Hochschultext

Hans Liebig

Rechnerorganisation

Hardware und Software digitaler Rechner

Unter Mitarbeit von T. Flik und K. Horn

Springer-Verlag
Berlin Heidelberg New York 1976

Dr.-Ing. HANS LIEBIG
Professor der Informatik
an der Technischen Universität Berlin
Institut für Technische Informatik

Dr.-Ing. THOMAS FLIK
Institut für Technische Informatik
der Technischen Universität Berlin

Dipl.-Ing. KLAUS HORN
Institut für Technische Informatik
der Technischen Universität Berlin

Mit 102 Abbildungen

ISBN-13: 978-3-540-07596-7 e-ISBN-13: 978-3-642-96311-7
DOI: 10.1007/978-3-642-96311-7

Library of Congress Cataloging in Publication Data
Liebig, Hans. Rechnerorganisation. (Hochschultext) Includes index. 1. Electronic digital computers.
I. Flik, T., 1943- joint author. II. Horn, Klaus, 1944- joint author. III. Title. QA76.5.L513 001.6'4'044 76-12556

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdruckes, der Entnahme von Abbildungen, der Funksendung, der Wiedergabe auf photomechanischem oder ähnlichem Wege und der Speicherung in Datenverarbeitungsanlagen bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten.

Bei Vervielfältigungen für gewerbliche Zwecke ist gemäß § 54 UrhG eine Vergütung an den Verlag zu zahlen, deren Höhe mit dem Verlag zu vereinbaren ist.

© by Springer-Verlag, Berlin/Heidelberg 1976.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buche berechtigt auch ohne besondere Kennzeichnung nicht zur Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Offsetdruck: fotokop wilhelm weihert kg, Darmstadt · Einband: Konrad Triltsch, Würzburg.

Vorwort

Bücher über Rechnerorganisation, Rechnerstrukturen oder Rechnerarchitektur behandeln üblicherweise die Hardware und die Software digitaler Rechner als getrennte Gebiete. Wenig oder gar nicht findet die Tatsache Berücksichtigung, daß die Funktion von Digitalrechnern erst durch das Zusammenwirken von Hardware und Software zustandekommt und daß beim Entwurf eines Rechners für jede Funktionseinheit zwischen den Möglichkeiten der Hardware- oder der Software-Realisierung gewählt werden kann. Das vorliegende Buch soll hier eine Lücke schließen, indem diese gegenüberstellende Betrachtungsweise konsequent durchgeführt und durch viele Beispiele untermauert wird.

Bei der Auswahl des Stoffes wurde der Schwerpunkt auf Organisationsformen von Rechenanlagen gelegt, die sich in der Praxis bewährt haben. Im Vordergrund steht nicht die Darstellung einer Vielzahl technisch verschiedener Ausführungen, sondern die Behandlung von Prinzipien der Wirkungsweise und des Entwurfs von Digitalrechnern. Das Buch ist somit für Studenten, Programmierer und Ingenieure zum Erlernen von Techniken des Rechnerentwurfs und der Assemblerprogrammierung bestimmt, ebenso wie für Rechnerbenutzer als Ergänzung zu Rechner- und Assembler-Handbüchern der Hersteller. Auf Angaben konkreter Werte von Leistungsmerkmalen heutiger Rechner wurde wegen des enormen Tempos der technischen Entwicklung weitgehend verzichtet. Diese insbesondere für Systemplaner wichtigen Fragen der Bewertung von Rechenanlagen aufgrund betrieblicher Erfordernisse und technologischer Möglichkeiten sind Stoff des 1975 im gleichen Verlag erschienenen Buches "Architektur digitaler Rechner" von E. Jessen.

Das vorliegende Buch ist zum Gebrauch neben Vorlesungen bestimmt und auch für das Selbststudium geeignet, wenn man die Mühe nicht scheut, die vielen aus der Praxis des Rechnerentwurfs und der Assemblerprogrammierung entnommenen Beispiele mit Bleistift und Papier durchzuarbeiten. Das Buch setzt Kenntnisse im Programmieren in einer höheren Programmiersprache und im Entwurf digitaler Systeme voraus; für beide Gebiete existiert eine Reihe guter Lehrbücher.

Die vorliegende Darstellung orientiert sich an der aus Prozessor, Speicher und Ein-Ausgabekanälen bestehenden Struktur heutiger Rechenanlagen. In Kapitel 1 werden Grundlagen der klassischen Rechnerorganisation behandelt: die maschinelle Durchführung von Berechnungen durch die Rechnerhardware und die maschinelle Vorbereitung von Programmen mit der Rechnersoftware. Die Rechnerhardware und die Rechnersoftware werden in den Kapiteln 2 und 3 wieder aufgegriffen. Während die Behandlung der Hardware in Kapitel 2 immer tiefer ins Detail vordringt bis hin zur Beschreibung eines mikroprogrammierbaren Mikroprozessors, entfernt sich die Behandlung der Software in Kapitel 3 immer mehr vom Detail und endet in einem Überblick über einige wichtige Betriebssystemfunktionen. In Kapitel 4 werden die Prinzipien der Hardware- und Softwareorganisation zusammengefaßt und einander gegenübergestellt. Assemblerprogrammierung und Prozessororganisation sind zwei Gebiete, die derart verzahnt zusammenwirken, daß uns eine einheitliche Behandlung in ein und demselben Kapitel zweckmäßig erschien. Die Modifizierung von Adressen, die Gegenüberstellung von Makrobefehlen und Prozeduren, die Wirkungsweise geschachtelter und rekursiver Unterprogramme sind wichtige Themen dieses Kapitels. In den Kapiteln 5 und 6 werden die Speicherorganisation bzw. die Ein-Ausgabeorganisation behandelt, wobei Hardware- und Software-Realisierungen sich teils abwechseln und teils einander gegenübergestellt sind. In diesen Kapiteln werden insbesondere modularer Speicheraufbau, virtuelle Speicher, modulare Rechnerkonfigurationen, Programmunterbrechungen und Ein-Ausgabekanäle behandelt. In beiden Kapiteln werden auch Realisierungen vorgestellt, die sich nicht an traditionelle Organisationsformen halten: zum Beispiel in Kapitel 5 die Realisierung der Speicherverwaltung durch Hardware und in Kapitel 6 die Realisierung der Ein- und Ausgabesteuerung durch Software.

Herrn Professor P. Rechenberg, Linz, sei für die Durchsicht von Teilen des Manuskriptes, für viele anregende Diskussionen und eine Reihe von Verbesserungsvorschlägen gedankt. Unser Dank gilt auch dem Springer-Verlag für die gute Ausstattung des Buches.

Berlin, im April 1976

H. Liebig
T. Flik
K. Horn

Inhaltsverzeichnis

1. Klassische Rechnerorganisation 1

1.1 Maschinelle Durchführung von Berechnungen 4

Informationsverarbeitung mit einem Rechenhilfsmittel (Vier-Spezies-Rechner) 4

Informationsverarbeitung mit einem Rechenautomaten (Von-Neumann-Rechner) 8

1.2 Maschinelle Vorbereitung von Programmen 12

Codierung von Programmen durch den Menschen (Codierer) 12

Codierung von Programmen durch die Maschine (Assembler) 16

1.3 Mikroalgorithmen für die vier Grundrechenarten 19

2-Komplement-Zahlen 20

Zählen und Komplementieren 23

Addition und Subtraktion 24

Multiplikation 25

Division 29

2. Rechnerhardware 37

2.1 Ein einfacher Rechner 40

Ein Einadreßrechner 40

Funktionsweise des Rechners 46

2.2 Schaltnetze und Speicher 51

Boolesche Funktionen in Matrixdarstellung 55

Codierer und Decodierer 59

Multiplexer, Demultiplexer und Assoziierer 61

Festspeicher, Randomspeicher und Assoziativspeicher 65

2.3 Schaltwerke und Prozessoren 72

Boolesche Algorithmen in Matrixdarstellung 76

Boolesche Mikroprogrammwerke 78

Sequentielle Mikroprogrammwerke 83

Mikroprogrammierte Rechner 88

3. Rechnersoftware 94
3.1 Ein einfacher Assembler 97
Eine Assemblersprache 97
Funktionsweise des Assemblers 103
3.2 Lader und Binder 108
Urlader, Absolutlader und verschiebender Lader 108
Binder und bindender Lader 112
3.3 Systemprogramme und Betriebssysteme 116
Manueller Betrieb 118
Stapelbetrieb 119
Teilnehmerbetrieb 122

4. Assemblerprogrammierung und Prozessororganisation 126
4.1 Verallgemeinerung des Von-Neumann-Rechners und Erweiterung des klassischen Assemblers 127
Ein Zweiadreßrechner 130
Ein Makroassembler 134
4.2 Adreßmodifizierungen durch Software (Assembler) und Hardware (Prozessor) 137
Konstantenadressierung 138
Indexadressierung 140
Basisadressierung 141
Indirekte Adressierung 143
4.3 Offene und geschlossene Unterprogramme 146
Einfache Assembler-Unterprogramme (Makrobefehle) 147
Einfache Prozessor-Unterprogramme (Prozeduren) 150
Statische und dynamische Speicherverwaltung 155
4.4 Geschachtelte und rekursive Unterprogramme 157
Makroaufruf in einer Makrodefinition 158
Prozeduraufruf in einer Prozedurdefinition 161

5. Speicherorganisation 171
5.1 Speichermodelle und periphere Speicher 173
Speicher mit wahlfreiem Zugriff 173
Speicher mit sequentiellem Zugriff 177
Typische periphere Speicher 181
5.2 Modularer Speicheraufbau 184
Konstante und variable Blockdecodierung 185
Unverschränkte und verschränkte Speicher 186
Speicher für seriellen und parallelen Betrieb 189

5.3 Virtuelle Speicher 192
Pufferorganisation mit Assoziativspeichern und -prozessoren 194
Segment- und Seitenorganisation zur Pufferung des Primärspeichers . . 199
Segment- und Seitenorganisation zur Pufferung des Sekundärspeichers . 202

5.4 Dynamische Speicherverwaltung einzelner Programme durch Hardware 208
Speicherverwaltung für Felder variabler Länge 209
Speicherverwaltung bei Blöcken 213
Speicherverwaltung bei Prozeduren 217

6. Ein- und Ausgabeorganisation 225
6.1 Rechnerperipherie und Ein-Ausgabegeräte 227
Prinzipien des Anschlusses peripherer Geräte 228
Prinzipien der Steuerung peripherer Geräte 230
Ein- und Ausgabegeräte 233

6.2 Modulare Konfigurationen 238
Zusammenschaltungen von Speichern und Prozessoren 240
Zusammenschaltungen von Prozessoren und Ein-Ausgabegeräten 242

6.3 Programmunterbrechungen 244
Sensor- und Unterbrechungsleitungen 244
Ein-Ausgabeprogramme 250
Reentrante Prozeduren 254

6.4 Ein-Ausgabekanäle, Ein-Ausgabeprozessoren und Mehrprogrammbetrieb 257
Ein-Ausgabekanäle 257
Ein-Ausgabeprozessoren 266
Mehrprogrammbetrieb 269

Literaturverzeichnis 272

Sachverzeichnis 273

Zusammenhang der einzelnen Abschnitte

Die Punkte einer Kette zeigen, welche Abschnitte zum Verständnis des durch den letzten Punkt der Kette markierten Abschnitts notwendig sind.

1. Klassische Rechnerorganisation 1
1.1 Maschinelle Durchführung von Berechnungen 4
1.2 Maschinelle Vorbereitung von Programmen 12
1.3 Mikroalgorithmen für die vier Grundrechenarten 19

2. Rechnerhardware 37
2.1 Ein einfacher Rechner 40
2.2 Schaltnetze und Speicher 51
2.3 Schaltwerke und Prozessoren 72

3. Rechnersoftware 94
3.1 Ein einfacher Assembler 97
3.2 Lader und Binder 108
3.3 Systemprogramme und Betriebssysteme 116

4. Assemblerprogrammierung und Prozessororganisation 126
4.1 Verallgemeinerung des Von-Neumann-Rechners und Erweiterung des klassischen Assemblers 127
4.2 Adreßmodifizierungen durch Software (Assembler) und Hardware (Prozessor) 137
4.3 Offene und geschlossene Unterprogramme 146
4.4 Geschachtelte und rekursive Unterprogramme 157

5. Speicherorganisation 171
5.1 Speichermodelle und periphere Speicher 173
5.2 Modularer Speicheraufbau 184
5.3 Virtueller Speicher 192
5.4 Dynamische Speicherverwaltung einzelner Programme durch Hardware 208

6. Ein- und Ausgabeorganisation 225
6.1 Rechnerperipherie und Ein-Ausgabegeräte 227
6.2 Modulare Konfigurationen 238
6.3 Programmunterbrechungen 244
6.4 Ein-Ausgabkanäle, Ein-Ausgabeprozessoren und Mehrprogrammbetrieb 257

1. Klassische Rechnerorganisation

Wenn wir von klassischer Rechnerorganisation sprechen, meinen wir die Organisation digitaler Rechner nach einem Konzept, das die Grundlage für die meisten universell einsetzbaren Digitalrechner bildet.

In der Geschichte der Entwicklung digitaler Rechenautomaten findet sich wohl der erste Ansatz zu diesem Konzept in den Arbeiten von Charles Babbage, der die von Schickard 1623, Pascal 1645, Leibniz 1674, Poleni 1709, Falcon 1725, Vaucanson 1741, Jacquard 1804 und anderen entwickelten Ideen zum Bau mechanischer Rechenmaschinen 1833 zusammenfaßte und den "perfekten Rechenautomaten", die "Analytical Engine" entwarf. Babbage sah für diese Maschine ein Rechenwerk sehr hoher Stellenzahl mit parallelem Zehnerübertrag vor, außerdem eine Lochbandsteuerung, eine Lochbandeingabe, ein Speicherwerk für 1000 Zahlen zu 50 Stellen und ein Druckwerk für die Ausgabe. Der Bau dieses Automaten scheiterte schließlich an den technischen Unzugänglichkeiten dieser Zeit. Babbage selbst wurde zu seinen Lebzeiten als Narr angesehen und fand mit seinen Ideen entsprechend wenig Resonanz, obwohl die von ihm vorgesehenen Baugruppen wie Rechenwerk, Speicherwerk, Leitwerk und Ein-Ausgabewerk sich in vielen der heutigen Elektronenrechner wiederfinden.

Ein weiterer Fortschritt war die von Hollerith eingeführte Anlage, die aus einem Kartenlocher, einem handbedienten elektromagnetischen Zähler und einer Sortiereinrichtung bestand. Sie wurde 1890 zur Erleichterung der Volkszählung in den USA eingesetzt. Damit konnte die Bearbeitungszeit von bisher 7 1/2 Jahren auf 2 1/2 Jahre reduziert werden. Hollerith entwarf ein Lochkartenformat, das der 20-Dollar-Note entsprach, und einen eigenen Code. Kartenform und Code werden auch heute noch verwendet.

Erst mit der Entwicklung der Elektromechanik in der Telegrafietechnik wurden die Voraussetzungen für den Bau größerer programmgesteuerter Rechenmaschinen geschaffen. Einer der Pioniere dieser Zeit war Konrad Zuse, der 1941 den ersten elektromechanischen Rechner Z3 fertigstellte. Die Rechenzeiten für die Addition lagen bei 0,3 s und für die Multiplikation bei 5 s. Die interne Zahlendarstellung er-

folgte im Dualzahlensystem in sog. Gleitkommadarstellung. Dies erforderte eine Dezimal-Dual- bzw. Dual-Dezimal-Konvertierung für die Ein- und Ausgabe. Das Rechenprogramm wurde von einem gelochten 8-spurigen Kinofilm abgetastet.

Gleichzeitig mit den Entwicklungen von Zuse wurden auch in den USA die ersten Relais-Rechner gebaut. So z.B. der "Automatic Sequence Controlled Calculator" ASCC, auch MARK I genannt, der von Aiken an der Harvard University entworfen und 1943 in Betrieb genommen wurde. Er war zwar äußerst leistungsfähig, jedoch nur beschränkt programmierbar. Die Rechenzeiten lagen etwas über denen der Z3 von Zuse. Der MARK I wurde für ballistische Berechnungen eingesetzt. Aiken schätzte, daß sein Rechner ungefähr 100 mal schneller rechnen kann als ein Mensch mit einer Tischrechenmaschine. Der Nachfolger MARK II, der aus 13 000 Relais aufgebaut war, wurde 1947 fertiggestellt. Die Rechenzeiten lagen bei 0,125 s für die Addition und 0,25 s für die Multiplikation von 10-stelligen Gleitkomma-Dezimalzahlen.

Einen weiteren großen Sprung machte die Rechnerentwicklung mit der Verwendung der Elektronenröhre als Bauelement für die elementaren Schaltvorgänge. Mit ihr konnten die Schaltzeiten gegenüber der Relaistechnik stark verringert werden. Sie wurde zur Grundlage der sog. ersten Generation elektronischer Rechenanlagen. Die steuerbare Elektronenröhre wurde 1910 von Lieben erfunden und 1913 von Langmuir weiterentwickelt. Eccles und Jordan entwickelten bereits 1919 das bistabile Flipflop, das sich für binäre Rechenschaltungen und Zähler eignete. Auf dieser Basis bauten Eckert und Mauchly 1944 an der University of Pennsylvania den ersten elektronischen digitalen Universalrechner, den "Electronic Numerical Integrator and Computer" ENIAC, dessen erster Einsatz 1946 in Los Alamos erfolgte. Mit Flipflops wurden 10-stellige Ringzähler aufgebaut und im Sinne der zehnstelligen Ziffernräder zum dezimalen Zählen verwendet. Der ENIAC wurde für die Berechnung ballistischer Tabellen eingesetzt. Er wurde von den Entwicklungsingenieuren als Ersatz für rechnende Angestellte angesehen. Die arithmetische Einheit ersetzte die Tischrechenmaschine, der Speicher den Notizblock und das Programm die Anweisungen, die der Mathematiker den Angestellten gab. Der ENIAC war aus ca. 18 000 Röhren aufgebaut und führte die Addition in 0,2 ms und die Multiplikation in 2 ms aus.

Maßgeblich beteiligt an der Entwicklung moderner Digitalrechner in den USA war John von Neumann, der 1945 als erster vorschlug, das Programm zusammen mit den Daten in einem gemeinsamen Speicher unterzubringen. Dies erlaubt neben der Verwendung von Programmsprüngen auch das Verändern der Befehle während eines Rechenvorganges. Somit können Programme wie Daten behandelt werden, da von der Maschine kein Unterschied zwischen Befehlen und Operanden gemacht wird. Die von ihm vorgeschlagenen Ideen finden sich in den meisten Rechnern der Nachfolgezeit wieder.

Der erste lauffähige Digitalrechner mit intern gespeichertem Programm ist jedoch nicht der von von Neumann entwickelte "Elektronic Discrete Variable Automatic Computer" EDVAC, der erst 1952 fertiggestellt wurde, sondern der daran angelehnte "Electronic Delay Storage Automatic Calculator" EDSAC von Wilkes, der 1949 an der Cambridge University in England in Betrieb genommen wurde. Als Speicher verwendete Wilkes einen akustischen Verzögerungsspeicher mit Quecksilber als Übertragungsmedium, wie er zur Datenspeicherung in Radaranwendungen verwendet wurde. Die Realisierung eines Flipflop-Speichers auf Röhrenbasis war bei größerem Speicherumfang zu teuer. Für die Arithmetik- und Steuereinheit wurden 4000 Röhren verwendet; außerdem standen Oszillographenröhren zur Darstellung der gespeicherten Information zur Verfügung.

In den USA wurde als erster lauffähiger Rechner mit intern gespeichertem Programm der von Eckert und Mauchly entworfene und an den EDVAC angelehnte "Binary Automatic Computer" BINAC gebaut und 1950 in Betrieb genommen. Ein weiterer Rechner von Eckert und Mauchly, der "Universal Automatic Computer" UNIVAC, wurde 1951 fertiggestellt. Er war der bis dahin bestausgeprüfte kommerzielle Rechner und wurde für die Volkszählung von 1950 eingesetzt. Er hatte ein gepuffertes Magnetbandsystem, das Geschwindigkeiten für Vorwärts- und Rückwärtslesen erlaubte, die bis 1968 kaum übertroffen wurden. Der UNIVAC kostete 150 000 Dollar.

Wir wollen die geschichtliche Darstellung der Entwicklung digitaler Rechenmaschinen an dieser Stelle mit einem Zitat aus einer der grundlegendsten Arbeiten[1] auf diesem Gebiet abschließen. Aufgrund der Tatsache, daß Burks, Goldstine und von Neumann mit dieser Veröffentlichung als erste eine umfassende Beschreibung ihres aus den vier Baueinheiten Rechenwerk, Speicherwerk, Leitwerk und Ein- Ausgabewerk bestehenden Rechners gaben, bezeichnet man Universalrechner, die diesem Konzept folgen, kurz als Von-Neumann-Rechner.

"Inasmuch as the completed device will be a general-purpose computing machine it should contain certain main organs relating to arithmetic, memory-storage, control and connection with the human operator. It is intended that the machine be fully automatic in character, i.e. independent of the human operator after the computation starts."

"It is evident that the machine must be capable of storing in some manner not only the digital information needed in a given computation and also the intermediate results of the computation, but also the instructions which govern the actual routine to be performed on the numerical data. In a special-purpose machine these instructions are an integral part of the device and constitute a part of its design structure. For an all-purpose machine it must be possible to instruct the device to carry out any com-

---

[1] Burks, A.W., H.H. Goldstine, J.v. Neumann: Preliminary discussion of the logical design of an electronic computing instrument. (1946). Zitiert nach: The Origins of Digital Computers. Randell, B. (Hrsg.). Berlin, Heidelberg, New York: Springer 1973, S. 371-372.

putation that can be formulated in numerical terms. Hence there must be some organ capable of storing these program orders. There must, moreover, be a unit which can understand these instructions and order their execution."

"Conceptually we have discussed above two different forms of memory: storage of numbers and storage of orders. If, however, the orders to the machine are reduced to a numerical code and if the machine can in some fashion distinguish a number from an order, the memory organ can be used to store both numbers and orders."

"If the memory for orders is merely a storage organ there must exist an organ which can automatically execute the orders stored in the memory. We shall call this organ the Control."

"Inasmuch as the device is to be a computing machine there must be an arithmetic organ in it which can perform certain of the elementary arithmetic operations. There will be, therefore, a unit capable of adding, subtracting, multiplying and dividing."

"The operations that the machine will view as elementary are clearly those which are wired into the machine. To illustrate, the operation of multiplication could be eliminated from the device as an elementary process if one were willing to view as a properly ordered series of additions. Similar remarks apply to division. In general, the inner economy of the arithmetic unit is determined by a compromise between the desire for speed of operation - a non-elementary operation will generally take a long time to perform since it is constituted of a series of orders given by the control - and the desire for simplicity, or cheapness, of the machine."

"Lastly there must exist devices, the input and output organ, whereby the human operator and the machine can communicate with each other."

In diesem Kapitel beschreiben wir grundlegende Gedankengänge, die die Organisation digitaler Rechner geprägt haben: in Abschnitt 1.1 die Automatisierung der Durchführung von Berechnungen, in Abschnitt 1.2 die Automatisierung der Vorbereitung von Programmen und in Abschnitt 1.3 die dem Rechnen mit Dualzahlen zugrundeliegenden Gesetzmäßigkeiten der vier Grundrechenarten.

1.1 Maschinelle Durchführung von Berechnungen

Wenn man eine umfangreichere Berechnung von einem Rechner durchführen läßt, so übergibt man ihm das Lösungsverfahren der Aufgabenstellung in der Form eines Programmes und die Rechengrößen der Aufgabenstellung als Daten. Der Rechner verarbeitet die Daten nach Maßgabe des Programmes und gibt die Ergebnisse der Aufgabenstellung als Nachricht weiter. Es erübrigt sich beinahe festzuhalten, daß man das Ergebnis vor der Rechnung nicht gekannt hat oder sich seiner unsicher war und daß die durch die Nachricht übermittelte Information diese Unsicherheit beseitigt hat.

Informationsverarbeitung mit einem Rechenhilfsmittel (Vier-Spezies-Rechner)

Wir sagen, Information ist beseitigte Unsicherheit. Die undifferenzierteste Information ist offenbar die beseitigte Unsicherheit, die durch die Auswahl aus nur zwei Mög-

lichkeiten entsteht. Ihr wird die Maßeinheit 1 bit zugrundegelegt. Daten oder eine Nachricht zur Verarbeitung bzw. Weitergabe einer Information von 1 bit können dementsprechend durch ein Zeichen aus einem Vorrat von 2 Binärzeichen (binary digits, Bits) dargestellt werden, z.B. durch eine der beiden Dualziffern 0 und 1. Daten oder eine Nachricht mit dem Informationsgehalt von n bit können durch ein Zeichen aus einem Vorrat von 2^n Zeichen oder einer als Einheit aufgefaßten Kombination von n Binärzeichen dargestellt werden, z.B. durch eine n-stellige Dualzahl. Solche als Einheit betrachteten Folgen von Zeichen werden Wörter genannt, und die Menge aller in einem bestimmten Zusammenhang gebildeten Wörter wird als Code bezeichnet. Die Anzahl der Zeichen eines Wortes bildet seine Wortlänge. Ist sie für alle Wörter eines Codes gleich, so spricht man von einem gleichlangen Code. Sind die Zeichen, aus denen die Codewörter bestehen, Binärzeichen (Bits), so spricht man von einem Binärcode. Gleichlange Binärcodes haben in der Rechnerorganisation eine besondere Bedeutung, da Daten und Nachrichten aus technischen Gründen in dieser Form gespeichert bzw. übertragen werden. Dabei werden die Wörter des Codes auf Datenträgern oder in Speichern aufbewahrt, auf bzw. in die sie geschrieben oder von bzw. aus denen sie gelesen werden können. Jedes Wort ist in eindeutiger Weise durch eine Adresse gekennzeichnet, durch deren Angabe es ausgewählt und die entsprechende Speicherzelle beschrieben bzw. ausgelesen werden kann. Ein Speicher, der nur ein Wort aufnehmen, aufbewahren und abgeben kann, wird oft als Register bezeichnet. Auch die Verarbeitung und Übertragung von Daten und Nachrichten erfolgt in digitalen Rechenanlagen meist in der Form von gleichlangen Binärcodes. Im Rechenwerk werden Daten als Dualzahlen mit gleicher Stellenanzahl verarbeitet, im Leitwerk werden Programme als Binärcodes mit gleicher Wortlänge verarbeitet und im Ein-Ausgabewerk werden Daten und Programme als verschlüsselte Zeichen mit gleicher Bitzahl übertragen. Bitgruppen von 8 Bits werden im Zusammenhang mit Rechenanlagen als Bytes bezeichnet.

Nach diesen allgemeinen Betrachtungen wenden wir uns dem eigentlichen Thema dieses Abschnittes zu und analysieren zunächst die Tätigkeit eines Menschen, der mit einem Vier-Spezies-Rechner eine etwas kompliziertere Rechnung durchführt. Ein Vier-Spezies-Rechner ist eine Rechenmaschine, mit der im wesentlichen die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division durchgeführt werden können. Die Eingabe der Operanden und die Auswahl der Operationen erfolgt über eine Tastatur. Die Ausgabe der Ergebnisse geschieht durch Ausdrucken oder Anzeigen der Zahlen.

<u>Ein Programmbeispiel.</u> Wir wollen uns vergegenwärtigen, wie ein Mensch mit einem solchen Rechner den Wert eines Polynoms berechnet, z.B. den Wert von

$$\begin{aligned} p &= a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 \\ &= ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0 \end{aligned}$$

mit $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$.

Der Rechner besitzt ein sog. Akkumulator-Register, das auch als Anzeige beim Eingeben und Ablesen von Zahlen dient. Mit dem Rechner können unter anderem folgende Operationen durchgeführt werden.

Taste | Variable	Wirkung
↓ M	lädt den Akkumulator mit dem Wert von M
+ M	addiert den Wert von M auf den im Akkumulator gespeicherten Wert
* M	multipliziert den Wert von M mit dem im Akkumulator gespeicherten Wert
⋮	

In der nächsten Tabelle sind die Operationen zur Polynomauswertung aufgelistet, zusammen mit den Werten der Koeffizienten a_0 bis a_3 und der Variablen x, für die der Wert des Polynoms berechnet werden soll. Die Größe z ist eine Konstante mit dem Wert 0, die zum "Löschen" des Akkumulators zu Beginn der Rechnung dient.

Reihenfolge der Operationen (Programm)	Namen und Werte der beteiligten Operanden
0: ↓z	z : 0
1: $+a_3$	a_3: 9
2: *x	a_2: 8
3: $+a_2$	a_1: 7
4: *x	a_0: 6
5: $+a_1$	x : 3
6: *x	p :
7: $+a_0$	
8: Schreibe den im Akkumulator enthaltenen Wert nach p	

Mit der Angabe des Programms und der Rechengrößen ist die Berechnung des Polynoms vollständig beschrieben, so daß der Wert von p mit dem Rechner ermittelt werden kann, ohne daß der Operateur, der den Rechner bedient, zu wissen braucht, daß es sich um eine Polynomauswertung handelt. Die letzte Operation wird nicht vom Rechner, sondern nur vom Operateur ausgeführt. Sie weist ihn an, den im Akkumu-

lator angezeigten Wert abzulesen und unter dem Variablennamen p zu notieren. Auf dem Papier wurde ein Platz "reserviert", da der Wert von p beim Formulieren des Programmes noch nicht bekannt war.

Tätigkeit des Operateurs. Dem Operateur liegt die Aufgabenstellung in der Form eines Programmes vor, das aus sog. Befehlen besteht. Der Operateur liest den ersten Befehl vom Papier ab und interpretiert ihn. Ein Befehl ist aufgeteilt in Operationscode (Name der Operation) und Operandenadresse (Name des Operanden). Er sucht den zur Operandenadresse gehörenden Wert des Operanden und liest ihn vom Papier ab. Anschließend tastet er ihn in den Rechner ein und betätigt die Operationstaste. Dann liest er den nächsten Befehl, interpretiert ihn und führt ihn in gleicher Weise aus. In Bild 1.1 ist die Tätigkeit des Operateurs durch Pfeile angedeutet, die, bei einem gedachten Zähler beginnend, in der Reihenfolge ihrer Numerierung durchlaufen werden. Immer wenn ein Befehl abgerufen und ausgeführt ist, wird der Zähler um 1 hochgezählt.

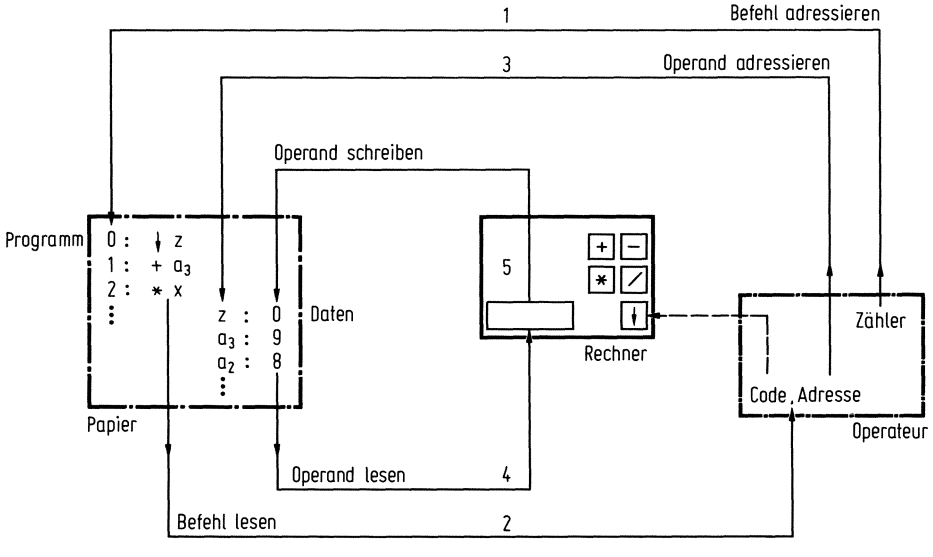


Bild 1.1 Ausführung eines Programmes mit einem Taschenrechner

Programme sind im allgemeinen nicht so "geradeaus" geschrieben wie unser Programmbeispiel zur Polynomauswertung, sondern enthalten "Schleifen", wodurch der lineare Ablauf durch Zyklen unterbrochen wird. Dies bedingt Befehle an den Operateur, den Zählerstand in Abhängigkeit von Bedingungen zu verändern. Im folgenden, erheblich komplizierteren Programm zur Polynomauswertung sind weitere Befehle an den Operateur vorgesehen, nämlich die Adresse in einem Befehl durch eine andere zu ersetzen bzw. die Adresse in einem Befehl mit einer anderen zu ver-

gleichen. Adreßersetzungsbefehle waren bereits in dem von Burks, Goldstine und von Neumann beschriebenen Rechnerkonzept vorgesehen.

Befehls-adresse	Programm
0	$\downarrow z$
1	$+a_3$
2	$*x$
3	Ersetze a_i des unter 1 gespeicherten Befehls durch a_{i-1}.
4	Ist die Adresse des unter 1 gespeicherten Befehls a_0, dann überspringe den nächsten Befehl.
5	Führe den unter 1 gespeicherten Befehl als nächsten Befehl aus.
6	$+a_0$
7	Schreibe den im Akkumulator enthaltenen Wert nach p.

Informationsverarbeitung mit einem Rechenautomaten (Von-Neumann-Rechner)

Ein Rechenautomat vermag Programme auszuführen, ohne daß ein Operateur zur Durchführung der Rechnung nötig ist. An seine Stelle tritt ein elektronisches Schaltwerk, das Leitwerk. Als Rechner dient ein elektronisches Rechenwerk, und anstelle des Papiers wird zur Speicherung von Programmen und Daten ein elektronisches Speicherwerk verwendet. Mit einem solchen Rechenautomaten heutiger Generation können mehr als eine Million Operationen pro Sekunde ausgeführt werden. Diesem Vorteil hoher Rechengeschwindigkeit steht jedoch der Nachteil mühseliger Programmierung gegenüber.

Das Programmbeispiel. Wir legen für unser Programmbeispiel einen einfachen Universalrechner zugrunde, dessen Speicher eine Wortlänge von 16 Bits hat. Soll ein

Befehlscode	Speicherzelle	Wirkung
0001		hält den Rechner am Ende des Programmes an
0101	M	speichert den Inhalt des Akkumulators nach M
1000	M	lädt den Akkumulator mit dem Inhalt von M
1100	M	addiert den Inhalt von M auf den Inhalt des Akkumulators
1110	M	multipliziert den Inhalt von M auf den Inhalt des Akkumulators
⋮		

Befehl gerade in ein Speicherwort passen, so müssen die 16 Bits auf Operationscode und Operandenandresse aufgeteilt werden. Wir wählen 4 Bits für den Operationscode und 12 Bits für die Operandenadresse, wodurch $2^4 = 16$ verschiedene Operationen codiert und $2^{12} = 4096$ verschiedene Operanden adressiert werden können. Operationscode und Operandenadressen werden als 4- bzw. 12-stellige Dualzahlen aufgefaßt. Mit dem Rechner können unter anderem die auf S. 8 unten zusammengestellten Operationen durchgeführt werden.

In der nächsten Tabelle sind die Befehle zur Polynomauswertung aufgelistet, zusammen mit den Werten der Operanden, die ab Zelle 10 im Speicher stehen. Der mit Sternchen versehene Inhalt der Speicherzelle 16 ist bedeutungslos, da er mit dem Ergebnis der Polynomauswertung überschrieben wird.

Adressen	Befehle und Operanden
00000	1000000000001010
00001	1100000000001011
00010	1110000000001111
00011	1100000000001100
00100	1110000000001111
00101	1100000000001101
00110	1110000000001111
00111	1100000000001110
01000	0101000000010000
01001	0001000000000000
01010	0000000000000000
01011	0000000000001001
01100	0000000000001000
01101	0000000000000111
01110	0000000000000110
01111	0000000000000011
10000	**********

Befehle und Operanden des Programmes werden als 0-1-Kombinationen bzw. Dualzahlen in ein und demselben Speicher aufbewahrt. Sie sind nur durch ihren Platz im Speicher zu unterscheiden: In den Zellen 0 bis 9 sind die Befehle und in den Zellen 10 bis 16 die Operanden gespeichert. Bei der Adresse 0 beginnend werden die Befehle der Reihe nach ausgeführt. Der letzte Befehl in der Zelle 9 verhindert, daß der erste Operand als Befehl interpretiert wird.

Funktion des Leitwerks. Die Tätigkeit des Operateurs bei der Durchführung einer Rechnung mit dem Vier-Spezies-Rechner wird beim Von-Neumann-Rechner durch die Funktion des Leitwerks ersetzt. Im Leitwerk sind Register zum Zählen und zum Zwischenspeichern der Befehle vorgesehen: der Befehlszähler und das Befehlsregister. Der Befehlszähler wird mit der Adresse des ersten Befehls geladen. Anschließend wird der erste Befehl aus dem Speicher gelesen, ins Befehlsregister gebracht und entschlüsselt. Je nach Befehlscode wird entweder der Operand aus dem Speicher gelesen und in den Akkumulator gebracht bzw. mit seinem Inhalt verknüpft, oder der Inhalt des Akkumulators wird in den Speicher geschrieben, oder der Befehl wird vom Leitwerk unmittelbar ausgeführt. Inzwischen ist der Befehlszählerstand um 1 erhöht worden, so daß der nächste Befehl in gleicher Weise interpretiert und ausgeführt werden kann. In Bild 1.2 ist die Funktion des Leitwerks durch

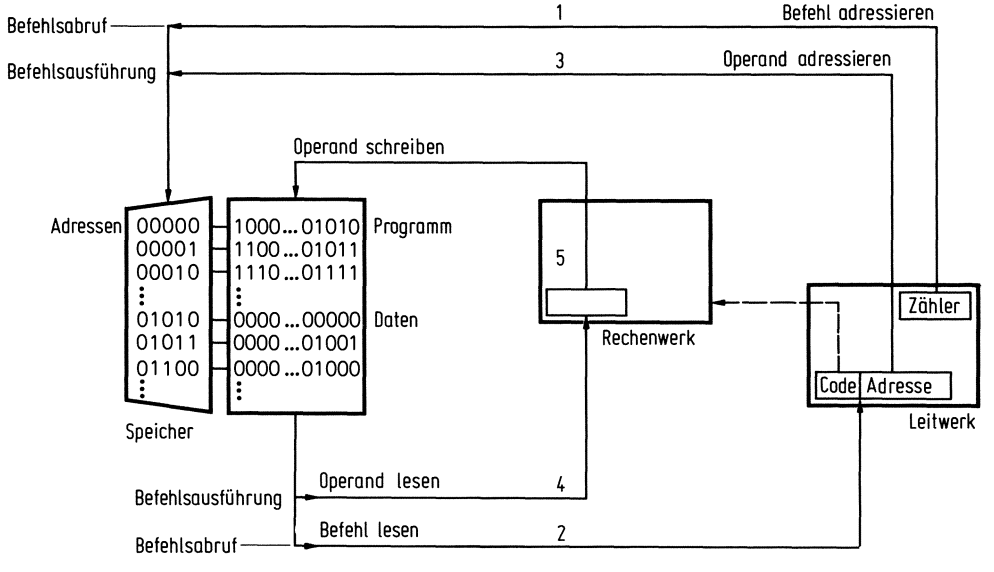


Bild 1.2 Ausführung eines Programmes mit einem Universalrechner

Pfeile angedeutet. Man erkennt deutlich die beiden Kreise aus Bild 1.1, die in Bild 1.2 über einen Speicher laufen und durch sog. Multiplexer und Demultiplexer ausgewählt werden. Dementsprechend erfolgt ein Rechenschritt in zwei Phasen: dem Befehlsabruf (durch 1 und 2 in Bild 1.2 gekennzeichnet) und dem Operandenabruf (3 und 4 in Bild 1.2) mit Ausführung der Operation (5 in Bild 1.2). Man bezeichnet die entsprechenden Phasen als Abruf- und als Ausführungsphase. Bild 1.3 zeigt in der Art einer Momentaufnahme die Verarbeitung des in Zelle 1 gespeicherten Befehls des Programmes zur Polynomauswertung, wobei die eingetragenen Markierungen mit Bild 1.2 übereinstimmen.

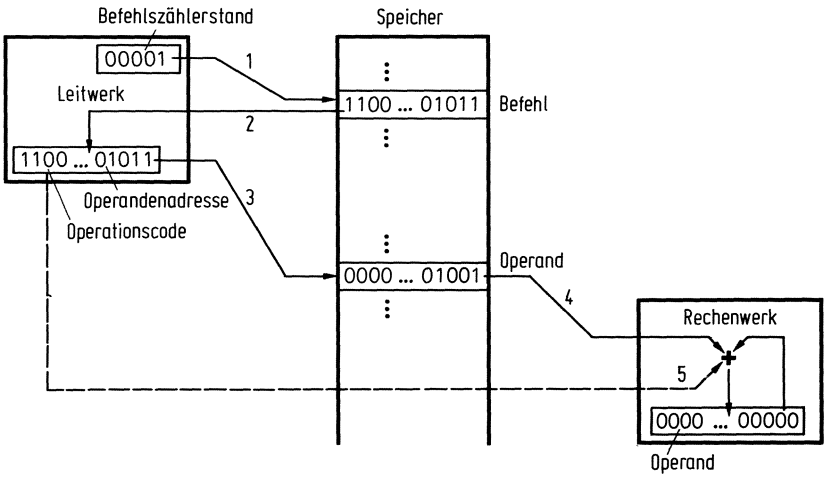


Bild 1.3 Ausführung des in der Zelle 1 gespeicherten Befehls

Ein-Ausgabewerk. Soweit beschrieben, wurde der Rechner als in sich abgeschlossen betrachtet. Um jedoch Befehle und Operanden in die Maschine eingeben und Resultate ausgeben zu können, bedarf es eines weiteren Werkes, des Ein-Ausgabewerkes. Dieses Werk ist das vierte Werk entsprechend dem Vorschlag von Burks, Goldstine und von Neumann. Bei diesem Konzept sind die Ein-Ausgabeleitungen direkt mit dem Akkumulator verbunden. Dementsprechend müssen Befehle vorgesehen werden, wie z.B. "Lies ein Wort von einem Peripheriegerät und bringe es in den Akkumulator" und "Nimm ein Wort vom Akkumulator und schreibe es in ein Peripheriegerät" bzw. "Gib ein Wort ein" oder "Gib ein Wort aus". Stünde z.B. im Programm zur Polynomauswertung der Wert von x nicht schon beim Schreiben des Programmes fest, oder sollte dasselbe Programm zur Berechnung des Polynoms mit verschiedenen Werten für das Argument x benutzt werden, so hätte man dem Programm zwei Befehle voranstellen müssen: "Gib den Wert von x ein" und "speichere den Inhalt des Akkumulators nach Zelle 15".

Bild 1.4 zeigt die vier Werke des Von-Neumann-Rechners in einer abschließenden Darstellung. Neben der charakteristischen Struktur sind es die folgenden Punkte, die einen Universalrechner nach Burks, Goldstine und von Neumann kennzeichnen:

1. die gleichartige Darstellung von Befehlen und Operanden,
2. die Speicherung von Programmen und Daten in einem gemeinsamen Speicher,
3. die Auslegung des Rechenwerkes für Dualzahlen in 2-Komplement-Darstellung,
4. Befehle zur Abfrage von Bedingungen und zum Verändern des Befehlszählerstands,
5. Befehle zur Ersetzung der Operandenadressen innerhalb von Programmen.

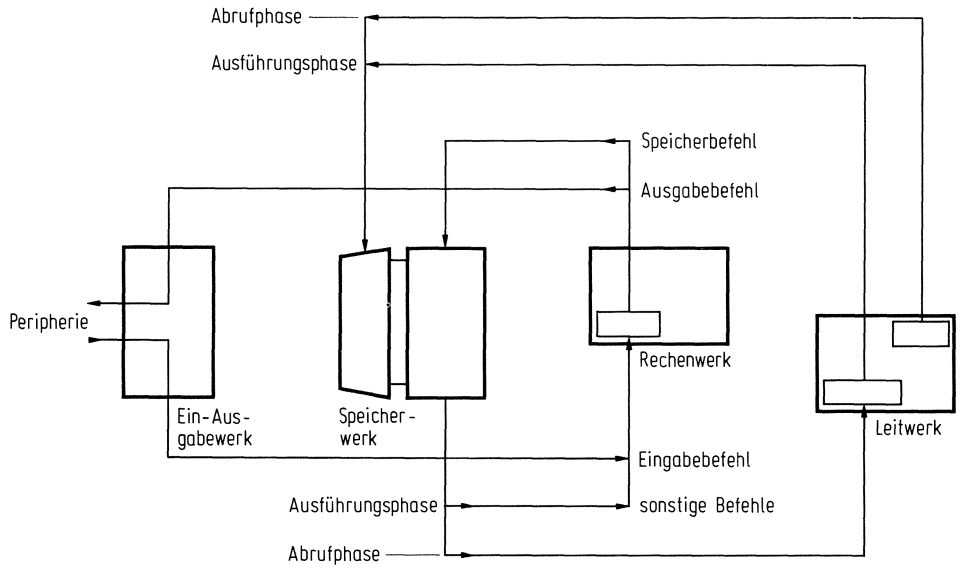


Bild 1.4 Struktur des Von-Neumann-Rechners

1.2 Maschinelle Vorbereitung von Programmen

Ein dem Menschen verständliches Programm besteht aus Symbolen für die Operationen, Buchstaben für die Namen und Dezimalzahlen für die Werte der Variablen. Ein dem Rechner verständliches Programm besteht demgegenüber aus 0-1-Kombinationen für die Operationen und Dualzahlen für die Adressen und die Werte der Operanden. Ein dem Menschen verständliches Programm kann in ein dem Rechner verständliches Programm umgeformt werden, so daß beide Programme den gleichen Algorithmus beschreiben. Diese Umformung hat zur Folge, daß das Programm in einer dem Menschen unverständlichen Form erscheint. Sie wird als Codierung bezeichnet, was sich mit der umgangssprachlichen Verwendung dieses Begriffes deckt, wenn man das Programm als Nachricht auffaßt.

Codierung von Programmen durch den Menschen (Codierer)

Unter dem Begriff Codierung versteht man die Umformung von Zeichen oder Zeichenfolgen in andere Zeichen oder Zeichenfolgen, wobei die Information, die die Zeichen bzw. Zeichenfolgen darstellen, unverändert bleibt. Die Zeichen eines Codes, dessen Zeichenvorrat mehr als zwei Zeichen umfaßt, können durch Folgen von Binärzeichen, d.h. durch Wörter eines Binärcodes dargestellt werden. Bild 1.5 und Bild 1.6 zeigen die Zuordnung der Zeichen eines symbolischen Codes (Alphabet der lateinischen Buchstaben, Dezimalziffern und Sonderzeichen) zu den Wörtern

eines Binärcodes mit einer Wortlänge von 12 Bits (Lochkartencode der IBM) bzw. 8 Bits (Extended Binary Coded Decimal Interchange Code - EBCDIC).

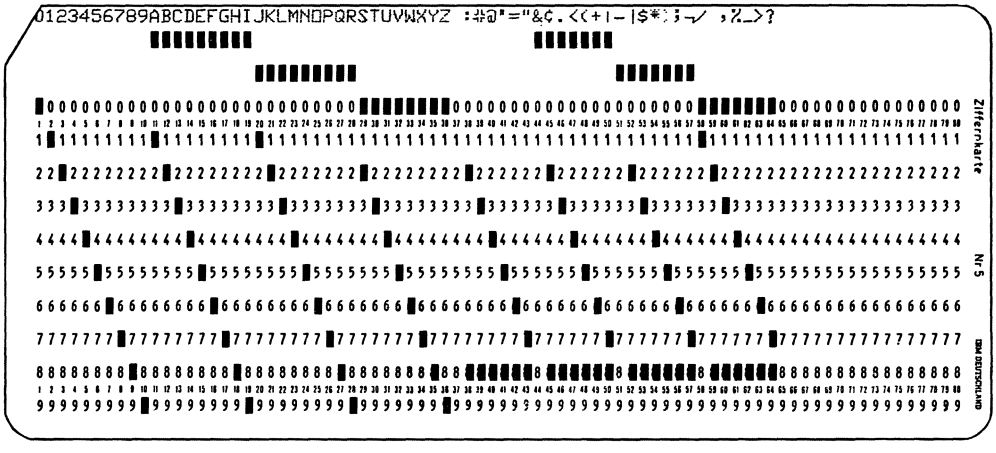


Bild 1.5 IBM-Lochkarte mit einem gelochten Zeichenvorrat

hintere Tetrade \ vordere Tetrade	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000					b̸	&	–									0
0001							/		a	j			A	J		1
0010									b	k	s		B	K	S	2
0011									c	l	t		C	L	T	3
0100									d	m	u		D	M	U	4
0101		↵							e	n	v		E	N	V	5
0110									f	o	w		F	O	W	6
0111									g	p	x		G	P	X	7
1000									h	q	y		H	Q	Y	8
1001									i	r	z		I	R	Z	9
1010					¢	!	^	:								
1011					.	$	,	#								
1100					<	*	%	@								
1101					(	)	_	'								
1110					+	;	>	=								
1111					|	¬	?	"								

b̸ Zwischenraum (Blank)
↵ Wagenrücklauf und Zeilenvorschub (New Line)

Bild 1.6 EBCDI-Code mit einem eingetragenen Zeichenvorrat

Der EBCDI-Code wie auch der Lochkartencode zeigen recht deutlich, wie wir es hier nicht nur mit Codierung im allgemeinsten Sinn als der Darstellung einer Nachricht in anderer Form, sondern mit einem wesentlich spezielleren Vorgang zu tun haben: "mit der Darstellung eines einzelnen Nachrichtenelementes, aus einer größeren Mannigfaltigkeit, durch eine Mehrzahl von Nachrichtenelementen aus einer (meist) kleine-

ren Mannigfaltigkeit. Dieser Vorgang heißt Codierung (im engeren Sinn), seine Umkehrung Decodierung."[1]

Nach diesen allgemeinen Betrachtungen kehren wir zu unserem eigentlichen Problemkreis zurück und analysieren zunächst die Tätigkeit eines Menschen, der ein Programm und die zugehörigen Daten in einer der Maschine verständlichen Form codiert.

Das Programmbeispiel. In der folgenden Tabelle ist das Programm zur Polynomauswertung in der dem Menschen und in der der Maschine verständlichen Form nebeneinander dargestellt.

Maschinenexterne, für den Menschen verständliche Form	Maschineninterne, für den Rechner verständliche Form
0: ↓z	00000: 1000000000001010
1: $+a_3$	00001: 1100000000001011
2: *x	00010: 1110000000001111
3: $+a_2$	00011: 1100000000001100
4: *x	00100: 1110000000001111
5: $+a_1$	00101: 1100000000001101
6: *x	00110: 1110000000001111
7: $+a_0$	00111: 1100000000001110
8: ↑p	01000: 0101000000010000
9: Halt	01001: 0001000000000000
z: 0	01010: 0000000000000000
a_3: 9	01011: 0000000000001001
a_2: 8	01100: 0000000000001000
a_1: 7	01101: 0000000000000111
a_0: 6	01110: 0000000000000110
x: 3	01111: 0000000000000011
p:	10000: **********

Jede Zeile auf der linken Seite der Tabelle enthält einen symbolischen Befehl mit zugehöriger dezimaler Befehlsadresse oder einen Variablenwert mit zugehörigem symbolischen Variablennamen. Jede Zeile auf der rechten Seite enthält die ent-

[1] Berger-Damiani, E.R. Nachrichtentheorie, Zahlensysteme und Codierung. Abschnitt 5.3 in Steinbuch, K., Weber, W.: Taschenbuch der Informatik, Band II. Berlin, Heidelberg, New York: Springer 1974, S. 94.

sprechende Information in codierter Form: einen Maschinenbefehl mit zugehöriger dualer Adresse bzw. einen dualen Operanden mit zugehöriger dualer Adresse (die Sterne in der letzten Zeile sollen andeuten, daß der Inhalt der Speicherzelle 16 bedeutungslos ist). Zeichen wie z.B. *, x, 3 bzw. 0, 1 sind Elemente von zwei Codes. Während *, x und 3 Elemente des maschinenexternen Codes sind, bilden 0 und 1 die Elemente des maschineninternen Codes. Bevor ein Programm von einem Rechner ausgeführt werden kann, muß es in den maschineninternen Code (Maschinen- oder Interncode) umgeformt werden.

Die Tätigkeit des Codierers. Zu Beginn seiner Tätigkeit liegt dem Codierer eine erste Tabelle vor, die sog. Zuordnungstabelle. Sie beschreibt die Zuordnungen der symbolischen zu den binär codierten Operatoren.

Die Codierung des symbolischen Programmes erfolgt in zwei Arbeitsgängen. Im ersten Arbeitsgang werden die Zeilen des Programms numeriert, so daß jede Zeilennummer der Adresse einer Speicherzelle entspricht. Die Variablennamen erhalten dadurch äquivalente Speicheradressen, die als Dualzahlen geschrieben werden. Die Zuordnungen zwischen symbolischen und dualen Adressen hält der Codierer in Form einer zweiten Tabelle fest, der sog. Symboltabelle.

Im zweiten Arbeitsgang werden die Befehle nacheinander codiert, indem der Codierer bei jedem Befehl in der Zuordnungs- und in der Symboltabelle nachsieht, wel-

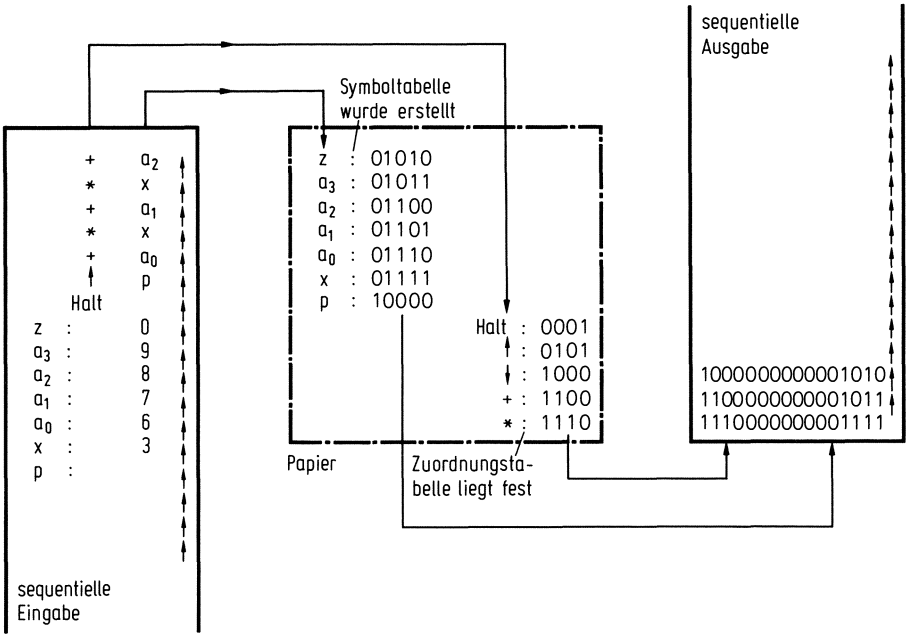


Bild 1.7 Codierung eines symbolischen Programmes von Hand

cher binäre welchem symbolischen Befehlscode und welche binäre welcher symbolischen Adresse zugeordnet ist. Der binär codierte Befehlscode und die binär codierte Adresse werden zu 16-Bit-Wörtern zusammengesetzt.

Nach der Codierung der Befehle erfolgt die Codierung der Daten, indem die vorgegebenen Werte der Operanden in 16-stellige Dualzahlen umgeformt werden. In Bild 1.7 ist die Tätigkeit des Codierers im zweiten Arbeitsgang beim Umsetzen des symbolischen Programms zur Polynomauswertung in das entsprechende Maschinenprogramm veranschaulicht. Es zeigt den Codiervorgang zu einem Zeitpunkt, in dem die ersten drei Zeilen des symbolischen Programms in binär codierter Form vorliegen. Die zur Codierung des symbolischen Programms notwendigen Tabellen sind im ersten Arbeitsgang erstellt worden (Symboltabelle) bzw. lagen zu Beginn der Tätigkeit des Codierers fest (Zuordnungstabelle).

Codierung von Programmen durch die Maschine (Assembler)

Das Codieren eines symbolischen Programms erfolgt nach festen Regeln, so daß es algorithmisch beschrieben und mit Hilfe eines Programms, des sog. Assemblers[1], von der Rechenmaschine durchgeführt werden kann. Wir stellen uns vor, daß der Assembler von Hand codiert wurde und in ablauffähiger Form im Speicher des Rechners vorliegt. Das symbolische Programm wird vom Assembler als Eingabedaten eingelesen, in das Maschinenprogramm umgesetzt und als Ausgabedaten ausgegeben. Da die Daten, mit denen der Rechner arbeitet, binär codiert sein müssen, muß auch das symbolische Programm in binär codierter Form vorliegen. Wählt man z.B. den Lochkartencode zur externen binären Darstellung des symbolischen Programms, so muß man sich beim Abfassen des Programmtextes auf den Zeichenvorrat dieses Codes beschränken. Das symbolische Programm muß also in einer ganz bestimmten Form, der sog. Assemblersprache, abgefaßt sein. In dieser Sprache werden Rechnerbefehle durch bestimmte mnemonische[2] Symbolfolgen anstelle einzelner Sonderzeichen und Adressen durch frei wählbare Buchstabenfolgen dargestellt, die mit Ausnahme des ersten Zeichens auch Ziffern enthalten können. Spezielle Assembleranweisungen versorgen den Assembler mit Informationen, die zur Durchführung der Assemblierung und zur Erzeugung des Datencodes nötig sind. Auch diese Anweisungen sind durch mnemonische Symbolfolgen festgelegt.

Das Programmbeispiel. Wir legen eine einfache Assemblersprache zugrunde, mit der unter anderem die folgenden Rechnerbefehle und Assembleranweisungen umgesetzt bzw. ausgeführt werden können.

---

[1] Aus dem Englischen: to assemble - zusammenfügen, montieren.
[2] Aus dem Griechischen: mneme - Gedächtnis.

Rechner-befehle	Wirkung
HLT	hält den Rechner am Ende des Programmes an
STA M	speichert den Inhalt des Akkumulators nach M
LDA M	lädt den Akkumulator mit dem Inhalt von M
ADD M	addiert den Inhalt von M auf den Inhalt des Akkumulators
MUL M	multipliziert den Inhalt von M auf den Inhalt des Akkumulators
⋮	

Assembler-anweisungen	Wirkung
ORG c	definiert den Anfang eines Programmes, wobei die Zahl c die Adresse seiner ersten Speicherzelle festlegt; die Adressen der folgenden Speicherzellen sind fortlaufend um 1 erhöht
RES c	reserviert so viele Speicherzellen, wie die Zahl c angibt; die Adresse der folgenden Speicherzelle ist um die Anzahl reservierter Zellen erhöht
DAT c	belegt eine Speicherzelle mit einer Dualzahl, die der Zahl c entspricht; die Adresse der folgenden Speicherzelle ist um 1 erhöht
END	zeigt das Ende des zu assemblierenden Programms an
⋮	

In der Tabelle auf S. 18 sind links die symbolischen Rechnerbefehle und Assembleranweisungen für das Programm zur Polynomauswertung aufgelistet. Die Beschreibung der Aufgabe in unserer einfachen Assemblersprache ist unmittelbar verständlich. Rechts sind das Programm und seine Daten in der sog. Maschinensprache dargestellt, wie sie vom Assembler erzeugt wird. Das Programm kann in dieser Form vom Rechner unmittelbar verarbeitet werden.

Funktion des Assemblers. Die Tätigkeit des Codierers bei der Assemblierung eines Programms wird durch die Funktion des Assemblers ersetzt. Der Assembler liest die symbolischen Rechnerbefehle und Assembleranweisungen zeichenweise ein und faßt die Zeichen zu Symbolen und Konstanten zusammen. Die Umsetzung der symbolischen Operationscodes in ihre interne Darstellung erfolgt über die im Assembler gespeicherte Zuordnungstabelle. Für die Zuordnung von symbolischen zu numerischen Adressen baut der Assembler die Symboltabelle auf. Die dualen Adressen werden von einem vom Assembler vorgesehenen Zähler bestimmt, der, ausgehend von der in der ORG-Anweisung angegebenen Zahl, den Maschinenwörtern Nummern zu-

Assemblercode			Maschinencode
	ORG	0	
	LDA	Z	1000000000001010
	ADD	A3	1100000000001011
	MUL	X	1110000000001111
	ADD	A2	1100000000001100
	MUL	X	1110000000001111
	ADD	A1	1100000000001101
	MUL	X	1110000000001111
	ADD	A0	1100000000001110
	STA	P	0101000000010000
	HLT		0001000000000000
Z	DAT	0	0000000000000000
A3	DAT	9	0000000000001001
A2	DAT	8	0000000000001000
A1	DAT	7	0000000000000111
A0	DAT	6	0000000000000110
X	DAT	3	0000000000000011
P	RES	1	**********
	END		

ordnet, die den Speicheradressen dieser Wörter entsprechen. Ein Symbol, das links vom Operationscode einer Anweisung steht, erhält als Adresse den Zählerstand des zugehörigen Maschinenwortes. Dabei ist es gleichgültig, ob das Symbol einen Maschinenbefehl oder einen Operanden bezeichnet. Konstanten werden vom Assembler von der dezimalen in die duale Form übergeführt.

Zur Eingabe eines in der Assemblersprache verfaßten Programmes muß das Programm auch in externer Darstellung, z.B. auf Lochkarten, in binärcodierter Form vorliegen. Der Lochkartenleser setzt die im Lochkarten-Code verschlüsselten Symbole z.B. in den EBCDI-Code um und übermittelt dem Rechner das Programm in der Form einer Bytekette (Bild 1.8).

Wir haben also mehrere verschiedene Codes des Programmes und seiner Daten vor uns: den symbolischen Externcode (Assemblercode), den binären Externcode als Lochkarten - und als EBCDI-Code und den binären Interncode (Maschinencode). Der Rechner wird in zwei Verarbeitungsarten benutzt: zunächst zur Codierung und dann zur Ausführung des Programms. In der Programm-Codierphase wird dem Rechner der binäre Externcode und in der Programm-Ausführungsphase der in der Codierphase

erzeugte binäre Interncode zugeführt. Ein Universalrechner kann also neben der Ausführung von arithmetischen Rechnungen auch zur Umformung von Symbolfolgen, d.h. zur Codierung im allgemeinen Sinne benutzt werden.

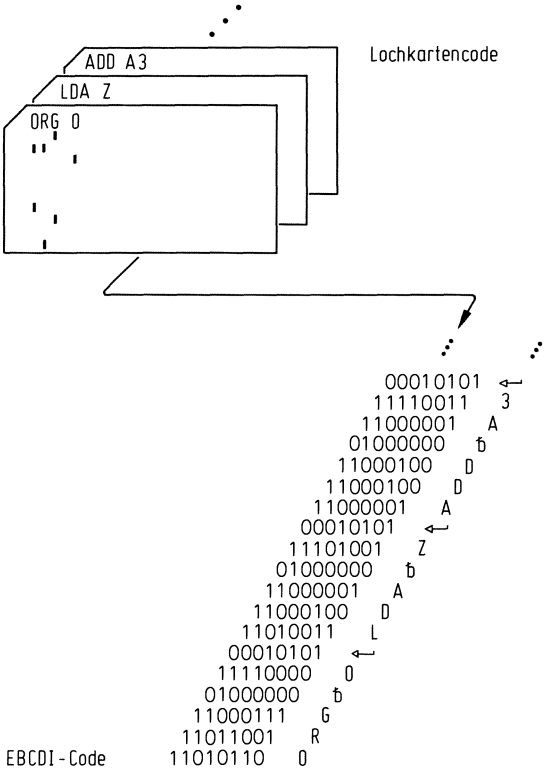


Bild 1.8 Umsetzung des Lochkartencodes in den EBCDI-Code am Beispiel eines Programmes

1.3 Mikroalgorithmen für die vier Grundrechenarten

Die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division sind uns durch den Umgang im täglichen Leben derart zur Gewohnheit geworden, daß wir uns der Regeln, nach denen wir die einzelnen Schritte einer solchen Operation ausführen, kaum noch bewußt sind. Zur technischen Realisierung dieser Rechenoperationen ist es aber notwendig, sie in eine Folge von Elementarhandlungen zu zerlegen und sie durch Algorithmen zu beschreiben. Als Elementarhandlungen werden im allgemeinen einfache arithmetische Operationen gewählt wie Zählen, Addition, Multiplikation mit 2, Division durch 2, Bildung des Komplementes, Abfragen auf Null usw. Diese Elementarhandlungen werden auch Mikrofunktionen und die aus ihnen be-

stehenden Algorithmen Mikroalgorithmen genannt. Wir beschränken uns hier auf Mikrofunktionen und Mikroalgorithmen, die die vier Spezies Addition, Subtraktion, Multiplikation und Division für ganze Dualzahlen in 2-Komplement-Darstellung beschreiben.

2-Komplement-Zahlen

Zur Darstellung positiver und negativer Dualzahlen benötigen wir (n-1) Bits für die Ziffern und 1 Bit für das Vorzeichen. Dann lassen sie sich gerade durch n Bits darstellen und in einer Speicherzelle der Wortlänge n speichern. Zwei Möglichkeiten sind zur Darstellung vorzeichenbehafteter Dualzahlen naheliegend und charakteristisch.

1. Die Darstellung positiver und negativer Dualzahlen als Vorzeichen-Zahlen.

Vorzeichen	Stelle n - 1 ↓ Stelle n - 2 ↓ ... Stelle 0 ↓
positiv	$0\ 1 \ldots 111 = +2^{n-1}-1$ ← größte Zahl
	$0\ 1 \ldots 110 = +2^{n-1}-2$
	$0\ 1 \ldots 101 = +2^{n-1}-3$
	⋮ ⋮
	$0\ 0 \ldots 010 = +2$
	$0\ 0 \ldots 001 = +1$
	$0\ 0 \ldots 000 = +0$
negativ	$1\ 0 \ldots 000 = -0$
	$1\ 0 \ldots 001 = -1$
	$1\ 0 \ldots 010 = -2$
	⋮ ⋮
	$1\ 1 \ldots 101 = -(2^{n-1}-3)$
	$1\ 1 \ldots 110 = -(2^{n-1}-2)$
	$1\ 1 \ldots 111 = -(2^{n-1}-1)$ ← kleinste Zahl

2. Die Darstellung positiver und negativer Zahlen als 2-Komplement-Zahlen.

Ausgehend von der größten positiven Zahl $+2^{n-1}-1$, wird hier von Zeile zu Zeile 1 als kleinste Einheit subtrahiert. Der Übertrag bei der Subtraktion +0-1 bewirkt, daß das Vorzeichen automatisch von 0 auf 1, d.h. von + auf - springt. Positive Zahlen und negative Zahlen mit gleichem Wert ergänzen sich zu 2^n, wenn die Vorzeichenstelle wie eine Ziffer mit der Wertigkeit 2^{n-1} aufgefaßt wird.

Vorzeichen	Stelle n - 1 ↓ Stelle n - 2 ↓ ... Stelle 0 ↓	
positiv	$0\ 1 \ldots 111 = +2^{n-1}-1$	← größte Zahl
	$0\ 1 \ldots 110 = +2^{n-1}-2$	
	$0\ 1 \ldots 101 = +2^{n-1}-3$	
	⋮ ⋮	
	$0\ 0 \ldots 010 = +2$	
	$0\ 0 \ldots 001 = +1$	
	$0\ 0 \ldots 000 = +0$	
negativ	$1\ 1 \ldots 111 = -1$	
	$1\ 1 \ldots 110 = -2$	
	$1\ 1 \ldots 101 = -3$	
	⋮ ⋮	
	$1\ 0 \ldots 010 = -2^{n-1}+2$	
	$1\ 0 \ldots 001 = -2^{n-1}+1$	
	$1\ 0 \ldots 000 = -2^{n-1}+0$	← kleinste Zahl

Man nennt zwei Zahlen b-komplementär bezüglich b^n, wenn ihre Summe b^n ergibt. Im Dualsystem ist b=2, und dementsprechend heißen Zahlen mit dieser Eigenschaft 2-Komplement-Zahlen. Andererseits ergibt die Summe einer positiven und der entsprechenden negativen Zahl 0, wenn die Vorzeichenstelle mit der Wertigkeit -2^{n-1} versehen wird. Der Wert einer 2-Komplement-Zahl $Q = q_{n-1}\, q_{n-2} \cdots q_0$ läßt sich durch

$$Q = q_{n-1}\, q_{n-2} \cdots q_0 = \sum_{i=0}^{n-2} q_i \cdot 2^i - q_{n-1} \cdot 2^{n-1} \qquad (1.1)$$

darstellen. Die Stelle q_{n-1} ist die Vorzeichenstelle. Wenn sie als Ziffer mit der Wertigkeit -2^{n-1} betrachtet wird, so läßt sich nach Gl. (1.1) der Betrag der Zahl errechnen.

<u>Zahlenring.</u> Wird von der kleinsten Zahl 10...000 eine 1 subtrahiert, so entsteht die größte Zahl 01...111, oder wird umgekehrt eine 1 zu 01...111 addiert, so entsteht 10...000. Das veranlaßt uns, die Zahlen nicht als Strecke, sondern als Ring darzustellen (Bild 1.9). Sobald auf diesem Ring die Grenze zwischen der größten und der kleinsten Zahl überschritten wird, muß dies als Bereichsüberschreitung angezeigt werden.

Der Zahlenring dient zur Veranschaulichung arithmetischer Operationen. Die einfachste arithmetische Operation ist das Zählen. Im Zahlenring wird Vorwärts- und Rückwärtszählen als Nach-Rechts- bzw. Nach-Links-Schreiten dargestellt. Damit wird einleuchtend, warum gerade diese Codierung der Zahlen für arithmetische Operationen günstig ist:

Vorwärtszählen: Eine Zahl - ob positiv oder negativ - wird durch Addition von +1 = 00...001 um 1 erhöht.

Rückwärtszählen: Eine Zahl - ob positiv oder negativ - wird durch Subtraktion von +1 = 00...001 um 1 vermindert.

Addition und Subtraktion können im Prinzip auf Vorwärts- und Rückwärtszählen zurückgeführt werden. Multiplikation und Division bestehen im Prinzip aus Addition und Subtraktion.

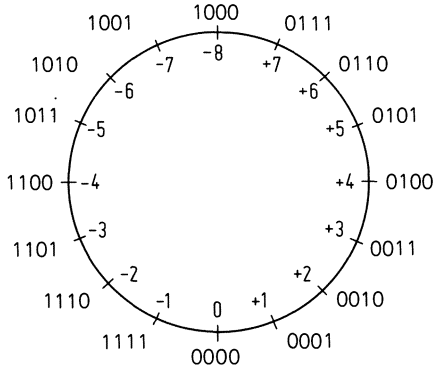


Bild 1.9 Zahlenring für die 2-Komplement-Zahlen -8 bis +7

Im folgenden geben wir Algorithmen für die gundlegenden Mikrofunktionen Zählen, Komplementieren, Addieren und Subtrahieren in Form von booleschen Algorithmen[1] an, d.h. in Form von Algorithmen, deren Anweisungen boolesche Funktionen sind. Die höheren Operationen Multiplizieren und Dividieren werden in Form von Mikroalgorithmen[2] beschrieben, d.h. in Form von Algorithmen, deren Anweisungen Mikrofunktionen enthalten. Dementsprechend haben die Operationszeichen + und · unterschiedliche Bedeutungen. In booleschen Algorithmen bezeichnen sie Disjunktion bzw. Konjunktion und in Mikroalgorithmen Addition bzw. Multiplikation. Zur Be-

---

[1] Wir interpretieren boolesche Algorithmen "ortssequentiell", d.h. bei n-maliger Iteration werden boolesche Funktionen n-mal aufgebaut, was durch iterative Schaltungen (Schaltketten) bewerkstelligt wird.

[2] Wir interpretieren Mikroalgorithmen "zeitsequentiell", d.h. bei n-maliger Iteration werden Mikrofunktionen n-mal ausgeführt, was durch sequentielle Schaltungen (Schaltwerke) bewerkstelligt wird.

zeichnung von Dualziffern und Dualzahlen legen wir Gl.(1.1) zugrunde. Eine Überschreitung des Zahlenbereichs wird durch den Wert 1 der Variablen ü angezeigt.

Zählen und Komplementieren

Die folgenden vier Algorithmen für Vorwärts- und Rückwärtszählen (1-Addition bzw. 1-Subtraktion) und Komplementieren sind boolesche Algorithmen. Gegeben ist eine ganze Dualzahl $X = x_{n-1}\, x_{n-2} \cdots x_0$ in 2-Komplement-Darstellung. Die Algorithmen berechnen die einzelnen Ergebnisziffern $Y = y_{n-1}\, y_{n-2} \cdots y_0$. Sie benutzen dazu die Hilfsgrößen (Überträge) $u_0, u_1, \ldots, u_{n-1}$ und zeigen eine Bereichsüberschreitung (Überlauf) durch ü an.

Algorithmus 1.1 Vorwärtszählen Y = X + 1

$u_0 = 1,$

<u>for</u> i <u>from</u> 0 <u>by</u> 1 <u>to</u> n - 1 <u>do</u>

$u_{i+1} = x_i \cdot u_i,$

$y_i = x_i \oplus u_i$

$ü = u_{n-1} \cdot \bar{x}_{n-1}$

Algorithmus 1.2 Rückwärtszählen Y = X - 1

$u_0 = 1,$

<u>for</u> i <u>from</u> 0 <u>by</u> 1 <u>to</u> n - 1 <u>do</u>

$u_{i+1} = \bar{x}_i \cdot u_i,$

$y_i = x_i \oplus u_i$

$ü = u_{n-1} \cdot x_{n-1}$

Algorithmus 1.3 Komplementieren Y = -X

$u_0 = 1,$

for i from 0 by 1 to n - 1 do

$u_{i+1} = \bar{x}_i \cdot u_i,$

$y_i = \bar{x}_i \oplus u_i$

$ü = u_{n-1} \cdot x_{n-1}$

Algorithmus 1.4 Komplementieren Y = -X

$u_0 = 0,$

for i from 0 by 1 to n - 1 do

$u_{i+1} = x_i + u_i,$

$y_i = x_i \oplus u_i$

$ü = \bar{u}_{n-1} \cdot x_{n-1}$

Die Algorithmen 1.3 und 1.4 entsprechen den bekannten Regeln zur Komplementbildung von Dualzahlen: Man vertausche 0 und 1 und addiere +1 bzw. man vertausche 0 und 1 ab der ersten von Null verschiedenen Ziffer. Sie unterscheiden sich wegen unterschiedlichen Anfangswerten $u_0 = 1$ bzw. $u_0 = 0$ voneinander. Auch für die Algorithmen 1.1 und 1.2 lassen sich boolesche Algorithmen mit dem Anfangswert $u_0 = 0$ angeben.

Addition und Subtraktion

Die folgenden beiden Algorithmen für Addition und Subtraktion sind boolesche Algorithmen. Gegeben sind zwei ganze Dualzahlen $X = x_{n-1}\, x_{n-2} \cdots x_0$ und $Y = y_{n-1}\, y_{n-2} \cdots y_0$ in 2-Komplement-Darstellung. Sie benutzen dazu die Hilfsgrößen (Überträge) $u_0, u_1, \ldots, u_{n-1}$ und zeigen eine Bereichsüberschreitung (Überlauf) durch ü an.

Algorithmus 1.5 Addition $Z = X + Y$[1]

$u_0 = 0,$

<u>for</u> i <u>from</u> 0 <u>by</u> 1 <u>to</u> n - 1 <u>do</u>

$u_{i+1} = x_i \cdot y_i \oplus (x_i \oplus y_i) \cdot u_i,$

$z_i = x_i \oplus y_i \oplus u_i$

$\ddot{u} = \bar{u}_{n-1} \cdot x_{n-1} \cdot y_{n-1} + u_{n-1} \cdot \bar{x}_{n-1} \cdot \bar{y}_{n-1}$

Algorithmus 1.6 Subtraktion $Z = X - Y$

$u_0 = 1,$

<u>for</u> i <u>from</u> 0 <u>by</u> 1 <u>to</u> n - 1 <u>do</u>

$u_{i+1} = x_i \cdot \bar{y}_i \oplus (x_i \oplus \bar{y}_i) \cdot u_i,$

$z_i = x_i \oplus \bar{y}_i \oplus u_i$

$\ddot{u} = \bar{u}_{n-1} \cdot x_{n-1} \cdot \bar{y}_{n-1} + u_{n-1} \cdot \bar{x}_{n-1} \cdot y_{n-1}$

Algorithmus 1.6 entspricht der Regel zur Subtraktion von 2-Komplement-Zahlen: Man bilde das Komplement von Y und addiere X, d.h. man vertausche 0 und 1 von Y und addiere X und 1 (vgl. Algorithmus 1.3).

Multiplikation

Die folgenden drei Algorithmen für die Multiplikation von 2-Komplement-Zahlen sind Mikroalgorithmen. Gegeben sind zwei ganze Dualzahlen $D = d_{n-1}\, d_{n-2} \ldots d_0$ (Multiplikand) und $Q = q_{n-1}\, q_{n-2} \ldots q_0$ (Multiplikator) in 2-Komplement-Darstellung. Die Algorithmen berechnen schrittweise das Ergebnis $R = r_{2n-1}\, r_{2n-2} \ldots r_0$. Können in einem Schritt auch mehrere Mikrooperationen gleichzeitig ausgeführt werden, so werden Kommas als Trennzeichen verwendet.

---

[1] Als ortssequentieller boolescher Algorithmus interpretiert, entsteht die aus Volladdiergliedern aufgebaute Additionsschaltkette.

Multiplikation durch Addieren. Die Multiplikation kann auf wiederholte Addition des Multiplikanden zurückgeführt werden. Dies läßt sich durch den folgenden Mikroalgorithmus beschreiben, wobei zu beachten ist, daß mit zunehmender Multiplikatorlänge immer mehr signifikante Ergebnisstellen entstehen.

Algorithmus 1.7 Multiplikation $R = D \cdot Q$

```
R := 0;
while Q ≠ 0 do
    if Q > 0 then  R := R + D,
                   Q := Q - 1
             else  R := R - D,
                   Q := Q + 1
```

Nebenbei sei bemerkt, daß mit R := X statt R := 0 als erster Anweisung und D = 1 ein Mikroalgorithmus für die Addition von X und Q entsteht.

Multiplikation durch Addieren und Schiften. Effizienter wird $D \cdot Q$ jedoch dadurch gebildet, daß der Multiplikand D mit jeder Multiplikatorziffer $q_i = 0$ bzw. $q_i = 1$ multipliziert und entsprechend der Stellenwertigkeit i der Ziffer gegenüber seiner ursprünglichen Position um i Stellen nach links verschoben akkumuliert wird. Bild 1.10 zeigt diesen Vorgang, veranschaulicht auf dem Zahlenring mit D als Einheit

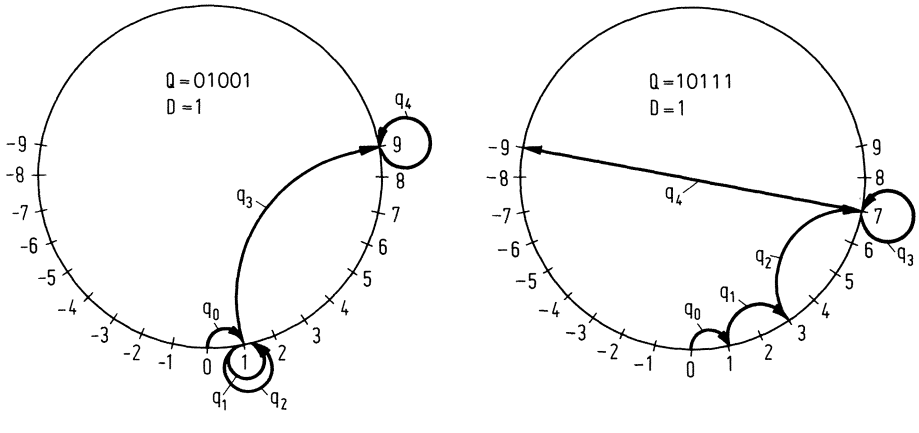


Bild 1.10 Multiplikation von $1 \cdot (+9)$ und $1 \cdot (-9)$ durch Addieren

und Q = 01001 bzw. Q = 10111. Dabei ist zu beachten, daß q_4 entsprechend Gl.(1.1) mit -2^4 gewichtet wird.

Statt D in jedem Schritt um eine Stelle nach links zu verschieben ($D := D \cdot 2$), kann genau so gut R jeweils um eine Stelle nach rechts verschoben werden ($R := R/2$). Zuvor muß D allerdings gegenüber R um n Stellen nach links verschoben werden ($D := D \cdot 2^n$), um in der richtigen Position akkumuliert zu werden. Je nachdem, ob die Multiplikatorziffer $q_i = 1$ oder 0 ist, wird D zu R addiert ($R := R + 1 \cdot D$) bzw. nicht zur R addiert ($R := R + 0 \cdot D$). R wird in jedem Fall um eine Position nach rechts verschoben. Bei der Verarbeitung der n-ten Multiplikatorziffer wird im Falle $q_{n-1} = 1$ D von R subtrahiert ($R := R - 1 \cdot D$). Wir geben den Algorithmus zur Multiplikation von D und Q mit Rechtsverschiebung von R an, da er meistens in dieser Form in Rechenwerken von Digitalrechnern verwendet wird.

Algorithmus 1.8 Multiplikation $R = D \cdot Q$

```
D := D · 2^n,
R := 0;
for i from 0 by 1 to n-2 do
    if q_i = 1 then R := R + D;
    R := R/2
if q_(n-1) = 1 then R := R - D;
R := R/2
```

Die Anzahl der Operationen in der do-Schleife bestimmt die Geschwindigkeit des Multiplikationsalgorithmus. Ist beispielsweise Q = -1, so werden insgesamt n Additionen und n Schifts ausgeführt. Bezüglich der Ausführungszeit der Multiplikation ist Algorithmus 1.8 ungeeignet, wenn der Multiplikator viele Einsen enthält, z.B. bei kleinen negativen Zahlen. Dieser Nachteil wird im folgenden Verfahren vermieden.

Multiplikation durch Addieren, Subtrahieren und Schiften. Man mustert Q von rechts nach links durch und addiert D auf R bei jedem Übergang von 0 auf 1 und subtrahiert D von R bei jedem Übergang von 1 auf 0. Bild 1.11 zeigt die Multiplikation $R = D \cdot Q$ mit Q = 01001 und Q = 10111, veranschaulicht durch den Zahlenring mit D als Einheit. Man sieht, wie sich Additionen und Subtraktionen von D fortwährend

abwechseln, wenn benachbarte Stellen ungleiche Werte haben. Der erste Schritt führt bei $q_0 = 1$ immer in den negativen Bereich, als ob eine Ziffer $q_{-1} = 0$ existieren würde.

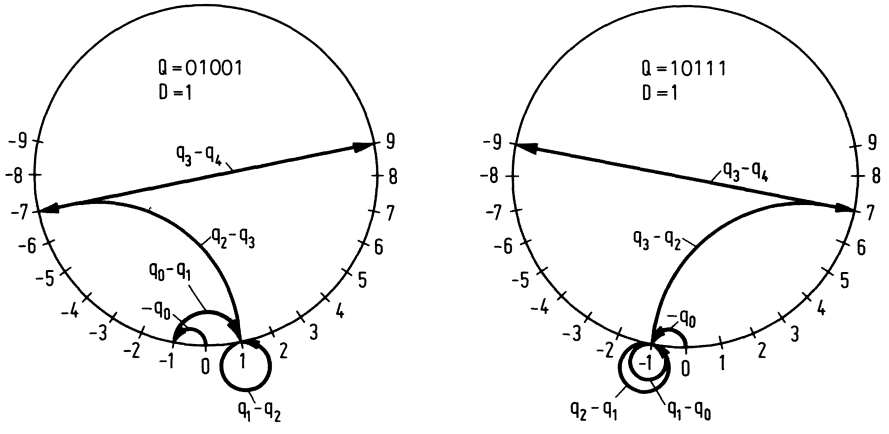


Bild 1.11 Multiplikation von $1 \cdot (+9)$ und $1 \cdot (-9)$ durch Addieren und Subtrahieren

Aus Bild 1.11 läßt sich ein Mikroalgorithmus für die Multiplikation von 2-Komplement-Zahlen entwickeln. Die Überlegungen sind ähnlich denjenigen, die zu Algorithmus 1.8 führten.

Algorithmus 1.9 Multiplikation $R = D \cdot Q$

```
D := D · 2^n,
R := 0;
if q_0 = 1 then  [ D := -D;
                   R := R + D ]
R := R/2;
for i from 1 by 1 to n - 1 do
    if q_i ≠ q_{i-1} then  [ D := -D;
                             R := R + D ]
    R := R/2
```

Verifizierung. Zur Verifizierung von Algorithmus 1.9 drücken wir zunächst die beiden if-Anweisungen durch Formeln aus, in die wir auch die Operation $R := R/2$ einbeziehen.

```
D := D · 2^n;
R_1 = (-q_0 · D)/2;

for i from 1 by 1 to n-1 do

    R_{i+1} = (R_i + (q_{i-1} - q_i) · D)/2
```

Wir lösen die do-Schleife entsprechend dem folgenden Schema auf, wobei wir $D = D \cdot 2^n$ setzen und $R_1 = -q_0 \cdot D/2$ als Anfangswert berücksichtigen.

$$
\begin{aligned}
R_n &= (R_{n-1} + (q_{n-2} - q_{n-1}) \cdot D \cdot 2^n)/2 \\
&\uparrow \\
R_{n-1} &= (R_{n-2} + (q_{n-3} - q_{n-2}) \cdot D \cdot 2^n)/2 \\
&\vdots \\
&\uparrow \\
R_2 &= (R_1 + (q_0 - q_1) \cdot D \cdot 2^n)/2 \\
&\uparrow \\
R_1 &= (-q_0 \cdot D \cdot 2^n)/2
\end{aligned}
$$

Durch wiederholtes Einsetzen entsteht

$$
\begin{aligned}
R_n &= (\ldots((-q_0 \cdot D \cdot 2^n/2 + (q_0 - q_1) \cdot D \cdot 2^n)/2 + \ldots + (q_{n-3} - q_{n-2}) \cdot \\
&\quad \cdot D \cdot 2^n)/2 + (q_{n-2} - q_{n-1}) \cdot D \cdot 2^n)/2 \\
&= D \cdot (-q_0 \cdot 2^0 + q_0 \cdot 2^1 - q_1 \cdot 2^1 + \ldots + q_{n-3} \cdot 2^{n-2} - q_{n-2} \cdot 2^{n-2} + \\
&\quad + q_{n-2} \cdot 2^{n-1} - q_{n-1} \cdot 2^{n-1} \\
&= D \cdot \left(\sum_{i=0}^{n-2} q_i \cdot 2^i - q_{n-1} \cdot 2^{n-1} \right) \\
&= D \cdot Q \qquad \text{vgl. Gl.(1.1)}
\end{aligned}
$$

Division

Die folgenden drei Algorithmen für die Division von 2-Komplement-Zahlen sind Mikroalgorithmen. Gegeben sind zwei ganze Dualzahlen $R = r_{2n-1}\, r_{2n-2} \ldots r_0$ (Dividend) und $D = d_{n-1}\, d_{n-2} \ldots d_0$ (Divisor) in 2-Komplement-Darstellung. Die Algorithmen berechnen die einzelnen Ergebnisziffern $Q = q_{n-1}\, q_{n-2} \ldots q_0$ (Quotient).

Die für Multiplikation und Division gemeinsam benutzten Bezeichnungen R, D und Q sollen darauf hinweisen, daß es sich um umkehrbare Operationen handelt. Dies kommt auch in den Verfahren und den Details der Algorithmen zum Ausdruck.

Division durch Subtrahieren. Bei der Lösung der Divisionsaufgabe Q = R/D wird gefragt, wieviel mal der Divisor D in den Dividenden R "hineinpaßt". Am einfachsten wird die Unbekannte Q dadurch ermittelt, daß D so oft von R subtrahiert wird, bis ein Rest bleibt, dessen Absolutwert kleiner als der Absolutwert von D ist. Die Anzahl der Subtraktionen ist gleich dem gesuchten Quotienten Q, d.h. D paßt Q-mal in R hinein. Der folgende Mikroalgorithmus beschreibt dieses Verfahren.

Algorithmus 1.10 Division Q = R/D

```
Q := 0;
while abs(R) ≥ abs(D) do
    if (R ≥ 0) ≡ (D ≥ 0) then  Q := Q + 1,
                               R := R - D
                  else         Q := Q - 1,
                               R := R + D
```

Division durch Subtrahieren und Rückstellen des Restes. Effizienter wird R/D dadurch gebildet, daß anstelle von D die Zweierpotenzen von D subtrahiert werden. Q entsteht ziffernweise, beginnend bei der höchsten Stelle. Bild 1.12 illustriert dies

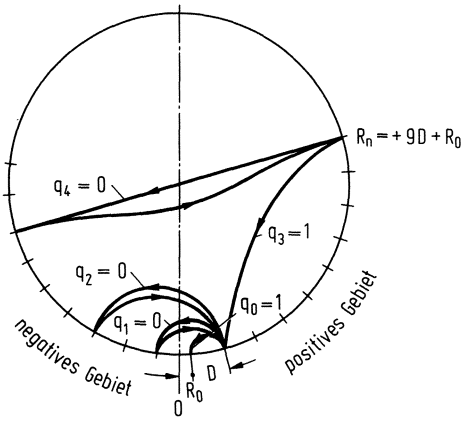


Bild 1.12 Illustration des Divisionsverfahrens mit Rückstellen des Rests

für $R_n = +Q \cdot D + R_0$, veranschaulicht auf dem Zahlenring mit D als Einheit. Dabei wird der Vergleich $abs(R) \geqslant abs(D) \cdot 2^i$ durch die Subtraktion $R := R - 2^i \cdot D$ und die Abfrage $R \geqslant 0$ ersetzt. Für $R < 0$ $(q_i := 0)$ muß die Subtraktion jedoch durch $R := R + 2^i \cdot D$ wieder rückgängig gemacht werden. Man spricht vom Divisionsverfahren mit Rückstellen des Zwischenrestes.

In Bild 1.12 ist die Subtraktion durch ← und die Addition durch → gekennzeichnet. Die einzelnen Ziffern q_i entstehen auf folgende Weise: Ausgehend von $R := R_n$

1. Versuch: $R - q_4 \cdot 2^4 D$ mit $q_4 = 1$ ergibt "Landung" im negativen Bereich, d.h. $q_4 := 0$ und erneut von R aus
2. Versuch: $R - q_3 \cdot 2^3 D$ mit $q_3 = 1$ ergibt "Landung" im positiven Bereich, d.h. $q_3 := 1$ "paßt"; von $R := R - 1 \cdot 2^3 D$ aus
3. Versuch: $R - q_2 \cdot 2^2 D$ mit $q_2 = 1$ ergibt "Landung" im negativen Bereich, d.h. $q_2 := 0$ und erneut von R aus
4. Versuch: $R - q_1 \cdot 2D$ mit $q_1 = 1$ ergibt "Landung" im negativen Bereich, d.h. $q_1 := 0$ und erneut von R aus
5. Versuch: $R - q_0 \cdot 1D$ mit $q_0 = 1$ ergibt "Landung" im positiven Bereich, d.h. $q_0 := 1$ "paßt".

Als Ergebnis entsteht $Q = q_4 q_3 q_2 q_1 q_0 = 01001 = 9$. Der Rest R_0 ist positiv.

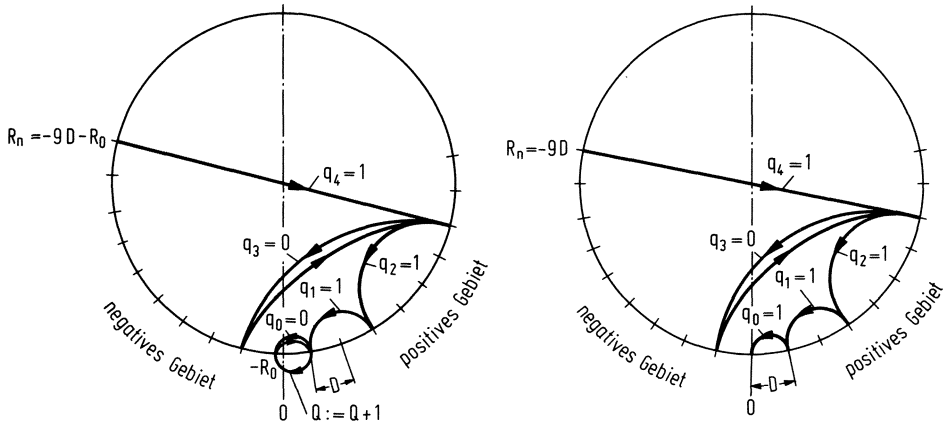


Bild 1.13 Restkorrektur bei der Division mit Rückstellen des Rests

Bild 1.13 illustriert dieses Verfahren für $R_n = -9 \cdot D - R_0$ und für $R_n = -9 \cdot D$. Im Falle $R_n = -9 \cdot D - R_0$ entsteht zunächst $Q = 10110 = -10$ mit positivem Rest. Dieses Ergebnis ist nicht falsch, da $-10 \cdot D$ plus positiver Rest R_n ergibt. Wird aber gefordert, daß der Rest das gleiche Vorzeichen wie R_n haben soll, so muß im

Falle $R_n > 0$ zusätzlich R um D vermindert werden (R := R - D) und dementsprechend Q um 1 erhöht werden (Q := Q + 1) (vgl. Algorithmus 1.10). Im Falle $R_n = -9 \cdot D$ entsteht Q = 10111 = -9 mit dem Rest 0. Dieses Ergebnis ist richtig, da 0 in der 2-Komplement-Darstellung per definitionem positiv ist.

Wir fassen zusammen: Eine Korrektur des Restes ist nicht notwendig, wenn $R_n \geqslant 0$ <u>oder</u> $R_0 = 0$ (vgl. Bild 1.12 und Bild 1.13 rechter Teil). Der Rest muß korrigiert werden, wenn das Gegenteil der Fall ist, d.h. wenn $R_n < 0$ <u>und</u> $R_0 \neq 0$ ist (vgl. Bild 1.13 linker Teil).

Das Divisionsverfahren mit Rückstellen des Zwischenrestes, soweit es bisher beschrieben wurde, gilt nur für positives D. Bei negativem D wird D zuerst komplementiert, dann wird weiter wie bei positivem D verfahren und anschließend Q komplementiert. Im folgenden Mikroalgorithmus ist das Komplementieren von D in die if-Anweisungen eingearbeitet.

<u>Algorithmus 1.11 Division $Q = (R_n - R_0)/D$ mit Rückstellen des Zwischenrestes</u>

```
D := D · 2^n,
R := R_n;

for i from n-1 by -1 to 0 do

    R := R · 2;
    if (R ≥ 0) ≡ (D ≥ 0) then R := R - D
                         else R := R + D;

    if (R ≥ 0) then  q_i := 1
               else  q_i := 0, if (D ≥ 0) then R := R + D
                                          else R := R - D

if (R_n < 0) und (R ≠ 0) then  if (D ≥ 0) then R := R - D
                               else R := R + D,
                               Q := Q + 1

if (D < 0) then Q := -Q
```

<u>Division durch Subtrahieren und Addieren ohne Rückstellen des Restes.</u> An Bild 1.12 sieht man, daß das Verfahren verkürzt werden kann, indem auf das Rückstellen des Zwischenrestes bei "Landung" im negativen Bereich verzichtet und statt dessen die

nächste Zweierpotenz $2^i \cdot D$ addiert wird. Man spricht vom Verfahren ohne Rückstellen des Zwischenrestes. Bild 1.14 veranschaulicht dieses Verfahren für $R_n = +9 \cdot D + R_0$ zum Vergleich mit Bild 1.12 und für $R_n = +8 \cdot D + R_0$.

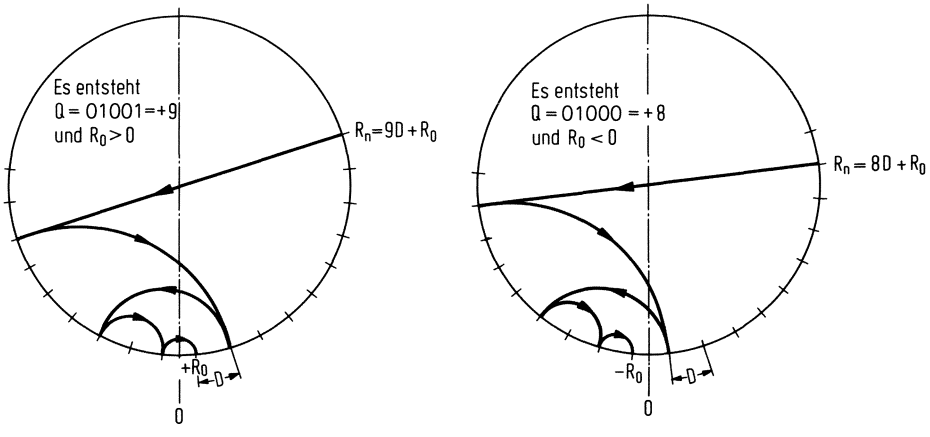


Bild 1.14 Division ohne Rückstellen des Rests bei ungerad- und geradzahligen Quotienten

Im Fall $R_n = 9 \cdot D + R_0$ entsteht $Q = 01001 = +9$ mit positivem Rest, im Falle $R_n = 8 \cdot D + R_0$ entsteht $Q = 01000 = +8$ mit negativem Rest. Es ist für dieses Verfahren charakteristisch, daß alle geraden Zahlen Q fälschlicherweise einen negativen Rest haben, der durch eine nachfolgende Addition von D korrigiert werden muß. Dadurch entsteht in jedem Fall ein positiver Rest, der gegebenenfalls erneut korrigiert werden muß (vgl. Restkorrektur in Algorithmus 1.11).

Zur Entwicklung eines Mikroalgorithmus für das Verfahren ohne Rückstellen des Zwischenrestes gehen wir von Algorithmus 1.11 aus. Die erste if-Anweisung wird umgeschrieben, so daß D komplementiert wird, wenn R und D gleiche Vorzeichen haben. Dadurch entfällt in der ersten if-Anweisung die Operation R := R - D. In der zweiten if-Anweisung entfällt das Rückstellen des Zwischenrestes. Durch das immer wiederkehrende Komplementieren von D bei gleichen Vorzeichen von R und D wird erreicht, daß man sich in der Darstellung des Ablaufes entsprechend Bild 1.14 immer in Richtung Null bewegt. Dies beinhaltet natürlich, daß auch zu Beginn des Algorithmus R und D komplementär zueinander sein müssen, und es tritt die Notwendigkeit, bei negativem D den Quotienten Q zu komplementieren, noch deutlicher hervor als in Algorithmus 1.11. Da sich hier außer dem Wert von R auch der Wert von D im Verlaufe des Algorithmus laufend ändert, muß neben dem ursprünglichen Vorzeichen von R auch das ursprüngliche Vorzeichen von D aufgehoben werden. Im folgenden Algorithmus bezeichnen wir die ursprünglichen Operanden mit R_n und D_n und die im Laufe des Algorithmus sich ändernden Operanden mit R und D.

Algorithmus 1.12 Division $Q = (R_n - R_0)/D_n$ ohne Rückstellen des Zwischenrestes

```
                    D := D_n · 2^n,
                    R := R_n;

                    for i from n-1 by -1 to 0 do

                        if (R ≥ 0) ≡ (D ≥ 0) then D := -D;
                        R := R · 2;
                        R := R + D;
                        if (R ≥ 0) then q_i := 1
                                   else q_i := 0

hier gilt
R_0 < 0, wenn R < 0  →  if (R ≥ 0) ≡ (D ≥ 0) then D := -D;

hier gilt
R_0 ≥ 0 immer        →  if (R < 0) then R := R + D;
                        if (R ≥ 0) ≡ (D ≥ 0) then D := -D;

hier gilt
R_0 ≤ 0, wenn R_n < 0
R_0 ≥ 0, wenn R_n ≥ 0 →  if (R_n < 0) und (R ≠ 0) then [R := R + D,
                                                        Q := Q + 1]
                        if (D_n < 0) then Q := -Q
```

Eine Überschreitung des Zahlenbereiches tritt im Falle

$$(R_n \geqslant 0) \equiv (D_n \geqslant 0) \not\equiv (Q \geqslant 0)$$

auf. Da $Q \geqslant 0$ bzw. $Q < 0$ in Algorithmus 1.12 bereits im ersten Schritt der do-Schleife ermittelt wird, kann der Ablauf bereits dort mit $(R \geqslant 0) \equiv (D \geqslant 0)$ abgebrochen und ü = 1 gesetzt werden.

Verifizierung. Zur Verifizierung betrachten wir nur die beiden ersten Abschnitte von Algorithmus 1.12. Das Ergebnis ist auch ohne die zweite Restkorrektur richtig, so daß sich die Einbeziehung des dritten Abschnittes erübrigt. Der vierte Abschnitt beschreibt lediglich die Komplementierung von Q bei negativem D_n.

Als erstes drücken wir Algorithmus 1.12 soweit wie möglich durch Formeln aus. Dann bilden wir zu diesem Algorithmus für die Division von 2-Komplement-Zahlen mit $R_0 = 0$ den inversen Algorithmus, wodurch ein Algorithmus für die Multiplikation von 2-Komplement-Zahlen entsteht. Dieser Multiplikationsalgorithmus ergibt

sich aus dem Divisionsalgorithmus, wenn man die einzelnen Schritte des Divisionsalgorithmus rückwärts verfolgt (vgl. Bild 1.14).

Division

```
D := D · 2^n;
if (R ≥ 0) then q_n = 1
           else q_n = 0;
for i from n-1 by -1 to 0 do
    R := R · 2 - (2 · q_{i+1} - 1) · D;
    if (R ≥ 0) then q_i = 1
               else q_i = 0
R := R - (q_0 - 1) · D
```

Multiplikation

```
D := D · 2^n;
R := 0;
q_n = 1 - q_{n-1};
R := R + (q_0 - 1) · D;
for i from 0 by 1 to n-1 do
    R := (R + (2q_{i+1} - 1) · D)/2
```

Für ungerade Zahlen Q bei 0 und für gerade Zahlen bei -D beginnend, wird R durch Additionen und Subtraktionen Schritt für Schritt aufgebaut. Ist beispielsweise Q = 01001 oder Q = 01000, so werden die Pfeile in Bild 1.14 umgekehrter Richtung durchlaufen. Dieses Multiplikationsverfahren hat allerdings keine praktische Bedeutung, da in jedem Schritt addiert bzw. subtrahiert und geschiftet wird und dementsprechend im Mittel mehr Schritte als in den Algorithmen 1.8 und 1.5 notwending sind. Wir lösen den inversen Divisionsalgorithmus wie bei der Verifizierung von Algorithmus 1.9 auf, indem wir die Rekursionsgleichungen ineinander einsetzen. Es entsteht

$$
\begin{aligned}
R_n &= ((\ldots((q_0 - 1) \cdot D \cdot 2^n + (2q_1 - 1) \cdot D \cdot 2^n)/2 + \ldots + \\
&\quad + (2q_{n-1} - 1) \cdot D \cdot 2^n)/2 - (2q_{n-1} - 1) \cdot D \cdot 2^n)/2 \\
&= D \cdot (q_0 \cdot 2^0 - 2^0 + q_1 \cdot 2^1 - 2^0 + q_2 \cdot 2^2 - 2^1 + \ldots + \\
&\quad + q_{n-1} \cdot 2^{n-1} - 2^{n-2} - q_{n-1} \cdot 2^n + 2^{n-1}) \\
&= D \cdot \left(\sum_{i=0}^{n-2} q_i \cdot 2^i - q_{n-1} \cdot 2^{n-1} \right) \\
&= D \cdot Q \qquad \text{vgl. Gl.(1.1)}
\end{aligned}
$$

Vergleicht man die Multiplikationsalgorithmen 1.8 und 1.9 und den inversen Divisionsalgorithmus 1.12 miteinander, so widerspiegeln sich in ihnen drei verschiedene Zahlensysteme. In Algorithmus 1.8 wird $2^i \cdot q_i$ akkumuliert, d.h. bei $q_i = +1$ addiert und bei $q_i = 0$ nicht addiert. In Algorithmus 1.9 wird $2^i \cdot (q_{i-1} - q_i)$ akkumuliert, d.h. bei $q_{i-1} - q_i = +1$ addiert, bei $q_{i-1} - q_i = -1$ subtrahiert und bei $q_{i-1} = q_i$ weder addiert noch subtrahiert. In Algorithmus 1.12 schließlich wird $2^i \cdot (2q_i - 1)$ akkumuliert, d.h. bei $2q_i - 1 = +1$ addiert und bei $2q_i - 1 = -1$ subtratrahiert. Das entspricht Zahlensystemen, deren Stellen 2^i mit Ziffern 0,+1 bzw. -1,0,+1 bzw. -1,+1 bewertet werden.

2. Rechnerhardware

Als Hardware[1] bezeichnet man die Gesamtheit der Baueinheiten (physical units) eines digitalen Systems. Hardware ist unveränderbar, solange die physikalische Konstruktion des Systems nicht verändert wird.

Die Rechnerhardware wird stärker von der Rechnertechnologie beeinflußt als von den Rechneranwendungen. So war es in der ersten Generation elektronischer Rechenmaschinen, die durch die Verwendung von Röhren als Schaltelemente gekennzeichnet ist, unmöglich, größere Rechner mit hoher Ausfallsicherheit zu bauen. Zum Beispiel trat bei dem von Eckert und Mauchly an der University of Pennsylvania gebauten Rechner ENIAC aufgrund der mittleren Lebensdauer von 1000 Stunden der verwendeten Elektronenröhen ca. alle 5 Minuten ein Hardwarefehler auf. Ein noch früheres Beispiel aus der Zeit der mechanischen Rechenmaschinen, die "Analytical Engine" von Babbage, zeigt, daß gewisse Rechnerkonzepte aufgrund fehlender technologischer Möglichkeiten erst gar nicht realisiert werden konnten.

Noch in der zweiten Generation elektronischer Rechenmaschinen, die durch die Verwendung von Transistoren als Schaltelemente gekennzeichnet ist, war es unwirtschaftlich, Rechner mit vielen Flipflopregistern zu bauen. Der wohl in dieser Zeit am weitesten verbreitete kommerzielle Rechner IBM 1401 arbeitete z.B. mit Index-"Registern", die im Kernspeicher untergebracht waren.

Aber auch in der dritten Generation elektronischer Rechenmaschinen, die durch die Verwendung integrierter Schaltkreise (integrated circuits) gekennzeichnet ist, war es noch nicht möglich, Flipflopspeicher höherer Kapazität zu bauen. Im größten Rechner der IBM-360er-Serie, dem IBM 360/85, hat man z.B. den Geschwindigkeitsunterschied zwischen Kernspeicher und Prozessor durch eine aufwendige Pufferorganisation auszugleichen versucht.

Erst in der nächsten Generation elektronischer Rechenmaschinen, die durch die Anwendung mittel- und hochintegrierter Technik (medium-scale integrated bzw. large-

---

[1] Aus dem Englischen: hardware - Eisenwaren.

scale integrated) gekennzeichnet sein wird, ist zu erwarten, daß sich der Geschwindigkeitsunterschied zwischen Speichern hoher Kapazität und schnellen Prozessoren erheblich reduziert. Auch ist es in dieser Technik eher vorstellbar, daß Assoziativspeicher in größerem Stil im Rechnerbau Eingang finden.

Der Einfluß der Technologie läßt sich vielleicht am besten durch den Vergleich von Rechnern verschiedener Generationen zeigen[1]. Der am MIT entwickelte WHIRLWIND (erste Generation) und die IBM 1800 (dritte Generation) haben sehr ähnliche Befehlssätze, wenn man von der Indizierung absieht, die bei der Planung des WHIRLWIND noch nicht bekannt war. Trotzdem sind ihre Prozessorstrukturen sehr unterschiedlich. Beim WHIRLWIND wurde die Datenübertragung zwischen den Datenendstationen und den peripheren Speichern über die Programmsteuerung des Zentralprozessors abgewickelt, wodurch die im Zentralprozessor vorhandenen Register und Verknüpfungsglieder aus Kostengründen mitbenutzt werden konnten. Bei der IBM 1800 war es dagegen ökonomischer, zusätzliche Funktionseinheiten für Spezialaufgaben vorzusehen. Sie besitzt mehrere Ein-Ausgabeprozessoren, die unabhängig vom Zentralprozessor arbeiten. Interessant ist es, den Einfluß der Technologie auf die Entwicklung der Kosten und der Leistung beider Rechner innerhalb eines Zeitraums von 10 Jahren zu betrachten: Das Kostenverhältnis zwischen WHIRLWIND und IBM 1800 beträgt 10 zu 1, das Verhältnis der Leistungsfähigkeiten 1 zu 5 und das Verhältnis der Maschinentaktzeiten ebenfalls 1 zu 5.

Nicht nur Kostenfragen und technologische Probleme der Elektronik, wie sie hier in Beispielen für die einzelnen Generationen elektronischer Rechenmaschinen dargestellt wurden, bestimmen die Hardware digitaler Rechenanlagen. Einen nicht zu unterschätzenden Einfluß üben auch mechanische und thermodynamische Probleme auf die Konstruktion von Rechenanlagen aus. Waren es z.B. früher die großen Abmessungen, so ist es heute die Wärmeentwicklung, die der Komplexität der Rechnerhardware eine Grenze setzt.

In diesem Kapitel beschreiben wir zunächst einen Digitalrechner, der sich stark an die von Burks, Goldstine und von Neumann vorgeschlagene Organisationsform anlehnt, und gehen dann mit der Beschreibung von Speichern und Prozessoren immer mehr ins Detail. Dabei werden zwei charakteristische Hardwarekonzepte gegenübergestellt. Das klassische Konzept mit Flipflop-Schaltwerken gestattet einen hohen Grad an Parallelität bei der Ausführung von Mikrooperationen. Dem steht das modernere Konzept der Rechneremulation gegenüber, das durch überwiegend serielle Ausführung von Mikro-

---

[1] Nach Bell, C.G. and Newell, A.: Computer Structures: Readings and Examples. New York: McGraw-Hill 1971, S. 55.

operationen gekennzeichnet ist. Emulation bedeutet, daß die Funktion des Rechners durch fest gespeicherte Mikroprogramme auf einem anderen, im allgemeinen kleineren Rechner nachgebildet wird. Der Vorteil der Emulation liegt in der Flexibilität, mit der Änderungen der Rechnerfunktion vorgenommen werden können, ihr Nachteil liegt in der langsameren Arbeitsgeschwindigkeit aufgrund der seriellen Ausführung der Mikrooperationen begründet. Die Technologie spielt bei der Emulation insofern eine Rolle, als die Zugriffszeit des Mikroprogrammspeichers erheblich unter der Zykluszeit des Programm- und Datenspeichers liegen muß.

Neben diesen beiden Hardwarekonzepten seien noch zwei weitere, gewissermaßen dazwischenliegende Konzepte erwähnt. Die Mikroprogrammsteuerung mit Schaltwerken, deren Flipflops durch Schaltermatrizen (in der zweiten Generation elektronischer Rechenmaschinen Diodenmatrizen) angesteuert werden, entspricht einerseits mit ihrer Minimierung des Aufwands an Schaltern der klassischen Schaltwerkstechnik, andererseits mit ihrer Auswechselbarkeit von fest gespeicherten Ablauffolgen der Emulationstechnik. Die Mikroprogrammsteuerung mit Hilfe von sog. mikroprogrammierbaren Rechnern (in der dritten Generation Spezialrechner mit langen Befehlswörtern) entspricht im Hinblick auf parallele Ausführung von Mikrooperationen mehr der Schaltwerkstechnik und im Hinblick auf die Ablaufsteuerung mehr der flexiblen Emulationstechnik.

In Abschnitt 2.1 wird ein einfacher Rechner in einer Form beschrieben, wie man sie in vielen Handbüchern (computer reference manuals) der Rechnerhersteller findet. Zunächst ist die Wirkung eines jeden Maschinenbefehls genau beschrieben, ohne auf die Details der einzelnen Abläufe einzugehen. Anschließend wird ein Ausschnitt aus der Funktionsbeschreibung des Rechners gegeben. Hier findet man die Abläufe einzelner Befehle im Detail. Die Darstellung der Abläufe durch Zustandsdiagramme ist nicht an eine bestimmte Realisierung gebunden. Sie kann vielmehr als Normdarstellung für parallel ablaufende Handlungen aufgefaßt werden.

In Abschnitt 2.2 werden kombinatorisch arbeitende Elemente digitaler Rechner (combinatorial circuits) eingeführt. Schaltnetze und Speicher werden in einer Form beschrieben, die sich für die strukturelle Beschreibung komplexerer digitaler Systeme besonders gut eignet. Die in diesem Abschnitt eingeführten Symbole für Codierer, Decodierer, Multiplexer, Demultiplexer, Randomspeicher und Assoziativspeicher werden in fast allen Abschnitten des Buches benutzt. Sie erlauben es unter anderem, zeitlich sequentielle Funktionen durch räumlich parallele Strukturen zu ersetzen.

In Abschnitt 2.3 werden sequentiell arbeitende Elemente digitaler Rechner (sequential machines) eingeführt. Schaltwerke und Prozessoren gliedern Digitalrechner in steuernde und gesteuerte Werke, die Befehle ausgeben bzw. ausführen. Diese Auftei-

lung gilt nicht nur für Rechner, sondern ganz allgemein für Systeme, die aus zwei Funktionseinheiten bestehen, von denen die eine die Reihenfolge von Aktionen steuert und die andere diese Aktionen auszuführen imstande ist. Die steuernde Einheit übermittelt der ausführenden Einheit Informationen über die Ausführung von Aktionen und die ausführende Einheit übermittelt der steuernden Einheit Informationen über ihren Zustand.

2.1 Ein einfacher Rechner

In Abschnitt 1.1 wurden im Rahmen eines Beispiels einige Befehle beschrieben, die durch einen Rechner verarbeitet werden können, ohne daß jedoch die für einen Universalrechner typischen Befehle behandelt wurden. Dies soll hier nachgeholt werden; und wir legen diesem Abschnitt einen einfachen Einadreßrechner mit einer Wortlänge von 16 Bits zugrunde.

Die Wortlänge bezieht sich auf den Speicher und die vom Rechnerbenutzer verwendbaren Register des Rechners. Sie ist ein wichtiges Merkmal eines Rechners, da sie einen Anhaltspunkt für seine Größe und seine Anwendung bietet. Wissenschaftliche Rechner besitzen wegen der notwendigen höheren Genauigkeit eine Wortlänge von 32 Bits und darüber, während technische Rechner wegen der niedrigeren Genauigkeit von Meßwerten meistens eine Wortlänge von 16 Bits haben. Kaufmännische Rechner weisen wegen der vorrangig benötigten Zeichenverarbeitung manchmal nur eine Wortlänge von 8 Bits auf, d.h. daß jedes gespeicherte Byte adressierbar ist. Kennzeichnend für die Wortlänge ist, daß sie im allgemeinen ein Vielfaches von 8 Bits beträgt, da auch mit technischen und wissenschaftlichen Rechnern Zeichen verarbeitet werden, z.B. beim Assemblieren von Programmen.

Ein Einadreßrechner

Bild 2.1 zeigt die Struktur des Rechners, die sich von Bild 1.4 hauptsächlich dadurch unterscheidet, daß das Rechenwerk neben dem Akkumulator A zwei weitere Register B und H enthält und das Ein-Ausgabewerk aus einem sog. Multiplexer-Demultiplexer besteht. Die beiden zusätzlichen Register des Rechenwerkes werden zum Multiplizieren und Dividieren benötigt, wobei die Register A und B zu einem Akkumulator doppelter Wortlänge zusammengefaßt werden. Der Multiplexer-Demultiplexer des Ein-Ausgabewerkes dient zum Anschluß mehrerer Ein-Ausgabegeräte an den Rechner, die über einen sog. Decodierer angewählt (adressiert) werden können. Wie in Bild 1.4 sind auch in Bild 2.1 die beiden Kreise für das Befehlholen (Bef.h.) und Operandholen (Opd.h.) über Sammelschienen, sog. Busse (buses), mit dem Speicher verbunden. Entsprechend der Funktion des Speichers sind ein Adreßbus, ein Lesebus und

ein (nicht gekennzeichneter) Schreibbus vorhanden. In Bild 2.1 nicht eingezeichnet sind die Tasten, Schalter und Anzeigelampen im Bedienungsfeld des Rechners, das einerseits dem Leitwerk und andererseits dem Ein-Ausgabewerk zugeordnet werden kann.

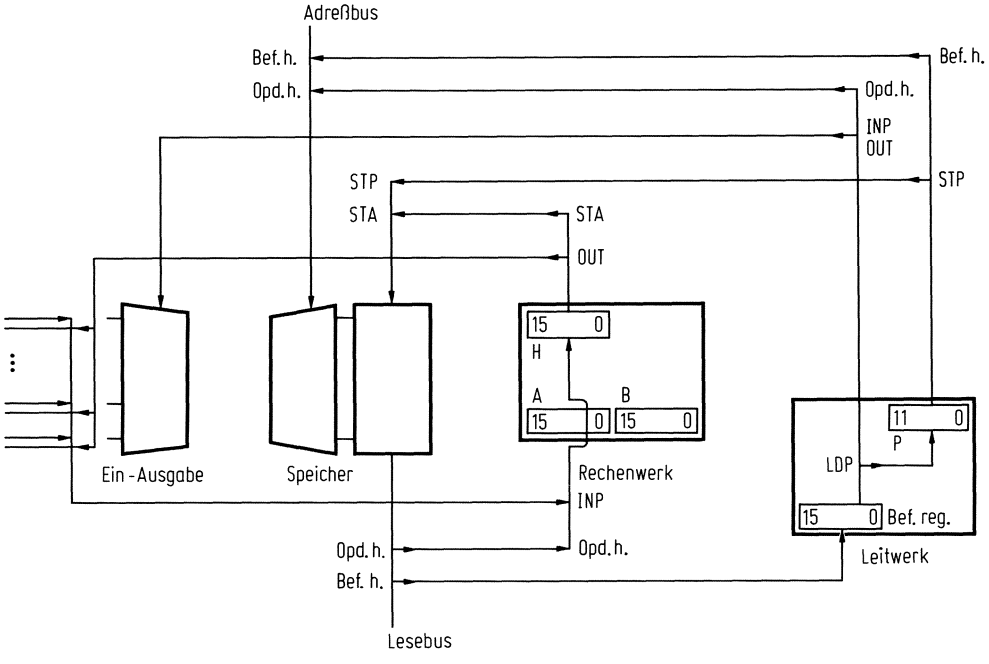


Bild 2.1 Struktur eines einfachen Einadreßrechners

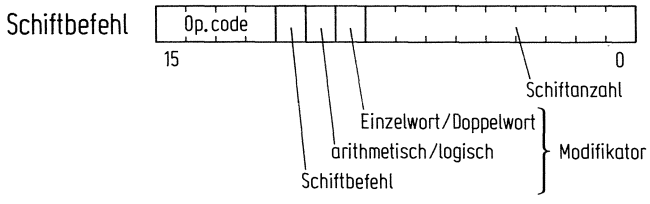
Befehls- und Operandenformat. Wie auch in Abschnitt 1.1 angenommen, sollen Befehle und Operanden genau die Länge eines Speicherwortes von 16 Bits haben.

Einadreßbefehl

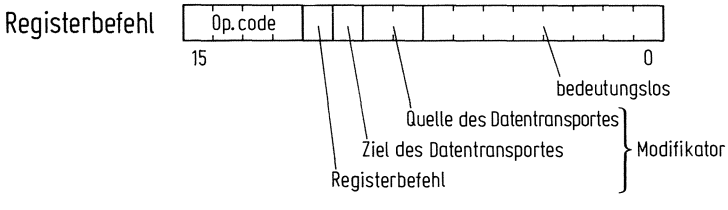
Op. code	Operandenadresse
15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0

Bei den Befehlen, die aus Operationscode und Operandenadresse bestehen, wird das Befehlswort in 4 Bits für den Operationscode und 12 Bits für die Adresse aufgeteilt.

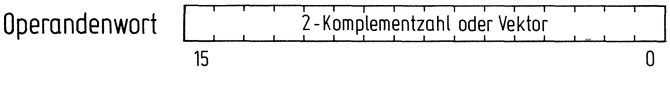
Bei den Schift- und Registerbefehlen können die 12 Bits des Adreßteils teilweise oder ganz zur Codierung des Befehls mitbenutzt werden, da kein Operand aus dem Speicher geholt zu werden braucht. Während die vorderen 4 Bits des Befehlswortes für alle Schift- und Registerbefehle denselben Code haben, werden die verschiedenen Schift- und Registerbefehle erst durch den erweiterten Befehlscode unterschieden, den wir Befehlsmodifikator nennen. Bit 11 enthält die Entscheidung, ob es sich um einen Schift- oder einen Registerbefehl handelt.



Die Schiftbefehle werden wie folgt unterschieden: Bit 10 gibt an, ob "arithmetisch" oder "logisch" geschiftet werden soll. Bit 9 gibt an, ob nur der Inhalt des Akkumulators A oder aber der Inhalt des erweiterten Akkumulators AB geschiftet werden soll. Bit 8 ist das Vorzeichen der Schiftanzahl und gibt an, ob nach links oder nach rechts geschiftet werden soll. Die verbleibenden Bits 7 bis 0 enthalten die Schiftanzahl als Konstante.



Die Registerbefehle werden folgendermaßen unterschieden: Bit 10 gibt das Zielregister des Datentransports an; das kann das A-Register oder das B-Register sein. Die Bits 9 und 8 bezeichnen das Quellregister des Datentransports; das kann das A-Register, das B-Register, ein gedachtes Register mit der Konstanten 0 oder ein gedachtes Register mit den negierten Bits von A als Inhalt sein.



Bei arithmetischen Operationen gibt Bit 15 das Vorzeichen an; die Zahlendarstellung erfolgt im 2-Komplement, d.h. negative und positive Zahlen gleichen Betrages ergänzen sich zu 2^{16}. Mit 16 Bits lassen sich $2^{16} = 65536$ verschiedene Dualzahlen bilden, das entspricht einem Zahlenbereich von -32768 bis +32767. Bei logischen Operationen besteht das Operandenwort aus 16 Bits, die als 0-1-Kombination des Wertes eines booleschen Vektors aufgefaßt werden.

Maschinenbefehle. Als Maschinenbefehle übernehmen wir die 5 Befehle mit den mnemonischen Codes LDA, STA, ADD, MUL und HLT aus Abschnitt 1.1 und führen eine Reihe weiterer Befehle ein. Bei allen Einadreßbefehlen sind 4096 Speicherzellen direkt adressierbar, M steht dabei stellvertretend für eine beliebige Speicheradresse oder eine Konstante. Die Register A, B und P (Befehlszähler) werden nur bei bestimmten Befehlen angesprochen, ihre Adresse ist implizit im Befehlscode enthalten.

Im Anschluß an die folgenden Befehlsbeschreibungen sind drei Beispiele angegeben, in denen die Wirkung der einzelnen Befehle im Zusammenhang mit anderen Befehlen illustriert wird.

LDA M (load A, Code: 8) lädt das Register A mit dem Inhalt der Speicherzelle M.

LDP M (load P, Code: 1) lädt das Register P mit der Adresse M. M ist die Adresse des Wortes, das als nächster Befehl interpretiert wird. Der Befehl LDP M bewirkt einen Programmsprung. In Verbindung mit dem CMP-Befehl wird er für Programmverzweigungen benutzt.

STA M (store A, Code: 4) speichert den Inhalt des Registers A in die Speicherzelle M.

STP M (store P, Code: 5) speichert den Inhalt des Registers P, d.h. die Adresse des auf STP M folgenden Befehls, in die Speicherzelle M. Der Befehl STP M wird benutzt, um bei einem Unterprogrammaufruf die Rücksprungadresse sicherzustellen.

ANA M (and, Code: 9) verknüpft die korrespondierenden Bits des Registers A und der Speicherzelle M durch die boolesche Operation "und" und bringt das Ergebnis in das Register A.

ORA M (or, Code: 10) verknüpft die korrespondierenden Bits des Registers A und der Speicherzelle M durch die boolesche Operation "oder" und bringt das Ergebnis in das Register A.

ADD M (add, Code: 11) addiert den Inhalt der Speicherzelle M auf den Inhalt des Akkumulators A und bringt das Ergebnis in den Akkumulator A. Wird bei der Addition der Zahlenbereich überschritten, so hält der Rechner an.

SUB M (subtract, Code: 12) subtrahiert den Inhalt der Speicherzelle M vom Inhalt des Akkumulators A und bringt das Ergebnis in den Akkumulator A. Wird bei der Subtraktion der Zahlenbereich überschritten, so hält der Rechner an.

MUL M (multiply, Code: 13) multipliziert den Inhalt der Speicherzelle M mit dem Inhalt des Akkumulators A und bringt das Ergebnis rechtsbündig in den auf doppelte Wortlänge erweiterten Akkumulator AB. Das Vorzeichen erscheint im Ergebnis in den beiden höchsten Akkumulatorstellen doppelt.

DIV M (divide, Code: 14) dividiert den Inhalt des auf doppelte Wortlänge erweiterten Akkumulators AB durch den Inhalt der Speicherzelle M und bringt das Ergebnis in den Akkumulator A. Wird bei der Division der Zahlenbereich überschritten, so hält der Rechner an. Der bei der Division entstehende Rest verbleibt im Register B.

CMP M (compare, Code: 15) vergleicht den Inhalt des Akkumulators A mit dem Inhalt der Speicherzelle M. Der auf CMP M folgende Befehl wird ausgeführt, wenn der Inhalt des Akkumulators A kleiner ist als der Inhalt der Speicherzelle M. Der auf

CMP M folgende Befehl wird übersprungen, wenn der Inhalt des Akkumulators A und der Inhalt der Speicherzelle M gleich sind. Die auf CMP M folgenden zwei Befehle werden übersprungen, wenn der Inhalt des Akkumulators größer ist als der Inhalt der Speicherzelle M. Der Befehl CMP M ermöglicht eine Programmverzweigung. Ihm folgen im allgemeinen ein oder zwei LDP-Befehle, wodurch ein bedingter Programmsprung entsteht.

INP M (input, Code: 6) gibt in den Akkumulator A den auf dem Eingabebus anstehenden Wert ein, sobald dieser vom Eingabegerät mit der Adresse M bereitgestellt worden ist.

OUT M (output, Code: 7) gibt den Inhalt des Akkumulators A auf den Ausgabebus aus, sobald das Ausgabegerät mit der Adresse M bereit ist, ihn zu übernehmen.

SAS M (shift arithmetic single, Code: 3, Modifikator: 0) schiftet den Inhalt des Akkumulators A um M Stellen nach links, wenn M ein positives oder kein Vorzeichen hat und deshalb Bit 8 im Befehlswort Null ist. Mit jedem Schiftschritt gelangt eine Null nach Bit 0 des Akkumulators. Hat M ein negatives Vorzeichen, so ist Bit 8 des Befehlswortes Eins und der Inhalt des Akkumulators A wird um M Stellen nach rechts geschiftet, wobei der Wert von Bit 15 des Akkumulators erhalten bleibt.

SAD M (shift arithmetic double, Code: 3, Modifikator: 1) schiftet den Inhalt des auf doppelte Wortlänge erweiterten Akkumulators AB um M Stellen nach links, wenn M ein positives oder kein Vorzeichen hat und deshalb Bit 8 im Befehlswort Null ist. Mit jedem Schiftschritt gelangt eine Null nach Bit 0 des B-Registers. Hat M ein negatives Vorzeichen, so ist Bit 8 des Befehlswortes Eins und der Inhalt des erweiterten Akkumulators AB wird um M Stellen nach rechts geschiftet, wobei der Wert von Bit 15 des A-Registers erhalten bleibt.

SLS M (shift logic single, Code: 3, Modifikator: 2) schiftet den Inhalt des Akkumulators A um M Stellen nach links, wenn M ein positives oder kein Vorzeichen hat und deshalb Bit 8 im Befehlswort Null ist. Bei jedem Schiftschritt gelangt dann der Wert von Bit 15 des A-Registers nach Bit 0 des A-Registers. Hat M ein negatives Vorzeichen, so ist Bit 8 des Befehlswortes Eins und der Inhalt des Akkumulators A wird um M Stellen nach rechts geschiftet, wobei jedesmal der Wert von Bit 0 des A-Registers nach Bit 15 des A-Registers gelangt.

SLD M (shift logic double, Code: 3, Modifikator: 4) schiftet den Inhalt des auf doppelte Wortlänge erweiterten Akkumulators AB um M Stellen nach links, wenn M ein positives oder kein Vorzeichen hat und deshalb Bit 8 im Befehlswort Null ist. Bei jedem Schiftschritt gelangt dann der Wert von Bit 15 des A-Registers nach Bit 0 des B-Registers. Hat M ein negatives Vorzeichen, so ist Bit 8 des Befehlswortes Eins und der Inhalt des erweiterten Akkumulators AB wird um M Stellen nach rechts geschiftet, wobei jedesmal der Wert von Bit 0 des B-Registers nach Bit 15 des A-Registers gelangt.

LAB (load A with B, Code: 3, Modifikator: 9) lädt das A-Register mit dem Inhalt des B-Registers.

LAZ (load A with zero, Code: 3, Modifkator: 10) lädt das A-Register mit Null.

LBA (load B with A, Code: 3, Modifikator: 12) lädt das B-Register mit dem Inhalt des A-Registers.

LBZ (load B with zero, Code: 3, Modifikator: 13) lädt das B-Register mit Null.

NGA (negate, Code: 3, Modifikator: 11) negiert jedes Bit des A-Registers.

NOP (no operation, Code: 0) besagt, daß der nächste Befehl aufgerufen wird. Im Adreßteil von NOP können Konstanten oder Adressen erscheinen, die auf die Ausführung des Befehls keine Wirkung haben.

HLT (halt, Code: 2) hält den Rechner an. Vom Bedienungsfeld aus kann er wieder in Betrieb gesetzt werden.

Beispiel 2.1 Einfache Befehlsfolgen

Die Befehle unseres Rechners umfassen mehr Operationen, als für seine Programmierung unbedingt notwendig sind. Auf bestimmte Befehle kann man verzichten und sie durch einen oder mehrere andere Befehle ersetzen. Zum Beispiel wird

NOP	ersetzt durch	SAS 0
LDA M	ersetzt durch	ANA M ORA M
SUB M	ersetzt durch	NGA ADD M NGA

■

Beispiel 2.2 Einfache Adreßmodifikationen

In unserem Rechner sind keine Hardware-Vorrichtungen zur sog. Modifikation von Adressen in den Befehlen vorgesehen. Im folgenden wird am Beispiel des LDA-Befehls gezeigt, wie die beiden wichtigsten Adreßmodifikationen, die Indizierung und die Substitution (vgl. Abschnitt 4.2: Indexadressierung und indirekte Adressierung), durch Befehlsfolgen nachgebildet werden können. Bei der Indizierung wird die Adresse M des LDA-Befehls um 1 erhöht, und bei der Substitution wird die Adresse des LDA-Befehls durch ihren Inhalt ersetzt. Es ist zu beachten, daß die Modifikationen in den angegebenen Programmstücken nicht rückgängig gemacht werden.

Substitution von M durch Inhalt

```
        LDA  BFHL
        ORA  M
        STA  BFHL
BFHL    LDA  0
```

Indizierung von M mit 1

```
        LDA  BFHL
        ADD  EINS
EINS    NOP  1
        STA  BFHL
BFHL    LDA  M
```

■

Beispiel 2.3 Einfacher Unterprogrammanschluß

Wird ein und dasselbe Programmstück oft an verschiedenen Stellen eines Programmes benutzt, so kann man es als sog. Unterprogramm schreiben (vgl. Abschnitt 4.3: Prozeduren). Am Beispiel eines Unterprogrammes zum Runden einer im erweiterten Akkumulator AB gespeicherten Zahl mit zwischen den Registern gedachtem Komma wird gezeigt, wie Unterprogrammaufruf und -rücksprung mit unserem Rechner organisiert werden können. Dabei wird die Rücksprungadresse durch Indizierung der Absprungadresse mit 1 gewonnen und der Rücksprung nach Substitution der Adresse im LDP-Befehl ausgeführt. Das Unterprogramm ist so geschrieben, daß es nach erfolgtem Aufruf wieder benutzbar ist.

Aufruf des Unterprogrammes RND

```
        STP  RND
        LDP  RND
```

Definition des Unterprogrammes RND

```
RND     NOP
        STA  TEMP
        LDA  BFHL
        ORA  RND
        ADD  EINS
        STA  RET
        LDA  TEMP
        CMP  NULL
        LDP  RET
        NOP
        LAZ
        SAD  1
        ADD  TEMP
RET     LDP  0
TEMP    RES  1
BFHL    LDP  0
NULL    DAT  0
EINS    DAT  1
```

■

Funktionsweise des Rechners

Die Beschreibung der Befehle liefert nur eine grobe Beschreibung der Funktion unseres Rechners, die für seine Programmierung wohl ausreicht. Bezieht man diese Beschreibung jedoch auf Bild 2.1, so kann man erkennen, daß der Ablauf eines jeden Befehls aus mehreren Schritten besteht, in denen bestimmte Operationen ausgeführt und bestimmte Bedingungen abgefragt werden. Die in einem Schritt ausgeführ-

ten Elementarhandlungen nennen wir Mikrobedingungen und Mikrooperationen und fassen sie zu einem Mikrobefehl zusammen. Eine Folge von Mikrobefehlen bildet ein Mikroprogramm, ähnlich wie eine Folge von Maschinenbefehlen ein Maschinenprogramm darstellt. Bei einem Mikroprogramm sind es die Mikrobefehle, und bei einem Programm sind es die Befehle, die als Elementarhandlungen angesehen werden. Wir haben also eine zweistufige Interpretationshierarchie vor uns.

Mikroprogrammsprache. Zur Beschreibung von Mikroprogrammen benutzen wir eine etwas speziellere Schreibweise als zur Beschreibung von Mikroalgorithmen und nennen sie Mikroprogrammsprache oder kurz Mikrosprache in Anlehnung an die Begriffe Assembler- oder Maschinensprache, wenngleich die Mikrosprache auch graphische Elemente enthält. In den Bezeichnungen der Operanden und Operationen sind wir bei Mikroprogrammen nicht so frei wie bei Mikroalgorithmen, da sich die Mikroprogramme auf ganz bestimmte Operanden und Operationen beziehen, z.B. auf die Speicher- und Registeroperanden bzw. die arithmetischen und logischen Operationen unseres Rechners. Bevor wir das Mikroprogramm unseres Rechners auszugsweise wiedergeben, beschreiben wir die wichtigsten Merkmale dieser Sprache.

Ein Speicher besteht aus einer matrixförmigen Anordnung von Speicherelementen, deren Zeilen die Zellen bilden, in denen die Speicheroperanden aufbewahrt werden. Um einen Operanden zu adressieren, muß neben dem Namen der Matrix auch die Nummer der Zeile angegeben werden, für den Operanden in der ersten Zelle der Matrix "Speicher" z.B. Speicher[0]. Ein Register besteht aus einer vektorförmigen Anordnung von Speicherelementen und bildet eine Zelle, in der ein Registeroperand aufbewahrt wird. Um den Operanden zu kennzeichnen, genügt es, den Namen des Vektors anzugeben. Sind jedoch nur bestimmte Elemente des Vektors gemeint, so müssen neben dem Namen des Vektors auch die Nummern der Spalten angegeben werden, für die Elemente 11 bis 0 des Vektors "Bef.reg." zum Beispiel Bef.reg. <11:0>. Speicher und Register sind durch Leitungsbündel miteinander verbunden, die aus einem "Vektor" von Einzelleitungen bestehen. Leitungsbündel und einzelne Leitungen werden wie Register bzw. Speicherelemente gekennzeichnet.

Es ist ein Unterschied, ob ein Operand in ein Register oder eine Speicherzelle geschrieben wird oder lediglich auf einem Leitungsbündel erscheint. Wir kennzeichnen dies durch unterschiedliche Transportsymbole. So drückt z.B. Adreßbus = P aus, daß der im Befehlszähler stehende Operand (der Befehlszählerstand) auf die Adreßleitung des Speichers geschaltet wird, während Bef.reg. := Lesebus bedeutet, daß der auf der Leseleitung des Speichers vorliegende Operand (ein Befehl) erst im nächsten Schritt im Befehlsregister erscheint. Der Speicherzyklus zum Holen eines Befehls kann also durch den Mikrobefehl

Adreßbus = P, Lesebus = Speicher [Adreßbus], Bef.reg. := Lesebus

oder kurz durch den Mikrobefehl

Bef.reg. := Speicher [P]

beschrieben werden.

Schließlich müssen wir noch Operationszeichen für arithmetische und logische Operationen vorsehen und die gleichzeitige (parallele) Ausführung von Mikrooperationen und Mikrobedingungen von der nacheinander erfolgenden (seriellen) Ausführung von Mikrobefehlen durch die Angabe von , bzw. ; trennen. Damit läßt sich z.B. der Ablauf des Befehls ADD mit den Definitionen

Code = Bef.reg.<15:12>, Adresse = Bef.reg.<11:0>

durch folgendes Mikroprogramm beschreiben.

Bef.reg. := Speicher [P], P := P + 1;

<u>if</u> Code = 11 <u>then</u> H := Speicher [Adresse];

A := A + H, <u>goto</u> ...

Um die bei Mikroprogrammen relativ oft vorkommenden parallelen Verzweigungen deutlicher ausdrücken zu können, wollen wir in die Sprache graphische Elemente aufnehmen. Jedem Mikrobefehl werden ein Kreis und ein durch strichpunktierte Linien gekennzeichneter Bereich zugeordnet (Zustand), in dem Mikrobedingungen und Mikrooperationen gleichzeitig ausgewertet bzw. ausgeführt werden. Weiter wird die Folge der Mikrobefehle durch Pfeile dargestellt, die durch links angeordnete Mikrobedingungen in eindeutiger Weise bezeichnet sind. Die zu einem Mikrobefehl gehörenden Mikrooperationen werden rechts aufgelistet, wobei den Mikrooperationen durch Doppelpunkte abgesetzte Mikrobedingungen für das Ausführen der Operation vorangestellt sein können.

Wir fassen zusammen: Speicher, Register (auch Leitungsbündel) und Speicherelemente (auch Einzelleitungen) werden als Variablen (Matrizen, Vektoren bzw. Skalare) aufgefaßt. Zur Kennzeichnung von Zeilen- und Spaltenindizes benutzen wir eckige ([]) bzw. spitze (<>) Klammern. Innerhalb solcher Klammern können mehrere Indizes erscheinen: Einzelindizes werden durch Kommas aufgelistet, Anfangs- und Endindizes werden durch einen Doppelpunkt getrennt. Die Durchschaltung von Operanden auf Leitungsbündel oder Leitungen wird durch das Gleichheitszeichen (=) beschrieben. Das Überschreiben von Operanden in Speicherzellen, Registern oder Speicherelementen wird durch das Ersetzungszeichen (:=) ausgedrückt. Als arithmetische Operationszeichen werden die üblichen Symbole verwendet. Anstelle symbolischer logischer Operationszeichen tritt unterstrichener Text. Das Gleichheitszeichen wird außerdem für Abfragen auf Gleichheit und für Definitionen benutzt. Die Folge der Mikrobefehle wird durch einen Graphen beschrieben; die durch := beschriebenen Mikrooperationen werden gleichzeitig mit den durch ← gekennzeichneten Übergängen zum nächsten Mikrobefehl ausgeführt.

Befehls- und Operandenabruf. Der Befehlsabruf ist für alle Befehle unseres Rechners gleich, weshalb ein Teil des Mikroprogramms von allen Befehlen gemeinsam durchlaufen wird. Bild 2.2 zeigt neben dem Befehlsabruf auch den Operandenabruf, den alle Befehle durchlaufen, die mit einem Speicheroperanden arbeiten. Die Befehlsausführung beschreiben wir in Bild 2.2 nur für die Befehle HLT, NOP und LDP und in den folgenden Bildern für die Befehle LDA, ANA, ORA, ADD, SUB und MUL.

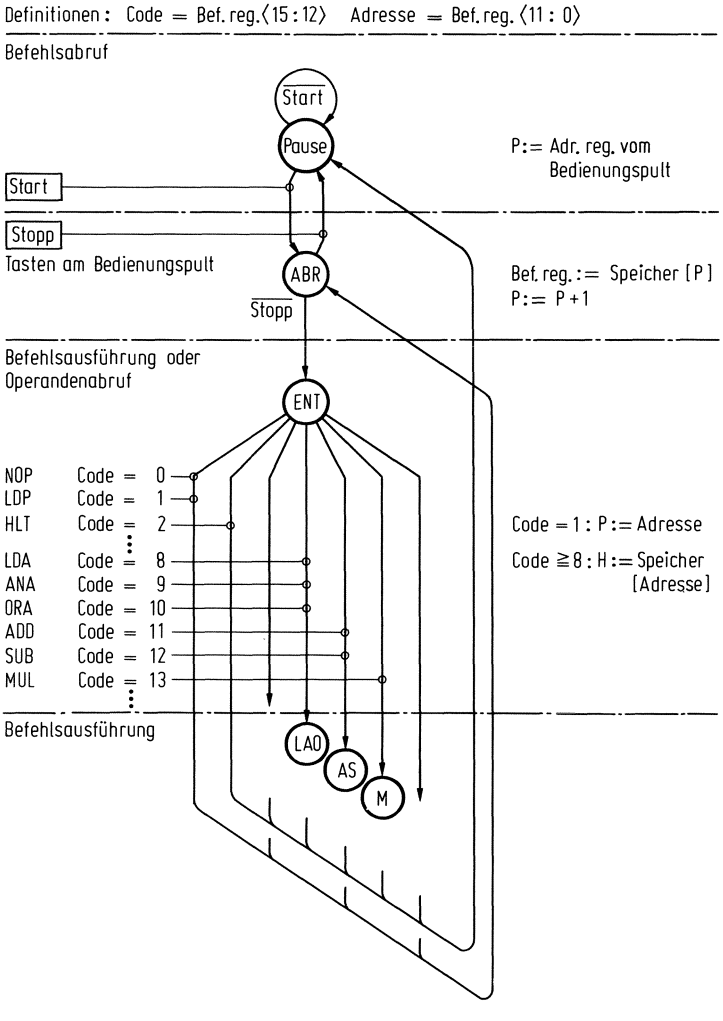


Bild 2.2 Mikroprogramm des Einadreßrechners

Mit dem Einschalten des Netzstromes wird der Rechner in einen Anfangszustand versetzt, den Zustand Pause. Vom Bedienungspult wird die dort eingestellte Startadresse in den Befehlszähler gebracht. Nach dem Drücken der Starttaste am Bedienungs-

pult gelangt der Rechner in den Zustand ABR (Abruf), in dem ein Befehl geholt und der Befehlszähler erhöht wird. Anschließend wird der mit ENT (Entschlüsselung) bezeichnete Mikrobefehl ausgeführt, der den Befehlscode abfragt, ausfürbare Befehle, wie z.B. den LDP-Befehl, ausführt oder für die logischen und arithmetischen Befehle den Operanden aus dem Speicher holt. Je nach Befehlscode wird mit unterschiedlichen Mikrobefehlen in unterschiedlichen Zuständen fortgefahren: für den Befehl HLT in Pause (Bild 2.2), für die Befehle NOP und LDP in ABR (Bild 2.2), für die Befehle LDA, ANA und ORA in LAO (Bild 2.3), für die Befehle ADD und SUB in AS (Bild 2.4) und für den Befehl MUL in M (Bild 2.5).

Logische Befehle. Bild 2.3 zeigt den Ausschnitt des Mikroprogramms für die logischen Befehle LDA, ANA und ORA. Je nach Befehlscode wird im Zustand LAO die entsprechende Mikrooperation ausgeführt, und anschließend wird nach ABR (Bild 2.2) gesprungen.

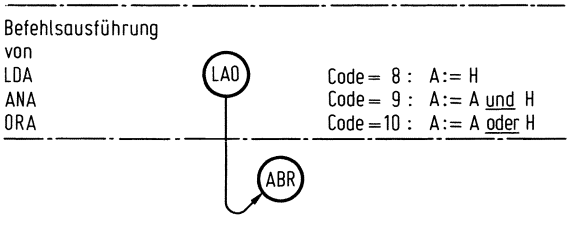


Bild 2.3 Ausschnitt aus dem Mikroprogramm des Einadreßrechners

Additions- und Subtraktionsbefehl. Bild 2.4 zeigt den Ausschnitt des Mikroprogramms für die arithmetischen Befehle ADD und SUB. Es folgt den Algorithmen 1.5 und 1.6 aus Abschnitt 1.3 für die Addition bzw. Subtraktion von 2-Komplement-Zahlen, die hier in einem Mikroprogramm vereinigt sind. Im Zustand AS wird der Speicherope-

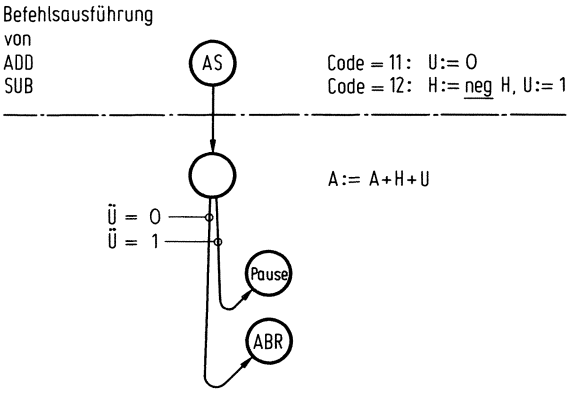


Bild 2.4 Ausschnitt aus dem Mikroprogramm des Einadreßrechners

rand gegebenenfalls komplementiert, anschließend mit dem Registeroperanden addiert, wobei gleichzeitig die Bereichsüberschreitung ausgewertet wird. Je nach Ergebnis dieser Auswertung wird nach ABR oder Pause (Bild 2.2) gesprungen.

Multiplikationsbefehl. Bild 2.5 zeigt den Ausschnitt des Mikroprogramms für den arithmetischen Befehl MUL. Es folgt im Prinzip dem Algorithmus 1.9 aus Abschnitt 1.5. Im Zustand M befindet sich der Multiplikator im A-Register und der Multiplikand im H-Register. Aufgrund des Wertes der niedrigsten Multiplikandenstelle wird eine Programmverzweigung vorgenommen und gleichzeitig unter anderem das A-Register gelöscht und der Multiplikand ins B-Register gebracht. In dem anschließenden Zyklus werden jeweils zwei nebeneinanderliegende Multiplikandenstellen abgefragt und die Teilprodukte akkumuliert. Wenn der Zykluszähler abgelaufen ist, steht in dem auf doppelte Wortlänge erweiterten Akkumulator AB das Ergebnis, und es wird nach ABR (Bild 2.2) gesprungen.

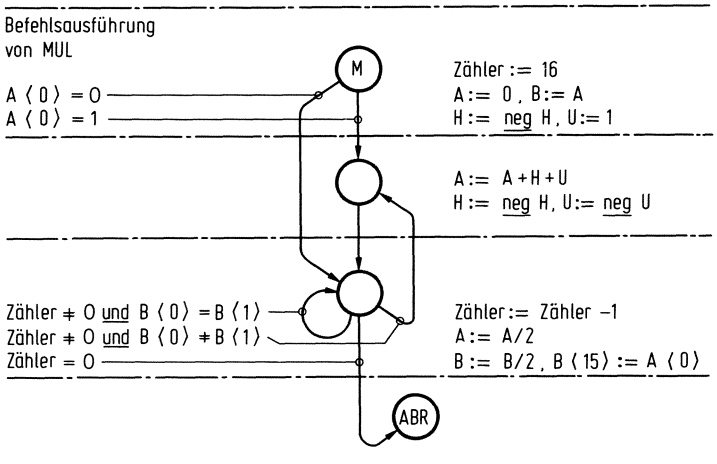


Bild 2.5 Ausschnitt aus dem Mikroprogramm des Einadreßrechners

2.2 Schaltnetze und Speicher

Komplexe digitale Systeme sind zusammengesetzt aus Schaltnetzen und Speichern, in Bild 2.1 z.B. aus Schaltnetzen zum Durchschalten von Leitungen oder zum Ausführen von arithmetischen Operationen und aus Speichern zum Aufbewahren von Befehlen und zum Aufbewahren und Verändern von Operanden.

Die Funktion solcher digitaler Systeme wird durch Mikroprogramme beschrieben, in Bild 2.2 z.B. in einer graphischen Mikrosprache, mit der ausgedrückt wird, wann

und unter welchen Bedingungen welche Operationen ausgeführt werden. Diese Sprache beschreibt die Funktion des digitalen Systems auf der Grundlage von Mikrobedingungen und Mikrooperationen und läßt dabei offen, wie diese Mikrofunktionen innerhalb eines Mikroschrittes ablaufen. Dies wiederum wird durch die "Sprache" der Booleschen Algebra beschrieben. Wir haben also Sprachen für mehrere Ebenen unterschiedlichen Detaillierungsgrades vor uns: bei einem Rechner die Maschinensprache, die Mikrosprache und die Boolesche Algebra. Die Boolesche Algebra beschreibt neben der Funktion auch die Struktur des Systems, so daß sie dementsprechend nicht weiter detailliert zu werden braucht.

In der Mikrosprache haben wir als Variablen Matrizen, Vektoren und Skalare benutzt. Auch in der Booleschen Algebra können mit Formeln, die Verknüpfungen von booleschen Vektoren und Matrizen beschreiben, komplexe Zusammenhänge gut lesbar dargestellt werden. Schreibt man die Vektor- und Matrixformeln in Form von Komponenten und rechnet sie entsprechend den Matrixoperationen aus, so entstehen Formeln, die funktionell im Sinne des Aussagenkalküls oder strukturell im Sinne von Schalterverbindungen interpretiert werden können. Sie enthalten damit alle Details, die in der Booleschen Algebra zur Darstellung logischer Zusammenhänge benötigt werden. Wir benutzen also Matrizen und Vektoren als geeignete Bindeglieder zwischen der Mikrosprache und der Booleschen Algebra, werden jedoch in der Booleschen Algebra die übliche mathematische Kurzschreibweise zur Darstellung von Vektoren, Matrizen und logischen Operationen verwenden, um die Formeln zur Beschreibung von Schaltnetzen und Speichern so übersichtlich wie möglich gestalten zu können.

Wir verabreden: Zeilenindizes werden als hochgestellte Indizes geschrieben (anstatt in eckige Klammern gesetzt). Spaltenindizes werden als tiefgestellte Indizes geschrieben (anstatt in spitze Klammern gesetzt). Wenn die Angabe der Indizes lediglich auf den Unterschied zwischen Skalaren, Spaltenvektoren, Zeilenvektoren und Matrizen hinweisen soll, werden sie weggelassen und statt dessen Buchstaben unterschiedlicher Art verwendet: für Skalare kleine, für Zeilenvektoren kleine fettgedruckte, für Spaltenvektoren große und für Matrizen große fettgedruckte Buchstaben. Skalare sind unzusammengesetzte Größen. Durch Nebeneinandersetzen und Untereinandersetzen von Skalaren entstehen Zeilenvektoren bzw. Spaltenvektoren als ranghöhere Größen. Durch Untereinandersetzen von Zeilenvektoren oder Nebeneinandersetzen von Spaltenvektoren entstehen Matrizen als wiederum ranghöhere Größen.

Wir geben im folgenden eine Zusammenstellung der wichtigsten Vektor- und Matrixoperationen, die durch ein Beispiel illustriert werden. Operationen für Größen eines bestimmten Ranges gelten - sinngemäß angewendet - auch für Größen niedrigeren Ranges. Bezüglich der Elementezahl der verknüpften Größen bestehen gewisse Bedingungen, die beachtet werden müssen.

Definition. Strukturoperationen

Wir kennzeichnen die Zusammenfassung von Elementen, die durch Nebeneinandersetzen oder durch Untereinandersetzen entstehen, durch eckige Klammern. Wir kennzeichnen den Übergang von Zeilenvektoren auf Spaltenvektoren und umgekehrt durch ein der Größe voran- und hochgestelltes T (Transposition).

Nebeneinandersetzen

$$\mathbf{C}_{1:(m+n)}^{1:t} = \left[\mathbf{A}_{1:m}^{1:t} \quad \mathbf{B}_{1:n}^{1:t}\right]$$

Untereinandersetzen

$$\mathbf{C}_{1:t}^{1:(m+n)} = \begin{bmatrix} \mathbf{A}_{1:t}^{1:m} \\ \mathbf{B}_{1:t}^{1:n} \end{bmatrix}$$

Transposition

$$\mathbf{C}_{1:m}^{1:n} = {}^{T}\mathbf{A}_{1:n}^{1:m} \quad \text{mit} \quad c_j^i = a_i^j$$

◆

Definition. Logische Operationen

Wir verwenden als variable Operationszeichen für die zweistelligen logischen Operationen Konjunktion ($\cdot$), Disjunktion ($+$), Äquivalenz ($\equiv$) und Antivalenz ($\not\equiv$) die beiden Symbole $\square$ und $\circ$. Die Negation wird durch Überstreichen des zum Klammerbereich gehörenden logischen Ausdruckes bezeichnet.

Durch die Vektorreduktion $\circ/\mathbf{a}$ bzw. $\circ/A$ wird - die Bezeichnung deutet es schon an - ein Vektor auf ein Skalar reduziert, indem sämtliche Elemente des Vektors durch $\circ$ verknüpft werden. Schreibt man $+$ für $\circ$, so entsteht mit $+/\mathbf{a}$ bzw. $+/A$ die Summe aller Elemente des Vektors. Die Reduktion erlaubt es, analoge Operationen auch für andere logische Verknüpfungen zu definieren.

Durch die Negation $\overline{\mathbf{A}}$ werden alle Elemente der Matrix $\mathbf{A}$ negiert. Durch die verallgemeinerte Addition $\mathbf{A} \circ \mathbf{B}$ werden Elemente mit gleichen Indizes verknüpft. Schreibt man $+$ für $\circ$, so entsteht mit $\mathbf{A} + \mathbf{B}$ die gewöhnliche Addition von zwei Matrizen. Die Verallgemeinerung der Addition erlaubt es, entsprechende Operationen für andere logische Verknüpfungen zu definieren.

Durch die verallgemeinerte Multiplikation $\mathbf{A} \square\circ \mathbf{B}$ werden die Elemente einer Zeile der Matrix $\mathbf{A}$ mit den Elementen einer Spalte der Matrix $\mathbf{B}$ durch $\circ$ verknüpft. Der

entstehende Vektor wird mit $\square$ zu einem Skalar reduziert. Auf diese Weise werden alle Zeilen von **A** mit allen Spalten von **B** verknüpft und reduziert. Aus einer jeden solchen Kombination ergibt sich ein Element der entstehenden Matrix. Schreibt man + für $\square$ und · für $\circ$, so entsteht mit **A** +· **B** die gewöhnliche Multiplikation von zwei Matrizen. Die verallgemeinerte Multiplikation erlaubt es, entsprechende Operationen für andere logische Verknüpfungen zu definieren.

Vektorreduktion

$$c = \circ/\mathbf{a}_{1:n} \quad \text{mit} \quad c = a_1 \circ a_2 \circ \ldots \circ a_n$$

$$c = \circ/A^{1:m} \quad \text{mit} \quad c = a^1 \circ a^2 \circ \ldots \circ a^m$$

Negation

$$\mathbf{C}^{1:m}_{1:n} = \overline{\mathbf{A}}^{1:m}_{1:n} \quad \text{mit} \quad c^i_j = \bar{a}^i_j$$

Verallgemeinerte Addition

$$\mathbf{C}^{1:m}_{1:n} = \mathbf{A}^{1:m}_{1:n} \circ \mathbf{B}^{1:m}_{1:n} \quad \text{mit} \quad c^i_j = a^i_j \circ b^i_j$$

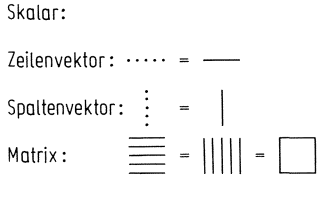
Verallgemeinerte Multiplikation

$$\mathbf{C}^{1:n}_{1:m} = \mathbf{A}^{1:n}_{1:t} \square\circ \mathbf{B}^{1:t}_{1:n} \quad \text{mit} \quad c^i_j = \square/(\mathbf{a}^i_{1:t} \circ B^{1:t}_i)$$

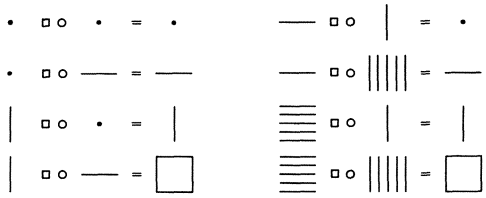
◆

Beispiel 2.4 Verallgemeinerte Multiplikation mit Vektoren und Matrizen

Wir stellen Vektoren und Matrizen durch Nebeneinander- und Untereinandersetzen ihrer jeweils rangtieferen Größen dar. Die entstehenden Strukturen entsprechen den üblichen Darstellungen von zu Vektoren und Matrizen zusammengefaßten Elementen. In Symbolen (· für Skalar, — und | für Zeilen- bzw. Spaltenvektor, □ für Matrix) kommt die Wirkung dieser Operationen besonders gut zum Ausdruck.



Wir benutzen diese Symbole, um verschiedene typische Anwendungen der verallgemeinerten Multiplikation zu zeigen. Die folgenden links dargestellten vier Fälle sind "Kombinationen": Jede Komponente des linken Operanden wird mit jeder Komponente des rechten Operanden kombiniert und durch ∘ verknüpft; das Zeichen ▫ ist bedeutungslos. Die folgenden rechts dargestellten vier Fälle sind "Reduktionen": Im Index korrespondierende Komponenten von Spalten des linken Operanden und von Zeilen des rechten Operanden werden durch ∘ verknüpft und mit ▫ reduziert.



Boolesche Funktionen in Matrixdarstellung

Die Beschreibung von Funktionen in der Sprache der Booleschen Algebra führt auf boolesche Funktionen; das sind Funktionen, deren Parameter boolesche Größen sind. Zum Beispiel bestehen die den Transport-Operationen "Adreßbus = P" und "Adreßbus = Adresse" entsprechenden Schaltfunktionen in Bild 2.2 aus den abhängigen booleschen Variablen $adreßbus_j$ entsprechend Adreßbus⟨j⟩ und den unabhängigen booleschen Variablen p_j entsprechend P⟨j⟩ und $adresse_j$ entsprechend Adresse⟨j⟩. In der Booleschen Algebra muß zwischen der Ausführung und der Nichtausführung dieser beiden Befehle unterschieden werden, was im Mikroprogramm Bild 2.2 dadurch zum Ausdruck gebracht wird, daß die Befehle aufgeschrieben bzw. nicht aufgeschrieben werden. Wir können dementsprechend auch die Befehle als boolesche Variablen auffassen, die 0 (Befehl nicht ausführen) oder 1 (Befehl ausführen) sein können. Wir kürzen

Adreßbus = P durch befh

Adreßbus = Adresse durch opdh

entsprechend den Bezeichnungen Bef.h. und Opd.h. in Bild 2.1 ab und beschreiben die Transportoperationen durch ein System boolescher Funktionen

$$adreßbus_0 = befh \cdot p_0 + opdh \cdot adresse_0$$
$$adreßbus_1 = befh \cdot p_1 + opdh \cdot adresse_1$$
$$\vdots$$
$$adreßbus_{11} = befh \cdot p_{11} + opdh \cdot adresse_{11}$$

das kürzer und damit übersichtlicher und aussagekräftiger durch Vektoren, Matrizen und darauf abgestimmte Operationen beschrieben werden kann.

$$\mathbf{adreßbus} = [\text{befh opdh}] +\cdot \begin{bmatrix} \mathbf{p} \\ \mathbf{adresse} \end{bmatrix}$$

Diese boolesche Funktion gibt die Struktur eines Schaltnetzes wieder, das Multiplexer genannt wird. Bevor wir solche speziellen Schaltnetze genauer besprechen, wollen wir zunächst auf boolesche Funktionen bzw. Schaltnetze allgemein eingehen.

Eine boolesche Funktion beschreibt die Zuordnung von Werten eines Vektors $\mathbf{x}$ (Eingangsgrößen) zu den Werten eines Skalars y oder eines Vektors $\mathbf{y}$ (Ausgangsgrößen) Ausgehend von der Wertetabelle läßt sich eine Funktion in ausgezeichneter disjunktiver Normalform als Formel darstellen.

$$y = f(x_0, x_1, \ldots, x_{n-1}) = \sum_{i=0}^{2^n-1} (m_i \cdot c_i). \tag{2.1}$$

Hierin bedeuten

m_i Minterm i, gebildet aus den Variablen $x_0, x_1, \ldots, x_{n-1}$

c_i Funktionswert i, zugehörig dem Minterm i

Entsprechend Gl.(2.1) ergibt sich die ausgezeichnete disjunktive Normalform einer booleschen Funktion aus der Summe (Disjunktion) all derjenigen Minterme m_i, deren zugehöriger Funktionswert $c_i = 1$ ist: Die Produkte (Konjunktionen) $m_i \cdot c_i$ vereinfachen sich für $c_i = 1$ zu m_i und für $c_i = 0$ zu 0, so daß die entsprechende Summe von Mintermen entsteht.

Der durch Gl.(2.1) beschriebene Zusammenhang läßt sich kürzer formulieren, wenn Vektoren, Matrizen und entsprechene Operationen benutzt werden: mit der Reduktion

$$y = f(\mathbf{x}) = +/({}^T\mathbf{m} \cdot C) \tag{2.2}$$

mit der verallgemeinerten Multiplikation

$$y = f(\mathbf{x}) = \mathbf{m} +\cdot C$$

oder genauer

$$y = f(\mathbf{x}) = \mathbf{m}(\mathbf{x}) +\cdot C. \tag{2.3}$$

Die Komponenten des Mintermvektors $\mathbf{m}$ sind Funktionen der zum Zeilenvektor $\mathbf{x}$ zusammengefaßten Variablen $x_0, x_1, \ldots, x_{n-1}$. C sind die zu einem Spaltenvektor zusammengefaßten Funktionswerte $c_0, c_1, \ldots, c_{2^n-1}$.

In den bisher aufgestellten Formeln müssen die Minterme in der Reihenfolge der natürlichen Zahlen angeordnet sein. Werden diese Zahlen als Dualzahlen aufgefaßt, so stimmt diese Folge mit der Reihenfolge der 0-1-Kombinationen in der Funktionstabelle überein. Dieser Nachteil eines starren und vollständigen Schemas in der Anordnung der Minterme wird vermieden, wenn die 0-1-Kombination zusammen mit **x** als Teil der Formel erscheinen und durch geeignet ausgeführte Operationen die Reihenfolge der Minterme "erzeugt" wird.[1]

$$y = f(\mathbf{x}) = (\mathbf{x} \cdot \equiv {}^{T}\mathbf{D}) +\cdot C \qquad (2.4)$$

Hierin bedeuten

- **x** Zeilenvektor der unabhängigen Variablen der Funktion
- **D** Matrix der 0-1-Kombination entsprechend der linken Seite der Wertetabelle
- C Spaltenvektor der Funktionswerte entsprechend der rechten Seite der Wertetabelle

Gl. (2.4) beschreibt die Darstellung der Funktion $y = f(\mathbf{x})$ durch eine Funktionsgleichung, in der die linke Seite der Wertetabelle durch eine Matrix **D** und die rechte Seite der Wertetabelle durch einen Spaltenvektor C dargestellt wird. Die Matrix **D**, deren Zeilen aus unterschiedlichen 0-1-Kombinationen bestehen, bildet den Definitionsbereich der Funktion. Der Spaltenvektor C, dessen Elemente 0 oder 1 sind, bildet den Wertebereich der Funktion. Die für jede Zeile $\mathbf{d}^i$ von **D** einerseits und jedes Element c^i von C andererseits getroffene Zuordnung zwischen einem Wert von **x** und einem Wert von y ist durch die Funktion f gegeben (Bild 2.6). Sie wird durch Gl. (2.4) beschrieben, in der die Größen **x**, **D** und C explizit erscheinen.

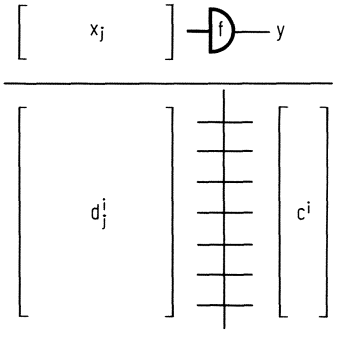


Bild 2.6 Boolesche Funktion als Wertetabelle (linke Seite: boolesche Matrix, rechte Seite: boolescher Vektor)

[1] Iverson, D.E.: A Programming Language. New York: John Wiley 1962, S. 251.

Wird diese Formel ausgerechnet, so entsteht die Funktion $y = f(\mathbf{x})$ in ausgezeichneter disjunktiver Normalform. Der erste Teil $(\mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D})$ ist ein Zeilenvektor, dessen Elemente Minterme sind, die in der durch **D** festgelegten Reihenfolge und Anzahl erscheinen. Durch die Multiplikation dieses Mintermvektors mit C werden dann diejenigen Minterme, deren korrespondierende Funktionswerte 1 sind, disjunktiv zusammengefaßt.

Zunächst wollen wir Gl.(2.4), die die Zuordnung f der Werte eines Vektors **x** zu einem Skalar y beschreibt, verallgemeinern zu einer Formel, die die Zuordnung $f_0, f_1, \ldots$ der Werte des Vektors **x** zu dem Feld von Skalaren $y_0, y_1, \ldots$ beschreibt, d.h. die Zuordnung **f** der Werte eines Vektors **x** zu einem Vektor **y** (Bild 2.7).

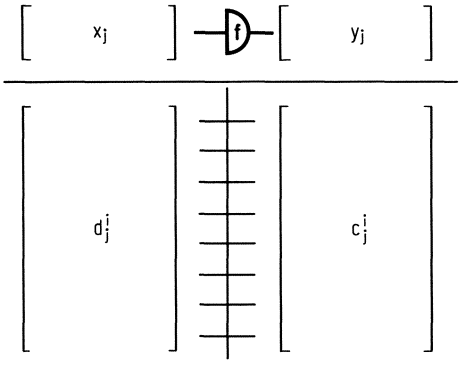


Bild 2.7 Boolesche Vektorfunktion als Wertetabelle (linke Seite: Decodiermatrix, rechte Seite: Codiermatrix)

Für ein Element y_j des Vektors **y** geschrieben, lautet Gl.(2.4)

$$y_j = f_j(\mathbf{x}) = (\mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D}) +\!\cdot\ C_j .$$

Für den Vektor **y** entsteht daraus die endgültige Formel zur Darstellung einer booleschen Funktion.[1]

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = (\mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D}) +\!\cdot\ \mathbf{C} \tag{2.5}$$

Mathematisch gesehen ist dies eine Abbildung einer Menge von 0-1-Kombinationen, die durch die Zeilen von **D** gegeben sind, auf eine Menge von 0-1-Kombinationen, die durch die Zeilen von **C** gegeben sind. Die Abbildung selbst ist definiert durch die Zuordnung **f** zwischen den Zeilen von **D** (das sind die Elemente des Definitionsbereichs) und den Zeilen von **C** (bestehend aus den Elementen des Wertebereichs).

Technisch gesehen beschreibt die Formel die Codeumsetzung eines Codes von Wörtern, die durch die Zeilen von **D** gegeben sind, in einen Code von Wörtern, die durch

[1] Giloi, W., Liebig, H.: Logischer Entwurf digitaler Systeme. Berlin: Springer 1973, S. 54.

die Zeilen von **C** gegeben sind. Die Umsetzung selbst ist definiert durch die Zuordnung der einzelnen Codewörter der Matrizen **D** und **C**.

Die Funktion $\mathbf{y} = \mathbf{f}(\mathbf{x})$ besteht aus Konjunktionen, die den durch **D** spezifizierten Mintermen entsprechen, die durch Disjunktionen entsprechend **C** zusammengefaßt sind. Da die Codewörter von **D** paarweise verschieden sind, schließen sich die Elemente des Mintermvektors gegenseitig aus. Für einen bestimmten Wert von **x** hat also nur ein Element des Mintermvektors den Wert 1, alle anderen haben den Wert 0. Faßt man die Zeilen zwischen **D** und **C** als Stellen eines Codewortes auf, so erscheint zwischen **D** und **C** als "Zwischencode" ein 1-aus-n-Code. Die Matrix **D** beschreibt die Decodierung von **x** in den 1-aus-n-Code, und **C** beschreibt die Codierung des 1-aus-n-Codes zu $\mathbf{y} = \mathbf{f}(\mathbf{x})$. **D** und **C** heißen Decodiermatrix bzw. Codiermatrix.

Codierer und Decodierer

Codierer und Decodierer werden in digitalen Systemen als Bausteine dort benutzt, wo es um die Umsetzung des 1-aus-n-Codes in einen beliebigen gleichlangen Binärcode (Codierung: Verschlüsselung eines Codes) und umgekehrt geht (Decodierung: Entschlüsselung des Codes). In Bild 2.1 werden Decodierer z.B. für die Auswahl von "einem aus n" Ein-Ausgabegeräten oder von "einer aus n" Speicherzellen benutzt.

Wir beschreiben Codierer und Decodierer wie auch später Multiplexer, Demultiplexer und Assoziierer durch ihre Funktion, durch ein Symbol, durch eine Formel und durch ein Schaltbild. Die Beschreibung der Funktion geben wir nicht in der Form der Mikrosprache, sondern in Form von Text. Später werden wir hauptsächlich die Symbole benutzen und nur dann durch Formeln oder Schaltbilder ergänzen, wenn notwendige Detailangaben unberücksichtigt bleiben würden.

Codierer. Bild 2.8 zeigt das Symbol des Codierers. Genau einer von n Eingängen hat den Wert 1, und die übrigen haben den Wert 0 (1-aus-n-Code). Dieser Eingangswert wird vom Codierer in einen Wert verschlüsselt (codiert), der am Ausgang er-

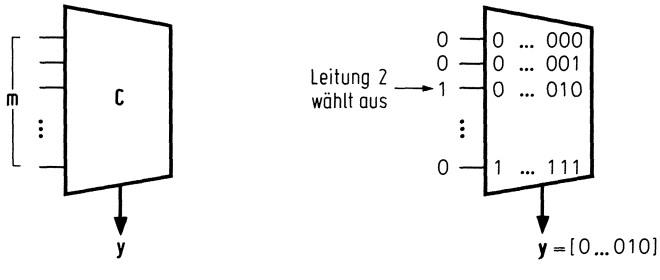


Bild 2.8 Codierer (links: Symbol, rechts: Wirkung)

scheint. Ist **C** = **Dual** die Matrix, deren Zeilen den Dualzahlen 0, 1, 2, ... entsprechen, so erscheint die Nummer der durch 1 markierten Eingangsleitung als Dualzahl am Ausgang. Entsprechendes gilt für jede andere Codierung von Dezimalzahlen.

Die Formel

$$\mathbf{y} = \mathbf{m} \cdot\!+ \mathbf{C} \quad \text{mit} \quad c_j^i \text{ als Konstanten} \tag{2.6}$$

beschreibt die Struktur, und Bild 2.9 zeigt den Aufbau des Codierers mit Verknüpfungsgliedern entsprechend der Darstellung von Gl. (2.6) in Komponentenform. Auffallend ist die Analogie der Struktur der Verbindungspunkte mit der Anordnung der Einsen in der Codiermatrix.

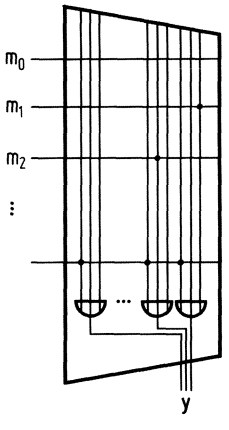


Bild 2.9 Schaltbild des Codierers

Decodierer. Bild 2.10 zeigt das Symbol des Decodierers. Der Eingangswert wird entschlüsselt (decodiert), so daß genau einer von n Ausgängen den Wert 1 hat, während die übrigen den Wert 0 haben (1-aus-n-Code). Ist **D** = **Dual**, so wird diejenige Ausgangsleitung durch 1 markiert, deren Nummer - als Dualzahl aufgefaßt - am Eingang anliegt. Entsprechendes gilt für jede andere Codierung von Dezimalzahlen.

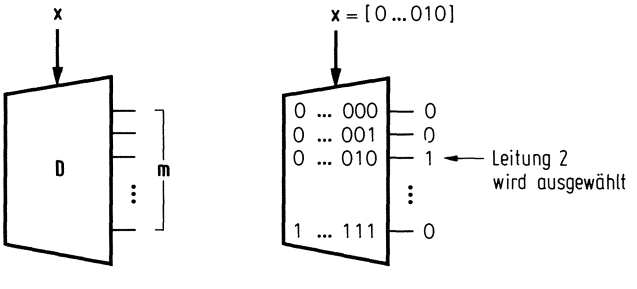


Bild 2.10 Decodierer (links: Symbol, rechts: Wirkung)

Die Formel

$$\mathbf{m} = \mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D} \quad \text{mit} \quad d_j^i \text{ als Konstanten} \tag{2.7}$$

beschreibt die Struktur, und Bild 2.11 zeigt den Aufbau des Decodierers mit Verknüpfungsgliedern entsprechend der Darstellung von Gl. (2.7) in Komponentenform. Auffallend ist die Analogie der Struktur der Verbindungspunkte mit der Anordnung der Nullen und Einsen in der Decodiermatrix.

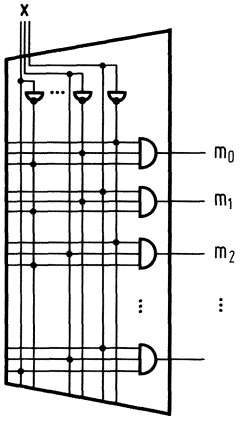


Bild 2.11 Schaltbild des Decodierers

Multiplexer, Demultiplexer und Assoziierer

Multiplexer und Demultiplexer werden in digitalen Systemen als Bausteine dann benutzt, wenn Leitungen oder Leitungsbündel zusammengefaßt bzw. aufgeteilt werden. In Bild 2.1 werden Multiplexer z.B. für das Durchschalten der Eingabeleitungen auf den Eingabebus oder der Befehls- oder Operandenadresse auf den Speichereingang benutzt.

Multiplexer, Demultiplexer und insbesondere Assoziierer werden vor allem in Speichern benötigt. Dort werden die Ausgänge der Speicherzellen von einem Multiplexer zu einem Speicherausgang zusammengefaßt, der Speichereingang wird durch Demultiplexer in die Eingänge der Speicherzellen aufgeteilt, und die am Eingang anliegende Adresse wird durch einen Assoziierer mit den gespeicherten Adressen verglichen.

Multiplexer. Bild 2.12 zeigt das Symbol des Multiplexers. Der Wert von **m** bestimmt, welches Leitungsbündel seine Eingangswerte an den Ausgang weitergibt. Vom Multiplexer wird derjenige Eingangswert auf den Ausgang durchgeschaltet, der durch eine 1 markiert ist.

Die Formel

$$\mathbf{y} = \mathbf{m} +\!\cdot \mathbf{C} \quad \text{mit} \quad c_j^i \text{ als Variablen} \tag{2.8}$$

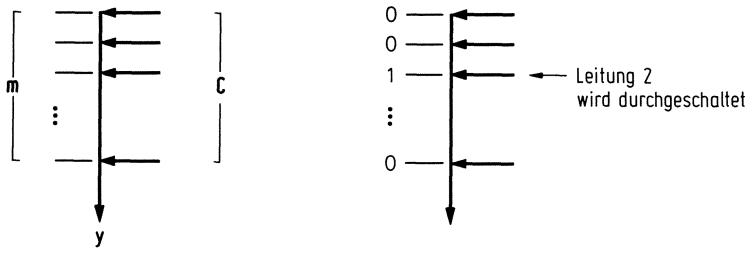


Bild 2.12 Multiplexer (links: Symbol, rechts: Wirkung)

beschreibt die Struktur, und Bild 2.13 zeigt den Aufbau des Multiplexers mit Verknüpfungsgliedern entsprechend der Darstellung von Gl. (2.8) in Komponentenform. Multiplexer werden vielfach mit Decodierern zusammengeschaltet. Man nennt dann die Werte des Decodierereingangs Multiplexeradressen. Sie werden durch den Decodierer entschlüsselt und wählen einen Multiplexereingang aus.

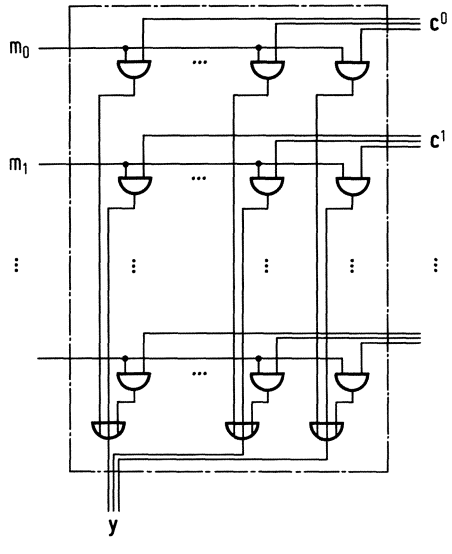


Bild 2.13 Schaltbild des Multiplexers

Beim Multiplexer ist es sinnwoll, u.a. auch die Kombination **m** = [0...00] zuzulassen; bei dieser Kombination wird kein Eingangswert auf den Ausgang durchgeschaltet. Hat der Multiplexer nur einen einzigen Eingang, so können die Oder-Glieder entfallen, und es entsteht eine einfache Torschaltung, die durch folgende Formel beschrieben wird.

$$\mathbf{y} = m +\cdot \mathbf{c} \quad \text{mit} \quad m = 0,1 \tag{2.9}$$

<u>Demultiplexer.</u> Bild 2.14 zeigt das Symbol des Demultiplexers. Der Wert von **m** bestimmt, welches Leitungsbündel am Ausgang die Eingangswerte weitergibt. Vom De-

multiplexer wird der Eingangswert auf denjenigen Ausgang durchgeschaltet, der durch eine 1 markiert ist.

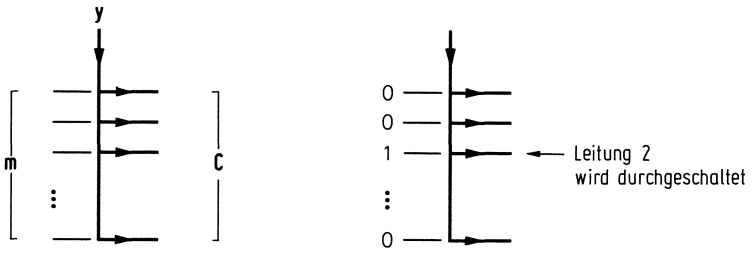


Bild 2.14 Demultiplexer (links: Symbol, rechts: Wirkung)

Die Formel

$$\mathbf{C} = {}^{T}\mathbf{m} +\cdot \mathbf{y} \tag{2.10}$$

beschreibt die Struktur, und Bild 2.15 zeigt den Aufbau des Demultiplexers mit Verknüpfungsgliedern entsprechend der Darstellung von Gl. (2.10) in Komponentenform. Wie Multiplexer werden Demultiplexer vielfach mit Decodierern zusammengeschaltet. Der Wert der Adresse am Decodierereingang wird entschlüsselt und wählt einen Demultiplexerausgang aus.

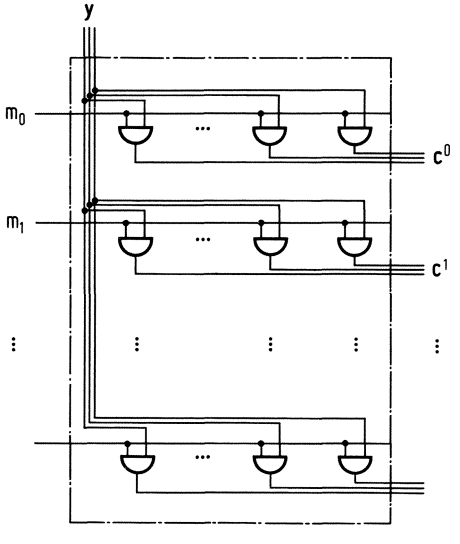


Bild 2.15 Schaltbild des Demultiplexers

Beim Demultiplexer ist es sinnvoll, u.a. auch die Kombination $\mathbf{m} = [1...11]$ zuzulassen. Bei dieser Kombination wird der Eingangswert auf alle Ausgänge durchgeschaltet. Ändert sich dieser Wert nie, so können die Und-Glieder durch starre Verbindungen ersetzt werden, und es entsteht eine einfache Verteilerschaltung. Im Sym-

bol deuten wir das an, indem die Verbindungen des Eingangs mit den Ausgängen durch Punkte dargestellt werden. Die Formel des Verteilers lautet

$$\mathbf{C} = {}^{T}\mathbf{i} +\cdot \mathbf{y} \quad \text{mit} \quad \mathbf{i} = [1 \ldots 11] \tag{2.11}$$

Assoziierer. Bild 2.16 zeigt das Symbol des Assoziierers. Der am Eingang **x** anliegende Wert wird mit allen Eingangswerten $\mathbf{c}^0, \mathbf{c}^1, \ldots$ verglichen, und im Falle der Übereinstimmung erhalten die zugehörigen (assoziierten) Ausgänge $m_0, m_1, \ldots$ den Wert 1, während die übrigen den Wert 0 annehmen. Ein weiterer Ausgang a zeigt an, ob mindestens eine (a=1) oder keine Übereinstimmung (a=0) gefunden wurde. Sind die Werte der Zeilenvektoren von **C** paarweise verschieden, so entsteht bei erfolgreicher Assoziation am Ausgang **m** des Assoziierers der 1-aus-n-Code.

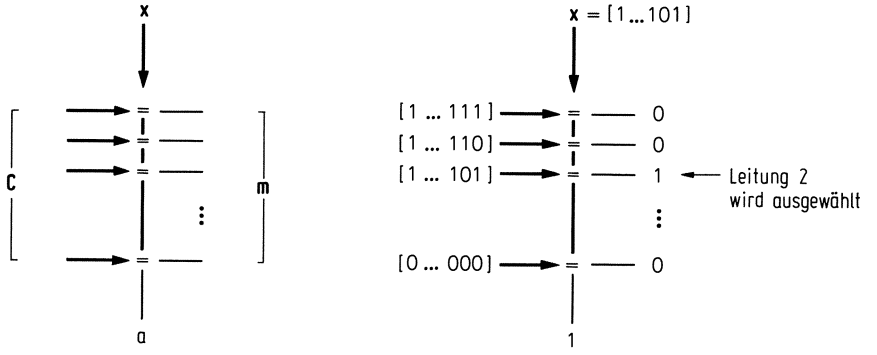


Bild 2.16 Assoziierer (links: Symbol, rechts: Wirkung)

Die Formeln

$$\left.\begin{array}{l} \mathbf{m} = \mathbf{x} \cdot\equiv {}^{T}\mathbf{D} \quad \text{mit} \quad d_j^i \text{ als Variablen} \\ a = +/\mathbf{m} \end{array}\right] \tag{2.12}$$

beschreiben die Struktur, und Bild 2.17 zeigt den Aufbau des Assoziierers mit Verknüpfungsgliedern entsprechend der Darstellung von Gl. (2.12) in Komponentenform. Man erkennt in jeder Zeile die Äquivalenz- und Und-Glieder der Vergleichsschaltnetze.

Sind die Werte der Zeilenvektoren von **C** nicht paarweise verschieden, so haben wir am Assoziiererausgang bei m assoziierten von n vorhandenen Ausgängen einen m-aus-n-Code vor uns. Soll ein Assoziierer in diesem Fall einen Multiplexer oder einen Demultiplexer steuern, so muß der m-aus-n-Code in den 1-aus-n-Code umgeformt werden. Dies kann durch ein zwischengeschaltetes Schaltnetz erfolgen, das aus den m

Markierungen diejenigen mit der niedrigsten Position auswählt. Schaltnetze dieser Art sind unter der Bezeichnung Prioritätenschaltnetze bekannt und werden überall dort verwendet, wo unter mehreren gleichzeitigen "Anmeldungen" diejenige mit der

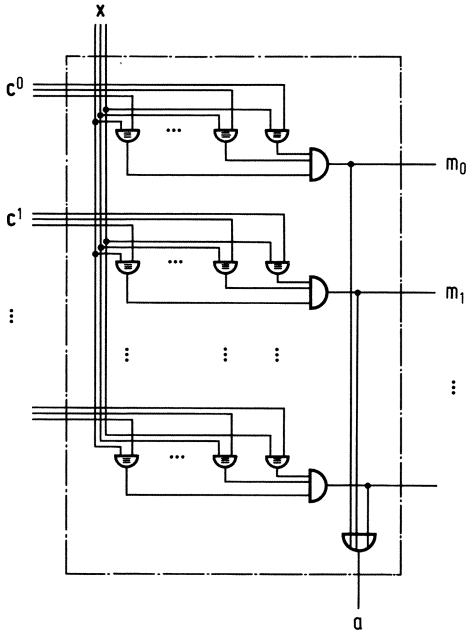


Bild 2.17 Schaltbild des Assoziierers

höchsten "Priorität" ausgewählt wird. In unserem Zusammenhang wird demjenigen Ausgang höchste Priorität zugeordnet, der räumlich zuoberst angeordnet ist. Wir geben die Funktionsgleichungen dieses Schaltnetzes in Form einer Schaltkette zusammen mit den Gleichungen für den Assoziierer an.

$$\left.\begin{array}{ll} \text{Assoziierer:} & \mathbf{k} = \mathbf{x} \cdot\equiv^{T}\mathbf{D} \\ & a = +/\mathbf{k} \\ \text{Prioritätenschaltnetz:} & m_0 = k_0;\ u_1 = k_0 \\ & m_i = u_i \cdot k_i;\ u_{i+1} = u_1 \cdot \bar{k}_1 \ \text{für}\ i = 1,2,3,\ldots,n-1 \end{array}\right] \qquad (2.13)$$

Festspeicher, Randomspeicher und Assoziativspeicher

Nur bei sehr kleinen Speichern mit wenigen Speicherzellen können alle Wörter gleichzeitig verändert oder abgefragt werden. In Bild 2.1 beispielsweise besteht der "Speicher" im Rechenwerk aus den Registern A, B und H, die voneinander völlig unabhängig sind. Bei Speichern größerer Kapazität kann jedoch immer nur ein Wort durch einen Schreib- oder Lesevorgang verändert bzw. abgefragt werden. Für alle Speicherzellen werden gemeinsame Schreib- und Leseleitungen benutzt, mit denen das ausge-

wählte Wort verbunden wird. Die Auswahl der Wörter erfolgt dadurch, daß zu jedem Wort eine Adresse gespeichert ist, die die Position des Wortes angibt. Dementsprechend besteht ein Speicher aus den beiden Teilen Adreßspeicher und Wortspeicher. Je nachdem ob der Adreßspeicher und der Wortspeicher aus Zellen unveränderlichen Inhalts bestehen, spricht man von Festspeichern, Randomspeichern oder Assoziativspeichern.

Festspeicher. Beim Festspeicher sind der Inhalt des Adreßspeichers und der Inhalt des Wortspeichers unveränderlich. Dementsprechend existieren weder für den Adreß- noch für den Wortspeicher Schreibvorrichtungen. Bild 2.18 zeigt das Symbol des Festspeichers. Er besteht aus einem Decodierer **D** als Adreßspeicher und einem Codierer **C** als Wortspeicher. Der Decodierer entschlüsselt die am Eingang **x** anliegende Adresse (Adressieren), und am Ausgang **y** des Codierers erscheint das zu dieser Adresse gehörende Wort (Lesen). Auch die Begriffe Zuordner und Code-

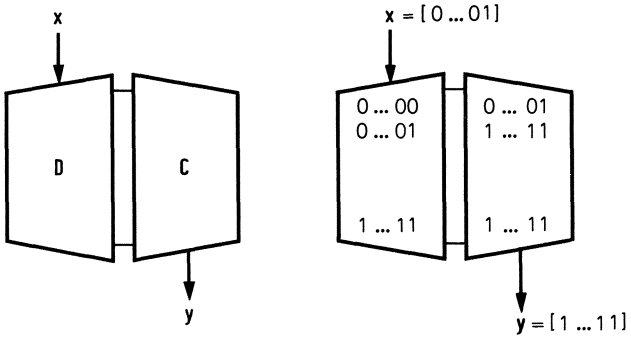


Bild 2.18 Festspeicher (links: Symbol, rechts: Lesevorgang)

umsetzer werden für die in Bild 2.18 gezeigte Zusammenschaltung aus Decodierer und Codierer benutzt (jedem Codewort von **D** ist ein Codewort von **C** zugeordnet; der im Decodierer fest gespeicherte Code wird in den im Codierer fest gespeicherten Code umgesetzt).

Die folgenden Formeln beschreiben die Struktur des Festspeichers. Sie sind mit Gln. (2.6) und (2.7) und auch mit Gl. (2.5) identisch. Somit kann ein Festspeicher als ein Schaltnetz angesehen werden, das entsprechend einer booleschen Vektorfunktion in ausgezeichneter disjunktiver Normalform aufgebaut ist.

$$\left.\begin{array}{ll} \text{Decodieren:} & \mathbf{m} = \mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D} \\ \text{Lesen:} & \mathbf{y} = \mathbf{m} +\!\cdot \mathbf{C} \end{array}\right] \qquad (2.14)$$

Nicht nur die ausgezeichnete, sondern jede beliebige disjunktive Normalform, also auch die disjunktive Minimalform läßt sich in der Art von Bild 2.18 darstellen. Als

Zwischencode entsteht dann im allgemeinen nicht der 1-aus-n-Code, so daß so viele Wörter disjunktiv zusammengefaßt werden, wie Einsen im Zwischencode erscheinen. Die Elemente der Matrix **D** in Gl.(2.14) werden dabei als dreiwertige Variablen aufgefaßt: $d_j^i = 0$ heißt, m_i enthält die Variable $\bar{x}_j$; $d_j^i = 1$ heißt, m_i enthält die Variable x_j; $d_j^i = *$ heißt, m_i enthält die Variable x_j nicht. Die Und-Glieder in Bild 2.11 sind nicht mehr mit jeder Variablen verbunden, und aus den vollständigen Konjunktionstermen, den Mintermen, werden unvollständige Konjunktionsterme, im Extremfall Primterme.

Im engeren Sinne werden als Festspeicher Schaltnetze entsprechend Gl.(2.14) bezeichnet, bei denen **D** = **Dual** unveränderbar ist und **C** nur einmal programmiert, d.h. vor Inbetriebnahme verändert werden kann. Man nennt sie - den Zustand nach Inbetriebnahme ansprechend - Read Only Memories (ROMs). Schaltnetze entsprechend Gln.(2.14), bei denen neben **C** auch **D** im oben beschriebenen Sinne programmierbar ist, werden demgegenüber - den Zustand vor Inbetriebnahme ansprechend - Programmable Logical Arrays (PLAs) genannt. ROMs und PLAs unterscheiden sich hauptsächlich dadurch, daß die Decodierung vollständig bzw. nur teilweise erfolgt. Wir benutzen Bild 2.18 als Symbol sowohl für ROMs als auch für PLAs und verwenden es darüber hinaus nicht nur als Symbol für zweistufige, sondern auch als Symbol für mehrstufige Schaltnetze.

Beispiel 2.5 Torschaltung als PLA

Gl.(2.9) beschreibt ein sehr einfaches Schaltnetz zum Durchschalten eines Leitungsbündels, das auch als PLA aufgebaut werden kann. Um uns an Gln.(2.14) anzupassen, ersetzen wir in Gl.(2.9) m durch x_n und **c** durch $[x_{n-1} \cdots x_2 x_1 x_0]$, setzen $\mathbf{y} = [y_{n-1} \cdots y_2 y_1 y_0]$ und beschreiben die Funktion durch eine Tabelle.

$x_n x_{n-1} \cdots x_2 x_1 x_0$	$y_{n-1} \cdots y_2 y_1 y_0$
0 * ... * * *	0 ... 0 0 0
1 * ... * * 1	* ... * * 1
1 * ... * 1 *	* ... * 1 *
1 * ... 1 * *	* ... 1 * *
⋮	⋮
1 1 ... * * *	1 ... * * *

Linke Seite und rechte Seite der Wertetabelle entsprechen den Matrizen **D** bzw. **C** in Gln.(2.14). Mit Ausnahme der ersten Zeile sind diese Matrizen in Bild 2.19 als PLA dargestellt, wobei die folgenden Symbole anstelle der üblichen Konjunktions- bzw. Disjunktionssymbole verwendet werden.

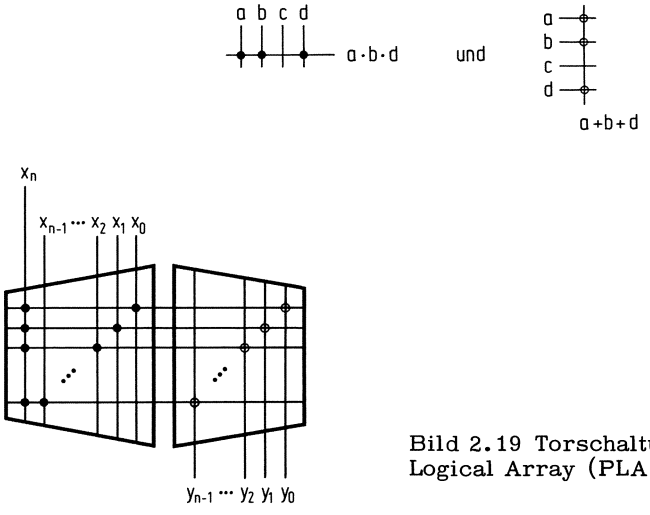


Bild 2.19 Torschaltung als Programmable-Logical Array (PLA)

■

Randomspeicher. Beim Randomspeicher ist der Inhalt des Adreßspeichers unveränderlich und der Inhalt des Wortspeichers veränderlich. Dementsprechend existieren für den Wortspeicher auch Schreibvorrichtungen. Bild 2.20 zeigt das Symbol des Randomspeichers. Er besteht aus einem Decodierer **D** als Adreßspeicher, aus matrixförmig angeordneten Speicherelementen **C** als Wortspeicher, einem Multiplexer und zwei Demultiplexern. Der Decodierer entschlüsselt die am Eingang **x** anliegende Adresse (Adressieren), und am Ausgang **y** des Multiplexers erscheint das zu dieser Adresse gehörende Wort (Lesen). Die beiden Demultiplexer übertragen das an ihren Eingängen **z** anliegende Wort in diejenige Speicherzelle, die durch den

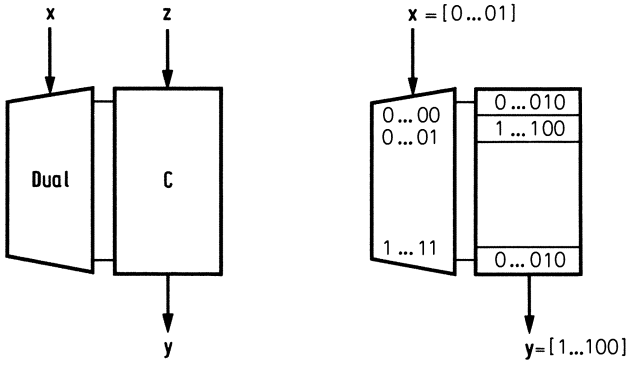


Bild 2.20 Randomspeicher (links: Symbol, rechts: Lesevorgang)

Decodierer ausgewählt wird (Schreiben). Sie steuern die Setz- bzw. Rücksetzeingänge der Speicherelemente an. Zwei Demultiplexer sind deshalb nötig, um zwischen 1-Schreiben (Setzeingang 1), 0-Schreiben (Rücksetzeingang 1) und Nicht-Schreiben (Setz-

eingang 0 und Rücksetzeingang 0) unterscheiden zu können. Die Zeiten, die vom Anlegen einer Adresse bis zum Erscheinen des zugehörigen Wortes bzw. dem Anlegen einer neuen Adresse vergehen, werden Zugriffs- bzw. Zykluszeit genannt.

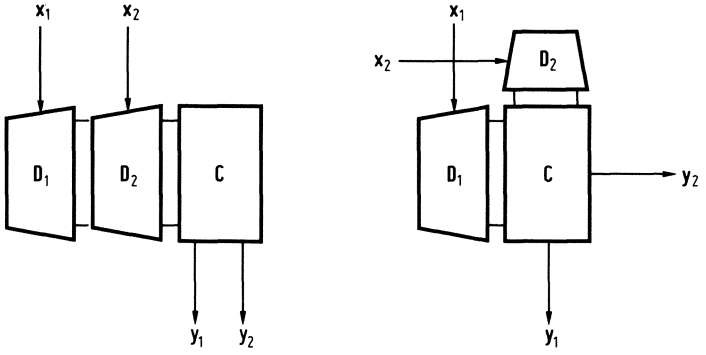
Die folgenden Formeln beschreiben die Struktur des Randomspeichers. Sie gehen aus den Gln.(2.6), (2.8) und (2.10) hervor. Mit **C** ist die Matrix der Ausgänge und mit **Cs** und **Cr** sind die Matrizen der Setz- bzw. Rücksetzeingänge der Speicherelemente bezeichnet. Steuerleitungen für Lesen und Schreiben sind der Einfachheit halber nicht berücksichtigt.

$$\left.\begin{array}{lll} \text{Decodieren:} & \mathbf{m} = \mathbf{x} \cdot\!\equiv {}^{T}\mathbf{Dual} \\ \text{Lesen:} & \mathbf{y} = \mathbf{m} +\!\cdot \mathbf{C} \\ \text{Schreiben:} & \mathbf{Cs} = {}^{T}\mathbf{m} +\!\cdot \mathbf{z} \\ & \mathbf{Cr} = {}^{T}\mathbf{m} +\!\cdot \bar{\mathbf{z}} \end{array}\right] \qquad (2.15)$$

Bei Randomspeichern erfolgen Lesen und Schreiben im allgemeinen zu verschiedenen Zeitpunkten, da der Multiplexer und die beiden Demultiplexer durch ein und denselben Decodierer angesteuert werden. Wenn zwei Decodierer vorgesehen werden, so können Lesen und Schreiben gleichzeitig erfolgen. Auch ist es mit mehreren Decodierern möglich, mehrere Speicherzellen gleichzeitig auszulesen oder zu beschreiben. Da die Adressen all dieser Speicher gewissermaßen frei wählbar sind, nennt man sie auch Speicher mit wahlfreiem Zugriff oder Random Access Memories (RAMs).

Beispiel 2.6 Zweifach auslesbare RAMs

Bild 2.21 zeigt zwei Randomspeicher, bei denen gleichzeitig zwei Zeilen bzw. eine Zeile und eine Spalte wahlfrei ausgelesen werden können. Die unter den Symbolen an-



$y_1 = (x_1 \cdot\!\equiv {}^{T}D_1) +\!\cdot C$
$y_2 = (x_2 \cdot\!\equiv {}^{T}D_2) +\!\cdot C$

$y_1 = (x_1 \cdot\!\equiv {}^{T}D_1) +\!\cdot C$
$y_2 = (x_2 \cdot\!\equiv {}^{T}D_2) +\!\cdot {}^{T}C$

Bild 2.21 Zwei Randomspeicher mit zweifacher Adreßdecodierung

gegebenen Formeln beschreiben den Aufbau der Leseeinrichtungen und definieren damit die Symbole. ■

Assoziativspeicher. Beim Assoziativspeicher sind der Inhalt des Adreßspeichers und der Inhalt des Wortspeichers veränderlich. Dementsprechend existieren sowohl für den Adreßspeicher als auch für den Wortspeicher Schreibvorrichtungen, die jedoch in verschiedenen Betriebsfällen unterschiedlich angesteuert werden. Bild 2.22 zeigt das Symbol des Assoziativspeichers. Er besteht aus matrixförmig angeordneten Speicherelementen **D** als Adreßspeicher, die mit einem Assoziierer zusammengeschaltet sind, und aus matrixförmig angeordneten Speicherelementen **C** als Wort-

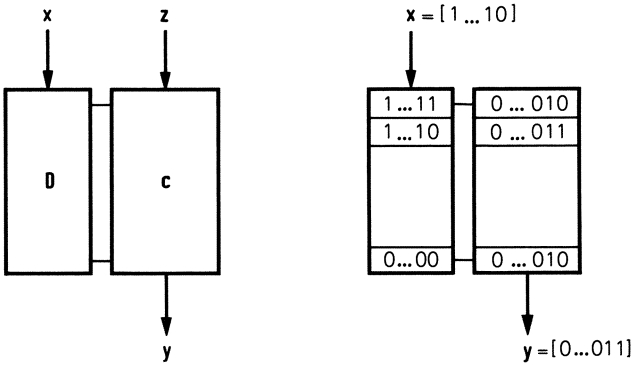


Bild 2.22 Assoziativspeicher (links: Symbol, rechts: Lesevorgang)

speicher, die mit einem Multiplexer und zwei Demultiplexern zusammengeschaltet sind. Die Ausgänge des Assoziierers steuern die Eingänge des Multiplexers und der Demultiplexer. Der Assoziierer vergleicht die am Eingang **x** anliegende Adresse mit allen im Adreßspeicher gespeicherten Adressen. Stimmt die anliegende mit einer gespeicherten Adresse in allen Bits überein, so wird ein mit dieser Adresse verbundenes Wort ausgewählt (Assoziieren), das am Ausgang **y** des Multiplexers erscheint (Lesen) oder das durch ein am Eingang **z** des Demultiplexers anliegendes Wort ersetzt wird (Schreiben).

Im Symbol sind nicht die Vorrichtungen zum Entladen und zum Laden des Assoziativspeichers enthalten. Darunter verstehen wir das Lesen bzw. Schreiben des aus Adreß- und Wortspeicher zusammengefaßten Speichers, bei dem eine Speicherzelle aus dem Paar (Adresse, Wort) besteht. Auch eine solche Zelle muß z.B. über

einen zusätzlichen Decodierer adressiert, über Multiplexer gelesen und über Demultiplexer beschrieben werden können. Die folgenden Formeln beschreiben die Struktur des Assoziativspeichers. Sie gehen aus den Gln.(2.8), (2.10) und (2.12) hervor. Mit **D** ist die Matrix der Ausgänge und mit **Ds** und **Dr** sind die Matrizen der Setz- bzw. Rücksetzeingänge der Speicherelemente des Adressenspeichers bezeichnet. **u** und **v** sind die Adressen und **w** das zu entladende bzw. das zu ladende Wort des als Einheit aus Adreß- und Wortspeicher aufgefaßten Speichers. Steuerleitungen für die verschiedenen Betriebsarten sind der Einfachheit halber nicht berücksichtigt.

$$\left.\begin{array}{ll}
\text{Assoziieren:} & \mathbf{m} = \mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D} \\
 & a = +/\mathbf{m} \\
\text{Lesen:} & \mathbf{y} = \mathbf{m} +\!\cdot \mathbf{C} \\
\text{Schreiben:} & \mathbf{Cs} = {}^{T}\mathbf{m} +\!\cdot \mathbf{z} \\
 & \mathbf{Cr} = {}^{T}\mathbf{m} +\!\cdot \bar{\mathbf{z}} \\
\text{Decodieren:} & \mathbf{k} = \mathbf{u} \cdot\!\equiv {}^{T}\mathbf{Dual} \\
\text{Entladen:} & \mathbf{v} = \mathbf{k} +\!\cdot [\mathbf{D}\,\mathbf{C}] \\
\text{Laden:} & [\mathbf{Ds}\ \mathbf{Cs}] = {}^{T}\mathbf{k} +\!\cdot \mathbf{w} \\
 & [\mathbf{Dr}\ \mathbf{Cr}] = {}^{T}\mathbf{k} +\!\cdot \overline{\mathbf{w}}
\end{array}\right] \qquad (2.16)$$

Adreßspeicher und Wortspeicher brauchen im Aufbau nicht so starr voneinander getrennt zu werden, wie es in Gln.(2.16) ausgedrückt ist. Vielmehr können beide Speicher zu einem Speicher vereinigt sein; dann bestimmt der Inhalt eines sog. Maskenregisters, welche Bits als "Adresse" und welche als "Wort" angesehen werden. Bei solchen Assoziativspeichern ist es möglich, durch die Adressierung des Teilinhaltes einer Speicherzelle den damit assoziierten Restinhalt auszulesen. Wir haben also nicht mehr eine konstante, sondern eine variable Aufteilung der Speicherzellen in "Adreß-" bzw. "Wortteil" vor uns. Da man bei all diesen Speichern einen Teil des Inhalts der Speicherzellen adressieren kann, nennt man sie auch inhaltsadressierbare Speicher oder Content Addressable Memories (CAMs).

Beispiel 2.7 Ersetzung eines CAM durch einen RAM

Bild 2.23 zeigt, wie sich ein Assoziativspeicher in einen aus Speicherelementen, Assoziierer und Codierer bestehenden Adreßspeicher und einen aus Decodierer, Multiplexer und Speicherelementen bestehenden Wortspeicher aufteilen läßt. Der Adreßspeicher kann durch einen Randomspeicher ersetzt werden, wenn in jedem Zeitpunkt die gespeicherten Adressen paarweise verschieden sind und die Kapazität des Wortspeichers eine Zweierpotenz der Adreßlänge ist.

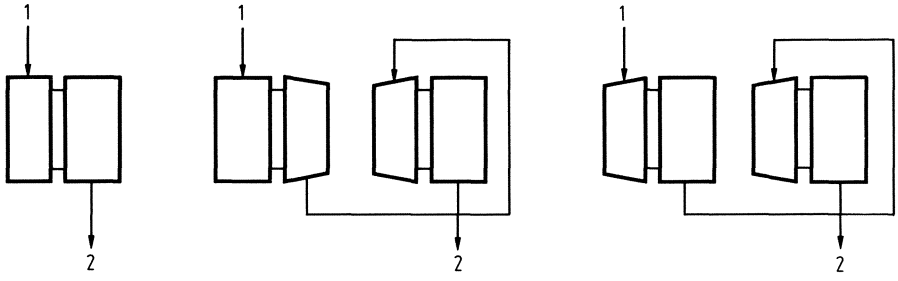


Bild 2.23 Aufteilung eines Assoziativspeichers in Adreßspeicher und Wortspeicher

Bild 2.24 zeigt, wie sich zwei hintereinandergeschaltete Randomspeicher, in denen z.B. Adressen und Wörter eines Assoziativspeichers gespeichert sind, durch einen Randomspeicher mit zwei Decodierern oder durch einen Randomspeicher mit einem Decodierer ersetzen lassen. Im letzten Fall werden zusätzlich ein Multiplexer und ein Demultiplexer zur zeitlichen Trennung des Datenflusses, ein Pufferregister zur Zwischenspeicherung der Daten und ein (nicht gezeichnetes) Schaltwerk zur Steuerung der Datenwege benötigt. Der in Bild 2.24 dargestellte Übergang von räumlich nebeneinander (parallel) ablaufenden Vorgängen in zeitlich nacheinander (seriell) ablaufende Vorgänge läßt sich von zwei auf mehrere Random-, Fest- oder Assoziativspeicher verallgemeinern.

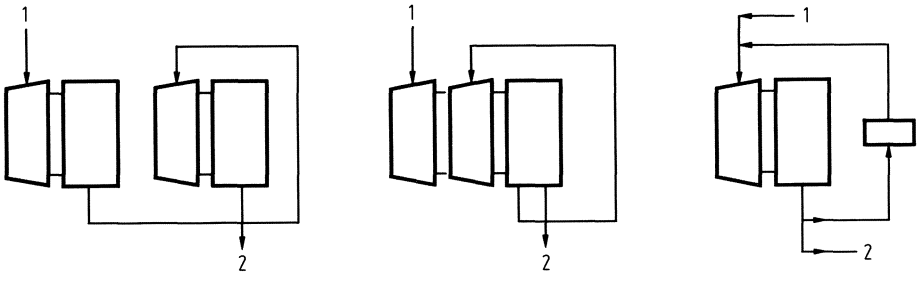


Bild 2.24 Übergang vom parallelen Betrieb zweier hintereinandergeschalteter Speicher auf seriellen Betrieb eines Speichers ■

2.3 Schaltwerke und Prozessoren

Zusammenschaltungen von Schaltnetzen und Speichern führen auf Schaltwerke, die sich in funktioneller Hinsicht passiv verhalten. Sie speichern Daten (Operanden), die von außen her durch Befehle abgefragt und verändert werden können. Um dies begrifflich zu unterstreichen, nennen wir sie Datenwerke. Das Gegenstück zu den Datenwerken bilden Schaltwerke, die sich in funktioneller Hinsicht aktiv verhalten.

Sie speichern Programme (Befehle), die nach außen hin die Abfrage und das Verändern von Daten veranlassen. Um dies zu unterstreichen, nennen wir sie Programmwerke.

Datenwerk und Programmwerk sind zwei sich gegenseitig ergänzende und aufeinander abgestimmte Teile eines umfassenderen, in gewisser Hinsicht autonomen Schaltwerks, das man digitales System, digitales Rechensystem, digitales Datenverarbeitungssystem oder kurz Digitalrechner oder Rechenanlage nennt.

Das Datenwerk eines Digitalrechners läßt sich wohl nicht physisch, so aber doch gedanklich wiederum in Daten- und Programmwerk untergliedern, so daß eine Hierarchie von Programmwerken entsteht. Das gedachte Programmwerk als Teil des Datenwerkes verarbeitet Maschinenprogramme, das physische Programmwerk als Teil des Digitalrechners verarbeitet Mikroprogramme. Der Rechner umfaßt also ineinander geschachtelt ein (gedachtes) Datenwerk, ein (gedachtes) Programmwerk und ein (physisches) Programmwerk, das wir zur begrifflichen Differenzierung Mikroprogrammwerk nennen. Bild 2.1 zeigt z.B. das Datenwerk unseres Einadreßrechners in einer strukturell betonten Darstellung. Bild 2.2 zeigt das Mikroprogrammwerk des Einadreßrechners in einer funktionell betonten Darstellung. Das Mikroprogrammwerk gibt bedingte Mikrobefehle aus, die im Datenwerk ausgeführt werden. Die Daten bestehen aus den Operanden und Befehlen des Maschinenprogramms.

Bild 2.25 zeigt in einer schematischen Darstellung das Zusammenwirken von Datenwerk und Programmwerk. Es gilt sowohl für die gedachte Aufteilung des Rechners in Daten- und Programmwerk als auch für die physische Aufteilung des Rechners in

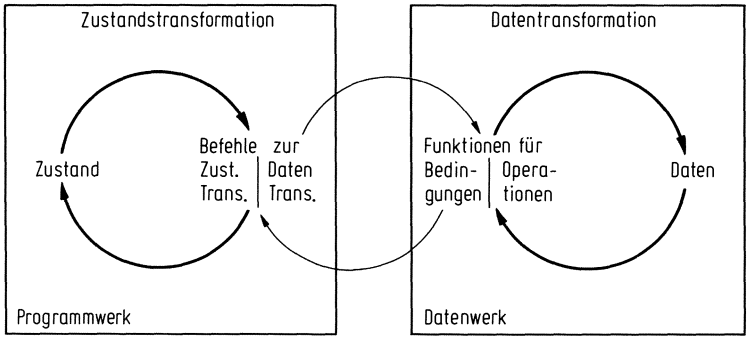


Bild 2.25 Aufteilung eines digitalen Systems (Digitalrechner) in Programmwerk und Datenwerk

Datenwerk und Mikroprogrammwerk. Das Programmwerk besteht aus einem Speicher für das unveränderliche oder zumindest als unveränderlich aufgefaßte Programm und einem Verarbeitungswerk, dessen Kern eine Speicherzelle zur Speicherung des Pro-

grammzustandes ist. Der Ablauf des Programmes kann damit als Transformation des Programmzustands umschrieben werden. Das Datenwerk besteht aus einem Speicher für die veränderlichen Daten und einem Verarbeitungswerk, dessen Kern ein Schaltnetz zur Verknüpfung der Daten ist. Der Ablauf des Programmes bewirkt im Datenwerk eine Transformation des Datenzustands.

Zur Illustration des Zusammenwirkens von Programm- und Datenwerk eines Digitalrechners wollen wir jetzt und auch im folgenden nicht mehr unseren Einadreßrechner, sondern stellvertretend dafür einen ganz einfachen Rechner zur Addition von zwei Dualzahlen verwenden. Alle aus diesem Beispiel gewonnenen Ergebnisse können verallgemeinert und auf den Entwurf von Digitalrechnern übertragen werden.

Beispiel 2.8 Additionsrechner

Wir wollen Programm- und Datenwerk[1] eines Rechners beschreiben, der die Summe von in zwei Registern A und H gespeicherten ganzen Dualzahlen ohne Berücksichtigung des Überlaufs nach dem folgenden Algorithmus berechnet. Dieser Algorithmus ist von Burks, Goldstine und von Neumann für ihren Universalrechner vorgesehen worden und noch in Rechnern der zweiten Generation verwendet worden.

```
do
  if Addieren then
  while H ≠ 0 do
    A := A exor H, H := H und A;
    if H ≠ 0 then H := H · 2
```

Die Operationen A := A exor H und H := H und A werden für alle Registerstellen gleichzeitig ausgeführt. Zusammen mit H := H · 2 bilden sie die Befehle, die im Programmwerk gespeichert vorliegen und in Abhängigkeit vom Zustand des Programms ausgegeben werden. Ausgeführt werden sie im Datenwerk; dort wird auch das Ergebnis der Operation H ≠ 0 ermittelt und dem Programmwerk als Bedingung zur Programmverzweigung oder zur Befehlsausführung mitgeteilt. Die Bedingung Addieren

[1] Da wir keine Interpretationshierarchie vor uns haben, brauchen gedachtes Programmwerk und vorhandenes Mikroprogrammwerk nicht unterschieden zu werden.

wird dem Algorithmus "von außen" zugeführt. Würde der Additionsalgorithmus in einem elektronischen Taschenrechner verwendet, so entspräche die Erfüllung dieser Bedingung dem Drücken der Additionstaste des Taschenrechners.

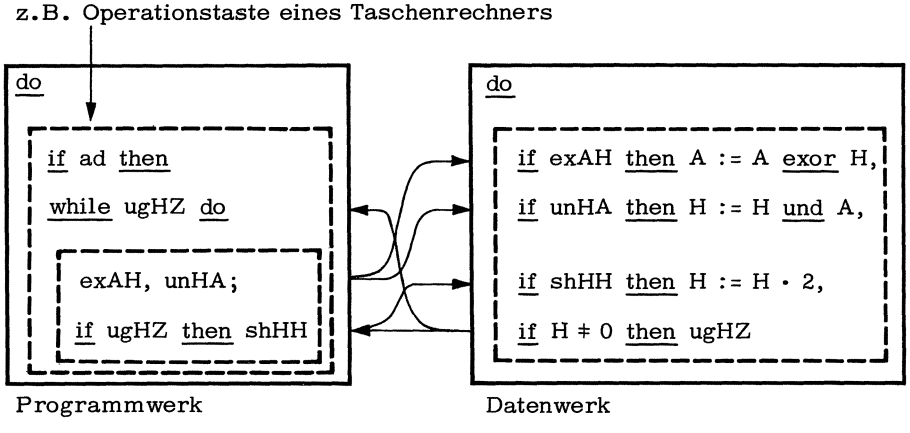
Die Aufteilung des Systems in ein steuerndes und ein ausführendes Werk kommt in dem oben wiedergegebenen Algorithmus nicht zum Ausdruck. Um sie explizit darzustellen, kürzen wir die Befehle

A := A <u>exor</u> H	durch	exAH
H := H <u>und</u> A	durch	unHA
H := H · 2	durch	shHH

und die Bedingungen

H ≠ 0	durch	ugHZ
Addieren	durch	ad

ab und formulieren den Algorithmus als auf das ausführende Werk zugeschnittenes Programm.



Je nachdem, ob Befehle und Bedingungen durch jeweils 1 Bit oder durch mehrere in Gruppen für den Befehlscode und die Operandenadressen aufgeteilte Bits codiert sind und ob in einem Schritt nur jeweils ein Befehl oder eine Bedingung bzw. mehrere Befehle und Bedingungen verarbeitet werden, entstehen charakteristische Programm- und Datenwerke, die sich wohl am besten durch die Adjektive boolesch/nichtboolesch bzw. sequentiell/nichtsequentiell kennzeichnen lassen. Wir greifen diese Begriffe später wieder auf. ■

Boolesche Algorithmen in Matrixdarstellung

Die Beschreibung von Algorithmen in der Sprache der Booleschen Algebra führt auf boolesche Algorithmen; das sind Algorithmen, deren Parameter boolesche Größen sind. Das dem Von-Neumann-Algorithmus entsprechende Additionsprogramm aus Beispiel 2.8 besteht aus Befehlen als Ausgangsgrößen, aus Bedingungen als Eingangsgrößen und aus Zuständen. Alle diese Größen können wir als boolesche Variablen auffassen, die 0 (Befehl nicht ausführen, Bedingung nicht erfüllt, Zustand nicht erlangt) oder 1 (Befehl ausführen, Bedingung erfüllt, Zustand erlangt) sein können. Im Beispiel haben wir diese Größen bereits in diesem Sinne verwendet, ohne daß wir ihnen jedoch boolesche Werte zugewiesen hätten. Wir haben damit der stillschweigend gefaßten Übereinkunft Rechnung getragen, Befehle und Bedingungen nur dann aufzuschreiben, wenn sie ausgeführt bzw. abgefragt werden sollen. Die Darstellung von Algorithmen durch die Boolesche Algebra zwingt uns jedoch dazu, neben der Ausführung von Befehlen und Bedingungen auch deren Nichtausführung zu vermerken. Die Beschreibung von Algorithmen erfolgt dann in Form von booleschen Algorithmen. Den angesprochenen Additionsalgorithmus können wir auch in einer Form schreiben, in der wir die Programmzustände besser erkennen können.

```
wait:   if Addieren goto loop else goto wait;
loop:   goto retn, A := A exor H, H := H und A;
retn:   if H ≠ 0 then goto loop else goto wait,
        if H ≠ 0 then H := H · 2
```

Dieses Programm ist die sprachliche Umschreibung eines Graphen und kann ebensogut durch boolesche Funktionsgleichungen beschrieben werden. Der besseren Übersichtlichkeit halber benutzen wir für die Befehle und Bedingungen die in Beispiel 2.8 eingeführten Abkürzungen und für und, oder und nicht die entsprechenden booleschen Operationszeichen.

$$\text{wait} := \text{wait} \cdot \overline{\text{ad}} + \text{loop} \cdot 0 + \text{retn} \cdot \overline{\text{ugHZ}}$$

$$\text{loop} := \text{wait} \cdot \text{ad} + \text{loop} \cdot 0 + \text{retn} \cdot \text{ugHZ}$$

$$\text{retn} := \text{wait} \cdot 0 + \text{loop} \cdot 1 + \text{retn} \cdot 0$$

$$\text{exAH} = \text{wait} \cdot 0 + \text{loop} \cdot 1 + \text{retn} \cdot 0$$

$$\text{unHA} = \text{wait} \cdot 0 + \text{loop} \cdot 1 + \text{retn} \cdot 0$$

$$\text{shHH} = \text{wait} \cdot 0 + \text{loop} \cdot 0 + \text{retn} \cdot \text{ugHZ}$$

Dieser boolesche Algorithmus gibt die Struktur eines Schaltwerkes wieder, das wir boolesches Mikroprogrammwerk nennen. Bevor wir solche Schaltwerke genauer besprechen, wollen wir zunächst auf boolesche Algorithmen bzw. Schaltwerke allgemein eingehen.

Ein boolescher Algorithmus beschreibt die Transformation **f** der Werte eines Vektors **z** (Zustände) und die Berechnung **g** der Werte eines Vektors **a** (Befehle, Ausgangsgrößen) in Abhängigkeit von den Werten eines Vektors **e** (Bedingungen, Eingangsgrössen). Betrachten wir zunächst das Paar boolescher Funktionen

$$\mathbf{z} := \mathbf{f}(\mathbf{e}) \tag{2.17}$$

$$\mathbf{a} = \mathbf{g}(\mathbf{z}) \tag{2.18}$$

Da die Werte von **z** als Codewörter eines 1-aus-n-Codes aufgefaßt werden können, lassen sich diese booleschen Funktionen durch zwei Matrizen **F** und **G** darstellen (Bild 2.26), deren Elemente f_j^i bzw. g_i^j 0 oder 1 sind, je nachdem ob vom Zustand

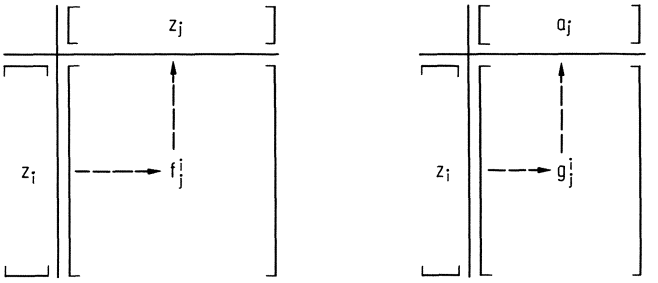


Bild 2.26 Übergangs- und Ausgangsmatrix einer booleschen Funktion

z_i der Zustand z_j erreicht werden kann (im Graphen ein Pfeil von z_i nach z_j) bzw. ob dem Zustand z_i die Ausgangsgröße a_j zugeordnet ist (im Graphen ein im Bereich von z_i erscheinendes a_j). **F** und **G** sind boolesche Matrizen mit 0 und 1 als Elementen, und Gln. (2.17) und (2.18) können mit der verallgemeinerten Multiplikation auch in

$$\mathbf{z} := \mathbf{z} +\cdot \mathbf{F} \tag{2.19}$$

$$\mathbf{a} = \mathbf{z} +\cdot \mathbf{G} \tag{2.20}$$

umgeschrieben werden.

Bei einem booleschen Algorithmus sind die Funktionen Gln. (2.19) und (2.20) jedoch nicht nur von den Zuständen **z**, sondern auch von den Eingangsgrößen **e** abhängig.

$$\mathbf{z} := \mathbf{f}(\mathbf{z},\mathbf{e}) \tag{2.21}$$

$$\mathbf{a} = \mathbf{g}(\mathbf{z},\mathbf{e}) \tag{2.22}$$

Auch diese Gleichungen können durch zwei Matrizen **F**(**e**) und **G**(**e**) dargestellt werden (Bild 2.27), deren Elemente $f_j^i(\mathbf{e})$ bzw. $g_j^i(\mathbf{e})$ Funktionen des Eingangsvektors **e** sind, je nachdem ob vom Zustand z_i unter der durch $f_j^i(\mathbf{e})$ ausgedrückten Bedin-

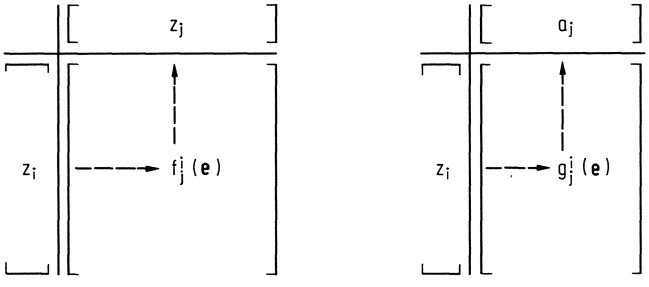


Bild 2.27 Übergangs- und Ausgangsmatrix eines booleschen Algorithmus

gung der Zustand z_j erreicht werden kann (im Graphen ein bezeichneter Pfeil von z_i nach z_j) bzw. ob dem Zustand z_i unter der durch $g_j^i(\mathbf{e})$ ausgedrückten Bedingung eine Ausgangsgröße a_j zugeordnet ist (im Graphen ein im Bereich von z_i erscheinendes bedingtes a_j). **F**(**e**) und **G**(**e**) sind boolesche Matrizen mit booleschen Ausdrücken als Elementen, und Gln. (2.21) und (2.22) lassen sich in Analogie zu Gln. (2.19) und (2.20) durch

$$\mathbf{z} := \mathbf{z} +\cdot \mathbf{F}(\mathbf{e}) \tag{2.23}$$

$$\mathbf{a} = \mathbf{z} +\cdot \mathbf{G}(\mathbf{e}) \tag{2.24}$$

ausdrücken. **F**(**e**) und **G**(**e**) heißen Übergangs- bzw. Ausgangsmatrix. Wird ein boolescher Algorithmus durch die Kombination von Gl. (2.23) und Gl. (2.20) beschrieben, so entstehen Schaltwerke, deren Ausgänge nur von den Zuständen abhängen[1], während aus der Kombination von Gl. (2.23) und Gl. (2.24) Schaltwerke entstehen, deren Ausgänge außer von den Zuständen auch von den Eingangsgrößen abhängen können[2].

Boolesche Mikroprogrammwerke

Boolesche Algorithmen beschreiben Funktion und Struktur von Programmwerken, die dadurch gekennzeichnet sind, daß ihre Befehle und Bedingungen "boolesch" codiert sind und daß ihre Programme "nicht sequentiell" ablaufen, d.h. daß in einem Schritt mehrere Bedingungen gleichzeitig mit mehreren Befehlen verarbeitet werden.

---

[1] In der Automatentheorie als Moore-Automaten bezeichnet.

[2] In der Automatentheorie als Mealy-Automaten bezeichnet.

Im Datenwerk sind im allgemeinen nicht alle möglichen Kombinationen von Operationen mit einem oder mehreren Operanden vorgesehen, sondern nur diejenigen, die in den Befehlen des Programmes vorkommen. Bestimmte Speicherzellen sind mit bestimmten Schaltnetzen verbunden, so daß ein spezialisiertes Datenwerk entsteht.

Im Programmwerk sind für jede Bedingung zwei Leitungen vorgesehen, deren Werte angeben, ob die Bedingung erfüllt, nicht erfüllt oder nicht berücksichtigt ist. Für jeden Befehl ist eine Leitung vorgesehen, deren Wert angibt, ob der Befehl im Datenwerk ausgeführt oder nicht ausgeführt wird. Das aus Bedingungen, Befehlen und Folgezuständen bestehende Programm liegt im allgemeinen in unveränderlicher Form vor (fest gespeichert, fest verdrahtet), da für das spezialisierte Datenwerk universelle Programme kaum geeignet sind.

Parallelverarbeitung. Bild 2.28 zeigt die Struktur eines booleschen Programmwerkes für parallele Verarbeitung von Mikrobefehlen und -bedingungen zusammen mit

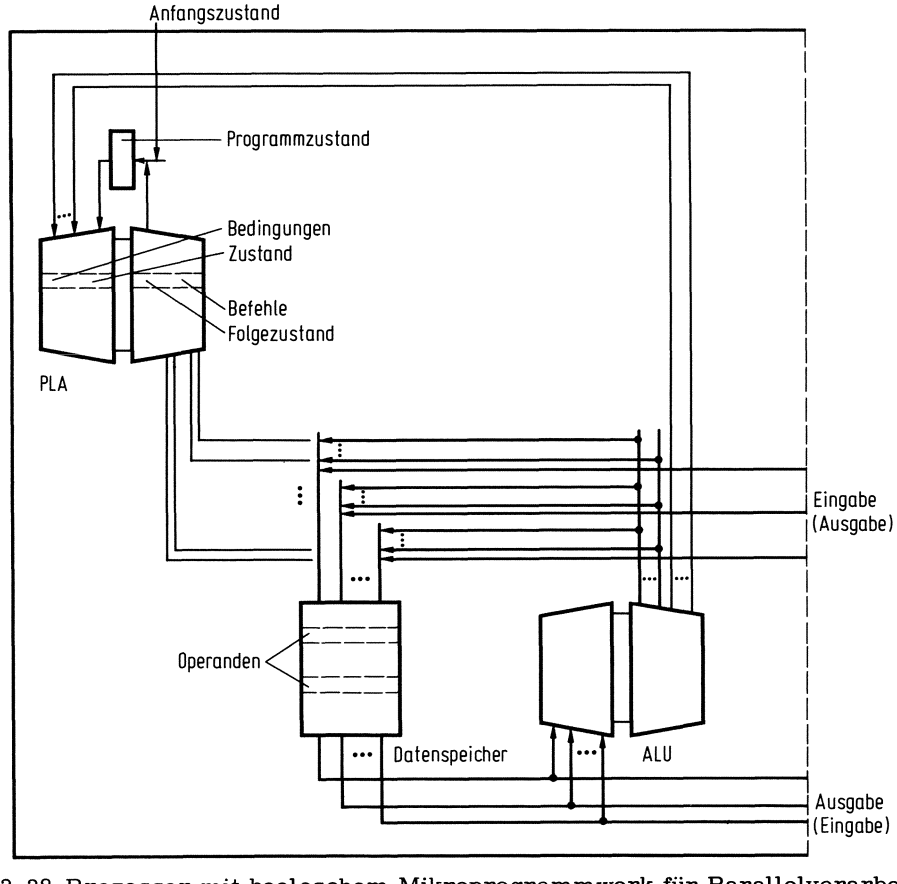


Bild 2.28 Prozessor mit booleschem Mikroprogrammwerk für Parallelverarbeitung (ALU steht als Abkürzung für arithmetisch-logische Einheit)

einem darauf abgestimmten Datenwerk. In einem Schritt werden im Festpeicher des Programmwerkes die Bedingungen aus dem Datenwerk abgefragt und zusammen mit dem Zustand decodiert. Der Folgezustand liegt an den Eingängen der Verzögerungsglieder an, und die Befehle steuern einen zweidimensionalen Multiplexer des Datenwerkes, der Verbindungen von bestimmten Speicherzellen mit bestimmten Schaltnetzen auswählt. Im nächsten Schritt erscheinen in den Verzögerungsgliedern des Programmwerkes der Folgezustand und in den Speicherzellen des Datenwerkes die Ergebnisse. Die Wirkungsweise des Systems wird später durch Beispiel 2.9 illustriert.

Bild 2.28 vermag die funktionelle und strukturelle Beschreibung nicht so genau wiederzugeben, wie es mit den Formeln der Booleschen Algebra möglich ist. Dazu beschreiben wir das Programmwerk durch boolesche Algorithmen ausgehend von Gln. (2.23) und (2.24) und codieren die Zustände, so daß weniger Flipflops zur Speicherung des Zustands benötigt werden. Wir bezeichnen den decodierten Zustandsvektor wie früher mit $\mathbf{z}$, den codierten Zustandsvektor mit $\mathbf{u}$ und die Codiermatrix mit **Code**, so daß wir für das Programmwerk die folgenden Funktionen erhalten.

$$\left.\begin{array}{ll} \text{Übergangsfunktion} & \mathbf{u} := \mathbf{z} +\cdot \mathbf{U}(\mathbf{e}) \\ \text{Ausgangsfunktion} & \mathbf{a} = \mathbf{z} +\cdot \mathbf{G}(\mathbf{e}) \\ \text{mit den Abkürzungen} & \mathbf{z} = \mathbf{u} \cdot\equiv {}^{T}\mathbf{Code} \\ & \mathbf{U}(\mathbf{e}) = \mathbf{F}(\mathbf{e}) +\cdot \mathbf{Code} \end{array}\right] \qquad (2.25)$$

$\mathbf{U}(\mathbf{e})$ nennen wir codierte Übergangsmatrix, da sie die Transformation des decodierten Zustandsvektors $\mathbf{z}$ in den codierten Zustandsvektor $\mathbf{u}$ beschreibt. Aus den Matrizen **Code**, $\mathbf{U}(\mathbf{e})$ und $\mathbf{G}(\mathbf{e})$ lassen sich die Schaltwerksgleichungen und die Schaltwerkstabelle entwickeln, aus der die Verdrahtung des Schaltwerkes bzw. die Programmierung des Festspeichers abgelesen werden kann.

Zur Beschreibung des Datenwerkes durch boolesche Funktionen gehen wir von Gl. (2.8) aus und schreiben $\mathbf{f}(\mathbf{C})$ anstelle von $\mathbf{C}$ und $\mathbf{c}$ anstelle von $\mathbf{y}$. $\mathbf{c}^0, \mathbf{c}^1, \ldots$ bezeichnen die Speicherzellen des Speichers $\mathbf{C}$, $\mathbf{f}(\mathbf{C})$ bezeichnet eine Vektorfunktion mit den Speicherzellen als unabhängigen Variablen. Für bestimmte Speicherzellen $\mathbf{c}^0, \mathbf{c}^1, \ldots$ sind Funktionen $\mathbf{f}^0(\mathbf{C})$, $\mathbf{f}^1(\mathbf{C}), \ldots$ vorgesehen, die durch Multiplexereingänge $\mathbf{m}^0, \mathbf{m}^1, \ldots$ ausgewählt werden. Wir fassen diese Funktionen zu den Matrizen $\mathbf{F}(\mathbf{C})$ und $\mathbf{M}$ zusammen, so daß wir das Datenwerk durch die folgenden Funktionen beschreiben können.

$$\left.\begin{array}{ll} \text{Übergangsfunktion} & \mathbf{C} := \mathbf{M} +\cdot \mathbf{F}(\mathbf{C}) \\ \text{Ausgangsfunktion} & \mathbf{a} = \mathbf{g}(\mathbf{C}) \end{array}\right] \qquad (2.26)$$

g ist die Vektorfunktion für die Ausgänge des Datenwerkes und enthält nur die Speicherzellen als unabhängige Variablen. Aus den beiden Funktionen lassen sich die Schaltwerksgleichungen ermitteln, aus denen die Verdrahtung des Schaltnetzes entwickelt werden kann.

Die Funktionsgleichungen für das Programmwerk und das Datenwerk hängen über die Eingangs- und Ausgangsgrößen zusammen. Die Eingänge des Programmwerkes sind die Ausgänge des Datenwerkes (Bedingungen). Die Eingänge des Datenwerkes sind die Ausgänge des Programmwerkes (Befehle), die beim Programmwerk in Form eines Zeilenvektors und beim Datenwerk in Form einer Matrix angeordnet sind. Aus der matrixförmigen Anordnung entsteht die vektorförmige Anordnung der Eingänge dadurch, daß die Zeilen der Matrix aneinandergesetzt und überflüssige Nullen eliminiert werden. Beide Werke sind in jedem Schritt synchronisiert.

Serienparallelverarbeitung. Um für den Festspeicher im Programmwerk ROMs anstelle von PLAs verwenden zu können, wird nur eine Leitung zur Abfrage von Bedingungen vorgesehen (Bild 2.29). Das hat zur Folge, daß die Ausgangsleitungen des Datenwerkes durch einen Multiplexer zusammengefaßt werden, der durch zusätzliche codierte Befehle gesteuert wird. Die Adreßlänge des Speichers, der das Programm aufnehmen soll, verkleinert sich also auf Kosten seiner Wortlänge. Seine Kapazität vergrößert sich, da die Anzahl der Zustände wächst und für jeden Zustand zwei Adreß- und Speicherwörter benötigt werden.

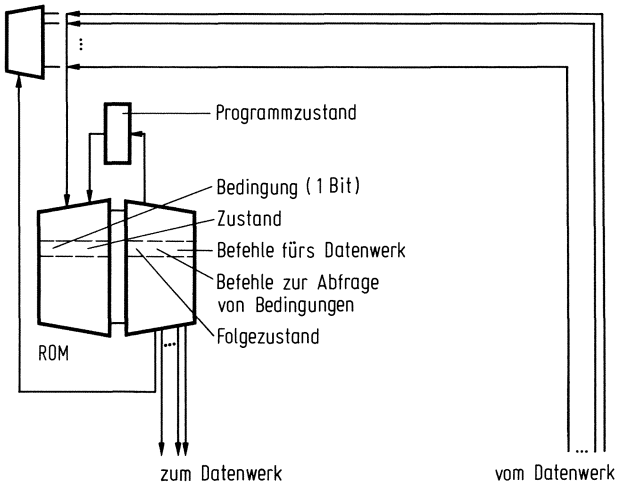


Abb. 2.29 Boolesches Mikroprogrammwerk für Serienparallelverarbeitung

Beispiel 2.9 Additionsrechner

Mit den Abkürzungen aus Beispiel 2.8 für die Befehle und Bedingungen können wir den Additionsalgorithmus durch einen booleschen Algorithmus in Form von zwei Vek-

torgleichungen beschreiben.

$$[\text{wait loop retn}] := [\text{wait loop retn}] +\cdot \begin{bmatrix} \overline{\text{ad}} & \text{ad} & 0 \\ 0 & 0 & 1 \\ \overline{\text{ugHZ}} & \text{ugHZ} & 0 \end{bmatrix} \tag{2.27}$$

$$[\text{exAH unHA shHH}] = [\text{wait loop retn}] +\cdot \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & \text{ugHZ} \end{bmatrix} \tag{2.28}$$

Programmwerk: Mit dualer Zustandscodierung entstehen aus diesen Gleichungen die Übergangs- und die Ausgangsfunktion des Programmwerkes, aus denen unmittelbar die Verdrahtungsfunktionen oder die Programmtabelle (vgl. Bild 2.30) gewonnen werden können.

$$[u_0\ u_1] := [\bar{u}_0\bar{u}_1\ \bar{u}_0u_1\ u_0\bar{u}_1] +\cdot \begin{bmatrix} 0 & \text{ad} \\ 1 & 0 \\ 0 & \text{ugHZ} \end{bmatrix} \tag{2.29}$$

$$[\text{exAH unHA shHH}] = [\bar{u}_0\bar{u}_1\ \bar{u}_0u_1\ u_0\bar{u}_1] +\cdot \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & \text{ugHZ} \end{bmatrix} \tag{2.30}$$

Datenwerk: Mit den Abkürzungen

$$\begin{aligned} \mathbf{a} &= [a_{15} \ldots a_1 a_0] \quad \text{für Register A} \\ \mathbf{h} &= [h_{15} \ldots h_1 h_0] \quad \text{für Register H} \\ \mathbf{z} &= [0 \quad \ldots 0\ 0\ ] \\ \mathbf{sh(h)} &= [h_{14} \ldots h_0 0\ ] \end{aligned}$$

lassen sich die Übergangs- und Ausgangsfunktion des Datenwerkes angeben, aus denen unmittelbar die Beschaltung der Registerflipflops gewonnen werden kann.

Übergangsfunktion:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{h} \end{bmatrix} := \begin{bmatrix} \text{ldAA} & 0 & \text{exAH} & 0 & 0 \\ 0 & \text{ldHH} & 0 & \text{unHA} & \text{shHH} \end{bmatrix} +\cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{h} \\ \mathbf{a} \not\equiv \mathbf{h} \\ \mathbf{a} \cdot \mathbf{h} \\ \mathbf{sh(h)} \end{bmatrix} \tag{2.31}$$

mit $\text{ldAA} = \overline{\text{exAH}}$ und $\text{ldHH} = \overline{\text{unHA}} \cdot \overline{\text{shHH}}$

Ausgangsfunktion:

$$ugHZ = \mathbf{h} +\!\!\!\!\neq {}^{T}\mathbf{z} \tag{2.32}$$

Bild 2.30 zeigt den Additionsrechner mit einem PLA als Programmspeicher und SR-Flipflop-Registern als Datenspeicher. Zum Verständnis der Wirkungsweise boolescher Mikroprogrammwerke simuliere man den Ablauf des Additionsprogramms auf dem Papier, wobei der Programmzustand anfangs als $[u_0 u_1] = [0\ 0]$ anzunehmen ist.

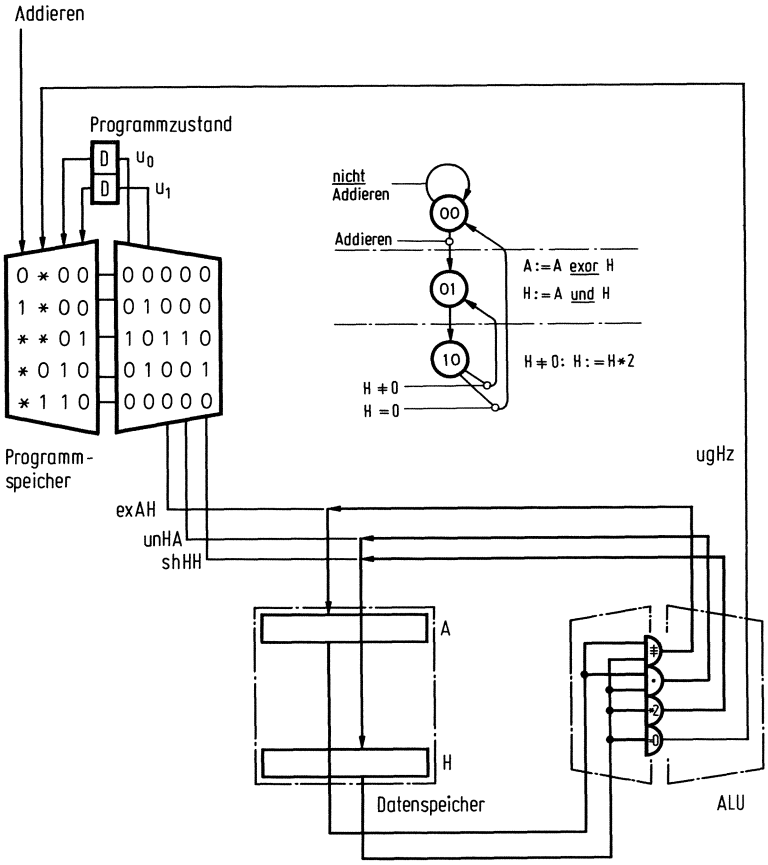


Bild 2.30 Rechner zur Addition von Dualzahlen mit booleschem Mikroprogrammwerk und darauf abgestimmtem Datenwerk ■

Sequentielle Mikroprogrammwerke

Boolesche Programmwerke führen auf digitale Systeme, die auf die Lösung einer bestimmten Aufgabe speziell zugeschnitten sind. Zwar ließe sich der Festspeicher durch einen Assoziativspeicher ersetzen, in dem zur Lösung verschiedener Aufgaben unter-

schiedliche Programme gespeichert werden können. Dann müßte aber auch das in Bild 2.28 als Festspeicher gezeichnete Schaltnetz im Datenwerk durch einen Assoziativspeicher ersetzt werden, oder es müßten alle infrage kommenden Verknüpfungen mit allen zur Verfügung stehenden Operanden parallel aufgebaut werden, was bei den meisten Problemen gar nicht ausgenutzt werden könnte.

So ist es natürlich, wenn für digitale Systeme, die für die Lösung von verschiedenen Aufgaben universell verwendbar sein sollen, von der "booleschen" auf die "nichtboolesche" Codierung und von der "nichtsequentiellen" auf die "sequentielle" Arbeitsweise übergegangen wird, kurz als horizontale bzw. vertikale Mikroprogrammierung bezeichnet.

Im Datenwerk genügt es, statt einer Vielzahl gleicher Verknüpfungsschaltnetze ($m \cdot n^2$ bei n Operanden für m zweistellige Operationen) einige wenige (m) vorzusehen. Statt einer Vielzahl von Leitungsbündeln (n^3) genügen einige wenige Sammelleitungsbündel (3), die mit den Schaltnetzen verbunden sind.

Im Programmwerk genügt es, statt einer Vielzahl Befehlsbits ($m \cdot n^2$) einige wenige ($\lceil \mathrm{ld}\, m \rceil \cdot 2 \cdot \lceil \mathrm{ld}\, n \rceil$)[1] vorzusehen, um Operationen und Operanden auszuwählen. Statt einer Vielzahl von Bedingungsbits genügt es, die Bedingungen wie Befehle mit booleschem Ergebnis zu behandeln. Wegen der sequentiellen Verarbeitung der Befehle lohnt es sich oft nicht mehr, zu jedem Befehl den Folgezustand, d.h. die Adresse des nächsten Befehls anzugeben. Vielmehr wird ein Zähler vorgesehen, der die Befehle sequentiell abruft und der nur durch Bedingungen oder durch Sprungbefehle in seiner Sequenz unterbrochen wird (Befehlszähler).

Serienverarbeitung. Bild 2.31 zeigt die Struktur eines sequentiellen Mikroprogrammwerkes mit einer Codierung der Mikrobefehle durch Befehlscode und Operandenadressen zusammen mit einem darauf abgestimmten Datenwerk. In einem Schritt werden im Randomspeicher des Programmwerkes der Befehlszählerstand decodiert, ein Befehl ausgelesen und der Befehlscode im Programmwerk und im Datenwerk decodiert. Davon abhängig wird im Programmwerk der neue Befehlszählerstand ermittelt, und im Datenwerk erscheinen an den Ausgängen des zweifach adressierbaren Randomspeichers die beiden Operanden, die zum Ergebnis der Operation verknüpft werden. Dieses Ergebnis liegt am Eingang des Randomspeichers an und ersetzt im nächsten Schritt den durch die erste Adresse ausgewählten Operanden. Gleichzeitig erscheint die Adresse des zeitlich nächsten Befehls im Befehlszähler. Die Wirkungsweise des Systems wird später durch Beispiel 2.10 illustriert.

---

[1] $\lceil x \rceil$ bedeutet die kleinste ganze Zahl größer oder gleich x.

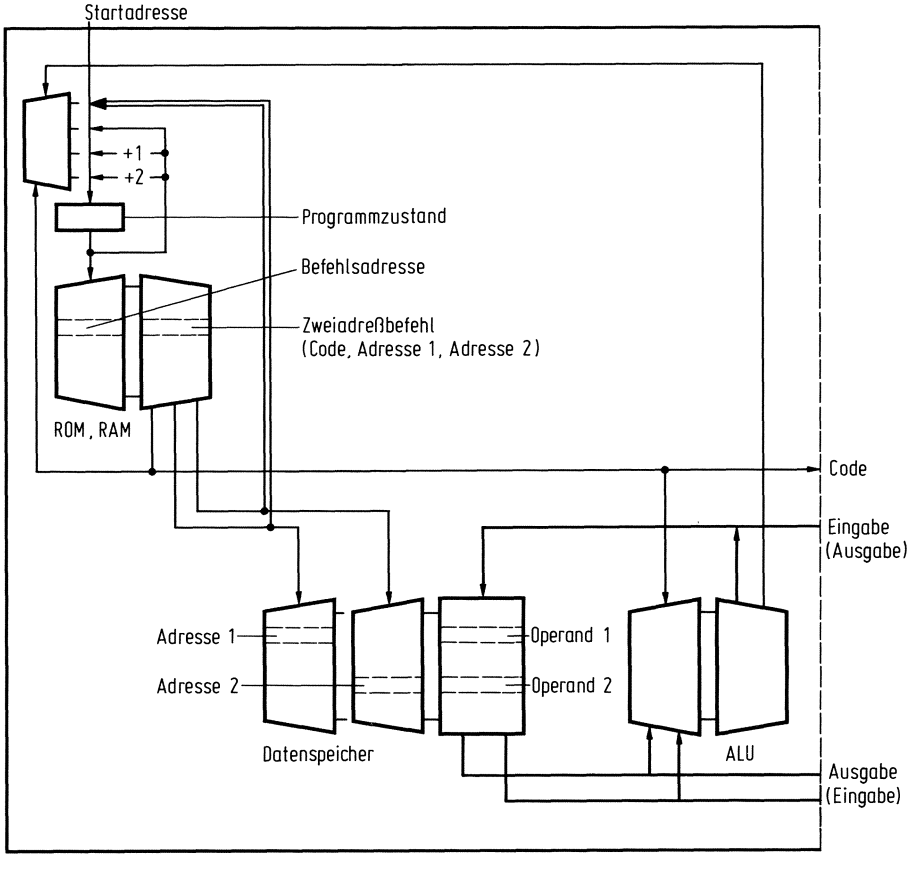


Bild 2.31 Prozessor mit sequentiellem Mikroprogrammwerk für Serienverarbeitung

Bild 2.31 vermag die funktionelle und strukturelle Beschreibung nicht so genau wie die Formeln der Mikrosprache wiederzugeben. Zur Beschreibung des Programmwerkes setzen wir die Existenz eines Sprungbefehls (goto), eines Haltebefehls (halt) und mindestens eines von zwei Operanden abhängigen Überspringbefehls (skip) voraus.

Befehl = Programmspeicher [P],
[Code Adresse1 Adresse2] = Befehl,
<u>if</u> Code = goto <u>then</u> P := [Adresse1 Adresse2],
<u>if</u> Code = halt <u>then</u> P := P,
<u>if</u> Code = skip <u>and</u> Bedingung <u>then</u> P := P+2
<u>else</u> P := P+1

Zur Beschreibung des Datenwerkes setzen wir neben einem Skipbefehl auch boolesche und einfache arithmetische Befehle voraus, die wir aber ebenso wie die Art der Bedingung lediglich durch Funktionen von Operanden allgemein ausdrücken.

Operand1 = Datenspeicher [Adresse1],
Operand2 = Datenspeicher [Adresse2],
<u>if</u> Code = skip
<u>then</u> Bedingung = Funktion (Operand1, Operand2)
<u>else</u> Operand1 := Funktion (Operand1, Operand2)

Die Formeln für das Programmwerk und das Datenwerk hängen über ihre Eingangs- und Ausgangsgrößen zusammen. Die Eingangsgröße des Programmwerkes und gleichzeitig Ausgangsgröße des Datenwerkes ist die Bedingung. Die Eingangsgrößen des Datenwerkes und gleichzeitig Ausgangsgrößen des Programmerkes sind Code, Adresse1 und Adresse2. Beide Werke sind in jedem Schritt synchronisiert.

<u>Serienparallelverarbeitung.</u> Wenn Programme viele Sprungbefehle aufweisen, ist es zweckmäßig, auch bei sequentiellen Programmwerken zu jedem Befehl die Adresse des nächsten Befehls (Folgeadresse) zu speichern (Bild 2.32). Dies entspricht der

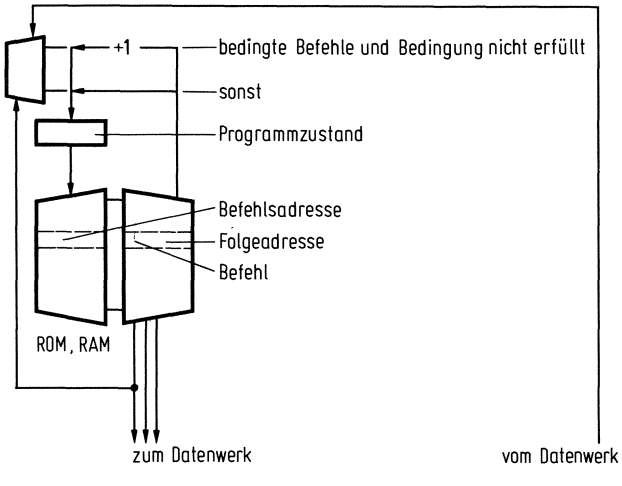


Bild 2.32 Sequentielles Mikroprogrammwerk für Serienparallelverarbeitung

parallelen Ausführung von zwei Befehlen in jedem Schritt, von denen einer immer ein Sprungbefehl ist. Bei bedingten Befehlen ist die Auswahl aus zwei Sprungzielen erforderlich: Der beim Befehl gespeicherte Sprung wird ausgeführt, wenn die Bedingung erfüllt ist, der nächste Befehl wird ausgeführt, wenn die Bedingung nicht erfüllt ist. Durch die parallele Ausführung von zwei Befehlen nimmt die Wortlänge des Speichers, der das Programm aufnimmt, bei abnehmender Kapazität zu, da ein Teil der Befehle horizontal statt vertikal angeordnet erscheint.

Beispiel 2.10 Additionsrechner

Mit den Abkürzungen aus Beispiel 2.8 für die Befehle und Bedingungen zuzüglich der Abkürzungen ldXA für X := A und unHX für H := H und X schreiben wir den Additionsalgorithmus als sequentielles Programm.

```
wait: if     ad
loop: then   ldXA;
             exAH;
             unHX;
             if     ugHZ
             then   shHH
                    goto loop
             else   goto wait
```

Programmwerk: Durch die Aufteilung der Befehle in Befehlscode und ein oder zwei Operandenadressen entsteht aus diesem Algorithmus ein Programm (vgl. Bild 2.33), das von dem durch die folgenden Formeln beschriebenen Programmwerk interpretiert werden kann. Mit den Formeln kann das Programmwerk unmittelbar aufgebaut werden.

```
Befehl = Programmspeicher [P],
[Code Adresse1 Adresse2] = Befehl;
if Code = go then P := [Adresse1 Adresse2]
if Code = ht then P := P
if Code = ug and Bedingung then P := P+2
                            else P := P+1
```

Datenwerk: Zur Ausführung der im Programmwerk gespeicherten Befehle ist ein Datenwerk vonnöten, das durch die folgenden Formeln beschrieben wird. Mit diesen Formeln kann das Datenwerk unmittelbar aufgebaut werden.

```
Operand1 = Datenspeicher [Adresse1],
Operand2 = Datenspeicher [Adresse2],
if Code = ld  then Operand1 := Operand2
if Code = ex  then Operand1 := Operand1 exor Operand2,
if Code = un  then Operand1 := Operand1 und  Operand2,
if Code = sh  then Operand1 := Operand2 · 2,
if Code = ug  then Bedingung = Operand1 ≠ Operand2
```

Bild 2.33 zeigt den Additionsrechner mit einem ROM als Programmspeicher und einem zweifach adressierbaren SR-Flipflop-RAM als Datenspeicher. Zum Verständnis der Wirkungsweise sequentieller Mikroprogrammwerke simuliere man den Ablauf des Additionsprogramms auf dem Papier, wobei der Programmzählerstand anfangs zu 0 anzunehmen ist.

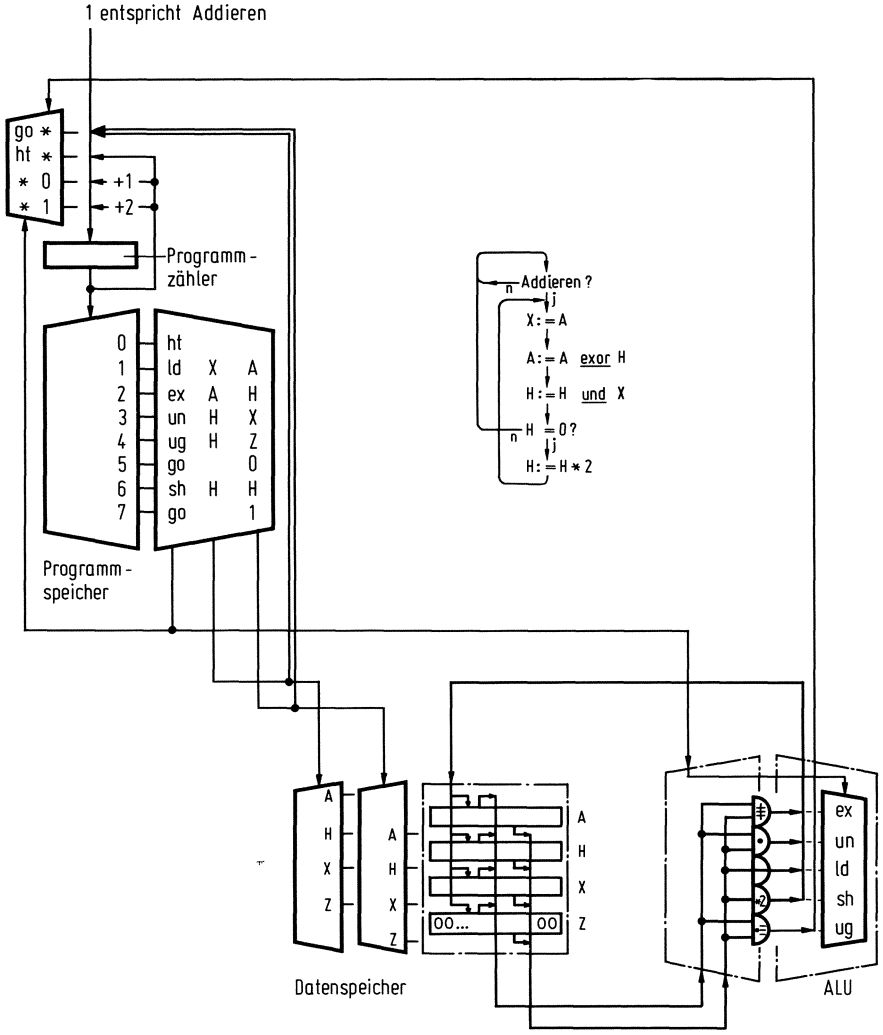


Bild 2.33 Rechner zur Addition von Dualzahlen mit sequentiellem Mikroprogrammwerk und darauf abgestimmtem Datenwerk ■

Mikroprogrammierte Rechner

Wie bereits am Anfang des vorhergehenden Abschnittes beschrieben, können mikroprogrammierte Digitalrechner nur gedanklich in ein Programmwerk und ein Datenwerk zerlegt werden. Programm-Speicher (PrSp) und Daten-Speicher (DaSp) lassen

sich physisch nicht trennen, und zur Unterscheidung von Befehlen und Operanden müssen Multiplexer (z.B. für Befehlholen und Operandholen bei unserem Einadreßrechner) und Pufferspeicher (z.B. Befehlsregister, A-Register, H-Register beim Einadreßrechner) vorgesehen werden. Programm-Verarbeitungswerk (PrVw) und Daten-Verarbeitungswerk (DaVw) lassen sich ebenfalls physisch nicht trennen, denn zumindest die Multiplexer können weder zu dem einen Werk (bei unserem Einadreßrechner das Leitwerk) noch zu dem anderen Werk (beim Einadreßrechner das Rechenwerk) gezählt werden. Programm- und Daten-Speicher werden genau wie Programm- und Daten-Verarbeitungswerk zu zwei Einheiten (PrDaSp und PrDaVw) zusammengefaßt, die als Speicher bzw. Prozessor bezeichnet werden. Dabei wird von der Vorstellung ausgegangen, jeder Befehl würde in einem Schritt verarbeitet[1].

Bei mikroprogrammierten Rechnern werden die Einheiten PrDaVw und PrDaSp zum Datenwerk zusammengefaßt (vgl. Bild 2.1 unseres Einadreßrechners), das mit einem Mikroprogrammwerk zusammenarbeitet (vgl. Bild 2.2 des Einadreßrechners). Hier haben wir die physische Aufteilung eines mikroprogrammierten Rechners in Programmwerk und Datenwerk vor uns. Das Programmwerk umfaßt den Mikroprogramm-Speicher (MpSp) und das Mikroprogramm-Verarbeitungswerk (MpVw), die wiederum - der Systematik folgend - mit den Einheiten PrDaVw bzw. PrDaSp des Datenwerkes zum Mikroprogramm-Programm-Daten-Verarbeitungswerk (MpPrDaSp) zusammengefaßt werden. Diese beiden neuen Einheiten bilden den Prozessor bzw. den Speicher des mikroprogrammierten Rechners, wobei jeder Mikrobefehl in einem Schritt verarbeitet wird. Bild 2.34 illustriert diese zweistufige Programmhierarchie eines Rechners mit den Befehlen als Elementarhandlungen in der oberen Stufe und mit den Mikrobefehlen als Elementarhandlungen in der unteren Stufe. Die dargestellte Hierarchie kann nach oben oder unten auf mehr als zwei Stufen erweitert werden. Der Rechner interpretiert auf jeder Stufe die Programme der nächsthöheren Stufe, z.B. interpretiert der Rechner auf der Mikroprogrammstufe die Maschinenprogramme.

Bei austauschbaren oder ladbaren Mikroprogrammen ist es damit möglich, verschiedene Maschinensprachen zu interpretieren und die Funktion verschiedener Rechenmaschinen zu simulieren. Dies ist u.a. beim Übergang zu neuen Modellen von wirtschaftlicher Bedeutung, um während der Zeit der Umrüstung auf die neue Maschine die Programme des Vorgängermodells auch auf dem Nachfolgemodell laufen lassen zu können. Man spricht von Emulation des Vorgängermodells.

---

[1] Dies trifft für Rechner zu, die kein Mikroprogrammwerk benötigen; sie lassen sich physisch in Programmwerk und Datenwerk zerlegen (vgl. Bild 2.31 mit RAMs genügend hoher Kapazität für Programme und Daten und Schaltnetzen genügend mächtiger Operationen zur Zustands- und Datentransformation).

Bild 2.34 erweckt den Eindruck, die Mikroprogramme seien zusammen mit den Maschinenprogrammen und den Daten gemeinsam in einem Randomspeicher untergebracht. Solange jedoch genügend schnelle Randomspeicher mit hoher Kapazität nicht herstellbar oder nicht wirtschaftlich sind, wird der Mikroprogrammspeicher als kleinerer, deutlich schnellerer Randomspeicher oder als schneller Festspeicher aufgebaut. Physisch wird er dann nicht mehr zum Speicher, sondern zum Prozessor gezählt. Damit ergeben sich die in Bild 2.35 dargestellten Möglichkeiten der physischen

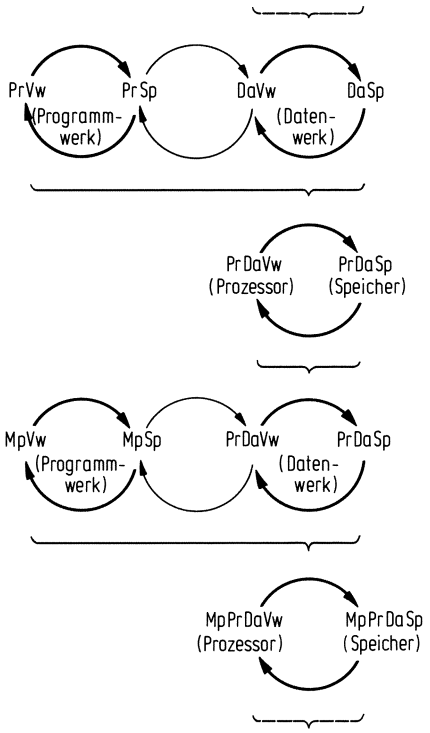


Bild 2.34 Aufteilung eines mikroprogrammierten Digitalrechners in Prozessor und Speicher

Aufteilung eines Digitalrechners in einen Prozessor mit booleschem bzw. sequentiellem Mikroprogrammwerk und einen Speicher für die Programme und die Daten. Dieser besteht aus einem "internen" Randomspeicher (Primärspeicher) und "externen", in Bild 2.35 nicht gezeichneten, peripheren Speichern (Sekundärspeicher). Solange Primärspeicher und Sekundärspeicher ausschließlich von ein und demselben Mikroprogrammwerk gesteuert werden, haben wir eine Rechenanlage vor uns, die aus einem Prozessor und mehreren Speichern besteht. Wenn jedoch z.B. die Übertragung von Programmen und Daten vom Sekundärspeicher in den Primärspeicher und umgekehrt von besonderen Mikroprogrammwerken gesteuert wird, haben wir bereits eine Rechenanlage vor uns, die aus mehreren Prozessoren und mehreren Speichern besteht.

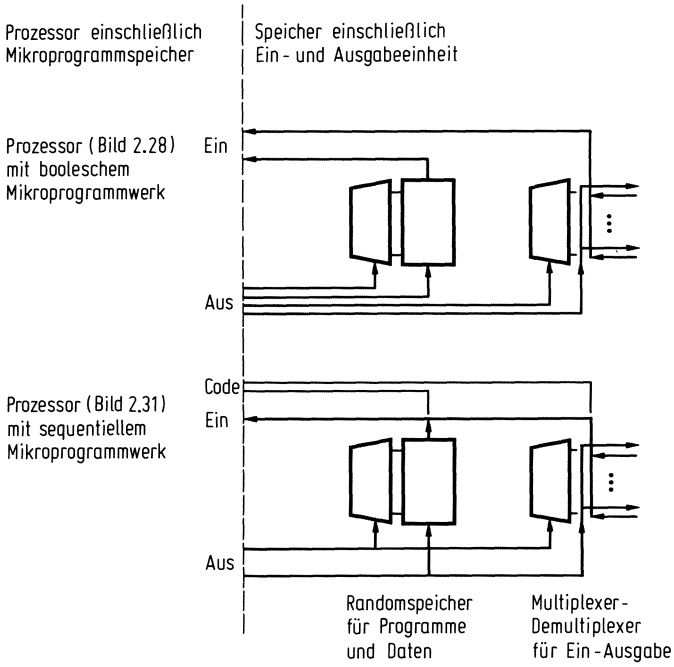


Bild 2.35 Struktur von Digitalrechnern mit booleschem bzw. mit sequentiellem Mikroprogrammwerk

In einem abschließenden Beispiel soll eine Möglichkeit gezeigt werden, wie man Rechenanlagen mit sog. Mikroprozessoren aufbauen kann. Diese Mikroprozessoren sind entweder einfache Prozessoren einschließlich Mikroprogrammspeicher, an die lediglich Primär- und Sekundärspeicher angeschlossen werden müssen, oder sie bestehen nur aus dem Programm-Daten-Verarbeitungswerk, um das herum neben Primär- und Sekundärspeichern auch das Mikroprogrammwerk aufzubauen ist. Letztere werden vielfach in "Scheiben" zu 4 Bits (bit slices) angeboten und lassen sich zu Prozessoren mit Wortlängen von Vielfachen von 4 Bits zusammensetzen. Die Bezeichnung Mikroprozessor ist eher aus dem mikrominiaturisierten Aufbau zu erklären als aus ihrer Funktion, Mikroprogramme zu verarbeiten.

Beispiel 2.11 Mikroprozessor

Bild 2.36 zeigt einen Digitalrechner mit einer Wortlänge von 16 Bits, dessen Kern ein Mikroprozessor[1] aus 4 parallelgeschalteten sog. Chips mit je 4 Bits bildet. Die Struktur des Mikroprozessors ist bereits seiner Funktion als Emulator eines Digitalrechners angepaßt. Primitive arithmetische Operationen können durch Mikrobefehle in einem Schritt ausgeführt werden, und zur effizienten Mikroprogrammierung höherer

[1] Vgl. Mikroprozessor MMI 6701 von Monolithic Memories.

arithmetischer Operationen sind spezielle Schaltnetze und Register vorgesehen. An die beiden Ein-Ausgabebusse des Mikroprozessors sind der Primärspeicher und der Multiplexer-Demultiplexer für die Peripheriegeräte angeschlossen. Mikroprozessor, Speicher und Peripheriegeräte bilden das Datenwerk der Rechenanlage. Sein Mikroprogrammwerk besteht aus dem Festspeicher zum Aufbewahren der Mikrobefehle, einem Register zum Zwischenspeichern ihrer Folgeadressen, einem Multiplexer zum Zusammenfassen der Mikrobedingungen und einem weiteren Festspeicher, in dem Startadressen der Mikroprogramme für die einzelnen Maschinenbefehle aufbewahrt werden.

Soll unser in Abschnitt 2.1 beschriebener Einadreßrechner auf dieser Anlage emuliert werden, so sind die Register und Flipflops des Rechners und die Schaltnetze fast vollständig im RAM bzw. in der ALU des Mikroprozessors zusammenzufassen. Die Mikroprogramme für den Befehls- und Operandenabruf (Bild 2.2) sowie die Befehlsausführungen (u.a. Bilder 2.3, 2.4, 2.5) sind der in Bild 2.36 gezeigten Konfiguration anzupassen.

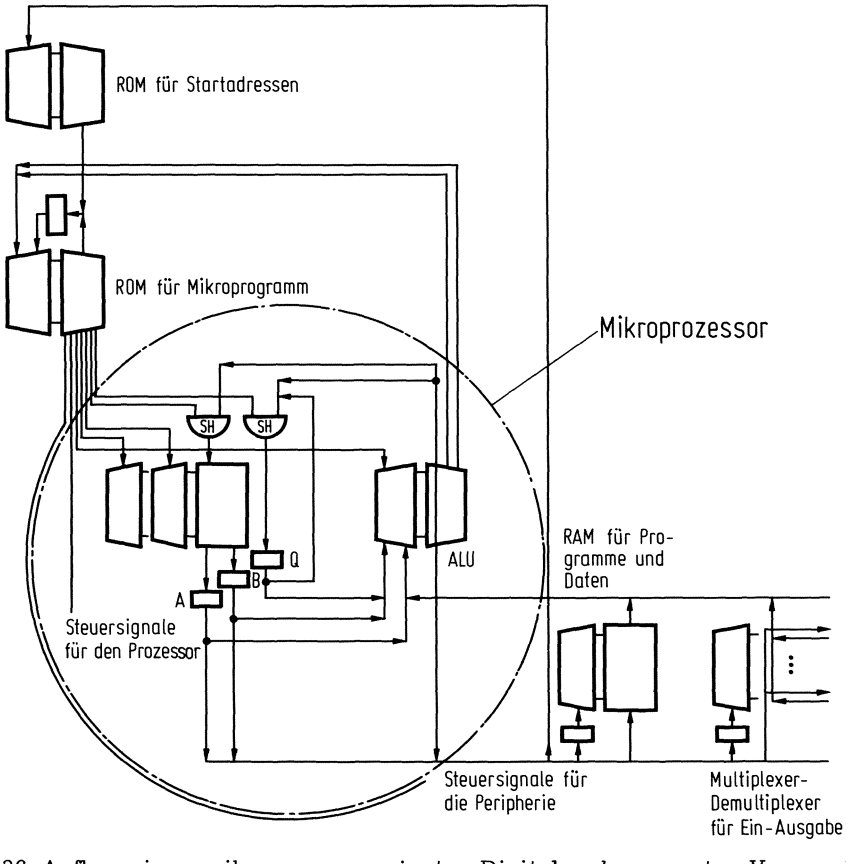


Bild 2.36 Aufbau eines mikroprogrammierten Digitalrechners unter Verwendung eines Mikroprozessors

Bild 2.36 zeigt nur eine von vielen Möglichkeiten, einen Mikroprozessor zu einem Digitalrechner zu erweitern. Man kann weniger als in Bild 2.36 gezeigt, aber auch mehr um den Mikroprozessor herumbauen. Zum Beispiel können die Startadressen auch im ROM für die Mikroprogramme untergebracht werden (entspricht indirekter Adressierung innerhalb des Festspeichers). Zum Beispiel können aber auch Befehlszähler und Befehlsregister gesondert aufgebaut werden und unmittelbar an den RAM für Programme und Daten angeschlossen werden (entspricht Leitwerk außerhalb des Mikroprozessors). ■

3. Rechnersoftware

Als Software[1] bezeichnet man die Gesamtheit der Programmeinheiten (program units) eines digitalen Systems. Software ist veränderbar, ohne daß die physikalische Konstruktion des Systems verändert zu werden braucht.

Die Rechnersoftware wird stärker von den Rechneranwendungen beeinflußt, weniger von der Rechnertechnologie. Das kommt deutlich im Adjektiv der Bezeichnung problemorientierte Sprachen zum Ausdruck, mit denen Rechenanlagen heute programmiert werden. Diese Sprachen gestatten es, Problemlösungen algorithmisch zu formulieren und nach einer Reihe von Vorbereitungsschritten von der Rechnerhardware ausführen zu lassen. Gegenüber den in der Mathematik gebräuchlichen Formeln zur Beschreibung einer Rechnung durch Beziehungen, die zwischen Größen bestehen, ermöglichen diese algorithmischen Sprachen die Beschreibung des Ablaufs einer Rechnung durch Handlungen, die mit Größen vorgenommen werden. Mit solchen künstlichen Sprachen ist ein neues Ausdrucksmittel entstanden, das weit über das Gebiet der Programmierung digitaler Rechenanlagen hinausreicht.

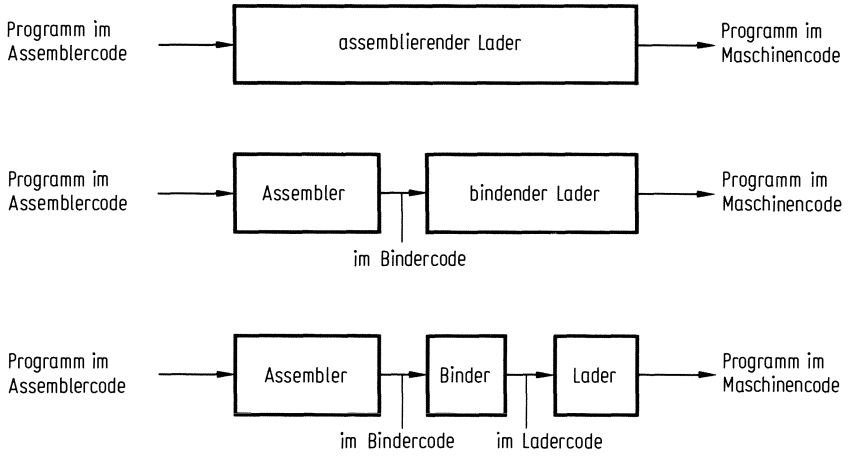
Jedoch standen diese Erkenntnisse nicht am Anfang der Entwicklung problemorientierter Sprachen. Den Anstoß dazu gaben wohl die Rechneranwender, die sich bei der Problemlösung durch ihr Instrument eher behindert als unterstützt sahen. So ist das Codieren in der Maschinensprache nicht nur eine höchst unbefriedigende Tätigkeit, sondern bewirkt durch die extreme Beschränkung der erlaubten Ausdrucksmittel auf Nullen und Einsen eine mangelhafte Übersichtlichkeit und mannigfaltige Programmierfehler. Ein erster Schritt zur Vermeidung der Darstellung eines Befehls durch 0-1-Kombinationen wurde bei der Programmierung des an der Manchester University gebauten MARK I getan. Die 20 Bits eines Befehls wurden nicht durch 20 Ziffern zur Basis 2, sondern durch 4 "Zeichen" zur Basis 32 dargestellt, die auf einem Lochstreifen von 5 Spuren nacheinander gestanzt vorlagen und beim Eingeben in den Speicher des Rechners aneinandergesetzt wurden. Bei der Programmierung des an der Cambridge University gebauten EDSAC ging man bereits einen Schritt weiter, indem man die Bits eines Befehls entsprechend ihrer Funktion zu Gruppen zusammenfaßte

---

[1] Aus dem Englischen im Gegensatz zu Hardware: soft - weich; ware - Ware.

und codierte. Für den Befehlscode wurde ein Buchstabe und für die Adresse eine Dezimalzahl verwendet. Dies machte bereits einen einfachen Codiervorgang notwendig, bei dem das symbolische Programm in das Maschinenprogramm umgeformt wurde. Aus diesen Ansätzen entwickelten sich einfache Assembler, die auch das Auflösen der Querbezüge von den in den Befehlen stehenden Adressen und den symbolisch adressierten Speicherplätzen ermöglichten.

Die ersten Assembler übersetzten symbolische Programme, die z.B. auf einem Lochstreifen codiert waren, in den Maschinencode, der direkt im Speicher abgelegt wurde. Dieses Vorgehen impliziert jedoch, daß auch umfangreiche symbolische Programme nur in einem Stück und nicht etwa in Teilen assembliert werden können und daß bereits im Maschinencode vorhandene Programme nicht in das symbolische Programm mit einbezogen werden können. Spätere Assembler formen deshalb symbolische Programme in einen Code um, der erst von einer Art Nachassembler, dem sog. bindenden Lader, in den Maschinencode umgeformt und im Speicher abgelegt wird. Diese Trennung des Assemblierens und nachträglichen Bindens von Programmteilen zu einem im Speicher stehenden, vollständigen, ablauffähigen Maschinenprogramm erfordert das Zwischenspeichern assemblierter Programme in einem Zwischencode, dem sog. Bindercode. Ein aus Assembler und bindendem Lader bestehendes Programmsystem impliziert jedoch, daß auch ein lange laufendes Maschinenprogramm nur unmittelbar nach dem Binden und nicht zu einem späteren Zeitpunkt, etwa wenn die Maschine nicht ausgelastet ist, ausgeführt werden kann. Heutige Programmsysteme trennen deshalb vielfach das Binden und das Laden, so daß auch das gebundene Programm in einem Zwischencode, dem sog. Ladercode, zwischengespeichert werden muß. Programme werden also zu unterschiedlichen Zeitpunkten vom Assembler in den Bindercode, vom Binder in den Ladercode und vom Lader in den Maschinencode umgeformt, in dem sie als lauffähiges Programm ausgeführt werden. Die folgende Abbildung zeigt



die hier angedeuteten Phasen der Entwicklung vom "assemblierenden Lader" zu einem Programmsystem aus Assembler, Binder und Lader. Für die verschiedenen in der Abbildung eingetragenen Codes eines Programmes sind auch andere Begriffe gebräuchlich, wie z.B. Quellcode für Assemblercode und Objektcode für Bindercode. Durch das Aufteilen der Programmvorbereitung in mehrere Arbeitsgänge müssen diese Codes zwischengespeichert werden, wozu aus Kostengründen langsamere externe Speicher, wie Platten- oder Bandspeicher benutzt werden. Um die dadurch entstehende Speicherhierarchie auch begrifflich zu unterstützen, bezeichnen wir den "internen" Speicher als Primärspeicher und die externen Speicher als Sekundärspeicher.

Die Programmaufbereitung und die Programmausführung wurden zuerst vom Programmierer selbst durchgeführt. Die Situation ist ähnlich der eines Programmierers, der auf einem elektronischen Taschenrechner rechnet: Ein technisches Gerät wird von einem menschlichen Operateur bedient, der auch nicht annähernd die Arbeitsgeschwindigkeit des Gerätes ausnutzen kann. Es erscheint naheliegend, den mit dem Taschenrechner arbeitenden Operateur durch ein Leitwerk zu ersetzen. Genauso naheliegend ist es, den mit einer Rechenanlage arbeitenden Operateur durch ein Betriebsprogramm zu ersetzen. Ein solches Programm ermöglicht es, die Rechnersoftware in ähnlicher Weise durch Betriebsanweisungen wie die Rechnerhardware durch Maschinenbefehle zu benutzen.

In diesem Kapitel beschreiben wir zunächst einen Assembler und einen Lader für den Einadreßrechner aus Kapitel 2 und entfernen uns dann mit der Beschreibung des Binders und der Systemprogramme mehr und mehr vom Detail.

In Abschnitt 3.1 wird ein einfacher Assembler in einer Form beschrieben, wie man sie in vielen Handbüchern (assembler reference manuals) der Rechnerhersteller findet. Zunächst ist die Wirkung einer jeden Assembleranweisung genau beschrieben, ohne auf die Details der einzelnen Abläufe einzugehen. Anschließend wird ein Ausschnitt aus der Funktionsbeschreibung des Assemblers gegeben. Hier findet man die Abläufe einzelner Assemblerfunktionen im Detail. Die Darstellung der Abläufe durch Flußdiagramme ist nicht an eine bestimmte Implementierung gebunden. Sie kann vielmehr als Normdarstellung für seriell ablaufende Handlungen aufgefaßt werden.

In Abschnitt 3.2 werden Lader und Binder als neben dem Assembler wichtigste Bestandteile der Grundsoftware beschrieben. Der Lader ist das elementarste für den Betrieb einer Rechenanlage notwendige Programm. Mit ihm steht der Operateur über das Bedienungsfeld in Verbindung, und aus ihm entwickelt sich der sog. Monitor als Kern des Betriebssystems. Der Binder ermöglicht es, in verschiedenen Programmiersprachen verfaßte Programmteile zu einem ablauffähigen Programm zu vereinigen und bildet damit die Verbindung zu höheren Programmiersprachen.

In Abschnitt 3.3 werden Systemprogramme und Betriebssysteme behandelt. Sie bilden die für die Programmierung und den Betrieb von Rechenanlagen notwendige Software, durch die der Benutzer Zugriff zu einer modernen Rechenanlage erhält. Für ihn ist die Software genau wie die Hardware Bestandteil des aus Hardware und Software bestehenden digitalen Systems.

3.1 Ein einfacher Assembler

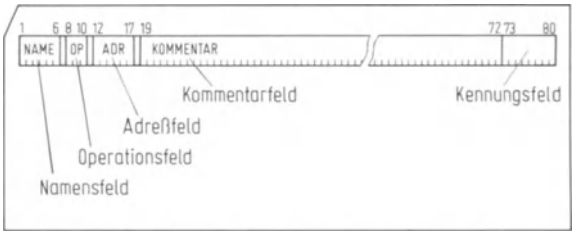
In Abschnitt 1.4 wurden im Rahmen eines Beispiels bereits die wichtigsten Anweisungen beschrieben, die durch einen Assembler verarbeitet werden können, ohne daß jedoch auf die Assemblersprache als ganzes eingegangen wurde. Dies soll hier nachgeholt werden; und wir legen diesem Abschnitt die Befehlsliste des Einadreßrechners aus Abschnitt 2.1 zugrunde. In den Befehlen erscheinen der Operationscode, die Operandenadressen und die Konstanten in symbolischer Schreibweise. Aus diesen Symbolen werden die Zeilen eines Programms gebildet, wobei üblicherweise das Format einer Zeile vorgeschrieben ist. Die möglichen Operationen und die Vorschriften für die Formulierung einer Programmzeile bilden die Assemblersprache eines Rechners.

Eine Assemblersprache

In unserer Assemblersprache unterscheiden wir zwischen symbolischen Maschinenbefehlen, Assembleranweisungen und Kommentaren. Maschinenbefehle (kurz Befehle) sind Operationen für den Prozessor, die während der Programmausführung (run time) ausgeführt werden. Befehle treten nach der Assemblierung in binär codierter Form im Ladercode auf und stehen zur Ausführungszeit im Speicher. Assembleranweisungen (kurz Anweisungen) sind Operationen für den Assembler, die während der Programmassemblierung (assembly time) ausgeführt werden. Sie erzeugen keine Befehlswörter im Ladercode und stehen auch nicht zur Ausführungszeit im Speicher. Einige Anweisungen erzeugen dennoch Ladercode, indem sie Konstanten definieren, Variablen initialisieren oder Speicherplätze reservieren. Kommentare können als Klartext in das symbolische Programm aufgenommen werden. Sie haben keine operative Wirkung und damit weder Einfluß auf die Assemblierung noch auf die Programmausführung. Sie dienen lediglich zur Dokumentation von Programmen.

Format einer Programmzeile. Bei der Festlegung des Formates einer Programmzeile soll davon ausgegangen werden, daß das symbolische Programm auf Lochkarten geschrieben oder als Lochkartenabbild auf einem anderen Speichermedium vorhanden ist. Jede Lochkarte nimmt gerade eine Programmzeile auf, für die 72 der 80 Lochkartenspalten zur Verfügung stehen. Da sich die Befehle und Anweisungen aus Teilen mit bestimmter Bedeutung wie Name, Operation, Adresse und Kommentar zusammensetzen, zerlegen wir das den ersten 72 Spalten entsprechende Feld einer Lochkarte in

Teilfelder wie folgt: Namensfeld, Operationsfeld, Adreßfeld und Kommentarfeld. Die Teilfelder werden durch je eine Leerspalte voneinander getrennt. Die Spalten 73 bis 80 bilden ein Kennungsfeld zur Lochkartenidentifikation. Sie werden vom Assembler nicht verarbeitet.



Das Namensfeld belegt die Spalten 1 bis 6. Es ist entweder leer oder enthält ein Symbol von maximal sechs Buchstaben oder Ziffern, wobei das erste Zeichen ein Buchstabe sein muß. Es erlaubt dem Programmierer, innerhalb seines Programms einen Bezug auf ein so markiertes Befehls- oder Datenwort oder eine Anweisung herzustellen. Das Operationsfeld belegt die Spalten 8 bis 10 und enthält den mnemonischen Operationscode eines Maschinenbefehls oder einer Assembleranweisung. Die Mnemone sind festgelegte Buchstabenkombinationen. Das Adreßfeld belegt die Spalten 12 bis 17. Es enthält entweder ein Symbol als symbolische Adresse oder eine Zahl als numerische Adresse oder Konstante. Ist das Adreßfeld leer, so ergänzt es der Assembler mit der numerischen Adresse bzw. der Konstanten 0. Sämtliche Eintragungen ins Namens-, Operations- und Adreßfeld müssen linksbündig stehen. Das Kommentarfeld belegt die Spalten 19 bis 72. Es kann beliebige Zeichenketten, die aus dem verfügbaren Zeichenvorrat gebildet werden können, aufnehmen. Programmzeilen, die nur aus Kommentar bestehen, müssen mit dem Sonderzeichen * in der Spalte 1 beginnen, um sie von den Maschinenbefehlen und Assembleranweisungen unterscheiden zu können.

Zur Darstellung der symbolischen Sprachelemente wird ein Zeichenvorrat, bestehend aus den Großbuchstaben A bis Z, den Ziffern 0 bis 9 und einigen Sonderzeichen verwendet. Namen werden, wie bereits beschrieben, aus Buchstaben und Ziffern (alphanumerische Zeichen) gebildet. Konstanten werden wahlweise in Dezimal- oder Sedezimalschreibweise angegeben. Der Zahlenbereich ist durch die Wortlänge von 16 Bits auf den Bereich -32768 bis +32767 begrenzt. Eine Dezimalzahl wird aus den Ziffern 0 bis 9 mit einem Vorzeichen + bzw. - oder ohne Vorzeichen gebildet. Eine Sedezimalzahl setzt sich aus den Sedezimalziffern 0,...,9,A,...,F[1] zusammen und wird zur Unterscheidung von Dezimalzahlen in zwei Hochkommas eingeschlossen und durch

[1] Sedezimalzahl ist eine Zahl des Zahlensystems mit der Basis 16.

ein vorangestelltes X gekennzeichnet. Sie umfaßt 4 Ziffern, wobei jede Ziffer 4 Bits eines Speicherworts belegt. Eine zu kurze bzw. zu lange Sedezimalzahl wird vom Assembler linksseitig mit Nullen aufgefüllt bzw. abgeschnitten.

Assembleranweisungen. Als Assembleranweisungen übernehmen wir die vier Anweisungen mit den mnemonischen Codes DAT, RES, ORG und END aus Abschnitt 1.2 und führen zusätzlich eine Anweisung mit dem Code EQU ein. Da die Bildungsvorschriften des Namens- und Adreßfeldes dieser Anweisungen nicht einheitlich sind, werden wir sie für jede Anweisung getrennt definieren. Symbole (s) oder Konstanten (c), die weggelassen werden können, werden in eckige Klammern gesetzt. Alternativen werden durch einen senkrechten Strich gekennzeichnet. Im Anschluß an die folgenden Anweisungsbeschreibungen sind drei Beispiele angegeben, die die Wirkung der Assembleranweisungen im Zusammenhang mit Maschinenbefehlen illustrieren.

[s] DAT c|s (generate data) generiert eine Konstante oder eine symbolische Adresse. Sie belegt im Maschinencode ein Wort, das die Konstante bzw. die numerische Adresse als Dualzahl enthält. Der Bezug von einem Befehl oder von einer anderen Anweisung auf dieses Rechnerwort kann über ein Symbol im Namensfeld der DAT-Anweisung hergestellt werden. Dieses Symbol ist jedoch nicht unbedingt erforderlich, da die entsprechende Speicherzelle auch über eine programmierte Adreßrechnung angesprochen werden kann.

[s] RES c (reserve) reserviert Speicherplatz für Variablen. Die Anzahl der zu reservierenden Speicherzellen ist durch die Konstante c festgelegt. Sie wird als Zusatzinformation für den Lader in den Ladercode aufgenommen. Nach dem Ladevorgang bleibt der Inhalt der reservierten Speicherzellen zunächst unbestimmt. Ein Symbol im Namensfeld hat, wenn nur eine Zelle reserviert wird, die gleiche Funktion wie bei der DAT-Anweisung. Wird mehr als eine Speicherzelle reserviert, so bezeichnet das Namenssymbol die erste von aufeinanderfolgenden Speicherzellen, d.h. die erste Zelle eines Feldes. Die weiteren Zellen des Feldes können über eine programmierte Adreßrechnung erreicht werden.

s EQU c|s (equate) ordnet dem Symbol im Namensfeld den Wert des Adreßfeldes zu. Das Adreßfeld enthält entweder eine Konstante oder ein Symbol. Ein Symbol ist nur dann zulässig, wenn es im Namensfeld einer der EQU-Anweisung vorangehenden Anweisung bereits aufgetreten ist. Sowohl das Namensfeld als auch das Adreßfeld müssen in jedem Fall belegt sein. Die EQU-Anweisung ist eine reine Assembleranweisung; sie erzeugt keinen Ladercode. Die EQU-Anweisung kann z.B. dazu verwendet werden, den numerischen Geräteadressen Namen zuzuordnen.

[s] ORG c (origin) gibt dem Assembler im Adreßfeld die Ladeadresse an, ab der die nachfolgenden Wörter in den Speicher geladen werden sollen. Die Konstante im Adreßfeld ist somit die Speicheradresse des ersten auf ORG folgenden Wortes. Die Adressen sämtlicher nachfolgender Wörter werden vom Assembler durch Addition der Ladeadresse ermittelt. Enthält ein Programm mehrere ORG-Anweisungen, so erfolgt die Adressierung zwischen zwei ORG-Anweisungen relativ zur Ladeadresse der ersten ORG-Anweisung. Ist ein Symbol im Namensfeld angegeben, so wird es als symbolische Ladeadresse interpretiert. Es hat die gleiche Funktion wie ein Symbol im Namensfeld der auf ORG folgenden Programmzeile. Die ORG-Anweisung selbst erzeugt keinen Maschinencode, sie fügt lediglich die Ladeadresse dem Ladercode als Information für den Lader bei.

END [s] (end) zeigt dem Assembler das Ende des zu assemblierenden Programms an. Programmzeilen, die nach der END-Anweisung stehen, werden vom Assembler nicht verarbeitet. Die symbolische Adresse im Adreßfeld gibt die Startadresse des Programms an; mit ihr erfolgt die Initialisierung des Befehlszählers. Diese Adresse steht im allgemeinen als Marke im Namensfeld des ersten Befehls des Programms. Ist das Adreßfeld leer, so wird vom Assembler die in der ersten ORG-Anweisung definierte Ladeadresse als Startadresse angenommen. Die END-Anweisung erzeugt keinen Maschinencode, sie ergänzt jedoch den Ladercode durch Angabe der in den Befehlszähler zu ladenden Startadresse.

Beispiel 3.1 Summieren der Zahlen 1 bis N

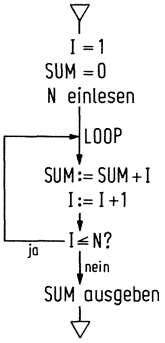
Im folgenden Programm sollen die ganzen Zahlen 1 bis N summiert und das Ergebnis über eine Konsolschreibmaschine mit der symbolischen Geräteadresse TYP ausgegeben werden. Der Wert für N soll zu Anfang über diese Schreibmaschine eingelesen werden[1]. Zusätzlich zum Assemblerprogramm ist der Algorithmus als Flußdiagramm dargestellt. Dieses Programmbeispiel stellt die Funktionsweise der Assembleranweisungen und einiger Maschinenbefehle vor, wobei auch eine der Verwendungsmöglichkeiten für die EQU-Anweisung gezeigt wird. Die Geräteadresse 2 des Ein-Ausgabegerätes wird durch EQU dem Symbol TYP zugeordnet. Damit kann das Ein-Ausgaberät in den Befehlen INP und OUT symbolisch adressiert werden. Soll das Programm z.B. mit einem anderen Ein-Ausgabegerät ablaufen, so braucht nur die absolute Geräteadresse in der EQU-Anweisung geändert und das Programm erneut assembliert zu werden, während die Ein-Ausgabebefehle unverändert bleiben, was vor allem bei größeren Programmen mit vielen Ein-Ausgabebefehlen nützlich ist.

---

[1] Der Übersichtlichkeit halber sind die Ein-Ausgabevorgänge vereinfacht dargestellt und die dabei notwendigen Umwandlungen der Zahlen vom externen Dezimalcode in den internen Dualcode und umgekehrt nicht berücksichtigt.

Flußdiagramm

SUM := 0+1+2+...+N



Assemblerprogramm

```
* SUMMIEREN DER ZAHLEN 1 BIS N
* ============================
        ORG 36      WILLKUERLICHE LADEADRESSE
I       DAT 1       VARIABLE MIT I=1 INITIALISIERT
EINS    DAT 1       KONSTANTE EINS MIT DEM WERT 1
SUM     DAT 0       VARIABLE MIT SUM=0 INITIALISIERT
N       RES 1       VARIABLE N
TYP     EQU 2       SYMBOLISCHE GERAETEADRESSE
*
START   INP TYP
        STA N       N EINLESEN
LOOP    LDA SUM
        ADD I
        STA SUM     SUM:=SUM+I
        LDA I
        ADD EINS
        STA I       I:=I+1
        CMP N       WENN
        NOP         I< N
        LDP LOOP    I = N, WEITER BEI LOOP
        LDA SUM     I>N, SUM AUSGEBEN
        OUT TYP
        HLT         PROGRAMMHALT
        END START   ENDE DER ASSEMBLIERUNG
```

Das Programm ist so organisiert, daß zuerst der Datenbereich und dann der Befehlsbereich angelegt wird. Der Programmstart ist durch den Adreßteil der END-Anweisung auf den als ersten auszuführenden Maschinenbefehl - hier durch das Symbol START bezeichnet - festgelegt. Daten- und Befehlsbereich belegen den Primärspeicher fortlaufend ab der Ladeadresse 36.

Das Programmbeispiel zeigt die Möglichkeit, eine Programmschleife zu bilden. Der Laufindex I der Schleife wird durch die Anweisung I DAT 1 initialisiert und durch die Befehle LDA I und ADD EINS mit jedem Durchlauf um 1 erhöht. Die Abfrage des Schleifenendes, d.h. der Vergleich von I mit der Anzahl N der geforderten Durchläufe, erfolgt durch den Vergleichsbefehl CMP N, wozu der Wert von I zuvor in den Akkumulator gebracht wird. Mit LDP LOOP erfolgt bei $I \leqslant N$ ein Rücksprung zum Schleifenanfang, der mit LOOP markiert ist. Der auf CMP folgende NOP-Befehl bewirkt die Vereinigung der Kriterien $I < N$ und $I = N$ zu $I \leqslant N$. ■

Beispiel 3.2 Programmliste

Für das Programm aus Beispiel 3.1 ist im folgenden die Programmliste angegeben, die beim Assemblieren des symbolischen Programms ausgegeben werden kann. Sie dient dem Programmierer zur Kontrolle und enthält neben dem symbolischen Programm auch die numerischen Adressen und die dazugehörigen Inhalte der einzelnen Speicherzellen in sedezimaler Darstellung. Treten Fehler beim Assemblieren auf, z.B. bei der Verwendung nicht definierter Symbole, so wird dies ebenfalls in der Liste angezeigt.

```
                1   *SUMMIEREN DER ZAHLEN 1 BIS N
                2   * ==========================
        011     3          ORG 36     WILLKUERLICHE LADEADRESSE
024     0001    4   I      DAT 1      VARIABLE MIT I=1 INITIALISIERT
025     0001    5   EINS   DAT 1      KONSTANTE EINS MIT DEM WERT 1
026     0000    6   SUM    DAT 0      VARIABLE MIT SUM=0 INITIALISIERT
027             7   N      RES 1      VARIABLE N
                8   TYP    EQU 2      SYMBOLISCHE GERAETEADRESSE
                9   *
028     6002    10  START  INP TYP
029     4027    11         STA N      N EINLESEN
02A     8026    12  LOOP   LDA SUM
02B     B024    13         ADD I
02C     4026    14         STA SUM    SUM:=SUM+I
02D     8024    15         LDA I
02E     B025    16         ADD EINS
02F     4024    17         STA I      I:=I+1
030     F027    18         CMP N      WENN
031     0000    19         NOP        I<N
032     102A    20         LDP LOOP   I=N, WEITER BEI LOOP
033     8026    21         LDA SUM    I>N, SUM AUSGEBEN
034     7002    22         OUT TYP
035     2000    23         HLT        PROGRAMMHALT
        028     24         END START  ENDE DER ASSEMBLIERUNG
```

Der im Sedezimalcode ausgedruckte Speicherinhalt stellt gleichzeitig mit geringfügingen Abweichungen den Ladercode dar. In der ersten Zeile ist für den Lader die Anzahl der einzulesenden Speicherwörter in sedezimaler Form angegeben. Dazu zählen die Datenwörter und die Maschinenbefehle. Die letzte Zeile enthält die Startadresse. Der im Sedezimalcode ausgedruckte Speicherinhalt ist bis auf die erste und die letzte Zeile mit dem Maschinencode indentisch, wenn der Ladercode von einem Absolutlader in den Speicher gelesen wird (vgl. Abschnitt 3.2: Urlader und Absolutlader). ■

Beispiel 3.3 Vektoraddition

Es soll ein Programmstück geschrieben werden, das innerhalb einer größeren Rechnung die Addition $\mathbf{x} := \mathbf{x}+\mathbf{y}$ von zwei Vektoren $\mathbf{x}$ und $\mathbf{y}$, also die Addition der Werte von zwei eindimensionalen Feldern X und Y durchführt. Dazu wird die Vektoraddition in eine Serie von Additionen korrespondierender Elemente aufgelöst. Der Einfachheit halber sollen die Werte der Vektoren ganze Zahlen sein. Wir nehmen an, daß die Werte der Felder X und Y in aufeinanderfolgenden Speicherzellen stehen und nicht mehr als 100 Elemente umfassen. Die Feldlänge N soll den Feldern vorangestellt sein.

Im folgenden Programm wird die Addition der beiden Felder X und Y in einer Programmschleife ausgeführt. Da die Feldelemente X(I), Y(I) für $I > 0$ nicht direkt adressiert werden können, muß bei jedem Schleifendurchlauf eine Adreßrechnung aus-

geführt werden. Dadurch kommt zum eigentlichen Programm noch eine Reihe von Organisationsbefehlen hinzu, die auch Speicherplatz und Ausführungszeit kosten. Hier zeigen sich die Grenzen des in Kapitel 2 definierten Rechners. Eine effizientere Feldverarbeitung kann nur erfolgen, wenn sie von der Hardware unterstützt wird (s. Abschnitt 4.2: Indexadressierung).

```
*VEKTORADDITION
*==============
       ORG 0
N      RES 1      FELDLAENGE N
X      RES 100    FELD X
Y      RES 100    FELD Y
I      RES 1      INDEX
NULL   DAT 0      KONSTANTE
EINS   DAT 1      KONSTANTE
        :
START   :
        :
       LDA NULL
       STA I      INDEX I:=0
BFHL1  LDA X
BFHL2  ADD Y
BFHL3  STA X      X(I):=X(I)+Y(I)
       LDA BFHL1
       ADD EINS
       STA BFHL1  ADRESSE IN BEFEHL 1 UM 1 ERHOEHEN
       LDA BFHL2
       ADD EINS
       STA BFHL2  ADRESSE IN BEFEHL 2 UM 1 ERHOEHEN
       LDA BFHL3
       ADD EINS
       STA BFHL3  ADRESSE IN BEFEHL 3 UM 1 ERHOEHEN
       LDA I
       ADD EINS
       STA I      INDEX UM 1 ERHOEHEN
       CMP N      WENN INDEX I
       LDP BFHL1  <N, DANN WEITER BEI BFHL1
        :
       END START
```

■

Funktionsweise des Assemblers

Die folgende Beschreibung legt einen sog. 2-Phasen-Assembler zugrunde, d.h. einen Assembler, der den Quellcode zweimal durchläuft. Die erste Phase umfaßt eine lexikalische und syntaktische Analyse der einzelnen Programmzeilen und die Bestimmung der Adreßbezüge. In der zweiten Phase findet die Codegenerierung statt.

Lexikalische und syntaktische Analyse. Die lexikalische und syntaktische Analyse wird vom Scanner (Abtaster) - einer Funktionseinheit des Assemblers - durchgeführt. Der Scanner liest dazu das symbolische Programm z.B. von einem Lochkar-

tenleser in den Primärspeicher, wo er Befehle und Anweisungen einzeln verarbeitet. Er tastet sie Zeichen für Zeichen ab und faßt die Zeichenketten zu Symbolen, Mnemonen und Konstanten zusammen; der Kommentar wird überlesen. Der Scanner benutzt dazu eine Zeichentabelle, in der der gesamte Zeichenvorrat z.B. in EBCDI-codierter Form gespeichert ist. Zu jedem Zeichen sind zusätzlich Kennbits zur Unterscheidung der Zeichengruppen Buchstabe, Ziffer, Sonderzeichen bzw. die dualen Entsprechungen der Dezimal- und Sedezimalziffern angegeben. Die Symbole, Mnemone, Konstanten und deren spaltenweise Anordnung werden auf syntaktische Richtigkeit überprüft. Die Überprüfung der Mnemone erfolgt durch Vergleich mit dem Inhalt der Zuordnungstabelle, in der sämtliche Mnemone der Assemblersprache und deren Darstellung im Maschinencode gespeichert sind. Konstanten werden vom Scanner von der symbolischen Darstellung in entsprechende Dualzahlen umgewandelt. Ein leeres Adreßfeld wird als 0 interpretiert. Fehlerhafte Schreibweisen werden in einer Fehlertabelle registriert und später in Form einer Fehlerliste ausgegeben.

Bild 3.1 zeigt einen Ausschnitt aus dem Abtastvorgang, bei dem das Namensfeld einer Programmzeile analysiert wird. Enthält das Namensfeld ein Symbol, so wird

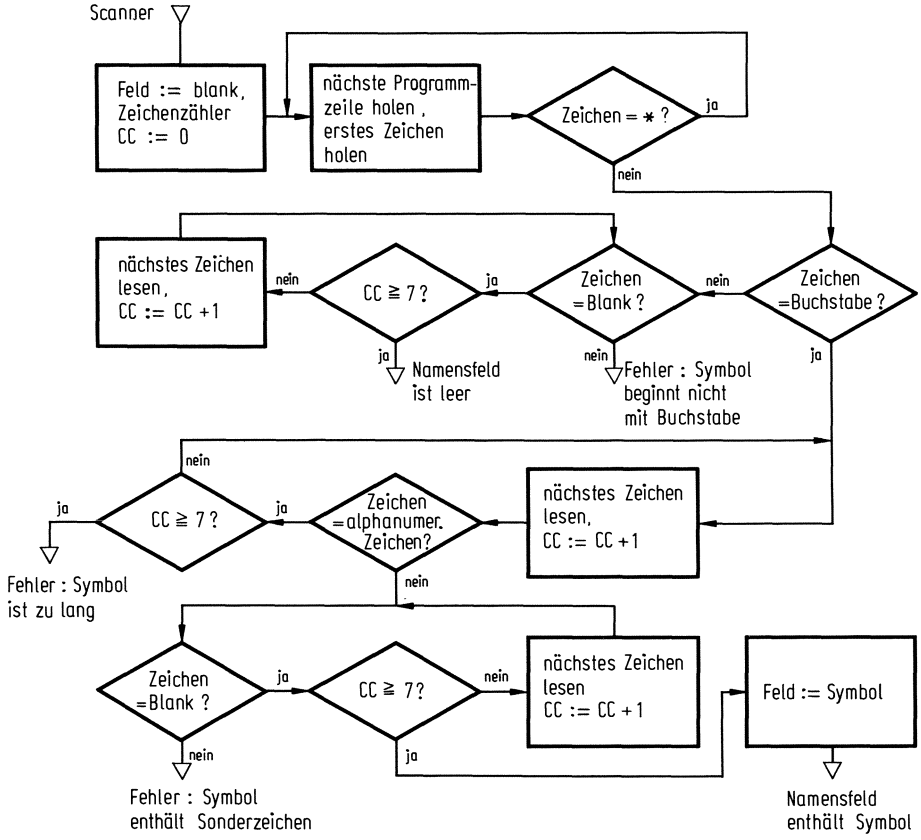


Bild 3.1 Analyse des Namensfeldes durch den Scanner in Phase 1

das Symbol zur Weiterverarbeitung in der Phase 1 in einem zuvor mit Blanks beschriebenen Speicherbereich FELD abgelegt. Ist kein Symbol vorhanden, bleibt der Bereich FELD unverändert. Ein fehlerhaftes Symbol verursacht eine Eintragung in die Fehlertabelle. In jedem Fall wird der Abtastvorgang mit der Analyse des Operationsfeldes fortgesetzt.

Bestimmung der Adreßbezüge. Um die Zuordnung zwischen den symbolischen Adressen eines Benutzerprogramms und den numerischen Speicheradressen herzustellen, benutzt der Assembler einen Zuordnungszähler und baut die Symboltabelle auf. Der Zuordnungszähler wird zu Beginn der Phase 1 auf den im Adreßfeld der ORG-Anweisung angegebenen Wert gesetzt. Während der Phase 1 wird dieser Zähler nach jeder übersetzten Programmzeile um die Anzal der sich daraus ergebenden Maschinencodewörter weitergezählt. Bei der Übersetzung eines Befehls oder einer Anweisung erhalten diese als Adresse den augenblicklichen Wert des Zuordnungszählers. Steht ein Symbol im Namensfeld, so wird ihm der Zählerstand als Wert zugeordnet. Dieses Symbol gilt damit als definiert. Es wird zusammen mit seinem Wert in die Symboltabelle eingetragen. Ein Symbol, das zuerst im Adreßfeld auftritt, gilt zunächst nur als verwendet, da ihm noch kein Wert zugewiesen werden kann. Es wird in die Symboltabelle eingetragen und erst dann durch eine numerische Adreßangabe ergänzt, wenn es im Namensfeld auftritt. Die Möglichkeit der Assemblerprogrammierung, ein Symbol zu verwenden bevor es definiert ist, erfordert eine Assemblierung in zwei Phasen[1]. Eine Ausnahme bei der Symboldefinition bildet die EQU-Anweisung. Bei ihr wird der Wert des Namens durch eine Konstante oder durch den Wert eines bereits definierten Symbols im Adreßfeld der EQU-Anweisung bestimmt.

Bild 3.2 zeigt einen Ausschnitt der Symboltabelle, die pro Eintragung sechs alphanumerische Zeichen eines Symbols z.B. in EBCDI-codierter Form, den Adreßwert

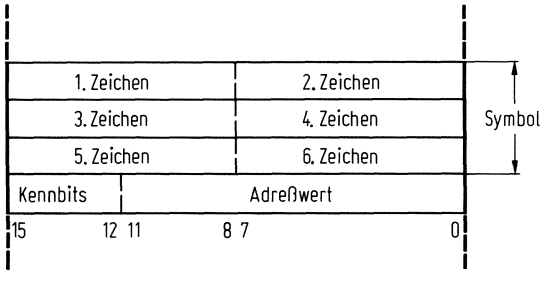


Bild 3.2 Ausschnitt aus einer Symboltabelle

[1] Prinzipiell ist hier auch eine einphasige Assemblierung möglich. Dazu müssen jedoch im Maschinencode die noch offenen Adreßbezüge über Verweise referenziert und nachträglich in einer Rückwärtsersetzung hergestellt werden.

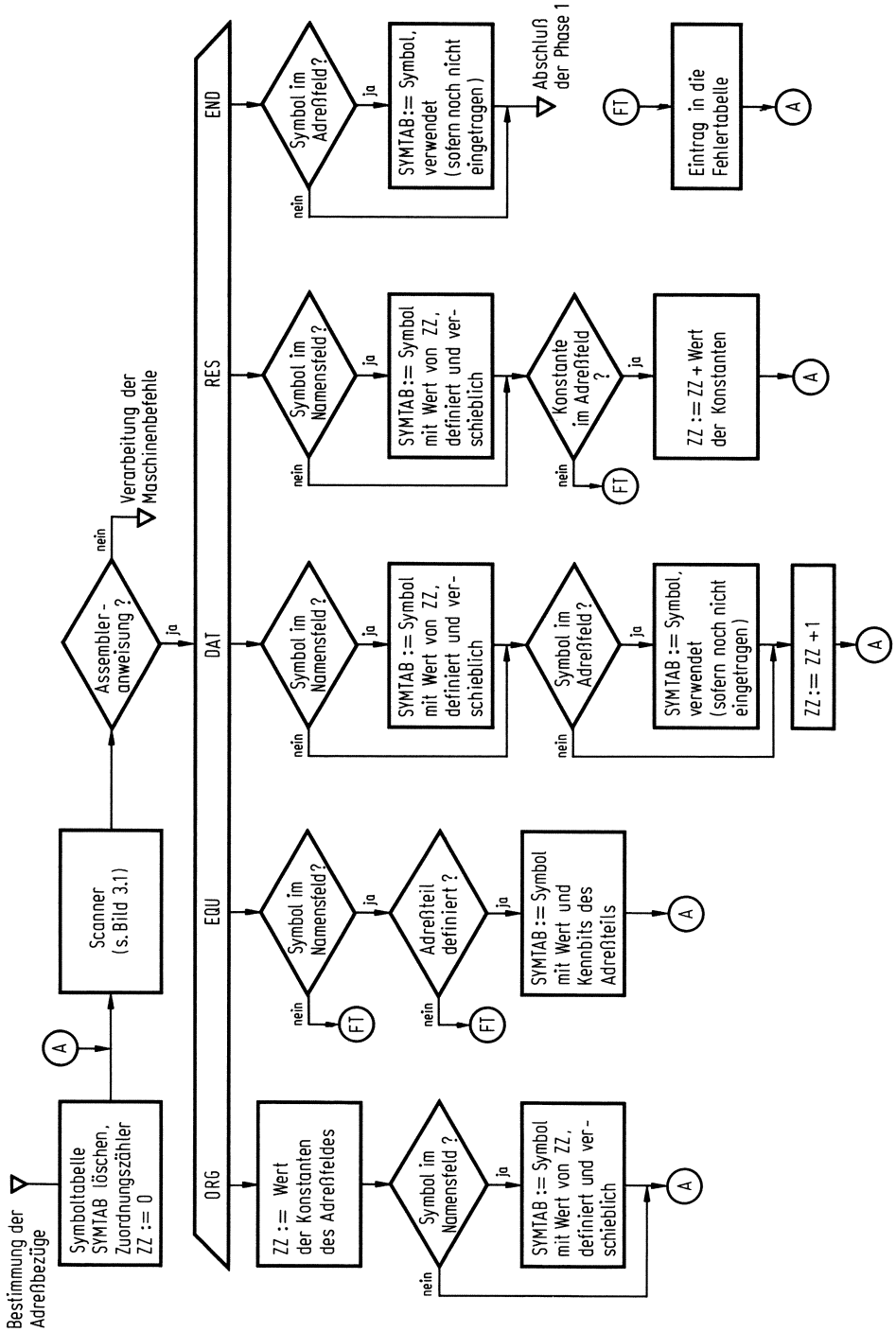


Bild 3.3 Bestimmung der Adreßbezüge bei den Assembleranweisungen

des Symbols und einige Kennbits aufnimmt. Eines der Kennbits stellt die Information definiert/verwendet dar. Ein zweites Kennbit zeigt an, ob der Wert eines Symbols "fest" (fixed) oder "verschieblich" (relocatable) ist. Verschieblich sind sämtliche Adressen, die sich auf den Zuordnungszähler beziehen, also relativ zur Ladeadresse gebildet werden. Man bezeichnet sie deshalb auch als relative Adressen. Feste Adressen sind absolute Adressen, die durch Konstanten dargestellt werden. Eine symbolische Adresse ist in unserer Assemblersprache nur dann eine feste Adresse, wenn ihr durch eine EQU-Anweisung eine Konstante zugewiesen wird. Die Verschieblichkeitsinformation ist für das spätere Laden des Maschinenprogramms in den Primärspeicher von Bedeutung und deshalb Bestandteil des Ladercodes (s. Abschnitt 3.2: Verschiebender Lader).

Bild 3.3 zeigt die Bestimmung der Adreßbezüge am Beispiel der fünf Assembleranweisungen ORG, EQU, DAT, RES und END. Eine fehlerhafte Anweisung verursacht eine Eintragung in die Fehlertabelle FT. Nach der Verarbeitung einer Anweisung erfolgt ein Sprung zum Punkt A, worauf der Scanner die nächste Programmzeile liest und analysiert. Die Verzweigung auf die Verarbeitung der Assembleranweisungen bzw. der Maschinenbefehle findet nach dem Abtastvorgang statt. Maschinenbefehle werden analog zur DAT-Anweisung verarbeitet.

Zum Abschluß der Phase 1, d.h. nach Verarbeitung der END-Anweisung, wird die Symboltabelle SYMTAB nach eventuell noch nicht erfüllten Adreßbezügen durchsucht. Diese bewirken eine Eintragung in die Fehlertabelle. Mit Abschluß der Phase 1 kann die Fehlerliste z.B. über einen Schnelldrucker ausgegeben werden.

Codegenerierung. Im Hinblick auf die begrenzte Primärspeicherkapazität werden im Verlauf der ersten Phase die einzelnen Programmzeilen zusammen mit den daraus ermittelten Zusatzinformationen - die Symboltabelle ausgenommen - als sog. Zwischencode auf einem Sekundärspeicher abgelegt. Dieser Zwischencode wird nun zur Erzeugung des eigentlichen Maschinencodes in der zweiten Phase in den Primärspeicher gelesen und zeilenweise verarbeitet.

Zunächst wird der Zuordnungszähler wie in der Phase 1 durch die ORG-Anweisung initialisiert. Im weiteren Verlauf wird er dann jeweils um die durch eine Programmzeile erzeugte Anzahl an Maschinencodewörtern weitergezählt. Der jeweilige Zählerstand gibt die Ladeadressen der Maschinencodewörter an und wird diesen als Laderinformation beigefügt. Als erste Information wird die in der Phase 1 ermittelte und vom Lader benötigte Programmlänge dem Maschinenprogramm vorangestellt. Bei der Verarbeitung eines Maschinenbefehls werden die binäre Entsprechung des symbolischen Operationscodes der Zuordnungstabelle und die numerischen Entsprechungen des symbolischen Adreßteils der Symboltabelle entnommen und zu einem 16 Bit langen Maschinencodewort zusammengefaßt. Die Symboltabelle liefert zusätzlich die

Laderinformation fest/verschieblich des Adreßteils (s. Abschnitt 3.2: Verschiebender Lader). Die numerischen Entsprechungen von Konstanten im Adreßfeld werden dem Zwischencode entnommen. Eine DAT-Anweisung führt ebenfalls auf ein 16 Bit langes Maschinencodewort, das analog der Codierung des Adreßteils von Maschinenbefehlen erzeugt wird. Eine RES-Anweisung bewirkt lediglich ein Weiterzählen des Zuordnungszählers um die Anzahl der zu reservierenden Zellen. Ein danach generiertes Maschinencodewort erhält als Laderinformation eine entsprechend erhöhte Ladeadresse. Die EQU-Anweisungen werden in der Phase 2 übersprungen. Mit Verarbeitung der END-Anweisung wird schließlich die Startadresse dem Maschinencode zur späteren Initialisierung des Befehlszählers beigefügt. Der Maschinencode wird zusammen mit der Information für den Lader als Ladercode auf einen Sekundärspeicher ausgegeben.

Parallel zur Codegenerierung erzeugt der Assembler eine Programmliste, die das gesamte Programm in symbolischer Form und das Maschinenprogramm mit der Laderinformation in sedezimaler Form enthält (s. Beispiel 3.2). Die Programmliste kann auf Anforderung des Benutzers z.B. auf einem Schnelldrucker ausgegeben werden.

3.2 Lader und Binder

Um ein Programm ausführen zu können, muß es durch ein Ladeprogramm - den Lader - in den Primärspeicher des Rechners gebracht werden. Besteht ein Programm aus mehreren voneinander abhängigen, jedoch getrennt assemblierten Programmoduln, so müssen zuvor von einem Bindeprogramm - dem Binder - die Querbezüge zwischen den einzelnen Moduln hergestellt werden. Dieser Bindevorgang kann als Nachassemblieren angesehen werden, da durch ihn jene Adreßbezüge hergestellt werden, die vom Assembler nicht aufgelöst werden können.

Urlader, Absolutlader und verschiebender Lader

Wir stellen uns vor, daß wir einen Rechner haben, dessen Speicher zunächst leer ist. Um ein Programm in diesen Speicher laden zu können, benötigen wir ein Ladeprogramm. Da dieses Ladeprogramm, um es ausführen zu können, zuvor selbst in den Speicher geladen werden muß, ist eine Initialisierungsphase (initial program loading) nötig. Hierzu werden im allgemeinen mehrere Hilfsladeprogramme benutzt, die sich nacheinander in den Speicher laden. Dabei "zieht" das jeweils zuletzt geladene Programm das nächste, umfangreichere Programm "nach". Wir wollen annehmen, daß der Maschinencode dieser Hilfsladeprogramme und des eigentlichen Laders als Blöcke auf einem Sekundärspeicher zur Verfügung stehen. Der erste Block umfasse das erste Hilfsladeprogramm.

Urlader. Um zunächst den ersten Block lesen zu können, wird über eine Tastatur des Bedienungsfeldes des Rechners ein sehr kurzes Ladeprogramm, der Urlader, in den Primärspeicher eingegeben und gestartet. Der Urlader liest den ersten Block vom Sekundärspeicher und bringt den so geladenen Hilfslader, der bereits mehr Befehle als der Urlader umfaßt, zur Ausführung. Der Hilfslader liest nun seinerseits den nächsten Block und damit ein umfangreicheres Hilfsladeprogramm ein. Dieser Nachladevorgang wird so oft wiederholt, bis der eigentliche Lader, mit dem schließlich jedes Programm geladen werden kann, im Primärspeicher steht. Damit ist der Initialisierungsvorgang abgeschlossen. Die Hilfsprogramme werden nicht mehr benötigt und können überschrieben werden. Man bezeichnet dieses schrittweise Laden auch als Selbstladen (bootstrap loading).

Bei moderneren Rechnern wird der Urlader in einem Festspeicher untergebracht, so daß er nicht jedesmal neu über das Bedienungsfeld eingegeben zu werden braucht. Die Initialisierung wird dann einfach durch einen Knopfdruck eingeleitet. Dabei muß zuvor am Bedienungsfeld noch die Adresse des Gerätes eingegeben werden, das zur Eingabe der Ladeprogramme benutzt wird.

Um auch den Lader nicht für jeden Ladevorgang erneut laden zu müssen, wird er nach der Initialisierung als residentes Programm in einem vor dem Überschreiben geschützten Speicherbereich gehalten. Die Initialisierung ist nur dann noch nötig, wenn die Rechenanlage installiert wird oder der Lader z.B. durch einen Systemfehler zerstört wird.

Absolutlader. Während des Ladens eines Benutzerprogramms verarbeitet der Lader zusätzlich zum eigentlichen Programmcode auch Steuerinformationen für den Ladevorgang. Dazu gehören im einfachsten Fall die vom Assembler erzeugten Speicher-

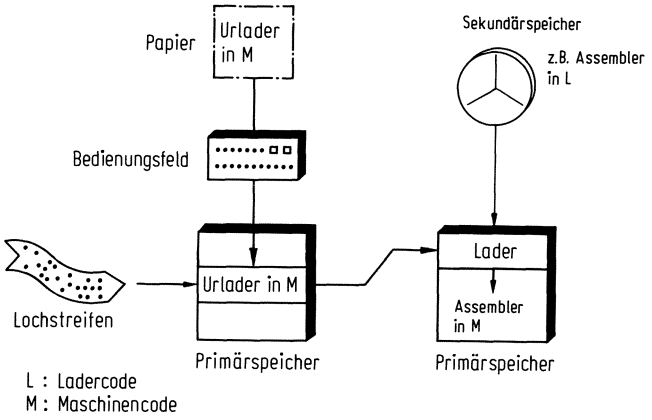


Bild 3.4 Laden des Urladers von Hand, Laden des Laders durch den Urlader und Laden des Assemblers durch den Lader

adressen des Maschinencodes, eine Angabe über die Länge des Maschinenprogramms und die Startadresse. Ein Lader, der den Maschinencode eines Programms unter den Adressen speichert, die vom Assembler ermittelt worden sind, wird als Absolutlader bezeichnet. Mit ihm kann ein einmal assembliertes Programm immer nur an dieselbe Stelle im Speicher geladen werden. Um es an anderer Stelle zu laden, muß es zuvor mit einer entsprechend abgeänderten ORG-Anweisung neu assembliert werden.

Bild 3.4 zeigt einen vereinfachten Initialisierungsvorgang, bei dem der Urlader über das Bedienungsfeld in den Primärspeicher gebracht wird und dann seinerseits den Lader direkt, d.h. ohne Hilfsladeprogramme, über einen Lochstreifenleser oder von Band in den Speicher lädt. Danach liest der Lader z.B. den Assembler vom Magnetband, wobei der Urlader im Speicher überschrieben wird.

Beispiel 3.4 Urlader und Absolutlader für den Einadreßrechner

Im folgenden wird die Initialisierungsphase für den Rechner aus Kapitel 2 am Beispiel eines Urladers und eines Absolutladers betrachtet, wobei der Urlader den Absolutlader direkt, d.h. ohne Hilfsladeprogramme in den Primärspeicher lädt (vgl. Bild 3.4).

Ladeadressen und Bitmuster des Urladers		Symbolischer Befehlscode und Kommentar zum Urlader		
36:	0000000000000001		1	Konstante 1.
37:	0110000000000101	INP	5	Schleifenbeginn: erstes Byte lesen.
38:	0011000000001000	SAS	8	Byte verschieben.
39:	0110000000000101	INP	5	Zweites Byte lesen.
40:	0100000000000000	STA	0	Wort speichern.
41:	1000000000101000	LDA	40	Indizierung: Speicheradresse
42:	1011000000100100	ADD	36	des in Zelle 40 gespeicherten
43:	0100000000101000	STA	40	Befehls um 1 erhöhen.
44:	1111000000000000	CMP	0	Schleifenende: wenn
45:	0001000000100101	LDP	37	(A) < (0): Sprung zu Adresse 37.
46:	0010000000000000	HLT		(A) = (0): Halt.

Der Maschinencode des Urladers wird am Bedienungsfeld des Rechners als "Bitmuster" über ein 16-Bit-Datenregister, dessen Inhalt sich durch 16 Tasten einstellen läßt, und eine LOAD-Taste in den Speicher eingegeben. Hierbei muß zu jedem Wert die Speicheradresse ebenfalls als Bitmuster in ein 12-Bit-Adreßregister des Bedienungsfeldes eingetastet werden. Der Start des Urladers erfolgt nach abgeschlossener Eingabe über eine START-Taste, nachdem zuvor die Startadresse 37 ins Adreßregister eingetastet wurde. Mit Betätigen der START-Taste wird diese Adresse in den Befehlszähler geladen und die Abrufphase eingeleitet.

Für das Einlesen des Absolutladers wird vorausgesetzt, daß der Absolutlader als reiner Maschinencode, d.h. ohne Zusatzinformation, auf einem Lochstreifenleser mit der Geräteadresse 5 bereitsteht. Die Informationsdarstellung auf dem Lochstreifen erfolgt in acht Datenspuren, so daß als Informationseinheit jeweils 1 Byte gelesen werden kann. Um ein Maschinencodewort einzulesen, sind also zwei INP-Befehle notwendig. Zwischen diesen beiden Befehlen muß ein Schiftbefehl stehen, der das zuerst gelesene Byte im Akkumulator um acht Stellen nach links verschiebt. Das Einlesen erfolgt in einer Programmschleife, in der die Maschinencodewörter fortlaufend in die Speicherzellen 0 bis 35 geladen werden. Die Anzahl der Schleifendurchläufe plus 1 (36) gibt der Absolutlader selbst in seinem ersten Codewort an, das in die Speicherzelle 0 geladen wird. Am Ende des Einlesevorgangs steuert der Urlader den Rechner durch einen HLT-Befehl in den Wartezustand. Der Absolutlader steht danach zur Ausführung bereit. Um ihn zu starten, wird über das Adreßregister des Bedienungsfeldes seine Startadresse 7 eingegeben und danach die START-Taste betätigt.

```
            ORG 0       ABSOLUTLADER
 0          STA 36
 1   LAENGE RES 1       LAENGE DES ZU LADENDEN PROGRAMMS
 2   ORT    RES 1       SPEICHERADRESSE DES MASCHINENCODEWORTES
 3   BFHL1  STA 0       BEFEHLE, DIE ZUR ADRESSMODIFIKATION
 4   BFHL2  LDP 0       VERWENDET WERDEN
 5   I      DAT 0       SCHLEIFENINDEX
 6   EINS   DAT 1
 7   START  INP 5       PROGRAMMSTART:PROGRAMMLAENGE EINLESEN (ERSTES BYTE)
 8          SAS 8
 9          INP 5       PROGRAMMLAENGE EINLESEN (ZWEITES BYTE)
10          STA LAENGE
11   LOOP   INP 5       SPEICHERADRESSE EINLESEN (ERSTES BYTE)
12          SAS 8
13          INP 5       SPEICHERADRESSE EINLESEN (ZWEITES BYTE)
14          STA ORT
15          LDA BFHL1   INDIREKTE ADRESSIERUNG DURCH ADRESSRECHNUNG
16          ORA ORT     MIT STA-BEFEHL
17          STA IND1
18          INP 5       MASCHINENCODEWORT EINLESEN (ERSTES BYTE)
19          SAS 8
20          INP 5       MASCHINENCODEWORT EINLESEN (ZWEITES BYTE)
21   IND1   STA 0
22          LDA I       SCHLEIFENINDEX ERHOEHEN
23          ADD EINS
24          STA I
25          CMP LAENGE  SCHLEIFENENDE ABFRAGEN
26          LDP LOOP    RUECKSPRUNG NACH LOOP,WENN(A)< LAENGE
27          LAZ         SCHLEIFENINDEX AUF DEN
28          STA I       ANFANGSWERT NULL SETZEN
29          INP 5       STARTADRESSE EINLESEN (ERSTES BYTE)
30          SAS 8
31          INP 5       STARTADRESSE EINLESEN (ZWEITES BYTE)
32          ORA BFHL2   STARTADRESSE ALS SPRUNGADRESSE IN DEN BEFEHL MIT
33          STA IND2    DEM NAMEN IND2 EINSETZEN
34          HLT         ENDE DES LADEVORGANGS
35   IND2   LDP 0       UEBERGABE DER PROGRAMMSTEUERUNG AN EINGELESENES PROGRAMM
            END START
```

Die Wirkungsweise des Absolutladers kann anhand eines zu ladenden Programms (vgl. Beispiel 3.2) und des Kommentars zum Absolutlader nachvollzogen werden. Der Lader liest zunächst das erste Codewort des Programms, das ihm die Anzahl der noch folgenden Doppelwörter angibt, in die Speicherzelle LAENGE ein. Jedes Doppelwort besteht aus einer Speicheradresse und dem unter dieser Adresse zu speichernden Wort[1]. Zur Speicherung dieses Wortes ist eine indirekte Adressierung erforderlich, die durch eine Oder-Verknüpfung des Befehls STA 0 mit der in der Zelle ORT abgelegten Speicheradresse durchgeführt wird. Nachdem alle Doppelwörter verarbeitet sind, liest der Lader als letztes Codewort die Startadresse des Programms ein und bringt dann den Rechner in den Wartezustand. Zuvor wird der ursprüngliche Programmzustand für den nächsten Ladevorgang hergestellt (betrifft die Befehle LAZ und STA I). Mit Betätigen der START-Taste führt der Lader seinen letzten Befehl LDP aus. Dieser lädt die Startadresse in den Befehlszähler und übergibt damit die Steuerung an das geladene Programm.

Verschiebender Lader. Soll ein Programm in einen freien Speicherbereich geladen werden, dessen Lage beim Schreiben des Programms nicht bekannt ist, so kann die Ladeadresse erst beim Ladevorgang angegeben werden. Zum Beispiel ist es bei größeren Programmen nützlich, sie in einzelne Abschnitte, sog. Segmente oder Moduln zu zerlegen, die getrennt assembliert aber gemeinsam geladen werden. Die Moduln dürfen sich beim Laden nicht gegenseitig überlappen, d.h. aus der Ladeadresse und der Länge des zuletzt geladenen Moduls muß die Ladeadresse des nächsten Moduls errechnet werden.

Der Absolutlader hat den Nachteil, daß die Ladeadresse nicht beim Ladevorgang vorgegeben werden kann. Um dies zu ermöglichen, macht man die Programme verschieblich. Da der Assembler bereits absolute Adressen in den Maschinencode einsetzt, genügt es nicht, daß der Lader den Code unverändert verschiebt. Dies würde zu falschen Adreßbezügen innerhalb des Programms führen. Man benötigt deshalb einen verschiebenden Lader, der während des Ladevorgangs auf die von der Verschiebung betroffenen Adreßwerte des Maschinencodes den Wert der neuen Ladeadresse addiert. Die Angabe, welches Codewort eine verschiebliche Adresse enthält, wird vom Assembler für jedes Maschinenwort in der Laderinformation durch ein Bit angegeben.

Binder und bindender Lader

Wie schon beim verschiebenden Lader beschrieben, werden größere Programme in Moduln aufgeteilt. Jeder Modul wird unabhängig von den übrigen assembliert, und

---

[1] Üblicherweise wird dem Lader nicht für jedes Codewort eine Ladeadresse mitgegeben. Es werden stattdessen vom Assembler Maschinencodeblöcke gebildet mit Angabe der Ladeanfangsadressen und der Blocklängen. Blockgrenzen treten an Stellen von einzelnen oder aufeinanderfolgenden RES-Anweisungen auf, da die regelmäßige Adreßfortzählung durchbrochen wird.

in einer anschließenden Bindephase werden die Moduln dann zu einem ladbaren Gesamtmodul gebunden. Dieses Vorgehen bringt eine Reihe von Vorteilen mit sich:

1. Die einzelnen Moduln können von mehreren Programmierern geschrieben werden.
2. Die Moduln sind wegen ihrer Kürze übersichtlicher und dadurch einfacher zu testen.
3. Bei der Änderung eines Moduls braucht nicht das gesamte Programm neu assembliert zu werden.
4. Im Programm können Standardmoduln z.B. aus einer Programmbibliothek benutzt werden, ohne daß sie assembliert zu werden brauchen.
5. Die vom Assembler erzeugten Moduln können mit Moduln, die von Übersetzern höherer Programmiersprachen erzeugt wurden, gebunden werden.
6. Die Anzahl der Symbole eines Programms wird nicht durch die maximal zulässige Länge der Symboltabelle des Assemblers beschränkt.

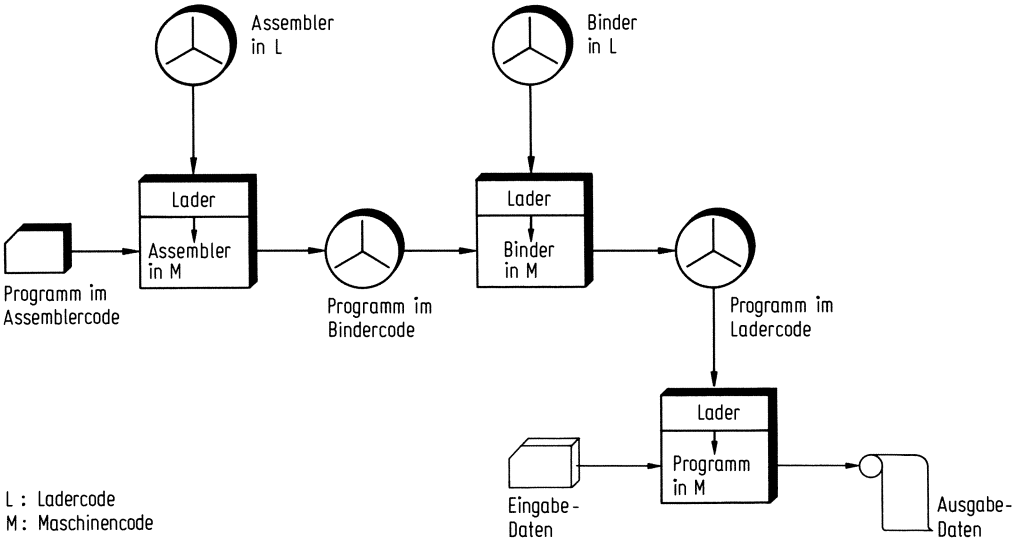


Bild 3.5 Assemblieren, Binden und Laden eines Programmes

Bild 3.5 zeigt das Assemblieren, das Binden und das Laden von Programmoduln; es schließt damit an den in Bild 3.4 dargestellten Ablauf an. Der vom Lader in den Primärspeicher geladene Assembler liest die Programmoduln nacheinander in den Speicher, übersetzt sie und gibt sie als Bindermoduln im Bindercode auf einen Sekundärspeicher aus. Der Bindercode enthält Informationen, die dem Binder jene Adreßbezüge anzeigen, die innerhalb eines Moduls vom Assembler nicht hergestellt werden konnten. Im nächsten Verarbeitungsschritt lädt der Lader den Binder in den Speicher, wobei er den Assembler überschreibt. Der Binder seinerseits liest die Bindermoduln nacheinander in den Speicher, wobei er die nachfolgenden Moduln so

verschiebt, daß sie an die bereits geladenen Moduln angefügt werden können. Gleichzeitig stellt er anhand der Binderinformation die Adreßbezüge zwischen den Moduln her. Die so gebundenen Moduln werden vom Binder als Ladermodul wieder auf einem Sekundärspeicher abgelegt. Schließlich wird der Ladermodul vom Lader zur Ausführung in den Speicher geladen, wobei eine Verschiebung des Gesamtmoduls möglich ist. Der Binder kann dabei überschrieben werden.

Damit der Assembler die notwendigen Informationen für den Bindevorgang bereitstellen kann, müssen ihm die externen Adreßbezüge, d.h. die Adreßbezüge zwischen den Moduln, mitgeteilt werden. Wir führen dazu zwei Assembleranweisungen EXT und ENT ein.

EXT s (external) bezeichnet ein Symbol, das innerhalb des Moduls benutzt wird, jedoch in einem anderen Modul definiert ist. Im Modul, in dem die EXT-Anweisung steht, tritt dieses Symbol im Adreßfeld von Anweisungen oder Befehlen auf.

ENT s (entry) bezeichnet ein Symbol, das innerhalb des Moduls definiert ist, jedoch in einem anderen Modul benutzt wird. Im Modul, in dem die ENT-Anweisung steht, muß dieses Symbol im Namensfeld einer Anweisung oder eines Befehls auftreten; es kann außerdem auch im Adreßfeld verwendet werden.

Jedes Symbol, das in einer EXT-Anweisung erscheint, nimmt der Assembler zusammen mit den Adressen der Speicherzellen, in denen es verwendet wird (Referenzadressen), in eine sog. Benutzungstabelle auf. Jedes Symbol, das in einer ENT-Anweisung angegeben ist, nimmt der Assembler zusammen mit der numerischen Entsprechung des Symbols und der Verschiebeinformation in eine sog. Definitionstabelle auf. Die Benutzungstabelle, die Definitionstabelle und die Angabe der Länge beider Tabellen werden dem Maschinencode eines Moduls als Binderinformation vorangestellt.

Während des Bindevorgangs erzeugt der Binder aus den Benutzungs- und Definitionstabellen aller Bindermoduln eine sog. Bindesymboltabelle. In ihr faßt er alle Eintragungen der Benutzungs- und Definitionstabellen mit gleichen symbolischen Adreßangaben zusammen, so daß der Bezug zwischen den Referenzadressen und den numerischen Entsprechungen der Symbole hergestellt ist. Unter Auswertung der Bindesymboltabelle ergänzt der Binder die fehlenden Adressen im Maschinencode der einzelnen Moduln.

Häufig werden der Bindevorgang und der Ladevorgang nicht nacheinander ausgeführt, sondern miteinander in einem Ablaufschritt verknüpft. Man nennt den Lader dann dementsprechend bindender Lader (linking loader).

Beispiel 3.5 Assemblieren und Binden zweier Programmoduln

Das folgende Programm, das sich aus den beiden Moduln MODUL1 und MODUL2 zusammensetzt, soll das Erzeugen der Binderinformation und deren Auswertung zeigen. Dazu sind die beiden Moduln im Assemblercode, im Bindercode und im Maschinencode dargestellt. Der Übersichtlichkeit halber wird auf die Angabe der Laderinformation verzichtet. Desgleichen sind die Operationscodes im Binder- und Maschinencode symbolisch angegeben und sämtliche Adressen und Operanden als Dezimalzahlen dargestellt.

Assemblercode		Bindercode			Maschinencode		
*	MODUL1						
*				2			
	ORG			1			
	EXT Z	Ben.-Tab.	Z :	12			
	EXT MOD2		MOD2:	99			
	ENT K	Def.-Tab.	K =	2			
X	DAT 123	0:		123	0:		123
Y	DAT 987	1:		987	1:		987
K	DAT 21	2:		21	2:		21
	⋮	⋮			⋮		
START	LDA X	10:	LDA	0	10:	LDA	0
	ADD Y	11:	ADD	1	11:	ADD	1
	STA Z	12:	STA	13	12:	STA	100
	MUL Z	13:	MUL	14	13:	MUL	100
	STA Z	14:	STA	-	14:	STA	100
	⋮	⋮			⋮		
	LDP MOD2	99:	LDP	-	99:	LDP	120
	END START						
*	MODUL2						
*				1			
	ORG			2			
	EXT K	Ben.-Tab.	K :	21			
	ENT Z	Def.-Tab.	Z =	0			
	ENT MOD2		MOD2=	20			
Z	RES 1	0:			100:		
	⋮	⋮			⋮		
MOD2	LDA Z	20:	LDA	0	120:	LDA	100
	ADD K	21:	ADD	-	121:	ADD	2
	STA Z	22:	STA	0	122:	STA	100
	⋮	⋮			⋮		
	HLT	199:	HLT		299:	HLT	
	END						

Beide Programmoduln beginnen mit ORG und werden mit END abgeschlossen. Das Symbol START im Adreßteil der END-Anweisung des ersten Moduls gibt die Startadresse des Programms an. MODUL1 berechnet aus den Größen X und Y seines Datenbereichs eine Größe Z, für die im Datenbereich von MODUL2 eine Speicherzelle reserviert ist. Später erfolgt ein Programmsprung zu der ebenfalls in MODUL2 definierten Programmadresse MOD2. Z und MOD2 sind somit zwei Adressen, die in MODUL1 mit EXT und in MODUL2 mit ENT zu kennzeichnen sind. In MODUL2 wird zur Größe Z die Konstante K addiert, die im Datenbereich von MODUL1 definiert ist. K muß deshalb in MODUL1 in einer ENT-Anweisung und in MODUL2 in einer EXT-Anweisung angegeben werden.

Der Bindercode von MODUL1 enthält dementsprechend eine Benutzungstabelle mit den Symbolen Z und MOD2. Da Z mehrere Referenzadressen hat, müssen diese entweder alle in der Benutzungstabelle aufgeführt sein oder aber über sie zugänglich sein. Im angegebenen Beispiel sind die Referenzadressen in einer Referenzkette gespeichert, die die Adreßteile der betroffenen Befehle als Speicherplätze benutzt. Das Symbol Z in der Benutzungstabelle verweist auf die erste Referenzadresse 12. Unter dieser ist im Adreßteil des STA-Befehls die nächste Referenzadresse 13 gespeichert. Die Referenzkette endet mit einem "leeren" Adreßteil im STA-Befehl mit der letzten Referenzadresse 14. Die Definitionstabelle von MODUL1 enthält das Symbol K und dessen numerische Entsprechung 2. Die Länge der Benutzungstabelle und die Länge der Definitionstabelle sind beiden Tabellen vorangestellt.

Analog zu MODUL1 besteht die Binderinformation für MODUL2 aus einer Benutzungstabelle, in der das Symbol K mit der Referenzadresse 21 angegeben ist, einer Definitionstabelle mit den Symbolen Z und MOD2 und deren numerischen Entsprechungen 0 bzw. 20 und den Längenangaben beider Tabellen. In beiden Moduln sind die noch unbestimmten Adreßspeicherplätze im Maschinencode, sofern sie nicht von der Referenzkette belegt sind, durch einen Querstrich gekennzeichnet.

Der Maschinencode beider Moduln zeigt die vom Binder über die Bindesymboltabelle hergestellten Adreßbezüge unter Berücksichtigung der Verschiebung von MODUL2 um die Anzahl 100 der von MODUL1 belegten Speicherzellen. Der Maschinencode beider Moduln bildet nun eine Einheit und kann zur Ausführung in den Speicher geladen werden. ■

3.3 Systemprogramme und Betriebssysteme

Wie wir in Abschnitt 3.2 gesehen haben, ist von der Erstellung eines Programms bis zu dessen Ausführung eine Reihe von Schritten notwendig, in deren Verlauf das symbolische Programm durch Systemprogramme wie Assembler, Binder und Lader

codiert wird. In Bild 3.5 sind Primär- und Sekundärspeicher nur der Anschaulichkeit halber mehrfach gezeichnet. Diesem Bild liegt vielmehr eine Rechnerkonfiguration zugrunde, wie sie in Bild 3.6 dargestellt ist. Um den Primärspeicher als zentralen Speicher sind die Sekundärspeicher als periphere Speicher gruppiert. Zu diesen gehören die beiden Lochkartenstapel, auf denen das symbolische Programm und seine Daten gespeichert sind und die über einen Lochkartenleser in den Primärspeicher geladen werden können. Für die Zwischencodes, wie Binder- und Ladercode, die vom Rechner weiterverarbeitet werden, steht eine Magnetbandeinheit (MT1) zur Verfügung. Eine weitere Magnetbandeinheit (MT0) dient zur Speicherung der Systemprogramme. Schließlich zählt auch das Papier, auf dem ein Schnelldrukker (LP) Listen anfertigt und die Resultate ausdruckt, zu den peripheren Speichern.

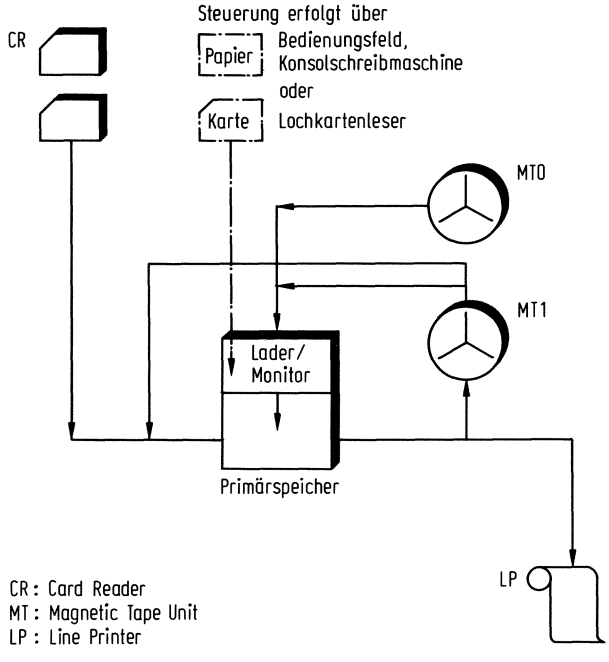


Bild 3.6 Beispiel einer Rechnerkonfiguration mit Konsolschreibmaschine, Lochkartenleser, Schnelldrucker und zwei Magnetbandgeräten

Zur Durchführung der einzelnen Schritte muß der Lader permanent in einem gegen Überschreiben möglichst geschützten Speicherbereich zur Verfügung stehen. Als erstes wird das Lochkartenpaket in den Kartenleser eingelegt. Dann wird der Assembler geladen und gestartet, der das symbolische Programm in den Bindercode umformt und es auf einem Magnetband zwischenspeichert. Dann wird der Binder geladen und gestartet, der das Programm ggf. zusammen mit anderen Moduln in den Ladercode umformt und auf dem Magnetband zwischenspeichert. Schließlich wird das Programm selbst geladen und ausgeführt, wodurch die im Kartenleser gestapelten Ein-

gabedaten in Ausgabedaten umgeformt werden, die auf dem Schnelldrucker ausgedruckt werden. Danach steht die Rechenanlage für die nächste Aufgabe zur Verfügung.

Manueller Betrieb

In der einfachsten Betriebsform wird der eben geschilderte Ablauf vom Benutzer selbst gesteuert. Er benutzt dazu das Bedienungsfeld, über das er manuell die einzelnen Anweisungen, wie Laden und Starten der Systemprogramme und des Benutzerprogramms, eingibt. Dabei muß er den Systemprogrammen über die Register des Bedienungsfeldes die Geräteadressen der jeweils benutzten Ein-Ausgabegeräte mitteilen. Zur Überwachung des Ablaufs sind vom Benutzer verschiedene Kontrollampen am Bedienungsfeld zu beachten. Sie geben ihm z.B. an, wann er den nächsten Ablaufschritt einleiten kann und ob während des letzten Ablaufschritts Fehler aufgetreten sind, auf die er mit entsprechenden Anweisungen reagieren muß.

Diese Art der manuellen Rechnerbedienung durch den Benutzer weist verschiedene Nachteile auf:

1. Der Zeitaufwand für die Rechnerbedienung ist im Vergleich zur Arbeitsgeschwindigkeit des Prozessors hoch.
2. Die Rechnerbedienung erfolgt häufig fehlerhaft, da sie von jedem Benutzer detaillierte Kenntnisse der Rechnerhardware und -software verlangt.
3. Die Fehlerinterpretation während des Programmlaufes ist ebenfalls sehr zeitaufwendig und nicht von jedem Benutzer optimal zu bewältigen.
4. Der laufende Wechsel der Benutzer einer Rechenanlage vermindert ihre Verfügbarkeit, z.B. durch unsachgemäße Behandlung von Hardware und Software.

Aus diesen Gründen geht man dazu über, für die Rechnerbedienung einen Operateur einzusetzen, der mit der Anlage vertraut ist. Der Benutzer hat jetzt keinen direkten Zugriff mehr zum Rechner. Er liefert sein Programm und die Daten als Lochkartenstapel beim Operateur ab und gibt ihm an, welche Ablaufschritte erforderlich sind. Den Lochkartenstapel und die Liste der Resultate kann er sich dann zu einem späteren Zeitpunkt beim Operateur wieder abholen.

Beispiel 3.6 Auftragsbeschreibung bei manuellem Betrieb

Für das Assemblieren und nachfolgende Ausführen eines Programmes liefert der Programmierer dem Operateur den Lochkartenstapel ab, der das symbolische Programm und die Eingabedaten umfaßt. Darüber hinaus gibt er ihm einen Zettel mit folgenden Anweisungen zur Bearbeitung seines Auftrags, wobei die in Bild 3.6 dargestellte Rechnerkonfiguration zugrunde gelegt ist.

Anweisungen an den Operateur	Kommentar
1. Auftrag: USER NAME	für einen Vermerk im Protokoll
2. Ordne zu: Symbolcode-Eingabe von Kartenleser, Binärcode-Ausgabe auf Magnetband 1, Listen-Ausgabe auf Schnelldrucker	für den Assembliervorgang
3. Spule zurück: Magnetband 1	für das Zwischenspeichern des Binärcodes
4. Assembliere: mit Symbolcode-Eingabe, Binärcode-Ausgabe, Listen-Ausgabe	
5. Ordne zu: Binärcode-Eingabe von Magnetband 1	für den Ladevorgang
6. Spule zurück: Magnetband 1	für das Laden des Binärcodes
7. Lade	
8. Starte	
9. Ende Auftrag	

Der Operateur führt die Anweisungen der Auftragsbeschreibung aus, wobei er die Zuordnungen der verschiedenen Programmcodes zu den externen Geräten dem Assembler und dem Lader über das Bedienungsfeld mitteilt. Die Anweisung "Assembliere" umfaßt das Laden und Starten des Assemblers; die letzte Karte des symbolischen Programms muß eine END-Anweisung enthalten, nach deren Verarbeitung der Assembler den Rechner anhält. Die Anweisung "Lade" umfaßt das Laden und Starten des Laders, der nach Beendigung des Ladevorganges den Rechner erneut anhält. Mit der Anweisung "Starte" wird das Benutzerprogramm gestartet. Der Rechner befindet sich im sog. Benutzermodus; der letzte Befehl des Maschinenprogramms muß ein HLT-Befehl sein, damit der Operateur die nächsten Anweisungen ausführen kann.

■

Stapelbetrieb

Die Arbeit des Operateurs bei der Verarbeitung eines Lochkartenstapels kann vereinfacht und damit der Gesamtrechenvorgang beschleunigt werden, wenn zur Steuerung des Ablaufs anstelle des Bedienungsfeldes ein bequemeres Ein-Ausgabegerät, z.B. eine Konsolschreibmaschine, verwendet wird. Dazu müssen jedoch die Anweisungen an den Rechner formalisiert werden, und es muß ein Systemprogramm existieren, das diese Anweisungen interpretieren und ausführen kann. Wir bezeichnen die Menge dieser Anweisungen als Kommandosprache und das zugehörige Systemprogramm als Monitor. In Bild 3.6 erfolgt die Steuerung nicht mehr über das Bedienungsfeld, sondern durch die Konsolschreibmaschine, und der vom Lader belegte permanente Primärspeicherbereich ist um das Monitorprogramm zu erweitern. Dem Monitor können nun auch zusätzliche Aufgaben übertragen werden, wie z.B. die Überwachung des Systemzustandes, das Ausdrucken von Informationen und die Reaktion auf fehlerhafte Abläufe.

Die manuelle Eingabe der Steueranweisungen über eine Konsolschreibmaschine ist für den Operateur bequemer und schneller auszuführen als die Eingabe über das Bedienungsfeld. Sie erfolgt jedoch im Vergleich zur Verarbeitungsgeschwindigkeit des Rechners immer noch sehr langsam. Man geht deshalb von der Konsolschreibmaschine auf ein schnelleres Eingabegerät, den Lochkartenleser, über. Die Steueranweisungen werden jetzt vom Benutzer auf Lochkarten gestanzt und als sog. Steuerkarten dem Lochkartenstapel des auszuführenden Programms beigefügt. Die Steuerkarten bilden zusammen mit den Programm- und Datenkarten einen in sich abgeschlossenen Auftrag (job). Dem Operateur obliegt jetzt im wesentlichen nur noch die Bedienung der peripheren Geräte, z.B. das Einlegen der Kartenstapel in den Kartenleser, das Auswechseln von Magnetbändern und die Versorgung des Schnelldruckers mit Papier.

Die Automatisierung des Ablaufs bietet die Möglichkeit, mehrere Jobs in einem großen Kartenstapel zusammenzufassen, die vom Rechner nacheinander ohne manuelle Beeinflussung verarbeitet werden. Lediglich die Reihenfolge der Job-Verarbeitung wird vom Operateur beim Zusammenstellen des Stapels festgelegt. Man nennt diese Betriebsform Stapelbetrieb (batch processing).

Gegenüber der Steuerung über die Konsolschreibmaschine weist der Monitor für den Stapelbetrieb einige Erweiterungen auf. Die Kommandosprache, die auch als Job Control Language oder Job Description Language bezeichnet wird, ist jetzt wesentlich umfangreicher. Sie ähnelt in der Vielfalt ihrer Anweisungen einer Programmiersprache und kann z.B. auch Anweisungen zur bedingten Verzweigung des Ablaufs vorsehen.

Der Monitor bildet zusammen mit den übrigen Systemprogrammen wie Assembler, Binder und Lader, das Betriebssystem der Rechenanlage. Dabei stellt der Monitor das übergeordnete Systemprogramm dar, das den Einsatz der restlichen Systemprogramme organisiert.

Moderne Rechenanlagen ermöglichen es, symbolische Programme nicht nur in der maschinennahen Assemblersprache, sondern auch in problemorientierten Sprachen wie FORTRAN, ALGOL, PL/1 oder COBOL zu formulieren. Die dafür erforderlichen Übersetzungsprogramme (compiler) sind ebenfalls Bestandteile des Betriebssystems. Darüber hinaus steht dem Benutzer eine Reihe von Standardprogrammen für immer wiederkehrende Aufgaben in Form einer Programmbibliothek zur Verfügung. Diese Standardprogramme können vom Benutzerprogramm als Unterprogramme aufgerufen werden; Beispiele sind Programme für die Berechnungen mathematischer Funktionen oder für die Umwandlung von Dezimal- in Dualzahlen. Testprogramme ermöglichen es, Probeläufe von Programmen vorzunehmen. So können mit "Trace-Routinen" z.B. die Registerinhalte für jeden Rechenschritt eines zuvor definierten Pro-

grammbereiches ausgegeben werden. "Dump-Routinen" ermöglichen das Ausdrucken von Bereichen des Arbeitsspeichers z.B. nach Beendigung einer Programmausführung. Mit "Update-Routinen" können das Systemband bzw. die Systemplatte verändert werden, z.B. durch Hinzufügen von weiteren System- bzw. Bibliotheksprogrammen. Diese Möglichkeiten sind jedoch im allgemeinen dem Systemprogrammierer vorbehalten, um die Verfügbarkeit der Anlage nicht zu gefährden.

Der Einsatz unterschiedlicher peripherer Geräte mit relativ komplizierten Ein-Ausgabe-Programmen in Assembler- oder Maschinensprache legt es nahe, die Ein-Ausgabe-Programme ebenfalls als Systemprogramme in das Betriebssystem zu integrieren und sie allen Benutzern zur Verfügung zu stellen. Der Benutzer erspart sich dadurch Programmierarbeit; gleichzeitig wird auch die Fehlerrate seiner Programme reduziert.

Da das Betriebssystem viel Speicherplatz benötigt, kann es nicht permanent im Primärspeicher untergebracht werden. Es würde den Speicherbereich eines laufenden Programmes zu stark einengen. Deshalb wird es auf einem Sekundärspeicher gehalten, und die einzelnen Systemprogramme werden nur bei Bedarf in den Primärspeicher geladen. Eine Ausnahme bilden der Lader und Teile des Monitors. Sie belegen permanent einen festen Bereich des Primärspeichers, der gegen Überschreiben durch Benutzerprogramme geschützt ist. Damit wird ein Zerstören dieser sog. "arbeitsspeicherresidenten Software" verhindert.

Beispiel 3.7 Job-Beschreibung bei Stapelbetrieb

In Bild 3.7 ist der Auftrag, wie er in Beispiel 3.6 vorlag, als Lochkartenpaket dargestellt, das neben dem Assemblerprogramm und den zugehörigen Eingabedaten auch

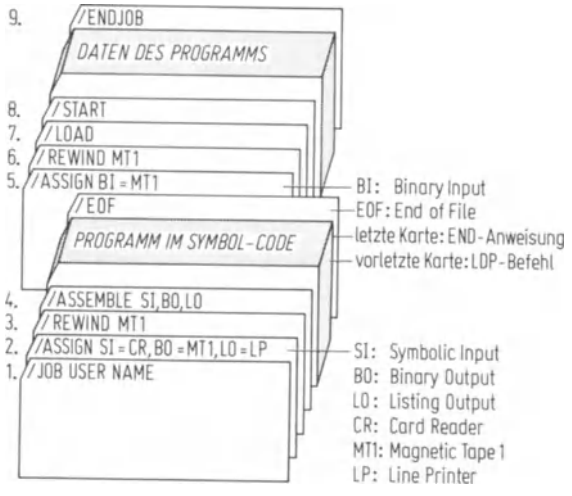


Bild 3.7 Kartenstapel mit Anweisungen zum Assemblieren, Laden und Starten eines Programms unter Zugrundelegung eines Rechners mit der Konfiguration Bild 3.6

die Steuerkarten für den Verarbeitungsablauf enthält. Die Job-Verarbeitung verläuft nach den in Beispiel 3.6 angegebenen Ablaufschritten, wobei den Steuerkarten die in Beispiel 3.6 angegebenen Nummern vorangestellt sind. Wird das Einlesen der Karten des Symbol-Codes vom Monitor vorgenommen, so muß die auf die letzte Karte des Symbol-Codes folgende Karte zur Kennzeichnung des Satz-Endes eine mit einem Schrägstrich versehene Monitoranweisung enthalten. Nach dem Starten des Programmes befindet sich der Rechner im Benutzermodus; und der letzte Befehl des Programmes muß ein Sprungbefehl sein, der den Monitor zur Verarbeitung der nächsten Steuerkarte aufruft.

Bei der Job-Beschreibung wurden als externe Speicher bewußt zwei Magnetbandeinheiten verwendet, um die einzelnen Speicheranforderungen des Betriebssystems zu unterscheiden. Bei einer Rechenanlage mit einer Magnetplatteneinheit wird man wegen des schnelleren Speicherzugriffs neben dem Betriebssystem auch den Binärcode auf dem Plattenspeicher ablegen. Die unterschiedlichen symbolischen Gerätebezeichnungen der Steuerkarten werden dann durch die Einheitsbezeichnung DF (disk file) ersetzt. Die Unterscheidung der Plattenbereiche wird vom Betriebssystem selbst vorgenommen. ■

Teilnehmerbetrieb

Beim Stapelbetrieb hat der Benutzer keinen direkten Kontakt mehr zum Rechner, und es treten für ihn relativ große Wartezeiten zwischen Abgabe seines Jobs und Erhalt der Resultate auf. Dies ist besonders während der Testphase eines Programmes nachteilig, wenn Fehler auftreten, die eine Programmänderung und damit eine erneute Programmaufbereitung notwendig machen.

Der Teilnehmerbetrieb bietet demgegenüber die Möglichkeit, viele Benutzer gleichzeitig mit dem Rechner arbeiten zu lassen. Dadurch ist die Rechenanlage auch dann gut ausgelastet, wenn jeder Teilnehmer über ein langsames Ein-Ausgabegerät wie eine Schreibmaschine oder ein Sichtgerät unmittelbar an den Rechner angeschlossen ist. Die Schreibmaschine bzw. das Sichtgerät werden zur Programmerstellung wie zur Ablaufsteuerung benutzt. Bei der Programmerstellung wird der Quellcode sofort auf den Sekundärspeicher gebracht, wodurch das lästige Hantieren mit Lochkarten entfällt und Korrekturen bequem ausgeführt werden können. Bei der Ablaufsteuerung "unterhält" sich der Benutzer mit dem Rechner in einer sog. Dialogsprache, wodurch das beim Stapelbetrieb leidige Warten entfällt und wichtige Mitteilungen dem Benutzer sofort übermittelt werden können.

Das Teilnehmerbetriebssystem vermittelt dem Benutzer den Eindruck, als habe er die Rechenanlage für sich allein zur Verfügung; der Rechner erhält Anforderungen von so vielen Benutzern, daß er während der Wartezeiten, die bei einem Benutzer

auftreten, seine Rechenkapazität den anderen Benutzern anbieten kann. Da ein rechenintensives Programm die übrigen Benutzeranforderungen blockieren würde, erfolgt die Aufteilung der Rechenkapazität in sog. Zeitscheiben (time slices). Das bedeutet, daß ein Benutzer immer nur für einen bestimmten Zeitabschnitt die Rechenanlage zugeteilt bekommt und daß sich alle Benutzer die Verfügung über die gesamte Anlage zeitlich teilen (time sharing). Dazu ist jedoch eine aufwendige Speicherverwaltung durch das Betriebssystem erforderlich. Ist der Primärspeicher des Rechners groß genug, um mehrere Programme gleichzeitig aufzunehmen, so müssen die Speicherbereiche der gerade inaktiven Programme gegen eine Beeinflussung durch das gerade aktive Programm geschützt werden (Speicherschutz für System- und Benutzerprogramme). Üblicherweise werden nur Teile von Programmen in den Primärspeicher geladen, da pro Zeitscheibe auch nur ein Teil eines Programmes verarbeitet werden kann. Diese Programmteile werden je nach Systemzustand zwischen dem Primärspeicher und einem Sekundärspeicher hin und her transportiert. Dem Betriebssystem obliegt dabei die Entscheidung, welche Bereiche des Primärspeichers ausgelagert werden müssen oder überschrieben werden können. Da ein Programmteil bei mehrmaligem Laden in den Primärspeicher unter unterschiedlichen physikalischen Adressen gespeichert sein kann, wird vom Betriebssystem eine Adreßumsetzung gefordert, d.h. die logischen Programmadressen werden zur Ausführungszeit auf die physikalischen Speicheradressen umgesetzt. Das Betriebssystem ist weiterhin in der Lage, Prioritäten der einzelnen Benutzerprogramme, die vom Operateur angegeben werden, bei der Verteilung der Rechenkapazität zu berücksichtigen. Darüber hinaus mißt es die von jedem Benutzer beanspruchte Rechenzeit, um eine Kostenabrechnung durchführen zu können.

Neben den beschriebenen internen Funktionen bietet ein Teilnehmerbetriebssystem viele Dienstleistungen, die auf den Dialogverkehr mit dem Benutzer zugeschnitten sind. Im folgenden Beispiel wird dies am Erstellen, Ausführen und Ändern eines Programms gezeigt.

Beispiel 3.8 Dialogverkehr bei Teilnehmerbetrieb

Der Benutzer gibt das Quellprogramm mit Hilfe eines Systemprogramms, des Editors, über eine Konsolschreibmaschine oder ein Datensichtgerät zeilenweise ein und kann dabei die innerhalb einer Zeile auftretenden Schreibfehler sofort korrigieren. Weiter lassen sich durch textverarbeitende Kommandos bereits eingegebene Programmteile oder ganze Programme modifizieren. Das editierte Programm wird durch ein Kommando als Verarbeitungseinheit (file) auf dem Sekundärspeicher abgelegt und kann übersetzt, geladen und ausgeführt werden.

Das folgende Protokoll soll einen Eindruck über die Möglichkeiten des Dialogverkehrs vermitteln. Das Programm zur Summierung der ganzen Zahlen 1 bis N wird

```
/ LOGIN USERID                    Benutzeridentifikation ins Logbuch eintragen!
------ LOGON
/ EDIT SUM ASSEMBLE               Editor zur Programmerstellung von SUM aufrufen!
------ EDIT NEW FILE
// INPUT                          Programmtext eingeben!
        ORG
I       RES     1
A       RES     1
N       RES     1
TYP     EQU     2
START   INP     N       TYP
        LD      I       =1
        LD      A       =0
LOOP    ADD     A       I
        ADD     I       =1
        CMP     I       N
        NOP
        GOTO    LOOP
        OUT     A       TYP
        GOTO    START
        END     START
// FILE                           Programmtext als File SUM ablegen!
/ ASSEMBLE SUM                    Programm SUM assemblieren!
------ ASSEMBLED
/ LOAD SUM                        Programm SUM laden!
------ LOADED
/ START SUM                       Programm SUM starten!
------ STARTED
1                                 1 eingeben (INP-Befehl)
1                                 Ausgeben von 1 (OUT-Befehl)
5                                 5 eingeben (INP-Befehl)
15                                Ausgeben von 15 (OUT-Befehl)
/ CANCEL                          Programmausführung abbrechen!
/ EDIT FAK ASSEMBLE               Editor zur Programmerstellung von FAK aufrufen!
------ NEW FILE
// GETFILE SUM                    Programmtext File SUM holen!
// FIND LOOP                      Zeile von LOOP im Programmtext finden!
// CHANGE / ADD / MUL /           ADD durch MUL in dieser Zeile ersetzen!
// FILE                           Programmtext als File FAK ablegen!
/ ASSEMBLE FAK                    Programm FAK assemblieren!
------ ASSEMBLED
/ LOAD                            Programm FAK laden!
------ LOADED
/ START                           Programm FAK starten!
------ STARTED
1                                 1 eingeben (INP-Befehl)
1                                 Ausgeben von 1 (OUT-Befehl)
5                                 5 eingeben (INP-Befehl)
120                               Ausgeben von 120 (OUT-Befehl)
/ CANCEL                          Programmausführung abbrechen!
/ LOGOUT                          Benutzer im Logbuch streichen!
```

als Quellcode (mit Zweiadreßbefehlen und Adreßmodifizierung im Vorgriff auf Kap.4) erstellt, assembliert, ausgeführt, abgebrochen, dupliziert und modifiziert in ein Programm zur Berechnung der Fakultät, erneut assembliert, geladen und ausgeführt. Den vom Benutzer eingegebenen Monitoranweisungen ist ein Schrägstrich und den vom Benutzer eingegebenen Editoranweisungen sind zwei Schrägstriche vorangestellt. Die "Antworten" des Rechners auf Monitor- bzw. Editoranweisungen sind durch sechs Bindestriche gekennzeichnet. ■

4. Assemblerprogrammierung und Prozessororganisation

Assemblerprogrammierung und Prozessororganisation sind zwei eng zusammenhängende Gebiete der Rechnerorganisation. Mit der Festlegung der Prozessoreigenschaften wird eine Auswahl von Handlungen bereitgestellt, die von der Hardware ausgeführt werden können und die in der Assemblersprache ausgedrückt werden müssen. Zu diesen Handlungen gehören neben den Maschinenbefehlen auch andere Vorgänge, die zur Ausführungszeit ablaufen, z.B. die Auswertung von Adreßmodifikationen.

Mit der Festlegung der Assemblersprache wird eine weitere Auswahl von Handlungen bereitgestellt, die nur mit Hilfe der Software ausgeführt werden können. Zu diesen Handlungen zählen neben den Assembleranweisungen wieder auch andere Vorgänge, die vor der Ausführungszeit, d.h. hauptsächlich zur Assemblierzeit, ablaufen, z.B. wieder die Auswertung von Adreßrechnungen.

Die vor der Ausführungszeit ausgeführten Handlungen und die Sprache, in der sie ausgedrückt werden, können sich von der Assemblersprache so weit entfernen, daß die Reihenfolge und das Format der Maschinenbefehle gar nicht mehr erkennbar sind. In höheren Programmiersprachen z.B. können arithmetische Formeln in mathematischer Schreibweise in einer Zeile geschrieben werden. Wenn die Programmiersprache solche Ausdrucksmittel enthält, die zur Ausführungszeit nicht unmittelbar von der Maschine ausgewertet werden können, dann müssen diese Handlungen vor der Ausführungszeit in eine Folge von Maschinenbefehlen übersetzt oder zur Ausführungszeit von der Maschine simuliert werden. In beiden Fällen wird die 1-zu-1-Zuordnung von Befehlen im Assemblercode zu Befehlen im Maschinencode aufgegeben zugunsten einer 1-zu-n-Zuordnung von in Programmiersprachen formulierten Anweisungen zu Befehlen im Maschinencode. Bei der Definition der Prozessorfunktion können deshalb viel einfachere Maschinenbefehle vorgesehen werden als es von der Programmiersprache her nötig erscheint. Andererseits dürfen bei der Definition der Programmiersprache die Möglichkeiten der Prozessororganisation für eine effizienoe Programmausführung nicht ganz außer acht gelassen werden.

in Abschnitt 4.1 wird eine Verallgemeinerung des Von-Neumann-Rechners zu einem weiadreßrechner mit symmetrischen Befehlen und modifizierbaren Adressen und eine Erweiterung des klassischen Assemblers zu einem Makroassembler mit Makrobe-

fehlen als Anweisungen und Ausdrücken als Adressen beschrieben. Dieses fiktive System ist einerseits genügend problemnah, um seine wichtigsten Anwendungen gut darstellen zu können, und andererseits genügend maschinennah, um sich leicht auf vorhandene Systeme übertragen zu lassen.

In Abschnitt 4.2 werden Adreßmodifizierungen beschrieben, die zur Assemblierzeit durch die Software, d.h. durch den Assembler, oder zur Ausführungszeit durch die Hardware, d.h. durch den Prozessor, ausgeführt werden. Aus der gegenüberstellenden Behandlung dieser in ihrer Wirkung innerhalb des Systems sehr unterschiedlichen Adressierungsmöglichkeiten läßt sich eine Reihe von Analogien von Assembler- und Prozessorfunktionen ableiten, die in den nächsten beiden Abschnitten weiter verfolgt werden.

In Abschnitt 4.3 werden offene und geschlossene Unterprogramme behandelt. Unterprogramme werden benutzt, um öfter auftretenden Programmcode in abkürzender Form schreiben und aufrufen zu können. Von der Assemblerprogrammierung kommend, ist es naheliegender, die geschriebene Abkürzung durch den das Unterprogramm definierenden Text zur Assemblierzeit zu ersetzen (Assembler-Unterprogramme, Makrobefehle). Von der Prozessororganisation kommend, ist es naheliegender, vom abkürzenden Aufruf aus den das Unterprogramm definierenden Maschinencode zur Ausführungszeit zu durchlaufen (Prozessor-Unterprogramme, Prozeduren). Hinsichtlich der Parameterübernahme unterscheiden sich Makrobefehle und Prozeduren dadurch, daß bei Makrobefehlen eine Textersetzung (call by name) und bei Prozeduren eine Adreßersetzung (call by reference) vorgenommen wird. Soll die Parameterübernahme bei Prozeduren wie bei Makrobefehlen erfolgen, so muß die Wirkung der Textersetzung zur Ausführungszeit des Programmes mit Hilfe eines sog. Laufzeitsystems (run-time system) simuliert werden.

In Abschnitt 4.4 werden geschachtelte, d.h. andere Unterprogramme aufrufende, und rekursive, d.h. sich selbst aufrufende Unterprogramme für die beiden Fälle behandelt, daß der Aufruf eines Makrobefehls innerhalb eines Makrobefehls und der Aufruf einer Prozedur innerhalb einer Prozedur erfolgt. Die beiden anderen Fälle, nämlich der Aufruf eines Makrobefehls innerhalb einer Prozedur und der Aufruf einer Prozedur innerhalb eines Makrobefehls; sind nicht minder wichtig. Zum Beispiel ermöglicht es der Prozeduraufruf innerhalb einer Makrodefinition, die Übersichtlichkeit des Makroaufrufes mit der Wirtschaftlichkeit der Prozedurdefinition zu vereinigen.

4.1 Verallgemeinerung des Von-Neumann-Rechners und Erweiterung des klassischen Assemblers

Bei dem in Abschnitt 2.1 beschriebenen Rechner, der sich eng an den Vorschlag von Burks, Goldstine und von Neumann anlehnt, wurden die beiden Befehlsgruppen Ein-

adreßbefehle und adressenlose Befehle unterschieden. Diese Bezeichnungen sind nicht ganz korrekt. Bei den Einadreßbefehlen wird eine Zelle des Primärspeichers (kurz eine Speicherzelle) explizit durch die Adresse im Befehl und die Adresse einer Zelle des Registerspeichers (kurz ein Register) implizit durch den Befehlscode angegeben (z.B. LDA M und LDP M). Bei den adressenlosen Befehlen wird entweder ein Operand direkt im Adreßteil und eine Registeradresse implizit im Befehlscode (Schiftbefehle) oder beide Registeradressen implizit im Befehlscode angegeben (Registerbefehle). Wir adressieren also in fast allen Befehlen zwei Operanden, ohne daß dies im Befehlsformat zum Ausdruck kommt. Die Befehle sind also weder adressenlose Befehle noch Einadreßbefehle sondern Zweiadreßbefehle, wobei die erste Adresse in jedem Fall eine Registeradresse und die zweite Adresse eine Speicheradresse oder eine Registeradresse ist (z.B. LDA M oder LAB). Eine Ausnahme bilden die Schiftbefehle und der Sprungbefehl insofern, als der zweite Operand nicht adressiert, sondern im Befehl direkt angegeben wird (z.B. SAS M und LDP M).

Wir wollen uns von dem unsystematischen Bild lösen, das der Von-Neumann-Rechner hinsichtlich seiner Befehlsstruktur darstellt. Wir systematisieren, indem wir Befehlscode und Operandenadressen voneinander trennen und zunächst nicht vorsehen, daß im Befehl Operanden direkt angegeben werden können (z.B. LD A M oder SA A M). Wir verallgemeinern, indem wir als erste Adresse jede beliebige Registeradresse und als zweite Adresse jede beliebige Speicher- oder Registeradresse zulassen (z.B. LD A M oder LD B M). Wir abstrahieren, indem wir die erste Adresse der zweiten Adresse gleichstellen, so daß jede Adresse Register- oder Speicheradresse sein kann (z.B. LD M A oder LD M N). Wie man an den Beispielen sieht, werden mit dem Befehlscode LD alle Transportoperationen bezeichnet: Transporte von Register zu Register (früher z.B. LAB), von Speicherzelle zu Register (früher z.B. LDA M), von Register zu Speicherzelle (früher STA M) und von Speicherzelle zu Speicherzelle (früher nicht vorgesehen).

Eine völlige Gleichstellung von Speicher- und Registeradressen würde bedingen, daß neben den Speicher- auch die Registeradressen in Analogie zu ORG, END, DAT, RES und EQU durch Assembleranweisungen deklariert werden müßten. Desgleichen wären in dieser Art auch die Ein- und Ausgabebusse zu kennzeichnen, wodurch die Ein- und Ausgabebefehle auch durch den Befehlscode LD dargestellt werden könnten. Wir wollen jedoch einen Kompromiß zwischen Systematik und Wirklichkeit schließen und wie früher nur Speicherzellen durch Assembleranweisungen deklarieren. Registerzellen werden wir durch fett gedruckte Symbole kennzeichnen[1], die in modular aufgebauten

---

[1] In ausgeführten Assemblern muß die Kennzeichnung von Registersymbolen in einer Form erfolgen, die vom Assembler verarbeitet werden kann, z.B. durch feste Symbole R0, R1, R2 usw., denen durch EQU-Anweisungen frei wählbare Symbole zugeordnet werden können oder deren Reihenfolge vom Assembler festgelegt wird.

Programmen in gleicher Bedeutung verwendet werden. Die Ein- und Ausgabebefehle werden in der früheren Form beibehalten. Wir vereinbaren, daß die erste Adresse stets eine Register- oder Speicheradresse und die zweite Adresse stets eine Busadresse ist.

Wie er bis jetzt als Verallgemeinerung des Von-Neumann-Rechners beschrieben wurde, ist dieser Rechner durch eine Aufteilung des Arbeitsspeichers in einen Primär- und einen Registerspeicher gekennzeichnet. Über die Wortlänge, die Kapazität und die Zugriffszeiten der Speicher soll zunächst nichts ausgesagt werden. Offensichtlich handelt es sich um ein Rechnermodell, das einen Rechner mit einem schnellen Registerspeicher und fehlendem Primärspeicher genauso wie einen Rechner mit einem langsamen Primärspeicher und fehlendem Registerspeicher als Grenzfälle einschließt. Wir verwenden deshalb beide Speicher, um Probleme, wie z.B. den Kompromiß zwischen Speicherbelegung und Ausführungszeit von Programmen, zu studieren. Außerdem muß es möglich sein, diese "abstrakte Maschine" relativ einfach auf "konkrete Maschinen" abzubilden. - Die Verallgemeinerung des Von-Neumann-Rechners ist in erster Linie als eine Art Verständigungsmittel zu betrachten, mit dem die Struktur und die Organisation digitaler Rechner so durchsichtig wie möglich, aber gleichzeitig so wirklichkeitsnah wie nötig beschrieben werden kann.

Beispiel 4.1 Summieren der Zahlen 1 bis N

Wir schreiben das Assemblerprogramm aus Beispiel 3.1 mit Zweiadreßbefehlen um. In Programm 1 beziehen sich die ersten Adressen auf Register und die zweiten Adressen auf Speicherzellen (entspricht einem Rechner mit Einadreßbefehlen). In Programm 2 arbeiten alle Befehle direkt auf den Primärspeicher, so daß sich ein Registerspeicher erübrigt (entspricht einem Rechner mit Zweiadreßbefehlen). In allen Befehlen ist die erste Adresse die "Zieladresse". Wir schreiben die Assembleranweisungen zur besseren Unterscheidung von Rechnerbefehlen im Gegensatz zu Kapitel 3 klein.

Programm 1

```
        org   36
NULL    dat   0
EINS    dat   1
LOOPA   dat   LOOP
TYP     equ   2
START   INP   N     TYP
        LD    A     NULL
        LD    I     EINS
LOOP    ADD   A     I
        ADD   I     EINS
        CMP   I     N
        NOP
        LD    P     LOOPA
        OUT   A     TYP
        HLT
        end   START
```

Programm 2

```
        org   36
EINS    dat   1
LOOPA   dat   LOOP
I       dat   1
SUM     dat   0
N       res   1
TYP     equ   2
START   INP   N     TYP
LOOP    ADD   SUM   I
        ADD   I     EINS
        CMP   I     N
        NOP
        LD    P     LOOPA
        OUT   SUM   TYP
        HLT
        end   START
```

Im Programm 1 sind mit **A**, **I** und **N** Registeradressen symbolisch bezeichnet. **P** ist die Adresse des Befehlszählers, der wie eine ausgezeichnete Zelle des Registerspeichers behandelt wird. Da für den Registerspeicher Assembleranweisungen in Analogie zu dat nicht vorgesehen sind, können die Variablen **A** und **I** nicht wie SUM und I im Programm 2 vom Assembler initialisiert werden. Dies übernehmen im Programm 1 die ersten beiden LD-Befehle. Die beiden Programme hinsichtlich ihrer Ausführungszeit zu vergleichen, würde Kenntnisse über die Anzahl von Speicherwörtern voraussetzen, die ein Befehlswort umfaßt. Man kann jedoch den Programmen entnehmen, daß im zweiten Programm bei jedem Schleifendurchgang fünf Speicheroperanden mehr als im ersten Programm zu holen sind. ■

Ein Zweiadreßrechner

Die Maschinensprache, die wir dem verallgemeinerten Von-Neumann-Rechner zugrunde legen, folgt unmittelbar aus Abschnitt 2.1. Wir erweitern sie durch den Übergang auf Zweiadreßbefehle und durch Möglichkeiten zur Modifizierung von Adressen. Die Ausführung dieser Adreßrechnungen durch ein Adressenwerk bildet den ersten Schritt zur Verbesserung der in Kapitel 2 beschriebenen Rechnerorganisation. Die Universalität des Von-Neumann-Rechners wird nicht eingeschränkt; vielmehr werden besonders häufig vorkommende Rechnungen bevorzugt verarbeitet. In Abschnitt 4.2 werden wir eine Reihe von Adreßmodifizierungen behandeln, die jedoch bereits in diesem Abschnitt mit aufgelistet sind.

Befehle des Rechners. Wir beschreiben die Befehle des Zweiadreßrechners, nach funktionellen Gesichtspunkten geordnet in folgender Reihenfolge: arithmetische Befehle, logische Befehle, Schiftbefehle, Steuerbefehle und Ein-Ausgabebefehle.

ADD N M addiert den Inhalt der Speicherzelle M auf den Inhalt der Speicherzelle N und bringt das Ergebnis in die Speicherzelle N.

SUB N M subtrahiert den Inhalt der Speicherzelle M vom Inhalt der Speicherzelle N und bringt das Ergebnis in die Speicherzelle N.

MUL N M multipliziert den Inhalt der Speicherzelle M mit dem Inhalt der Speicherzelle N und bringt das Ergebnis in die Speicherzelle N.

DIV N M dividiert den Inhalt der Speicherzelle N durch den Inhalt der Speicherzelle M und bringt das Ergebnis in die Speicherzelle N. Der bei der Division entstehende Rest geht verloren.

CMP N M vergleicht den Inhalt der Speicherzelle N mit dem Inhalt der Speicherzelle M. Der auf CMP folgende Befehl wird ausgeführt, wenn der Inhalt der Speicherzelle N kleiner ist als der Inhalt der Speicherzelle M. Der auf CMP folgende Befehl wird

übersprungen, wenn die Inhalte der Speicherzellen N und M gleich sind. Die auf CMP folgenden zwei Befehle werden übersprungen, wenn der Inhalt der Speicherzelle N größer ist als der Inhalt der Speicherzelle M.

LD N M[1] lädt die Speicherzelle N mit dem Inhalt der Speicherzelle M.

AN N M verknüpft die korrespondierenden Bits der Speicherzellen N und M durch die boolesche Operation "und" und bringt das Ergebnis in die Speicherzelle N.

OR N M verknüpft die korrespondierenden Bits der Speicherzellen N und M durch die boolesche Operation "oder" und bringt das Ergebnis in die Speicherzelle N.

NG N M negiert jedes Bit der Speicherzelle M und lädt das Ergebnis in die Speicherzelle N.

SA N M schiftet den Inhalt der Speicherzelle N um soviele Stellen, wie in der Speicherzelle M angegeben. Ist der Inhalt von M größer Null, so wird nach links geschiftet, und von rechts werden Nullen nachgezogen; ist er kleiner Null, so wird nach rechts geschiftet, und es wird der Wert des Vorzeichens nachgezogen.

SZ N M schiftet den Inhalt der Speicherzelle N um soviele Stellen, wie in der Speicherzelle M angegeben. Ist der Inhalt von M größer Null, so wird nach links geschiftet, wobei das äußerste linke Bit bei jedem Schiftschritt ganz rechts wieder geladen wird; ist er kleiner Null, so wird nach rechts geschiftet, wobei das äußerste rechte Bit bei jedem Schiftschritt ganz links wieder geladen wird.

NOP besagt, daß der nächste Befehl aufgerufen wird.

HLT läßt den Rechner in einen Wartezustand laufen. Wir benötigen den HLT-Befehl, um auf ein Unterbrechungssignal zu warten und um den Rechner am Ende eines Programmes anzuhalten.

INP N G wartet, bis der vom Eingabegerät gesendete Wert auf dem Eingabebus mit der Multiplexeradresse G erscheint und lädt ihn dann in die Zelle N.

OUT N G schaltet den Inhalt der Zelle N auf den Ausgabebus mit der Demultiplexeradresse G und wartet, bis dieser Wert vom Ausgabegerät empfangen worden ist.

Struktur des Rechners. Bild 4.1 zeigt die Struktur des Datenwerkes des Zweiadreßrechners, der Übersichtlichkeit halber in einer Ausführung, in der jeder Befehl in einem Schritt verarbeitet wird (vgl. Bild 2.31 und Fußnote auf S. 89). In Bild 4.1

---

[1] Bei Verwendung als Sprungbefehl ist die erste Adresse die symbolische Befehlszähleradresse **P**; diese Adresse ist nicht frei wählbar.

ist angedeutet, wie die vier Funktionseinheiten Primärspeicher, Registerspeicher, arithmetisch-logische Einheit und Ein-Ausgabe-Einheit ausgewählt werden. Jeder Adresse im Befehl ist ein Bit zugeordnet, das sie als Adresse des Primär- oder des Registerspeichers ausweist. Bei den Befehlen INP und OUT ist dieses Bit bei der Zweitadresse bedeutungslos, da sich die Zweitadresse auf den Ein-Ausgabe-Multiplexer-Demultiplexer bezieht. Diese beiden Bits und die Codes der INP- und OUT-Befehle steuern die Multiplexer und Demultiplexer der Operandenbusse, wodurch die entsprechenden Funktionseinheiten ausgewählt werden.

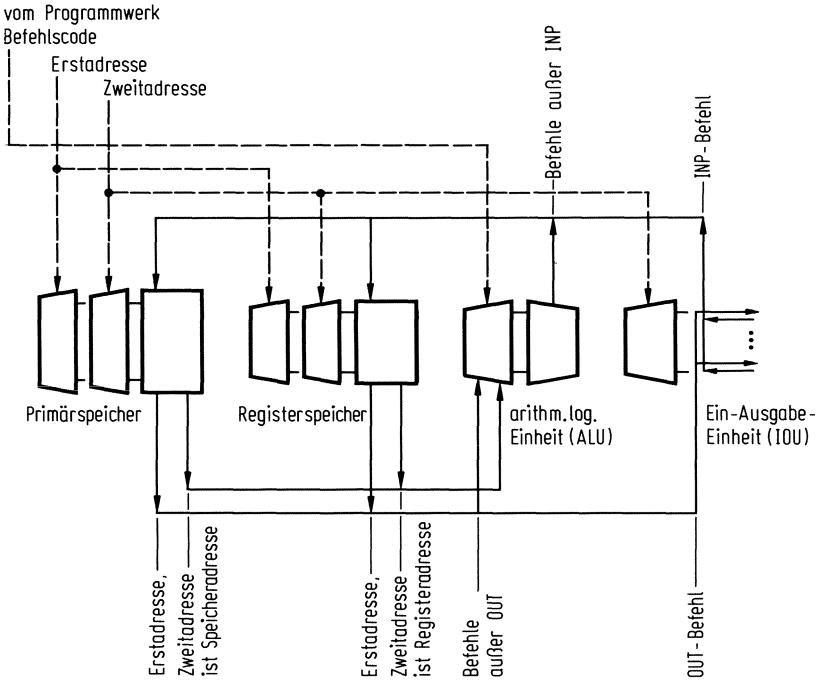


Bild 4.1 Datenwerk eines Zweiadreßrechners

Adreßmodifizierungen. Wir listen die Möglichkeiten des Zweiadreßrechners (auch für einfache Rechner zutreffend) zur Modifizierung von Adressen einzeln auf und geben anschließend die Wirkung kombinierter Modifizierungen an.

Direktorperand 'N. Die im Befehl angegebene Adresse N wird als Operand interpretiert.

Indirekte Adressierung *N. Die im Befehl angegebene Adresse N weist auf eine Speicherzelle, in der die Adresse des Operanden gespeichert ist. Die Operandenadresse ergibt sich dadurch, daß die Adresse N durch ihren Inhalt ersetzt wird.

Vorindizierung N,**R**. Die im Befehl angegebene Adresse N wird zum Inhalt des Registers **R** addiert, wodurch sich die Adresse des Operanden ergibt. Die Addition wird vor der Adreßersetzung durchgeführt, wenn Vorindizierung und indirekte Adressierung kombiniert werden.

Nachindizierung N,,**R.** Die im Befehl angegebene Adresse N wird zum Inhalt des Registers **R** addiert, wodurch sich die Adresse des Operanden ergibt. Die Addition wird nach der Adreßersetzung durchgeführt, wenn Nachindizierung und indirekte Adressierung kombiniert werden.

Diese Adreßmodifizierungen können miteinander kombiniert werden. Bild 4.2 zeigt in einer schematischen Darstellung, in welcher Reihenfolge die verschiedenen Modifikationen im Adressenwerk vorgenommen werden. Mit R[**I**] und R[**J**] sind die Inhalte der Registerspeicherzellen **I** bzw. **J** und mit P[N] der Inhalt der Primärspeicherzelle N bezeichnet. Da die Unterscheidung zwischen Register- und Primärspeicher bereits durch die Schreibweise ihrer Adressen festgelegt ist, verwenden wir anstelle der eckigen Klammern mit vorangestelltem R oder P oft nur runde Klammern. Dadurch ist es auf einfache Weise möglich, zwischen Adressen und ihren Inhalten zu unterscheiden, so daß auch das Rechnen mit Adressen vom Rechnen mit Operanden unterschieden werden kann. Bei der indirekten Adressierung wird der Inhalt der Adresse wiederum als Adresse interpretiert und kann erneut modifiziert werden. Eine Adresse besteht aus einem oder mehreren Bits für die Vorindizierung, einem Bit für die indirekte Adressierung, einem oder mehreren Bits für die Nachindizierung, einem Bit zur Unterscheidung zwischen Adresse und Operand und einer "Konstanten", die je nach Modifizierung unterschiedlich interpretiert wird: als Registeradresse, als absolute, relative oder indirekte Speicheradresse oder als relativer, direkter oder indirekter Operand. Zur Illustration vergegenwärtige man sich die Wirkung der Befehle LD **A** X und LD **A** '*X mit X dat 1 und X dat 1,**R.**

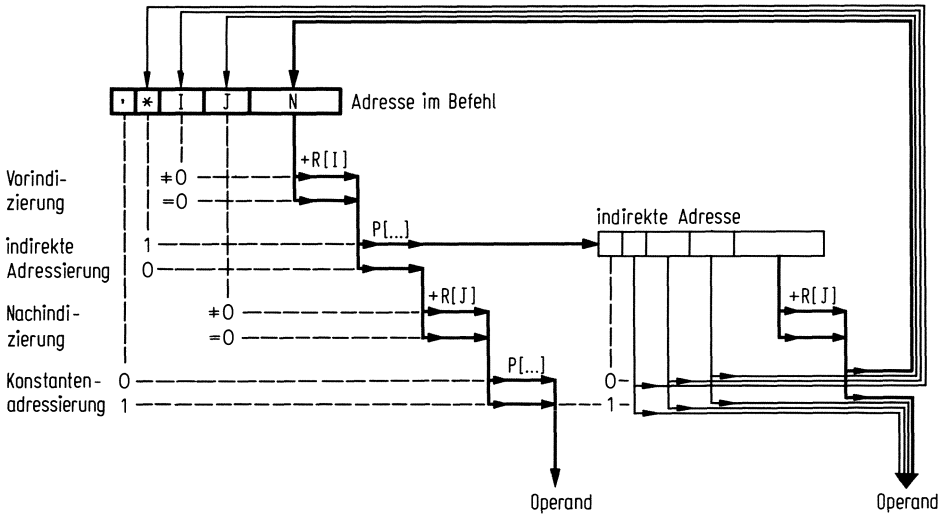


Bild 4.2 Vom Prozessor verarbeitete Adreßmodifizierungen; der Zweig unten rechts wird nur durchlaufen, wenn sowohl die Adresse im Befehl als auch die indirekte Adresse hochgestellte Kommas enthalten

Ein Makroassembler

Wir gehen von dem in Abschnitt 3.1 beschriebenen Assembler aus und ergänzen ihn durch eine Reihe weiterer Anweisungen, die insbesondere zur Übersichtlichkeit von Assemblerprogrammen beitragen. Diese Übersichtlichkeit wird durch die Einführung sog. Makrobefehle erreicht. Die Gliederung von umfangreichen Programmen durch Makrobefehle bildet den ersten Schritt zur Verbesserung der in Kapitel 3 beschriebenen Assemblerprogrammierung. Die Erweiterung des klassischen Assemblers zum Makroassembler erlaubt es u.a., nicht existierende Maschinen, wie den in diesem Kapitel vorgestellten Zweiadreßrechner, zu definieren und dessen Programme auf existierenden Maschinen ablaufen zu lassen. Im Abschnitt 4.3 befassen wir uns ausführlich mit Makrobefehlen; die dabei notwendigen Assembleranweisungen sind jedoch bereits in diesem Abschnitt mit aufgelistet.

Anweisungen des Assemblers. Wir beschreiben die Anweisungen des Makroassemblers, nach funktionellen Gesichtspunkten geordnet in folgender Reihenfolge: Zuordnungszähleranweisungen, Datenanweisungen, Symbolanweisungen, Steueranweisungen, Makroanweisungen und Bindeanweisungen.

[s] org [e] gibt mit dem Wert des Ausdruckes[1] e die Ladeadresse des nachfolgenden Programmcodes an. Dieser Wert wird einem Symbol s im Namensteil zugewiesen. Bei fehlendem Ausdruck wird die Ladeadresse als Null angenommen.

[s] aorg [e] hat dieselbe Funktion wie org, jedoch ist der auf aorg bezogene Maschinencode absolut, d.h. er kann vom Lader nicht verschoben werden.

[s] res e reserviert soviele Speicherzellen, wie der Wert des Ausdruckes e angibt. Ein Symbol s im Namensteil gibt die symbolische Adresse des ersten Speicherwortes an.

[s] dat a generiert ein Datenwort mit dem Wert des im Adreßteil a auftretenden Ausdrucks. Ist dies eine Adresse, so werden zusätzliche Angaben zur Adreßmodifizierung durch den Prozessor ebenfalls gespeichert. Ein Symbol s im Namensteil gibt die symbolische Adresse des Wortes an.

[s] text "Zeichenfolge" generiert Datenwörter, die mit den Zeichen der Zeichenfolge belegt werden. Pro Datenwort werden je zwei Zeichen in EBCDI-codierter Form gespeichert. Die Zeichenfolge wird von links nach rechts verarbeitet; das erste Datenwort kann mit einem Symbol im Namensfeld markiert werden.

---

[1] Erklärung von "Ausdruck" s. Bild 4.3.

s equ e setzt das Symbol s im Namensteil gleich dem Wert des Ausdruckes e im Adreßteil. Eine nachfolgende Änderung des Ausdruckes hat eine Änderung des Wertes von s zur Folge.

s set e ersetzt den Wert von s im Namensteil durch den Wert des Ausdruckes e im Adreßteil. Der Ausdruck kann s mit seinem bisher gültigen Wert enthalten. Innerhalb eines Programmes können mehrere set-Anweisungen mit gleichen Symbolen im Namensteil benutzt werden.

[s] goto e[1] verweist mit dem Wert des Ausdruckes e auf diejenige Programmzeile, mit der die Assemblierung fortzusetzen ist. Ein Symbol s im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

[s] if e ∘ e setzt die Assemblierung mit der nachfolgenden Programmzeile fort, wenn die beiden Ausdrücke e die Vergleichsrelation ∘ erfüllen. Sonst wird die nachfolgende Zeile bei der Assemblierung übersprungen. Ein Symbol s im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

[s] end [e] bezeichnet die letzte zu assemblierende Zeile eines Programms. Der Wert des Ausdruckes e gibt die Startadresse des Programms an. Ist kein Ausdruck angegeben, so wird die mit der ersten org-Anweisung definierte Ladeadresse angenommen. Ein Symbol im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

[s] makro $[p]_0^n$ bringt den Assembler vom Normalmodus in den Makroersetzungsmodus. Das Adreßfeld der Anweisung enthält die 0 bis n formalen Parameter p des Makrobefehls. Ein Symbol s im Namensteil gibt die symbolische Adresse des ersten im Makrobefehl auftretenden Befehls oder der ersten im Makrobefehl auftretenden Anweisung an.

[s] mend kennzeichnet das Ende eines Makrobefehls und bringt den Assembler vom Makroersetzungsmodus in den Normalmodus. Ein Symbol s im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

[s] ext s gibt an, daß das Symbol s im Adreßteil in diesem Programmodul verwendet, jedoch in einem anderen Modul definiert ist. Das Symbol s im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

[s] ent s gibt an, daß das Symbol s im Adreßteil in diesem Programmodul definiert, jedoch auch in anderen Moduln verwendet ist. Das Symbol s im Namensteil kann als Sprungziel einer goto-Anweisung benutzt werden.

---

[1] Der Einfachheit halber verwenden wir für LD-**P**-Befehle und für goto-Anweisungen gleichartige Sprungmarken.

Funktion des Makroassemblers. Die Aufgabe eines Makroassemblers ist es, den Rumpf einer Makrodefinition, d.h. die Befehlsfolge zwischen den Anweisungen makro und mend, an jenen Stellen in das Programm einzusetzen, an denen der Makrobefehl verwendet wird. Gleichzeitig werden dabei die formalen Parameter der Makrodefinition durch die aktuellen Parameter im Makroaufruf ersetzt. Wir wollen uns diesen Ersetzungsvorgang bei einem 2-Phasen-Assembler genauer ansehen[1].

In der Phase 1 verarbeitet der Makroassembler zunächst die Makrodefinitionen. Die Anweisung makro versetzt ihn in den Makrodefinitionsmodus, in dem der Makroname und der Makrorumpf in einer sog. Makrodefinitionstabelle aufgenommen werden. Die im Makrorumpf vorkommenden formalen Parameter werden dabei gemäß der Reihenfolge ihres Auftretens in der Parameterliste der makro-Anweisung durch fortlaufende Nummern ersetzt. Die Anweisung mend bringt den Makroassembler wieder in den Normalmodus.

In der Phase 2 verarbeitet der Makroassembler die Makroaufrufe. Er vergleicht jeden symbolischen Operationscode zunächst mit den in der Makrodefinitionstabelle eingetragenen Namen. Bei Übereinstimmung geht er in den Makroersetzungsmodus über. Die Befehle des Makrorumpfes entnimmt er der Makrodefinitionstabelle und fügt sie - beginnend am Ort des Aufrufes - an die bereits generierte Befehlsfolge an. Dabei werden die durch Nummern gekennzeichneten formalen Parameter des Makrorumpfes durch die aktuellen Parameter des Aufrufes ersetzt. Die Zuordnung der Parameter ist durch die Reihenfolge in den beiden Parameterlisten gegeben. Nach der Ersetzung wird die Assemblierung im Normalmodus mit den auf den Aufruf folgenden Befehlen und Anweisungen fortgesetzt. Die Makrodefinitionen werden in der Phase 2 übergangen.

Symbolische Operationscodes, die nicht in der Makrodefinitionstabelle stehen, werden wie bisher mit den Eintragungen in der Zuordnungstabelle verglichen. Der Assembler bleibt im Normalmodus. Diese Reihenfolge in der Interpretation des Operationscodes erlaubt es, den Code der Standardbefehle als Makrobefehle mit veränderter Funktion zu benutzen.

Die Steueranweisungen if und goto finden häufig in Makrodefinitionen Verwendung. Mit ihnen lassen sich z.B. Makrobefehle schreiben, bei denen der Makroassembler je nach Art des Makroaufrufes unterschiedliche Befehlsfolgen generiert. Die Befehlsgenerierung hängt dabei von Bedingungen ab, die als aktuelle Parameter beim Aufruf (Phase 2) übergeben werden und in den if- und goto-Anweisungen des Makrorumpfes ausgewertet werden. Man spricht in diesem Zusammenhang auch von bedingter Assemblierung.

Adreßmodifizierungen. Wir listen die Möglichkeiten des Makroassemblers (auch für einfache Assembler zutreffend) zur Modifizierung von Adressen einzeln auf und geben anschließend die Wirkung kombinierter Modifizierungen an.

Literal =e. Der Wert des im Befehl angegebenen Ausdrucks e wird als Operand benutzt.

Addition $e_1 + e_2$. Die Werte der Ausdrücke e_1 und e_2 werden addiert. Sind e_1 und e_2 absolute Ausdrücke, so ist auch ihre Summe ein absoluter Ausdruck; ist einer von beiden ein verschieblicher Ausdruck, so ist auch ihre Summe ein verschieblicher Ausdruck. Es ist nicht zulässig, daß beide Ausdrücke verschieblich sind.

---

[1] In Anlehnung an die Darstellung von Barron, D.W.: Assembler und Lader. München: Hanser 1970, S. 82ff.

Subtraktion $e_1 - e_2$. Die Werte der Ausdrücke e_1 und e_2 werden subtrahiert. Sind e_1 und e_2 entweder zwei absolute oder zwei relative Ausdrücke, so ist ihre Differenz ein absoluter Ausdruck; anderenfalls ist ihre Differenz ein relativer Ausdruck.

Diese Adreßmodifizierungen können miteinander kombiniert werden. Bild 4.3 zeigt in einer schematischen Darstellung, welche Möglichkeiten von Adreßmodifikationen erlaubt sind. Jedes Schema wird von links nach rechts durchlaufen, wobei Alternativen durch Verzweigungen und Wiederholungen durch Schleifen gekennzeichnet sind. Feste Symbole sind in Kreisen dargestellt; $ hat den aktuellen Stand des Zuordnungszählers als Wert. Die in Kästen verwendeten Symbole und Konstanten sind frei wählbar im Sinne von Abschnitt 3.1. Zur Illustration möglicher Konstruktionen vergegenwärtige man sich die Bildung von =-3+$ und von P1+P1+3 mit P1 set 3.

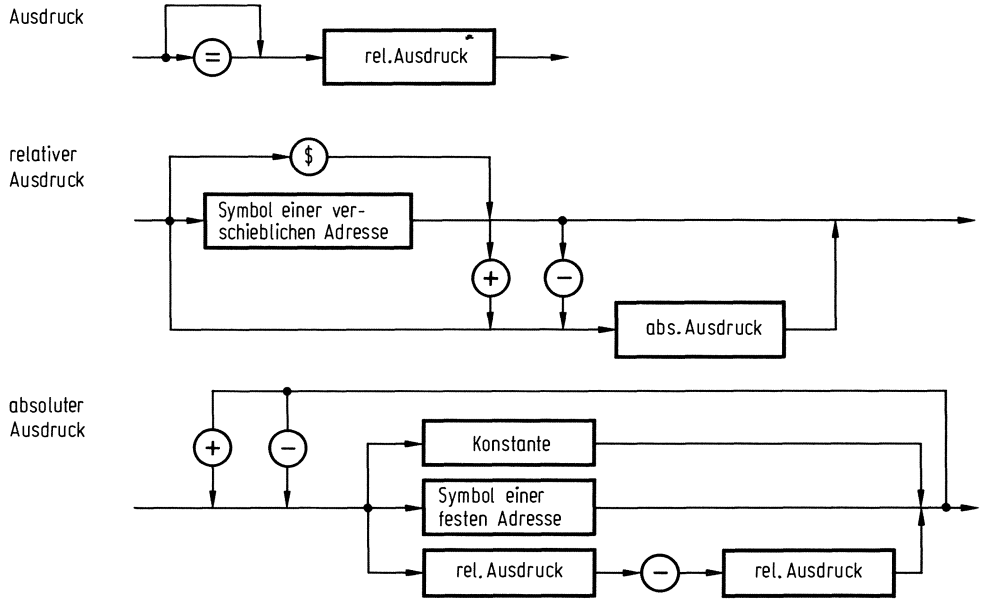


Bild 4.3 Vom Assembler verarbeitete Adreßmodifizierungen

4.2 Adreßmodifizierungen durch Software (Assembler) und Hardware (Prozessor)

Wir beschreiben Adreßmodifikationen, die bei der Assemblierung von Programmen durch die Rechnersoftware und die bei der Ausführung von Programmen durch die Rechnerhardware vorgenommen werden, unter gemeinsamer Überschrift. Auf diese Weise treten ihre Ähnlichkeiten in der Anwendung und ihre Unterschiede in der Wirkung deutlicher zutage.

Bereits im Von-Neumann-Rechner gab es eine Möglichkeit zur Adreßmodifizierung, nämlich die Ersetzung einer Adresse im Befehl durch eine andere Adresse. Für diese Adreßsubtitution wurden von Burks, Goldstine und von Neumann spezielle Befehle vorgesehen. Adreßrechnungen, wie etwa das Hochzählen von Adressen bei der Verarbeitung von Feldern, lassen sich damit etwas einfacher als in Beispiel 3.3 programmieren. Dessen ungeachtet kann dies in Assemblersprache noch weit übersichtlicher geschrieben und vom Prozessor wesentlich effizienter ausgeführt werden, wenn das Hochzählen der Adresse von einem Adressenrechenwerk vorgenommen wird. Anstelle besonderer Befehle wird die Anweisung für das Hochzählen der Adresse direkt hinter der zu modifizierenden symbolischen Adresse angegeben. Dadurch wird die Schleife kürzer, übersichtlicher und kann schneller ausgeführt werden.

Konstanten wurden bisher durch Variablen mit unveränderten Werten dargestellt, was die Übersichtlichkeit von Programmen nicht fördert. Dem Programm aus Beispiel 3.1 ist nicht unmittelbar anzusehen, welche Symbole Konstanten und welche Variablen bezeichnen. Auch dieser Unterschied soll durch Adreßmodifizierungen ausgedrückt werden können, so daß die in den Befehlen erscheinenden symbolischen Adressen sich nur noch auf Variablen beziehen.[1]

Die beiden zitierten Beispiele 3.1 und 3.3 werden wir im folgenden laufend zur Illustration der Adreßmodifizierungen heranziehen, obwohl sie nicht immer als repräsentativ bezüglich ihrer Verwendung anzusehen sind. Es empfiehlt sich, sie mit Bleistift und Papier zu simulieren, um sich die unterschiedliche Wirkung von Software- und Hardwaremodifizierungen zur Assemblier- bzw. zur Ausführungszeit zu verdeutlichen.

Konstantenadressierung

Die Konstantenadressierung ermöglicht es, den Wert einer Konstanten oder eine symbolische Adresse, die als Konstante aufgefaßt wird, direkt im Adreßteil eines Befehls anzugeben. Die Art der Adreßmodifikation muß in der Assemblersprache durch ein Symbol gekennzeichnet werden.

Assembler. Die Konstante wird im Adreßfeld der zweiten (bei CMP auch der ersten) Adresse angegeben. Um sie für den Assembler von einer numerischen Adresse zu unterscheiden, wird ihr ein Gleichheitszeichen vorangestellt (ADD I =1 oder LD **P** =LOOP in Beispiel 4.2). Diese syntaktische Einheit wird als Literal bezeichnet. Der Assem-

---

[1] Diese Aussage muß relativiert werden: Wenn man alle symbolischen Adressen ausschließlich als Variablennamen auffaßte, würde man sich der Möglichkeit begeben, in den Fällen mit Adressen wie mit Variablen zu rechnen, in denen die im Rechner vorgesehenen Adreßmodifikationen nicht ausreichen.

bler ersetzt beim Assemblieren des Programms alle Literale durch Adressen und erzeugt - diesen zugeordnet - den Code der Konstanten. In der Assemblersprache können durch das Gleichheitszeichen Konstanten von anderen Größen unterschieden werden (vgl. Bild 4.3). Beim Assemblieren werden aus ihnen Variablen mit konstantem Wert. In der Maschinensprache sind sie nicht mehr von Variablen zu unterscheiden.

Prozessor. Die Konstante wird im Adreßfeld der zweiten (bei CMP auch der ersten) Adresse angegeben. Um sie für den Assembler von einer numerischen Adresse zu unterscheiden, wird ihr ein hochgestelltes Komma vorangestellt (ADD I '1 oder LD **P** 'LOOP in Beispiel 4.2). Diese syntaktische Einheit wird als Direktoperand bezeichnet. Der Assembler setzt aufgrund des hochgestellten Kommas im Befehl das Direktoperandbit und setzt den Code der Konstanten an die Stelle der Adresse. Der Prozessor kann dadurch zwischen einem Direktoperanden und einer Adresse unterscheiden (vgl. Bild 4.2).

Beispiel 4.2 Summieren der Zahlen 1 bis N

Wir verwenden das Programm 2 aus Beispiel 4.1 und ersetzen diejenigen dat-Anweisungen, die zur Definition von Konstanten dienen, durch Adreßmodifikationen. Das linke Programm benutzt Literale. Es entspricht hinsichtlich Speicherbedarf und Ausführungszeit dem Programm 2 aus Beispiel 4.1. Das rechte Programm benutzt Direktoperanden. Verglichen mit dem linken Programm benötigt es zwei Speicherplätze weniger, und in der Schleife entfällt zweimal das Lesen der Operanden. Besonders gut kommt in beiden Programmen der Unterschied zwischen den Konstanten (1 und LOOP), den Variablen (SUM und N) und der Geräteadresse (TYP) zum Ausdruck.

Konstantenadressierung im Assembler

```
        org   36
I       dat   1
SUM     dat   0
N       res   1
TYP     equ   2
START   INP   N      TYP
LOOP    ADD   SUM    I
        ADD   I      =1
        CMP   I      N
        NOP
        LD    P      =LOOP
        OUT   SUM    TYP
        HLT
        end   START
#1      dat   1      vom Assembler erzeugt
#2      dat   LOOP   und hinzugefügt
```

Konstantenadressierung im Prozessor

```
        org   36
I       dat   1
SUM     dat   0
N       res   1
TYP     equ   2
START   INP   N      TYP
LOOP    ADD   SUM    I
        ADD   I      '1
        CMP   I      N
        NOP
        LD    P      'LOOP
        OUT   SUM    TYP
        HLT
        end   START
```

■

Indexadressierung

Bei fast allen Algorithmen der Datenverarbeitung spielen Felder eine große Rolle, deren Elemente entsprechend der byte- oder wortorganisierten Speicher eindimensional gespeichert sind. Ein solches Feld hat eine Adresse, die auf das erste Element des Feldes verweist. Die übrigen Elemente werden durch Addition einer Distanzangabe zu dieser Feldanfangsadresse erreicht. Diese arithmetische Adreßrechnung kann sowohl während der Assemblierung durch die Software als auch während der Ausführung des Programms durch die Hardware vorgenommen werden. Die Distanzangaben werden dementsprechend bereits zur Assemblierzeit bzw. sie werden erst zur Ausführungszeit benötigt. Die Art der Adreßmodifikation muß in der Assemblersprache durch Symbole angegeben werden: das eine Symbol dient zur Kennzeichnung der Adreßoperation (der Addition), und das andere Symbol zur Kennzeichnung des Operanden (der Distanz).

Assembler. Das Operationszeichen für die Indexadressierung während der Assemblierzeit ist das Pluszeichen. Der erste Operand (die Feldadresse) ist eine Primärspeicheradresse, und der zweite Operand (die Distanz) wird durch ein Symbol angegeben (ADD X+I Y+I in Beispiel 4.3). Der Assembler berechnet den Wert dieses Ausdruckes, wobei die Adresse und die Distanz zur Assemblierzeit bekannt sein müssen (vgl. Bild 4.3). Dieser Wert ist die numerische Adresse, die im assemblierten Maschinenbefehl erscheint. Im Adreßfeld eines Befehles treten also einfache arithmetische Ausdrücke auf, und es ist naheliegend, in diesen Ausdrücken auch höhere arithmetische Operationen als die Addition zu verwenden[1]. Auf diese Weise gelangt man zu komplizierten Formeln, die innerhalb des Adreßfeldes auftreten und die vom Assembler ausgewertet werden. Die mehrfache Verwendung von arithmetischen Operationen ist durch gewisse Einschränkungen eingeengt, die auf der Forderung nach verschieblichen Programmen beruhen. Sie erfordert genaue Beachtung der syntaktischen und semantischen Regeln zur Bildung von Adreßausdrücken.

Prozessor. Das Operationszeichen für die Indexadressierung während der Ausführungszeit ist das Komma. Der erste Operand (die Feldadresse) ist eine Primärspeicheradresse, und der zweite Operand (die Distanz) ist in einem Register gespeichert (dem Indexregister), das durch ein Symbol angegeben wird (ADD X,I Y,I in Beispiel 4.3). Der Assembler setzt aufgrund des Kommas die symbolische in die numerische Indexregisteradresse um. Der Prozessor addiert zur Ausführungszeit den Inhalt des Indexregisters zur Primärspeicheradresse. Dieser Wert ist die effektive Adresse, mit der der Operand des Befehls adressiert wird (vgl. Bild 4.2).

---

[1] Neben den arithmetischen sind auch boolesche Ausdrücke in vielen Assemblern für die Adreßmodifizierung zur Assemblierzeit vorgesehen.

Beispiel 4.3 Vektoraddition

Wir greifen auf das in Beispiel 3.3 wiedergegebene Programmstück zur Addition von zwei Feldern zurück und geben Ausschnitte aus zwei Assemblerprogrammen an, in denen die Adreßrechnung während der Assemblierphase bzw. während der Ausführungsphase durchgeführt wird. Nach dem Assemblieren dieser beiden Programme entstehen zwei vollständig verschiedene Maschinenprogramme. Das linke Programm ist ein Geradeausprogramm, bei dem die Feldlänge bereits zur Assemblierzeit bekannt sein muß. Das rechte Programm ist ein Schleifenprogramm, bei dem die Feldlänge erst zur Ausführungszeit bekannt zu sein braucht. Bei beiden Programmen wird jedoch der Einfachheit halber angenommen, daß die Feldlänge beim Schreiben der Programme bekannt ist. Das Geradeausprogramm benötigt sehr viel mehr Speicherplatz als das Schleifenprogramm. Die Ausführungszeit des Geradeausprogramms ist etwas geringer als die des Schleifenprogramms. Die Schreibarbeit beider Programme ist etwa die gleiche.

Indexadressierung im Assembler

```
        org
N       set   100
X       res   100
Y       res   100
         :
START    :
         :
I       set   0
LOOP    ADD   X+I    Y+I
I       set   I+1
        if    I≠N
        goto  LOOP
         :
        end   START
```

Indexadressierung im Prozessor

```
        org
N       dat   100
X       res   100
Y       res   100
         :
START    :
         :
        LD    I     '0
LOOP    ADD   X,I   Y,I
        ADD   I     '1
        CMP   I     N
        LD    P     'LOOP
         :
        end   START
```

■

Basisadressierung

Für die Indexadressierung ist charakteristisch, daß die im Befehl angegebene Primärspeicheradresse konstant bleibt und sich der Indexregisterinhalt ändert. Die typische Anwendung ist das Adressieren von Feldelementen in einer Schleife, wobei die Feldadresse konstant und die Distanz variabel ist.

Umgekehrt ist es möglich, zur Adressierung der Feldelemente die im Befehl angegebene Primärspeicheradresse zu verändern und den Indexregisterinhalt konstant zu halten. Man spricht dann nicht mehr von Index- sondern von Basisadressierung. Die Feldadresse wird in einem Register (dem Basisregister) gespeichert, und die Distanz wird als relative, numerische Adresse im Befehl angegeben.

Man bezeichnet diese Adreßmodifizierung auch als relative Adressierung. Die Anwendung der relativen Adressierung betrifft nicht so sehr die Adressierung von Feldern, sondern mehr die Adressierung von Programmen oder Datenbereichen. Der Programmanfang, d.h. die Adresse des ersten Befehls, oder der Datenanfang, d.h. die Adresse des ersten Operanden, wird als Basisadresse benutzt, und alle Adressen werden relativ dazu angegeben. Dann ist es möglich, Programme und Daten in unterschiedliche Bereiche des Speichers zu laden, ohne daß alle Adressen der Programme verändert zu werden brauchen. Lediglich die Basisadressen müssen geändert werden, so daß sie mit dem jeweiligen variablen Programm- bzw. Datenanfang übereinstimmen.

Soweit beschrieben, ist die Basisadresse als konstant angenommen. Eine Variante der relativen Adressierung entsteht, wenn die Basisadresse variabel ist. Als variable Basisadresse für Programmbereiche kann der Zuordnungszählerstand des Assemblers oder der Befehlszählerstand des Prozessors gewählt werden[1].

In der symbolischen Schreibweise und in der Ausführung unterscheidet sich die Basisadressierung nicht von der Indexadressierung, obwohl bei der Basisadressierung auch negative Distanzen erscheinen können.

Assembler. Wie bei der Indexadressierung wird die Operation durch + und die Basisadresse symbolisch bezeichnet (ADD 1+R 101+R in Beispiel 4.4). Der Zuordnungszählerstand wird durch das Symbol $ gekennzeichnet. Er wird durch die org-Anweisung gesetzt und hat als aktuellen Stand den Wert, der dem Befehl entspricht, in dem das $-Zeichen verwendet wird (LD **P** = -3+$ in Beispiel 4.4).

Prozessor. Wie bei der Indexadressierung wird die Operation durch ein Komma und die Basisadresse symbolisch bezeichnet (z.B. ADD 1,**R** 101,**R**). Der Befehlszählerstand wird durch das Symbol **P** gekennzeichnet. Der Befehlszähler wird durch einen Ladebefehl gesetzt und hat als aktuellen Stand den Wert, der dem nächsten Befehl entspricht, in dem das Symbol **P** verwendet wird (LD **P** '-4,**P** in Beispiel 4.4).

Beispiel 4.4 Vektoraddition

Wir benutzen das Schleifenprogramm aus Beispiel 4.3 und adressieren die Operanden relativ zu einer Datenbasisadresse und die Befehle relativ zum Zuordnungs- bzw. Befehlszählerstand. Im linken Programm wird die Datenbasisadresse zur Assemblierzeit und im rechten Programm zur Ausführungszeit gesetzt. Die Basisadressie-

---

[1] Als variable Basisadresse für Datenbereiche kann bei Kellerspeichern ein Register benutzt werden, das als Vorwärts-Rückwärtszähler ausgebildet ist (vgl. Fußnote auf S. 157).

rung wird dementsprechend ebenfalls zur Assemblier- bzw. Ausführungszeit vorgenommen. Die Programme arbeiten mit Daten, die in den Zellen 1000, 1001 bis 1100 und 1101 bis 1200 gespeichert sind.

Relative Adressierung im Assembler

```
        aorg   1000
N       res    1
X       res    100
Y       res    100
        :
        org
START   :
R       set    1000
        LD     I       =0
        ADD    1+R,I   101+R,I
        ADD    I       =1
        CMP    I       0+R
        LD     P       =-3+$
        :
        end    START
```

Relative Adressierung im Prozessor

```
        aorg   1000
N       res    1
X       res    100
Y       res    100
        :
        org
START   :
        LD     R       '1000
        LD     I       '0
        ADD    1,R,I   101,R,I
        ADD    I       '1
        CMP    I       0,R
        LD     P       '-4,P
        :
        end    START
```

Um die Programme an beliebiger Stelle (den Datenbereich 1000 bis 1200 ausgenommen) zu speichern, muß beim linken Programm die Sprungadresse, die der Assembler zu 1 berechnet hat, um die Programmbasisadresse erhöht werden. Diese in der Ladephase vorgenommene relative Adressierung wird vom verschiebenden Lader ausgeführt. Beim rechten Programm ist dies nicht nötig, da die relative Adressierung erst in der Ausführungsphase vorgenommen wird.

Die relative Adressierung ist in unserem Zusammenhang für die dynamische Speicherverwaltung von Bedeutung (vgl. Abschnitt 4.3: Statische und dynamische Speicherverwaltung). ■

Indirekte Adressierung

Bei der indirekten Adressierung erscheint im Befehl nicht die Operandenadresse, sondern die Adresse einer Speicherzelle, die die Operandenadresse enthält. Bereits die Basisadressierung erlaubt es, Operanden indirekt zu adressieren. Wir schreiben z.B. XX set X für den Assembler bzw. LD **XX** X für den Prozessor und beziehen uns innerhalb des Programms auf 0+XX bzw. 0,**XX**, um mit dem Operanden X zu arbeiten (z.B. LD A 0+XX bzw. LD A 0,**XX**). Die indirekte Adressierung ist vor allem in der Unterprogrammtechnik von Bedeutung (vgl. Abschnitt 4.3). Soll nämlich ein- und dasselbe Programm mehrfach benutzt werden, so braucht es deshalb nicht unbe-

dingt auch mehrfach niedergeschrieben oder mehrfach gespeichert zu werden. Stattdessen genügt es, die Niederschrift bzw. die Speicherung des Programms nur einmal vorzunehmen und die Operandenadressen indirekt statt direkt anzugeben. Bei der Assemblierung bzw. bei der Ausführung des Programms werden die indirekten durch direkte Operandenadressen ersetzt.

Assembler. Die indirekte Adressierung zur Assemblierzeit besteht aus Symbolersetzungen mit Hilfe von set-Anweisungen. Es bedarf somit keiner besonderen Kennzeichnung zur Unterscheidung von direkten und indirekten Adressen im Befehl (CMP **I** P1 mit P1 set N in Beispiel 4.5).

Prozessor. Die indirekte Adressierung zur Ausführungszeit besteht aus einer Substitution, wobei die im Befehl angegebene Adresse durch ihren Inhalt ersetzt wird. Dieser Inhalt ist die Adresse des Operanden. Um an einen indirekt adressierten Operanden zu gelangen, sind zwei Speicherzugriffe notwendig. Um die indirekte von der direkten Adressierung unterscheiden zu können, wird für den Assembler vor die symbolische Adresse ein hochgestellter Stern gesetzt (CMP **I** *P1 mit P1 res 1 in Beispiel 4.5). Der Assembler setzt aufgrund dieses Sternes im Befehl das Indirektbit. Der Prozessor wertet dieses beim Ausführen des Programmes aus und ersetzt die indirekte durch die direkte Adresse. Dies ist die Adresse, mit der der Operand adressiert wird (vgl. Bild 4.2).

Beispiel 4.5 Vektoraddition

Wir schreiben zwei kurze Programmstücke zur Addition von zwei Vektoren, in denen der jeweilige Feldanfang und die Feldlänge indirekt adressiert werden. Unmittelbar vor der Assemblierung bzw. vor der Ausführung dieser Programmstücke werden die direkten Adressen bereitgestellt. Dann folgt eine Anweisung an den Zuordnungszähler ($) bzw. ein Befehl an den Befehlszähler (**P**), an anderer Stelle mit Assemblieren bzw. Ausführen fortzufahren (Aufruf der Additionsprogramme). Am Ende des Additionsprogramms steht eine Anweisung bzw. ein Befehl, mit Assemblieren bzw. Ausführen unmittelbar nach dem Aufruf fortzufahren. Das bedingt das "Sich-Merken" des Zuordnungs- bzw. Befehlszählerstandes des auf den Aufruf folgenden Befehles.

Im linken Programm finden die Substitutionen der indirekten durch die direkten Operandenadressen während der Assemblierzeit statt, indem mit Hilfe der set-Anweisungen die Symbole P1,P2,P3 durch die Symbole N,X,Y bzw. N,X,Z ersetzt werden. Im rechten Programm finden die Substitutionen der indirekten durch die direkten Operandenadressen während der Ausführung der einzelnen Befehle statt, indem die Indirektbits ausgewertet und vorübergehend die Adressen P1,P2,P3 durch die Adressen N,X,Y bzw. N,X,Z ersetzt werden. Dazu ist es notwendig, für P1,P2,P3 Speicherplätze zu reservieren und unmittelbar vor dem Aufruf des Additionsprogrammes die Adressen N,X,Y bzw. N,X,Z in diese Zellen zu transportieren.

Adreßsubstitution in der Assemblierphase

```
                org
                 :
Operanden  N    res   1
           X    res   100
           Y    res   100
           Z    res   100
                 :
      START      :
                 :
1. Aufruf  P1   set   N
           P2   set   X
           P3   set   Y
           P0   set   L0
           goto VADD
      L0         :
                 :
2. Aufruf  P1   set   N
           P2   set   X
           P3   set   Z
           P0   set   L1
           goto VADD
      L1         :
                 :
                 :
                 :
           HLT
           goto FIN
Unterprogramm
      VADD LD    I     '0
           ADD   P2,I  P3,I
           ADD   I     '1
           CMP   I     P1
           LD    P     '-4,P
           goto  P0
      FIN  end   START
```

Adreßsubstitution in der Ausführungsphase

```
                org
                 :
      N          res   1
      X          res   100
      Y          res   100
      Z          res   100
                 :
      START      :
                 :
                 LD    P1    'N
                 LD    P2    'X
                 LD    P3    'Y
                 LD    P0    'L0
                 LD    P     'VADD
      L0         :
                 :
                 LD    P1    'N
                 LD    P2    'X
                 LD    P3    'Z
                 LD    P0    'L1
                 LD    P     'VADD
      L1         :
                 :
                 HLT
      P0         res   1
      P1         res   1
      P2         res   1
      P3         res   1
      VADD       LD    I     '0
                 ADD   *P2,,I *P3,,I
                 ADD   I     '1
                 CMP   I     *P1
                 LD    P     '-4,P
                 LD    P     P0
                 end   START
```

An diesem Beispiel sieht man gut, wie sich die Wirkung von Konstantenadressierung und indirekter Adressierung zusammengenommen aufheben: zum Beispiel wird mit LD P1 'N die Adresse N nach P1 gebracht (Konstantenadressierung), und hernach wird im Befehl CMP **I** *P1 auf diese Adresse indirekt Bezug genommen (indirekte Adressierung). Stattdessen hätte die zweite Adresse im CMP-Befehl mit einem (hier nicht vorgesehenen) Adreßersetzungsbefehl durch N ersetzt werden können, wodurch die indirekte Adressierung entfallen wäre. Dann hätten aber die Befehlswörter selbst wieder als Operanden benutzt werden müssen (vgl. Beispiel 3.3). Die indirekte Adressierung ermöglicht es jedoch, Adressen von den Befehlen zu trennen und als gesonderte Größen zu behandeln.

Das linke und das rechte Programm unterscheiden sich in der Ausführungsphase stark voneinander. Das linke Programm ist vom Assembler dupliziert worden und steht zwei-

fach im Speicher. Die Adressierung der Operanden ist direkt. Die Ausführungszeit eines solchen Programms ist dementsprechend wesentlich geringer als die des rechten Programms. Das rechte Programm steht nur einfach im Speicher. Der Aufruf und die Parameterübergabe kosten Speicherplatz und Ausführungszeit. Diese Form der Programmierung wird man dementsprechend nur für größere Programme anwenden, bei denen man sich eine doppelte oder mehrfache Speicherung nicht leisten kann. ■

4.3 Offene und geschlossene Unterprogramme

Um z.B. komplizierte Aufgaben übersichtlich zu beschreiben, ersetzt man immer wieder auftretende Teilaufgaben durch abkürzende Definitionen. Den sich ändernden Größen dieser Teilaufgaben werden Parameter zugeordnet, die als formale Bezüge bei der Definition der Teilaufgabe dienen und denen vor der Benutzung der Teilaufgabe aktuelle Größen zugewiesen werden. Bei komplizierten Programmen entsprechen den Teilaufgaben immer wiederkehrende Befehlsfolgen. Stellt man sich die Struktur komplexer Programme hierarchisch gegliedert vor, so operiert ein solches Teilprogramm unter einem Hauptprogramm. Es wird Unterprogramm genannt und benutzt bei seiner Ausführung die aktuellen Parameter stellvertretend für die bei seiner Niederschrift benutzten formalen Parameter. Das Hauptprogramm ruft das Unterprogramm mit seinem Namen auf. Es übergibt die Steuerung dem Unterprogramm, das seinerseits nach Beendigung seiner Aufgabe die Steuerung an das Hauptprogramm zurückgibt.

Auch der Begriff des Unterprogramms läßt sich sowohl auf den Prozessor als auch auf den Assembler anwenden, und wir sprechen von Prozessor- bzw. Assembler-Unterprogrammen. Ein Prozessor-Unterprogramm ist ein nur einmal vorhandenes Stück Maschinencode, das von einem Maschinenprogramm, das ausgeführt werden soll, von verschiedenen Stellen aus aufgerufen werden kann. Ein Assembler-Unterprogramm ist ein nur einmal vorhandenes Stück Assemblercode, das vom Assembler an verschiedenen Stellen des Assemblerprogramms, das assembliert werden soll, eingesetzt wird. Das Assembler-Unterprogramm spart Schreibarbeit, das Prozessor-Unterprogramm darüber hinaus Speicherplatz. Für den Befehlszähler sind Assembler-Unterprogramme zur Ausführungszeit "offen" (Bild 4.4 links) und Prozessor-Unterprogramme zur Ausführungszeit "geschlossen" (Bild 4.4 rechts). Man nennt dementsprechend die Assembler-Unterprogramme offene Unterprogramme und die Prozessor-Unterprogramme geschlossene Unterprogramme.

In Abschnitt 4.2 haben wir bereits Unterprogramme zur Illustration der indirekten Adressierung benutzt. Die dort verwendete Organisation des Aufrufs und der Para-

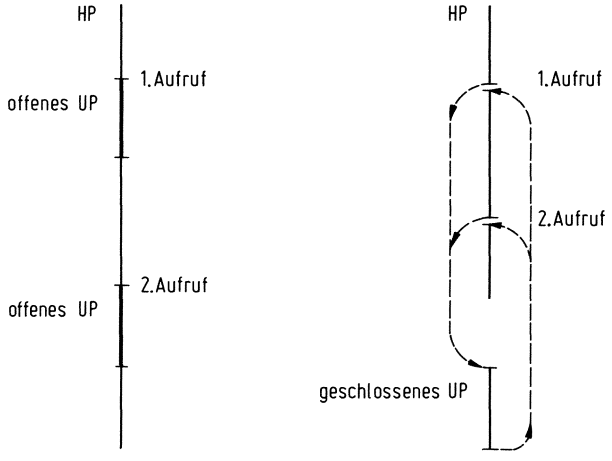


Bild 4.4 Illustration des Unterschieds zwischen offenen und geschlossenen Unterprogrammen

meterübergabe ist jedoch sehr unübersichtlich. Um diesen Nachteil zu beheben, geben wir im Hauptprogramm nur den Unterprogrammnamen und die Parameter an, während die organisatorischen Maßnahmen ins Unterprogramm gedrängt werden.

Einfache Assembler-Unterprogramme (Makrobefehle)

Der Assembler ist ein symbolverarbeitendes Systemprogramm, und der Gestaltung des Unterprogrammaufrufes sind keine engen Grenzen gesetzt. Um ihn übersichtlich zu machen, gleichen wir den Aufruf von Assembler-Unterprogrammen den Maschinenbefehlen soweit wie möglich an: Maschinenbefehle bestehen aus dem Befehlscode und den beiden Operandenadressen; Unterprogrammaufrufe bestehen aus dem Unterprogrammnamen und im allgemeinen mehreren Operandenadressen. Insofern stellen Unterprogrammaufrufe eine Verallgemeinerung von Befehlsaufrufen dar. Wegen der Übereinstimmung in der Schreibweise von Befehlen und Assembler-Unterprogrammen werden diese auch als Makrobefehle (kurz Makros) bezeichnet.

Makroaufruf: NAME adr1 adr2 adr3 ...

Von der Definition des Makrobefehls hängt es ab, ob die Parameter als Adressen oder als andere Größen, z.B. Operanden, aufgefaßt werden.

Makrodefinition. Zur Definition von Makrobefehlen benutzen wir die beiden Anweisungen "makro" und "mend", die - ähnlich wie org und end für ein Programm - den Anfang und das Ende für den Makrobefehl festlegen. Im Namensfeld der makro-Anweisung steht der Name des Makrobefehls und im Adreßfeld die formalen Parameter, die als Adressen in den Befehlen der Makrodefinition wieder erscheinen und nur innerhalb

des durch "makro" und "mend" begrenzten Bereichs gültig sind. Die Art der aktuellen Parameter (z.B. Adressen, Operanden) muß mit der Art der formalen Parameter übereinstimmen.

Parameterbehandlung. Während der Assemblierung prüft der Assembler den Befehlscode, ob er als Makrobefehl definiert ist. In diesem Falle setzt er anstelle des Aufrufes den zwischen "makro" und "mend" stehenden Text in die Befehlsfolge des Hauptprogrammes ein und ersetzt dabei die formalen durch die aktuellen Parameter, die von Aufruf zu Aufruf verschieden sein können. Das Ersetzen der formalen durch die aktuellen Parameter entspricht der mit set-Anweisungen in Beispiel 4.5 explizit durchgeführten indirekten Adressierung. Diese Art der Parameterübernahme wird als "call by name" bezeichnet. Dabei werden zuerst die formalen symbolischen Adressen durch die aktuellen symbolischen Adressen und dann durch numerische Adressen ersetzt.[1] Es handelt sich also hierbei um eine reine Substitution von Symbolfolgen, d.h. von Namen. Die Parameterübernahme "call by name" entspricht der Parameterübernahme "call by value", wenn ein Parameter nicht einen Namen, sondern einen Wert darstellt. Zur optimalen Erzeugung von Befehlscode im Hinblick auf Speicherplatz lassen sich Makrobefehle mit einer Parameterübernahme "call by value" besonders gut verwenden. Die folgenden Beispiele dienen dem Verständnis dieser Zusammenhänge, und es empfiehlt sich, die Assemblierung der Programme selbst durchzuspielen.

Beispiel 4.6 Vektoraddition (call by name)

Wir schreiben das in Beispiel 4.5 angegebene Additionsprogramm als Assembler-Unterprogramm und rufen es in einem Hauptprogramm zweimal mit zum Teil unterschiedlichen Parametern auf. Soll ein solches Unterprogramm verschiedenen Benutzern zugänglich gemacht werden, z.B. als Unterprogramm einer Programmbibliothek, so muß seine Wirkung genau beschrieben werden. Dabei müssen auch eventuell auftretende Nebeneffekte angegeben werden, wie im Beispiel die Benutzung des Index-Registers I, das nach Programmausführung durch den Prozessor den aktuellen Wert des ersten Parameters enthält.

---

[1] Im Gegensatz zur Parameterübernahme "call by value", wobei bereits vor dem Aufruf die symbolischen Adressen durch numerische Adressen ersetzt und erst dann die formalen symbolischen Adressen durch die aktuellen numerischen Adressen ersetzt werden.

```
                org                                  Haupt- und Unterprogramm
                 :
Makrobefehl
 VADD           makro   P1      P2      P3     Definition des Makros VADD
                LD      I       '0
 L              ADD     P2,I    P3,I
                ADD     I       '1
                CMP     I       P1
                LD      P       'L
                mend
                 :
 N              res     1                      Anzahl der Elemente von X, Y, Z
 X              res     100                    Vektor X
 Y              res     100                    Vektor Y
 Z              res     100                    Vektor Z
                 :
Hauptprogramm
 START           :                              erster Befehl des Hauptprogramms
                 :
                VADD    N       X       Y      erster Aufruf des Makros VADD;
                 :                             der zwischen makro und mend stehende
                 :                             Text wird vom Assembler eingesetzt, wobei
                 :                             P1,P2,P3 durch N,X,Y ersetzt werden.
                VADD    N       X       Z      zweiter Aufruf des Makros VADD;
                 :                             der zwischen makro und mend stehende
                 :                             Text wird vom Assembler eingesetzt, wobei
                 :                             P1,P2,P3 durch N,X,Z ersetzt werden.
                end     START                                                          ■
```

Beispiel 4.7 Vektoraddition (Feldlänge call by value)

Wir schreiben einen Makrobefehl zur Vektoraddition, bei dem die Länge der Felder (Parameter P1) bereits zur Assemblierzeit bekannt ist. Je nachdem, welchen Wert P1 hat, kann der Assembler entsprechend dem aktuellen Wert von P1 verschiedene Befehlsfolgen erzeugen, so daß der Speicherplatzbedarf minimal wird. Man sieht, wie die beiden Programme aus Beispiel 4.3 in einem Makrobefehl vereinigt sind. Beim Aufruf VADD 1 X Y wird beispielsweise nur der Code für den Befehl ADD X Y erzeugt. Beim Aufruf von VADD 5 X Y wird der Code für fünf Additionsbefehle mit

```
VADD    makro   P1      P2      P3     Makrodefinition von VADD
        if      P1>5
        goto    L
I       set     0
LA      ADD     P2+I    P3+I           P1-malige Codegenerierung für P1≤5
I       set     I+1
        if      I≠P1
        goto    LA
        goto    FIN
L       LD      I       '0             einmalige Codegenerierung für P1>5
LP      ADD     P2,I    P3,I
        ADD     I       '1
        CMP     I       'P1
        LD      P       'LP
FIN     mend                                                                          ■
```

aufeinanderfolgenden Adressen und beim Aufruf von VADD P1 X Y mit P1 > 5 wird der Code eines aus fünf Befehlen bestehenden Schleifenprogramms erzeugt. ■

Beispiel 4.8 Matrixaddition

Wir benutzen den in Beispiel 4.7 definierten Makrobefehl, um zwei Matrizen A und B von der Dimension 10 mal 10 zeilenweise zu addieren. Dabei wird die nach dem Prinzip "call by name" vorgenommene Ersetzung der in den Befehlen auftretenden formalen Parameter besonders deutlich. Der Assembler ersetzt den Aufruf durch das in Beispiel 4.7 definierte Programmstück, wobei die in den Adreßfeldern der

```
        org                                  Hauptprogramm
         :
A       res     10*10                        Matrix A
B       res     10*10                        Matrix B
         :
START    :
         :
        LD      J      '0
L       VADD    10     A,J     B,J           Makroaufruf von VADD (s. Bsp. 4.7)
        ADD     J      '10
        CMP     J      '10*10
        LD      P      'L
         :
        end     START
```

Makrodefinition stehenden Symbole durch die entsprechenden Symbolfolgen des Makroaufrufs ersetzt werden. Das folgende Programmstück zeigt die Befehle zur Matrixaddition nach der Makroersetzungsphase.

```
         :
        LD      J      '0
L       LD      I      '0      Makrobefehl VADD in der vom Assembler
        L ADD   A,J,I  B,J,I   erzeugten Form
        ADD     I      '1
        CMP     I      '10
        LD      P      'L
        ADD     J      '10
        CMP     J      '10*10
        LD      P      'L
         :
```

■

Einfache Prozessor-Unterprogramme (Prozeduren)

Der Prozessor ist hinsichtlich des Unterprogrammaufrufes an sein Befehlsformat gebunden: Maschinenbefehle bestehen aus dem Befehlscode und den beiden Operandenadressen; Unterprogrammaufrufe bestehen aus dem Unterprogrammnamen und im allgemeinen mehreren Operandenadressen. Der gleiche übersichtliche Aufruf wie bei

Assembler-Unterprogrammen ist bei Prozessor-Unterprogrammen nicht möglich. Trotzdem soll die am Anfang dieses Abschnittes aufgestellte Forderung nach Übersichtlichkeit soweit wie möglich erfüllt werden. Dazu schreiben wir nicht wie in Beispiel 4.5 unmittelbar vor dem Aufruf im Hauptprogramm eine Reihe von Befehlen, durch die die aktuellen Adressen in die formal definierten Speicherplätze des Unterprogramms transportiert werden, sondern tragen die aktuellen Adressen lediglich in unmittelbar nach dem Aufruf folgende Speicherzellen ein. Der Aufruf des Unterprogramms erfolgt durch einen Sprung zum ersten Befehl des Unterprogramms, der den Namen des Unterprogramms trägt. Damit die Programmsteuerung nach Ablauf des Unterprogramms wieder dem Hauptprogramm übergeben werden kann, muß beim Aufruf der Befehlszählerstand im Register- oder im Primärspeicher zur Ermittlung der Rücksprungadresse zwischengespeichert werden. Unterprogramme dieser Art werden zur Unterscheidung von Makrobefehlen als Prozeduren bezeichnet.

```
Prozeduraufruf:  LD   R      '1,P
                 LD   P      'NAME
                 dat  adr1
                 dat  adr2
                 dat  adr3
```

Von der Definition der Prozedur hängt es ab, ob die Parameter als Adressen oder als andere Größen, z.B. Operanden, aufgefaßt werden.

Prozedurdefinition. Zur Definition einer Prozedur benutzen wir die beiden Anweisungen "org" und "end", die den Anfang und das Ende der Prozedur für den Assembler festlegen, sofern die Prozedur getrennt vom Hauptprogramm assembliert wird. Für den Prozessor ist der Anfang der Prozedur durch den mit dem Namen der Prozedur gekennzeichneten ersten Maschinenbefehl und das Ende der Prozedur durch einen Sprung ins Hauptprogramm festgelegt. Die Rücksprungadresse errechnet sich aus der in der Zelle R gespeicherten ersten Parameteradresse plus der Länge des Parameterfeldes. Dort steht derjenige Befehl des Hauptprogramms, der nach der Ausführung der Prozedur als nächster Befehl ausgeführt werden soll.

Parameterbehandlung. Je nachdem ob die Adresse R sich auf den Registerspeicher oder den Primärspeicher bezieht, muß man unterschiedliche Adreßmodifikationen benutzen, um an die Operanden heranzukommen. Die in der Prozedur verwendeten formalen Adressen der Operanden müssen zur Ausführungszeit vom Prozessor durch die im Aufruf angegebenen aktuellen Adressen ersetzt werden, die von Aufruf zu Aufruf verschieden sein können. Dieses Ersetzen von indirekten durch direkte Adressen wird mit der Basisadressierung durchgeführt, wenn die Adresse des ersten Parameters im Registerspeicher steht, und es wird mit Hilfe der indirekten Adressierung durchgeführt, wenn die Adresse des ersten Parameters im Primärspeicher steht. Im allgemeinen werden die auf den Aufruf folgenden Parameter in die Prozedur übernommen. Dazu werden in der Prozedur Speicherzellen reserviert, deren Adressen in den

Befehlen der Prozedur indirekt verwendet werden. Diese Art der Parameterübernahme wird als "call by reference" bezeichnet. Dabei werden die Adressen der Operanden als Parameter in die Prozedur übernommen und indirekt adressiert. Bei der Parameterübernahme "call by value" hingegen werden die Werte der Operanden als Parameter in die Prozedur übernommen und direkt adressiert. Dabei beansprucht zwar die Parameterübernahme mehr Zeit, aber die Ausführungszeit der Prozedur wird unter Umständen trotzdem geringer. Die folgenden Beispiele dienen dem Verständnis dieser Zusammenhänge, und es empfiehlt sich, die Ausführung der Programme selbst durchzuspielen.

Beispiel 4.9 Vektoraddition (call by reference)

```
                org                          Haupt- und Unterprogramm
                 :
Prozedur
  P0            res   1                      Platzreservierung für die Rücksprungadresse
  P1            res   1                      und die Parameter (Parameterauffangfeld)
  P2            res   1
  P3            res   1
  VADD          LD    P1        0,R          Übernahme der Parameter nach "call by reference"
                LD    P2        1,R
                LD    P3        2,R
                LD    P0        '3,R         Übernahme der Rücksprungadresse
                LD    I         '0           "eigentliche" Prozedur mit indirekter
  L             ADD   *P2,,I    *P3,,I       Adressierung der formalen Parameter P1 bis P3
                ADD   I         '1
                CMP   I         *P1
                LD    P         'L
                LD    P         P0           Rücksprung
                 :
  N             res   1                      Anzahl der Elemente von X, Y, Z
  X             res   100                    Vektor X
  Y             res   100                    Vektor Y
  Z             res   100                    Vektor Z
                 :
Hauptprogramm
  START          :                           erster Befehl des Hauptprogramms
                 :
                LD    R         '1,P         erster Aufruf der Prozedur VADD; der Prozessor
                LD    P         'VADD        fährt mit der Ausführung bei VADD fort
                dat   N                      Parameterversorgungsfeld
                dat   X
                dat   Y
                 :
                LD    R         '1,P         zweiter Aufruf der Prozedur VADD; der Prozessor
                LD    P         'VADD        fährt mit der Ausführung bei VADD fort
                dat   N                      Parameterversorgungsfeld
                dat   X
                dat   Z
                 :
                end   START
```

Wir schreiben das in Beispiel 4.5 angegebene Additionsprogramm als Prozessor-Unterprogramm und rufen es in einem Hauptprogramm zweimal mit unterschiedlichen Parametern auf. Die Adresse des Parameterfeldes wird im Register **R** gespeichert. Die Parameter werden vom Hauptprogramm in einen Bereich des Unterprogramms transportiert, der für die Befehle des Programms unmittelbar zugänglich ist. Die als Feld zusammengefaßten Parameter werden im Hauptprogramm oft als Parameterversorgungsfeld und im Unterprogramm als Parameterauffangfeld bezeichnet. Dementsprechend erscheinen im Versorgungsfeld die aktuellen Parameter, die von Aufruf zu Aufruf verschieden sein können. Die Adressen des Auffangfeldes sind die Adressen der formalen Parameter. Die zum Transport der Parameter notwendigen Befehle und eventuell weitere Organisationsbefehle stehen im Unterprogramm. Dadurch wird der (öfter vorkommende) Aufruf kurz und das Hauptprogramm übersichtlicher. ■

Beispiel 4.10 Vektoraddition (Feldlänge call by value)

Wir modifizieren die in Beispiel 4.9 angegebene Prozedur, so daß einer der Eingangsparameter durch "call by value" übernommen wird. Der CMP-Befehl braucht dann nicht mehr mit indirekter Adresse geschrieben zu werden, wodurch sich die Ausführungszeit des Programms verkürzt. Der Transport und damit das doppelte Speichern der anderen Parameteradressen kann entfallen, wenn Vor- und Nachindizierung kombiniert zur Adreßmodifizierung benutzt werden. Die Prozedur benötigt dadurch 6 Speicherplätze weniger, und es werden 3 Befehle weniger ausgeführt.

```
        org                                 Prozedurdefinition von VADD
        ent   VADD
P1      res   1                             Reservierung für den Wert des 1. Parameters
VADD    LD    P1        *0,R                Übernahme des 1. Parameters "call by value"
        LD    I         '0                  "eigentliche" Prozedur
L       ADD   *1,R,I    *2,R,I
        ADD   I         '1
        CMP   I         P1
        LD    P         'L
        LD    P         '3,R
        end
```

Die Prozedur kann unabhängig vom Hauptprogramm assembliert werden. Sie ist mit einer end-Anweisung abgeschlossen und somit ein vollständiges, in sich abgeschlossenes Programm. Auf diese Weise können große Programme aus einzelnen assemblierten Programmoduln zusammengesetzt werden. Die im Hauptprogramm im Adreßfeld der Befehle benutzten und in den Unterprogrammen im Namensfeld auftretenden Adressen müssen für das Binden durch ent-Anweisungen gekennzeichnet werden. ■

Beispiel 4.11 Matrixaddition

Wir benutzen die in Beispiel 4.10 definierte Prozedur, um zwei Matrizen A und B von der Dimension 10 mal 10 zeilenweise zu addieren. Dabei wird die kombinierte Adreßmodifizierung auch auf die Parameteradressen ausgedehnt.[1]

```
        org                          Hauptprogramm
        ext   VADD
        :
A       res   10*10                  Matrix A
B       res   10*10                  Matrix B
        :
START   :
        :
        LD    J       '0
L       LD    R       '1,P           Prozeduraufruf von VADD (s. Bsp. 4.10)
        LD    P       'VADD
        dat   '10                    Konstante
        dat   A,J                    indizierte Variable
        dat   B,J                    indizierte Variable
        ADD   J       '10
        CMP   J       '10*10
        LD    P       'L
        :
        end   START
```

Wird in der Prozedur VADD entsprechend Beispiel 4.10 die Adresse *1,**R**,**I** ausgewertet, so hat sie nach der indirekten Adressierung den Wert (1+(**R**)), was der Adresse A,**J** entspricht. Zum Wert von A wird anschließend der Inhalt des Indexregisters **I** addiert, was der Adresse A+(**I**), **J** entspricht. Schließlich wird die Adresse A+(**I**),**J** ausgewertet, wodurch sich ihr Wert zu A+(**I**)+(**J**) ergibt, was ei-

```
        :
        LD    J       '0
L       LD    I       '0       äquivalentes Programm entsprechend der
 L      ADD   A,I,J   B,I,J    Prozedur VADD
        ADD   I       '1
        CMP   I       '10
        LD    P       'L
        ADD   J       '10
        CMP   J       '10*10
        LD    P       'L
        :
```

[1] Ohne das Unterprogramm VADD zu benutzen, hätte wegen der eindimensionalen Speicherung der Matrizenelemente das Programm wie VADD auch mit einer einzigen Schleife programmiert werden können.

ner gedachten Adresse A,I,J entspricht. Mit dieser Adresse wird der Operand geholt (vgl. Bild 4.2). Das Programm ist in seiner Wirkung dem auf S. 154 unten dargestellten Programm äquivalent. ■

Statische und dynamische Speicherverwaltung

Der Datenbereich einer Prozedur besteht aus den Speicherzellen für die Parameter, für die Rücksprungadresse und für die lokalen Größen der Prozedur. Wird dieser Bereich als ein Feld aufgefaßt, dann ist es möglich, alle diese Objekte relativ zu der Anfangsadresse des Feldes zu adressieren. In unseren Beispielen hatten wir diese Plätze durch den Assembler reservieren lassen, und es war deshalb zur Ausführungszeit nicht möglich, diesen Bereich auszudehnen. Man spricht von statischer Speicherverwaltung.

Ein gut strukturiertes Programm mit vielen Prozeduren benötigt bei statischer Speicherverwaltung für alle Prozeduren eigene Datenbereiche, die zusammen mit dem Datenbereich des Hauptprogramms den Speicher füllen. Vom Prozessor wird aber zu jedem Zeitpunkt nur das Hauptprogramm oder eine dieser Prozeduren ausgeführt, so daß entweder nur mit den Daten des Hauptprogramms oder mit den Daten des Hauptprogramms und einer dieser Prozeduren gearbeitet wird. In der Ausführungsphase des Programms werden also die Daten des Hauptprogramms fortwährend und die Daten der gerade ausgeführten Prozedur nur vorübergehend benötigt. Wird nun dieser zusätzlich zum Hauptprogramm benötigte Speicherbereich der Prozedur erst zur Ausführungszeit beim Prozeduraufruf reserviert und beim Verlassen der Prozedur wieder freigegeben, so pulsiert die Speicherplatzbelegung, und man spricht von dynamischer Speicherverwaltung.

Die dynamische Speicherreservierung erfolgt im Gegensatz zur statischen Speicherreservierung nicht bei der Assemblierung, sondern erst bei der Ausführung des Programms. Dazu ist es nötig, die Prozedurdaten vom Prozedurcode zu trennen. Die Adresse des ersten freien Speicherplatzes muß bei der Prozedur zur Ausführungszeit bekannt sein. Sie wird in ein Register gebracht, so daß die von der Prozedur benutzten Größen relativ zu dieser als Datenbasis aufgefaßten Adresse adressiert werden können. Bei der Ausführung der Prozedur muß die Adresse des ersten freien Speicherplatzes um die in der Prozedur reservierten Speicherplätze erhöht werden. Wenn die Basis zwischengespeichert wird, kann innerhalb dieser Prozedur in gleicher Weise wiederum eine Prozedur aufgerufen werden. Beim Verlassen der Prozedur ist der vor dem Prozeduraufruf vorgefundene Zustand wiederherzustellen. Die Organisation des geschachtelten Aufrufes von Prozeduren, d.h. des Aufrufes von Prozeduren innerhalb von Prozeduren wird im folgenden Abschnitt genauer behandelt.

Beispiel 4.12 Vektoraddition

In Beispiel 4.9 besteht der Datenbereich der Prozedur VADD aus den reservierten Plätzen für die Rücksprungadresse und die drei Parameter. Wir reservieren einen weiteren Platz für den Inhalt des Indexregisters, so daß der im Hauptprogramm gültige Registerinhalt nach der Ausführung der Prozedur wieder vorgefunden wird. Dies entspricht der Speicherreservierung für eine lokale Größe der Prozedur.

Platzreservierung durch den Assembler

```
        org   1000
        :
N       res   1
X       res   100
Y       res   100
        :
START   :
        :
        LD    R       '1,P
        LD    P       'VADD
        dat   N
        dat   X
        dat   Y
        :
B       res   5
VADD    LD    0+B     '3,R
        LD    1+B     0,R
        LD    2+B     1,R
        LD    3+B     2,R
        LD    4+B     I
        LD    I       '0
L       ADD   *2+B,,I *3+B,,I
        ADD   I       '1
        CMP   I       *1+B
        LD    P       'L
        LD    I       4+B
        LD    P       0+B
        end   START
```

Platzreservierung durch den Prozessor

```
        org   1000
        :
N       res   1
X       res   100
Y       res   100
        :
START   LD    B       '2000
        LD    F       '2000
        :
        LD    R       '1,P
        LD    P       'VADD
        dat   N
        dat   X
        dat   Y
        :
VADD    ADD   F       '5
        LD    0,B     '3,R
        LD    1,B     0,R
        LD    2,B     1,R
        LD    3,B     2,R
        LD    4,B     I
        LD    I       '0
L       ADD   *2,B,I  *3,B,I
        ADD   I       '1
        CMP   I       *1,B
        LD    P       'L
        LD    I       4,B
        LD    F       B
        LD    P       0,B
        end   START
```

Im linken Programm wird der Speicher statisch verwaltet. Durch B res 5 werden vom Assembler 5 Speicherplätze reserviert, und dieser Bereich wird mit B als Basisadresse zur Assemblierzeit relativ adressiert. Im rechten Programm wird der Speicherbereich dynamisch verwaltet. Durch ADD **F** '5 werden vom Prozessor 5 Speicherplätze reserviert. Vor dem Prozeduraufruf ist in den beiden Registern **F** und **B** die Adresse der ersten Zelle des dynamisch verwalteten Speicher-

bereichs gespeichert, so daß der Datenbereich der Prozedur zur Ausführungszeit relativ zum Inhalt von **B** adressiert werden kann. Unmittelbar vor dem Verlassen der Prozedur wird die Datenbasisadresse der Prozedur wieder in das Register **F** geladen.[1] Der Inhalt von **B** zeigt also immer auf die Datenbasis der gerade aktiven Prozedur und der Inhalt von **F** auf den ersten freien Speicherplatz. ■

4.4 Geschachtelte und rekursive Unterprogramme

Wenn von Haupt- und Unterprogrammen die Rede ist, so legt das eine zweistufige Programmhierarchie nahe. Wir wollen im folgenden den absoluten Begriff Hauptprogramm durch den relativen Begriff Oberprogramm ersetzen, wenn vom Programm über dem Unterprogramm die Rede ist. Das Oberprogramm ist das aufrufende und das Unterprogramm ist das aufgerufene Programm. Das Oberprogramm kann Haupt- oder Unterprogramm sein, wodurch die Programmhierarchie von zwei auf mehrere Stufen erweitert wird. Es entsteht ein "Baum" ineinander geschalteter Unterprogramme mit dem Hauptprogramm als "Wurzel". Jedes Unterprogramm, das seinerseits ein oder mehrere Unterprogramme aufruft, ist gleichzeitig Oberprogramm. Auf diese Weise läßt sich eine umfangreiche Aufgabe derart strukturieren, daß auf der obersten Ebene eine verständliche, in viele Unteraufgaben gegliederte und in sich geschlossene Beschreibung der Aufgabe vorliegt. Auf den unteren Ebenen sind die Unteraufgaben in gleicher Weise in sich geschlossen beschrieben und ihrerseits in weitere Unteraufgaben zerlegt. Je tiefer man in diesem Aufgaben-Stapel vordringt, desto detailreicher wird die Beschreibung. In der untersten Stufe ist die Beschreibung der Unteraufgaben genau auf die Werkzeuge zugeschnitten, mit denen sie ausgeführt werden. In der Programmierungstechnik wird dieses Vorgehen als Methode der schrittweisen Verfeinerung bezeichnet oder kurz schrittweise Programmentwicklung genannt. "Die allgemeingültigste Strategie der Programmierung ist wohl das Prinzip, einen Prozeß in einzelne Handlungen, und damit das Programm in einzelne Anweisungen, zu zerlegen. Bei jedem dieser Zerlegungsschritte muß überlegt und sichergestellt werden, daß

1. die Lösung der Teilaufgabe die Lösung der Gesamtaufgabe beinhaltet,
2. die festgelegte Folge von Teilhandlungen sinnvoll ist und
3. die gewählte Zerlegung Anweisungen ergibt, die dem verfügbaren Werkzeug (Programmsprache) angemessen sind."[2]

---

[1] Diese Organisation des dynamischen Speichers ähnelt der Organisation eines Kellerspeichers. Dort ersetzt man die Adressierung relativ zu der konstanten Basisadresse (**B**) durch die Adressierung relativ zu der variablen Basisadresse (**F**), wobei man negative Distanzen in Kauf nehmen muß. Das Register **B** wird nicht benötigt, und das Register **F** wird als Zähler ausgebildet. Der dynamische Speicher wird als Stack bezeichnet, **F** enthält den Stackpointer, und zwei Befehle (PUSH und POP) füllen und leeren den Stack.

[2] Wirth, N.: Systematisches Programmieren. Stuttgart: Teubner 1972, S. 120.

In jeder Stufe ist jedes Unterprogramm ein in sich abgeschlossenes Programm mit Eingangs- und Ausgangsparametern, so daß es unabhängig von den anderen geschrieben und verifiziert werden kann. Dieser modulare Programmaufbau ermöglicht es, Unterprogramme von verschiedenen Programmierern getrennt voneinander codieren und testen zu lassen. Alle Unterprogramme werden hernach schrittweise zusammengesetzt, so daß in jeder Ebene das Problem mit einem unterschiedlichen Grad an Detailinformation vollständig beschrieben ist.

Bei geschachtelten Unterprogrammen ist es denkbar, daß sich ein Unterprogramm mittelbar oder unmittelbar selbst aufruft. Unterprogramme, die dies von ihrer Organisation her erlauben, nennt man rekursive Unterprogramme. Rekursive Algorithmen kommen vor allem in einigen Zweigen der nichtnumerischen Datenverarbeitung vor (z.B. beim Programmieren von Spielen). In unserem Zusammenhang sind rekursive Prozeduren deshalb von Interesse, da sie eine Unterklasse der reentranten Prozeduren sind, die in Abschnitt 6.4 behandelt werden.

Makroaufruf in einer Makrodefinition

Der geschachtelte Aufruf von Assembler-Unterprogrammen ist unproblematisch, weil die Adreßersetzungen bei der Parameterübernahme statisch, d.h. bereits zur Assemblierzeit des Programmes, vorgenommen werden. Der Aufruf eines Makrobefehls innerhalb eines Unterprogrammes unterscheidet sich prinzipiell nicht vom Aufruf eines Makrobefehles innerhalb eines Hauptprogrammes. In der Reihenfolge, in der die Makrobefehle vom Assembler verarbeitet werden, wird Maschinencode erzeugt, und die formalen werden durch die aktuellen Parameter ersetzt. Lokale Datenbereiche werden innerhalb der Makrobefehle vorgesehen. Die Speicherverwaltung ist statisch. Die Schachtelung von Makrobefehlen kann ganz außerordentlich zu einer übersichtlichen Programmierung beitragen, was allerdings auf Kosten der Effizienz der Programme gehen kann.

Beispiel 4.13 Matrixaddition als Makrobefehl

Wir schreiben einen Makrobefehl zur Addition von zwei gleich großen quadratischen Matrizen A und B von der Dimension 10 mal 10. Wie in Beispiel 4.8 wird die Addition der Matrizen auf die Addition korrespondierender Zeilenvektoren zurückgeführt. Man vergleiche dazu den Makroaufruf von VADD mit den Befehlsaufrufen von ADD und CMP hinsichtlich der Parameter bzw. Adressen. Der Makroassembler ersetzt den Aufruf von MADD durch den auf S. 159 unten angegebenen Code, wobei der Übersichtlichkeit halber die symbolische Darstellung der Befehle und Anweisungen beibehalten worden ist.

```
        org                                  Hauptprogramm
        ⋮
A       res     10*10                        Matrix A
B       res     10*10                        Matrix B
        ⋮
START   ⋮
        ⋮
        MADD    '10     A       B            Makroaufruf von MADD
        ⋮
MADD    makro   P1      P2      P3           Makrodefinition für MADD;
        LD      P       'NQ+1                P1 Rang, P2 und P3 Matrizen
NQ      res     1
        LD      J       '0
        LD      NQ      P1
        MUL     NQ      NQ
L       VADD    P1      P2,J    P3,J         Makroaufruf von VADD
        ADD     J       P1
        CMP     J       NQ
        LD      P       'L
        mend
        ⋮
VADD    makro   P1      P2      P3           Makrodefinition für VADD;
        LD      I       '0                   P1 Rang, P2 und P3 Vektoren
L       ADD     P2,I    P3,I                 Befehlsaufruf von ADD
        ADD     I       '1
        CMP     I       P1                   Befehlsaufruf von CMP
        LD      P       'L
        mend
        end     START
```

```
⋮
┌──────────────────────────────────────────────────────┐
│        LD      P       'NQ+1   Code des Makros MADD  │
│NQ      res     1                                     │
│        LD      J       '0                            │
│        LD      NQ      '10                           │
│        MUL     NQ      NQ                            │
│L  ┌────────────────────────────────────────────────┐│
│   │    LD      I       '0      Code des Makros VADD│
│   │L   ADD     A,J,I   B,J,I                       │
│   │    ADD     I       '1                          │
│   │    CMP     I       '10                         │
│   │    LD      P       'L                          │
│   └────────────────────────────────────────────────┘│
│        ADD     J       '10                           │
│        CMP     J       NQ                            │
│        LD      P       'L                            │
└──────────────────────────────────────────────────────┘
⋮
```

■

Der rekursive Aufruf von Assembler-Unterprogrammen ist ebenfalls unproblematisch, wenn strikt an der beschränkten Gültigkeit der Namen festgehalten wird. Bei jedem Auftreten des Makrobefehles muß vom Assembler ein Exemplar Maschinencode einschließlich der lokalen Datenbereiche entsprechend der Definition des Makrobefehles erzeugt werden. Dieser Prozeß immer wiederkehrender Schachtelung ein und desselben Makrobefehles muß irgendwann sein Ende finden. Dazu wird der rekursive Aufruf übersprungen, und der Assembler gelangt an die mend-Anweisung des rekursiven Makrobefehles.

Beispiel 4.14 Fakultät

Wir benutzen zur Demonstration des rekursiven Aufrufes eines Makrobefehles die Berechnung der Fakultät von n nach den Formeln

$$n! = 1 \quad \text{für} \quad n = 1$$
$$n! = n \cdot (n-1)! \quad \text{für} \quad n > 1$$

Da der Assembler in Abhängigkeit vom Argument mehrere Exemplare des Makrobefehls erzeugen soll, muß der Wert dieses Parameters zur Assemblierzeit bekannt sein. Wir schreiben einen Makrobefehl FAK P1 P2 mit folgender Wirkung: Die Fakultät des Wertes von P2 wird in die Zelle mit der Adresse P1 geschrieben.

```
        org                     Hauptprogramm
         :
Y       res     1
         :
START    :
         :
        FAK     Y       4       Makroaufruf von FAK zur Berechnung von 4!
         :
FAK     makro   P1      P2      Makrodefinition von FAK
        LD      P1      'P2
        if      P2=1
        goto    FIN
        LD      P       'Z+1
Z       res     1
        LD      Z       'P2
P2      set     P2-1
        FAK     P1      P2      Wiederaufruf von FAK
        MUL     P1      Z
FIN     mend
        end     START
```

Nach der Ersetzung des Makrobefehles durch Maschinenbefehle tritt die Blockstruktur des Programmes besonders deutlich hervor. Die im Namensfeld angegebenen Symbole sind nur innerhalb des jeweils betrachteten Blockes gültig. Im vollständig

assemblierten Maschinenprogramm drückt sich das dadurch aus, daß für ein und dieselbe symbolische Adresse unterschiedliche numerische Adressen stehen.

```
        :
Y       res     1
        :
START   :
        :
        LD      Y     '4      FAK Y 4 entspr. 1. Aufruf
        LD      P     'Z+1
Z       res     1
        LD      Z     '4
        LD      Y     '3      FAK Y 3 entspr. 2. Aufruf
        LD      P     'Z+1
Z       res     1
        LD      Z     '3
        LD      Y     '2      FAK Y 2 entspr. 3. Aufruf
        LD      P     'Z+1
Z       res     1
        LD      Z     '2
        LD      Y     '1      FAK Y 1 entspr. 4. Aufruf
        MUL     Y     Z
        MUL     Y     Z
        MUL     Y     Z
        :
```

■

Prozeduraufruf in einer Prozedurdefinition

Der geschachtelte Aufruf von Prozessor-Unterprogrammen ist problematischer, weil die Adreßersetzungen bei der Parameterübernahme dynamisch, d.h. zur Ausführungszeit des Programmes, durchgeführt werden. Der Aufruf einer Prozedur innerhalb eines Unterprogrammes unterscheidet sich nur dann nicht vom Aufruf einer Prozedur innerhalb eines Hauptprogrammes, wenn bei der Parameterübernahme alle in den dat-Anweisungen angegebenen Adreßmodifizierungen ausgeführt werden.

Der rekursive Aufruf von Prozessor-Unterprogrammen ist ebenfalls problematischer, weil die Speicherreservierungen des Unterprogrammes dynamisch, d.h. zur Programmausführungszeit, durchgeführt werden. Bei jeder Ausführung der Prozedur muß vom Prozessor ein neuer Datenbereich eröffnet werden, so daß der belegte Speicherbereich ständig wächst. Dieser Prozeß des immer wiederkehrenden Aufrufes ein und derselben Prozedur muß irgendwann sein Ende finden. Dazu wird der rekursive Aufruf übersprungen und der Prozessor gelangt an den Rücksprungbefehl der rekursiven Prozedur.

Die Programmierung geschachtelter und rekursiver Prozeduren wird durch die Befehle zur Parameterübernahme und zur Organisation des dynamischen Speichers recht unübersichtlich. Um die Organisationsbefehle so weit wie möglich in den Hintergrund zu drängen, erweitern wir die Befehle unserer Assemblersprache durch Makrobefehle,

die wie Maschinenbefehle benutzt werden können. Wir definieren so eine niedere Programmiersprache, die aus den durch die Hardware bestimmten Maschinenbefehlen und den durch die Software bestimmten Systemmakros besteht. In unserem Zusammenhang handelt es sich um Makros für den Aufruf einer Prozedur (CALL) einschließlich der Parameterübertragung in den dynamischen Speicher, für das Verlassen einer Prozedur (RETURN), für das Definieren des Ursprungs des dynamischen Speichers (ORG), für das dynamische Reservieren von Speicherzellen (RES) und für den unbedingten Sprung (GOTO). Wir listen diese Makros zunächst auf[1] und geben nachfolgend zwei Beispiele, die ihre Wirkung verstehen helfen und ihre Anwendungsmöglichkeiten zeigen sollen.

Makrobefehl CALL. Wir gehen einen Schritt rückwärts, um zwei Schritte vorwärts machen zu können. Die Befehle zur Reservierung der Speicherplätze für die Rücksprungadresse, die Datenbasisadresse und die Parameter sowie die Befehle zur Speicherung bzw. Übernahme dieser Größen in den dynamischen Speicher verlegen wir aus dem Unter- ins Oberprogramm und fassen sie zum Systemmakro CALL zusammen.

Makroaufruf: CALL P1 P2
P1 : Name der Prozedur
P2 : Anzahl der aktuellen Parameter

CALL dient zum Aufrufen einer Prozedur. Der erste Parameter von CALL trägt den Namen der Prozedur und gibt die Adresse des ersten Befehls der Prozedur an. Der zweite Parameter gibt die Anzahl der in die Prozedur zu übertragenden Prozedurparameter an, die in den auf CALL folgenden Zeilen aufgeführt sind. Die Prozedurparameter werden in den dynamischen Speicherbereich der Prozedur entsprechend "call by reference" übergeben, wobei evtl. auftretende Adreßmodifizierungen mit Ausnahme von Direktoperanden ausgeführt werden (vgl. Bild 4.3). Sie sind in der durch die Datenbasisadresse der Prozedur adressierten Zelle und den darauf folgenden Zellen gespeichert. In der ersten und der zweiten Zelle vor der durch die Datenbasisadresse adressierten Zelle sind die Datenbasisadresse und die Rücksprungadresse für das Oberprogramm gespeichert. Die Datenbasisadresse der Prozedur steht im Register **B**, und die Adresse der ersten freien Zelle des dynamischen Speichers steht im Register **F**.

---

[1] Wer sich um die Details der Makrodefinitionen nicht kümmern will, fasse die Makrobefehle als Maschinenbefehle auf.

Makrodefinition:

```
CALL   makro   P1         P2          Prozeduraufruf
       ext     P1                    Prozedurname extern deklariert
D      equ     P2+4
       ADD     F          'P2+2      zum Reservieren von P2+2 Zellen
       LD      -D,F       '2*P2+3,P  zum Speichern der Rücksprung- und
       LD      -D+1,F     B          Datenbasisadresse
I      set     0                     zur Parameterübergabe
L      LD      -D+2+I,F   '*P2+1,P   entsprechend "call by reference"
I      set     I+1
       if      I≠P2
       goto    L
       LD      B          '-D+2,F    zum Laden der Register B (Datenbasis der
       LD      P          'P1        Prozedur) und P (Sprung in die Prozedur)
       mend
```

Makrobefehl RETURN. Wir fassen die Befehle zum Freigeben der durch eine Prozedur belegten Speicherplätze, zum Laden der Datenbasisadresse des Oberprogramms und zum Rücksprung ins Oberprogramm beim Verlassen einer Prozedur zum Systemmakro RETURN zusammen.

Makroaufruf: RETURN

RETURN dient zum Rücksprung aus einer Prozedur ins Oberprogramm. Er stellt den vor dem Prozeduraufruf gültigen Inhalt der Register **F**,**B** und **P** wieder her und gibt den in der Prozedur reservierten dynamischen Speicherbereich frei.

Makrodefinition:

```
RETURN makro          Prozedurrückkehr
       LD     F B     zum Freigeben des dynamisch reservierten
       LD     B -1,F  Speicherplatzes und zum Rücksprung
       LD     P -2,F  ins Oberprogramm
       mend
```

Makrobefehl ORG. Wir fassen die Befehle zum Laden der Datenbasisadresse und der Adresse der ersten Zelle des dynamischen Speichers zum Systemmakro ORG zusammen.

Makroaufruf: ORG P1

P1 : Basisadresse des dynamischen Speicherbereichs

ORG definiert den Ursprung des dynamischen Speicherbereichs zur Ausführungszeit. Der Parameter des Makrobefehls weist auf die erste Zelle des dynamischen Speicherbereichs. Die ersten beiden Plätze des durch ORG reservierten Speicherbereichs sind für die Rücksprung- und Datenbasisadresse des Hauptprogramms im Falle eines Prozeduraufrufs bestimmt. Dementsprechend ist die Adresse der ersten freien Zelle um 2 höher als die Datenbasisadresse.

Makrodefinition:

```
ORG     makro  P1          Ursprung des dynamischen Speichers
        LD     B    'P1
        LD     F    'P1+2
        mend
```

Makrobefehl RES. Anstelle des ADD-**F**-Befehls benutzen wir zur Verdeutlichung der dynamischen Reservierung von Speicherplatz den Systemmakro RES.

Makroaufruf: RES P1

P1 : Anzahl der dynamisch reservierten Speicherzellen

RES reserviert Speicherplätze zur Ausführungszeit. Der Parameter des Makrobefehls gibt an, wieviele Speicherplätze reserviert werden.

Makrodefinition:

```
RES     makro  P1         Reservierung im dynamischen Speicher
        ADD    F    'P1
        mend
```

Makrobefehl GOTO. Anstelle des LD-**P**-Befehls benutzen wir zur Verdeutlichung des Sprunges den Systemmakro GOTO.

Makroaufruf: GOTO P1

P1 : Adresse des nächsten Befehls

GOTO bewirkt ein Weiter an anderer Stelle im Programm. Der Parameter des Makrobefehls gibt an, welcher Befehl als nächster ausgeführt wird.

Makrodefinition:

```
GOTO    makro  P1              Programmsprung
        LD     P    'P1-$-1,P
        mend
```

Die folgenden beiden Beispiele für den Aufruf geschachtelter und rekursiver Prozeduren veranschaulichen Anwendung und Wirkung dieser Systemmakros und zeigen, wie wirkungsvoll und übersichtlich damit programmiert werden kann.

Beispiel 4.15 Matrixaddition als Prozedur

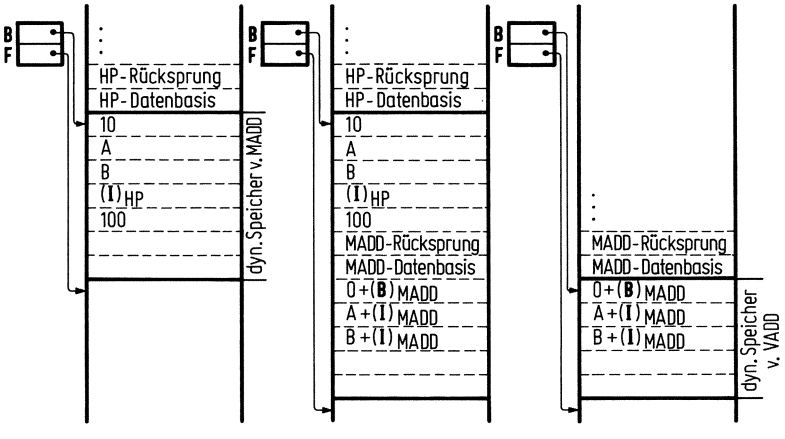
Wir schreiben eine Prozedur zur Addition von zwei gleich großen quadratischen Matrizen A und B von der Dimension 10 mal 10. Wie in Beispiel 4.11 wird die Addition der Matrizen auf die Addition korrespondierender Zeilenvektoren zurückgeführt. Die Daten des Hauptprogrammes sind statisch ab Zelle 1000 und die Daten der Prozeduren dynamisch ab Zelle 2000 gespeichert. Man vergleiche den Prozeduraufruf von VADD mit den Befehlsaufrufen von ADD und CMP hinsichtlich der Parameter bzw. Adressen.

```
        org     1000            Hauptprogramm
        :
A       res     10*10           Matrix A
B       res     10*10           Matrix B
N       dat     10              Dimension 10
        :
START   ORG     2000            Ursprung des dynamischen Speichers
        :
        CALL    MADD    3       Prozeduraufruf von MADD
        dat     N
        dat     A
        dat     B
        :
        end     START
```

```
        org                     Prozedurdefinition von MADD
        ent     MADD
MADD    LD      0,B     *0,B    "call by value" für den 1. Parameter
        RES     2               Reservierung für Index und Quadrat
        LD      3,B     I       Speichern des Indexregisters
        LD      4,B     0,B     Bilden des Quadrates des 1. Parameters
        MUL     4,B     4,B
        LD      I       '0
L       CALL    VADD    3       Prozeduraufruf von VADD
        dat     0,B
        dat     *1,B,I
        dat     *2,B,I
        ADD     I       0,B
        CMP     I       4,B
        GOTO    L
        LD      I       3,B     Laden des Indexregisters
        RETURN
        end
```

```
        org                     Prozedurdefinition von VADD
        ent     VADD
VADD    LD      0,B     *0,B    "call by value" für den 1. Parameter
        RES     1               Reservierung und
        LD      3,B     I       Speichern des Indexregisters
        LD      I       '0
L       ADD     *1,B,I  *2,B,I  Befehlsaufruf von ADD
        ADD     I       '1
        CMP     I       0,B     Befehlsaufruf von CMP
        GOTO    L
        LD      I       3,B     Laden des Indexregisters
        RETURN
        end
```

Die Wirkung des Prozeduraufrufs von CALL VADD wird durch die folgenden Bilder illustriert. Drei "Schnappschüsse" zeigen den dynamisch reservierten Speicherbereich vor dem Prozeduraufruf, nach der Parameterübergabe und vor dem Sprung zum ersten Befehl der Prozedur. Wie man sieht, umfaßt der durch **B** und **F** markierte Teil des dynamischen Speichers vor dem Prozeduraufruf den Datenbereich des Oberprogrammes (linkes Bild), während des Prozeduraufrufes den Datenbereich des Oberprogrammes und des Unterprogrammes (mittleres Bild) und nach dem Prozeduraufruf den Datenbereich des Unterprogrammes (rechtes Bild) jeweils einschließlich der beiden vorsorglich reservierten Zellen für die Rücksprung- und die Datenbasisadresse. Während des Aufrufes überlappen sich die Datenbereiche des Ober- und Unterprogrammes (mittleres Bild), so daß in dieser Zeit die Parameter mit der Datenbasisadresse des Oberprogrammes modifiziert werden können.



Dadurch, daß mit CALL die Parameterübergabe im Oberprogramm vorgenommen wird, ist es nicht wie in den früheren Beispielen nötig, die Anfangsadresse des Parameterfeldes in einem Register zwischenzuspeichern. Die assemblierte Aufruffolge ähnelt dem in Beispiel 4.5 angedeuteten Prozeduraufruf. Zum besseren Verständnis der Wirkung der benutzten Systemmakros geben wir auf S. 167 das assemblierte Unterprogramm MADD an, wobei symbolische Befehlscodes und symbolische Adressen beibehalten sind.

Die Parameter werden in den Speicherbereich der Prozedur durch LD-Befehle transportiert, deren zweite Adressen mit dem Indirektadreß-Stern versehen sind, weil die Parameter als Adressen aufgefaßt werden. Da die Adressen jedoch wie Operanden transportiert werden sollen, muß bei den LD-Befehlen zusätzlich zum Stern noch das Direktoperanden-Komma angegeben werden. Dadurch wird erreicht, daß wegen des Sterns die beim Parameter angegebenen Adreßmodifizierungen durchgeführt werden und wegen des Kommas die so entstandene effektive Adresse als Operand angesehen wird. Die LD-Befehle übergeben auf diese Weise die modifizierten Parameter der

Prozedur, in deren Datenbereich sie als effektive Adressen gespeichert werden. Damit können im Hauptprogramm definierte Adressen auf bequeme Weise durch vielstufig geschachtelte Prozeduren transportiert werden, ohne daß zur Ausführungszeit Ketten von Adreßersetzungen durchzuführen wären.

```
MADD  LD   0,B     *0,B
      ADD  F       '2       Code von RES
      LD   3,B     I
      LD   4,B     0,B
      MUL  4,B     4,B
      LD   I       '0
      ADD  F       '5       Code von CALL
      LD   -7,F    '9,P     Rücksprungadresse
      LD   -6,F    B        und Datenbasis speichern
      LD   -5,F    '*4,P    Parameter nach "call by reference"
      LD   -4,F    '*4,P    übergeben, dabei
      LD   -3,F    '*4,P    Adressen modifizieren (vgl. Bild 4.3)
      LD   B       '-5,F    Datenbasisadresse ändern
      LD   P       'VADD    Absprung
      dat  0,B
      dat  *1,B,I
      dat  *2,B,I
      ADD  I       0,B
      CMP  I       4,B
      LD   P       '-14,P   Code von GOTO
      LD   I       3,B
      LD   F       B        Code von RETURN
      LD   B       -1,F
      LD   P       -2,F     Rücksprung                          ■
```

Beispiel 4.16 Fakultät

Wir benutzen als Beispiel zur Demonstration des rekursiven Aufrufes einer Prozedur die Berechnung der Fakultät nach dem folgenden in ALGOL 60 geschriebenen Algorithmus.

```
procedure  fak (y,x);
value x;   integer y,x;
    begin  y := x;
           if x = 1 then goto fin else
           begin y := y-1;
                 fak (y,y);
                 y := y * x
           end
fin: end
```

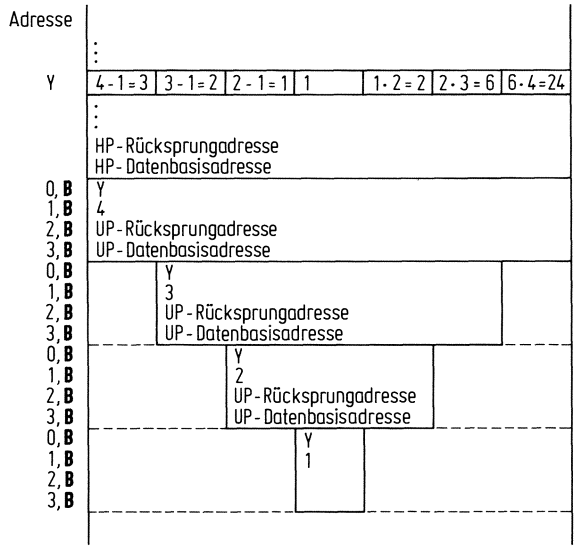
Wir schreiben eine Prozedur FAK mit folgender Wirkung: Die Fakultät des Wortes mit der Adresse des zweiten Parameters wird in die Zelle mit der Adresse des ersten Parameters geschrieben.

```
        org     1000        Hauptprogramm
        :
Y       res     1
X       dat     4
        :
START   ORG     2000
        :
        CALL    FAK     2   Prozeduraufruf von FAK zur Berechnung von 4!
        dat     Y
        dat     X
        :
        end     START
```

```
        org                 Prozedurdefinition von FAK
        ent     FAK
FAK     LD      1,B     *1,B   "call by value" für den 2. Parameter
        LD      *0,B    1,B
        CMP     1,B     '1
        NOP
        GOTO    FIN
        SUB     *0,B    '1
        CALL    FAK     2      Wiederaufruf von FAK
        dat     *0,B
        dat     *0,B
        MUL     *0,B    1,B
FIN     RETURN
        end
```

Der dynamisch verwaltete Datenspeicher wird mit jedem Eintritt in die Prozedur um 4 Plätze vergrößert, wodurch 2 Plätze für die beiden Parameter und vorsorglich 2 Plätze für die Rücksprung- und die Datenbasisadresse reserviert werden. Mit jedem Austritt aus FAK werden diese 4 Plätze wieder freigegeben, so daß die Prozedur entsprechend ihrer Rekursionstiefe immer mit dem jeweils zugehörigen Datenbereich operiert. Das Schema auf S. 169 oben zeigt die dynamische Speicherbelegung zusammen mit dem sich fortlaufend verändernden Wert von Y.

Wir wollen als nächstes das Programm so schreiben, daß nur ein Parameter angegeben zu werden braucht, der beim Aufruf das Argument und bei der Rückkehr das Ergebnis enthält. Zwei Lösungen sind im folgenden angegeben. Im linken Programm wird die im Hauptprogramm benutzte Operandenadresse als Parameter verwendet, so daß bei jedem Prozeduraufruf die gleiche Adresse weitergegeben wird. Im rechten Programm wird eine in der Prozedur benutzte Operandenadresse als Parameter verwendet, so daß bei jedem Prozeduraufruf eine andere Adresse weitergegeben wird.



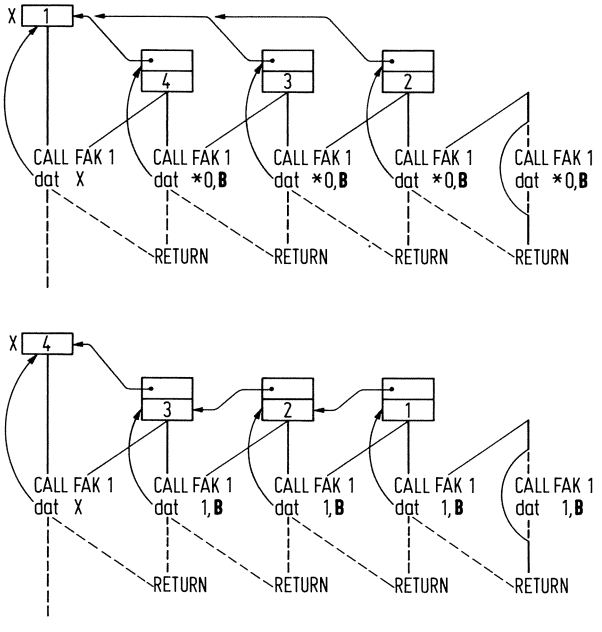

```
        org
        :
X       dat     4
        :
START   ORG     2000
        :
        CALL    FAK     1
        dat     X
        :
        end     START
```

```
        org
        :
X       dat     4
        :
START   ORG     2000
        :
        CALL    FAK     1
        dat     X
        :
        end     START
```

```
        org
        ent     FAK
FAK     CMP     *0,B    '1
        NOP
        GOTO    FIN
        RES     1
        LD      1,B     *0,B
        SUB     *0,B    '1
        CALL    FAK     1
        dat     *0,B
        MUL     *0,B    1,B
FIN     RETURN
        end
```

```
        org
        ent     FAK
FAK     CMP     *0,B    '1
        NOP
        GOTO    FIN
        RES     1
        LD      1,B     *0,B
        SUB     1,B     '1
        CALL    FAK     1
        dat     1,B
        MUL     *0,B    1,B
FIN     RETURN
        end
```

Die Parameteradresse des linken Programmes bezieht sich über beliebig viele Rekursionschritte auf den Datenbereich des Hauptprogrammes und die des rechten Programmes über nur einen Rekursionsschritt auf den Datenbereich des Oberprogrammes. Das folgende Schema illustriert diesen Gegensatz, wobei die eingetragenen Operandenwerte dem letzten Durchlauf der rekursiven Prozedur entsprechen.



Die Vorteile, die sich aus der Organisation eines dynamisch verwalteten Datenspeichers ergeben, sind mit dem Preis bezahlt worden, mit numerischen relativen statt mit symbolischen absoluten Operandenadressen programmieren zu müssen. Dies tritt noch deutlicher hervor, wenn man das Hinzufügen von "**B**" zu den numerischen Adressen dem Assembler überträgt oder wenn die Ausführung der Vorindizierung mit **B** im Prozessor automatisch ausgeführt wird. Sind darüber hinaus die hier benutzten Makrobefehle Maschinenbefehle, so treten die Vorindizierungen und die Register **B**,**F** und **P** nach außen gar nicht mehr in Erscheinung. Dieser "neue" Rechner arbeitet mit automatischer dynamischer Speicherverwaltung, und Aufgabe des "neuen" Assemblers wäre es, wieder die symbolische Adressierung der Operanden zu ermöglichen. Im folgenden, letzten Programm dieses Abschnittes wird dieser Schritt in Richtung eines "neuen" Systems angedeutet (vgl. Beispiel 5.10 aus Abschnitt 5.4).

```
FAK   org
      ent     FAK
      CMP     *0    '1
      NOP
      GOTO    FIN
      RES     1
      LD      1     *0
      SUB     1     '1
      CALL    FAK   1
      adr     1
      MUL     *0    1
FIN   RETURN
      end
```

```
FAK   proc    P1
      ent     FAK
      CMP     *P1   '1
      NOP
      GOTO    FIN
      RES     X     1
      LD      X     *P1
      SUB     X     '1
      CALL    FAK   1
      dat     X
      MUL     *P1   X
FIN   RETURN
      end
```

■

5. Speicherorganisation

Die Speicherorganisation ist eines der zentralen Themen der Rechnerorganisation. Das zeigt sich daran, daß man Speicher fordert, deren Kapazität möglichst groß ist und bei denen die Zeit für den Zugriff auf die gespeicherte Information möglichst kurz ist. Darüber hinaus strebt man einen kleinen Aufbau und geringe Kosten an. Diesen Forderungen stehen die physikalischen Grenzen der Speicherkapazität und der Zugriffszeit sowie die technischen und wirtschaftlichen Grenzen des Aufbaus und der Kosten gegenüber. Da die Idealvorstellungen technologisch derzeit nicht realisierbar sind, versucht man, sie organisatorisch nachzubilden.

Die Speichertechnologie, wie sie sich in der Vergangenheit darbot, war dadurch gekennzeichnet, daß es schnellere Speicher geringerer Kapazität und langsamere Speicher höherer Kapazität gab, wobei die Kosten eine große Rolle spielten. Das ist der Grund, weshalb sich eine Speicherhierarchie von Registerspeicher, Primärspeicher und Sekundärspeicher herausgebildet hat. Die extern gespeicherten Programme und Daten der Benutzer müssen diese Hierarchie durchlaufen, und nur derjenige Ausschnitt gespeicherter Programme und Daten befindet sich jeweils im Registerspeicher, der gerade vom Prozessor verarbeitet wird. Aufgabe der Speicherorganisation ist es also, Ausschnitte des Programmes und seiner Daten auszublenden und vorausschauend durch die Speicherhierarchie in den Registerspeicher zu transportieren.

Aber auch die Prozessororganisation ist in starkem Maße von der Speichertechnologie betroffen. Es ist nicht nur von theoretischem Interesse, Prozessoren zu entwikkeln, die mit Speichern arbeiten, bei denen Registerspeicher, Primärspeicher und vielleicht sogar Sekundärspeicher in einem Randomspeicher oder in einem Assoziativspeicher oder in Speichern mit anderen Zugriffsarten vereinigt sind. Solche Prozessoren orientieren sich an dem Wunsch nach einem genügend schnellen Speicher ausreichend hoher Kapazität. Wenn es solche Speicher gäbe, so vereinfachte sich die Rechnerorganisation beträchtlich. Solange es solche Speicher jedoch nicht gibt, ist es Aufgabe der Rechnerorganisation, sie zu simulieren. Dies drückt sich im Prinzip des virtuellen Speichers aus, der den Benutzern den Eindruck vermittelt, einen Speicher praktisch unbegrenzter Kapazität zur Verfügung zu haben. Dieses Prinzip

wird in abgewandelter Form auf den Prozessor und auf die Peripherie übertragen, so daß man sogar von virtuellen Rechnern spricht: Beim Time-sharing-Betrieb teilen sich viele Benutzer räumlich in den Speicher und zeitlich in den Prozessor und die Peripherie. Jeder Benutzer hat dabei den Eindruck, die Anlage stünde ihm allein zur Verfügung.

In Abschnitt 5.1 werden Speichermodelle und periphere Speicher behandelt. Anhand der Speichermodelle werden die strukturellen Prinzipien der Speicherung und des Zugriffs systematisch dargestellt. Anschließend werden die peripheren Speicher in ihrer technisch-physikalischen Wirkung kurz beschrieben. Der Kernspeicher nimmt dabei insofern eine Sonderstellung ein, als er heute der am weitesten verbreitete Primärspeicher ist, in Zukunft aber mehr und mehr von Halbleiterspeichern verdrängt wird.

In Abschnitt 5.2 wird modularer Speicheraufbau, der in gewissem Sinne eine technische Notwendigkeit ist, dazu ausgenutzt, die Zugriffsmöglichkeiten zu verbessern und die Arbeitsgeschwindigkeiten zu erhöhen. Wir setzen bei diesen Betrachtungen voraus, daß auch bei parallel betriebenen Speicherblöcken die Verarbeitung des Befehlsstroms durch den Prozessor sequentiell geschieht, wodurch die technisch höchstmögliche Arbeitsgeschwindigkeit modular aufgebauter Speicher nicht immer erreicht wird.

In Abschnitt 5.3 werden virtuelle Speicher beschrieben. Anstelle eines einzigen schnellen und großen Speichers werden hierarchisch organisierte Speichersysteme verwendet, bei denen der Sekundärspeicher durch den Primärspeicher und der Primärspeicher durch den Registerspeicher gepuffert werden, wobei sich im Pufferspeicher jeweils ein Ausschnitt des Inhalts des gepufferten Speichers befindet. Als Pufferspeicher eignen sich in erster Linie Assoziativspeicher, weswegen wir die Diskussion der virtuellen Speicher mit dieser Organisationsform beginnen.

In Abschnitt 5.4 wird die dynamische Speicherverwaltung einzelner Programme durch Hardware beschrieben. Zur dynamischen Reservierung und Freigabe von Speicherplatz eignen sich wiederum in erster Linie Assoziativspeicher als Primärspeicher, weswegen wir auch hier die Diskussion der dynamischen Speicherverwaltung mit einem entsprechenden Rechnermodell beginnen. Wird die Freigabe von Speicherplatz in der Weise durchgeführt, daß der zuletzt belegte Platz immer als erster wieder freigegeben wird, so muß die Reservierung innerhalb geschachtelter Blöcke vorgenommen werden, und die Freigabe darf nur beim Austritt aus einem Block erfolgen. In ähnlicher Weise wird auch bei der Platzreservierung innerhalb geschachtelter Prozeduren verfahren, so daß die Belegung des Speichers während der Ausführung eines Programmes pulsiert und zu keinem Zeitpunkt "Löcher" im belegten Speicherplatz entstehen.

5.1 Speichermodelle und periphere Speicher

Hinsichtlich des Aufsuchens der gespeicherten Information kann man zwei grundsätzlich unterschiedliche Arten von Speichern unterscheiden. Bei der ersten Art ist die Information unbeweglich, und eine veränderliche Adresse (Zeiger) weist auf ein gespeichertes Wort. Die Adresse entspricht einer Ortsangabe. Codierte Adressen werden durch ein Schaltnetz decodiert. Man spricht von Speichern mit wahlfreiem Zugriff (Bild 5.1). Bei der zweiten Art ist die Information beweglich, und eine unveränderliche Adresse weist auf ein gespeichertes Wort. Die Adresse entspricht einer Zeitangabe. Codierte Adressen werden durch ein Schaltwerk decodiert. Man spricht von Speichern mit sequentiellem Zugriff (Bild 5.1).

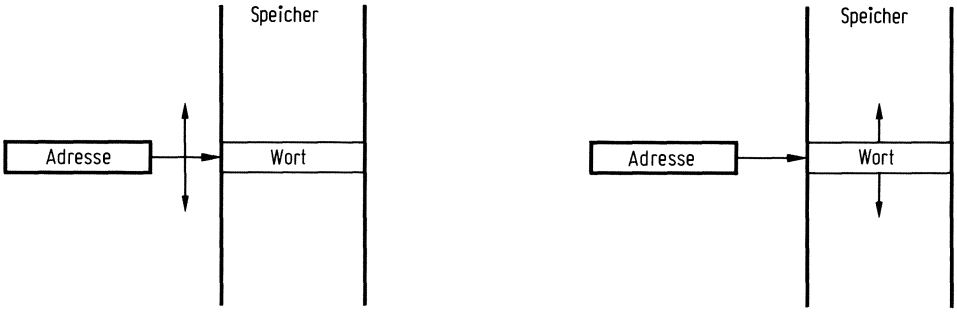


Bild 5.1 Illustration des Unterschieds zwischen Speichern mit wahlfreiem und sequentiellem Zugriff

Bei wahlfreien Speichern erreicht man zu einem Zeitpunkt ein Wort, das an einem bestimmten Ortspunkt adressiert wird. Wahlfreie Speicher sind geeignet für die Verarbeitung einzelner Wörter.

Bei sequentiellen Speichern erreicht man an einem Ortspunkt ein Wort, das zu einem bestimmten Zeitpunkt adressiert wird. Die Information bewegt sich an einer Stelle vorbei, so daß aufeinanderfolgende Wörter unmittelbar gelesen oder geschrieben werden können. Sequentielle Speicher sind geeignet für die Verarbeitung von Blöcken, die aus aufeinanderfolgenden Wörtern bestehen.

Speicher mit wahlfreiem Zugriff

Wörter und Adressen liegen in paralleler Darstellung vor. Die Auswahl eines Wortes erfolgt durch Angabe eines Ortspunktes. In Abschnitt 2.2 ist die Struktur solcher Speicher für den Lesevorgang durch Gl.(2.5) beschrieben:

$$\mathbf{y} = (\mathbf{x} \cdot\!\equiv {}^{T}\mathbf{D}) +\!\cdot \mathbf{C} \tag{5.1}$$

Darin bedeutet **C** Wortspeicher
D Adreßspeicher
x Adresse
y Wort

Die Elemente von **D** und **C** in Gl.(5.1) können konstant oder variabel sein. Bild 5.2 zeigt Gl.(5.1) in symbolischer Darstellung für den Fall, daß sowohl **D** als auch **C** variabel sind.

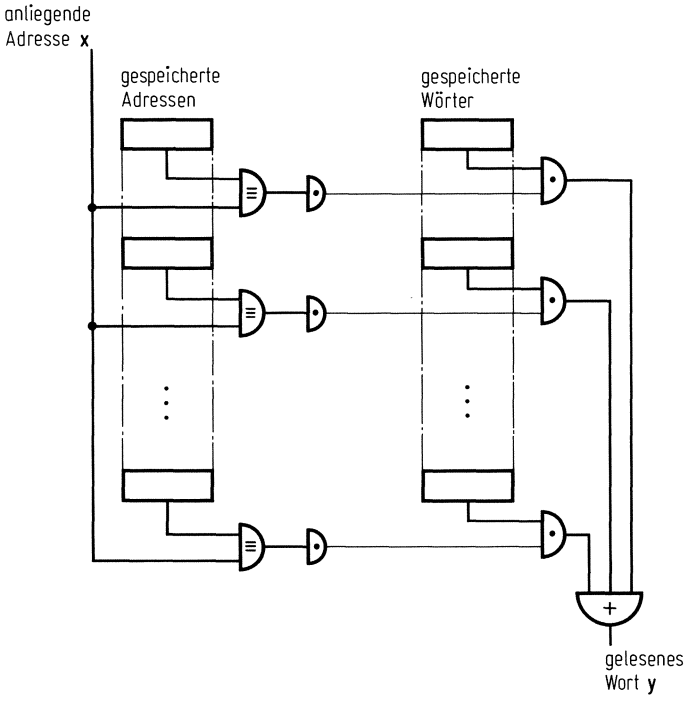


Bild 5.2 Prinzip eines Speichers mit wahlfreiem Zugriff

Die außen anliegende Adresse wird mit allen konstant oder variabel gespeicherten Adressen verglichen, und bei Übereinstimmung der angelegten mit einer gespeicherten Adresse wird das entsprechende konstant oder variabel gespeicherte Wort ausgegeben.

Speichertypen. Je nachdem, ob **D** und **C** konstant oder variabel sind, ergeben sich vier Typen von Speichern mit wahlfreiem Zugriff, die in Bild 5.3 dargestellt sind.

Sind **D** bzw. **C** konstant, so wird die Information fest durch die Anordnung der Bauelemente gespeichert. Diese Bauelemente als Teile von Verknüpfungsgliedern können unterschiedlicher physikalischer Natur sein, z.B. magnetische Schaltkerne oder

Halbleiter-Bauelemente. Sind **D** bzw. **C** variabel, so wird die Information veränderlich in speziellen Elementen gespeichert. Diese Speicherelemente können zwei Zustände entsprechend den zu speichernden booleschen Werten annehmen, und auch hier existiert eine Fülle physikalisch unterschiedlicher technischer Elemente, wie z.B. magnetische Speicherkerne oder Halbleiter-Flipflops.

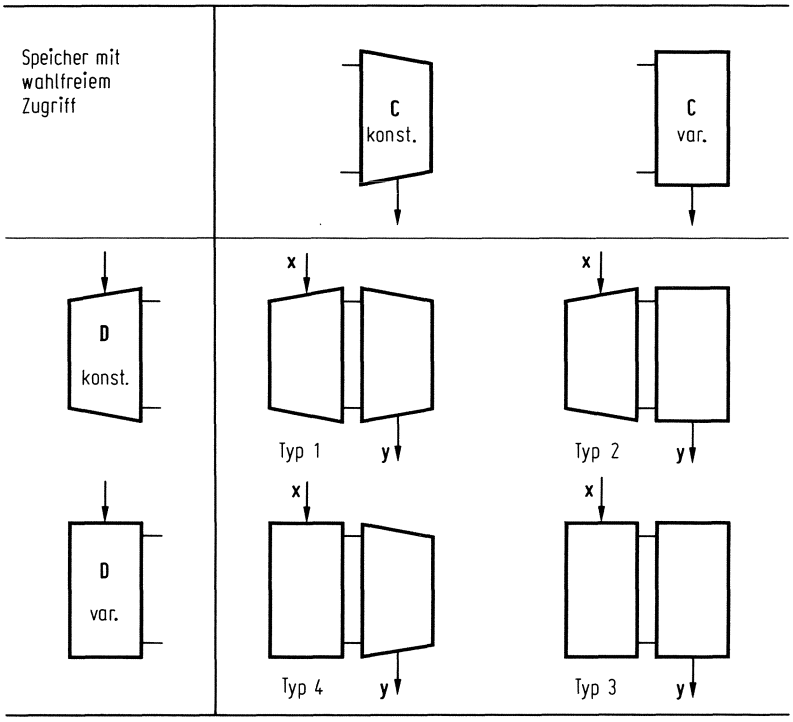


Bild 5.3 Vier Typen von Speichern mit wahlfreiem Zugriff

Bezeichnungen. Entsprechend den vielfältigen Möglichkeiten, die sich durch unterschiedliche Anwendungen ergeben, in denen Speicher eingesetzt werden, und die sich durch die verschiedenartigen Bauelemente ergeben, aus denen Speicher aufgebaut sind, haben sich für die einzelnen Speichertypen verschiedene Bezeichnungen eingebürgert. In der auf S. 176 wiedergegebenen Tabelle sind die unserer Systematik folgenden, die in Deutschland zum Teil genormten und die im englischen Sprachgebrauch hauptsächlich verwendeten Bezeichnungen zusammengestellt. Darüber hinaus existiert eine Fülle weiterer Bezeichnungen, die ebenfalls durch den Einsatz des Speichers (z.B. Schaltnetz, Codeumsetzer für ROMs; Register, Primärspeicher für RAMs; Suchspeicher für CAMs) bzw. durch die Bauelemente des Speichers (z.B. Diodenmatrizen, Schaltkernmatrizen für ROMs; Flipflopspeicher, Kernspeicher für RAMs; integrierte Assoziativspeicher für CAMs) geprägt werden. Die amerikanischen Abkürzungen

Speicher	Bezeichnungen
Typ 1	konstant decodierter Konstantenspeicher Festspeicher Read Only Memory (ROM)
Typ 2	konstant decodierter Variablenspeicher Speicher mit wahlfreiem Zugriff Random Access Memory (RAM)
Typ 3	variabel decodierter Variablenspeicher Assoziativspeicher Content Addressable Memory (CAM)
Typ 4	variabel decodierter Konstantenspeicher

stellen wohl den besten Kompromiß zwischen Kürze, Systematik und Aussagekraft dar. Wir wollen jedoch die der deutschen Norm entsprechenden bzw. angelehnten Begriffe Festspeicher, Randomspeicher und Assoziativspeicher weiterverwenden.

Unserer Darstellung folgend, unterscheiden sich Festspeicher von Randomspeichern hauptsächlich dadurch, daß die gespeicherten Wörter konstant bzw. variabel sind, und Randomspeicher von Assoziativspeichern dadurch, daß die gespeicherten Adressen konstant bzw. variabel sind. Dies impliziert, daß bei Randomspeichern Vorrichtungen zum Schreiben von Wörtern und bei Assoziativspeichern Vorrichtungen zum Schreiben von Adressen und Wörtern vorhanden sein müssen (vgl. Abschnitt 2.2).

Beispiel 5.1 LIFO-Speicher

Aus Speichern mit wahlfreiem Zugriff lassen sich Speicher mit sequentiellem Zugriff, bei denen die Reihenfolge des Zugriffs durch bestimmte Algorithmen festgelegt ist, besonders einfach aufbauen. Bild 5.4 zeigt einen Speicher mit dem Zugriffsalgorithmus Last-in-first-out (LIFO), d.h. das als letztes eingeschriebene Wort wird als erstes ausgelesen. Beim Füllen des Speichers werden die Wörter in der Art eines Stapels aufeinandergeschichtet, der beim Leeren des Speichers von oben wieder abgebaut wird. Eine bewegliche Marke (Adresse) zeigt auf die erste freie Zelle über dem obersten Wort des Stapels. Der Speicher wird gefüllt, indem das am Eingang EIN anliegende Wort in diese Zelle eingegeben und anschließend die Marke um eine Position nach oben geschoben wird. Er wird geleert, indem zuerst die Marke um eine Position nach unten geschoben und dann das auf dem Ausgang AUS erscheinende oberste gespeicherte Wort ausgegeben wird. Dem in Bild 5.4 gezeigten LIFO-Speicher mit unbeweglich gespeicherter Information und beweglichem Zeiger stehen LIFO-Speicher mit beweglich gespeicherter Information und feststehendem Zeiger gegenüber. Beim Füllen eines solchen Speichers werden die Wörter in eine Art Keller hinuntergestoßen und

beim Leeren wieder heraufgezogen. Dabei wird bei jedem Speicherzugriff der gesamte Kellerinhalt um eine Speicherzelle verschoben. Diesen Vorstellungen entsprechend nennt man einen LIFO-Speicher auch kurz Stapel (stack) oder Keller.

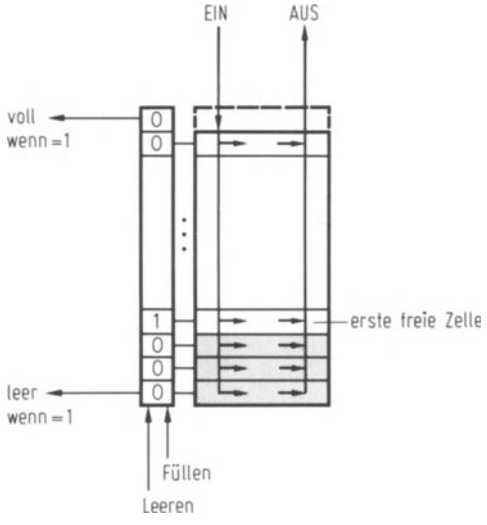


Bild 5.4 LIFO-Speicher, aufgebaut aus einem Speicher mit wahlfreiem Zugriff für Eingabe und Ausgabe ■

Speicher mit sequentiellem Zugriff

Wörter und Adressen liegen in serieller Darstellung vor. Die Auswahl eines Wortes erfolgt durch Angabe eines Zeitpunktes. Die Matrizen in Gl.(5.1) reduzieren sich zu Vektoren, deren Werte sich von einem Zeitpunkt zum nächsten ändern.

$$\mathbf{y} = (\mathbf{x} \cdot\!\equiv^{T} \mathbf{d}) +\!\cdot\ \mathbf{c} \tag{5.2}$$

$$\mathbf{c}_{i+1}^{t+1} = \mathbf{c}_i^t \quad \text{oder} \quad \mathbf{c}^{t+1} = \mathbf{f}(\mathbf{c}^t) \tag{5.3}$$

$$\mathbf{d}_{i+1}^{t+1} = \mathbf{d}_i^t \quad \text{oder} \quad \mathbf{d}^{t+1} = \mathbf{g}(\mathbf{d}^t) \tag{5.4}$$

Darin bedeutet
- $\mathbf{c}$ gespeichertes Wort
- $\mathbf{d}$ gespeicherte Adresse
- $\mathbf{x}$ Adresse
- $\mathbf{y}$ Wort

Im Unterschied zu Gl.(5.1) können die Elemente von $\mathbf{d}$ und $\mathbf{c}$ in Gl.(5.2) nur variabel sein. Bild 5.5 zeigt Gl.(5.2) in symbolischer Darstellung, wobei jeweils die ersten Alternativen von Gln.(5.3) und (5.4) verwendet sind, d.h. daß Adressen $\mathbf{d}_i$ und Wörter $\mathbf{c}_i$ als konstant oder variabel gespeicherte Folgen vorliegen. Die zweiten

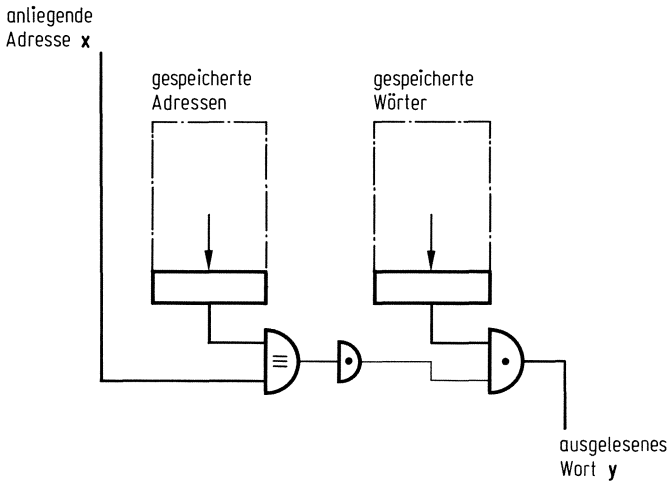


Bild 5.5 Prinzip eines Speichers mit sequentiellem Zugriff

Alternativen dieser Gleichungen deuten die Erzeugung konstanter Wort- bzw. Adreßfolgen durch antonome Schaltwerke an.

Konstant oder variabel gespeicherte Adressen und Wörter bewegen sich synchron an einer Vorrichtung zum Lesen vorbei, und bei Übereinstimmung der von außen angelegten Adresse mit der vorbeilaufenden gespeicherten Adresse wird das entsprechende gespeicherte Wort ausgegeben.

Speichertypen. Die Art des Zugriffes auf sequentielle Speicher wird durch die Bewegung der Information genauer bestimmt. Ist die Bewegung eine Translation, so spricht man von Speichern mit linearem Zugriff, und ist sie eine Rotation, so spricht man von Speichern mit zyklischem Zugriff. In Bild 5.6 sind vier Typen von Speichern dargestellt, die aus der Art des Zugriffes kombiniert mit der Art der Adressierung entstehen. Die Adressierung kann durch einen Zähler erfolgen (Gl.(5.4), rechter Teil) oder durch gespeicherte Adreßfolgen (Gl.(5.4), linker Teil, $\mathbf{d}_i$ Variablen)[1]. Der erste Fall entspricht der festen und der zweite Fall der variablen Adreßdecodierung bei Speichern mit wahlfreiem Zugriff.

Der Transport gespeicherter Daten kann mit Speichermedien unterschiedlicher physikalischer Natur geschehen. Die Information bewegt sich entweder auf einem unbeweglichen Datenträger, z.B. magnetischen Verzögerungsleitungen oder Halbleiter-Schieberegistern, oder sie bewegt sich mit dem Datenträger, z.B. Lochstreifen oder Magnetplatten.

---

[1] Die Möglichkeiten, die sich aus Kombinationen mit Gl.(5.3), rechter Teil, oder Gl.(5.4), linker Teil mit konstantem $\mathbf{d}_i$, ergeben, sind in Bild 5.6 nicht berücksichtigt.

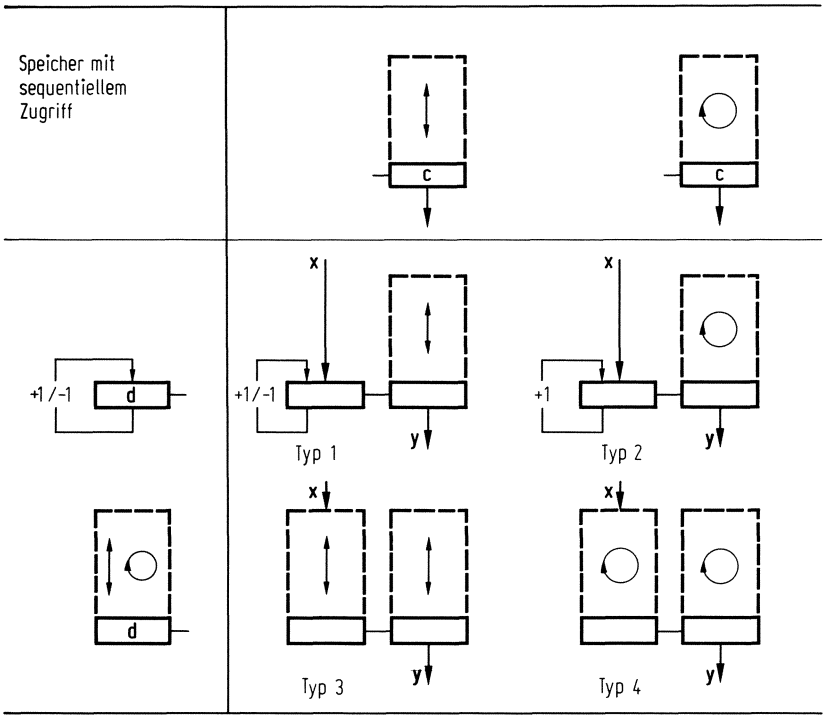


Bild 5.6 Vier Typen von Speichern mit sequentiellem Zugriff

Bezeichnungen. Wie bei den Speichern mit wahlfreiem Zugriff ist auch hier eine Reihe von Bezeichnungen gebräuchlich. Wir wollen die aus unserer Darstellung hervorgehenden systematischen Bezeichnungen in Form einer Tabelle zusammenstellen und jeweils durch ein Beispiel illustrieren.

Speicher	Bezeichnungen
Typ 1	Speicher mit linearem Zugriff, z.B. Magnetband
Typ 2	Speicher mit zyklischem Zugriff, z.B. Magnettrommel
Typ 3	Assoziativspeicher mit linearem Zugriff, z.B. Magnetband mit gespeicherten Adressen
Typ 4	Assoziativspeicher mit zyklischem Zugriff, z.B. Magnettrommel mit gespeicherten Adressen

Auch bei sequentiellen Speichern hat sich eine Fülle weiterer Bezeichnungen eingebürgert, die durch den Einsatz des Speichers (z.B. Schieberegister für Typ 1, Rundschiftregister für Typ 2) oder durch das Speichermedium (z.B. Magnetbandspeicher für Typ 1, Trommelspeicher für Typ 2) geprägt werden.

Bei allen vier Speichertypen hat es den Anschein, als ob die gespeicherte Information an einem Fenster vorbeiliefe und ausgeblendet würde. Speicher mit sequentiellem Zugriff werden vorwiegend als periphere Speicher benutzt, auf deren technischen Aufbau im Anschluß an das folgende Beispiel kurz eingegangen wird.

Beispiel 5.2 FIFO-Speicher

Wie der LIFO-Speicher ist auch der FIFO-Speicher ein Speicher mit sequentiellem Zugriff, bei dem die Reihenfolge des Zugriffs durch einen bestimmten Algorithmus festgelegt ist. Bild 5.7 zeigt einen Speicher mit einer Kombination von wahlfreiem

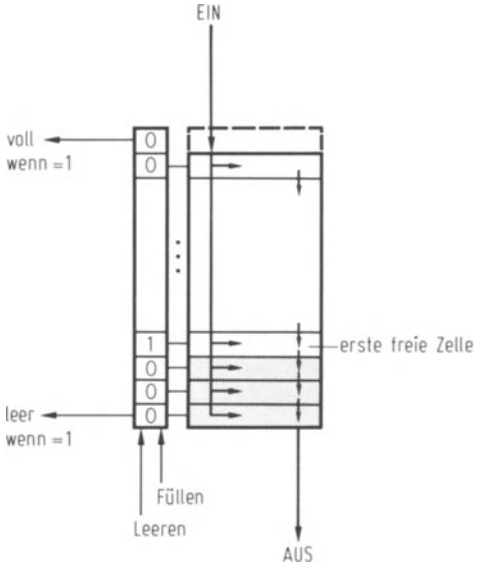


Bild 5.7 FIFO-Speicher, aufgebaut aus einem Speicher mit wahlfreiem Zugriff für die Eingabe und sequentiellem Zugriff für die Ausgabe

Zugriff, d.h. beweglichem Zeiger bei feststehender Information, und sequentiellem Zugriff, d.h. feststehendem Zeiger bei beweglicher Information. Der Speicher arbeitet nach dem Zugriffsalgorithmus First-in-first-out (FIFO), d.h. das als erstes eingeschriebene Wort wird als erstes ausgelesen. Beim Füllen des Speichers werden die Wörter wie beim LIFO-Speicher in der Art eines Stapels aufeinandergeschichtet, der jedoch beim Leeren im Gegensatz zum LIFO-Speicher von unten abgebaut wird, wobei die darüberliegenden Wörter nachrutschen. Eine bewegliche Marke (Adresse) zeigt auf die erste freie Zelle über dem obersten Wort des Stapels. Der Speicher wird gefüllt, indem der am Eingang EIN anliegende Wert in diese Zelle eingegeben und anschließend die Marke um eine Position nach oben geschoben wird. Er wird geleert, indem der gesamte Inhalt des Speichers zusammen mit der Marke

um eine Position nach unten geschoben und gleichzeitig das auf dem Ausgang AUS erscheinende unterste gespeicherte Wort ausgegeben wird. Wie der LIFO-Speicher so kann auch der FIFO-Speicher unmittelbar als sequentieller Speicher aufgebaut werden. Seine gespeicherten Wörter kann man sich als Menschen vorstellen, die vor einem Abfertigungsschalter Schlange stehen. Ein neu hinzukommendes Wort muß sich hinten anstellen und solange warten, bis es vorne abgefertigt wird. Wie der LIFO-Speicher so kann der FIFO-Speicher aber auch aus einem Speicher aufgebaut werden, der nur wahlfreien Zugriff erlaubt. Dann sind zwei Marken (Adressen) nötig, die auf den Anfang und das Ende des belegten Teiles des Speichers zeigen, und der Speicher wird als Ring aufgefaßt. So werden Bereiche von Randomspeichern vielfach zur Pufferung von Daten eingesetzt. Diesen Vorstellungen entsprechend nennt man einen FIFO-Speicher auch Warteschlange (queue) oder Ringpuffer. ■

Typische periphere Speicher

Im folgenden wird der technische Aufbau einiger typischer peripherer Speicher kurz beschrieben. Der Magnetkernspeicher nimmt in diesem Abschnitt insofern eine Sonderstellung ein, als er heute noch hauptsächlich als Primärspeicher verwendet wird. Er wird jedoch durch den Einsatz von Halbleiterspeichern als Primärspeicher immer mehr verdrängt. Als Sekundärspeicher werden heute in erster Linie Magnettrommel-, Magnetplatten- und Magnetbandspeicher verwendet. Beim Magnetblasenspeicher wird im Gegensatz zum elektromotorischen Transport des magnetischen Materials bei Magnetplatte, -band und -trommel die Information im Trägermaterial durch magnetische Felder verschoben.

Kernspeicher. Als Speicherelemente werden Ringkerne aus ferromagnetischem Material verwendet, deren Außendurchmesser zwischen 0,2 und 4 mm liegt. Ein Speichersystem enthält einige Hundert bis zu 10 Milliarden Kerne.

Die Speicherwirkung der Kerne beruht auf der remanenten Magnetisierung des Materials. Die Steuerung der Magnetisierungsrichtung erfolgt durch Stromimpulse über verschiedene Leitungen, die durch die Kerne gefädelt sind. Beim Lesen der Information wird auf einem ebenfalls durch die Kerne gefädelten Lesedraht eine Spannung induziert, wobei die im Kern gespeicherte Information gelöscht wird. Soll sie erhalten bleiben, so muß sie in einem Register (dem Wortregister) zwischengespeichert und im Anschluß an das Lesen wieder in den Speicher geschrieben werden. Bei jedem Zugriff auf den Speicher werden also nacheinander eine Lese- und eine Schreibphase durchlaufen. Beide Phasen zusammen bezeichnet man als Speicherzyklus, dessen Dauer (etwa 1 µs) ein wesentliches Merkmal eines Kernspeichers darstellt. Die Zeit, die vergeht, bis die gelesene Information im Wortregister zur Verfügung steht, wird als Zugriffszeit bezeichnet. Sie ist ein weiteres charakteristisches Merkmal eines Speichers. Da aus technischen Gründen die Lesephase im allgemeinen kürzer ist als die Schreibphase, liegt die Zugriffszeit von Kernspeichern bei etwa einem Drittel der Zykluszeit. Bild 5.8 zeigt das Blockschaltbild eines Kernspeichers. Die Adresse muß während eines ganzen Zyklus im Adreßregister zur Verfügung stehen. Die Adreßdecodierung erfolgt zum Teil über Diodenmatrizen und Schaltkerne und zum Teil in den Speicherkernen selbst.

Soll in den Speicher geschrieben werden, so wird das Wortregister am Ende der Lesephase von außen überschrieben, und in der anschließenden Schreibphase gelangt der Inhalt dieses Registers in die ausgewählte Speicherzelle.

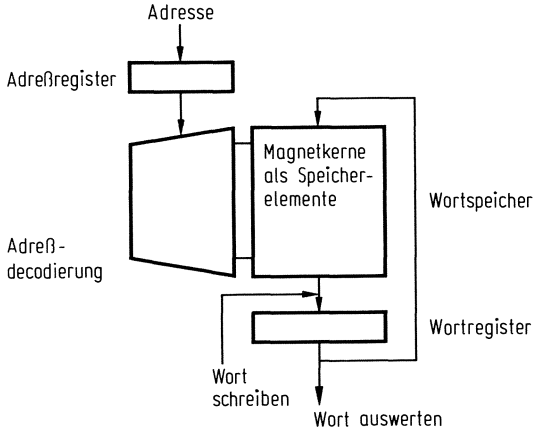


Bild 5.8 Prinzipschaltbild eines Kernspeichers

Magnettrommelspeicher. Als Speichermedium wird die Oberfläche einer rotierenden Trommel benutzt, die aus einer magnetisierbaren Eisenoxydschicht besteht. Die zu speichernde Information erzeugt mit Hilfe von Schreibköpfen magnetisierte Spuren auf der Trommel, aus denen durch Leseköpfe die Information wieder gewonnen werden kann. Man unterscheidet zwischen feststehenden und beweglichen Magnetköpfen. Damit die Montage der Köpfe keine Schwierigkeiten bereitet, sind die Köpfe in einer Schraubenlinie um die Trommel herum angeordnet. Die mittlere Zugriffszeit ist gleich der halben Umdrehungszeit der Trommel. Bei den Trommeln mit beweglichen Magnetköpfen kommt zu der mittleren Zugriffszeit noch die Zeit für die Positionierung der Magnetköpfe hinzu.

Magnetplattenspeicher. Bei den Magnetplattenspeichern dienen die beschichteten Oberflächen von kreisförmigen Platten als Speichermedium. Meist sind mehrere Platten in Form eines Plattenstapels übereinander montiert. Wie beim Trommelspeicher verwendet man auch hier feststehende oder bewegliche, auf einem Luftkissen gleitende Magnetköpfe.

Im Gegensatz zum Trommelspeicher gibt es bei den Plattenspeichern Ausführungen, bei denen der Plattenstapel auswechselbar ist. Damit ist die Speicherkapazität theoretisch unbegrenzt.

Magnetbandspeicher. Als Speichermedium dient ein flexibles Kunststoffband, auf dem eine dünne Eisenoxydschicht aufgebracht ist. Die Länge der 1/2 Zoll (12,7 mm) breiten Bänder liegt zwischen 500 und 1000 m. Da die Relativgeschwindigkeit des Speichermediums klein ist, liegt das Band beim Lesen und Schreiben am Magnetkopf an.

Eine Magnetbandeinheit muß in der Lage sein, das Band in kurzer Zeit auf die Maximalgeschwindigkeit zu beschleunigen. Wegen der Trägheit der Spulen wird zur Zugentlastung des Bandes zwischen Köpfen und Spulen je eine Bandschleife erzeugt. Ein Vakuumkanal und eine Lichtschranke sorgen für die richtige Länge der Schleifen.

Magnetblasenspeicher. Ferromagnetische Materialien besitzen im allgemeinen eine sog. Domänenstruktur: Sie sind in Bereiche (Weißsche Bezirke) unterteilt, in denen die Magnetisierung eine einheitliche Richtung hat. In Schichten von wenigen µm Stärke erstrecken sich die Domänen über die gesamte Dicke des Speichermaterials und lassen

Größenordnungen einiger typischer Kennwerte von Halbleiter- und Magnetspeichern (Stand April 1976)

		Speicherkapazität [Bits je Chip bzw. Modul][a]	Zugriffszeit bzw. Zykluszeit	Datenrate [Wörter/s]	Leistungsverbrauch je Bit	Kosten [Pf/Bit]	Verwendungszweck
Halbleiterspeicher	CAM bipolar	16	20 ns	10^7	50 mW	500	Cache-Memories
	MOS	128	100 ns	10^6	1 mW	–[b]	
	RAM bipolar	1 K	20 ns	10^7	1 mW	5	Register, Puffer- und Primärspeicher
	MOS	4 K	200 ns	10^6	100 µW	1	
	ROM bipolar	10 K	50 ns	10^7	100 µW	1	Mikroprogrammspeicher
	MOS	16 K	500 ns	$2 \cdot 10^6$	100 µW	1	
Magnetspeicher	Kern	5 M	1 µs	10^6	10 µW	3	Primärspeicher
	Trommel	100 M	10 ms	10^6	10 µW	10^{-1}[c]	Sekundärspeicher (Hintergrund- oder Massen-Speicher)
	Platte	1000 M[d]	10 ms	10^6	10 µW	10^{-3}[c]	
	Band	500 M	1 s bis 1 min	10^5	10 µW	10^{-4}[c]	
	Magnetblasen	10 M	1 µs	10^6	1 µW	10^{-2}	

[a] $1\,K = 2^{10} = 1024$, $1\,M = 2^{20} = 1048576$
[b] nur als Muster entwickelt
[c] Kosten ohne Antriebseinheit
[d] Plattenstapel

sich durch ein senkrecht zur Schicht angelegtes Magnetfeld zu kleinen Zylindern von ca 6 μm Durchmesser formen, die als Magnetblasen bezeichnet werden. Sie sind innerhalb der Speicherschicht verschieblich und werden mit Hilfe eines in der Schichtebene rotierenden Drehfeldes schrittweise auf vorgegebenen Bahnen bewegt. Die Bahnen werden durch Muster aus magnetisierbarem Metall (Permalloy) vorgeschrieben, die auf der Speicherschicht angebracht sind.

Übersicht. Die Tabelle auf S. 183 gibt einen Überblick über die Größenordnungen einiger Kennwerte typischer Magnetspeicher. Zum Vergleich sind auch die derzeit gebräuchlichen Halbleiterspeicher mit aufgeführt. Aus der Tabelle ist zu ersehen, daß bei steigender Speicherkapazität die Kosten pro Bit abnehmen. Allerdings steigt dabei die Zugriffszeit erheblich. Die zur Zeit (1976) noch in der Entwicklung befindlichen Magnetblasenspeicher versprechen hier eine beachtliche Leistungssteigerung, da sie trotz großer Kapazität kurze Zugriffszeiten bei geringem Leistungsverbrauch ermöglichen.

Bei der physikalischen Ausführung von Halbleiterspeichern unterscheidet man die sog. bipolare von der unipolaren Technik, die sich auf die Geschwindigkeit und den Leistungsverbrauch der Speicher auswirkt. Bei bipolaren Speichern finden npn- oder pnp-Transistoren Verwendung, bei den unipolaren Speichern Feldeffekt-Transistoren. Unipolare Speicher werden auch als MOS-Speicher (metal oxide semiconductor) bezeichnet.

5.2 Modularer Speicheraufbau

Speicher mit wahlfreiem Zugriff, deren Struktur hinsichtlich der Adressierung und des Zugriffes durch Gl.(5.1) beschrieben ist, werden aus technischen Gründen aus mehreren Blöcken aufgebaut. Für den Benutzer von Rechenanlagen ergibt sich aus diesem modularen Aufbau der Vorteil, den Ausbau des Speichers stufenweise vornehmen zu können. Für den Konstrukteur von Rechenanlagen bietet der modulare Speicheraufbau die Möglichkeit, unter anderem die einzelnen Blöcke "assoziativ" zu adressieren, sie untereinander zu verschränken oder sie parallel zu betreiben. Durch die assoziative Blockauswahl lassen sich die Vorteile des Assoziativspeichers mit der Wirtschaftlichkeit des Randomspeichers kombinieren, und durch verschränkten oder parallelen Betrieb kann die Geschwindigkeit des Speichers erhöht werden. Wir erörtern diese Gesichtspunkte im folgenden ausführlicher.

Modulare Speicherorganisation bedingt die Aufteilung der Speicheradresse in zwei Teile, d.h. die Staffelung der Decodierung in Blockauswahl und Wortauswahl. Wir bezeichnen die Adresse für die Blockdecodierung als Blockadresse $\mathbf{u}$ und die Adresse für die Wortdecodierung als Wortadresse $\mathbf{v}$. Beide Adressen zusammengesetzt ergeben die Adresse $\mathbf{x} = [\mathbf{u}\ \mathbf{v}]$, mit der ein beliebiges Wort in einem beliebigen Block adressiert werden kann. Wir betrachten einen Speicher aus n Blöcken und bezeichnen die Decodiermatrix für die Blöcke mit $\mathbf{D}$ und die Decodiermatrizen für die Wörter in den Blöcken mit $\mathbf{E}_0, \mathbf{E}_1, \ldots, \mathbf{E}_{n-1}$. Die zugehörigen Speicherblöcke für die Wörter

bezeichnen wir mit $\mathbf{C}_0, \mathbf{C}_1, \ldots, \mathbf{C}_{n-1}$. Damit entsteht aus Gl. (5.1)

$$\mathbf{y} = \left([\mathbf{u}\,\mathbf{v}] \cdot\!\equiv {}^{T}\begin{bmatrix} ({}^{T}\mathbf{i} +\!\cdot\, \mathbf{d}_0) & \mathbf{E}_0 \\ ({}^{T}\mathbf{i} +\!\cdot\, \mathbf{d}_1) & \mathbf{E}_1 \\ \vdots & \vdots \\ ({}^{T}\mathbf{i} +\!\cdot\, \mathbf{d}_{n-1}) & \mathbf{E}_{n-1} \end{bmatrix} \right) +\!\cdot \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{n-1} \end{bmatrix} \tag{5.5}$$

Der durch diese Gleichung beschriebene Zusammenhang zwischen $\mathbf{y}$ und $[\mathbf{u}\,\mathbf{v}]$ kann auch in der folgenden Form geschrieben werden:

$$\mathbf{y} = (\mathbf{u} \cdot\!\equiv {}^{T}\mathbf{D}) +\!\cdot \left(\left(({}^{T}\mathbf{i} +\!\cdot\, \mathbf{v}) \cdot\!\equiv {}^{T}\begin{bmatrix} \mathbf{E}_0 \\ \mathbf{E}_1 \\ \vdots \\ \mathbf{E}_{n-1} \end{bmatrix} \right) +\!\cdot \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{n-1} \end{bmatrix} \right) \tag{5.6}$$

Die Wortadresse $\mathbf{v}$ wird über einen Verteiler an alle n Speicherblöcke gegeben, deren Ausgänge über einen Multiplexer zusammengefaßt werden, der von $\mathbf{u}$ über einen Decodierer gesteuert wird. Dementsprechend braucht die Blockdecodierung nur einmal aufgebaut zu werden. Bild 5.9 zeigt die Struktur des modularen Speichers unter der Annahme, daß die Elemente von $\mathbf{D}$ und $\mathbf{E}$ konstant und die von $\mathbf{C}$ variabel sind.

Obwohl die Größen $\mathbf{D}, \mathbf{E}$ und $\mathbf{C}$ in Gl. (5.6) Konstanten oder Variablen in bunter Mischung sein können, wollen wir uns im folgenden darauf beschränken, daß $\mathbf{C}$ grundsätzlich Variable ist und nur die verschiedenen Varianten diskutieren, die dadurch entstehen, daß $\mathbf{D}$ und $\mathbf{E}$ Konstanten oder Variablen sein können. Von den vier noch verbleibenden Möglichkeiten seien die beiden wichtigsten herausgegriffen: die der konstanten und die der variablen Decodierung von Randomspeicherblöcken.

Konstante und variable Blockdecodierung

Konstantes $\mathbf{D}$ in Gl. (5.6) bedeutet: Einer von n Speicherblöcken wird durch variable te Decodierung der Blockadresse ausgewählt. Bei gleichbleibender Adreßlänge $\mathbf{x}$ gehen die beiden Grenzfälle $\mathbf{x} = \mathbf{u}$ und $\mathbf{x} = \mathbf{v}$ in Randomspeicher über, die aus einem Block aufgebaut sind; Gl. (5.6) vereinfacht sich entsprechend.

Variables $\mathbf{D}$ in Gl. (5.6) bedeutet: Einer von n Speicherblöcken wird durch variable Decodierung der Blockadresse ausgewählt. Die Blockadresse kann größer sein, als es für die Anzahl der Blöcke notwendig wäre. Jeder Speicherblock hat eine veränderliche Adresse, die nicht mit seiner laufenden Nummer übereinzustimmen braucht, und die realen (physikalischen) Adressen der Blöcke sind ersetzt durch virtuelle (logische) Adressen. Auch bei "fehlenden" Blöcken läßt sich dadurch ein fortlaufend zusammenhängender Speicher herstellen. Bei gleichbleibender Adreßlänge von

x = [**u**v] gehen die beiden Grenzfälle **x** = **u** und **x** = **v** in einen Assoziativspeicher bzw. in einen Randomspeicher über, die jeweils aus einem Block bestehen; Gl. (5.6) vereinfacht sich entsprechend.

Bild 5.9 zeigt das Prinzip dieser beiden Adressierungsarten von Speicherblöcken für Halbleiterspeicher entsprechend der in Gl. (5.6) beschriebenen Struktur, wobei **D** als konstant oder variabel zu interpretieren ist. Für Kernspeicher muß der Adreßverteiler durch einen Demultiplexer ersetzt werden, da wegen des Informationsverlustes beim Lesen nur ein Block adressiert werden darf. Zum Regenerieren der Information wird das gelesene Wort **y** in einem Register zwischengespeichert und als Wort **z** wieder geschrieben.

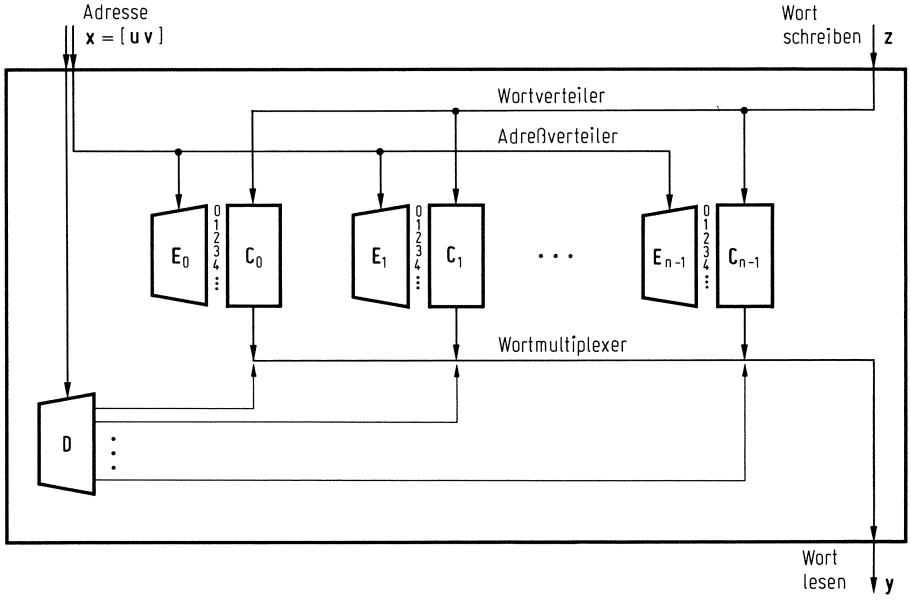


Bild 5.9 Blockauswahl beim unverschränkten Speicher

Unverschränkte und verschränkte Speicher

Die physikalische Aufteilung eines Speichers in einzelne unabhängige Moduln kann zur Erhöhung der Geschwindigkeit des Speichers ausgenutzt werden, wenn jeder Modul gegenüber den anderen mit versetztem Zyklus arbeitet. Dies ist besonders bei Kernspeichern wegen des zur Regenerierung notwendigen Wiedereinschreibens vorteilhaft. Beim unverschränkten Betrieb eines modular aufgebauten Kernspeichers brauchen Adreß- und Wortregister für alle Blöcke nur einmal vorgesehen zu werden. Die einzelnen Blöcke werden seriell betrieben, und die Zyklen der Blöcke sind nahtlos aneinandergereiht. Beim verschränkten Betrieb eines modular aufgebauten Kernspeichers werden Adreß- und Wortregister hingegen für jeden Block aufgebaut. Die

einzelnen Blöcke werden parallel betrieben, so daß die Zyklen der Blöcke sich gegenseitig überlappen. n Blöcke (wegen der einfachen Organisation ist n im allgemeinen eine Zweier-Potenz, z.B. 4 oder 8) werden "verzahnt" nacheinander angewählt, so daß sich die Übertragungsrate ver-n-facht (z.B. vervier- oder verachtfacht). Da Programm- und auch Datenstrukturen in hohem Maße sequentiell organisiert sind, werden aufeinanderfolgende Adressen rundum nebeneinanderliegenden Blöcken zugeordnet. Die niederwertigen Adreßbits sind demnach für die Blockadressierung und die höherwertigen Adreßbits für die Wortadressierung innerhalb der Blöcke zu verwenden. Die Anordnung von **u** und **v** in Gl.(5.6) beschreibt den nicht verschränkten Speicher (**x** = [**u v**]). Für den voll verschränkten Speicher müssen **u** und **v** in Gl.(5.6) vertauscht werden (**x** = [**v u**]). Eine Variante entsteht, wenn **u** aufgeteilt wird und die Teile als niedrigste und höchste Adreßbits verwendet werden; wir sprechen von teilverschränkten Speichern. Bild 5.10 zeigt einen voll verschränkten

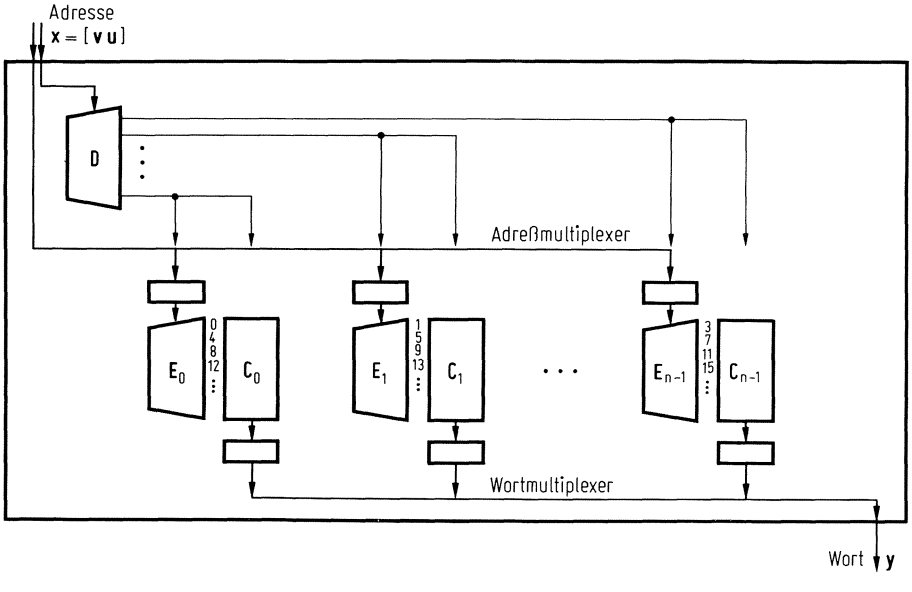
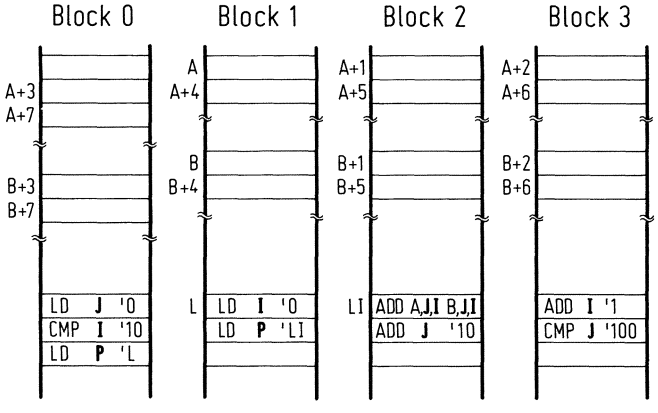


Bild 5.10 Blockauswahl beim verschränkten Speicher (Schreibbus nicht gezeichnet)

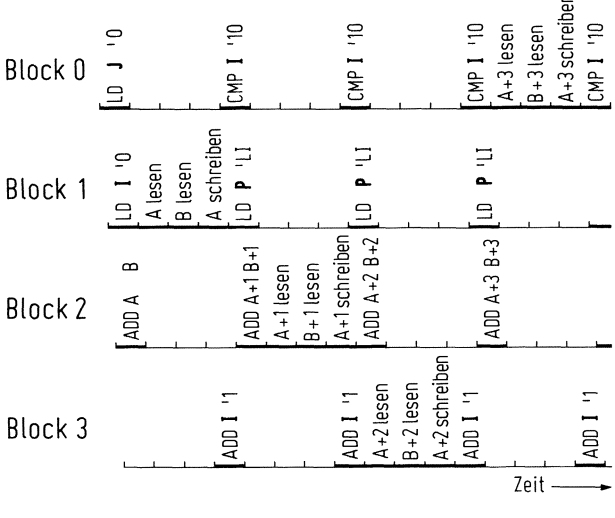
Kernspeicher, wobei aus Gründen der Übersichtlichkeit die Leitungen zum Schreiben und die Leitungen zum Regenerieren für die einzelnen Blöcke nicht gezeichnet sind (vgl. Bild 5.8). Jeder Block benötigt im Gegensatz zum unverschränkten Speicher aufgrund des parallelen Betriebes eigene Pufferregister, eigene Schreib- und Leseverstärker und eine eigene Speichersteuerung. Durch die verschränkte Adressierung und die zeitlich versetzten Zyklen der einzelnen Blöcke können Wörter sequentiell mit n-facher Geschwindigkeit am Ausgang **y** des Wortmultiplexers abgegriffen werden. Das folgende Beispiel soll den Betrieb eines Speichers mit verschränkter Adressierung illustrieren.

Beispiel 5.3 Verschränkter Speicherbetrieb*

Wir untersuchen die Wirkung eines verschränkten Speichers mit vier Blöcken anhand des Programmes zur Matrixaddition aus Beispiel 4.8. Befehle und Operanden seien wie folgt in den einzelnen Blöcken gespeichert.



Wir nehmen an, daß die Speicherzyklen der vier Blöcke gegeneinander um ein Viertel der Zykluszeit versetzt sind und daß die Verarbeitungszeit des Prozessors ein Viertel der Zykluszeit beträgt. Der Block (i+1) mod 4 wird in jedem Fall, auch wenn er über längere Zeit nicht adressiert wurde, ein Viertel der Zykluszeit später als der Block i mod 4 "angestoßen". Damit ist der sequentielle Ablauf des Programmes sichergestellt. Befehle und Operanden werden zwar sich überlappend gelesen, aber streng sequentiell ausgeführt bzw. verändert. Das folgende Schema soll den Ablauf des Programmes mit überlappenden Zyklen illustrieren, wobei in den Befehlen die modifizierten Adressen angegeben sind.



* Vgl. z.B. Speicherverschränkung bei der Rechenanlage TR 440 von AEG-TELEFUNKEN.

Wie an diesem Schema zu sehen ist, werden für die Ausführung der inneren Schleife des Programms 4 Speicherzyklen benötigt. Der gleiche Prozeß in einem Speicher mit unverschränkter Adressierung benötigte demgegenüber 7 Speicherzyklen. Die Ausführungszeit des Programms hat sich - grob gerechnet - durch die Verschränkung halbiert. Bei dieser Rechnung ist nicht berücksichtigt, daß Zweiadreßbefehle, deren Operanden im Primärspeicher stehen, wegen ihrer großen Länge mehr als ein Speicherwort belegen können. Wegen der Adressierung aufeinanderfolgender Speicherwörter würde sich das Verhältnis zugunsten des verschränkten Betriebes verschieben. ■

Speicher für seriellen und parallelen Betrieb

Obwohl bei einem modular aufgebauten Speicher mit verschränkter Adressierung alle Blöcke überlappend parallel arbeiten, erscheint er nach außen hin als seriell ar-

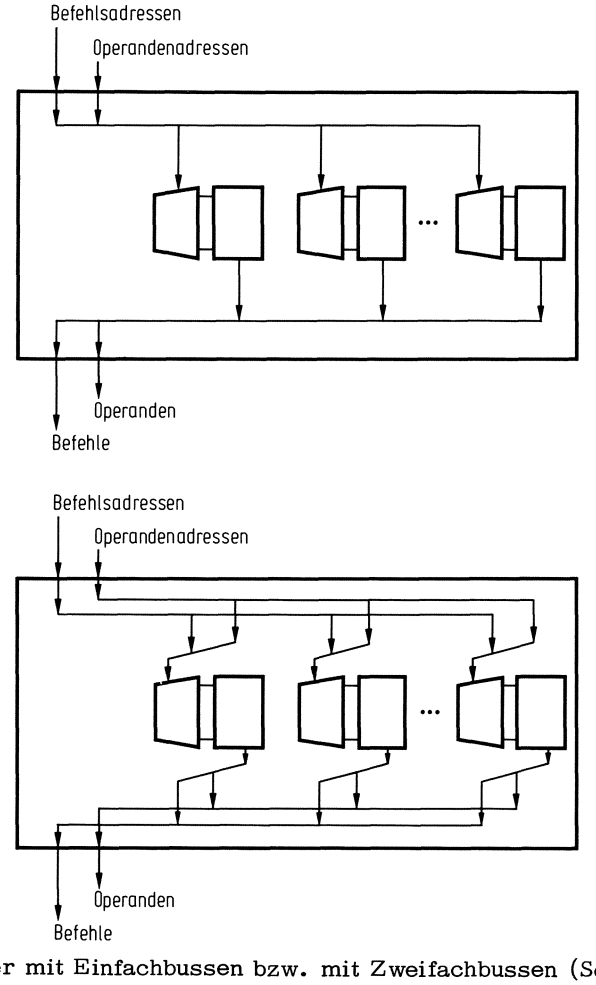


Bild 5.11 Speicher mit Einfachbussen bzw. mit Zweifachbussen (Schreibbusse nicht gezeichnet)

beitender Speicher mit variabler kürzerer Zykluszeit. Das liegt in der Art seiner "Schnittstelle zur Außenwelt". In Bild 5.10 existieren jeweils nur ein Adreßbus und ein Datenbus. Es versteht sich von selbst, daß diese Einfachbusse nur seriell betrieben werden können, und wenn mehrere äußere Moduln an den Speicher angeschlossen werden sollen, so müssen sie über Multiplexer und Demultiplexer mit den Einfachbussen verbunden werden. Erst der Übergang auf Mehrfachbusse schafft die Möglichkeit, daß mehrere äußere Moduln parallel mit dem Speicher arbeiten können. Bei Einprozessoranlagen sind diese äußeren Moduln zum Beispiel das Programmwerk und das Datenwerk, bei Mehrprozessoranlagen können es mehrere Prozessoren sein. Im Gegensatz zu Einfachbussystemen benötigen Mehrfachbussysteme "zweidimensionale" Decodierer. In Bild 5.11 sind oben ein Einfachbus- und unten ein Zweifachbussystem mit ein- bzw. zweidimensional ansteuerbaren Multi- und Demultiplexern dargestellt. Man erkennt beim Vergleich der beiden Bilder deutlich den Mehraufwand des Zweifachbus- gegenüber dem des Einfachbussystems. Allgemeine Fragen des Zusammenschlusses von Moduln werden in Abschnitt 6.2 (Modulare Konfigurationen) behandelt. Abschließend wollen wir ein Beispiel geben, das gleichermaßen einen Einblick in die Größenordnung eines entsprechend Bild 5.11 (unterer Teil) aufgebauten Speichers wie in den parallelen Betrieb eines modularen Systems verschaffen soll.

Beispiel 5.4 Paralleler Speicherbetrieb

Wir betrachten einen modular aufgebauten Speicher, der bis zu 8 Blöcke mit einer Kapazität von 32K Wörtern je Block umfassen kann.[1] Die Adresse eines Wortes innerhalb eines Blocks hat demnach eine Länge von 15 Bits. Zur Adressierung von acht Blöcken sind weitere drei Bits erforderlich. Insgesamt beträgt die Adreßlänge einer Speicheradresse 18 Bits. Der Speicher wird teilverschränkt adressiert. Die Blockauswahl ist in Bild 5.12 symbolisch dargestellt. Die Speicherblöcke mit geraden Adressen sind links und die mit ungeraden Adressen sind rechts angeordnet. Diese Blockreihen sind zu Speicherbänken zusammengefaßt, die ihrerseits nicht verschränkt adressiert werden. Die beiden Speicherbänke können voll parallel betrieben werden, wenn in einem der beiden Adreßregister immer nur gerade Adressen und im anderen immer nur ungerade Adressen erscheinen. In Bild 5.12 ist dieser Betriebsfall durch dicke Linien angedeutet.

Wird dieser Speicher in einer Einprozessoranlage betrieben, so trennt man Befehle und Operanden und speichert sie grundsätzlich in verschiedenen Bänken. Diese Aufteilung, die vom Logischen her zur Klarheit beiträgt, erfährt dadurch vom Physika-

---

[1] Vgl. modulare Speicherorganisation bei der Rechenanlage UNIVAC 1108.

lischen her eine wünschenswerte Unterstützung. Befehle werden z.B. nur mit geraden und Operanden nur mit ungeraden Adressen versehen. Wenn ein Befehl einen Operanden holt, kann parallel dazu der nächste Befehl gelesen werden.

Für den in Kapitel 4 beschriebenen Rechner mit den Möglichkeiten, in einem Befehl zwei Speicherzellen zu adressieren und Direktoperanden zu verwenden, bringt diese Art der modularen Speicherorganisation kaum einen Geschwindigkeitsvorteil. Die

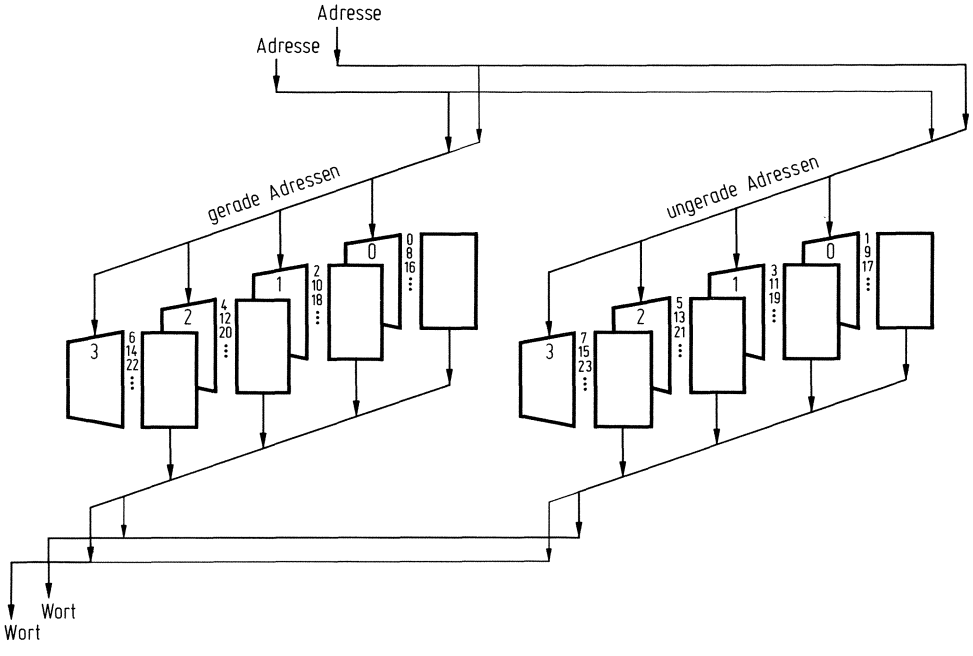
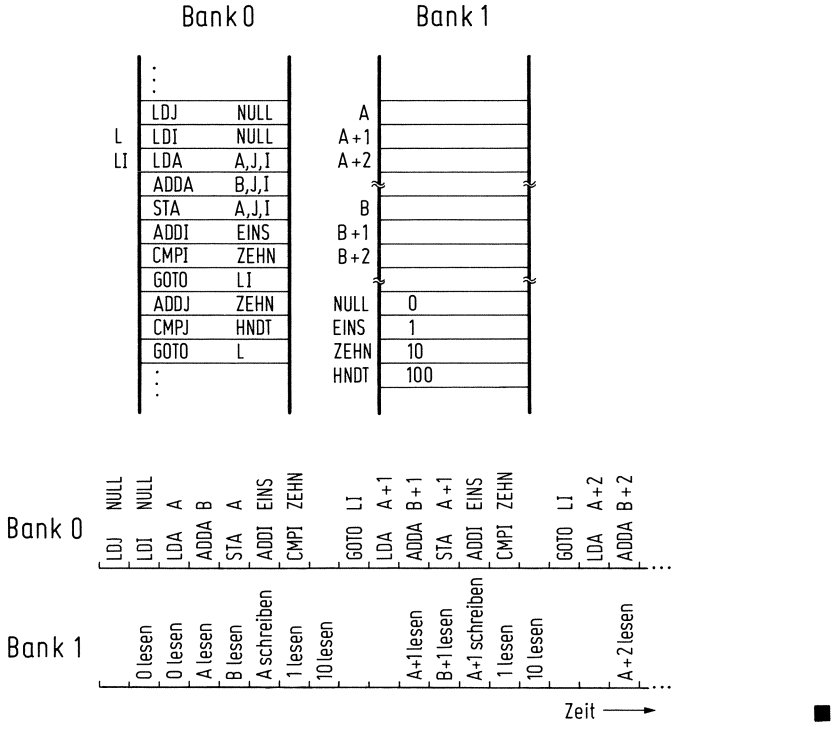


Bild 5.12 Blockauswahl bei einem teilweise verschränkten Speicher mit Zweifachbussen (Schreibbusse nicht gezeichnet)

beste Ausnutzung des parallelen Betriebes eines solchen Speichers ergibt sich für Einadreßrechner ohne Direktoperandenverarbeitung. Das Lesen von Befehlen und Operanden wechselt sich gegenseitig ab, und beide Speicherbänke arbeiten bis auf "Störungen" (z.B. durch Sprungbefehle) voll parallel. Wir wollen wieder das Programm für die Matrixaddition aus Beispiel 4.8 zur Veranschaulichung des parallelen Betriebes benutzen, wobei wir zu Einadreßbefehlen übergehen. Das dabei zugrundegelegte Rechnermodell entspricht dem in Kapitel 2 entwickelten Einadreßrechner mit doppelter Indizierung. Im folgenden Schema sind entsprechend ihrer Speicherung in den beiden unabhängigen Bänken das Programm und die Daten nebeneinander angeordnet. Darunter ist der Ablauf des Programmes skizziert, wobei in den Befehlen die modifi-

zierten Adressen eingetragen sind. Verglichen mit einem seriell arbeitenden unverschränkten Speicher halbiert sich die Ausführungszeit des Programmes fast.



5.3 Virtuelle Speicher

Wenn einem Programm und seinen Daten zur Ausführungszeit ein Speicherbereich zur Verfügung steht, dessen Kapazität unzureichend ist, so erscheinen in den Befehlen zum Teil absolute Adressen, die in Wirklichkeit nicht existieren. Diese Adressen sind demnach nur scheinbar vorhanden und beziehen sich auf einen scheinbaren Speicher größerer Kapazität. Zur Verfügung steht aber nur der wirklich vorhandene Speicher mit beschränkter Kapazität. Man nennt den scheinbar vorhandenen Speicher virtuellen Speicher und den wirklich vorhandenen Speicher realen Speicher. Dementsprechend spricht man von virtuellen und von realen Adressen.

Stehen nur Teile des Programmes und seiner Daten im realen Speicher (Bild 5.13), so müssen vor oder während der Ausführung des Programmes die virtuellen Adressen auf reale Adressen abgebildet werden. Die Zerteilung des Programmes kann nach logischen Gesichtspunkten erfolgen, z.B. dadurch, daß zusammengehörende Befehlsfolgen zusammengefaßt werden, oder sie kann nach physikalischen Gesichtspunkten

erfolgen, z.B. dadurch, daß die Programmstücke so groß wie der zur Verfügung stehende Speicherbereich gewählt werden.

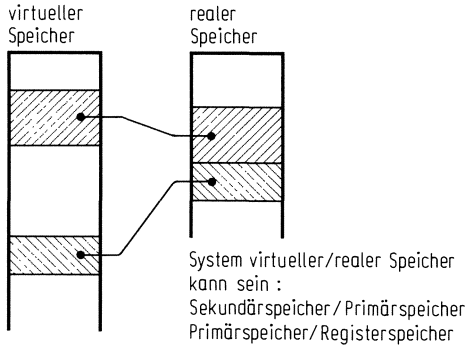


Bild 5.13 Illustration der Abbildung des virtuellen auf den realen Speicher

Zur Organisation des Programmablaufs ist es nötig, die als nächste auszuführenden Programm- bzw. Datensegmente in den realen Speicher zu laden. Dies kann durch Betriebsprogramme erfolgen (dann übernimmt das Betriebssystem die Ein- und Auslagerungen) oder muß im Programm vorgesehen werden (dann übernimmt das Programm diese Lade- und Entladevorgänge). Da die Ausführung eines Programmes ein dynamischer Vorgang ist, läßt sich oft nicht sagen, welche Programmteile mit welchen Daten als nächste ausgeführt werden sollen (z.B. bei Programmverzweigungen). Es wird also nicht in jedem Fall möglich sein, ein statisch segmentiertes Programm vorsorglich so zu laden, daß immer gerade derjenige Ausschnitt im realen Speicher steht, der unmittelbar verarbeitet werden soll.

Die dynamische Adreßabbildung und das dynamische Verwalten der Segmente eines Programmes und seiner Daten kann durch eine Speicherorganisation erreicht werden, die von Programmen oder von Schaltwerken unterstützt wird. Im folgenden werden wir die verschiedenen Möglichkeiten der Adreßabbildung eingehend erörtern, wogegen das Ein- und Auslagern nur exemplarisch in einem Beispiel behandelt wird.

Am einfachsten und wirkungsvollsten ist die Abbildung von virtuellen auf reale Adressen zu erzielen, wenn ein Assoziativspeicher als realer Speicher benutzt wird. Ein Zeiger weist auf die erste freie Speicherzelle, und die virtuellen Befehls- und Operandenadressen werden zusammen mit den korrespondierenden Befehls- und Operandenwörtern direkt in den Assoziativspeicher geladen. Die realen Adressen entsprechen den lokalen Adressen, was besonders deutlich wird, wenn man sich den Assoziativspeicher in ein hierarchisch aufgebautes Speichersystem zerlegt denkt: In Bild 5.14 erkennt man im unteren Speicher den realen Speicher. Er wird von einem Adreßspeicher gesteuert, dessen Adressen länger sind, als es für seine Adressierung notwendig wäre. Aus technischen Gründen ist es nur möglich, Assoziativspeicher kleiner Kapazität als reale Speicher zu verwenden, so daß dieses Konzept nur

für die zweite der in Bild 5.13 angegebenen Möglichkeiten in Frage kommt, nämlich der Pufferung des Primärspeichers durch den Registerspeicher. Dabei spielt der Primärspeicher die Rolle des virtuellen und der Registerspeicher die Rolle des realen Speichers. Wir wollen diese Organisationsform virtueller Speicher als erste behandeln.

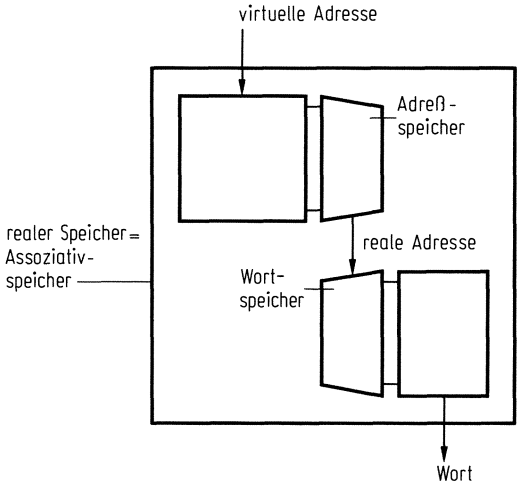


Bild 5.14 Aufteilung eines Assoziativspeichers in Adreß- und Wortspeicher zu Illustration der Abbildung virtueller auf reale Adressen

Pufferorganisation mit Assoziativspeichern und -prozessoren

Im verallgemeinerten Konzept des Von-Neumann-Rechners aus Kapitel 4 hatten wir ein hierarchisch aufgebautes Speichersystem vorgesehen, bestehend aus einem schnelleren Registerspeicher und einem langsameren Primärspeicher. In Beispiel 4.1 wurden unter Zugrundelegung dieses Konzepts die Ausführungszeiten von zwei Programmen verglichen, deren Daten einmal im Registerspeicher und einmal im Primärspeicher vorlagen. Der Vergleich fiel zugunsten des ersten Programmes aus, da im zweiten Programm um Größenordnungen höhere Zeiten für den Zugriff auf die Operanden im Primärspeicher entstanden. Der Unterschied in den Ausführungszeiten beider Programme würde noch krasser ausfallen, wenn außer den Operanden auch die Befehle des ersten Programmes im Registerspeicher stünden. Der Vergleich der beiden Programme entspräche dann dem Vergleich zwischen zwei Rechnern mit jeweils einem schnelleren und einem langsameren Speicher.

Da sich die beiden gegensätzlichen Forderungen nach hoher Kapazität und hoher Geschwindigkeit technisch nicht in einem einzigen Speicher vereinigen lassen, verwendet man aus mehreren Speichern unterschiedlicher Kapazität und Geschwindigkeit aufgebaute Speichersysteme und strebt Organisationsformen an, die die vorteilhaften

Eigenschaften in den Vordergrund und die nachteiligen Eigenschaften in den Hintergrund treten lassen.

Pufferorganisation mit Assoziativspeicher. Bild 5.15 zeigt ein Speichersystem mit einem Assoziativspeicher kleiner Kapazität als Registerspeicher und einem Randomspeicher wesentlich größerer Kapazität als Primärspeicher. Im Primärspeicher befindet sich das Programm zusammen mit seinen Daten und im Registerspeicher der

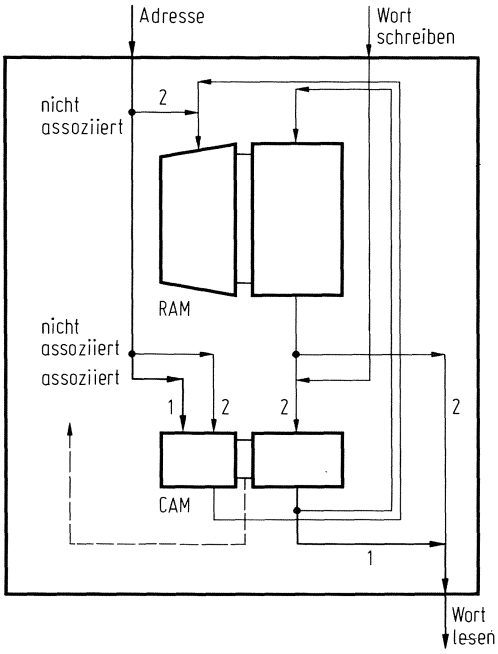


Bild 5.15 Primärspeichersystem, bestehend aus einem Randomspeicher als Primärspeicher und einem Assoziativspeicher als Pufferspeicher

aktuelle Ausschnitt des Programmes und der Daten. Wird das Speichersystem mit einer Befehls- oder Operandenadresse angesprochen, so werden folgende Schritte durchlaufen, denen die in Bild 5.15 eingetragenen Nummern entsprechen:

1. Die Adresse wird mit allen im Assoziativspeicher gehaltenen Adressen verglichen. Bei Übereinstimmung wird das zugehörige Wort aus dem Assoziativspeicher ausgelesen.
2. Bei Nichtübereinstimmung wird die Adresse in den Randomspeicher geleitet, zusammen mit dem aus dem Randomspeicher ausgelesenen Wort in den Assoziativspeicher geschrieben und das ausgelesene Wort an den Ausgang des Speichersystems übertragen.
3. In Schritt 2 muß eine "alte" Eintragung im Assoziativspeicher durch eine "neue" ersetzt werden. Eine konstante Eintragung (Befehl, Konstante) kann unmittelbar über-

schrieben werden, wohingegen eine veränderte variable Eintragung zuvor in den Randomspeicher gebracht werden muß. Dies ist in Bild 5.15 durch die eingezeichnete Doppelleitung angedeutet.

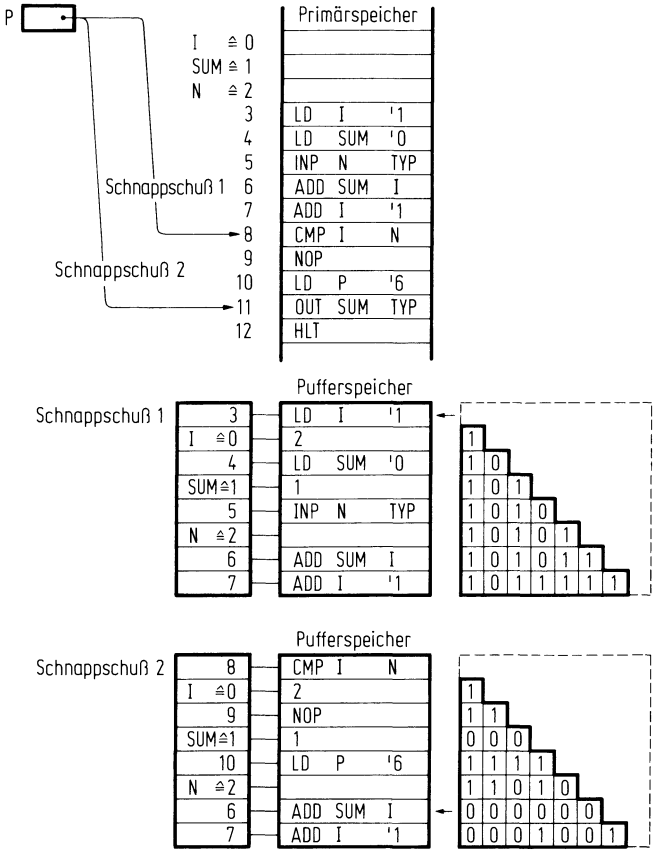
Die Organisation dieses Betriebes wird durch einen Algorithmus festgelegt, der bestimmt, welche alte Eintragung im Assoziativspeicher überschrieben wird. Das kann zum Beispiel diejenige sein, die am längsten nicht ersetzt wurde (first in first out) oder diejenige, die am längsten nicht benutzt wurde (least recently used). Solche "Alterungsmechanismen" werden durch Schaltwerke realisiert, so daß das Speichersystem einem Arbeitsspeicher gleicht, der die große Kapazität des Primärspeichers mit der hohen Geschwindigkeit des Registerspeichers vereinigt.

Die hier beschriebene Form der Pufferorganisation kommt der Maxime der Programmierungstechnik entgegen, die innerste Schleife eines Programmes so schnell wie möglich zu machen: Übersteigt die Anzahl der Befehle und Operanden der Schleife nicht die Kapazität des Registerspeichers, so wird die Schleife nach dem anfänglichen Laden mit der Geschwindigkeit ausgeführt, die der Registerspeicher erlaubt. Das folgende Beispiel soll diese Art der Pufferorganisation illustrieren, wobei der Alterungsmechanismus Least-recently-used (LRU) ausführlicher beschrieben wird.

Beispiel 5.5 Alterungsmechanismen

Wir wählen das in den Beispielen 4.1 und 4.2 benutzte einfache Schleifenprogramm für das Summieren der Zahlen 1 bis N. Im folgenden Schema ist dargestellt, wie dieses Programm im Primärspeicher ab Adresse 0 untergebracht ist. Unter dem Primärspeicher sind zwei Darstellungen des Registerspeichers gezeichnet, die zwei verschiedenen Zeitpunkten des Programmablaufs entsprechen. Der erste "Schnappschuß" zeigt den Registerspeicher, unmittelbar nachdem er gefüllt ist; die erste Zelle ist die, auf die am längsten nicht zugegriffen wurde. Der zweite "Schnappschuß" zeigt den Registerspeicher nach der erstmaligen Ausführung des Sprungbefehls; die siebente Zelle ist die, auf die am längsten nicht zugegriffen wurde.

Rechts neben den Abbildungen des Registerspeichers sind die als Dreiecksmatrix aufgebauten Flipflops des Alterungsmechanismus mit ihren aktuellen Eintragungen gezeigt. Wenn der Pufferspeicher leer ist, sind die Flipflops der Dreiecksmatrix mit Einsen besetzt. Diejenige Zelle des Assoziativspeichers enthält die am längsten nicht benutzte Eintragung, deren Zeile nur Nullen und deren Spalte nur Einsen enthält. Der Inhalt dieser Zelle wird ersetzt, sobald eine neue Eintragung im Assoziativspeicher notwendig wird. Bei jedem Bezug auf den Assoziativspeicher, das entspricht beim Lesen der identifizierten und beim Schreiben der am längsten nicht benutzten Eintragung, werden die in der Alterungsmatrix eingetragenen Bits dadurch auf den neuesten Stand gebracht, daß die entsprechende Zeile auf Eins und die entsprechende Spalte



auf Null gesetzt werden. Diejenige Zelle des Assoziativspeichers enthält also die jüngste Eintragung, deren Zeile nur Einsen und deren Spalte nur Nullen enthält. Wird die jüngste Zelle erneut benutzt, so ändert sich der Stand der Alterungsmatrix nicht. Wird eine andere Zelle benutzt, so werden all jene Zelleneintragungen um Eins gealtert, die jünger sind als die aufgerufene Eintragung. Wird insbesondere die "älteste" Zelle benutzt, so werden alle anderen Eintragungen gealtert.

Für Pufferspeicher, die nur Operanden enthalten, wird vorteilhaft die Least-recently-used Alterung benutzt, da auf Operanden meistens wahlfrei zugegriffen wird. Damit können zum Beispiel die Inhalte einer Vielzahl zuletzt benutzter virtueller Indexregister in einigen wenigen Assoziativregistern gepuffert werden.[1]

Für Pufferspeicher, die nur Befehle enthalten, kann anstelle der relativ aufwendigen Least-recently-used-Alterung die einfachere First-in-first-out-Alterung benutzt werden, bei der die zeitlich erste Eintragung als erste ersetzt wird. Dabei wird die

---

[1] Vgl. assoziative Indexregister der Rechenanlage TR 440 von AEG-TELEFUNKEN.

Tatsache ausgenutzt, daß die Befehle hintereinander gespeichert und mit Ausnahme von Vergleichs- und Sprungbefehlen nacheinander ausgeführt werden. Da Befehle außerdem unveränderlich sind, entfällt das Rückschreiben in den Primärspeicher, und Bild 5.15 vereinfacht sich entsprechend. Der im Primärspeicher gehaltene Befehlsstrom "fließt" durch den Registerspeicher und wird dort unmittelbar verarbeitet. Der Registerspeicher erscheint als Fenster, durch das ein Teil des im Primärspeicher stehenden Programmes ausgeblendet wird. Ist eine Schleife im Puffer untergebracht, so "steht" das Fenster, sonst bewegt es sich und blendet einige wenige Befehle vor und einige mehr Befehle nach dem Befehlszählerstand des Prozessors aus[1]. Vielfach wird dieser Mechanismus gekoppelt mit einem vorausschauenden Befehlstransport in den Puffer, so daß Sprungbefehle unter bestimmten Umständen unmittelbar ausgeführt und damit sozusagen eliminiert werden können. ■

Aus der Sicht des Registerspeichers ist der Primärspeicher in der in Bild 5.15 gezeigten Anordnung ein virtueller Speicher. Der Registerspeicher verfügt über den Adreßraum des virtuellen Speichers, hat aber nur die Kapazität des realen Speichers.

Von der Kapazität hängt in erster Linie die Wirksamkeit des Puffers ab. Übersteigt beispielsweise die Anzahl der zu einer Schleife zählenden Befehle die Kapazität des Puffers, so wird bei den beschriebenen Alterungsmechanismen im ungünstigsten Fall immer derjenige Befehl ersetzt, der als nächster verarbeitet werden soll, und die Geschwindigkeit des Speichersystems sinkt unter die des Primärspeichers. Für Puffer hoher Kapazität werden Assoziativspeicher kaum verwendet, da sie zu teuer sind und den teilweise sequentiellen Strukturen von Programmen und Daten nicht angepaßt sind. Wir werden deshalb im folgenden verschiedene Lösungen für die Pufferorganisation unter Verwendung von Randomspeichern vorstellen: zunächst ein mehr prinzipielles Konzept mit einem Assoziativprozessor und davon ausgehend das Konzept der Segment- und Seitenorganisation, das sowohl bei der Pufferung des Primärspeichers als auch bei der Pufferung des Sekundärspeichers in unterschiedlichen Varianten verwendet wird.

Pufferorganisation mit Randomspeicher und Assoziativprozessor. Als Pufferspeicher wird anstelle des Assoziativspeichers ein Randomspeicher großer Kapazität vorgesehen, dem über ein Additionsschaltnetz ein Assoziativprozessor vorgeschaltet ist (Bild 5.16). Mit diesem Assoziativprozessor kann im Gegensatz zu einem Assoziativspeicher ein Intervallvergleich durchgeführt werden. Dazu enthält er die virtuelle Anfangs- und die virtuelle Endadresse derjenigen Programm- und Datensegmente, die im Puffer stehen. Diese Segmente stellen zusammenhängende Ausschnitte aus den Aktivitätszentren eines Prozesses dar. Den virtuellen Anfangs- und Endadressen eines je-

---

[1] Vgl. Befehlspuffer der Rechenanlage CD 7600 von Control Data.

den Segmentes sind reale Basisadressen zugeordnet. Eine Pufferspeicheradresse - das ist eine reale Adresse aus der Sicht des (in Bild 5.16 nicht gezeichneten) Primärspeichers und gleichzeitig eine virtuelle Adresse aus der Sicht des Pufferspeichers - wird mit den beiden im Assoziativprozessor eingetragenen Segmentadressen verglichen. Liegt sie in dem durch die beiden Segmentadressen definierten Intervall, so befindet sich das dem virtuellen Segment zugeordnete reale Segment im Randomspeicher. Die reale Pufferadresse errechnet sich dabei aus virtueller Pufferadresse minus virtueller Segmentanfangsadresse plus realer Segmentanfangsadresse. Der Vorteil gegenüber der Verwendung eines Randomspeichers mit vorgeschaltetem Assoziativprozessor anstelle eines Assoziativspeichers großer Kapazität liegt darin, daß nur die Anfangs- und Endadressen einiger weniger Segmente gespeichert zu werden brauchen. Demgegenüber stehen die Nachteile, daß der Assoziativprozessor den Vergleich einer Größe zwischen zwei Schranken ausführen können muß und daß die Verwaltung des Puffers erheblich komplizierter wird.

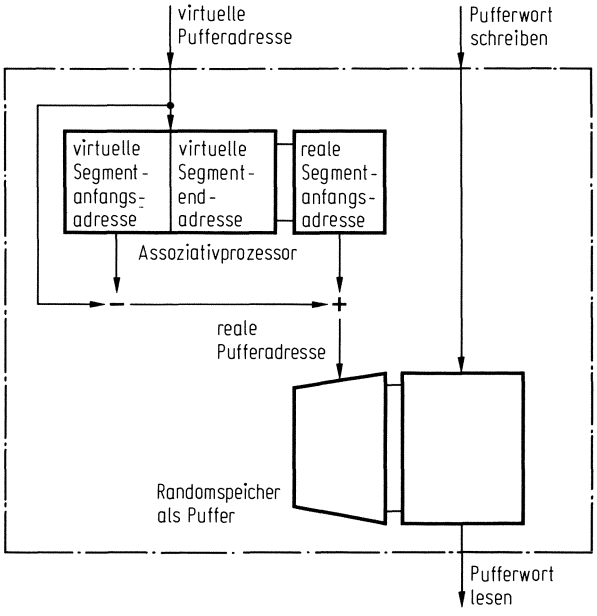


Bild 5.16 Pufferspeichersystem, bestehend aus einem Randomspeicher als Pufferspeicher und vorgeschaltetem Assoziativprozessor zur Adreßabbildung

Segment- und Seitenorganisation zur Pufferung des Primärspeichers

Die Nachteile des eben beschriebenen mehr prinzipiellen Konzeptes können vermieden werden, wenn die Segmente (variable Größe) über eine virtuelle Segmentadresse und eine reale Wortadresse (Distanz) adressiert werden (Bild 5.17). Anstelle des Assoziativprozessors tritt ein Assoziativspeicher, der nur noch die Zuordnungen von

virtuellen zu realen Segmentadressen enthält. Die Pufferadresse wird durch Addition der Wortadresse zur realen Segmentadresse gebildet.

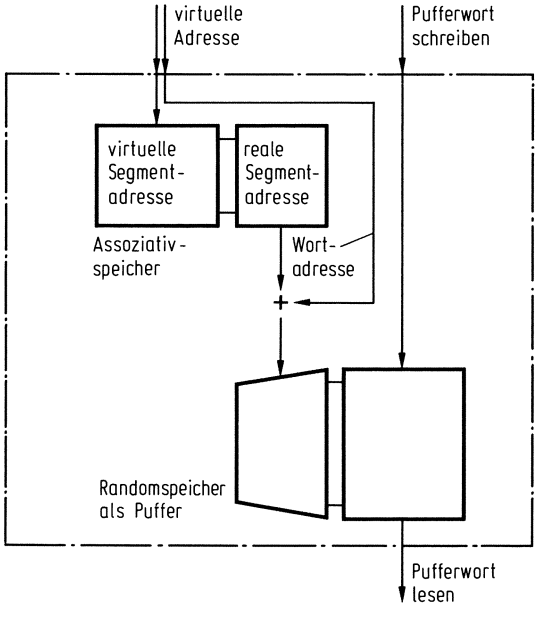


Bild 5.17 Pufferspeichersystem für die Segmentorganisation

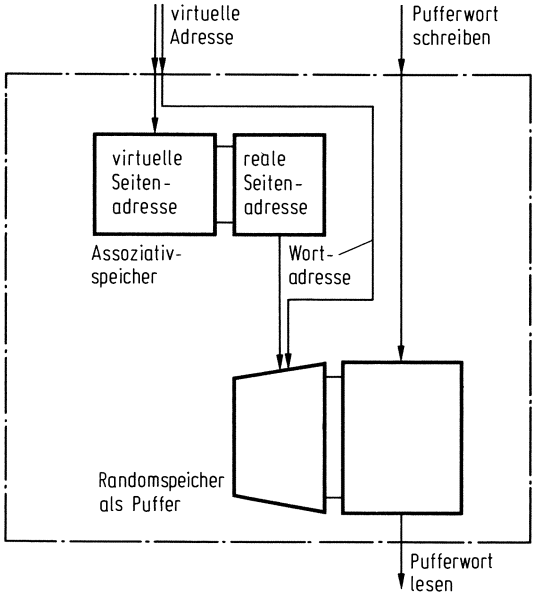


Bild 5.18 Pufferspeichersystem für die Seitenorganisation

Die Nachteile des oben beschriebenen prinzipiellen Konzeptes lassen sich auch dadurch vermeiden, daß man den Primärspeicher und den Pufferspeicher in Seiten (konstante Größe) aufteilt und die Seiten über eine virtuelle Seitenadresse und eine reale Wortadresse (Distanz) adressiert (Bild 5.18). Die Zuordnung von virtuellen zu realen Seitenadressen erfolgt wieder über einen Assoziativspeicher. Die Pufferadresse wird durch Aneinandersetzen von Wortadresse und realer Seitenadresse gebildet. Dieses Konzept zeichnet sich durch Einfachheit in der Struktur und der Organisation aus und ähnelt den in Abschnitt 5.2 behandelten Mischformen von Assoziativ- und Randomspeichern. Ihm liegt die Vorstellung zugrunde, Programme und Daten als Text zu betrachten, der in einem Buch gespeichert ist und sich über viele Seiten erstreckt. Nur die im Moment interessierenden Seiten sind unmittelbar zugänglich und können Zeile für Zeile gelesen werden. Es versteht sich von selbst, daß dieses Zugänglichmachen organisiert werden muß. Das folgende Beispiel soll Vorstellungen über die Größenordnungen solcher Pufferspeicher vermitteln.

Beispiel 5.6 Depot-Speicher (Cache Memory)*

Wir legen einen Primärspeicher mit einer Kapazität von 1024 Seiten und einen Pufferspeicher mit einer Kapazität von 16 Seiten zugrunde. Jede Seite umfasse 1024 Bytes. Zur Adressierung einer Seite im Primärspeicher benötigt man 10 Bits und zur Adressierung einer Seite im Pufferspeicher 4 Bits. Der Assoziativspeicher, der die aus der Sicht des Puffers virtuellen Seitenadressen von 10 Bits Länge auf die realen Seitenadressen von 4 Bits Länge abbildet, besteht entsprechend der Kapazität des Puffers aus 16 Zellen. Bild 5.19 vermittelt einen Eindruck dieser Zusammenhänge, ohne daß damit der wirkliche Aufbau des Systems verbunden sein soll.

Um die Transportzeiten möglichst kurz zu halten, werden 16 Bytes parallel in den Puffer transportiert. Eine Seite mit 1024 Bytes wird also in 64 Taktzeiten vom Primär- in den Pufferspeicher transportiert. Da mit 1024 Bytes als Transporteinheit unter Umständen "zu viel" Information transportiert wird, unterteilt man eine Seite in Transportblöcke von 64 Bytes, die in 4 Zeiteinheiten zu transportieren sind, und transportiert nur diejenigen Blöcke, die benötigt werden. Auf diese Weise erreicht man ein einigermaßen ausgewogenes Verhältnis zwischen Kapazität und Ladezeit der gepufferten Programm- und Datenausschnitte.

Wird eine Seite adressiert, die nicht im Pufferspeicher steht, so wird die am längsten nicht benutzte Seite durch die gerade adressierte Seite in den Primärspeicher verdrängt. Im Pufferspeicher ist diese Seite die zuletzt benutzte, und alle anderen Seiteneintragungen werden um eine "Einheit" gealtert (Least-recently-used-Alterung).

---

* In Anlehnung an den Pufferspeicher der Rechenanlage IBM 360/85.

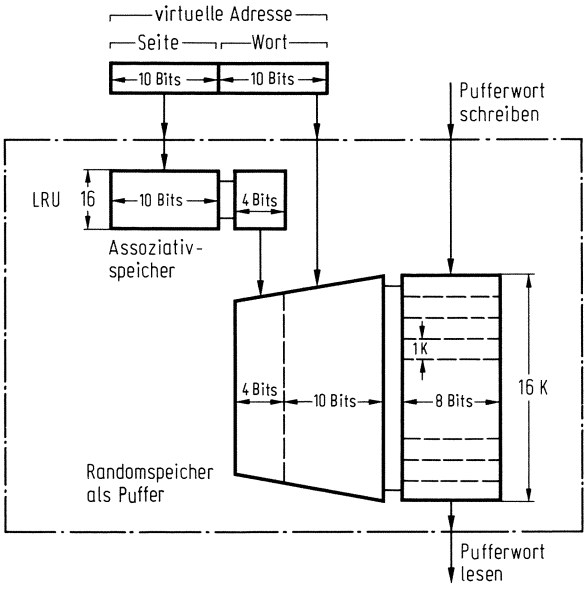


Bild 5.19 Darstellung der Adressierung eines Bytes bei einem Pufferspeichersystem, mit dem 16 von insgesamt 1024 Seiten der Größe 1K Bytes gespeichert werden können ■

Segment- und Seitenorganisation zur Pufferung des Sekundärspeichers

Segment- und Seitenorganisation sind im Prinzip bereits bei der Pufferung des Primärspeichers durch den Registerspeicher beschrieben. Vielfach werden diese Begriffe jedoch erst auf der nächst höheren Stufe der Speicherhierarchie benutzt, nämlich bei der Pufferung des Sekundärspeichers durch den Primärspeicher. Der Unterschied liegt in erster Linie in den höheren Kapazitäten, mit denen man es auf dieser Stufe zu tun hat.

Für die Pufferung des Primärspeichers durch den Registerspeicher ist die charakteristische Lösung die, den Registerspeicher als Assoziativspeicher aufzubauen und die Pufferorganisation in Hardware zu implementieren, d.h. sie auch für den Systemprogrammierer unsichtbar zu machen. Für die Pufferung des Sekundärspeichers durch den Primärspeicher ist die charakteristische Lösung die, den Primärspeicher als Randomspeicher aufzubauen, die Pufferorganisation durch Hardware lediglich zu unterstützen und sie im wesentlichen dem Systemprogrammierer zu übertragen. Algorithmen für die Verwaltung des Randomspeichers, für Segment- bzw. Seitentransporte, für Verdrängungen von Tabelleneintragungen u.s.w. obliegen dem Betriebssystem. Für den Benutzer der Rechenanlage ist auch hier die Pufferorganisation nicht sichtbar. Er schreibt seine Programme, als ob ihm ein Primärspeicher immer ausreichender Kapazität zur Verfügung stünde. Es braucht ihn nicht zu kümmern, daß der Primärspei-

cher als realer Speicher eine begrenzte Kapazität hat und daß der Sekundärspeicher als virtueller Speicher benutzt wird. Für ihn stellt sich das Speichersystem als ein Speicher dar, der die relativ hohe Geschwindigkeit des Primärspeichers mit der relativ großen Kapazität des Sekundärspeichers in sich vereint.

Die Segmentorganisation ist im Prinzip in Bild 5.17 und die Seitenorganisation in Bild 5.18 dargestellt. Im Unterschied zur Pufferung des Primärspeichers werden jedoch bei der Pufferung des Sekundärspeichers nicht sämtliche Zuordnungen der virtuellen zu den realen Segment- bzw. Seitenadressen im Assoziativspeicher gehalten. Das liegt an den höheren Kapazitäten der auf dieser Stufe benutzten Speicher, was eine höhere Kapazität des Assoziativspeichers notwendig machen würde. Stattdessen werden die Adreßzuordnungen im Randomspeicher als Tabellen aufbewahrt, und lediglich ein (nicht notwendigerweise zusammenhängender) Ausschnitt aus der entsprechenden Tabelle befindet sich im Assoziativspeicher. Der Assoziativspeicher arbeitet bei dieser Organisationsform als Puffer des Randomspeichers, genauer als Puffer desjenigen Speicherbereichs, der die Tabelle enthält. Die in Bild 5.15 dargestellte Anordnung vereinfacht sich dabei insoweit, als die Leitungen für das Rückschreiben vom Assoziativspeicher in den Randomspeicher entfallen.

Ein Speichersystem zur Pufferung des Sekundärspeichers arbeitet also wie ein Pufferspeicher ähnlich Bild 5.17 bzw. Bild 5.18, wobei aber die Tabellen für die Abbildung der virtuellen in die realen Segment- bzw. Seitenadressen im Randomspeicher stehen, der seinerseits wiederum entsprechend der in Bild 5.15 dargestellten Pufferorganisation verwaltet wird. Verzichtet man auf die Pufferung der Tabellen, so muß bei jeder virtuellen Adressierung des Randomspeichers vor dem Auslesen des Wortes die entsprechende Tabelle durchsucht werden, was bei sequentieller Tabellenorganisation einer indirekten Adressierung mit Vorindizierung entspricht. Die Bilder 5.20 und 5.21 zeigen die Speicherorganisation für die Segment- bzw. Seitenadressierung zur Pufferung des Sekundärspeichers schematisch, wobei die Leitungen für das Schreiben in den Assoziativspeicher der Einfachheit halber nicht gezeichnet sind. Beide Organisationsformen unterscheiden sich dadurch, daß die Aufteilung des Randomspeichers bei der Segmentadressierung in Bereiche beliebiger Größe erfolgt, während man sich den Randomspeicher bei der Seitenadressierung in gleichgroße Rahmen aufgeteilt denken muß, von denen jeder genau eine Seite aufnehmen kann. Das bedeutet, daß die reale Adresse einmal durch Summieren und einmal durch Aneinandersetzen von Segment- bzw. Seitenadresse und Wortadresse gewonnen wird.

Bei der Segment- und der Seitenadressierung werden nacheinander folgende Schritte durchlaufen, denen die in den Bildern 5.20 und 5.21 eingetragenen Nummern entsprechen:

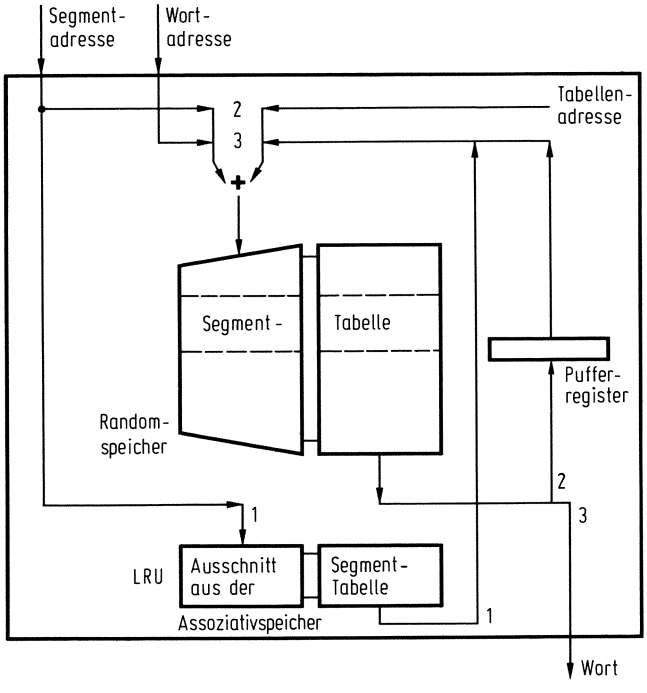


Bild 5.20 Primärspeichersystem für Segmentadressierung (segmenting) mit Pufferung der Segmenttabelle

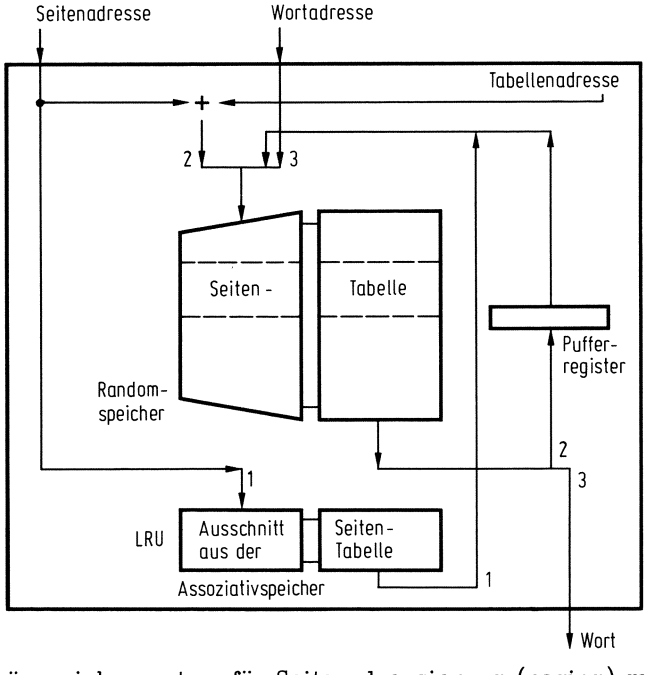


Bild 5.21 Primärspeichersystem für Seitenadressierung (paging) mit Pufferung der Seitentabelle

1. Die von außen angelegte virtuelle Segment- bzw. Seitenadresse wird mit den im Assoziativspeicher eingetragenen virtuellen Segment- bzw. Seitenadressen verglichen; stimmen angelegte und eingetragene virtuelle Adressen überein, so wird die entsprechende reale Segment- bzw. Seitenadresse aus dem Assoziativspeicher entnommen.

2. Stimmen angelegte und eingetragene virtuelle Adresse nicht überein, so wird zur virtuellen Segmentadresse die Adresse der ersten Tabelleneintragung im Randomspeicher addiert und aus dem Randomspeicher die entsprechende reale Segment- bzw. Seitenadresse entnommen. Gleichzeitig wird im Assoziativspeicher die am längsten unbenutzte Tabelleneintragung durch das Paar von virtueller und realer Segment- bzw. Seitenadresse ersetzt (in den Bildern 5.20 und 5.21 nicht gezeichnet).

3. Die Wortadresse wird zu der realen Segmentadresse addiert bzw. mit der realen Seitenadresse zusammengesetzt; mit dieser Adresse wird der Randomspeicher adressiert, und das entsprechende Wort dem Randomspeicher entnommen.

Segment- und Seitenorganisation werden in den verschiedensten Varianten angewendet. Bei der Segmentorganisation ist die Speicherverwaltung wesentlich schwieriger zu bewerkstelligen als bei der Seitenorganisation; Segmentwechsel bedingen wegen der unterschiedlich großen Segmente eines oder mehrerer Programme komplizierte Ersetzungsalgorithmen. Bei der Seitenorganisation ist die Zerstückelung zusammenhängender Bereiche nachteilig gegenüber der Segmentorganisation; die Zerteilung eines oder mehrerer Programme in gleich große Seiten berücksichtigt nicht die Struktur der Programme.

Speicherschutz. Werden von einem Rechner mehrere Programme gleichzeitig verarbeitet, so ist ein Speicherschutzmechanismus erforderlich, der z.B. die von den gerade inaktiven Programmen belegten Primärspeicherbereiche gegen eine Beeinflussung durch das gerade aktive Programm schützt (vgl. Abschnitt 3.3: Teilnehmerbetrieb). Dazu werden für jedes Segment bzw. jede Seite Speicherschutzbedingungen eingeführt, wie z.B.

- uneingeschränkter Schreib- und Lesezugriff;
- kein Schreibzugriff, jedoch Lesezugriff auf Befehle und Operanden;
- kein Schreibzugriff, Lesezugriff nur auf Operanden;
- überhaupt kein Zugriff.

Die Überprüfung der Speicherschutzbedingungen erfolgt bei der Ermittlung der realen Segment- bzw. Seitenadresse. Hierzu werden in der Segment- bzw. Seitentabelle zusätzlich zur realen Segment- bzw. Seitenadresse die entsprechenden Speicherschutzbedingungen angegeben, die dann beim Zugriff auf die Tabelle ausgewertet werden. Bei der Segmentadressierung muß zusätzlich überprüft werden, ob die Wortadresse den Bereich des angesprochenen Segmentes nicht überschreitet. Dazu wird in der Tabelle der

Segmentadressen zu jedem Segment auch dessen Länge oder dessen obere Bereichsadresse angegeben. Der Bereichsschutz erfolgt durch Vergleich der Wortadresse mit den Bereichsgrenzen. Die Verletzung einer Speicherschutzbedingung führt zu einer Programmunterbrechung und Aktivierung des Betriebssystems.

Beispiel 5.7 Segmentierte Seitenorganisation

Eine erweiterte Form der Seitenorganisation ist die Seitenorganisation mit überlagerter Segmentierung, wobei die Segmente eine feste Anzahl an Seiten umfassen.[1] Wir wollen dazu einen virtuellen Speicher von 16 M Bytes betrachten. Der virtuelle Adreßraum ist in 4 K Seiten zu je 4 K Bytes unterteilt, und jeweils 256 Seiten sind

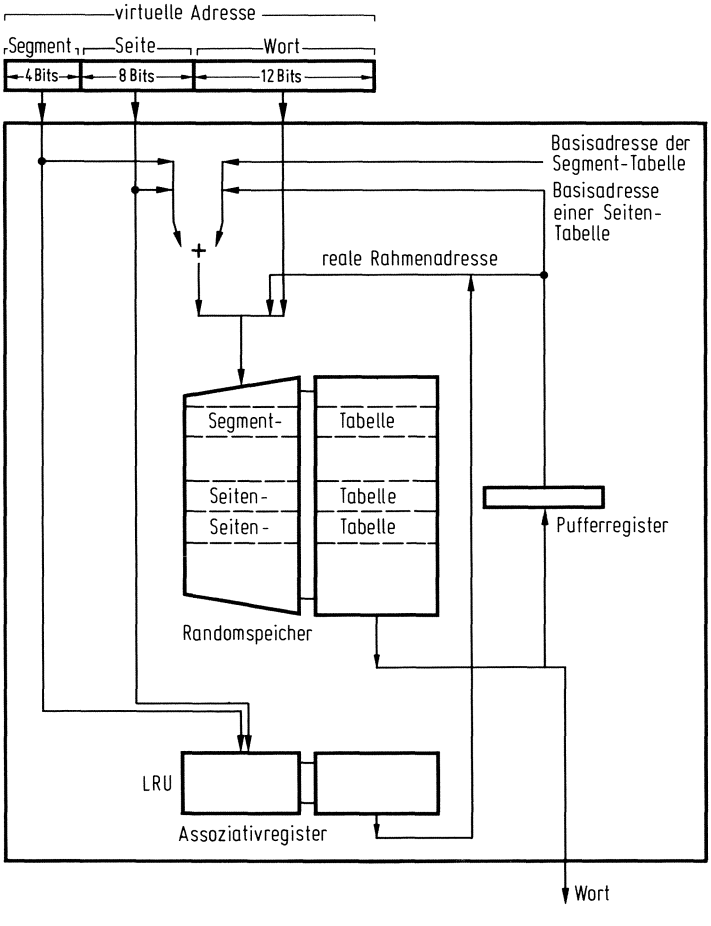
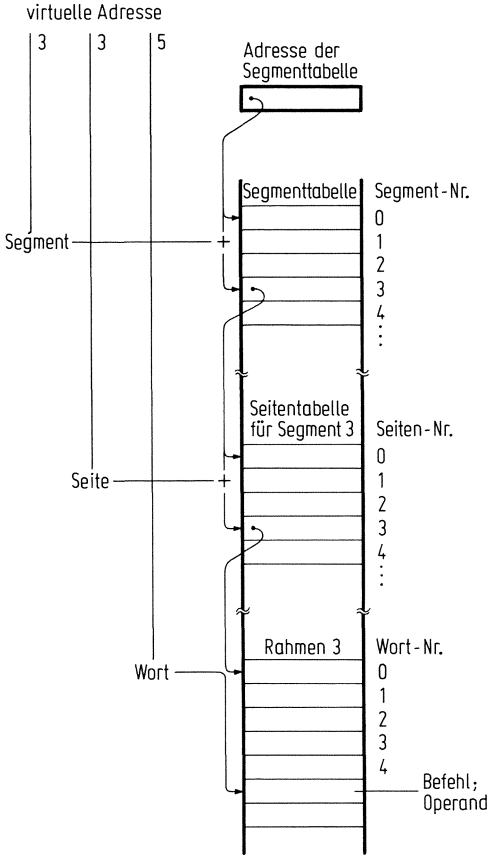


Bild 5.22 Primärspeichersystem für Seitenadressierung mit überlagerter Segmentadressierung

[1] In Anlehnung an die Seitenorganisation der Rechenanlage IBM 360/67.

zu einem Segment zusammengefaßt, so daß der gesamte Adreßraum durch 16 Segmente überdeckt wird. Die Seite soll die kleinste Transporteinheit zwischen Sekundär- und Primärspeicher darstellen. Der Randomspeicher ist als realer Speicher in Rahmen von 4K Bytes aufgeteilt, so daß jeweils eine Seite in einen Rahmen hineinpaßt. Die Adreßabbildung vom virtuellen in den realen Adreßraum zeigen Bild 5.22 und das folgende Schema.



Die virtuelle Adresse besteht aus einer 4-Bit-Segmentadresse, einer 8-Bit-Seitenadresse und einer 12-Bit-Wortadresse (Byteadresse). Über die Anfangsadresse der Segmenttabelle und die virtuelle Segmentadresse erhält man zunächst die Anfangsadresse einer der 16 Seitentabellen. Mit der virtuellen Seitenadresse ergibt sich aus der Seitentabelle die reale Adresse des dieser Seite zugeordneten Rahmens. Eine Zusammensetzung der Rahmenadresse mit der Wortadresse führt schließlich auf die reale Adresse des Wortes im Primärspeicher. Die Organisation des Assoziativspeichers und der im Primärspeicher abgelegten Segment- und Seitentabelle erfolgt in der bei der Seitenorganisation beschriebenen Weise. ■

5.4 Dynamische Speicherverwaltung einzelner Programme durch Hardware

Steht einem Programm ein Speicher zur Verfügung, der dynamisch verwaltet wird, so müssen zur Adressierung der Operanden in den symbolischen Befehlen unserer Assemblersprache numerische Adressen verwendet werden, die zu Basisadressen addiert werden (vgl. Abschnitt 4.3). Für den Programmierer bedeutet dies einen Schritt zurück von der bequemen Assemblerprogrammierung mit frei wählbaren symbolischen Adressen zur unbequemen Maschinenprogrammierung mit fest vorgegebenen numerischen Adressen. Am deutlichsten tritt dieser Rückschritt in Abschnitt 4.4 bei der Ersetzung der Assembleranweisung res durch den Makrobefehl RES hervor, wobei die frei wählbare symbolische Adresse der Assembleranweisung nicht mehr im Makrobefehl erscheint (z.B. RES 100 gegenüber A res 100).

Um diesen unbefriedigenden Zustand zu überwinden, betrachten wir als erstes das Modell eines Rechners, in dem ein Assoziativspeicher als Primärspeicher benutzt wird. Wir führen frei wählbare symbolische Adressen wieder ein, die in codierter Form in den Adreßteil des Assoziativspeichers eingetragen werden. Die dynamische Platzreservierung wird durch einen Maschinenbefehl durchgeführt, der die symbolische Adresse des zu reservierenden Wortes in die erste freie Speicherzelle schreibt und den Zeiger auf die erste freie Speicherzelle um eins erhöht. Der Verwendung eines Assoziativspeichers als Primärspeicher stehen jedoch Nachteile entgegen: Es lassen sich mit einem RES-Befehl nicht mehrere Speicherzellen reservieren (z.B. für Felder), es lassen sich keine indizierten oder relativen Adressen verwenden (z.B. bei Feldern und bei Listen), und es lassen sich Adressen nicht mehrfach bezeichnen (z.B. für Parameter von Prozeduren).

Wir wählen deshalb einen hierarchisch organisierten Speicher als Primärspeicher und untersuchen als zweites das Modell eines Rechners mit einem Randomspeicher, dem ein Assoziativspeicher über ein Addierschaltnetz vorgeschaltet ist und als drittes das Modell eines Rechners mit einem Randomspeicher, dem wiederum ein Randomspeicher über ein Addierschaltnetz vorgeschaltet ist (Bild 5.23). Mit diesen hierarchisch gegliederten Organisationsformen können die oben geschilderten Nachteile vermieden werden. Darüber hinaus zeichnen sie sich durch eine klare räumliche Trennung der verschiedenen gespeicherten Informationen aus: In den oberen Speichern sind nur Adressen gespeichert (Adreßspeicher), zu denen ein Index addiert werden kann, und in den unteren Speichern sind Befehle und Operanden gespeichert (Befehls- und Operandenspeicher). Beim dritten Modell muß als Nachteil gegenüber dem zweiten Modell der Assembler die Umwandlung der symbolischen in numerische Adressen vornehmen. Dazu werden unterschiedliche symbolische Adressen in der Reihenfolge ihres Auftretens durchnumeriert und gleiche symbolische durch gleiche numerische Adressen ersetzt. Dieser Vorgang entspricht der lexikalischen Analyse

bei der Übersetzung höherer Programmiersprachen. Ein weiterer Nachteil des dritten Modells gegenüber dem zweiten Modell ist, daß keine virtuellen Adressen benutzt werden können. Der Vorteil des dritten Modells gegenüber dem zweiten Modell liegt jedoch in der unkomplizierten Struktur des Speichersystems begründet, was sich in einfacherer Organisation und einfacherem Aufbau auswirkt. Aus diesen Gründen verwenden wir das dritte Modell als Grundlage eines Rechners, mit dem einige wichtige Funktionen zur Ausführungszeit eines Programmes (dynamisch) vorgenommen werden, die sonst zur Assemblierzeit des Programmes (statisch) vorgenommen werden. Wir nennen dieses Organisationskonzept schlagwortartig dynamischer Assembler.

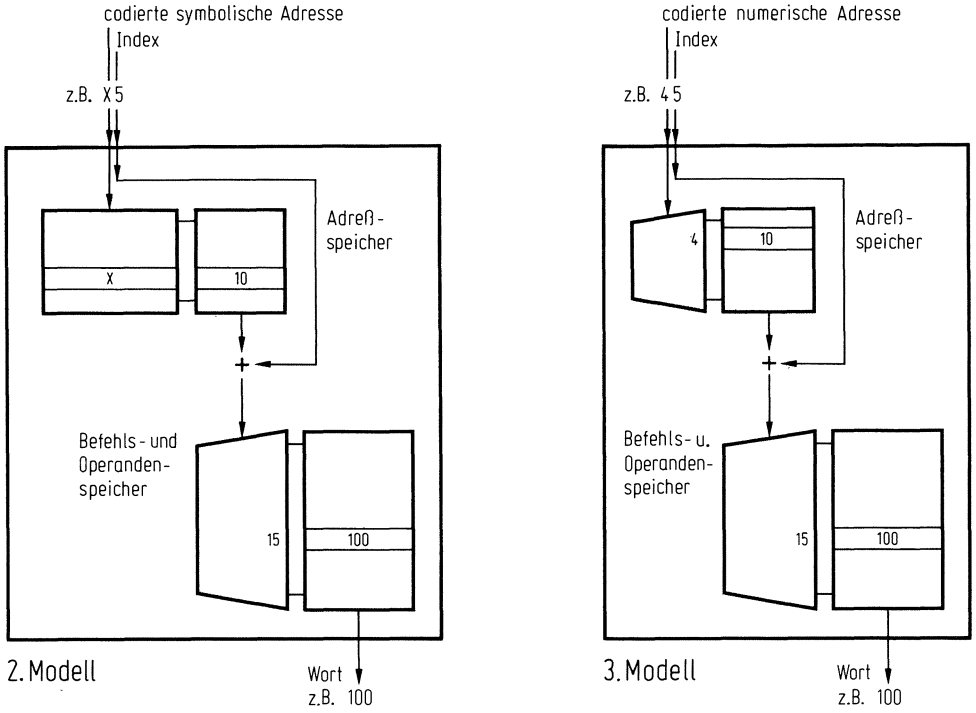


Bild 5.23 Primärspeichersystem für dynamische Speicherverwaltung mit Randomspeicher und vorgeschaltetem Assoziativspeicher (links) bzw. Randomspeicher (rechts)

Speicherverwaltung für Felder variabler Länge

Zur dynamischen Reservierung von Speicherplatz für symbolisch adressierte Felder variabler Länge gehen wir vom dritten Speichermodell aus, das in Bild 5.24 etwas ausführlicher dargestellt ist. Der besseren Anschauung wegen trennen wir die Befehle von den Operanden und führen dementsprechend getrennte Speichereinheiten ein, wie

Operandenspeicher und Befehlsspeicher bzw. Datenadreßspeicher und Programmbasisadreßspeicher.

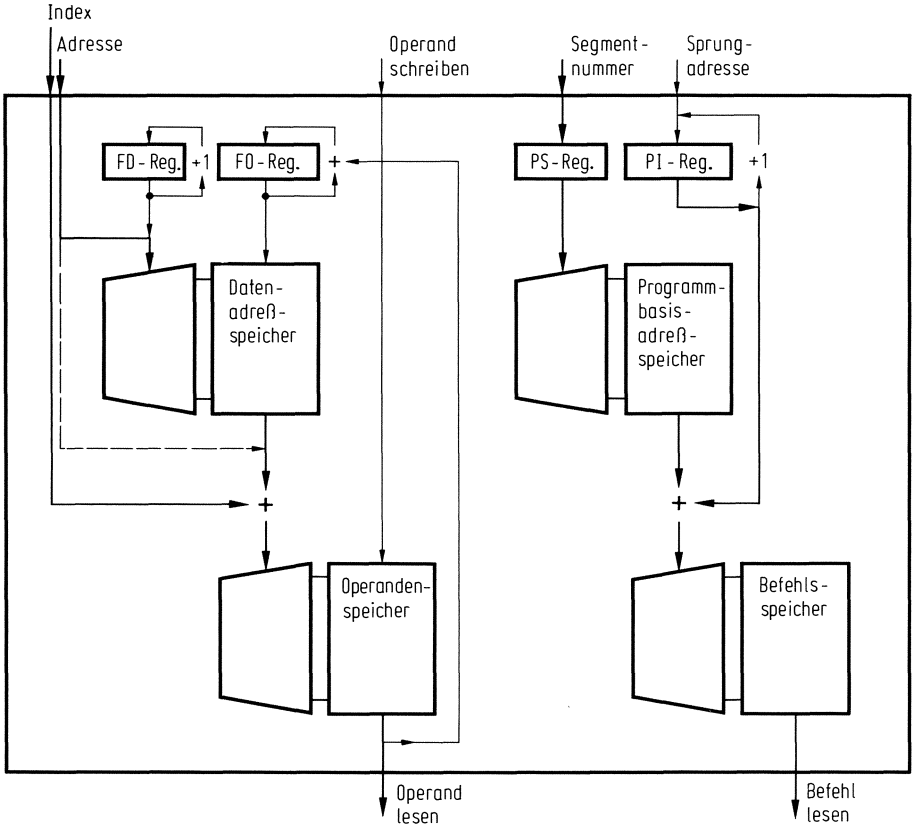


Bild 5.24 Primärspeichersystem zur Speicherverwaltung von Feldern variabler Länge (linke Seite: Datenorganisation, rechte Seite: Programmorganisation)

Datenorganisation. Im Operandenspeicher unterscheiden wir einen statischen und einen dynamischen Speicherbereich und sprechen dementsprechend von statischen bzw. dynamischen Operanden, Variablen, Parametern und Daten.

Der statische Speicherbereich belegt die Zellen mit niedrigen Adressen im Operandenspeicher. Die Adressierung der statischen Operanden erfolgt unter Umgehung des Datenadreßspeichers, was einer direkten Adressierung mit der Möglichkeit der Indizierung entspricht (gestrichelte Adreßleitung in Bild 5.24). Zur Platzreservierung im statischen Speicherbereich wird die res-Anweisung benutzt.

Der dynamische Speicherbereich schließt sich an den statischen Speicherbereich an. Die Adressierung der dynamischen Operanden ist nur über den Datenadreßspeicher möglich, was einer indirekten Adressierung mit der Möglichkeit der Nachindizierung

entspricht (durchgezogene Adreßleitung in Bild 5.24). Zur Verwaltung des Datenadreß- und des Operandenspeichers sind die beiden Register FD und FO nötig. Das FD-Register ist als Zähler ausgeführt und enthält die Adresse des ersten freien Speicherplatzes im Datenadreßspeicher während der Laufzeit eines Programmes. Das FO-Register enthält entsprechend die Adresse des ersten freien Speicherplatzes im Operandenspeicher. Der Inhalt von FO wird jeweils um die Anzahl der im Operandenspeicher belegten Zellen erhöht.

Zur Platzreservierung im dynamischen Speicherbereich sehen wir den Maschinenbefehl RES vor, der im Gegensatz zu dem im Abschnitt 4.4 eingeführten Makrobefehl die Verwendung symbolischer Feldanfangsadressen erlaubt.

RES s N reserviert zur Ausführungszeit ein Feld im Operandenspeicher mit der unter der Adresse N angegebenen Anzahl an Feldelementen. Dazu wird bei seiner Ausführung zunächst der Inhalt des FO-Registers im Datenadreßspeicher unter der im FD-Register gespeicherten Adresse abgelegt. Anschließend wird der Inhalt des FD-Registers um 1 und der des FO-Registers um die durch den Inhalt von N angegebene Feldlänge erhöht.

Die Zuordnung der symbolischen Feldanfangsadresse s zur jeweils belegten Datenadreßspeicherzelle ist durch die fortlaufende Numerierung der Adreßsymbole bei der Assemblierung und die entsprechende Fortschaltung des FD-Registers bei der Ausführung des Programms hergestellt. Das Symbol s ist nur zur Assemblierzeit von Bedeutung und wird zur Festlegung einer numerischen Adresse benutzt, die dann bei den nachfolgenden Befehlen anstelle des Symbols in den Maschinencode eingesetzt wird. Im Maschinencode des RES-Befehls ist diese Adresse nicht vorhanden; zur Ausführungszeit entspricht ihr der Inhalt des FD-Registers. RES ist eine Mischung von Assembleranweisung und Maschinenbefehl. Im Assemblercode erscheint RES als Zweiadreß- und im Maschinencode als Einadreßbefehl. Diese implizit hergestellte Zuordnung der zur Assemblierzeit ermittelten zu den zur Ausführungszeit benutzten Adressen bedingt, daß RES-Befehle während der Programmausführung weder übersprungen noch in Programmschleifen verwendet werden dürfen.

Die Adressen von statischen und dynamischen Operanden werden vom Assembler unterschiedlich ausgewertet und im Maschinencode der Befehlswörter durch ein Bit gekennzeichnet, ähnlich wie in Abschnitt 4.4 die Adressen der dynamischen Operanden durch die Bits für die Basisadressierung gekennzeichnet wurden.

Programmorganisation. Die Organisation des Befehlsspeichers geht von segmentierten Programmen aus. Die Segmente werden aus dem Hauptprogramm und den Prozeduren gebildet. Die Adressierung eines Befehls ist nur über den Programmbasisadreßspeicher möglich, in dem die Programmbasisadressen der Segmente stehen (Bild 5.24).

Dazu wird jedem Programmsegment eine Segmentnummer zugeordnet, die vor der Ausführung des Segments in das Programmsegmentregister PS geladen wird und so den Programmbasisadreßspeicher adressiert. Zur Programmbasisadresse wird ein Programmindex addiert, der im Programmindexregister PI steht. Das PI-Register entspricht dem Befehlszähler; sein Inhalt wird zu Beginn der Ausführung eines Segments auf 0 gesetzt.

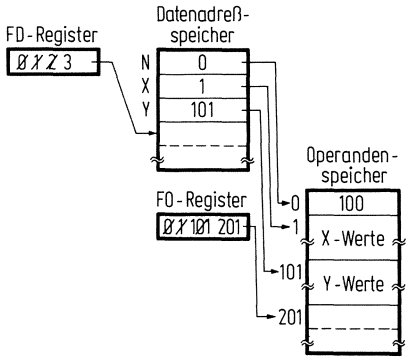
Beispiel 5.8 Vektoraddition mit dynamischer Speicherbelegung

Die dynamische Speicherverwaltung mit RES-Befehlen soll anhand des Programmbeispiels der Vektoraddition veranschaulicht werden. Im folgenden Schema ist im oberen Teil das Assemblerprogramm links mit den symbolischen Adressen N, X und Y und rechts mit den vom Assembler generierten numerischen Adressen 0, 1 und 2 dargestellt. Im unteren Teil wird die Belegung des Datenadreßspeichers und des Operandenspeichers zusammen mit den Inhalten der Register FD und FO gezeigt. Die eingetragenen Werte entsprechen einem "Schnappschuß" nach der Ausführung des Befehls READ Y N (lies soviele Wörter nach Y, wie der Inhalt von N angibt).

Programm:

```
        org
START   RES    N      '1          RES    0      '1
        READ   N      '1          READ   0      '1
        RES    X      N           RES    1      0
        RES    Y      N           RES    2      0
        READ   X      N           READ   1      0
        READ   Y      N           READ   2      0
         :                          :
        LD     I      '0          LD     I      '0
LOOP    ADD    X,I    Y,I         ADD    1,I    2,I
        ADD    I      '1          ADD    I      '1
        CMP    I      N           CMP    I      0
        GOTO   LOOP                GOTO
         :                          :
        end    START
```

Speicherbelegung:



Zu Beginn der Programmausführung sind der Datenadreßspeicher und der Operandenspeicher leer; die Register FD und FO enthalten die numerischen Adressen 0 als Zeiger auf den jeweils ersten freien Speicherplatz in diesen beiden Speichern. Während der Programmausführung wird mit jedem RES-Befehl der Inhalt des Registers FO als Feldanfangsadresse in die durch das FD-Register adressierte Zelle des Datenadreßspeichers gebracht. Der Inhalt des Registers FO wird um den Wert des in RES angegebenen Operanden erhöht, und der Inhalt des Registers FD wird um 1 erhöht. Das Bild zeigt sämtliche Inhalte der Register FD und FO bis zum Zeitpunkt des Schnappschusses. Es wird angenommen, daß für N der Wert 100 als Feldlänge eingelesen wurde, so daß insgesamt 201 Speicherzellen für die Operanden reserviert sind. Der READ-Befehl sei als Makrobefehl definiert und erlaube das Einlesen ganzer Datenfelder. ■

Speicherverwaltung bei Blöcken

Mit der Einführung des RES-Befehls können wir Speicherplatz dynamisch reservieren, ohne daß wir jedoch die Möglichkeit vorgesehen haben, Speicherplatz dynamisch wieder freigeben zu können. Zu diesem Zweck strukturieren wir Programme durch Blöcke in der Weise, daß ein Programm aus einem äußeren Block besteht, in den nebeneinander oder ineinander weitere Blöcke eingebettet sind. Das dynamische Reservieren von Speicherplatz erfolgt durch RES-Befehle innerhalb der Blöcke, und beim Verlassen eines Blockes wird der im Block reservierte Speicherplatz wieder freigegeben. Das bedingt eine Speicherorganisation, bei der beim Eintritt in einen Block die Adresse des ersten freien Speicherplatzes als Blockbasisadresse "eingekellert" und beim Verlassen eines Blockes wieder "ausgekellert" wird. Innerhalb eines Blockes wird die Adresse des ersten freien Speicherplatzes bei jedem RES-Befehl hochgezählt. Die Speicherbelegung pulsiert wie beim Aufruf und Verlassen von Prozeduren.

Durch die Reservierung und Freigabe von Speicherplatz auf der Basis der Blockschachtelung ist gewährleistet, daß immer nur der zuletzt dynamisch reservierte Datenspeicherplatz freigegeben werden kann (Keller). Die gezielte Freigabe reservierter Speicherplätze mitten in einem Datenbereich setzt eine andere Speicherorganisation voraus (Halde).

Für unser Speichermodell bedeutet die Datenverwaltung in Form von Blöcken die Hinzunahme einer weiteren Speicherebene zur Aufnahme der Blockbasisadressen. Mit der Datenblockung ändert sich auch die Datenadressierung. Ein Operand wird jetzt durch ein Adreßpaar (ST, RA) adressiert. Die statische Blockschachtelungstiefe ST, die durch die Blockstruktur des Programms bestimmt wird (vgl. ALGOL 60), wählt die Datenblockbasisadresse aus, zu der die Relativadresse RA des Operanden addiert wird. Das entsprechend modifizierte Speichermodell zeigt Bild 5.25.

Datenorganisation. Das neue Speichermodell Bild 5.25 ist gegenüber dem bisherigen Modell Bild 5.24 durch den Blockbasisadreßspeicher und den FO-Speicher erweitert, die beide über einen gemeinsamen Decodierer adressiert werden. Diesem Decodierer ist ein Zählregister FB vorgeschaltet, dessen Inhalt jeweils auf den ersten freien Speicherplatz der beiden Speicher zeigt.

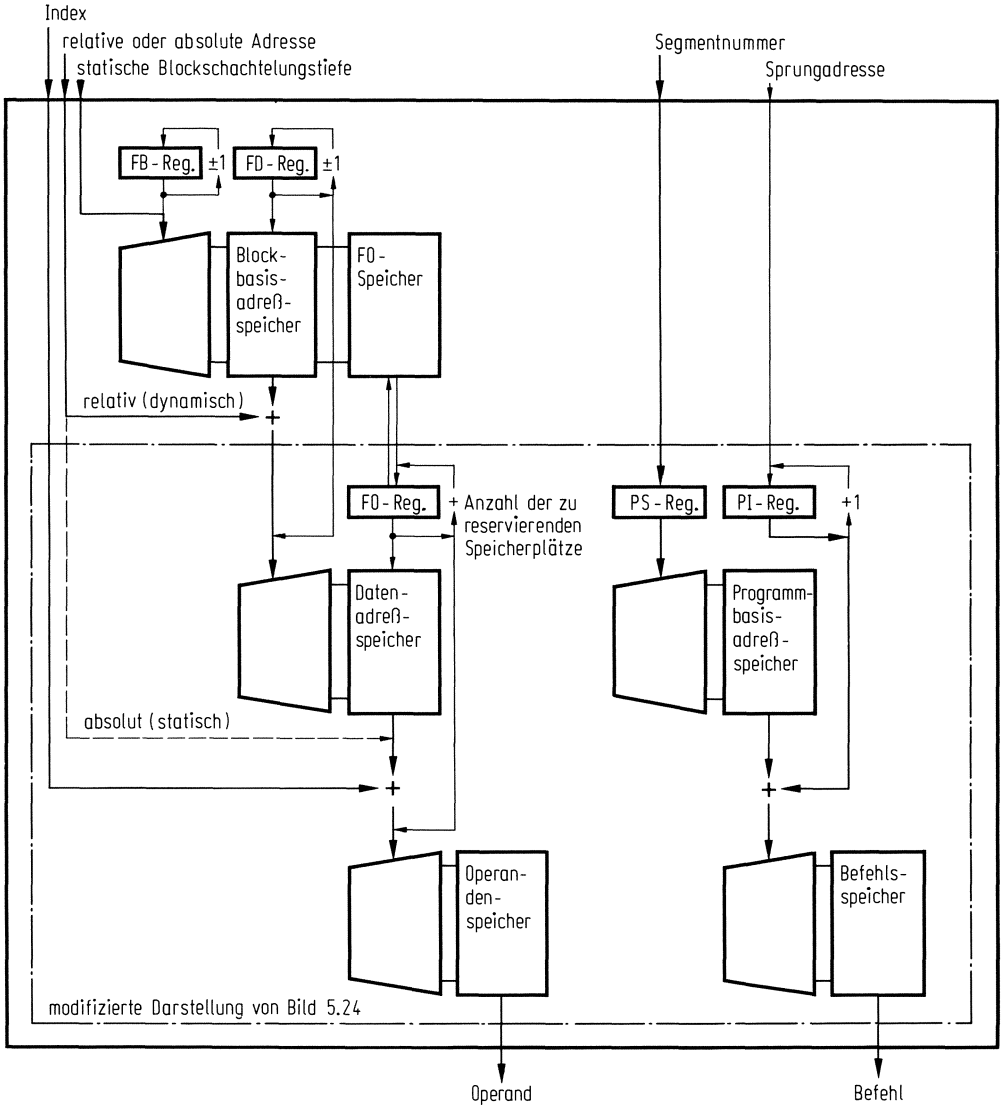


Bild 5.25 Primärspeichersystem zur Speicherverwaltung von Blöcken und Feldern variabler Länge (linke Seite: Datenorganisation, rechte Seite: Programmorganisation)

In Analogie zur Assembleranweisung res können wir auch die Anweisungen org und end "dynamisieren". Wir führen dazu die Maschinenbefehle ORG und END ein und übertragen ihnen die Speicherorganisation des Blockeintritts und des Blockendes.

ORG eröffnet jeweils einen neuen Speicherbereich im Datenadreßspeicher und im Operandenspeicher. Dazu werden die Inhalte der Register FD und FO (Bereichsbasisadressen) im Blockbasisadreßspeicher bzw. im FO-Speicher unter der im FB-Register stehenden Adresse gespeichert. Anschließend wird der Inhalt des FB-Registers um 1 erhöht.

END gibt die durch den zugehörigen Programmblock belegten Speicherbereiche im Datenadreßspeicher und Operandenspeicher frei. Dazu wird zunächst der Inhalt des Registers FB um 1 vermindert. Anschließend werden der Blockbasisadreßspeicher und der FO-Speicher mit der im FB-Register stehenden Adresse adressiert und die dort gespeicherten Bereichsbasisadressen in die Register FD bzw. FO geladen.

Der FO-Speicher wird allein für die Speicherplatzfreigabe im Operandenspeicher benötigt. Der Blockbasisadreßspeicher hingegen wird für die Speicherplatzfreigabe im Datenadreßspeicher und für die laufende Adressierung der Operanden eingesetzt. Während der Programmausführung sind sämtliche Blockbasisadressen über die Adreßleitung für die statische Blockschachtelungstiefe (ST) zugänglich. Daten, deren Blockbasisadresse über die Blockschachtelungstiefe des momentan aktiven Programmblocks adressiert wird, sind dessen lokale Daten; Daten, deren Blockbasisadresse über eine kleinere Blockschachtelungstiefe adressiert wird, sind dessen globale Daten.

Programmorganisation. Die Programmorganisation hat sich in unserem neuen Modell gegenüber dem bisherigen nicht grundsätzlich verändert, so daß sich die Bilder 5.24 und 5.25 diesbezüglich nicht unterscheiden.

Zusätzlich zu ihrer Funktion als Maschinenbefehle werden ORG und END vom Assembler als Hinweise auf die Blockstruktur des Programms zur Erzeugung der relativen Operandenadressen und des Maschinencodes von Sprungbefehlen (GOTO) ausgewertet. In unterschiedlichen Programmblöcken können ohne weiteres gleiche symbolische Sprungmarken verwendet werden. Bei der numerischen Ersetzung einer Sprungmarke im Adreßteil eines GOTO-Befehls sucht der Assembler das Sprungziel zunächst in dem Block, der den GOTO-Befehl enthält, und dann in jenen Blöcken geringerer Schachtelungstiefe, die diesen Block einschließen. Als Sprungziel gilt jene Marke, die in der Blockschachtelung dem GOTO-Befehl am nächsten liegt. Ein Sprungbefehl, dessen Sprungziel in einem Programmblock geringerer Schachtelungstiefe liegt, gibt bei seiner Ausführung ähnlich wie der END-Befehl die entsprechenden Blöcke im Datenadreßspeicher

und im Operandenspeicher frei. Dazu muß der Assembler im Maschinencode des GOTO-Befehls den Betrag $\geqslant 1$ angeben, um den der Inhalt des FB-Registers zu vermindern ist.

Beispiel 5.9 Vektoraddition mit dynamischer Speicherbelegung und -freigabe

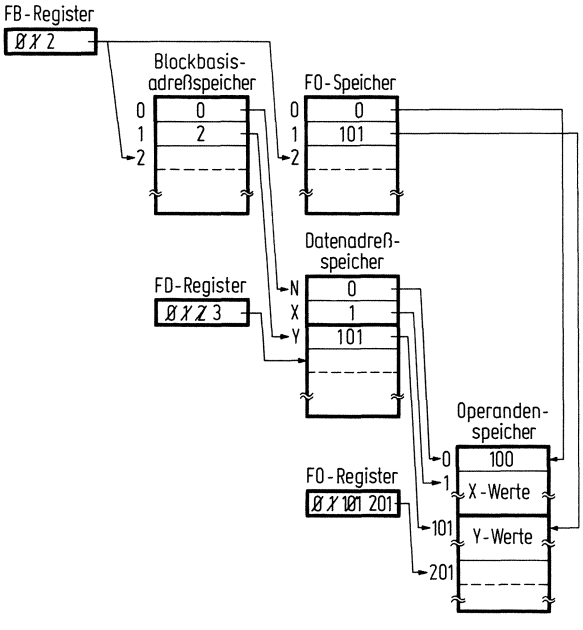
Die Verarbeitung blockstrukturierter Programme mit unserem Speichermodell soll am Beispiel der Addition dreier Vektoren **x**, **y** und **z** nach dem folgenden etwas wirklichkeitsfremden Programm gezeigt werden.[1]

```
org                                  Programm zur Vektoraddition
ORG                                  Beginn des äußeren Blocks
RES     N       '1                   Speicherplatz N reservieren
READ    N       '1                   1 Wort nach N lesen
RES     X       N                    Speicherplatz X reservieren
READ    X       N                    (N) Wörter nach X lesen
        ORG                          Beginn des ersten inneren Blocks
        RES         Y       N        Speicherplatz Y reservieren
        READ        Y       N        (N) Wörter nach Y lesen
        LD          I       '0
L       ADD         X,I     Y,I
        ADD         I       '1
        CMP         I       N
        GOTO        L
        END                          Speicherplatzfreigabe Y
        ORG                          Beginn des zweiten inneren Blocks
        RES         Z       N        Speicherplatz Z reservieren
        READ        Z       N        (N) Wörter nach Z lesen
        LD          I       '0
L       ADD         X,I     Z,I
        ADD         I       '1
        CMP         I       N
        GOTO        L
        END                          Speicherplatzfreigabe Z
:
END                                  Speicherplatzfreigabe N,X,...
end
```

Die Operation **x** := **x**+**y**+**z** wird in die beiden Einzeloperationen **x** := **x**+**y** und **x** := **x**+**z** zerlegt, die nacheinander ausgeführt werden. Die Vektoren **x**, **y** und **z** werden im Speicher als Felder X, Y und Z mit der Länge N dargestellt. X und N werden im äußeren Block der Schachtelungstiefe 0 vereinbart, so daß sie während der gesamten Programmausführung zur Verfügung stehen. Die Ausführung der beiden Einzeloperationen erfolgt in zwei inneren Blöcken der Schachtelungstiefe 1, in denen auch die Speicherbereiche für Y bzw. Z reserviert werden. (Zur Demonstration der unterschiedlichen Blockschachtelungstiefen sind die inneren Blöcke gegen-

[1] In Wirklichkeit würde man die Vektoren **y** und **z** beide ins Feld Y lesen und die Operation **x** := **x**+**y** zweimal ausführen.

über dem äußeren Block etwas eingerückt dargestellt.) Beim Verlassen des ersten inneren Blockes wird mit END der Speicherbereich Y freigegeben, so daß er für den zweiten inneren Block zur Verfügung steht. Hier werden die Y-Werte durch die Z-Werte überschrieben. Nach Verlassen des zweiten inneren Blockes wird auch der Speicherbereich Z wieder freigegeben, so daß dann nur noch die Größen X und N des äußeren Blocks existieren. Durch die Blockstruktur wird also jeweils nur so viel Speicherplatz belegt, wie momentan benötigt wird.



Das obige Schema zeigt die Speicherbelegung und die Inhalte der Register FB, FD und FO nach Einlesen der Y-Werte im ersten inneren Block. Über die statische Blockschachtelungstiefe 1 erfolgt der Zugriff auf die Basisadresse 2 der lokalen Daten Y. Ferner ist im Blockbasisadreßspeicher die Basisadresse 0 der globalen Daten N und Y über die statische Blockschachtelungstiefe 0 zugänglich. Zu einem späteren Zeitpunkt, nach Einlesen der Z-Werte im zweiten inneren Block, würde die Speicherbelegung genauso aussehen, nur wäre Y durch Z ersetzt. ■

Speicherverwaltung bei Prozeduren

Die Verwaltung des dynamischen Speichers beim Durchlaufen eines Blockes ähnelt der Speicherverwaltung bei der Ausführung von Prozeduren. In der in Abschnitt 4.4 geschilderten Organisation wurden beim Aufruf einer Prozedur durch CALL die Datenbasisadresse und der Befehlszählerstand abgespeichert und die modifizierten Pa-

rameteradressen in den dynamischen Speicher übernommen. Innerhalb der Prozedur konnte Speicherplatz dynamisch reserviert werden, der samt dem Speicherplatz für die Parameter und die Statusinformation beim Verlassen der Prozedur durch RETURN wieder freigegeben wurde.

Die Gründe, weshalb die in Abschnitt 4.4 beschriebene Programmorganisation bei geschachtelten und rekursiven Prozeduren etwas unübersichtlich erscheint, liegen in der Verwendung numerischer, mit dem Basisadreßregister modifizierter Adressen, in der Speicherung von Statusinformation und Daten in ein und demselben dynamischen Speicher und in der Verwaltung des freien Speichers im Hinblick auf reentrante Prozeduren (vgl. Abschnitt 6.3).

Durch die Einführung des RES-Befehls können wir in der hier geschilderten Organisation wieder symbolische Adressen verwenden. Darüber hinaus ist Information mit unterschiedlicher Bedeutung auch in unterschiedlichen Speichern abgelegt. Die durch diese Aufteilung entstandene Übersichtlichkeit spiegelt sich in der Struktur unseres Speichermodells Bild 5.25 wider, das wir für die Organisation von Prozeduraufruf und -rückkehr um eine zusätzliche Speicherebene erweitern (Bild 5.26).

Datenorganisation. Das neue Speichermodell Bild 5.26 ist gegenüber dem bisherigen Modell Bild 5.25 durch einen Prozedurbasisadreßspeicher und den FD-Speicher erweitert, die beide über einen gemeinsamen Decodierer adressiert werden. Diesem Decodierer ist ein Zählregister DT (dynamische Prozedurschachtelungstiefe) vorgeschaltet, dessen Inhalt jeweils auf den zuletzt belegten Speicherplatz der beiden Speicher zeigt. Die beiden Speicher werden zur Verwaltung der innerhalb einer Prozedur zusammengefaßten Blöcke eingesetzt, analog zum Blockbasisadreßspeicher und FO-Speicher, die beide zur Verwaltung der zu Blöcken zusammengefaßten Daten dienen.

Beim Aufruf einer Prozedur wird zunächst der Inhalt des DT-Registers um 1 erhöht und dann der Inhalt des FB-Registers, der den ersten freien Speicherplatz im Blockbasisadreßspeicher anzeigt, als Prozedurbasisadresse in der durch das DT-Register adressierten ersten freien Speicherzelle des Prozedurbasisadreßspeichers abgelegt. Gleichzeitig wird der Inhalt des FD-Registers, der auf die erste freie Speicherzelle im Datenadreßspeicher zeigt, in die entsprechende Zelle des FD-Speichers gebracht. Da der Aufruf einer Prozedur gleichzeitig auch als Eintritt in einen neuen Block gilt, laufen zusätzlich die beim vorhergehenden Speichermodell Bild 5.25 beschriebenen Vorgänge des Blockeintritts ab.

Der Prozeduraufruf erfolgt durch den Maschinenbefehl CALL, dessen Verwendung bei der Beschreibung der Parameterübergabe später eingehender gezeigt wird. Das Verlassen einer Prozedur erfolgt durch den Maschinenbefehl RETURN, der den von der

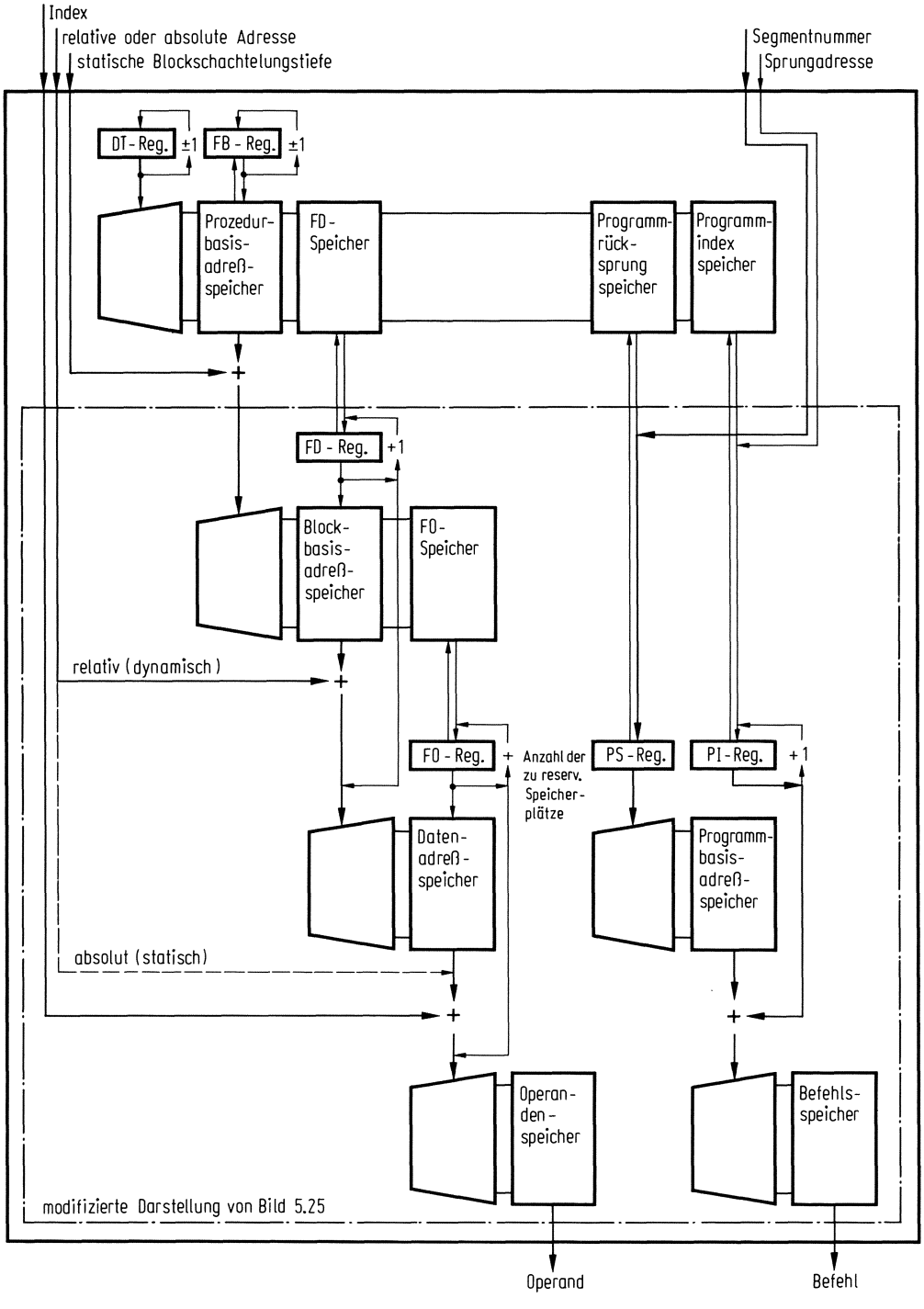


Bild 5.26 Primärspeichersystem zur Speicherverwaltung von Prozeduren, Blöcken und Feldern variabler Länge (linke Seite: Datenorganisation, rechte Seite: Programmorganisation)

Prozedur belegten Datenspeicherplatz freigibt. Dabei wird die durch das DT-Register adressierte Prozedurbasisadresse als neuer Zeiger auf den ersten freien Speicherplatz im Blockbasisadreßspeicher in das FB-Register geladen. Gleichzeitig wird der neue Zeiger auf die erste freie Zelle im Datenadreßspeicher aus dem FD-Speicher in das FD-Register gebracht. Mit dem neuen Inhalt des FB-Registers wird der FO-Speicher adressiert und dessen Inhalt in das FO-Register geladen. Damit weist der neue Zeiger auf den ersten freien Speicherplatz im Operandenspeicher. Der Inhalt des DT-Registers wird um 1 vermindert.

Anmerkung: Die Freigabe des von einer Prozedur im Datenadreßspeicher belegten Speicherplatzes muß nicht unbedingt über den FD-Speicher erfolgen. Man erhält den Zeiger auf den ersten freien Speicherplatz im Datenadreßspeicher, wie im vorangegangenen Modell beschrieben, auch indirekt über den Blockbasisadreßspeicher. Damit entfällt der FD-Speicher in Bild 6.26. Die gleiche Betrachtung, nur eine Speicherebene tiefer, läßt sich für den FO-Speicher in Bild 5.25 anstellen. Auch hier ergibt sich der Zeiger auf den ersten freien Speicherplatz im Operandenspeicher indirekt über den Datenadreßspeicher. Diese Betrachtung für den FO-Speicher gilt jedoch nicht für Bild 5.26, da, wie aus dem nachfolgenden Beispiel 5.10 zu ersehen ist, durch die Parameterübergabe "call by reference" dieser indirekte Bezug verfälscht wird. Der Verzicht auf den FD-Speicher in Bild 5.26 bzw. auf den FO-Speicher in Bild 5.25 äußert sich in einer langsameren Speicherplatzfreigabe, die durch die indirekten Adreßbezüge entsteht.

Die Adressierung der Operanden erfolgt wie bisher mit den Adreßpaaren (ST, RA), wobei ST die statische Blockschachtelungstiefe innerhalb einer Prozedur angibt. Der Gesamtdatenbereich einer Prozedur wird nicht explizit adressiert, sondern implizit durch den momentanen Inhalt des DT-Registers, also dynamisch. Dies setzt voraus, daß eine Prozedur immer über RETURN verlassen wird. Durch das Zählregister DT wird gewährleistet, daß jeder Prozeduraufruf dynamisch eine neue Datenumgebung erzeugt, so auch bei rekursiven Prozeduren.

Um innerhalb einer Prozedur auch Operanden adressieren zu können, die außerhalb ihrer Datenumgebung liegen, sind zwei Möglichkeiten vorgesehen: die Parameterübergabe, auf die wir noch eingehen werden, und die statische Adressierung des Operandenspeichers. In Prozeduren können ebenso wie im Hauptprogramm statische Größen mit der res-Anweisung vereinbart werden und durch die Anweisungen ext und ent auch anderen Prozeduren zugänglich gemacht werden. Die mit ext gekennzeichneten externen Größen nennen wir "global bezüglich der Prozedurgrenze". Größen die innerhalb einer Prozedur vereinbart werden, sind entsprechend "lokal bezüglich der Prozedurgrenze", sie können wiederum "lokal bezüglich der Blockgrenze" oder "global bezüglich der Blockgrenze" sein.

Programmorganisation. Die Programmorganisation erfolgt im wesentlichen wie bisher beschrieben (Bilder 5.24 und 5.25). Zur Verwaltung der Prozedurrücksprungadressen werden jedoch zwei zusätzliche Speicher benötigt: ein Programmrücksprungspeicher und ein Programmindexspeicher (Bild 5.26). Beim Prozeduraufruf werden die Inhalte des PS-Registers (Segmentnummer) und des PI-Registers (Programmindex) in diesen beiden Speichern abgelegt. Sie stellen die Rücksprunginformation dar, mit der die beiden Register bei der Prozedurrückkehr wieder geladen werden. Die Adressierung der beiden Speichereinheiten erfolgt über das DT-Register und ist mit der Adressierung des Prozedurbasisadreßspeichers und des FD-Speichers identisch.

Zusätzlich zu den Befehlen RES, ORG und END führen wir die dynamischen Entsprechungen der Assembleranweisungen dat und equ als Maschinenbefehle DAT und EQU ein:

DAT s|@ N reserviert zur Ausführungszeit eine Zelle im Operandenspeicher und lädt sie mit dem durch die Adresse N bezeichneten Operanden. Dieser kann entweder ein Direktoperand oder der Inhalt einer Zelle des Operandenspeichers sein. Die Reservierung und Belegung der zum neuen Operanden s gehörenden Datenadreßspeicherzelle über die beiden Register FD und FO und die Zuordnung der Adresse s zu dieser Zelle erfolgt analog zum RES-Befehl. Der Inhalt des FO-Registers wird dabei jedoch nur um 1 erhöht. Wird DAT als Parametertransportbefehl benutzt (call by value), so ist s durch das Platzhaltesymbol @ zu ersetzen.

EQU s|@ N reserviert zur Ausführungszeit eine Zelle im Datenadreßspeicher und lädt sie mit der zur symbolischen Adresse N gehörenden numerischen Operandenspeicheradresse. Diese kann entweder ein Direktwert oder der Inhalt einer Datenadreßspeicherzelle sein. Die Reservierung der zum neuen Operanden s gehörenden Datenadreßspeicherzelle und deren Zuordnung zur Adresse s erfolgt analog zum RES-Befehl. Der Inhalt des FO-Registers bleibt jedoch unverändert. Wird EQU als Parametertransportbefehl benutzt (call by reference), so ist s durch das Platzhaltesymbol @ zu ersetzen.

Der DAT- und der EQU-Befehl dürfen wie der RES-Befehl bei der Programmausführung weder übersprungen noch in Programmschleifen verwendet werden. Die zur Ausführung der beiden Befehle notwendigen Datenpfade sind aus Gründen einer überschaubaren Darstellung in Bild 5.26 nicht alle eingezeichnet.

Mit den beiden Befehlen DAT und EQU sind schließlich alle fünf Assembleranweisungen aus Kapitel 3 (res, org, end, dat, equ) als Maschinenbefehle eingeführt, so daß die bisherigen statischen Assemblerfunktionen zusätzlich auch dynamisch ausgeführt werden können.

Parameterübergabe. Der Aufruf einer Prozedur erfolgt mit dem Maschinenbefehl CALL, die Parameterübergabe mit den Maschinenbefehlen DAT und EQU.

```
Prozeduraufruf:  CALL   NAME   Parameterzahl
                 EQU    @      adr1
                 EQU    @      adr2
                 DAT    @      adr3
                  :
```

Mit dem Befehl CALL wird der Aufruf eingeleitet. NAME bezeichnet den Prozedurnamen; er wird im Maschinencode durch die Segmentnummer ersetzt. "Parameterzahl" gibt die Anzahl der auf CALL folgenden Parametertransportbefehle an.

Durch CALL werden zunächst - wie bei der Daten- und Programmorganisation beschrieben - die neuen Datenbasisadressen (die Inhalte der Register FB, FD und FO) und die Segmentnummer des aufrufenden Programms (der Inhalt des Registers PS) unter der neuen dynamischen Schachtelungstiefe gerettet. Ist die Parameterzahl größer als Null, so wird - in Abänderung der bisher beschriebenen Organisation - die dynamische Schachtelungstiefe des aufrufenden Programmes wiederhergestellt. Danach werden die Parametertransportbefehle ausgeführt. Während der Parameterübergabe, die noch unter der Steuerung des aufrufenden Programmes erfolgt, sind mit dessen Datenumgebung auch die aktuellen Parameter zugänglich. Die transportierten Parameter belegen jedoch Zellen, die nach abgeschlossenem Prozeduraufruf zur lokalen Datenumgebung der aufgerufenen Prozedur gehören.

Durch EQU @ adr1 wird die durch adr1 im Datenadreßspeicher adressierte Operandenadresse als Parameter in die erste freie Speicherzelle des Datenadreßspeichers gebracht (call by reference). Durch DAT @ adr3 wird der durch adr3 im Operandenspeicher adressierte Operand als Parameter in die erste freie Speicherzelle des Operandenspeichers gebracht (call by value). Bei beiden Parametertransportbefehlen ist die symbolische Zieladresse durch das Platzhaltesymbol @ ersetzt, da sie zum Adressierungsbereich der aufgerufenen Prozedur gehört und deshalb auch dort vereinbart werden muß.

Nach Abschluß des Parametertransports wird die dynamische Schachtelungstiefe um 1 erhöht. Damit sind der Datenadreßblock und der Operandenblock der aufgerufenen Prozedur aktiviert und die dort stehenden Parameter adressierbar. Jetzt wird auch der Inhalt des PI-Registers als Rücksprunginformation in den Programmindexspeicher gebracht. Der eigentliche Sprung zur Prozedur erfolgt, indem das PS-Register mit der neuen Segmentnummer geladen und das PI-Register auf Null gesetzt wird. Diese abschließenden Ausführungsschritte sind Bestandteil des CALL-Befehls, der durch den Parametertransport in seiner Ausführung unterbrochen wird.

Die Vereinbarung der formalen Parameter in der Prozedur wird durch die Assembleranweisung proc mit dem Namen der Prozedur im Namensfeld und den formalen Parametern im Adreßfeld vorgenommen.

```
Prozedurdefinition:  NAME  proc     par1  par2  par3 ...
                            :
                           RETURN
```

Der Assembler ordnet den formalen Parametern par1, par2, par3, ... die numerischen Relativadressen 0, 1, 2, ... zu, die der Speicherplatzzuordnung bei der Parameterübergabe entsprechen.

Beispiel 5.10 Fakultät

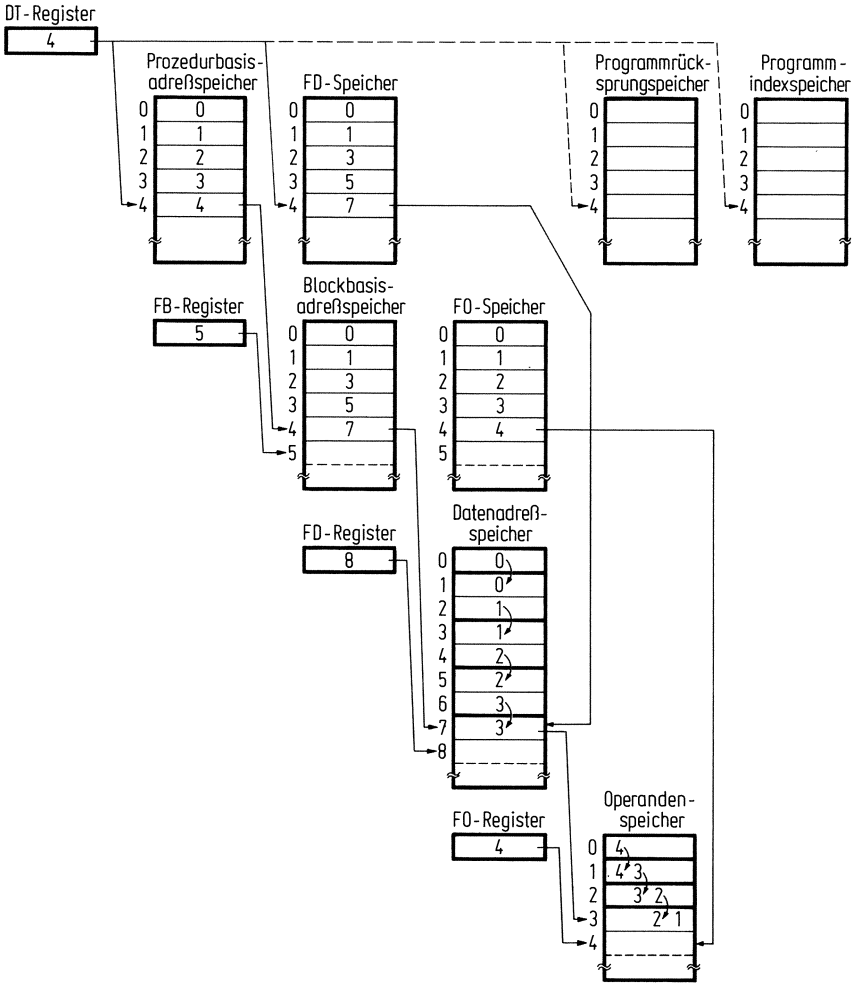
Am Beispiel der rekursiven Prozedur FAK soll die Speicherverwaltung bei rekursiven Prozeduraufrufen gezeigt werden. FAK berechnet die Fakultät einer Zahl X, für die im Hauptprogramm ein Speicherplatz dynamisch reserviert wird. Die Adresse der Zahl X wird beim Aufruf von FAK als Parameter übergeben und dient auch zur Speicherung des Resultats (vgl. Beispiel 4.16).

```
        org                 Hauptprogramm
        ORG
START   RES     X     '1
         :
        CALL    FAK   1     Prozeduraufruf von FAK
        EQU     @     X     Parameterübergabe (call by reference)
         :
        END
        end     START

        org                 Vereinbarung der Prozedur FAK
FAK     proc    P1          Vereinbarung des formalen Parameters P1
        CMP     P1    '1
        NOP
        GOTO    FIN
        DAT     X     P1
        SUB     X     '1
        CALL    FAK   1     Wiederaufruf von FAK
        EQU     @     X     Parameterübergabe (call by reference)
        MUL     P1    X
FIN     RETURN
        end
```

Das folgende Schema zeigt einen Schnappschuß der Speicherbelegung nach der Parameterübergabe des vierten Aufrufs von FAK. Bei jedem Aufruf wurde im Datenadreßspeicher und im Operandenspeicher ein neuer lokaler Speicherbereich reserviert; gleichzeitg wurde die dynamische Schachtelungstiefe jeweils um 1 erhöht. Die statische Schachtelungstiefe blieb mit 0 unverändert, da keine ORG-END-Blöcke vorkommen. Daraus resultiert die fortlaufende Numerierung der Adressen im Prozedur-

basisadreßspeicher, d.h. daß immer nur der zuletzt generierte Datenblock dynamisch adressierbar ist. Im gezeigten Schnappschuß ist dies im Operandenspeicher die Zelle 3. Die Pfeile im Datenadreßspeicher geben die Parameterübergabe durch EQU an, die Pfeile im Operandenspeicher bezeichnen den Operandentransport durch DAT. (Anstatt "DAT X P1" könnte auch die Befehlsfolge "RES X '1, LD X P1" stehen.)



Mit der Entflechtung der rekursiven Aufrufe über den Rücksprungbefehl RETURN werden die Speicherbereiche im Datenadreßspeicher und Operandenspeicher nach und nach wieder freigegeben. Die aus der Multiplikation resultierenden Zwischenergebnisse entstehen in der durch den Parameter P1 angesprochenen Operandenspeicherzelle des jeweils aufrufenden Programmes. ■

6. Ein- und Ausgabeorganisation

Die Ein- und Ausgabeorganisation umfaßt die Kommunikation des Rechners mit seiner Peripherie. Dazu gehören nicht nur die typischen Ein-Ausgabegeräte wie Schreibmaschine oder Lochkartenleser und -stanzer, sondern genauso die peripheren Speicher wie Magnetplatten- oder Magnetbandspeicher, die gewissermaßen nur eine Erweiterung der Speicherkapazität der Rechenanlage darstellen. Diese Art der Peripherie ist typisch für wissenschaftliche und kaufmännische Rechenanlagen, und man spricht von kommerzieller Peripherie.

Zur Kommunikation des Rechners mit seiner Peripherie gehören auch die Ein-Ausgabevorgänge zur Steuerung und Regelung von Prozessen. Dabei kann der Rechner seine übergeordnete Funktion verlieren und als eine von vielen Funktionseinheiten arbeiten. Im Gegensatz zu anderen Einheiten eines solchen Systems erhält er seine besondere Funktion erst durch seine Programmierung. Vom Rechner aus gesehen zählen freilich noch immer alle Anlagen um ihn herum zur Peripherie, und man spricht von Prozeßperipherie.

Sind mehrere Rechner zu einem Rechnernetz zusammengeschlossen, so erfolgt die Kommunikation zwischen den Rechnern ebenfalls in Form von Ein-Ausgabevorgängen. Auch hier kann einem einzelnen Rechner nicht mehr unbedingt eine zentrale Bedeutung zugeordnet werden. Vielmehr haben bestimmte Rechner des Netzes besondere Eigenschaften, die sie für die Bearbeitung spezieller Probleme geeignet machen. Über die Peripherie des Netzes sind sie einem jeden Benutzer zugänglich.

Für die Organisation der Kommunikation des Rechners mit seiner Peripherie oder mit anderen Rechnern ist kennzeichnend, daß er - als Funktionseinheit betrachtet - mit vielen anderen Funktionseinheiten sehr unterschiedlicher Geschwindigkeit zusammenarbeitet, die gleichzeitig in Betrieb sind. Für die Hardwaretechnik stehen die Probleme der Signalübertragung und der Zusammenschaltung der einzelnen Einheiten im Vordergrund, und für die Softwaretechnik spielen die Probleme des Wartens und der Synchronisation der auf den Einheiten parallel laufenden Prozesse eine dominierende Rolle. Dabei ist es von untergeordneter Bedeutung, ob diese "Prozessoren" durch Software veränderbar oder durch Hardware unveränderbar programmiert sind.

Selbst das einfachste Ein-Ausgabegerät kann als Prozessor betrachtet werden, und dementsprechend werden bereits die einfachsten Ein-Ausgabevorgänge, wie z.B. die Übertragung eines Bytes vom oder zum Rechner, durch Warteschleifen synchronisiert, die in die Ein-Ausgabeprogramme eingefügt werden.

In Abschnitt 6.1 wird die Rechnerperipherie, nach Struktur und Funktion gegliedert, zunächst mehr nach grundsätzlichen Gesichtspunkten behandelt. Anschließend werden kurz typische Ein- und Ausgabegeräte beschrieben, wodurch ein Bezug zu Abschnitt 5.1 entsteht. Eine formale Gleichbehandlung der Ein-Ausgabegeräte und der peripheren Speicher ergibt sich dadurch, daß einerseits Ein-Ausgabegeräte gewisse Eigenschaften von Speichern haben und andererseits periphere Speicher wie Ein-Ausgabegeräte angeschlossen sind. Der Unterschied ist in erster Linie darin zu sehen, daß die peripheren Speicher ein Teil der Speicherhierarchie des Rechners sind, wogegen die Ein-Ausgabegeräte die Verbindung der Anlage mit dem Benutzer darstellen.

In Abschnitt 6.2 werden modulare Konfigurationen beim Aufbau von Rechenanlagen aus Speichern, Prozessoren und Ein-Ausgabegeräten behandelt. Die allgemeinste Struktur der Zusammenschaltung von Moduln mit n unterschiedlichen Funktionen ergibt sich, wenn alle Moduln gleicher Funktion mit jeweils allen anderen Moduln gleicher Funktion über Schalter verbunden werden. Auf diese Weise entsteht ein n-dimensionales Gitter von Leitungen, in dessen Kreuzungspunkten Schalter eingebaut sind. Es versteht sich von selbst, daß dieser Aufbau nicht nur vom Aufwand her, sondern auch von der Organisation her schnell an Grenzen stößt und man sich stattdessen bei größeren Systemen hierarchischer Strukturen bedient.

In Abschnitt 6.3 werden Programmunterbrechungen zur Bearbeitung von mehreren in sich abgeschlossenen Aufgaben oder Teilaufgaben eingeführt. Damit kann die Wirkung mehrerer parallel arbeitender Prozessoren durch einen einzigen sequentiell arbeitenden Prozessor simuliert werden. Aufgaben, die von Natur aus besser durch parallel als durch seriell auszuführende Handlungen beschrieben werden können, werden aus wirtschaftlichen Gründen meistens nur von einem einzigen Prozessor seriell bearbeitet. Die Programmierung solcher Aufgaben muß dann so erfolgen, als ob die auszuführenden Programme von mehreren Prozessoren parallel verarbeitet würden, obwohl nur ein einziger Prozessor zur Verfügung steht. Deshalb ist es angebracht, nicht nur bei Programmen mit eigenen Prozessoren, sondern auch bei Programmen, die aufgrund von Programmunterbrechungen ablaufen, von Prozessen zu sprechen.

In Abschnitt 6.4 werden zunächst Ein-Ausgabekanäle behandelt. Prinzipiell können Ein-Ausgabevorgänge vom Prozessor des Rechners selbst vorgenommen werden, was aber wegen des großen Geschwindigkeitsunterschieds zu den peripheren Geräten zu langen Wartezeiten des Prozessors führt. Um zu einer besseren Ausnutzung der Anlage

zu kommen, verlagert man die Steuerung der Ein-Ausgabevorgänge vom Zentralprozessor auf Ein-Ausgabekanäle oder Ein-Ausgabeprozessoren. Während bei Ein-Ausgabekanälen die Steuerung der Ein-Ausgabevorgänge ein für allemal fest programmiert ist (Hardware) kann sie bei Ein-Ausgabeprozessoren von Fall zu Fall ausgewechselt werden (Software). Ein-Ausgabekanäle sind dementsprechend Ein-Ausgabeprozessoren mit unveränderlichem Programm, und Rechenanlagen sowohl mit Ein-Ausgabekanälen als auch Ein-Ausgabeprozessoren können als Mehrprozessorsysteme aufgefaßt werden. Für diese Systeme ist kennzeichnend, daß sie einen Primärspeicher gemeinsam benutzen.

Als Abschluß der Ein-Ausgabeorganisation wird schließlich Mehrprogrammbetrieb kurz behandelt. Dabei entfernen wir uns von den Details, die in den Abschnitten 6.3 und 6.4 in diesem Zusammenhang beschrieben worden sind. Dieser Stoff geht über den Rahmen des Buches hinaus und gehört bereits in das Gebiet der Betriebssysteme. Mehrprogrammbetrieb ist notwendig für effizient arbeitende Stapelbetriebssysteme genau so wie für Teilnehmer- oder Realzeitbetriebssysteme. Dabei dominieren die Softwareprobleme; bestimmte Hardwarevoraussetzungen sind jedoch unentbehrlich.

6.1 Rechnerperipherie und Ein-Ausgabegeräte

Die Umgebung eines Rechners wird als Rechnerperipherie bezeichnet. Peripheriegeräte haben wir bereits bei den peripheren Speichern kennengelernt, die in Abschnitt 5.1 beschrieben sind. Periphere Speicher (z.B. Magnetbandgeräte) und Ein- und Ausgabegeräte (z.B. Lochkartenleser und -stanzer) lassen sich begrifflich nicht scharf voneinander abgrenzen. Ein Magnetband speichert binäre Information durch unterschiedliche Magnetisierungen und Lochkarten durch gestanzte oder nicht gestanzte Löcher. Ein Lochkartengerät kann als peripherer Speicher angesehen werden, da Lochkarten wie Magnetbänder vom Rechner beschrieben und gelesen werden. Andererseits kann auch ein Magnetbandgerät als Ein- und Ausgabegerät bezeichnet werden, da die auf dem Band codierten Zeichen wie beim Lochkartengerät oder bei einer elektrischen Schreibmaschine in den Akkumulator oder den Primärspeicher des Rechners übertragen werden.

Die Ein-Ausgabeorganisation so verschiedener Peripheriegeräte wie Magnetbandgeräte und elektrische Schreibmaschinen ist hinsichtlich Struktur und Funktion von vielen Gemeinsamkeiten geprägt, so daß sich ihre Beschreibung in ein und demselben Kapitel rechtfertigt. Selbst wenn die Umgebung eines Rechners aus Prozeßperipherie (wie z.B. in einem Walzwerk) oder wieder aus Rechnern besteht (wie z.B. in einem Rechnerverbundsystem), ergeben sich hinsichtlich Struktur und Funktion der Ein-Aus-

gabeorganisation keine wesentlich neuen Gesichtspunkte. Freilich ist der Betrieb eines Rechners in derart unterschiedlichen Systemen entsprechend vielgestaltig, was in den vielfältigen Betriebsarten der verschiedenen Betriebssysteme zum Ausdruck kommt.

Prinzipien des Anschlusses peripherer Geräte

Zu den beiden wesentlichen Moduln eines Rechners, dem Prozessormodul und dem Speichermodul, kommt als dritter der Ein-Ausgabemodul hinzu. Erst damit ist die Möglichkeit gegeben, daß der Rechner mit seiner Peripherie kommunizieren kann.

Wir verstehen im folgenden unter Prozessormodul das Leit- und Rechenwerk eines Rechners, unter Speichermodul den gesamten Primärspeicher des Rechners und unter Ein-Ausgabemodul die Gesamtheit der Peripheriegeräte. Bild 6.1 zeigt die beiden grundsätzlichen Möglichkeiten der Zusammenschaltung dieser Moduln durch zwei

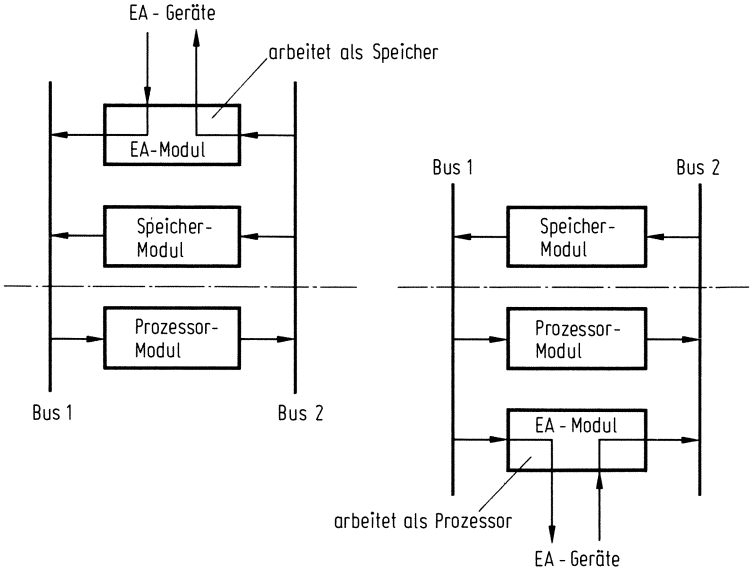


Bild 6.1 Illustration der unterschiedlichen Betriebsweisen des Ein-Ausgabemoduls

Busse. Die links dargestellte Kombination entspricht der Organisation des in Kapitel 2 beschriebenen Einadreßrechners, während die rechts dargestellte Kombination dem Konzept des Zweiadreßrechners aus Kapitel 4 folgt. In der linken Darstellung arbeitet der Ein-Ausgabemodul in derselben "Richtung" wie der Speichermodul. Die Übertragung eines Wortes aus der Peripherie erfolgt über den Prozessor, im einfachsten Fall über den Akkumulator. Dies ist dann günstig, wenn das übertragene Wort sofort verarbeitet werden soll, z.B. beim Analysieren eines Textes. Wird der Ein-Ausgabemodul

im weitesten Sinne als Speicher betrachtet, so haben wir eine Konfiguration vor uns, bestehend aus einem Prozessormodul und zwei Speichermoduln, die vom Prozessor im Multiplexbetrieb bedient werden.

In der rechten Darstellung arbeitet der Ein-Ausgabemodul in der "Richtung" des Prozessormoduls. Die Übertragung eines Wortes aus der Peripherie erfolgt in den Speicher. Dies ist dann günstig, wenn das übertragene Wort gespeichert werden soll, bevor es verarbeitet wird, z.B. bei der Übertragung der Elemente eines Feldes. Wird der Ein-Ausgabemodul im weitesten Sinne als Prozessor betrachtet, so haben wir eine Konfiguration vor uns, bestehend aus zwei Prozessormoduln und einem Speichermodul, der von den beiden Prozessoren im Multiplexbetrieb benutzt wird.

Nimmt man zu den INP- und OUT-Befehlen für Byteleitungen des verallgemeinerten Von-Neumann-Rechners noch die im folgenden beschriebenen CNT- und TST-Befehle für Bitleitungen hinzu, so wird die in Bild 4.1 gezeigte Ein-Ausgabe-Einheit (IOU) komplizierter (Bild 6.2). Sie besteht dann aus je einem Multiplexer-Demultiplexer

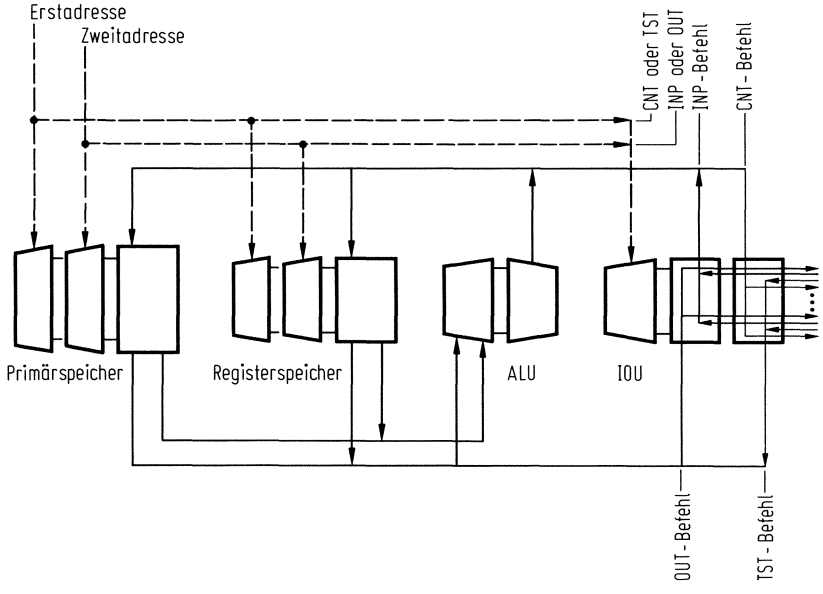


Bild 6.2 Datenwerk des Zweiadreßrechners Bild 4.1 mit erweiterter Ein-Ausgabeeinheit (IOU)

für die Byteleitungen und für die Bitleitungen, von denen der erste entsprechend der rechten und der zweite entsprechend der linken Darstellung von Bild 6.1 arbeitet, wenn Registerspeicher und Primärspeicher als ein Speichermodul und das Verknüpfungsschaltnetz als Prozessormodul betrachtet werden. Byteleitungen werden hauptsächlich zur Übertragung von Daten benutzt, die im Primärspeicher zwischengespeichert werden; Bitleitungen werden zum Steuern und Testen der Peripherie verwendet, deren Zustand in der arithmetisch-logischen Einheit (ALU) unmittelbar verarbeitet wird.

Werden Register- und Primärspeicher als getrennte Moduln betrachtet, so haben wir die Erweiterung der rechten Darstellung von Bild 6.1 auf zwei Speichermoduln vor uns. Auch Prozessor- oder Ein-Ausgabemoduln können mehrfach vorhanden sein. Geht man weiter von Einfachbussen auf Mehrfachbusse über, so entstehen modulare Konfigurationen, wie sie im nächsten Abschnitt behandelt werden. Denkt man sich z.B. den Primärspeicher in Bild 6.2 aus zwei unabhängigen "Bänken" aufgebaut, so können bei der Verwendung von Zweifachbussen der Prozessormodul mit der einen und der Ein-Ausgabemodul mit der anderen Bank zusammenarbeiten, ohne sich gegenseitig zu stören. Eine Voraussetzung für diesen parallelen Betrieb ist, daß der Prozessormodul und der Ein-Ausgabemodul über eigene Befehlswerke verfügen. Man nennt dann den Prozessormodul Zentralprozessor und den Ein-Ausgabemodul Ein-Ausgabekanal oder Ein-Ausgabeprozessor.

Prinzipien der Steuerung peripherer Geräte

Die Befehle des verallgemeinerten Von-Neumann-Rechners gliedern sich entsprechend der in Bild 6.2 gezeigten Struktur in zwei Gruppen: die Befehle für die arithmetisch-logische Einheit und die Befehle für die Ein-Ausgabe-Einheit. Für beide Einheiten existiert ein gemeinsames (in Bild 6.2 nicht gezeichnetes) Leitwerk. Die arithmetischen und logischen Befehle schalten die arithmetisch-logische Einheit, und die Ein-Ausgabe-

Ein-Ausgabebefehle		Entsprechungen	
INP	N G ↑ ↑ Geräteadresse Speicheradresse	LD	N G ↑ ↑ Geräteadresse Speicheradresse
OUT	N G ↑ ↑ Geräteadresse Speicheradresse	LD	G N ↑ ↑ Speicheradresse Geräteadresse
CNT	G N ↑ ↑ Speicheradresse Geräteadresse	LD	G N ↑ ↑ Speicheradresse Geräteadresse
TST	G N ↑ ↑ Speicheradresse Geräteadresse	CMP	G N ↑ ↑ Speicheradresse Geräteadresse

befehle schalten die Ein-Ausgabe-Einheit an die beiden Datenbusse. Die beiden Ein-Ausgabebefehle INP und OUT entsprechen den logischen Befehlen Laden und Speichern, und es ist naheliegend, auch Ein-Ausgabebefehle einzuführen, die den arithmetischen Befehlen entsprechen. Beschränkt man sich auf die für die Funktion der Ein-Ausgabe unbedingt notwendigen Befehle, so ergibt sich obige Modifizierung bzw. Erweiterung

der Befehlsliste. Die Befehle INP und OUT beziehen sich auf den Transport von Daten aus der Peripherie in den Speicher und umgekehrt. Durch die Angabe der Multiplexer-Demultiplexer-Adresse im Zweitadreßteil des Befehls wird die Verbindung zu einem der Peripheriegeräte hergestellt. Peripheriegeräte von kommerziellen Rechnern sind im allgemeinen byteorientiert. Dementsprechend werden durch die Befehle INP und OUT jeweils nur acht Bits parallel übertragen. Die Datenleitungen zwischen Multiplexern-Demultiplexern und den Geräten sind auf acht Bits ausgebaut. Wir vereinbaren, daß diese Leitungen für die Eingabe rechtsbündig und für die Ausgabe linksbündig mit den Datenbussen verbunden sind. Peripheriegeräte von Prozeßrechnern sind byteorientiert, was periphere Speicher und Ein-Ausgabegeräte angeht, und wortorientiert, was die Prozeßperipherie betrifft. Dementsprechend sind die Datenleitungen zwischen Multiplexern-Demultiplexern und Geräten voll ausgebaut. Darüber hinaus ist es oft nützlich, zum Steuern oder Testen der Peripherie nur ein einziges Bit zu übertragen. Dies wird im allgemeinen nicht durch die Befehle INP oder OUT, sondern durch die Befehle CNT und TST bewerkstelligt.

Die Befehle CNT und TST dienen zum Steuern und Testen der Peripherie. Im Gegensatz zum Multiplexer-Demultiplexer für die Befehle INP und OUT arbeitet der Multiplexer-Demultiplexer für die Befehle CNT und TST in derselben Richtung wie der Speicher. Mit CNT kann dadurch ein Wort aus dem Speicher oder eine im Befehl angegebene Konstante für die Dauer der Ausführung des Befehls an die Peripherie ausgegeben werden. Mit TST kann der Wert einer Größe aus der Peripherie während der Ausführung des Befehls mit einem Wort aus dem Speicher oder einer im Befehl angegebenen Konstanten verglichen werden. Diese Befehle eignen sich zum Stellen oder Abfragen eines Bits genauso wie zum Ausgeben oder Abfragen eines Steuerwortes. Bei kommerziellen Rechnern sowie bei Prozeßrechnern werden sie zum Betrieb der Peripheriegeräte benutzt, z.B. für das Rückspulen eines Magnetbandes, für das Stellen von Schaltern oder das Abfragen von Sensoren. Je nachdem ob auf diesen Leitungen ein Bit, ein Wort oder nur ein Teil eines Wortes übertragen wird, sind die Multiplexereingänge und die Demultiplexerausgänge nur mit einer Leitung versehen, voll oder nur zum Teil ausgebaut. Wir vereinbaren, daß diese Leitungen rechtsbündig mit den Datenbussen verbunden werden.

Ein-Bit-Eingabe- und Ein-Bit-Ausgabeleitungen werden vielfach als Sensor- und Steuerleitungen (sense lines bzw. control lines) bezeichnet. Selbst bei den einfachsten Ein-Ausgabe-Programmen werden sie zur Synchronisation der Peripheriegeräte mit dem Prozessor benötigt, wie das folgende Beispiel zeigt.

Beispiel 6.1 Schreibmaschinenausgabe

Wir schreiben ein einfaches Ausgabeprogramm, mit dem ein Text auf einer Schreibmaschine ausgegeben werden kann. Die Byteleitung für die Zeichenübertragung zur Schreib-

maschine und die Sensorleitung für die Auswertung des Zustands der Schreibmaschine sind unter der Adresse 10 am Ein-Ausgabe-Multiplexer-Demultiplexer angeschlossen (Bild 6.3). Vor der Übertragung eines jeden Zeichens wird die Schreibmaschine mit Hilfe der Sensorleitung "gefragt", ob das erste Zeichen geschrieben werden kann oder ob das Schreiben des vorhergehenden Zeichens beendet ist und das nächste Zeichen geschrieben werden kann. Ist die Schreibmaschine für eine Ausgabe bereit, so wird jeweils aus dem Feld mit der Anfangsadresse ALFABET ein Wort, bestehend aus 2 Zeichen, in ein Pufferregister (Akkumulator) übertragen. Das links im Puffer stehende Zeichen wird ausgegeben und der Pufferinhalt um ein Zeichen nach links geschiftet. Sind zwei Zeichen eines Wortes ausgegeben, wird das nächste Wort in das Pufferregister gebracht und gleichermaßen verfahren. Sind alle Wörter auf diese Weise verarbeitet, so ist die Ausgabe des "Satzes" beendet.

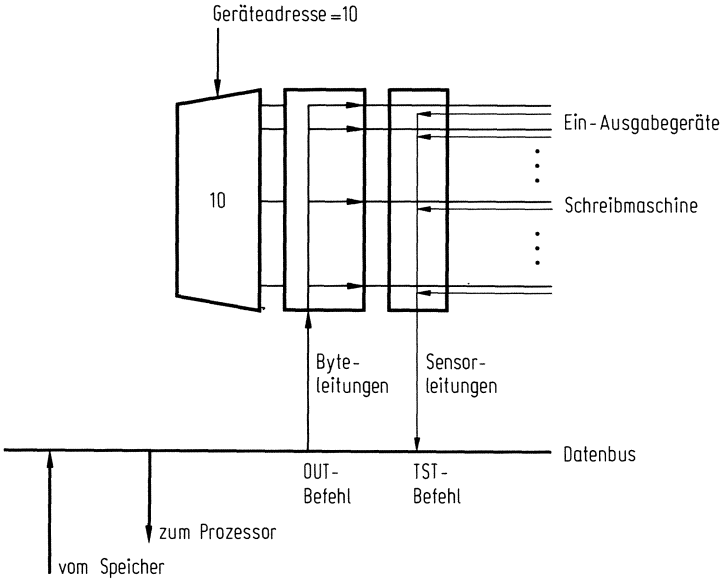


Bild 6.3 Ausschnitt aus der Ein-Ausgabeeinheit mit Anschluß einer Schreibmaschine über eine Byte- und eine Sensorleitung (Ausgabemodus)

Wir schreiben das Ausgabeprogramm als Makro mit dem Namen TYPE und geben als ersten Parameter die Anzahl der auszugebenden Wörter und als zweiten Parameter die Adresse des auszugebenden Textes an.

Im Ausgabeprogramm benutzen wir zwei Zähler: den einen, um das Ende der Ausgabe eines aus zwei Zeichen bestehenden Wortes mitzuteilen (Register **J**), und den zweiten, um das Ende der Ausgabe eines aus 13 Wörtern bestehenden Satzes anzuzeigen (Register **I**). Das sog. Wortendesignal tritt praktisch bei jeder Übertragung von zu Wörtern zusammengesetzten Bytes auf (Sensorleitung TY).

Ausgabeprogramm (Prozessorprogramm mit Sensor-Leitung)

```
        org                 Hauptprogramm
         :
ALFABET text    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
         :
START    :
         :
        TYPE    13 ALFABET  Schreibe 13 Wörter auf der Schreibmaschine
         :
        end     START
```

```
TYPE    makro   P1    P2     Makrodefinition von TYPE
TY      equ     10           Multiplexer-Demultiplexer-Adresse Schreibmaschine
        LD      I     '0     Wortzähler setzen
LO      LD      A     P2,I   Wort in A zwischenspeichern
        LD      J     '0     Bytezähler setzen
L1      TST     TY    '1     auf Ende des Druckvorganges eines Zeichens warten
        GOTO    L1
        OUT     A     TY     Ausgabe des höchstwertigen Bytes von A
        SA      A     '8     Linksschift von A um 8 Stellen
        ADD     J     '1     Bytezählerstand erhöhen
        CMP     J     '2     auf "Wortende" abfragen
        GOTO    L1
        ADD     I     '1     Wortzählerstand erhöhen
        CMP     I     'P1    auf "Satzende" abfragen
        GOTO    LO
        mend
```

Zur Pufferung der Bytes wird oft ein spezielles Register zwischen Multiplexer-Demultiplexer und Datenbus vorgesehen, das mit einem sog. Bytezähler gekoppelt ist. Diese durch Hardware erweiterte Ein-Ausgabeeinheit wird Ein-Ausgabekanal genannt. Vielfach werden darüber hinaus im Kanal zum Zählen und Adressieren der Wörter zwei weitere Zähler vorgesehen, wodurch die Übertragung eines Satzes vom Kanal völlig selbständig abgewickelt werden kann. Ein-Ausgabekanäle werden in Abschnitt 6.4 genauer behandelt, wobei die hier vorgestellte Schreibmaschinenausgabe wiederholt als Beispiel benutzt wird. ■

Ein- und Ausgabegeräte

Im folgenden wird ein grober Überblick über die gebräuchlichen Ein- und Ausgabegeräte gegeben und ihr technischer Aufbau kurz beschrieben.

Schreibmaschine. Das einfachste Gerät zur Kommunikation zwischen Mensch und Rechner ist die elektrische Schreibmaschine. Je nach den Anforderungen an die Datenstation wird ein handelsüblicher Fernschreiber oder eine speziell als Peripherie-

gerät entwickelte Konsolschreibmaschine benutzt. Während im Fernschreiber die Umsetzung des auf der Tastatur angeschlagenen Zeichens in einen seriell übertragenen Binärcode und umgekehrt mechanisch über Wählschienen und nockengesteuerte Hebel vollzogen wird, erfolgt die Codierung in der Konsolschreibmaschine über Schaltnetze oder Schaltwerke.

Lochstreifengeräte. Auf dem Lochstreifen ist die Information parallel auf 5 bis 8 Spuren gespeichert. Ein Loch bedeutet eine Eins, kein Loch eine Null. Außerdem enthält jeder Lochstreifen eine Transportspur mit etwas kleineren Löchern. Die Transportspur dient dem Streifenvorschub, wobei ein Transportrad mit kleinen Stiften in die Löcher greift. Bei manchen Geräten erfolgt der Streifentransport wie beim Magnetbandgerät über Antriebsrollen.

Man unterscheidet die beiden Betriebsarten Einzelabruf und Dauerdurchlauf. Beim Einzelabruf wird das Lochstreifengerät zur Übertragung jeweils eines Zeichens von außen angestoßen und bleibt bis zum nächsten Aufruf wieder stehen. Beim Dauerdurchlauf arbeitet das Gerät mit maximaler Geschwindigkeit. Damit zwingt es den angeschlossenen Einheiten seinen Arbeitsrhythmus auf, und man benötigt zur Synchronisation einen Pufferspeicher.

Beim Stanzen werden im allgemeinen alle Stanzstempel durch denselben Motor über einen Exzenter angetrieben. Das ankommende Signal steuert für jede Spur über einen kleinen Magneten einen Mitnehmer, der die Motorbewegung auf den Stempel überträgt bzw. nicht überträgt.

Das Lesen erfolgt ebenfalls für alle Spuren gleichzeitig. Beim Einzelabrufbetrieb kann die Abtastung durch Fühlstifte erfolgen, die durch Löcher im stillstehenden Streifen passen und so einen Kontakt schließen. Beim Dauerdurchlaufbetrieb ist ein mechanisches Abtasten nur mit Kontaktbürsten möglich, die einen Kontakt zu einer unter dem Streifen mitlaufenden Walze schließen, sobald ein Loch im Streifen unter ihnen hindurchläuft. Da die mechanische Abtastung jedoch nur begrenzte Geschwindigkeiten zuläßt, wird bei schnellen Lesern mit weißem oder infrarotem Licht gelesen. Dabei gibt eine Photozelle einen Impuls ab, sobald Licht aus einer über jeder Datenspur liegenden Lichtquelle durch ein Loch im Streifen auf sie fällt. Ebenso schnell arbeiten kapazitive Leser, bei denen durch ein Loch im Papier des Datenträgers die Dielektrizitätskonstante und damit die Kapazität eines Kondensators in einem Schwingkreis verändert wird.

Lochkartengeräte. Die Lochkartengeräte arbeiten im Prinzip sehr ähnlich wie die Lochstreifengeräte. Eine Lochkarte kann 80 Zeichen enthalten, die jeweils mit 12 Bits codiert sind. Gegenüber dem Lochstreifen weist sie folgende Vorteile auf: Sie ist stabiler und ein Datensatz wird durch sie in kleinere Einheiten unterteilt, die leicht ausgetauscht oder umsortiert werden können. Außerdem läßt sich auf ihnen die gespeicherte Information leicht noch in Klarschrift darstellen, was für ihren Gebrauch außerhalb des Rechners nützlich ist.

Zur Verarbeitung liegen die Karten zunächst gestapelt in einem Magazin. Von dort werden sie mit Transportwalzen oder Schiebern der Lese- bzw. Stanzvorrichtung zugeführt. Auch hier unterscheidet man Dauerdurchlauf und Einzelabrufbetrieb. Nur bei kleineren Geräten erfolgt das Lesen und Stanzen spaltenweise, d.h. Zeichen für Zeichen wie beim Lochstreifen. Die meisten Lochkartengeräte arbeiten zeilenweise. Da die Information jedoch spaltenweise codiert ist, wird dann ein Pufferspeicher benötigt, der mindestens einen Karteninhalt aufnehmen kann. Einen ebenso großen Pufferspeicher benötigen die Geräte, die den gesamten Karteninhalt auf einmal lesen bzw. stanzen. Zur Vermeidung von Fehlern wird jeder Stanzvorgang bei Lochstreifen und Lochkarten durch sofort anschließendes Lesen überprüft. Ebenso sollte das Lesen durch zwei voneinander unabhängige Systeme erfolgen.

Die später angegebene Tabelle ermöglicht einen Überblick über die erreichbaren Stanz- und Lesegeschwindigkeiten bei Lochstreifen und Lochkarten.

Zeilendrucker. Da die Schreibgeschwindigkeit selbst einer modernen Schreibmaschine mit 20 Zeichen pro Sekunde für die Ausgabe großer Datenmengen viel zu niedrig ist, werden stattdessen Schnelldrucker eingesetzt, die jeweils eine ganze Zeile mit bis zu 160 Zeichen gleichzeitig ausdrucken. Neben der Druckleistung sind ihre wesentlichen Kennzeichen der Zeichenvorrat, die Zeilenlänge, das Schriftbild, die Geschwindigkeit, mit der Leerzeilen übersprungen werden können, die Anzahl der möglichen Durchschläge und die Geräuschentwicklung. Man unterscheidet mechanische und nichtmechanische Drucker.

Die vier Hauptbestandteile aller mechanischen Drucker sind Typenträger, Anschlagvorrichtung, Färbevorrichtung und Papiertransportvorrichtung. Bei allen modernen Zeilendruckern bewegt sich der Typenträger mit konstanter, hoher Geschwindigkeit am Papier und der Anschlagvorrichtung vorbei. Für jede Zeichenposition in einer Zeile ist ein kleiner Anschlaghammer vorgesehen, der in dem Moment betätigt wird, in dem das zu druckende Schriftzeichen auf dem Typenträger vor ihm "vorbeifliegt". Man spricht deshalb vom fliegenden Druckverfahren. Um ein klares Schriftbild zu erhalten, muß der Anschlag des Hammers sehr kurz und kräftig und zu einem genau definierten Zeitpunkt erfolgen. Das Einfärben erfolgt meist wie bei der Schreibmaschine mit Hilfe von Farbbändern. Jeder Typenträger muß alle verwendeten Schriftzeichen mindestens einmal enthalten. Je häufiger sich der Zeichensatz auf dem Typenträger wiederholt, desto schneller kann er den einzelnen Anschlaghämmern angeboten werden, wodurch die Druckleistung erhöht wird. Je nach der Art der Typenträger unterscheiden wir Trommeldrucker, Kettendrucker, Gliederdrucker und Stabdrucker.

Bei dem Versuch, die Ausgabegeschwindigkeit mechanischer Drucker weiter zu erhöhen, steigt im allgemeinen der Verschleiß derart an, daß ein längerfristiges fehlerfreies Arbeiten der Geräte nicht mehr gewährleistet ist. Da oft Magnetbänder als Ausgabepuffer verwendet werden, ist man bemüht, die Drucker der Ausgabegeschwindigkeit der Bandgeräte anzupassen. Dies gelingt nur mit einigen nichtmechanischen Zeilendruckern. Es werden elektrostatische, elektromagnetische, elektrochemische, elektrothermische und photographische Verfahren angewendet.

Das elektrostatische Verfahren ist auch als xerographischer Druckprozeß bekannt und erfordert sechs Arbeitsgänge: Die Oberfläche einer selenbeschichteten Trommel wird zunächst gleichmäßig positiv aufgeladen. Bei der anschließenden Belichtung mit der Kopiervorlage wird die Ladung entsprechend den weißen Stellen der Vorlage von der Trommel wieder abgeleitet. Nun wird die Trommel mit negativ geladenem Farbpulver bestreut, das nur an den positiv geladenen Stellen haften bleibt. Unter leichtem Druck rollt die xerographische Trommel über Papier ab, auf dem sich so das Bild abzeichnet. Abschließend wird das Farbpulver auf dem Papier fixiert und die Trommel wieder gereinigt. In ähnlicher Weise läuft das elektromagnetische Druckverfahren ab, nur wird hier statt der Selentrommel ein magnetisierbarer Hilfsträger und statt geladenem magnetisches Farbpulver verwendet. Bei den elektrochemischen Verfahren wird ein Stromimpuls über metallische Schreibstifte durch das befeuchtete Papier (meist Spezialpapier) auf eine Metallwalze geleitet. Dieser Stromimpuls bewirkt eine lokale Einfärbung des Papiers entweder durch Ionisation und elektrolytischen Abbau einer Elektrode oder durch Änderung des pH-Wertes, d.h. der Ionenkonzentration im Papier.

Die elektrothermischen Druckverfahren verwenden ein mehrschichtiges Spezialpapier, dessen weiße Oberfläche durch einen über die Schreibstifte fließenden Strom so stark erhitzt wird, daß der schwarze Untergrund sichtbar wird. Bei den photographischen Druckverfahren wird mit Hilfe des Elektronenstrahls einer Kathodenstrahlröhre wie beim Bildschirmgerät das gewünschte Schriftbild erzeugt und auf einem Mikrofilm festgehalten.

Die folgende Tabelle gibt einen Überblick über die Arbeitsgeschwindigkeiten verschiedener Ein- und Ausgabegeräte.

Gerätetyp	maximale Arbeitsgeschwindigkeit	Ein- bzw. Ausgabemedien
Lochstreifenstanzer	300 Zeichen/s	Lochstreifen
Lochstreifenleser mit Fühlstiften	20 " "	"
Lochstreifenleser mit Kontaktbürsten	200 " "	"
optischer oder kapazitiver Lochstreifenleser	2000 " "	"
Lochkartenstanzer	650 Zeichen/s ≙ 500 Karten/min	Lochkarte
Lochkartenleser	2000 Zeichen/s ≙ 1500 Karten/min	"
Schreibmaschine	20 Zeichen/s	normales Papier
Stabdrucker	600 Zeilen/min	" "
Kettendrucker	1300 " "	" "
Gliederdrucker	2000 " "	" "
Trommeldrucker	1600 " "	" "
elektrostatischer Drucker	30000 " "	Spezialpapier[1]
elektromagnetischer Drucker	2400 " "	normales Papier[2]
elektrochemischer Drucker	6000 " "	Spezialpapier[1]
elektrothermischer Drucker	200 " "	Spezialpapier
photographischer Drucker	30000 " "	Mikrofilm

[1] Verwendung von Normalpapier ist bei manchen Geräten möglich, jedoch sinkt dann die Ausgabegeschwindigkeit beträchtlich.

[2] Bei manchen Geräten ist Spezialpapier erforderlich, doch erhöht sich damit die Ausgabegeschwindigkeit nicht.

Analog-Digital-Umsetzer (ADU). Beim Einsatz von Digitalrechnern für die Datenerfassung und -verarbeitung in technischen Prozessen fallen nicht nur digitale, sondern vorwiegend auch analoge Meßwerte an. Um sie in den Rechner eingeben zu können, müssen sie in eine digitale Form umgesetzt werden. Das geschieht durch sog. Analog-Digital-Umsetzer. Bei der Umsetzung eines analogen Meßwertes in einen digitalen Meßwert wird festgestellt, wieviel mal eine Maßeinheit in der analogen Größe enthalten ist. Dies kann auf zwei verschiedene Arten geschehen: Entweder zählt man die Anzahl der Einheiten ab, die von einem Bezugspunkt an digital aufeinander addiert werden müssen, bis sie der zu messenden Größe gleichen, oder man benutzt Vergleichsmaßstäbe mit mehreren verschieden feinen Skalen. Dabei wird zwischen zwei Marken einer gröberen Skala mit der jeweils feineren Skala interpoliert, so wie man auf einem Maßband erst die Meter, dann die Zentimeter und schließlich die Millimeter abliest.

Eine direkte Umsetzung analoger elektrischer Größen in einen digitalen Wert ist nicht möglich. Man vergleicht den Ausgang eines Digital-Analog-Umsetzers über einen Komparator mit der Meßgröße und erhöht oder vermindert den Digitalwert so lange, bis er den Meßwert erreicht. Dieses Erhöhen bzw. Vermindern kann in gleichmäßigen kleinen Zählschritten erfolgen oder ähnlich dem Divisionsverfahren ohne Rückstellen des Rests in Vorwärts- und Rückwärtsschritten wechselnder Richtung mit jeweils halbierter Schrittweite (vgl. Algorithmus 1.12).

Digital-Analog-Umsetzer (DAU). Als Umkehrung zur Umsetzung analoger in digitale Größen ist es bei der Datenverarbeitung in technischen Prozessen notwendig, digitale Größen in analoger Form auszugeben. Das geschieht durch sogenannte Digital-Analog-Umsetzer.

Die Digital-Analog-Umsetzung elektrischer Größen ist verhältnismäßig einfach mit Hilfe eines schaltbaren Spannungsteilers zu bewerkstelligen. Jedes Bit der digitalen Größe steuert über einen Schalttransistor einen Präzisionswiderstand in einem Widerstandsnetzwerk. Diese Widerstände unterscheiden sich um jeweils eine Zweierpotenz voneinander. Zur Erzeugung geometrischer Größen werden dagegen entweder Schrittmotoren eingesetzt, die mit jedem Impuls eines Zählwerks um eine genau definierte Strecke bzw. einen Winkel weiterlaufen, oder man nimmt die Umsetzung indirekt vor. Dabei werden die tatsächliche Position (Istwert) und die digital angegebene Größe (Sollwert) jeweils in eine elektrische Spannung umgesetzt. Beide Spannungen werden miteinander über einen Differenzverstärker verglichen, dessen Ausgang einen Stellmotor treibt. Erst wenn die Differenzspannung Null ist (Istwert gleich Sollwert), kommt der Stellmotor zum Stillstand.

X-Y-Schreiber. Ein typisches Anwendungsbeispiel für die eben beschriebenen Digital-Analog-Umsetzer sind die X-Y-Schreiber (plotter). Sie benötigen je einen DAU für die Positionierung in X- und Y-Richtung und zusätzlich eine Information, ob der Schreibstift gehoben oder gesenkt werden soll. Für Plotter werden beide möglichen Umsetzungsarten angewendet: Schrittmotoren mit Zählern oder Stellmotoren mit Differenzverstärkern.

Bildschirmgeräte. Für das interaktive Arbeiten am Rechner ist das Bildschirmgerät (display) in vielen Anwendungsbereichen ein unentbehrliches Hilfsmittel geworden. Das Kernstück jedes Bildschirmgerätes ist eine Braunsche Röhre (Kathodenstrahlröhre). Die zur X- und Y-Ablenkung sowie zur Helligkeitssteuerung des Kathodenstrahls erforderlichen Daten sind in digitaler Form im Bildwiederholungsspeicher abgelegt und werden über DAUs an die Röhre übermittelt. Um ein flackerfreies Bild zu erhalten, muß der gesamte Bildwiederholungsspeicher mindestens 25 mal pro Sekunde ausgelesen werden. Die Bilder können wie beim Fernsehen punktweise erzeugt werden (Fernsehrasterverfahren) oder aber aus einzelnen Linien zusammengesetzt werden (Vektorverfahren). Die letztere Möglichkeit liefert ein besseres Bild, erfordert aber bedeutend mehr Aufwand. In sog. Vektorgeneratoren werden durch Hardware aus den Anfangs- und Endkoordinaten die variablen Ablenkspannungen für den Schreibstrahl sowie die in Abhängigkeit von der Schreibgeschwindigkeit zu ändernde Spannung zur Steuerung der Strahlintensität erzeugt. Will man über eine Tastatur alphanumerische Zeichen an bestimmten Positionen des Bildschirms darstellen, so wird ein Zeichengenerator benötigt, der aus den digitalen Angaben über Zeichenart, -größe und -position analoge Signale zur Steuerung des Schreibstrahls erzeugt. Mit Hilfe von graphischen Tabletts, Rollkugeln, Steuerknüppeln und ähnlichen Geräten, deren Positionen über ADUs umgesetzt und dann über die Steuereinheit dem Bildwiederholungsspeicher zugeführt werden, lassen sich Markierungen, Zeichnungen und Bewegungen auf dem Bildschirm darstellen. Mit dem Lichtgriffel (light pen) lassen sich direkt am Bildschirm Bildelemente markieren oder neu darstellen.

Klarschriftleser. Im allgemeinen muß der Mensch eine Information für den Rechner, die ihm schriftlich oder mündlich zugeht, zunächst in eine der Maschine verständliche Form umcodieren. Dies geschieht, indem er die Information über eine Schreibma-

schine, ein Lochstreifen- oder -kartengerät oder ein Bildschirmgerät an den Rechner weitergibt. Dieses Verfahren ist ineffektiv, wenn viele Belege verarbeitet werden müssen, wie dies in der kommerziellen Datenverarbeitung der Fall ist. Deshalb wurden Geräte entwickelt, die geschriebene Zeichen erkennen und in eine dem Rechner verständliche Form umsetzen. Die automatische Zeichenerkennung erfolgt grundsätzlich in drei Schritten: Zunächst wird das Bild des Zeichens (meist optisch) abgetastet. Dann werden seine besonderen Merkmale festgestellt, wobei die Kriterien, die zur Auswahl der Merkmale herangezogen werden, je nach Verfahren sehr verschieden sind. Anschließend werden die ermittelten Merkmale mit den in einem Speicher niedergelegten Merkmalen aller definierten Zeichen verglichen. Im Falle der Übereinstimmung der wesentlichen Merkmale wird das codierte Zeichen gespeichert. Wird mit keinem oder mit mehreren Zeichen Übereinstimmung festgestellt, so wird das Zeichen als nicht erkannt aussortiert.

Die Bestimmung spezifischer Merkmale und die anschließende Klassifikation ist verhältnismäßig einfach, wenn es sich um gedruckte Schrift mit ständig gleichbleibendem Schriftbild handelt. Die Erkennung unterschiedlicher Druckschriften oder gar handgeschriebener Schrift wirft dagegen Probleme auf, die z.Z. noch nicht befriedigend gelöst sind, ohne daß dem Schreiber scharfe Restriktionen auferlegt werden. Noch größer werden die Probleme bei der automatischen Spracherkennung.

6.2 Modulare Konfigurationen

Wir wollen die in Abschnitt 6.1 angegebenen Zusammenschaltungen von Prozessor-, Speicher- und Ein-Ausgabemoduln in einen allgemeineren Rahmen stellen. Die einzelnen Moduln sind über Busse verbunden, und Multiplexer und Demultiplexer schalten die Verbindungen durch.

Einfach- und Mehrfachbussysteme

Bild 6.4 zeigt die Struktur eines Einfachbussystems, wobei die Multiplexereingänge in der Matrix **A** und die Demultiplexerausgänge in der Matrix **B** zusammengefaßt

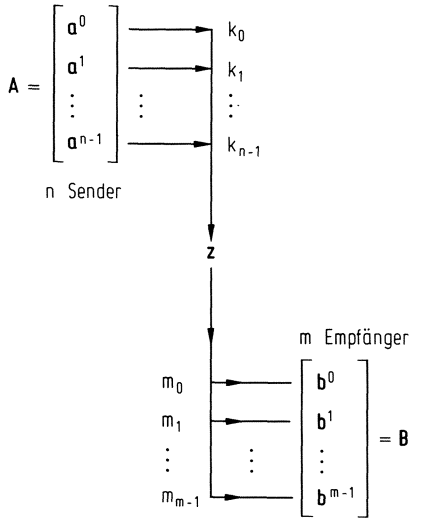


Bild 6.4 Sternförmige Zusammenschaltung mehrerer Sender und Empfänger über ein Einfachbussystem

sind. $\mathbf{a}^0$, $\mathbf{a}^1$ bis $\mathbf{a}^{n-1}$ bezeichnen die Ausgangsleitungen von n Moduln (den Sendern), und $\mathbf{b}^0$, $\mathbf{b}^1$ bis $\mathbf{b}^{m-1}$ bezeichnen die Eingangsleitungen von m Moduln (den Empfängern). Mit den in Abschnitt 2.2 eingeführten Formeln zur Beschreibung von Multiplexern und Demultiplexern läßt sich die Struktur der in Bild 6.4 dargestellten Zusammenschaltung durch zwei Gleichungen, nämlich

$$\mathbf{B} = {}^{T}\mathbf{m} \mathbin{+\cdot} \mathbf{z}$$
$$\mathbf{z} = \mathbf{k} \mathbin{+\cdot} \mathbf{A}$$

oder durch eine Gleichung beschreiben:

$$\mathbf{B} = {}^{T}\mathbf{m} \mathbin{+\cdot} (\mathbf{k} \mathbin{+\cdot} \mathbf{A}) \tag{6.1}$$

Wegen der Gültigkeit des Assoziativgesetzes für die Operation $+\cdot$ kann Gl.(6.1) auch in folgender Form durch eine Gleichung, nämlich

$$\mathbf{B} = ({}^{T}\mathbf{m} \mathbin{+\cdot} \mathbf{k}) \mathbin{+\cdot} \mathbf{A} \tag{6.2}$$

oder durch zwei Gleichungen beschrieben werden:

$$\mathbf{C} = {}^{T}\mathbf{m} \mathbin{+\cdot} \mathbf{k}$$
$$\mathbf{B} = \mathbf{C} \mathbin{+\cdot} \mathbf{A}$$

Die Interpretation der Gln.(6.1) und (6.2) unterscheidet sich wesentlich. Gl.(6.1) beschreibt die "sternförmige" Zusammenschaltung von n Sendern und m Empfängern. Zu einem Zeitpunkt ist genau ein Sender mit genau einem Empfänger verbunden. Gl. (6.2) beschreibt dagegen die "matrixförmige" Zusammenschaltung von n Sendern und

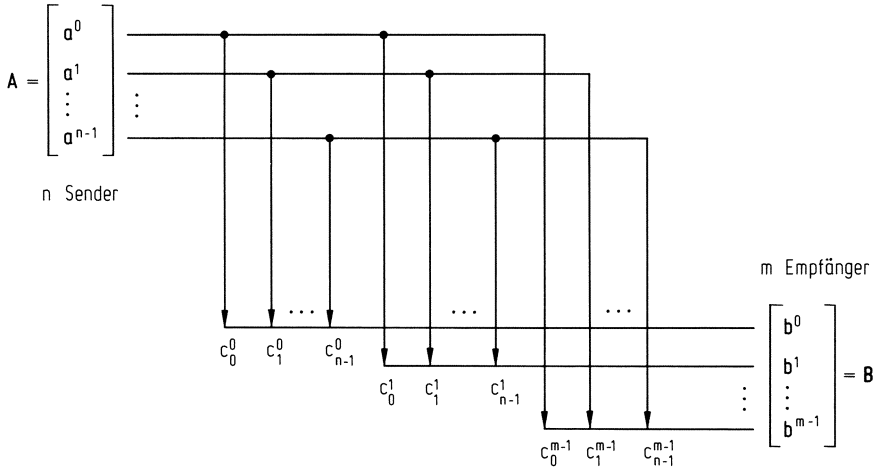


Bild 6.5 Matrixförmige Zusammenschaltung mehrerer Sender und Empfänger über ein Mehrfachbussystem

m Empfängern. Zu einem Zeitpunkt können mehrere Sender mit mehreren Empfängern verbunden sein. Bild 6.5 zeigt die Interpretation von Gl.(6.2) symbolisch. Man sieht, wie die n Sender mit den m Empfängern durch n mal m Leitungen verbunden sind.

In Abschnitt 5.2 sind diese beiden Arten der Zusammenschaltung von Moduln für den seriellen und parallelen Betrieb von Speichern benutzt worden. Die sternförmige Zusammenschaltung entsprechend Bild 6.4 erlaubt grundsätzlich nur seriellen Betrieb, wohingegen die matrixförmige Zusammenschaltung entsprechend Bild 6.5 auch parallelen Betrieb ermöglicht.

Zusammenschaltungen von Speichern und Prozessoren

Werden Speicher und Prozessoren als Moduln in Einfach- oder Mehrfachbussystemen verwendet, so entstehen Rechnerkonfigurationen, die für seriellen und für parallelen Betrieb geeignet sind. Wir geben im folgenden je ein Konfigurationsbeispiel für eine Einfach- und für eine Mehrfachbuszusammenschaltung an.

Beispiel 6.2 Rechnerstruktur für serielle Verarbeitung

Bild 6.6 zeigt vier Primärspeichermoduln, die mit zwei Prozessormoduln, einem Zentral- und einem Ein-Ausgabeprozessor (CPU bzw. IOU) über ein Einfachbussystem verbunden sind. Wie in Bild 6.6 gezeichnet, sind Datenübertragungen nur zwischen Prozessor- und Speichermoduln, nicht aber zwischen Prozessor- oder Speichermoduln untereinander möglich. Die sternförmige Zusammenschaltung erlaubt zu einem Zeitpunkt nur eine Prozessor-Speicher-Verbindung.

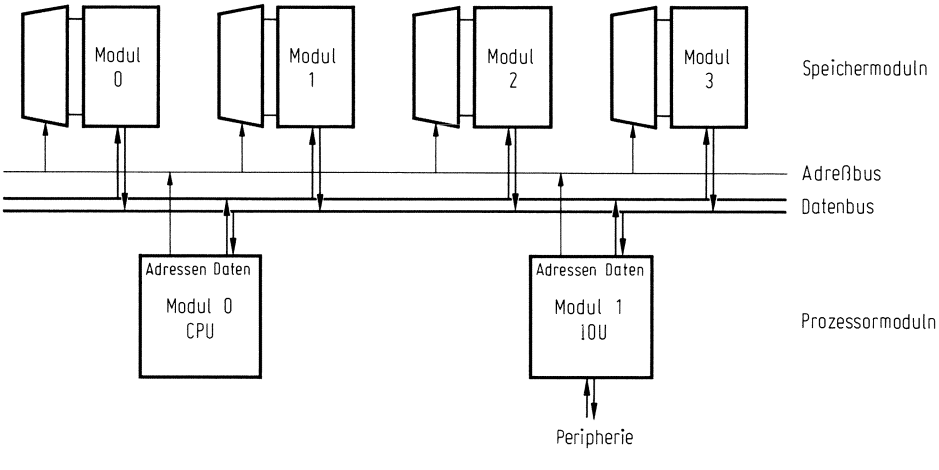


Bild 6.6 Datenübertragung zwischen Prozessoren und Speichern auf Einfachbussen

Diese Konfiguration ist typisch für Rechner, deren Ein- und Ausgabe mit einem Ein-Ausgabekanal durchgeführt wird.[1] Der Kanal benötigt die Busse zur Übertragung eines Wortes für die Dauer eines Speicherzyklus. Dabei "stiehlt" er dem Zentralprozessor diesen Zyklus; man spricht von Cycle Stealing (s. Abschnitt 6.4). ■

Beispiel 6.3 Rechnerstruktur für parallele Verarbeitung

Bild 6.7 zeigt dieselbe Anzahl von Speicher- und Prozessormoduln wie Bild 6.6. Im Unterschied zu Bild 6.6 sind in Bild 6.7 jedoch die Busse zweifach vorgesehen. Durch die matrixförmige Zusammenschaltung ist es möglich, daß der Zentralprozessor und der Ein-Ausgabeprozessor mit jeweils einem Speichermodul zusammenarbeiten.

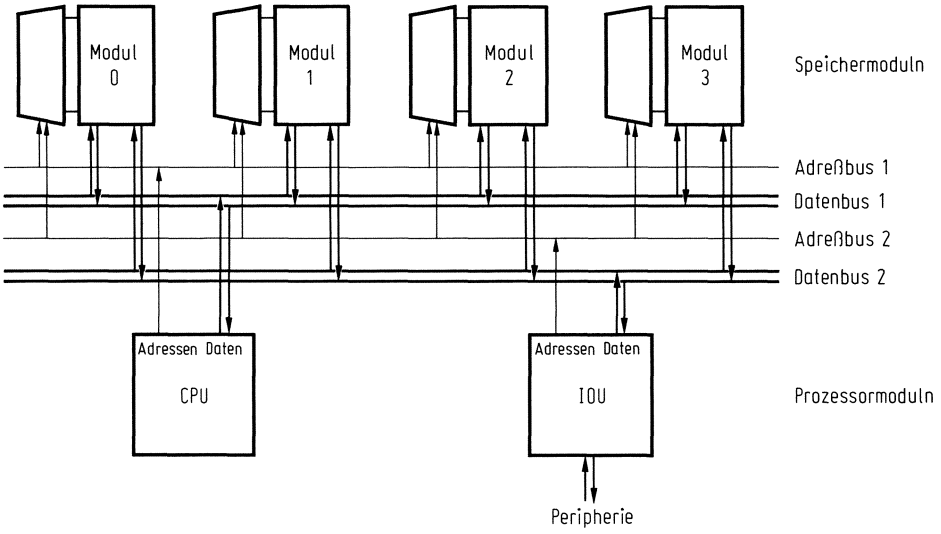


Bild 6.7 Datenübertragung zwischen Prozessoren und Speichern auf Mehrfachbussen

Diese Konfiguration ist typisch für Rechner, deren Ein-Ausgabe mit einem Ein-Ausgabeprozessor durchgeführt wird.[2] Der Ein-Ausgabeprozessor arbeitet mit dem Speicher, ohne Zeit vom Zentralprozessor zu beanspruchen, solange beide unterschiedliche Moduln adressieren. Dabei ist er "direkt" mit einem Speichermodul verbunden; man spricht von Direct Memory Access[3]. ■

---

[1] Vgl. z.B. Konfiguration einer XDS-SIGMA-2-Anlage.

[2] Vgl. z.B. Konfiguration einer XDS-SIGMA-7-Anlage.

[3] Von Direct Memory Access wird teilweise auch gesprochen, wenn in einem Einfachbussystem der Ein-Ausgabekanal nicht mit dem Prozessor, sondern mit dem Speicher verbunden ist (vgl. Bild 6.1).

Zusammenschaltungen von Prozessoren und Ein-Ausgabegeräten

Werden Prozessoren und Ein-Ausgabegeräte als Moduln in Einfach- oder Mehrfachbussystemen verwendet, so entstehen Peripherie-Konfigurationen für seriellen Betrieb, für scheinbar parallelen Betrieb und für wirklich parallelen Betrieb. Wir geben wieder je ein Konfigurationsbeispiel für eine Einfach- und für eine Mehrfachbuszusammenschaltung an.

Beispiel 6.4 Anschluß peripherer Geräte für serielle Verarbeitung

Bild 6.8 zeigt einen Ein-Ausgabebus, der einen Ein-Ausgabeprozessor (IOU) mit mehreren Ein-Ausgabe-Steuereinheiten von Peripheriegeräten (PCU) verbindet. An jede Steuereinheit sind entweder ein oder mehrere Geräte angeschlossen. Wegen der sternförmigen Zusammenschaltung ist zu einem Zeitpunkt nur die Verbindung einer der Steuereinheiten mit dem Ein-Ausgabeprozessor möglich. Die Organisation erfolgt je nach Prozessororganisation im Selektor-, Blockmultiplex-, oder Bytemultiplexbetrieb.

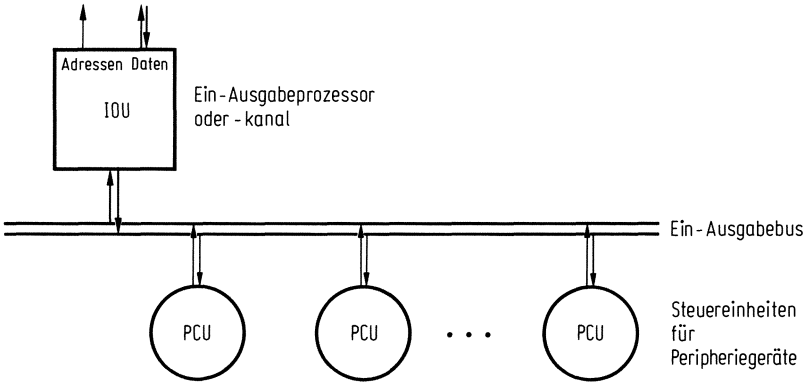


Bild 6.8 Datenübertragung zwischen einem Ein-Ausgabeprozessor oder -kanal und Ein-Ausgabegeräten auf Einfachbussen

Beim Selektorbetrieb wird die Verbindung zwischen dem Ein-Ausgabeprozessor und einer Steuereinheit während des gesamten Ein-Ausgabevorganges aufrechterhalten. Der Datenbus kann selbst dann nicht für Ein-Ausgabevorgänge mit anderen Steuereinheiten verwendet werden, wenn z.B. beim Einstellen eines Plattenarmes Wartezeiten auftreten, in denen keine Daten übertragen werden. Der Selektorbetrieb erlaubt eine sehr schnelle Datenübertragung und eignet sich für den Anschluß schneller Peripheriegeräte.

Beim Blockmultiplexbetrieb kann im Gegensatz zum Selektorbetrieb ein Ein-Ausgabevorgang unterbrochen werden, z.B. wenn durch die Suchbewegung eines Platten-

armes Wartezeiten hervorgerufen werden. Die Kanalstatusinformation wird im Ein-Ausgabeprozessor und in der Steuereinheit gespeichert. Der Ein-Ausgabevorgang wird fortgesetzt, wenn die unterbrochene Einheit für den Datentransfer bereit und der Kanal wieder frei ist. Ein Blockmultiplexkanal kann auch im Selektorbetrieb arbeiten. Er ist für den Anschluß schneller Peripheriegeräte geeignet.

Beim Bytemultiplexbetrieb werden wie beim Blockmultiplexbetrieb mehrere Ein-Ausgabevorgänge zeitlich verzahnt ausgeführt. Unterbrechungen sind jeweils nach der Übertragung eines Datenbytes möglich. Die Statusinformation der unterbrochenen Ein-Ausgabevorgänge wird in sog. Kanalregistern gespeichert. Der Bytemultiplexkanal eignet sich für den Anschluß langsamer Peripheriegeräte.

Die in Bild 6.8 dargestellte Konfiguration findet man bei den meisten heute gebräuchlichen Rechenanlagen. Sind in einer Anlage sehr viele Peripheriegeräte vorhanden, so werden oft mehrere Ein-Ausgabekanäle oder -prozessoren benutzt, wobei jeder Kanal bzw. Prozessor mit bestimmten Ein-Ausgabegeräten verbunden ist.[1] Die Kanäle bzw. Prozessoren sind entsprechend der unterschiedlichen Geschwindigkeiten der einzelnen Ein-Ausgabegeräte für Selektor-, Blockmultiplex- oder Bytemultiplexbetrieb ausgelegt; man spricht von Selektor-, Blockmultiplex- oder Bytemultiplexkanälen oder -prozessoren. ■

Beispiel 6.5 Anschluß peripherer Geräte für parallele Verarbeitung

Der Nachteil der im Beispiel 6.8 geschilderten Konfigurationen ist, daß mehrere Peripheriegeräte nicht gleichzeitig bedient werden können, da sie an ein und demselben

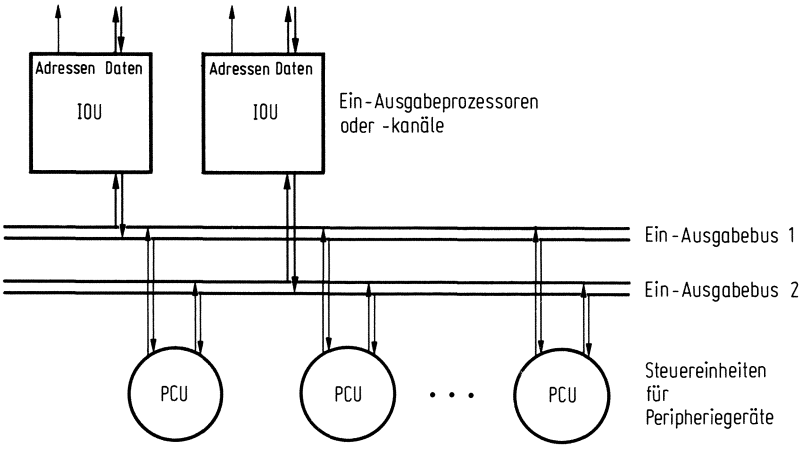


Bild 6.9 Datenübertragung zwischen Ein-Ausgabeprozessoren oder -kanälen und Ein-Ausgabegeräten auf Mehrfachbussen

[1] Vgl. z.B. Konfiguration einer XDS-SIGMA-7-Anlage.

Bus angeschlossen sind. Abhilfe schafft hier ein Ein-Ausgabesystem nach Bild 6.9, bei dem wegen der matrixförmigen Zusammenschaltung jede Steuereinheit mit einem der gerade verfügbaren Ein-Ausgabeprozessoren gekoppelt werden kann.

Die in Bild 6.9 dargestellte Konfiguration setzt Selektorprozessoren voraus, damit ein und dasselbe Ein-Ausgabegerät während der Datenübertragung nicht mit einem anderen Prozessor verbunden werden kann.[1] Wegen der "fließenden" Zuordnung der Prozessoren und Steuereinheiten spricht man von Floating Channels (s. Abschnitt 6.4). ■

6.3 Programmunterbrechungen

In Abschnitt 6.1 hatten wir Sensorleitungen benutzt, mit denen der Zustand eines Peripheriegeräts abgefragt wurde (vgl. Beispiel 6.1). Die Verwendung des Passiv bei der Formulierung dieses Satzes soll die Passivität der Peripherie unterstreichen: Der Zeitpunkt der Reaktion des Prozessors auf ein Signal aus der Peripherie wird durch Ausführung eines TST-Befehls im laufenden Programm bestimmt. Eine Sensorleitung wird erst dann abgefragt, wenn das Programm am Zustand der Peripherie interessiert ist.

Eine ganz andere Situation entsteht, wenn die Peripherie ihre Passivität aufgibt und in Aktivität umkehrt. Dann bestimmt die Peripherie den Zeitpunkt der Reaktion des Prozessors auf ein Signal aus der Peripherie, und das laufende Programm wird unterbrochen. Man nennt dies eine Programmunterbrechung (interrupt) und die Leitungen, die eine Unterbrechung hervorrufen können, Unterbrechungsleitungen (interrupt lines). Erst durch die Möglichkeit der Programmunterbrechungen kann der Prozessor nicht nur als zentrales, sondern auch als untergeordnetes Element eines Prozesses eingesetzt werden, z.B. im Kernkraftwerk zur Überwachung der physikalischen Reaktion der Kernspaltung oder im Flugtrainer zur Simulation der Landung eines Flugzeuges. Solche Rechner sind in der Lage, sofern sie schnell genug sind, einem Prozeß in seinem "wirklichen" Zeitablauf zu folgen. Systeme mit diesen Eigenschaften werden Realzeitsysteme genannt. Man findet sie überall dort, wo Steuerungen oder Regelungen auftreten, die nicht mit beliebig langsamer Geschwindigkeit simuliert werden können, z.B. bei "maschinellen" Prozessen wie der Reaktorsteuerung oder bei "menschlichen" Prozessen wie der Flugsimulation.

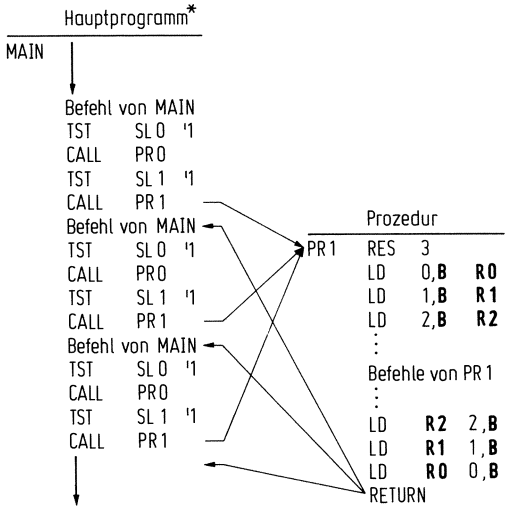
Sensor- und Unterbrechungsleitungen

Auch mit Sensorleitungen kann prinzipiell auf Signale aus der Peripherie unmittelbar reagiert werden, jedoch auf Kosten von Speicherplatz und Programmausführungszeit.

---

[1] Vgl. typische Konfigurationen von B 6700-Anlagen von Burroughs.

Je nach geforderter Reaktionszeit sind TST- und CALL-Befehle nach jeweils einer Gruppe von mehreren Befehlen oder nach jedem Befehl einzufügen, mit denen die einzelnen Sensorleitungen ausgewertet werden. Das folgende Schema veranschaulicht drastisch den Mehrbedarf an Speicherplatz und Ausführungszeit bei der Auswertung von zwei Sensorleitungen SL0 und SL1 innerhalb des Hauptprogrammes MAIN, denen zwei Unterprogramme PR0 und PR1 zugeordnet sind.



Dieser äußerst unökonomischen Softwarelösung mit Sensorleitungen steht die wesentlich ökonomischere Hardwarelösung mit Unterbrechungsleitungen gegenüber. Das Abfragen der Sensorleitungen nach einem jeden Befehl des Programms wird in das Mikroprogramm eines jeden Befehls verlegt, und anstelle der auf jeden TST-Befehl folgenden Speicherzelle wird jeder Leitung eine Speicherzelle zugeordnet, in die der CALL-Befehl eingetragen wird. Auf diese Weise werden Sensorleitungen zu Unterbrechungsleitungen. Eine Eins auf einer Unterbrechungsleitung bewirkt die Unterbrechung des laufenden Programms zwischen dem Ende der Ausführung eines Befehls und dem Beginn des Abrufes des auf ihn folgenden Befehls. Anstatt diesen nächsten Befehl auszulesen, wird die der Unterbrechungsleitung zugeordnete Speicherzelle ausgelesen und der darin enthaltene CALL-Befehl ausgeführt. Der Befehlszähler- und der Basisregisterstand des unterbrochenen Programms werden in den dynamischen Speicher[1] gebracht, und anschließend wird in dem Unterprogramm fortgefahren, das der Unter-

---

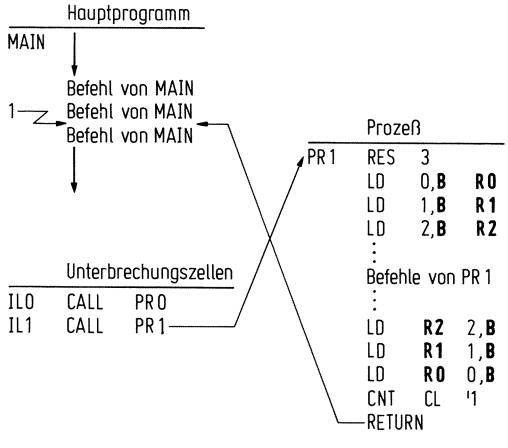
* Um nicht noch mehr Organisationsbefehle zu benötigen, werden die Prozeduren PR0 und PR1 ausgeführt, wenn auf den zugeordneten Sensorleitungen SL0 bzw. SL1 nicht Einsen, sondern Nullen erscheinen.

[1] Vielfach ist es üblich, den Zustand des Prozessors in einen dem Unterprogramm zugeordneten statischen Speicherbereich zu retten.

brechung zugeordnet ist. Die Eins auf der Unterbrechungsleitung muß gespeichert werden und darf erst dann gelöscht werden, wenn das Unterprogramm beendet ist und der Prozessor in den Zustand gebracht worden ist, den er vor der Programmunterbrechung hatte. Das Löschen der gespeicherten Eins ist die letzte Aktion vor dem Unterprogramm-Rücksprung. Dazu wird der Befehl CNT CL '1 benutzt, der für die Dauer seiner Ausführung einen Impuls auf einer Eingangsleitung des Unterbrechungssystems erzeugt (Steuerleitung CL).

Wir nennen Unterprogramme, die aufgrund von Unterbrechungssignalen ausgeführt werden, Prozesse, um sie begrifflich von Unterprogrammen unterscheiden zu können, die aufgrund eines Prozeduraufrufes ausgeführt werden.

Im folgenden Schema ist gezeigt, wie Signale aus der Peripherie durch Unterbrechungsleitungen unmittelbar ausgewertet werden. Wir setzen voraus, daß den Unterbrechungsleitungen IL0 und IL1 die beiden absoluten Primärspeicheradressen IL0 und IL1 zugeordnet sind, in denen die Befehle zum Aufruf der beiden Prozesse PR0 und PR1 gespeichert sind. Eine Eins auf Leitung IL1 bewirkt, daß zwischen zwei Be-



fehlen von MAIN der Prozess PR1 ausgeführt wird. Die Reaktionszeit des Prozessors auf dieses Signal wird durch die Ausführungszeit eines Befehls von MAIN plus den Ausführungszeiten der LD-Befehle für das Retten der Registerinhalte bestimmt. Ist selbst diese kurze Zeitspanne im Realzeitbetrieb zu lang, so müssen Rechner mit mehreren Registerspeichern benutzt werden, so daß das Retten und Laden der Registerinhalte durch das Auswählen des zu einer Unterbrechung gehörenden Registerspeichers ersetzt werden kann.

Bei der Auswertung von Sensorleitungen werden im Hauptprogramm alle Leitungen der Reihe nach abgefragt. Im vorn angegebenen Schema kommt SL0 eine höhere

Priorität zu als SL1, denn PR0 wird vor PR1 ausgeführt, wenn sowohl SL0 als auch SL1 gleich Eins sind. Es ist jedoch nicht möglich, wenn SL0 nach SL1 Eins wird, trotz höherer Priorität von SL0 die SL1 zugeordnete Prozedur PR1 zu unterbrechen. Dazu müßten innerhalb von PR1 wiederum nach jedem Befehl von PR1 TST-SL0-Befehle eingefügt werden, was die Software-Lösung noch ineffizienter machen würde.

Bei der Auswertung von Unterbrechungssignalen wird die Reihenfolge der Abfrage der Unterbrechungsleitungen in den Mikroprogrammzyklus übernommen. Die Hardware benötigt dazu ein Prioritätenschaltnetz, das das Unterbrechungssignal auf der Leitung mit dem niedrigsten Index durchschaltet und alle Signale auf Leitungen mit höherem Index blockiert. Niedrige Indizes der Unterbrechungsleitungen entsprechen höheren Prioritäten der Unterbrechungssignale.

Unterbrechungssystem. Bild 6.10 zeigt in einer groben Darstellung die Struktur des Unterbrechungssystems mit dem Prioritätenschaltnetz und zwei Flipflopregistern als

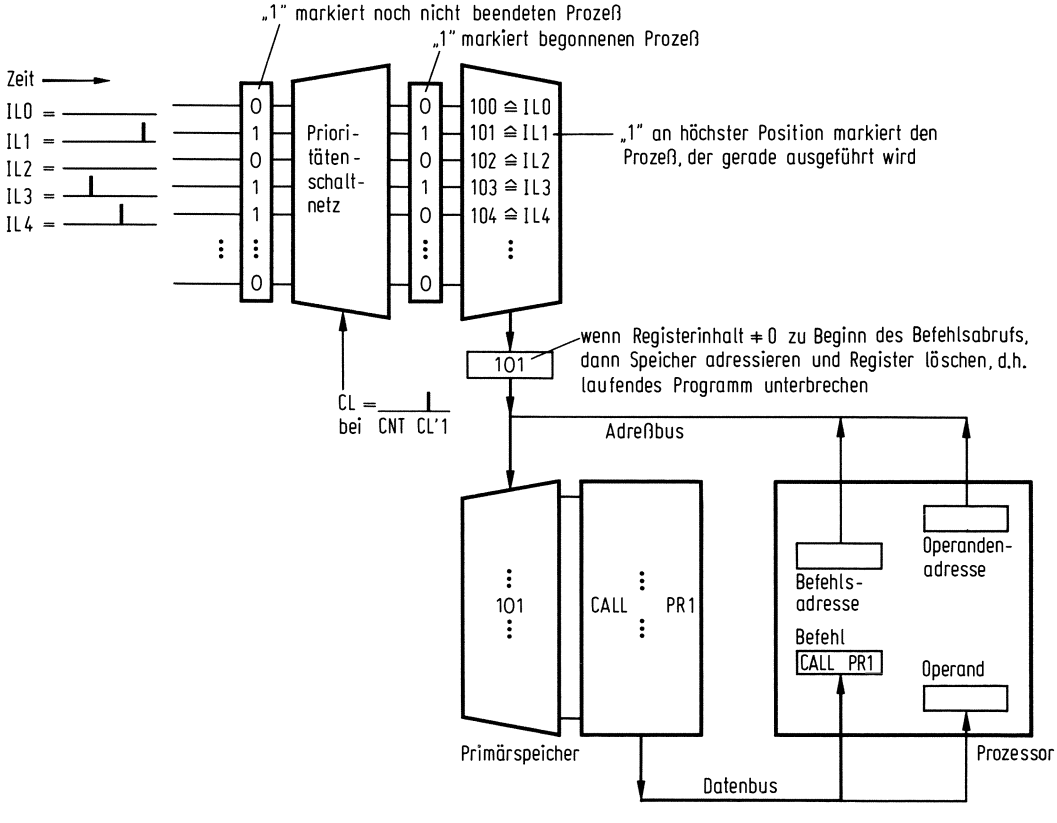
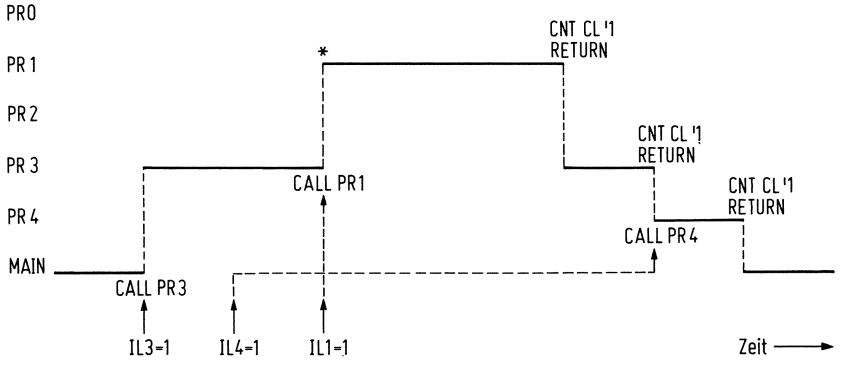


Bild 6.10 Unterbrechungssystem mit Prioritätensteuerung

Kernstück. Eine bestimmte Unterbrechungssituation ist als Beispiel eingetragen: Das später als IL4=1 und IL3=1 erschienene Unterbrechungssignal IL1=1 unterbricht den Prozeß PR3 und erzwingt einmalig das Auslesen des unter der absoluten Adresse 101 entsprechend IL1 gespeicherten Wortes CALL PR1 und dessen Interpretation als Befehl. Dies wird veranlaßt durch die Eins am Ausgang des Prioritätenschaltnetzes in Verbindung mit der Null in dem dem Schaltnetz nachgeschalteten Register, die danach auf Eins wechselt. Wird im Prozeß PR1 der Befehl CNT CL '1 ausgeführt, so werden die höchsten Einsen in den Registern vor und nach dem Prioritätenschaltnetz gelöscht und dadurch am Ausgang des Schaltnetzes die nächst tiefere Eins markiert, was ohne Wirkung bleibt, da im rechten Flipflop bereits eine Eins steht. Durch den Befehl RETURN wird im Prozeß PR3 mit dem nächsten Befehl nach der Unterbrechung fortgefahren. Wird im Prozeß PR3 der Befehl CNT CL '1 ausgeführt, so wird am Ausgang des Prioritätenschaltnetzes wiederum die nächst tiefere Eins markiert. Da aber im rechten Flipflop eine Null steht, wird der Speicher mit 104 entsprechend IL4 adressiert, der Befehl CALL PR4 ausgelesen und der Prozeß PR4 ausgeführt. Der Befehl CNT CL '1 im Prozeß PR4 löscht die Flipflops vor und nach dem Prioritätenschaltnetz, und die RETURN-Befehle bewirken den Rücksprung ins Hauptprogramm. Bei der Schilderung des Ablaufs dieser Unterbrechungssituation ist vorausgesetzt, daß nach IL1=1 kein weiteres Unterbrechungssignal eintrifft.

Das nacheinander erfolgte Eintreffen von mit Prioritäten versehenen Unterbrechungssignalen wird genau abgebildet auf das nacheinander erfolgende Ausführen der zugeordneten Prozesse. Die Programmorganisation entspricht der baumstrukturierten Organisation geschachtelter Prozeduren, und die Speicherorganisation entspricht der pulsierenden Organisation des dynamischen Speichers. Das folgende Schema zeigt, wie drei in der Reihenfolge IL3=1, IL4=1 und IL1=1 eintreffende Unterbrechungssignale durch drei Prozesse PR1, PR3 und PR4 abgearbeitet werden. Der in der Zeichnung eingetragene Stern markiert den Zeitpunkt, der der in Bild 6.10 dargestellten Unterbrechungssituation entspricht.



In Bild 6.10 sieht man, wie im Register nach dem Prioritätenschaltnetz diejenigen Prozesse gekennzeichnet sind, die begonnen wurden, und wie im Register vor dem Prioritätenschaltnetz diejenigen Prozesse gekennzeichnet sind, die noch nicht beendet worden sind. Dabei ist vorausgesetzt, daß die Befehlsfolge CNT CL '1, RETURN nicht unterbrechbar ist. - Vielfach findet man in ausgeführten Rechenanlagen spezielle Mikroprogramme oder Befehle für den Aufruf bzw. die Rückkehr aus Prozessen. Neben solchen CALL- und RETURN-Befehlen sind auch spezielle Befehle für das Unterbrechungssystem, z.B. ENABLE- und DISABLE-Befehle zum Ein- und Ausschalten des gesamten Unterbrechungssystems und ARM- und DISARM-Befehle zum Ein- und Ausschalten bestimmter Unterbrechungsleitungen üblich. Schließlich gibt es auch sog. Einzelwort-Programmunterbrechungen, die nach der Ausführung eines einzelnen Befehls bereits erledigt sind und bei denen das Löschen der Unterbrechungsflipflops von der Hardware ausgeführt wird. Im folgenden Beispiel geben wir zwei Prozesse an, die die Betriebsorganisation einer Rechenanlage unterstützen.

Beispiel 6.6 Netzausfall

Unmittelbare Reaktionen eines Prozessors sind insbesondere als Antwort auf "Notstands-Situationen" erforderlich, die - wenn nicht innerhalb kürzester Zeit reagiert

```
        aorg   100          Inhalt der Zellen 100 und 101
        ext    LOAD
FAIL    CALL   SAVE         Rufe Unterprogramm SAVE
RETN    GOTO   LOAD         Weiter bei Unterprogramm LOAD
        end
```

```
        org                Definition des Netzausfallprozesses
        ent    SAVE
        ent    LOAD
X       res    1
SAVE    RES    3
        LD     0,B    R0   Retten der Registerinhalte in den
        LD     1,B    R1   dynamischen Speicher
        LD     2,B    R2
         :
        LD     X      B
        HLT
LOAD    LD     B      X
         :
        LD     R2     2,B  Laden der Register aus dem
        LD     R1     1,B  dynamischen Speicher
        LD     R0     0,B
        CNT    CL     '1   Unterbrechung löschen
        RETURN             Rücksprung
        end
```

wird - zur "Katastrophe" führen können. Auch rechnerintern können solche Situationen entstehen, z.B. bei Ausfall des Stromnetzes. Werden für diesen Fall nicht innerhalb kürzester Zeit die flüchtigen Speicherinhalte der Maschine in einen nichtflüchtigen Speicher* gerettet, so ist bei der Wiederkehr des Stromes die weitere Ausführung des Programmes nicht möglich.

Wir sehen für Stromausfall höchste und für Stromwiederkehr zweithöchste Priorität vor. Die den Unterbrechungsleitungen zugeordneten Speicherzellen haben die absoluten symbolischen Adressen FAIL und RETN mit ihren festen numerischen Entsprechungen 100 und 101. Den beiden Unterbrechungsleitungen sind zwei Programmstücke SAVE und LOAD zugeordnet. Im Programm auf S. 249 wird eine aorg-Anweisung anstelle der sonst üblichen org-Anweisung benutzt, da ein Teil des Codes durch den verschiebender Lader nicht irgendwohin, sondern an eine ganz bestimmte Stelle im Speicher geladen werden muß. - Würde auch der Netzschalter an dieses Unterbrechungssystem angeschlossen, so könnte der Prozessor beim Einschalten dort fortfahren, wo er beim Ausschalten unterbrochen wurde. Dies gilt naturgemäß nicht für unterbrochene Ein-Ausgabevorgänge oder andere in Realzeit ablaufende Vorgänge, so daß das Betriebssystem der Anlage an der Auswertung der Unterbrechungen beteiligt werden muß. ■

Ein-Ausgabeprogramme

In Beispiel 6.1 ist ein einfaches Ausgabeprogramm angegeben, in dem eine Sensorleitung verwendet wird, um auf die Beendigung des Druckvorganges des Ein-Ausgabegerätes zu warten. Nehmen wir an, daß die Schreibmaschine für das Drucken eines Zeichens 0,1 s und der Rechner für einen Speicherzyklus 1 µs benötigen. Dann durchläuft der Prozessor die Befehle der Ausgabeschleife in einem Bruchteil der Zeit, die zum Drucken eines Zeichens notwendig ist. In der verbleibenden Zeitspanne wartet der Prozessor. Verwendet man statt der Sensorleitung eine Unterbrechungsleitung, so kann der Prozessor nach dem Starten des Ausgabevorganges im Hauptprogramm fortfahren und wird alle 0,1 s vom Ein-Ausgabegerät unterbrochen, das die Bereitschaft zum Drucken eines weiteren Zeichens signalisiert. Bei jeder Unterbrechung wird das Ausgabeprogramm einmal durchlaufen, und es wird ein Zeichen ausgegeben. Nach der Ausgabe des letzten Zeichens wird der Ein-Ausgabevorgang gestoppt, wodurch das Unterbrechungssignal unterdrückt wird, und es wird ohne Unterbrechung im Hauptprogramm fortgefahren.

Für das Starten und das Stoppen von Ein-Ausgabevorgängen definieren wir einen Makrobefehl INOUT, der wie die früher definierten Systemmakros auch als Maschinenbe-

---

* Z.B. Kernspeicher als Primärspeicher oder Plattenspeicher als Sekundärspeicher.

fehl ausgeführt sein kann. Beim Starten eines Ein-Ausgabevorganges wird zunächst ein Wort an die Peripherie übertragen, das Startinformationen für den Betrieb des Ein-Ausgabegerätes enthält, z.B. "Drucken auf Schreibmaschine mit der Geräteadresse 10". Anschließend wird ein Wort aus der Peripherie hereingeholt und getestet. Wenn der Zustand des Gerätes keine ordnungsgemäße Datenübertragung erlaubt, wird die Ausführung eines Diagnose-Programms eingeleitet. Andernfalls wird der auf INOUT folgende Befehl ausgeführt, und das Gerät antwortet mit einem Unterbrechungssignal, das die Ausführung des Ein-Ausgabeprogramms veranlaßt. Beim Stoppen des Ein-Ausgabevorganges werden dem Ein-Ausgabegerät Stopp-Anweisungen übergeben, z.B. "Ende des Druckens auf Schreibmaschine mit der Geräteadresse 10". Wie beim Starten wird auch hier ein Wort aus der Peripherie in den Prozessor geholt und getestet. Wenn die Datenübertragung fehlerhaft war, wird wiederum das Diagnose-Programm aufgerufen. Anderenfalls wird der auf INOUT folgende Befehl ausgeführt.

Makrobefehl INOUT. Wir fassen die für das Starten und Stoppen eines Ein-Ausgabevorganges notwendigen Befehle zum Systemmakro INOUT zusammen.

Makroaufruf: INOUT P1 P2
P1 : Geräteadresse
P2 : Steuerwort

Der Makrobefehl INOUT startet oder stoppt den Ein-Ausgabevorgang mit einem Peripheriegerät. Der erste Parameter von INOUT gibt die Geräteadresse an, und der zweite Parameter enthält ein Wort mit Steuerinformation. Ist beim Starten das Peripheriegerät nicht betriebsbereit oder stellt man nach dem Stoppen fest, daß die Datenübertragung fehlerhaft war, wird die Prozedur DIAGN aufgerufen.

```
INOUT   makro   P1     P2    Starten und Stoppen
        CNT     P1     'P2   Steuerwort ausgeben
        TST     P1     '0    wenn Testwort
        CALL    DIAGN        < 0, dann rufe Diagnose-Prozedur
        mend
```

Im folgenden Beispiel wird die Programmierung eines Ein- und Ausgabevorganges unter Verwendung einer Sensor- und einer Unterbrechungsleitung gegenübergestellt, wobei auch der Makrobefehl INOUT benutzt wird.

Beispiel 6.7 Schreibmaschinenausgabe

Wir greifen die in Beispiel 6.1 behandelte Ausgabe eines Textes auf der Schreibmaschine auf und stellen zwei Programme gegenüber, die eine Sensorleitung zur Abfrage bzw. eine Unterbrechungsleitung zur Übermittlung des Gerätezustands benutzen. Wir ver-

wenden anstelle des Registers **A** die Speicherzelle BB (Bytepuffer), anstelle des Registers **I** die Speicherzelle WC (Wortzähler), anstelle des Registers **J** die Speicherzelle BC (Bytezähler), und eine weitere Speicherzelle WP (Wortzeiger). Wie in Beispiel 6.1 schreiben wir den Aufruf der Ausgabeprogramme als Makrobefehle.

Der hauptsächliche Unterschied zwischen diesen beiden Programmen ist, daß das Sensorleitungs-Programm den Prozessor für die Dauer des gesamten Ausgabevorganges belegt, während das Unterbrechungsleitungs-Programm den Prozessor nur für die Dauer der Übertragung und Auswertung der einzelnen Zeichen beansprucht. In der Zeit, in der der Prozessor im ersten Fall auf das Sensorsignal wartet, kann

Ausgabeprozedur (Prozessor-Unterprogramm mit Sensorleitung)

```
TYPE   makro   P1    P2     Definition des Makrobefehls TYPE
       ext     WC
       ext     WP
       ext     BC
TY     equ     10           Geräte-Adresse Schreibmaschine
       LD      WC    'P1    lade Wortzähler
       LD      WP    'P2    lade Wortzeiger
       LD      BC    '2     lade Bytezähler
STRT   equ     1            Startinformation
       INOUT   TY    STRT   Start Ausgabe
       CALL    TYP          Prozeduraufruf
       mend
```

```
       org                  Definition der Ausgabeprozedur TYP
       ent     WC
       ent     WP
       ent     BC
       ent     TYP
TY     equ     10           Geräte-Adresse Schreibmaschine
WC     res     1            Wortzähler
WP     res     1            Wortzeiger
BC     res     1            Bytezähler
BB     res     1            Bytepuffer
TYP    LD      BB    *WP    lade Bytepuffer mit Speicherwort
       SUB     WC    '1     vermindere Wortzähler
       ADD     WP    '1     erhöhe Wortzeiger
L      TST     TY    '1     wenn Sensorleitung TY
       GOTO    L            =0, dann zurück zu Marke L
       OUT     BB    TY     =1, dann gib aus Bytepuffer aus
       SUB     BC    '1     vermindere Bytezähler
       SA      BB    '8     schifte Bytepuffer um ein Byte
       CMP     '0    BC     wenn Bytezähler
       GOTO    L            >0, dann zurück zu Marke L
       LD      BC    '2     =0, dann setze Bytezähler
       CMP     '0    WC     wenn Wortzähler
       GOTO    TYP          >0, dann zurück zur Marke TYP
STOP   equ     3            Stoppinformation
       INOUT   TY    STOP   Stopp Ausgabe
       RETURN               Rückkehr ins aufrufende Programm
       end
```

er im zweiten Fall im Hauptprogramm fortfahren und z.B. auf einer zweiten Schreibmaschine durch einen zweiten Ausgabeprozeß Text ausdrucken. Für den Außenstehenden arbeiten beide Schreibmaschinen parallel, während sie in Wirklichkeit vom Prozessor nacheinander bedient werden. Man spricht von scheinbar paralleler Arbeitsweise (Zeitmultiplexbetrieb).

Ausgabeprozeß (Prozessor-Unterprogramm mit Unterbrechungsleitung)

```
TYPE   makro   P1    P2     Definition des Makrobefehls TYPE
       ext     WC
       ext     WP
       ext     BC
TY     equ     10           Geräte-Adresse Schreibmaschine
       LD      WC    'P1    lade Wortzähler
       LD      WP    'P2    lade Wortzeiger
       LD      BC    '2     lade Bytezähler
STRT   equ     1
       INOUT   TY    STRT   Start Ausgabe
       mend
```

Speicherzelle für Unterbrechungsleitung TY

```
CALL     TYP          wenn Unterbrechungsleitung TY=1, dann Prozeduraufruf TYP
```

```
       org                  Definition des Ausgabeprozesses TYP
       ent     WC
       ent     WP
       ent     BC
       ent     TYP
TY     equ     10           Geräte-Adresse Schreibmaschine
WC     res     1            Wortzähler
WP     res     1            Wortzeiger
BC     res     1            Bytezähler
BB     res     1            Bytepuffer
TYP    CMP     BC    '2     wenn Bytezähler
       GOTO    L            <2, dann weiter bei Marke L
       LD      BB    *WP    =2, dann lade Bytepuffer mit Wort
       SUB     WC    '1             vermindere Wortzähler
       ADD     WP    '1             erhöhe Wortzeiger
L      OUT     BB    TY     gib aus Bytepuffer aus
       SUB     BC    '1     vermindere Bytezähler
       SA      BB    '8     schifte Bytepuffer um ein Byte
       CMP     '0    BC     wenn Bytezähler
       GOTO    FIN          >0, dann weiter bei Marke FIN
       LD      BC    '2     =0, dann setze Bytezähler
       CMP     '0    WC     wenn Wortzähler
       GOTO    FIN          >0, dann weiter bei Marke FIN
STOP   equ     3
       INOUT   TY    STOP   =0, dann Stopp Ausgabe
FIN    CNT     CL    '1     lösche Unterbrechung
       RETURN               Rückkehr ins unterbrochene Programm
       end
```

■

Reentrante Prozeduren

Wenn - wie eben geschildert - zwei Ein-Ausgabevorgänge im Zeitmultiplexbetrieb ablaufen, so kann sich folgende in Bild 6.11 dargestellte Situation ergeben: Der eine Ein-Ausgabeprozeß ruft (aufgrund einer Fehlermeldung vom ersten Ein-Ausgabegerät) die Diagnose-Prozedur auf (1). Der andere Ein-Ausgabeprozeß unterbricht die Ausführung der Prozedur (2) und ruft (aufgrund einer Fehlermeldung vom zweiten Ein-Ausgabegerät) die Diagnose-Prozedur erneut auf (3). Die Prozedur wird also vom zweiten Prozeß aufgerufen, obwohl sie vom ersten Prozeß nicht vollständig ausgeführt wurde. Damit ist eine ähnliche Situation wie beim rekursiven Aufruf von Prozeduren entstanden: Ein und dieselbe Prozedur wird aufgerufen, obwohl sie noch nicht beendet ist. Als Prozedur des ersten Prozesses befindet sie sich in einem statischen und als Prozedur des zweiten Prozesses in einem dynamischen Zustand der Ausführung (4). Nach der Ausführung der Prozedur wird in den zweiten Prozeß zurückgesprungen (5), und nach dessen Ausführung kehrt der Prozessor an die Stelle der Prozedur zurück, an der sie früher unterbrochen wurde (6). Die Prozedur muß zu diesem Zeitpunkt ihren früheren Zustand vorfinden, um ordnungsgemäß weiter ausgeführt und beendet werden zu können (7). Man nennt Prozeduren, die dieser Bedingung gehorchen, wiedereintrittsfeste oder reentrante Prozeduren.

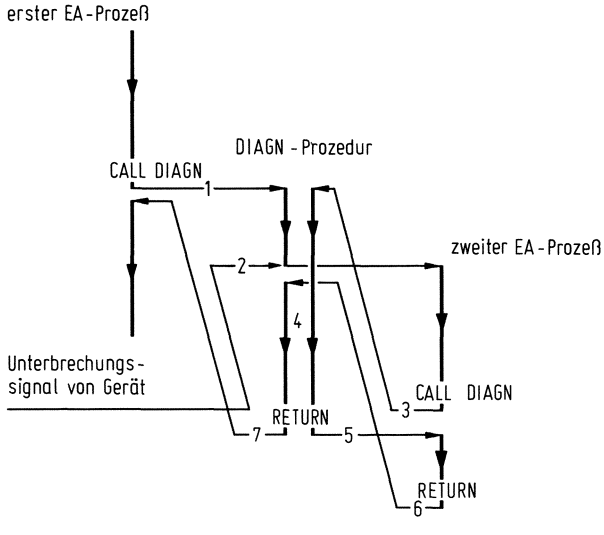


Bild 6.11 Zweimaliges Aufrufen einer reentranten Prozedur am Beispiel des Diagnose-Programms

Um den Zustand der Prozedur zu erhalten, dürfen die Daten der Prozedur beim Wiedereintritt nicht überschrieben werden. Das kann dadurch erreicht werden, daß der Speicher dynamisch organisiert wird, da die Datenbereiche der Prozedur dann so oft existieren wie "reentrante Aufrufe" erfolgen. In Bild 6.12 sieht man links, wie der Prozeß P0, nachdem er die Prozedur PR aufgerufen hat, durch den Prozeß P1 unterbrochen wird und wie dieselbe Prozedur PR vom Prozeß P1 aus erneut aufgerufen wird. Rechts ist der dynamisch organisierte Speicher dargestellt. Die eingetragenen Ziffern markieren jeweils die erste Zelle des freien Speicherbereiches und sind den im linken Bild eingetragenen Zeitpunkten zugeordnet. Zwischen den Punkten 1 und 2 ist der Speicherbereich für die von der Prozedur benutzten Daten des ersten Aufrufs reserviert, und zwischen den Punkten 3 und 4 ist der Speicherbereich für die von der Prozedur benutzten Daten des zweiten Aufrufs reserviert. Aufgrund der dynamischen Speicherorganisation sind diese Bereiche doppelt vorhanden. Man sieht, wie der erste Bereich dem Prozeß P0 und der zweite Bereich dem Prozeß P1 zugeordnet ist.

Der in Bild 6.12 gezeigte dynamische Speicher setzt eine Programmorganisation voraus, bei der nur in dem als erstem auszuführenden Prozeß ein ORG-Befehl gegeben wird. Dementsprechend haben wir einen einzigen zusammenhängenden dynamischen Speicherbereich für alle Prozesse vor uns. Werden in mehreren Prozessen ORG-Befehle gegeben, so entstehen mehrere unzusammenhängende dynamische Speicherbereiche, und im Extrem hat jeder Prozeß eigene statische und dynamische Speicherbereiche.

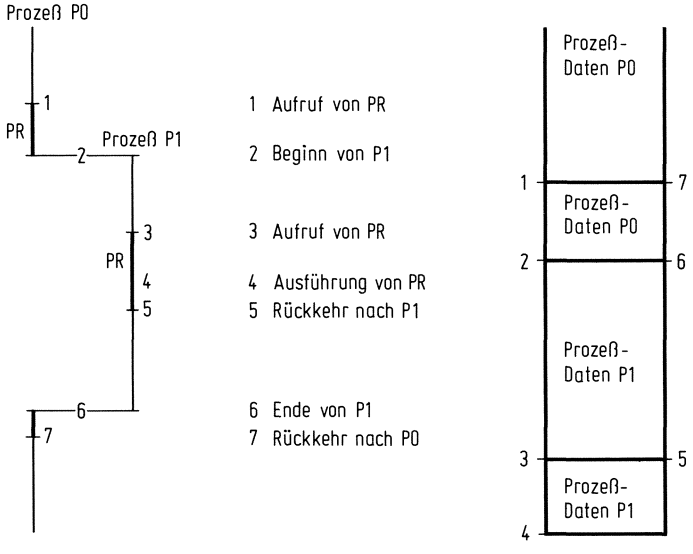


Bild 6.12 Zeitlicher Ablauf und pulsierende Speicherbelegung bei zweimaligem Aufruf einer reentranten Prozedur

Eine reentrante Prozedur kann also von mehreren Prozessen benutzt werden. Ihre unveränderlichen Bestandteile (das Programm und die Konstanten) brauchen nur einmal im Speicher zu stehen, aber ihre veränderlichen Teile (der Status und die Variablen) müssen so oft gespeichert werden, wie die Prozedur unterbrochen und aufgerufen wird. Dies wird einsichtiger, wenn die veränderlichen Bestandteile der Prozedur als Daten der Prozesse und die unveränderlichen Bestandteile der Prozedur als Code betrachtet werden, der von mehreren Prozessen gemeinsam benutzt wird. Mehrere Prozesse "teilen" sich beim wiederholten Aufruf unterbrochener Prozeduren in den statisch gespeicherten Code (shared code), ähnlich wie sich mehrere Programme beim Aufruf von Prozeduren mit extern definierten Parametern in die statisch gespeicherten Daten teilen (shared data). Reentrante Prozeduren können keine solche statischen Parameter benutzen.

Reentrante Prozeduren unterscheiden sich von rekursiven Prozeduren dadurch, daß der Wiedereintritt in die Prozedur vom Programmierer nicht festgelegt und dementsprechend der Wiederaufruf der Prozedur auch nicht programmiert ist. Bei rekursiven Prozeduren wird die Stelle und damit der Zeitpunkt des Wiederaufrufs durch das Auftreten des CALL-Befehls vom Programmierer bestimmt, während bei reentranten Prozeduren der Zeitpunkt des Wiedereintritts vom Unterbrechungssystem bestimmt wird und damit unvorhersagbar ist. Prozeduren werden rekursiv geschrieben, wenn das zu lösende Problem rekursiver Natur ist. Prozeduren werden reentrant geschrieben, wenn sie von mehreren Prozessen benutzt werden. Da eine reentrante im Gegensatz zu einer rekursiven Prozedur nach jedem Befehl unterbrochen und erneut aufgerufen werden kann, können sich reentrante und rekursive Prozeduren in ihrer Programmorganisation unterscheiden. Bei reentranten Prozeduren muß immer erst Speicherplatz dynamisch reserviert werden, bevor er mit Daten belegt wird, und er darf

erst freigegeben werden, nachdem er von weiterbenötigten Daten geräumt wurde. Die in Abschnitt 4.4 eingeführten Makrobefehle ORG, CALL und RETURN sind bezüglich der Organisation des dynamischen Speichers für den Aufruf reentranter Prozeduren definiert. Reentrante Organisation schließt rekursiven Aufruf von Prozeduren ein, während umgekehrt rekursive Organisation reentranten Aufruf von Prozeduren nicht einzuschließen braucht. Im folgenden Beispiel wird dieser Unterschied in den Details der Maschinenprogrammierung deutlich.

Beispiel 6.8 Fakultät in reentranter und rekursiver Organisation

Wir betrachten die rekursive Prozedur FAK entsprechend Beispiel 4.16, die von zwei verschiedenen Prozessen aufgerufen wird.

Prozeß P0

```
      :
X     res 1
      :
      CALL  FAK 1
      dat   X
      :
```

Prozedur FAK

```
FAK   CMP     0,B   '1
      NOP
      GOTO    FIN
      RES     1
      LD      1,B   0,B
      SUB     0,B   '1
      CALL    FAK   1
      dat     0,B
      MUL     0,B   1,B
FIN   RETURN
```

Prozeß P1

```
      :
X     res 1
      :
      CALL  FAK 1
      dat   X
      :
```

Nachfolgend ist links die Erweiterung von CALL FAK 1 angegeben, die der in Abschnitt 4.4 gegebenen Definition des Makrobefehls CALL folgt und die sowohl für rekursive wie für reentrante Prozeduren benutzt werden kann. Rechts ist eine andere Erweiterung gezeigt, die einer Definition von CALL folgt, die zwar für rekursive, aber nicht für reentrante Prozeduren benutzt werden kann.

```
ADD   F     '3
LD    -5,F  '5,P
LD    -4,F  B
LD    -3,F  '*2,P
LD    B     '-1,F
LD    P     'FAK
```

```
→ LD    -2,F  '6,P
→ LD    -1,F  B
→ LD    0,F   '*3,P
→ LD    B     F
  ADD   F     '3
  LD    P     'FAK
```

Im linken Programmstück kann der Prozessor nach jedem Befehl unterbrochen werden, da im dynamischen Speicher Zellen reserviert werden, bevor sie belegt werden. Im rechten Programmstück hingegen werden Zellen im dynamischen Speicher zuerst belegt und dann reserviert. Wenn der Prozessor in dieser Situation unterbrochen wird (im Programm durch Pfeile markiert), so werden sie mit neuen Werten überschrieben, und am Ende der Unterbrechung findet der unterbrochene Prozeß nicht mehr seine ursprünglichen Werte vor. ■

6.4 Ein-Ausgabekanäle, Ein-Ausgabeprozessoren und Mehrprogrammbetrieb

Zur Verbindung eines Rechners mit seinen Ein-Ausgabegeräten sind technische Vorrichtungen wie Leitungsbündel und Signalverstärker vonnöten, die Daten übertragen ohne sie zu verändern und Ein-Ausgabekanäle genannt werden. Sind - wie z.B. in Bild 6.2 gezeigt - an einen Rechner mehrere Ein-Ausgabegeräte angeschlossen, so erfolgt ihre Verbindung mit dem Rechner über Multiplexer und Demultiplexer. Mit einem solchen einfachen Multiplexkanal muß der Ein-Ausgabevorgang in allen Einzelheiten programmiert werden, wobei mehrere Ein-Ausgabevorgänge zeitlich aufgeteilt nebeneinander ablaufen können (vgl. Beispiel 6.7). Mit einem Selektorkanal ist ein solcher Zeitmultiplexbetrieb nicht möglich, da der Multiplexer-Demultiplexer durch ein im Kanal vorgesehenes Register zur Adressierung des Ein-Ausgabegerätes über die Zeitdauer eines gesamten Ein-Ausgabevorganges mit ein und demselben Gerät verbunden ist. Das ist der erste Schritt zur Verschnellerung von Ein-Ausgabevorgängen durch zusätzliche Hardware. Der zweite Schritt betrifft den Einbau von Registern zur Zwischenspeicherung und zum Schiften der Datenbytes und zum Zählen der übertragenen Bytes eines Wortes. Dazu ist ein einfaches Steuerwerk erforderlich, wodurch bereits einige Befehle in den Ein-Ausgabeprozessen entfallen. Der dritte Schritt betrifft darüber hinaus den Einbau von Registern zur Zwischenspeicherung der Primärspeicheradresse und zum Zählen der zu übertragenden Worte eines Satzes. Auch dies erfordert ein Steuerwerk für das Hinaufzählen der Wortadresse und das Herunterzählen der Wortanzahl, wodurch fast alle Befehle zur Datenübertragung in den Ein-Ausgabeprozessen entfallen. Auf diese Weise werden dem Kanal immer mehr Funktionen übertragen: die Auswahl des Gerätes, das Zusammensetzen der Bytes zu Wörtern, die Übertragung der Worte eines Satzes und schließlich die Übertragung eines Satzes in oder aus mehreren unzusammenhängenden Speicherbereichen, die untereinander durch Adressen (Zeiger) verbunden sind.

Ein-Ausgabekanäle

Wie bereits gesagt, verfügen Ein-Ausgabekanäle neben Registern mit speziellen Funktionen auch über Schaltwerke mit speziellen Funktionen, die die Reihenfolge der im Kanal ablaufenden Vorgänge steuern. Man kann die Register als Datenwerk und das Schaltwerk als Programmwerk eines Prozessors mit unverständlich gespeichertem Programm und damit einen Ein-Ausgabekanal als einen für die Bedürfnisse der Ein- und Ausgabe entworfenen Spezialprozessor betrachten. Ein-Ausgabekanäle führen je nach Ausbaustufe immer mehr Teile von Ein-Ausgabeprozessen aus, wodurch der Prozessor von der Durchführung des Ein-Ausgabevorganges entlastet wird. In der letzten Ausbaustufe wird der Prozessor nur noch am Ende der Datenübertragung unterbrochen, um diejenigen Funktionen des Ein-Ausgabevorganges durch Software auszuführen, die durch Hardware entweder nur schlecht (z.B. Test- und Steuerfunktionen unterschiedlicher Ein-Ausgabegerä-

te) oder gar nicht (z.B. Synchronisation der Datenübertragung und -verarbeitung) ausgeführt werden können.

Kanal zur Übertragung von Wörtern. Bild 6.13 zeigt einen Selektorkanal mit einem Geräteadreßregister **DA**, einem Bytepufferregister **BB** und einem Bytezähler **BC**. Das Geräteadreßregister wird mit der Adresse des Ein-Ausgabegerätes geladen, wodurch der Bytepuffer mit dem Ein-Ausgabegerät verbunden wird. Der Bytezähler wird

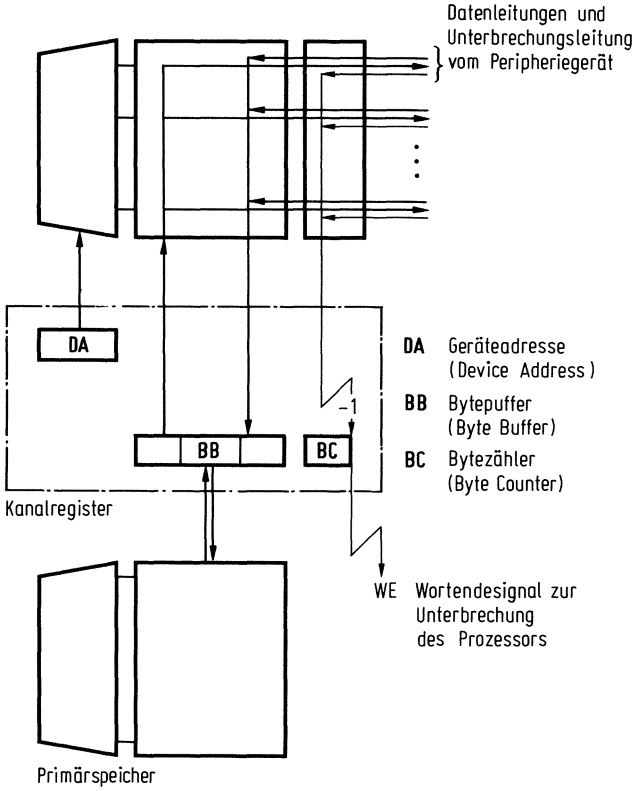


Bild 6.13 Gepufferter Ein-Ausgabekanal zur Übertragung von Wörtern (buffered channel)

mit der Anzahl der Bytes pro Wort geladen und bei jeder vom Ein-Ausgabegerät hervorgerufenen Unterbrechung heruntergezählt. Gleichzeitig wird ein Byte aus dem Bytepuffer ein- oder ausgegeben und der Inhalt des Bytepuffers um 1 Byte nach links geschiftet. Der Stand des Bytezählers gibt an, wann der Bytepuffer bei der Eingabe gefüllt bzw. wann er bei der Ausgabe geleert ist, so daß ein Wort aus dem Puffer in den Speicher bzw. aus dem Speicher in den Puffer übertragen werden kann, was jedoch nicht mehr von der Kanalsteuerung ausgeführt wird. Vielmehr erzeugt der Bytezähler des Kanals ein Signal auf einer Unterbrechungsleitung WE (Wortende), das den Pro-

zessor veranlaßt, das gerade laufende Programm zu unterbrechen und im Programm des Ein-Ausgabeprozesses fortzufahren. Dieses Signal bedeutet im Fall der Eingabe "Bytepuffer gefüllt" (das Wort kann in den Speicher eingegeben werden) und im Falle der Ausgabe "Bytepuffer leer" (das nächste Wort kann aus dem Speicher ausgegeben werden). Das Unterbrechungssignal vom Ein-Ausgabegerät wird also durch die Kanalhardware ausgewertet, während das Wortendesignal von der Rechnersoftware verarbeitet wird.

Diese Art der Datenübertragung nennt man gepufferte Übertragung und den Ein-Ausgabekanal einen gepufferten Kanal (buffered channel).

Kanal zur Übertragung von Sätzen. Bild 6.14 zeigt einen Selektorkanal, der neben dem Bytepufferregister **BB** und dem Bytezähler **BC** ein Wortzeigerregister **WP** und einen Wortzähler **WC** enthält. Der Wortzähler wird mit der Anzahl der zu übertragenden Worte eines Satzes (Feldlänge) geladen und der Wortzeiger mit der Anfangsadresse des Speicherbereiches (Feldadresse), in den der Satz eingegeben oder aus dem er ausgegeben wird. Das Sammeln bzw. Verteilen der Bytes eines Wortes wird wie beim gepufferten Kanal im Bytepuffer vorgenommen. Das vom Bytezählerstand abgeleitete Wortendesignal unterbricht jedoch nicht den Prozessor, sondern wird unmittelbar im Kanal ausgewertet, wodurch die Programmierung der Übertragung eines Wortes aus dem Bytepuffer in den Speicher oder umgekehrt entfällt. Die Kanalhardware übernimmt die Funktion der Prozessorsoftware und verbindet den Kanal mit dem Speicher für die Dauer eines Speicherzyklus. Sind der Prozessor und der Kanal an ein und demselben Speichermodul angeschlossen, so wird die Übertragung eines Wortes vom bzw. in den Kanal vorrangig gegenüber der Übertragung eines Wortes vom bzw. in den Prozessor durchgeführt, wodurch dem Programm, das der Prozessor gerade ausführt, jeweils ein Zyklus "gestohlen" wird. Bei jeder Übertragung eines Wortes zwischen Kanal und Speicher wird der Wortzeiger um 1 erhöht und der Wortzähler um 1 vermindert, wodurch die Übertragung des nächsten Wortes vorbereitet wird. Der Wortzeiger zeigt also in jedem Moment auf die Anfangsadresse des sich mit jeder Übertragung "verkleinernden" Feldes, während der Wortzählerstand die Anzahl der noch zu übertragenden Worte angibt. Sind alle Worte des Satzes übertragen, so erzeugt der Wortzähler ein Signal auf der Unterbrechungsleitung SE (Satzende), wodurch der Prozessor in seinem laufenden Programm unterbrochen wird und zur Beendigung des Ein-Ausgabevorganges im Ein-Ausgabeprozeß fortfährt. Das Unterbrechungssignal vom Ein-Ausgabegerät wie auch das Wortendesignal werden also von der Kanalhardware ausgewertet, während das Satzendesignal von der Rechnersoftware verarbeitet wird.

Diese Art der Datenübertragung nennt man auch Vorrangübertragung (cycle stealing).

Kanal zur Übertragung verketteter Sätze. In Bild 6.14 ist links neben dem Wortzähler ein zusätzliches Flipflop **CB** eingezeichnet, das für die Übertragung sog. verketteter Sätze in bzw. aus unzusammenhängenden Speicherbereichen von Bedeutung ist. Dieses Verkettungsbit ist von vornherein Null, wenn ein einziger Speicherbereich mit aufeinanderfolgenden Adressen angesprochen wird. Es ist zunächst Eins, wenn mehrere unzusammenhängende Speicherbereiche mit jeweils aufeinanderfolgenden Adressen angesprochen werden.

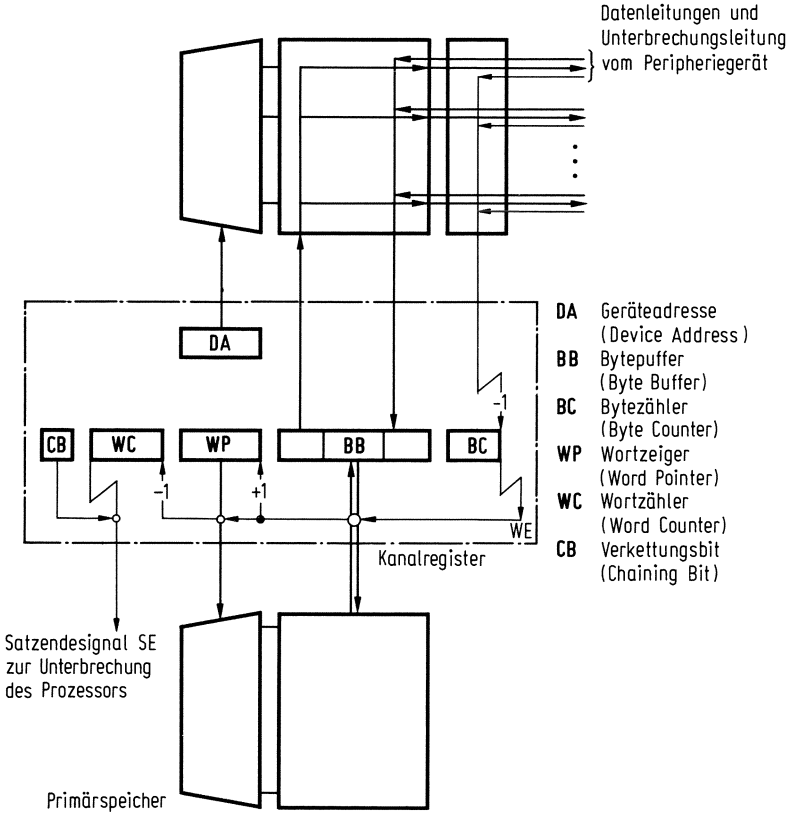


Bild 6.14 Gepufferter Ein-Ausgabekanal zur Übertragung von Sätzen in bzw. aus unzusammenhängenden Speicherbereichen (cycle stealing, gather-read/scatter-write)

Wie beim Kanal mit Vorrangübertragung werden nach dem anfänglichen Laden der Kanalregister die Daten in das erste Feld eingegeben bzw. aus ihm ausgegeben. Wenn das Verkettungsbit Null ist, wird das Satzendesignal zum Prozessor weitergeleitet. Wenn es hingegen Eins ist, werden die Kanalregister **WP** und **WC** mit der Feldadresse und der Feldlänge des nächsten Speicherbereichs geladen. Das Verkettungsflipflop wird erneut gesetzt, wenn ein weiterer Speicherbereich folgt; es wird hingegen gelöscht, wenn der letzte der unzusammenhängenden Speicherbereiche zu übertragen

ist. Die zur Übertragung von zusammenhängenden Sätzen der Peripherie in bzw. aus unzusammenhängenden Speicherbereichen benötigte Steuerinformation ist nicht in den ein- oder auszugebenden Sätzen enthalten, sondern muß in einer oder mehreren auf die Speicherbereiche folgenden Zellen bereitgestellt werden.

Diese Art der Datenübertragung nennt man Übertragung mit sammelndem Lesen und streuendem Schreiben (gather-read/scatter-write).

Beispiel 6.9 Schreibmaschinenausgabe

Wir betrachten wieder die Ausgabe eines Satzes durch die Schreibmaschine und zeigen, wie sich die Ausgabeprogramme gegenüber den Programmen aus Beispiel 6.7 vereinfachen. Links sind die Befehle für einen gepufferten Kanal und rechts die Befehle für einen Kanal mit Vorrangübertragung angeordnet. Verglichen mit dem Aus-

Kanalprogramm (buffered channel)

```
TYPE   makro   P1      P2
       ext     WC
       ext     WP
       LD      DA      '10
       LD      WC      'P1
       LD      WP      'P2
       LD      BC      '2
STRT   equ     1
       INOUT   0,DA    STRT
       mend
```

Kanalprogramm (cycle stealing)

```
TYPE   makro   P1      P2
       LD      DA      '10
       LD      WC      'P1
       LD      WP      'P2
       LD      BC      '2
STRT   equ     1
       INOUT   0,DA    STRT
       mend
```

Unterbrechungsleitung WE

```
       CALL    IOTR
```

Unterbrechungsleitung SE

```
       CALL    IOEND
```

```
       org     Ausgabeprozeß
       ent     WC
       ent     WP
       ent     IOTR
WC     res     1
WP     res     1
IOTR   CMP     WC      '0
       NOP
       GOTO    L0
       LD      BB      *WP
       SUB     WC      '1
       ADD     WP      '1
       GOTO    L1
STOP   equ     3
L0     INOUT   0,DA    STOP
L1     CNT     CL      '1
       RETURN
       end
```

```
       org     Ausgabeprozeß
       ent     IOEND
STOP   equ     3
IOEND  INOUT   0,DA    STOP
       CNT     CL      '1
       RETURN
       end
```

gabeprozeß aus Beispiel 6.7 wird die Funktion der beiden Ein-Ausgabekanäle deutlich: die im jeweils betrachteten Prozeß nicht auftretenden Befehlsfolgen sind durch Mikroprogramme in den Kanälen ersetzt und durch Steuerwerke realisiert.

Sind mehrere Selektorkanäle über Multiplexer-Demultiplexer an den Speicher angeschlossen, so muß der für die Datenübertragung benutzte Kanal durch die Angabe einer Kanaladresse ausgewählt werden. Dabei wird nicht wie in Abschnitt 6.3 jedem Ein-Ausgabegerät, sondern jedem Ein-Ausgabekanal ein Software-Unterbrechungssignal zugeordnet. Der Ein-Ausgabeprozeß gibt am Ende des Ein-Ausgabevorganges den Kanal frei, der seinerseits das Ein-Ausgabegerät stoppt, und die Rückmeldung des Gerätes wird ausgewertet. Sind beim Ein-Ausgabevorgang Fehler aufgetreten, wird im Diagnose-Programm gerätespezifisch reagiert. Der Freigabe des Kanals nach dem Ein-Ausgabevorgang steht vor dem Laden der Kanalregister das Prüfen gegenüber, ob der Kanal freigegeben worden ist. Die Auswertung des Kanalzustandes ist in den vorstehenden Programmen nicht berücksichtigt. Ebenfalls unberücksichtigt ist, daß bei der Programmierung von Ein-Ausgabevorgängen mehrere Prozesse parallel ablaufen können, was im allgemeinen Synchronisationsmaßnahmen erfordert.

■

Kanalauswahl. Hat ein Rechner mehrere Selektorkanäle und sind bestimmte Ein-Ausgabegeräte nur an bestimmte Kanäle angeschlossen (vgl. Beispiel 6.4), so muß beim Laden der Kanalregister - wie eben beschrieben - neben der Adresse für die Multiplexer-Demultiplexer zwischen Kanal und Geräten (Geräteadresse) auch die Adresse für die Multiplexer-Demultiplexer zwischen Speicher und Kanälen (Kanaladresse) angegeben werden. Eine solche Konfiguration erlaubt es nicht, daß ein Ein-Ausgabevorgang mit einem Gerät abgewickelt wird, dessen Kanal bereits mit einem anderen Gerät arbeitet. Dieser Nachteil wird behoben, wenn jeder Kanal mit jedem Ein-Ausgabegerät verbunden ist (vgl. Beispiel 6.5). Dann ist es nicht mehr erforderlich, neben der Geräteadresse auch die Kanaladresse anzugeben, denn die Datenübertragung mit einem bestimmten Ein-Ausgabegerät kann über irgendeinen Kanal erfolgen, der nicht bereits mit anderen Ein-Ausgabegeräten arbeitet.

Bild 6.15 zeigt, wie mehrere Selektorkanäle oder -prozessoren über Multiplexer-Demultiplexer matrixförmig mit den Ein-Ausgabegeräten verbunden sind. Ein Decodierschaltwerk, dessen Zustand über Sensor- und Steuerleitungen vom Prozessor abgefragt und verändert werden kann, wählt unter den freien Kanälen den mit der niedrigsten Adresse aus und verbindet ihn mit dem Speicher. Beim Starten eines Ein-Ausgabevorganges wird zunächst die Sensorleitung FR (free) abgefragt, ob mindestens ein Kanal frei ist. Ist kein Kanal frei, so wird der Ein-Ausgabevorgang verzögert, andernfalls wird der erste freie Kanal belegt und über die Steuerleitung NX (next) der nächste Kanal als erster freier Kanal ausgewählt. Beim Stoppen eines Ein-Ausgabevorganges wird der mit dem Ein-Ausgabegerät verbundene Kanal angesprochen und mit der Steu-

erleitung CR (channel released) freigegeben. Es ist jetzt nicht mehr möglich, das Satzendesignal eines Kanals als Softwareunterbrechung zur Beendigung des Ein-Ausgabevorganges zu benutzen, da keine feste Zuordnung zwischen Kanal und angeschlossenem Gerät besteht. Vielmehr muß mit dem Satzendesignal die Unterbrechungsleitung des gerade mit dem Kanal verbundenen Ein-Ausgabegerätes durchgeschaltet werden, wodurch der dem Gerät zugeordnete Ein-Ausgabeprozeß mit der Priorität des Gerätes (nicht des Kanals) ausgeführt wird. Entsprechendes gilt bei der Datenübertragung zwischen Kanal und Speicher für das Wortendesignal. Das Wortendesignal schaltet ebenfalls die Unterbrechungsleitung des Ein-Ausgabegerätes durch, wodurch die Übertragung eines Wortes mit der Priorität des Gerätes (nicht des Kanals) durchgeführt wird.

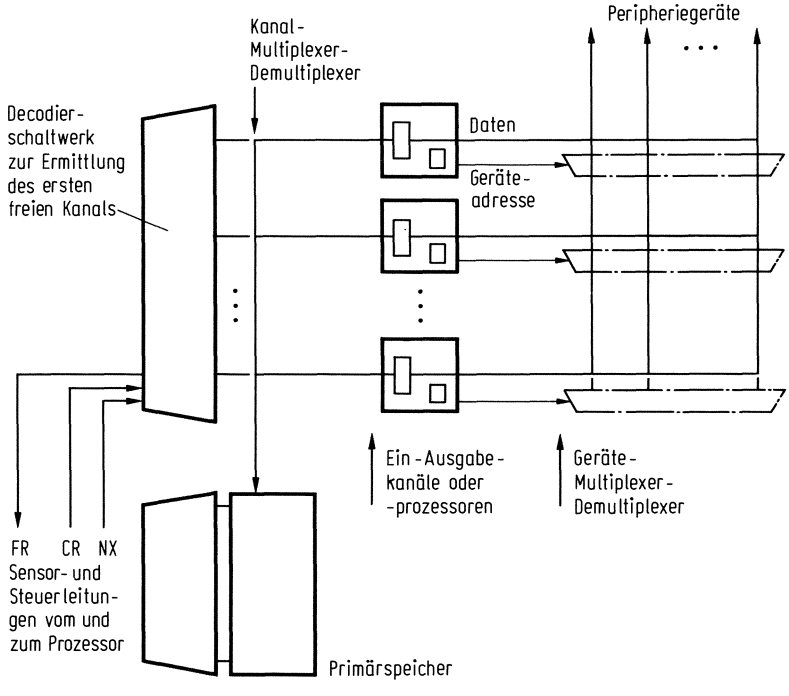


Bild 6.15 Ein-Ausgabewerk mit automatischer Kanalauswahl und fließender Zuordnung von Ein-Ausgabegeräten zu Ein-Ausgabekanälen (floating channels)

Beim Starten eines Ein-Ausgabevorganges ist - wie beschrieben - der erste freie Kanal mit Prozessor und Speicher verbunden. Mit den Befehlen LD **DA** ..., LD **WC** ..., LD **WP** ... und INOUT 0,**DA** STRT werden die Register dieses Kanals geladen und der Ein-Ausgabevorgang gestartet; mit dem Befehl CNT NX '1 wird der nächste freie Kanal mit Prozessor und Speicher verbunden. Beim Stoppen des Ein-Ausgabevorganges erfolgt eine Programmunterbrechung, und anstelle des ersten freien Kanals ist derjenige Kanal mit Prozessor und Speicher verbunden, dessen Satzendesignal die Unterbrechung

hervorgerufen hat. Mit dem Befehl INOUT 0,**DA** STOP wird über das Geräteadreßregister des Kanals der Ein-Ausgabevorgang gestoppt, und der Kanal wird durch den Befehl CNT CR '1 freigegeben. Dieser Befehl wirkt auf die Auswahlsteuerung des Kanalsystems in ähnlicher Weise wie der Befehl CNT CL '1 auf die Prioritätensteuerung des Unterbrechungssystems. Wir vereinbaren, daß die Steuerleitung CR auch die Steuerleitung CL beeinflußt, so daß mit CR=1 auch CL=1 wird, d.h. Ein-Ausgabekanal und Ein-Ausgabeprozeß werden gleichzeitig freigegeben.

Ein-Ausgabekanäle dieser Art nennt man - wie bereits früher beschrieben - wegen der fließenden Zuordnung von Kanälen und Ein-Ausgabegeräten gleitende Kanäle (floating channels).

Beispiel 6.10 Matrixaddition mit Bändern

Wir skizzieren ein Programmstück, das die Elemente von zwei Matrizen der Größe 100 mal 100 in Blöcken zu 100 Elementen von zwei Magnetbandgeräten 0 und 1 in zwei Pufferbereiche A und B des Primärspeichers eingibt, anschließend addiert und in einem dritten Pufferbereich C ablegt und schließlich auf einem dritten Magnetbandgerät mit der Nummer 2 ausgibt. Die drei Bandgeräte 0 bis 2 sind an gleitende Kanäle mit Vorrangübertragung angeschlossen und können mit den Geräteadressen 1 bis 3 ausgewählt werden. Das Laden der Kanäle erfolgt durch zwei Systemmakros READ und WRITE, die als Parameter die Nummer des Bandgerätes (P1), die Anzahl der zu übertragenden Worte (P2) und die Anfangsadressen der Pufferbereiche (P3) enthalten.

```
READ   makro   P1     P2     P3
TAPE   equ     1+P1
STRT   equ     11
L      TST     FR     '1
       GOTO    L
       LD      DA     'TAPE
       LD      WC     'P2
       LD      WP     'P3
       CNT     NX     '1
       INOUT   0,DA   STRT
       mend
```

```
WRITE  makro   P1     P2     P3
TAPE   equ     1+P1
STRT   equ     12
L      TST     FR     '1
       GOTO    L
       LD      DA     'TAPE
       LD      WC     'P2
       LD      WP     'P3
       CNT     NX     '1
       INOUT   0,DA   STRT
       mend
```

Die drei Ein-Ausgabevorgänge werden von den Kanälen selbständig abgewickelt und laufen mit der vollen Geschwindigkeit der Bandgeräte ab, so daß der Eindruck eines parallelen Betriebes der Bandgeräte entsteht. Das Prozessorprogramm wird dabei um die Anzahl der Speicherzyklen verzögert, die die Ein-Ausgabevorgänge insgesamt benötigen. Zur Organisation der Daten-Ein-Ausgabe und der Daten-Verarbeitung werden im Programm "Haltepunkte" vorgesehen. Dadurch wird verhindert, daß Daten verarbeitet werden, bevor sie eingelesen worden sind. Das Warten bzw. Fortschreiten im Programm hängt vom Zustand der jeweiligen Variablen ab, die dem Haltepunkt zugeordnet ist und der bestimmte Werte zugewiesen werden können. Um die

auf diese Weise durchgeführte "Synchronisation" zwischen dem Prozessorprogramm und den Ein-Ausgabeprozessen in den Programmen deutlich zu machen, verwenden wir zwei Makrobefehle WAIT und SET mit je zwei Parametern. Die ersten Parameter bezeichnen Zustandsvariablen, die zweiten Parameter beziehen sich auf einen Vergleichswert dieser Variablen. WAIT verzögert bedingt den Ablauf eines Programms: das Programm wartet in WAIT, wenn die Werte der Variablen gleich sind, es schreitet fort, wenn diese Werte nicht übereinstimmen. SET ersetzt den Wert der ersten Variablen durch den Wert der zweiten Variablen.

```
WAIT  makro  P1   P2
      CMP    P1   P2
      LD     P    '+1,P
      LD     P    '-3,P
      mend
```

```
SET   makro  P1   P2
      LD     P1   P2
      mend
```

Die Addition der 100-mal-100-Matrizen erfolgt in den zwei Phasen des scheinbar gleichzeitigen Lesens und Schreibens der Bandgeräte 0, 1 und 2 (Bänder arbeiten) und der Ausführung der Addition der zwischengepufferten Zelleninhalte der Matrizen (Prozessor arbeitet). Diese beiden Phasen wechseln sich gegenseitig ab: der Prozessor wartet, während die Bandgeräte arbeiten; die Bandgeräte warten, während der Prozessor arbeitet. Zur Synchronisation dieses Betriebes werden den Pufferbereichen A, B und C drei Zustandsvariablen ZA, ZB und ZC zugeordnet, die mit 0 (Puffer leer) initialisiert werden. Sobald die Bereiche A und B von den Bandgeräten 0 und 1 gefüllt sind, werden am Ende der Eingabeprozesse die Variablen ZA und ZB auf 1 (Puffer voll) gesetzt. Dann führt der Prozessor elementweise die Addition aus, füllt dabei den Puffer C und setzt am Ende des Additionsprozesses die Variable ZC auf 1 (Puffer voll). Während nun der Puffer C vom Bandgerät 2 geleert wird, können die beiden anderen Bandgeräte 0 und 1 die Puffer A und B füllen.

Den Unterbrechungssignalen von den drei Ein-Ausgabegeräten, die auf den Satzendeleitungen der Kanäle durchgeschaltet werden, sind die drei symbolischen Adressen

```
     Unterbrechung MTP0
MT0  CALL   MTP0
```

```
     Unterbrechung MTP1
MT1  CALL   MTP1
```

```
     Unterbrechung MTP2
MT2  CALL   MTP2
```

```
     org    Prozeß MTP0
     ext    ZA
     ent    MTP0
STOP equ    13
MTP0 INOUT  0,DASTOP
     SET    ZA   '1
     CNT    CR   '1
     RETURN
     end
```

```
     org    Prozeß MTP1
     ext    ZB
     ent    MTP1
STOP equ    13
MTP1 INOUT  0,DASTOP
     SET    ZB   '1
     CNT    CR   '1
     RETURN
     end
```

```
     org    Prozeß MTP2
     ext    ZC
     ent    MTP2
STOP equ    13
MTP2 INOUT  0,DASTOP
     SET    ZC   '0
     CNT    CR   '1
     RETURN
     end
```

MT0, MT1 und MT2 (magnetic tape 0,1 und 2) zugeordnet. In diesen Speicherzellen stehen CALL-Befehle, die die Ein-Ausgabeprozesse MTP0, MTP1 und MTP2 aufrufen.

Hauptprogramm

```
        org                          Hauptprogramm
        ent     ZA
        ent     ZB
        ent     ZC
A       res     100                  Puffer für Elemente der Matrix A
ZA      res     1                    Zustand von Puffer A
B       res     100                  Puffer für Elemente der Matrix B
ZB      res     1                    Zustand von Puffer B
C       res     100                  Puffer für Elemente der Matrix C
ZC      res     1                    Zustand von Puffer C
         :
START    :
         :
        LD      I       '0
        SET     ZC      '0           setze Zustand von C 'leer'
L0      SET     ZA      '0           setze Zustand von A 'leer'
        SET     ZB      '0           setze Zustand von B 'leer'
        READ    0       100     A    lies von Band 0 100 Wörter nach A
        READ    1       100     B    lies von Band 1 100 Wörter nach B
        WAIT    ZA      '0           warte, wenn Zustand von A 'leer'
        WAIT    ZB      '0           warte, wenn Zustand von B 'leer'
        WAIT    ZC      '1           warte, wenn Zustand von C 'voll'
        LD      J       '0           addiere A und B nach C
L1      LD      C,J     A,J
        ADD     C,J     B,J
        ADD     J       '1
        CMP     J       '100
        GOTO    L1
        SET     ZC      '1           setze Zustand von C 'voll'
        WRITE   2       100     C    schreibe auf Band 2 100 Wörter aus C
        ADD     I       '1
        CMP     I       '100
        GOTO    L0
         :
        end     START                                                    ■
```

Ein-Ausgabeprozessoren

Daß es sich bei der Programmierung von Ein-Ausgabevorgängen um die Organisation von parallel ablaufenden Prozessen handelt, tritt besonders deutlich in Beispiel 6.10 hervor. Die Datenübertragung wird vollständig über die Ein-Ausgabekanäle abgewikkelt, die wie eigenständige Prozessoren parallel zum Zentralprozessor arbeiten. Lediglich am Ende einer Datenübertragung wird das Programm unterbrochen und der Zentralprozessor für kurze Zeit dem Ein-Ausgabekanal zur Verfügung gestellt. Dies kann man so deuten, daß sich der Kanal mangels eigener Leistungsfähigkeit den Zentralprozessor für kurze Zeit ausleiht. Damit kann der Kanal nicht nur für die Dauer der Datenübertragung, sondern auch am Ende der Datenübertragung als Prozessor angesehen werden; zuerst arbeitet er als Prozessor mit unveränderlich gespeichertem Mikroprogramm und dann als Prozessor mit veränderlich gespeichertem Maschi-

nenprogramm. Bei Kanälen mit Vorrangübertragung beschränkt sich die Inanspruchnahme des Prozessors auf die Beendigung des Ein-Ausgabevorganges (Beispiel 6.9). Bei gepufferten Kanälen wird der Zentralprozessor außer für die Beendigung des Ein-Ausgabevorganges auch für die Übertragung der gepufferten Wörter zwischen Kanal und Speicher benutzt (Beispiel 6.9). Bei ungepufferten Kanälen schließlich wird der gesamte Ein-Ausgabevorgang vom Zentralprozessor im Unterbrechungsrhythmus des Ein-Ausgabegerätes durchgeführt (Beispiel 6.7). Selbst dann handelt es sich bei der Programmierung von Ein-Ausgabevorgängen um die Organisation parallel ablaufender Prozesse, wenn ein einziger Prozessor alle Ein-Ausgabevorgänge nacheinander im Unterbrechungsmodus abwickelt. Der eine Prozessor führt dabei die Aufgaben nacheinander aus, die den einzelnen Ein-Ausgabekanälen und dem Zentralprozessor zur gleichzeitigen Ausführung zugeteilt sind.

Ein-Ausgabekanäle arbeiten zwar wie Prozessoren, ihre Funktion wird jedoch bei ihrer Konstruktion ein für alle Mal festgelegt, und es ist nicht möglich, sie zu ändern, ohne ihre physikalische Konstruktion zu ändern. Anders verhält es sich bei Ein-Ausgabeprozessoren. Ihre Funktion wird erst durch ihre Programmierung festgelegt und kann jederzeit unterschiedlichen Anforderungen angepaßt werden. Zu diesem Zweck können Ein-Ausgabeprozessoren auf Speicherbereiche zugreifen, in denen ihre Programme und Daten aufbewahrt sind und die zur Datenübertragung dem Zentralprozessor zumindest teilweise zugänglich sein müssen.

Ein-Ausgabeprozessoren können ganz auf die Bedürfnisse der Ein-Ausgabeorganisation zugeschnittene Spezialprozessoren sein (programmierbare Kanäle). Sie können aber auch die Eigenschaften eines im Vergleich zum Zentralprozessor einfachen Universalrechners haben. Dann ist es möglich, mit dem Ein-Ausgabeprozessor außer der reinen Datenübertragung auch Datenvor- und -nachverarbeitung durchzuführen und ihm damit einen Teil der Aufgaben des Zentralprozessors zu übertragen. Ein-Ausgabeprozessoren können schließlich - zumindest als Modell - die gleichen Eigenschaften des Universalrechners haben wie der Zentralprozessor. In allen Fällen haben wir hierarchisch gegliederte Organisationsformen von Mehrprozessor- oder Mehrrechnersystemen vor uns.

Im letzten Beispiel dieses Kapitels wird eine Möglichkeit gezeigt, in einem Mehrprozessorsystem der in Bild 6.15 gezeigten Struktur die Ausgabe eines Textes auf einer Schreibmaschine mit einem Ein-Ausgabeprozessor durchzuführen. Zur Programmierung des Prozessors benutzen wir die Befehle unseres Zweiadreßrechners, verwenden jedoch weder dynamische noch statische Primärspeicherbereiche, sondern bringen die Operanden in den Registern des Ein-Ausgabeprozessors unter. Auf diese Weise entsteht ein Programm mit einfachen Befehlen, das man sich von einem einfachen Einadreßrechner als Ein-Ausgaberechner ausgeführt denken kann.

Beispiel 6.11 Schreibmaschinenausgabe

Wie in den früheren Beispielen definieren wir einen Makrobefehl TYPE mit zwei Parametern, die die Anzahl der auszugebenden Worte und die Anfangsadresse des auszugebenden Satzes enthalten. Der Makrobefehl lädt die Register des ersten freien Ein-Ausgabeprozessors in gleicher Weise wie früher die Register des ersten freien Ein-Ausgabekanals. Wir setzen voraus, daß ein freier Prozessor (wie früher ein freier Kanal) inaktiv ist, so daß er nicht mit einem CALL-Befehl unterbrochen werden muß, sondern nur durch einen GOTO-Befehl aktiviert zu werden braucht. Weiter setzen wir voraus, daß die Ein-Ausgabeprozessoren und der Zentralprozessor an einen gemeinsamen Speicher angeschlossen sind und gleichzeitig arbeiten können, ohne sich gegenseitig zu stören, so daß der Ausgabeprozeß unter Benutzung einer Sensorleitung programmiert werden kann. Diese Einzelbitleitung vom Ein-Ausgabegerät hat für den Ein-Ausgabeprozessor die gleiche Funktion wie die Einzelbitleitung beim Ein-Ausgabekanal, nur daß sie nicht als Unterbrechungsleitung im Kanal-Mikroprogramm, sondern als Sensorleitung im Prozessor-Programm abgefragt wird.

Die Programmierung des Ausgabeprozesses erfolgt in ähnlicher Weise wie in Beispiel 6.1 unter ausschließlicher Benutzung von Registern, jedoch folgt die Programmstruktur mehr dem Ausgabeprogramm aus Beispiel 6.5, wobei wie in Beispiel 6.9 bei der Programmierung von Ein-Ausgabekanälen Register an die Stelle der Primärspeicherzellen treten. Die Punkte in der Befehlsfolge des Ein-Ausgabeprozesses sollen andeuten, daß mit einem Ein-Ausgabeprozessor im Gegensatz zu einem Ein-Ausgabekanal mehr als nur die Datenübertragung bewerkstelligt werden kann, z.B. beim Ausgeben von Zahlen die Konvertierung von Dual- in Dezimalzahlen (vgl. Beispiel 4.1). Schließlich wird durch den letzten Befehl des Ausgabeprozesses der Befehlszähler auf dem gleichen Stand gehalten, so daß der Ein-Ausgabeprozessor wartet. Unmittelbar davor hat er dem Decodierschaltwerk zur Auswahl des ersten freien Prozessors mitgeteilt, daß er selbst frei ist.

```
TYPE   makro   P1   P2     Definition des Makrobefehls TYPE
       ext     TYP
L      TST     FR   '1     wenn Sensorleitung FR
       GOTO    L           =0, dann zurück zu Marke L (kein Prozessor frei)
       LD      WC   'P1    =1, dann lade Wortzähler
       LD      WP   'P2    lade Wortzeiger
       LD      BC   '2     lade Bytezähler
       LD      PC   TYP    lade Befehlszähler des ersten freien Prozessors
       CNT     NX   '1     markiere nächsten freien Prozessor
       mend
```

```
       org                 Definition des Ausgabeprozesses TYP
       ent     TYP
TY     equ     10          Geräteadresse Schreibmaschine
STRT   equ     1
TYP    INOUT   TY   STRT   Start Ausgabe
L0      .
        .
        .
       LD      BB   0,WP   lade Bytepuffer mit Speicherwort
       SUB     WC   '1     vermindere Wortzähler
       ADD     WP   '1     erhöhe Wortzeiger
L1     TST     TY   '1     wenn Sensorleitung TY
       GOTO    L1          =0, dann zurück zu Marke L1
       OUT     BB   TY     =1, dann gib aus Bytepuffer aus
       SUB     BC   '1     vermindere Bytezähler
       SA      BB   '8     schifte Bytepuffer um 1 Byte
       CMP     '0   BC     wenn Bytezähler
       GOTO    L1          >0, dann zurück zu Marke L1
       LD      BC   '2     =0, dann setze Bytezähler
       CMP     '0   WC     wenn Wortzähler
       GOTO    L0          >0, dann zurück zu Marke L0
STOP   equ     3
       INOUT   TY   STOP   =0, dann Stopp Ausgabe
       CNT     CR   '0     Prozessor- und Unterbrechungsfreigabe
L2     GOTO    L2          warte
       end
```

Wir haben in diesem Kapitel das Beispiel der Ausgabe eines Satzes auf der Schreibmaschine in sechs Varianten behandelt. In allen Programmen werden zur Speicherung der Operanden statische Speicher benutzt, so daß sie mit Ausnahme des für den Ein-Ausgabeprozessor geschriebenen Programmes nicht als reentrant betrachtet werden können. Das Programm für den Ein-Ausgabeprozessor braucht nur einmal gespei-

chert zu sein und kann von mehreren Ein-Ausgabeprozessoren gleichzeitig benutzt werden, da jeder Prozessor seine Operanden in eigenen Registern hält. Sollte zur Speicherung der Operanden der Primärspeicher benutzt werden, so müßte wie in den anderen Fällen der Speicher dynamisch organisiert werden. Dazu müßten CALL-Befehle mit Parametern verwendet werden, was beim Einprozessorsystem mit Unterbrechungssteuerung die Programmierung der Zuordnung von Unterbrechungsleitungen und -adressen (variable Speicherung der Unterbrechungsadressen) und beim Mehrprozessorsystem mehrere dynamische Speicherbereiche unterschiedlichen Ursprungs (mehrfache Verwendung von ORG-Befehlen) voraussetzen würde. Beim Mehrprozessorsystem ergäbe sich dabei die Möglichkeit, daß die dem Zentralprozessor untergeordneten Ein-Ausgabeprozessoren solange unabhängig arbeiten, wie sie nicht vom Zentralprozessor benötigt werden. Würden weiter statt spezieller Ein-Ausgabeprozessoren universelle Prozessoren mit Ein-Ausgabekanälen verwendet, so wären Organisationsformen möglich, in denen z.B. das Betriebsprogramm vom "Zentralprozessor" und die Benutzerprogramme von den "Ein-Ausgabeprozessoren" bearbeitet würden. Stehen jedoch nicht so sehr prinzipielle Lösungen, sondern mehr Fragen der Effizienz im Vordergrund, so wird man dennoch die Bearbeitung dieser verschiedenen Aufgaben einem einzigen Prozessor mit Ein-Ausgabekanälen übertragen, ähnlich wie in unseren Beispielen der Schreibmaschinen-Ausgabeprozeß nicht von einem Ein-Ausgabeprozessor, sondern entweder von einem Ein-Ausgabekanal oder vom Zentralprozessor selbst ausgeführt wird. ■

Mehrprogrammbetrieb

Wenn auf einer Rechenanlage ein Programmlauf beendet sein muß, bevor der nächste gestartet werden kann, wenn also immer nur ein Programm in Betrieb ist, so kann man von Einprogrammbetrieb sprechen. In diesem Betrieb der Rechenanlage ist der Prozessor bei Ein-Ausgabevorgängen oft nur mangelhaft ausgelastet. Bei einem mit einer Sensorleitung programmierten Ein-Ausgabevorgang wartet der Prozessor, bis das dem Ein-Ausgabevorgang zugeordnete Sensorsignal das Weiterrechnen im Ein-Ausgabeprogramm erlaubt, und erst nach der Beendigung des Ein-Ausgabevorganges wird im Hauptprogramm fortgefahren. Dadurch wird das Ein-Ausgabeprogramm auf natürliche Weise mit dem Hauptprogramm synchronisiert, und es kann nicht vorkommen, daß das Hauptprogramm mit Daten arbeitet, die vom Ein-Ausgabeprogramm noch nicht ein- bzw. ausgegeben worden sind.

Bei einem mit einer Unterbrechungsleitung programmierten Ein-Ausgabevorgang rechnet der Prozessor sofort nach dem Anstoßen des Ein-Ausgabevorganges im Hauptprogramm weiter. Jedes dem Ein-Ausgabevorgang zugeordnete Unterbrechungssignal unterbricht das Hauptprogramm, und der Prozessor wird für die Dauer einer Wortübertragung an das Ein-Ausgabeprogramm ausgeliehen. Dabei muß die Beendigung des Ein-Ausgabevorganges auf künstliche Weise mit dem Hauptprogramm synchronisiert werden, um zu vermeiden, daß das Hauptprogramm mit Daten arbeitet, die noch gar nicht ein- bzw. ausgegeben worden sind. Auch hier wartet der Prozessor, wenn auch im Hauptprogramm.

Um den Prozessor besser auslasten zu können, werden die im Einprogrammbetrieb auftretenden Wartezeiten des Prozessors für die Bearbeitung weiterer Programme benutzt, und man geht vom Ein- zum Mehrprogrammbetrieb über. Steht für den Mehrprogrammbetrieb nur ein einziger Zentralprozessor zur Verfügung, so muß die Ausführung der einzelnen Programme zeitlich nacheinander ablaufen. Der Prozessor muß von einem zum anderen Programm umgeschaltet werden, ähnlich wie er bei Programmunterbrechungen vom Hauptprogramm zum Ein-Ausgabeprogramm umgeschaltet wird. Im Gegensatz zum Unterbrechungsbetrieb erfolgt jedoch das Umschalten des Prozessors bei Mehrprogrammbetrieb über die Systemsoftware: Im Prinzip werden alle WAIT-Befehle als Systemaufrufe aufgefaßt, und der Prozessor wird vom sog. Benutzermodus (slave mode) in den Systemmodus (master mode) übergeführt. Solche Aufrufe des Betriebssystems (supervisor calls) arbeiten wie programmierte Programmunterbrechungen. Man nennt sie Fallen (traps). Durch sie werden wie bei Unterbrechungen Zellen angesprochen, die unter der Regie des aufgerufenen (im allgemeinen des Betriebssystems)

und nicht etwa des aufrufenden Programmes stehen. Das Arbeiten des Prozessors im Systemmodus sollte so wenig Zeit wie möglich beanspruchen, da diese Zeit für das Ausführen der Aufträge der Benutzer verloren geht. Bei der Organisation eines Betriebssystems ist deshalb darauf zu achten, den Überhang (overhead) klein im Vergleich zum Durchsatz (throughput) zu halten.

In einer Situation, in der ein Benutzerprogramm auf die Beendigung eines Ein-Ausgabevorganges warten muß, wird über das Betriebssystem ein anderes Benutzerprogramm aufgerufen. Ist der Ein-Ausgabevorgang beendet, so wird im ersten Benutzerprogramm fortgefahren. Das zweite Programm wird unterbrochen und muß warten. Bei einem erneuten Warten des ersten Programmes auf die Beendigung eines Ein-Ausgabevorganges wird im zweiten Programm an der unterbrochenen Stelle fortgefahren, und mit Beendigung des Ein-Ausgabevorganges wird im ersten Programm weitergerechnet. Die Bearbeitung der Programme erfolgt, als ob sie in einem Keller gespeichert wären, d.h. dasjenige Programm, das zuletzt in die Reihe der wartenden Programme aufgenommen wurde, wird bei Freiwerden des Prozessors als erstes weiter ausgeführt. Dieses Last-in-first-out-Prinzip ist besonders günstig bei Stapelverarbeitung, bei der ein Stapel von unabhängigen Benutzerprogrammen nach Prioritäten in der Rechenanlage verarbeitet wird, so daß der Durchsatz der Anlage möglichst hoch ist. Erfolgt die Bearbeitung der Programme hingegen, als wären sie in einer Schlange aufgereiht, so wird das zuerst in die Reihe der wartenden Programme aufgenommene Programm bei Freiwerden des Prozessors als erstes weiter ausgeführt. Dieses First-in-first-out-Prinzip eignet sich besonders für den sog. Vielfachzugriff (multi-access), bei dem mehrere nur teilweise geladene Benutzerprogramme der Reihe nach stückweise ausgeführt werden, so daß die Antwortzeit für einen jeden Benutzer möglichst klein wird.

Für den Mehrprogrammbetrieb eignen sich Prozessoren mit mehreren Registersätzen gleicher Funktion; man spricht von verschiedenen Prozessor-Umgebungen (environments), die den Status verschiedener Programme kennzeichnen. Wenn zum Beispiel das Betriebssystem anstelle des Wartens auf das Ende eines Ein-Ausgabevorganges die Ausführung eines anderen Programmes veranlaßt, wählt es die entsprechende Umgebung aus und findet das Programm in dem Zustand, in dem es dieses zu einem früheren Zeitpunkt verlassen hatte. Die Beendigung des Ein-Ausgabevorganges wird durch ein Unterbrechungssignal vom Ein-Ausgabegerät angezeigt. Das Betriebssystem schaltet den zum ersten Programm gehörenden Registersatz auf. Diese Umgebung beinhaltet den Status dieses Programmes, und der Prozessor kann ohne Verzögerung im Benutzermodus weiterarbeiten.

Steht nicht für jedes Programm ein eigener Registersatz zur Verfügung, so müssen für einen Teil der Programme oder für alle Programme eigene Speicherbereiche für die Statusinformation im Primärspeicher vorgesehen werden. Das Betriebssystem muß bei einem Programmwechsel den Registerspeicher des Prozessors umladen, so daß der Prozessor jeweils den Status des zu verarbeitenden Programmes vorfindet. Zum Programmstatus gehören außer den Inhalten der Register auch alle in Flipflops gespeicherten Bedingungen (flags) oder Steuerwörter (control words), die im allgemeinen zu einem oder mehreren Programmstatuswörtern (program status words) zusammengefaßt werden. Kurz gesagt zählen zum Status eines Programmes die Inhalte aller Speicherelemente des Prozessors, die bei der Unterbrechung des Programms zu seiner späteren Fortsetzung wieder benötigt werden.

Häufig werden Rechner mit zwei Registersätzen ausgestattet, einem als Umgebung der Betriebsprogramme und einem als Umgebung der Benutzerprogramme. Dadurch wird ein schnelles Umschalten von Benutzer- auf Systemprogramme und zurück möglich, und lediglich der Status der Benutzerprogramme muß umgeladen werden.

Schnelles Umschalten vom Benutzer- in den Systemmodus ist auch aus Gründen der Überwachungs- und Verwaltungsaufgaben des Betriebssystems notwendig. Üblicherweise läuft das gesamte Unterbrechungssystem unter der Regie des Betriebssystems. Es kann Prioritäten verteilen, es kann Unterbrechungen ignorieren, oder es kann sie nach anderen Strategien als dem einfachen LIFO-Prinzip auswerten. Unterbrechungen

betreffen in vielen Fällen den Betrieb der Rechenanlage selbst. Man denke zum Beispiel an den Ausfall des Stromnetzes (vgl. Beispiel 6.6). Auch die Unterbrechungen bei Ein-Ausgabevorgängen beeinflussen den Betrieb der Anlage (vgl. Beispiel 6.7). So ist es naheliegend, auch die Initiierung der Ein-Ausgabevorgänge dem Betriebssystem zu übertragen. Dadurch beschränkt sich für den Benutzer die Ein- und Ausgabeprogrammierung auf Systemaufrufe ähnlich den in unseren Beispielen verwendeten Makroaufrufen TYPE, READ und WRITE (vgl. Beispiele 6.9 und 6.10).

Für den Betrieb der Rechenanlage mit mehreren unabhängigen Benutzerprogrammen ist es geradezu notwendig, die Abwicklung aller Ein-Ausgabevorgänge dem Betriebssystem zu übertragen, denn nicht die einzelnen Benutzer, sondern nur das Betriebssystem hat die Übersicht über die im Einsatz befindlichen Betriebsmittel, wie Ein-Ausgabekanäle und Ein-Ausgabegeräte. Ähnliches gilt für die Ein- und Auslagerungen beim Laden von Programmen für den Primärspeicher und die peripheren Speicher. Nur das Betriebssystem als den Benutzerprogrammen übergeordnetes Systemprogramm ist in der Lage, die verschiedenen Wünsche der einzelnen Benutzer nach unterschiedlichen Gesichtspunkten zu koordinieren. Damit einzelne Benutzer diese unter übergeordneten Gesichtspunkten vorgenommene Verteilung der Betriebsmittel nicht zu ihren Gunsten einseitig verändern können, sollten Ein-Ausgabebefehle und verschiedene Systembefehle, z.B. zur Speicherverwaltung oder für das Unterbrechungssystem, nur im Systemmodus ausgeführt werden. Man bezeichnet solche Befehle als privilegierte Befehle (privileged instructions).

Durch die gemeinsame Verwaltung aller Betriebsmittel durch das Betriebssystem wird ausgeschlossen, daß zum Beispiel zwei Programme, von denen das erste mit einem ersten Ein-Ausgabegerät und das zweite mit einem zweiten Ein-Ausgabegerät arbeiten und gleichzeitig das erste Programm das zweite Gerät und das zweite Programm das erste Gerät benutzen wollen, sich gegenseitig blockieren, wenn jedes Programm auf die Beendigung des Wartezustands des anderen wartet. Solche Situationen können auftreten, wenn die parallel laufenden Prozesse nicht ordnungsgemäß synchronisiert sind; sie werden als Verklemmungen (dead-locks) bezeichnet. In diesem Sinne können auch Ein-Ausgabekanäle und Ein-Ausgabegeräte als Prozessoren angesehen werden, und die Programmierung von Ein-Ausgabevorgängen läuft auf die Programmierung sog. nichtsequentieller Prozesse hinaus (parallel processing). Zur Verwaltung der Betriebsmittel gehört neben der Buchführung der Betriebsmittelanforderungen auch die Strategie der Betriebsmittelverteilung. Verschiedene zum Teil sich widersprechende Forderungen werden an das System gestellt, zum Beispiel maximale Auslastung der einzelnen Teile der Anlage und minimale Antwortzeit des Systems. Die vorhin beschriebenen FIFO- und LIFO-Prinzipien sind die einfachsten solcher Fahrpläne (schedules) genannten Strategien. Sie entsprechen allerdings nicht den Kompromissen, die hier oft geschlossen werden müssen.

Betriebssysteme für Mehrprogrammbetrieb sind sehr komplexe Programme, deren Erstellung vom Softwareingenieur die Beherrschung der einschlägigen Programmierungstechniken und ein gewisses Verständnis für die Hardware moderner Rechenanlagen verlangt. Auf der anderen Seite benötigt der Hardwareingenieur neben fundiertem Wissen über technologische Möglichkeiten in zunehmendem Maße auch Kenntnisse über die Software moderner Rechenanlagen.

Literaturverzeichnis

Barron, D.W.: Assembler und Lader. München: Hanser 1970.

de Beauclair, W.: Rechnen mit Maschinen. Braunschweig: Vieweg 1968.

Bell, C.G., Newell, A.: Computer Structures: Readings and Examples. New York: McGraw-Hill 1971.

Chu, Y.: Computer Organization and Microprogramming. Englewood Cliffs: Prentice-Hall 1972.

Colin, A.J.T.: Betriebssysteme. München: Hanser 1973.

Enslow, P.H.: Multiprocessors and Parallel Processing. New York: Wiley 1974.

Gear, C.W.: Computer Organization and Programming. New York: McGraw-Hill 1969.

Güntsch, F.R., Schneider, H.-J.: Einführung in die Programmierung digitaler Rechenautomaten. Berlin, New York: de Gruyter 1972.

Hasselmeier, H., Spruth, W.G. (Hrsg.): Rechnerstrukturen. München, Wien: Oldenbourg 1974.

Hill, F.J., Peterson, G.R.: Digital Systems: Hardware Organization and Design. New York: Wiley 1973.

Hobbs, L.C. (ed.): Parallel Processor Systems, Technologies and Applications. New York: Spartan Books 1970.

Hoffmann, R.: Rechenwerke und Mikroprogrammierung. München, Wien: Oldenbourg 1976.

Iliffe, J.K.: Basic Machine Principles. London: Macdonald 1972.

Jessen, E.: Architektur digitaler Rechenanlagen. Berlin, Heidelberg, New York: Springer 1975.

Lorin, H.: Parallelism in Hardware and Software. Englewood Cliffs: Prentice-Hall 1972.

Randell, B. (ed.): The Origins of Digital Computers: Selected Papers. Berlin, Heidelberg, New York: Springer 1973.

Schecher, H.: Funktioneller Aufbau digitaler Rechenanlagen. Berlin, Heidelberg, New York: Springer 1973.

Steinbuch, K., Weber, W. (Hrsg.): Taschenbuch der Informatik. Berlin, Heidelberg, New York: Springer 1974.

Tanenbaum, A.S.: Structured Computer Organization. Englewood Cliffs: Prentice-Hall 1976.

Sachverzeichnis

Abrufphase 10
Absolutlader 109
Additonsalgorithmus 25
- nach von Neumann 74, 76
Additionsrechner 74, 81, 87
Adreßbezüge 105
- , externe 114
Adreßbus 40
Adreßersetzung 127
- , dynamische 161
- , statische 158
Adreßumsetzungsbefehle 8
Adreßfeld 98
Adressierung, indirekte 112, 132, 143
- , relative 142
- , unverschränkte 189
- , verschränkte 187
Adreßmodifizierung eines Makroassemblers 136
- eines Zweiadreßrechners 132
- , programmierte 45
Adreßrechnung 102
Adreßregister 181
Adreßspeicher 66, 174, 193, 208
Adreßumsetzung 123
Adresse 5
- , absolute 107
- , aktuelle 151
- , effektive 140
- , feste 107
- , indirekte 144
- , logische 185
Adresse, reale 185, 192
- , relative 107
- , symbolische 17, 98
- , virtuelle 185, 192
ADU (Analog-Digital-Umsetzer) 236
Akkumulator 6
- eines Einadreßrechners 40
- - - mit doppelter Wortlänge 40
aktuelle Parameter 146
- von Makrobefehlen 136, 148
- von Prozeduren 162
algorithmische Sprache 94
Alterungsmechanismus 196
ALU (arithmetisch-logische Einheit) 79, 132
Analog-Digital-Umsetzer (ADU) 236
Anweisungen 97
- eines einfachen Assemblers 99
- eines Makroassemblers 134
arithmetisch-logische Einheit (ALU) 79, 132
Assembler 16, 95
- , dynamischer 209
- , Funktion 17, 103
Assembleranweisung 16, 97
Assemblercode 18, 96
Assemblersprache 16, 97
Assembler-Unterprogramm 127, 146
assembly time 97
Assoziativprozessor 198
Assoziativspeicher 70, 176
- mit linearem Zugriff 179

Assoziativspeicher mit zyklischem Zugriff 179
- , Aufteilung in Adreß- und Wortspeicher 72
- , Formeln 71
- , integrierter 175
- , Symbol 70
Assoziierer 64
- , Formeln 64
- , Schaltbild 65
- , Symbol 64
Aufruf, geschachtelter 158, 161
- , reentranter 254, 256
- , rekursiver 160, 161, 256
Auftrag 120
Auftragsbeschreibung 119
Ausdruck 137
- , absoluter 137
- , relativer 137
Ausführungsphase 10

B 6700 244
Basisadresse 142
Basisadressierung 141
Basisregister 141
batch processing 120
Bedienungsfeld 110
Bedingungen 270
bedingte Assemblierung 136
Befehl 7, 17
- , privilegierter 271
- , symbolischer 14
Befehle eines Zweiadreßrechners 130
Befehlsabruf 10
- eines Einadreßrechners 49
Befehlscode, binärer 16
- , symbolischer 16
Befehlsformat eines Einadreßrechners 41f
Befehlsmodifikator 41
Befehlsregister 10
Befehlsspeicher 208, 210
Befehlszähler 10
- , Initialisierung 108
Benutzermodus 119, 269
Benutzerprogramm 118
Benutzungstabelle 114
Bereichsüberschreitung 21
Betriebsanweisung 96
Betriebssystem 96, 120
Bibliotheksprogramm 121
Bildschirmgeräte 237
Binärcode 5, 13
Binärzeichen 5
binary digit 5
Binder 95, 108, 113
Bindercode 95
Binderinformation 114
Bindermodul 113
Bindesymboltabelle 114
Bit 5
b-komplementär 21
Blockadresse 184
Blockauswahl 184
Blockbasisadresse 213
Blockbasisadreßspeicher 214
Blockdecodierung 184
Blockmultiplexbetrieb 242
Blockmultiplexkanal 243
Blockmultiplexprozessor 243
Blockschachtelung 213
Blockstruktur 215
Boolesche Algebra 52
boolesche Funktion 55
- - in Matrixdarstellung 55
- - , Definitionsbereich 57
- - , disjunktive Normalform 56
- - , Wertebereich 57
- - , Wertetabelle 57
- Matrizen 57
- Vektoren 57
boolescher Algorithmus 22

boolescher Algorithmus, Ausgangsgrößen 77
- - , Ausgangsmatrix 78
- - , Bedingungen 77
- - , Befehle 77
- - , Eingangsgrößen 77
- - , Übergangsmatrix 78
- - , Zustände 77
bootstrap loading 109
buffered channel 258
Bus (bus) 40
Byte 5
Bytekette 18
Bytemultiplexbetrieb 243
Bytemultiplexkanal 243
Bytemultiplexprozessor 243
byteorientierte Rechner 231
Bytepuffer 252
Bytepufferregister 258
Bytezähler 233, 252, 258

Cache Memory 201
call by name 127, 148
call by reference 127, 152, 162, 222
call be value 148, 152, 222
CAM (Content Addressable Memory) 71, 175, 176
CD 7600 198
Code 5
- , gleichlanger 5
- , maschinenexterner 15
- , maschineninterner 15
- , symbolischer 12
Codegenerierung 103, 107
Codeumsetzer 66, 175
Codeumsetzung 58
Codewort 5
Codierer 59
- , Formel 60
- , Schaltbild 60
- , Symbol 59
Codiermatrix 59
Codierung 12, 14
compiler 120
Content Addressable Memory (CAM) 71, 175, 176
control line 231
control word 270
Cycle Stealing 241, 259

Daten 4
- , globale 215
- , lokale 215
Datenadreßspeicher 210
Datenbasisadresse 142, 162
Daten-Verarbeitungswerk 89
Datenwerk 72
DAU (Digital-Analog-Umsetzer) 237
dead-lock 271
Decodiermatrix 59
Decodierung 14, 59
Definitionstabelle 114
Demultiplexer 62
- , Formel 63
- , Schaltbild 63
- , Symbol 63
Dezimalzahl 98
Dialogsprache 122
Dialogverkehr 123
Digital-Analog-Umsetzer (DAU) 237
digitales Datenverarbeitungssystem 73
- Rechensystem 73
- System 73
Diodenmatrix 175
Direct Memory Access 241
Direktoperand 132, 139
Division 29
- durch Subtrahieren 30
- durch Subtrahieren und Rückstellen des Rests 30
- durch Subtrahieren und Addieren ohne Rückstellen des Rests 32

Dualziffer 5
Dump Routine 121
Durchsatz 270
dynamische Adreßabbildung 193
- Prozedurschachtelungstiefe 218
- Speicherreservierung 161
- Speicherverwaltung 155, 172, 208
dynamischer Operand 210
- Speicherbereich 210

EBCDI-Code 13
Editor 123
Einadreßbefehl 41, 128
Einadreßrechner 40
- , Befehle 42
- , Struktur 41
Ein-Ausgabebefehl 128
Ein-Ausgabe-Einheit (IOU) 132, 229
Ein-Ausgabegeräte 226, 233, 258
Ein-Ausgabe-Kanal 226, 230, 233
Ein-Ausgabemodul 228, 230
Ein-Ausgabeprogramme 250
Ein-Ausgabeprozessor 227, 230
Ein-Ausgabe-Steuereinheit (PCU) 242
Ein-Ausgabewerk 5, 11
Einfachbus 190
Einfachbussystem 238
einphasige Assemblierung 105
Einzelwort-Programmunterbrechung 249
elektromechanische Rechenmaschinen 1
elektronische Rechenmaschinen 2
- - , erste Generation 2, 37
- - , zweite Generation 37
- - , dritte Generation 37
Elementarhandlung 19, 47
Emulation 39, 89
Emulator 91
environment 270
erster freier Speicherplatz 155
Extended Binary Coded Decimal Interchange Code (EBCDIC) 13
Externcode, binärer 18
- , symbolischer 18

Fakultät 167, 223, 256
Falle 269
Fehlerliste 104
Fehlertabelle 104
Festspeicher 66, 176
- , Formeln 66
- , Symbol 66
FIFO-Speicher 180
file 123
first-in-first-out (FIFO) 180
flag 270
Flipflopspeicher 175
Floating Channel 244, 264
formale Parameter 146
- von Makrobefehlen 135, 147
- von Prozeduren 222

gather-read/scatter write 261
gepufferter Kanal 258
- - zur Übertragung von Sätzen 259
gepufferte Übertragung 259
Geräteadreßregister 258
Geräteadresse 230, 262
geschachtelter Aufruf 158, 161
geschützter Speicherbereich 109
gleitender Kanal 264
globale Größe 220

Halbleiterspeicher 183
Hardware 37
Hauptprogramm 146, 157
hierarchisches Speichersystem 193
Hilfsladeprogramm 108

IBM 360/67 206
- 360/85 201
Indexadressierung 140
Indexregister 140

Indexregister, assoziative 197
indirekte Adressierung 112, 132, 143
Indizierung 45
Information 4
Informationsgehalt 5
Initialisierungsphase 108
initial program loading 108
Interncode 15
- , binärer 18
Interpretationshierarchie 47
interrupt 244
interrupt line 244
IOU (Ein-Ausgabe-Einheit) 132, 229

job 120
Job Control Language 120
Job Description Language 120
Job-Verarbeitung 120
- -Beschreibung 121

Kanal, gepufferter 258
- - , zur Übertragung von Sätzen 259
- - - - - Wörtern 258
- , gleitender 264
- , programmierbarer 267
Kanaladresse 262
Kanalauswahl 262
Kanalregister 243
Kanalstatusinformation 243
Keller 177, 213
Kellerspeicher 157
Kennungsfeld 98
Kernspeicher 175, 181
Klarschriftleser 237
Kommandosprache 119
Kommentar 97
Kommentarfeld 98
kommerzielle Peripherie 225
Komplementieren 24
Konstante 98
Konstantenadressierung 138
Konstantenspeicher, konstant decodierter 176
- , variabel decodierter 176

Ladeadresse 107
Ladeprogramm 108
Lader 95, 108
- , assemblierender 96
- , bindender 95, 114
- , verschiebender 112
Ladercode 95, 102, 108
Laderinformation 107
Ladermodul 114
last-in-first-out (LIFO) 176
Laufzeitsystem 127
least recently used (LRU) 196
Leitwerk 5, 8
- , Funktion 10
Lesebus 40
lexikalische Analyse 103
LIFO-Speicher 176
linking loader 114
Literal 136, 138
Lochkartencode der IBM 13
Lochkartengeräte 234
Lochstreifengeräte 234
logische Operationen 53
lokale Größe 220
lokaler Datenbereich 158
LRU (least recently used) 196

Magnetbandspeicher 179, 182
Magnetblasenspeicher 182
Magnetplattenspeicher 182
Magnetspeicher 183
Magnettrommelspeicher 182
Makro 147
Makroassembler 134
- , Funktion 136
Makroaufruf 127, 136, 147
Makrobefehl 127, 134

Makrodefinition 127, 136, 147
Makrodefinitionstabelle 136
Makroname 136
Makrorumpf 136
Maschinenbefehle 15, 97
- eines Einadreßrechners 42
Maschinencode 15, 18, 95, 107
Maschinenprogramm 16
Maskenregister 71
master mode 269
Matrixaddition 150, 154, 158, 164
264
Matrixoperationen 53
mechanische Rechenmaschinen 1
Mehrfachbus 190
Mehrfachbussystem 239
Mehrprogrammbetrieb 269
Mehrprozessorsystem 227, 267
Mehrrechnersystem 267
Mikroalgorithmus 20, 22, 47
Mikrobedingung 47
Mikrobefehl 47
Mikrofunktion 19
Mikrooperation 47
Mikroprogramm 47
- eines Einadreßrechners 49
mikroprogrammierter Digitalrechner 88
Mikroprogrammierung, horizontale 84
- , vertikale 84
Mikroprogramm-Speicher 89
Mikroprogrammsprache 47
Mikroprogramm-Verarbeitungswerk 89
Mikroprogrammwerk 73
- , boolesches 78
- - , für Parallelverarbeitung 79
- - , für Serienparallelverarbeitung 81
- , sequentielles 83
- - , für Serienverarbeitung 84
- - , für Serienparallelverarbeitung 86
Mikroprozessor 91
Mikrosprache 47
Minterm 56
Mnemon 98
modulare Konfiguration 226, 230
modularer Speicheraufbau 184
Monitor 96, 119
multi-access 270
Multiplexbetrieb 229
Multiplexer 61
- , Formel 61
- , Schaltbild 62
- , Symbol 62
Multiplexeradresse 62
Multiplikation 25
- durch Addieren 26
- durch Addieren und Schiften 26
- durch Addieren, Subtrahieren und
Schiften 27

Nachassembler 95
Nachassemblieren 108
Nachindizierung 133
Name 98
Namensfeld 98, 104
Nebeneinandersetzen 53
Negation 54
Normalform, disjunktive 56

Oberprogramm 157
Objektcode 96
Operandenabruf 10
- eines Einadreßrechners 49
Operandenadresse 7
Operandenformat eines Einadreßrechners
41
Operandenspeicher 208, 210
Operandenwort 42
Operationscode 7
- , mnemonischer 98
Operationsfeld 98
overhead 270

paging 204
paralleler Betrieb 240
parallel processing 271
Parameterauffangfeld 153
Parameterbehandlung 148, 151
Parameterfeld 151
Parameterliste 136
Parametertransport 222
Parametertransportbefehl 222
Parameterübergabe 166, 221
Parameterversorgungsfeld 153
PCU (Ein-Ausgabe-Steuereinheit) 242
Peripherie-Konfiguration 242
PLA (Programmable Logical Array) 67
Primärspeicher 90, 96, 175
Prioritätenschaltnetz 65, 247
privileged instruction 271
problemorientierte Sprache 94, 120
Programm 4
- , residentes 109
- , symbolisches 15, 95
- , verschiebliches 112
Programmable Logical Array (PLA) 67
Programmbasisadresse 211
Programmbasisadreßspeicher 210
Programmbibliothek 120
Programmblock 215
programmierbarer Kanal 267
Programmindexspeicher 221
Programmliste 101, 108
Programmodul 108, 112
Programmrücksprungspeicher 221
Programmschleife 101
Programmsegment 112, 211
Programmsprung 43
- , bedingter 44
Programmstatus 270
Programmstatuswort (program status word) 270
Programmunterbrechung 226, 244
Programm-Verarbeitungswerk 89
Programmwerk 73
Programmzeile 97
- , Format 97
Prozedur 127, 151
- , geschachtelte 161, 218
- , reentrante 254
- , rekursive 161, 218, 255
- , wiedereintrittsfeste 254
Prozeduraufruf 127, 151, 218, 222
Prozedurbasisadresse 218
Prozedurbasisadreßspeicher 218
Prozedurdefinition 127, 151, 223
Prozedurrücksprungadresse 221
Prozeß 226, 246
- , nichtsequentieller 271
Prozessor 89
- mit booleschem Mikroprogrammwerk 91
- mit sequentiellem Mikroprogrammwerk 91
Prozessormodul 228
Prozessor-Umgebung 270
Prozessor-Unterprogramm 127, 146, 151
Prozeßperipherie 225
Pufferspeicher 195
Pufferspeichersystem 199, 202

Quellcode 96
Quellregister 42
queue 181

Rahmen 203
Random Access Memory (RAM) 69, 175, 176
Randomspeicher 68, 176
- mit zweifacher Adreßdecodierung 69
- , Formeln 69
- , hintereinandergeschaltete 72
- , Symbol 68
Read Only Memory (ROM) 67, 175, 176
reale Pufferadresse 199
- Segmentanfangsadresse 199

reale Seitenadresse 201, 203
Realzeitsystem 244
Rechenmaschinen, mechanische 1
- , elektromechanische 1
- , elektronische 2
- - , dritte Generation 37
- - , erste Generation 2, 37
- - , zweite Generation 37
Rechenwerk 5, 8
Rechnerkonfiguration 240
Rechnernetz 225
Rechnerperipherie 227
reentranter Aufruf 254, 256
Referenzadresse 114
Referenzkette 116
Register 5, 175
Registerbefehl 42
Registersatz 270
Rekursionstiefe 168
rekursiver Aufruf 160, 161, 256
Relativadresse 213
relative Adressierung 142
Ringpuffer 181
ROM (Read Only Memory) 67, 175, 176
Rücksprungadresse 151, 162
Rückwärtszählen 23
Rundschiftregister 179
run time 97
run-time system 127

sammelndes Lesen/streuendes Schreiben 261
Sammelschiene 40
Satzendesignal 260
Scanner 103
Sedezimalzahl 98
Sedezimalziffer 98
Segment 199
Segmentadresse 199
- , reale 203
- , virtuelle 199, 203
Segmentadressierung 203
segmenting 204
Segmentorganisation 198, 200, 202
Segmenttabelle 204
Seite 201
Seitenadressierung 203
Seitenorganisation 198, 200, 202
- , segmentierte 206
Seitentabelle 204
Sekundärspeicher 90, 96
Selektorbetrieb 242
Selektorkanal 243, 258
sense line 231
Sensorleitung 231, 244
serieller Betrieb 240
shared code 255
- data 255
slave mode 269
Software 94
- , arbeitsspeicherresidente 121
Speicher 5, 65, 89
- , dynamischer 161, 166, 245, 255
- für parallelen Betrieb 189
- , inhaltsadressierbarer 71
- mit linearem Zugriff 178
- mit sequentiellem Zugriff 173, 177
- mit wahlfreiem Zugriff 173, 176
- mit zyklischem Zugriff 178
- , peripherer 181, 226
- , realer 192
- , unverschränkter 186
- , verschränkter 187
- , virtueller 172, 192
Speicherbank 190
Speicherbereich, dynamischer 210
- , geschützter 109
- , statischer 210
Speicherblock 184
Speicherhierarchie 171
Speichermodul 228
Speicherplatzfreigabe 215

Speicherschutz 123, 205
Speicherschutzbedingung 205
Speicherwerk 8
Speicherzelle 5
Speicherzyklus 181
Substitution 45
Subtraktionsalgorithmus 25
Suchspeicher 175
supervisor call 269
Symbol 98, 107
- , definiertes 105
- , festes (fixed) 107
- , verschiebliches (relocatable) 107
- , verwendetes 105
Symboltabelle 15, 105, 107
Synchronisation 231, 265
Systemmakro 162
Systemmodus 269
Systemprogramm 96, 116

Schaltkernmatrix 175
Schaltnetz 67, 175
schedule 271
Schieberegister 179
Schiftbefehl 42
Schreibbus 41
Schreibmaschine 233
Schreibmaschinenausgabe 231, 251, 261, 267

stack 157, 177
stackpointer 157
Stapel 177
Stapelbetrieb 120
Startadresse 108
Starten und Stoppen eines Ein-Ausgabevorganges 251
Startinformation 251
statische Blockschachtelungstiefe 213
- Speicherverwaltung 155, 158
statischer Operand 210
statischer Speicherbereich 210
Status 270
Statusinformation 270
Steuerinformation 251
Steuerkarte 120
Steuerleitung 231
Steuerwort 231, 270
Strukturoperationen 53

Teilnehmerbetrieb 122
Testprogramm 120
Textersetzung 127
throughput 270
time sharing 123
time slice 123
Torschaltung 62
- als PLA 67
TR 440 188, 197
Trace-Routine 120
Transformation des Datenzustands 74
- - Programmzustands 74
Transposition 53
trap 269
Trommelspeicher 179

Überhang 270
Überlauf 23
Übersetzungsprogramm 120
Übertrag 23
UNIVAC 1108 190
Unterbrechungsleitung 245
Unterbrechungssystem 247
Untereinandersetzen 53
Unterprogramm 146, 157
- , geschachteltes 127, 158
- , geschlossenes 127, 146
- , offenes 127, 146
- , rekursives 127, 158
Unterprogrammaufruf 147, 150
- bei einem Einadreßrechner 46
Unterprogrammrücksprung bei einem Einadreßrechner 46

unverschränkte Adressierung 189
Update-Routine 121
Urlader 109

Variablenspeicher, konstant decodierter 176
- , variabel decodierter 176
Vektoraddition 102, 141, 142, 144, 148, 149, 152, 153, 156, 212 216f
Vektoroperationen 53
Vektorreduktion 54
Verallgemeinerte Addition 54
- Multiplikation 54
verkettete Sätze 260
Verkettungsbit 260
Verklemmung 271
Verschieblichkeitsinformation 107
verschränkte Adressierung 187
Verteilerschaltung 63
Vielfachzugriff 270
Vier-Spezies-Rechner 5
virtuelle Pufferadresse 199
- Segmentanfangsadresse 199
- Seitenadresse 201, 203
Von-Neumann Rechner 3, 8, 11
- , Struktur des 12
- , Verallgemeinerung des 129
Vorindizierung 132
Vorrangsübertragung 259
Vorwärtszählen 23
Vorzeichen-Zahl 20

Warteschlange 181
Wort 5
Wortadresse 184
- , reale 199
Wortauswahl 184
Wortdecodierung 184
Wortendesignal 232, 258
Wortlänge 5, 40
- von kaufmännischen Rechnern 40
- - technischen Rechnern 40
- - wissenschaftlichen Rechnern 40
wortorientierte Rechner 231
Wortregister 181
Wortspeicher 66, 174, 194
Wortzähler 252, 259
Wortzeiger 252
Wortzeigerregister 259

XDS SIGMA 2 241
- SIGMA 7 241, 243
X-Y-Schreiber 237

Zahlenring 21
Zeichen 5
Zeichentabelle 104
Zeichenvorrat 98
Zeilendrucker 235
Zeitmultiplexbetrieb 253
Zeitscheibe 123
Zentralprozessor 230
Zieladresse 129
Zielregister 42
Zugriffszeit 69, 181, 182
Zuordner 66
Zuordnungstabelle 15, 104, 107
Zuordnungszähler 105
Zustand 48
Zweiadreßbefehl 128
Zweiadreßrechner, Befehle eines 130
- , Struktur eines 131
Zwei-Komplement-Zahl 20
- , Wert 21
Zwei-Phasen-Assembler 103
zweistufige Programmhierarchie 89, 157
Zwischencode 95, 107
Zykluszeit 69