Interner Bericht
DESY F35-85-01
Februar 1985

Datenverarbeitung für Physiker

Vorlesung, gehalten im WS 1983/84

E. Lohrmann

II. Institut für Experimentalphysik, Universität Hamburg

Eigentum der DESY Bibliothek
Property of DESY library
Zugang: 19. MRZ. 1985
Accessions:
Leihfrist: 7 Tage
Loan period: 7 days

DESY behält sich alle Rechte für den Fall der Schutzrechtserteilung und für die wirtschaftliche Verwertung der in diesem Bericht enthaltenen Informationen vor.

DESY reserves all rights for commercial use of information included in this report, especially in case of filing application for or grant of patents.

" Die Verantwortung für den Inhalt dieses Internen Berichtes liegt ausschließlich beim Verfasser "

Inhalt

1 Einführung in die Informationstheorie **1**
1.1 Informationsmenge 1
1.2 Kodierung 2
1.3 Information und 2. Hauptsatz der Thermodynamik 8
1.4 Informationsübertragung 8
1.5 Fehlerprüfung- und Korrektur 11
1.6 Analogsignale 12
1.6.1 Abtasttheorem 12
1.6.2 Kanalkapazität 13
Literatur zu Kapitel 1 15

2 Organisation von Rechnern **16**
2.1 Die von Neumann-Maschine 16
2.2 Darstellung von Zahlen und Symbolen 17
2.3 Instruktionen 20
2.4 Adressierungsmethoden 23
2.4.1 Indexregister 23
2.4.2 Indirekte Adressierung 24
2.4.3 Virtuelle Adressierung 25
2.5 Speicher 25
2.6 Externe Speichermedien 27
2.6.1 Magnetband 27
2.6.2 Plattenspeicher 28
2.6.3 Massenspeicher 28
2.7 Ein/Ausgabe 29
2.8 Programmunterbrechung (Interrupt) 30
2.9 Systeme von Rechnern 31
Literatur zu Kapitel 2 32

3 PASCAL **33**
3.1 Vorbemerkung 33
3.2 Programmaufbau 33
3.3 Vokabular 34
3.4 Datentypen 35

3.5 Der ausführende Teil 36
3.6 Programmbeispiele 37
Literatur zu Kapitel 3 40

4 Erstellung von Programmen **41**
4.1 Allgemeines 41
4.2 Hilfsmittel zur Festlegung des Programmablaufs 41
4.3 Stilfragen der Programmierung 46
4.4 Compiler 49
Literatur zu Kapitel 4 53

5 Programmoptimierung **54**
5.1 Allgemeine Regeln 54
5.2 Spezielle Regeln 54
Literatur zu Kapitel 5 60

6 Programmprüfung und Fehlersuche **61**
6.1 Überblick 61
6.2 Test von Einzelmoduln 62
6.2.1 Programmlesung 62
6.2.2 Entwurf von Testfällen 63
6.3 Gesamt-Test 65
6.4 Finden und Beseitigen von Fehlern 65
Literatur zu Kapitel 6 68

7 Algorithmen **69**
7.1 Eine Studie zur numerischen Genauigkeit 69
7.2 Rekursion 70
7.3 Divide et impera 73
7.4 Sortieren 76
7.5 Das stabile Eheproblem 78
Literatur zu Kapitel 7 79

8 Datenstrukturen **80**
8.1 Listen 80
8.2 Durchsuchen von Listen 83
8.3 Baumstrukturen und Graphen 84
8.4 Datenbanken und Informationssysteme 87
8.5 Datensicherung 89
8.6 Datenfernübertragung 89
8.6.1 Fern-Netze 90
8.6.2 Lokale Netze 92
Literatur zu Kapitel 8 93

9 Vergangenheit und Zukunft der Informatik **94**
9.1 Zur Geschichte der Informatik 95
9.2 Turing-Maschinen 95
9.3 Das Halt-Problem 99
9.4 Die Grenzen der Von-Neumann Maschine 100
9.5 Anhang 103
Literatur zu Kapitel 9 105

Liste der Abbildungen

1.1 $S(p)$ 2
1.2 Kodebaum 3
1.3 Häufigkeitsverteilung von Rechnerworten 6
Häufigkeitsverteilung der 16-bit Rechnerworte zur Darstellung der Daten des TASSO-Experiments.
1.4 Maxwell'scher Dämon 9
1.5 Informationsübertragung 10
1.6 Zweidimensionale Paritätsprüfung 11
1.7 $s'(f)$ 12

2.1 Schema einer VNM 17
2.2 Schema einer zentralen Recheneinheit(CPU) 18
2.3 Zahlenbereich 20
2.4 Zugriffszeiten 26
2.5 Speicherhierarchie 26
2.6 Schema eines Plattenspeichers 28
2.7 Organisation der Ein/Ausgabe 29
SE=Steuereinheit: Sie enthält Energieversorgung und mechanische Steuerung, sie kann eines oder mehrere E/A-Geräte versorgen. An einem Selektorkanal hängt nur eine Steuereinheit, ein Multiplexer-Kanal kann mehrere Steuereinheiten bedienen, dabei kann es u.U. zu Wartezeiten kommen. Zur weiteren Entlastung des Hauptrechners kann ein 'Frontend'-Rechner eingesetzt werden, welcher Aufgaben wie Kode-Umwandlung, Formatkontrolle und Fehlerbehandlung übernehmen kann.
2.8 Multiprozessorsystem 31
Multiprozessorsystem über Datenleitungen (Bus) verknüpft. CPU=Zentrale Recheneinheit, M=Speichermodul, T=Terminal, E/A=Ein/Ausgabeeinheit allgemein, LAN=local area network (s.Abschn.8.6). Zur Vermeidung von Engpässen gibt es mehrere parallele Busse, die jeweils eine CPU mit einem M-Modul (lokaler Bus) verbinden, und solche (BUS 1, BUS 2), die mehrere Rechner bzw. Rechnersysteme verbinden (nach H.-J. Stuckenberg).
4.1 Strukturierte Programmierung 42
4.2 verboten 42
4.3 Struktogramme 43
4.4 Struktogramm zum Einlesen eines Kartenbildes 44
4.5 HIPO 44
4.6 Schritte zur Erstellung eines lauffähigen Programms 50
Der Binder fügt dem Benutzer-Programm die von diesem benötigten Unterprogramm zu, die auf einer Platte oder einem anderen Speichermedium stehen können. Der Lader verwandelt die relativen Adressen der einzelnen Programmteile in absolute Adressen.

4.7 Compiler 51

6.1 Fehlerkurve 64
6.2 Induktion 66
6.3 Deduktion 67

7.1 Divide et impera 74

8.1 Keller(stack) 81
8.2 Schlange(Queue) 81
8.3 Verkette Liste 82
8.4 Noch eine verkettete Liste 83
8.5 Baumstruktur 85
8.6 Polnische Notation 86
8.7 Auswertung eines arithmetischen Ausdrucks in polnischer Notation 87
8.8 Datenbank 88
8.9 Fernnetz 91
8.10 Lokale Netze 92

9.1 Turing-Maschine 96
9.2 Klammerprüfer 98
9.3 Universelle Turing-Maschine 99
t=Pseudoband der TM; T=Beschreibung der TM, z.B. in Form ihrer Quintupel.
9.4 Chips 101
Zahl der auf einem Chip integrierbaren Schaltelemente (1K=1000) in den Jahren von 1972 bis 1982.
9.5 Mittlere zum Schalten eines Elements benötigte Energie in den Jahren 1960-1980 102
9.6 Mittlere Verlustleistunge pro Schaltelement gegen die Schaltzeit aufgetragen 104
Die eingezeichneten Grenzkurven entsprechen Temperaturen von 300 K bzw.4K; die mit QM bezeichnete Kurve gibt die Begrenzung durch die Quantenmechanik an. Für die Begrenzung durch die Wärmeentwicklung wurden zulässige Leistungsdichten von $1 W/cm^2$ (flächenmäßig) bzw. $1 W/cm^3$ (räumliche Packung) angenommen.

Liste der Tabellen

1.1 Kodierung von Buchstaben 4
1.2 Die häufigsten 36 Wörter der deutschen Sprache 7

2.1 Beispiele verschiedener Kodes 21

4.1 Entscheidungstabellen 45

8.1 Das ISO-Sieben-Schichten-Modell 90

Kapitel 1

Einführung in die Informationstheorie

"I have seen the truth and it makes no sense."

1.1 Informationsmenge

Genau wie der Physiker große Schwierigkeiten hat, zu sagen, was Energie oder Masse "eigentlich ist", ist dies auch bei dem Begriff Information nicht so einfach. Wohl aber kann der Physiker die Größe einer Energiemenge oder die Größe einer Masse mit Hilfe von Meßvorschriften angeben, und diese Quantifizierung ist für das weitere Verständnis von Vorgängen, die mit Energie oder Masse zutun haben, entscheidend.

Die Informationstheorie quantifiziert Information durch Definition der Informationsmenge, anhand einer Meßvorschrift. Damit kann sie u.a. allgemeine Gesetzmäßigkeiten bei der Übertragung und Speicherung von Informationsmengen behandeln. Sie wurde begründet von C. E. Shannon durch verschiedene Publikationen in den Jahren 1942-1948. Grundlegende Ausführungen finden sich auch in dem Buch "Cybernetics" von N. Wiener.

Definition: *Die Einheit der Informationsmenge ist die jenige Informationsmenge, die es erlaubt, eine Entscheidung zwischen zwei gleich wahrscheinlichen Fällen zu treffen. Diese Einheit heißt* 1 bit.

Hieraus kann man die Informationsmenge einer Nachricht ableiten. Eine Nachricht geht von einer Informationsquelle aus. Sie bestimmt eine Auswahl aus einer Menge möglicher Nachrichten und kann Empfänger zu einem bestimmten Verhalten veranlassen.

Eine Informationsquelle produziert eine Sequenz von Symbolen (Zeichen). Eine solche Sequenz von Zeichen heißt *Nachricht*. Die Liste aller zulässigen Zeichen heißt *Alphabet*. Die Zeichen der Nachricht werden also aus diesem Alphabet ausgewählt.

Beispiele: Zeichen = Buchstaben; Alphabet = (a,b,c, ..., z)
Zeichen = Zahlen; Alphabet = (0,1,2,3,4,5,6,7,8,9)
Zeichen = Wörter; Alphabet = Wörterbuch

Wenn nun z.B. die Zeichen einer Nachricht aus einem Alphabet von N Symbolen entnommen werden, und wenn alle Symbole mit gleicher Wahrscheinlich p in der Nachricht vorkommen, so gestattet der Empfang eines Zeichens einen bestimmten Fall aus N gleich wahrscheinlichen Fällen auszuwählen. Eine solche Auswahl ist äquivalent $\log_2 N$ binären Auswahlentscheidungen. Damit ist die Informationsmenge der Nachricht pro Zeichen

$$H = \log_2 N = \text{ld } N$$

oder mit der Wahrscheinlichkeit

$$\begin{aligned} p &= 1/N \\ H &= -\text{ld } p \quad [\text{bit}] \end{aligned}$$

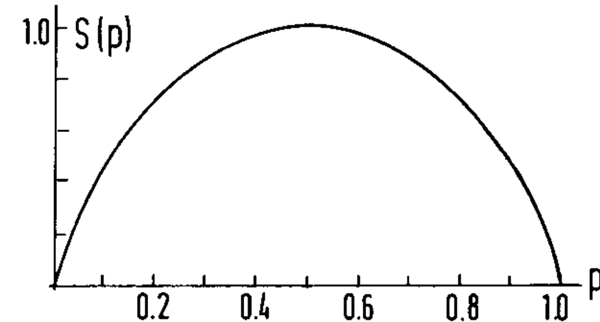


Abbildung 1.1: $S(p)$

Wir betrachten nun den allgemeinen Fall, daß die Zeichen nicht gleich wahrscheinlich sind. Im Mittel über viele Nachrichten möge das Zeichen i mit der Wahrscheinlichkeit p_i vorkommen. In einer Nachricht mit insgesamt M Zeichen (d.h. Länge M) kommen also im Mittel $N_i = M \cdot p_i$ Zeichen der Sorte i vor. Der gesamte Informationsgehalt einer Nachricht der Länge M ist also

$$\begin{aligned} &\textstyle\sum_i \ (\text{Zahl der Zeichen der Sorte}_i) \cdot (\text{Informationsmenge/Zeichen})_i \\ &= \textstyle\sum_i (M \cdot p_i) \cdot (-\text{ld } p_i) \end{aligned}$$

Die Informationsmenge pro Zeichen der Nachricht erhält man durch Division der Gesamtinformationsmenge durch die Zahl der Zeichen M. Die mittlere Informationsmenge pro Zeichen ist also

$$H = -\sum_i p_i \text{ ld } p_i \tag{1.1}$$

Diese wichtige Formel definiert die mittlere Informationsmenge pro Zeichen einer Nachricht. Ihre Einheit ist [*bit*]. Sie gilt nur, wenn die Zeichen der Nachricht voneinander statistisch unabhängig sind. Hierbei ist anzumerken, daß diese Definition rein formaler und statistischer Art ist. Die Bedeutung der Nachricht und ihre potentielle Tragweite spielen keine Rolle. So kann z.B. die Informationsmenge eines chinesischen Textes, der einem normalen Menschen unverständlich ist, sehr wohl rein formal auch ohne Kenntnisse der chinesischen Sprache nach Gleichung 1.1 bestimmt werden.

Die Einheit bit ist nicht zu verwechseln mit der Zahl der Binärzeichen (0 oder 1) in einer Binärzahl, die gemeinhin ebenfalls in bits angegeben wird. Um Verwechslungen zu vermeiden, werden wir in diesem Kapitel diese letztere Zahl als "binits" (binary digits) bezeichnen.

Wichtiger Spezialfall: Alphabet = (0,1). Eine Quelle, die solche Zeichen ausspuckt, heißt *Binärquelle*. Ist die Wahrscheinlichkeit des Zeichens 0 gegeben durch $p(0) = p$, so ist $p(1) = 1 - p$ und

$$H = -p \text{ ld } p - (1-p) \text{ ld } (1-p) = S(p) \tag{1.2}$$

Diese oft vorkommende Funktion wird mit $S(p)$ abgekürzt (S wie Shannon)(siehe Abb. 1.1).

Der Funktionsverlauf von $S(p)$ entspricht dem naiven Gefühl für die Informationsmenge einer Nachricht. Falls $p \to 0$ oder $p \to 1$, geht die Informationsmenge/Zeichen $\to 0$, d.h. eine Nachricht aus lauter 0en oder 1en enthält sehr wenig Information.

1.2 Kodierung

Unter Kodierung versteht man die eineindeutige Übersetzung der Symbole eines Alphabets in diejenigen

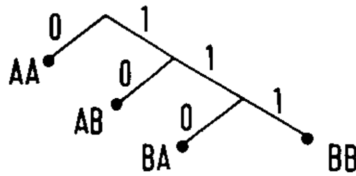


Abbildung 1.2: Kodebaum

eines anderen Alphabets. Dazu dient eine umkehrbar eindeutige Zuordnung der Elemente der beiden Alphabete anhand einer Liste: *Kodebuch*. Ein wichtiger Spezialfall ist die Übersetzung der Symbole eines Alphabets in Binärzahlen. Beispiel:

Alphabet 1	Alphabet 2
AA	0
AB	10
BA	110
BB	111

Das binäre Alphabet 2 hat die wichtige *Präfixeigenschaft*: Kein (binäres) Kodewort ist Vorderteil eines anderen.
Diese Eigenschaft ist notwendig, wenn die Symbole des Alphabets 2 ohne Trennsymbole in einer Nachricht aneinander gereiht werden sollen. (Sie ist nur dann im Prinzip entbehrlich, wenn alle Symbole dieselbe Länge haben). Die Präfixeigenschaft kann durch Konstruktion eines *Kodebaums* gewährleistet werden (1.2):

Kodierungstheorem: *Gegeben sei eine Nachrichtenquelle, die Nachrichten mit im Mittel L bits/Nachricht erzeugt. Dann ist es möglich, eine Sequenz von Nachrichten als Sequenz von Binärzeichen (0,1) so zu kodieren, daß im Mittel weniger als $L+\epsilon$ (mit $\epsilon > 0$) Binärzeichen (=binits)/Nachricht benötigt werden. Es ist nicht möglich, mit weniger als L Binärzeichen/Nachricht auszukommen.*

Beispiele:

(i) Huffman Coding: Verfahren zur optimalen Kodierung von Nachrichten in Binärzahlen nach folgendem Algorithmus:

1) Ordne alle Symbole des Alphabets nach fallender Wahrscheinlichkeit p_i in einer Liste.
2) Das Symbol mit dem kleinsten Wert p_i erhält 1, das mit dem zweitkleinsten Wert erhält 0, die vorne an eventuell bereits vorhandenen Zeichen des binären Kodes angefügt werden.
3) Vereinige die beiden Symbole von Punkt 2) zu einer Teilliste, addiere ihre Wahrscheinlichkeiten.
4) Falls kein weiteres Symbol in der Liste: Ende
Sonst: Füge die vereinigte Teilliste gemäß der Summenwahrscheinlichkeit in den richtigen Platz der Gesamtliste ein.
5) Gehe nach 2)

Beispiel:

	p_i	Kode		p_i	Kode		p_i	Kode		p_i	Kode
A	0.66		A	0.66		A	0.66		A	0.66	0
B	0.16		B	0.16		CDE	0.18	0	BCDE	0.34	1
C	0.15		C	0.15	0	B	0.16	1			
D	0.02	0	DE	0.03	1						
E	0.01	1									

Buchstabe	p_i	"Optimaler Kode"	Fernschreibkode	
·	.151	000	00100	
E	.147	001	10000	
N	.088	010	00110	
R	.068	0110	01010	
I	.064	0111	01100	
S	.054	1000	10100	
T	.047	1001	00001	
D	.044	1010	10010	
H	0.43	10110	00101	
A	.043	10111	11000	
U	.032	11000	11100	
L	.029	11001	01001	
C	.027	11010	01110	
G	.027	11011	01011	
M	.021	111000	00111	
O	.018	111001	00011	
B	.016	111010	10011	
Z	.014	111011	10001	
W	.014	111100	11001	
F	.014	1111010	10110	
K	.009	1111011	11111	
V	.007	1111100	01111	
Ü	.006	1111101	·	
P	.005	1111110	01101	
Ä	.005	11111110	·	
Ö	.003	111111110	·	$\Sigma p_i (EB)_i = 4.294$ bit
J	.002	1111111110	11010	
Y	.0002	11111111110	10101	$\Sigma p_i \text{ ld } 1/p_i = 4.115$ bit
Q	.0001	111111111110	11101	
X	.0001	111111111111	10111	
Σ	1.000			

Tabelle 1.1: Kodierung von Buchstaben

ERGEBNIS: E=1011, D=1010, C=100, B=11, A=0

Diese Kodierung hat die Präfixeigenschaft. Sie ist in dieser einfachen Form optimal, indem sie mit der geringsten Zahl von Binärzeichen in der Kodierung auskommt. Sie läßt sich verbessern, wenn man Gruppen von Symbole des ursprünglichen Alphabets zu einem neuen Alphabet so geschickt zusammenfaßt, daß die Wahrscheinlichkeiten der neuen Gruppen von Symbolen von der Form 2^{-m} sind, m = ganz, und dann neu kodiert.

Ein Maß für die Effizienz einer binären Kodierung ist die *Koderedundanz R*

$$R = B - H \tag{1.3}$$

Hierbei ist H die mittlere Informationsmenge/Zeichen und B die mittlere Zahl der Binärstellen (binits) der binären Kodierung. Ist also p_i die Wahrscheinlichkeit des Auftretens des i-ten Symbols im ursprünglichen Alphabet, und ist dieses Symbol in eine Binärzahl mit m_i Stellen (binits) übersetzt worden, so ist

$$B = \sum_i m_i p_i \tag{1.4}$$

Beispiele zu 1.1 und 1.2:

1. Kodierung von Buchstaben: Tab.1.1 zeigt die Symbole des Alphabets mit ihrer Häufigkeit p_i des Vorkommens in der deutschen Sprache. Nach Gl. 1.1 ist die mittlere Informationsmenge/Buchstabe H = 4.12 bit.

Der Fernschreibkode übersetzt dies in 5-stellige Binärzeichen. Die Redundanz dieser Kodierung ist also

$$R = 5 - 4.12 = 0.88 \quad [\text{bit}]$$

Besser ist der "optimale Kode", der die Buchstaben in Binärzahlen verschiedener Länge übersetzt,und zwar so, daß häufig vorkommende Buchstaben in kurzen Binärzahlen kodiert werden.
Hier ist nach Gl.1.4 $B = 4.29$ und die Redundanz

$$R = 4.29 - 4.12 = 0.17 \quad [\text{bit}]$$

2. TASSO-Experiment: Dieses am PETRA-Speicherring laufende Experiment speichert die Daten jeweils eines Trigger-Ereignisses in rund 3200 Rechnerworten von je 16 binits. Es ist also B=16. Das Alphabet besteht aus den 2^{16} möglichen Binärworten von je 16 binits. Abb. 1.3 zeigt die Häufigkeitsverteilung dieser 2^{16} Symbole, wie sie als Mittel über ein Stück der Datennahme des Detektors ermittelt wurde. Nach Gl. 1.1 erhält man als mittlere Informationsmenge pro 16-binit Rechnerwort H = 10.2 bits/Wort. Die Redundanz ist also

$$R = 16 - 10.2 = 5.8 \quad [\text{bit}]$$

Bei geschickterer Kodierung der Daten hätte man also im Mittel und im Prinzip mit 10.2 binits statt mit 16 auskommen können und somit z.B. 36 % der zur Datenspeicherung benutzten Bänder und der Pufferspeicher einsparen können. In der Praxis ist allerdings die oben erzielte Redundanz bereits sehr gut, und im übrigen gilt der Satz *"Sparsamkeit schafft Helden"*.

3. Informationsgehalt der Sprache: Die Buchstaben eines Textes sind korreliert, z.B. folgt der Buchstabe *u* häufiger auf das *q* als jeder andere Buchstabe. Infolgedessen ist es strenggenommen nicht richtig, die Informationsmenge/Buchstabe wie in Beispiel 1) zu berechnen. Wegen der Korrelation zwischen Buchstaben ist die wahre mittlere Informationsmenge/Buchstabe kleiner. Eine bessere Näherung ist es, als Alphabet die Worte der Sprache zu nehmen. Tab. 1.2 zeigt die Wahrscheinlichkeitsverteilung einiger Worte der deutschen Sprache (abgeschrieben aus einem älteren Buch). Für diese gilt näherungsweise das Zipf'sche Gesetz:

$$p_i \approx \frac{0.1}{i}$$

wobei p_i die Wahrscheinlichkeit für das Auftreten des i-häufigsten Wortes ist.

Aus dieser Häufigkeitsverteilung rechnet man mit Gl. 1.1

$$H_W = -\sum_{i=1}^{N} p_i \operatorname{ld} p_i \approx \sum_{i=1}^{N} \frac{0.1}{i} \cdot \operatorname{ld}(10i).$$

N ist gegeben durch die Randbedingung $\sum_{i=1}^{N} \frac{0.1}{i} = 1$.
Man erhält $H_W \approx 11.8$ bits/Wort und mit im Mittel 5.6 Buchstaben pro Wort ist nunmehr die mittlere Informationsmenge pro Buchstabe

$$H = 11.8/5.6 = 2.1 \quad [\text{bit}]$$

Dies ist zu vergleichen mit dem Wert 4.12 bit aus der naiven, ungenauen Rechnung in Beispiel 1). In Wirklichkeit sind die 2.1 bit immer noch zu hoch, da auch die Worte der Sprache korreliert sind. Genauere Abschätzungen ergeben, daß die mittlere Informationsmenge/Buchstabe etwa 1.3 bis 1.5 bit betragen dürfte.

Hieraus folgt eine (nicht ganz ernstzunehmende) Anwendung: Die "Buddenbrooks" von Thomas Mann haben etwa $1.2 \cdot 10^6$ Buchstaben, mithin 1.7 Mbit Information. Das Buch "Quantum Mechanics" von Schiff enthält mit $8 \cdot 10^5$ Buchstaben etwa 1.1 Mbit Information. Einem Werk der Informationstheorie entnehme ich, daß der Mensch etwa 100 bit/s lesen und etwa 10 bit/s erfassen kann. Demnach müsste man die "Buddenbrooks" in 4.7 h lesen und den "Schiff" in 31 h erfassen können.

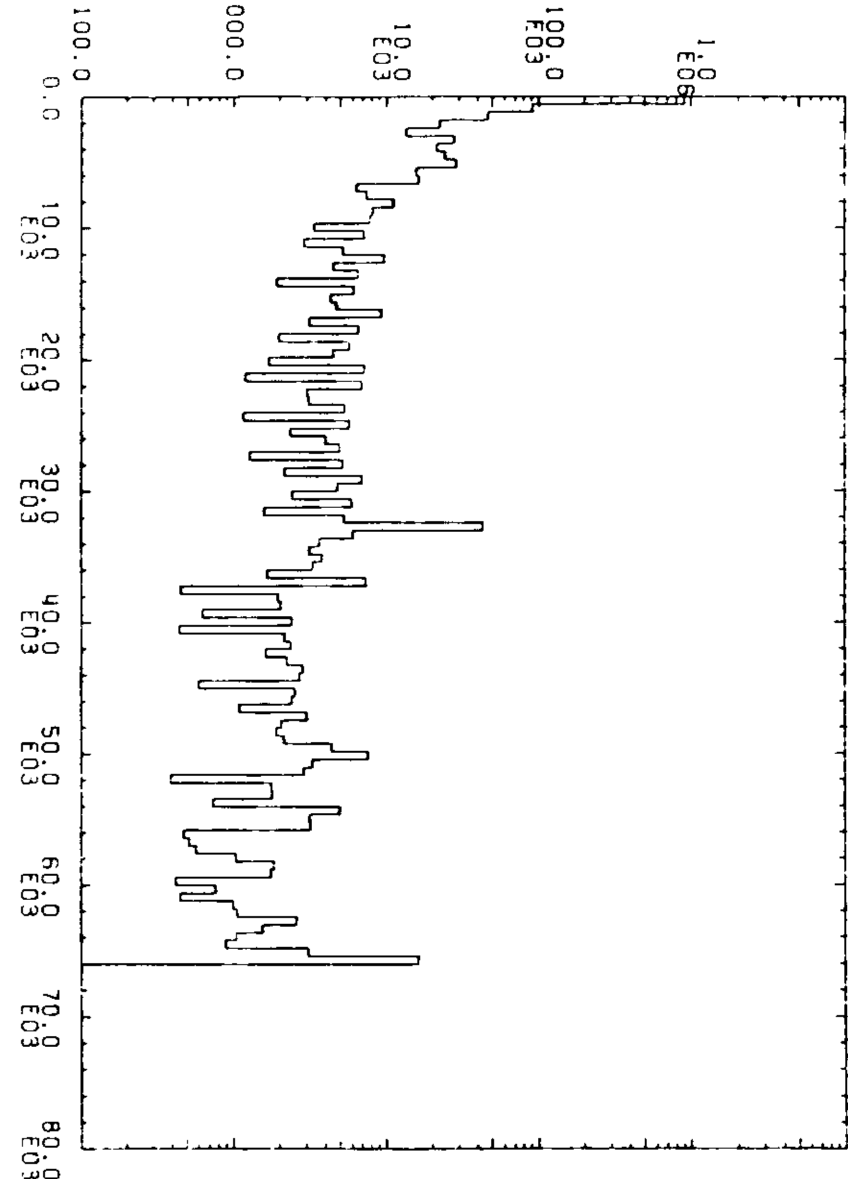


Abbildung 1.3: Häufigkeitsverteilung von Rechnerworten

	Die häufigsten Wörter		Die häufigsten Hauptwörter	
	Wort	Wahrscheinlichkeit p_i in %	Wort	Wahrscheinlichkeit p_i in %
1	die	3.28	Zeit	0.13
2	der	3.24	Herr	0.089
3	und	2.94	Jahre	0.071
4	zu	2.37	Weise	0.067
5	in	1.96	Mann	0.065
6	ein	1.40	Herren	0.063
7	an	1.34	Leben	0.063
8	den	1.30	Paragraph	0.062
9	auf	1.17	Hand	0.058
10	das	1.16	Herrn	0.058
11	von	1.08	Menschen	0.055
12	nicht	1.06	Seite	0.053
13	mit	1.01	Welt	0.049
14	dem	0.95	Frau	0.048
15	des	0.95	Art	0.046
16	aus	0.94	Gott	0.045
17	sie	0.94	Gott	0.043
18	ist	0.89	Liebe	0.041
19	so	0.89	Frage	0.039
20	sich	0.85	Augen	0.039
21	daß	0.81	Vater	0.039
22	er	0.80	Stadt	0.039
23	es	0.80	Teil	0.038
24	vor	0.77	Natur	0.038
25	ich	0.75	Nacht	0.036
26	über	0.71	Sache	0.036
27	da	0.69	Recht	0.036
28	nach	0.68	Wort	0.035
29	eine	0.64	Tag	0.028
30	auch	0.56	Gesetz	0.028
31	durch	0.55	Fall	0.027
32	als	0.54	Stellung	0.025
33	um	0.52	Lebens	0.024
34	bei	0.51	Mittel	0.024
35	wie	0.50	Weg	0.023
36	für	0.50	Haus	0.022

Tabelle 1.2: Die häufigsten 36 Wörter der deutschen Sprache

1.3 Information und 2. Hauptsatz der Thermodynamik

In der klassischen statistischen Thermodynamik betrachtet man ein System von N gleichen Teilchen, die miteinander in Wechselwirkung stehen. Es seien jeweils $n_i = N \cdot p_i$ Teilchen in einem Zustand der Energie E_i. Die Zahl der Möglichkeiten, wie man N Teilchen in die M Energiezellen $(i = 1, 2, 3, \ldots, M)$ verteilen kann, ist

$$W = N! / \prod_{i=1}^{M} n_i!$$

Die Entropie des Systems ist dann gegeben durch

$$S = k \cdot \ln W \qquad (k = \text{Boltzmann'sche Konstante})$$

Mit Hilfe des Stirlingschen Satzes

$$x! \simeq (x/e)^x \cdot (2\pi x)^{\frac{1}{2}} \approx (x/e)^x$$

erhält man

$$\ln x! \approx x \ln x - x$$

und

$$\ln W = N \ln N - N - \sum_{i=1}^{M} (n_i \ln n_i - n_i) = N \ln N - \sum n_i \ln n_i$$

da

$$\sum n_i = N$$

Mit

$$\begin{aligned} n_i &= p_i \cdot N && \text{wird} \\ \ln W &= -N \cdot \sum p_i \ln p_i && \text{und damit} \\ S &= -k \cdot N \cdot \sum p_i \ln p_i \end{aligned}$$

Diese Form hat große Ähnlichkeit mit der Definition der Informationsmenge nach Gl. 1.1, und wird dieselbe deshalb manchmal ebenfalls als "Entropie" bezeichnet. Einen Zusammenhang zwischen Entropie und Information zeigt Wiener auf am Beispiel des Maxwell'schen Dämons Abb. 1.4.

Der Dämon sitzt an einer Öffnung zwischen den beiden mit einem Gas gefüllten Gefäßen A und B. Die Öffnung hat eine Klappe, die sich ohne Energieaufwendung öffnen und schließen lässt. Immer wenn ein überdurchschnittlich schnelles Gasmolekül von B nach A fliegen will, schließt der Dämon die Klappe, während er sie öffnet, wenn ein entsprechendes Molekül von A nach B fliegt. Auf diese Weise baut sich eine Temperaturdifferenz zwischen A und B auf in Verletzung des 2. Hauptsatzes. Was ist hier los? Man muß den Dämon in das System einbeziehen. Die Information, die er über die Bewegung der Gasmoleküle erhält, erhöht seine Entropie, so daß für das Gesamtsystem der 2. Hauptsatz erfüllt ist. Ich habe allerdings noch keine quantitative Durchrechnung dieser Idee gesehen.

1.4 Informationsübertragung

Abb.1.5 zeigt das Schema einer Informationsübertragung.

Die über den Übertragungskanal übertragene Informationsmenge ΔH dividiert durch die Zeit Δt nennt man *Informationsfluß*

$$F = \frac{\Delta H}{\Delta t} = N H_0 \quad [\text{bit/s}]$$

wobei N = Zahl der übertragene Zeichen/Zeit
und H_0 = mittlere Informationsmenge/Zeichen

Dieses Maß ist nicht zu verwechseln mit der Zahl der übertragenen Binärzeichen (binits)/Zeit. Die Einheit dieses Maßes heißt *baud* = binits/s, nach Baudot (1845-1903), einem französischen Ingenieur. Nach dem Kodierungstheorem ist der Informationsfluß in bit/s stets kleiner, höchstens gleich der Zahl baud, d.h. der Zahl der übertragenen Binärzeichen/Zeit.

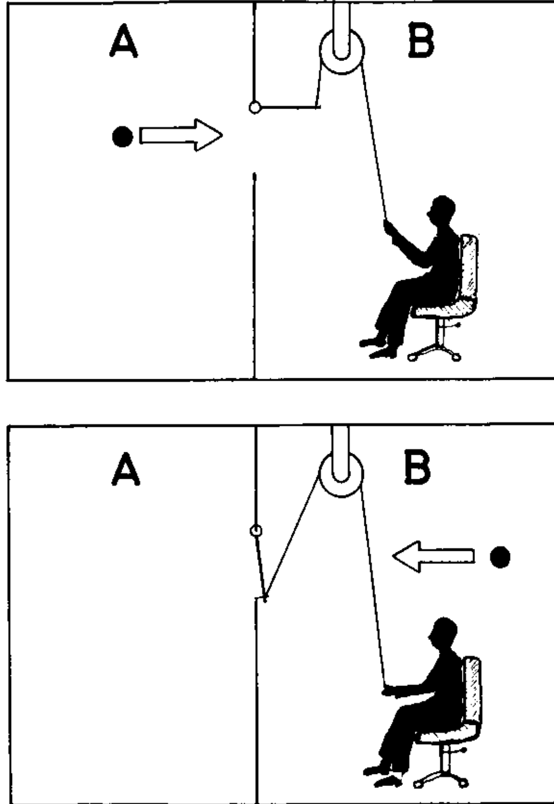


Abbildung 1.4: Maxwell'scher Dämon

Abbildung 1.5: Informationsübertragung

Ein Übertragungskanal übertrage m Zeichen in der Zeit Δt, die Übertragungszeit für ein Zeichen ist also

$$t = \frac{\Delta t}{m}$$

und sie sei für alle Zeichen gleich.

Der Übertragungskanal arbeite ohne Störung, d.h. jedes Zeichen am Eingang wird ungestört auf den Ausgang übertragen. Dann bezeichnet man als *Kanalkapazität*

$$C = \frac{1}{t} \max \left\{ -\sum_{i=1}^{m} p_i \,\mathrm{ld}\, p_i \right\} \quad [\mathrm{bit/s}] \tag{1.5}$$

wobei die Wahrscheinlichkeiten p_i der m Symbole des Alphabets so zu wählen sind, daß C ein Maximum wird. Eine optimale Wahl ist z.B. $p_i = 1/m$ und damit wird

$$C = \frac{1}{t} \,\mathrm{ld}\, m$$

In der Praxis ist die Datenübertragung über den Kanal natürlich gestört. Die Frage ist nach der Kanalkapazität mit Störung. Dazu behandeln wir zuerst als einfaches Beispiel die Übertragung eines Binärkodes.

Es sei q die Wahrscheinlichkeit für einen Übertragungsfehler eines binits (0 bzw. 1 wird als 1 bzw. 0 übertragen). Die ursprüngliche Informationsmenge sei M. Die über den gestörten Kanal empfangene Informationsmenge sei M'. Sie ist fehlerhaft. Die Fehler lassen sich korrigieren durch die Zusatz- (=Korrektur-) Informationsmenge M_C, so daß gilt

$$M = M' + M_C$$

Nach dem Kodierungstheorem läßt sich eine Nachrichtenmenge M mit $N \geq M$ binits kodieren. Als Korrekturinformation könnte man ein Wort von ebenfalls N binits übermitteln, welches eine 1 an den Stellen hat, wo das übertragene binit der Hauptinformation korrekt ist, und eine 0, da wo ein Übertragungsfehler vorgekommen ist. Die Häufigkeit der Nullen ist also q, und

$$M_C = -N(q \,\mathrm{ld}\, q - (1-q)\,\mathrm{ld}\,(1-q)) = NS(q)$$

und somit ist

$$\begin{aligned} M' &= M - M_C \\ &= M - N \cdot S(q) \end{aligned}$$

und mit $M \approx N$:

$$M' = M(1 - S(q)) \tag{1.6}$$

Man beachte, daß $M' = M$ sowohl für $q = 0$ als auch für $q = 1$ in diesem einfachen Modell.

In der Praxis läßt sich eine Fehlerkorrektur mit dem geschilderten Modell natürlich nicht durchführen. Realistische Verfahren der Fehlerkorrektur werden im folgenden Abschnitt beschrieben; eine etwas kompliziertere Überlegung führt dann ebenfalls auf Gl. 1.6.

Allgemein gilt:

$$M' = -\sum_i p_i \,\mathrm{ld}\, p_i - \left(-\sum_i p_i \sum_j p_{ij} \,\mathrm{ld}\, p_{ij} \right)$$

0	1	1	1	0	0	1
1	0	0	1	0	1	1
1	0	1	0	0	1	1
1	1	0	0	1	1	0
1	0	0	0	1	0	0
1	1	0	0	1	0	1
1	1	0	0	1	1	0

Abbildung 1.6: Zweidimensionale Paritätsprüfung

wo p_i = Wahrscheinlichkeit des $i$ten Symbols auf der Empfängerseite
p_{ij} = Wahrscheinlichkeit, das Signal i zu erhalten, unter der Voraussetzung, daß j gesendet wurde.

Die Kanalkapazität eines gestörten Kanals ist mit Gl. 1.6:

$$C' = \frac{1}{t}\max\{M'\} \simeq \frac{1}{t}\max\left\{-\sum p_i \operatorname{ld} p_i\right\}\cdot(1-S(q)) \approx C\cdot(1-S(q))$$

Satz: *Eine Informationsquelle erzeuge einen Informationseinfluß von R bits/s. Es ist möglich, über einen gestörten Kanal mit Kapazität C' Sequenzen von Symbolen so zu übertragen, daß die Wahrscheinlichkeit einer fehlerhaften Übertragung gegen Null geht, vorausgesetzt daß $C' > R$ ist.*

1.5 Fehlerprüfung- und Korrektur

Der oben genannte Satz erfordert für seine Realisierung Mittel, um Übertragungsfehler zu erkennen und zu korrigieren. Einige Möglichkeiten sind:

(i) Paritätsbit: Zu einer Binärzahl wird 1 bit hinzugefügt (Paritätsbit), so daß die Gesamtzahl der Einsen in dem Block gerade bzw. ungerade ist. Auf diese Weise kann genau ein Fehler entdeckt (aber nicht korrigiert werden). Eine Erweiterung in zwei oder mehr Dimensionen ist möglich. Beim Schreiben auf Band kann z.B. an der seitlichen Begrenzung ein Paritätsbit zugefügt werden und zusätzlich nach dem Schreiben einer vorgegebenen Zahl von Wörtern ein Wort zugefügt werden, so daß die Summe der Einsen längs dieses Bandabschnitts gerade bzw. ungerade ist (longitudinales Prüfbit), s. Abb. 1.6. Eine zweidimensionale Paritätsprüfung kann zwei Fehler erkennen und einen Fehler korrigieren.

(ii) Hamming-Kode: Dieser Kode kann genau einen Fehler in der Nachricht korrigieren. Beispiel: Zu einer Nachricht von 11 binits werden 4 binits zugefügt, die einen Fehler zu korrigieren gestatten nach folgendem Schema:

binit No.	12	11	10	9	8	7	6	5	4	3	2	1
12=				×	×		×		×	×	×	×
13=			×	×		×		×	×	×	×	
14=		×	×		×		×	×	×	×		
15=	×	×		×		×	×	×	×			

Dabei werden die binits Nummer 12-15 als Paritätsbits aus den angekreuzten binits bestimmt.

(iii) Spiegelung: Die Nachricht wird über den Übertragungskanal an den Absender zurückgeschickt und mit der ursprünglichen abgeschickten Nachricht verglichen. So können fehlerhaft übertragene Teile der Nachricht erkannt und notfalls nochmals übertragen werden. Dieses Verfahren ist aufwendig, aber einfach.

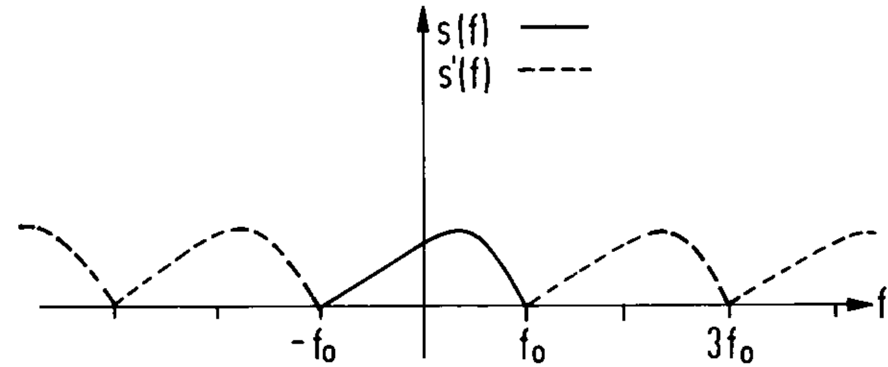


Abbildung 1.7: $s'(f)$

1.6 Analogsignale

1.6.1 Abtasttheorem

Dieses Theorem, das auf Whittaker und Shannon zurückgeht, spielt eine Rolle bei der binären Kodierung analoger Signale. Man geht aus von einem kontinuierlichen Signal, gegeben als Zeitfunktion $u(t)$. Das Fourierspektrum dieses Signals sei begrenzt von 0 bis zu einer oberen Frequenzgrenze f_0.

Schreibt man das Fourierintegral (Frequenz$f = \omega/2\pi$)

$$u(t) = \int_{-\infty}^{+\infty} s(f)e^{2\pi i f t}df$$

so ist also nach Voraussetzung die Fouriertransformierte $s(f)$ lediglich für $-f_0 \leq f \leq f_0$ von Null verschieden.

Man ergänzt nun die Funktion $s(f)$ periodisch (Periode $2f_0$), erhält dadurch die Funktion $s'(f)$ (Abb. 1.7). Die ergänzte Funktion $s'(f)$ kann wegen ihrer Periodizität als Fourierreihe geschrieben werden:

$$s'(f) = \sum_{n=-\infty}^{+\infty} a_n e^{-2\pi i n f/2f_0} \tag{1.7}$$

wobei die Fourierkoeffizienten gegeben sind durch

$$2f_0 a_n = \int_{-f_0}^{+f_0} s'(f)e^{2\pi i n f/2f_0}\,df \tag{1.8}$$

Da $s(f)$ außerhalb des Intervalls $(-f_0, f_0)$ gleich Null ist und innerhalb des Intervalls mit $s'(f)$ übereinstimmt, ist

$$u(t) = \int_{-\infty}^{+\infty} s(f)e^{2\pi i f t}\,df = \int_{-f_0}^{+f_0} s(f)e^{2\pi i f t}\,df = \int_{-f_0}^{f_0} s'(f)e^{2\pi i f t}\,df \tag{1.9}$$

Setzt man $t = n/2f_0$, so wird aus Gl.(1.9)

$$u\left(\frac{n}{2f_0}\right) = \int_{-f_0}^{f_0} s'(f)e^{2\pi i f n/2f_0}\,df = 2f_0 a_n \tag{1.10}$$

(mit Gl.1.8)

Setzt man die aus Gl.(1.10) bestimmten Werte der a_n in Gl.(1.7) ein, so erhält man

$$s'(f) = \frac{1}{2f_0} \sum_{n=-\infty}^{+\infty} u(n/2f_0) e^{-2\pi i n f/2f_0}$$

und dieses eingesetzt in Gl.(1.9)

$$u(t) = \int_{-f_0}^{f_0} s'(f) e^{2\pi i f t}\, df = \frac{1}{2f_0} \int_{-f_0}^{+f_0} \sum_{n=-\infty}^{+\infty} u(n/2f_0)\, e^{2\pi i f(t-n/2f_0)}\, df \qquad (1.11)$$

Wenn die Funktion $u(t)$ die (zeitliche) Dauer T hat, so ist sie also vermöge Gl.(1.11) vollständig darstellbar durch die endlich vielen (nämlich $2f_0T$) Funktionswerte $u(t)$ an den Stützstellen $t = n/2f_0$, $n = 0, 1, 2, \ldots, 2f_0T$.

Die Ausrechnung des Integrals Gl.(1.11) liefert als Endergebnis

$$u(t) = \sum_{n=0}^{n=2f_0T} u(n/2f_0) \cdot \frac{\sin(2\pi f_0 t - n\pi)}{2\pi f_0 t - n\pi}$$

welches nochmals explizit zeigt, daß das Signal $u(t)$ der Dauer T vollständig bestimmt ist durch die $2f_0T$ Werte der Funktion an den Stützstellen $t = n/2f_0$, $n = 0 \ldots 2f_0T$.

1.6.2 Kanalkapazität

Ein Übertragungskanal möge die Bandbreite f_0 haben, d.h. er überträgt Signale im Frequenzbereich 0 bis f_0Hz. Gefragt ist die Kanalkapazität, also die maximale von dem Kanal übertragbare Informationsmenge dividiert durch die hierzu benötigte Zeit.

Dieselbe ist

$$C = f_0 \text{ ld } (1 + S/R) \quad [\text{bits}/s] \qquad (1.12)$$

wobei S = Senderleistung und R = Rauschleistung.

Plausible Herleitung: Man betrachte eine Datenübermittelung, die insgesamt die Zeit T währt. Nach dem Abtasttheorem ist die gesamte von einem kontinuierlichen Signal übertragbare Informationsmenge in den $n = 2f_0T$ Amplitudenmeßwerten des Signals enthalten. Die Amplitude des Signals ist proportional der Wurzel der Gesamtleistung (=Senderleistung + Rauschleistung), also

$$U = const. \sqrt{R + S}$$

Die Zahl m der sinnvoll zu bildenden und zu übertragenden Amplitudengrößen ist durch die Amplitude ΔU der Rauschleistung

$$\Delta U = const. \sqrt{R}$$

begrenzt zu

$$m = U/\Delta U = \sqrt{1 + S/R}$$

Die Kanalkapazität ist nach Gl. 1.5

$$C = \frac{1}{t} \max \left\{ -\sum p_i \text{ ld } p_i \right\} = \frac{1}{t} \text{ ld } m$$

falls man (optimal) die Wahrscheinlichkeit aller Amplitudengrössen gleich wählt, also $p_i = 1/m$. Die Zeit t für die Übertragung eines Zeichens

= Gesamtdauer/Zahl der in dieser Zeit übertragenen Amplitudenmeßwerte

$= T/2f_0T$, also

$$C = 2f_0 \text{ ld } \sqrt{1 + S/R} = f_0 \text{ ld } (1 + S/R)$$

An dieser bemerkenswerten Formel sieht man, daß man mangelnde Bandbreite durch ein großes Signal/Rauschverhältnis kompensieren kann und im Prinzip über eine schmalbandige Leitung große Datenmengen übertragen kann. In der Praxis stößt man allerdings bald an Grenzen, da C nur langsam mit S/R ansteigt. Weiterhin sieht man, daß eine Übertragung von Information stets mit der Übertragung von Energie gekoppelt ist, da $C = 0$ falls $S = 0$. Damit gilt auch für die Übertragung von Information die Lichtgeschwindigkeit als Grenzgeschwindigkeit,

die nicht überschritten werden kann. Dies gilt für $R \neq 0$. Man könnte nun daran denken, die Rauschleistung R beliebig klein zu machen, jedoch das geht nicht. Hierfür sorgen quantenmechanische Effekte, sowie die Unmöglichkeit, den absoluten Nullpunkt der Temperatur genau zu erreichen.

Literatur

[1] H. Zemanek. *Elementare Informationstheorie*, R. Oldenbourg, 1982

[2] *Handbook of Automation and Control* Band 1, Kapitel 16, J. Wiley and Sons, 1982

[3] P. Frey. *Informationstheorie*, Akademieverlag Berlin, 1982

[4] *Communication Systems Engineering Handbook*, Mc Graw Hill, 1982

[5] A. Sommerfeld. *Vorlesungen über theoretische Physik*, Band 5 (Kap.1.3)

[6] N. Wiener. *Cybernetics*, Hermann, Paris

Kapitel 2

Organisation von Rechnern

"It works better
if you plug it in"

2.1 Die von Neumann-Maschine

Die meisten Rechner sind VNM. Ihr wesentliches Charakteristikum ist das sequenzielle Verarbeiten einer linearen Folge von Instruktionen. Insgesamt ist eine VNM charakterisiert durch die folgenden Eigenschaften:

(i) Fähigkeit zur Ausführung arithmetischer und logischer Operationen

(ii) Fähigkeit zur Ablaufsteuerung

(iii) Fähigkeit zur gleichartigen Speicherung von Daten und Instruktionen

(iv) Fähigkeit zur Kommunikation mit dem Benutzer

(v) Steuerung durch schriftliche Instruktionen

(vi) Sequentielle Abarbeitung der Instruktionen.

Die Abb. 2.1 zeigt das Schema einer VNM. Die Pfeile geben den Datenfluß an.

Daten und Instruktionen stehen als Binärworte in einem Speicher. Die Worte im Speicher sind durchnummeriert. Über diese Nummer, die man Adresse nennt, kann die an dieser Stelle gespeicherte Information abgerufen werden, oder es kann unter dieser Adresse (neue) Information abgespeichert werden. Da man beliebige Speicherplätze ohne Rücksicht auf die Reihenfolge des Zugriffs ansprechen kann, nennt man dies 'Random Access Memory (RAM)'. Im Gegensatz dazu stehen Speicher, die nur das Lesen, nicht das Schreiben von Information gestatten ('Read Only Memory' ROM).

Die Verarbeitung der Daten erfolgt in der zentralen Recheneinheit ('Central Processing Unit' CPU). Fig. 2.2 zeigt schematisch und vereinfacht die CPU der IBM 3033. Die Pfeile geben den Datenfluß an. Instruktionen werden entweder aus einem schnellen Zwischenspeicher (CACHE) oder aus dem Hauptspeicher in die CPU geholt. Dies geschieht unter der Kontrolle des Instruktionsregisters, welches jeweils die Adresse der nächsten auszuführenden Instruktionen enthält. Die Instruktion wird dekodiert, d.h. die Dekodiereinheit stellt anhand des Instruktionswortes fest, welche Art die auszuführende Operation ist, unter welcher Adresse die Information steht, die bearbeitet werden soll, und wohin das Ergebnis der Operation abgespeichert werden soll. Sie stellt diese Information aus dem Speicher oder aus Registern der CPU bereit und steuert dann die arithmetisch-logische Einheit (ALU) an. Dieselbe führt die gewünschte arithmetische (+,−,·,/) oder logische (UND, ODER, NICHT, EXKLUSIVODER) Verknüpfung der Operanden durch.

Die CPU hat einen Instruktionspuffer und eine Instruktionsschlange, wodurch mehrere Instruktionen auf Vorrat aus dem Speicher geholt und dekodiert werden können. Dadurch kann eine Parallelarbeit von Speicherzugriffen, Dekodierung und der Arbeit der ALU erreicht werden, was die effektive Rechengeschwindigkeit erhöht. Demselben Ziel dient ein 8facher Pfad zum Hauptspeicher.

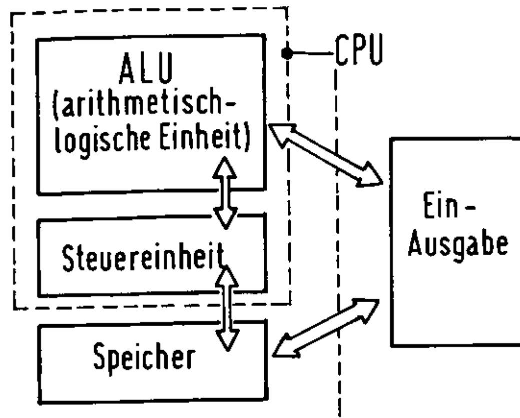


Abbildung 2.1: Schema einer VNM

Die CPU und der Speicher tauschen über die Ein-Ausgabeeinheit Information mit der Aussenwelt aus, in dem sie Binärworte aus dem Speicher oder der CPU an die Datenregister der E/A-Einheit übertragen (WRITE), oder Binärworte aus den Registern der E/A-Einheit übernehmen (READ).

Die Realisierung des Rechners aus elektronischen Schalt- und Speichereinheiten soll an dieser Stelle nicht beschrieben werden (s. die angegebene Literatur). In der Regel werden Bauelemente benutzt, die viele elementare Flip-flop-Speicher oder logische Gatter auf einer Siliziumscheibe (Chip) enthalten. Einfache integrierte Bauelemente ('Integrated Circuit' IC) enthalten einige bis einige Dutzend Bauelemente. Die sehr schnelle technische Entwicklung hat es ermöglicht, immer mehr Bauelemente auf einer Si-Scheibe unterzubringen (Large Scale Integration 'LSI') bis hin zu Chips, die 64 000 byte Speicher oder eine ganze CPU enthalten (Very Large Scale Integration 'VLSI'), (s. Abschnitt 9.4).

2.2 Darstellung von Zahlen und Symbolen

Die Einheit für die Darstellung von Zahlen, Symbolen, Instruktionen in den Registern des Speichers und der Zentralen Recheneinheit (CPU) sind gebräuchlicherweise das *binit*, von jetzt ab *bit* genannt, die 4-bit Hexadezimalzahl [1] (0,1,2, ... ,9,A,B, ... ,F), das byte (meist ein 8-bit Wort), und das Rechnerwort (gebräuchlich sind Wortlängen von 8, 16, 32 bits).

Ganze Zahlen werden als Dualzahlen gespeichert. Das vorderste bit kann zur Speicherung des Vorzeichens dienen, meist $+ \hat{=} 0$, $- \hat{=} 1$.
Beispiel: Speicherung von +10, in einem 16-bit Rechnerwort

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
oder:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+10	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Allgemein üblich ist eine zweite Art der Darstellung negativer Zahlen: das Zwei-Komplement. Es ist definiert durch:
Zweikomplement = Einskomplement + 1
Man erhält das Einskomplement, indem man in dem Zahlenwort jede 0 durch 1 ersetzt und jede 1 durch 0.
Man stellt dann $-b$ durch das Zweikomplement von b dar.
Dann gilt:

$$a - b = a + (\text{2-Komplement von } b)$$

[1] Beispiele: $9_{10} = 9_{16}; 12_{10} = C_{16}$

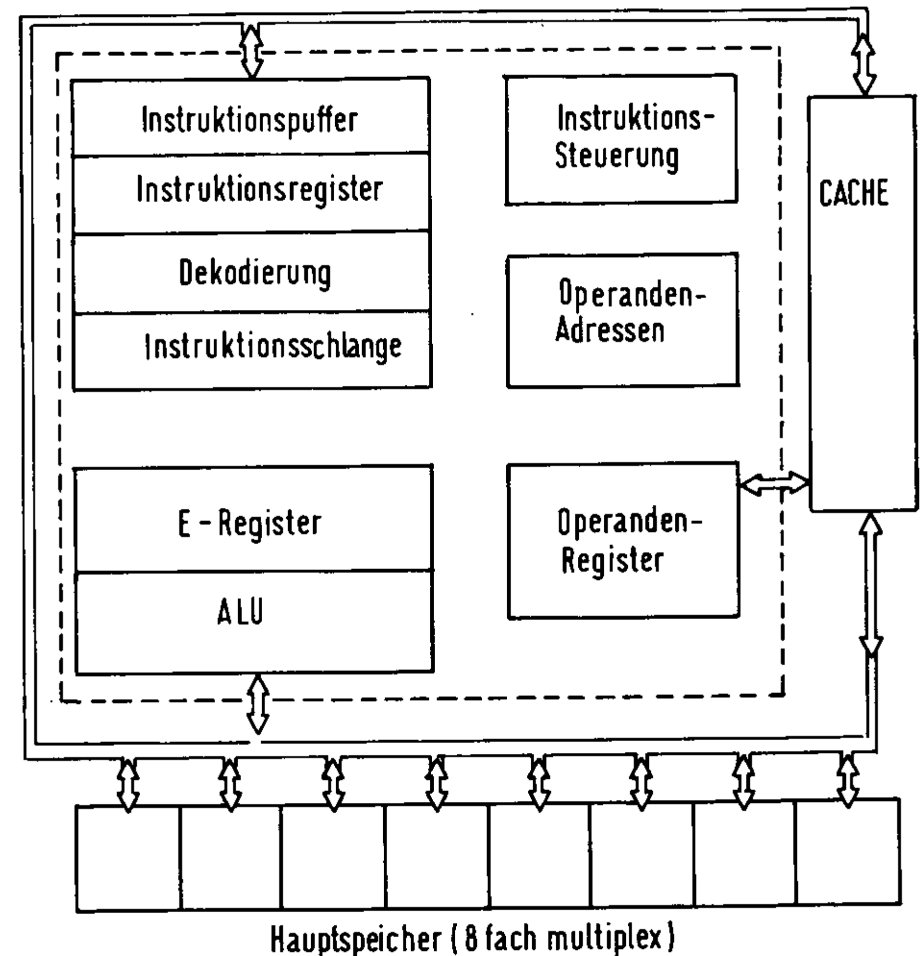


Abbildung 2.2: Schema einer zentralen Recheneinheit(CPU)

Da es leicht ist, von einer Zahl das Zweikomplement zu bilden, erspart dieses Vorgehen ein besonderes Rechenwerk für die Subtraktion.
Beispiel:

(i)	0 0 0 0 1 1 0 0	= +12
(ii)	0 0 0 0 1 0 1 0	= +10
(iii)	1 1 1 1 0 1 0 1	= Einskomplement von 10
(iv)	1 1 1 1 0 1 1 0	= Zweikomplement von 10 = −10
	0 0 0 0 0 0 1 0	= 12 − 10 = 12 + Zweikomplement von 10 = (i) + (iv); eine 1 läuft vorne über.

Man beachte, daß es zwei Arten von 0 in der Darstellung geben kann: +0 und −0, falls die Darstellung negativer Zahlen durch ein Vorzeichenbit erfolgt.

Echte Dualbrüche werden gespeichert in der Form

$$b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + b_3 \cdot 2^{-3} + \ldots,$$

wobei b_1, b_2, ... das vorderste, zweitvorderste, ... bit darstellen.
Beispiel: $11/16 = 0.1011 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16}$. Das duale System hat seine Tücken. Z.B. ist die dezimale Zahl $0.1_{10} = \frac{1}{10}$ ein unendlicher (periodischer) Dualbruch:

$$0.1_{10} = \frac{1}{10} = 0.00011001100_2 \ldots$$

Wegen der endlichen Wortlänge der Maschine ist dies nicht genau darstellbar und deshalb findet die Maschine z.B. 10 * 0.1(dezimal) ≠ 1.0 und die Physikerin kann mit Abfragen wie

```
IF((1. - 10. * 0.1).EQ.0.) ...
```

ihre Wunder erleben.

Gleitkommazahlen werden benutzt, wenn ein großer Zahlenbereich abgedeckt werden muß. Man schreibt sie in der Form

$$\text{Zahl} = F \cdot 2^E$$

und speichert F und E

$$E = \text{Exponent}$$
$$F = \text{Mantisse ('Fraction')}$$

F wird dargestellt als echter Binärbruch

$$(|F| < 1)$$

in der Form

$$(1 - 2b_0) \cdot (b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \ldots)$$

wenn das erste bit (b_0) als Vorzeichenbit benutzt wird. Es ist aber auch die Darstellung negativer Mantissen in Zwei-Komplementform üblich. Beispiel einer Darstellung (IBM, einfache Genauigkeit):

E	V	M
1 ... 7	8	9 ... 32

Der Exponent hat 6 bits plus ein Vorzeichenbit; bei der Darstellung im Speicher wird 2^5 zu E addiert, so daß die Darstellung von E stets ≥ 0 ist. V ist das Vorzeichenbit der Mantisse.

Die größte so darstellbare Zahl ist $\sim 2^{2^6-1}$, die kleinste darstellbare positive Zahl ist $\sim \frac{1}{2} \cdot 2^{-63} \simeq 2^{-64}$, falls die Mantisse wie üblich normiert ist.

"Normiert" bedeutet, daß $|F| \geq \frac{1}{2}$ ist, d.h. das erste bit der Mantisse ist stets eine 1. Auf diese Weise wird die maximale relative Genauigkeit bei der Zahlendarstellung erreicht. Diese ist bei einer 24-bit Mantisse etwa $2^{-25} \sim 3 \cdot 10^{-8}$.

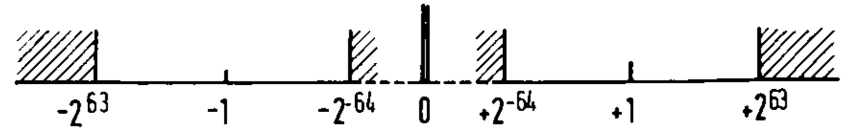


Abbildung 2.3: Zahlenbereich

Bei der IBM 3081 (und anderen IBM-Maschinen) ist das Element der Zahlendarstellung eine 4-bit Hexadezimalzahl. Für die Mantisse wird lediglich gefordert, daß die vorderste Hexadezimalzahl ≠ 0 ist; die Mantisse ist also i.a. nicht normiert und man verliert im Mittel 1.5 bits, in Extremfällen 3 bits an Genauigkeit.

Die Abb. 2.3 zeigt den mit dieser Darstellung zugänglichen Zahlenbereich. Man beachte die Sonderstellung der 0, für deren Darstellung eine besondere Konvention getroffen werden muß, z.B. alle bits = 0.

Falls man höhere Rechengenauigkeit benötigt, kann man zwei Worte zusammenfassen und so zu einer doppelt genauen Gleitkommazahl kommen. Die IBM benutzt in dieser Darstellung eine 56-bit Mantisse.

Für die Darstellung der Gleitkommazahlen existieren Normen. Trotzdem hat es der Individualismus der Menschen geschafft, selbst innerhalb dieser Normen Auslegungen ('Dialekte') zu finden, so daß selbst "genormte" Gleitkommadarstellungen nicht notwendigerweise kompatibel sind.

Fehler beim numerischen Rechnen erfolgen infolge der endlichen Genauigkeit der Zahlendarstellung. Falls das Ergebnis einer numerischen Rechnung in der vorgegebenen Zahl von Stellen nicht registriert werden kann, muß es entweder gerundet ('round') werden, oder die hintersten Stellen werden einfach abgeschnitten ('truncate'). Es sei nochmals erwähnt, daß so harmlos aussehende Zahlen wie 0.1 (dezimal) zu periodischen Dualbrüchen führen und deshalb nicht genau dargestellt werden können. Rundefehler können zu bösartigen Effekten führen, dies ist nicht verwunderlich, falls man bedenkt, daß eine Rechenmaschine mehr als 10^6 Rundefehler/s machen kann. Eine Prüfung auf Rundefehler bzw. Vermeidung derselben kann meist durch eine Kontrollrechnung mit doppelter Genauigkeit erfolgen. Oft ist die Wahl eines geeigneten Algorithmus von entscheidender Bedeutung für die Einhaltung der notwendigen numerischen Genauigkeit. Eine Überschreitung des Zahlenbereichs bei Gleitkommaoperationen heißt 'Overflow', eine Unterschreitung 'Underflow', Division durch 0 führt auf 'divide check'. Das Betriebssystem zeigt solche Fehler meist an; sie sind nützliche Hinweise auf Programmfehler. Eine Überschreitung des Zahlenbereichs bei Operationen mit ganzen Zahlen wird dagegen oft nicht angezeigt – dies ist eine große Gefahr.

Darstellung nicht numerischer Daten:

Hierbei handelt es sich um die Darstellung von Buchstaben, von Zahlen und von Sonderzeichen ('characters'). Diese werden meist in einem 6-bit oder einem 8-bit (byte) Binärwort verschlüsselt. Zur Freude der Anwender gibt es mehrere Konventionen:

BCD (binary coded decimal)

EBCDIC (Extended BCD interchange code)

ASCII (American standard code for information interchange).

Tab. 2.1 zeigt diese.

2.3 Instruktionen

Die Architektur eines Rechners spiegelt sich in seinem Instruktionssatz wieder. Jede Instruktion hat grundsätzlich drei Anteile:

(i) Art der Operation: 'was'

Character	BCD code	EBCDIC code	ASCII code	BCD cards	EBCDIC cards
blank	110 000	0100 0000	0100 0000	no punch	no punch
.	011 011	0100 1011	0100 1110	12,8,3	12,8,3
(	111 100	0100 11 01	0100 1000	0,8,4	12,8,5
+	010 000	0100 1110	0100 1011	12	12,8,6
$	101 011	0101 1011	0100 0100	11,8,3	11,8,3
*	101 100	0101 1100	0100 1010	11,8,4	11,8,4
)	011 100	0101 1101	0100 1001	112,8,4	11,8,5
-	100 000	0110 0000	0100 1101	11	11
/	110 001	0110 0001	0100 1111	0,1	0,1
,	111 011	0110 1011	0100 1111	0,8,3	0,8,3
'	001 100	0111 1101	0100 0111	4,8	8,5
=	001 011	0111 1110	0101 1101	3,8	8,6
A	010 001	1100 0001	1010 0001	12,1	12,1
B	010 010	1100 0010	1010 0010	12,2	12,2
C	010 011	1100 0011	1010 0011	12,3	12,3
D	010 100	1100 0100	1010 0100	12,4	12,4
E	010 101	1100 0101	1010 0101	12,5	12,5
F	010 110	1100 0110	1010 0110	12,6	12,6
G	010 111	1100 0111	1010 0111	12,7	12,7
H	011 000	1100 1000	1010 1000	12,8	12,8
I	011 001	1100 1001	1010 1001	12,9	12,9
J	100 001	1101 0001	1010 1010	11,1	11,1
K	100 010	1101 0010	1010 1011	11,2	11,2
L	100 011	1101 0011	1010 1100	11,3	11,3
M	100 100	1101 0100	1010 1101	11,4	11,4
N	100 101	1101 0101	1010 1110	11,5	11,5
O	100 110	1101 0110	1010 1111	11,6	11,6
P	100 111	1101 0111	1011 0000	11,7	11,7
Q	101 000	1101 1000	1011 0001	11,8	11,8
R	101 001	1101 1001	1011 0010	11,9	11,9
S	110 010	1110 0010	1011 0011	0,2	0,2
T	110 011	1110 0011	1011 0100	0,3	0,3
U	110 100	1110 0100	1011 0101	0,4	0,4
V	110 101	1110 0101	1011 0110	0,5	0,5
W	110 110	1110 0110	1011 0111	0,6	0,6
X	110 111	1110 0111	1011 1000	0,7	0,7
Y	111 000	1110 1000	1011 1001	0,8	0,8
Z	111 001	1110 1001	1011 1010	0,9	0,9
0	000 000	1111 0000	0101 0000	0	0
1	000 001	1111 0001	0101 0001	1	1
2	000 010	1111 0010	0101 0010	2	2
3	000 011	1111 0011	0101 0011	3	3
4	000 100	1111 0100	0101 0100	4	4
5	000 101	1111 0101	0101 0101	5	5
6	000 110	1111 0110	0101 0110	6	6
7	000 111	1111 0111	0101 0111	7	7
8	001 000	1111 1000	0101 1000	8	8
9	001 001	1111 1001	0101 1001	9	9

Tabelle 2.1: Beispiele verschiedener Kodes

(ii) Adresse d.h. Nummer des Speicherplatzes oder des Registers, wo der Operand steht (Operand = Information, auf welche die Operation wirkt): 'woher',

(iii) Adresse, wo das Ergebnis der Operation niedergelegt werden soll: 'wohin'.
Die Anteile (ii) und (iii) können aus mehreren Adressen bestehen oder fehlen.

Im folgenden werden einige Haupttypen von Instruktionen beispielhaft beschrieben. In der Praxis können andere, ähnliche oder kompliziertere Instruktionen implementiert sein.

Arithmetische Operationen:

Diese sind zu unterscheiden in Operationen, die auf ganze Zahlen und solche, die auf Gleitkommazahlen wirken.
Beispiel:
Addition einer Zahl, die unter der Adresse Nr. X gespeichert ist, zu einer Zahl, die unter der Adresse Y gespeichert ist, und speichern des Ergebnisses unter der Adresse Z.
Man schreibt dies so:

$$C(X) + C(Y) = C(Z)$$

$C(X)$ = Inhalt ('content') des Speicherplatzes Nr. X.
Diese Additionsinstruktion ist im Speicher der Maschine natürlich in Form eines Binärwortes gespeichert ('Objekt bzw. Maschinenkode'). Zur Betrachtung durch Menschen wird er in Buchstaben mit einer eingängigen mnemonischen Bezeichnung übersetzt – in der Regel besteht eine 1-1 Korrespondenz, man nennt den so entstehenden Kode 'Assemblerkode'. Die obige Additionsinstruktion könnte z.B. im Assemblerkode folgendermaßen aussehen:

ADD(X,Y,Z)

Diese Instruktion enthält 3 Adressen. Man kann die obige Addition auch durch eine Folge von Instruktionen mit einer einzigen Adresse darstellen, Beispiel:

	Kommentar
LOAD X	Bringe den Inhalt von Speicherplatz X in das Akkumulator-Register der CPU (Zentrale Recheneinheit): C(X) → C(AK).
ADD Y	Addiere den Inhalt von Speicherplatz Y zum Inhalt des Akkumulators: C(AK) + C(Y) → C(AK).
STORE Z	Bringe den Inhalt des Akkumulators in den Speicherplatz Z: C(AK) → C(Z).

Logische Operationen:

Gebräuchlich sind AND, OR, EOR (oder XOR) (exklusives Oder), NOT (Negation). Sie sind definiert durch die Wahrheitstafeln

AND (∩)

	0	1
0	0	0
1	0	1

OR (∪)

	0	1
0	0	1
1	1	1

EOR (⊕)

	0	1
0	0	1
1	1	0

NOT (¯)

	0	1
	1	0

Nützliche Beziehungen:

$$\left.\begin{array}{l}\overline{X \cap Y} = (\overline{X} \cup \overline{Y}) \\ \overline{X \cup Y} = (\overline{X} \cap \overline{Y})\end{array}\right\} \text{Morgan's Theorem}$$

$$X \oplus Y = (X \cup Y) \cap (\overline{X} \cup \overline{Y})$$

Beweis am einfachsten mit den Wahrheitstafeln. X, Y nennt man Bool'sche Variable. Sie können lediglich zwei Werte annehmen: Wahr (true, 1) und falsch (false, 0). In manchen Rechnern wirken die logischen Instruktionen auf Worte von mehr als einem bit.

Verschiebeoperationen

MOVE(X,Y) : C(X) → C(Y)

Verzweigungsinstruktionen:

Rechner besitzen im Steuerteil der CPU einen Instruktionszähler (IZ). Dies ist ein Register, welches die Adresse der nächsten auszuführende Instruktion enthält. Im infachensten Fall erhöht der Rechner den Inhalt des Instruktionszählers um 1, sobald eine Instruktion ausgeführt ist. Es werden somit die in aufeinanderfolgenden Plätzen des Speichers gespeicherten Instruktionen nacheinander ausgeführt. Die Verzweigungsinstruktionen gestatten, diese Sequenz der Instruktionen zu ändern.

Man unterscheidet:

(i) Unbedingte Verzweigung (J = jump):
J(X) bedeutet: X → C(IZ)
Die Zahl X wird in den Instruktionszähler geladen. Der Rechner führt also als nächste Instruktion die unter der Adresse Nr.X gespeicherte aus.

(ii) Bedingte Verzweigung:
JPL(X) (jump on plus)
JMN(X) (jump on minus)
JZE(X) (jump on zero)
JNZ(X) (jump on non-zero)
Der Rechner führt als nächste Instruktion die unter der Adresse X gespeicherte aus, falls der Inhalt des Akkumulators:
C(AK) > 0 : JPL
C(AK) < 0 : JMN
C(AK) = 0 : JZE
C(AK) ≠ 0 : JNZ
Falls die Bedingung nicht erfüllt ist, macht er in der normalen Sequenz weiter. Meist sind auch kompliziertere Bedingungen implementiert.

Verschiebe-Instruktionen:

Diese verschieben die bits in einem Wort um eine im Instruktionskode angegebene Zahl von Binärstellen nach rechts oder links und füllen die freiwerdenden Stellen mit Nullen auf.

Ein/Ausgabeinstruktionen:

Sie dienen dem Verkehr mit den Ein/Ausgabekanälen oder Ein/Ausgaberegistern.

2.4 Adressierungsmethoden

2.4.1 Indexregister

Sie dienen zur automatischen Modifikation von Adressen von Instruktionen. Ihr Nutzen ist am einfachsten an einem Beispiel ersichtlich. Angenommen, man hat die Aufgabe, die folgende Matrizenaddition durchzuführen:

$$A(I) = B(I) + D(I) \text{ für } I = 1,2,\ldots,1000.$$

Jeweils die erste Zahl der Matrizen B(I), D(I) sei unter der Adresse B1, D1 gespeichert, die übrigen Zahlen konsekutiv. Das Ergebnis soll konsekutiv in die Adressen beginnend mit A1 gespeichert werden. Es steht also A(1) unter der Adresse A1, A(2) unter der Adresse A1+1, u.s.w. Diese Aufgabe läßt sich ohne Indexregister durch das folgende Assemblerprogramm lösen:

			Kommentar
N	DEC	1000	Im Speicherplatz Nr.N wird die Dezimalzahl $1000 = 0000001111101000_2$ gespeichert
X	LOAD	B1	Im Speicherplatz Nr. X steht die Instruktion: C(B1) → C(AK).
Y	ADD	D1	Im Speicherplatz Nr. Y steht die Instruktion: C(D1) + C(AK) → C(AK).
Z	STORE	A1	Im Speicherplatz Nr. Z steht die Instruktion: C(AK) → C(A1)
	LOAD	X	C(X) → C(AK)
	ADD	=1	C(AK) → C(AK) + 1
	STORE	X	C(AK) → C(X): Diese Sequenz der letzten drei Instruktionen erhöht die Adresse der Instruktion LOAD B1 um 1, vorausgesetzt diese ist in den hintersten bits des Instruktionswortes gespeichert: Beispiel für eine Modifikation des Programms durch das Programm selbst.
	LOAD	Y	dto
	ADD	=1	
	STORE	Y	
	LOAD	Z	
	ADD	=1	
	STORE	Z	
	LOAD	N	
	SUB	=1	subtrahiere 1, im Akkumulator steht nun 999.
	STORE	N	
	JPL	X	füge als nächsten Befehl den im Speicherplatz Nr. X gespeicherten aus, falls der Inhalt des Akkumulators > 0 ist, sonst gehe zur nächsten darunterstehenden Instruktion über.

Dieses Programm hat einige Nachteile:

(i) es ist lang

(ii) die Adresse muß rechts in den Instruktionen gespeichert sein

(iii) das Programm modifiziert sich selbst - dies führt fast immer zu Schwierigkeiten, z. B. falls das Programm nach einer Programmunterbrechung weiterlaufen soll.

Mit Indexregistern sieht das Programm so aus:

	LOAD INDEX = 0,1	Lade eine 0 in das Indexregister Nr. 1: 0 → C(I1)
X	LOAD B1,1	C(B1 + C(I1)) → C(AK)
	ADD D1,1	C(D1 + C(I1)) + C(AK) → C(AK)
	STORE A1,1	C(AK) → C(A1 + C(I1))
	ADD TO INDEX 1,1	C(I1) + 1 → C(I1)
	COMPARE INDEX 1000,1	falls C(I1) = 1000, setze den Inhalt von I1 gleich 0, sonst tue nichts.
	TIX X,1	Nimm als nächsten Befehl den in der Speicheradresse X, falls C(I1) ≠ 0.

2.4.2 Indirekte Adressierung

Der Adressteil der Instruktion gibt die Adresse des Wortes im Speicher an, das die Adresse des Operanden

enthält. Beispiel:

LOAD * X (* bedeutet indirekte Adressierung)
LOAD * X : C(C(X)) → C(AK)

Die indirekte Adressierung wird häufig für den Rücksprung aus Subroutinen benutzt. Diese Aufzählung erschöpft nicht die Zahl der Adressierungsmöglichkeiten.

2.4.3 Virtuelle Adressierung

Maschinen mit virtueller Adressierung haben nicht das ganze Programm im schnellen adressierbaren Speicher. Das Programm steht vielmehr auf einem sekundären Speichermedium (langsamer Speicher oder Platte). Es wird jeweils nur ein Stück ('Seite', 'page') des Programms in den schnellen Arbeitsspeicher geladen; falls es durchlaufen ist, wirddas nächste Stück geladen. Man kann damit Programme von gewaltiger Länge bearbeiten. Der Nachteil besteht in dem zusätzlichen Aufwand, der auch durch das Ein- und Auslagern der Seiten entsteht. Außerdem erfordern diese Maschinen einen zusätzlichen Aufwand zur Berechnung der Adressen der aktuellen Operanden.

2.5 Speicher

Information kann auf verschiedenen Medien gespeichert werden. Ein wesentliches Unterscheidungsmerkmal ist die Zugriffszeit. Unter Zugriffszeit versteht man die mittlere Zeit zum Auffinden bzw. Speichern einer Informationseinheit. Als Informationseinheit hat man meist ein byte (=8-bit Wort) oder Worte größerer Länge (gebräuchlich sind 16 und 32 bits).

Ein zweites wesentliches Merkmal ist die Adressierungsmöglichkeit: Entweder kann zu jeder beliebigen Adresse zugegriffen werden ('ramdom access') oder es ist nur ein sequenzieller Zugriff zur nächstfolgender Speicheradresse möglich.

Das folgende ist eine Aufzählung von Speichermedien geordnet nach ihrer Zugriffszeit ("Speicherhierachie"):

Register	in der zentralen Recheneinheit (CPU)
CACHE-Speicher	aus integrierten Schaltelementen aufgebaut, mit besonders kurzer Zugriffszeit.
Hauptspeicher	meist aus hochintegrierten Schaltelementen aufgebaut, langsamer als der CACHE-Speicher
Plattenspeicher	auf Magnetplatten
Massenspeicher	enthalten addressierbare Magnetbandrollen, deren Inhalt unter Programmkontrolle auf den Plattenspeicher gebracht werden kann.
Magnetbänder	

Register, CACHE-Speicher und Hauptspeicher gestatten einen Direktzugriff per Adresse, während bei Plattenspeicher, Massenspeicher und Magnetbändern die Information nur zu sequenziellem Zugriff zugänglich ist. Die Abb. 2.4 zeigt für die verschiedenen Speichermedien die mittlere Zugriffszeit gegen die Speicherkapazität gebräuchlicher Speicher aufgetragen. Die folgenden Regeln, die natürlich nur grobe Richtwerte liefern können, gelten empirisch:

$$\text{Zugriffszeit x Speichergröße} \sim \text{Konstante}$$
$$\text{Zugriffszeit x Speicherpreis/bit} \sim \text{Konstante}$$

Man geht deswegen in der Praxis so vor, daß man Daten, die sehr häufig und mit kurzer Zugriffszeit benötigt werden, im direkt adressierbaren Speicher unterbringt, und die übrigen Daten, die nicht so häufig benötigt werden, je nach der voraussichtlichen Häufigkeit des Zugriffs auf Platten, Massenspeicher und Bänder.

Die Abb. 2.5 zeigt ein Beispiel der hierarchischen Anordnung von Speichermedien in einer Rechenmaschine mit CACHE-Speicher, Hauptspeicher und virtueller Adressierung: Der sehr schnelle CACHE-Speicher (Größe

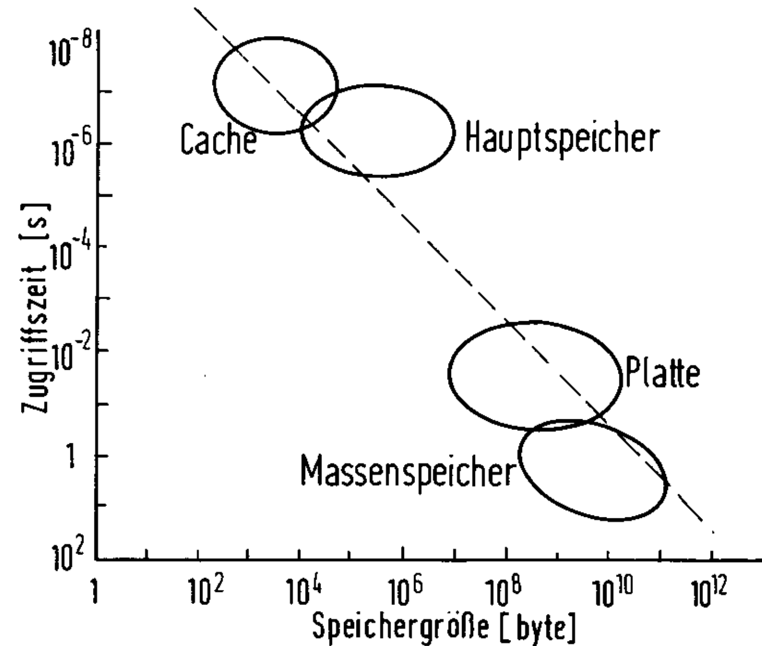


Abbildung 2.4: Zugriffszeiten

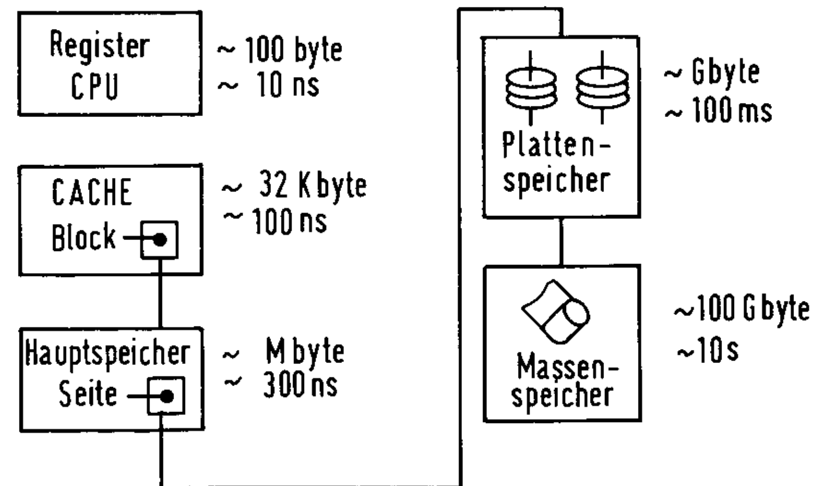


Abbildung 2.5: Speicherhierarchie

typisch einige kbyte bis 64 kbyte (IBM 3033)) ist in Blöcke (kleinere Speichereinheiten) eingeteilt, die nach den Erfordernissen des ablaufenden Programmes aus dem Hauptspeicher geladen werden. Die Strategie wird so gewählt, daß nach Möglichkeit alle vom ablaufenden Programm benötigte Information im CACHE-Speicher ist. Großrechner haben zur besseren Ausnutzung der CPU meist mehrere Programme parallel im Speicher, und sie werden abwechselnd ausgeführt. Die Hauptspeicher moderner Großrechner können einige Mbyte fassen, doch reicht dies oft für die gleichzeitige Speicherung mehrerer Großprogramme nicht aus. Man bezieht deshalb in die Hierarchie den Plattenspeicher mit ein. Der Hauptspeicher wird in Segmente ("Seiten") eingeteilt, und die Information wird unter der Kontrolle des Betriebssystems von der Magnetplatte mit einer Seite als Einheit in den Hauptspeicher gebracht. Dieses Verfahren gestattet den Einsatz sehr großer Programme ohne Einschränkung durch die Größe des Hauptspeichers. Da ein Teil des Programms auf der Platte ausgelagert sein kann, nennt man diese Technik 'virtuelle Adressierung'. Man erkauft sie sich durch eine Verlangsamung des Programmablaufs im Mittel, hervorgerufen durch die Notwendigkeit, Programmteile zwischen Hauptspeicher und Platte austauschen zu müssen.

2.6 Externe Speichermedien

2.6.1 Magnetband

Gebräuchlich ist heutzutage Magnetband von 1 Zoll Breite, auf dem die Daten in 9 parallelen Spuren gespeichert werden. Eine der 9 Spuren enthält in der Regel die Paritätsbits zu Paritätsprüfung der übrigen 8 Spuren. Die Schreibdichte in der Längsrichtung des Bandes beträgt je nach Typ 800, 1600, 3200 oder 6250 bits/Zoll (bits per inch = bpi). Die Bandlänge beträgt etwa 800 m, so daß ein Band mit 6250 bpi maximal 800 x (100/2.54) x 6250$\simeq$200 Mbytes speichern kann (in der Praxis kriegt man maximal etwa 160 Mbytes drauf). Als erste Information des Bandes steht meisten eine Bandmarke ('label'), welche das Band identifiziert, und die auch noch andere Informationen (z.B. über das Bandformat) enthalten kann.

Die Einheit beim Lesen oder Beschreiben eines Bandes ist der Block oder Datensatz ('Record'). Unter der Länge eines Blocks oder 'Records' versteht man die Zahl der bits, die in Längsrichtung konsekutiv stehen, und dies ist bei einem 9-Spur Band mit Paritätsbit auch die Zahl der bytes, die in einem Block zusammengefaßt sind. Jeweils nach dem Lesen oder Schreiben eines Blocks wird das Band angehalten. Um ein einwandfreies Lesen bzw. Schreiben unabhängig von Anlaufeffekten des Bandes zu gewährleisten, werden die Blöcke durch ein Stückchen unbeschriebenes Band getrennt ('Interrecordgap'), seine Länge heißt 'Kluftlänge', sie beträgt 0.6 Zoll (in Sonderfällen 0.3 Zoll). Um eine gute Bandausnutzung zu erhalten, muß man große Blocklängen wählen. Abschreckendes Beispiel: Schreiben eines FORTRAN-Programmes auf Band, als Blocklänge wird die Standardlänge 80 bytes einer FORTRAN-Instruktion benutzt. Sie entspricht der Zahl der Spalten auf einer Lochkarte, damit lassen sich 80 BCD-Zeichen bzw. bytes speichern. Die Länge eines solchen Blocks auf einem 6250 bpi-Band beträgt 80/6250 = 0.013 Zoll. Die Bandausnutzung bei einer Länge von 0.6 Zoll für ein 'interrecordgap' beträgt also $0.013/(0.013 + 0.6) \approx 2\%$. Man sieht hieran, wie nützlich es ist, große Blocklängen zu verwenden. Da man aber die Information, die zu einem Block gehört, in einem Pufferspeicher im Rechner zwischenspeichern muß, benötigen sehr lange Blöcke sehr viel Zwischenspeicher. Außerdem kommen sehr lange physische Blöcke oft auch mit der logischen Struktur der Daten in Konflikt. Diese widerstrebenden Forderungen führen jeweils auf eine günstigste Blocklänge. Hat man eine Bandausnutzung von mehr als etwa 80%, lohnt es sich meist nicht, die Blöcke noch länger zu machen.

Werden Bänder beschrieben, so wird automatisch bereits darauf stehende Information gelöscht. Um einen Schutz gegen unbeabsichtigtes Löschen zu gewähren, haben Bänder einen Schreibring. Wird er entfernt, kann das Band nicht beschrieben werden.

Da das Band nur sequentiell gelesen bzw. beschrieben werden kann, läst sich neue Information nur am Ende der schon bestehenden anfügen. Will man neue Information mitten drin zufügen, oder alte Information durch neue ersetzen, so muß das Band umkopiert werden.

Bänder können mit den verschiedensten Formaten beschrieben werden. Die bytes können Teile von Binärworten der verschiedensten Wortlänge bedeuten, sie können Zeichen im BCD oder ASCII oder einem anderen Kode bedeuten, die Blocklänge kann fest oder variabel sein, das Band kann 7 oder 9 Spuren haben, es kann 800, 1600, 3200, oder 6250 bpi haben, es kann eine Bandmarke haben oder keine, etc. Hat man ein Band zu lesen, von dem man diese Dinge nicht weiß, so ist man in einer traurigen Lage. Man werfe das Band weg oder bittet einen Spezialisten um Hilfe.

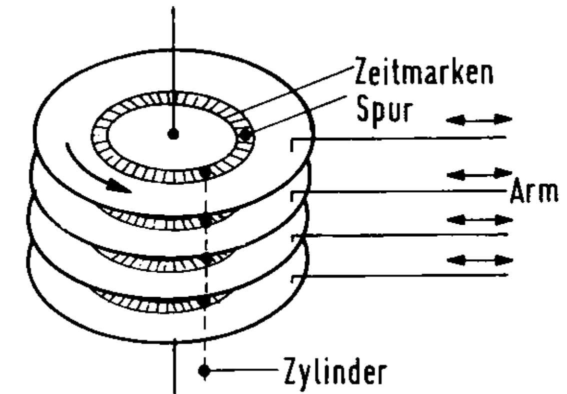


Abbildung 2.6: Schema eines Plattenspeichers

Um beim Anlaufen bzw. Abstoppen des Bandes möglichst wenig Masse bewegen zu müssen, hat man Bandeinheiten mit geistreicher Mechanik entwickelt.

2.6.2 Plattenspeicher

Die Abb. 2.6 zeigt eine Skizze. Die Information ist auf jeder Platte auf Spuren mit konstantem Radius angeordnet. Meistens sind mehrere Platten übereinander in einem Stapel. Alle Spuren auf diesen Platten, die denselben Radius haben, bilden einen Zylinder. Daten, die auf einem Zylinder liegen, sind ohne Bewegung der Lese/Schreibarme zugänglich. Die (mechanische) Bewegung der Schreibarme und die Umdrehungszeit der Platte bestimmen die grundlegenden Zeitkonstanten für den Zugriff zu Information auf der Platte. Die Spuren enthalten Zeitmarken (Formatierung), so daß Information gezielt angesteuert werden kann.

In diesem Sinn ist die Platte adressierbar und zusammen mit formatierten Magnetbändern wird sie als DAC (direct access device) bezeichnet. Auf ihnen kann Information durch andere (geringerer oder gleicher Länge) ersetzt werden.

Als Beispiel für die Ausführung eines großen Plattenspeichers sei hier der Plattenspeicher IBM 3350 aufgeführt. Er hat 555 Zylinder und 30 Spuren/Zylinder. Auf jede Spur gehen 19 kbytes. Die Gesamtkapazität ist also 317 Mbyte. Die Umdrehungszeit ist 16 ms.

Neben der oben skizzierten Ausführung mit mehreren Platten in einem Laufwerk und beweglichen Lese-/Schreibarmen gibt es Platten mit festen Lese/Schreibköpfen, Wechselplatten und Platten mit einfacher billiger Ausführung (Diskette, Floppy disc).

2.6.3 Massenspeicher

Er enthält eine Reihe von bandähnlichen Kasetten, die vollautomatisch ausgewählt und gelesen bzw. beschrieben werden können – es ist sozusagen ein von Robotern bedientes Bandlager.

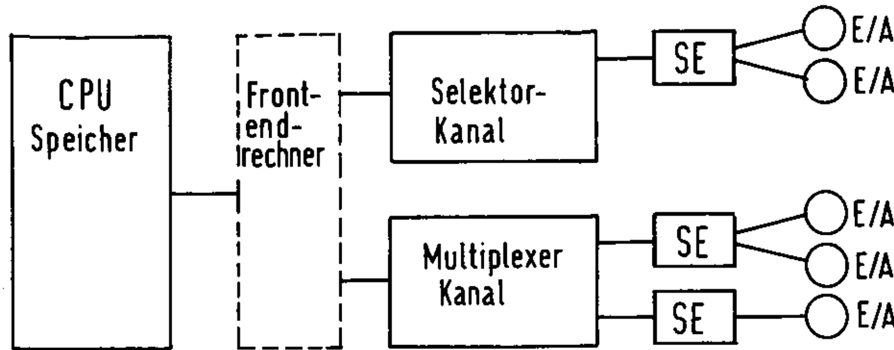


Abbildung 2.7: Organisation der Ein/Ausgabe

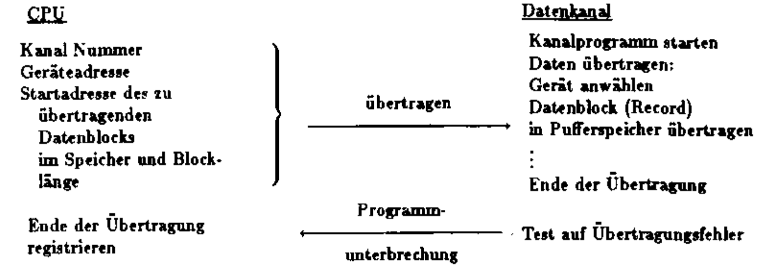
Die untenstehende Tabelle gibt eine Übersicht über die wichtigsten externen Speichermedien.

	Magnetband	Plattenspeicher	Massenspeicher
Beispiel	IBM 3420	IBM 3350	IBM 3850
Kapazität [Mbyte]	200 (6250 bpi)	317 pro Laufwerk	169000 (Anlage DESY)
Übertragungs-rate [kbytes/s]	320/1250 Lesen/Schreiben (6250 bpi)	1200	Übertragung auf Plattenspeicher
Zugriffszeit Min/Mittel/Max	1 min/10 min/ 1 Woche	10/25/50 ms	15–30 s

2.7 Ein/Ausgabe

Hierunter versteht man die Übertragung von Daten zwischen dem Rechner und externen Geräten (z.B. Bandgeräte, Plattenspeicher, Kartenleser, Drucker, Sichtgerät, Konsole). Da die externen Geräte mechanisch bewegte Teile haben, sind sie im Vergleich zur Geschwindigkeit des Rechners sehr langsam. Es gibt eine Reihe von Verfahren, die verhindern sollen, daß der Rechner durch die E/A-Operationen ungebührlich stark gebremst wird. Im gebräuchlichsten Verfahren wird die E/A-Operation delegiert an einem Spezialrechner. Diesen nennt man (E/A)-Kanal. Die Abb. 2.7 zeigt schematisch einen Rechner mit E/A Geräten und Kanälen.

Kanäle ermöglichen das Übertragen von Daten zwischen dem Speicher des Rechners und einem E/A-Gerät ohne Beanspruchung der CPU des Rechners. Dies zeigt das folgende Beispiel für das Wechselspiel zwischen Rechner und Datenkanal bei einem Schreibvorgang auf ein externes Gerät.



DMA (direct memory access) benutzt keine standardisierten E/A-Kanäle. Bei diesem Verfahren wird ein Wort direkt von einem äußeren Register in einen bestimmten Speicherplatz übertragen bzw. ein Speicherwort direkt in ein äusseres Register übertragen. Dazu unterbricht man den Rechner mit einem Signal (DMA-Request), und stellt die Speicheradresse in einem Register bereit. Zur Übertragung des Datenwortes benutzt man einen Rechnerzyklus ('Cycle-stealing') oder die Einrichtung eines Multiport-Memories (MPM). Man benutzt DMA zum Anschluss unkonventioneller Geräte oder für Operationen, die zeitkritisch sind. Allerdings muß auch der Benutzer die geschilderten Steuersignale selbst zurechtbasteln. Bei Maschinen mit virtueller Adressierung muß man darauf achten, daß einem der Rechner nicht Teile des Speichers während der DMA Operation unter dem Hintern wegzieht, und muß für die Umrechnung virtuelle → physikalische Speicheradresse Sorge tragen.

2.8 Programmunterbrechung (Interrupt)

Rechner müssen die Möglichkeit haben, ein laufendes Programm zu unterbrechen. Bedingungen hier für können sein:

- illegale arithmetische Operation (overflow, underflow, divide check)
- illegale Instruktion
- zugewiesene Rechenzeit überschritten
- Maschinenfehler (z.B. Auftreten eines Paritätsfehlers)
- Ein/Ausgabebedingung (z.B. Ende einer E/A Operation)

Eine Programmunterbrechung kann auf ein Interruptsignal hin technisch folgendermaßen ablaufen (einfaches Beispiel):

(1) Springen auf einen festen Speicherplatz, wo das Interrupt- verwaltungs- und Bedienungsprogramm losgeht.

(2) Retten des Instruktionszählers (IZ), der die Rücksprungadresse in das laufende Programm enthält, auf einen dafür vorgesehenen Speicherplatz (X).

(3) Interruptstatuswort (ISW) setzen. Dieses regelt die Priorität zwischen verschiedenen Interupts – es kann ja ein neuer Interrupt eintreten, solange ein alter noch abgearbeitet wird.

(4) Register der CPU retten, falls nötig.

(5) Aktion durchführen.

(6) Register wieder herstellen, falls nötig.

(7) ISW zurücksetzen.

(8) Rücksprung ins laufende Programm: J * X

Um mehrere Interrupts zeitlich überlappend bedienen zu können, kann man mehrere Prioritätsebenen für die Bedienung von Interrupts einführen.

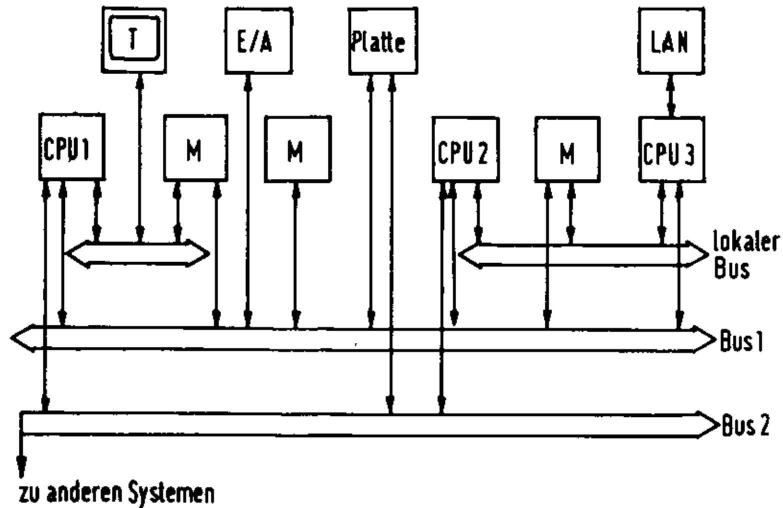


Abbildung 2.8: Multiprozessorsystem

2.9 Systeme von Rechnern

Die maximale Rechengeschwindigkeit ist schlußendlich begrenzt durch die endlichen Signallaufzeiten zwischen den Elementen des Rechners. Höhere Rechengeschwindigkeiten lassen sich im Prinzip durch Parallelarbeiten von Rechner erreichen. Diese Dinge, die gegenwärtig eine wichtige Entwicklungslinie darstellen, können hier nicht behandelt werden.
Klassische Verfahren sind Pipeline-Rechner,
Vektorrechner und Multiprozessoren.
Als Beispiel für ein Multiprozessorsystem zeigt Abb. 2.8, wie man eine Reihe von Rechnern, Speichermoduln und externen Geräten über gemeinsame Datenleitungen (Bus) verbinden kann. Dies gestattet es, mehrere Aufgaben gleichzeitig mit den Recheneinheiten CPU1, CPU2,... in Angriff zu nehmen. Diese können über den Bus an gemeinsame Programme und Daten zugreifen und die E/A-Einheiten ansprechen. Ein solches System ist im Prinzip durch Anhängen immer weiterer Rechner und Speichereinheiten erweiterungsfähig. Die Kontrolle des Datenverkehrs über den Bus erfordert besondere Aufmerksamkeit. Hierzu gibt es umfangreiche Literatur.

Vektor- und Pipeline-Rechner gestatten es, mehrere Instruktionen in ein und demselben Programm gleichzeitig zu bearbeiten. Voraussetzung ist, daß die Ergebnisse der Operationen nicht von vorhergehenden Operationen abhängen, und daß das bearbeitete Programmstück keine Verzweigungsinstruktionen enthält. Die Leistungsfähigkeit solcher Rechner hängt demgemäss stark von der Programmstruktur ab.

Literatur

[1] C. W. Gear. *Computer Organization and Control*, McGraw Hill

[2] *CERN School of Computing*, Zinal 1982, CERN 83-03 (Kap.2.9)

[3] V. Schmidt. *Digital elektronisches Praktikum*, Teubner

[4] H.-J. Stuckenberg. *Digitale Logik*, G. Braun

[5] H.-J. Stuckenberg. *Die Silizium-Hardware der Rechenmaschinen*, Interner DESY-Bericht F56-84-01

Kapitel 3

PASCAL

WARUM?... "goes my
conviction, that the language
in which the student is taught
to express his ideas profoundly
influences his habits of thought
and invention, and that the disorder
governing these languages directly
imposes itself onto the programming
style of the students".
N.Wirth, "Introduction to PASCAL"

3.1 Vorbemerkung

Das Ziel dieses Kapitels ist es, einen Eindruck der Möglichkeiten von PASCAL zu vermitteln. Es ersetzt keinesfalls ein Lehrbuch für diese Sprache.

Es folgen einige Erklärungen zur Notation.

- Fett gedruckte Worte sind reservierte Namen (**keywords**): sie dürfen nicht als Namen von Konstanten, Variablen, Typen oder Prozeduren benutzt werden.
- Namen für Konstanten, Variablen, Typen, Prozeduren werden mit dem Wort Namen (identifier) bezeichnet.
- Ausdrücke in < > sind nicht Bestandteil der Sprache. Sofern sie im Programm vorkommen, sind sie durch entsprechende Ausdrücke zu ersetzen.
- Alles was in { } steht, kann beliebig oft wiederholt werden.
- Das Symbol ::= enthält auf seiner rechten Seite die Definition des links stehende Begriffs.
- Das Symbol ; markiert das Ende einer Instruktion.

3.2 Programmaufbau

Jedes Programm besteht aus einer Sequenz der sieben Programmteile A-G.

A. Programm-Überschrift
<Programm-Überschrift> ::= **program** <Name> (<Dateiname1>, <Dateiname2>, ...);
Beispiel: **program** convert (output);

B. Marken-Deklaration
label <label1> {, <label>};
Alle Marken müssen deklariert werden.

C. Konstantendefinition
const <Name> = <Konstante>; {<Name> = <Konstante>;}
Beispiel: **const** addin = 32; mult = 1.8; separator = '...';
Alle Konstanten müssen definiert werden.

D. Typen-Definition
type <Name> = <Typ>; {<Name> = <Typ>;}
Alle Typen müssen deklariert werden.
Die Typendeklaration und die darin liegenden Möglichkeiten sind ein wichtiger Bestandteil der Sprache und werden später erklärt.

E. Variablen-Deklaration
var <Name> {, <Name>}: <Typ>; {<Name>{, <Name>}:<Typ>;}
Beispiel: **var** X1, X2: real; Zähl, i: integer; fertig: boolean; füllsel: char;
Alle Variablen müssen deklariert werden.

F. Unterprogramm- und Unterfunktionsdeklaration (Prozedur)
-falls verwendet.

G. Operationsteil
Dieser Teil enthält die Instruktionen. die das Programm ausführen soll.

3.3 Vokabular

<Buchstabe> ::= A | B | ... | Z

<Ziffer> ::= 0 | 1 | 2 | ... | 9

<Spezialsymbol> ::= + | − | * | / | < | > | | ; | ↑ | = | . | , | : | ' | (|) | { | } | ¬ | [|]

<Name> ::= <Buchstabe>{<Buchstabe oder Ziffer>}

<Zeichen (character)> ::= <Buchstabe> | <Ziffer> | <Spezialsymbol>

<Text (string)> ::= '<Zeichen>{<Zeichen>}'
3 Beispiele: 'A', ';', 'Dies ist ein Text'

<Reservierte Namen> ::= siehe später (3.)

Operationen

:=	Zuweisung
+	Addition oder Vereinigungsmenge
−	Subtraktion oder Differenzmenge
*	Multiplikation oder Durchschnitt von Mengen
div	Division (ganze Zahlen)
/	Division (ganze oder reelle Zahlen), Ergebnis reell
mod	Modul (ganze Zahlen)
=	Gleichheit (die Ausdrücke müssen vom selben Typ sein)
<	kleiner als
>	größer als
<=	kleiner oder gleich (bei Mengen: Teilmenge)
>=	größer oder gleich
not	Negation (wirkt auf boolsche Variable)
or	logisches oder (wirkt auf boolsche Variable)
and	logisches und (wirkt auf boolsche Variable)
in	Test auf Existenz eines Elements einer Menge

Standard-Namen

Konstanten:	false, true
Typen:	integer, boolean, real, char, text
Programmparameter:	input, output

Funktionen

(i) arithmetische:
abs(x) = | x |, sqr(x) = x*x, sin(x), arctan(x), exp(x), ln(x), sqrt(x) = $\sqrt{x}$, trunc(x), round(x)

(ii) boolsche:
odd(x) = true if x is odd, else = false, eoln(f) = true if, while writing the textfile f, the end of the line is reached, eof(f)

(iii) Zeichenmanipulation:
succ(x) = der in einer Reihe auf x folgende Wert, pred(x) = der vorhergehende Wert. Außerdem gibt es eine Reihe von Prozeduren zur Dateimanipulation und Zeigerverwaltung.

Reservierte Namen

div, mod, nil, in, or, and, not,if, then, else, case of, repeat, until, while, do, for, to, downto, begin, end, with, goto, const, var, type, array, record, set, file, function, procedure, label, packed, program

3.4 Datentypen

Standarddatentypen sind: integer (ganze Zahl), real (reelle Zahl), boolean (Boolsche Variable, hat einen der beiden Werte wahr oder falsch), character (Zeichen).
Eigene Datentypen kann man über den **type**-Namen definieren.

Beispiel:

type boolean = (false, true); color = (red, green, blue);
    weekday = (monday, wednesday);

Dies macht zusammen mit den Funktionen succ, pred Bildungen möglich wie:

if c > green **then** c := pred(c)

Erklärung:
falls die Variable c (vom Typ color) hinter (>) dem Namen green kommt in der Aufzählung der Namen in ( ), die zu dem Typ color gehören, dann ersetze c durch den Namen, der c in der ( ) vorhergeht (pred), also: falls c = blue, wird c durch die Anweisung durch green ersetzt, anderenfalls (c = green oder = red) passiert garnichts.

Manchmal wäre es mühsam, alle zu einem Typ gehörenden Namen explizit aufzuzählen. Um das zu vermeiden, führt man *Bereichstypen* (subrange types) ein.

Beispiel:

type letter = ('a' .. 'z');
    day = (monday .. friday);

Diese Typendefinitionen schließen alle zwischen 'a' und 'z' bzw. monday and friday liegenden Elemente mit ein. Dies ist durch die gerade verwendete Version der Sprache näher definiert.

Strukturierte Datentypen enthalten eine Strukturierung oder Gliederung. Es gibt die folgenden:

Vector(array): Er enthält lauter gleichartige Elemente. Er wird durch einen oder mehrere Indizes bezeichnet, die es gestatten, auf ein ganz bestimmtes Element des Vektors zuzugreifen.

Datei (file): besteht aus beliebig vielen gleichartigen Elementen, die ihrerseits wieder strukturiert sein können. Zugriff über get/put bzw. über read/write.

Datensatz (record): Hierzu sind ausführlichere Erklärungen nötig: Ein Datensatz besteht aus einer gegebenen Anzahl nicht notwendig gleichartiger Komponenten. Die Komponenten heißen Felder, sie haben einen Namen (field identifier), der sie bezeichnet. Sie werden über diesen Namen, nicht über Indizes angesprochen. Die Felder brauchen nicht vom selben Typ zu sein.

Beispiele:

type datum = **record** mo: (jan .. dez);
        tag:(1 .. 31); jahr: integer **end**
    Spielzeug = **record** art:(ball, puppe, buch);
        preis: real; empfang: datum;
        gefallen : (gut, etwas, schlecht);
        kaput, verloren : boolean **end**

Falls eine Variable (s) vom Typ Spielzeug definiert ist, kann man die Felder wie folgt zuweisen:

s.art := ball
s.preis := 12.50
s.gefallen := etwas
s.kaput := false

Datensätze (records) können geschachtelt werden, wie im vorigen Beispiel bei Spielzeug, wo empfang vom Typ datum ist, welches seinerseits als Datensatz definiert ist. Dies zeigt ein weiteres Beispiel:

type person = **record** name: **record** vorn, famn: alfa
        end;
steunr: integer; sex: (maenl, weibl);
geburtstag: datum;
case famstd: status **of**
    verh: (vdatum:datum);
    gesch: (gdatum:datum);
    ledig: (unabh:boolean)
    end

Dies zeigt an einem Beispiel die **case**-Anweisung. famstd kann verh, gesch oder ledig sein, der Typ von status wird durch die Namen verh, gesch, ledig geregelt. Beispiel für Felderzuweisung:

person.name.famn:= 'lohrmann'
person.sex:= maenl
person.famstd:= verh
person.verh.mo:= mai

Programmbeispiel für eine Kalenderroutine bei Überschreiten der Monatsgrenze:

if datum.mo = 12 **then**
    begin datum.mo:= 1, datum.jahr:= datum.jahr+1
    end
else datum.mo:= datum.mo+1

3.5 Der ausführende Teil

Dieser enthält als das eigentliche Programm die auszuführenden Instruktionen, während die Programmteile A-F lediglich Definitionen enthalten, welche natürlich nicht ausgeführt werden.

Die wichtigsten Elemente beim Aufbau des ausführenden Teils sind die folgenden:

(i) Zuweisung (assignment): Einer Variablen wird ein gewisser Wert zugewiesen, Beispiele:

```
     Zahl   := Zahl + 1
     sqrp   := sqr(p)
        y   := sin(x) + cos(x)
 gefunden   := y > z
```

Beispiel 1) zeigt, daß das := - Zeichen kein = -Zeichen ist. In Beispiel 4) ist der Typ von gefunden boolean.

(ii) Zusammengesetzte Anweisung (compound statement): Eine Reihe von Anweisungen kann als zusammengehörig bezeichnet werden, wenn sie zwischen **begin** ... und **end** eingeschlossen sind.

(iii) Schleifen (loops). Es gibt davon drei Typen, je nachdem wo die Abbruchbedingung der Schleife stehen soll:

```
while <Ausdruck> do <Anweisung>
repeat <Anweisung>{;<Anweisung>}until<Ausdruck>
for <Scheifenkontrollvariable>:= <Anfangswert>
to/downto <Endwert>
do <Anweisung>
```

(iv) Verzweigungen:

```
if <Ausdruck> then <Anweisungen>
if <Ausdruck> then <Anweisung> else<Anweisung>
```

Hier ist der Ausdruck vom Typ boolean; hat er den Wert true, so wird die auf **then** folgende Anweisung ausgeführt, andernfalls geschieht nichts (1. Fall) oder es wird (2. Fall) die auf **else** folgende Anweisung ausgeführt.

3.6 Programmbeispiele

(i) Berechne

$$\sum_{i=1}^{n} \frac{1}{i}$$

```
program comp(input,output)
var n : integer; h : real;
begin read(n); write(n); h := 0;
      while n > 0 do
      begin h := h + 1/n; n := n - 1;
      end
writeln(h);
end
```

Dieses Programm, wenn man ihm n = 10 zu lesen gibt, produziert die Ausgabe:
10 2.92896

2. Version:

```
program comp(input, output)
var m : integer; h : real;
begin read(n), write(n); h:= 0;
if n>0 then;
begin repeat h:= h + 1/n; n:= n - 1;
until n = 0;
end
writeln(h)
end
```


3. Version:

```
program comp(input, output)
var i,n : integer; h : real;
begin read(n); write(n); h:= 0;
for i = n downto 1 do h:= h + 1/i;
writeln(h)
end
```

(ii) Berechnung einer ganzzahligen Potenz x^y unter Verwendung von Zwischenergebnissen der Multiplikation. Ausdrücke in {} sind Kommentare und gehören nicht zu den Anweisungen des Programms.

```
program potenzierung (input, output)
var e,y : integer; u,x,z : real;
begin read(x,y): write(x,y); z:= 1; u:= x; e:= y;
while e> 0 do
begin {z * u ** e = x ** y, e > 0}
      while not odd(e) do
            begin e:= e div 2; u:= sqr(u)
            end
      e:= e - 1; z:= u * z
end
writeln(z) {z = x^y}
end
```

(iii) Suche nach dem größten und kleinsten Element eines Vektors, zeigt die Verwendung von Prozeduren (Unterprogrammen)

```
program main (input, output)
const n = 20;
var a:= array [1.. n]of integer;
      i,j : integer

procedure minimax
var i : 1 .. n; u, v, min, max : integer
begin min:= a[1 ]; max := min; i := 2
      while i < n do
      begin u:= a[i]; v:= a[i+1];
      if u> v then
      begin if u>max then max := u;
            if v<min then min := v;
      end else
      begin if v>max then max := v;
            if u<min then min := u;
      end
            i := i + 2
      end

      if i = n then
            if a[n]>max
                  then max := a[n];
            else if a[n]< min
                  then min := a[n];
            writeln(max,min)
      end {minmax}

begin {lies liste}
      for i := 1 to n do
```

```
        begin read(a[i]); write (a[i])
        end; minimax;
        for i := 1 to n do
        begin read(j); a[i]:= a[i]
              + j; write (a[i]);
        end; minimax;
        end
```

(iv) Mengen:

```
program woche (output)
type tage=(mo,di,mi,do,fr,sa,so);
        woche=setof tage{definiert eine Menge};
var wch,arb,frei : woche; t : tage;
procedure check (s : woche)
var t : tage
begin write (' ');
        for t := mo to so do
if t in s then write('x')else write('o');
        writeln
        end{check};
        begin arb:= []{leere Menge};
              frei:= []; wch:= [mo .. so];
              t:= sa; frei:= [t]+frei+[so];
              check(frei);
              arb:= wch-frei; check(arb);
              if frei<= wch then write('0');
              if wch>=arb then write ('k');
        if not (arb>=frei) then write ('jack');
        if[sa]<=arb then write('vergisses');
        writeln
        end
```

Dieses Programm erzeugte die folgende Ausgabe:

```
0 0 0 0 0 x x
x x x x x 0 0
o k j a ck
```

Literatur

[1] K. Jensen und N. Wirth. "*PASCAL*", Springer Verlag

Kapitel 4

Erstellung von Programmen

'Wenige schreiben, wie ein Architekt baut, der zuvor einen Plan entworfen und bis ins einzelne durchdacht hat; vielmehr die meisten nur so, wie man Domino spielt. Kaum daß sie ungefähr wissen, welche Gestalt im ganzen herauskommen wird, und wo das alles hinaus soll. Viele wissen selbst dies nicht, sondern schreiben, wie die Korallenpolypen bauen. Subroutine fügt sich an Subroutine, und es geht, wohin Gott will'
frei nach Schopenhauer

4.1 Allgemeines

Schritte bei der Erstellung eines Programms:

1) Festlegung der Ausgabe, Festlegung der Eingabe: mit der Festlegung der Ausgabe ist die Form der gewünschten Lösung dokumentiert. Die Ausgabe soll als erstes Namen und Funktion des Programmes, Datum und Version, und dann die Eingabedaten zur Kontrolle und Dokumentation enthalten. Danach folgen die Ergebnisse. Die Bedeutung der Zahlen des Ergebnisses muß im Klartext mit ausgedruckt werden.

2) Festlegung des Programmablaufs: S. Kap. 4.2

3) Programmierung (Kodierung): S. Kap. 4.3 und Kap. 5

4) Programm-Prüfung: S. Kap. 6

5) Dokumentierung und Programmbeschreibung.

4.2 Hilfsmittel zur Festlegung des Programmablaufs

(i) Strukturierte Programmierung:
Jedes Programm wird aufgebaut aus einer (geschachtelten) linearen Folge von Grundstrukturblöcken (Moduln), von denen es drei verschiedene Sorten gibt. (Siehe Abb. 4.1) Jedes Modul hat genau einen Eingang und einen Ausgang. Jeder Kasten in einem Modul kann selbst wieder ein Modul sein: Schachtelung. Die Schleife existiert in drei Spielarten je nach der logischen Anordnung der Abbruchbedingung der Wiederholung. Es ist streng verboten, in eine Schleife mit einem GO TO hereinzuspringen. Man vermeide überhaupt

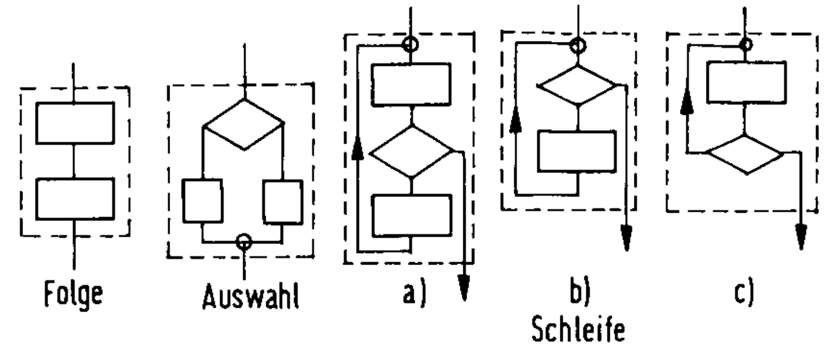


Abbildung 4.1: Strukturierte Programmierung

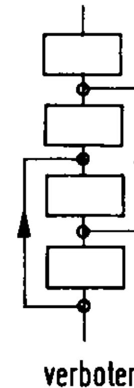


Abbildung 4.2: verboten

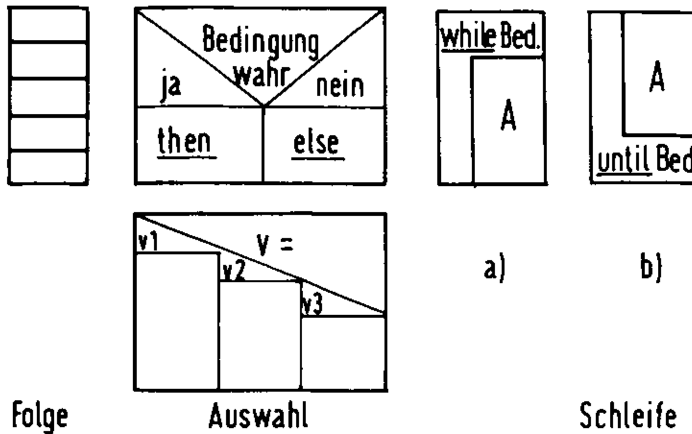


Abbildung 4.3: Struktogramme

GO TO soweit als möglich, weil diese Anweisung die logische Struktur des Programms unübersichtlich macht.

Eine Anordnung des Programs nach diesen Prinzipien macht die Sache übersichtlich und erleichtert das Testen,da man sich die Moduln einzeln vornehmen kann.

Für die Aufstellung der Struktur gibt es zwei Verfahren, die man auch in Kombination anwenden kann:

- "top down":
  Hier wird die Aufgabe zuerst in großen Umrissen, d.h.in allgemeinen Moduln festgelegt. Dann erfolgt eine schrittweise Festlegung der Details durch Ausfüllen der großen Blöcke in Schachtelmanier.
- "bottom up":
  Man beginnt mit den wichtigen Moduln, die die wichtigsten Aufgaben festlegen. Davon ausgehend wird das Programm dann nach allen Seiten ergänzt.
  Hilfsmittel für das strukturierte Programmieren sind Subroutinen und Makros.

(ii) Struktogramme:
Sie eignen sich gut zur Darstellung mehrfacher logischer Verzweigungen. Darstellung der drei Strukturelemente in Abb. 4.3: Ein Beispiel (Programm zum Einlesen von Lochkarten bzw. ihrem Bild auf Band) zeigt Abb. 4.4.

(iii) HIPO (Hierarchy plus Input-Process-Output):
Dieses von IBM stammende Verfahren gestattet, die Verknüpfung des Programms mit den Ein- und Ausgabedateien darzustellen.

Im einfachsten Fall sieht ein solches Schema folgendermaßen aus (Abb. 4.5).

(iv) Entscheidungstabellen:
Es ist eine Erfahrungstatsache, daß bei der Umsetzung logischer Entscheidungsstrukturen in ein Programm besonders häufig Fehler gemacht werden. Der Mensch ist eben ein zutiefst unlogisches Wesen. Deshalb sind Entscheidungstabellen ein wichtiges Hilfsmittel, die logische Entscheidungsstruktur zu dokumentieren. Manchmal begreift man mit ihrer Hilfe überhaupt erst die eigene Logik. Entscheidungstabellen sind weiterhin auch ein wichtiges Hilfsmittel zum Austesten des Programms, welches die Struktur angeblich darstellen soll.Eine Entscheidungstabelle in der kanonischen Normalform führt alle einzelnen Bedingungen tabellarisch auf. Im Programm werden sie durch boolsche Variable dargestellt. Ist die Zahl der Bedingungen n, so gibt es 2^n mögliche verschiedene Kombinationen (Regeln), die im Prinzip zu 2^n verschiedenen

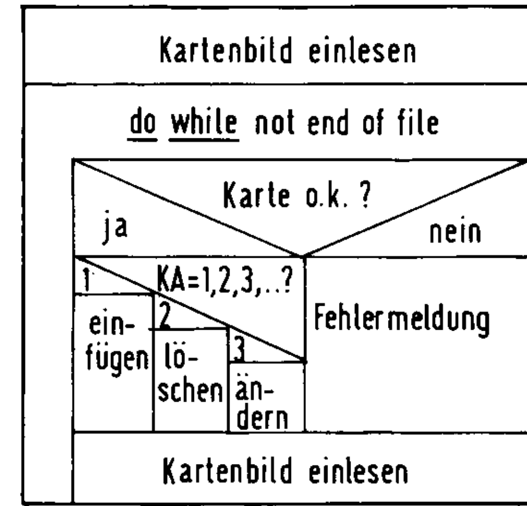


Abbildung 4.4: Struktogramm zum Einlesen eines Kartenbildes

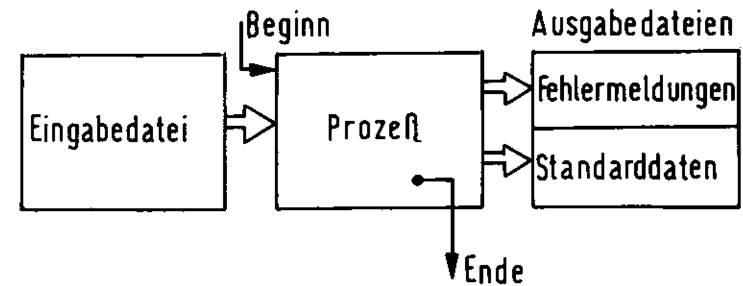


Abbildung 4.5: HIPO

Tabelle 4.1

Regel Nr.	1	2	3	4	5	6	7	8
Bedingung 1	T	T	T	T	F	F	F	F
Bedingung 2	T	T	F	F	T	T	F	F
Bedingung 3	T	F	T	F	T	F	T	F
Handlung: 1 FEHLER =	5	5	1	5	5	3	2	4

T = wahr , F = falsch

Regel Nr.	1	2	3	4
gutes Wetter?	T	T	F	F
Fahrrad kaputt?	T	F	T	F
Handlung	S	R	S	S

S = Fahrt mit Straßenbahn
R = Fahrt mit Fahrrad

Tabelle 4.1: Entscheidungstabellen

Aktionen führen können, wie Tabelle 4.1 zeigt. Eine solche Entscheidungstabelle könnte man z.B. mit Kommentar-Karten im Programm unterbringen, damit man weiß, was das Programm soll. Für die Umsetzung der Entscheidungstabelle in ein Programm gibt es Programmpakete. Man kann auch den gesunden Menschenverstand benutzen.

4.3 Stilfragen der Programmierung

Unter diesem Punkt werden eine Reihe von Regeln für die Praxis zusammengefasst. Diese zielen in erster Linie daraufhin, das Programm übersichtlich, leicht verständlich, wartungs- und benutzerfreundlich zu machen. Beachte: Der Autor des Programms gehört auch zu den Benutzern, und hat in der Regel nach einiger Zeit vergessen, wie sein eigenes Programm funktioniert.

Um ein Programm übersichtlich und die Logik einfach zu machen, darf man in vielen Fällen eine Einbuße an Rechengeschwindigkeit oder ein Mehr an Speicherplatz in Kauf nehmen, da die Personalkosten des Programmierers die Kosten der Rechner-Hardware übertreffen. Ausnahmen gibt es bei sehr rechenzeitintensiven und sehr häufig benutzten Programmen, z.B. bei Standard-Auswerte-Programmen, Plot-Routinen u.a.

Die folgenden Dinge dienen dem o.a. Zweck:

(i) Optische Gliederung des Programms:
Abschnitte durch Leerzeilen oder auffällige Kommentare markieren. Einrücken von Moduln, z.B. von DO-Loops. Strukturieren von Instruktionen:
schlecht:

```
IF(A.GE.5.)GOTO5
```

gut:

```
IF(A .GE. 5.) GO TO 5
```

schlecht:

```
6001  FORMAT(1H0,3X, ...
     110X, ... )
```

gut:

```
6001  FORMAT(1 H0, 3X, ...
     1          10X, ... )
```

(ii) Daten- und Aktionsteil trennen (s. den Abschnitt über PASCAL): Damit ist gewährleistet, daß am Programmkopf alle Konstanten definiert sind (alphabetisch ordnen). Es erspart die Suche durch das ganze Programm, wenn man den Wert einer Konstanten wissen oder ändern will. Man vergißt bei Änderungen weniger leicht eine konsequente Berücksichtigung aller Konstanten.

(iii) Namenwahl:
Man wähle nur mnemotechnisch gut überlegte Namen für die Konstanten und Variablen.

schlecht:

```
IF(STAND.EQ.3) ...
```

gut:

```
LOGICAL VERH
IF(STAND.EQ.3) VERH=.TRUE.
IF(VERH) ...
```

Keine kurzen Namen im COMMON wählen, also nicht: COMMON A,B,X
Gefahr: irgendwo im Programm setzt man X=...als temporäre Variable und hat vergessen, daß X im COMMON steht, mit katastrophalen Konsequenzen.

(iv) Alle FORMAT-Instruktionen und Fehlertexte an einer Stelle sammeln.

(v) Schnittstellen und Parameterübergabe an Subroutinen sorgfältig dokumentieren. Zu Beginn jeder Subroutine muß stehen, was die SR tut, weiterhin sollen Datum und Autor von Programmänderungen vermerkt werden. Jedes Programm hat Eingabeparameter. Als Ergebnis liefert es Ausgabeparameter ab. Dies muß für jedes Programm dokumentiert sein. In diesem Sinne muß dokumentiert sein, welche Variablen des COMMON das Programm benutzt und welche es verändert.

(vi) Bei der Programmierung keine Tricks anwenden - einfach und übersichtlich schreiben. Dabei können (von Ausnahmefällen s.o. abgesehen) Einbuße an Rechengeschwindigkeit und eine Vergrößerung der Zahl der Instruktionen in Kauf genommen werden. Beispiel:

schlecht:

```
      DO 1 I = 1,N
      DO 1 J = 1,N
    1 X(I,J) = (I/J) * (J/I)
```

gut (und schneller):

```
      DO 1 J = 1,N
      DO 2 I = 1,N
    2 X(I,J) = 0.
    1 X(J,J) = 1.
```

Logische Strukturen sind besonders fehleranfällig. Man vermeide Sprungbefehle (GOTO) zugunsten boolscher Variablen. Beispiel:

schlecht:

```
      LO=0
      IF(A.EQ.B) GO TO 10
      GO TO 20
   10 IF(C.EQ.D) LO=1
   20 CONTINUE
```

gut:

```
      LOGICAL LO
      LO = (A.EQ.B).AND.(C.EQ.D)
```

schlecht:

```
      XO(J)=(2 * MOD(J,2)-1) * DD
```

gut:

```
      XO(J)=DD
      IF(MOD(J,2).EQ.0) XO(J) = -DD
```

(vii) Möglichst keine Zahlen mit Ausnahme von 0,1,2 im Programm verwenden, also schlecht:

```
      DO 50 I = 1, 12
      A(I) = B(I,31)
      u.s.w.
```


gut:

```
      DATA NI,NB/12,31/
      DO 50 I = 1, NI
      A(I) = B(I,NB)
```

Grund: Falls man später die 12 oder 31 ändern muß, ist die Gefahr groß, eine Stelle im Programm zu übersehen, wo die Zahl auch noch vorkommt.

(viii) Weitere Maßnahmen, die das Programm übersichtlicher machen:
Strukturierung durch Verwendung von Subroutinen und Funktions-Subroutinen. Dieselben sollten nicht zu lang sein.
Benutzung von Bibliotheksroutinen, soweit möglich. Dieselben sind meist effizienter als selbstgeschriebene. Voraussetzung: Man muß (genau!) wissen, was die Routine tut - gegebenenfalls testen.
Setzen von Klammern:

schlecht:

```
      X=A * B/C * D
```

gut:

```
      X=((A * B)/C) * D
```

Die "Statement"-Nummern sind ein Mittel zur Strukturierung - in ihrer Aufeinanderfolge muß Ruhe und Ordnung herrschen.
Logische Ausdrücke sind eine große potentielle Fehlerquelle. Man halte sie so einfach wie möglich.

(ix) Verabscheuenswürdige, schlechte Praktiken:
Benutzen von Zwischenvariablen.
Benutzen des arithmetischen IF. Es ist logisch schwerer zu durchschauen und oft gefährlich (s.w.u.).
Benutzen von Sprungbefehlen (GO TO) im Übermaß. Sie machen die logische Struktur unübersichtlich.
Keine 'mixed mode', d.h. Mischung von Ausdrücken vom Typ INTEGER und REAL, wie $M = 1.2 + 5$

Dies ist uneffektiv, da die Maschinen jedesmal eine Typ-Umwandlung vornehmen muß.
Abfragen auf Gleichheit zweier Zahlen vom Typ REAL, z.B:

```
      X = 1.
      IF(X.EQ.(0.1 * 10.)) ...
```

was die Maschine damit macht, weiß Gott allein. Wenn man es doch meint, tun zu müssen, schreibe man:

```
      DATA EPS/1.E - 5/
      IF(ABS(X - 0.1 * 10.).LT.EPS) ...
```

(x) Passe die Datenstruktur der Aufgabe an, verlege die Komplexität des Problems in die Datenstruktur und nicht in die Logik. Beispiel Kalender:
schlecht:

```
      IF(NMONAT.EQ.1)  NTAGE=31
      IF(NMONAT.EQ.2)  NTAGE=28
      IF(NMONAT.EQ.12) NTAGE=31
```

gut:

```
      INTEGER KTAGE(12)/31,28,31, ... , 31/
      NTAGE=KTAGE(NMONAT)
```

(xi) Lesen von Dateien:

```
C      Lies den ersten Datensatz (Record)
   10 READ(6)A
C      Pruefung von A oder eines Teils von A darauf, ob
C      es der letzte Record der Datei ist, angezeigt durch
C      einen ''End-of-file-Markierer'' EOFM.
C      Man verlasse sich nie auf das Abzaehlen der Records,
C      um auf die End-Bedingung der eingelesenen Daten zu kommen.
       IF(A.EQ.EOFM) GOTO 100
C      Es folgt das Programm zur Behandlung der eingelesenen Daten
       GO TO 10
C      Ende dieses Programmteils
C      Endbehandlung
  100 CONTINUE
```

(xii) Regeln für Ein/Ausgabe:

Verwende Standard-Eingabeformate
Verwende, wenn möglich, formatfreie Eingabe,
z.B. 17.853 { }3.58 { }0.05 u.s.w.
Finde das Datenende nicht durch Abzählen, sondern
durch eine Daten-Ende-Markierung
Wiederhole die eingegebenen Daten in der Ausgabe
Prüfe Eingabedaten auf Konsistenz, Plausibilität,
Vollständigkeit
Fange schlechte Eingabedaten ab
Dokumentiere die Ausgabe sorgfältig und ausführlich.

(xiii) Regeln für Kommentare:

Grobe Faustregel: im Mittel 1 Kommentarkarte/5 Instruktionen.
An jedem Programmanfang:
Eingabeparameter,
Ausgabeparameter, Zweck der Routine
Dokumentiere alle wichtigen Variablen und Konstanten
Dokumentiere Datum und Autor von Programmänderungen
Erkläre Algorithmen und die Physik von Teilschritten
z.B. mit Pseudocode und Entscheidungstabellen

4.4 Compiler

Rechenmaschinen verstehen Instruktionen nur in Form von Binärworten. Ein ausführbares Programm, wie es z.B. im Speicher eines Rechners steht, besteht also aus solchen Binärworten. Es heißt Maschinenprogramm, Objektcode oder Maschinencode. Es wäre für den Menschen sehr mühsam und fehlerträchtig, ein solches Maschinenprogramm von Hand zu erstellen. Deswegen hat man Programme entwickelt, welche Instruktionen in einer höheren Programmiersprache (z.B. ALGOL, COBOL, FORTRAN, PASCAL) in Maschinencode übersetzen. Solche Programme nennt man Compiler. Als Zwischenstufe hat man eine Sprache entwickelt, welche die Instruktionen von Maschinencode auf mnemotechnisch vernünftig gewählte Instruktionen abbildet (Assemblersprache).
Die Übersetzung des Assemblercodes in Maschinencode ist verhältnismässig einfach. Programme, welche dies leisten, nennt man Assembler.

Abb. 4.6 zeigt die Programmschritte zur Erstellung eines ausführbaren Programms aus einem in einer höheren Programmiersprache geschriebenen Programm. Erstaunlicherweise kann man Compiler in einer höheren Programmiersprache schreiben. Man kann z.B. einen FORTRAN-Compiler in FORTRAN schreiben. Dies ist nützlich, wenn man den Compiler ändern will oder ihn von einem Rechner auf einen anderen verpflanzen will. Die Abb. 4.7 zeigt schematisch, wie es gemacht wird. Der A-Code ist ein Zwischen-Micky-Maus-Assemblercode,

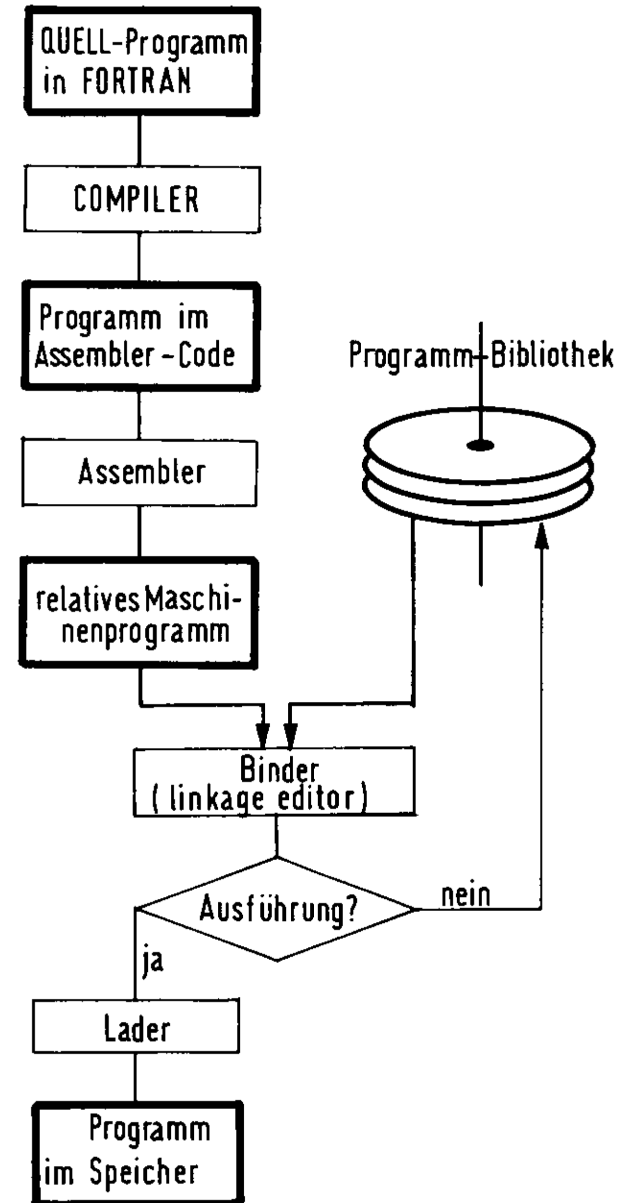


Abbildung 4.6: Schritte zur Erstellung eines lauffähigen Programms

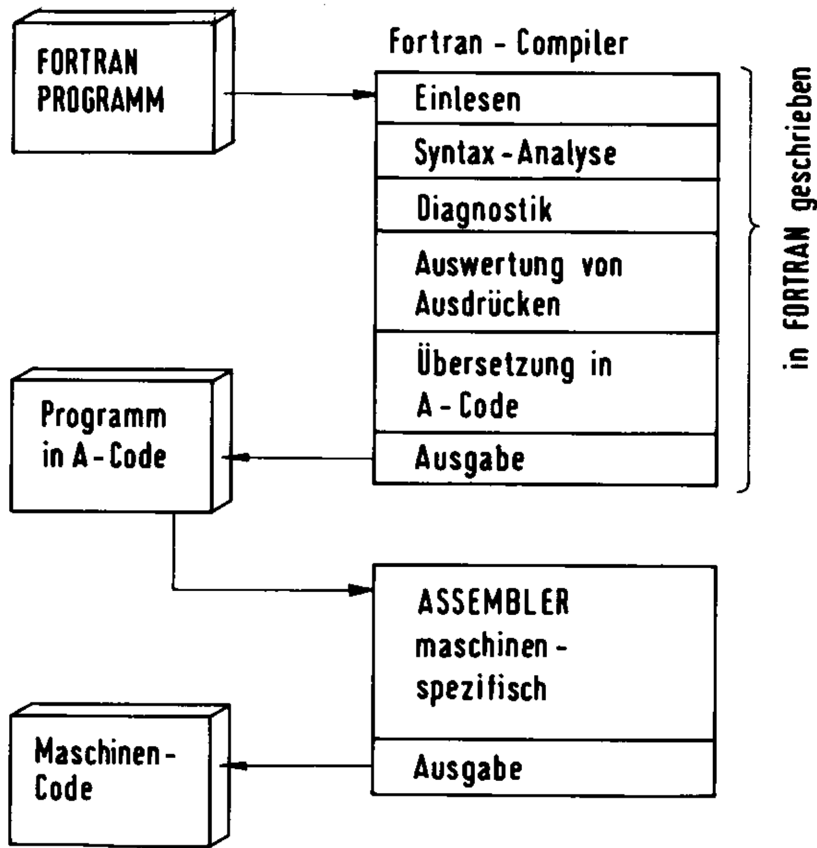


Abbildung 4.7: Compiler

der maschinennah ist und i.a. leicht in die spezielle Maschinensprache übersetzt werden kann. Nur dieser Teil muß der speziellen Maschine angepasst werden. Beispiel: Ref.BCPL

Literatur

[1] O. Hell. *Code in Haste, Debug at Leisure*, DESY Int.Rep.R1-81/03, Sehr zu empfehlen; viele der Beispiele dieses Kapitels sind daraus entnommen.

[2] F. Singer. *Programmieren in der Praxis*, Teubner

[3] BCPL. *The Language and its Compiler*, Richards/Whitby-Stevens, Cambridge University Press (Kap.4.4)

Kapitel 5

Programmoptimierung

"Don't do it"

5.1 Allgemeine Regeln

Die Entwicklung immer schnellerer und kostengünstigerer Rechenwerke, und der Anstieg der Lohnkosten bei der Erstellung von Programmen hat zu einem Umdenken bei der Kostenoptimierung für die Erstellung von Programmen geführt.

Es ist heute i.a. günstiger, Programme so zu schreiben, daß sie durchsichtig, leicht zu verstehen, leicht zu ändern und leicht zu warten sind. Dafür können Kompromisse hinsichtlich der Effektivität des Kodes gemacht werden. Dazu kommt, daß moderne optimierende Compiler i.a. eine ziemlich gute Übersetzung in Maschinensprache liefern, selbst wenn das zugrundeliegende FORTRAN Programm ziemlich lausig geschrieben ist.

Hieraus ergeben sich folgende Regeln:
Mache das Programm einfacher, klarer und sicherer gegen Fehler, bevor du es schneller machst. Opfere nicht Klarheit für kleine Gewinne an Schnelligkeit. Benutze keine undurchsichtigen Tricks, um das Programm schneller zu machen. Lasse den Compiler die Routine-Optimierung machen. Unterstütze ihn durch klare Programmlogik, vermeide rückwärtsgerichtete GO TO,"Spaghetti" u.a. mehr.
10-90-Regel: 10% des Programms benutzen 90% der Rechenzeit. Beginne die Optimierung mit diesen Programmteilen. Hier lohnen sich bei Programm mit Rechenzeiten > Wochen u.U. wilde Tricks, Umschreiben in Assembler u.ä. Um diese Teile zu finden, benutze man Zeitmeß (sog. Spy-)Routinen des Rechenzentrums (vergiss nicht, sie hinterher wieder zu entfernen). Oft sind es die innersten DO-Schleifen, welche zeitkritisch sind.
Mehr als jede noch so tolle Optimierung kann die Wahl eines guten Algorithmus, Änderungen in der Struktur oder Logik bringen.

5.2 Spezielle Regeln

Hier sollen einige einfache Regeln beschrieben werden, welche Programme effizienter machen ohne Einbuße von Klarheit, und die i.a. nicht die Domäne optimierender Compiler sein können, weil der Compiler die Absichten des Programmierers nicht kennt.

1) Benutze wenn möglich Systemroutinen, sie machen den Code durchsichtiger und sind meist schneller.
   Beispiel:
   schlecht:

```
IF (X.LT.Y) GO TO 30
```

```
      IF (Y.LT.Z) GO TO 50
      SMALL = Z
      GO TO   70
   30 IF (X.LT.Z) GO TO 60
      SMALL = Z
      GO TO   70
   50 SMALL = Y
      GO TO   70
   60 SMALL = X
   70 CONTINUE
```

gut:

```
      SMALL = AMIN1(X,Y,Z)
```

2) Hilf dem Compiler optimieren durch passende Schreibweisen.

Beispiel:

schlecht:

```
      A = Z*X*B*Y
      C = D*X*Y*Z
```

gut:

```
      A = (X*Y*Z)*B
      C = (X*Y*Z)*D
```

3) Indizierte Variable:
Versuche, die Zahl der Indizes (Dimension) niedrig zu halten, da die Berechnung hochdimensionaler Indizes viel Zeit erfordert. Dies sollte jedoch nicht auf Kosten der logischen Durchsichtigkeit geschehen. Versuche, möglichst wenige DO-Schleifen zu schachteln, da dann oft die Indexregister effizienter eingesetzt werden. Lasse den Rechner die Indexregister zur Indexberechnung benutzen. Beispiele:

schlecht:

```
      I5 = I+5                    DO  1 I = 1,10
      J3 = J-3                    J =  3*I + 4
      K2 = K*2                  1 X(J) = Y(J)+C
      A(I5,J3,K2) = ...
```

gut:

```
      A(I+5,J-3,K*2) = ...        DO  1 I = 7,34,3
                                1 X(I) = Y(I)+C
```

Gleiche Dimensionierung macht Indexberechnung effizienter:

schlecht:

```
      DIMENSION A(9,9), B(5,9)
      DO  1     I = 1,9
    1 A(1,I) = B(1,I)
```

gut:

```
      DIMENSION A(9,9), B(9,9)
      DO  1     I = 1,9
    1 A(1,I) = B(1,I)
```


schlecht:

```
      LOGICAL*1 K(9)
      LOGICAL*4 L(9)
```

gut:

```
      LOGICAL*4 K(9), L(9)
```

4) Gestatte dem Rechner die Einhaltung natürlicher Wortgrenzen in der Anlage des COMMON, also sollen im COMMON REAL*8-Variable vor LOGICAL/REAL/INTEGER*4 und diese vor INTEGER*2 und diese vor LOGICAL*1 stehen.

5) Schleifen: Vermeide kurze Schleifen - hoher Verwaltungsaufwand für Einrichten der Schleife und Endabfrage.

schlecht:

```
      DO 1 I = 1,2               DO  10  J = 1,3
    1 A(I) = 0.               10 KK(3*J) = 3+J
```

gut:

```
      A(1) = 0.                  KK(3) = 4
      A(2) = 0.                  KK(6) = 5
                                 KK(9) = 6
```

Bei mehreren Schleifen schreibe die am öftesten wiederholte als die innerste, wenn möglich. Dies verringert die Zahl der Initialisierungen und Endabfragen.

Dieser Trick halbiert den Verwaltungsaufwand in der Schleife (N = gerade):

```
      DO  1 I= 2,N,2
      A(I-1) = B(I-1)*C(I-1)
1     A(I)   = B(I) * C(I)
```

Ziehe gemeinsame Aktionen aus der Schleife heraus:

schlecht:

```
      DO  1    I = 1,N
      DO  1    K = 1,N
      C(I,K)   = 0.
      DO  1    J = 1,N
    1 C(I,K) = C(I,K)+A(I,J)*B(J,K)
```

gut:

```
      DO  1    I = 1,N
      DO  1    K = 1,N
      SUM = 0.
      DO  2    J = 1,N
    2 SUM = SUM + A(I,J)*B(J,K)
    1 C(I,K) = SUM
```

6) Verzweigungen:
Das logische IF ist schneller als das arithmetische

schlecht:

```
      IF(I-2) 2,1,2,
```

gut:

```
IF(I.EQ.2) GO TO 1
```

Bei mehreren

```
IF(cond.1) GO TO ... ← häufigstes
IF(cond.2) GO TO ...
```

hintereinander, schreibe das am häufigsten erfüllte als erstes.

7) Subroutinen und Funktionen:
Beachte, daß An- und Rücksprung einer Subroutine einen erheblichen Verwaltungsaufwand erfordern (Retten verschiedener Register), also keine zu kurzen Subroutinen schreiben, falls diese häufig aufgerufen werden.

Funktionsroutinen können manchmal aufwendig sein. Beachte dies:

schlecht:

```
I = 4
A = B**I
```

gut:

```
A = B**4
```

schlecht:

```
A = SIN(TH) + 1.
B = 3.-COS(TH)**2
```

gut:

```
SINT = SIN(TH)
A = SINT+1.
B = 3.-(1.-SINT**2)
```

schlecht:

```
X = SQRT(A)*SQRT(B)
```

gut:

```
X = SQRT(A*B)
```

aber schlecht:

```
IF(A.LT.0.)A = -A
IF(I.LT.0) I = 0
```

gut:

```
A = ABS(A)
I = MAX(I,0)
```

8) Arithmetik:
+ - ist schneller als * ist schneller als / ist schneller als ** also:

schlecht:

```
2.*A, A/2., A**2, A**0.5
```


gut:

```
A + A, 0.5 * A * A, SQRT(A)
```

schlecht:

```
Y=A3*X**3+A2*X**2+A1*X+A0
```

gut:

```
Y=((A3 * X + A2) * X + A1) * X + A0

  (Horner-Schema)
```

schlecht:

```
A = B/C/D/E
```

gut:

```
A = B/(C*D*E)
```

9) keine "mixed mode" verwenden, d.h. keine Ausdrücke verwenden, die Variablen oder Konstanten verschiedenen Typs enthalten, da die Typ-Umwandlung viel Zeit erfordert.

schlecht:

```
IF(Y.EQ.4)  J = 10.
```

gut

```
IF(Y.EQ.4.) J = 10
```

schlecht:

```
  DO 1   I = 1,N
1 A(I) = I
```

gut:

```
  X = 1.
  DO  1  I = 1,N
  A(I) = X
1 X = X + 1.
```

10) Manipulation von Text, z.B. in einem Editor oder Compiler, kann sehr uneffektiv sein.

11) E/A-Gabe: Wähle die größte vernünftige Blocklänge, unformatierte E/A-Gabe ist besser als formatierte. (Ausnahme: Übergang von einem Rechner zu einem anderen, dann ist oft EBCDIC Format gut). Die Eigenart des IBCOM-Aufrufs auf IBM-Anlagen bringt es mit sich, daß folgendes gut/bzw. schlecht ist:

schlecht:

```
            WRITE(6) (A(I), B(I), I = 2,10)
                 -9 Aufrufe von IBCOM
```

gut:

```
            WRITE(6) (A(I), I = 2,10), (B(I), I = 2,10)
                 -2 Aufrufe von IBCOM
```

12) Optimierende (und andere) Compiler haben manchmal ihre Tücken - sie sind u.U. nicht ganz comme il faut. Beispiel aus der Praxis (aus einem TASSO-Auswerteprogramm):

```
        SUBROUTINE LACLUS (IHD, IDE, IMOD)
        COMMON/LACL/NCLUST,ICLUST(100)
        IDONE=0
        NCLUST=1
100     IDONE=IDONE+1
        IHD=ICLUST(IDONE)
        IF(IDONE.GT.NCLUST)GO TO 200
        CALL LAUMGT(IHD,IDEV,IMOD)
        NCLUST=NCLUST
        GO TO 100
200     RETURN
        END
```

Die sinnlose Instruktion NCLUST=NCLUST ist notwendig, andernfalls sagt sich der Compiler folgendes: Zwischen Instruktion 100 und 200 ist eine Schleife. Offensichtlich (?!) wird in dieser Schleife NCLUST nicht verändert, also tues in ein Indexregister rein, und damit wird es wirklich nicht verändert. Der Compiler ahnt nicht, daß die Subroutine LAUMGT über den COMMON die variable NCLUST doch ändert.

Literatur

[1] M. Metcalf. *"FORTRAN Optimisation"*, Academic Press

[2] D. Notz. *"TASSO Note 261"*

Kapitel 6

Programmprüfung und Fehlersuche

"Jedes Programm
enthält mindestens
einen Fehler"
(Volksweisheit)

6.1 Überblick

Jedes Programm enthält Fehler. Erfahrungswerte für ungeprüfte Programme sind 4-8 Fehler/100 Instruktionen. Entsprechend wichtig ist die Programmprüfung. Typischerweise erfordert die Prüfung etwa 50% des gesamten Software-Aufwandes. Bei der Korrektur von Fehlern können sich neue Fehler einschleichen. Deshalb ist es praktisch unmöglich, ein sehr großes Programm völlig fehlerfrei zu kriegen.

Hauptsatz der Programmprüfung:
Ziel einer Programmprüfung ist es, Fehler zu finden.

Dies heißt also:

1) Ein erfolgreicher Test ist einer, der einen Fehler findet (der ja da ist); ein erfolgloser Test ist einer, der keinen Fehler findet.

2) Es ist nicht Ziel einer Programmprüfung zu "beweisen", daß ein Programm richtig arbeitet.

Hieraus folgt auch, daß ein Programmierer (psychologische) Schwierigkeiten hat, sein eigenes Programm zu prüfen - er sollte dies tunlichst vermeiden.

Programmprüfung erfordert systematisches Vorgehen, nicht nach der Methode des 'big bang':
Programm auf die Maschine nehmen, und sehen wo der Rauch aufsteigt.

Die systematische Programmprüfung läuft in den folgenden Stufen ab, die im folgenden näher erklärt werden:

i) Test von Einzelmoduln bzw. Einzelprogrammen

ii) Test des Gesamt-Programmsystems

iii) Funktions-Akzeptanz-Installationstest

iv) Abbruchbedingung der Tests.

6.2 Test von Einzelmoduln

6.2.1 Programmlesung

Der erste Schritt ist die Programmlesung.
Am besten erfolgt sie mit 3-4 Teilnehmern, die das Programm vorher gelesen haben. Es gibt einen Moderator; gefundene Fehler werden nicht sofort verbessert. Als Notbehelf kann die Programmiererin das Programm ihrem Mann (Freund) vorlesen - es ist nicht nötig, daß derselbe etwas vom Programmieren versteht.

Die Programmlesung verwendet zur Fehlersuche die beiden folgenden Verfahren:

(i) Achten auf häufig gemachte Fehler anhand einer Liste.

(ii) Durchspielen einfacher Fälle (z.B. zur Prüfung von End-Abfragen u.a.m.)

Liste typische Fehler:

- Der Wert einer benutzten Variablen ist nicht definiert.
- Werte von Indizes überschreiten ihre Grenzen.
- Falscher Typ für den Index.
- Ein Zeiger hat keine zugeordnete Speicherstelle.
- Datenstrukturen und Argumentlisten im Hauptprogramm und in der Subroutine sind verschieden.
- Verzählen um $\pm$ 1 bei Indizes und Abfragen.
- Konstanten oder Variablen sind nicht initialisiert.
- 'Mixed mode'.
- Der Zahlenbereich wird über- oder unterschritten.
- Nenner kann =0 werden.
- Nichtbeachtung von Rundefehlern, z.B. bei der Darstellung von 0.1.
- Fehler in der Logik sind häufig, falsche Verwendung von GT, LT, GE, LE, EQ, NE, AND, OR, NOT.
- Mätzchen in der Integerarithmetik, z.B. ist u.U. $2(I/2) \neq I$.
- Abbruchbedingung von Schleifen falsch.
- FORMAT und READ/WRITE passen nicht zusammen.
- Schreibfehler - tückisch! z.B. I statt 1, O statt 0, MEAN statt MEEN.
- Überlauf von INTEGER-Größen.
- COMMON stimmen nicht überein.
- Eingabedaten nicht überprüft.
- Eine Variable wird unbeabsichtigt für mehrere Zwecke benutzt, z.B. eine, die schon im COMMON steht.
- Es folgen noch einige abschreckende Beispiele von Fehlern aus der Praxis:

```
      K = 6
      WRITE(K,218) ...
      DO 25 K=1,N
      .
      .
   25 CONTINUE
      WRITE(K,218) ...      (!)
```

Noch ein Beispiel:

```
      COMMON/QPHAS/Q(40,5)
      DO 303 J = 1,10
303   Q(IS + NT,J) = Q(31,J)
```

Noch ein Beispiel (Zerstörung einer Eins):

```
      CALL SUB(1)
      .
      .
      .
      SUBROUTINE SUB(IND)
      IND = 0
      .
      .
```

6.2.2 Entwurf von Testfällen

Hierbei geht es um das systematische Prüfen des Programmes mit entsprechend gewählten Testfällen. Man unterscheidet zwei Klassen von Prüfverfahren:

(i) 'black box' - Das Verfahren ist E/A-orientiert. Man bietet dem Programm bestimmte Eingabedaten an und prüft, ob das erwartete Ergebnis herauskommt. Regeln für das Auswählen von Testfällen sind:

   Jeder Testfall sollte eine möglichst große Klasse von möglichen Eingaben abdecken; keine Klasse vergessen werden.

   Testdaten sollten auf, knapp unter und knapp über den Bereichs- und Definitionsgrenzen von Variablen gewählt werden, da dies helfen kann, Schwachstellen im Programm zu finden. Um die Stabilität des Programms (z.B. gegen Hardware-Fehler) zu prüfen, biete man ihm auch grotesk falsche und illegale Eingabeparameter an, einzeln und in Kombination. Man denke sich ausgefallene Fälle aus, um das Programm zu testen.

(ii) 'White Box': Das Verfahren ist Logik-orientiert.

   Man entwirft Testfälle so, daß alle Kombinationen von allen logischen Verzweigungen im Programm mindestens einmal durchlaufen werden. Dies ist oft nicht konsequent durchfürbar, weil die Zahl der Möglichkeiten sehr groß werden kann.

(iii) Hier sind noch einige allgemeine Regeln für das Durchführen von Tests:

   Ehe Du einen Test ausführst, lege das zu erwartende Resultat fest.

   Prüfe das Ergebnis jedes Tests genau.

   Dokumentiere das Ergebnis von Tests, da sie eine große Investition an Arbeitszeit darstellen.

   Konzentriere Dich beim Testen auf solche Moduln, in denen bereits eine Menge von Fehler gefunden wurden, denn die Wahrscheinlichkeit, daß dort weitere Fehler sind, ist besonders groß. Dies zeigt Abb. 6.1. Diese erstaunliche Abbildung wird von G. J. Myers als eine Erfahrungstatsache berichtet. Intuitiv könnte es bedeuten, daß gewisse Programme eben sehr viel fehleranfälliger sind als andere, und viele gefundene Fehler ein Indiz für das Vorliegen eines solchen Problemfalles sind, in dem sich dann eben auch noch weitere Fehler finden. Dazu gehört auch die Bemerkung von G. J. Myers, daß in einem der S/370 Betriebssysteme von IBM 47% der von Benutzern gefundenen Fehler in 4% der Moduln des Systems zu finden waren.

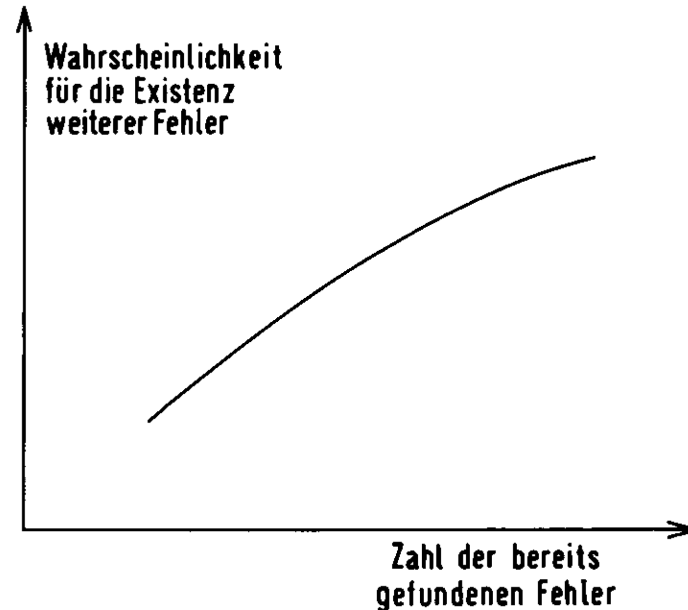


Abbildung 6.1: Fehlerkurve

6.3 Gesamt-Test

Hat man die Einzelprogramme nach Abschnitt (6.2) geprüft, so muß man sich als nächstes das System in seiner Gesamtheit vornehmen. Hierfür gibt es zwei Philosophien:

(i): Jeden Modul einzeln testen - dies erfordert Aufrufroutinen und DUMMY-Subroutinen für die CALL SR-Instruktionen. Dann alle Moduln im Programm vereinigen und laufen lassen: Big Bang.

(ii): Inkrementelle Tests:

a) von oben nach unten: Man beginnt mit der Hauptroutine, prüft sie, fügt dann die unmittelbar gerufenen Subroutinen an, prüft diese Kombination, und geht so nach unten auf einem kritischen Pfad auf die E/A-Routinen zu.

b) von unten nach oben: Man beginnt mit den innersten Subroutinen, prüft sie, fügt die rufende Subroutine an, prüft die Kombination u.s.fort.

In der Praxis ist oft die Methode (ii)b) die beste.

An den Test des Gesamtsystems schließt sich der FUNKTIONS-AKZEPTANZ-INSTALLATIONSTEST an. Hierfür sollte man die angegebene Literatur zu Rate ziehen.

DIE ABBRUCHBEDINGUNG:

Wann soll man die Suche nach weiteren Fehlern aufgeben?

schlecht:

- falls die zum Testen vorgegebene Zeit verstrichen ist
- falls alle Tests o.k. sind (beide Bedingungen können leicht durch nichts tun erfüllt werden)

gut:

- falls alle nach den Regeln von Abschn.6.2.2 durchgeführten Tests o.k. sind
- falls eine vorgegebene Zahl von Fehlern gefunden ist
- falls die Zahl der gefundenen Fehler/Zeitintervall sinkt.

6.4 Finden und Beseitigen von Fehlern

Gesetzt, ein Test oder die Praxis zeigen, daß ein Programm fehlerhaft arbeitet. Dann müssen die Fehler gefunden werden. Dies tut die Programmiererin am besten selbst. Regeln und Ratschläge hierzu:

- Vermeide Brute-force Methoden, wie Speicherausdrucke, wahlloses Verstreuen von Kontroll-Schreibbefehlen im Programm, komplizierte Debugger u.ä.
- Compiler mit sofortigem Zugang zu den Diagnostikausdrucken sind gut.
- Schlafe darüber.
- Erzähl Dein Problem jemand anderem.
- Gehe nicht blindlings vor.
- Wo ein Fehler ist, sind oft noch mehr (vgl. Abb. 6.1)
- Korrigiere keine Fehler rein.
- DENKE!
  a) durch Induktion (Abb. 6.2)
  b) durch Deduktion (Abb. 6.3)

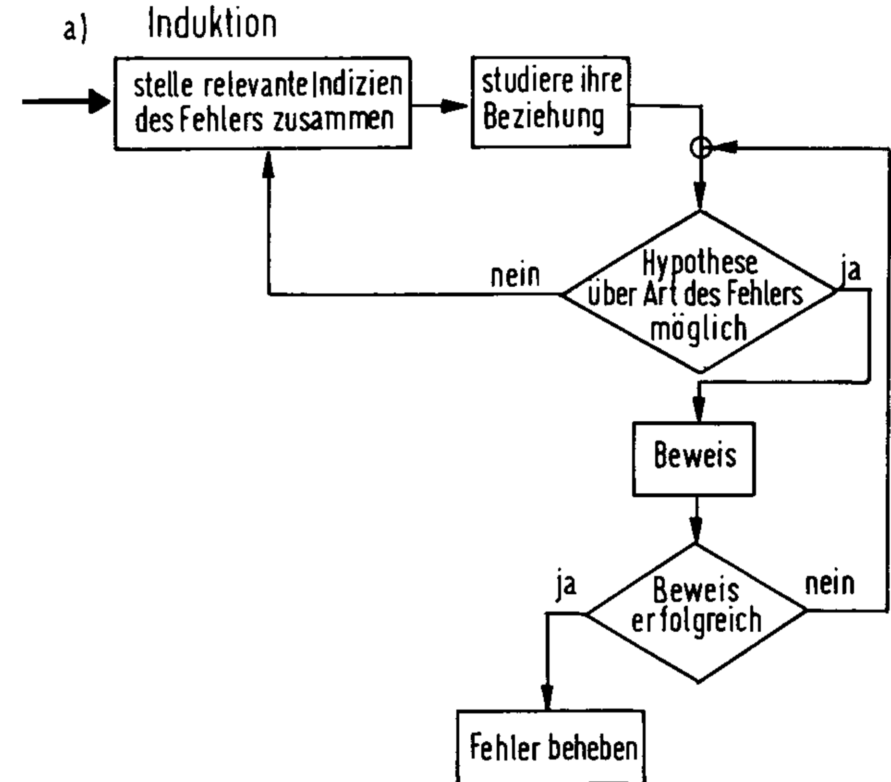


Abbildung 6.2: Induktion

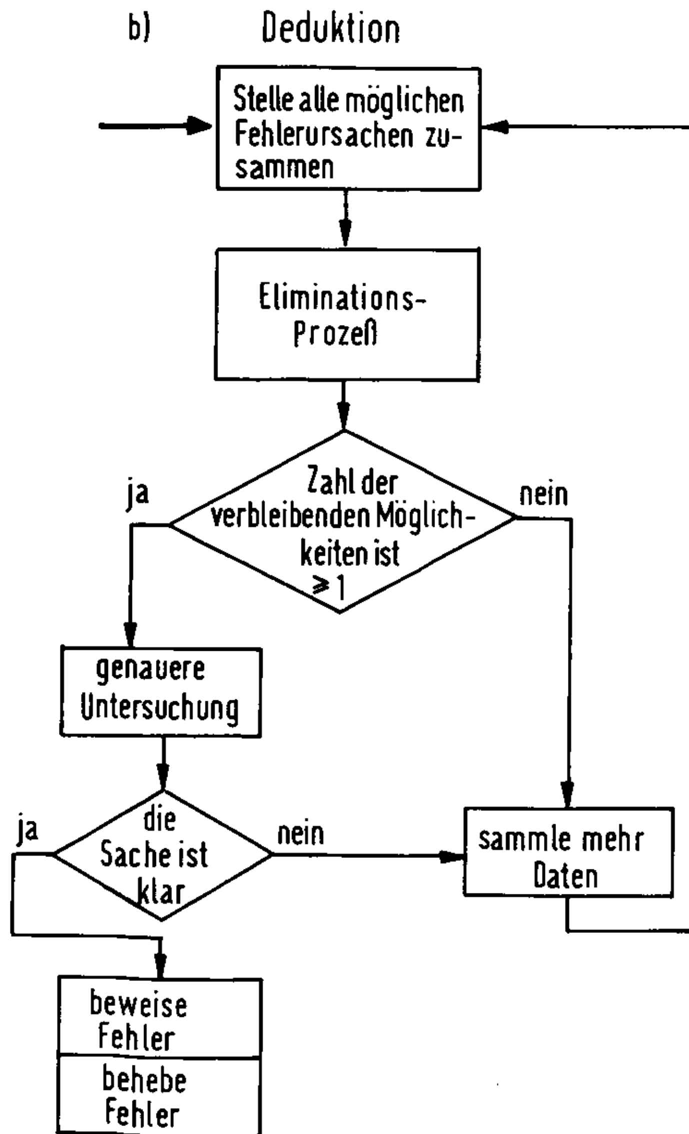


Abbildung 6.3: Deduktion

Literatur

[1] G. J. Myers. *The Art of Software Testing*, John Wiley and Sons

Kapitel 7

Algorithmen

"Palmström führt
ein Polizeipferd vor.
Dieses wackelt mehrmals
mit dem Ohr
und berechnet den
ertappten Tropf
logarhythmisch
und auf Spitz und Knopf".
(Christian Morgenstern)

7.1 Eine Studie zur numerischen Genauigkeit

Es sei

$$W = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots - \frac{1}{n}, n = \text{gerade}$$

Auf einem Rechner werde die Berechnung von W auf drei verschiedenen Weisen programmiert:

$$W1 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots - \frac{1}{n}$$

$$W2 = -\frac{1}{n} + \frac{1}{n-1} - \cdots - \frac{1}{2} + 1$$

$$W3 = (1 + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{n-1}) - (\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n})$$

Auf einer IBM 3081-Anlage erhält man bei Verwendung von einfach genauen Gleitkommazahlen die folgenden Abweichungen vom wahren Wert infolge von Rundungsfehlern:

	n=10 000	n=50 000
$\ln 2 - W$	$50 \cdot 10^{-6}$	$10 \cdot 10^{-6}$
$\lvert W - W1 \rvert$	$274 \cdot 10^{-6}$	$1391 \cdot 10^{-6}$
$\lvert W - W2 \rvert$	0	0
$\lvert W - W3 \rvert$	$33 \cdot 10^{-6}$	$30 \cdot 10^{-6}$
Rechenzeit	60 ms	140 ms

Rundefehler sind tückisch! Faustregel:
Bei Additionen mit den kleinen Zahlenwerten anfangen.

7.2 Rekursion

Eine rekursive Subroutine (Prozedur) ist ein Programm, welches sich selbst als Unterprogramm aufruft. Diese Technik gestattet oft die elegante Formulierung sehr mächtiger Algorithmen. Die scheinbar unschuldige Formulierung kann auf sehr komplexe Gebilde führen.

Beispiel für einen sehr einfachen rekursiven Aufruf: Berechnung von n!

```
C     AUFRUF
      J=IFACT (7)
      .
      .
      .
      FUNCTION IFACT(N)
      IF(N.GT.1) GO TO 110
      IFACT=1
      GO TO 130
C     REKURSIVER AUFRUF
  110 IFACT=N*IFACT(N-1)
  130 RETURN
      END
```

Dieses FORTRAN-Programm würde so nicht laufen, weil FORTRAN-Compiler keine rekursiven Aufrufe zulassen.

Ein Beispiel einer komplizierten (doppelt rekursiven) Funktion ist Ackermann's Funktion A(m,n). Sie ist folgendermaßen definiert:

if m=0 **then** A(m,n): = n + 1
else if n=0 **then** A(m,n): = A(m − 1,1)
else A(m,n): = A(m − 1,A(m,n −1))

also ist:

```
A(0,0)=1,A(0,1)=2, A(0,2)=3
A(1,0)=A(0,1)=2, A(2,0)=A(1,1)
A(1,1)=A(0,A(1,0))=A(0,2)=3, etc.
```

Diese komplizierte Sache kann man in FORTRAN programmieren, siehe A. Colin Day, loc.cit.
Allgemein kann man jede rekursive Routine in FORTRAN programmieren. Man kann die rekursive Routine natürlich nicht über eine CALL Subroutine ansprechen, sondern muß sie explizit im Programm einbauen ('open coded subroutine'). Dazu muß man sich seine eigene Verwaltung der Vor- und Rücksprungadressen sowie des Rettens der Subroutinen - Parameter bauen. Dies geschieht mit einem Kellerspeicher (Stack), d.h. durch Speichern der relevanten Werte in einem eindimensionalen Vektor. Eine Integervariable (Stack-Zeiger) enthält den Index der letzten in den Stack reingespeicherten Variablen.

Beispiel n! :

```
      DIMENSION NPA(100),NRET(100)
C     Zwei stacks fuer Parameter und Ruecksprungadresse
C     Stack-Zeiger
      DATA NST/0/
C     nun kommt Hauptprogramm mit Subroutinen-Aufruf
      NST=NST+1
C     7 !
      NPA(NST)=7
      NRET(NST)=1
C     Ruecksprungadresse, 1=Hauptprogramm
```

```
C       Sprung in die Subroutine
        GC TO 100
        Ergebnis, Fortsetzung Hauptprogramm
   10   J=IFACT
        .
        .
        .
C       nun folgt die rekursive n!-Routine
C       Abbruch-Test
  100   IF(NPA(NST).GT.1) GO TO 110
        IFACT=1
        GO TO 130
C       nun folgt rekursiver Aufruf
  110   NST=NST+1
        IF(NST.GT.100) GO TO 1000
        NPA(NST)= NPA(NST-1)-1
        NRET(NST)=2
        GO TO 100
  120   IFACT=NPA(NST) * IFACT
C       Ruecksprung,NRT=Ruecksprungadresse
  130   NRT=NRET(NST)
        NST=NST-1
        IF(NRT.EQ.1) GO TO   10
        IF(NST.LE.0) GO TO 1001
        GO TO 120
C       Fehlerbehandlung
 1000   WRITE(6,1002)
        STOP
 1001   WRITE(6,1003)
        STOP
 1002   FORMAT(1H ,'STACK OVERFLOW')
 1003   FORMAT(1H ,'STACK UNDERFLOW')
C       Ende der Subroutine
```

Das folgende Beispiel zeigt den Fall einer weniger trivialen Funktion, die als Funktion ihres Arguments so ungeheuer schnell anwächst, daß man das allgemeine Verhalten mathematisch nicht formulieren kann.

Die Funktion NRES der zwei Parameter NA und NN ist folgendermaßen definiert:

```
        DATA NA,NN/...,.../
        CALL IP(NA,NN)
C       Resultat ist NRES
        NRES=NA
        SUBROUTINE IP(NA,NN)
        IF(NN.EQ.0)GO TO 2
        DO 1 K=1,NA
        CALL IP((NN-1),NA)
    1   CONTINUE
        RETURN
    2   NA=NA+1
        RETURN
        END
```

Beispiel:

NA=2
NN=0, NRES=3
NN=1, NRES=4

NN=2, NRES=$2 \cdot 2^2$
NN=3, NRES=$2 \cdot 2^2 \cdot 2^{2 \cdot 2^2} = 2^{11}$
NN=4, $\log_2$NRES=2^{2048}

```
        FORTRAN-Programm zur Berechnung von NRES:
C       Stacks fuer: NA,NN, Zwischenwert, Ergebnis,
C                         Ruecksprungadresse
        DIMENSION NAV(10000),NNV(10000),NREPV(10000),
                  NEV(10000),NRTV(10000)
        DATA NDIM,NST/10000,0/
C       NST=Stackzeiger
C       Parameter
        DATA NA,NN/2,2/
        WRITE (6,1002) NA,NN
 1002   FORMAT(1H ,'NA=',I4,'NN=',I4)
C       Funktionsaufruf
        NST=NST+1
        NAV(NST)=NA
        NNV(NST)=NN
        NRTV(NST)=1
        GO TO 100
   10   WRITE(6,1000) NRES
 1000   FORMAT(1H ,'Resultat=',I 10)
        STOP
C       rekursive Funktion
  100   NNT=NNV(NST)
        IF(NNV(NST).NE.0) GO TO 200
        NEV(NST)=NAV(NST)+1
C       RETURN
        GO TO 300
  200   NI=NAV(NST)
C       setze Schleife
        NREPV(NST)=NAV(NST)
C       CALL IP(NN-1,NA)
  210   NST=NST+1
        IF(NST.LE.NDIM) GO TO 212
        WRITE(6,1001) NST
 1001   FORMAT(1H , 'stack fehler,NST=',I5)
        STOP
  212   NAV(NST)=NI
        NNV(NST)=NNT-1
        NRTV(NST)=2
        GO TO 100
  220   CONTINUE
        NEV(NST)=NRES
        NI=NEV(NST)
        NREPV(NST)=NREPV(NST)-1
        LOOP=NREPV(NST)
        IF(LOOP.EQ.0) GO TO 300
        GO TO 210
C       Ende Schleife
  300   NRT=NRTV(NST)
        NRES=NEV(NST)
        NST=NST-1
        IF(NRT.EQ.1) GO TO 10
        IF(NST.GT.0) GO TO 220
        WRITE(6,1001) NST
```

```
      STOP
      END
```

7.3 Divide et impera

Oft ist es vorteilhaft, ein Problem in ähnlich strukturierte Teilprobleme zu unterteilen, und diese dann (rekursiv) zu verknüpfen.

1. Beispiel: Wir behandeln die bekannte Aufgabe, aus einer Anzahl von N Zahlen die größte und die kleinste herauszusuchen. Die Standard-Lösung:

```
      DIMENSION ELM(1000)
      XMAX=ELM(1)
      KMAX=1
      DO 10 K=2,N
      IF(ELM(K).LT.XMAX) GO TO 10
      XMAX=ELM(K)
      KMAX=K
   10 CONTINUE
      XMIN=XMAX
      DO 20 K=1,N
      IF(K.EQ.KMAX) GO TO 20
      IF(ELM(K).GT.XMIN) GO TO 20
      KMIN=K
      XMIN=ELM(K)
   20 CONTINUE
```

benötigt 2N−2 Vergleiche (Gleitkommasubtraktionen).

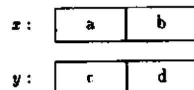
Dem gegenüber steht die Methode des 'divide et impera':
Man halbiert den Satz von N Zahlen, bestimmt von jeder Hälfte das größte und kleinste Element, und fährt rekursiv fort: jede der Hälften wird wieder halbiert usw. Die Zahl der Vergleiche in diesem Fall bestimmt sich folgendermassen: Angenommen, N sei von der Form $N=2^{n+1}$. Nach n maligem Halbieren hat man 2^n Paare von je zwei Zahlen.Man sortiert diese Paare(z.B. kleinstes Element vorneweg). Dies erfordert 2^n Vergleiche. Man bildet jetzt aus den 2^n größten Zahlen $2^n/2$ Paare, sortiert sie wieder mit $2^n/2$ Schritten. Dasselbe macht man mit den kleinsten Zahlen: $2^n/2$ Schritte. Insgesamt hat man also $2^n/2+2^n/2=2^n$ Schritte. Man wiederholt die letzten zwei Schritte mit den $2^n/2$ Paaren und benötigt dazu $2^n/2$ Schritte. Insgesamt benötigt man also

$$2^n + 2^n + 2^n/2 + 2^n/4 + 2^n/8 + \cdots = 3 \cdot 2^n = 1.5N - 2$$

Schritte.

Dazu muß ein gewisser Verwaltungsaufwand zum Halbieren etc. gezählt werden, so daß man insgesamt $1.5N+\epsilon$ Schritte braucht, zu vergleichen mit den 2N−2 Schritten im Normalverfahren. Wie bei allen diesen Algorithmen lohnt sich das vornehme Verfahren nur für große Werte von N. Die Abb. 7.1 zeigt nochmals die Idee.

2. Beispiel: Multiplikation zweier Zahlen zu je n bits. Die Zahl der hierzu nötigen Operationen ist im einfachsten Fall $O(n^2)$.
Ein (für große n) besseres Verfahren ist das folgende:
Teile die beiden Faktoren x und y in je zwei Hälften:



$$x \cdot y = (a \cdot 2^{n/2} + b) \cdot (c \cdot 2^{n/2} + d)$$

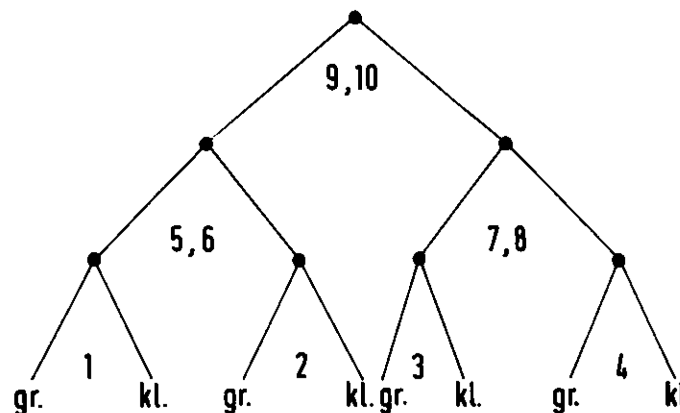


Abbildung 7.1: Divide et impera

$$= a \cdot c \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

$$xy = ac \cdot 2^n + [(a+b)(c+d) - ac - bd] \cdot 2^{n/2} + bd \qquad (7.1)$$

Falls die Multiplikation mit 2^n bzw. $2^{n/2}$ vernachlässigbar wenig Zeit kostet (es sind einfache Verschiebeoperationen) und falls der Aufwand für eine Addition klein ist verglichen mit dem für eine Multiplikation, so ist das Verfahren Gl.(7.1) günstig, da man lediglich die drei Produkte ac, (a+b)(c+d), und bd zu berechnen hat. Die Kosten dieser Operation sind im wesentlichen die Kosten für die Multiplikation dreier Zahlen der Länge n/2:

$$K(n) = 3 \cdot \frac{n}{2} \cdot \frac{n}{2} + kn \qquad (7.2)$$

Hierbei wurde noch ein Korrekturterm kn für die Additionen hinzugefügt.

Dieses Verfahren kann man nun rekursiv fortzusetzen, d.h. man kann zur Berechnung der Produkte ac usw. die Faktoren a, c, $\cdots$ wieder in zwei Teile teilen, und nach dem Algorithmus Gl.(7.1) verfahren. Für die Kosten der $\frac{n}{2} \times \frac{n}{2}$ Multiplikation gilt dann eine zu Gl.(7.2) entsprechende Gleichung. Durch Einsetzen und immer weiteres Einsetzen erhält man eine iterative Verschachtelung. Durch Induktion erhält man als Lösung von Gl.(7.2)

$$K(n) = 3k \cdot n^{lg_2 3} \sim 2kn \qquad (7.3)$$

Beweis: Für n=1 gilt

$$K(1) = k, \quad o.k.$$

Nimm an, $K(m) = 3km^{lg_2 3} - 2km$ sei bewiesen, dann gilt

$$\begin{aligned} K(2m) &= 3k(2m)^{lg_2 3} - 4km \\ &= 3k \cdot 3 \cdot m^{lg_2 3} - 4km \\ &= 3K(m) + 2mk \end{aligned}$$

Setze am Schluß 2m=n und man erhält Gl.(7.2).
Also sind die Kosten

$$K(n) \sim 3k \cdot n^{1.69}$$

zu vergleichen mit $K \sim n^2$ des Primitivverfahrens.
Dieses Verfahren lohnt sich allerdings nur für große Werte von n, z.B. bei Primzahlzerlegungen, die neuerdings für Verschlüsselungsprobleme in Mode gekommen sind.

Für kleine Werte von n kann bereits die Zeit zum Heranholen und Abspeichern der Operanden eine Rolle spielen; da Speicher billig geworden ist, lohnt sich eine Überlegung, ob man für kleine Werte von n nicht ganz auf eine Multiplikation verzichtet und statt dessen die Produkte aus einer Tabelle abliest.

Beispiel 3: Strassens Algorithmus zur Matrix-Multiplikation:

$$A \cdot B = C$$

Die Dimension der Matrizen sei n.
Zerlegung in Untermatrizen der halben Dimension:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} \qquad c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21} \qquad c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Im Normalverfahren sind die Kosten der hier beschriebenen Matrixmultiplikation

$$K(n) \simeq 8K(n/2) + 4n^2/4$$

(8 Multiplikationen, 4 Additionen)
Trick, um mit 7 Multiplikationen und 15 Additionen auszukommen:

$$\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}
\end{aligned}$$

$$\begin{aligned}
c_{11} &= m_1 + m_2 - m_4 + m_6 \\
c_{12} &= m_4 + m_5 \\
c_{21} &= m_6 + m_7 \\
c_{22} &= m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

Setzt man das Verfahren durch immer weitere Halbierung der Matrizen fort, so erhält man für die Kosten

$$K(n) \approx O(n^{\lg 7}) = O(n^{2.81})$$

zu vergleichen mit $O(n^3)$ beim Normalverfahren. Wegen des Beweises s. Hopcroft und Ullman, loc.cit.

Oft kann man bei der Ketten-Multiplikation mehrere Matrizen durch richtiges Setzen der Klammern erheblich Rechenzeit sparen, wie das folgende Beispiel zeigt:

$$M = M_1(10 \cdot 20) \cdot M_2(20 \cdot 50) \cdot M_3(50 \cdot 1) \cdot M_4(1 \cdot 100)$$

In der Klammerung

$$M = M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$$

ist die Zahl der benötigten Multiplikationen:

$$= 50 \cdot 1 \cdot 100 + 20 \cdot 50 \cdot 100 + 10 \cdot 20 \cdot 100 = 125000$$

Berechnet man dagegen M mit der Klammerung

$$M = (M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$$

so ist die Zahl der Multiplikationen

$$20 \cdot 50 \cdot 1 + 10 \cdot 20 + 10 \cdot 1 \cdot 100 = 2200 \quad (!)$$

7.4 Sortieren

Sortieren wird in der DV an vielen Stellen benutzt. Es lohnt sich deshalb, sich für das Sortieren langer Listen effizienten Algorithmen zu überlegen.

(i) Einfaches Verfahren

```
      DIMENSION NUM(10000)
C     N Zahlen NUM(K) sind zu sortieren
      DATA N/100/
      N1 = N - 1
      DO 1 K = 1,N1
      MIN=K
      J  =K+1
      DO 2 L=J,N
      IF(NUM(L).LT.NUM(K)) MIN=L
    2 CONTINUE
      NX=NUM(K)
      NUM(K)=NUM(MIN)
    1 NUM(MIN)=NX
```

Dieses erfordert Kosten von der Größenordnung

$$K \approx O(N^2/2)$$

(Um das kleinste Element zu finden, braucht man N-1 Vergleiche, für das zweitkleinste N-2, und (N-1)mal so weiter, insgesamt rund $N^2/2$ Vergleiche.

(ii) Bubble Sort: Vergleiche je 2 aufeinanderfolgende Elemente der Liste, falls nicht sortiert, werden sie vertauscht:

```
      N1=N-1
      DO 1 K=1, N1
      L=N-K
      NFLAG=0
      DO 2 M=1,L
      IF(NUM(M+1).GE.NUM(M)) GO TO 2
C     Vertausche die zwei Elemente
      NUMX=NUM(M)
      NUM(M)=NUM(M+1)
      NUM(M+1)=NUMX
      NFLAG=1
    2 CONTINUE
      IF(NFLAG.EQ.0) GO TO 3
    1 CONTINUE
    3 CONTINUE
C     Ende
```

Die Kosten sind i.a. $O(N^2/2)$; sie sind aber kleiner, wenn die Liste schon teilweise sortiert ist.

(iii) Eimersortieren: Dies geht, falls die zu sortierenden Zahlen ganze Zahlen sind, und man mindestens so viele Speicherplätze hat wie die größte vorkommende Zahl. Eimersortieren ist sehr effektiv.

Die zu sortierende Liste bestehe aus N ganzen Zahlen NUM(N), die im Bereich 1 bis M liegen. Dann:

```
      DIMENSION NUM(N), NEIMER (M)
C     setze alle Elmente der Eimermatrix = 0
      CALL VZERO (NEIMER)
      DO 1 K=1,N
```

```
C        Fuelle in Eimer
       1 NEIMER (NUM(K))=NEIMER(NUM(K))+1
         NZ=0
         DO 2 K=1,M
         L=NEIMER(K)
       3 IF(L.EQ.0)GO TO 2
         NZ=NZ+1
         L=L-1
         NUM(NZ)=K
         GO TO 3
       2 CONTINUE
```

Die Kosten sind O(M+N).

Ein damit verwandtes Verfahren, welches auch klappt, wenn man nicht so viele Speicherplätze hat, oder wenn die Zahlen reell sind, benutzt das Verfahren der "hash-Adressierung" (s.Kap.8.2).

(iv) Sortieren durch Vereinigen ('Merge') von Teillisten: Für lange Listen ist dies ein sehr effizientes Verfahren. Das Schema des Verfahrens ist das folgende:

(1) Teile die Liste der Länge N in P Teillisten der Länge N/P auf.

(2) Sortiere jede der P Teillisten, der Aufwand je Teilliste ist $(N/P)^2/2$, insgesamt also

$$\frac{1}{2}(N/P)^2 \cdot P = N^2/(2P)$$

(3) Vereinige die P sortierten Teillisten, dazu:

(3.1) Vergleiche jeweils die kleinsten Elemente der P Teillisten.

(3.2) Bringe das kleinste Element in die endgültige Liste, entferne es aus der entsprechenden Teilliste, gehe nach 3.1). Die Kosten des Schrittes 3) sind N·P, da man für jedes der N Elemente alle P Teillisten durchgehen muß.
Die gesamten Kosten sind also

$$K = N^2/(2P) + NP$$

Dies ist i.a. kleiner als $N^2/2$.

(v) Iterative Anwendung von (iv): Man unterteilt die P Teillisten wieder je P-fach und fährt damit fort, bis Listen der Länge P übrigbleiben und man aufhören muß. Die Zahl der Teilschritte n bestimmt sich aus der Gleichung

$$P^n = N, n = \ln N / \ln P$$

Die Endlisten werden geordnet, das kostet pro Liste $P^2/2$, insgesamt gibt es N/P solche Endlisten zu P Elementen, also sind die Kosten

$$(P^2/2) \cdot (N/P) = \frac{1}{2} \cdot NP$$

Nun werden je P Listen vereinigt, die Kosten dafür sind N· P(s.o.). Insgesamt sind n solcher Vereinigungsschritte nötig. Die Gesamtkosten sind also

$$K = \frac{1}{2}NP + n \cdot NP = NP\left(\frac{\ln N}{\ln P} + \frac{1}{2}\right) \approx NP \cdot \left(\frac{\ln N}{\ln P}\right)$$

Die Kostenfunktion hat ein Optimum für P=e, und damit wird

$$K = O(N \ln N)$$

Dies ist so ziemlich das beste, das man erreichen kann.

Ein Verfahren, welches nach diesen Ideen arbeitet, allerdings eine Aufteilung in je zwei Listen von nicht notwendig gleicher Länge benutzt, ist Quicksort (Hoare u. Scowen, Com.ACM 4 (1961)321):


```
         SUBROUTINE QUICKSORT(A,M,N)
         DIMENSION A(10 000)
C        A=Vektor, der sortiert werden soll. Der Index I
C        soll von M bei N gehen,M < N <= 10 000 in
C        diesem Beispiel
         IF(M.GE.N)STOP
         CALL PART(A,M,N,I)
         CALL QUICKSORT(A,M,I)
C        rekursive Aufrufe
         CALL QUICKSORT(A, I+1, N)
         END
         SUBROUTINE PART(A,M,N,I)
C        waehlt ein Element mit Index I in A zwischen den
C        Indizes M und N zufaellig aus. Ordnet Listen so, dass alle
C        A(K) <= A(I) falls K <= I und A(K) > A(I) falls K > I.
```

Die Kosten von Quicksort liegen zwischen $N^2/2$ und $N \lg N$.

Vergleich der Rechenzeit verschiedener Sortierverfahren am Beispiel von 500 Namen:

Bubble sort	1.26	s
Standard-sortieren	0.7	s
Sortieren durch Vereinigen	0.1	s
Quicksort FORTRAN	0.06	s
Quicksort assembler	0.02	s
Hash Adressierung	0.08	s

7.5 Das stabile Eheproblem

Das Problem betrifft die stabile Verpaarung von n Männern und n Frauen. Jeder Mann ordnet die Frauen in einer Prioritätsliste von 1 bis n, jede Frau ordnet die n Männer ebenso in einer Liste.

Eine Ehe ist stabil, falls für jede von 2 Paarungen (Index 1,2) die beiden folgenden Bedingungen gelten (M=Mann, F=Frau):

(i) (M_1 schätzt F_1 höher als F_2).oder.(F_2 schätzt M_2 höher als M_1).

(ii) (M_2 schätzt F_2 höher als F_1).oder.(F_1 schätzt M_1 höher als M_2).
Es existiert eine stabile Verpaarung. Beweis?

Literatur

[1] Brand. *Algorithmen zur praktischen Mathematik*, R. Oldenbourg

[2] Aho, Hopcroft und Ullmann. *The Design and Analysis of Computing Algorithms*, Addison-Wesley

[3] V. Blobel. *Methoden der Datenanalyse in der Hochenergiephysik*, DESY F14-81/01

[4] N. Wirth. *Algorithmen und Datenstrukturen*, CERN Computer School, Zinal 1984

[5] A. Colin Day. *FORTRAN Techniques*, Cambridge University Press

[6] M. Minsky. *Finite and Infinite Machines*, Prentice Hall

[7] J. H. Wilkinson. *Rundungsfehler*, Springer Verlag

[8] G. Engeln-Müllges/F. Reutter. *Formelsammlung zur numerischen Mathematik mit Standard FORTRAN Programmen*, BI Wiss. Verlag

Kapitel 8

Datenstrukturen

"We do things by routine here. You have followed the routine and found out what you wanted to know. It is the best way. It is the only way. It is very regular, and very slow, but it is very certain" "Yes, certain death. It has been, to the most of our tribe..."
Mark Twain, The Facts in the Great Beef Contract.

8.1 Listen

Die einfachste Datenstruktur ist eine Liste. Sie ist die sequentielle Aneinanderreihung von Informationselementen (Eintragungen).

Grundoperationen an Dateien sind:

1) Zufügen einer Eintragung

2) Löschen einer Eintragung

3) Modifizieren einer Eintragung

4) Finden von Zusatzinformation nach einem Stichwort.

Wir behandeln nun Listen in der Reihenfolge zunehmender Komplexität:

(i) Keller (Stack)
Er besteht aus einer konsekutiven Folge von Speicherplätzen (Vektor), in welche Information nacheinander abgelegt werden kann. Ein Index (Stackzeiger) gibt die Adresse der letzten gefüllten Position an. Neue Information kommt auf den anschließenden Platz (PUSH), sie wird durch Versetzen des Zeigers entfernt (POP):
last in, first out (LIFO), s. Abb. 8.1.

(ii) Schlange (Queue)
Es gibt hier zwei Zeiger (H=hinten, V=vorne). Information wird bei V zugeführt, bei H entfernt: first in - first out (FIFO). Die zwei Zeiger laufen hintereinander her. Der Schlangenspeicher kann zyklisch sein, d.h. auf die letzte Adresse des (physischen) Speicher folgt wieder die erste. Abb. 8.2.

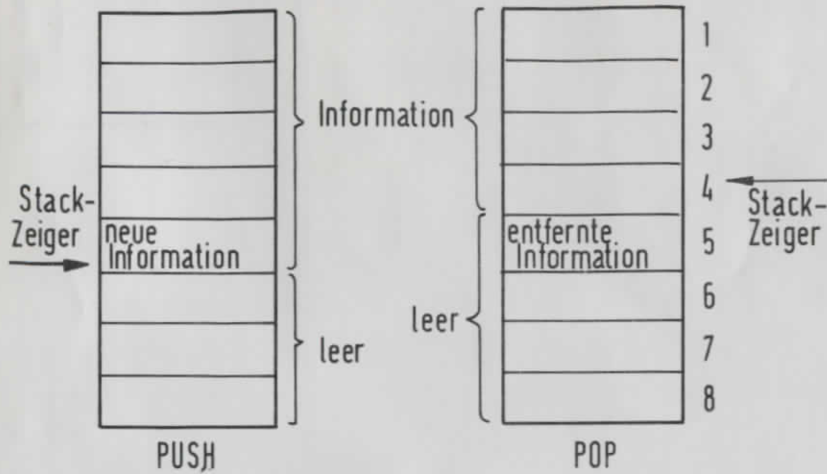


Abbildung 8.1: Keller(stack)

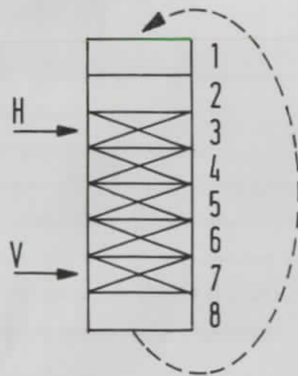


Abbildung 8.2: Schlange(Queue)

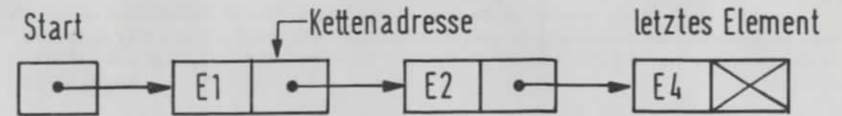


Abbildung 8.3: Verkette Liste

(iii) Verkettete Listen

Oft können sukzessive Elemente in einer Liste nicht in sukzessiven Speicherplätzen eingetragen werden. Dann muß jedes Element die Adresse der nachfolgenden Eintragung enthalten. Man nennt diese Adressen Zeiger:
Sie zeigen auf die Adresse des nächsten nachfolgenden Elementes. (Abb. 8.3).

Beispiel für die Anordnung einer verkettenden Liste im Speicher:

Speicherplatz (Adresse)	Element	Zeiger
1	-	2
2	El.1	4
3	El.4	0
4	El.2	5
5	El.3	3

Auf Platz 1 steht die Adresse (Zeiger) des ersten Elements, das Ende der Liste wird durch den Zeiger 0 bezeichnet.

Zeiger sind vor allem nützlich, falls jede Eintragung mehrere Speicherplätze (nicht notwendig immer gleichviele) benötigt. Dann lassen sich Operationen wie Einfügen oder Entfernen von Eintragungen lediglich durch Verändern der Zeiger bewerkstelligen.

Listen und Zeiger in FORTRAN:

```
      DIMENSION LINK(500),ELEM(500)
C     LINK=Zeiger,ELEM=Eintragungen
      ISTART=5
C     Startadresse
```

Die Liste 1.0, 1.2, 2.1, 6.3, 3.5 kann folgendermaßen gespeichert werden:

K=	ELEM(K)	LINK(K)
1	1.2	3
2	3.5	0
3	2.1	4
4	6.3	2
5	1.0	1

Einschieben eines auf Platz 6 im Speicher einzutragenden Elements XY zwischen die Elemente No.2 und 3 (d.h. zwischen 1.2 und 2.1)

```
      LINK(6)=LINK(1)
      LINK(1)=6
      ELEM(6)=XY
```

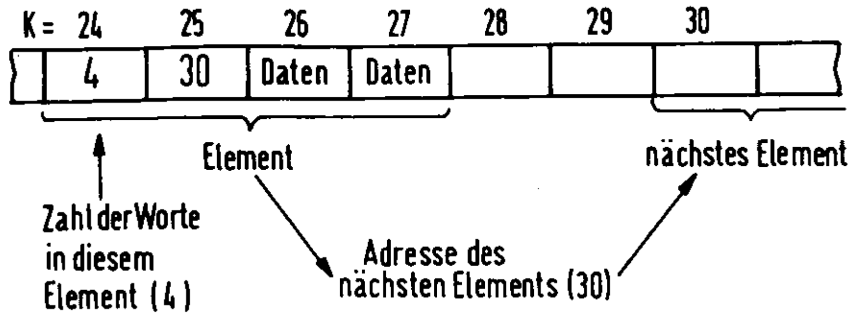


Abbildung 8.4: Noch eine verkettete Liste

Entfernen des Elements hinter Element 2 (K=1):

```
LINK(1)=LINK(LINK(1))
```

Eine andere Art der Verkettung seigt Abb. 8.4.

(iv) Doppelt verkettete Listen:
Doppelt verkettete Listen enthalten in jedem Element swei Zeiger: Der eine Zeiger enthält die Adresse des nachfolgenden Elements, der andere Zeiger die Adresse des vorhergehenden Elements. Dies gestattet es, die Liste vorwärts und rückwärts su lesen, und ein Element vor einem gegebenen Element einsutragen oder su löschen. Dafür ist die Verwaltung der Zeiger etwas komplizierter.

8.2 Durchsuchen von Listen

Das Durchsuchen von Listen sur Auffindung eines Elements ist eine der Standardoperationen. Hierfür sind eine Reihe von Methoden entwickelt worden.

8.2.1) Unterbringung der Daten in einem assoziativen Speicher (CAM = content addressable memory). Hier kann man eine Speicherselle direkt durch ihren Inhalt adressieren. Dieses Verfahren ist sehr schnell, doch sind solche Speicher teuer.

8.2.2) Sequenzielles Suchen: Man vergleicht alle Elemente des Speichers der Reihe nach, bis das gewünsche Wort gefunden ist. Falls N=Zahl der Eintragungen in der Datei, und M=Zahl der Suchoperationen, ist der Zeitaufwand im Mittel proportional $N \cdot M/2$.

8.2.3) Binäres Suchen: Dazu wird zuerst nach dem Inhalt ("Stichworten") sortiert, und swar nach einem Index, der eine grösser/kleiner Beziehung zu definieren gestattet ("nach der Grösse sortieren"). Das Ergebnis seien N Stichworte, ihr Index I laufe von 1 bis N. Das Suchen geht nach folgendem Algorithmus:

```
  (i)   L=1,
        M=N+1

 (ii)   falls M=L : Element nicht in Liste, stop

(iii)   I=INT((M+L)/2)
```



```
 (iv)   vergleiche Eintragung zum Index I mit dem
        gesuchten Stichwort:
        falls Eintragung (I) > Stichwort : M=I
        falls Eintragung (I) < Stichwort : L=I
        falls Eintragung (I) = Stichwort : gefunden, stop

  (v)   gehe nach (ii).
```

Der Aufwand für dieses Verfahren ist proportional su

$$3N \ln N + 2M(\operatorname{ld} N + 1)$$

wobei der erste Summand das Sortieren, der sweite das binäre Suchen bezeichnet.

8.2.4) Hash - Adressierung: Man benötigt mehr Speicherplätze als su sortierende Elemente. Man berechnet die Adresse des Stichwortes als Funktion des Stichwortes. Diese Funktion F heißt Hash-Funktion F(Stichwort). Sie sollte die Elemente möglichst gleichmässig über die Adressen verteilen. Die Funktion F muß eindeutig und monoton sein.

Eine bekannte Anwendung der Hash-Adressierung ist das Anlegen von Variablen-Listen durch Compiler. Dabei muß man die Variablennamen in Adressen umrechnen, indem man z.B. die Buchstaben in Zahlen 1-26 verwandelt und schreibt:

$$F = \text{erster Buchstabe} * 26 + \text{sweiter Buchstabe}$$

(Hierbei müssen sich natürlich alle Variablen in ihren ersten swei Buchstaben unterscheiden).

Algorithmus sum Suchen:

```
  (i)   Berechne Adresse des Stichwortes : AD=F(Stichwort)

 (ii)   Falls Stichwort = Inhalt (AD) : gefunden, stop.

(iii)   Falls Inhalt (AD) = leer : Stichwort nicht in
                                   der Liste, stop

 (iv)   AD:=AD+1, gehe nach (ii)
```

Algorithmus sum Einordnen

```
  (i)   Berechne Adresse des Stichworts AD=F(Stichwort)

 (ii)   Falls Inhalt (AD) = leer : Stichwort eintragen, stop

(iii)   Falls Inhalt (AD) = nicht leer : AD:=AD+1, gehe nach (ii)
```

8.3 Baumstrukturen und Graphen

Hierarchisch strukturierte Daten lassen sich vorteilhaft in Baumstrukturen unterbringen. Zur Festlegung der Baumstruktur können wieder Zeiger dienen.

Ein Baum ist ein gerichteter Graph mit den folgenden Eigenschaften:

1) Es gibt genau einen Knoten mit null einlaufenden Linien: Wursel, Beginn des Baumes

2) Jeder Knoten (mit Ausnahme der Wurzel) hat genau eine einlaufende Linie

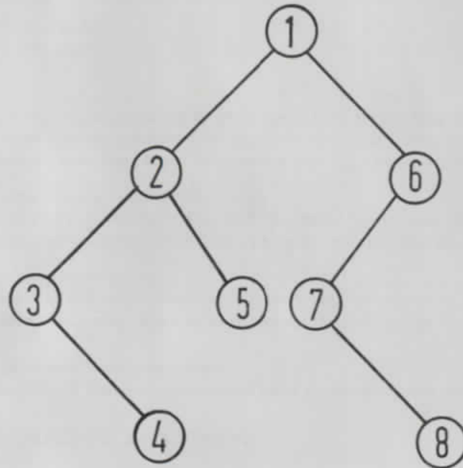


Abbildung 8.5: Baumstruktur

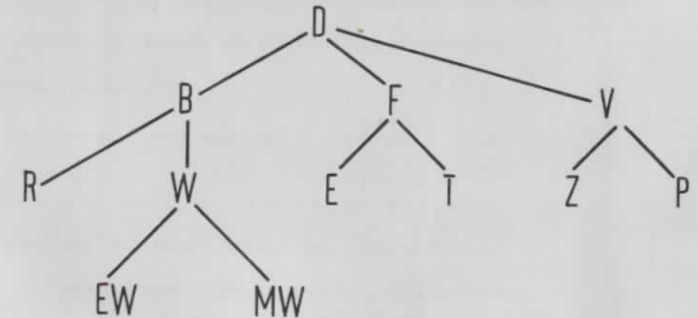
3) Von der Wurzel zu jedem Knoten gibt es genau einen Pfad, d.h. es darf keine Schleifen geben.

Ein Spezialfall ist der binäre Baum: jeder Knoten hat ≤ 2 auslaufende Linien, s. Abb. 8.5.
Repräsentation der in Abb. 8.5 gezeigten Struktur im Speicher:

Knoten Nr.	L	R
1	2	6
2	3	5
3	0	4
4	0	0
5	0	0
6	7	0
7	0	8
8	0	0

L(KN) und R(KN) bezeichnen die Knotennummer der links- (rechts-)seitigen Verbindung des Knotens KN.
Rekursiver Algorithmus zum Durchnumerieren (Durchsuchen) aller Knoten eines binären Baumes:

```
C       Hauptprogramm
        LAUFNR=1
        CALL SCAN(NWURZEL)
C       NWURZEL = Knotennr. der Wurzel
        STOP
        END
        SUBROUTINE SCAN(KN)
        IF(L(KN).EQ.0) GO TO 1
C       existiert ein linker Zweig? nein:GO TO 1
        CALL SCAN (L(KN))
      1 NUMMER(KN)=LAUFNR
C       nummeriere Knoten Nr.KN
```



$$D\,\lambda^3\,B\lambda^2\,R,W\lambda^2\,EW,MW,F\lambda^2\,E,T,V\,\lambda^2 Z,P$$

Abbildung 8.6: Polnische Notation

```
        LAUFNR=LAUFNR+1
        IF(R(KN).EQ.0)RETURN
C       RETURN falls kein rechter Knoten
        CALL SCAN(R(KN))
        END
```

Die POLNISCHE NOTATION (nach dem polnischen Mathematiker Lukasiewicz) ist eine andere Repräsentationsmöglichkeit von Baumstrukturen. Sie benutzt ein n-faches Baum-Verzweigungssymbol λ^n, Beispiel s. Abb. 8.6. Die polnische Notation wird mit Vorteil bei der Auswertung arithmetischer Ausdrücke angewandt.
Beispiel:

$$X = (P1 + P2 * P3) * (P4 + P5/P6) + P7/P8$$

wird verwandelt in die Matrix Abb. 8.7: Der Ausdruck schreibt sich in polnischer Notation folgendermaßen, wobei +, *, / jeweils für λ^2 steht: (es folgt ein numerisches Beispiel für die Ausrechnung)

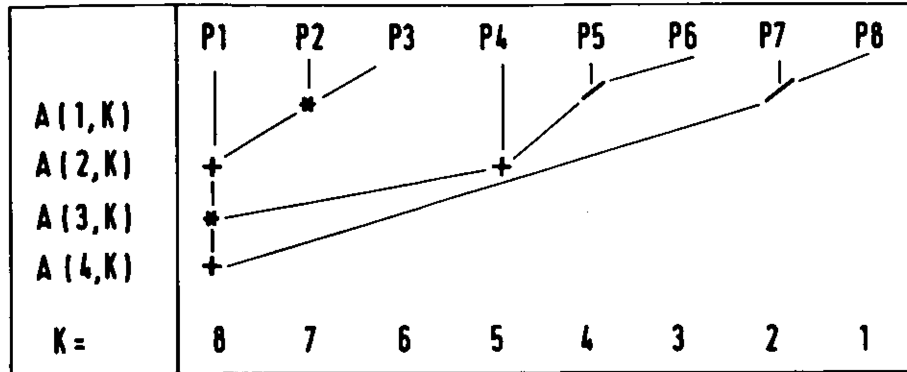


Abbildung 8.7: Auswertung eines arithmetischen Ausdrucks in polnischer Notation

```
+ * + P1  * P2 P3  +  P4 / P5  P6  /  P7  P8

+ * +  6  *  2  1  +   3 /  5  10  /   1   3

+ * +  6  *  2  1  +   3 /  5  10     0.33

+ * +  6  *  2  1  +   3   0.5 0.33

+ * +  6  *  2  1     3.5 0.33

+ * +  6     2  3.5   0.33

+ *    8    3.5 0.33

+     28    0.33

     28.33
```

Algorithmen zur Verarbeitung von Ausdrücken in polnischer Notation finden sich z.B. bei Rice and Rice, loc. cit.

Wesentlich komplexer als Baumstrukturen sind GRAPHEN. Ein Graph ist eine Menge von Knoten und (Verbindungs-)Linien. Falls die Linien geordnete Paare von Knoten verbinden, heißt der Graph GERICHTET.

Im Gegensatz zu Baumstrukturen kann ein Graph Maschen enthalten. Eine Beschreibung der Struktur kann durch Matrizen oder Zeigerlisten erfolgen, siehe z.B. Aho, et. al. loc. cit.

8.4 Datenbanken und Informationssysteme

DATENBANKEN sind organisierte Datensammlungen, in welchen die Trennung zwischen Daten und ihrer Verwendung explizit ist, und wo der Datenintegrität genügend Beachtung geschenkt wird.

Die Abb. 8.8 zeigt das Schema einer Datenbank. Der Verkehr der Benutzer mit den Daten läuft über ein zentrales Datenverwaltungsprogramm. Jeder Benutzer erhält nur Zugriff zu dem Teil der Daten, die er unbedingt braucht. Mehrere Benutzer können gleichzeitig auf die Daten zugreifen. Alle Funktionen zur Datendefinition, Datenorganisation und Datenintegrität geschehen nach einem einheitlichen Konzept.

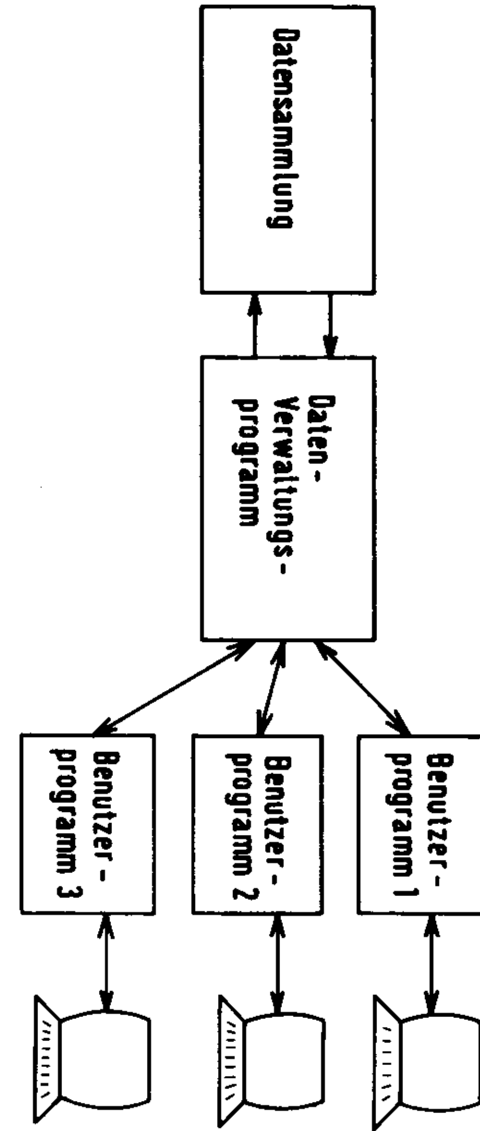


Abbildung 8.8: Datenbank

Für diese Vorteile muß man einen erheblichen Aufwand für das Datenverwaltungsprogramm und eine gewisse Bürokratisierung in Kauf nehmen.

Wegen Einzelheiten sei auf die gute Einführung bei Bauknecht und Zehnder loc.cit. verwiesen.

INFORMATIONSSYSTEME enthalten neben den Fakten einer Datenbank auch eine Sammlung von Methoden und Schlußregeln, sie verkörpern in einer gewissen Weise "Wissen" und "Erfahrung". Der Zugriff zu diesen Systemen erfolgt über eine Dialogsprache. Die Systeme sind in der Lage, Schlußfolgerungen und Datenverknüpfungen durchzuführen. Sie heißen deshalb auch wissenserarbeitende Systeme oder Expertensysteme.

Beispiele: Systeme zur medizinischen Diagnostik auf Spezialgebieten, Computer-Diagnostik, Konfiguration von Rechnersystemen (DEC), CAD (computer aided design, mechanisch und elektronisch), Ölsuche, Brückenbau, Entwurf von VLSI. Dies ist ein Gebiet mit großer Zukunft. Inzwischen gibt es bereits Programme zur Erstellung von solchen Expertensystemen.

8.5 Datensicherung

(i) Sicherung gegen Verlust und Verfälschung:
Die Sicherung muß jedes Glied der Datenverarbeitungs-, Übertragungs- und Speicherkette erfassen: CPU, Speicher, Übertragungskanäle, Sicherung von Bändern und Platten gegen Fehlfunktionen von Hard- und Software, Sicherung gegen Brand, Diebstahl, Dummheit, Sabotage, Ausfall von Klimaanlagen, Sicherung vor Benutzern und anderen Menschen.
Maßnahmen: Hard- und Software fail-soft Maßnahmen, Redundanz, gestaffeltes System von Sicherheitskopien.

Prüfziffern - Beispiel: IBM - Modulo 10 - Verfahren:

```
     Grundzahl                Pruefziffer

 7  9  9  1  2  8  1          2

14  9 18  1  4  8  2          Berechnung der Pruefziffer:
                              jede 2. Ziffer mal 2

1+4+9+1+8+1+4+8+2=38          Quersumme

40-38=2                       Ergaenzung zum naechsten
                              Zehner = Pruefziffer.
```

(ii) Sicherung gegen verbotenen Zugriff:

Maßnahmen: Passwort mit (un)periodischer Änderung, nachträgliche Kontrolle von Lese- und Schreibvorgängen, Kontrolle von Leseversuchen an gesperrten Daten. Eine Sicherung ist nur so stark wie das schwächste Glied. Dieses ist meist irgendein Mensch.

(iii) Sicherung gegen Mißbrauch durch Fachleute:
Maßnahmen: Trennung der Kompetenzen.

(iv) Datenschutz: Dies ist ein politisches und juristisches Problem.

8.6 Datenfernübertragung

Die folgenden Kriterien sind zum Entwurf oder der Beurteilung von Systemen zur Datenfernübertragung wichtig.

Schicht Nr.	Protokoll	Erklärung, Beispiele
7	Anwendungsprotokolle	
6	Darstellungsprotokolle	Interpretation der Daten, z.B. Sprache, Format
5	Sitzungsprotokoll 'Session'	Auf- Abbau der Verbindung auf Benutzerebene, Datenfluß- kontrolle, Synchronisation
4	Ende-zu-Ende- Transportprotokoll	Netzwerk unabhängige Verbindung zweier Teilnehmer
3	Netzwerkprotokoll	Transport durch das Netz, logische Kanalvermittlung
2	Leitungsprotokoll 'link'	Prüfung auf Zuverlässigkeit, Fehlerkorrektur HDLC
1	Physikalisches Protokoll	Darstellung der Signale, physischer Auf-, Abbau der Leitung

Tabelle 8.1: Das ISO-Sieben-Schichten-Modell

		Beispiel: DATEX-P der Bundespost
1)	'Fern' oder 'lokal'? 'lokal': einige cm bis zu einigen km. 'fern' meist über Postleitungen. Lokale Netze haben meist hohe Datenraten (~ Mbit/s), Fernnetze haben aus Kostengründen meist niedrige Datenraten.	'Fern': Anschlüsse im globalen Rahmen.
2)	Standards, Protokolle: Protokolle sind Konventionen beim Aufbau und Abbruch einer Verbindung sowie bei der Datenübertragung selbst	X.25 Paketübertragung (packet-switching)
3)	Rechner und Betriebssysteme. Auf welchen Rechnern sind die Protokolle implementiert?	auf dem meisten Rechnern
4)	Angebotene Dienste, Einschränkungen (z.B. Terminal-access, RJE, ...)	Zugang mit Konsolen über PAD (packet assembly-disassembly)
5)	Kosten: Man unterscheidet feste Kosten, übertragungs-, zeitabhängige und übertragungsvolumenabhängige Kosten	
6)	Qualität (Zuverlässigkeit, Fehlertoleranz)	gute Fehlerkorrektur
7)	Datenübertragungsrate	<9.6 kbaud

8.6.1 Fern-Netze

Abb. 8.9 zeigt das Schema: Fernnetze haben oft viele Teilnehmer mit den unterschiedlichsten Geräten. Damit sie miteinander in Verbindung treten können, bedarf es Vereinbarungen über die formalen Erfordernisse beim Auf- und Abbau der Verbindungen so wie über das Format der übertragenen Daten. Diese Vereinbarungen nennt man Protokolle. Sie sollen eine Standardisierung und weitgehende Unabhängigkeit vom Gerätetyp bewirken. Nach ihrer Festlegung bedürfen die Protokolle der Implementierung auf den verschiedenen Geräten.

Angestrebt wird eine Standardisierung der Protokolle im internationalen Rahmen. Ein Gerüst ist das ISO-sieben-Schichten-Modell , für dessen unterste Schichten bereits internationale Vereinbarungen bestehen (Tab. 8.1).

In dem ISO-Modell nennt man Schicht Nr.1-4 die Transportebene, Schicht Nr.5-7 ist die Anwendungsebene. Das X.25-Protokoll umfaßt die Schichten Nr.1-3.

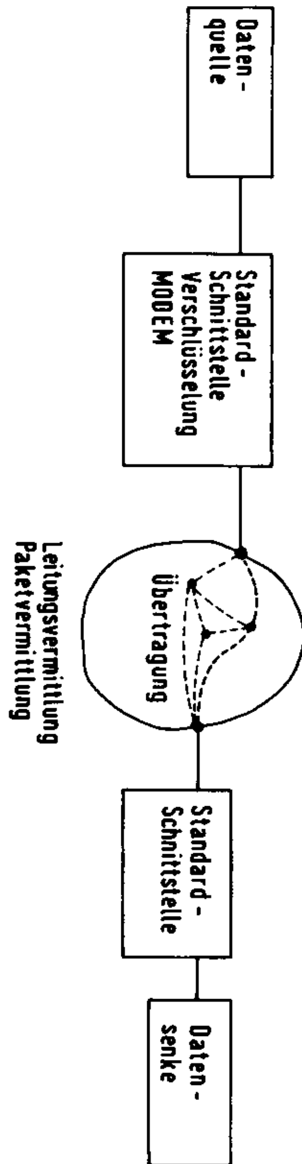


Abbildung 8.9: Fernnetz

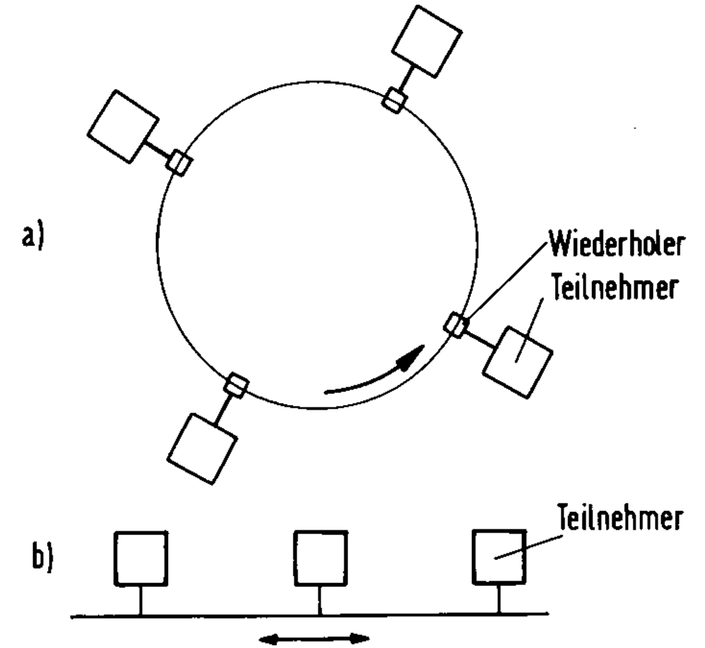


Abbildung 8.10: Lokale Netze

8.6.2 Lokale Netze

Lokale Netze sind oft für hohe Datenraten ausgelegt. Hier werden zwei Modelle vorgestellt (s. Abb. 8.10).

(a) Ringleitung: Jeder Teilnehmer ist an den Ring über einen Empfänger mit Wiederholer angeschlossen, der im Ring drinsitzt. Falls viele Empfänger und Wiederholer im Ring sitzen, kann dies Probleme der Zuverlässigkeit schaffen. Falls ein Teilnehmer über den Ring senden will, muß er sich Zugang verschaffen. Dazu gibt es verschiedene Verfahren, z.B. 'token passing' (Übergabe eines bestimmten Kennworts ermöglicht die Übertragung) oder 'empty slot' (im ringförmig zirkulierender Datenstrom gibt es (zeitliche) Lücken in welche neue Daten eingeschossen werden können).

(b) Bus: Die Übertragung erfolgt durch Adressierung eines 'Geräts' (Rechner, Speichermodul, E/A-Gerät) auf dem Bus. Schnittstellen sind Register. Ein gebräuchliches Zugangsverfahren ist CSMA/CD (carrier sense multiple access with collision detection): Ein Gerät, welches senden möchte, wartet, bis kein Verkehr auf dem Bus herrscht. Dann beginnt es zu senden. Falls ein zweites Gerät dieselbe Idee hatte und ebenfalls (fast) gleichzeitig zu senden beginnt, stoppen beide Geräte ihre Übertragung und nehmen sie mit willkürlichen Verzögerungen wieder auf.

Literatur

[1] K. Bauknecht und C. A.Zehnder. *Grundzüge der DV*, Teubner (s.a. weitere dort angegebene Literatur)

[2] L. Ponzin. *Loacal Area Networks*, 1982 CERN School of Computing

[3] R.W. Dobinson. *Bus Basics*, 1982 CERN School of Computing

[4] Rice and Rice. *Introduction to Computer Science*, Rinehart und Winston

[5] Aho, Hopcroft und Ullmann. *The Design and Analysis of Computer Algorithms*, Addison-Wesley

Kapitel 9

Vergangenheit und Zukunft der Informatik

*'Porro quod tu logistice, idem
ego mechanice nuper tentavi,
et machinam extruxi undecim
integris et sex mutilatis rotulis
constantem, quae datos numeros
computet, addat, subtrahat,
multiplicet, dividatque. Rideres clare,
si praesens cerneres, quomodo
sinistros denarium vel centenarium
supergressos sua sponte coacervet,
aut inter subtrahendeum ab eis
aliquid suffuretur.'*

W. Schickard an J. Kepler, 20.9.1623

9.1 Zur Geschichte der Informatik

ZEITTAFEL

1623	W. Schickard (1592-1631) Prof. für Hebräisch, Aramäisch, Chaldäisch und Syrisch an der Universität Tübingen	Erste funktionierende mechanische Rechenmaschine: sechs Dezimalstellen, vier Grundrechenarten
1641	B. Pascal	Grundkonzept moderner
1672	G. W. Leibniz	Rechenmaschinen: Speicher, Dualsystem, logische Operationen
1823	Ch. Babbage Prof. in Cambridge	Erstes Großrechner- konzept: Rechenwerk, Steuersystem, Speicher mit 1000 50-stelligen Zahlen, 50 000 Räder
1890	H. Hollerith	Lochkartenmaschinen für die US Volkszählung
1937-1941	K. Zuse Bauingenieur	Z1-Z3: Erste moderne elektrische Rechenmaschine: Binärsystem, Rechenwerk, Gleitkomma, gelochter Film als Programmspeicher, 3000 Relais, 64 Zahlen- Speicher, 20 Operationen/s.
1944	H. H. Aitken	Mark I: Lochstreifenleser, Programm auf Stecktafeln, 60 Zahlen-Speicher, 16m lang, Multiplikation 6 s
1946	Eckert und Mauchley University of Pennsylvania	Eniac:Electronic Numerical Integrator and Computer; dezimale Rechenweise, 1800 Röhren, 1500 Relais, Programm fest verdrahtet, Multiplikation 2.8 ms
1949	Universität Cambridge England	Edsac: gespeichertes Programm (Ultraschallspeicher), Binärsystem, erste J.v. Neumann-Maschine
1951	Remington-Rand	UNIVAC: erster kommerzieller Großrechner, Hg-Verzögerungsspeicher
1956	Erste Generation IBM 704, IBM 709	Kernspeicher, Datenkanäle, 12 μs Zykluszeit, 36 bit Wort
1960	Zweite Generation IBM 7090	Transistoren, 2.2 μs Zykluszeit
1967	Dritte Generation IBM 360-Reihe	Integrierte oder hybride Bausteine, Si-Technologie, LSI (large scale integration), 700 ns Zykluszeit
1982	Vierte Generation Mikrorechner,IBM 3081	VLSI (very large scale integration)
1988(?)	Fünfte Generation	Ga As-Technik, Expertensysteme.

9.2 Turing-Maschinen

Alan M. Turing, ein englischer Mathematiker, hat 1936 eine grundlegende Arbeit veröffentlicht, in der er sich mit Algorithmen[1] auseinandersetzt.

Unter einem Algorithmus versteht man eine Reihe von Instruktionen, die Schritt für Schritt ausgeführt zur Lösung einer Aufgabe führen.

Fragen:

(i) Wann sind die Instruktionen eindeutig?

(ii) Wann terminiert der Algorithmus?

Die 2. Frage wird in Kapitel 9.3 behandelt. Turing beschäftigt sich mit Frage (i). Seine Antwort: Die Regeln und damit ein Algorithmus sind dann eindeutig gegeben, wenn sie eine einfache Maschine, die Turingmaschine, ausführen kann.

[1] nach al Chwarismi, einem arabischen Mathematiker, der 820 ein Rechenbuch veröffentlichte

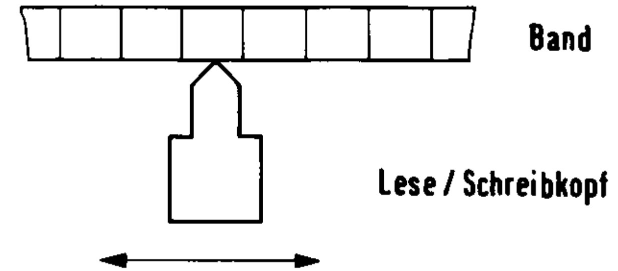


Abbildung 9.1: Turing-Maschine

Die Umkehrung dieses Satzes ist die Turingsche Vermutung:
Jeder Algorithmus läßt sich auf einer Turingmaschine ausführen. Heute wird dieser Satz durch eine große Menge Erfahrung und weitere Forschung gestützt.

Das Schema der Turing-Maschine zeigt Abb. 9.1.

Die Turingmaschine hat als Speichermedium ein Band, das beidseitig unbegrenzt lang ist. Es ist ursprünglich leer, bis auf ein Stück endlicher Länge. Die Maschine selbst ist charakterisiert durch eine endliche Zahl von Zuständen. Sie hat einen Lese/Schreibkopf, mit dem sie Symbole lesen bzw. auf das Band schreiben kann. Der Lese/Schreibkopf kann von einem Feld des Bandes auf das nächste nach rechts oder links bewegt werden.

Die Maschine arbeitet so, daß sie von einem Zustand zum nächsten geht. Die vollständige Abfolge von Operationen, die dazu nötig ist, nennt man einen Zyklus. Ein Zyklus läuft folgendermaßen ab:

1) Der Start erfolgt im Zustand q_i

2) die Maschine liest das Symbol s_j vom Band,

3) die Maschine schreibt, abhängig von Zustand q_i und vom gelesenen Symbol s_j, ein Symbol $s_{ij} = F(q_i, s_j)$ auf dasselbe Feld des Bandes

4) die Maschine rückt den Lese/Schreibkopf um ein Feld nach rechts bzw. links, abhängig von q_i und s_j: $D(q_i, s_j)$

5) die Maschine geht in einen neuen Zustand q_{ij} über, abhängig vom vorhergehenden Zustand q_i und vom gelesen Symbol s_j: $q_{ij} = G(q_i, s_j)$.

6) Die Maschine geht nach 2)

Die Maschine ist also charakterisiert durch ihre Zustände, beschrieben durch Quintupel, die möglichen Kombinationen von Zustand q_i und Eingabesymbol s_j zugeordnet sind:

Quintupel:

q_i	Zustand
s_j	gelesenes Symbol
$s_{ij} = F(q_i, s_j)$	geschriebenes Symbol
$q_{ij} = G(q_i, s_j)$	neuer Zustand
$d_{ij} = D(q_i, s_j)$	Kopfbewegung nach links oder rechts

Die Turingmaschine gehört in die Klasse der unendlichen Automaten, da sie ein Band unbegrenzter Länge hat. Man bezeichnet ihn deshalb besser als unbegrenzten Automaten.

Beispiele von Turingmaschinen:

1) Paritätszähler: Die Maschine startet auf einem Feld des Bandes am Beginn einer Binärzahl.Sie hat zwei Zustände, je einen für eine gerade bzw. ungerade Zahl von 1en, auf die sie getroffen ist. Sie bewegt sich nach rechts, liest die 0en und 1en der Zahl, bis sie auf das Symbol B trifft, welches das Ende der Zahl markiert. Dann druckt sie eine 1, falls die Zahl der 1en im Wort ungerade ist, sonst eine 0.

   Das folgende sind die Quintupel einer Maschine. die dies leistet:

q_i	s_j	q_{ij}	s_{ij}	d_{ij} (1=rechts 0=links)
0	0	0	0	1
0	1	1	0	1
0	B	Halt	0	-
1	0	1	0	1
1	1	0	0	1
1	B	Halt	1	-

2) Klammerprüfer: Die Maschine prüft Anordnungen von rechten und linken Klammern daraufhin, ob alle Klammern richtig verpaart sind, d.h. ob jede rechte Klammer ihre entsprechende linke Klammer hat.

   Als Beispiel: Die Kombination ((()(()))) ist richtig
   Die Kombination (()()))(()(()()) ist falsch.
   Das Prinzip des Klammerprüfers: Gehe nach rechts zur ersten), dann nach links zur ersten (, entferne beide, gehe wieder nach rechts usw. Falls am Ende keine Klammer mehr übrig ist, war die Kombination richtig verpaart, sonst nicht.

   Die Begrenzung des Klammerausdrucks auf dem Band sei durch je ein A vorne und hinten markiert.

   Die Turingmaschine ist durch den folgenden Satz von Quintupeln gegeben:

q_i	s_j	q_{ij}	s_{ij}	d_{ij} 1=rechts0=links
0	)	1	X	0
0	(	0	(	1
0	A	2	A	0
0	X	0	X	1
1	)	1	)	0
1	(	0	X	1
1	A	Halt	0	-
1	X	1	X	0
2	)	-	-	-
2	(	Halt	0	-
2	A	Halt	1	-
2	X	2	X	0

   Die Maschine wird im Zustand 0 auf dem ersten Feld neben dem linksseitigen A gestartet.

   Eine andere, etwa übersichtlichere Darstellung, ist das Zustandsdiagramm (es ist nicht ganz so allgemein wie die Liste der Quintupel). Das Zustandsdiagramm des Klammerprüfers ist in Abb. 9.2 zu sehen.

   In diesem Zustandsdiagramm bezeichnen die Quadrate Zustände der Maschine. R bedeutet eine Bewegung des Lese/Schreibkopfs nach rechts, L nach links. Symbole am Beginn der Pfeile bezeichnen die gelesenen Symbole, in der Mitte der Pfeile sind die geschriebenen Symbole angebracht.

   Die Klammerprüfaufgabe kann nicht von einem endlichen Automaten gelöst werden. Ein endlicher Automat hat eine endliche Zahl von Zuständen und ein Band mit einer endlichen Zahl von Feldern. Damit kann der endliche Automat nicht Klammerausdrücke unbegrenzter Länge bearbeiten. da die Maschine bei der Suche nach zusammengehörigen Klammern über Stücke des Bandes von unbegrenzter Länge zurückgehen muß.

UNIVERSELLE TURINGMASCHINE: Dies ist eine Turingmaschine, die interpretativ jede beliebige Turingmaschine simuliert, deren Beschreibung (etwa in der Form von Quintupeln) auf dem Band gespeichert ist.

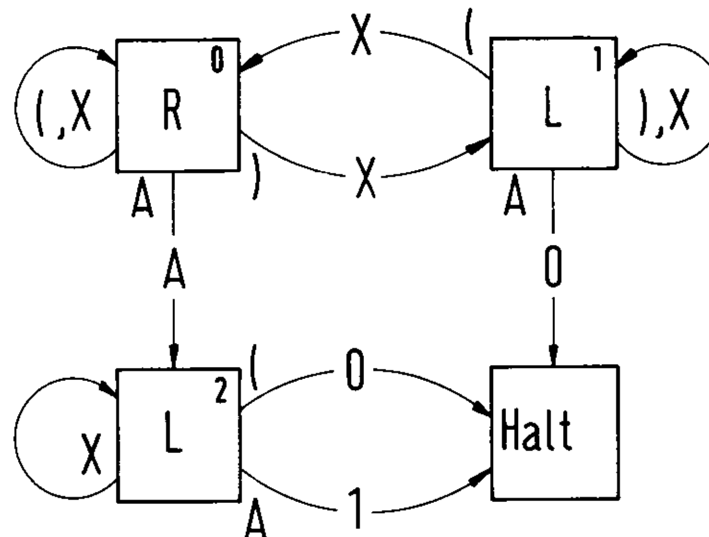


Abbildung 9.2: Klammerprüfer

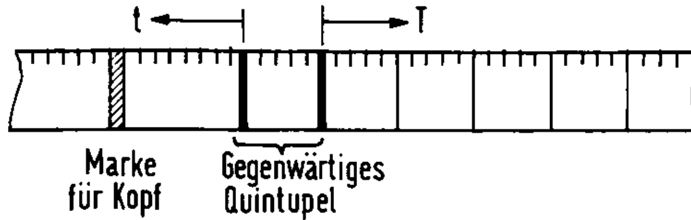


Abbildung 9.3: Universelle Turing-Maschine

Dies macht sie so, daß sie Schritt für Schritt ein Symbol auf dem Band liest, vom aktuellen Quintupel das zu schreibende Symbol bestimmt und schreibt, die Lesemarke nach rechts bzw. links schiebt in Abhängigkeit von dem gelesenen Symbol und dem gegenwärtigen Zustand, dann sucht sie den neuen Quintupel auf dem Band, der zum gegenwärtigen Zustand und zum gelesenen Symbol gehört u.s.f.

Abb. 9.3 zeigt die Organisation des Bandes für die universelle Turingmaschine In Abb. 9.3 steht nur die linke Seite des Bandes für Rechenzwecke zur Verfügung, da die rechte Seite das Programm (die Liste der Quintupel) enthält. Dies ist jedoch keine Schwierigkeit. Man kann z.B. vereinbaren, daß auf der linken Bandseite die Felder mit geradzahliger Feldnummer die linke Seite des alten Bandes bedeuten und die Felder mit ungeradzahligen Feldnummern die rechte Seite des alten Bandes.

9.3 Das Halt-Problem

Die Beantwortung des Halt-Problems beinhaltet eines der tiefsten und wichtigsten Ergbnisse der theoretischen Informatik.

Das Problem: Gibt es einen Algorithmus, der für jede Turingmaschine mit zugehörigem Band entscheiden kann, ob die Turingmaschine nach einer endlichen Zahl von Schritten anhält?

Die erstaunliche und wichtige Antwort auf diese Frage lautet: NEIN. Nach Turings These bedeutet dies, daß es auch keine Turingmaschine gibt, die dies leistet.

Dieses Ergebnis impliziert u.a. die Unmöglichkeit, ein allgemeines Programm zu erstellen, welches die Fehlerfreiheit beliebiger Programme prüft.

Man kann diesen Satz auch mit der Idee der universellen Turingmaschine formulieren: Es gibt keine Turingmaschine, die mit dem Band t_T, d_T (t_T=Turingband der Maschine T, d_T Beschreibung der Maschine T auf dem Band) als Eingabe prüfen kann, ob der so beschriebene Prozess nach einer endlichen Zahl von Schritten abbricht.

Man könnte zur Lösung des Problems zunächst auf die Idee kommen, eine obere Schranke für die Zahl von Zyklen zu berechnen, die auftreten kann, falls die Maschine stoppt. Dann würde es genügen zu beobachten, ob die Maschine mit der Zahl ihrer Zyklen diese Schranke überschreitet; wenn sie es tut, bricht der Prozess nie ab, und man hätte seine Antwort. Dies wäre ein akzeptabler Algorithmus. Dies klingt gut, weil die Zahl der Zustände und Symbole und die Zahl der ursprünglich beschriebenen Felder auf dem Band und die Zahl ihrer möglichen Kombinationen endlich sind, und es müßte doch gelingen, hieraus eine Schranke zu berechnen. Der nachfolgende Beweis der Unmöglichkeit, die Frage zu entscheiden, zeigt jedoch, daß man eine solche Schranke nicht berechnen kann. Ich vermute, daß dies mit der sehr großen (unberechenbar großen) Zahl von Zyklen zusammenhängt, die manche rekursive Programme implizieren.

Nun zum Beweis der Unmöglichkeit einer Entscheidung: Er wird dadurch geführt, daß man das Gegenteil annimmt und hieraus einen Widerspruch konstruiert.

Man nimmt also an, es gäbe eine Maschine D, welche, angesetzt auf das universelle Turingband (t_T, d_T)

entscheidet, ob der durch (t_T, d_T) beschriebene Prozess nach einer endlichen Zahl von Zyklen stoppt, und dies muss für alle Bänder funktionieren. Die Maschine D hat also zwei Halt-Ergebnis-Endzustände: Der eine sagt: Der Prozess stoppt, der andere sagt: Der Prozess stoppt nicht.

Nach Voraussetzung soll dies für alle Bänder möglich sein. Dann muß es auch für das spezielle Band (d_T, d_T) möglich sein. Dies bezeichnet eine Turingmaschine der Beschreibung d_T, welche ihre eigene Beschreibung d_T als Eingabeband hat. Dies ist äquivalent mit einem Eingabeband, welches nur die rechten Seite beschrieben hat (die linke Seite kann man sich notfalls je immer überkopiert denken). Als lezten Schritt bringt man an der ursprünglichen allgemeinen Entscheidungsmaschine D eine Modifikation an: Der Halt-Ergebnisausgang, der zur Antwort:'Algorithmus stoppt' gehört, wird so verändert, daß er in eine nicht endende Schleife führt; die Maschine D stoppt also nie, wenn sie zum Ergebnis kommt, daß der untersuchter Prozess stoppt.

Der andere Ergebnisausgang wird nicht modifiziert; falls die Antwort der Untersuchung ist: 'Algorithmus stoppt nicht', dann stoppt die Maschine D mit dieser Antwort.

Diese modifizierte Maschine nennen wir D^*. Für sie können wir sagen:

$D^*(d_T)$ hält an, falls der untersuchte Prozess T nicht anhält.

$D^*(d_T)$ hält nicht an, falls der untersuchte Prozess T anhält.

Die Frage ist nun: Was passiert, wenn man der Maschine D^* ihre eigene Beschreibung d_{D^*} zur Prüfung vorlegt? Dies ist zulässig, da das Verfahren für jeden Algorithmus funktionieren soll.

Nach dem oben gesagten lautet das Ergebnis:

$D^*(d_{D^*})$ hält an, falls der Prozess D^* nicht anhält.

$D^*(d_{D^*})$ hält nicht an, falls der Prozess D^* anhält.

Dies ist ein Widerspruch, also existiert D^* nicht, also existiert D nicht.

Davon abgeleitet sind die folgenden anderen Nicht-Entscheidbarkeitssätze:

Es ist unmöglich, einen Algorithmus aufzustellen, der im allgemeinen Fall entscheidet, ob ein Programm jemals ein bestimmtes Symbol schreibt.

Die Unentscheidbarkeit bleibt auch bestehen, wenn man sich bei der Aufgabenstellung darauf spezialisiert, nur Turingmaschinen mit ursprünglich leerem Band zu betrachten. Man kann nämlich das allgemeine Turingproblem auf das eben erwähnte zurückführen, wenn man den Inhalt des Bandes (endlich viele Felder) den Zuständen der Maschine hinzufügt.

Im Gegensatz zu der Unmöglichkeit, für einen beliebigen Algorithmus die Frage des Haltens zu entscheiden, kann dies für einen ganz bestimmten vorgelegten Algorithmus sehr wohl möglich sein. Man kann für diesen die Unentscheidbarkeit nicht beweisen.

Der Beweis geschieht durch 'Reductio ad absurdum'. Man nimmt an, die Unentscheidbarkeit, ob das Ding stoppt, sei bewiesen. Da es sich aber um einen ganz bestimmten Prozess handelt, kann man es ausprobieren. Falls er nach endlich vielen Schritten stoppt, weiß man, daß er stoppt, die Behauptung der Unentscheidbarkeit wäre widerlegt. Da dies nicht sein darf, kann die Maschine nie stoppen, d.h. die Frage ist ebenfalls dahingehend entschieden, daß die Maschine nie stoppt, wieder im Gegensatz zur Annahme einer Unentscheidbarkeit.

9.4 Die Grenzen der Von-Neumann Maschine

Wie schnell können seriell arbeitende Rechenmaschinen werden? Abb. 9.4 zeigt die historische Entwicklung der Zahl der auf einem Chip integrierten Schaltelemente. Dieser Fortschritt hat direkt mit der Steigerung der Rechengeschwindigkeit zu tun, wie folgende Überlegungen zeigen: Sind R und C typische, für die verwendete Technologie charakteristische Werte von Widerstand und Kapazität eines Schaltelements, so ist die charakteristische Schaltzeit Δt gegeben durch

$$\Delta t \sim RC$$

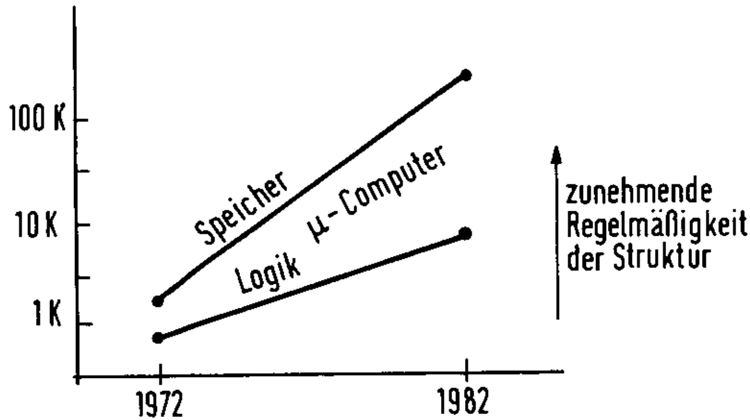


Abbildung 9.4: Chips

Die Rechengeschwindkeit ist etwa proportional zu $1/\Delta t$. Eine Verkleinerung der Schaltelemente führt u.a. zu einer Verkleinerung von C und damit zu einer Verkleinerung der Schaltzeit. Auf der anderen Seite ist die Verlustleistung

$$P \sim I \cdot U \sim \frac{U^2}{R} \sim \frac{U^2 C}{\Delta t}$$

oder

$$P \cdot \Delta t \sim U^2 C$$

Dies heißt: In einer gegebenen Technologie (U,C) ist hohe Rechengeschwindigkeit mit hoher Leistungdichte, d.h. Wärmeentwicklung, gekoppelt, und dies setzt einer beliebigen Steigerung der Rechengeschwindigkeit eine Grenze. Man muß also trachten, den Energieaufwand pro Schaltvorgang $P \cdot \Delta t$, möglichst gering zu halten, und da er proportional zu U^2 C ist, sinkt er mit fortschreitender Miniaturisierung, d.h. mit kleinerem C. Den im Laufe der Jahre erzielten Fortschritt zeigt Abb. 9.5. Kann dies unbegrenzt so weitergehen? Nein.

Die Thermodynamik sorgt für eine Begrenzung: Die pro Schaltvorgang umgesetzte Energie muß $\gg$ kT sein, sonst wird die Maschine durch thermodynamisch bedingte Zustandsänderungen gestört.
Also:

$$P \cdot \Delta t \sim \Delta E \gg kT$$

oder

$$P \gg kT/\Delta t$$

Die Quantentheorie sorgt für eine weitere Begrenzung:

$$\Delta E \cdot \Delta t > \hbar$$

$$\Delta E \sim P \cdot \Delta t > \hbar/\Delta t$$

also

$$P > \hbar/\Delta t^2$$

Man sieht, daß man schlußendlich hohe Rechengeschwindigkeit ungeachtet aller Fortschritte der Technologie durch hohe Schaltleistungen erkaufen muß.

Wie weit kann man dies treiben? Dies ist eine Frage der Kühltechnik und der angestrebten Zuverlässigkeit.

Strebt man eine Schaltzeit Δt an, so dürfen die Schaltelemente höchstens die Entfernung $c \cdot \Delta t$ voneinander haben, sonst bestimmen die Laufzeiten die Rechengeschwindigkeit (c=Lichgeschwindigkeit). Auf der Fläche A

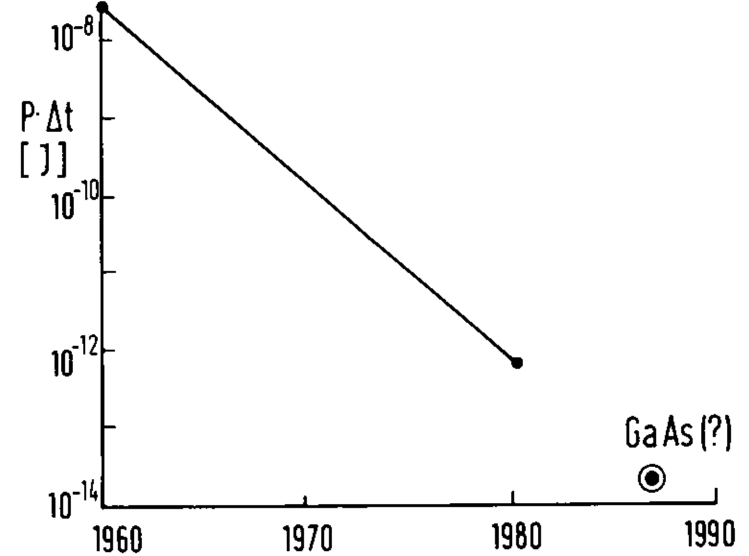


Abbildung 9.5: Mittlere zum Schalten eines Elements benötigte Energie in den Jahren 1960-1980

sitzen also mindestens $A/(c\Delta t)^2$ Schaltelemente, ihre Wärmeverlustleistung ist also mindestens $P \cdot A/(c \cdot \Delta t)^2$. Diese Zahl darf einen bestimmten Wert nicht übersteigen, sonst verbrennt das Ding. Die aus der Annahme einer maximal zulässigen Verlustleistung von W/cm^2 resultierende Grenze sowie die aus der Thermodynamik und der Quantentheorie folgenden Grenzen sind in Abb. 9.6 zusammengestellt. Die Abb. 9.6 enthält auch einen sehr vorläufigen Punkt, der aus der Verwendung von GaAs statt Si als Halbleitermaterial resultiert und eine Steigerung der Rechengeschwindigkeit um 1–2 Größenordnungen versprechen könnte. Schaltzeiten von 0.17 ns sind bis jetzt erreicht worden.

Eine andere Technologie, die sehr hohe Schaltgeschwindigkeiten verspricht, beruht auf dem Josephson-Effekt. Die dazu erforderlichen Temperaturen des flüssigen Heliums haben in der Praxis bisher und erwartungsgemäß zu großen Schwierigkeiten geführt.

Wir haben die Grenzen der gegenwärtigen Maschinen auf der Grundlage einer Technologie mit R-C basierenden Schaltelementen als Beispiel behandelt. Andere Schaltprinzipien, etwa mit optischen Elementen oder auf molekularer Basis sind denkbar. Aber auch für diese bleibt Abb.9.6 gültig.

Zum Schluß noch ein Gebot der Praxis: Mit jeder Steigerung der Rechengeschwindigkeit muß eine entsprechend erhöhte Zuverlässigkeit aller Rechnerkomponenten garantiert sein. Glücklicherweise war diese Forderung beim Fortschritt der Schaltelement-Integrierung erfüllt. Sie kann sich aber trotzdem bei sehr schnellen oder sehr großen System zum Problem Nr.1 entwickeln.

9.5 Anhang

Ist es nicht sehr bedauerlich und ein negativer Kulturindikator, daß die Kenntnis der lateinischen Sprache immer mehr zurückgeht?

Die Übersetzung des Mottos zu Kapitel 9 lautet:

'Dasselbe, was Du rechnerisch gemacht hast, habe ich in letzter Zeit auf mechanischem Wege versucht, und eine aus elf vollständigen und sechs verstümmelten Rädchen bestehende Maschine konstruiert, welche gegebene Zahlen augenblicklich automatisch zusammenrechnet: addiert, subtrahiert, multipliziert und dividiert. Du würdest hell auflachen, wenn du zuschauen könntest, wie sie die Stellen links, wenn es über einen Zehner oder Hunderter weggeht, ganz von selbst erhöht bzw. beim Subtrahieren ihnen etwas wegnimmt!

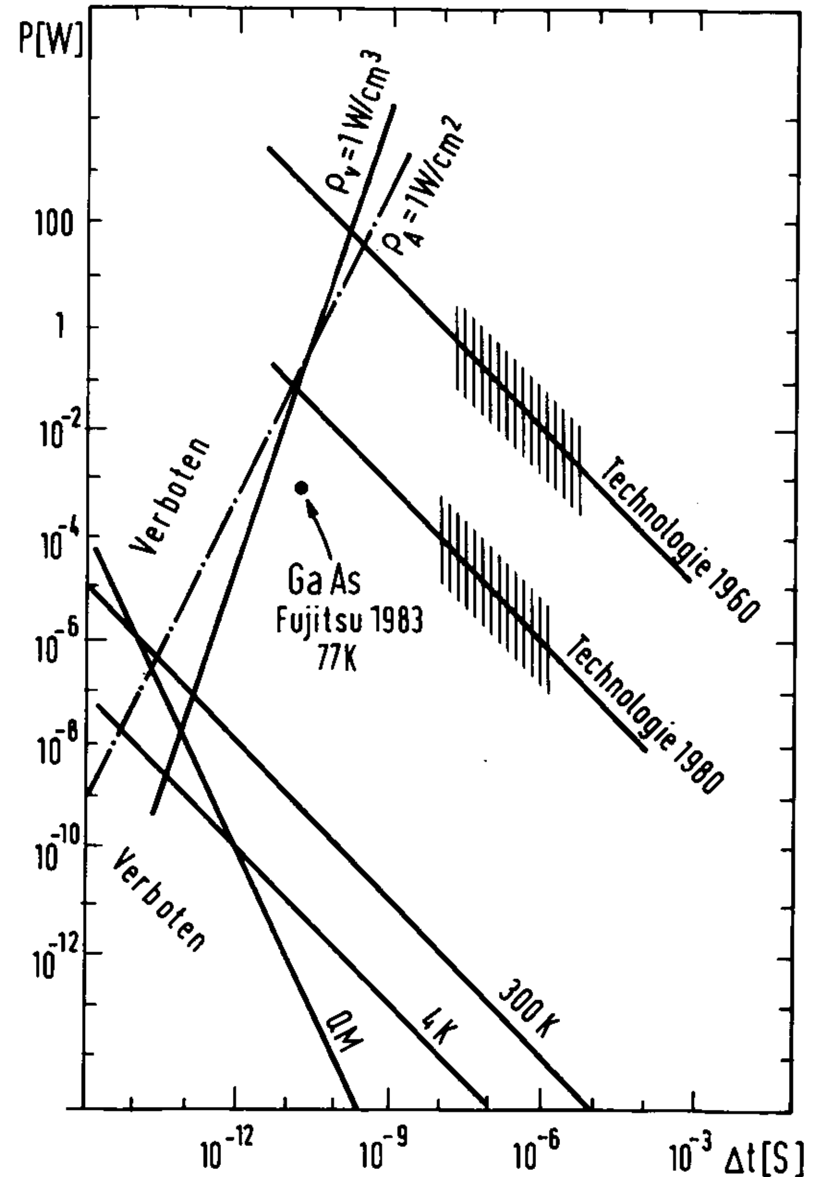


Abbildung 9.6: Mittlere Verlustleistunge pro Schaltelement gegen die Schaltzeit aufgetragen

Literatur

[1] S. Rosen. *Electronic Computers*, in Computing Surveys, 1 (1969) 7

[2] W. de Beauclair. *Prof. Schickards Tübinger Rechenmaschine*, kleine Tübinger Schriften Heft 4

[3] M. Minsky. *Computation*, Prentice Hall

Danksagung

Mein besonderer Dank gilt Frau M. Stuckenberg für die Erstellung des Manuskripts mit dem System LATEX. Den folgenden Kollegen bin ich für viele hilfreiche Hinweise, Verbesserungsvorschläge und Kritik sehr dankbar: M. Behrends, H. Dilcher, E. Freytag, O. Hell, G. Hochweller, D. Lüke, P.-K. Schilling und H.-J. Stuckenberg. Herrn W. Knaut danke ich für das Anfertigen der Abbildungen.