 (Seite 70).

Alle Befehle nehmen Typprüfungen vor. Bei den numerischen Befehlen wird bei Argumenten der Typen `FLONUM` und `BIGNUM` und bei Überläufen zum Wirtsprozessor verzweigt. Das gilt auch für den Befehl `COMPARE`, wenn er auf Zahlen, die nicht beide vom Typ `FIXNUM` sind, arbeitet.

## A.1 Zugriffe auf Variablen

| Zugriffe auf Variablen | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `PUSHLOCAL` | 1 | A | Auslesen einer lokalen Variablen |
| `PUSHVAR` | 2 | B | Auslesen einer übergeordneten Variablen |
| `PUSHGLOBAL` | 1 | A | Auslesen einer globalen Variablen |
| `UNBOUNDP` | 0 | A | Prüfung, ob oberstes Stackobjekt Bindung hat |
| `STORELOCAL` | 1 | A | Zuweisung an lokale Variable |
| `STOREVAR` | 2 | B | Zuweisung an übergeordnete Variable |
| `STOREGLOBAL` | 1 | A | Zuweisung an globale Variable |

## A.2 Zugriffe auf Konstanten

| Zugriffe auf Konstanten | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `PUSHLIT` | 1 | A | Konstante aus Literalbereich |
| `PUSHCLOSURE` | 1 | A | Codevektor aus Literalbereich und die aktuelle Bindungsumgebung bilden Closure |
| `PUSHPINT` | 1 | A | Zahl im Bereich von 0 bis 255 |
| `PUSHNINT` | 1 | A | Zahl im Bereich von $-256$ bis $-1$ |
| `PUSHT` | 0 | A | Boole'scher Wert `#T` |
| `PUSHF` | 0 | A | `#F` bzw. leere Liste |

## A.3 Sprung- und Verzweigungsbefehle

| Sprung- und Verzweigungsbefehle | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `BRANCH` | 2 | B | unbedingter Sprung |
| `BRANCHF` | 2 | B | bedingter Sprung |
| `BRANCHT` | 2 | B | bedingter Sprung |

## A.4 Aufruf und Rückkehr von Prozeduren

| Aufruf und Rückkehr von Prozeduren | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `PUSHCONTROL` | 0 | A | Anlegen eines Controlframes |
| `CALL` | 0 | A | nicht endrekursiver Prozeduraufruf |
| `STACKTRCALL` | 0 | A | endrekursiver Prozeduraufruf, Bindungsumgebung des Aufrufers im Stack |
| `HEAPTRCALL` | 0 | A | endrekursiver Prozeduraufruf, Bindungsumgebung des Aufrufers im Heap |
| `HEAPCONTEXT` | 0 | A | Kopieren der Bindungsumgebung in den Heap |
| `SLIDE` | 1 | A | endrekursiver Aufruf der gerade aktiven Prozedur |
| `RETURN` | 0 | A | Rückkehr von der Prozedur |
| `SUBR` | 1 | A | Aufruf des Wirtsprozessors |
| `CALLCC` | 0 | A | Standardprozedur `call-with-current-continuation` |

## A.5 Typprüfungen und Vergleiche

| Typprüfungen und Vergleiche | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `TYPECHK` | 1 | B | implementiert alle Typprädikate |
| `EQ` | 0 | A | Standardprozedur `eq?` |
| `COMPARE` | 1 | B | Vergleich von Zahlen und Zeichen |

## A.6 Listen- und Vektorbefehle

| Listen- und Vektorbefehle | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| CONS | 0 | A | Standardprozedur `cons` |
| CAR | 0 | A | Standardprozedur `car` |
| CDR | 0 | A | Standardprozedur `cdr` |
| SETCAR | 0 | A | Standardprozedur `set-car!` |
| SETCDR | 0 | A | Standardprozedur `set-cdr!` |
| MLIST | 1 | A | Standardprozedur `list`, $\leq 255$ Argumente |
| LISTN | 0 | A | Standardprozedur `list`, $> 255$ Argumente |
| MAKEVECTOR | 0 | A | Standardprozedur `make-vector` |
| VECTORSET | 0 | A | Standardprozedur `vector-set!` |
| VECTORREF | 0 | A | Standardprozedur `vector-ref` |
| VECTORLENGTH | 0 | A | Standardprozedur `vector-length` |
| MAKECODEVECTOR | 0 | A | Prozedur `make-codevector` |
| CODEVECTORSET | 0 | A | Prozedur `codevector-set!` |
| MVECTOR | 1 | A | Standardprozedur `vector`, $\leq 255$ Argumente |
| MVECTORN | 0 | A | Standardprozedur `vector`, $> 255$ Argumente |

## A.7 Numerische Befehle

| Numerische Befehle | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| DEC | 0 | A | Standardprozedur `-1+` |
| INC | 0 | A | Standardprozedur `1+` |
| NEG | 0 | A | Standardprozedur `-` für ein Argument |
| ADD | 0 | A | Standardprozedur `+` für 2 Argumente |
| SUB | 0 | A | Standardprozedur `-` für 2 Argumente |
| MUL | 0 | A | Standardprozedur `*` für 2 Argumente |
| DIVOP | 0 | A | Standardprozedur `quotient` für 2 Argumente |
| REM | 0 | A | Standardprozedur `remainder` für 2 Argumente |
| MINUSP | 0 | A | Standardprozedur `minus?` |
| ZEROP | 0 | A | Standardprozedur `zero?` |

## A.8 Zeichen- und zeichenkettenorientierte Befehle

| Zeichen- und zeichenkettenorientierte Befehle | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| CHARTOINT | 0 | A | Standardprozedur `char`→`integer` |
| INTTOCHAR | 0 | A | Standardprozedur `integer`→`char` |
| MAKESTRING | 0 | A | Standardprozedur `make-string` |
| STRINGLENGTH | 0 | A | Standardprozedur `string-length` |
| STRINGREF | 0 | A | Standardprozedur `string-ref` |
| STRINGSET | 0 | A | Standardprozedur `string-set!` |

## A.9 Sonstige Befehle

| Sonstige Befehle | | | |
|---|---|---|---|
| Bytecodebefehl | Argumente | Format | Bemerkung |
| `DUPLICATE` | 0 | A | Duplizieren des obersten Stackobjekts |
| `DROP` | 0 | A | Entfernen des obersten Stackobjekts |
| `STOP` | 0 | A | Anhalten des Scheme-Prozessors |

# Anhang B

# Gehäuse und Anschlussbelegung des Scheme-Prozessors

Zwanzig gefertigte Prototypen des Scheme-Prozessors wurden im November 1989 geliefert. Sie sind in 88poligen Keramikgehäusen des Typs PGA-88 montiert. Das Gehäuse bzw. die Anschlußbelegung der Prozessoren sind in Abbildung B.1 bzw. in Tabelle B.1 angegeben.

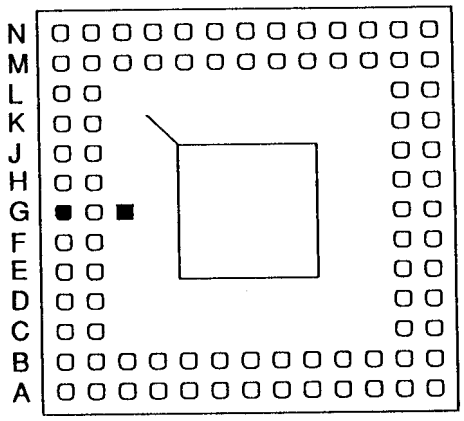



Abbildung B.1: Das Gehäuse PGA-88 der Scheme-Prozessors.

Die im Vergleich mit industriellen Prozessoren relativ hohe Zahl von Masse- und Betriebsspannungsanschlüssen ist nötig, weil bei Hauptspeicherschreibzugriffen dieses Prozessors die überwiegende Zahl aller Zellen gleichzeitig als Ausgang treiben. Die Entwurfsregeln verlangen dann pro acht Treiberzellen einen Masseanschluß [Siem88]. Bei industriellen Prozessoren können Versorgungsspannungsbahnen auf dem Chip für solche Fälle besonders breit ausgelegt werden. In VENUS-S sind solche Eingriffe durch den Benutzer nicht vorgesehen.

| Pin | Signal | Pin | Signal | Pin | Signal | Pin | Signal |
|---|---|---|---|---|---|---|---|
| G1 | A6 | A7 | LTEST | G13 | A19 | N7 | D7 |
| F2 | A7 | B8 | SCANCLK | H12 | A18 | M6 | D6 |
| F1 | A8 | A8 | — | H13 | A17 | N6 | Masse |
| E1 | A9 | A9 | Masse | J13 | A16 | N5 | CLKWRITE |
| E2 | A10 | B9 | FC0 | J12 | A15 | M5 | +5 V |
| D1 | A11 | A10 | FC1 | K13 | A14 | N4 | D5 |
| D2 | BGACK | B10 | FC2 | K12 | A13 | M4 | D4 |
| C1 | LDS | A11 | SCANOUT | L13 | A12 | N3 | D3 |
| B1 | Masse | A12 | RESET | M13 | Masse | N2 | D2 |
| C2 | +5 V | B11 | SCANIN | L12 | — | M3 | D1 |
| A1 | — | A13 | — | N13 | — | N1 | D0 |
| B2 | — | B12 | — | M12 | D15 | M2 | DTACK |
| B3 | UDS | C12 | Masse | M11 | D14 | L2 | AS |
| A2 | — | B13 | STKCLK | N12 | D13 | M1 | R/W |
| B4 | CS | D12 | UPCCLK | M10 | D12 | K2 | Masse |
| A3 | BGIN | C13 | A23 | N11 | D11 | L1 | BUSCLK |
| A4 | BR | D13 | A22 | N10 | D10 | K1 | +5 V |
| B5 | IRQ | E12 | +5 V | M9 | Masse | J2 | A1 |
| A5 | WAIT2 | E13 | +5 V | N9 | CLKREAD | J1 | A2 |
| B6 | +5 V | F12 | A21 | M8 | +5 V | H2 | A3 |
| A6 | ROMCLK | F13 | A20 | N8 | D9 | H1 | A4 |
| B7 | — | G12 | — | M7 | D8 | G2 | A5 |

Tabelle B.1: Die Anschlußbelegung der Scheme-Prozessors.

# Anhang C

# Externe Beschaltung des Scheme-Prozessors

Der Scheme-Prozessor ist für den Einsatz in 68000-Rechnern entworfen. Er generiert alle benötigten Signale und wird ohne zusätzliche Logik direkt an den 68000-Bus angeschlossen. Der Scheme-Prozessor enthält Treiberzellen, die 5.5 mA Dauerstrom aufnehmen (bei logisch 0) und 3 mA Dauerstrom liefern (bei logisch 1). Kurzzeitig dürfen beide Ströme bis zu 9 mA betragen. Diese Treiberleistungen erlauben es, den Scheme-Prozessor direkt in vorhandene 68000-Rechner einzusetzen. Externe Treiber sind daher nicht nötig.

## C.1 Der Taktgenerator

Der Scheme-Prozessor wird mit mehreren Takten versorgt: Im Operationswerk erhält der autonom arbeitende Buszugriffsmodul einen Takt, der nur zum Takt des 68000-Rechners synchron sein muß. Alle anderen Takte für das Operations- und das Steuerwerk müssen in einem festen zeitlichen Verhältnis zueinander stehen. Das Operationswerk erhält die Takte `CLKREAD` und `CLKWRITE`, die als `ReadClock` und `WriteClock` in Abbildung 8.19 auf Seite 106 dargestellt wurden. Das Steuerwerk (Abbildung 8.10 auf Seite 90) erhält vier Takte: `ROMCLK` steuert das ROM, dessen Daten nach der negativen Flanke ausgelesen werden können. `UPCCLK` und `STKCLK` übernehmen mit der steigenden Flanke in den Mikroprogrammzähler bzw. in den Mikroprogrammstack. Mit der steigenden Flanke von `SCANCLK` werden Daten in die Scanpath- und Pipelineregister übernommen.

Diese Takte sind aus zwei Gründen einzeln extern anzulegen: Einerseits müssen sie zum Test der Prozessoren auf Fertigungsfehler separat steuerbar sein. So werden im Betrieb zum Beispiel `UPCCLK` und `STKCLK` immer miteinander verbunden. Während des Testens darf aber beim Einschieben in den Scanpath, wobei `UPCCLK` verwendet wird, nicht der Mikroprogrammstack verändert werden, da er sonst nicht mehr testbar wäre. Andererseits können durch die einzeln zugänglichen Takte Verzögerungszeiten auf den gefertigten Chips gemessen werden, und es können Taktfrequenzen für die tatsächlichen Timingbedingungen und nicht für den ganzen Bereich der vom Hersteller garantierten Bedingungen ausgelegt werden. Im hier verwendeten Prozeß ACMOS3 von Siemens streuen die Verzögerungszeiten nach der Fertigung im Bereich der Faktoren 0,6 bis 1,6 um den „typischen“ Wert, der im Simulationsmodell von VENUS-S eingetragen ist [Siem88].

Der Betrieb des Scheme-Prozessors erfolgt im Atari ST mit den in Abbildung C.1 angegebenen Takten. Der Takt `ROMCLK` konnte mit `UPCCLK` und `STKCLK` zusammengefaßt werden, weil beide Takte mit unterschiedlichen Flanken übernehmen. In der Abbildung wird ein Takt von 8 MHz

erzeugt, weil er sich besonders einfach aus den im Atari vorhanden 32 MHz ableiten läßt. Zur Erzeugung der Takte aus dem 32 MHz-Takt dient im Atari ST die in Abbildung C.2 angegebene Schaltung.

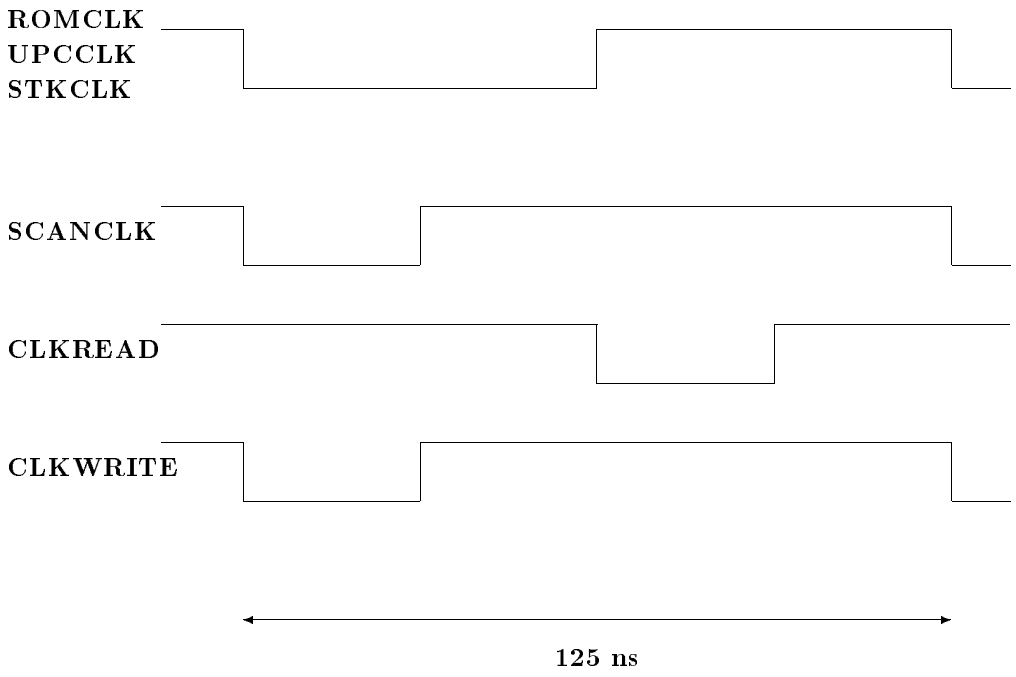



Abbildung C.1: Die Takte des Scheme-Prozessors.

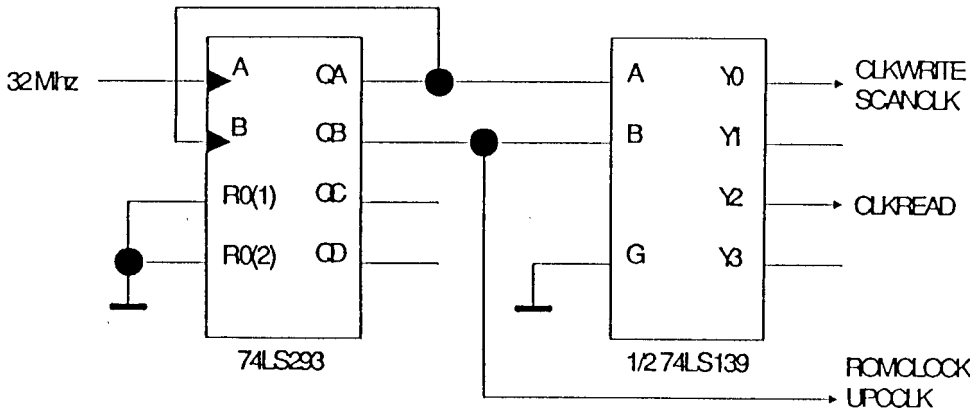



Abbildung C.2: Der Taktgenerator für den Scheme-Prozessor im Atari ST.

## C.2 Die Chipselectlogik

Eine externe Beschaltung der Prototypen wird benötigt, um das Bausteinauswahlsignal (Chipselect CS) zu generieren, das den Registern des Scheme-Prozessors Adressen für Zugriffe durch den 68000-Rechner zuordnet. Diese Adressen sind vom verwendeten Rechner abhängig; die Chipselectlogik ist daher immer rechnerspezifisch. Im Atari ST werden die Register des Scheme-Prozessors an die Adresse $F00000 gelegt, da diese noch unbelegt ist und mit geringem Beuteileaufwand dekodiert werden kann. Abbildung C.3 zeigt den Chipselektdekoder für den Atari ST, der die Adressen $F00000 bis $F7FFFF dekodiert, welche alle im Atari ST nicht verwendet werden.

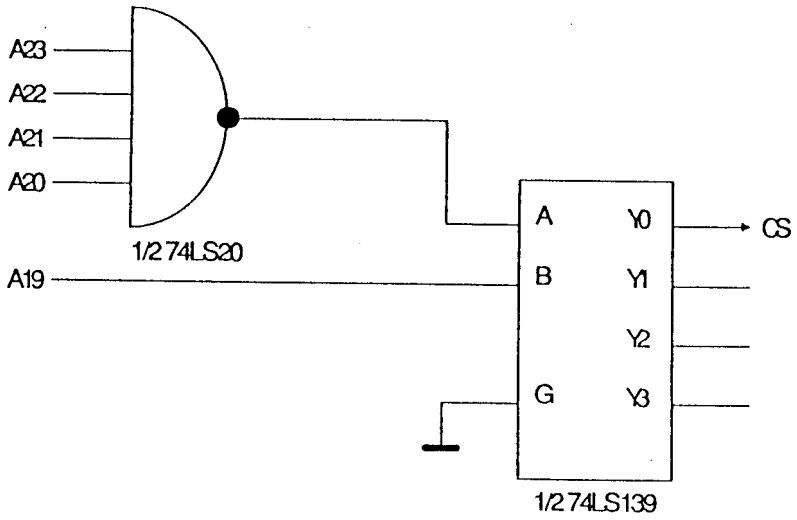



Abbildung C.3: Die Chipselectlogik für den Scheme-Prozessor im Atari ST.

Die gezeigte Hälfte des Dekoderbausteins 74LS139 wird in der Chipselectlogik als Oder-Gatter eingesetzt, da dieser Baustein im Taktgenerator nicht vollständig genutzt wird und auf diese Weise ein weiteres Standard-IC eingespart werden kann. Taktgenerator und Chipselectlogik bestehen zusammen aus nur drei Standard-ICs.

Die externe Logik wird derzeit in ein PAL verlagert, so daß außer dem Scheme-Prozessor nur noch ein zusätzlicher Baustein auf der Platine sein wird. Mit dem PAL werden dann auch genau die in Abbildung C.1 angegebenen Takte erzeugt; im derzeitigen diskreten Aufbau weichen die Takte durch aufgrund der unterschiedlichen Verzögerungszeiten von der Abbildung ab.

# Anhang D

# Test der gefertigten Prozessoren

Der Test der gefertigten Prozessoren erfolgt auf Fertigungsfehler. Prinzipiell werden die beim IC-Entwurf für den Fertigungstest entworfenen Testmuster durch einen Testmustergenerator an jedes der gefertigten Chips angelegt und die Ausgänge des Chips durch einen Analysator mit den erwarteten (simulierten) Reaktionen verglichen.

Beim Scheme-Prozessor bestand das Problem, daß die zur Verfügung stehenden Testgeräte nicht für die große Anzahl von Anschlüssen der Scheme-Prozessoren ausgelegt waren. Die Testgeräte sind nur für Chips mit maximal einem Viertel der Anschlüsse des Scheme-Prozessors geeignet.

Ein für den Fertigungstest bei Mikroprozessoren eingesetztes Verfahren besteht im Auslesen des Mikroprogramms in einem speziellen Testmodus. Wenn das Mikroprogramm korrekt ist, wird der Mikroprozessor durch das Ausführen von Befehlen getestet. Auf diese Weise wird auch der 68000-Prozessor getestet [DaBr85].

Der Test des Scheme-Prozessors erfolgt auch in zwei Schritten und berücksichtigt die Einschränkung der Testgeräte:

1. Testen des Steuerwerks mit den Testgeräten.
   Das Mikroprogramm-ROM wird ausgelesen, und die Folgeadreßlogik wird getestet. Da diese Operationen den Scanpath verwenden, werden nur wenige Prozessoranschlüsse benötigt.

2. Testen des Operationswerks durch Betreiben im Atari ST.
   Durch Diagnoseprogramme werden Registerbank und ALU getestet. Die Diagnoseprogramme umfassen sowohl Programme auf dem Atari ST (Lesen und Schreiben der Scheme-Prozessor-Register) als auch Programme in Bytecode, die der Scheme-Prozessor ausführt.

Ferner wird auch die Stromaufnahme der Prüflinge gemessen, um Hinweise auf mögliches fehlerhaftes elektrisches Verhalten (z.B. interne Kurzschlüsse) zu erhalten.

| Chip | Operations-werk | Steuer-werk | Strom | Erklärung |
|---|---|---|---|---|
| 1 | | | | Bonddrähte defekt |
| 2 | Fehler | o.k. | 10,0 mA | Bit #31 in Registerbank falsch |
| 3 | Fehler | o.k. | 10,1 mA | Registerbank defekt |
| 4 | | Fehler | 10,5 mA | 4886 Fehler im Mikroprogramm-ROM |
| 5 | o.k. | o.k. | 9,5 mA | |
| 6 | o.k. | o.k. | 9,4 mA | |
| 7 | | Fehler | 11,2 mA | 44 Fehler im Mikroprogramm-ROM |
| 8 | | Fehler | 7,1 mA | 10 Fehler im Mikroprogramm-ROM |
| 9 | | Fehler | 6.5 mA | 1 Fehler im Mikroprogramm-ROM |
| 10 | o.k. | o.k. | 10,5 mA | |
| 11 | o.k. | o.k. | 7,5 mA | |
| 12 | o.k. | o.k. | 8,0 mA | |
| 13 | o.k. | o.k. | 11,5 mA | |
| 14 | Fehler | o.k. | 7,5 mA | Register `cur_byte` und `tos` defekt |
| 15 | o.k. | o.k. | 11,0 mA | |
| 16 | o.k. | o.k. | 8,1 mA | |
| 17 | Fehler | o.k. | 7,0 mA | Registerbank defekt |
| 18 | o.k. | o.k. | 8,5 mA | |
| 19 | o.k. | o.k. | 11,1 mA | |
| 20 | o.k. | o.k. | 8,1 mA | |

Tabelle D.1: Ergebnisse des Tests der Prozessoren.

# Literaturverzeichnis


[TI] *2$\mu$m CMOS Standard Cell Data Book.* Texas Instruments Incorporated, 1986.

[68000] *MC68000 16-bit Microprocessor User's Manual.* Motorola, Inc., Second Edition, January 1980.

[AbSu85] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[AbSu88] Harold Abelson and Gerald Jay Sussman. Lisp: A Language for Stratified Design. *Byte Magazin*, McGraw Hill, New York, 207–218, February 1988.

[Abra86] Jacob A. Abraham. Fault Modeling in VLSI. In T. W. Williams, *VLSI Testing*, North-Holland, 1986.

[Aho89] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau.* Addison-Wesley Deutschland, 1989.

[Alle78] John Allen. *Anatomy of LISP.* McGraw Hill, New York, 1978.

[BaJe86] David H. Bartley and John C. Jensen. The Implementation of PC Scheme. *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 86–93, 1986.

[Bake78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[Barb81] Mario R. Barbacci. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers.* Vol C-30, No 1, January 1981.

[Bata82] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 Architecture - System and Chip. *Proceedings, Conference on Advanced Research in VLSI*, 69–77, 1982.

[BeWu88a] Andrew A. Berlin and Henry M. Wu. Scheme86: A System for Interpreting Scheme. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 116–123, 1988.

[BeWu88b] Andrew A. Berlin and Henry M. Wu. *Scheme86: A System for Interpreting Scheme.* AI Memo No. 1040, M.I.T., Cambridge, Mass., April 1988.

[Betz89] David M. Betz. *XScheme: An Object-oriented Scheme.* Version 0.16 Manual, January 1989.

[Boss87] Patrick W. Bosshart, C. Robert Hewes, Michael D. Ales, Mi-Chang Chang, Kwog Kit Chau, Kingsley Fasham, Charles C. Hoac, Theodore W. Houston, Vibhu Kalyan, Stephen L. Lusky, Shivaling S. Mahant-Shetti, Douglas J. Matzke, Kamalesh N. Ruparel, Joe F. Sexton, Ching-Hao Shaw, Thirumalai Sridhar, Donald Stark, and Ah Lyan Lee. A 553K-Transistor LISP Processor Chip. *IEEE Journal of Solid-State Circuits.* Vol. sc-22, No. 5, October 1987.

[BrGa84] Rodney A. Brooks and Richard P. Gabriel. A Critique of Common Lisp. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1–8, 1984.

[Broo84] Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 256–262, 1984.

[BuLa87] Rainer Buschke, Klaus Lagemann. SIMUEVA: A CAD Program Package for Efficient Analysis of Simulation Results. *Proceedings of COMPEURO 1987*, Hamburg, 451–454, 1987.

[CSch84] *MIT Scheme Manual, Seventh Edition.* Cambridge, Mass., September 1984.

[ChHe87] Paul Chow and John Hennessy. Reduced Instruction Set Computer Architectures. In Veljko M. Milutinović, *Computer Architecture*, North-Holland, New York, 1987.

[Chai84] Jérôme Chailloux, Matthieu Devin, and Jean-Marie Hullot. LeLisp, a Portable and Efficient LISP System. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 113–122, 1984.

[Chen70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[Chur41] Alonzo Church. *The Calculi of Cambda-Conversion.* Princeton University Press, Princeton, N. J., 1941.

[Clee79] M. W. van Cleemput. Hierarchical Design for VLSI — Problems and Advantages. *Proceedings of the CALTECH Conference on VLSI*, 259–274, January 1979.

[Clin84] William Clinger. The Scheme 311 compiler: An Exercise in Denotational Semantics. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 356–364, 1984.

[Clin85] William Clinger, editor. *The revised revised report on Scheme, or an uncommon Lisp.* MIT Artificial Intelligence Memo 848, August 1985.

[Clin88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation Strategies for Continuations. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 124–131, 1988.

[DaBr85] R. Gary Daniels and William C. Bruce. Built-In Self-Test Trends in Motorola Microprocessors. *IEEE Design & Test*, 64–71, April 1985.

[Davi78] Scott Davidson and Bruce D. Shriver. An Overview of Firmware Engineering. *IEEE Computer*, 21–33, Vol. 11, No. 5, May 1978.

[Davi83] Scott Davidson. High Level Mikroprogramming — Current Usage, Future Prospects. *ACM SIGMICRO*, 193–200, Vol. 14, No. 4, December 1983.

[Deut80] L. Peter Deutsch. ByteLisp and its Alto Implementation. *Conference Record of the 1980 Lisp Conference*. 231–242, 1980.

[Dybv87] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[ES2a] *ES2 ROM Generator*. European Silicon Structures, 1989.

[ES2b] *ES2 Solo 2030 Family*. European Silicon Structures, 1989.

[Eich85] Werner Eicher. *Ein hardware-unterstützter LISP-Interpreter*. Dissertation, Universität Kaiserslautern, 1985.

[Eise88] Michael Eisenberg. *Programming In Scheme*. Scientific Press, Redwood City, CA, 1988.

[FeYo69] Robert R. Fenichel, and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[Feel86] Marc Feeley. *Deux Approches á L'implementation du Language Scheme*. M.Sc. Thesis, Département d'Informatique et de Recherche Opérationelle, University of Montreal, May 1986

[Fell83] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher T. Haynes. *Scheme 311 version 4 Reference Manual*. Bloomington, Indiana, February 1983.

[Frie85] Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker, and Mitchell Wand. *Scheme 84 Interim Reference Manual*. Bloomington, Indiana, January 1985.

[Gabr85] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, Cambridge, Mass, 1985.

[Gabr87] Richard P. Gabriel. Memory Management in LISP. *AI Expert*, (2):32–38, 1987.

[GeKu83] D. D. Gajski, R. H. Kuhn. Guest Editor's introduction: New VLSI tools. *Computer*, 16(12):11–14, December 1983.

[Gilo83] Wolfgang K. Giloi and Bruce D. Shriver (Ed.). *Methodoligies for Computer System Design*. Proceedings of IFIP WG 10.1 Working Conference, Elsevier Science Publishers, Amsterdam, 1983.

[GrHe82] T. R. Gross, J. L. Hennesey. Optimizing Delayed Branches. *Proceedings of 15th Annual Workshop on Microprogramming*, ACM, October 1982, 114ff.

[Gree77] Richard Greenblatt. *LISP Machine Progress Report*. AI Memo No. 444, M.I.T., Cambridge, Mass., August 1977.

[HaFr84] Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 18–24, 1984.

[Hafe89] Christian Hafer. COLIBRI-Architektur. In Jürgen Ebert, *Alternative Konzepte für Sprachen und Rechner*, Bericht 6/89, EWH Koblenz.

[Hans69] Wilfred J. Hansen. Compact List Representation: Definition, Garbage Collection, and System Implementation. *Communications of the ACM*, 12(9):499–507, 1969.

[Hayn84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and Coroutines. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 293–298, 1984.

[Hayn86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines With Continuations. *Journal of Computer Languages*, 11(3/4):143–153, 1986.

[Hend80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall International, London, 1980.

[Hoff83] Rolf Hoffmann. *Rechenwerke und Mikroprogrammierung.* R. Oldenbourg, München, 1983.

[Holl80] Jack Holloway, Guy Lewis Steele, Jr., Gerald Jay Sussman, and Alan Bell. *The SCHEME-79 Chip.* AI Memo No. 559, M.I.T., Cambridge, Mass., 1980.

[Hörb86] Egon Hörbst, Martin Nett und Heinz Schwärtzel. *VENUS — Entwurf von VLSI-Schaltungen.* Springer Verlag, Berlin, 1986.

[JeWi78] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report.* Springer-Verlag, New York, 1978.

[KeRi78] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall International, Englewood Cliffs, 1978.

[Knut68] Donald E. Knuth. *The Art of Computer Programming Vol. 1 Fundamental Algorithms.* Addison-Wesly, Reading, Mass., 1968.

[Kong89] Shing I. Kong, David D. Lee, Randy H. Katz, David A. Hodges, and David A. Patterson. From Chip To System: The SPUR CPU Experience. *Proceedings of the VLSI'89 Conference*, IFIP, 55–64, Munich, August 16–18, 1989.

[Kran86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction.* 219–233. ACM June 1986.

[Kran88] David Andrew Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, May 1988.

[Lage87] Klaus Lagemann. *Rechnerstrukturen.* Springer-Verlag, Berlin Heidelberg, 1987.

[Laub76] Joachim Laubsch, Dieter Krause, Klaus Hess, Werner Schatz. *TR440 MacLisp - Handbuch.* Institut für Informatik der Universität Stuttgart, 1976.

[LiHe83] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LiSc82] P. Liell, E. Schaaf. *Die Hardwarebeschreibungssprache Karl II — Sprachbeschreibung.* 2. Auflage, Universität Kaiserslautern, April 1982.

[LoRa89a] Jörg Lohse, Reinhard Rauscher. A Scheme-Coprocessor for Cheap Microcomputers. *Proceedings Intern. AMSE Conference "Signals & Systems"*, AMSE Press, Vol. 3, p. 103–110, Brighton (U.K.), July 12–14, 1989.

[LoRa89b] Jörg Lohse, Reinhard Rauscher. A VLSI Coprocessor Implementing the Programming Language Scheme. *Proceedings of the VLSI'89 Conference*, IFIP, 275–284, Munich, August 16–18, 1989.

[Lohs85] Jörg Lohse. *ST Pascal.* CCD, Eltville, 1985.

[Lohs86] Jörg Lohse. *Personal Pascal.* Optimized System Software, San Jose, CA, 1986.

[Lohs87] Jörg Lohse. *Ein Plotprogramm für SIGRED-Schaltungen.* Mitteilung 2/87 des Arbeitsbereich TECH, September 1987. Erhältlich von: Gesellschaft für Mathematik und Datentechnik, Projekt E.I.S., Birlinghoven.

[Lohs88] Jörg Lohse. *Architektur, Befehlssatz und Entwurfswerkzeuge des ersten Hamburger Scheme-Prozessors.* Mitteilung des Fachbereichs Informatik FBI-HH-M-162/88, Universität Hamburg, September 1988.

[Lohs89] Jörg Lohse, Reinhard Rauscher, Bernd Schütz. A Coprocessor for Implementing the LISP-Dialect Scheme. *Proceedings of the EUROMICRO'89 Conference*, 355–363, Cologne, September 1989.

[MaDe80] Larry M. Masinter and L. Peter Deutsch. Local Optimization in a Compiler for Stack-based Lisp Machines. *Conference Record of the 1980 Lisp Conference.* 223–230, 1980.

[MaWi89] Dieter Maurer, Reinhard Wilhelm. MaMa — eine Abstrakte Maschine zur Implementierung funktionaler Programmiersprachen. *Informatik Forschung und Entwicklung.* Springer Verlag, Heidelberg, 67–88, April 1989.

[Marw86] Peter Marwedel. *Ein Software-System zur Sythese von Rechnerstrukturen und zur Erzeugung von Mikrocode.* Habilitation, Universität Kiel, 1986.

[Matt87] Gene Matthews, Robert Hewes, and Steve Krueger. Single-chip processor runs Lisp environments. *Computer Design.* 69–76, May 1987.

[McCa60] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 184–195, April 1960.

[McDe80] Drew McDermott. An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-scoped LISP. *Conference Record of the 1980 Lisp Conference.* 154–162, 1980.

[MeCo80] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Co., Reading, Mass, 1980.

[Milu87] Veljko M. Milutinović (Ed.). *Computer Architecture.* North-Holland, New York, 1987.

[Moon84] David A. Moon. Garbage Collection in Large Lisp Systems. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 235–246, 1984.

[Moon85] David A. Moon. Architecture of the Symbolics 3600. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 76–83, 1985.

[Much80] Steven S. Muchnick and Uwe F. Pleban. A Semantic Comparison of Lisp and Scheme. *Conference Record of the 1980 Lisp Conference*, 56–65, 1980.

[Muel84] Robert A. Mueller. *Automated Microcode Synthesis.* UMI Research Press, Ann Arbor, 1984.

[Myer87] Glenford J. Myers. *Advances in Computer Architecture.* John Wiley & Sons, New York, 1987.

[Naur63] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[Nowa86] Lothar Nowak. *SAMP: Entwurf und Realisierung eines neuartigen Rechnerkonzeptes.* Dissertation und Bericht Nr. 8615, Universität Kiel, November 1986.

[Padg86] Julian Padget, Jérôme Chailloux, Thomas Christaller, Ramon DeMantaras, Jeff Dulton, John Fitch, Timm Krumnack, Eugen Neidl, Eric Papon, Stephen Pope, Christian Queinnec, Luc Steels, and Herbert Stoyan. Desiderata for the standardisation of LISP. *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 54–66, 1986.

[Patt85] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8–21, 1985.

[Paye89] Michael Payer. *Graphentheoretische Ansätze für CAD-Werkzeuge in der CMOS-Technik.* Dissertation, Universität Hamburg, Juli 1989.

[Pend86] Joan M. Pendleton, Shing I. Kong, Emil W. Brown, Frank Dunlap, Christopher Marino, David M. Ungar, David A. Patterson, and David A. Hodges. A 32-bit Microprocessor for Smalltalk. *IEEE Journal of Solid State Circuits.* Vol. 21, No 5, October 1986.

[Peyt87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, 1987.

[PlTh87] Andrew R. Pleszkun, Matthew J. Thazhuthaveetil. The Architecture of Lisp Machines. *IEEE Computer*, Vol. 20, No. 3, 35–44, IEEE, March 1987.

[PrWi86] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design.* Prentice-Hall International, Englewood Cliffs, 1986.

[Prat75] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall International, Englewood Cliffs, 1975.

[Raus87] Reinhard Rauscher. *Ein praktikables Verfahren zur interaktiven Mikroprogrammtransformation und deren formaler automatischer Verifikation.* Dissertation, Universität Hamburg, 1987.

[Rees84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T manual, fourth edition.* Yale University, January 1984.

[Rees86] Jonathan A. Rees and William Clinger (editors). Revised$^3$ Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[Rowa80] William Rowan. A Lisp Compiler producing Compact Code. *Conference Record of the 1980 Lisp Conference.* 216–222, 1980.

[Roza84] Guillermo J. Rozas. *Liar, an Algol-like Compiler for Scheme.* S. B. Thesis, MIT Department of Electrical Engineering and Computer Science, January 1984.

[SaJa84] Emanuell Saint-James. Recursion is More Efficient than Iteration. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* 228–234, 1984.

[ScSt84] Richard Schooler and James W. Stamos. *Proposal for a Small Scheme Implementation.* MIT/LCS/TM-267, M.I.T., Cambridge, Mass., 1984.

[ScWa67] H. Schorr and W. M. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10(8):501–506, 1967.

[Sem87] *MacScheme+Toolsmith User's Guide.* Version 1.5, Semantic Microsystems, Beaverton, Oregon, 1987.

[Shep87] Susan J. Shepard. LISP on a Micro. *AI Expert*, (2):73–76, 1987.

[SiMi87] Alexander A. Silbey and Veljko M. Milutinović. Advanced Microprocessors and High-Level Language Processor Architectures. In Veljko M. Milutinović, *Computer Architecture*, North-Holland, New York, 1987.

[Siem88] *Zellenkatalog Standardzellen ACMOS3.* Stand VENUS-S3.3E2. Siemens AG, Abteilung ZTI PPM 13, München, Juli 1988.

[Smit88] Jerry Smith. *Introduction to Scheme.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[StHe88] Peter Steenkiste and John Hennessy. Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization. *IEEE Computer*, Vol. 21, No. 7, 34–45, IEEE, July 1988.

[StSu76] Guy Lewis Steele, Jr. and Gerald Jay Sussman. *Lambda, the Ultimate Imperative.* AI Memo No. 353, M.I.T., Cambridge, Mass., March 1976.

[StSu78] Guy Lewis Steele, Jr. and Gerald Jay Sussman. *The revised report on Scheme, a dialect of Lisp.* MIT Artificial Intelligence Memo 452, January 1978.

[StSu80a] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Design of a Lisp-based Processor. *Communications of the ACM*, 23(11):628–645, November 1980.

[StSu80b] Guy Lewis Steele, Jr. and Gerald Jay Sussman. The Dream of a Lifetime: a Lazy Variable Extent Mechanism. *Conference Record of the 1980 Lisp Conference*, 163–172, 1980.

[Stee76] Guy Lewis Steele, Jr. *Lambda, the Ultimate Declarative.* AI Memo 379, M.I.T., Cambridge, Mass., November 1976.

[Stee77] Guy Lewis Steele, Jr. Macaroni is Better than Spaghetti. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages.* 60–66, August 1977

[Stee78] Guy Lewis Steele, Jr. *Rabbit: a Compiler for Scheme.* Technical Report AI-TR-474, M.I.T., Cambridge, Mass., May 1978.

[Stee80] Guy Lewis Steele, Jr. *Destructive Reordering of CDR-Coded Lists.* AI Memo 587, M.I.T., Cambridge, Mass., August 1980.

[Stee84] Guy Lewis Steele, Jr. *Common Lisp the Language.* Digital Press, 1984.

[Stei88] Uwe Steinhausen. *VENUS Compendium.* Gesellschaft für Mathematik und Datentechnik, Projekt E.I.S., Birlinghoven, November 1988.

[Stum89] Hans Stumper. *Beobachtungen bei DRC und ERC des Hamburger Scheme-Chips.* Persönliche Mitteilung. Gesellschaft für Mathematik und Datentechnik, Projekt E.I.S., Birlinghoven, Juni 1989.

[SuSt75] Gerald Jay Sussman and Guy Lewis Steele, Jr. *Scheme: an Interpreter for Extended Lambda Calculus.* AI Memo No. 349, M.I.T., Cambridge, Mass., December 1975.

[Suss81] Gerald Jay Sussman, Jack Holloway, Guy Lewis Steele, Jr., and Alan Bell. Scheme-79 - Lisp on a Chip. *IEEE Computer*, 14(7):10–21, July 1981.

[Suss88] Gerald Jay Sussman. *Scheme-81.* Persönliche Mitteilung, E-mail, 11. Oktober 1988.

[Teit74] W. Teitelman. *Interlisp Reference Manual.* Xerox Palo Alto Research Center, 1974.

[ThSc89] Nguyen Huu Chau Thuy, Peter Schnupp. *Wissensverarbeitung und Expertensysteme.* R. Oldenbourg Verlag, München Wien, 1989.

[Ullm84] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[WG89] IEEE/MSC/P1178 Working Group on Scheme. *Unapproved Minutes of the Third Meeting.* MIT, Cambridge, MA, 7 July 1989.

[WaTh85] R. A. Walker, D. E. Thomas. A model of design representation and synthesis. *Proceedings of the 22nd Design Automation Conference*, 453–459, June 1985.

[Wads71] C. P. Wadsworth. *Semantics and Pracmatics of the lambda calculus.* PhD thesis, Oxford, 1971.

[West89] Neil West. The Ivory Lisp Processor Project — Some Experiences with Big VLSI. *Proceedings of the VLSI'89 Conference*, IFIP, 67–78, Munich, August 16–18, 1989.

[WhFa84] Skef Wholey and Scott E. Fahlman. The Design of an Instruction Set for Common Lisp. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 150–158, 1984.

[Wilk51] Maurice V. Wilkes. The Best Way to Design an Automatic Machine. *Proceedings Manchester University Computer Inaugural Conference.* Ferrante, London, July 1951.

[Will86] Thomas W. Williams (Ed.). *VLSI Testing.* North-Holland, 1986.

[Wirt81] Niklaus Wirth. *Compilerbau.* 2. Auflage, Teubner, Stuttgart, 1981.

[Wu88] Henry M. Wu. *Scheme86: An Architecture for Microcoding a Scheme Interpreter.* AI Memo No. 953, M.I.T., Cambridge, Mass., August 1988.

[Wulf75] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

# Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich diese Dissertation selbst verfaßt habe und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 27. November 1990

(Jörg Lohse)

# Lebenslauf

| | |
|---|---|
| 7. 3. 1960 | geboren in Hamburg als erstes Kind des Poliers Klaus Lohse und der Buchhalterin Gerda Lohse, geb. Jacobsen |
| 1967 bis 1971 | Besuch der Hauptschule Franzosenkoppel in Hamburg |
| 1971 bis 1980 | Besuch des naturwissenschaftlichen und neusprachlichen Gymnasiums Rispenweg in Hamburg |
| 1980 | Abitur am Gymnasium Rispenweg |
| 1980 | Beginn eines Studiums der Physik an der Universität Hamburg |
| 1982 | Aufnahme eines Studiums der Informatik an der Universität Hamburg |
| 1984 | Vordiplom in Informatik mit Nebenfach Physik an der Universität Hamburg |
| 1986 | Diplom in Informatik mit Nebenfach Physik an der Universität Hamburg |
| seit 1986 | tätig als wissenschaftlicher Mitarbeiter am Fachbereich Informatik der Universität Hamburg |