TECHNISCHE
UNIVERSITÄT
DARMSTADT

Technische Grundlagen der Informatik II (Rechnertechnologie II)

Skript zu den Vorlesungskapiteln 1 bis 5 im Sommersemester 2005 (enthält nur die Buchkapitel 0 bis 3 und Teile von 6.2)

Fachbereich Informatik
Fachgebiet Rechnerarchitektur
R. Hoffmann

Dieses Skript behandelt nur einen Teil der Vorlesungsthemen. Für die Vorlesung sind nur die Kapitel 0 bis 3 und 6.2 bis 6.2.2 dieses Skripts relevant. Die anderen Abschnitte sind nur zur weitergehenden Information gedacht.

Teile des Skriptes sind 1993 im Buch
Rechnerentwurf: Rechenwerke, Mikroprogrammierung, RISC
von Hoffmann, Rolf
im Oldenbourg-Verlag, München
ISBN 3-468-22174-4
erschienen.

© Copyright 2004 R. Hoffmann
Alle Rechte vorbehalten.
Ausgabe 2005, Stand April 2005

Inhaltsverzeichnis

Vorwort v

Vereinbarungen und Symbole vii

0 Zeichencodierung 1
0.1 Morse-Code 1
0.2 Fernschreiber 2
0.3 Grundlagen von optimierten Codes 3
0.3.1 Shannon-Fano-Code 4
0.3.2 Huffman-Code 4
0.3.3 Vergleich 5
0.3.4 Fehlerbehandlung 5
0.4 Gebräuchliche Zeichencodes 8
0.4.1 ASCII 8
0.4.2 EBCDIC 10
0.4.3 IBM PC Code 10
0.4.4 Unicode 11

1 Zahlendarstellungen 15
1.1 Zahl zur Basis b 15
1.2 Vorzeichen-Betrag-Darstellung 16
1.3 Zweikomplementzahl 17
1.4 Einskomplementzahl 19
1.5 Binärcodierte Dezimalzahl 20
1.6 Gleitkommazahl 22
1.6.1 IEEE-32-Bit-Gleitkommmaformat 27
1.6.2 IEEE-64-Bit-Gleitkommmaformat 30

2 Definition der Hardware-Beschreibungssprache HDL 31
2.1 Struktur eines HDL-Programms . . . 34
2.2 Kommentare, Marken, Namen . . . 34
2.3 Datenformate . . . 34
2.4 Konstanten . . . 35
2.5 Basistypen und Zuweisungen . . . 36
2.6 Deklaration von Variablen . . . 37
2.7 Variablenzugriff, Indizierung . . . 39
2.8 Operatoren . . . 40
2.8.1 Vektoroperatoren . . . 40
2.8.2 Matrixoperatoren . . . 42
2.9 Ausdrücke, Formatanpassung . . . 44
2.10 Permanente Anweisungen . . . 44
2.11 Asynchrone Prozesse . . . 46
2.12 Synchrone Automaten . . . 49
2.13 Equal-Deklaration, Const-Deklaration . . . 51
2.14 Subunits . . . 52
2.15 Makro-Operationen . . . 52
2.16 HDL-Programmbeispiele . . . 53
2.17 Verschiedene Arten von Mikroprogrammen . . . 54
2.18 HDL-Syntax . . . 57

3 Mikroalgorithmen und Rechenwerke für die Grundrechenarten 65
3.1 Addition . . . 66
3.1.1 Addition von Dualzahlen . . . 66
3.1.2 Subtraktion allgemein . . . 77
3.1.3 Addition von Vorzeichenzahlen . . . 77
3.1.4 Addition von Zweikomplementzahlen . . . 79
3.1.5 Addition von Einskomplementzahlen . . . 82
3.1.6 Doppelwort-Addition . . . 84
3.1.7 Addition von binärcodierten Dezimalzahlen . . . 84
3.1.8 Addition von Gleitkommazahlen . . . 88
3.2 Multiplikation . . . 90
3.2.1 Multiplikation von Dualzahlen . . . 90
3.2.2 Multiplikation von Zweikomplementzahlen . . . 94
3.2.3 Parallele Multiplikation . . . 103
3.2.4 Doppelwort-Multiplikation . . . 107
3.2.5 Multiplikation von binärcodierten Dezimalzahlen . . . 109
3.2.6 Multiplikation von Gleitkommazahlen . . . 110
3.3 Division . . . 111
3.3.1 Division von Dualzahlen . . . 113
3.3.2 Division von Zweikomplementzahlen . . . 117
3.3.3 Parallele Division . . . 124

3.3.4 Division von binärcodierten Dezimalzahlen 124
3.3.5 Division von Gleitkommazahlen 127
3.4 Konvertierung zwischen Zahlensystemen 128
3.4.1 Summationsmethode 128
3.4.2 Divisionsmethode 130
3.4.3 Multiplikationsmethode 131

6 Rechnerentwurf 133
6.1 Einige Entwurfsaspekte 133
6.2 Entwurf eines Beispiel-Rechners 144
6.2.1 DINATOS-Architektur 145
6.2.2 Direkte Hardware-Steuerung 147
6.2.3 Mikroprogrammierbarer Rechner PIRI 153
6.2.4 Bewertung der Lösungen 166

Literaturverzeichnis 169

Sachverzeichnis 175

Vorwort

Das Buch behandelt die Architektur und den Entwurf von CISC (Complex Instruction Set Computer) und RISC (Reduced Instruction Set Computer). CISC sind die klassischen Rechner mit umfangreichen Befehlssätzen, die überwiegend mit Hilfe von Mikroprogrammen (Firmware-Interpretation) implementiert werden. RISC sind Rechner mit reduzierten Befehlssätzen, die mit Hilfe von Hardware-Steuerwerken (festverdrahtet) implementiert werden.

Themenschwerpunkte sind

- Zahlendarstellungen,
- Struktur von arithmetischen Rechenwerken und ihre Steuerung durch Mikroalgorithmen (Hardware-Algorithmen),
- der Aufbau von Hardware- und Mikroprogramm-Steuerwerken,
- das Prinzip der Mikroprogrammierung zur Realisierung von Befehlssätzen (Funktionelle-Architektur) durch Interpretation,
- Entwurfsgesichtspunkte für Mikroprogramm-Steuerwerke (wie Speicherplatzminimierung, Kopplung von Steuerwerk und Operationswerk, Pipelining auf der Mikrobefehlsebene),
- Entwurf einer Beispiel-Rechnerarchitektur DINATOS mit einem Hardware-Steuerwerk auf der Register-Transfer-Ebene,
- Entwurf eines mikroprogrammierbaren Rechners (Mikromaschine) PIRI und die Emulation (Interpretation durch ein Mikroprogramm) der DINATOS-Architektur,
- RISC-Achitekturkonzepte wie Pipelining und Cache-Speicher, Superskalar-Prinzip,

- Architektur der RISC-Rechner Berkeley RISC I/II, Sun SPARC und Motorola 88110.

Diese Themen werden vornehmlich unter den Aspekten *Funktionsweise* (Funktionelle-Architektur) und *Register-Transfer-Implementierung* (Register-Transfer-Architektur) behandelt. Um diese Aspekte auch formal besser fassen zu können, wurde die Hardware-Beschreibungssprache HDL vom Autor definiert (1975-1977). Sie ermöglicht die Beschreibung von Architekturen auf verschiedenen Ebenen (Schaltnetz-parallel, algorithmisch-asynchron, Register-Transfer-synchron). Das Mehrebenen-Sprachkonzept wurde später auch in andere Hardware-Bescheibungssprachen wie VHDL aufgenommen. Aufgrund ihrer verhältnismäßigen Einfachheit, besseren Lesbarkeit und bestimmter semantischer Eigenschaften wird HDL gegenüber VHDL als Beschreibungswerkzeug bevorzugt.

Die Mikroprogrammierung bildet einen Schwerpunkt des Buches, die zur Zeit nicht mehr die praktische Bedeutung hat, wie in den 60er und 70er Jahren. Sie war und ist ein Mittel, um die Software-Kompatibilität zu gewährleisten und um umfangreiche Befehlssätze zu implementieren. Diese Technik wird heute noch in CISC-Rechnern angewendet und sie könnte auch in zukünftigen hybriden CISC/RISC-Rechnern wieder verstärkt zum Einsatz kommen. Die Mikroprogrammierung kann auch als eine sehr hardwarenahe Maschinenprogrammierung angesehen werden, ähnlich wie die RISC-Maschinenprogrammierung. In der Tat weisen die Mikrobefehle sehr große Ähnlichkeiten mit den RISC-Maschinenbefehlen (festes Format, Register-Register-Befehle, Ausführung in wenigen Takten, Pipelining) auf, so daß die Konzepte der Mikroprogammierung auf die RISC-Maschinen übertragen werden können und umgekehrt. Ein mikroprogrammierbarer Rechner entspricht prinzipiell einem RISC-Rechner mit *Harvard-Architektur* (getrennte Programm- und Datenspeicher), wenn man den Mikroprogrammspeicher als Maschinenprogrammspeicher benutzt.

Ich möchte allen danken, die inhaltliche Anregungen gaben und/oder bei der drucktechnischen Aufbereitung behilflich waren: Herr Prof. Dr. Liebig (TU-Berlin), Frau Jörs, Herr Dr. Völkmann, Herr Hochberger und Herr Baumgart. Mein besonderer Dank gilt Frau Schwarzkopf, die einen großen Teil des Textes in TEX verarbeitet hat und dabei viele Fragezeichen angebracht hat. Herr Halbach hat mich inbesondere bei der schnellen Erstellung der Bilder unterstützt, wofür ich mich herzlich bedanke.

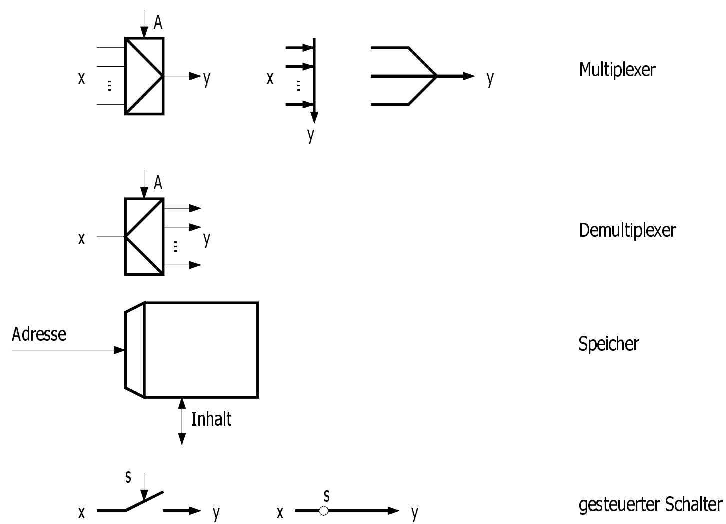
Darmstadt, im Sommer 1993 Rolf Hoffmann

Herr Heenes hat auf Verbesserungen hingewiesen, die Anfertigung der aktuellen LaTeX-Version für Lehrzwecke angeregt und konnte Herrn Meuth dafür gewinnen. Herr Meuth hat Fehler beseitigt und äquivalente Verilog-Programme entworfen. Herr Halbach hat die Skriptversion angefertigt. Herr Amend hat die Grafiken überarbeitet. Allen Beteiligten sei für ihre Mithilfe herzlich gedankt.

Darmstadt, im Herbst 2005 Rolf Hoffmann

Vereinbarungen und Symbole

$\{k : m\} = \{k, k+1, k+2, \ldots, m\}$	Menge ganzer Zahlen von k bis m
$X[n] = X(n:1) = X_n X_{n-1} \ldots X_1$	Dualzahl (Kapitel 1 und 3)



$X[n] = X(n-1:0)$	Vektor boole in HDL (Kapitel 2, 4–7)
$A_j = A(j)$	j-tes Element eines booleschen Vektors
$A^i = A(i,) = A(i)$	i-te Zeile einer booleschen Matrix
$A_j = A(,j)$	j-te Spalte einer booleschen Matrix
$A^i_j = A(i,j)$	Element (i,j) einer booleschen Matrix
0, 1	Null, Eins
`o, l`	boolesche Werte False und True

. Und, $\vee$ Oder, $+$ Plus, $*$ Mal, / Geteilt durch

Erklärung weiterer HDL-Operatoren siehe Abschnitt 2.8

0 Zeichencodierung[1]

Zur Darstellung geschriebener Sprache werden Symbole verwendet. Früher wurden diese in Steintafeln geritzt oder auf Papyrus-Rollen geschrieben. Mit der Entdeckung im Jahre 1773, elektrische Impulse zu entfernten Orten senden zu können, kam die Idee der Telegrafie auf, die der deutsche Mediziner Samuel Thomas von Sömmering (1755 – 1830) 1809 in einem ersten Experiment durchführte[2], nachdem der italienische Physiker Alessandro Giuseppe Antonio Anastasio Graf Volta (1745-1827) erstmals Elektrizität chemisch in einer Batterie speichern konnte. Danach wurden Geräte entwickelt, die über eine Zeigerstellung die gesendeten Zeichen darstellen konnten. 1844 baute Samuel Morse (1791 – 1872) unter Mitwirkung seines Mitarbeiters A. Vail in den USA eine Maschine, die über nur eine Datenleitung Nachrichten auf einem Papierstreifen mitschreiben konnte, kodiert in kurze und lange Signale.[3]

Im folgenden sind – im historischen Ablauf – verschiedene Methoden dargestellt, um Zeichen in eine andere Form zu bringen, unter anderem auch für fehlersensitive Verfahren. Die Darstellung endet mit der heute gebräuchlichen Form der Zeichenspeicherung in Computer.

0.1 Morse-Code

Die Codierung ist ein Verfahren, das Zeichen aus einem Alphabet in ein anderes Alphabet (im mathematischen Sinne) eineindeutig abbildet, d. h. von einem Quellalphabet (z. B. unsere Zeichen) in ein Zielalphabet (die Zeichen des Codes). Für die Umwandlung von 26 Buchstaben in ein Alphabet von zwei Zeichen (kurze und lange Signale) gibt es verschiedene Möglichkeiten.

[1] Dieses Kapitel wurde von Mathias Halbach geschrieben.

[2] Für jeden der Buchstaben sowie für jede Ziffer gab es je eine Leitung. Auf der Senderseite wurden per Batterie Signale auf die Leitung gegeben. Je Leitung gab es auf der Empfängerseite ein Röhrchen mit einer säurehaltigen Flüssigkeit, die bei einem Stromfluss Luftblasen empor steigen ließen. Darüber konnten Nachrichten gesendet werden.

[3] Hier schon lässt sich der Wandel von der parallelen zur seriellen Technik sehen.

Eine davon ist die von Samuel Morse.

Samuel Morse war ein Professor für Kunst und Design an der New York University. Das Alphabet von Morses Original-Codierung bestand aus Zahlen, die fest einer Kombination aus Punkten, Strichen und Pausen (= Wörtergrenzen) zugeordnet sind.

Die bekanntere Version wurde von seinem Mitarbeiter Alfred Vail entwickelt, der das Alphabet der englischen Buchstaben, Zahlen und Interpunktion als Quellalphabet verwendete. Die Codewörter (das Zielalphabet) besteht aus fünf Symbolen: Punkt, Strich, kurze Pause (zwischen den Zeichen), mittlere Pause (zwischen verschiedenen Wörtern) und lange Pausen (zwischen Sätzen).

Dabei wurde die Codierung der einzelnen Quellzeichen nach der Häufigkeit des Vorkommens optimiert. Heraus kam der heute bekannte Morsecode, der Groß- und Kleinschreibung nicht unterscheidet.

A	·—	I	··	Q	— — ·—	Y	— · ——	5	· · · · ·
Ä	· — ·—	J	· — ——	R	· — ·	Z	— — ··	6	— · · · ·
B	— · ··	K	— · —	S	· · ·	CH	— — ——	7	— — · · ·
C	— · —·	L	· — ··	T	—	,	— — · · ——	8	— — — · ·
D	— · ·	M	——	U	· · —	.	· — · — ·—	9	— — — — ·
E	·	N	—·	Ü	· · ——	1	· — — — —	0	— — — — —
F	· · —·	O	— — —	V	· · ·—	2	· · — — —	@	· — — · —·
G	— — ·	Ö	— — —·	W	· — —	3	· · · — —		
H	· · ··	P	· — —·	X	— · ·—	4	· · · · —		

Abbildung 1: Morse-Code für die deutschen Zeichen

0.2 Fernschreiber

Bei einer späteren Weiterentwicklung der Telegraphie wurden zur Aufzeichnung der Wörter Papierstreifen verwendet. Wenn jetzt die Codierung parallel und nicht mehr sequentiell aufgezeichnet wird, reichen fünf parallele, gleichlange Signale aus, um mehr als die 26 Zeichen des Alphabets darzustellen.

Eine solche Form des Codes wurde 1880 von J. M. E. Baudot (1845 – 1903) erfunden und als Fernschreibercode Nr. 1 (CCITT-Code No. 1) bekannt. Später wurde von Donald Murray ein ähnlicher Code entwickelt: Der Fernschreibercode Nr. 2 (Murray-Code, CCITT-Code No. 2). Dieser Code wird auch heute fälschlicherweise sehr oft als „Baudot-Code“ bezeichnet.

Da neben den Buchstaben auch Zahlen und Zeichen übertragen werden sollten, reichen jedoch die $2^5 = 32$ Möglichkeiten der Darstellung nicht aus. Daher wurde eine Doppelbelegung der

einzelnen Codes vorgenommen. Die Auswahl, welcher der beiden Codes verwendet werden soll, wird durch Steuersignale geregelt, die ebenfalls codiert sind. Die Umschaltung zum Buchstabenmodus erfolgt mit LS (Letter Shift), die zum Ziffernmodus mit FS (Figure Shift).

Code-Nr.	Dualcode	Buchstabe	Ziffer	Code-Nr.	Dualcode	Buchstabe	Ziffer
1	11000	A	–	17	11101	Q	1
2	10011	B	?	18	01010	R	4
3	01110	C	:	19	10100	S	’
4	10010	D	WAY	20	00001	T	5
5	10000	E	3	21	11100	U	7
6	10110	F	(N/A)	22	01111	V	=
7	01011	G	(N/A)	23	11001	W	2
8	00101	H	(N/A)	24	10111	X	/
9	01100	I	8	25	10101	Y	6
10	11010	J	BEL	26	10001	Z	+
11	11110	K	(	27	00010	CR	
12	01001	L	)	28	01000	LF	
13	00111	M	.	29	11111	LS	
14	00110	N	,	30	11011	FS	
15	00011	O	9	31	00100	Space	
16	01101	P	0	32	00000	(Unused)	

Abbildung 2: CCITT-Code No. 2

Hierbei gab es auch zum ersten Mal spezielle Zeichen für einen Zeilenvorschub (LF, Line Feed), Wagenrücklauf (CR, Carriage Return[4]) sowie eine Klingel (BEL, Bell; als Rufzeichen) und ein spezielles Zeichen für eine Anfrage (WAY, Who Are You?). Es gab auch ein Leerraumzeichen (Space) sowie nicht definierte (N/A, Not Assigned) und nicht zulässige Varianten (Unused).

0.3 Grundlagen von optimierten Codes

Eine andere, wieder sequentielle Variante ist die Codierung mit zwei Zeichen (0 und 1) als Zielalphabet, wobei für jedes Element des Quellalphabets eine möglichst kurze Darstellung verwendet werden sollte, ähnlich wie dies bei dem Morse-Code der Fall ist.

[4] Bei den damaligen Schreibmaschinen gab es einen „Wagen“, auf dem das Papier in horizontaler Richtung an der Schreibstelle vorbei geführt wurde. Die einzelnen Buchstaben und Zeichen wurde somit immer an die gleiche Stelle geschrieben und je Symbol wurde der Wagen mit dem Papier ein Stück weiter nach links bewegt. Um zurück an den Anfang der Zeile zu gelangen, ist ein Wagenrücklauf nach rechts notwendig, so dass die Schreibstelle wieder in der ersten Spalte positioniert ist.

0.3.1 Shannon-Fano-Code

Claude Elwood Shannon (30. April 1916 – 24. Februar 2001) erstellte als Erster in dem Artikel „A Mathematical Theory of Communication“ die mathematische Grundlage, wie ein optimaler Code erhalten werden kann. Im gleichen Artikel aus dem Jahre 1948 führte er auch die von J. W. Tukey (1915 – 2000) vorgeschlagene Bezeichnung „Bits“ als Abkürzung von ***bi**nary digi**ts*** ein, um die Länge von zu übertragenden Informationen auf der Basis 2 („ein“ oder „aus“) zu beschreiben.

Ausgangspunkt für einen optimalen Code ist eine Wahrscheinlichkeitstabelle für jedes Element des Quellalphabets. Dabei kann das Quellalphabet sowohl aus einzelnen Zeichen als auch aus ganzen Wörtern, Satzzeichen oder Kombinationen daraus bestehen.

Die Wahrscheinlichkeiten werden abfallend sortiert, so dass das am häufigsten vorkommende Zeichen an erster Stelle steht. Die Liste wird halbiert, so dass in beiden Teilen die Summe der Wahrscheinlichkeiten (annähernd) gleich ist. Der oberen Hälfte wird eine 0 zugewiesen, der unteren eine 1. Sind mehrere Zeichen in einer Hälfte, so wird die Teilung für die jeweilige Hälfte wiederholt und es ergibt sich die nächste Codeziffer. Das Ergebnis sind unterschiedlich lange Codeworte für die einzelnen Zeichen.

Zeichen	**Wahrscheinlichkeit**				**Code**
a	0,35		$0 : \Sigma = 0,35$		00
b	0,17	0: $\Sigma = 0,52$	$1 : \Sigma = 0,17$		01
c	0,17	1: $\Sigma = 0,48$	$0 : \Sigma = 0,17$		10
d	0,16		$1 : \Sigma = 0,31$	$0 : \Sigma = 0,16$	110
e	0,15			$1 : \Sigma = 0,15$	111

Abbildung 3: Beispiel für eine Shannon-Fano-Codierung

Dabei ist die so genannte Fano-Bedingung (benannt nach Robert Fano, geb. 1917) eingehalten: Kein Codewort ist Präfix (Anfangsbestandteil) eines anderen Codewortes.

0.3.2 Huffman-Code

Eine besser generierbare Codierung wurde von David Huffman (1925 – 1999, ein ehemaliger Student von Robert Fano) 1952 entwickelt. Dabei ist die Herangehensweise umgekehrt zu dem Shannon-Fano-Code.

Die einzelnen Wahrscheinlichkeiten des Quellalphabets (analog zu Blättern eines Baumes) werden aufgeschrieben. Die beiden Blätter mit der geringsten Wahrscheinlichkeit werden zu einem Teilbaum zusammengefasst, die Wahrscheinlichkeit ist die Summe der beiden Blätter. Die Zusammenfassung wird fortgesetzt, wobei Blätter und Teilbäume gleichwertig sind. Ist keine Zusammenfassung mehr möglich, dann ergibt sich der Code durch den Weg – ausgehend von

der letzten Zusammenfassung (der Wurzel) bis zu dem jeweiligen Zeichen (Blatt) –, ein linker Zweig wird mit 0 codiert, ein rechter mit 1.

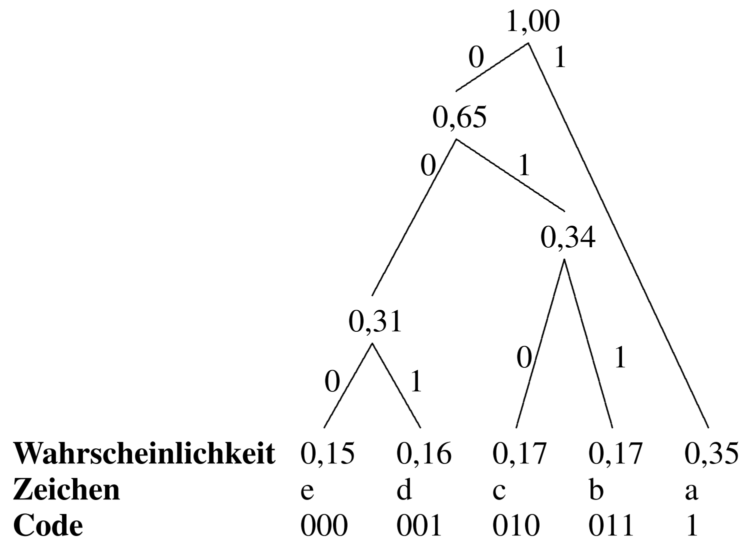


Abbildung 4: Beispiel für den Baum bei einer Huffman-Codierung

0.3.3 Vergleich

Die Effizienz einer Codierung kann mit der durchschnittlichen Codewort-Länge ($\sum_{i=1}^{n} P(a_i) *$ Codelength$_i$) bestimmt werden. Mit der Codierung aus den Abbildungen 3 und 4, die beide die gleiche Wahrscheinlichkeitsverteilung haben, ergibt sich für die Shannon-Fano-Codierung ein Wert von 2,31, für den Huffman-Code der bessere Wert 2,30.

0.3.4 Fehlerbehandlung

Da stark komprimierte Codes sehr fehleranfällig sind, ist es sinnvoll, eine Fehlererkennung und -korrektur mit in die Codierung zu integrieren. Das dies möglich ist, wurde von Shannon bewiesen. Richard Hamming (1915 – 1998) hat dann das erste Beispiel erstellt.

Um die Fehleranfälligkeit bestimmen zu können, ist es notwendig, zwischen allen Codeelementen die Unterschiede bestimmen zu können. Dazu werden für jeweils zwei Elemente die einzelnen Bits der Codewörter miteinander verglichen. Die Anzahl der Unterschiede wird mit „Hamming-Distanz“ (h) bezeichnet. Die Hamming-Distanz eines Codes ist der Minimalwert, der sich zwischen allen Paaren von Codewörtern ergibt.

Für eine Fehlererkennung ist eine Hamming-Distanz h von mindestens 2 notwendig, um die Änderung eines Bits zu erfassen. Für eine Fehlerkorrektur ist aber mindestens $h \geq 3$ notwendig,

um bei genau einem Fehler nur einen, ursprünglich wahrscheinlichen Code zu erhalten.

Eine Alternative zur Fehlerbehandlung einzelner Zeichen ist die Betrachtung eines ganzen Blockes, d. h. mehrere Zeichen werden zu einem Block zusammengefasst und eine „Prüfsumme“ ermittelt. Diese wird mit übertragen und beim Empfänger gleichzeitig generiert. Unterscheiden sich die beiden Werte, gab es einen Übertragungsfehler.

Angewendet wird dies in der zyklischen Blockprüfung (Cyclic Redundancy Checking, CRC). Bei diesem ist es sogar möglich, die empfangenen Daten inklusive Prüfsumme der CRC-Berechnung zuzuführen; ist das Ergebnis (die Prüfsumme) ungleich 0, gab es einen Übertragungsfehler.

Das CRC-Verfahren basiert auf polynomialer Arithmetik modulo 2, d. h. es gibt keine Überträge, wie sie im Dezimalen zum Beispiel bei $1+19$ auftreten (die Summe wäre dann 10 und nicht 20). Werden Bits (zur Basis 2) betrachtet, so sind Addition und Subtraktion wie folgt definiert:

- $0+0=0, \quad 0-0=0$
- $0+1=1, \quad 0-1=1$
- $1+0=1, \quad 1-0=1$
- $1+1=0, \quad 1-1=0$

In beiden Fällen ergibt sich die logische XOR-Funktion. Für die CRC-Berechnung wird auch noch Multiplikation und Division benötigt. Wenn diese analog der üblichen schriftlichen Berechnung abläuft, ist erkennbar, das sie einer Schiebeoperation und XOR-Verknüpfung entspricht und keine weiteren Operationen benötigt werden.

Für die Berechnung einer Prüfsumme wird der Bitdatenstrom[5] als Ganzes betrachtet und vorab um die Anzahl der Bits der Prüfsumme erweitert, der Wert dieser Bits ist am Anfang einheitlich 0. Im nachfolgenden wird die Kombination der beiden als Nachricht bezeichnet. Die Prüfsumme ist der Rest einer Division der Nachricht durch das CRC-Polynom. Das Polynom wird – im Falle des CRC-16-Algorithmus – mit $x^{16}+x^{15}+x^2+1$ oder 0x8005 angegeben. Ein anderer Algorithmus, der eine 4 Byte (32 Bit) breite Prüfsumme erstellt und u. a. für Ethernet, FDDI und ZIP eingesetzt wird, hat den Namen CRC-32 und das Polynom $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ (0x04C11DB7).

Das Verfahren für eine einfache Berechnung ist analog der schriftlichen Division. Basis ist ein Register, das die Breite der Prüfsumme hat; dies ist ein Bit weniger als der Wert des größten Exponenten des CRC-Algorithmus. Dieses Register wird z. B. mit 0-Bits initialisiert. Danach

[5] Da Daten normalerweise aus mehreren Bytes bestehen, wird mit dem ersten Byte angefangen – und zwar vom höchstwertige Bit (7. Position) bis zum niederwertigsten (0. Position).

wird die Nachricht in der niederwertigsten Seite (nach links) hineingeschoben. Wird dabei ein 1-Bit „hinausgeschoben“, ist das CRC-Polynom durch den momentanen Registerinhalt teilbar und wird deshalb abgezogen („Register XOR CRC-Polynom“ wird berechnet). Sobald die Nachricht vollständig verarbeitet ist, steht die Prüfsumme im Register.

Dies lässt sich durch byteweises Arbeiten beschleunigen. Die einzelnen Shift- und XOR-Operationen können – für ein bestimmtes oberstes Byte (das hinausgeschoben wird) – zusammengefasst und im Voraus berechnet werden. Da es für ein Byte nur 256 mögliche Werte gibt, ist es möglich, für jedes mögliche Byte die Vorausberechnung durchzuführen und in einer Tabelle abzuspeichern. Damit reduziert sich die Berechnung des CRC auf ein Achtel des Aufwandes.

Da zum einen die Nachricht am Ende aus 0-Bytes besteht (Anzahl der Bits der Prüfsumme), zum anderen am Anfang das Register mit 0 initialisiert ist – und beides nicht direkt in die Prüfsumme einfließt –, kann auch das Einfügen des Datenstroms mit einer XOR-Verknüpfung des hinausgeschobenen Bytes erfolgen, dieses Zwischenergebnis wird als Tabellenindex verwendet und dann mit XOR auf das Register angewendet. Dies liefert das gleiche Ergebnis wie vorhin.

Bei einer seriellen Übertragung kann es vorkommen, dass das niederwertigste Bit zuerst übertragen wird. Da CRC-Algorithmen auch für diesen Fall zur Verfügung stehen, gibt es entsprechende Variationen in der Software. Es gibt aber auch die Möglichkeit, den Schiebealgorithmus zu drehen, d. h. nach rechts zu schieben. Dadurch dreht sich der vorhin besprochene Tabelleninhalt, aber der Ablauf bleibt prinzipiell der gleiche.

Um die möglichen Varianten zu erweitern, kann das Register mit Einsen statt Nullen initialisiert und zum Abschluss mit XOR eines konstanten Wertes versehen werden. Für den CRC-32-Algorithmus ist dies für beide Fälle 0xFFFFFFFF. Die Initialisierung ungleich Null hat den Vorteil, dass eine Anfangssequenz von Null-Bytes zu einer Veränderung der Prüfsumme führt.

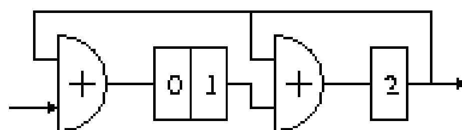


Abbildung 5: CRC-Umsetzung in Hardware für das Polynom $x^3 + x^2 + 1$

Alternativ kann die (serielle) Berechnung auch in Hardware erfolgen. Das Prinzip ist dabei das Gleiche: Je Bit wird ein taktgesteuertes Register verwendet. Der Eingang des Registers ist mit einem XOR-Operator versehen, dessen Eingänge zum einen mit dem Ausgang des Vorgängerregisters, zum anderen mit dem Ausgang des höchststelligen Registers (rückkoppelnd) verbunden ist. Verlässt ein 0-Bit das letzte Register, geschieht nichts weiter. Anders bei einem 1-Bit. Es muss das Polynom von den Registern subtrahiert werden. Im einzelnen heißt dies: Dort, wo ein Koeffizient $\neq 0$ vorhanden ist, ist der rückgekoppelte XOR-Eingang 1. Andernfalls bleibt dieser XOR-Eingang 0, das XOR-Gatter kann somit auch entfallen und der Registereingang direkt mit dem Vorgängerregisterausgang verbunden werden.

0.4 Gebräuchliche Zeichencodes

Für die Arbeit mit heutigen Computersystemen werden Zeichencodes gleicher Länge für alle Zeichen eingesetzt. Allerdings sind verschiedene Systeme im Einsatz, je nachdem, wie viele Bytes ein Zeichen umfasst. So kann es vorkommen, dass auf dem gleichen Rechner mehrere Codierungen (und damit auch Codelängen) parallel in Betrieb sind.

0.4.1 ASCII

Der erste Code, der sich auch in den heute gebräuchlichen Codierungen wiederfindet, benötigt 7 Bit und wurde 1968 entwickelt. In diesem sind neben den Buchstaben in großer und kleiner Schreibweise auch Ziffern, Interpunktion und Steuerzeichen (Zeilenumbruch etc.) enthalten. Dieser Code wird mit ASCII (für American Standard Code for Information Interchange) oder – vom Comité Consultatif International Téléphonique et Télégraphique, heute International Telecommunication Union (ITU) – auch mit CCITT-Code No. 5 bezeichnet. Die einzelnen Zeichen sind in der Abbildung 6 aufgeführt, die nicht druckbaren Steuerzeichen sind in der Abbildung 7 erläutert.

+	**0**	**16**	**32**	**48**	**64**	**80**	**96**	**112**
0	NUL	DLE	␣	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	”	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS	–	=	M	]	m	}
14	SO	RS	.	>	N	ˆ	n	˜
15	SI	US	/	?	O	_	o	DEL

Abbildung 6: ASCII (z. B. „x“ hat den Code $112 + 8 = 120$)

Übertragungssteuerzeichen		
SOH	start of heading	Kopfanfang
STX	start of text	Textanfang
ETX	end of text	Textende
EOT	end of transmission	Übertragungsende
ENQ	enquiry	Stationsanforderung
ACK	acknowledge	positive Rückmeldung
DLE	data line escape	Datenübertragungsumschaltung
NAK	negative acknowledge	negative Rückmeldung
SYN	synchronous idle	Synchronisierung
ETB	end of transmission block	Datenübertragungsende Block
Formatsteuerzeichen		
BS	backspace	Rückwärtsschritt
HT	horizontal tabulation	Zeichen-Tabulator
LF	line feed	Zeilenschritt
VT	vertical tabulation	Zeilen-Tabulator
FF	form feed	Formularvorschub
CR	carriage return	Wagenrücklauf
Code-Erweiterungszeichen		
SO	shift out	Dauerumschaltung
SI	shift in	Rückschaltung
ESC	escape	Escape
Gerätesteuerzeichen		
DC1	device control one	Gerätesteuerzeichen eins
DC2	device control two	Gerätesteuerzeichen zwei
DC3	device control three	Gerätesteuerzeichen drei
DC4	device control four	Gerätesteuerzeichen vier
Informationstrennzeichen		
US	unit separator	Teilgruppen-Trennzeichen
RS	record separator	Untergruppen-Trennzeichen
GS	group separator	Gruppen-Trennzeichen
FS	file separator	Hauptgruppen-Trennzeichen
Sonstige Steuerzeichen		
NUL	null	Null
BEL	bell	Klingel
CAN	cancel	ungültig
EM	end of medium	Aufzeichnungsende
SUB	substitute	Substitution
DEL	delete	Löschen
␣	space	Leerzeichen

Abbildung 7: Im ASCII verwendete Steuerzeichen

0.4.2 EBCDIC

Der Extended Binary-Coded-Decimal Interchange Code ist eine von IBM entwickelte Erweiterung der BCD-Zahlendarstellung, die bei den verwendeten 8 Bit viele Freiräume zwischen den einzelnen Zeichen und Kontrollsignalen z. B. für nationale Zeichensätze lässt. EBCDIC wird praktisch ausschließlich auf IBM-Großrechnern verwendet.

+	**0**	**16**	**32**	**48**	**64**	**80**	**96**	**112**	**128**	**144**	**160**	**176**	**192**	**208**	**224**	**240**
0	NUL				SP	&	–									0
1							/		a	j			A	J		1
2									b	k	s		B	K	S	2
3									c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	EOB	UC					f	o	w		F	O	W	6
7	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
8									h	q	y		H	Q	Y	8
9									i	r	z		I	R	Z	9
10			SM		¢	!	ˆ	:								
11					.	$	,	#								
12					<	*	%	@								
13					(	)	_	’								
14					+	;	>	=								
15					|	¬	?	”								¤

Abbildung 8: EBCDIC (z. B. „x“ hat den Code $160 + 7 = 167$)

NUL	all zero-bits	PF	punch off	HT	horizontal tab	LC	lower case
DEL	delete	RES	restore	NL	new line	BS	backspace
IL	idle	BYP	bypass	LF	line feed	EOB	end of block
PRE	prefix	SM	set mode	PN	punch on	RS	reader stop
UC	upper case	EOT	end of transmission	SP	space		

Abbildung 9: Kontrollzeichen des EBCDIC

0.4.3 IBM PC Code

Als 1981 die ersten Personal-Computer (PC) von IBM auf den Markt kamen, wurden Computer populärer und der Wunsch nach einer Erweiterung mit Umlauten und sonstigen, nationalen Schriftzeichen (lateinischer Buchstaben) kam auf. Eine Lösung ist, selten benötigte Zeichen z. B. „{“ in einen Umlaut umzuwandeln. Dies hat den Nachteil, dass statt der geschweiften Klammer immer ein „ä“ erscheint, auch wenn die Klammer benötigt wird.

Um nun alle Zeichen parat zu haben, wurden die 7 Bit des ASCII um ein 8. Bit ergänzt. Damit ist es dann möglich „im oberen Bereich“ Umlaute und andere Sonderzeichen unterzubringen, dargestellt in der Abbildung 10. Da aber damit noch immer nicht alle Zeichen des europäischen Sprachraumes verfügbar waren, gab es noch weitere Schriftsätze. Der Basisschriftsatz hat die Nummer 437 erhalten, Variationen z. B. die Nummer 850, bei der dann Teile der Doppellinientabellen durch Umlaute ersetzt wurden.

HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►	(SP)	0	@	P	`	p	Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q	ü	æ	í	▒	┴	╤	ß	±
-2	☻	↕	"	2	B	R	b	r	é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	Θ	•
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|	î	£	¼	╝	╠	▄	∞	^{n}
-D	♪	↔	-	=	M	]	m	}	ì	¥	¡	╜	═	▌	φ	^{2}
-E	♫	▲	.	>	N	^	n	~	Ä	Pts	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂	Å	ƒ	»	┐	╧	▀	∩	(RSP)

Abbildung 10: Zeichen des PC-ASCII, Codetabelle 437

IBM (International Business Machines Corp.) war es übrigens auch, die den Begriff „Byte“ für 8 Bits einführten. In den Normungsgremien ist bis dato nur von „Oktets“ die Rede.

0.4.4 Unicode

Neben den englischen Zeichen gibt es noch weitere Zeichensätze – nicht nur die Umlaute, sondern auch griechische Buchstaben, kyrillisch, hebräisch und arabisch, aber auch Schriftzeichen

aus dem asiatischen Raum (z. B. das indische Devanagari). Es ist offensichtlich, dass 8 Bit nicht mehr ausreichen, wenn jedes dieser Zeichen eine eigene Codierung erhalten soll.

Daher wurde der Standard Unicode (kurz: UCS, Universal Character Set) entwickelt. Die einzelnen Zeichen werden mit der Codierung U+0000 bis U+10FFFF angegeben, wobei die Kombinationen von U+D800 bis U+DFFF nicht erlaubt sind. Die direkte Darstellung in 4 Bytes wird mit UTF-32 bezeichnet, wobei UTF für Unicode Transformation Format steht und 32 die Anzahl der Bits angibt.

Man kann aber auch mit weniger als 4 Bytes alle Zeichen kodieren, indem eine variable Wortlänge und spezielle Zahlenwerte als Markierung verwendet werden. Für die Speicherung einzelner Zeichen in 16 Bit (UTF-16) werden aus der bitweisen Darstellung des Unicodes *000u uuuu aaaa aabb bbbb bbbb* die beiden Codewörter (*1101 10ww wwaa aaaa*, *1101 11bb bbbb bbbb*), wobei $wwww = uuuuu - 1$ und $uuuuu \geq 1$ gilt. Für $uuuuu = 0$ reicht ein Zeichen aus (*aaaa aabb bbbb bbbb*).

Stehen für ein Zeichen nur noch 8 Bit (1 Byte, UTF-8) zur Verfügung, verkompliziert sich die Sache: Es können 1 bis 4 Bytes benötigt werden. Die Zusammensetzung der einzelnen Bytes ist dann für den binär dargestellten Unicode *000u uuuu zzzz ayyy ybxx xxxx*

$$\rightarrow \begin{cases} \mathit{0bxx\,xxxx} \text{ für } \forall u = 0 \wedge \forall z = 0 \wedge a = 0 \wedge \forall y = 0 \\ \mathit{110y\,yyyb\ 10xx\,xxxx} \text{ für } \forall u = 0 \wedge \forall z = 0 \wedge a = 0 \wedge yyyy > 0 \\ \mathit{1110\,zzzz\ 10ay\,yyyb\ 10xx\,xxxx} \text{ für } uuuuu = 0 \wedge zzzz > 0 \\ \mathit{1111\,0uuu\ 10uu\,zzzz\ 10ay\,yyyb\ 10xx\,xxxx} \text{ für } uuuuu > 0 \end{cases}$$

Ein Beispiel ist in Abbildung 11 zu sehen. Die Aufteilung der Zeichen von U+0000 bis U+FFFF ist in Abbildung 12 dargestellt. Die Norm ist in „The Unicode Consortium. The Unicode Standard, Version 4.0.0, defined by: The Unicode Standard, Version 4.0 (Boston, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1)“ oder Online unter http://www.unicode.org/versions/Unicode4.0.0/ beschrieben.

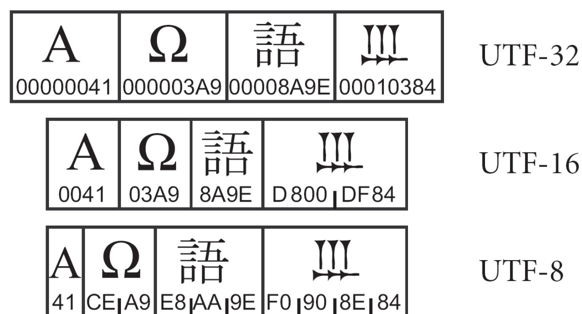


Abbildung 11: Unicode Encoding Forms

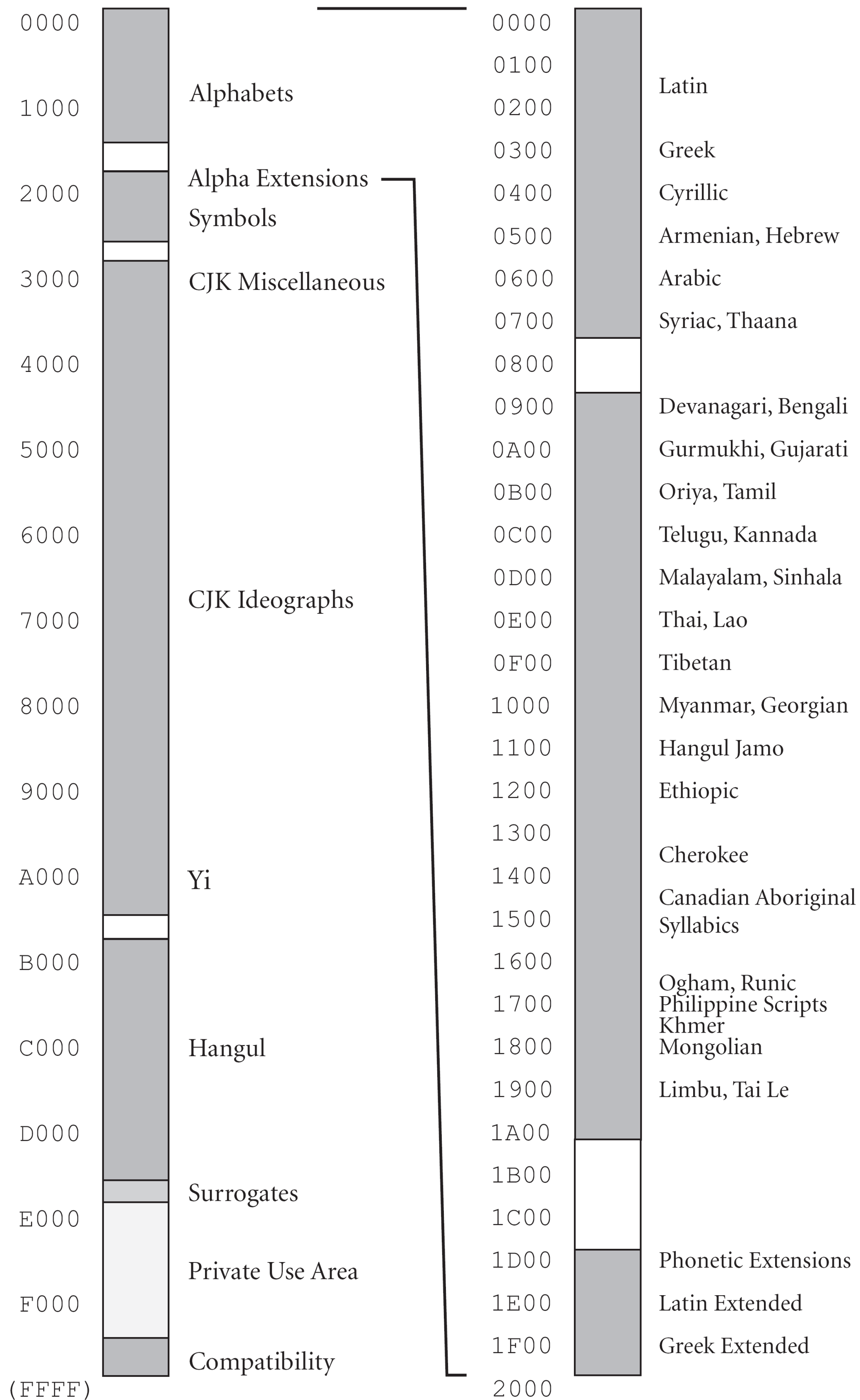


Abbildung 12: Zuordnung bei Unicode, Bereich U+0000 bis U+FFFF

Dez	Hex	Z.	Dez	Hex	Z.	Dez	Hex	Z.	Dez	Hex	Z.	Dez	Hex	Z.
32	20		33	21	!	34	22	”	35	23	#	36	24	$
37	25	%	38	26	&	39	27	’	40	28	(	41	29	)
42	2A	*	43	2B	+	44	2C	,	45	2D	–	46	2E	.
47	2F	/	48	30	0	49	31	1	50	32	2	51	33	3
52	34	4	53	35	5	54	36	6	55	37	7	56	38	8
57	39	9	58	3A	:	59	3B	;	60	3C	<	61	3D	=
62	3E	>	63	3F	?	64	40	@	65	41	A	66	42	B
67	43	C	68	44	D	69	45	E	70	46	F	71	47	G
72	48	H	73	49	I	74	4A	J	75	4B	K	76	4C	L
77	4D	M	78	4E	N	79	4F	O	80	50	P	81	51	Q
82	52	R	83	53	S	84	54	T	85	55	U	86	56	V
87	57	W	88	58	X	89	59	Y	90	5A	Z	91	5B	[
92	5C	\	93	5D	]	94	5E	ˆ	95	5F	_	96	60	‘
97	61	a	98	62	b	99	63	c	100	64	d	101	65	e
102	66	f	103	67	g	104	68	h	105	69	i	106	6A	j
107	6B	k	108	6C	l	109	6D	m	110	6E	n	111	6F	o
112	70	p	113	71	q	114	72	r	115	73	s	116	74	t
117	75	u	118	76	v	119	77	w	120	78	x	121	79	y
122	7A	z	123	7B	{	124	7C	|	125	7D	}	126	7E	˜
161	A1	¡	162	A2	¢	163	A3	£	164	A4	¤	165	A5	¥
166	A6	¦	167	A7	§	168	A8	¨	169	A9	©	170	AA	ª
171	AB	«	172	AC	¬	173	AD	SHY	174	AE	®	175	AF	¯
176	B0	°	177	B1	±	178	B2	²	179	B3	³	180	B4	´
181	B5	µ	182	B6	¶	183	B7	·	184	B8	¸	185	B9	¹
186	BA	º	187	BB	»	188	BC	¼	189	BD	½	190	BE	¾
191	BF	¿	192	C0	À	193	C1	Á	194	C2	Â	195	C3	Ã
196	C4	Ä	197	C5	Å	198	C6	Æ	199	C7	Ç	200	C8	È
201	C9	É	202	CA	Ê	203	CB	Ë	204	CC	Ì	205	CD	Í
206	CE	Î	207	CF	Ï	208	D0	Ð	209	D1	Ñ	210	D2	Ò
211	D3	Ó	212	D4	Ô	213	D5	Õ	214	D6	Ö	215	D7	×
216	D8	Ø	217	D9	Ù	218	DA	Ú	219	DB	Û	220	DC	Ü
221	DD	Ý	222	DE	Þ	223	DF	ß	224	E0	à	225	E1	á
226	E2	â	227	E3	ã	228	E4	ä	229	E5	å	230	E6	æ
231	E7	ç	232	E8	è	233	E9	é	234	EA	ê	235	EB	ë
236	EC	ì	237	ED	í	238	EE	î	239	EF	ï	240	F0	ð
241	F1	ñ	242	F2	ò	243	F3	ó	244	F4	ô	245	F5	õ
246	F6	ö	247	F7	÷	248	F8	ø	249	F9	ù	250	FA	ú
251	FB	û	252	FC	ü	253	FD	ý	254	FE	þ	255	FF	ÿ

160 A0 ␣ (nbsp, non-breakable space, nicht-unterbrechbares Leerzeichen)

Abbildung 13: Einige Zeichen im Unicode-Bereich U+0020 bis U+00FF

1 Zahlendarstellungen

In der Mathematik unterscheidet man Zahlen mit bestimmten Eigenschaften: natürliche Zahlen, ganze Zahlen, rationale Zahlen, reelle Zahlen, komplexe Zahlen usw. Eine Zahl wird als ein Element aus einer dieser Wert-Mengen definiert. Wenn wir ein bestimmtes Element angeben wollen, benutzen wir eine allgemein verständliche Darstellung. Mit „2“ meinen wir z. B. das zweite Element aus der Menge der natürlichen Zahlen. Eine Zahl, die durch Zeichen repräsentiert wird, nennen wir *konkrete Zahl*. Im täglichen Leben verwenden wir üblicherweise Dezimalzahlen, die jedermann versteht. Genauso gut hätte man eine andere Zahlendarstellung, z. B. die Oktalzahlen, standardisieren können. Meist ist die Bedeutung (Wert) einer Zahl wichtiger als ihre Darstellung. Der Wert einer konkreten Zahl kann als *abstrakte Zahl* bezeichnet werden, da er unabhängig von der Darstellung existiert. Für eine Rechenanlage ist dagegen die Darstellung von großer Bedeutung, weil die Operationen nur auf Zeichen und nicht auf Werten ausgeführt werden können. Um eine Zahlendarstellung, z. B. ein Bitmuster, richtig interpretieren zu können, muß eine Vorschrift zur Ermittlung ihres Wertes bekannt sein. Wir können formal definieren: Eine (*konkrete*) *Zahl* ist ein Paar (*Darstellung, Wert*), und es existiert eine Abbildung von der Menge der Darstellungen in die Menge der Werte (vergleiche [Bau-2, S. 59]). Man sagt auch, die Darstellung bzw. die Zahl besitzt einen bestimmten Wert. Natürlich dürfen verschiedene Darstellungen den selben Wert besitzen (z. B. 0,1 = $1/10$). In Abhängigkeit von der gewählten Abbildung kann die selbe Darstellung aber auch verschiedene Werte repräsentieren. – Im folgenden werden die für Rechenanlagen wichtigsten Zahlendarstellungen behandelt.

1.1 Zahl zur Basis b

Eine positive ganze Zahl zur Basis b (b-näre Zahl, b-adische Zahl, Stellenwertzahl, Positionssystem) wird durch eine Folge von n Ziffern X_i dargestellt, die die Werte zwischen 0 und $b-1$ annehmen können. Die Darstellung $(X_n \dots X_1)_b$ besitzt den Wert der Summe aus $X_i * b^{i-1}$. Dieser Zusammenhang wird meist durch die „Gleichung“

$$(X_n \dots X_1)_b \;=\; \sum_{i=1}^{n} X_i * b^{i-1} \tag{1.1}$$

beschrieben. Genaugenommen existiert eine bijektive Abbildung von der Menge der n-stelligen Ziffernfolgen $\{0 : b-1\}^n$ auf die Menge der positiven ganzen Zahlen $\{0 : b^n - 1\}$. Dabei sei $\{m : n\}$ eine Abkürzung für die Menge $\{m, m+1, \dots, n\}$. Die obige Beziehung können wir für gebrochene b-näre Zahlen wie folgt erweitern:

$$(X_n \dots X_1, X_0 \dots X_{-k})_b \;=\; \sum_{i=-k}^{n} X_i * b^{i-1}\ . \tag{1.2}$$

An dieser Stelle sei bemerkt, daß die Basis keine positive Zahl sein muß, sondern auch negativ oder komplex sein kann. In diesem Buch wollen wir uns nicht mit den speziellen Eigenschaften dieser und anderer Zahlensysteme auseinandersetzen, sondern uns auf die Dualzahlen (Basis $b = 2$) und die binärcodierten Dezimalzahlen ($b = 10$) konzentrieren, da sie in den Rechenanlagen am leichtesten gespeichert und verarbeitet werden können.

Für die folgenden Abschnitte müssen wir einige Vereinbarungen treffen: Durch einen kleinen Buchstaben wird der positive oder negative Wert einer darzustellenden Zahl gekennzeichnet, z. B. x. Zur Darstellung wird eine Folge von n Bits benutzt, symbolisch abgekürzt durch $X[n]$ (sprich: X mit n Stellen). Um die Darstellung aus dem Wert berechnen zu können (und umgekehrt), vereinbaren wir, daß $X[n]$ gleichzeitig den Wert der entsprechenden n-stelligen Dualzahl besitzt:

$$X[n] = X_n X_{n-1} \dots X_1 \;=\; \sum_{i=1}^{n} X_i * 2^{i-1}\ . \tag{1.3}$$

Mit der Schreibweise „$X[n]$“ ist also zum einem dieser Wert (den wir auch als *dualen Wert* bezeichnen wollen) gemeint, zum anderen die Darstellung durch eine Folge von n Bits: $X[n] = (n$ Bits, dualer Wert). Durch diese Vereinbarung kann z. B. bei Zweikomplementzahlen (Abschnitt 1.3) die Abbildung zwischen Wert und Darstellung durch eine Abbildung zwischen Wert und dualem Wert ersetzt werden. Wenn aus dem Zusammenhang hervorgeht, daß n Bits zur Darstellung benutzt werden, dann werden wir statt $X[n]$ einfach X schreiben. Um Dualzahlen von Dezimalzahlen besser unterscheiden zu können, wollen wir für die Dualziffern die Symbole „o“ und „l“ benutzen.

1.2 Vorzeichen-Betrag-Darstellung

Eine naheliegende Darstellung für positive und negative Zahlen ist die *Vorzeichen-Betrag-Darstellung*. Dabei wird die Zahl x durch ein Vorzeichenbit V_x und den Betrag $|x|$ dargestellt.

Meist wird $V_x = \circ$ für positive x-Werte und $V_x = 1$ für negative x-Werte vereinbart.

Eine positive Dualzahl $X[n]$ mit Vorzeichenbit V_x soll *Vorzeichenzahl* $(V_x, X[n])$ heißen:

$$X[n] = |x| \quad \text{und} \quad V_x = \begin{cases} \circ & \text{für} \quad x \geq 0 \\ 1 & \text{für} \quad x \leq 0 \text{ bzw. } x < 0. \end{cases} \tag{1.4}$$

$$x = (1 - 2V_x) * X[n] \quad = \quad (\overline{V_x} - V_x) * X[n] \,. \tag{1.5}$$

Bei n Stellen für den Betrag lassen sich alle Werte $|x| \leq 2^n - 1$ darstellen. Die Null kann dabei als „positive Null" ($\circ$, 0) und als „negative Null" (1, 0) dargestellt werden. Wenn man die negative Null verbietet, dann gestalten sich die Addition und Subtraktion schwieriger.

Die Vorzeichenzahl ist eine naheliegende und einfache Darstellungsform für positive und negative Zahlen. Vorzeichenzahlen sind für die Multiplikation und Division gut geeignet (Vorzeichen und Betrag werden getrennt behandelt), nicht aber für die Addition und Subtraktion. Deshalb bevorzugt man meist die Zwei- oder Einskomplement-Darstellung.

1.3 Zweikomplementzahl

Die *Zweikomplementzahl* (auch abgekürzt 2-*Komplementzahl*) ist die am häufigsten verwendete Darstellung für positive und negative Zahlen. Der duale Wert der Zweikomplementzahl $X[n]$ stimmt mit dem darzustellenden Wert x überein, wenn x positiv ist. Ist x jedoch negativ, dann wird von der Konstanten 2^n der Wert $|x|$ abgezogen, damit ein positiver Wert $X[n]$ entsteht. Wir können eine bijektive Abbildung $V2K$ definieren:

$$\begin{aligned} V2K: \quad \{-2^{n-1} : 2^{n-1} - 1\} \quad &\rightarrow \quad \{0 : 2^n - 1\} \\ x \quad &\mapsto \quad X[n] \,. \end{aligned} \tag{1.6}$$

Die Vorschrift zur Bildung des dualen Wertes $X[n]$ lautet:

$$X[n] = x \bmod 2^n = \begin{cases} x & \text{für} \quad x \geq 0 \\ 2^n + x & \text{für} \quad x < 0 \,. \end{cases} \tag{1.7}$$

Für die spätere Ableitung von arithmetischen Algorithmen in Zweikomplementdarstellung wird hauptsächlich die Rückabbildung $V2K^{-1}$ benötigt:

$$\begin{aligned} x \quad &= \quad X[n] - X_n * 2^n = \begin{cases} X[n] & \text{für} \quad X_n = \circ \\ X[n] - 2^n & \text{für} \quad X_n = 1 \,. \end{cases} \\ x \quad &= \quad X[n-1] - X_n * 2^{n-1} \end{aligned} \tag{1.8}$$

An dem Bit X_n kann man erkennen, ob der dargestellte Wert x positiv ($X_n = \mathtt{l}$) oder negativ ($X_n = \mathtt{o}$) ist. Dieses Bit wird deshalb auch als Vorzeichenbit (Sign) einer 2-Komplementzahl bezeichnet. Eine unangenehme Eigenschaft von 2-Komplementzahlen ist es, daß der Definitionsbereich unsymmetrisch zur Null liegt. Es läßt sich zwar der Wert $x = -2^{n-1}$ darstellen, nicht aber der Wert $x = +2^{n-1}$. Der Zusammenhang zwischen dem dualen Wert der 2-Komplementdarstellung $X[n]$ und dem darzustellenden Wert x läßt sich durch einen Zahlenring (Abbildung 1.1) veranschaulichen. Addieren wir z. B. auf $X[3] = \mathtt{ooo}$ mehrfach `ool`, dann durchlaufen wir den Zahlenring schrittweise im Uhrzeigersinn, und die Werte x nehmen zunächst um $+1$ zu. Beim Übergang von $X = \mathtt{oll}$ auf `loo` ändert sich aber der Wert x plötzlich von $+3$ auf -4. Dieser Übergang wird bei der Addition als Bereichsüberschreitung bezeichnet, weil der Wert $+4$ (Nachfolger von $+3$) nicht mehr mit drei Stellen dargestellt werden kann. Der Übergang von `lll` nach `ooo` ergibt sich durch Addition von `ool` auf `lll` (= `l ooo`) durch Abschneiden oder Subtraktion der höchstwertigen Stelle. Die entstandene höchstwertige Stelle wird als Übertrag (Carry) bezeichnet und meist für weiterführende Operationen gerettet.

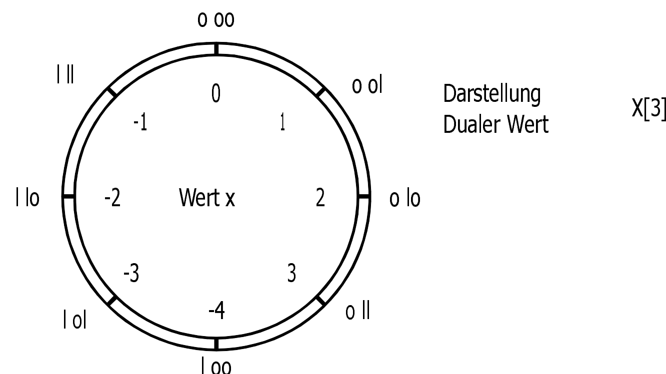


Abbildung 1.1: Zahlenring für $n = 3$ Stellen

Wie sieht eine 2-Komplementzahl aus, bei der wir den Definitionsbereich auf $\{-2^{m-1} : 2^{m-1} - 1\}$ beschränken, wobei $m \leq n$ gilt? Die 2-Komplementzahl hat dann $n - m + 1$ führende Einsen, wenn x negativ ist, und $n - m + 1$ führende Nullen, wenn x positiv ist. Wenn wir eine 2-Komplementzahl um p Stellen erweitern wollen, dann müssen wir das Vorzeichenbit X_n genau p mal links an die 2-Komplementzahl anfügen (arithmetische Erweiterung, Sign Extension).

Wir wollen uns jetzt überlegen, wie die 2-Komplementzahl $X[n]$ verändert werden muß, damit sie anstelle des Wertes x den Wert $-x$ darstellt. Die 2-Komplementzahl, die den Wert $-x$ darstellt, heiße $X'[n]$. Ist x positiv, dann ergibt sich mit Gleichung (1.7) $X[n] = |x|$ und $X'[n] = 2^n - |x|$, und daraus folgt:

$$X'[n] \quad = \quad 2^n - X[n] \,. \tag{1.9}$$

Diese Differenz nennt man *das Komplement von* $X[n]$ *gegen* 2^n. Ist x negativ, dann ergibt sich die gleiche Beziehung, die sich noch umformen läßt. Dazu definieren wir die negierte Dualzahl

$\overline{X[n]}$, bei der jedes einzelne Bit negiert ist. Es gilt dann:

$$X[n] + \overline{X[n]} = \sum_{i=1}^{n}(X_i + \overline{X_i}) * 2^{i-1} = 2^n - 1 = \texttt{1l...l} . \tag{1.10}$$

Darin bedeutet `ll...l` eine Bitkette der Länge n. Wenn wir $X[n]$ aus Gleichung (1.9) eliminieren, dann ergibt sich

$$X'[n] = \overline{X[n]} + 1 . \tag{1.11}$$

Die 2-Komplementzahl wird also komplementiert, indem alle Bits negiert werden und eine 1 hinzuaddiert wird. Die 2-Komplementzahl kann auch seriell komplementiert werden, indem man von der niedrigstwertigen Stelle ausgeht und nach links fortschreitet: Alle rechts stehenden Nullen bleiben stehen. Die erste Dualziffer, die gleich `l` ist, bleibt ebenfalls unverändert. Alle Bits, die weiter links stehen, werden negiert.

Beispiel: $X[6] = \texttt{olo loo} \longrightarrow X'[6] = \texttt{lol loo}$

Bei der Komplementierung muß man darauf achten, daß die 2-Komplementzahl `loo...o`, die den Wert -2^{n-1} repräsentiert, wegen der Zahlenbereichsüberschreitung nicht komplementiert werden darf.

1.4 Einskomplementzahl

Die *Einskomplementzahl* (Abkürzung: 1-Komplementzahl) unterscheidet sich von der 2-Komplementzahl nur dadurch, daß das Komplement gegen $2^n - 1$ gebildet wird. Wir definieren die folgende bijektive Abbildung:

$$\begin{aligned} V1K: \quad \{-2^{n-1}+1 : 2^{n-1}-1\} &\rightarrow \{0 : 2^n - 2\} \\ x &\mapsto X[n] . \end{aligned} \tag{1.12}$$

Zwischen dem darzustellenden Wert x und dem dualen Wert der 1-Komplementzahl $X[n]$ gelten die folgenden Beziehungen:

$$X[n] = x \bmod (2^n - 1) = \begin{cases} x & \text{für } x \geq 0 \\ 2^n - 1 + x & \text{für } x < 0 \end{cases} \tag{1.13}$$

$$x = X[n] - X_n * (2^n - 1) = \begin{cases} X[n] & \text{f. } X_n = \texttt{o} \\ X[n] - 2^n + 1 & \text{f. } X_n = \texttt{l} . \end{cases} \tag{1.14}$$

Auch bei dieser Darstellung repräsentiert das höchstwertige Bit X_n das Vorzeichen von x. Würden wir für $X[n]$ auch den Wert $2^n - 1$ zulassen, dann ließe sich $x = 0$ sowohl durch $X[n] = \texttt{oo...o}$ als auch durch $X[n] = \texttt{ll...l}$ darstellen.

Die 1-Komplementzahl läßt sich besonders leicht komplementieren, da sich $X[n]$ und $X'[n]$ zu $2^n - 1$ ergänzen:

$$\begin{aligned} X[n] + X'[n] &= 2^n - 1 = \mathtt{ll\ldots l} = X[n] + \overline{X[n]} \\ X'[n] &= \overline{X[n]}\,. \end{aligned} \qquad (1.15)$$

Das Einskomplement ergibt sich also durch Negation aller Bits. Im Vergleich zur 2-Komplementzahl läßt sich die 1-Komplementzahl leichter komplementieren, aber sie verhält sich bei der Addition wegen der manchmal notwendigen Korrektur um Eins ungünstiger.

Zum Abschluß wollen wir die bisherigen Zahlendarstellungen an einem Beispiel (Abb. 1.2) vergleichen. Wir nehmen an, daß 3 Bits zur Darstellung zur Verfügung stehen. Für positive Werte von x sind die drei Darstellungen äquivalent. Der Wert -4 kann als Vorzeichenzahl und als 1-Komplementzahl nicht mehr dargestellt werden.

Wert x dezimal	Vorzeichen zahl $V_x X[2]$	2K-Zahl $X[3]$	1K-Zahl $X[3]$
-4	----	l oo	----
-3	l ll	l ol	l oo
-2	l lo	l lo	l ol
-1	l ol	l ll	l lo
0	o oo (l oo)	o oo	o oo (l ll)
1	o ol	o ol	o ol
2	o lo	o lo	o lo
3	o ll	o ll	o ll

Abbildung 1.2: Definitions- und Wertebereiche für $n = 3$ Stellen

1.5 Binärcodierte Dezimalzahl

Viele Rechenanlagen rechnen intern mit Dualzahlen, weil die meisten Speicherelemente nur zwei Zustände besitzen. Außerdem lassen sich für Dualzahlen sehr einfache Rechenwerke mit logischen Bauelementen konstruieren. Wir als Menschen haben uns dagegen so stark an die Dezimalzahlen gewöhnt, daß wir nur ungern mit anderen Zahlensystem umgehen wollen. Wir verlangen deshalb, daß wir in einen Rechner Dezimalzahlen eingeben können, und daß er die Ergebnisse in der gleichen Form ausgibt. Wenn der Rechner intern mit Dualzahlen rechnet, dann müssen am Anfang und Ende der Rechnung Konvertierungen vorgenommen werden. Abgesehen davon, daß für diese Umwandlung Hardware und/oder Software benötigt werden, entstehen bei der Konvertierung von gebrochenen Zahlen Konvertierungsfehler. So kann z. B. die Dezimalzahl 0,1 nicht als Dualzahl mit endlich vielen Ziffern dargestellt werden.

Wenn wir intern mit Dualzahlen rechnen, dann können sich Fehler ergeben, die z. B. in der kommerziellen Datenverarbeitung nicht toleriert werden können. Aus diesen Gründen können viele Rechner auch intern mit Dezimalzahlen rechnen. Auch Taschenrechner rechnen intern meist mit Dezimalzahlen. Wie werden die Dezimalzahlen im Rechner gespeichert? – Jede einzelne Dezimalziffer wird durch eine Gruppe von Bits codiert. Dabei hat sich insbesondere die Codierung durch eine 4-stellige Dualzahl durchgesetzt. Wir sprechen dann von einer binärcodierten Dezimalzahl (Abkürzung: BCD-Zahl). Eine 4-Bit-Gruppe wird allgemein als Tetrade bezeichnet; die unerlaubten Codierungen 10 bis 15 werden als Pseudotetraden bezeichnet.

Beispiel: $59_{10}= \mathtt{olol\ lool}_{BCD}$

Die Darstellung von negativen BCD-Zahlen kann entweder durch Vorzeichen und Betrag erfolgen oder durch eine Darstellung im 9- oder 10-Komplement. Das 9-Komplement ist für $|x| < 4\,99\ldots9$ wie folgt definiert:

$$X_{9K} = \begin{cases} x & \text{für} \quad x \;\geq\; 0 \\ (10^m - 1) + x & \text{für} \quad x \;<\; 0\,. \end{cases} \tag{1.16}$$

Dabei besitzt sowohl x als auch X_{9K} m Dezimalstellen. Die Vorzeichenstelle, die jetzt aus 4 Bits besteht, ist 0 bis 4 für $x \geq 0$ und 5 bis 9 für $x < 0$. Die Zahl X'_{9K} sei die Zahl, die $-x$ repräsentiert. Sie ergibt sich aus X_{9K} durch Komplementbildung:

$$X'_{9K} = (10^m - 1) - X_{9K} = 99\ldots9 - X_{9K}\,. \tag{1.17}$$

Das bedeutet, daß jede einzelne Dezimalziffer dadurch komplementiert wird, daß die Differenz zu 9 gebildet wird. Das 9-Komplement einer binärcodierten Dezimalziffer kann dadurch gebildet werden, daß zu jeder BCD-Ziffer $X_{9K}\langle i\rangle$ der Wert $\mathtt{ollo}$ (+6) addiert wird und danach alle Bits negiert werden. Dabei entsteht kein Übertrag.

$$\begin{aligned} X_{9K}\langle i\rangle &= 15 - (6 + X_{9K}\langle i\rangle) \\ &= \mathtt{llll} - (\mathtt{ollo} + X_{9K}\langle i\rangle) = \overline{\mathtt{ollo} + X_{9K}\langle i\rangle}\,. \end{aligned} \tag{1.18}$$

Außer dem BCD-Code sind eine Reihe anderer Codierungen für Dezimalziffern vorgeschlagen worden, wie z. B. der Excess-3-Code oder der Aiken-Code, die beide selbst-komplementierend sind. Das bedeutet, daß das 9-Komplement einer Dezimalziffer durch Negation der Bits entsteht. Diese und andere Codes haben eine Reihe interessanter Eigenschaften [Spe], auf die hier nicht weiter eingegangen wird.

Das 10-Komplement einer darzustellenden Zahl x aus dem Bereich $-5\,00\ldots0 \leq x \leq 4\,99\ldots9$ mit m Dezimalstellen ergibt sich nach der Vorschrift

$$X_{10K} = \begin{cases} x & \text{für} \quad x \;\geq\; 0 \\ 10^m + x & \text{für} \quad x \;<\; 0\,. \end{cases} \tag{1.19}$$

Wie beim 9-Komplement ist das Vorzeichen implizit in der höchstwertigen BCD-Ziffer enthalten ($(+)$ für 0, 1, 2, 3, 4 und $(-)$ für 5, 6, 7, 8, 9). Wird der Definitionsbereich im 9-Komplement auf $|x| \leq 0,99\ldots 9 * 10^{m-1}$ und im 10-Komplement auf $-10^{m-1} \leq x \leq 0,99\ldots 9 * 10^{m-1}$ beschränkt, dann ist die erste Ziffer der m-stelligen Komplementdarstellung 0 für $x \geq 0$ und 9 für $x < 0$.

Die Zahl X'_{10K} sei die Zahl, die $-x$ repräsentiert. Man erhält sie dadurch, daß man zuerst das 9-Komplement bildet und dann $+1$ addiert:

$$\begin{aligned} X'_{10K} &= 10^m - X_{10K} = (10^m - 1) - X_{10K} + 1 \\ &= (99\ldots 9 - X_{10K}) + 1 \,. \end{aligned} \tag{1.20}$$

Es ist verboten, $X_{10K} = 500\ldots 0$ $(x = -500\ldots 0)$ zu komplementieren. Das Komplement läßt sich auch dadurch bilden, daß von rechts nach links ziffernweise vorangegangen wird. Alle rechts stehenden Nullen bleiben unverändert. Die erste Ziffer, die ungleich 0 ist, wird gegen 10 komplementiert. Die restlichen Ziffern werden gegen 9 komplementiert. Das Komplement der ersten Ziffer $\neq 0$ gegen 10 ergibt sich bei der BCD-Darstellung durch Addition von $+5 =$ `olol` und anschließender Negation.

Beispiel: X_{10K} = 509400 $\quad x = -490600$
X'_{10K} = 490600 $\quad x = +490600$

Binärcodierte Dezimalzahlen in 10-Komplement-Darstellung lassen sich nach folgendem Verfahren komplementieren $(X_{10K}, x) \rightarrow (X'_{10K}, -x)$:

1. Negiere alle Bits und addiere Eins dual $(\overline{X_{10K}} + 1)$. Merke alle Überträge zwischen den Tetraden (BCD-Stellen).

2. Subtrahiere 6 von allen BCD-Stellen, die keinen Übertrag erzeugt haben. Die Subtraktion kann auf die Addition von -6 im 2-Komplement (= `lolo`) zurückgeführt werden, wobei die Überträge zwischen den Tetraden nicht weitergeleitet werden dürfen.

1.6 Gleitkommazahl

Durch ein Codewort mit n Bits lassen sich 2^n verschiedene Zahlen repräsentieren. Wenn wir das Codewort z. B. als Vorzeichenzahl interpretieren, dann lassen sich die Werte von $-(2^{n-1} - 1)$ bis $+(2^{n-1} - 1)$ in Abständen von 1 darstellen. Dabei haben wir uns ganze Zahlen vorgestellt, mit einem fiktiven Komma, das ganz rechts steht. Wenn wir das Komma um m Stellen nach links verschieben, dann bedeutet dies eine Division durch 2^m. Die Abstände zwischen den Zahlen betragen jetzt 2^{-m}, wodurch sich näherungsweise rationale Zahlen darstellen lassen. Wird das Komma um m Stellen nach rechts verschoben, dann lassen sich noch sehr große Zahlen darstellen, allerdings nur mit dem Abstand 2^m. Wir sprechen von einer Festkommazahl, wenn wir uns das Komma an einer festen Stelle vorstellen. Verwenden wir zusätzliche Bits, die die Stellung des Kommas angeben, dann sprechen wir von einer *Gleitkommazahl* (floating

point number) oder einer halblogarithmischen Darstellung.

Der Vorteil der Gleitkommazahl gegenüber der Festkommazahl besteht nun darin, daß ein wesentlich größerer Wertebereich mit der gleichen Anzahl von Bits dargestellt werden kann, so daß bei arithmetischen Operationen nur selten die Bereichsgrenzen (Überlauf-, Unterlaufbedingung) überschritten werden. Der Programmierer braucht somit den Programmablauf weniger zu analysieren bzw. zu kontrollieren als bei Festkommazahlen. Außerdem ist die relative Darstellungsgenauigkeit in etwa konstant und unabhängig von der Größe der Zahl. Nachteile der Gleitkommazahlen sind der wesentlich höhere Verarbeitungsaufwand bei arithmetischen Operationen und die möglichen Fehler, die durch die Rundung auf die darstellbaren Werte bei der Durchführung arithmetischer Operationen entstehen. Um diese Rundungsfehler kontrollieren zu können, wurde die sogenannte *Intervallarithmetik* [Kie] eingeführt, auf die hier nicht näher eingegangen wird.

Eine Gleitkommazahl, die den Wert x darstellt, besteht aus der Mantisse mx und dem Exponenten ex zur Basis b.

$$x = mx * b^{ex} \tag{1.21}$$

Für die folgenden Betrachtungen wollen wir die praktisch immer eingehaltene Bedingung annehmen, daß b mit der Basis der Darstellung der Mantisse übereinstimmt und daß für ex nur ganze Zahlen zugelassen sind. Ohne zusätzliche Bedingungen würde für den gleichen Wert x eine Reihe von Darstellungen (mx, ex) existieren. Die Darstellung wird erst dann eindeutig, wenn der Wertebereich der Mantisse beschränkt wird. Wir definieren eine normierte (normalisierte) Darstellung:

$$x = v_x * \underbrace{mx_{\text{norm}} * b^k}_{\widehat{mx}_{\text{norm}}} * b^{ex} \tag{1.22}$$

Dabei stellt mx_{norm} die positive normalisierte Mantisse dar, v_x das Vorzeichen $(+1, -1)$ und k eine ganzzahlige Konstante. Bei der Darstellung der normalisierten Mantisse durch eine r-stellige b-näre Zahl eignet sich die Normalisierungsbedingung

$$\frac{1}{b} \leq mx_{\text{norm}} \leq 1 - \frac{1}{b^r} < 1 \tag{1.23}$$

am Besten. Damit ist das Komma der normalisierten Mantisse vor der höchstwertigen Ziffer vereinbart. Die kleinste normalisierte Mantisse wird dann durch $,100\ldots0$ und die größte durch $,(b-1)(b-1)\ldots(b-1)$ dargestellt. Die normalisierte Mantisse ist also daran zu erkennen, daß die erste Ziffer hinter dem Komma $\neq 0$ ist. Alle Darstellungen, die mit 0 hinter dem Komma beginnen, sind verboten und damit redundant. Bei der Basis $b = 2$ ist die Hälfte, bei $b = 16$ sind $1/16$ aller möglichen Darstellungen redundant. Die Konstante b^k bewertet die normalisierte Mantisse und definiert damit eine andere Kommastellung für die Mantisse $\widehat{mx}_{\text{norm}}$. Für die Realisierung von arithmetischen Operationen auf Gleitkommazahlen ist es am günstigsten, $k = 0$ zu wählen. Wenn $k = r$ gewählt wird, ist die Mantisse $\widehat{mx}_{\text{norm}}$ immer eine ganze Zahl. Wenn $k = 1$ gewählt wird (IEEE-Gleitkommaformat, siehe Abschnitt 1.6.1), dann

liegt sie zwischen 1 und 2. Für die Darstellung der Mantisse wird vorzugsweise die Vorzeichen-Betrag-Darstellung benutzt. Dabei ist das Vorzeichenbit $V_x = \mathtt{o}$ für $x \geq 0$ $(v_x = +1)$ oder $V_x = \mathtt{1}$ für $x < 0$ $(v_x = -1)$. Diese Darstellung ist insbesondere für die Multiplikation und Division von Vorteil. Aber es findet auch die Darstellung im 1-Komplement (oder allgemeiner im $(b-1)$-Komplement) Verwendung. Dadurch können Addition und Subtraktion einfach realisiert werden. Die Normalisierungsbedingung ist dann erfüllt, wenn die Vorzeichenstelle (höchstwertiges Bit) und die darauffolgende Stelle ungleich sind. Rechnet man ausschließlich mit normalisierten Mantissen im 1-Komplement, so kann das Vorzeichenbit eingespart werden, da es immer negiert zu dem darauf folgenden Bit sein muß. Die Darstellung der Null durch $mx_{\text{norm}} = 0$ ist dann aber nicht mehr möglich.

Die Zweikomplementdarstellung wird kaum verwendet, weil der Wertebereich unsymmetrisch zur Null liegt. Die Einhaltung der Normalisierungsbedingungen und die arithmetischen Algorithmen werden durch die notwendigen Sonderbehandlungen umständlich. Um die Darstellung bei negativer Mantisse eindeutig zu machen, muß entweder die Mantisse $mx = -1$ (*MX* = `1,oo...o`) oder die Mantisse $mx = -0,5$ (*MX* = `1,1oo...o`) verboten werden. Für die einfache Abwicklung der Gleitkommaoperationen ist es zweckmäßiger, $mx = -1$ zu verbieten und $mx = -0,5$ zu erlauben. Dadurch wird die obige Normalisierungsbedingung (1.23) eingehalten; die Mantisse ist dann immer betragsmäßig kleiner als 1 und der darstellbare Zahlenbereich liegt symmetrisch zur Null. Allerdings erfordert es jetzt einen höheren Aufwand, die Normalisierungsbedingungen für die Zweikomplement-Mantisse $M[n]$ abzufragen:

$$\begin{aligned} \text{NORM} = {} & (M_n M_{n-1} = \mathtt{o1}) \\ & \vee (M_n M_{n-1} = \mathtt{1o}).(M[n-2] \neq 0) \\ & \vee (M_n M_{n-1} = \mathtt{11}).(M[n-2] = 0) \,. \end{aligned} \tag{1.24}$$

Würde man dagegen $mx = 0,5$ verbieten, so wäre die Normalisierungsbedingung einfacher durch $M_n \not\equiv M_{n-1}$ abzufragen, aber dann würden die arithmetischen Algorithmen wesentlich aufwendiger werden, weil für positive Mantissen eine andere Normalisierungsbedingung als für negative Mantissen gelten würde. Um diesen Schwierigkeiten aus dem Weg zu gehen, bevorzugt man die Vorzeichen-Betrag-Darstellung.

Für den ganzzahligen Exponenten wird ein Wertebereich von $ex_{\min}$ bis $ex_{\max}$ definiert, wobei meist $ex_{\max} = -ex_{\min}$ definiert wird. Für $k = 0$ ergibt sich damit für x ein Wertebereich von

$$b^{ex_{\min}-1} \leq x \leq b^{ex_{\max}} * (1 - \frac{1}{b^r}) \,.$$

Der Abstand zwischen zwei aufeinanderfolgenden x-Werten beträgt

$$\Delta x = \frac{1}{b^r} * b^{ex} \,.$$

Wenn die niedrigstwertige Stelle der Mantisse nicht sicher ist, dann schwankt der relative Fehler zwischen

$$\frac{1}{b^r - 1} \leq \text{ relativer Fehler } \leq \frac{1}{b^{r-1}} \,.$$

Die relative Darstellungsgenauigkeit von Gleitkommazahlen ist also in etwa konstant.

Bei der Durchführung von arithmetischen Operationen kann der zulässige Wertebereich von x überschritten werden. Wird die obere Grenze überschritten, so spricht man von Überlauf (Overflow bzw. Exponentenüberlauf), wird die untere Grenze unterschritten, so spricht man von Unterlauf (Underflow bzw. Exponentenunterlauf).

Für die Darstellung des Exponenten eignet sich die 1-Komplement- oder 2-Komplementdarstellung, da sie die Addition und Subtraktion der Exponenten unterstützt, die bei der Durchführung der Gleitkommaoperationen notwendig sind. Einen besonderen Vorteil bietet die Darstellung des Exponenten als sogenannte *Charakteristik* (biased exponent). Sie ergibt sich aus der Abbildungsfunktion

$$EX\,[n] = ex + 2^{n-1}\ .$$

(Diese Darstellung unterscheidet sich von der 2-Komplementdarstellung nur dadurch, daß das Vorzeichenbit negiert ist; bei der Addition von zwei Charakteristiken muß das Vorzeichenbit anschließend negiert werden.) Der Vorteil dieser Darstellung ist, daß die Größer/Kleiner-Relation von ex auch in der Darstellung *EX* erhalten bleibt und sich somit Gleitkommazahlen leichter vergleichen lassen. Zu diesem Zweck bringt man *EX* im linken Teil eines Maschinenwortes und die normalisierte Mantisse mx_{norm} im rechten Teil unter. Dadurch lassen sich positive Werte auf einmal miteinander vergleichen. Möchte man den Vergleich auch auf negative x-Werte ausdehnen, so fügt man für $x > 0$ linksseitig ein `1`-Bit an; für $x < 0$ negiert man die Darstellung und fügt ein `o`-Bit an (Abb. 1.3).

$x > 0$:	`1`	*EX*	mx_{norm}
$x < 0$:	`o`	$\overline{EX}$	$\overline{mx_{\text{norm}}}$
$x = 0$:	`1`	0	beliebig

Abbildung 1.3: Günstige Gleitkommadarstellung

Bei der normalisierten Darstellung mit endlich vielen Stellen ist es nicht möglich, die Null direkt zu codieren. Die Null läßt sich z. B. durch $ex = -\infty$, durch $mx_{\text{norm}} = 0$ oder durch ein zusätzliches Kennbit darstellen. Folgende Darstellungen der Null sind gebräuchlich, wobei *EX* $= 0$ dem Wert $-2^{n-1} = ex_{\text{min}} - 1$ entspricht:

1. (*EX* $= 0$, mx_{norm} beliebig)
2. (*EX* beliebig, $mx_{\text{norm}} = 0$)
3. (*EX* $= 0$, $mx_{\text{norm}} = 0$)

Die erste Möglichkeit bietet den Vorteil, bei der Darstellung der normalisierten Mantissen ein Bit einzusparen, da Vorzeichenbit und 1. Stelle hinter dem Komma immer ungleich sind (1-Komplement) bzw. immer gleich bei der Darstellung nach Abbildung 1.3. Manchmal werden auch zwei Darstellungen benutzt, die erste wird dann als sehr kleiner, nicht mehr darstellbarer Wert (unechte Null) betrachtet, die zweite oder dritte als „echte“ Null.
Die Wahl der Basis wirkt sich bei gleicher Bitanzahl für *EX* und mx wie folgt aus: Durch eine größere Basis wachsen der Darstellungbereich, der Abstand zwischen zwei aufeinanderfolgenden Werten und der maximale relative Fehler exponentiell. Dafür sinkt die Wahrscheinlichkeit, das Ergebnis nach einer arithmetischen Operation normalisieren zu müssen. Werden 4 Binärstellen zu einer Gruppe zusammengefaßt, so ist $b = 16$ und zum Normalisieren ist dann ein Schift über 4 Binärstellen notwendig.

Beispiele in realisierten Anlagen, einfache Genauigkeit:

1. IBM/Siemens-Großrechner: $b = 16,\ k = 0,\ ex\,\epsilon\,\{-64 : +63\}$

	$(V_x,$	*EX*,	$mx_{\text{norm16}})$	mit $1,\ 7,\ 24$ Bits
$x > 0$:	$(\texttt{0},$	$ex + 2^6,$	$mx_{\text{norm16}})$	
$x = 0$:	$(\texttt{0},$	0,	0)	
$x < 0$:	$(\texttt{1},$	$ex + 2^6,$	$mx_{\text{norm16}})$	

   $(\texttt{0}, \texttt{101 000}, 100\,000_{16}) = 1,0$

   $(\texttt{1}, \texttt{100 000}, 200\,000_{16}) = -2/16$

   größte positive Zahl : $(\texttt{0}, \texttt{111 1111}, FFF\,FFF_{16}) \approx 7,23 * 10^{75}$

   kleinste positive Zahl: $(\texttt{0}, \texttt{000 0000}, 100\,00_{16}) \approx 5,39 * 10^{-79}$

2. DEC-32-Bit-Gleitkommaformat: $b = 2,\ k = 0,\ ex\,\epsilon\,\{-127 : +127\}$

	$(V_x,$	*EX*,	*FX*)	mit $1,\ 8,\ (1) + 23$ Bits
$x > 0$:	$(\texttt{0},$	$ex + 2^7,$	*FX*)	
$x = 0$:	$(\texttt{0},$	0,	beliebig)	
$x < 0$:	$(\texttt{1},$	$ex + 2^7,$	*FX*)	

   Das Vorzeichenbit V_x ist $\texttt{0}$ für positive Zahlen und $\texttt{1}$ für negative Zahlen. Die Basis ist 2. Die Charakteristik (biased exponent) entsteht durch Addition von 128 (bias) auf den (echten) Exponenten. Die Mantisse ist auf Beträge < 1 und $\geq 0,5$ normalisiert. Dadurch ist das Bit hinter dem Komma immer $\texttt{1}$, weswegen es nicht dargestellt wird. Die normalisierte Mantisse ohne dieses Bit heißt „fraction“ *FX*$= mx_{\text{norm2}} - 0,5$.

   Der Wert der normalisierten Gleitkommazahl ergibt sich aus der Beziehung

   $$x = (-1)^{V_x} * 2^{EX-128} * (0,5 + FX)\ .$$

   Reservierte Operanden werden durch ($\texttt{1}$, 0, *SYMBOL*) repräsentiert. Dabei wird *SYMBOL* nicht als numerischer Wert, sondern als Kontrollinformation oder Symbol interpretiert. Dadurch können ungültige Operationen und Überlaufbedingungen dargestellt werden. Ferner können Kontrollinformationen durch aufeinanderfolgende Berechnungen geschleust werden.

Zahlenbeispiele zu 2):

$$\begin{aligned} x &= +3,5 = 0,875 * 2^2 \\ &= \texttt{o,lll oooo oooo oooo oooo oooo} * 2^{130-128} \\ &= (\texttt{o},\ 130,\ 0,375 = 0,875 - 0,5) \\ &= (\texttt{o, looo oolo, llo oooo oooo oooo oooo oooo}) \end{aligned}$$

$$\begin{aligned} x &= -11,375 = -\texttt{loll,oll} * 2^0 = \texttt{o,lolloll} * 2^4 \\ &= (\texttt{l},\ 132,\ 0,2109375 = 0,7109375 - 0,5) \\ &= (\texttt{l, looo oloo, oll ollo oooo oooo oooo oooo}) \end{aligned}$$

größte positive Zahl: $(\texttt{o, llll llll, ll...l}) = 2^{127} * (1 - 2^{-24}) \approx 1,70 * 10^{38}$

kleinste positive Zahl: $(\texttt{o, oooo oool, oo...o}) = 2^{-127} * 0,5 \approx 2,93 * 10^{-39}$

1.6.1 IEEE-32-Bit-Gleitkommmaformat

Darstellung

V_x	*EX*	F_x
1	8	23

Das **Vorzeichenbit (Sign)** ist

$$V_x = \begin{cases} \texttt{l} & \text{für negative Zahlen} \\ \texttt{o} & \text{für positive Zahlen .} \end{cases}$$

Die **Charakteristik** *EX* (biased exponent, bias = 127) ensteht durch Addition von $127 = 2^{8-1} - 1$ auf den echten Exponenten ex (unbiased exponent, true exponent), siehe Tabelle Abb. 1.4:

$$\begin{aligned} EX[8] &= ex + 2^7 - 1 = ex + 127 && \text{für } ex = -126, \ldots, +127 \\ ex &= EX - 127 && \text{für } EX = 1, \ldots, 254 \end{aligned}$$

Die **Mantisse** $MX = 1,FX$ wird auf Werte zwischen 1 und 2 normalisiert, genauer $1 \leq MX \leq 2 - 2^{-23}$. Dargestellt werden nur die 23 Binärstellen hinter dem Komma (fraction *FX*), da die Stelle vor dem Komma immer gleich 1 ist.

Bemerkung: Nach der obigen Notation (Gleichung 1.22, Seite 23) gilt $MX_{IEEE} = \widehat{mx}_{\text{norm}} = mx_{\text{norm}} * 2^1$.
Der **Wert der normalisierten Gleitkommazahl** ergibt sich aus der Beziehung

$$\begin{aligned} x &= (-1)^{V_x} * 2^{EX-127} * (1, FX) \\ &= \begin{cases} +2^{EX-127} * (1, FX) & \text{für } V_x = \texttt{o} \\ -2^{EX-127} * (1, FX) & \text{für } V_x = \texttt{l} \end{cases} \end{aligned}$$

ex	EX	Interpretation
128	255 = `llll llll`	kein gültiger Exponent; zur Darstellung von ∞ und Kontrollcodes
127	254 = `llll lllo`	
126	253 = `llll llol`	
$\vdots$		gültige Exponenten
0	127 = `olll llll`	für normalisierte
-1	126 = `olll lllo`	Gleitkommazahlen
$\vdots$		
-126	1 = `oooo oool`	
-127	0 = `oooo oooo`	kein gültiger Exponent; zur Darstellung von Null bzw. Kleiner Zahl

Abbildung 1.4: Gültige und ungültige Exponenten

Betragsmäßig größter und kleinster normalisierter Wert:

$$\begin{aligned} x_{\max} &= 2^{127} * (2 - 2^{-23}) \approx 3,4 * 10^{38} \\ x_{\min} &= 2^{-126} * (1,0) \approx 1,17 * 10^{-38} \end{aligned}$$

Zahlenbeispiele:

$$\begin{aligned} x &= +3,5 = 1,75 * 2^1 \\ &= \texttt{l,llo oooo oooo oooo oooo oooo} * 2^{128-127} \\ &= (\texttt{o},\ 128,\ 0,75) \\ &= (\texttt{o, olll lll, llo oooo oooo oooo oooo oooo}) \\ x &= -11,375 = -\texttt{loll,oll} * 2^0 = \texttt{l,olloll} * 2^3 \\ &= (\texttt{o},\ 130,\ 0,375) \\ &= (\texttt{o, looo oolo, oll ollo oooo oooo oooo oooo}) \end{aligned}$$

Weitere Interpretationen der Darstellung:

a) Die Null

Darstellung der positiven Null: (0, 0, 0)
Darstellung der negativen Null: (1, 0, 0)

b) Kleine Zahl (denormalized number)

Darstellung: $(V_x, 0, \neq 0)$ Betrag: $0 < |x| < 2^{-126}$
Kleine Zahlen repräsentieren sehr kleine Werte, die kleiner als $x_{\min}$ und größer als Null sind. Der Wert ergibt sich zu
$x = (-1)^{V_x} * 2^{EX-126} * (0,FX)$.

c) Große Zahl, ∞

Darstellung: $(V_x, 255, 0)$ Betrag: $|x| > x_{\max}$
∞ kann sowohl ein positives als auch ein negatives Vorzeichen besitzen.

d) Ungültige Zahl (Not a Number, *NAN*)

Darstellung: $(V_x, 255, \neq 0)$
Ungültige Zahlen werden als Kontrollcodes interpretiert. Entweder entstehen bestimmte Kontrollcodes bei der Ausführung unzulässiger Operationen (z. B. $0*\infty$) oder sie werden dazu benutzt, Statuswerte durch eine Serie von Operationen zu schleusen.

e) Überlauf (Overflow)

Überlauf kann bei der Addition, Multiplikation und Division auftreten, wenn nach dem Runden ein Ergebnis $|x| > x_{\max}$ entsteht.

f) Unterlauf (Underflow)

Unterlauf kann bei der Addition, Multiplikation, Division und Reziprokwertbildung $(1/x)$ auftreten, wenn nach dem Runden ein Ergebnis $0 < |x| < x_{\min}$ entsteht. Als Ergebnis wird entweder eine „Kleine Zahl" erzeugt oder ± 0.

g) Besondere Operationen

Besondere Operationen können für die Verarbeitung von ∞, kleinen Zahlen und durch die Unterscheidung von positiver und negativer Null definiert werden.

h) Rundung bei einem nicht darstellbaren Rest R

Vier Rundungsarten stehen zur Auswahl:

- Rundung zur nächsten darstellbaren Zahl:

  $|x|, R \rightarrow |x| + 1$ für $R \geq 0,5$ (*MSB* von R ist 1)
  $\rightarrow |x|$ für $R < 0,5$ (*MSB* von R ist 0)

- Rundung in Richtung $+\infty$:

  $+|x|, R \rightarrow +|x| + 1$ für $R \neq 0$
  $-|x|, R \rightarrow -|x|$

- Rundung in Richtung $-\infty$:
  $+|x|, R \quad \rightarrow \quad +|x|$
  $-|x|, R \quad \rightarrow \quad -|x| - 1 \qquad$ für $R \neq 0$
- Rundung in Richtung 0
  $|x|, R \quad \rightarrow \quad |x|$

1.6.2 IEEE-64-Bit-Gleitkommmaformat

Das IEEE-64-Bit-Gleitkommaformat (Double) entspricht dem IEEE-32-Bit-Gleitkommaformat (Single) in seiner Aufbaustruktur. Für das Vorzeichenbit V_x wird 1 Bit, für die Charakteristik werden 11 Bits und für die Mantisse werden 52 Bits benutzt. Die Darstellung wird wie folgt interpretiert:

1. Falls *EX* $= 1, \ldots, 2046$, dann wird eine normalisierte Zahl mit dem Wert $x = (-1)^{V_x} * 2^{EX-1023} * (1,FX)$ dargestellt.

2. Falls *EX* $= 0$ und *FX*$= 0$, dann ist $x = (-1)^{V_x} * 0$.

3. Falls *EX* $= 0$ und *FX* $\neq 0$, dann wird eine nicht normalisierte (kleine) Zahl mit dem Wert $x = (-1)^{V_x} * 2^{EX-1022} * (0,FX)$ dargestellt.

4. Falls *EX* $= 2047$ und *FX* $= 0$, dann ist $x = (-1)^{V_x} * \infty$.

5. Falls *EX* $= 2047$ und *FX* $\neq 0$, dann wird *NAN* dargestellt.

Für noch höhere Ansprüche an die Rechengenauigkeit kann das IEEE-Gleitkommaformat auf $(1, \geq 15, \geq 63)$ Bits erweitert werden.

2 Definition der Hardware-Beschreibungssprache HDL

Ein System, das einen Algorithmus ausführt, besteht in der Regel aus einer Anzahl übereinanderliegender Ebenen, die sich hinsichtlich ihrer Operationen, ihrer Realisierung und ihrer kennzeichnenden Parameter unterscheiden. Betrachten wir die üblichen Ebenen eines Rechnersystems. Die oberste Ebene heißt *Systemebene*, die durch die Gesamtheit der realisierten Funktionen, die Verarbeitungsleistung, die Zuverlässigkeit, die Kompatibilität, die Erweiterbarkeit, den Stromverbrauch usw. gekennzeichnet ist. Sie wird durch die System- und Anwendersoftware sowie die gesamte Hardware realisiert.

Die nächste Ebene heißt *Funktionelle-Architekturebene* (auch *Verhaltensebene* oder *Algorithmische-Ebene* genannt). Sie beschreibt die Funktionsweise aus Sicht des Programmierers auf der Maschinenebene, die durch den Maschinenbefehlssatz gekennzeichnet ist, d. h. durch die dem Benutzer zugänglichen Operationen auf Datenspeichern, Ein-/Ausgabesignalen und der Programmablaufkontrolle. Weiterhin zählt zur Funktionellen-Architekturebene die Gliederung der Maschine in ihre Funktionseinheiten und deren Zusammenspiel. (Auf den Begriff „Architektur" wird im Abschnitt 5.2 genauer eingegangen.)

Die nächste Ebene in dieser Hierarchie wird als *Register-Transfer-Ebene* oder *Mikroprogrammebene* bezeichnet. Sie ist durch die Gliederung in Komponenten (wie Register, Speicher, ALU, Busse) und ihr Zusammenspiel gekennzeichnet. Das Zusammenspiel wir durch die Verdrahtung und durch einen Steuerablauf (Mikroprogramm, Mikroalgorithmus, Hardware-Steuerwerk) mit Mikrobefehlen/Mikrooperationen auf den Komponenten festgelegt. Der Steuerablauf legt die Interpretation der Maschinenbefehle und damit die Funktionelle-Architektur fest.

Die sich daran anschließende *Logik-Ebene* (auch Gatter-Ebene oder Hardware-Ebene genannt) dient schließlich zur Realisierung der Register-Transfer-Komponenten und der Mikrobefehle/Mikrooperationen; sie besteht aus den einzelnen Bauelementen wie Registern und Verknüpfungsschaltnetzen mit ihrem logischen, zeitlichen, elektrischen und physikalischen

Verhalten. Bei der Realisierung bestimmter logischer Funktionen muß auf dieser Ebene geprüft werden, ob die Voraussetzungen für die gewünschte logische Funktion, wie Zeitschranken, Flankensteilheit, Stromversorgung, Fan-out, Fan-in, Temperaturschranken usw. erfüllt sind. Unterhalb der Logikebene können weitere Ebenen identifiziert werden: die *Schaltkreisebene* stellt elementare Komponenten wie Transistoren, Dioden, Widerstände usw. zur Realisierung der Logikebene zur Verfügung; die *Physikalische Ebene* ist durch die verwendeten Materialien und räumlichen Abmessungen (z. B. Layout) der Komponenten gekennzeichnet. Die Einteilung eines Systems in diese Ebenen unterliegt einer gewissen Willkür, weil oftmals keine genügend strengen Kriterien zu ihrer Abgrenzung vorliegen. Trotzdem ist es sinnvoll, solche Ebenen zu definieren, damit Abstraktionen/Modelle eingesetzt werden können und damit beim Entwurf eine Aufgabenteilung vorgenommen werden kann.

Der Entwurf von der Systemebene zur Funktionellen-Ebene wird als *Funktioneller-Entwurf*, von der Funktionellen-Ebene zur Register-Transfer-Ebene wird als *Register-Transfer-Entwurf* und von der Register-Transfer-Ebene zur Logik-Ebene als *Logischer-Entwurf* bezeichnet. Der *Physikalische-Entwurf* erstreckt sich von der Logik-Ebene zur Physikalischen-Ebene. Auf jeder Ebene sind bestimmte Eigenschaften von besonderer Bedeutung: auf der Systemebene die Eigenschaften des gesamten Systems, auf der Funktionellen-Architektur-Ebene die Eigenschaften der Systemkomponenten, auf der Register-Transfer-Ebene die Eigenschaften der Bausteine (Operations- und Steuerwerke) und auf der Logikebene die Eigenschaften der einfachen logischen Bauelemente. Um insbesondere das funktionelle und zeitliche Verhalten sowie Modularisierungen auf den verschiedenen Ebenen beschreiben zu können, bedient man sich neben anderen Beschreibungsmitteln spezieller formaler Hardware-Beschreibungssprachen.

Eine Hardware-Beschreibungssprache dient als Beschreibungsmittel zur Kommunikation zwischen Hardware- und Software-Fachleuten, zur Spezifikation von zu erstellenden und zur Dokumentation von fertigen Systemen und zur Veranschaulichung von Hardware-Strukturen und -Algorithmen in der Lehre. Mit Hilfe eines Compilers und eines Simulators läßt sich das funktionelle und zeitliche Verhalten von digitalen Systemen überprüfen. Eine Hardware-Beschreibungssprache kann auch für die manuelle oder rechnergesteuerte Synthese von Hardware (Hardware-Compiler) oder zur Erstellung von Mikroprogrammen (Firmware) Verwendung finden.

Zur Beschreibung der Struktur von Rechnersystemen wurde die Sprache PMSL [Knu], eine Weiterentwicklung vom PMS [Bel] definiert. Für die Architekturebene werden teilweise höhere Programmiersprachen wie APL [Ive] benutzt, besser sind aber spezielle Sprachen wie ISP [Bel] geeignet. Für die Register-Transfer-Ebene und die Logikebene wurden eine Reihe von Register-Transfer-Sprachen wie CDL [Chu-1], CASSANDRE [Mer], DDL [Dul], RTS [Pil-1], ERES [Grd], HBS [Hor], PHPL [Anl], KARL [Har], CONLAN [Pil-2], REGLAN [Men], VHDL [Iee] u. a. definiert. Die meisten Sprachen konzentrieren sich auf die Beschreibung einer Ebene.

Die vom Autor definierte Sprache HDL (**H**ardware **D**escription and Microprogramming

Language) [Hof-2, Hof-3] ermöglicht es dagegen, ein digitales System auf verschiedenen Ebenen zu beschreiben, und zwar in seiner Funktion (Funktionelle-Architektur) und in seiner Implementierung durch synchrone Schaltwerke (Register-Transfer-Architektur) oder durch asynchrone Schaltwerke und Schaltnetze mit Laufzeiten (Teil der Logik-Ebene). Dadurch ist es möglich, die einzelnen Phasen des Entwurfs innerhalb eines Sprachrahmens zu beschreiben. Dieses „Mehrebenen-Konzept“ wurde auch später in der Sprache VHDL verwirklicht, die z. Zt. in der Praxis die größte Bedeutung erlangt hat. Diese Sprache wird dabei, außer für Simulationszwecke, auch für die Logik-Synthese eingesetzt. VHDL besitzt einige Mängel, wie die zu starke Orientierung an der Programmiersprache ADA, die Komplexität und teilweise die Umständlichkeit bei der Modellierung einfacher Sachverhalte. Aus diesen Gründen soll in diesem Buch die relativ einfache Hardware-Beschreibungssprache HDL benutzt werden, die auch aus didaktischen Gründen besser geeignet ist, die darzustellenden Sachverhalte leicht verständlich zu präsentieren.

Bei der Definition der Sprache HDL wurde versucht, den folgenden Anforderungen zu genügen: Eine Hardware-Beschreibungssprache sollte eine einfache, konsistente Syntax und Semantik besitzen, so daß sie leicht lesbar und erlernbar ist. Deshalb muß sie sich an allgemein akzeptierte Symbole und Konstrukte anlehnen wie z. B. die strukturierenden Anweisungen (While-Schleife, For-Schleife usw.) von höheren Programmiersprachen. Sie sollte so allgemein sein, daß mit ihr die Abläufe auf den verschiedenen Ebenen beschrieben werden können, ohne von der augenblicklichen Hardware-Technologie abhängig zu sein. Andererseits sollte die Sprache die Möglichkeit bieten, eine Isomorphie-Beziehung zwischen der Beschreibung und der Implementierung auszudrücken. Deshalb sollten die in der Hardware häufig benutzten Datenspeicher als elementare Datentypen bereitgestellt werden, und asynchrone und synchrone Zuweisungen sollten unterschieden werden. Da die Speicherelemente in Vektor- oder Matrixform vorliegen, sollten Vektoren und Matrizen die elementaren Datentypen sein, und es sollten dafür leistungsfähige Operatoren zur Verfügung stehen. Da in einem digitalen System im allgemeinen mehrere synchrone und asynchrone Vorgänge gleichzeitig ablaufen, müssen einfache Mittel zur Beschreibung der Sequentialität und der Parallelität zur Verfügung stehen. Die für synchrone Systeme notwendigen Takte müssen definierbar sein, und für asychrone Systeme müssen spezielle Operatoren vorhanden sein. Da ein digitales System im allgemeinen aus einem Netz von gekoppelten Untersystemen besteht, muß die Sprache es zulassen, modulare Strukturen zu definieren.

Im folgenden wird die Sprache HDL einführend beschrieben. Dabei wurde mehr Wert auf leichte Verständlichkeit als auf völlige Exaktheit gelegt. Dem Leser sei angeraten, parallel zum Lesen der folgenden Abschnitte die Syntaxdiagramme (Abschn. 2.18) zu verfolgen. Weitere Einzelheiten über die Sprache sind in [Hof-3] zu finden.

2.1 Struktur eines HDL-Programms

Ein simulierbares HDL-Programm besteht mindestens aus einer „Unit“, die die Funktionsweise oder Realisierung einer digitalen Funktionseinheit beschreibt. Eine Unit beginnt mit dem Schlüsselwort `unit` und endet mit dem Schlüsselwort `uend`. Auf `unit` folgt der Name der Funktionseinheit, dann folgen der Deklarationsteil und der Ausführungsteil. Der Ausführungsteil einer Unit enthält mindestens eine der drei Beschreibungsformen:

1. Permanente Anweisungen, eingeschlossen in „`perm ... pend`“
2. Asynchrone Prozesse, eingeschlossen in „`loop ... lend`“
3. Synchrone Automaten, eingeschlossen in „`on clock ... noc`“

Die 1. Beschreibungsform ermöglicht die Beschreibung von Schaltnetzen, die boolesche Verknüpfungen, feste Verbindungen und Zeitverzögerungen enthalten. Die 2. Beschreibungsform eignet sich insbesondere zur funktionellen Beschreibung von Systemen mit Hilfe zyklischer asynchroner Prozesse. Die 3. Beschreibungsform dient zur Beschreibung synchroner Automaten.

2.2 Kommentare, Marken, Namen

Ein Kommentar besteht aus einer beliebigen Zeichenfolge, die von zwei Anführungszeichen begrenzt wird. Eine Marke (label) besteht aus einer Folge von Zeichen, die Buchstaben oder Ziffern oder das Hochkommazeichen sein können. Eine Marke wird zur Kennzeichnung von asynchronen Prozessen und asynchronen Anweisungen benutzt. Ein Name besteht aus mindestens einem Buchstaben, gefolgt von Buchstaben, Ziffern oder Hochkommas. Namen dienen zur Kennzeichnung von Variablen, Programmen, Unterprogrammen und bestimmten anderen Größen.

2.3 Datenformate

Folgende Datenformate können definiert werden:

`B`	Bit
`V(b:a)`	Vektor mit n Bits, $n = \|b - a + 1\|$
`M(d:c,b:a)`	Matrix aus $m * n$ Bits, $m = \|d - c + 1\|$

Die Indizes sind positive Konstanten, die wahlweise aufsteigend oder absteigend definiert sein können, beginnend bei `0` oder `1`. Das Datenformat Matrix kann je nach Anwendung verschieden interpretiert werden, z. B. als rechteckige Matrix, bestehend aus m Zeilen und n Spalten, oder als strukturierter Vektor, bestehend aus m Bitgruppen zu je n Bits. Für die Definition der Operatoren sind die Vektorelemente aufgereiht von rechts (a) nach links (b) und die Zeilen einer

Matrix von oben (d) nach unten (c) angeordnet zu denken. Das Datenformat Vektor schließt das Datenformat Bit mit ein, das Datenformat Matrix schließt die Datenformate Vektor und Bit mit ein.

2.4 Konstanten

Ein *boolesche* Konstante ist eine Folge der booleschen Werte „o“ (false) oder „l“ (true). Das Datenformat ist Bit bzw. Vektor. Zur Darstellung der booleschen Werte werden Sonderzeichen oder die Buchstaben o und l benutzt, um diese häufig auftretenden Konstanten einfach darstellen und von Dezimalzahlen unterscheiden zu können. Ein *oktale* Konstante besteht aus einer Folge von Oktalziffern 0,1,..., 7, die von zwei $-Zeichen begrenzt wird. Eine oktale Konstante ist eine Abkürzung für eine boolesche Konstante. Ersetzt man die 0,1,..., 7 Oktalziffern durch ooo, ool,..., lll, dann ergibt sich die äquivalente boolesche Konstante. Eine *hexadezimale* Konstante besteht aus einer Folge von Hexadezimalziffern 0,1,..., F, die von zwei Hochkommas begrenzt wird. Eine hexadezimale Konstante ist eine Abkürzung für eine boolesche Konstante. Ersetzt man die Hexadezimalziffern 0,1,..., F durch oooo, oool,..., llll, dann ergibt sich die äquivalente boolesche Konstante. Eine *dezimale* Konstante besteht aus einer Folge von Dezimalziffern 0,1,..., 9. Sie ist einer booleschen Vektor-Konstanten mit k Bits äquivalent, die als Dualzahl interpretiert, dem Wert der dezimalen Konstanten entspricht. Die Größe k ist implementierungsabhängig (z. B. $k = 16, 32$). *Einfache* Konstanten besitzen das Datenformat „Vektor“ und werden als boolesche, oktale, hexadezimale oder dezimale Konstanten geschrieben. Eine *allgemeine* Konstante besitzt das Datenformat „Matrix“. Sie wird aus einer Folge von Konstanten gebildet, die durch das Trennzeichen „..“ voneinander getrennt sind. Jede Konstante definiert einen (Zeilen-)Vektor der Matrix. Die Anzahl n der Vektorelemente muß für alle Konstanten gleich groß sein.

Beispiel:

```
'01'..'25'..'F9' = oooo oool
                   oolo olol
                   llll lool
```

Wird zwischen zwei Konstanten das Trennzeichen „:“ benutzt (A:B), so wird dadurch eine Matrix definiert, deren Zeilen folgende Werte besitzen:

```
A..(A+1)..(A+2).. ... B     für A<B
A..(A-1)..(A-2).. ... B     für A>B
A                           für A=B
```

Eine *ternäre* Konstante besteht aus einer Folge von „o“, „l“ oder „@“. Sie wird zum Vergleich mit booleschen Vektoren in Ausdrücken benutzt, wobei die durch „@“ markierten Bit-Positionen nicht verglichen werden. Vergleichkonstanten sind entweder Konstanten oder ternäre Konstanten und werden im Case-Statement zum Vergleich benutzt.

2.5 Basistypen und Zuweisungen

Es werden die drei Basistypen `signal`, `boole` und `register` unterschieden. Die übrigen Datentypen lassen sich mit Hilfe der Basistypen erklären. Der Basistyp `signal` dient zur Weiterleitung der binären Werte `o` und `l`. Ein `signal` entspricht hardwaremäßig einer gerichteten Signalleitung (Drahtverbindung). Ein `signal` kann selbst keine Werte speichern. Es nimmt am Ausgang den Wert an, der am Eingang angelegt wird. Der Eingang muß direkt oder indirekt mit einer speichernden Variablen verbunden sein, damit der Wert am Ausgang nicht undefiniert ist. Die Verbindung des Wertes `Z` mit dem `signal X` wird im Programm durch `X==Z` ausgedrückt.

Der Basistyp `boole` kann den Wert wahr (`l`) oder falsch (`o`) speichern. Er entspricht einer einfachen Speicherzelle mit zwei Zuständen, wie sie z. B. durch ein RS-Flipflop realisiert wird. Der Typ `boole` wird am häufigsten in einem HDL-Programm verwendet. Er wird zur Beschreibung der im realen System konkret vorhandenen Speicher (z. B. Halbleiterspeicher) verwendet oder aber als (abstrakter) Hilfsspeicher zur Darstellung eines bestimmten Systemverhaltens. Im Programm wird die Zuweisung eines Wertes `Z` an eine Variable `X` vom Basistyp `boole` durch `X:=Z` ausgedrückt.

Der Basistyp `register` setzt sich aus zwei Basistypen `boole` zusammen, die als „Master“ und „Slave“ hintereinander geschaltet sind. Ein `register` ist ein taktgesteuertes Speicherelement, das zur Beschreibung von synchronen digitalen Systemen verwendet wird. Das Eingangssignal wird in dem Master mit der positiven Taktflanke übernommen und mit der negativen Taktflanke an den Slave weitergegeben. Der Ausgang bleibt zwischen zwei negativen Taktflanken konstant (Abb. 2.1). Ein `register` verhält sich damit wie ein zweiflankengesteuertes Flipflop (delay-skew-, data-lock-out-, double-edge-trigger-Flipflop). Durch Verändern des Tastverhältnisses des Taktes lassen sich auch andere Flipflop-Typen simulieren. Eine synchrone Registerzuweisung hat die Form:

`on` *Takt* `clock` *Register* ← *Expression* `noc` .

In Abhängigkeit von den *Takt*-Flanken erfolgt die Zuweisung des booleschen Ausdrucks *Expression* an das Register *Register*. Falls in dieser Zuweisung der Taktname *Takt* weggelassen wird, so wird ein bestimmter Standard-Simulationstakt vorausgesetzt. Einflankengesteuerte Flipflops lassen sich wie folgt beschreiben:

`on rise` *Takt* `clock` *Register* ← *Expression* `noc` .

In diesem Fall werden die Teiloperationen (`Master:=D`) und (`Q:=Master`) unmittelbar hintereinander ausgeführt werden, und zwar nach dem Auftreten der positiven Taktflanke. Durch Angabe von `on fall ... clock` werden diese Operationen nach der negativen Taktflanke ausgeführt.
Stehen mehrere Registerzuweisungen unter dem Einfluß des gleichen Takts, so werden zuerst

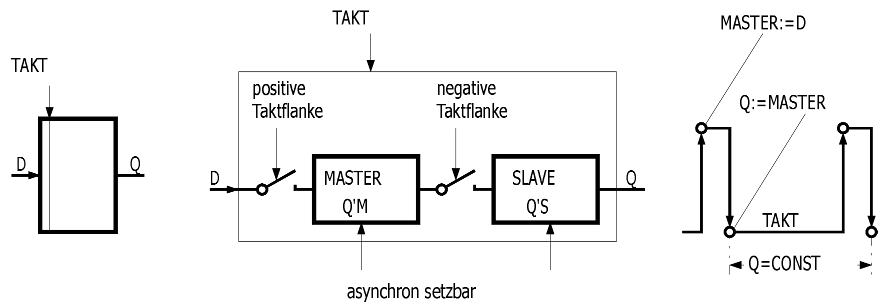


Abbildung 2.1: Verhalten eines Registers

alle Zuweisungen an die verschiedenen Master und danach alle Zuweisungen an die verschiedenen Slaves ausgeführt. Dadurch kann die synchrone Verarbeitung simuliert werden. In speziellen Anwendungsfällen möchte man auch einen direkten Zugriff auf den Master und den Slave eines Registers haben. Durch Anfügen von `'M` an den Registernamen ist der Master, und durch Anfügen von `'S` ist der Slave wie eine boolesche Variable ansprechbar. Ein Register `Q` kann z. B. durch die Anweisung `Q'S:=o` asynchron zurückgesetzt werden.

2.6 Deklaration von Variablen

Interne Variablen werden nach dem Typ-Schlüsselwort `signal`, `boole` oder `register` durch Angabe des Namens und des Formats deklariert und durch Komma voneinander getrennt.

Beispiel:

```
signal      LINE, BUS(16:1);
boole       RS'FF, LATCH(0:3), MEMORY(255:0,7:0);
register    AC(15:0), CARRY;
```

Durch diese Deklarationen werden ein Signal-Bit `LINE`, ein Signal-Vektor `BUS` zu 16 Bit, ein Boole-Bit `RS'FF`, ein Boole-Vektor `LATCH`, eine Boole-Matrix `MEMORY` mit 256 Zeilen zu je 8 Bit, ein Register `AC` zu je 16 Bit und ein Register-Bit `CARRY` erklärt.

Um die Indexgrenzen nicht immer hinschreiben zu müssen, kann die Kurzschreibweise „`[n]`“ anstelle von „`(n-1:0)`“ benutzt werden:

```
X(n-1:0) = X[n]
X(m-1:0, n-1:0) = X[m,n] = X[m][n]
```

Eine Output-Variable entspricht einer internen Variablen mit der zusätzlichen Eigenschaft, daß ihr Wert aus der Unit heraus zu anderen Units oder der Umgebung geführt wird. Es werden drei

verschiedene Output-Variablen unterschieden in Abhängigkeit davon, ob der Wert einer Signal-Variablen, einer Boole-Variablen oder einer Register-Variablen nach außen geführt wird. Die Unterscheidung erfolgt durch das auf `output` folgende Schlüsselwort.

```
output signal   S;              "= output S"
output boole    B(7:0);         "= B[8]"
output register R(7:0, 3:0);    "= R[8,4]=R[8][4]"
```

In dem Beispiel ist `S` ein Output-Signal-Bit, `B` ein Output-Boole-Vektor, und `R` eine Output-Register-Matrix. Die Zuweisung eines Wertes zu einer Output-Variablen erfolgt im Programm durch den Zuweisungsoperator, der dem spezifischen Typ entspricht.

Eine Input-Variable repräsentiert einen Wert, der von einer Output-Variablen einer anderen Unit oder der Umgebung stammt. Eine Input-Variable kann nur gelesen werden und darf deshalb nicht auf der linken Seite einer Zuweisung stehen. Input-Variablen werden nach dem Schlüsselwort `input` in gleicher Form wie Signal-Variablen deklariert.

Beispiel:

```
input RW, DIN(0:4), ADDR(0:9);
```

Durch diese Deklaration werden ein Input-Bit `RW` und die Input-Vektoren `DIN` und `ADDR` deklariert.

Eine Clock-Variable ist eine Boole-Bit-Variable, die implizit durch die Deklaration eines Taktgenerators erklärt wird. Eine Clock-Variable wechselt periodisch ihren Wert, wobei die Dauer der `o`-Phase und `l`-Phase festgelegt werden kann. Die Dauer wird in ganzzahligen Vielfachen von folgenden Zeiteinheiten angegeben: Sekunden (`sc`), Millisekunden (`ms`), Mikrosekunden (`us`), Nanosekunden (`ns`) und Picosekunden (`ps`).

Beispiel:

```
clock C1(l=100 ns, o=200 ns), C2(o=50 ns, l=250 ns);
```

Durch diese Deklaration werden zwei Taktgeneratoren erklärt, mit den Clock-Variablen `C1` und `C2`. `C1` ist `l` für 100 Nanosekunden und `o` für 200 ns, `C2` ist `o` für 50 ns und `l` für 250 ns. Während der Simulation starten alle Taktgeneratoren gleichzeitig mit ihren zuerst definierten Phasen. Ein Taktgenerator ist eine Abkürzung für einen bestimmten asynchronen Prozess (siehe Abschnitt 2.11). Der obige Taktgenerator für `C1` entspricht dem folgenden asynchronen Prozeß:

```
boole C1;
loop CLOCK: C1:=l; wait 100 ns; C1:=o; wait 200 ns lend
```

In synchronen Automaten (siehe Abschnitt 2.12) wird jeder synchrone Zustand durch einen in eckige Klammern eingeschlossenen Namen gekennzeichnet. Für jeden Namen wird implizit

ein Register-Bit deklariert; z. B. erfolgt durch die Zustandsfolge

```
[Z0]...[Z1]...[Z2]...
```

implizit die Deklaration `register Z0,Z1,Z2`. Auf die so implizit deklarierten State-Register-Variablen kann wie auf Register-Variablen zugegriffen werden.

2.7 Variablenzugriff, Indizierung

Ein bestimmtes Bit eines Vektors wird durch Angabe des in runden Klammern eingeschlossenen Index ausgewählt. Der Index kann ein beliebiger Ausdruck sein. Besitzt der Ausdruck das Datenformat Vektor, so wird er als Dualzahl interpretiert, dessen Wert den Index definiert. Besitzt der Ausdruck das Datenformat Matrix, so wird dadurch eine Folge von Indizes definiert, die durch die einzelnen Zeilen, interpretiert als Dualzahlen, festgelegt ist.

Beispiel:

`boole V(1:8);`

Aufruf	Äquivalente Schreibweise	
`V(2)`	V_2	2. Bit
`V(1:4)`	$V_{1:4}$	Die ersten 4 Bits
`V(1..3..5)`	$V_{1..3..5}$	Teilvektor, bestehend aus den Bits V_1,V_3,V_5
`V(1..1..2..2)`	$V_{1..1..2..2}$	Teilvektor, bestehend aus den Bits V_1,V_1,V_2,V_2

Wie das letzte Beispiel zeigt, dürfen die gleichen Indizes mehrmals auftreten, allerdings nur im Lesezugriff. Mehrfacher Schreibzugriff auf das gleiche Bit ist verboten.

Die Indizierung einer Matrix erfolgt durch einen Index für die Spalte(n) und einen Index für die Zeile(n). Wird Zeilen- bzw. Spaltenindex weggelassen, so sind alle Zeichen bzw. Spalten gemeint.

Beispiel:

`boole M(0:255,15:0);`

Aufruf	`M(2,) = M(2)`	M^2	2. Zeile (Wort)
	`M(2,7)`	M^2_7	Bit (2,7)
	`M(,7)`	M_7	7. Spalte
	`M(0:127,7:0)`	$M^{0:127}_{7:0}$	Teilmatrix, definiert durch die Zeilen 0 – 127 und die Spalten 7 – 0

2.8 Operatoren

2.8.1 Vektoroperatoren

Vektoroperatoren erwarten das Datenformat Vektor oder Bit. Sind einer oder beide Operanden Matrizen, dann werden sie vor Ausführung der Operation in Vektoren überführt, indem die Zeilen von oben nach unten hergenommen werden und von rechts nach links fortschreitend aneinandergefügt werden. Für die Definitionen (Abb. 2-2, Abb. 2-3) der Operatoren gilt `A(n:1), B(p:1), C(r:1).` Wenn bei dyadischen Operatoren die Vektoren ungleich lang sind, dann definiert der links stehende Vektor `A` die Länge des Ergebnisses. Der Vektor `B` wird vor der Operation an die Länge von `A` angepaßt, indem entweder die überzähligen höherwertigen Bits abgeschnitten werden oder `o...o` ergänzt wird.

Operation C=	Erklärung	r
-A	Vorzeichenwechsel, Zweikomplement ~A+1	n
A+B	Addition (A+B) mod 2^{n}	n
A++B	Addition mit Übertrag (A+B) mod 2^{n+1}	n+1
A-B	Subtraktion (A+(-B)) mod 2^{n}	n
A**B	Multiplikation C = Produkt positiver Zahlen	n+p
A*B	Eingeschränkte Multiplikation (A**B) mod 2^{n}	n
A/B	Division, B≠0: C = ganzzahliger Quotient	n
A mod B	Modulofunktion, B≠0: A-(A/B)*B	n
A%B	Relationen, % ∈ {<, ≤, >, ≥, =, ≠} A, B: positive Dualzahlen: C=A%B	1
A<=>B	Identität if (Datenformat und Werte gleich) then C=l else C=o	1

Abbildung 2.2: Vektoroperationen

Operation C=	Erklärung	r
K shr A	Shift Right C = A um K Stellen nach rechts verschoben, wobei o-Bits nachge- zogen werden. Wenn K fehlt, dann gilt K=1.	n
K shl A	Shift Left wie "Shift Right", nur nach links.	n
B inshr A	Insert Shift Right, p<n: C = B_A(n : 1+p)	n
B inshl A	Insert Shift Left, p<n: C = A(n-p : 1)_B	n
K cir A	Circular Shift Right C = A um K Stellen zyklisch nach rechts verschoben. Wenn K fehlt, dann gilt K=1.	n
K cil A	Circular Shift Left wie "Circular Shift Right", nur nach links.	n
B mux A	Multiplexoperator, $p=2^n*k$: $C = B\big((A+1)*k : (A*k + 1)\big)$	k
B dux A	Demultiplexoperator $C\big((A+1)*p : (A*p + 1)\big) = B$ Alle sonstigen Bits von C werden gleich o gesetzt.	$p*2^n$
A:B	Indexgenerator C = A..(A+1)..(A+2).. .. ..B für A<B C = A..(A-1)..(A-2).. .. ..B für A>B C = A für A=B	r=n s = $\lvert B-A \rvert$ +1

Abbildung 2.3: Vektoroperatoren

Beispiele:

```
-oolo = lllo                olol + lo = olll

llll + ol = loooo           oool - lo = llll

loo ** lll = ollloo         oolo * lll = lllo

ollloo / loo = ooolll
```

```
llol mod loo = oool          lllo > lo = l

lool <=> olool = o           shr lloo = ollo

2 shr lloo = ooll            2 shl ooll = lloo

ll inshr ooolll = lloool     ll inshl ooolll = olllll

2 cir oolo = looo            2 cil oolo = looo

oo ol lo ll mux lo = ol      ol dux lo = oo ol oo oo

3:1 = oooo oooo oooo oool
      oooo oooo oooo oolo
      oooo oooo oooo ooll
```

2.8.2 Matrixoperatoren

Matrixoperatoren können auf Matrizen, Vektoren und Bits angewandt werden. Für die Definitionen (Abb. 2-4) gelten die Datenformate `A(m:1,n:1)`, `B(q:1,p:1)`, `C(s:1,r:1)`.

Operation C=	Erklärung	s,r
A_B	Zusammenfügen horizontal m=q: B wird rechts an A angefügt.	m,n+p
A..B	Zusammenfügen vertikal n=p: B wird an A unten angefügt.	m+q,n
tra A	Transposition $C^{i}_{j} = A^{j}_{i}$, Abkürzung ^{T}A	n,m
~A	Negation $C^{i}_{j} = \sim A^{i}_{j}$ für alle (i,j)	m,n
A % B	Boolesche Operationen % ∈ {.,v,≡,≢} (Und, Oder, Äquivalenz, Disvalenz)	
	n=p, m=q : $C^{i}_{j} = A^{i}_{j} \;\%\; B^{i}_{j}$	m,n
	n=1, m=1 : $C^{i}_{j} = A^{1}_{1} \;\%\; B^{i}_{j}$	q,p
	p=1, q=1 : $C^{i}_{j} = A^{i}_{j} \;\%\; B^{1}_{1}$	m,n
%/A	Reduktion % ∈ {.,v,≡,≢} $C = \{A_1 \;\%\; A_2 \;\%\; \ldots \;\%\; A_n\}$	m,1
A %* B	Logisches Matrixprodukt %* ∈ {.v , v. , ≡. , ≢v} $C^{i}_{j} = \%/(A^{i} * \mathrm{tra}\ B_{j})$	m,p

Abbildung 2.4: Matrixoperationen

Beispiele:

```
oo_ll = ooll          oo..ll = oo
                               ll

tra oo = ool          ~lllo = oool
    ol   olo
    lo

lool v lloo = llol  lool.l = lool

v/ lool = l           ./ lool = o

ol v. ll = ol
lo    ol   ll
```

2.9 Ausdrücke, Formatanpassung

Ein Ausdruck besteht aus Konstanten, Variablen und Funktionsaufrufen, die durch monadische oder dyadische Operatoren miteinander verknüpft werden. Zuerst werden die monadischen Operatoren ausgeführt. Danach werden die dyadischen Operatoren von links nach rechts fortschreitend ausgeführt. Die dyadischen Operatoren haben keine fest vorgegebene Priorität, aber es kann zwischen zwei Prioritätsstufen gewählt werden: Wenn der zwischen den Operanden stehende Operator nicht durch Blanks eingegrenzt ist, dann wird diese Operation vor jener Operation ausgeführt, bei der der Operator durch Blanks eingegrenzt ist. Weiterhin wird ein in runde Klammern eingeschlossener Ausdruck vorrangig behandelt. Das Ergebnis der Abarbeitung eines Ausdrucks ist ein Wert mit einem bestimmten Datenformat.

Beispiele:

```
4 + 2*3   = 10     '3' * lll-1+(8/2) = '6'
                         ^^^^^^
                         =llo
                         ^^^^^^^^^^^^^^
4+2*3     = 18             = olo
```

Der Wert `B`, der sich aus der Abarbeitung eines Ausdrucks ergibt, kann mit Hilfe eines Zuweisungsoperators (`:=`, `==`, `<-`) an eine Variable `A` übergeben werden. Wenn das Datenformat des Wertes nicht mit dem Datenformat der Variablen übereinstimmt, wird der Wert an das Datenformat der Variablen angepaßt. Man kann sich die Anpassung veranschaulichen, indem man sich die beiden Matrizen `A` und `B` als Flächen vorstellt, die man so übereinander legt, daß die rechten oberen Ecken übereinstimmen. Die Zuweisung erfolgt dann für die übereinanderliegenden Bits. Besitzt die Matrix `A` mehr Zeilen oder Spalten als die Matrix `B`, so werden die dadurch bestimmten Bits auf `o` gesetzt.

Beispiele:

```
A(4:1)    := 1      =>  A=oool
A(3:0)    := 'F'    =>  A=llll
C(1:4)    := 10     =>  C=oolo
D(0:3,1:4):= 1:3    =>  D=oool..oolo..ooll..oooo
```

2.10 Permanente Anweisungen

Feste Verbindungen und boolesche Schaltnetze werden durch *permanente Anweisungen* beschrieben, die durch Kommas getrennt und in „`perm`" und „`pend`" eingeschlossen werden. Permanente Anweisungen sind *permanente Zuweisungen*, *verzögerte Zuweisungen*, *permanente If-Statements* oder *permanente Case-Statements*. Die permanente Zuweisung dient zur

Übergabe von Werten an Signal-Variablen ohne Zeitverzögerung. Durch die permanente Zuweisung `Y==`*Ausdruck* besitzt die Signal-Variable `Y` immer den aktuellen Wert des *Ausdruck*s. Wenn sich der Wert des Ausdrucks ändert, dann ändert sich unmittelbar auch der Wert von `Y`. Mehrere permanente Anweisungen werden durch Kommas voneinander getrennt. Permanente Zuweisungen entsprechen im wesentlichen booleschen Gleichungen und dienen zur Beschreibung von Schaltnetzen. Die Reihenfolge der einzelnen permanenten Anweisungen im Programm impliziert keine entsprechende Abarbeitungsfolge. Alle permanenten Anweisungen stehen also – semantisch gesehen – gleichberechtigt nebeneinander. Im Programm dürfen mehrere permanente Zuweisungen an die gleiche Signalvariable stehen. In diesem Fall werden alle Werte durch die Oder-Funktion miteinander verknüpft: z. B. `Y==X1, Y==X2, Y==X3` entspricht `Y==X1 v X2 v X3`.

Die *verzögerte Zuweisung* ergibt sich aus der permanenten Zuweisung durch den Zusatz `delay` *time*. Als Zeiteinheiten sind `sc` (Sekunden), `ms` (Millisekunden), `us` (Mikrosekunden), `ns` (Nanosekunden), `ps` (Picosekunden) zulässig. (Beispiel: `X==delay 100 ns Y`). Die Verzögerung erfolgt dadurch, daß alle Wertänderungen auf der rechten Seite in einem Pufferspeicher für die Dauer der Verzögerungszeit zwischengespeichert werden. Verzögerte Zuweisungen dürfen nicht in einem permanenten If- oder Case-Statement stehen.

Mit Hilfe des *permanenten If-Statements* kann in Abhängigkeit von einer Bedingung eine Signal-Variable einen von zwei Werten annehmen. Hardwaremäßig entspricht das permanente If-Statement einem gesteuerten Schalter oder einem 2:1-Multiplexer. Das permanente If-Statement „`if X then Y==A else Y==B fi`“ ist gleichbedeutend mit `Y==A.X v B.~X`. Die Bedingung X muß das Datenformat Bit besitzen. Der `else`-Teil kann weggelassen werden. Im `then`- oder `else`-Teil darf im allgemeinen eine Liste von permanenten Ereignissen stehen. Ein permanentes Ereignis ist eine permanente Anweisung, ein permanentes If-Statement oder ein permanentes Case-Statement.

Das *permanente Case-Statement* ist eine Verallgemeinerung des permanenten If-Statements. In Abhängigkeit von 2^n verschiedenen Werten, die ein Bedingungsvektor der Länge n annehmen, können maximal 2^n verschiedene permanente Ereignisse ausgewählt werden. Es gibt zwei Formen des Case-Statements, das „`case ... of`“ und das „`case ... then`“. Das permanente Case-Then-Statement

```
case C then
?C0: Y == A0
?C1: Y == A1 ...
?Cn: Y == An
else Y == Ae esac
```

entspricht

```
Y == A0.K0 v A1.K1 v ... An.Kn
     v Ae. K0. K1 ...  Kn          (*) .
```

unter Verwendung der Abkürzung `Ki = (C=Ci)`. Fehlt in dem obigen Case-Statement der `else`-Teil, dann entfällt der durch `(*)` gekennzeichnete Teil. Fehlt in einer Zeile die permanente Zuweisung `Y==Ai`, dann entfällt der Term `Ai.Ki`. Die Bedingungen `Ki` müssen sich nicht gegenseitig ausschließen, so daß alle `Ai` durch Oder miteinander verknüpft werden, für die `(C=Ci)=1` ist. `Ci` kann eine Liste von Vergleichskonstanten `Ci0,Ci1,...,` `Cim` sein. Es gilt dann `Ki = (C = Ci0)v(C = Ci1) v...(C = Cim)`. Anstelle der permanenten Anweisungen `Y==Ai` darf allgemein eine Liste von permanenten Ereignissen stehen.

Mit Hilfe des permanenten Case-Then-Statements lassen sich z. B. UND/ODER-Matrizen (PLA, programmable logical array) oder bedingte Mehrfach-Durchschaltungen beschreiben. Das permanente Case-Of-Statement unterscheidet sich vom permanenten Case-Then-Statement dadurch, daß sich alle Bedingungen `Ki` gegenseitig ausschließen müssen; d. h. es darf höchstens eine erfüllte Bedingung geben, wodurch auch nur höchstens eine Liste von permanenten Ereignissen ausgeführt wird. Das Case-Of-Statement eignet sich z. B. zur Beschreibung von Multiplexern, Demultiplexern und kleinen Festwertspeichern.

2.11 Asynchrone Prozesse

Ein *asynchroner Prozeß* besteht aus einer Folge von *asynchronen Statements*, die zyklisch wiederholt werden. Die asynchronen Statements des Prozesses werden durch Semikolon voneinander getrennt und durch „`loop`“ und „`lend`“ begrenzt. Die durch Semikolon getrennten Statements werden nacheinander ausgeführt. Ein asynchroner Prozeß kann sequentielle und kollaterale Unterprozesse enthalten, die nur aus einfachen asynchronen Statements bestehen dürfen. Einfache asynchrone Statements dürfen nicht durch Sprungmarken gekennzeichnet sein und keine Kontrolloperationen (`goto, wait, start, stop`) enthalten. Sind im Ausführungsteil mehrere asynchrone Prozesse vorhanden, so müssen sie durch Marken (process-label) gekennzeichnet werden. Ist nur ein einziger Prozeß vorhanden, so braucht er nicht gekennzeichnet zu werden. Jeder asynchrone Prozeß beschreibt eine Hardware- oder Software-Aktivität. In einer Unit werden im allgemeinen mehrere gleichzeitig ablaufende Aktivitäten beschrieben. Der Ablauf und die Synchronisation von Prozessen kann durch die Kontroll-Operationen gesteuert werden.

Asynchrone Prozesse enthalten hauptsächlich asynchrone Zuweisungen „`:=`“ an Boole-Variablen. Daneben können simultane Zuweisungen der Form `[A:=W1„B:=W2„C :=W3„...]` ausgeführt werden, indem zuerst alle Werte `W1,W2,W3, ...` berechnet, dann zwischengespeichert und anschließend an die Variablen `A,B,C, ...` übergeben werden. Dadurch lassen sich Register-Variablen simulieren.

Das *asynchrone If-Statement* ermöglicht die durch eine Bedingung gesteuerte alternative Ausführung zweier Folgen von asynchronen Statements. Wenn die Bedingung erfüllt ist, werden die auf `then` folgenden Statements ausgeführt, ansonsten die auf `else` folgenden

Statements. Der `else`-Teil kann entfallen. Die auf `then` oder `else` folgenden Statements können beliebige asynchrone Statements sein.

Das *asynchrone Case-Statement* ermöglicht die bedingte Ausführung von maximal 2^n alternativen Statements S_i in Abhängigkeit von einem Bedingungsvektor der Länge n. Das asynchrone Case-Statement

```
case C then      "entspricht mit Ki = (C=Ci)"
?C0: S0          if K0 then S0 fi;
?C1: S1          if K1 then S1 fi;
...              ...
?Cn: Sn          if Kn then Sn fi;
else Se          if  K0. K1... Kn then Se fi .
```

Der `else`-Teil kann entfallen. Wenn mehrere Bedingungen `Ki` erfüllt sind, so werden alle zugehörigen asynchronen Statements `Si` in der durch die Übersetzung vorgegebenen Reihenfolge ausgeführt. `Ci` kann eine Liste von Vergleichskonstanten sein (vergl. permanentes Case-Statement). Das asynchrone `case C of` unterscheidet sich vom `case C then` dadurch, daß die Bedingungen `Ki` sich gegenseitig ausschließen müssen.

Das *asynchrone While-* und *Until-Statement* dient zur Wiederholung einer Folge von asynchronen Statements `S`. Beim While-Statement wird die Bedingung `C` vor der Ausführung von `S` abgefragt, während sie beim Until-Statement nach der Ausführung abgefragt wird. Beim Until-Statement wird `S` also mindestens einmal ausgeführt:

```
             "entspricht"
while C do  L: if C then goto K fi;
S              S;
od             goto L;
            K: ...

             "entspricht"
until C do  L : S;
S           if C then goto K else goto L fi;
od          K: ...
```

Das *asynchrone For-Statement* dient zur wiederholten Ausführung einer Folge von asynchronen Statements `S(k)`, wobei ein Laufindex `k` zwischen zwei Grenzen `R` und `T` herauf- bzw. heruntergezählt wird:

```
                        "entspricht"
for k := R to T do  k := R; t := T;
                    if k<t then d:= 1 fi;
                    if k>t then d:=-1 fi;
S(k)                L: S(k); if k=t then goto M fi;
                    k:=k+d;
od                  goto L;
                    M: ...
```

Die Laufvariable `k` muß vom Datentyp `boole` sein und das Datenformat Vektor besitzen. Bei der Übersetzung werden zwei Hilfsvariablen `t` und `d` erzeugt, die vom gleichen Datentyp und Datenformat sind. Zu Beginn werden die Ausdrücke `R` und `T` berechnet und die Werte den Variablen `k` und `t` zugewiesen. Dann wird das Inkrement `d` ermittelt und anschließend wird die Schleife durchlaufen.

Einfache asynchrone If-, Case-, While- und For-Statements sind dadurch definiert, daß sie nur einfache asynchrone Statements ohne Kontrolloperationen enthalten.

Die *Kontroll-Operationen* dienen zum Verzweigen (`goto`), zum Warten (`wait`) und zur Synchronisation innerhalb von Prozessen. Sie dürfen nicht innerhalb von sequentiellen oder kollateralen Unterprozessen verwendet werden. Die Sprunganweisung „`goto LABEL`“ ermöglicht einen Sprung zu dem durch „`LABEL:`“ gekennzeichneten Statement. Durch die Anweisung „`wait until C;`“ wird die Ausführung des Prozesses solange unterbrochen, bis die Bedingung `C` erfüllt ist. Die Anweisung „`wait until rise C;`“ entspricht „`wait until ~C; wait until C;`“. Die Anweisung „`wait until fall C;`“ ist äquivalent zu „`wait until C; wait until ~C;`“. Durch die Anweisung „`wait N us`“ wird die Ausführung des Prozesses solange unterbrochen, bis die simulierte Zeit um `N` Mikrosekunden fortgeschritten ist. Anstelle von `us` können auch die Zeiteinheiten `sc`

(Sekunden), `ms` (Millisekunden), `ns` (Nanosekunden) und `ps` (Picosekunden) benutzt werden. Mit Hilfe der `start`- und `stop`-Anweisung ist es möglich, die zyklischen asynchronen Prozesse zu synchronisieren.

Ein *sequentieller Unterprozeß* besteht aus einer in Klammern eingeschlossenen Folge von einfachen asynchronen Statements, die keine Kontroll-Operationen und nicht durch Marken gekennzeichnet sein dürfen. Ein asynchroner Unterprozeß ist eine logische Zusammenfassung einer Folge von einfachen asynchronen Statements. Asynchrone Unterprozesse werden hauptsächlich innerhalb von kollateralen Unterprozessen verwendet.

Ein *kollateraler Unterprozeß* wird durch in eckige Klammern eingeschlossene einfache asynchrone Statements definiert: `[C1,C2,..., Cn]`. Diese Schreibweise besagt, daß die asynchronen Statements `C1` bis `Cn` gleichzeitig ausgeführt werden *können*, aber nicht müssen; d. h. sie können auch in beliebiger Reihenfolge nacheinander ausgeführt werden, ohne daß eine andere Wirkung erzielt wird. Sie dürfen daher weder direkt noch indirekt (z. B. über Signal-Variablen) dieselben Variablen mehrfach schreiben oder schreiben und lesen. Durch einen kollateralen Unterprozeß mit seinen kollateralen Statements kann also die Möglichkeit oder der Wunsch nach gleichzeitiger Ausführung ausgedrückt werden. Die kollateralen Statements können wieder sequentielle Unterprozesse sein, d. h. kollaterale und sequentielle Unterprozesse können ineinander geschachtelt werden.

2.12 Synchrone Automaten

Ein *synchroner Automat* besteht aus einer Folge von Zuständen, die durch einen Takt gesteuerte Registerzuweisungen und Zustandsübergänge enthalten.

```
on T clock
     Zustandsaktionen-0
[Z1] Zustandsaktionen-1
[Z2] Zustandsaktionen-2
...
[Zn] Zustandsaktionen-n .
noc
```

Der Takt `T` ist eine beliebige Variable mit dem Datenformat Bit. Wenn kein Takt angegeben wird, dann wird ein Standard-Takt angenommen, der während der Simulationsphase definiert werden kann. Die Zustandsfolge besteht aus den Zustandsaktionen-0, die auch entfallen können, und aus n>0 Zuständen. Jeder Zustand wird durch ein Zustandsregisterbit `Zi` gekennzeichnet. Die Zustandsregisterbits sind vom Basistyp `register`. Zu den Zustandsaktionen-0 kann man sich ein Zustandsregister `Z0` zugeordnet vorstellen, das immer auf `1` gesetzt ist. Deshalb werden die Zustandsaktionen-0 immer (erneut mit jedem Takt) ausgeführt. Die Zustandsaktionen-i>0 werden nur dann ausgeführt, wenn sich der Automat im Zustand

`Zi` befindet, d. h. wenn die zugehörige Zustandsvariable `Zi=1` ist. Der Automat darf sich gleichzeitig in mehreren Zuständen befinden; bei den meisten Automatenbeschreibungen befindet sich der Automat jedoch immer nur in einem Zustand.

Eine *Zustandsaktion* ist eine permanente Zuweisung, ein Register-Transfer, eine synchrone Sprunganweisung (`next`), eine asynchrone Sequenz, eine If-Zustandsaktion oder eine Case-Zustandsaktion.

Wenn sich unter den Zustandsaktionen-0 permanente Zuweisungen befinden, dann werden sie genauso wie permanente Statements interpretiert; stehen permanente Zuweisungen innerhalb von Zustandsaktionen-i>0, dann erfolgt eine Übersetzung in das permanente If-Statement mit `Zi` als Bedingung.

Beispiel:

```
[Z1] Y == W  "entspricht"  if Z1 then Y == W fi .
```

Bei einem Registertransfer `R<-D` wird der Wert `D` unmittelbar nach Auftreten der positiven Taktflanke in den Master `R'M` übernommen und unmittelbar nach der negativen Flanke an den Slave `R'S` weitergegeben; der Wert `R` am Ausgang des Registers ist gleich dem Wert von `R'S`. Der obige Registertransfer ist äquivalent zu dem asynchronen Prozeß

```
loop wait until T;   R'M:=D; wait until ~T; R'S := R'M lend.
```

Anstelle des Taktes `T` kann auch der negierte Takt `~T`, die ansteigende Flanke des Taktes „`rise T`“ oder die abfallende Flanke „`fall T`“ benutzt werden. Die Angabe „`rise T`“ bedeutet, daß nach der ansteigenden Taktflanke zuerst alle Master-Zuweisungen und unmittelbar danach alle Slave-Zuweisungen erfolgen.

Die *synchrone Sprunganweisung* `next` ermöglicht mit dem Takt synchronisierte Zustandsübergänge. Ein Zustandsübergang erfolgt wie ein Registertransfer nach dem Master-Slave-Prinzip. Der Zustandsübergang `[Zi] next Zj` ist dem Registertransfer `Zi<-o`, `Zj<-1` an die implizit erklärten Registerbits äquivalent.

Eine *asynchrone Sequenz* ist eine Folge von einfachen asynchronen Statements, die nach dem Eintritt in einen Zustand durchlaufen werden soll. Sie wird in die Schlüsselwörter „`asyn ... aend`“ eingeschlossen.

Die *If-Zustandsaktion* ermöglicht die bedingte Ausführung zweier alternativer Zustandsaktionen. Wenn die Bedingung erfüllt ist, werden die auf `then` folgenden Zustandsaktionen ausgeführt, ansonsten die auf `else` folgenden Zustandsaktionen. Der `else`-Teil kann entfallen.

Die *Case-Zustandsaktion* ermöglicht die bedingte Ausführung von maximal 2^n alternativen Zustandsaktionen `Ai` in Abhängigkeit von einem Bedingungsvektor der Länge n.

```
case C then     "entspricht mit Ki=(C=Ci)"
?C0 : A0        if K0 then A0 fi,
?C1 : A1        if K1 then A1 fi,
...             ...
?Cn : An        if Kn then An fi,
else Ae         if  K0. K1... Kn then Ae fi
esac
```

Die Bedingungen `Ki` müssen sich *nicht* gegenseitig ausschließen. `Ci` kann eine Liste von Vergleichskonstanten sein, wie beim permanenten Case-Statement. Der `else`-Teil kann entfallen. Die Case-Zustandsaktion `case C of` unterscheidet sich vom `case C then` dadurch, daß sich die Bedingungen `Ki` gegenseitig ausschließen müssen.

2.13 Equal-Deklaration, Const-Deklaration

Die *Equal-Deklaration* dient dazu, bereits deklarierten Variablen oder Teilen von ihnen einen neuen Namen, eine neue Indizierung oder ein neues Datenformat zuzuordnen. Die so neu definierten Variablen heißen *Equal-Variablen*. Insbesondere dient die Equal-Deklaration zur hierarchischen Zerlegung einer Variablen in Teile. Die gewünschten Teile werden durch Indizierung mit Konstanten spezifiziert. In jeder Equal-Deklaration dürfen auf der rechten Seite nur direkt deklarierte oder durch vorhergehende Equal-Deklarationen definierte Equal-Variablen auftreten. Das Datenformat der Equal-Variablen auf der linken Seite muß mit dem Datenformat der indizierten Variablen auf der rechten Seite nicht übereinstimmen. Die Anzahl der Bits aber muß auf beiden Seiten gleich sein. Die Zuordnung der Bits erfolgt dadurch, daß die Zeilen von oben nach unten und die Bits von rechts nach links gezählt werden. Die Indizierung auf der rechten Seite darf nicht mehrmals den gleichen Index aufweisen, um eine 1:1-Zuordnung zwischen den Bits zu gewährleisten. Die Equal-Deklaration darf nicht auf Output-Variablen angewandt werden.

Mit Hilfe der *Const-Deklaration* können Konstanten durch symbolische Namen ersetzt werden.

Beispiel:

```
boole INSTRUCTION(16:1);  register R(3:0,8:1);

equal OPCODE(4:1)       = INSTRUCTION(16:13),
      ADDRESS(12:1)     = INSTRUCTION(12:1),
      A16(16:1)         = R(1:0), BIT8(3:0) = R(,8);
const ZERO = '0000'     , MINUS1 = 'FFFF';
```

Durch diese Equal-Deklaration wird `INSTRUCTION` in die beiden Teile `OPCODE` und `ADDRESS` zerlegt, `A16` als 16-Bit-Register `R(1,)_R(0,)` definiert, `BIT8` zu `R(3,8)_R(2,8)_R(1,8)_R(0,8)` definiert, und es werden zwei symbolische Konstanten vereinbart.

2.14 Subunits

Digitale Systeme bestehen im allgemeinen aus einer Vielzahl von miteinander verbundenen Funktions- bzw. Baueinheiten (Units). Jede Unit kann mehrere *Subunits* besitzen. Hardwaremäßig kann man sich die Subunits z. B. als integrierte Schaltkreise auf einer gedruckten Schaltung oder als Funktionseinheiten auf einem Chip vorstellen. Die Subunits werden untereinander und mit der Unit über ihre Eingangs- und Ausgangsvariablen miteinander verbunden. Ansonsten existieren keine weiteren gemeinsamen Variablen zwischen den Units. Eine Subunit ist eine Kopie einer zuvor definierten Unit, die selbst wieder Subunits enthalten kann. Somit lassen sich Hierarchien von Units aufbauen.

Bevor die Subunit kopiert werden kann, muß sie als Original-Unit definiert werden. Anschließend können dann mit Hilfe der Copy-Deklarationen Subunits erzeugt werden, die einen anderen Namen bekommen können. Von einer Original-Unit können mehrere Subunits mit verschiedenen Namen kopiert werden. Die Eingangs- und Ausgangsvariablen der Subunits müssen in gleicher Weise wie die Original-Unit deklariert werden, können aber einen anderen Namen erhalten. Die Verbindung der Subunits untereinander und mit der Unit erfolgt durch permanente Zuweisungen an die Input-Variablen.

Beispiel:

```
unit  FULLADDER; input A, B, C; output D, S;
perm  S==A+B+C,  D==A.B v A.C v B.C pend uend

unit   ADD; input A(4:1), B(4:1), CIN; output S(4:1), COUT;
copy   FULLADDER :1:2:3:4; input A, B, C; output D, S; cend
perm   A1==A(1), A2==A(2), A3==A(3), A4==A(4),
       B1==B(1), B2==B(2), B3==B(3), B4==B(4),
       C1==CIN, C2==D1, C3==D2, C4==D3, COUT==D4,
       S ==S4_S3_S2_S1 pend uend
```

Die Addierschaltkette (`unit ADD`) besteht aus 4 hintereinandergeschalteten Volladdierern (Subunits `FULLADDER1,2,3,4`). Zuerst wird die Original-Unit `FULLADDER` definiert, die dann 4 mal durch „`copy FULLADDER:1,2,3,4`“ kopiert wird. Die Namen der Eingangs- und Ausgangsvariablen der Subunits werden neu vereinbart, wobei zur Unterscheidung der verschiedenen Kopien die Marken `1,2,3,4` an die Namen angefügt werden. Stellt man nur eine Kopie von einem Original her, dann braucht sie nicht durch eine Marke gekennzeichnet zu werden. Anschließend werden die Verbindungen definiert.

2.15 Makro-Operationen

Makro-Operationen dienen zur Modifikation des Quelltextes vor der Übersetzung. Die Makro-Deklaration dient dazu, einen meist längeren Quelltextteil unter einem Namen `M` zu

definieren: macro <M> ::= TEXT mend. Befindet sich an beliebiger Stelle im Programm nach der Definition der Aufruf <M>, so wird an dieser Stelle der vereinbarte TEXT eingesetzt. Eine syntaktische und semantische Prüfung des Textes erfolgt erst nach der Ersetzung. Die Makro-Deklaration muß vor dem Aufruf erfolgen. In einer Makro-Deklaration dürfen vorher definierte Makros aufgerufen werden.

Das generierende For-Statement kann an einer beliebigen Stelle im Ausführungsteil stehen. Es erzeugt aus einem Quelltextteil S(K1, K2,... Kn) eine Liste von Quelltextteilen, indem nacheinander für K1 bis Kn die in einer Liste definierten Werte eingesetzt werden. Die formalen Parameter Ki müssen Buchstaben sein, die nicht gesondert deklariert werden. Im Quelltextteil darf weder vor noch nach den Ki ein Buchstabe stehen. Das generierende For-Statement

```
generate for K1 := C11, C12, ... C1m
for K2 := C21, C22,... C2m
... for Kn := Cn1, Cn2,...Cnm do
S(K1, K2, ... Kn) od * endgenerate
```

erzeugt die Liste

```
S(C11, C21, ... Cn1) * S(C12, C22, ... Cn2)
* ... S(C1m, C2m, ... Cnm).
```

Die Listenwerte Cij können beliebige Zeichenketten sein, die kein Komma enthalten dürfen. Das Zeichen * ist ein beliebiges Trennzeichen, das auch entfallen kann. Das generierende For-Statement eignet sich z. B. zur Beschreibung von Schaltketten.

2.16 HDL-Programmbeispiele

Beispiel 1: Vierfach-Nand-Gatter mit Zeitverzögerung

```
unit SN'7400N; input A(1:4), B(1:4); output Y(1:4);
perm Y == delay 10 ns ~(A.B) pend
uend
```

Beispiel 2: Flankengetriggertes D-Flipflop

```
unit D'FLIPFLOP'7474; input CLOCK,D; output boole Q;
loop wait until rise CLOCK; Q:=D lend
uend
```

Beispiel 3: Serielles 2-Komplement Y(n:1) = ~ X(n:1)+1

```
unit COMPLEMENT; input Xi, output Yi;
on clock
[Z0] Yi == Xi, if Xi=o then next Z0 else next Z1 fi
[Z1] Yi == ~Xi
noc uend
```

2.17 Verschiedene Arten von Mikroprogrammen

Anhand eines Beispiels, der Multiplikation positiver Dualzahlen, soll gezeigt werden, wie unterschiedlich ein Mikroalgorithmus (durch Hardware oder Mikrooperationen zu realisierender Algorithmus) dargestellt werden kann. Gesucht ist das Produkt

$$P = X * Y = (X_n \dots X_1) * (Y_n \dots Y_1) = X * \sum_{i=1}^{n} Y_i * 2^{i-1} \ .$$

Durch Umformung in ein Horner-Schema und der Abkürzung $Xh = X * 2^n$ ergibt sich folgendes rekursive Gleichungssystem:

$$\begin{aligned} P^0 &= 0 \\ P^1 &= (P^0 + Xh * Y_1) * 2^{-1} \\ &\vdots \\ P^n &= (P^{n-1} + Xh * Y_n) * 2^{-1} \end{aligned}$$

Es läßt sich leicht algorithmisch als HDL-Programm (z. B. für n=8) formulieren:

```
boole X(8:1), Y(8:1), Xh(16:1), P(16:1),
      i(4:1); "Laufvariable"
loop  P:=0; Xh:= X_oooo oooo;
for   i:= 1 to 8 do P:= (P + Xh*Y(i))/2 od
stop  lend
```

Softwaremäßig betrachtet beschreibt dieses Programm hinreichend genau die serienparallele Multiplikation. Aus der Sicht der Hardware-Realisierung ist die Benutzung der Operatoren `*` und `/` aber zu aufwendig. HDL ermöglicht es, diesen Algorithmus hardwarenah, d. h. unter Verwendung der typischen realisierbaren Mikrooperationen (hier Addition und Schift) zu beschreiben:

```
boole   X(8:1), Y(8:1), P(17:1),
i(3:1); "Laufvariable"
loop    P:= o oooo oooo_Y
for     i:= 0 to 7 do
if P(1) then P(17:9):= P(16:9)++X fi; P:= shr P od
stop lend
```

Die beiden letzten HDL-Programme sind Beispiele für *abstrakte Mikroprogramme*. Ein abstraktes Mikroprogramm dient dazu, die Funktionsweise einer Funktionseinheit abstrahiert von ihrer Realisierung zu beschreiben. Es eignet sich daher zur Spezifikation beim Entwurfsvorgang. Der Vorteil des zweiten Mikroprogramms ist der, daß es Realisierungshinweise bezüglich der zu verwendenden oder verwendeten Operatoren enthält. HDL ermöglicht es, eine ganze Palette abstrakter Mikroprogramme zu formulieren, so daß der Top-Down-Entwurf

bis hin zur konkreten Realisierung beschrieben werden kann. Die abstrakten Mikroprogramme können kollaterale und simultane Anweisungen enthalten, so daß auch die möglichen bzw. notwendigen parallelen Vorgänge beschreibbar sind.

Weiterhin läßt sich der Mikroalgorithmus als synchrones Mikroprogramm beschreiben:

```
register X(8:1), Y(8:1), P(17:1), i(3:1);
on clock
[Z1] P <- o oooo oooo_Y, next Z2,
"for i:= 0 to 7 do" i<-0
[Z2] if P1(1) then P(17:9) <- P(16:9)++X fi, next Z3
[Z3] P <- shr P,
"od" if i<7 then next Z2 else next Z1 fi, i <- i+1
noc
```

Ein *synchrones Mikroprogramm* dient hauptsächlich zur Beschreibung des Verhaltens eines zu realisierenden oder realisierten synchronen Automaten. Es kann aber auch als abstraktes Mikroprogramm mit simultanen Anweisungen interpretiert werden, wenn von dem Vorhandensein eines bestimmten Taktes abstrahiert wird.

Zur Beschreibung der Mikroalgorithmen im Kapitel 3 werden vorzugsweise synchrone Zustandsdiagramme benutzt, die graphische Darstellungen der synchronen Mikroprogramme sind. Sie sind anschaulicher, weil die Zustandsübergänge deutlicher hervortreten. Zur Vereinfachung werden außerdem Wiederholungen von Zuständen durch rückführende Pfeile ersetzt, die mit der Anzahl der Durchläufe versehen werden können. Außerdem wird gekennzeichnet, welcher Pfeil zuerst und welcher zuletzt durchlaufen wird. Das synchrone Zustandsdiagramm für das Beispiel zeigt Abb. 2.5.

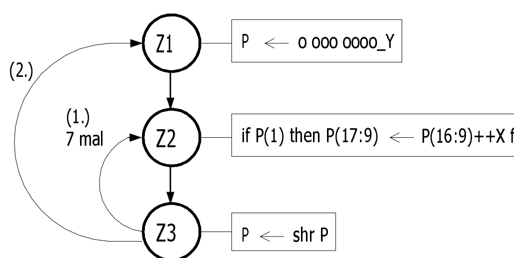


Abbildung 2.5: Synchrones Zustandsdiagramm

Ein Mikroalgorithmus, der Funktionswerte berechnet, die nur von den Eingangswerten und nicht von einem inneren Zustand abhängen, lassen sich als Schaltnetze oder Schaltketten beschreiben und realisieren. Die zugehörige Beschreibungsform soll *paralleles Mikroprogramm* heißen. Es besteht aus einem Satz von booleschen Gleichungen (permanenten Anweisungen),

wobei die Reihenfolge der Aufschreibung keine Rolle spielt (nicht-prozedurale Schreibweise). Das Beispiel lautet als paralleles Mikroprogramm:

```
signal X(4:1), Y(4:1), P(1:4,5:1), S(8:1);
P(1,) ==              Y.X(1),
P(2,) == P(1,5:2) ++ Y.X(2),
P(3,) == P(2,5:2) ++ Y.X(3),
P(4,) == P(3,5:2) ++ Y.X(4),
S     == P(4,)_P(3,1)_P(2,1)_P(1,1) "S=X**Y".
```

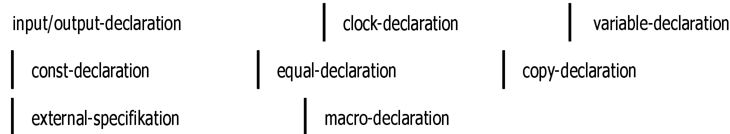
Zur Verkürzung der Schreibweise könnten die drei mittleren Gleichungen aus einer Gleichung mit variablem Index mit Hilfe des generierenden For-Statements generiert werden.

Schaltnetze oder Schaltketten, die aus einer Hintereinanderschaltung von Gattern oder logischen Funktionsbausteinen (wie Addierer, Multiplizierer) bestehen, werden auch als *ortssequentielle* (asynchrone) Schaltungen [Lie] bezeichnet. Dieser Bezeichnungsweise folgend könnte man parallele Mikroprogramme auch als *ortsseqentielle Mikroprogramme* bezeichnen. Ebensogut könnte man sie als *Datenflußprogramme* bezeichnen, da die Eingangsdaten (im Beispiel `X` und `Y`) über die Zwischenwerte (im Beispiel `P`) zum Ergebnis (im Beispiel `S`) fließen.

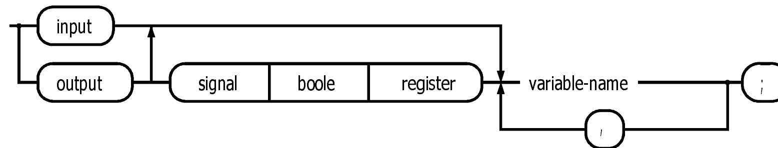
2.18 HDL-Syntax

1. unit

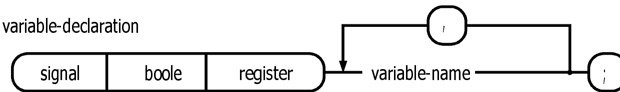
2. declaration



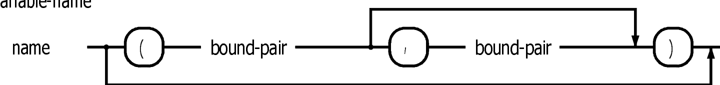
3. input/output-declaration



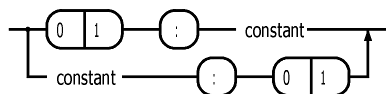
4. variable-declaration



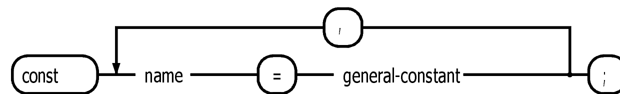
5. variable-name



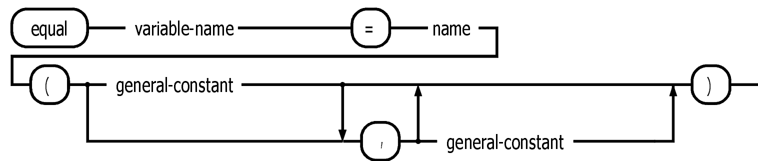
6. bound-pair



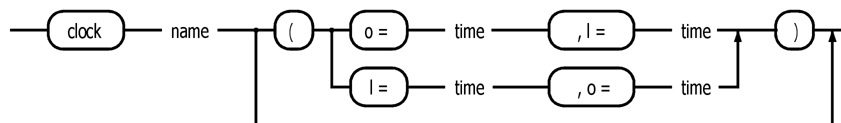
7. const-declaration



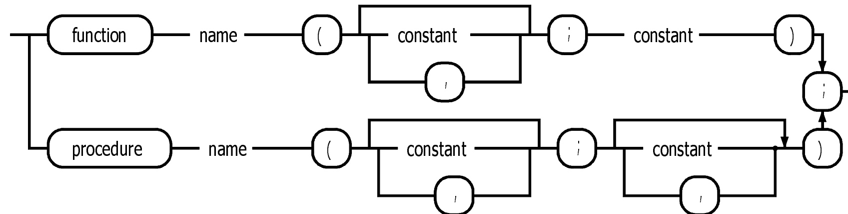
8. equal-declaration



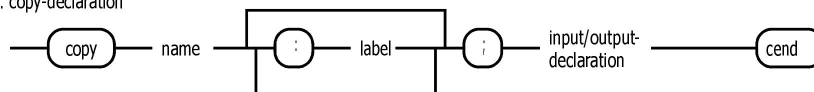
9. clock-declaration



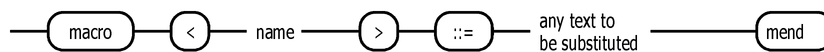
10. external-specification



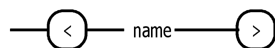
11. copy-declaration



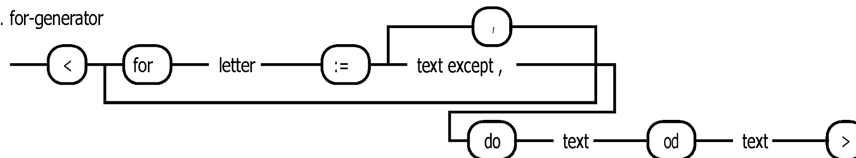
12. macro



13. macro-call

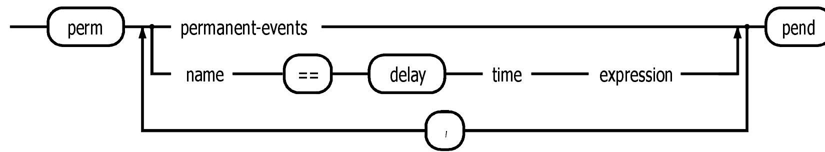


14. for-generator

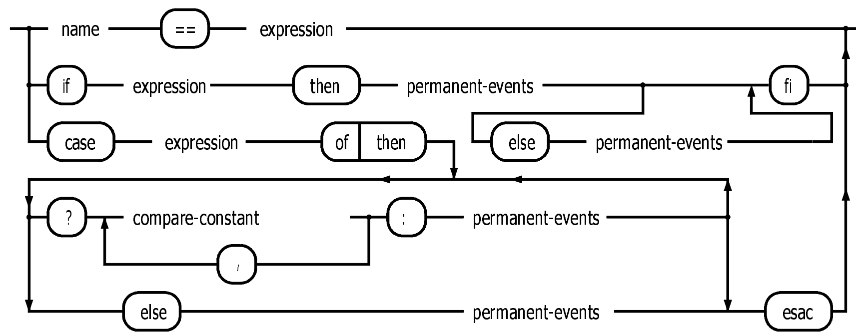


15. execution-part

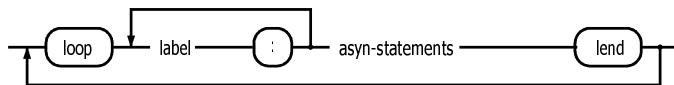
16. permanent-statements



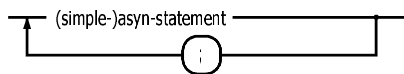
17. permanent-events



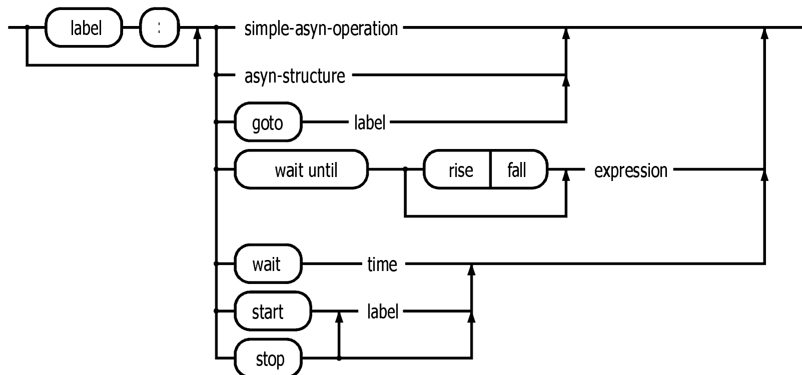
18. asynchronous-processes



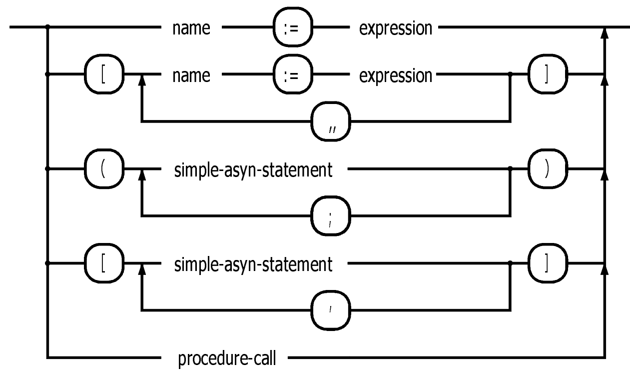
19. (simple-)asyn-statements



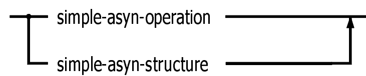
20. asyn-statement



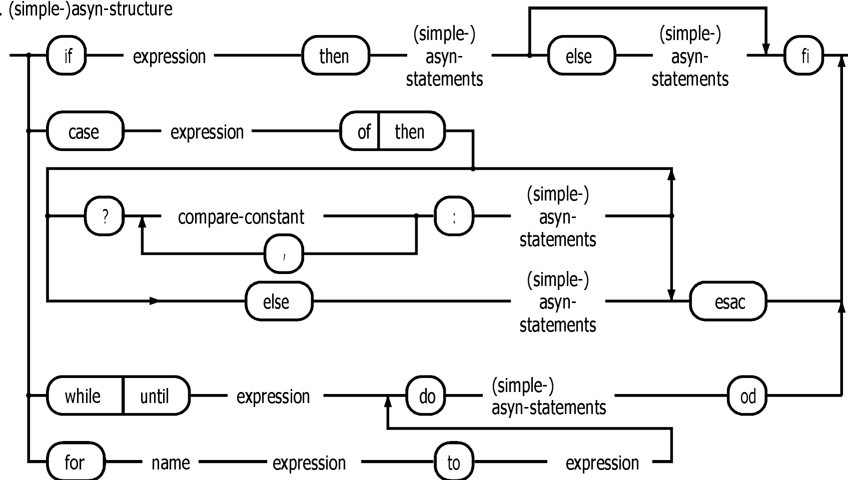
21. simple-asyn-operation



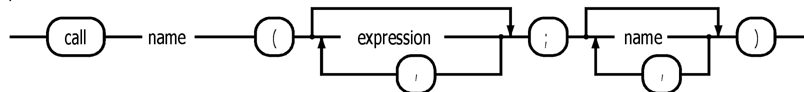
22. simple-asyn-statement



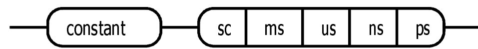
23. (simple-)asyn-structure



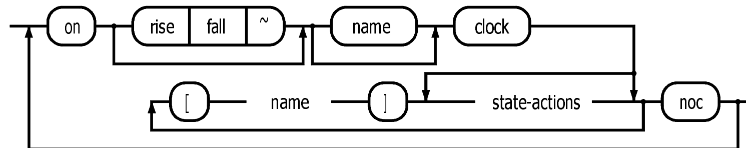
24. procedure-call



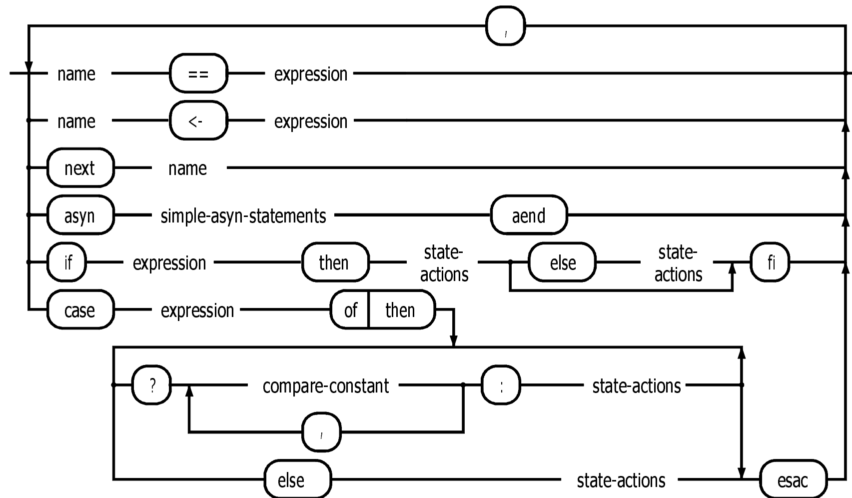
25. time



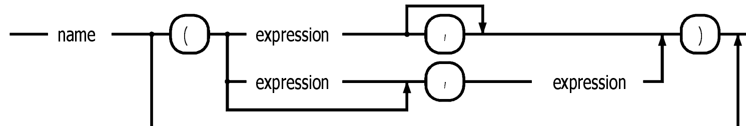
26. automata-sequence



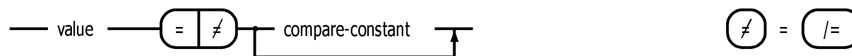
27. state-actions



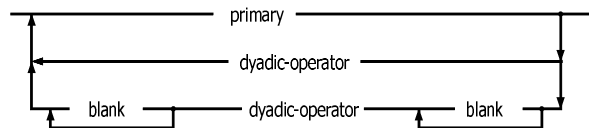
28. variable



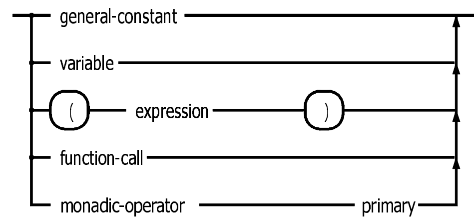
29. expression



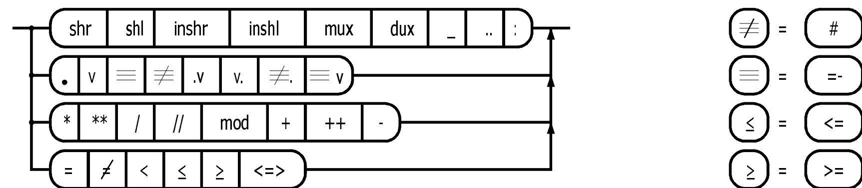
30. value



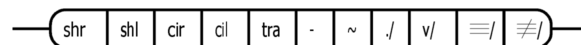
31. primary



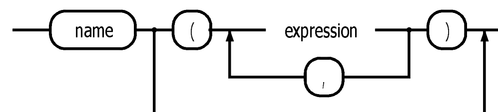
32. dyadic-operator



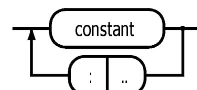
33. monadic-operator



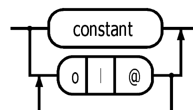
34. function-call



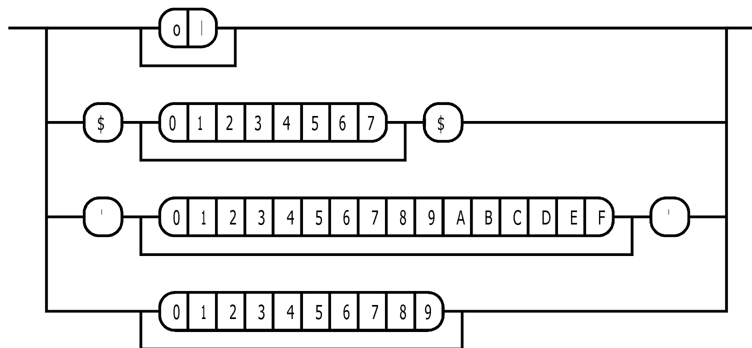
35. general-constant



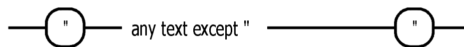
36. compare-constant



37. constant

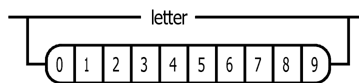


38. comment



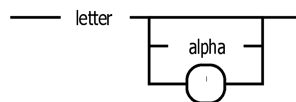
39. label

40. alpha



41. letter

42. name



3 Mikroalgorithmen und Rechenwerke für die Grundrechenarten

In diesem Kapitel werden hardware-orientierte Algorithmen (*Mikroalgorithmen*) und Realisierungen für die Grundrechenarten (Rechenwerke) bei verschiedenen Zahlendarstellungen behandelt.

Verallgemeinert gesehen dienen Rechenwerke zur Berechnung von Funktionen. Eine Funktion kann am einfachsten durch eine Tabelle realisiert werden, die in einem normalen Speicher abgelegt wird. Obwohl diese Realisierung aufgrund der technologischen Fortschritte immer mehr an Bedeutung gewinnt, so ist doch der Aufwand bei größeren Wortlängen der Operanden nicht mehr vertretbar. Der Aufwand läßt sich erheblich verringern, wenn die der Funktion innewohnenden Eigenschaften herauskristallisiert werden und durch eine Hintereinanderschaltung von Teilfunktionen – entweder räumlich/asynchron durch Schaltkettenglieder oder zeitlich/synchron durch Teilfunktionen auf Registerinhalten – realisiert werden.

Die Zerlegung einer Funktion in Teilfunktionen kann als Algorithmus formuliert werden. Der Algorithmus definiert, in welcher Weise bestimmte Schaltkettenglieder räumlich bzw. Registeroperationen zeitlich aufeinanderfolgen müssen. Da der Algorithmus die richtige Aufeinanderfolge der hardwaremäßig zur Verfügung stehenden Teilfunktionen (*Mikrofunktionen*) steuert, wird er auch als *Mikroalgorithmus* oder *Hardware-Algorithmus* bezeichnet. Ein Mikroalgorithmus wird als *Mikroprogramm* bezeichnet, wenn er in einer hardware-orientierten Sprache (Hardware-Beschreibungssprache, Mikroprogrammsprache) formuliert wurde. Bei der räumlich-sequentiellen Realisierung als Schaltkette sind alle Teilfunktionen als Schaltkettenglieder physikalisch vorhanden und müssen gemäß dem Mikroalgorithmus miteinander verbunden werden. Dabei ist eine explizite Steuerung nicht erforderlich, sie steckt implizit in der Verdrahtung. Dagegen wird bei der zeitlich-sequentiellen Realisierung zusätzlich ein Steuerwerk benötigt, das ganz oder teilweise im Rechenwerk integriert sein kann. Das Rechenwerk ist in der Lage, Daten zu speichern, zu transportieren und Mikrofunktionen auszuführen. Das Steuerwerk veranlaßt das Rechenwerk, gemäß dem Mikroalgorithmus in jedem Schritt

1. Operanden auszuwählen,
2. bestimmte Mikrofunktionen auszuführen und
3. Zwischenergebnisse/Ergebnisse zu speichern.

Das Auswählen der Operanden, Durchführen der Mikrofunktionen und das Speichern der Ergebnisse wird auch als *Mikrooperation* bezeichnet.

3.1 Addition

3.1.1 Addition von Dualzahlen

Die Addition von Dualzahlen wird meist ziffernweise unter Berücksichtigung der Überträge durchgeführt. Für die Addition von Dualziffern stehen Halb- und Volladdierer zur Verfügung, die zur Realisierung von Addierschaltnetzen hintereinandergeschaltet werden. Am einfachsten, aber langsamsten kann die Addition durch serielle Addierwerke realisiert werden; wird eine hohe Verarbeitungsgeschwindigkeit verlangt, so werden Addierer mit Übertragsvorausberechnung verwendet.

3.1.1.1 Halbaddierer und Volladdierer

Ein *Halbaddierer* ist ein Schaltnetz, das die Summe von zwei Dualziffern A_i und B_i bildet. Die Summe kann die Werte 0, 1, 2 annehmen; sie wird durch zwei Bits, das Summenbit S_i und das Übertragsbit C_{i+1}, dual codiert. Die logische Funktion wird durch die Wahrheitstabelle (Abb. 3.1a) oder durch die booleschen Gleichungen (Abb. 3.1b) beschrieben. Eine Realisierung mit *NOR*-Gattern zeigt Abb. 3.1d und als Schaltzeichen wird hier Abb. 3.1c verwendet. Halbaddierer sind nur für die „halbe“ Addition geeignet, weil ein zusätzlicher Übertrag aus der vorhergehenden Stelle nicht berücksichtigt werden kann. Aus zwei Halbaddierern läßt sich ein Volladdierer aufbauen.

Ein *Volladdierer* ist ein Schaltnetz, das die Summe von drei Dualziffern A_i, B_i und C_i bildet. Er kann die „volle“ Addition durchführen, weil er den Übertrag C_i aus der vorhergehenden Stelle berücksichtigt. Die Summe kann die Werte 0, 1, 2, 3 annehmen; sie wird durch zwei Bits, das Summenbit S_i und das Übertragsbit C_{i+1}, dual codiert. Die logische Funktion kann durch die Wahrheitstabelle (Abb. 3.2a) oder durch die booleschen Gleichungen (Abb. 3.2b) beschrieben werden. Als Schaltzeichen wird hier Abb. 3.2c verwendet. Ein Volladdierer läßt sich in Abhängigkeit von den zur Verfügung stehenden Gattern und der zulässigen Verzögerungszeit auf verschiedene Arten realisieren. Eine mögliche Realisierung zeigt Abb. 3.2d.

3.1.1.2 Addierer mit Übertragsweiterleitung

Ein Additionsschaltnetz zur Addition von zwei n-stelligen Dualzahlen wird am einfachsten in Form von n hintereinander geschalteten Volladdierern realisiert (Abb. 3.3). Durch jeden Vol-

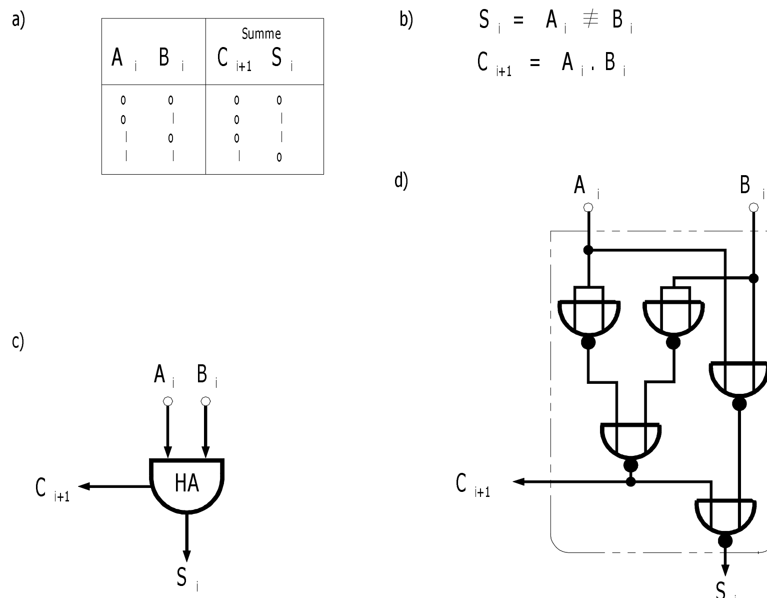


Abbildung 3.1: Halbaddierer

laddierer wird eine Stelle verarbeitet, wobei der Übertrag aus der vorhergehenden Stelle berücksichtigt wird. Der hereingehende Übertrag C_1 (carry in) ist normalerweise gleich $\circ$, außer bei speziellen Operationen (Doppelwort-Addition, Komplementbildung). Der herausgehende Übertrag C_n (carry out) ist gleich dem höchstwertigen Summenbit S_{n+1}. Die $(n+1)$-stellige Summe kann maximal den Wert $2^{n+1} - 1 = (2^n - 1) + (2^n - 1) + 1$ annehmen. C_n kann bei einer Addition mit doppelter Wortlänge zwischengespeichert und anschließend als hereingehender Übertrag berücksichtigt werden. Im ungünstigsten Fall muß das Durchlaufen des Übertrags durch alle Stufen abgewartet werden, um die Addition zu beenden. Dieser Addierer heißt deswegen auch *Carry-Ripple-Addierer*. Die Additionszeit ist proportional zur Anzahl der zu verarbeitenden Stellen. Sie läßt sich durch eine vorausschauende Berechnung der Überträge verringern.

3.1.1.3 Addierer mit Übertragsvorausberechnung

Durch die Methode der *Übertragsvorausberechnung* (Carry-Look-Ahead) gelingt es, die Additionszeit zu verringern. Beim Carry-Ripple-Addierer werden die Überträge nacheinander durch die n rekursiven Gleichungen

$$C_{i+1} = A_i.B_i \vee (A_i \vee B_i).C_i \qquad i = 1, 2, \ldots, n \qquad (3.1)$$

berechnet. Realisiert man die Übertragsberechnung nach diesen Gleichungen, dann werden für die erste Stufe 3 und für alle weiteren Stufen je 2 Gatterlaufzeiten E zur Berechnung benötigt $(3 + (n-1) * 2)$. Die Zeit zur Berechnung der Überträge C_3, C_4 usw. läßt sich reduzieren,

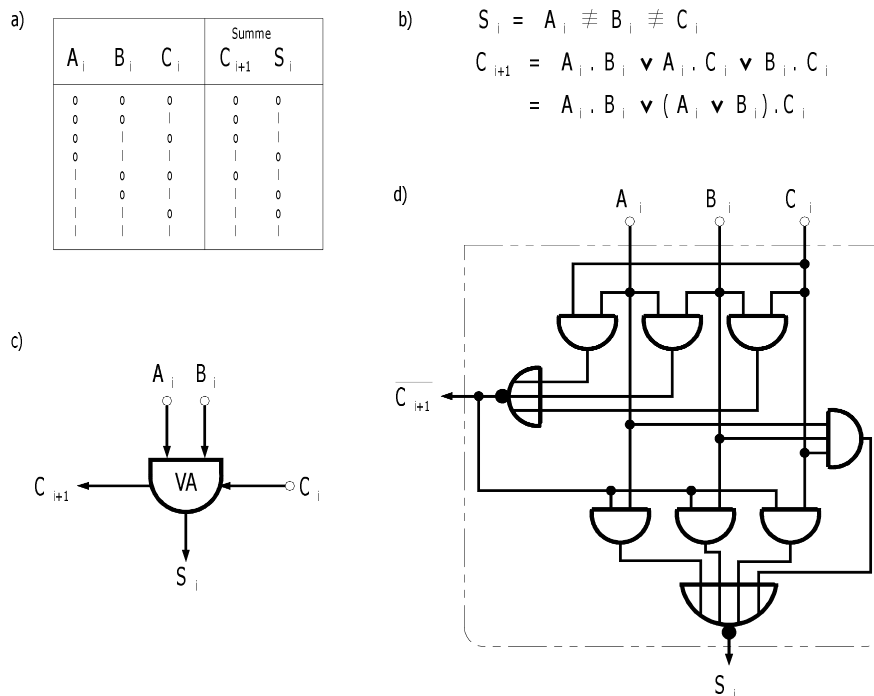


Abbildung 3.2: Volladdierer

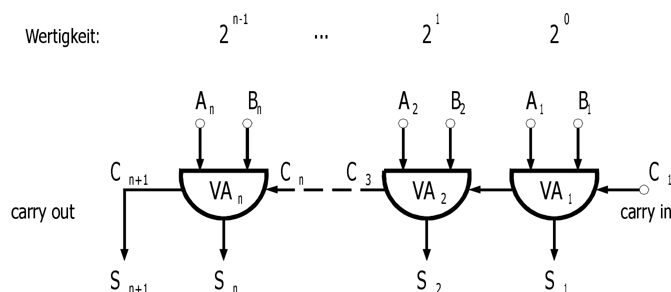


Abbildung 3.3: Additionsschaltnetz für Dualzahlen

indem die Gleichungen ineinander eingesetzt werden. Durch Zusammenfassen der beiden Übertragungsgleichungen für $i = 1$ und $i = 2$ ergibt sich z.B.

$$C_3 = A_2.B_2 \vee (A_2 \vee B_2).A_1.B_1 \vee (A_2 \vee B_2).(A_1 \vee B_1).C_1 \; . \tag{3.2}$$

Nach dieser Beziehung läßt sich C_3 ebenso schnell wie C_2 berechnen. Dafür hat sich der Realisierungsaufwand erhöht. Durch Zusammenfassen von drei Übertragungsgleichungen ergibt sich eine noch aufwendigere Beziehung, deren Realisierung es erlaubt, auch den Übertrag C_4 in der gleichen Zeit von $3E$ zu berechnen. In den so zusammengefaßten Übertragsgleichungen wiederholen sich die Terme $G_i = A_i.B_i$ und $P_i = A_i \vee B_i$ sehr oft, so daß

sie zweckmäßigerweise als Hilfsfunktionen realisiert werden. G_i wird als Übertragserzeugung (carry generate) und P_i als Übertragsweiterleitung (carry propagate) bezeichnet. Denn der alte Übertrag C_i wird nach Gleichung (3.1) weitergeleitet, wenn P_i = 1 ist, und ein neuer Übertrag wird erzeugt, wenn G_i = 1 ist. Mit diesen Hilfsfunktionen lauten die Gleichungen für einen 4-Bit-Addierer mit Übertragsvorausberechnung:

	Gesamtrechenzeit	
$C_2 = G_1 \vee P_1C_1$	$3\,E$	(3.3)
$C_3 = G_2 \vee P_2G_1 \vee P_2P_1C_1$	$3\,E$	(3.4)
$C_4 = G_3 \vee P_3G_2 \vee P_3P_2G_1 \vee P_3P_2P_1C_1$	$3\,E$	(3.5)
$C_5 = G_4 \vee P_4C_4$	$6\,E$	(3.6)
$S_1 = A_1 \not\equiv B_1 \not\equiv C_1$	$2\,E$(für ex.oder)	(3.7)
$S_2 = A_2 \not\equiv B_2 \not\equiv C_2$	$5\,E$	(3.8)
$S_3 = A_3 \not\equiv B_3 \not\equiv C_3$	$5\,E$	(3.9)
$S_4 = A_4 \not\equiv B_4 \not\equiv C_4$	$5\,E$	(3.10)

(Der Übersichtlichkeit halber wurde in diesen Gleichungen die Und-Verknüpfung zwischen den Variablen weggelassen.)

Da die Berechnung der Summenbits $5\,E$ benötigt, braucht die Berechnung von C_5 nicht schneller durchgeführt werden. Für C_5 wird deshalb die vereinfachte Berechnung (3.6) verwendet, die C_4 voraussetzt.

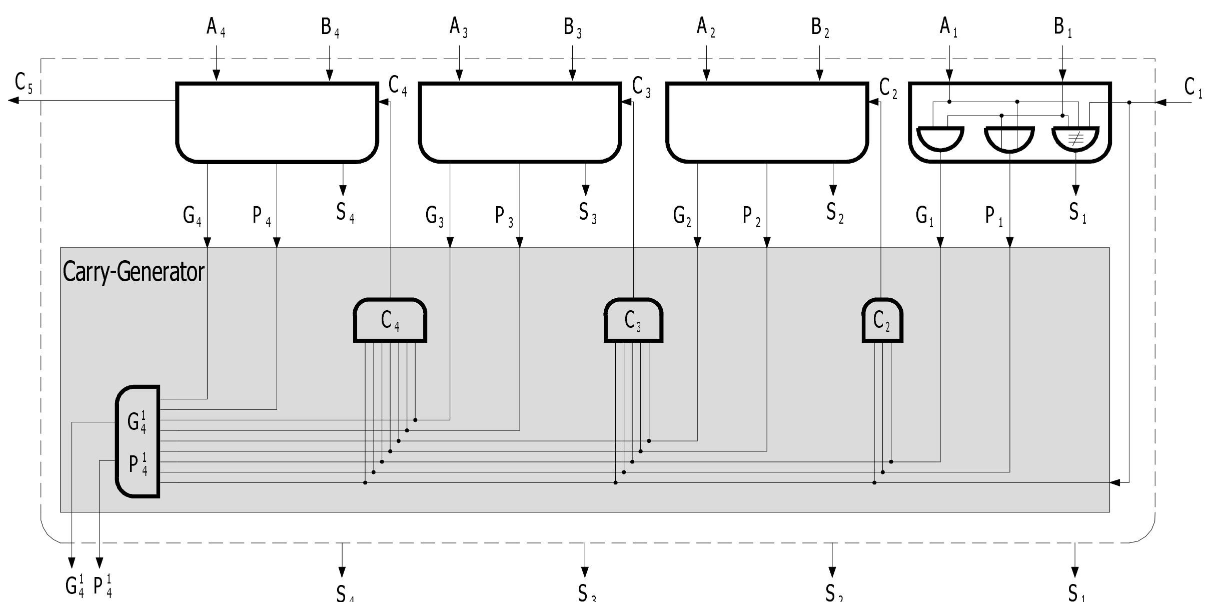


Abbildung 3.4: 4-Bit-Addierer mit Übertragsvorausberechnung

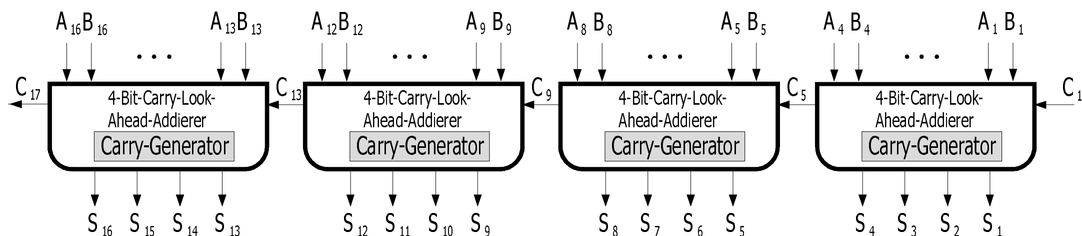


Abbildung 3.5: 16-Bit-Addierer bestehend aus vier 4-Bit-Addierern mit Übertragsvorausberechnung

Da der Aufwand zur Realisierung der Übertragsfunktionen mit der dritten Potenz der Stellenzahl steigt, werden im allgemeinen nur 4-Bit-Addierer mit vollständiger Übertragsvorausberechnung realisiert (Abb. 3.4). Um z. B. einen 16-Bit-Addierer zu realisieren, können 4-Bit-Addierer mit Übertragsvorausberechnung nach dem Ripple-Carry-Prinzip hintereinander geschaltet werden (Abb. 3.5). Die Additionszeit beträgt dann das Vierfache des einzelnen 4-Bit-Addierers.

Diese Additionszeit läßt sich durch Einführen einer zweiten Carry-Generator-Ebene etwa auf die Hälfte reduzieren. Man bezeichnet die Realisierung der Funktionen (3.3), (3.4), (3.5) einschließlich der beiden Funktionen

$$G_4^1 = G_4 \vee P_4G_3 \vee P_4P_3G_2 \vee P_4P_3P_2G_1 \quad (3.11)$$
$$P_4^1 = P_4P_3P_2P_1 \quad (3.12)$$

als Carry-Generator und die Funktion G_4^1 als Übertragserzeugung 1. Ordnung und P_4^1 als Übertragsweiterleitung 1. Ordnung.

Schaltet man die Hilfsfunktionen 1. Ordnung auf einen weiteren Carry-Generator in einer zweiten Ebene (Abb. 3.6), dann werden die Überträge C_5, C_9, C_{13} gleichzeitig mit erhöhter Geschwindigkeit berechnet. Gegenüber dem 16-Bit-Addierer nach Abb. 3.5 verringert sich die Additionszeit etwa auf die Hälfte. Eine ausführliche Betrachtung der Carry-Look-Ahead-Technik ist in [Gil] zu finden, praktische Realisierungen in [Mor].

3.1.1.4 Von Neumannsche Addition

Nach VON NEUMANN läßt sich ein einfaches Additionswerk konstruieren, das nur die booleschen Funktionen „Exclusiv-Oder“ und „Und“ (Halbaddierer) sowie einen Linksschift benötigt. Dieses Additionswerk bildet die Summen $S = X + Y$ nach folgendem Algorithmus:

1. Zu Beginn (*START* $=$ `1`) wird der Summand X in den Akkumulator *AC* und der Summand Y in das Übertragsregister *CY* übernommen.

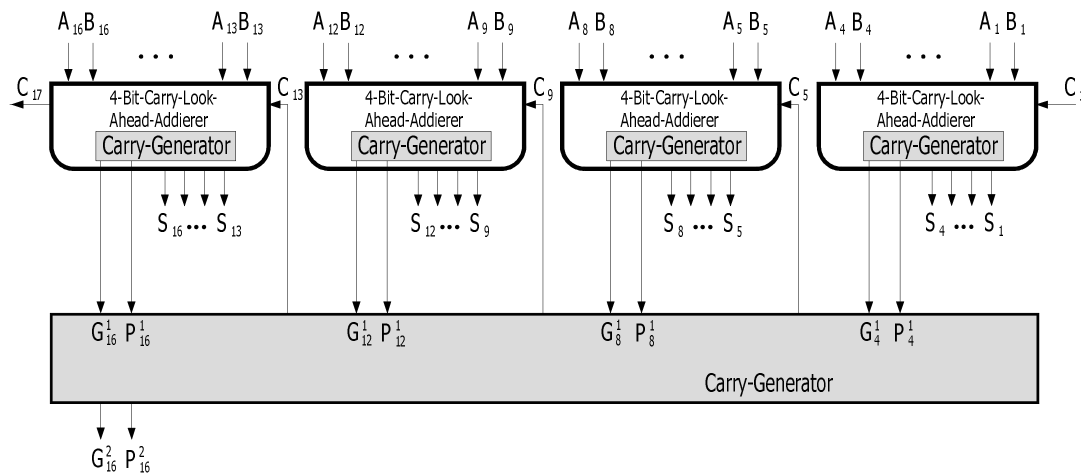


Abbildung 3.6: 16-Bit-Addierer mit zwei Carry-Generator-Ebenen

2. Zu dem Akkumulator werden der 2. Summand bzw. die vorhergehenden Überträge addiert ($AC \leftarrow AC \not\equiv CY$), und es werden die neuen Überträge $AC.CY$ gebildet, die zur Übertragsweiterleitung um eine Stelle nach links verschoben werden: $CY \leftarrow \text{shl}(AC.CY)$.

   Der 2. Schritt wird solange wiederholt, bis alle Überträge ($ENDE = \mathtt{1}$) verarbeitet worden sind.

Abb. 3.7 zeigt den Mikroalgorithmus zur Addition 4-stelliger Dualzahlen als synchrones Mikroprogramms in HDL. Das Steuerwerk geht mit dem *START*-Signal in den Verarbeitungszustand $S2$, in dem die Überträge schrittweise berücksichtigt werden. Wenn keine Überträge mehr entstehen, geht das Steuerwerk in den Wartezustand $S1$ zurück und das *ENDE*-Signal nimmt den Wert $\mathtt{1}$ an. Den letzten Verarbeitungsschritt könnte man durch eine modifizierte *ENDE*-Abfrage ($\text{shl}\,(AC.CY) = 0$) einsparen.

```
unit ADD;
input X(3:0), Y(3:0), START;
output S(3:0), ENDE;
register AC(3:0), CY(3:0);
perm ENDE == (CY=oooo), S == AC pend
on clock
[S1]  if START  then AC <-  X, CY  <-  Y,  next S2
                else                       next S1 fi
[S2]  if ENDE   then AC <-  AC  ≢  CY, CY  <-  shl(AC.CY), next S2
                else                                      next S1 fi
noc uend
```

Abbildung 3.7: Addition durch logische Funktionen

3.1.1.5 Serielles Addierwerk

Während man heute die Addition meist wortparallel durch ein Schaltnetz durchführt, wurde sie früher aus Kostengründen seriell durchgeführt. Dadurch werden nicht nur logische Verknüpfungsglieder eingespart, sondern es werden auch wesentlich weniger Leitungsverbindungen benötigt.

Ein serielles Addierwerk besteht aus zwei n-stelligen Schieberegistern, einem Volladdierer und einem Übertragungsspeicher (Beispiel $n = 4$, Abb. 3.8). Es bildet nacheinander in n Schritten aus jeweils zwei Bits der Summanden und dem vorhergehenden Übertrag ein Summenbit und ein neues Übertragsbit. Der Ablauf der Summenbildung wird durch ein Steuerwerk mit folgendem Algorithmus gesteuert:

1. Wenn das Startsignal erfolgt, dann werden die Summanden in die Register übernommen und das Übertragsregister wird gelöscht ($A \leftarrow X,\ B \leftarrow Y,\ C \leftarrow 0$).

2. Das Summenbit S und das Übertragungsbit *CNEU* werden mit Hilfe eines Volladdierers aus den niedrigstwertigen Registerbits A_0, B_0 und dem vorhergehenden Übertrag C gebildet.

3. Der neue Übertrag wird gespeichert ($C \leftarrow$ *CNEU*), und die Register werden um eine Stelle nach rechts geschoben. Gleichzeitig wird das Summenbit in das freiwerdende höchstwertige Bit des A-Registers eingeschrieben ($A \leftarrow \circ_A_{n-1} \ldots A_1, B \leftarrow S_B_{n-1} \ldots B_1$). Die Summe steht im B-Register und der letzte Übertrag (carry out) im C-Register.

   Die Schritte 2 und 3 werden n mal wiederholt.

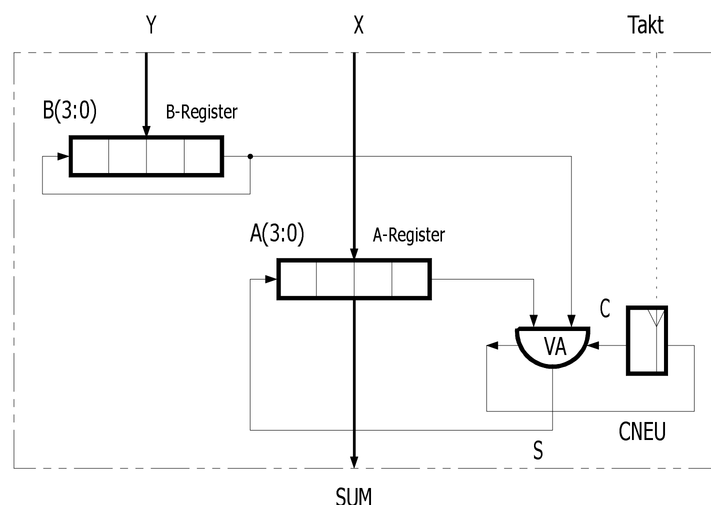


Abbildung 3.8: Serielles Addierwerk für Dualzahlen

In der Abb. 3.9 ist dieser Mikroalgorithmus in Form eines Zustandsdiagramms mit den Sprachmitteln von HDL dargestellt. Die Registerzuweisungen erfolgen gleichzeitig mit den Zustandsänderungen, synchronisiert durch den Takt.

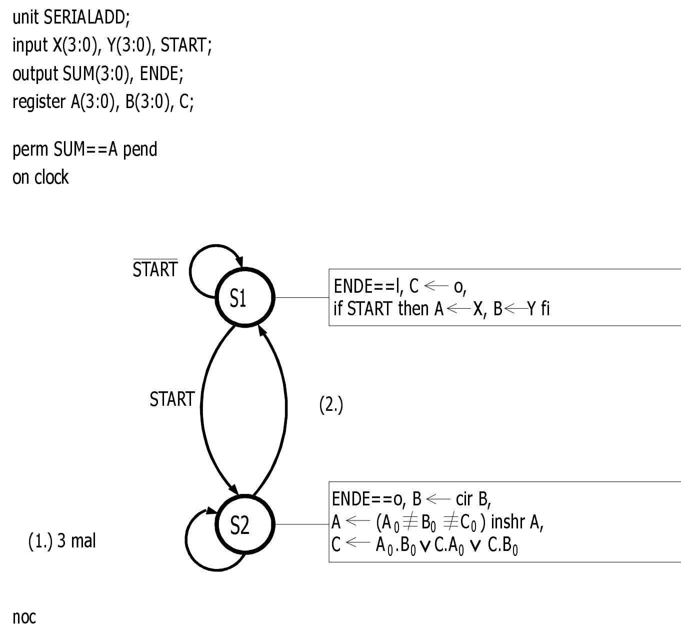


Abbildung 3.9: Mikroprogramm für die serielle Addition

3.1.1.6 Akkumulation von Dualzahlen

Die Akkumulation von m Dualzahlen erfolgt am einfachsten durch das Rechenwerk nach Abb. 3.10. Die (Zwischen)Summe wird in den taktgesteuerten Speichern $S1, S2, \ldots$ gespeichert. Zwei Varianten können unterschieden werden, je nachdem, ob für die Überträge taktgesteuerte Speicher vorgesehen werden oder nicht. Sind keine Speicher für die Überträge vorhanden, dann wird mit jedem Taktschritt ein neuer Summand zu der Zwischensumme addiert ($S \leftarrow S + X$). Das Taktintervall muß mindestens $n * T$ groß sein, bei n Stellen und der Verzögerungszeit T durch einen Volladdierer. Bei m Summanden beträgt die Rechenzeit $t = m * n * T$. (Das Summenregister S sei am Anfang gelöscht.) Werden dagegen für die Überträge Speicher vorgesehen, so kann das Taktintervall auf T verringert werden. Dabei werden die Überträge mit jedem Takt nur um einen Schritt nach links geschoben. Nach der Verarbeitung der m Summanden werden noch maximal $(n-1)$ Taktschritte zur Verarbeitung der noch in den Speichern stehenden Überträge benötigt. Die gesamte Rechenzeit beträgt somit $t = m * T + (n-1) * T$. Sie ist damit für $m > 1$ immer kleiner als bei der 1. Variante. Die 2. Variante ist ein Beispiel für ein Fließband-(Pipeline-)Rechenwerk: Die Verarbeitung des Übertrags erfolgt schrittweise, aber gleichzeitig für mehrere Summanden in verschiedenen Bearbeitungsstufen. Durch eine Unterteilung in Bearbeitungsstufen läßt sich bei langen Vektoren die gesamte Rechenzeit erheblich

verkürzen.

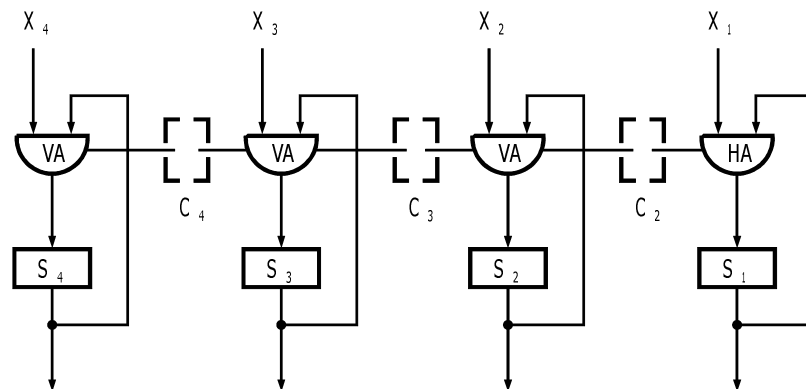


Abbildung 3.10: Rechenwerk zur Akkumulation von Dualzahlen

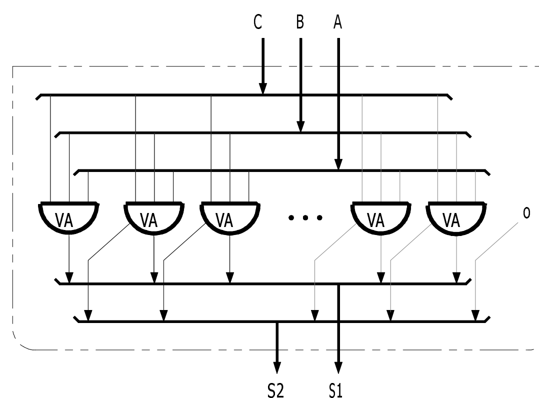


Abbildung 3.11: Carry-Save-Addierer (CSA)

Eine weitere Verkürzung der Rechenzeit läßt sich durch spezielle Additionsschaltnetze erzielen. Als Grundbaustein dient der sogenannte *Carry-Save-Addierer* (CSA), der aus drei Summanden zwei Teilsummen (Abb. 3.11) bildet. Der Carry-Save-Addierer besteht aus n Volladdierern. Die Summenbits werden zu der Teilsumme $S1$ zusammengefaßt und die Übertragsbits zu der Summe $S2$. Die Übertragsbits werden also nicht wie beim Carry-Ripple-Addierer sofort weitergeleitet, sondern werden als $S2$ „gerettet“ und müssen in den darauffolgenden Addierstufen verarbeitet werden. Dadurch, daß die Überträge nicht intern weitergeleitet werden, verursacht jeder CSA nur die Laufzeit T, unabhängig von der Anzahl der Stellen n.
Eine lineare Schaltkette zur Akkumulation von n Dualzahlen besteht aus $m - 2$ hintereinander geschalteten CSA und einem abschließenden Addierer, der die 2 letzten Summen verarbeitet (Abb. 3.12). Die erste Stufe verarbeitet 3 Summanden, jede weitere einen Summanden. Gegenüber $m - 1$ hintereinandergeschalteten Carry-Ripple-Addierern ergibt sich nur dann ein Geschwindigkeitsvorteil, wenn die Abschlußaddition weniger als $n * T$ benötigt. Deshalb muß für

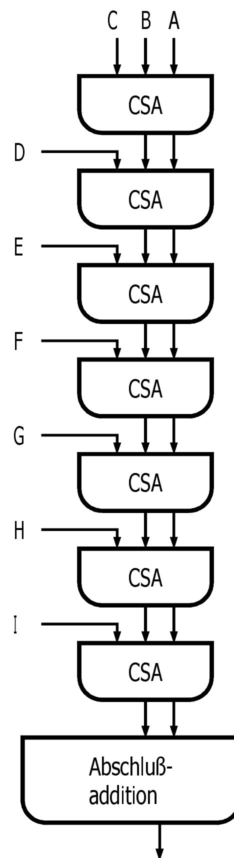


Abbildung 3.12: Akkumulation von 9 Summanden mit Carry-Save-Addierern

die Abschlußaddition ein schneller Addierer eingesetzt werden, z. B. ein Carry-Look-Ahead-Addierer.
Die Verarbeitungsgeschwindigkeit läßt sich weiter erhöhen, wenn mehrere CSA in einer Stufe parallel arbeiten. Abb. 3.13 zeigt ein Beispiel für einen sogenannten Additionsbaum (adder tree) nach WALLACE [Wal] für $m = 9$ Summanden. Die Anzahl der CSA beträgt wie bei der linearen Schaltkette $m - 2 = 7$, die Anzahl der CSA-Stufen hat sich aber in dem Beispiel auf 4 reduziert, wodurch sich die Rechenzeit entsprechend verringert. Bei einer anderen Anzahl von Summanden lassen sich ähnliche Additionsbäume konstruieren. Weitergehende Betrachtungen darüber sind in [Wal] und [Gil] zu finden. Das CSA-Prinzip kann auch in Multiplikations- und Divisionsschaltnetzen eingesetzt werden, in denen Zwischenergebnisse akkumuliert werden.

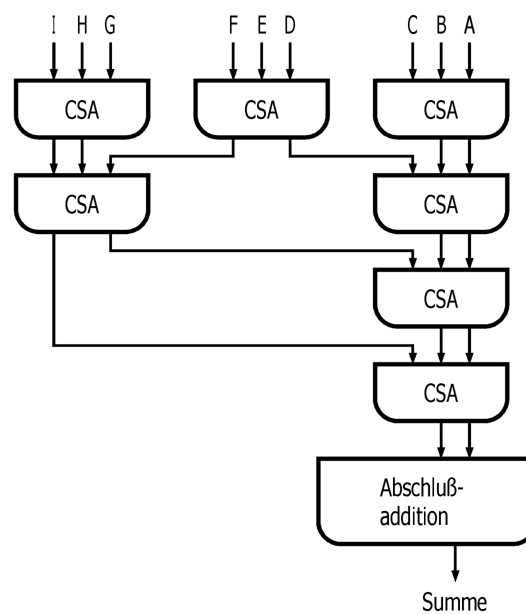


Abbildung 3.13: Additionsbaum nach WALLACE für die Akkumulation von 9 Summanden

3.1.2 Subtraktion allgemein

Die Subtraktion von zwei positiven Dualzahlen läßt sich direkt mit Hilfe von Vollsubtrahierern ausführen. Ein Vollsubtrahierer bildet die Differenz der Bits $A_i - B_i - C_i$, die die Werte $+1, 0, -1, -2$ annehmen kann. Sie wird durch das Differenzbit D_i mit der Wertigkeit 1 und dem Borgebit C_{i+1} mit der Wertigkeit -2 dual codiert. Die logischen Gleichungen lauten:

$$D_i = A_i \not\equiv B_i \not\equiv C_i \tag{3.13}$$

$$C_{i+1} = B_i.C_i \lor \overline{A_i}.B_i \lor \overline{A_i}.C_i \,. \tag{3.14}$$

Durch Hintereinanderschalten von Vollsubtrahierern und Weiterleitung der Borgebits entsteht ein Subtraktionsschaltnetz. Es ist aber auch möglich, ein gegebenes Additionsschaltnetz für die Subtraktion zu benutzen. Dazu werden alle B_i negiert, der „carry in“-Eingang auf `1` gesetzt und der entstehende Übertrag 2^n ignoriert (subtrahiert), denn es gilt $B + \overline{B} = 2^n - 1$ und $A - B = (A + \overline{B} + 1) - 2^n$. Entsteht kein Übertrag, so ist $B > A$, so daß auf diese Weise auch ein Größenvergleich vorgenommen werden kann. Oft wird die Subtraktion nicht direkt realisiert, sondern auf die Addition eines negativen Summanden zurückgeführt: $a - b = a + (-b)$. Voraussetzung dafür ist ein Additionswerk, das negative Zahlen verarbeiten kann sowie die Operation „Vorzeichenwechsel“ $b \rightarrow -b$ für die gewählte Zahlendarstellung. Bei der Subtraktion in der gebräuchlichsten Zweikomplementdarstellung wird der notwendige Vorzeichenwechsel durch eine Komplementierung gegen 2^n erreicht, d. h. die Zweikomplementzahl B muß in $2^n - B = \overline{B} + 1$ umgeformt werden. Bei der Subtraktion in der Einskomplementdarstellung muß das Komplement gegen $2^n - 1$ gebildet werden, d. h. die Einskomplementzahl B muß in $\overline{B}$ umgeformt werden.

Die Subtraktion $A - B$ von m-stelligen Dezimalzahlen wird auf die Addition des Komplements von B gegen 10^m (Zehnerkomplement) oder gegen $10^m - 1 = 99 \ldots 9$ (Neunerkomplement) zurückgeführt (vergl. Komplementbildung bei BCD-Zahlen, Abschnitte 1.5 und 3.1.7).

3.1.3 Addition von Vorzeichenzahlen

Die Addition von Vorzeichenzahlen erfordert eine getrennte Berechnung für das Vorzeichen und den Betrag der Summe. Sie beinhaltet die Subtraktion, indem das Vorzeichen des zweiten Summanden gewechselt wird. In Abhängigkeit von den Vorzeichen und den Beträgen der Summanden müssen 4 Fällen unterschieden werden :

	Bedingung	Summe	Vorzeich. V_s
1.	$\lvert x\rvert + \lvert y\rvert \leq 2^n - 1,\ V_x = V_y$	$S[n] = X + Y$	$V_x = V_y$
2.	$\lvert x\rvert + \lvert y\rvert > 2^n - 1,\ V_x = V_y$	$S[n+1] = X + Y$	$V_x = V_y$
3.	$\lvert x\rvert > \lvert y\rvert,\quad V_x \neq V_y$	$S[n] = X - Y$	V_x
4.	$\lvert x\rvert < \lvert y\rvert,\quad V_x \neq V_y$	$S[n] = Y - X$	V_y

Im 2. Fall wird eine zusätzliche Stelle zur Darstellung des Ergebnisses benötigt und die Überlaufbedingung *OV* muß gesetzt werden. Aufwendig ist die Abfrage auf größer/kleiner im 3. und

4. Fall. Diese Abfrage kann umgangen werden, indem zuerst für $V_x \neq V_y$ probeweise der Fall 3 durchgeführt wird. Ergibt sich bei der Subtraktion ein negatives Ergebnis – was bei der Durchführung im Zweikomplement leicht feststellbar ist (kein Übertrag) –, dann wird das Ergebnis komplementiert. Die vier Fälle werden durch folgenden Algorithmus erfaßt:

if $V_x = V_y$ then $[S := X + Y$, *OV*$:= C,\ V_s := V_x = V_y]$;
  if *OV* then *Summe* $= S[n+1]$
    else *Summe* $= S[n]$ fi
fi,
if $V_x \neq V_y$ then $[S := X - Y = X + \overline{Y} + 1,\ V_s := 0]$;
  if C then „$S \geq 0$“ *Summe* $= S[n]$,
    else „$S < 0$“ [*Summe* $= -S[n] = (\overline{S}+1)[n]$,
      V_s :=1] fi
fi

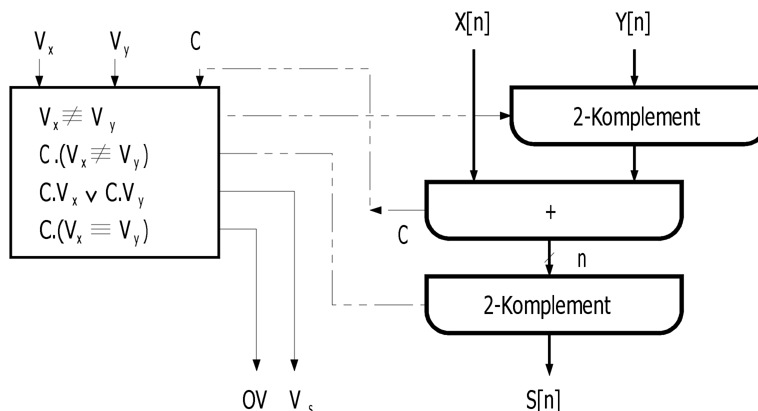


Abbildung 3.14: Rechenwerk für die Addition von Vorzeichenzahlen

Wenn die Darstellung der negativen Null (Vorzeichen = 1, Betrag = 0) verboten werden soll, dann muß durch eine Sonderbehandlung verhindert werden, daß eine negative Null bei $V_x \neq V_y$ und $|x| = |y|$ entstehen kann.

Abb. 3.14 zeigt eine mögliche Realisierung, wobei die Subtraktion auf die Addition im Zweikomplement zurückgeführt wird. Für $V_x = V_y$ entspricht der Übertrag C (carry out) der Stelle S_{n+1} und der Überlaufbedingung *OV*, und für $V_x \neq V_y$ entspricht $C = \circ$ der Bedingung $X - Y < 0$. An dem Rechenwerk bzw. einem äquivalenten Algorithmus wird deutlich, daß diese Zahlendarstellung zu aufwendigen Realisierungen führt. Deshalb wird die Addition und Subtraktion von Festkommazahlen in praktisch allen Rechenanlagen im 2-, 1-, 10- oder 9-Komplement durchgeführt, weil dann das Rechenwerk durch ein einfaches Additionsschaltnetz ersetzt werden kann. Nur bei der Addition/Subtraktion von Gleitkommazahlen mit Vorzeichen-Betrag-Darstellung muß dieser oder ein ähnlicher Algorithmus verwendet werden.

3.1.4 Addition von Zweikomplementzahlen

Gegeben seien die beiden Zweikomplementzahlen $X[n]$ und $Y[n]$, gesucht ist die Summe in Zweikomplementdarstellung. Zur Ableitung eines geeigneten Algorithmus verwenden wir die Rückabbildungsgleichung (1.8: $x = X[n] - X_n * 2^n$) für x und y:

$$s = x + y = X[n] - X_n * 2^n + Y[n] - Y_n * 2^n \ . \tag{3.15}$$

Je nach Größe und Vorzeichen von x und y werden zur Darstellung der Summe n bzw. $n+1$ Stellen benötigt:

$$s = S[n] - S_n * 2^n \tag{3.16}$$

$$s = S[n+1] - S_{n+1} * 2^{n+1} \ . \tag{3.17}$$

Durch Gleichsetzung von (3.15) mit (3.16) bzw. (3.17) ergibt sich die Summe in Zweikomplementdarstellung

$$S[n] = X[n] + Y[n] \quad \text{„kein Überlauf“} \\ + (S_n - X_n - Y_n) * 2^n \tag{3.18}$$

$$S[n+1] = X[n] + Y[n] \quad \text{„Überlauf“} \\ + (2S_{n+1} - X_n - Y_n) * 2^n \ . \tag{3.19}$$

Aus diesen beiden Gleichungen lassen sich in Abhängigkeit von X_n, Y_n, S_n und S_{n+1} sechs verschiedene Fälle entwickeln. Eine gewisse Schwierigkeit ist die Tatsache, daß S_n bzw. S_{n+1} auf beiden Seiten der Gleichungen vorkommt. Dazu überlegen wir uns, daß die Bits S_{n+1} und S_n in Abhängigkeit von den Vorzeichen X_n, Y_n und der Größe der Summe s nur bestimmte Werte annehmen können, z. B. aus $X_n = 0$ und $Y_n = 0$ und $s \leq 2^{n-1} - 1$ folgt $S_n = 0$. Setzt man diese Abhängigkeiten zwischen X_n, Y_n, S_{n+1}, S_n in die Gleichungen (3.18) und (3.19) ein, so ergibt sich die Tabelle Abb. 3.15.

Fall	X_n	Y_n	S_{n+1}	S_n	Summe
a1	0	0	(0)	0	$S[n] = X + Y$
a2	0	0	0	1	$S[n+1] = X + Y$ „Überlauf“
b1	1	1	(1)	1	$S[n] = X + Y - 2^n$
b2	1	1	1	0	$S[n+1] = X + Y$ „Überlauf“
c1	0	1	(1)	0	$S[n] = X + Y - 2^n$
c2	0	1	(0)	1	$S[n] = X + Y$

Abbildung 3.15: Die verschiedenen Fälle der Addition im Zweikomplement

In den Fällen a2 und b2 findet eine Bereichsüberschreitung (Überlauf, Overflow) statt, da das Ergebnis nicht mehr mit n Stellen darstellbar ist. Die hinzukommende Stelle S_{n+1} kann dann

```
                                C  OV                              C  OV
a1)  +2      o.o l o                     a2)  +7      o.l l l
     +2      o.o l o                          +7      o.l l l
     +4    o o.l o o    o o                   +14   o.l l l o    o l

b1)  -1      l.l l l                     b2)  -7      l.o o l
     -1      l.l l l                          -7      l.o o l
     -2    l l.l l o    l o                   -14   l.o o l o    l l

c1)  +2      o.o l o                     c2)  +1      o.o o l
     -1      l.l l l                          -7      l.o o l
     +1    l o.o o l    l o                   -6    o l.o l o    o o
```

C = Carry, OV = Overflow

Abbildung 3.16: Typische Zahlenbeispiele für die Addition im Zweikomplement

als Vorzeichenstelle interpretiert werden. Die Stelle S_{n+1} entspricht dem Übertrag (carry out). Der Übertrag, der in den Fällen b1 und c1 auftritt, gehört nicht zur Darstellung des n-stelligen Ergebnisses und wird deshalb abgeschnitten $(X + Y - 2^n = (X + Y) \bmod 2^n)$.

Es trägt sehr zum Verständnis bei, wenn man sich die sechs typischen Fälle anhand von Beispielen (Abb. 3.16) klar macht.

Der große Vorteil der Addition im Zweikomplement liegt darin, daß alle Fälle in zwei Schritten zusammengefaßt werden können:

1. $S[n] = X + Y$ bilden und den Übertrag S_{n+1}merken. Die Überlaufbedingung berechnen.
2. Wenn kein Überlauf aufgetreten ist (*OV*=o), dann ist die Summe gleich $S[n]$, ansonsten gleich $S[n+1]$.

Die Addition im Zweikomplement führt dadurch zu einem sehr einfachen Rechenwerk (Abb. 3.17). Der Überlauf berechnet sich nach der Beziehung

$$OV = (X_n \equiv Y_n).(H_{n+1} \not\equiv H_n)\ . \qquad (3.20)$$

Dabei sind $H_i (i = 1, \ldots, n)$ die Ausgänge des Additionsschaltnetzes mit $H_{n+1} = C_{n+1}$ (Abb. 3.17). Wenn die Überträge C_i des Additionsschaltnetzes verfügbar sind, dann kann auch die einfachere Funktion $OV = (C_{n+1} \not\equiv C_n)$ verwendet werden.
Es ist nicht unbedingt erforderlich, den Überlauf parallel zur Addition zu berechnen. Er kann auch nachträglich aus X_n, Y_n, S_n berechnet werden. So eignet sich das folgende Mikroprogramm für die softwaremäßige Implementierung:

```
register X[n], Y[n], S[n+1], OV;
signal H[n+1];
perm H == X++Y pend "Additionsschaltnetz"
on clock

[1]        S <— H, OV <— (Xn ≡ Yn ).(Hn+1 ≢ Hn ), next 2

[2]        "if OV=o  then  Summe = S[n]
                     else  Summe = S[n+1] fi"

noc
```

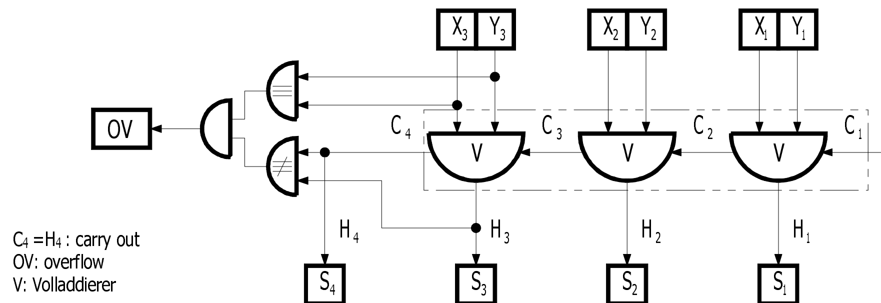


Abbildung 3.17: Zweikomplement-Addition und Rechenwerk für n = 3

boole $X[n], Y[n], S[n]$;
$S := X + Y$;
if $(X_nY_nS_n = \texttt{oo1}) \vee (X_nY_nS_n = \texttt{11o})$ then *OV* $:= \texttt{1}$ fi
„if *OV* then *Summe* $= \overline{S_n}_S = X_n_S$, evtl. Rechtsschift
        else *Summe* $= S$ fi“

Wenn bei der Addition ein Überlauf aufgetreten ist, dann reichen die vorgesehen n Stellen nicht mehr aus. Wenn man weiter rechnen will, dann muß man die Summe $S[n+1] = \overline{S_n}\,S[n]$ um eine Stelle nach rechts verschieben (Division durch 2) und sich merken, daß das Ergebnis doppelt so groß geworden ist. Verallgemeinert man dieses Prinzip, so gelangt man zur Addition im Gleitkomma, die im Abschnitt 3.1.8 beschrieben wird. Es gibt noch eine andere einfache Möglichkeit, auf den möglichen Überlauf zu reagieren, indem eine sogenannte Schutzstelle (realisiert z. B. im Apollo-Guidance-Computer) vorgesehen wird. Dabei wird der Wertebereich von x und y beschränkt, indem vor der Addition $X_n = X_{n-1}$ und $Y_n = Y_{n-1}$ vorausgesetzt werden, d. h. jeder Summand besitzt zwei gleiche Vorzeichenbits. Nach der Addition läßt sich der Überlauf an der Bedingung $S_n \neq S_{n-1}$ erkennen.

Es soll noch einmal daran erinnert werden, daß das Zweikomplement den Nachteil hat, daß der darstellbare Zahlenbereich im Negativen um Eins größer ist als im Positiven. Die Algorithmen für die Komplementbildung, die Multiplikation und Division vereinfachen sich wesentlich, wenn die betragsmäßig größte negative Zahl im Zweikomplement ($\texttt{1.oo...o}$) verboten wird. Das kann dadurch geschehen, daß beim Auftreten dieser Zahl eine Fehlerbedingung oder die Überlaufanzeige *OV* gesetzt wird.

3.1.5 Addition von Einskomplementzahlen

Es soll die Summe von zwei Einskomplementzahlen gebildet werden. Unter Verwendung der Rückabbildungsgleichung (1.14) für x und y ergibt sich

$$\begin{aligned} s \ = x + y = \ & X[n] - X_n * (2^n - 1) \\ & + Y[n] - Y_n * (2^n - 1) \,. \end{aligned} \tag{3.21}$$

Zur Darstellung der Summe werden n Stellen benötigt, wenn keine Bereichsüberschreitung auftritt, und $n+1$ Stellen, wenn sie auftritt:

$$s = S[n] - S_n * (2^n - 1) \tag{3.22}$$

$$s = S[n+1] - S_{n+1} * (2^{n+1} - 1) \,. \tag{3.23}$$

Durch Gleichsetzen von (3.21) mit (3.22) und (3.23) erhalten wir die Summe in Einskomplementdarstellung:

$$\begin{aligned} S[n] &= X[n] + Y[n] \quad \text{„kein Überlauf“} \\ & \quad + (S_n - X_n - Y_n) * (2^n - 1) \end{aligned} \tag{3.24}$$

$$\begin{aligned} S[n+1] &= X[n] + Y[n] \quad \text{„Überlauf“} \\ & \quad + (2S_{n+1} - X_n - Y_n) * (2^n - 1) S_{n+1} \,. \end{aligned} \tag{3.25}$$

Bevor wir aus diesen Beziehungen die verschiedenen Fälle formal ableiten, veranschaulichen wir uns die 6 typischen Fälle (Tabelle Abb. 3.18).

Aus diesen Beispielen können wir entnehmen, daß wir das richtige Ergebnis nur dann erhalten, wenn in den Fällen b1, b2 und c1 eine Eins hinzuaddiert wird. Diese Einserkorrektur (end around carry) muß genau dann durchgeführt werden, wenn der Übertrag (carry out) $S_{n+1} =$ `1` geworden ist. Diese Aussagen werden formal bestätigt, indem in den Gleichungen (3.24) und (3.25) die gültigen Kombinationen von X_n, Y_n, S_{n+1}, S_n eingesetzt werden (Tabelle Abb. 3.19). Die Subtraktion von 2^n in den Fällen `b1, c1` wird einfach durch Abschneiden auf n Stellen realisiert: $X + Y + 1 - 2^n = (X + Y + 1) \bmod 2^n$.

Wie bei der Addition im Zweikomplement wird auch hier nur ein einfaches Additionsschaltnetz benötigt (Abb. 3.20). Der herausgehende Übertrag (carry out) wird einfach mit dem hereingehenden Übertrag (carry in) verbunden, wodurch die Einserkorrektur automatisch durchgeführt wird. Allerdings ist mit dieser Rückkopplung eine zusätzliche Zeitverzögerung verbunden. Wenn wir vor der Wahl stehen, für ein Rechenwerk entweder das Eins- oder das Zweikomplement zu verwenden, dann werden wir uns im allgemeinen für das Zweikomplement entscheiden, weil dann ein normales Additionschaltnetz verwendet werden kann, keine negative Null stört und keine Einserkorrektur notwendig ist. Bei der seriellen Addition, bei der in jedem Schritt ein Summenbit und ein Übertragsbit berechnet werden, würde die Einserkorrektur zu einem zweiten Durchlauf führen. Allerdings darf man nicht außer Betracht lassen, daß im Zweikomplement die betragsmäßig größte negative Zahl eine Sonderbehandlung erfordert. Im Einskomplement gestalten sich dagegen Multiplikation, Division und Komplementbildung einfacher.

```
                               C  OV                            C  OV
a1)  +2        o.o l o                a2)  +7       o.l l l
     +2        o.o l o                     +7       o.l l l
     +4      o o.l o o     o  o           +14     o.l l l o     o  l

b1)  -1        l.l l o                b2)  -7       l.o o o
     -1        l.l l o                     -7       l.o o o
     -3      l l.l o o                    -15     l.o o o o
     +1              l                     +1             l
     -2      l l.l o l     l  o           -14     l.o o o l     l  l

c1)  +2        o.o l o                c2)  +1       o.o o l
     -1        l.l l o                     -7       l.o o o
      0      l o.o o o                     -6       l.o o l     o  o
     +1              l
     +1      l o.o o l     l  o
```

C = Carry, OV = Overflow

Abbildung 3.18: Typische Zahlenbeispiele für die Addition im Einskomplement

Fall	X_n	Y_n	S_{n+1}	S_n	Summe
a1	o	o	(o)	o	$S[n] = X + Y$
a2	o	o	o	l	$S[n+1] = X + Y$ „Überlauf“
b1	l	l	(l)	l	$S[n] = X + Y + 1 - 2^n$
b2	l	l	l	o	$S[n+1] = X + Y + 1$ „Überlauf“
c1	o	l	(l)	o	$S[n] = X + Y + 1 - 2^n$
c2	o	l	(o)	l	$S[n] = X + Y$

Abbildung 3.19: Die verschiedenen Fälle der Addition im Einskomplement

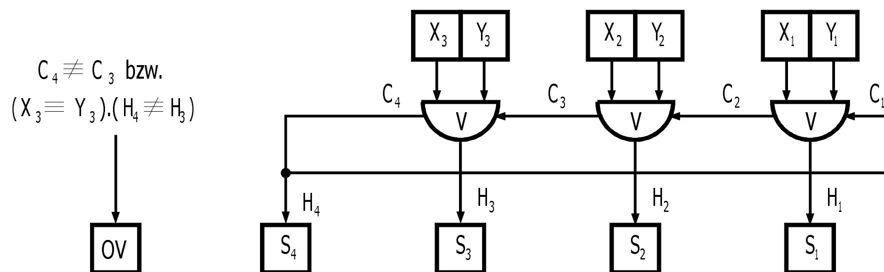


Abbildung 3.20: Rechenwerk für die Einskomplement-Addition für n = 3 Stellen

3.1.6 Doppelwort-Addition

Angenommen wir besitzen ein Rechenwerk, das 16-stellige Dualzahlen addieren kann. Falls wir eine 32-Bit-Addition benötigen, müssen wir sie auf die 16-Bit-Addition zurückführen. Dabei werden zuerst die niederwertigen 16 Bits addiert und der Übertrag zwischengespeichert. Anschließend werden die höherwertigen 16 Bits unter Berücksichtigung des vorhergehenden Übertrags addiert. Dieses Prinzip kann sowohl auf positive Dualzahlen als auch auf Zweikomplementzahlen angewandt werden, und es kann entsprechend auf mehr als zwei Worte erweitert werden (Mehrwort-Addition).

Das folgende (Mikro-)Programm in der Sprache HDL beschreibt die Addition der Zweikomplementzahlen $X = X2_X1$ plus $Y = Y2_Y1$.

boole *OV*, $(X2, X1, Y2, Y1)[n], (S2, S1)[n+1]$;
$S1 := X1 +\!\!+ Y1$; „$S1_{n+1}$ = *CARRY*1"
$S2 := X2 +\!\!+ Y2 + S1_{n+1}$; „$S2_{n+1}$ = *CARRY*2"
OV $:= (X2_n \equiv Y2_n).(S2_{n+1} \not\equiv S2_n)$
$\vee(S2[n]_S1[n] = \mathtt{1oo...o})$ „Sonderfall"
„if *OV* = o then *Summe* $= S2[n]_S1[n]$
else *Summe* $= S2[n+1]_S1[n]$ fi"

Die Überlaufbedingung wird zweckmäßigerweise auch dann gesetzt, wenn die kleinste darstellbare Zahl ($S = \mathtt{1oo...o}$) entsteht.

3.1.7 Addition von binärcodierten Dezimalzahlen

Die Addition von binärcodierten Dezimalzahlen wird auf die dezimale Addition der einzelnen BCD-Ziffern $X\langle i\rangle$ und $Y\langle i\rangle$ unter Berücksichtigung der Überträge zurückgeführt. Die dezimale Addition der BCD-Ziffern wird meist auf die *duale* Addition (Addition von Dualzahlen) zurückgeführt, wobei eine Korrektur mit +6 durchgeführt werden muß, falls eine unerlaubte Ziffer > 9 entsteht.

$Z[5]$		$S[5]$	
0	o oooo	0	o oooo
1	o oool	1	o oool
2	o oolo	2	o oolo
3	o ooll	3	o ooll
4	o oloo	4	o oloo
5	o olol	5	o olol
6	o ollo	6	o ollo
7	o olll	7	o olll
8	o looo	8	o looo
9	o lool	9	o lool

$Z[5]$		$S[5]$	
10	o lolo	16	l oooo
11	o loll	17	l oool
12	o lloo	18	l oolo
13	o llol	19	l ooll
14	o lllo	20	l oloo
15	o llll	21	l olol
16	l oooo	22	l ollo
17	l oool	23	l olll
18	l oolo	24	l looo
19	l ooll	25	l lool

Abbildung 3.21: Dualzahl $Z[5]$ und BCD-Zahl $S[5]$

Die duale Addition von zwei BCD-Ziffern und dem Übertrag C aus der vorhergehenden Ziffern-Addition ergibt die fünfstellige duale Zwischensumme

$$Z[5] = X\langle i\rangle \;\text{++}\; Y\langle i\rangle + C\,, \tag{3.26}$$

die zwischen 0 und 19 liegen kann (Tabelle Abb. 3.21). Gesucht ist aber die Zwischensumme in BCD-Darstellung $S[5]$, die sich aus $Z[5]$ durch Korrektur mit $+6$ ergibt, wenn $Z[5] > 9$ ist:

$$S[5] \;=\; Z[5] + 6 * (Z[5] > 9)\,. \tag{3.27}$$

Dabei stellt das Bit S_5 den neuen Übertrag in die nächste BCD-Stelle dar und ist ein Kennzeichen für $Z[5] > 9$. Dieses 1. Verfahren läßt sich formal wie folgt beschreiben:

boole $(X, Y, S)[4n], Z[5]$;	„$Z_5 = C =$ Übertrag“
$Z_5 := 0$;	„Übertragsspeicher löschen“
for $i := 1$ to n do	
$Z := X\langle i\rangle \text{++} Y\langle i\rangle + Z_5$;	„Duale Addition“
if $Z >$ `lool` then $Z := Z+$`ollo` fi;	„Korrektur“
$S\langle i\rangle := Z[4]$	
od	

Dabei sind die 4 Bits der i-ten BCD-Stelle $(4i : 1 + 4(i-1))$ durch $\langle i\rangle$ gekennzeichnet. Seriell läßt sich die BCD-Addition mit Hilfe von zwei 4 Bit breiten Schieberegistern und einem Dezimaladdierer durchführen (Abb. 3.22).

Zahlenbeispiel: $98 + 47 = 145$

```
  lool looo
+ oloo olll
```

```
        llll      > 9
        ollo      + 6      Korrektur
      l olol
   lllo           > 9
   ollo           + 6      Korrektur
 l oloo olol      = 145
```

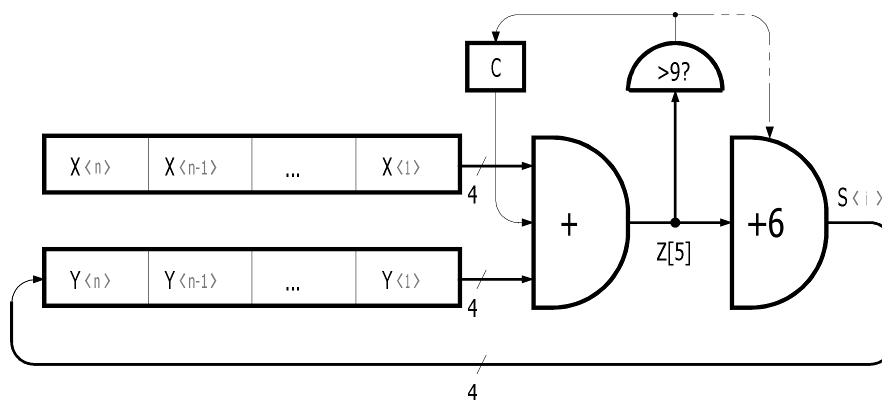


Abbildung 3.22: Serielle Addition von BCD-Zahlen

Das 1. Verfahren hat zwei Nachteile: (1) die Abfrage > 9 ist hardwaremäßig oder softwaremäßig relativ aufwendig, und (2) die BCD-Ziffern müssen nacheinander korrigiert werden.

Deshalb wird die Abfrage > 9 meist auf die Abfrage > 15 zurückgeführt, indem auf die duale Zwischensumme probeweise $+6$ addiert wird. Entsteht ein Übertrag (der dem Dezimalübertrag entspricht), so ist die duale Zwischensumme > 9 und die Korrektur war richtig. Entsteht kein Übertrag, so muß die Korrektur durch Subtraktion von 6 rückgängig gemacht werden. Formal ergibt sich der Dezimalübertrag zu

$$(Z[5] > 9) \quad = \quad ((Z[5] + 6) > 15) = (Z[5] + 6)_5 \; . \tag{3.28}$$

Damit ergibt sich aus (3.27)

$$\begin{aligned} S[5] &= Z[5] + 6 * (Z[5] + 6)_5 \\ S[5] &= (Z[5] + 6) - 6 * \overline{(Z[5] + 6)_5} \; . \end{aligned} \tag{3.29}$$

Das 2. Verfahren lautet somit für ein paralleles Rechenwerk:

1. Zu jeder BCD-Ziffer $X\langle i\rangle$ des ersten Summanden +6 addieren. Überträge zwischen den Tetraden (4-Bit-Gruppen) können nicht auftreten.

2. Dazu den zweiten Summanden dual addieren: $S := (X + 66\ldots6) + Y$. Alle Überträge zwischen den Tetraden für den 3. Schritt merken.

3. Von jeder BCD-Stelle, die keinen Übertrag erzeugt hat, 6 abziehen (durch Addition von $-6 =$ `lolo` in Zweikomplementdarstellung). Dabei dürfen die entstehenden Überträge zwischen den Tetraden nicht weitergeleitet werden.

Ein Rechenwerk für diesen Algorithmus erfordert, neben der dualen Addition, für jede Tetrade einen Übertragsspeicher, um in Abhängigkeit davon die Korrektur im 3. Schritt durchführen zu können. Außerdem muß zur Durchführung des 3. Schrittes die Weiterleitung des Übertrages nach jeder Tetrade unterbrechbar sein.

Zahlenbeispiel: $18 + 47 = 65$

oool	looo	1	8	
+ ollo	ollo	+6	+6	
olll	lllo	7	14	
oloo	olll	+4	+7	
l	olol	+1	5	$14 + 7 = 21 = 15_{16}$
o lloo		12		
lolo		+10		entspricht -6
(l) ollo	olol	(1) 6		$12 + 10 = 22 = 16_{16}$
ollo	olol			$= 65$

Das 3. hier geschilderte Verfahren besteht aus zwei Teilen: duale Addition (1.) und BCD-Korrekturaddition (2. und 3.).

1. Addiere die Summanden dual ($Z := X + Y$) und merke alle Überträge C_i zwischen den Tetraden.

2. Addiere testweise zu allen Tetraden dual $+6$ ($T := Z + 66\ldots6$) und merke alle neuen Überträge D_i zwischen den Tetraden.

3. Ist entweder ein Übertrag C_i oder ein Übertrag D_i entstanden, so wird $T\langle i\rangle$ als gültige Summenziffer übernommen, andernfalls $Z\langle i\rangle$ plus D_{i-1} (if $C_i \vee D_i$ then $S\langle i\rangle := T\langle i\rangle$ else $S\langle i\rangle := Z\langle i\rangle + D_{i-1}$ fi).

Ein Additionswerk für BCD-Zahlen eignet sich gleichzeitig zur Addition von 10-Komplement-Zahlen. Der Wert der Summe darf dann zwischen $-50 \ldots 0$ und $49 \ldots 9$ schwanken, ohne daß eine Bereichsüberschreitung auftritt. Eine Bereichsüberschreitung kann daran erkannnt werden, daß bei gleichen Vorzeichen der Summanden durch die Addition ein Vorzeichenwechsel hervorgerufen wird. Das Vorzeichen ist implizit in der höchstwertigen BCD-Ziffer enthalten (vergleiche Abschnitt 1.5).

Die Subtraktion von zwei positiven BCD-Zahlen (bei der Vorzeichen-Betrag-Darstellung) kann auf die Addition des 10-Komplements zurückgeführt werden:

1. Addiere zum ersten Operanden den negierten zweiten Operanden plus Eins. Merke die Übertäge in die nächsten BCD-Stellen (nur für den 2. Schritt).

2. Subtrahiere 6 von allen BCD-Stellen, die keinen Übertrag erzeugt haben. Die Subtraktion wird auf die Addition von -6 im 2-Komplement (`1o1o`) zurückgeführt, wobei die Überträge nicht weitergeleitet werden dürfen. Merke den Übertrag aus der höchstwertigen BCD-Stelle.

3. Wenn ein Übertrag aus der höchstwertigen BCD-Stelle entstanden ist, dann ist das Ergebnis negativ in 10-Komplement-Darstellung. Wenn eine Vorzeichen-Betrag-Darstellung gewünscht wird, dann muß das Vorzeichen negativ gesetzt werden und das Ergebnis muß rückkomplementiert werden. Zur Bildung des 10-Komplements wird zuerst das 2-Komplement durch Negation und Addition von +1 gebildet. Von allen BCD-Stellen, die keinen Übertrag erzeugt haben, muß 6 abgezogen werden (durch Addition von `1o1o`).

3.1.8 Addition von Gleitkommazahlen

Im Gegensatz zur Addition von Festkommazahlen ist die Addition von Gleitkommazahlen wesentlich komplizierter, weil die Exponenten und Mantissen getrennt behandelt werden müssen. Gesucht ist die normalisierte Mantisse ms und der Exponent es der Summe

$$s \;=\; x + y = ms * b^{es} = mx * b^{ex} + my * b^{ey} \;. \tag{3.30}$$

Bevor die Addition der Mantissen vorgenommen werden kann, muß der kleinere Exponent an den größeren angepaßt werden. Dazu muß die Mantisse, die zu dem kleineren Exponenten gehört, um so viele Stellen nach rechts geschoben werden, wie die Differenz der Exponenten beträgt.

$$ms * b^{es} \;=\; \begin{cases} (mx + my * b^{-(ex-ey)}) * b^{ex} & ey \leq ex \\ (my + mx * b^{-(ey-ex)}) * b^{ey} & ex \leq ey \;. \end{cases} \tag{3.31}$$

Im Anschluß an die Mantissen-Addition wird die Summe meist normalisiert. Besitzt die (positive) Mantisse ms k führende Nullen, so wird die Operation

$$\begin{aligned} ms &:= ms * b^k && \text{„Linksschift um } k \text{ Stellen" und} \\ es &:= es - k && \text{„Exponent erniedrigen"} \end{aligned} \tag{3.32}$$

durchgeführt. Beim Linksschift der Mantisse werden von rechts Nullen nachgezogen. Um den mittleren Rundungsfehler zu verringern, kann man auch einmal die Ziffer $b/2$ nachziehen (Rundung auf Intervallmitte). Die Mantissen-Addition kann auch mit höherer Genauigkeit durchgeführt werden, so daß die Stellen, die beim Anpassen der Exponenten nach rechts geschiftet werden, nicht verloren gehen. Diese Stellen können dann beim Normalisieren/Runden nach der Mantissen-Addition wieder verwendet werden. Die Vermeidung bzw. Minimierung von Rundungsfehlern ist bei numerischen Berechnungen von großer Bedeutung [Kie]. Die Subtraktion annähernd gleich großer Zahlen sollte möglichst vermieden werden (durch

Umformulierung des Programms, z. B. $x_1 - x_2 + x_3 - x_4 = (x_1 + x_3) - (x_2 + x_4)$, weil dabei führende Stellen ausgelöscht und beim anschließenden Normalisieren vorher nicht vorhandene Nullen nachgezogen werden.

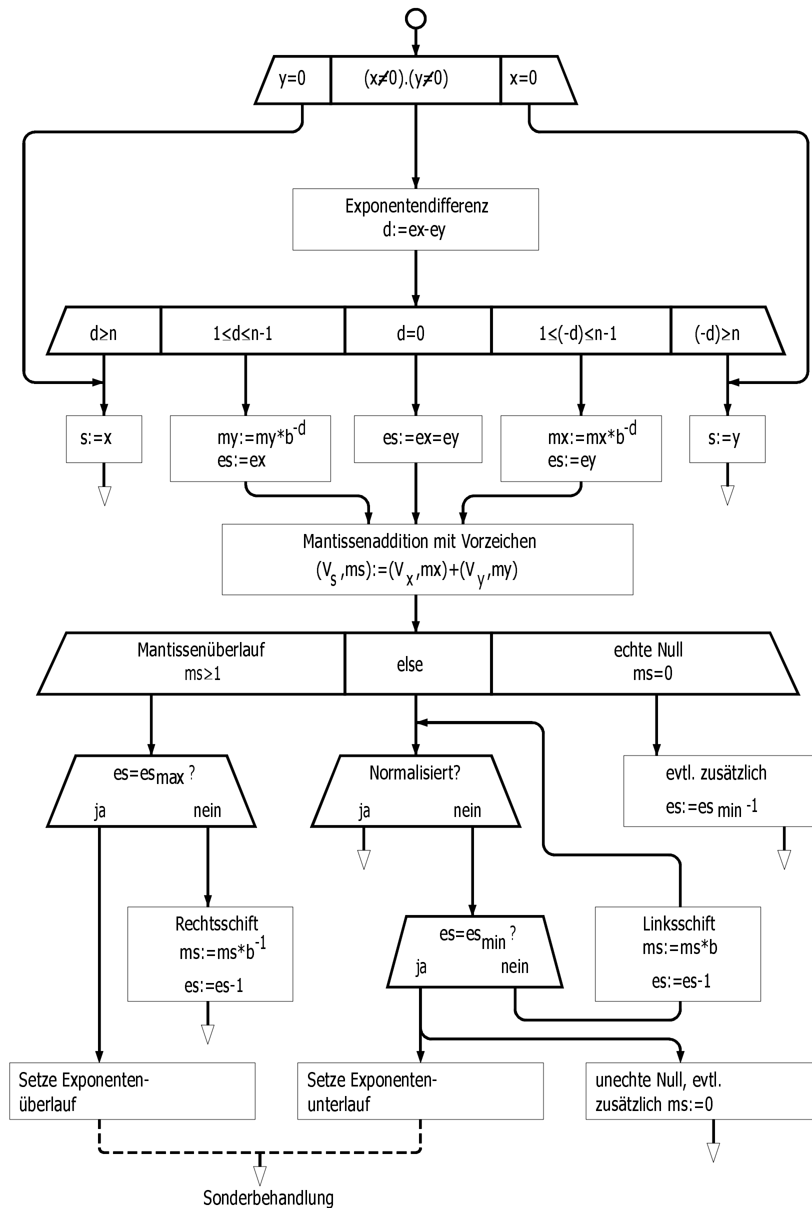


Abbildung 3.23: Addition von Gleitkommazahlen
$(V_s, ms, es) := (V_x, mx, ex) + (V_y, my, ey)$

Das Erniedrigen der Exponenten beim Normalisieren kann das Unterschreiten des zulässigen Exponentenbereichs $es \ \epsilon \ \{es_{\min} : es_{\max}\}$ zur Folge haben. Das Ergebnis ist dann so klein

geworden, daß es nicht mehr darstellbar, aber noch nicht $= 0$ ist. Das Ergebnis wird entweder als unechte Null (Kleiner Wert) weitergeführt oder gleich der echten Null gesetzt. Die echte Null entsteht nur dann, wenn $x = -y$ ist, weil dann die Summenmantisse $= 0$ wird. Die Null wird häufig durch $(ex_{\min} - 1)$ und $mx = 0$ (vergl. Abschnitt 1.6) codiert. Wird der Exponent als Charakteristik dargestellt, dann entspricht $(ex_{\min} - 1)$ der Charakteristik *EX*$= 0$. Die Bildung der Differenz d der Exponenten kann durch Addition des 2-Komplements *EY*$'$ auf *EX* und anschließende Negation der Vorzeichenbits vorgenommen werden.

Bei der Mantissen-Addition kann ein Mantissenüberlauf entstehen, d. h. $|ms| \geq 1$. Es wird also eigentlich eine Stelle mehr zur Darstellung benötigt. Durch einen Schift nach rechts mit Nachziehen der entstandenen Übertragsstelle wird die Mantisse normalisiert. Der Exponent muß dabei um 1 erhöht werden. Das gelingt aber nur, wenn der Exponent noch nicht seinen Maximalwert erreicht hat, anderenfalls muß die Exponentenüberlaufbedingung gesetzt werden.

Der Ablauf der Gleitkommaaddition ist als Flußdiagramm (Abb. 3.23) dargestellt. Dabei gilt: Die Mantissen mx, my sind positiv, normalisiert (< 1 und $\geq b - 1$) mit n Stellen hinter dem Komma. Bei der Mantissen-Addition werden die Vorzeichen V_s, V_x, V_y berücksichtigt (vergl. Abschnitte 1.2, 1.6 und 3.1.3).

3.2 Multiplikation

3.2.1 Multiplikation von Dualzahlen

Die Multiplikation der (positiven) Dualzahl $X[n]$ mit der (positiven) Dualzahl $Y[m]$ ergibt ein $(n + m)$-stelliges Produkt $P[n + m] = X[n] * Y[m]$. Meist wird die gleiche Stellenzahl $n = m$ gewählt. Aus Gründen der Übersichtlichkeit vereinbaren wir im folgenden $X = X[n]$ und $Y = Y[n]$.

Wir wollen uns nun überlegen, wie man die Multiplikation auf einfachere Operationen wie Addition, Schift und logische Operationen zurückführen kann. Das Auflösen in eine Folge einfacher Operationen kann nach folgendem Schema erfolgen:

1. Spezifikation der Operation op (X, Y)
2. Zerlegung der Operanden in Teiloperanden mit dem Ziel, die Operation in Teiloperationen zu zerlegen; z. B. werden die Operanden in eine arithmetische oder logische Summe von Teiloperanden zerlegt.
3. Anwendung der Operation auf alle Paare von Teiloperanden und Aufstellung einer Folge von Teiloperationen, z. B. in Form eines rekursiven Gleichungssystems.
4. Umwandlung in ein Programm, abstraktes Mikroprogramm, synchrones Mikroprogramm oder paralleles Mikroprogramm, je nachdem, ob eine Realisierung als sequentielles Programm, synchrones Schaltwerk oder Schaltnetz angestrebt wird (vergl. Abschn. 2.17).

Eine derartige Vorhergehensweise bezeichnet man als Top-Down-Methode, bei der das komplexe Gesamtproblem schrittweise in einfachere Teilprobleme zerlegt wird. Die angegebenen vier Schritte charakterisieren eine mögliche Top-Down-Methode zur Zerlegung arithmetischer Operationen.

Wenn der Multiplikator $Y[n]$ in eine Summe zerlegt wird, dann ergibt sich:

$$P = X * Y = X * Y_1 + 2^1 * X * Y_2 + \cdots + 2^{n-1} * X * Y_n \; . \tag{3.33}$$

Diese Beziehung läßt sich in ein HORNER-Schema umformen, wobei es zwei Möglichkeiten gibt. Man kann zuerst das höchstwertige Bit Y_n auswerten und das Teilprodukt $X * Y_n$ bilden, oder man wertet zuerst das niedrigstwertige Bit Y_1 aus.

Fall 1: Beginn mit der höchstwertigen Stelle Y_n

Ausklammern von (3.33) ergibt:

$$P = \underbrace{(\cdots(\underbrace{\underbrace{(X * Y_n)}_{P^1} 2 + X * Y_{n-1})}_{P^2} 2 + \cdots + X * Y_2)2 + X * Y_1}_{P^n} \; .$$

Daraus ergibt sich das rekursive Gleichungssystem

$$\begin{aligned} P^0 &= 0 \\ P^1 &= 2 * P^0 + X * Y_n \\ P^2 &= 2 * P^1 + X * Y_{n-1} \\ &\vdots \\ P^n &= 2 * P^{n-1} + X * Y_1 \; . \end{aligned} \tag{3.34}$$

Der Algorithmus kann jetzt in einer höheren Programmiersprache formuliert werden, z. B. in HDL:

boole $X[n], Y[n], P[2n]$; „Produkt $P = X$ mal Y“
$P := 0$;
for $i := 0$ to $n - 1$ do
$P := P * 2 + X * Y_{n-i}$ od

Wie sieht jetzt eine Hardware-Realisierung der Multiplikation aus? Dazu wird der Algorithmus in eine harwarenahe Darstellung überführt. Hier wird ein synchrones Mikroprogramm (Abb. 3.24) auf der Basis von HDL benutzt. Zur besseren Übersicht wird der Ablauf als Zustandsdiagramm dargestellt. Die Zustandsübergänge („next“ in HDL) erfolgen synchronparallel mit den Registerzuweisungen. Das Rechenwerk, das sich aus dem Mikroprogramm herleitet, ist zu

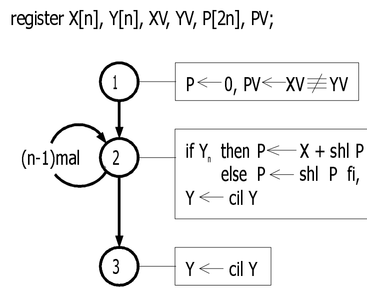


Abbildung 3.24: Serien-parallele Multiplikation, Fall 1

aufwendig, da es ein $2n$-stelliges Additionsschaltnetz erfordert. Aus diesem Grunde wird man die Multiplikation nicht mit dem höchstwertigen Bit Y_n beginnen.

Fall 2: Beginn mit der niedrigstwertigen Stelle Y_1

Mit der Abkürzung $\widehat{X} = X * 2^n$ ergibt sich aus (3.33):

$$P = \underbrace{(\cdots(\underbrace{\underbrace{(\widehat{X} * Y_1) * 2^{-1}}_{P^1} + \widehat{X} * Y_2) * 2^{-1}}_{P^2} + \cdots + \widehat{X} * Y_n) * 2^{-1}}_{P^n} \; .$$

Daraus folgt das Rekursionsschema:

$$\begin{aligned} P^0 &= 0 \\ P^1 &= (P^0 + \widehat{X} * Y_1) * 2^{-1} \\ P^2 &= (P^1 + \widehat{X} * Y_2) * 2^{-1} \\ &\vdots \\ P^n &= (P^{n-1} + \widehat{X} * Y_n) * 2^{-1} \; . \end{aligned} \tag{3.35}$$

Ein etwas verkürztes Rekursionsschema ergibt sich, wenn $\widehat{X} = Y * 2^{n-1}$ gesetzt wird, so daß die letzte Multiplikation mit 2^{-1} entfallen kann. Der zu (3.35) gehörige Algorithmus lautet mit den Sprachmitteln von HDL:

boole $X[n], Y[n], P[2n]$; „Produkt $P = X$ mal Y“
$P := 0$;
for $i := 1$ to n do
$P := (P + X_\texttt{oo...o} * Y_i)/2$ od .

`X_oo...o` bedeutet eine Verschiebung von `X` um n Stellen nach links (das entspricht $X * 2^n$), und die Division durch 2 = `1o` kann durch einen Schift nach rechts um eine Stelle bewerkstelligt werden. Die Multiplikation mit Y_i kann durch die Und-Funktion ersetzt werden.

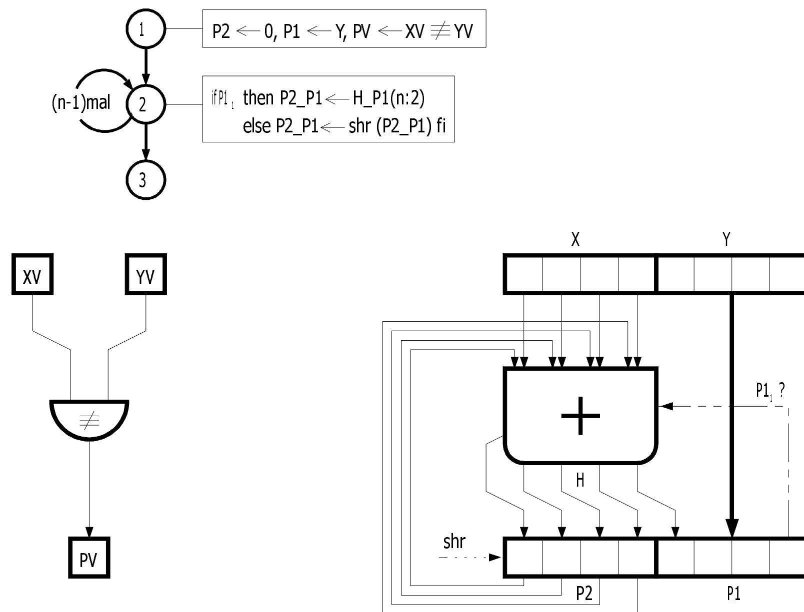


Abbildung 3.25: Serien-parallele Multiplikation mit Rechenwerk

Aus diesem Algorithmus läßt sich wieder ein synchrones Mikroprogramm und ein zugehöriges Rechenwerk entwickeln (Abb. 3.25). Das Multiplikationsrechenwerk wurde in vielen Rechenanlagen nach diesem Prinzip konstruiert. Dabei entspricht $P2$ dem Akkumulator und $P1$ dem erweiterten Akkumulator. Das Register für Y kann eingespart werden, wenn Y am Anfang in $P1$ steht.

Zahlenbeispiel: $(+15) * (-9) = -135$
$V_x = \text{o} \quad |X| = \text{llll}$
$V_y = \text{l} \quad |Y| = \text{lool}$

```
Zustand
   1                P2_P1  ←  oooo lool  Vp ← 1
   2    H = o  llll P2_P1  ←  olll lloo
   2                P2_P1  ←  ooll lllo
   2                P2_P1  ←  oool llll
   2    H = l  oooo P2_P1  ←  looo olll  „gleich − 135“
   3
```

Häufig wird die Mikrooperation $P2_P1 \leftarrow H_P1_{n:2}$ in zwei aufeinanderfolgende Operationen zerlegt, und zwar in

$$\textit{CARRY_}P2 \leftarrow P2 \mathbin{+\!\!+} X; \quad P2_P1 \leftarrow \textit{CARRY}\ \text{inshr}\ (P2_P1)\ .$$

Die erste Mikrooperation entspricht einer Addition, die auch anderweitig benötigt wird, und die zweite einem Rechtsschift über beide Akkumulatoren.

3.2.2 Multiplikation von Zweikomplementzahlen

3.2.2.1 Grundsätzliche Methode

Wenn zwei n-stellige Zweikomplementzahlen $X = X[n]$ und $Y = Y[n]$ dual miteinander multipliziert werden, so ergibt sich ein $2n$-stelliges Zwischenergebnis, das noch korrigiert werden muß, um das Produkt $P[2n]$ in Zweikomplementdarstellung zu erhalten. Die notwendigen Korrekturen lassen sich durch das Einsetzen der Rückabbildungsgleichungen (1.8) für x, y und p ermitteln.

$$\begin{aligned} p &= x * y = (X - X_n * 2^n) * (Y - Y_n * 2^n) \quad \text{und} \\ p &= P[2n] - P_{2n} * 2^{2n}\ . \end{aligned}$$

Durch Gleichsetzung ergibt sich daraus

$$P[2n] = (X - X_n * 2^n) * (Y - Y_n * 2^n) + P_{2n} * 2^{2n}\ . \tag{3.36}$$

Die Vorzeichenstelle muß folgender Beziehung gehorchen:

$$P_{2n} = X_n \not\equiv Y_n = \overline{X_n}.Y_n \vee X_n.\overline{Y_n} = \overline{X_n}.Y_n + X_n.\overline{Y_n}\ . \tag{3.37}$$

Einsetzen von (3.37) in (3.36) ergibt

$$\begin{aligned} P[2n] &= X * Y - X_n * 2^n Y - Y_n * 2^n X \\ &+ \underbrace{X_n Y_n 2^{2n} + \overline{X}_n Y_n 2^{2n}}_{=Y_n 2^{2n}} + \underbrace{X_n \overline{Y_n} 2^{2n} + \overbrace{X_n Y_n 2^{2n}}_{}}_{=X_n 2^{2n}} \overbrace{}^{=0} - X_n Y_n 2^{2n}\ . \end{aligned}$$

Für den Fall $X_n = Y_n = \mathtt{1}$ muß 2^{2n} abgezogen werden, indem das Übertragsbit in die Stelle 2^{n+1} mit der Wertigkeit 2^{2n} nicht beachtet wird $\pmod{2^{2n}}$. Damit erhält man allgemein:

$$\begin{aligned} P[2n] = (X * Y \quad &+ \quad X_n * 2^n (2^n - Y) \\ &+ \quad Y_n * 2^n (2^n - X)) \bmod 2^{2n}\ . \end{aligned} \tag{3.38}$$

Diese Beziehung läßt sich in 4 Fälle aufspalten:

1. $X_n = \circ\,, Y_n = \circ$: $P[2n] = X * Y$
   x und y sind positiv: es ist keine Korrektur notwendig.

2. $X_n = \texttt{o}\, , Y_n = \texttt{l}$: $P[2n] = X * Y + 2^n(2^n - X)$
   x ist positiv, y negativ: das Komplement von X, verschoben um n Stellen nach links, muß addiert werden.

3. $X_n = \texttt{l}\, , Y_n = \texttt{o}$: $P[2n] = X * Y + 2^n(2^n - Y)$
   x ist negativ, y positiv: das Komplement von Y, verschoben um n Stellen nach links, muß addiert werden.

4. $X_n = \texttt{l}\, , Y_n = \texttt{l}$:
   $P[2n] = X * Y + 2^n(2^n - Y) + 2^n(2^n - X) - [2^{2n}]$
   x und y sind negativ: die Komplemente von X und Y, verschoben um n Stellen nach links, müssen addiert werden. Der entstehende Übertrag wird nicht beachtet.

Zahlenbeispiel für den 4. Fall:

```
X[5] = l o o l l           x = -13
Y[5] = l o l l o           y = -10

l o o l l * l o l l o
            o o o o o
          l o o l l
        l o o l l
      o o o o o
    l o o l l
    l l o l o o o l o
  o l l o l                +2^5(2^5 - X[5])
  o l o l o                            +2^5(2^5 - Y[5])
l o o l o o o o o l o
```

Eine prinzipielle Lösung für das Rechenwerk zeigt Abb. 3.26. Außer dem Multiplikationsschaltnetz ist ein Additionsschaltnetz für die Korrektur notwendig. Bei einer Realisierung als Schaltkette wird man versuchen, die notwendigen Korrekturen schon während der Multiplikation schrittweise zu berücksichtigen, d. h. die Kettenglieder führen dann schon Teilkorrekturen durch.
Bei der bisher betrachteten Methode wird auch der Sonderfall `loo...o` mal `loo...o` (-2^{n-1} mal -2^{n-1}) noch richtig erfaßt. Schließen wir diesen Sonderfall aus, dann genügen $2n-1$ Stellen zur Darstellung des Produkts. Wie bei der 2-Komplement-Addition ist es also durchaus sinnvoll, die Zahl `loo...o` zu verbieten. Bei den weiteren Multiplikationsmethoden wird versucht, mit möglichst wenig Schritten auszukommen und die Korrekturen schon während der eigentlichen Multiplikationen durchzuführen. Eine gute Darstellung der meisten der hier geschilderten Verfahren ist auch in [Chu-2] zu finden.

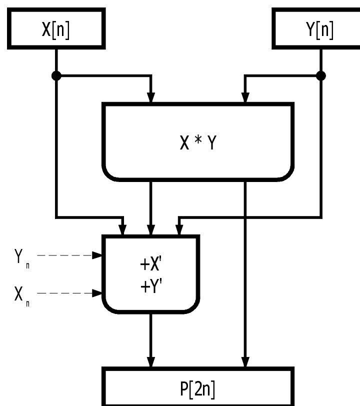


Abbildung 3.26: Rechenwerk für die Multiplikation im Zweikomplement

3.2.2.2 Burks-Goldstine-von-Neumann-Methode

Bei dieser Methode werden die Zweikomplementzahlen ohne Vorzeichen multipliziert. Die betragsmäßig größten Operanden ($-2^{n-1} = \texttt{loo...o}$) sind verboten. Die notwendigen Korrekturen lassen sich wieder unter Verwendung der Rückabbildungsgleichung $X = X[n] - X_n * 2^n$ und der Beziehung $X[n] = X[n-1] + X_n * 2^{n-1}$ ermitteln:

$$\begin{aligned} P[2n-1] &= X[n-1] * Y[n-1] + X_n 2^{n-1}(2^n - Y[n-1]) \\ &\quad + Y_n 2^{n-1}(2^n - X[n]) - X_n Y_n 2^{2n-1}\,. \end{aligned} \tag{3.39}$$

Der Term $-X_n Y_n 2^{2n-1}$ bedeutet, daß der entstehende Übertrag in die Stelle P_{2n} nicht beachtet werden darf, d. h. das Ergebnis wird modulo 2^{2n-1} betrachtet.

Zahlenbeispiel:

```
X[5] = l o o l l            x = -13
Y[5] = l o l l o            y = -10

o o l l * o l l o
          o o o o
        o o l l
      o o l l
    o o o o
    o o l o o l o
  o l l o l              +2^4(2^5 - X[5])
  l l o l o                          +2^4(2^5 - Y[4])
l o l o o o o o l o
```

Nachteilig bei dieser Methode ist, daß im Anschluß an die Multiplikation zwei Korrekturen durchgeführt werden müssen. Man kann nun die Korrektur mit dem Komplement von Y schon

während der Multiplikation erledigen, so daß nur noch die Korrektur mit dem Komplement von X übrig bleibt. Dadurch braucht Y auch nicht mehr bis zum Schluß aufgehoben zu werden.

Die Korrektur von $2^{n-1}(2^n - Y[n-1])$ in (3.39) kann wie folgt umgeformt werden:

$$\begin{aligned} 2^{n-1}(2^n - Y[n-1]) &= 2^{n-1}(2^{n-1} - Y[n-1] + 2^{n-1}) \\ &= 2^{n-1}(Y'[n-1] + 2^{n-1}) \\ &= 2^{n-1}(\overline{Y[n-1]} + 2^{n-1} + 1) \end{aligned} \quad (3.40)$$

Die Methode von BURKS-GOLDSTINE-VON-NEUMANN [Bur] besteht nun darin, während der Multiplikation eine $\mathtt{l}$ nachzuziehen, wenn $X_n = \mathtt{l}$ und $Y_i = \mathtt{o}$ ist. Damit wird die Korrektur um $2^{n-1} * \overline{Y[n-1]}$ schrittweise durchgeführt. Am Schluß muß dann noch die Konstante $2^{n-1}(2^{n-1}+1)$ dazu addiert werden. Die Addition der Konstanten kann für den Fall $X_n = \mathtt{l}$ und $Y_n = \mathtt{l}$ mit der Korrektur $2^{n-1}(2^n - X[n])$ zusammengefaßt werden, so daß insgesamt höchstens ein Korrekturschritt erforderlich ist.

Zahlenbeispiel zur Methode von BURKS-GOLDSTINE-VON-NEUMANN:

```
X[5] = l o o l l            x = -13
Y[5] = l o l l o            y = -10

o o l l * o l l o
        l o o o o      .
      + o o l l        .
      o l o l l        .
    + o o l l          .    Nachziehen von Y_i (überstrichen)
    o l o o o          .
  + o o o o            .
  l o l o o            .
+ l o o o l            .    Korrekturkonstante 2^(n-1)(2^(n-1) + 1)
+ o l l o l                 2^(n-1)(2^n - X[n])
l o l o o o o o l o
```

Eine ähnliche Methode besteht darin, den vollständigen Multiplikanden $X[n]$ mit dem Multiplikator $Y[n-1]$ ohne Vorzeichen zu multiplizieren. Unter Verwendung der Rückabbildungsgleichung $y = Y[n-1] - Y_n * 2^{n-1}$ ergibt sich:

$$\begin{aligned} P[2n-1] &= X[n] * Y[n-1] + X_n 2^n (2^{n-1} - Y[n-1]) \\ &\quad + Y_n 2^{n-1}(2^n - X[n]) - X_n Y_n 2^{2n-1} \, . \end{aligned} \quad (3.41)$$

Der Term $-X_n Y_n 2^{2n-1}$ bedeutet, daß ein auftretender Übertrag in die Stelle P_{2n} ignoriert werden muß.

Zahlenbeispiel:

```
      X[5] = l o o l l          x = -13
      Y[5] = l o l l o          y = -10

      l o o l l * o l l o
                  o o o o o
                l o o l l
              l o o l l
            o o o o o
          o l l o l             +2^4(2^5 - X[5])
        l o l o                              +2^5(2^4 - Y[4])
      l o l o o o o o o l o
```

Auch bei dieser Methode kann durch Nachziehen einer l für $X_n = \mathtt{l}$ und $Y_i = \mathtt{o}$ die Korrektur mit dem Komplement von Y schrittweise durchgeführt werden. Am Schluß muß noch eine l, verschoben um n Stellen, addiert werden.

3.2.2.3 Erste Methode von Robertson

Die nachträgliche Korrektur mit dem Komplement von Y kann auch dadurch vermieden werden, daß X nach links mit seinem Vorzeichen erweitert wird. Dadurch ist sichergestellt, daß sich beim Aufaddieren der Partialprodukte $X * Y_i$ für $X_n = \mathtt{l}$ (x negativ) und $Y_n = \mathtt{o}$ (y positiv) immer eine negative Zwischensumme ergibt. Das Produkt ergibt sich mit $y = Y[n-1] - Y_n * 2^{n-1}$ zu

$$p = x * Y[n-1] + Y_n * 2^{n-1}(-x) \; . \tag{3.42}$$

Bei der schrittweisen Auswertung dieser Gleichung werden x und p im Zweikomplement mit einer ausreichenden Anzahl von Stellen angesetzt.

Zahlenbeispiel:

```
      X[5] = l o o l l          x = -13
      Y[5] = l o l l o          y = -10


      l o o l l * o l l o
                o o o o o o .
              + l o o l l   .   Wenn Zwischenergebnis
              l l o o l l   .   negativ, dann immer
            + l o o l l     .   l  nachziehen
          l o l l o o       .
        + o o o o o         .
      l l o l l o               = x * Y[n-1]
    + o l l o l                              +2^4(2^5 - X[5])
    l o l o o o o o o l o
```

Die Methode von ROBERTSON hat den Vorteil, daß die Korrektur mit dem Komplement von Y durch das Nachziehen von Einsen nach dem Aufaddieren der Teilprodukte vorgenommen werden kann. Das Zwischenergebnis wird das erste Mal negativ, wenn $X_n = \mathtt{l}$ und $Y_i = \mathtt{l}$ ist. Damit alle weiteren Zwischenergebnisse negativ bleiben, muß danach immer eine $\mathtt{l}$ nachgezogen werden.

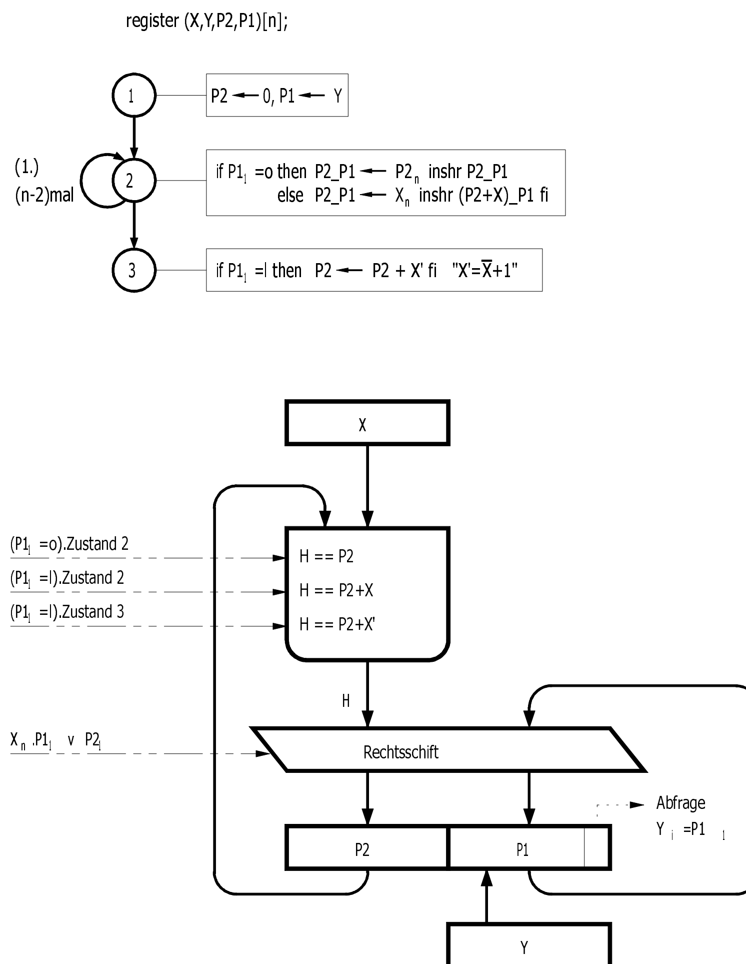


Abbildung 3.27: Zustandsdiagramm für die serien-parallele Multiplikation im Zweikomplement

Abb. 3.27 zeigt den Ablauf und das Rechenwerk der serien-parallelen Addition nach der 1. Methode von ROBERTSON. Im Zustand 2 wird für $X_i = \mathtt{l}$ $(P1_1 = \mathtt{l})$ der Multiplikand x zum Akkumulator $P2$ addiert, und das Vorzeichen X_n wird nachgezogen. Wenn $Y_i = \circ$ ist, dann wird das vorhandene Vorzeichen $P2_n$ nachgezogen (arithmetischer Schift). Im Zustand 3 wird für $P1_1 = \mathtt{l}$ das Komplement von X addiert. Das $(2n-1)$-stellige Produkt im Zweikomplement steht dann linksbündig in $P2$ und $P1$, mit dem Vorzeichen $P2_n$. Das Bit

$P1_1$ enthält das alte Vorzeichen von Y und ist für das Produkt ohne Bedeutung. Durch einen arithmetischen Schift nach rechts könnte man $P1_1$ herausschieben, und das Produkt stünde dann rechtsbündig mit zwei Vorzeichen in $P2$ und $P1$.

Zahlenbeispiel zur ersten Methode von ROBERTSON:

```
    X = l o o l l       x = -13       Y = l o l l o       y = -10
    X'= o l l o l
Zustand
  1     P2_P1 <- o o o o o   l o l l o
  2     P2_P1 <- o o o o o   o l o l l      arithmetischer Schift
                     l o o l l                  +X
  2     P2_P1 <- l l o o l   l o l o l      Schift, X(n) nachziehen
                (l)o l l o o                    +X
  2     P2_P1 <- l o l l o   o l o l o      Schift, X(n) nachziehen
  2     P2_P1 <- l l o l l   o o l o l      arithmetischer Schift
                     o l o o o                  +X', Korrektur
  3     P2_P1 <- o l o o o   o o l o l
  4
```

3.2.2.4 Zweite Methode von Robertson

Die erste Methode von ROBERTSON erfordert eine nachträgliche Korrektur mit dem Komplement von X, wenn Y negativ ist. Prinzipiell besteht die Möglichkeit, auch das Komplement von X schon seriell während der schrittweisen Addition zu berücksichtigen. Allerdings muß dann zusätzlich das Komplement von X zur Verfügung stehen, und eine weitere Additionsstelle muß bereitgestellt werden. Um die nachträgliche Korrektur zu vermeiden, hat ROBERTSON [Rob] vorgeschlagen, x und y vor der Multiplikation zu negieren, wenn y negativ ist. Dann ist y immer positiv, und die Multiplikation wird auf die beiden Fälle $X_n = \mathtt{o}$, $Y_n = \mathtt{o}$ und $X_n = \mathtt{l}$, $Y_n = \mathtt{o}$ reduziert. Bei dieser Methode ist also am Anfang ein zusätzlicher Schift erforderlich, wenn $Y_n = \mathtt{l}$ ist. Das Rechenwerk wird dadurch komplizierter und der Multiplikator $Y =$ `loo...o` muß wegen der nicht möglichen Komplementbildung verboten werden.

3.2.2.5 Algorithmus von Booth

Der Algorithmus von BOOTH (A. D. BOOTH und K. H. V. BOOTH [Boo]) erfordert keine nachträglichen Korrekturen. Er basiert auf der Zerlegung einer Dualzahl $Y[n]$ in eine $n + 1$-stellige ternäre Zahl, deren Ziffern die Werte `+l, o, -l` annehmen können. Durch eine einfache Umformung ergibt sich

$$Y[n] \;=\; \sum_{i=1}^{n} Y_i * 2^{i-1}$$

$$= \sum_{i=1}^{n+1} (Y_{i-1} - Y_i) * 2^{i-1} \quad \text{mit } Y_{n+1} = Y_0 = \circ \; . \tag{3.43}$$

Beispiel:

$$+10 = \texttt{lolo}_2 = (\texttt{+l,-l,+l,-l,o}) \;=\; \begin{array}{l}\texttt{llllo}\\ \texttt{+-+-}\end{array}$$

Für eine Zweikomplementzahl $Y[n]$ gilt $y = Y[n] - Y_n * 2^n$. Ersetzt man $Y[n]$ durch (3.43), dann folgt

$$y = \sum_{i=1}^{n} (Y_{i-1} - Y_i) * 2^{i-1} \quad \text{mit } Y_0 = \circ \; . \tag{3.44}$$

Diese Beziehung ist eine andere Form der Rückabbildungsgleichung vom Zweikomplement auf den Wert y. Die Ziffern der ternären Zahl ergeben sich dabei direkt aus der Differenz zweier aufeinanderfolgender Ziffern der Zweikomplementzahl.

Die Produktbildung können wir, wie bei den Vorzeichenzahlen (Abschnitt 3.2.1, Fall 2), in ein Gleichungssystem umformen:

$$p = x * y = x * \sum_{i=1}^{n} (Y_{i-1} - Y_i) * 2^{i-1} \tag{3.45}$$

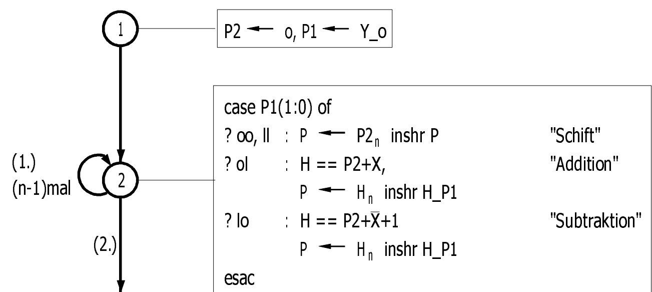
Mit $\widehat{x} = x * 2^n$ folgt daraus

$$\begin{aligned}
p^0 &= 0 \\
p^1 &= (p^0 - (Y_1 - 0) * \widehat{x}) * 2^{-1} \\
p^2 &= (p^1 - (Y_2 - y_1) * \widehat{x}) * 2^{-1} \\
&\;\vdots \\
p^n &= (p^{n-1} - (Y_n - Y_{n-1}) * \widehat{x}) * 2^{-1}
\end{aligned} \tag{3.46}$$

Die letzte Multiplikation mit 2^{-1} kann entfallen, wenn $\widehat{x} = x * 2^{n-1}$ gesetzt wird. Diese Gleichungen können wie folgt interpretiert werden: In Abhängigkeit von zwei aufeinanderfolgenden Ziffern der Zweikomplementzahl $Y[n]$ wird x addiert, subtrahiert, oder es wird keine Operation durchgeführt (abgesehen von dem anschließenden Schift):

$$\begin{aligned}
Y_i Y_{i-1} = \texttt{o l}&: \quad \text{Addition} \\
\texttt{l o}&: \quad \text{Subtraktion} \\
\texttt{o o}&: \quad \text{keine Operation} \\
\texttt{l l}&: \quad \text{keine Operation} \; .
\end{aligned}$$

```
register X[n], Y[n], P(2*n:0);
signal H[n];   "Ausgang des Additionsschaltnetzes"
equal P2[n] = P(2*n:n+1), P1[n] = P(n:1);
```



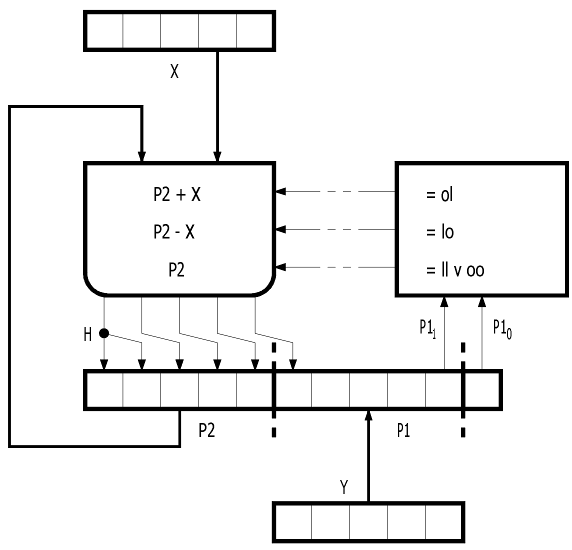


Abbildung 3.28: Mikroalgorithmus und Rechenwerk für die Multiplikation nach BOOTH

Abb. 3.28 zeigt das synchrone Zustandsdiagramm und das Rechenwerk für die Multiplikation von 2-Komplementzahlen nach dem Algorithmus von BOOTH. Subtraktion und Addition wird zweckmäßigerweise im 2-Komplement durchgeführt, da die Operanden im 2-Komplement zur Verfügung stehen. Dann sind die Teilergebnisse Pi auch 2-Komplementzahlen. Da zwei aufeinanderfolgende Additionen oder Subtraktionen nicht möglich sind, kann kein Überlauf auftreten. An jede Operation schließt sich ein arithmetischer Rechtsschift an. Der letzte Rechtsschift, der eigentlich überflüssig ist, bewirkt, daß das Produkt rechtsbündig in den Registern $P2$ und $P1$ steht.

Zahlenbeispiel zur Multiplikation nach BOOTH:

```
        x = -13                   y = -10
        X = l o o l l              Y = l o l l o
        X = o l l o l              Y = l l o l o
                                     -  +   -
    Zustand
      1     P <- o o o o o,l o l l o,o
      2     P <- o o o o o,o l o l l,o     Schift
                 o l l o l                 +X'
      2     P <- o o l l o,l o l o l,l     Schift
      2     P <- o o o l l,o l o l o,l     Schift
                 l o l l o                 +X
      2     P <- l l o l l,o o l o l,o     Schift
                 o l o o o                 +X'
      2     P <- o o l o o,o o o l o,l     Schift
```

3.2.3 Parallele Multiplikation

Die bisher behandelten Multiplikationsmethoden liefen serien-parallel ab. Die Teilprodukte wurden parallel berechnet, aber seriell aufaddiert. Diese Algorithmen können natürlich noch weiter zerlegt werden, indem auch die Addition seriell ausgeführt wird. Das dauert aber im allgemeinen zu lange. Um auf besonders kurze Ausführungszeiten zu kommen, kann die Multiplikation parallel durch ein Schaltnetz ausgeführt werden. Die einfachste Lösung wäre es, alle möglichen Produkte in Form einer Tabelle in einem Festwertspeicher abzuspeichern. Allerdings würde man für einen 16-Bit-Multiplizierer bereits einen Speicher mit 2^{32} Speicherworten benötigen. Eine andere Möglichkeit besteht darin, die Multiplikations-Tabelle logisch zu minimieren und in Form einer Schaltmatrix, die als programmierbarer Baustein (PLA) hergestellt werden kann, zu realisieren. Der Aufwand läßt sich dadurch beträchtlich reduzieren, aber es werden speziell programmierte Schaltmatrizen benötigt. Der Aufwand ist dann aber immer noch ziemlich hoch, weil die Realisierung durch ein zweistufiges Schaltnetz (als Schaltmatrix) nicht die strukturellen Eigenschaften dieser Funktion berücksichtigt. Denn die Multiplikation kann in eine Hintereinanderschaltung von Teilfunktionen zerlegt werden, und das ist gerade die allgemein bekannte Realisierung durch eine zweidimensionale Schaltkette. Obwohl diese Lösung nicht die kleinste Rechenzeit der Tabellenlösung aufweist, ist sie ein guter Kompromiß zwischen Aufwand und Rechenzeit.

Die bisher betrachteten Multiplikationsalgorithmen können direkt für eine Schaltkettenrealisierung herangezogen werden, denn auch in einer Schaltkette wird das Ergebnis schrittweise berechnet, wenn auch asynchron. Eine naheliegende, einfache Schaltkette zur Multiplikation positiver Dualzahlen auf der Basis des Carry-Ripple-Addierers zeigt Abb. 3.29 für $n = m = 4$. Für die Multiplikation eines n-stelligen Multiplikanden $X[n]$ mit einem m-stelligen Multipli-

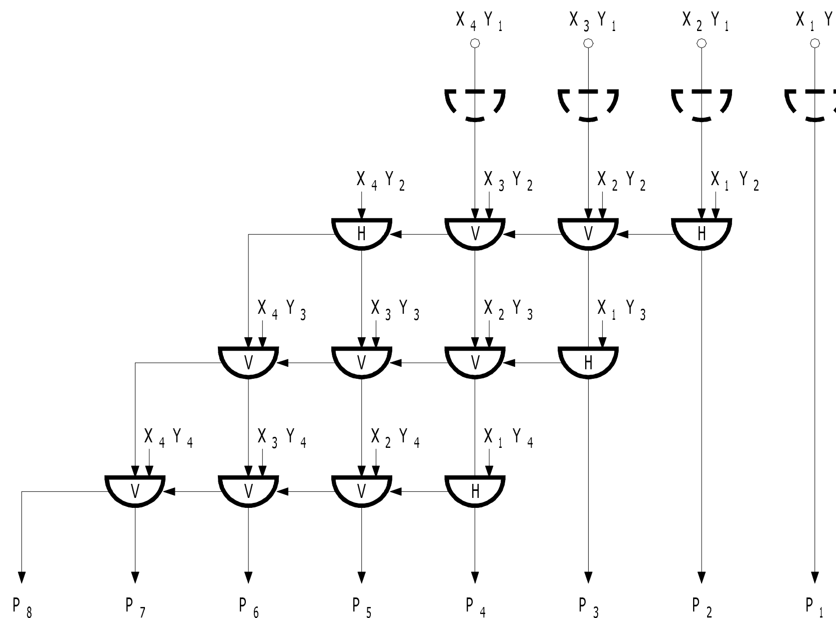


Abbildung 3.29: Multiplikationsschaltnetz für Dualzahlen

kator $Y[m]$ werden $(m-1)*n-m$ Volladdierer und n Halbaddierer benötigt. Die maximale Rechenzeit beträgt $t=[n+(m-2)*2]*T$, mit $T=$ Laufzeit durch einen Voll- oder Halbaddierer. Durch Anwendung der *Carry-Save-Technik* (siehe Abschnitt 3.1.1.6), d.h. durch eine Verarbeitung der Überträge in der nächsten Stufe, läßt sich die Multiplikationsschaltkette nach Abb. 3.30 entwickeln. Die Rechenzeit hat sich dabei auf $t=(m-2)*T$ plus die Zeit für die Abschlußaddition verringert.

Auf die im folgenden behandelten Schaltketten läßt sich die Carry-Save-Technik ebenso anwenden. Dabei wird der Übertrag nicht „horizontal“ vom Volladdierer (i,j) in den Volladdierer $(i,j+1)$, sondern „schräg nach unten“ in den Volladdierer $(i+1,j+1)$ weitergeleitet. Dabei ist i der Stufenindex (zählt von oben nach unten) und j der Spaltenindex (zählt von rechts nach links).

Wenn man eine größere Zahl von Multiplikationen hintereinander ausführen will (z. B. bei der Verarbeitung von Vektoren, Vektorrechenwerke in Hochleistungsrechnern), dann wird man die Multiplikationsschaltketten als Pipeline unter Verwendung der Carry-Save-Technik aufbauen. Dazu müssen zwischen den s Stufen Register eingefügt werden. Um das erste Ergebnis zu berechnen, werden s Takte benötigt. Mit jedem Takt kann dann ein weiteres Ergebnis berechnet werden, weil sich in den Registern der vorausgehenden Stufen schon die nachfolgenden Teilergebnissse befinden. Bei langen Vektoren kann man durch diese Pipeline-Technik die Rechenzeit beinahe auf $1/s$ reduzieren.

Multiplikationsschaltketten für die Multiplikation von Zweikomplementzahlen zeigen die Abb.

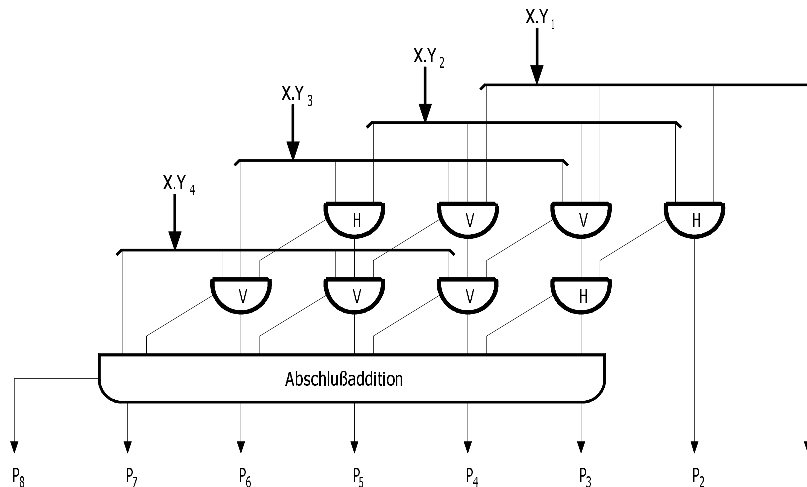


Abbildung 3.30: Multiplikationsschaltnetz mit Carry-Save-Addition

3.31 und 3.32. Die Schaltkette (Abb. 3.31) arbeitet nach der 1. Methode von ROBERTSON. Die Oder-Gatter dienen dazu, die Korrektur mit dem Komplement von Y zu erzeugen. Dazu wird die Funktion $X_n.Y_i \vee P_{2n}$ nachgezogen (vergl. Abb. 3.27). Das Komplement von X wird anschließend addiert, falls $Y_n = \mathtt{l}$ ist. Die Schaltkette besitzt nur $2n-1$ Ausgänge, da nur ein Vorzeichen benötigt wird. Wenn die Schaltkette auch den Sonderfall `loo...o` mal `loo...o` berechnen soll, dann muß das Schaltnetz für Y um eine Stelle erweitert werden. Für das Produkt $X[n]$ mal $Y[m]$ beträgt die maximale Rechenzeit $t = [n + (m-2) * 2] * T$ Laufzeiten. Die Rechenzeit läßt sich reduzieren, wenn die einzelnen Addierschaltnetze mit Übertragsvorausberechnung (Carry Look Ahead) versehen werden oder die Carry-Save-Technik eingesetzt wird.
Die Schaltkette von BAUGH und WOOLEY (Abb. 3.32) arbeitet im wesentlichen nach der Methode von BURKS-GOLDSTINE-VON-NEUMANN [BaW]. Mit dem Ansatz

$$\begin{aligned} p &= P[2n] - P_{2n} * 2^{2n} , \\ x &= X[n-1] - X_n * 2^{n-1} , \\ y &= Y[n-1] - Y_n * 2^{n-1} \end{aligned}$$

ergibt sich

$$\begin{aligned} P[2n] &= X[n-1] * Y[n-1] + X_n 2^{n-1}(\overline{Y[n-1]} + 1) \\ &\quad + Y_n 2^{n-1}(\overline{X[n-1]} + 1) + \text{Korrektur} \end{aligned} \tag{3.47}$$

$$\text{Korrektur} = 2^{2n-2}(X_n \vee Y_n) + 2^{2n-1}(X_n \vee Y_n) - X_n Y_n 2^{2n} . \tag{3.48}$$

Der letzte Term entspricht dabei einem Übertrag in die Stelle 2^{2n}, der nicht beachtet wird. Für die Korrektur lassen sich auch andere Beziehungen aufstellen, vergleiche [Bln]. Die maximale Rechenzeit beträgt $t = (3n-2) * T$ für das $2n$-stellige Produkt. Im Gegensatz

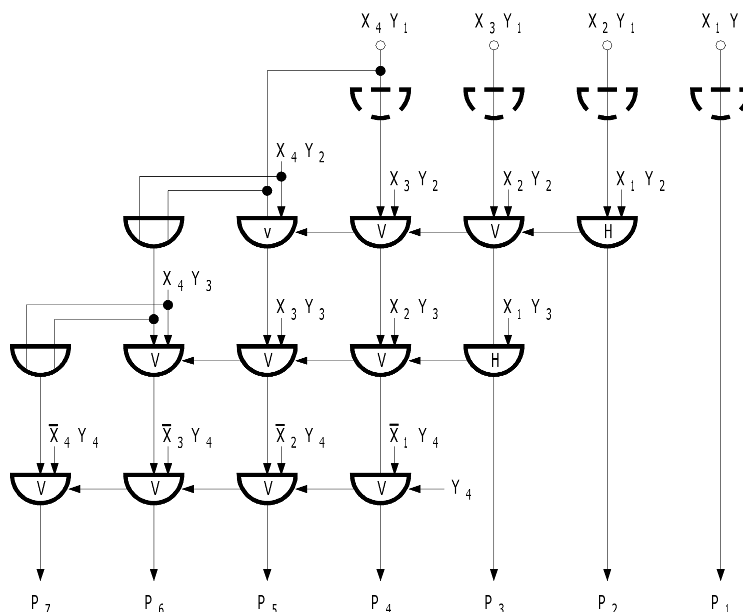


Abbildung 3.31: Multiplikationschaltkette für Zweikomplementzahlen

zu der Schaltkette nach der 1. Methode von ROBERTSON wird auch hier noch das Produkt `lo...o` mal `lo...o` richtig gebildet. Wird dieser Sonderfall nicht benötigt, dann kann das Exklusiv-Oder-Gatter für P_8 entfallen.

Man kann auch den Algorithmus von BOOTH in eine Schaltkette umsetzen (Abb. 3.33). In Abhängigkeit von den beiden Steuersignalen Y_i und Y_{i-1} muß jede Zelle addieren, subtrahieren oder nur durchschalten können. Die logischen Gleichungen für eine Zelle lauten

$$\mathit{ADD} = \overline{Y_i}.Y_{i-1} \qquad \mathit{SUB} = Y_i.\overline{Y_{i-1}}$$

$$S = A := \begin{cases} A \not\equiv B \not\equiv C & \text{für } \mathit{ADD}, \mathit{SUB} \\ A & \text{sonst} \end{cases}$$

$$C := \begin{cases} A.B \vee A.C \vee B.C & \text{für } \mathit{ADD} \\ \overline{A}.B \vee \overline{A}.C \vee B.C & \text{für } \mathit{SUB} \\ \text{Don't care} & \text{sonst} \end{cases}$$

Für den Sonderfall $X[n] = \texttt{loo...o}$ ergibt sich ein falsches Ergebnis. Dieser Multiplikand sollte möglichst verboten werden, denn die Komplementbildung ist dafür nicht möglich. Falls dieser Sonderfall dennoch erfaßt werden soll, dann muß die Schaltkette nach links um eine Stelle (Schutzstelle) erweitert werden.

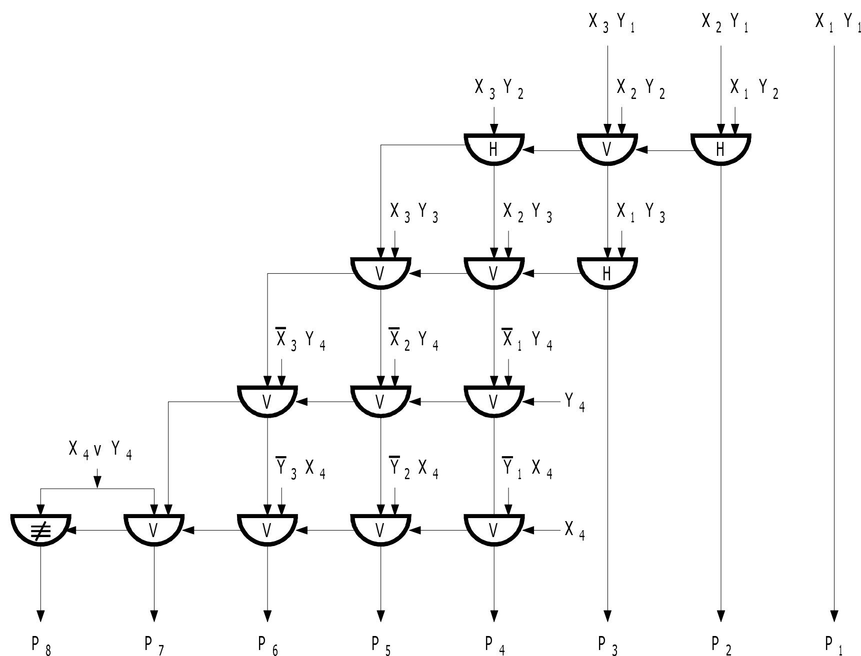


Abbildung 3.32: Multiplikationsschaltkette nach BAUGH/WOOLEY

Dieser Algorithmus wurde in einem Multiplizierbaustein der Firma „Advanced Micro Devices“ realisiert. Jeder Baustein kann eine 4×2 Bit-Multiplikation im 2-Komplement durchführen, so daß er zwei Zeilen der Schaltkette (Abb. 3.33) ersetzen kann. Der interne Aufbau des Bausteins besteht aus einem Schifter, Komplementierer und Addierer. In Abhängigkeit von drei aufeinanderfolgenden Bits kann dieser Baustein fünf Operationen durchführen: Addieren, Subtrahieren, Doppelt Addieren oder Doppelt Subtrahieren (durch internes Schiften) und Durchschalten.

Multiplikationsschaltketten, die für die Gleitkommamultiplikation benutzt werden, können vereinfacht werden, weil die Mantissen eine feste Länge haben. Das doppelt so lange Produkt muß auf die Hälfte der Stellenzahl gerundet werden, so daß ein Teil der Multiplikationsschaltkette auf der rechten Seite entfallen kann. Für einen 8×8 Bit-Multiplizierer brauchen dann nur 11 Stellen berechnet zu werden, die auf 8 gerundet werden, und für einen 16×16 Bit-Multiplizierer brauchen nur 20 Stellen berechnet zu werden. Der Fehler ist dann kleiner als die Hälfte des Wertes der niedrigstwertigen Stelle des gültigen Produkts. (Der maximale Fehler läßt sich ermitteln, indem die nicht verarbeiteten Stellen mit Einsen besetzt und aufaddiert werden.)

3.2.4 Doppelwort-Multiplikation

Wie bei der Doppelwort-Addition soll die Rechengenauigkeit auf das Doppelte erhöht werden. Die beiden $2n$-stelligen positiven Dualzahlen $A = A2_A1$ und $B = B2_B1$ sollen unter

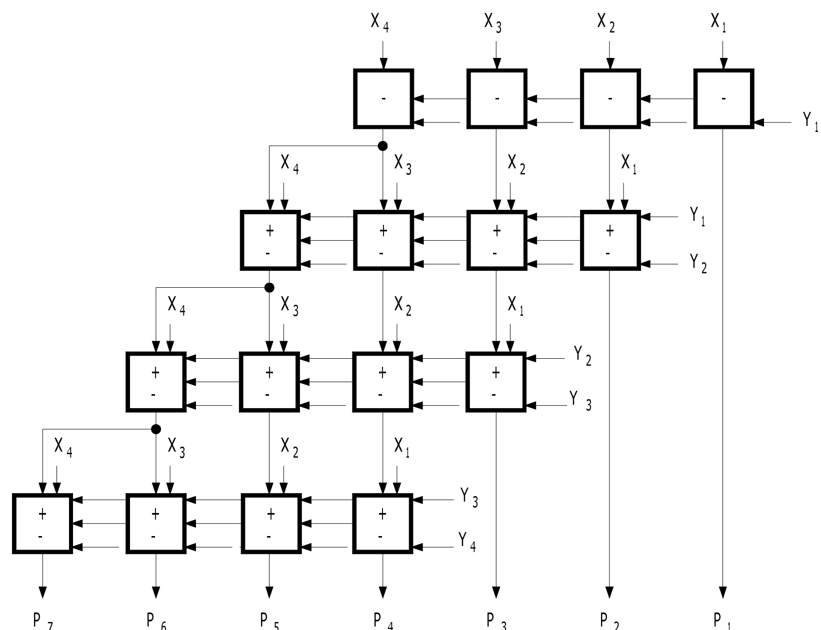


Abbildung 3.33: Multiplikationschaltkette nach BOOTH

Verwendung der $n \times n$-stelligen Multiplikation (realisiert als Festwertspeicher, Schaltnetz, Rechenwerk oder Befehl) miteinander multipliziert werden. Der Algorithmus ergibt sich aus der Beziehung:

$$\begin{aligned} P &= (A2_A1) * (B2_B1) = (A2 * 2^n + A1) * (B2 * 2^n + B1) \\ P &= (A2 * B2) * 2^{2n} + (A2 * B1 + A1 * B2) * 2^n + A1 * B1\ . \end{aligned}$$

Stehen ausreichend viele Hardware-Bauelemente zur Verfügung, dann läßt sich diese Formel direkt als Schaltnetz realisieren (Abb. 3.34). Wenn nur je ein Multiplikationsbaustein und ein Additionsbaustein zur Verfügung stehen, dann kann das Produkt $P = P4_P3_P2_P1$ schrittweise berechnet werden. $H1$ bis $H7$ sind Zwischenspeicher für die Teilprodukte, und C bis F dienen zum Speichern der Überträge.

Ein sequentielles Programm lautet:

boole $(A2, A1, B2, B1, H8, H7, H6, H5, H4, H3, H2, H1,$
$P4, P3, P2, P1)[n], C, D, E, F;$
1. $H2_H1 := A1 * B1;$ $\quad H4_H3 := A1 * B2;$
2. $H6_H5 := A2 * B1;$ $\quad H8_H7 := A2 * B2;$
3. $P1 := H1;$ $\quad C_P2 := H2 +\!\!+ H3;$
4. $D_P2 := P2 +\!\!+ H5;$ $\quad E_P3 := H4 +\!\!+ H7 + C;$
5. $F_P3 := P3 +\!\!+ H6 + D;$
6. $P4 := H8 + E + F;$

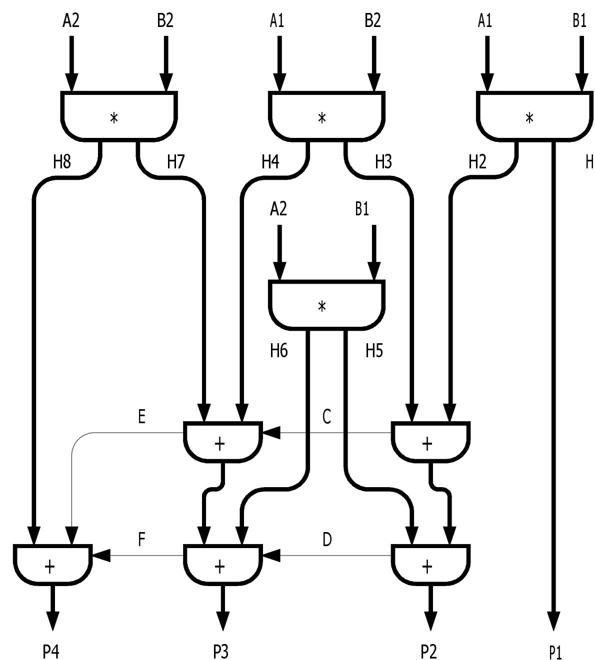


Abbildung 3.34: Parallele Doppelwort-Multiplikation

Das Programm benötigt zur Ausführung 10 Operationen. Wenn zwei Operationen parallel ausgeführt werden können, so werden nur noch 6 Schritte benötigt, entsprechend den oben aufgeführten Zeilen. Dabei sind die Zeilen 2 und 3 vertauschbar. Wenn 4 Operationen parallel ausgeführt werden können, dann können die 1. und 2. Zeile zusammengefaßt werden, so daß sich die Anzahl der Schritte auf das Minimum von 5 reduziert. Eine weitere Reduzierung der Schrittanzahl ist nur möglich, indem mächtigere Operatoren (z. B. Addition von 3 Operanden mit Überträgen) zur Verfügung gestellt werden.

3.2.5 Multiplikation von binärcodierten Dezimalzahlen

Da die Multiplikation von binärcodierten Dezimalzahlen aus Gründen des Aufwands meist nur seriell erfolgt, wird hier nur ein serielles Rechenwerk betrachtet. Zur Multiplikation von n-stelligen BCD-Zahlen ($X = X_n X_{n-1} \dots X_1,\ Y = Y_n Y_{n-1} \dots Y_1$) werden ein n-stelliges Multiplikandenregister X, ein $(n+1)$-stelliger Akkumulator $P2$, ein n-stelliges Multiplikatorregister $P1$ und ein Addierer für BCD-Ziffern (4-Bit-Addierer mit Pseudotetradenkorrektur) mit Übertragsflipflop C benötigt (Abb. 3.35). Alle Register sind 4 Bit breit, um die BCD-Ziffern stellenweise speichern und verarbeiten zu können. Im folgenden Algorithmus wird für die i-te Registerstelle à 4 Bit die Schreibweise $\langle i \rangle$ benutzt:

1. Das Multiplikandenregister X wird mit dem Multiplikanden X und das Multiplikatorregister $P1$ mit dem Multiplikator Y geladen. Der Akkumulator wird gelöscht ($P2 := 0$).

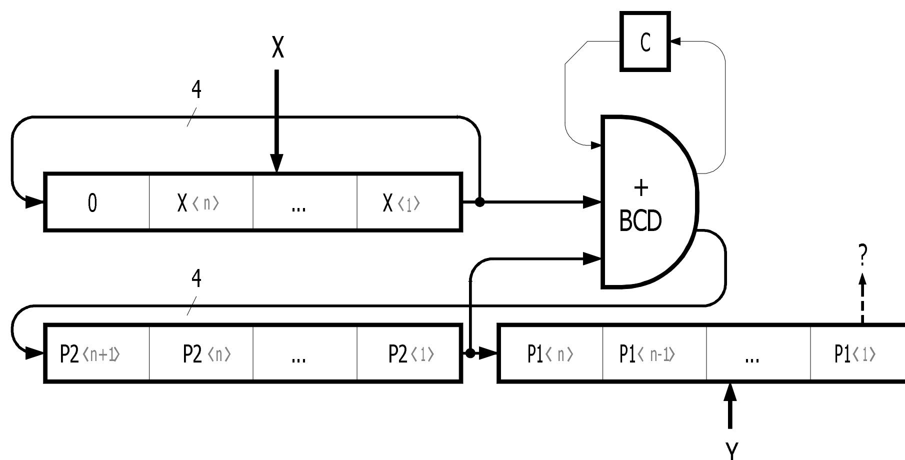


Abbildung 3.35: Serielles Multiplizierwerk für binärcodierte Dezimalzahlen

Die folgenden Schritte 2 und 3 werden n mal ausgeführt.

2. Die Stelle $P1\langle 1\rangle$ des Multiplikatorregisters (entspricht zuerst $Y\langle 1\rangle$, dann $Y\langle 2\rangle$, usw.) wird abgefragt. Dann wird der Multiplikand X $Y\langle i\rangle$-mal in den Akkumulator addiert ($P2 := P2 + X * Y\langle i\rangle$). Jede einzelne Addition erfordert $n+1$ Schritte, wenn sie, wie hier, seriell durchgeführt wird.

3. Die beiden Register ($P2_P1$) werden um eine Stelle nach rechts geschoben, wobei die Ziffer $P2\langle i\rangle$ nach $P1\langle n\rangle$ gelangt und $P2\langle n+1\rangle := 0$ gesetzt wird.

4. Das Produkt mit $2n$ BCD-Stellen steht in den beiden Registern ($P2_P1$), wobei $P2\langle n+1\rangle = 0$ ist.

Es läßt sich leicht verifizieren, daß durch diesen Algorithmus, nämlich

$P := 0;\ Z := X * 10^n;$
for $i := 1$ to n do $P := (P + Z * Y_i)/10$ od

die dezimale Multiplikation durchgeführt wird. Das Vorzeichen des Produkts wird bei vorzeichenbehafteten BCD-Zahlen getrennt ermittelt.

3.2.6 Multiplikation von Gleitkommazahlen

Die Multiplikation von Gleitkommazahlen ist im Vergleich zur Addition von Gleitkommazahlen einfach. Die Mantissen der beiden Operanden werden miteineinander multipliziert und die Exponenten zur Basis b werden unabhängig davon addiert.

$$\begin{aligned} p &= x * y = mp * b^{ep} = (mx * b^{ex}) * (my * b^{ey}) \\ &= (mx * my) * b^{ex+ey} \end{aligned}$$

Geht man für *mx* und *my* von normalisierten Mantissen $1/b \leq m < 1$ aus, dann liegt die Produkt-Mantisse zwischen den Grenzen

$$b^{-2} \leq |mp| < 1 \; . \tag{3.49}$$

Das bedeutet, daß das Produkt nicht notwendigerweise normalisiert ist. Wenn die höchstwertige Ziffer der Mantisse gleich Null ist, dann muß die Mantisse um eine Stelle nach links geschoben werden, und der Exponent muß um Eins erniedrigt werden.

Wird der Wertbereich des Exponenten nach oben oder unten überschritten, dann wird die Überlaufanzeige (Overflow) oder die Unterlaufanzeige (Underflow) gesetzt.

Es folgt eine allgemeine, algorithmische Beschreibung für normalisierte Gleitkommazahlen, in der die Darstellungsform noch nicht festgelegt ist. Anweisungen, die in eckigen Klammern eingeschlossen und durch ein Komma getrennt sind, können parallel ausgeführt werden:

if $(x = 0) \vee (y = 0)$ then $p := 0$
else $mp := mx * my$;
  if $1/b \leq |mp| < 1$ „normalisiert“
    then $ep := ex + ey$
  else „$b^{-2} \leq |mp| < 1/b$“ „Normalisieren“
  $[ep := ex + ey - 1, mp := mp * b]$ fi;
  [if $ep > ep_{\max}$ then „Exponentenüberlauf“ fi,
  if $ep < ep_{\min}$ then „Exponentenunterlauf, evtl. $p := 0$“ fi]
fi.

Wenn *mx* und *my* n-stellig sind, sollte die Multiplikation mindestens auf $n + 1$ Stellen genau ausgeführt werden, damit beim Normalisieren eine gültige Ziffer nachgezogen werden kann. Als Darstellungsform für die Mantisse eignet sich am besten die Vorzeichen-Betrag-Darstellung oder Einskomplementdarstellung. Von der Benutzung der Zweikomplementdarstellung für die Mantisse ist abzuraten, da Sonderbehandlungen bei der Multiplikation und dem Normalisieren für die betragsmäßig größte negative Mantisse erforderlich werden.

3.3 Division

Die Division ist die schwierigste Grundrechenart, für die eine Reihe ganz unterschiedlicher Berechnungsmethoden existiert. Die einfachste, aber aufwendigste Methode besteht darin, alle Quotienten in einer Tabelle (Festwertspeicher oder minimiertes Schaltnetz) direkt abzuspeichern. Meist werden die Quotientenziffern jedoch ziffernweise durch fortlaufende Subtraktionen und Vergleiche ermittelt. Bei den iterativen Verfahren (z. B. NEWTONsche Näherung) wird die Genauigkeit des Quotienten schrittweise verbessert, ausgehend von einem Startwert. Die Rechenzeit wird dabei umso kleiner, je genauer der Startwert (größere Startwertabelle) gewählt wird. In diesem Abschnitt werden nur die klassischen Divisionsverfahren

behandelt, die auf der Addition und Subtraktion basieren, und nicht iterative Methoden, die die Multiplikation voraussetzen, siehe z. B. [Wal, Str].

In den folgenden Betrachtungen beschränken wir uns auf die ganzzahlige Division, bei der der *Quotient* (ganzzahlig) und der *Rest* aus der Beziehung

$$(\textit{Rest}) = (\text{Dividend}) - (\textit{Quotient}) * (\text{Divisor}) \quad \text{bzw.}$$

$$(\textit{Quotient}) = \frac{(\text{Dividend}) - (\textit{Rest})}{(\text{Divisor})}$$

ermittelt werden.

Der echte Quotient (rationale Zahl) ergibt sich durch Addition von (Rest)/(Divisor) zu dem ganzzahligen Quotienten.

Für die ganzzahlige Division sind im wesentlichen drei Verfahren gebräuchlich:

1. die Vergleichsmethode,
2. die Methode mit Rückstellen des Restes und
3. die Methode ohne Rückstellen des Restes.

Bei der allgemein bekannten Vergleichsmethode wird durch einen Vergleich festgestellt, wie oft der Divisor in den höherwertigen Teil des Dividenden paßt. Diese Anzahl ergibt eine Quotientenstelle. Der Dividend wird dann sukzessive um das Teilprodukt (Quotientstelle mal Divisor) verringert und steht dann für einen erneuten Vergleich zur Verfügung. Die Division ist beendet, wenn der Rest kleiner als der Divisor ist. Für die Realisierung dieser Methode benötigt man ein Vergleichsschaltnetz mit b Entscheidungen, wenn b die Basis des Zahlensystems ist.

Besonders einfach gestaltet sich die Division für Dualzahlen, weil das Vergleichsschaltnetz nur die Aussagen „Divisor paßt“ und „Divisor paßt nicht“ liefern muß. Wenn der Divisor in den höherwertigen Teil des Dividenden paßt, dann wird das Quotientenbit $= 1$ gesetzt, und der Divisor wird stellenrichtig vom Dividenden subtrahiert. Im anderen Falle wird das Quotientenbit $= 0$ gesetzt und keine Subtraktion durchgeführt. Mit jedem neuen Schritt wird eine weitere Stelle des Dividenden mit zum Vergleich heranzogen.

Bei der Methode „Mit Rückstellen des Restes“ wird das Vergleichsschaltnetz eingespart, indem der Divisor maximal $(b-1)$-mal vom höherwertigen Teil des Dividenden abgezogen wird. Ergibt sich bei der k-ten Subtraktion ein negativer (Zwischen-)Rest, dann ist das ein Kennzeichen dafür, daß der Divisor nur $(k-1)$-mal in den betrachteten Teil des Dividenden paßt. Bei Auftreten eines negativen (Zwischen-)Restes wird eine Korrektur durch Addition des Divisors (Rückstellen des Restes) angeschlossen. Diese Methode wird im Dividierwerk für binärcodierte Dezimalzahlen (Abschnitt 3.3.4) angewandt.

Die Methode „Ohne Rückstellen des Restes“ kommt mit weniger Schritten als die Methode mit Rückstellen des Restes aus. Dabei wird ein negativer (Zwischen-)Rest nicht zurückgestellt, sondern im nächsten Rekursionsschritt weiterverarbeitet, indem der Dividend maximal $(b-1)$-mal addiert wird. Immer wenn ein positiver (Zwischen-)Rest entsteht, wird im darauffolgenden Rekursionsschritt subtrahiert und umgekehrt. Ensteht zum Schluß des Verfahrens ein negativer Rest, so muß dieser durch Addition des Divisors korrigiert werden. Die Methode ohne Rückstellen des Restes wird im folgenden Abschnitt für Dualzahlen entwickelt.

3.3.1 Division von Dualzahlen

Bei der Divisionsaufgabe sind der Dividend p und der Divisor d vorgegeben. Gesucht ist der ganzzahlige Quotient q und der Rest r, wobei die Beziehung

$$r = p - q * d \qquad \text{mit} \qquad 0 \leq r < d \tag{3.50}$$

erfüllt sein muß. Die Division wird zunächst für Dualzahlen mit einem vorgestelltem Bit, das den Wert $\circ$ hat, betrachtet.

$$\begin{aligned} p = P[2n-1] &= \circ\, P_{2n-2} \dots P_1 \\ q = Q[n] &= \circ\, Q_{n-1} \dots Q_1 \\ d = D[n] &= \circ\, D_{n-1} \dots D_1 \\ r = R[n] &= \circ\, R_{n-1} \dots R_1 \, . \end{aligned}$$

Die vorgestellte Bitstelle kann bei Vorzeichenzahlen zur Codierung des Vorzeichens benutzt werden. Diese Stelle wird auch bei der späteren Betrachtung der Division im Zweikomplement benötigt.

Bei der Division kann ein Divisionsüberlauf entstehen, wenn der Quotient nicht mit den verfügbaren Stellen dargestellt werden kann. Wenn $n-1$ Stellen für den Betrag des Quotienten vorgesehen sind, dann darf er nicht größer als $2^{n-1}-1$ werden. Wenn kein Überlauf auftreten soll, dann müssen die folgenden Bedingungen erfüllt sein:

$$\begin{aligned} 0 &\leq q \leq 2^{n-1} - 1 \\ 0 &\leq q * d + r \leq (2^{n-1} - 1) * d + r < 2^{n-1} * d \\ 0 &\leq p < 2^{n-1} * d \, . \end{aligned} \tag{3.51}$$

Der um $n-1$ Stellen verschobene Divisor d muß also größer als der Dividend p sein. Diese Bedingung muß vor Beginn der eigentlichen Division abgefragt werden. Dieser Test kann durch ein Vergleichsschaltnetz oder durch die Test-Subtraktion $(p-2^{n-1}*d < 0)$ erfolgen.

Wir wollen zuerst die Vergleichsmethode entwickeln. Dabei wird vom Dividenden schrittweise (Quotient mal Divisor) abgezogen, wobei gleichzeitig die unbekannten Quotientenbits ermittelt

werden.

$$R = P - (Q_n Q_{n-1} \cdots Q_1) * D$$
$$R = P - Q_n * D * 2^{n-1} - Q_{n-1} * D * 2^{n-2} - \cdots Q_1 * D \quad (3.52)$$
$$R = ((P2^{-(n-1)} - Q_n * D)2 - Q_{n-1} * D)2 - \cdots Q_1 * D \,. \quad (3.53)$$

Daraus folgt das Rekursionsschema

$$\begin{aligned} R^n &= P * 2^{-(n-1)} - Q_n * D \\ R^{n-1} &= R^n * 2 - Q_{n-1} * D \\ &\vdots \\ R^1 &= R^2 * 2 - Q_1 * D \,. \end{aligned} \quad (3.54)$$

Bei der Vergleichsmethode wird zuerst überprüft, ob der Divisor in den höherwertigen Teil des Dividenden paßt ($D \leq 2R^i$). Wenn der Divisor paßt, dann wird das Quotientenbit $Q_{i-1} = \mathtt{1}$ gesetzt und die Subtraktion $R^{i-1} = 2R^i - D$ führt zu einem positiven (Zwischen-)Rest. Die Realisierung des Vergleichs erfordert eine Vergleichsoperation oder ein Vergleichsschaltnetz, die bei der *Methode mit Rückstellen des Restes* eingespart werden können. Der i-te Schritt $R^{i-1} := R^i * 2 - Q_{i-1} * D$ kann wie folgt umgeformt werden:

$R^{i-1} := R^i * 2 - D;$
if $R^{i-1} < 0$ then $Q^{i-1} := \mathtt{0}$; $R^{i-1} := R^{i-1} + D$; „Rückstellen"
else $Q^{i-1} := \mathtt{1}$;   fi .

Dabei wird in jedem Schritt zuerst der Divisor probeweise subtrahiert; wird der Teilrest R^{i-1} negativ, dann war die Subtraktion unberechtigt ($Q_{i-1} = \mathtt{0}$), und der Teilrest wird durch Addition von D zurückgestellt.

Das umständliche Rückstellen des Restes wird bei der *Methode ohne Rückstellen des Restes* vermieden. Eine Umformung der Gleichung (3.52) ergibt

$$\begin{aligned} R = {} & P - D * 2^{n-1} + (1 - 2Q_n)D * 2^{n-2} + \cdots \\ & + (1 - 2Q_2)D + (1 - Q_1)D \,. \end{aligned} \quad (3.55)$$

Daraus folgt das Rekursionsschema (3.56)

$$\begin{aligned} R^n &= P * 2^{-(n-1)} - D && \rightarrow \; Q_n = (R_n \geq 0) \\ & && \text{if } Q_n = \mathtt{1} \text{ then „Überlauf"} \\ R^{n-1} &= R^n * 2 + (1 - 2 * Q_n) * D && \rightarrow \; Q_{n-1} = (R_{n-1} \geq 0) \\ &\vdots \\ R^1 &= R^2 * 2 + (1 - 2 * Q_2) * D && \rightarrow \; Q_1 = (R_1 \geq 0) \\ R &= R^1 + (1 - Q_1) * D && . \end{aligned}$$

Im ersten Schritt wird durch Subtraktion von D die Überlaufbedingung $p < 2^{n-1} * d$ abgefragt. Wenn $R_n < 0$ ist, dann tritt kein Überlauf auf, und Q_n ist gleich $\circ$. Im zweiten Schritt wird der Divisor D addiert, und Q_{n-1} ist gleich $\mathtt{1}$ für $R^{n-1} \geq 0$. In den darauffolgenden Schritten wird D subtrahiert bzw. aufaddiert, je nachdem, ob der vorhergehende Teilrest positiv oder negativ ist. Entsteht am Ende ein negativer Rest $R^1 \quad (Q_1 = \circ)$, dann muß als Korrektur der Divisor addiert werden. Die teilweise notwendige Subtraktion wird zweckmäßigerweise im 2-Komplement durch Addition des Komplements $\overline{D} + 1$ durchgeführt. Das negierte Vorzeichen des Teilrestes gibt dann direkt das gesuchte Quotientenbit an $(Q_i = \overline{R_n^i})$.

Im folgenden wird ein Dividierwerk für positive Dualzahlen beschrieben, das die Division eines $2n$-stelligen Dividenden ($\circ\ P_{2n-1} \ldots P_1\ \circ$) durch einen n-stelligen Divisor ($\circ\ D_{n-1} \ldots D_1$) realisiert. Als Ergebnis entsteht ein n-stelliger Quotient ($\circ\ Q_{n-1} \ldots Q_1$) und ein n-stelliger Rest ($\circ\ R_{n-1} \ldots R_1$). Die Vorzeichen müssen getrennt behandelt werden.

Das Dividierwerk (Abb. 3.36) enthält ein Register D zur Aufnahme des Divisors, zwei Register R und Q, die zu Beginn den Dividenden enthalten und am Ende den Rest und den Quotienten aufnehmen, sowie einen Addierer/Subtrahierer. Die Division erfolgt nach dem Algorithmus ohne Rückstellen des Restes (vergl. Abb. 3.37):

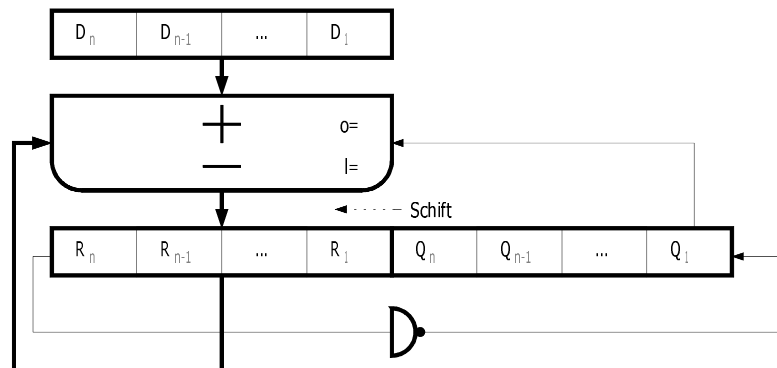


Abbildung 3.36: Dividierwerk für Dualzahlen

1. Das Register D wird mit dem Divisor und die Register R_Q werden mit dem Dividenden geladen.

2. Der Divisor wird vom Inhalt des R-Registers, der den höherwertigen Teil des Dividenden enthält, subtrahiert, um einen möglichen Divisionsüberlauf festzustellen ($R \leftarrow R - D$). Die Subtraktion muß einen negativen Zwischenrest R ergeben (d. h. $R_n = \mathtt{1}$), denn nur dann ist der Quotient mit n Stellen darstellbar. Ist $R_n = \circ$, dann wird die Überlaufanzeige gesetzt und die Division abgebrochen.

   Die Schritte 3 und 4 werden $(n - 1)$ mal durchgeführt.

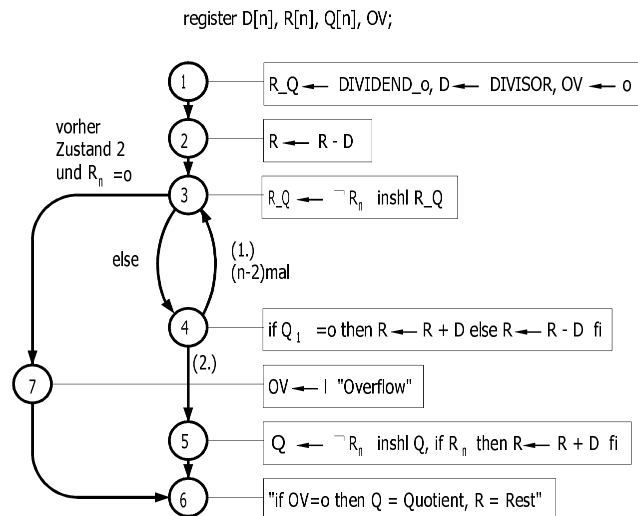


Abbildung 3.37: Mikroalgorithmus für die Division von Dualzahlen

3. Die Register R_Q werden um eine Stelle nach links geschoben und Q_1 wird gleich R_n gesetzt.

4. Wenn $Q_1 = \circ$ ist, dann wird D zu R addiert $(R \leftarrow R + D)$, andernfalls wird D von R subtrahiert $(R \leftarrow R - D)$.

5. Nur das Q-Register wird nach links geschoben, und R_n wird nachgezogen. Wenn der Rest negativ ist ($R_n = \mathtt{1}$), dann muß die Korrektur $R \leftarrow R + D$ durchgeführt werden.

Es besteht die Möglichkeit, die Zustände 3 und 4 des Zustandsdiagramms zusammenzufassen, um dadurch die Rechenzeit zu verkürzen.

Zahlenbeispiel: $34/5 = 6$ Rest 4

```
                P_Q = o l o o o l o = DIVIDEND_o
                  D = o l o l       = DIVISOR
                 D' = l o l l
Zustand
  1          R_Q ← o l o o o l o o
  2            R ← l l l l            +D'
  3          R_Q ← l l l o l o o o    shl
  4            R ← o o l l            +D
  3          R_Q ← o l l l o o o l    shl
  4            R ← o o l o            +D'
  3          R_Q ← o l o o o o l l    shl
  4            R ← l l l l            +D'
  5          R_Q ← o l o o o l l o    +D, shl Q
                       4       6
```

3.3.2 Division von Zweikomplementzahlen

Die Lösung der Divisionsaufgabe $p - q * d = r$ wird nun auf negative Zahlen, dargestellt im Zweikomplement, erweitert. Es zeigt sich, daß relativ aufwendige Korrekturen zur Erzielung des richtigen Resultats erforderlich sind. Unter Verwendung der Rückabbildungsgleichung (1.8) ergibt sich der Rest für einen $(2n-1)$-stelligen Dividenden zu

$$\begin{aligned} R[n] &= P[2n-1] - P_{2n-1} * 2^{2n-1} + R_n * 2^n \\ &\quad - (Q[n] - Q_n * 2^n) * (D[n] - D_n * 2^n) \ . \end{aligned} \tag{3.57}$$

Dabei ist zu beachten, daß das Vorzeichen des Quotienten und des Restes gleich $P_{2n-1} \not\equiv D_n$ sein muß. Als Dividend wird die betragsmäßig größte negative Zahl $P =$ `loo...o` verboten, da sie nicht komplementiert werden kann. In Abhängigkeit von den Vorzeichen lassen sich 4 Fälle unterscheiden:

Fall 1: p und d positiv

Die Division wird wie für positive Dualzahlen nach dem Algorithmus im Abschnitt 3.3.1 durchgeführt.

Fall 2: p negativ, d positiv

Im Prinzip kann dieser Fall auf Fall 1 zurückgeführt weden, indem P vorher und Q und R nachher komplementiert werden. Umständlich ist dabei die Komplementbildung des $(2n-1)$-stelligen Dividenden, der meist in zwei Registern abgespeichert wird. Eleganter, aber auch nicht wesentlich einfacher ist es, direkt mit den Zweikomplementzahlen zu rechnen. Die dafür

notwendigen Korrekturen werden im folgenden ermittelt. Grundsätzlich ist festzustellen, daß die bisher betrachteten Divisionsalgorithmen nur für positive Dualzahlen funktionieren. Die Methoden „Mit“ und „Ohne Rückstellen des Restes“ gehen von einem positiven Dividenden und einem positiven Divisor aus, wobei am Ende des Algorithmus nur ein positiver Rest erlaubt ist. Ein negativer Dividend wird im Zweikomplement durch eine positive Dualzahl repräsentiert, wodurch bereits die Forderung nach einem positiven Dividenden und Divisor erfüllt ist. Da der Divisionsalgorithmus nur einen positiven „Rest“ liefert, muß untersucht werden, welche Bedeutung diesem „Pseudorest“ zukommt und wie der generierte „Pseudoquotient“ korrigiert werden muß. Je nach Größe des Restes löst der Algorithmus nämlich zwei unterschiedliche Gleichungen.

Fall 2a: Rest gleich Null

Es wird die Gleichung $0 = p - q * d$ gelöst, die im Zweikomplement lautet:

$$0 = P - Q * D + D * 2^n - 2^{2n-1} \; . \tag{3.58}$$

Unter Verwendung von (3.55) für die Division ohne Rückstellen des Restes ergibt sich:

$$0 = \underbrace{P - D * 2^{n-1} + D * 2^n}_{= +D * 2^{n-1}} + (1 - 2 * Q_n) * D * 2^{n-2} + \cdots \; . \tag{3.59}$$

Im Gegensatz zur Division von positiven Dualzahlen wird am Anfang der Divisor addiert und nicht subtrahiert, wobei gleichzeitig der Term -2^{2n-1} (vergl. 3.58) durch Abschneiden des entstehenden Übertrages berücksichtigt wird. In dem Rekursionsschema (3.56) ist nur die erste Zeile zu ersetzen:

$$R^n = P * 2^{-(n-1)} + D \quad \longrightarrow \quad Q_n = (R^n \geq 0) \; . \tag{3.60}$$

Im Fall 2 und Fall 4, der auf Fall 2 zurückgeführt wird, kann der betragsmäßig größte negative Quotient $q = -2^{n-1}$ entstehen. Möchte man diesen Wert von der weiteren Verarbeitung wegen der zu erwarteten Komplikationen bei der Komplementbildung ausschließen, so setzt man die Überlaufbedingung bei $R^n \leq 0$, ansonsten bei $R^n < 0 \;\; (Q_n = o)$.

Fall 2b: Negativer Rest

Für $-d < r < 0$ wird die Gleichung

$$\widehat{r} \;\; = \;\; r + d = p - (q - 1) * d = p - (\widehat{q} * d) \tag{3.61}$$

gelöst [Hof-4], denn der Wert des $2K$-Dividenden ist um den Rest kleiner. Der Divisionsalgorithmus liefert einen Pseudorest $\widehat{r}$, der die Bedingung $0 < \widehat{r} < d$ erfüllt und einen Pseudoquotienten $\widehat{q}$. Zur Ermittlung des gesuchten Quotienten und des Restes müssen folgende Korrekturen angeschlossen werden:

$$r := \widehat{r} - d \qquad \text{und} \qquad q := \widehat{q} + 1 \; . \tag{3.62}$$

In Zweikomplementdarstellung wird also zunächst folgende Gleichung aufgelöst:

$$\widehat{R} = P - \widehat{Q} * D + D * 2^n - 2^{2n-1} \,. \tag{3.63}$$

Zur Auflösung dieser Beziehung wird das gleiche Rekursionsschema wie für den Fall 2a verwendet. Ob bei dieser Auflösung der richtige Quotient oder der Pseudoquotient entstanden ist, läßt sich erst zum Schluß feststellen. Wenn dann der Rest gleich Null ist, dann ist der Quotient richtig. Ist er > 0, dann muß der Divisor abgezogen und der Pseudoquotient um 1 erhöht werden.

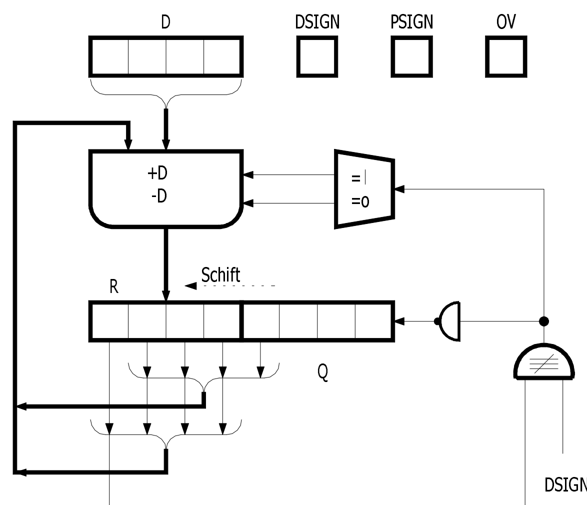


Abbildung 3.38: Rechenwerk für die Division im Zweikomplement

Fall 3: p **positiv,** d **negativ**

Dieser Fall wird auf Fall 1 zurückgeführt, indem der Divisor vorher $(D := 2^n - D)$ und der Quotient nachher $(Q := \overline{Q} + 1)$ komplementiert werden. Die Komplementbildung des Quotienten kann schon während der Auflösung vorgenommen werden, indem die anfallenden Quotientenbits negiert werden. Zum Schluß muß dann noch $+1$ addiert werden, damit die Komplementbildung vollständig wird.

Fall 4: p **negativ,** d **negativ**
Dieser Fall wird auf Fall 2 zurückgeführt, indem der Divisor vorher $(D := 2^n - 1)$ und der Quotient nachher komplementiert werden. Ensteht kein Pseudorest $(\widehat{R} = 0)$, dann ist $Q := \widehat{Q} + 1$; wenn ein Pseudorest $(\widehat{R} > 0)$ entsteht, dann müssen die Korrekturen $R := R - D$ und $Q := 2^n - (\widehat{Q} + 1) = \overline{\widehat{Q}}$ durchgeführt werden. Dabei kann die Negation des Quotienten schon bitweise während der Abarbeitung erfolgen.

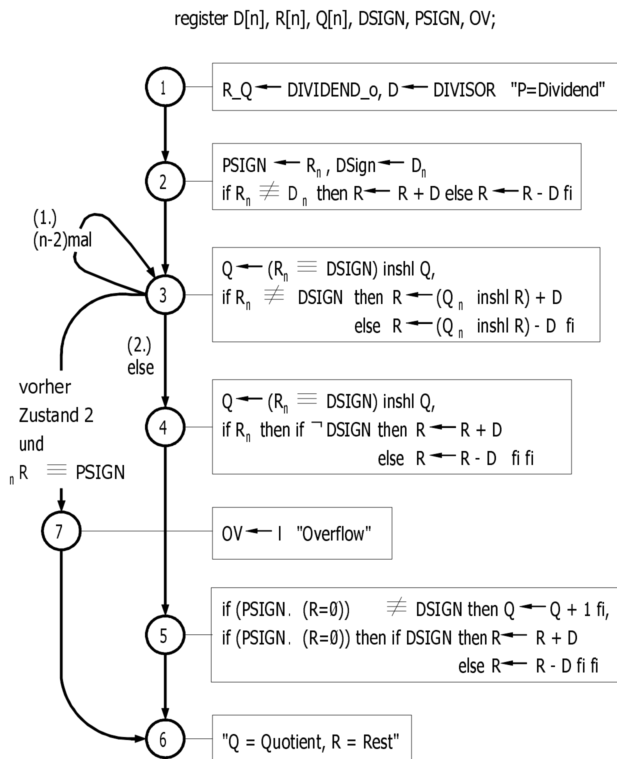


Abbildung 3.39: Mikroalgorithmus für die Division im Zweikomplement

Abb. 3.38 zeigt das Rechenwerk und Abb. 3.39 das synchrone Mikroprogramm für die Division im Zweikomplement. Daran anschließend werden die verschiedenen Fälle der Division anhand von Beispielen (Abb. 3.40) veranschaulicht.

Zur Verdeutlichung werden das 3. und 6. Zahlenbeispiel aus Abb. 3.40 nach dem Algorithmus aus Abb. 3.39 schrittweise durchgerechnet.

3. Beispiel: $7/3 = 2$ Rest 1

```
           P = o.o l l l     p = 7
           D = o.l l         d = 3
           D'= l.o l
Zustand
  1      R_Q   <- o o l l l o
  2                [+D']
         R     <- l l o             PSIGN ← o, DSIGN ← o
  3                [+D ]            shl
         R_Q   <- o o o l o o
  3                [+D']            shl
         R_Q   <- l l o o o l
  4                [+D ]            shl Q
         R_Q   <- o o l o l o       „Rest u. Quotient, keine Korr."
                    1     2
```

6. Beispiel: $-7/3 = -2$ Rest -1

```
           P  = l.l o o l     p = -7
           D  = o.l l         d =  3
           D' = l.o l

Zustand
  1      R_Q   <- l l o o l o
  2                [+D ]
         R     <- o o l             PSIGN ← o, DSIGN ← o
  3                [+D']            shl
         R_Q   <- l l l l o l
  3                [+D ]            shl
         R_Q   <- o l o o l o
  4                                 shl Q
         R_Q   <- o l o l o l
  5                [+D'] [+1 ]      „Korrektur"
         R_Q   <- l l l l l o       „Rest und Quotient"
                   -1    -2
```

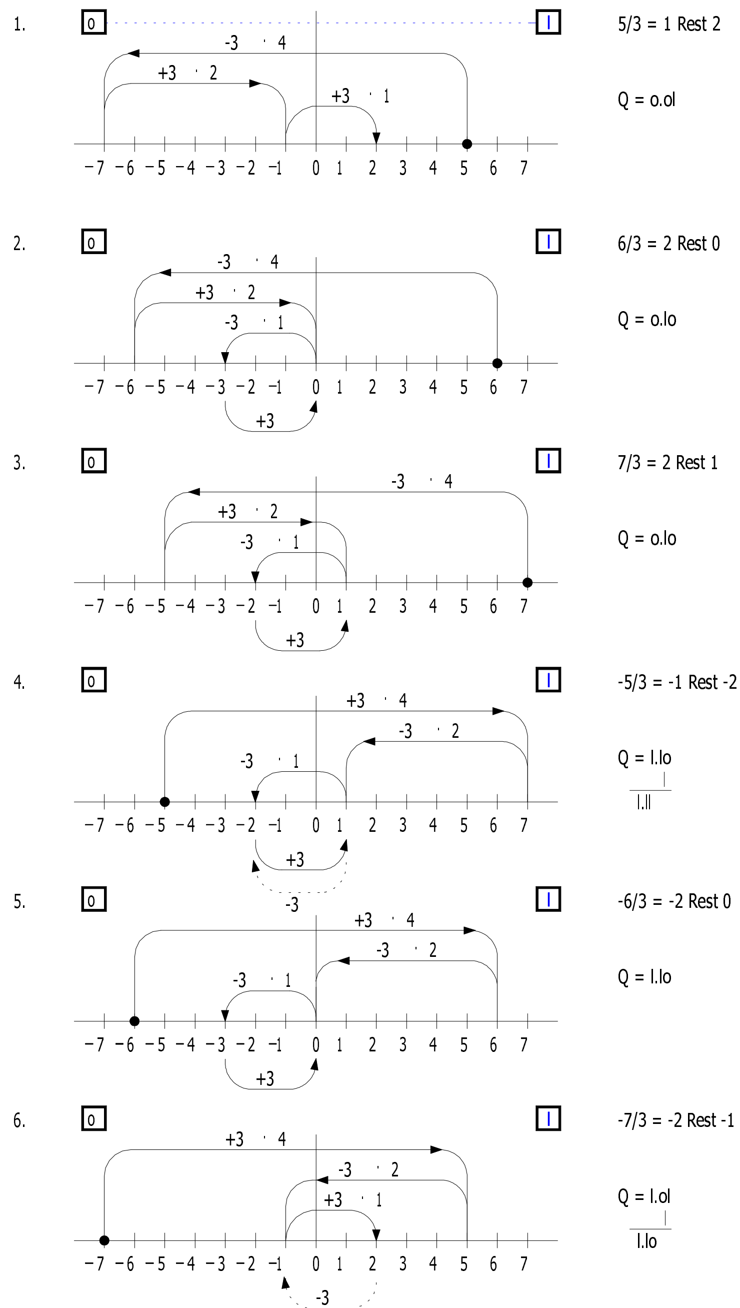


Abbildung 3.40: Anschauliche Beispiele für die Division im Zweikomplement

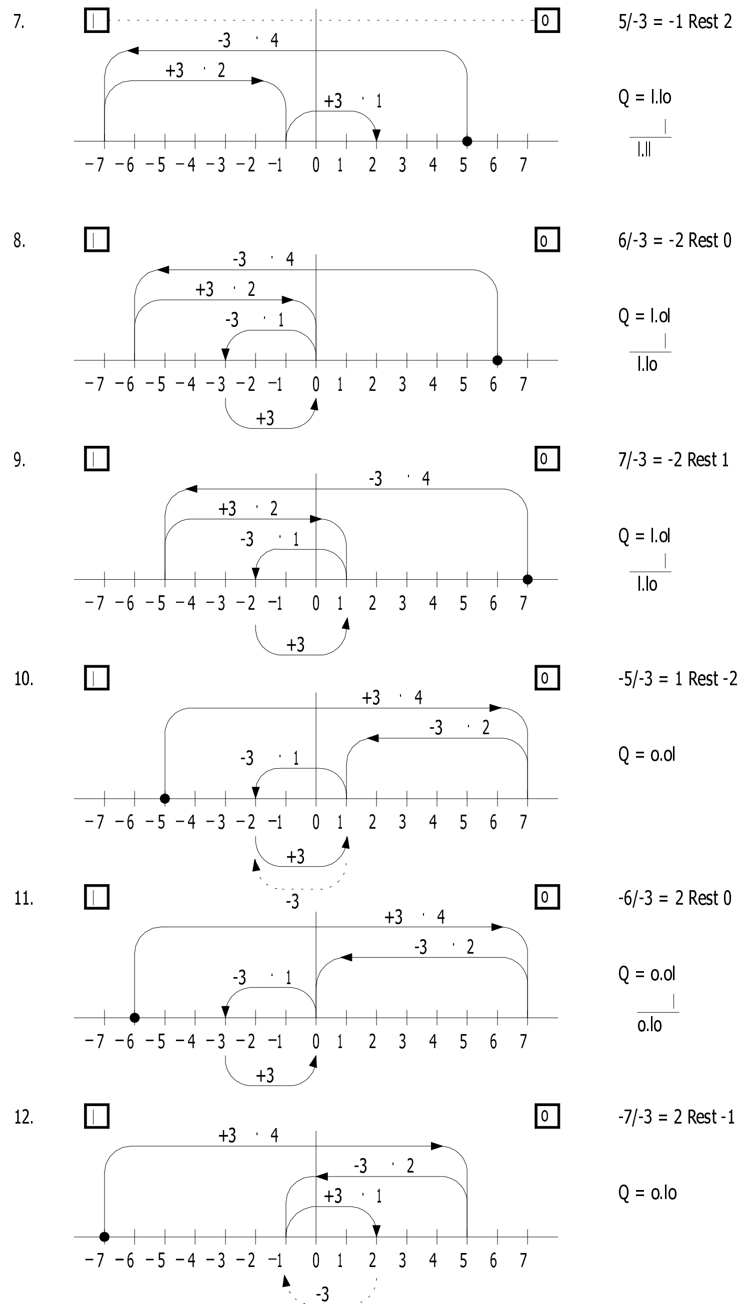


Abbildung 3.41: Anschauliche Beispiele für die Division im Zweikomplement

3.3.3 Parallele Division

Die bisher betrachteten Divisionsalgorithmen lassen sich natürlich auch als parallele Schaltkette realisieren, indem jeder Zeit-Schritt in einen Raum-Schritt (Schaltkettenglied) umgesetzt wird. Auch hier erweist sich die „Methode ohne Rückstellen des Restes“ als die günstigste. Abb. 3.42 zeigt eine Divisionsschaltkette für positive Dualzahlen. Dabei wird der 7-stellige Dividend P durch den 4-stelligen Divisor D dividiert und der 3-stellige Quotient Q und der 4-stellig Rest R werden berechnet. Jede Stufe der Schaltkette besteht aus einem Addierer/Subtrahierer, der durch das von links eintreffende Steuersignal kontrolliert wird. Die Subtraktion wird im Zweikomplement durchgeführt. Die Verbindung des Divisors D mit den Addierern/Subtrahierern ist in Abb. 3.42 nicht eingezeichnet.

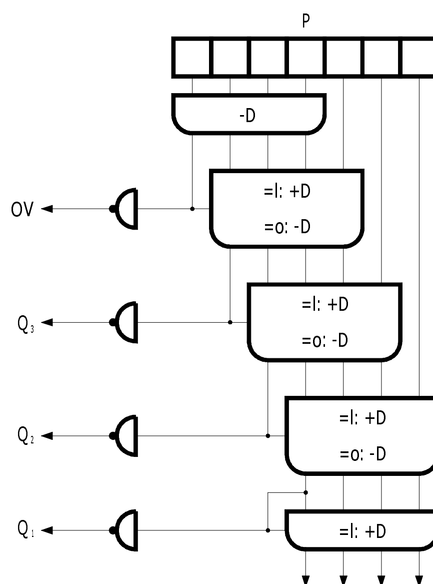


Abbildung 3.42: Divisionsschaltkette für Dualzahlen

Die parallele Division für Zweikomplementzahlen erfordert eine aufwendige Schaltkette, die durch eine räumliche Abwicklung des Algorithmus nach Abb. 3.37 gewonnen werden kann. Abb. 3.43 zeigt als Beispiel das Divisionsschaltnetz für einen 7-stelligen Dividenden P und einen 4-stelligen Divisor D. Durch die vorletzte Stufe wird der Rest bzw. Pseudorest positiv gemacht. Ist bei negativen Dividenden der Pseudorest $= 0$, dann wird in der letzten Stufe der Pseudorest korrigiert und der Pseudoquotient um $+1$ erhöht (Gleichung 3.62).

3.3.4 Division von binärcodierten Dezimalzahlen

Die Division von binärcodierten Dezimalzahlen wird anhand eines Dividierwerks beschrieben, das die Division eines $(2n-2)$-stelligen positiven Dividenden $(P_{2n-2} \dots P_1)_{\mathrm{BCD}}$ durch einen

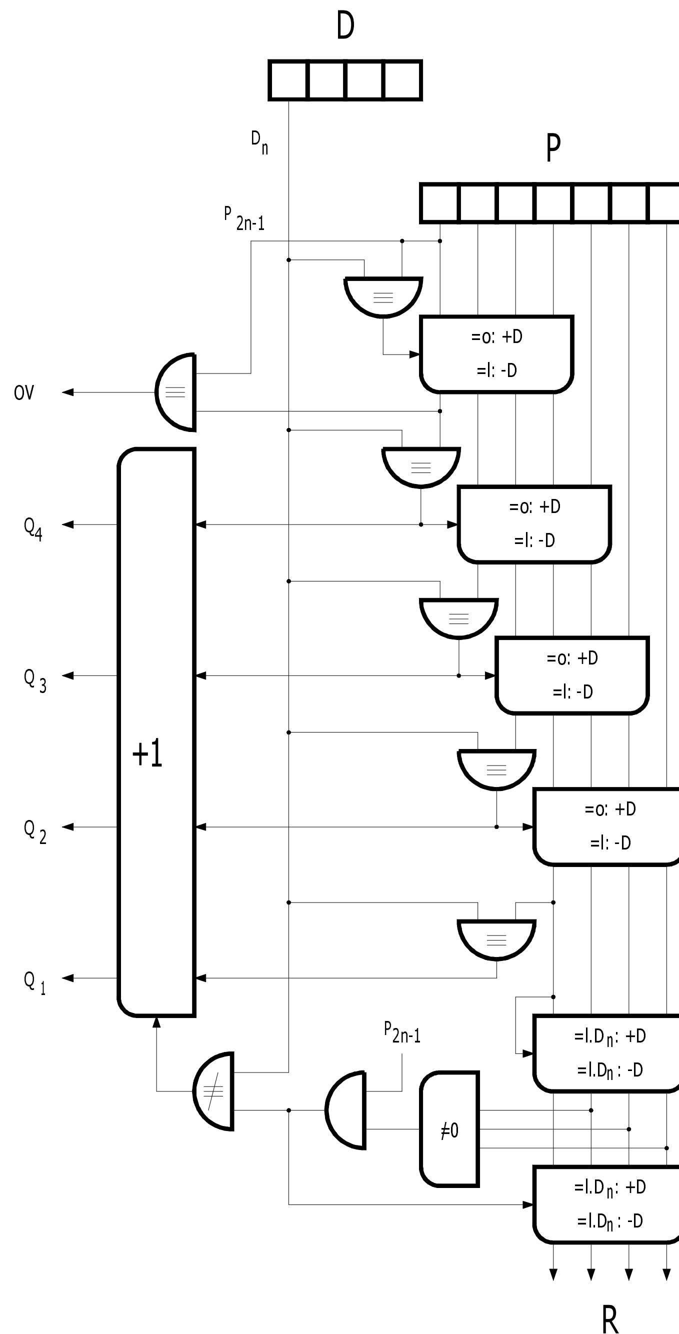


Abbildung 3.43: Divisionsschaltkette für Zweikomplementzahlen

$(n-1)$-stelligen positiven Divisor $(D_{n-1} \ldots D_1)_{\mathrm{BCD}}$ seriell durchführt. Als Ergebnis ensteht der Quotient $(0\ Q_{n-1} \ldots Q_1)_{\mathrm{BCD}}$ und der Rest $(0\ R_{n-1} \ldots R_1)_{\mathrm{BCD}}$. Die Vorzeichen müssen getrennt behandelt werden. Das Dividierwerk (Abb. 3.44) enthält ein Register D zur Aufnahme des Divisors, zwei Register R und Q, die zu Beginn den Dividenden enthalten und am Ende

den Rest und den Quotienten aufnehmen, sowie einen BCD-Addierer/Subtrahierer für eine BCD-Stelle. Alle Register sind 4 Bit breit.

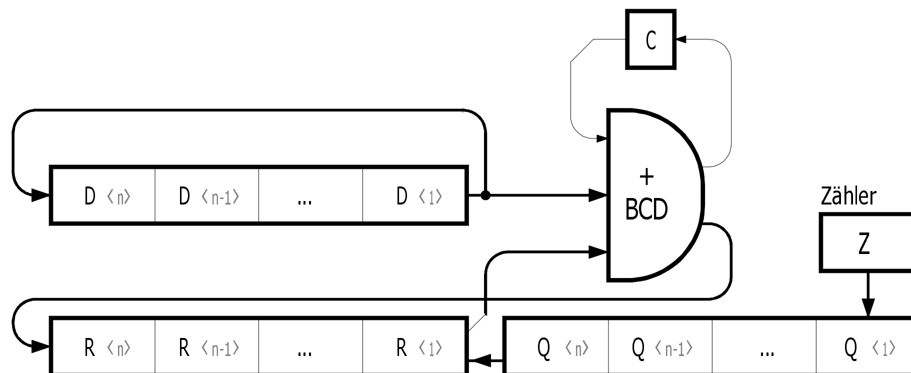


Abbildung 3.44: Dividierwerk für binärcodierte Dezimalzahlen

Die Division erfolgt nach der Methode mit Rückstellen des Restes, wobei die Schreibweise $\langle i\rangle$ für die i-te Registerstelle á 4 Bit benutzt wird:

1. Das Register D wird mit dem Divisor $(0\,D_{n-1}\ldots D_1)_{\text{BCD}}$ und die Register R_Q werden mit dem Dividenden $(0\,P_{2n-2}\ldots P_1\,0)_{\text{BCD}}$ geladen. Der Zähler Z wird gelöscht $(Z := 0)$.

2. Der Divisor wird vom Inhalt des R-Registers, der den höherwertigen Teil des Dividenden enthält, teilweise subtrahiert, um einen möglichen Divisionsüberlauf festzustellen $(R := R - D)$. Die Subtraktion erfolgt wie die Addition ziffernweise in n Schritten. Ergibt die Subtraktion ein positives Ergebnis $(R\langle n\rangle = 0)$, dann wird die Überlaufanzeige gesetzt und die Division abgebrochen. Bei einem negativen Ergebnis $(R\langle n\rangle \neq 0)$ wird die Testoperation rückgängig gemacht $(R := R + D)$.

   Die folgenden Schritte 3 und 4 werden $(n-1)$-mal wiederholt.

3. Die Register R_Q werden um eine Stelle nach links geschoben, und der Inhalt des Zählers wird nach $Q\langle 1\rangle$ gebracht $(Q\langle 1\rangle := Z)$.

4. Von dem R-Register, das den Zwischenrest enthält, wird der Divisor so oft abgezogen, bis der Zwischenrest < 0 wird $(R\langle n\rangle \neq 0)$. Der Zähler zählt dabei die Anzahl der ausgeführten Subtraktionen bis auf die letzte. Daran anschließend wird die letzte Subtraktion wieder rückgängig gemacht $(R := R + D)$.

5. Das Q-Register wird um eine Stelle nach links geschoben, und $Q\langle 1\rangle$ wird mit dem Inhalt des Zählers geladen $(Q\langle 1\rangle := Z)$.

3.3.5 Division von Gleitkommazahlen

Zwei Gleitkommazahlen werden dividiert, indem ihre Mantissen dividiert und ihre Exponenten zur Basis b subtrahiert werden:

$$q \quad = \quad x/y = mq * b^{eq} = (mx/my) * b^{ex-ey} \,. \tag{3.64}$$

Geht man von normalisierten Mantissen aus, d. h. $1/b \le |mx| < 1$ und $1/b \le |my| < 1$, dann liegt die Quotienten-Mantisse zwischen den Grenzen

$$1/b < mq < b \,. \tag{3.65}$$

Die Mantisse kann also betragsmäßig größer als 1 werden. Wenn dann die entstandene Übertragsstelle noch verfügbar ist, kann die Mantisse zum Normalisieren um eine Stelle nach rechts geschoben und der Exponent um Eins erhöht werden. Wenn die Übertragsstelle nicht verfügbar ist, d. h. wenn bei der Division der Mantissen nur Quotienten < 1 auftreten dürfen, dann muß vor der Division der Dividend der Mantisse um eine Stelle nach rechts geschoben werden. Dann liegt die Quotienten-Mantisse zwischen b^{-2} und 1, so daß die Mantisse ggf. durch einen Linksschift normalisiert werden muß. Die vorher herausgeschobene Stelle des Dividenden kann zur Erhöhung der Genauigkeit zum Normalisieren aufgehoben werden. Der folgende Algorithmus beschreibt die Division für normalisierte Gleitkommazahlen, wobei davon ausgegangen wurde, daß die Übertragsstelle gemerkt werden kann. In eckige Klammern eingeschlossene und durch Kommas getrennte Anweisungen können parallel ausgeführt werden.

[if $(x = 0).(y = 0)$ then „Undefiniert“ fi,
if $(x \neq 0).(y = 0)$ then „Unendlich groß“ fi,
if $(x = 0).(y \neq 0)$ then $q := 0$, „Null“ fi,

if $(x \neq 0).(y \neq 0)$ then $mq := mx/my$; „Übertragsstelle merken“
    if $1/b \le |mq| < 1$ „normalisiert“
    then $eq := ex - ey$
    else „$1 \le |mq| < b$“[$eq := ex - ey + 1, mq := mq/b$] fi;
    [if $eq > eq_{\max}$ then „Exponentenüberlauf“ fi,
    if $eq < eq_{\min}$ then „Exponentenunterlauf, evtl. $q := 0$“ fi]
fi]

Bei der Division können verschiedene Fehler auftreten:

1. Undefiniertes Ergebnis $(0/0)$
2. Division durch 0 $(x/0 = \infty)$
3. Quotient zu groß (Exponentenüberlauf)
4. Quotient zu klein (Exponentenunterlauf).

Die beiden ersten Fälle werden oft zu dem Fehler „Division durch Null“ zusammengefaßt. Manchmal werden auch die Fälle 1, 2 und 3 zu einer allgemeinen Überlaufbedingung zusammengefaßt, und bei Exponentenunterlauf (unechte Null) kann der Quotient gleich Null gesetzt werden. Auf solche Rundungen muß der Anwender gefaßt sein, so daß die numerischen Ergebnisse mit der entsprechenden Skepsis interpretiert werden müssen.

3.4 Konvertierung zwischen Zahlensystemen

In diesem Abschnitt werden Konvertierungen von ganzen Zahlen zur Basis a in Zahlen zur Basis b betrachtet. In Rechenanlagen sind hauptsächlich Konvertierungen zwischen Dualzahlen (interne Darstellung) und Dezimalzahlen (externe Darstellung oder interne Darstellung als BCD-Zahl) notwendig.

Gegeben sei eine m-stellige ganze Zahl zur Basis a (Darstellung im Quellsystem), die in eine n-stellige ganze Zahl zur Basis b (Darstellung im Zielsystem) konvertiert werden soll.

$$A = (A_m A_{m-1} \dots A_1)_a \quad \rightarrow \quad (B_n B_{n-1} \dots B_1)_b = B \tag{3.66}$$

Die beiden Zahlen sollen natürlich den gleichen Wert $(A = B)$ besitzen. Die Werte lassen sich als Potenzsumme oder durch das Horner-Schema berechnen:

$$\begin{aligned} A &= A_m * a^{m-1} + \dots + A_2 * a + A_1 && (3.67) \\ B &= B_n * b^{n-1} + \dots + B_2 * b + B_1 && (3.68) \\ A &= (\cdots (A_m * a + A_{m-1}) * a + \dots + A_2) * a + A_1 && (3.69) \\ B &= (\cdots (B_n * b + B_{n-1}) * b + \dots + B_2) * b + B_1 && (3.70) \end{aligned}$$

Die verschiedenen Konvertierungsverfahren lassen sich aus diesen Beziehungen durch Gleichsetzen herleiten.

3.4.1 Summationsmethode

Die Summationsmethoden ergibt sich durch Gleichsetzung von

$$\sum_{i=1}^{m} A_i * a^{i-1} \quad = \quad B \; .$$

Um B zu erhalten, müssen aus den Ziffern A_i (dargestellt im Quellsystem zur Basis a) die Summanden $A_i * a^{i-1}$ (dargestellt im Zielsystem zur Basis b) ermittelt werden und im Zielsystem aufaddiert werden. Die Summanden werden am einfachsten einer Tabelle (Abb. 3.45) entnommen: $\mathit{TAB}(A, i) = A_i * a^{i-1}$. Nach dieser Methode lassen sich Hexadezimalzahlen in Dezimalzahlen wie folgt umwandeln:

Hexadezimal-ziffer	Stellenwertigkeit					
	$*16^0$	$*16^1$	$*16^2$	$*16^3$	$*16^4$	$*16^5$
0	00	000	0000	00000	000000	00000000
1	01	016	0256	04096	065536	01048576
2	02	032	0512	08192	131072	02097152
3	03	048	0768	12288	196608	03145728
4	04	064	1024	16384	262144	04194304
5	05	080	1280	20480	327680	05242880
6	06	096	1536	24576	393216	06291456
7	07	112	1792	28672	458752	07340032
8	08	128	2048	32768	524288	08388608
9	09	144	2304	36864	589824	09437184
A	10	160	2560	40960	655360	10485760
B	11	176	2816	45056	720896	11534336
C	12	192	3072	49152	786432	12582912
D	13	208	3328	53248	851968	13631488
E	14	224	3584	57344	917504	14680064
F	15	240	3840	61440	983040	15728640
	Äquivalenter dezimaler Summand					

Abbildung 3.45: Konvertierungstabelle für Hexadezimalzahlen in Dezimalzahlen

1. Lies aus der Tabelle für jede Hexadezimalziffer (Zeile) mit ihrer zugehörigen Stellenwertigkeit (Spalte) den äquivalenten dezimalen Summanden.
2. Addiere alle Summanden dezimal. Die Summe ist die gesuchte Dezimalzahl.

Dieser Algorithmus eignet sich auch zur Umwandlung von Dualzahlen in BCD-Zahlen. Dazu werden die Dualzahlen durch Zusammenfassen von Gruppen zu je 4 Bits in Hexadezimalzahlen überführt und die Tabellenwerte als BCD-Zahlen codiert, die dann BCD-mäßig aufaddiert werden.

Zahlenbeispiel: $\mathrm{F18C}_{16}$

$$\begin{array}{rrr} = & \mathrm{F000} & 61440 \\ + & 100 & 256 \\ + & 80 & 128 \\ + & \mathrm{C} & 12 \\ \hline & & 61836_{10} \end{array}$$

Die Umwandlung von Dualzahlen in BCD-Zahlen kann auch mit Hilfe einer vereinfachten Tabelle erfolgen, in der die Potenzen von 2 in BCD-Darstellung enthalten sind. Der Algorithmus lautet dann

$B := 0;$
for $i := 1$ to m do if $A_i = \mathtt{1}$ then $B := B +_{\text{BCD}} (2^{i-1})_{\text{BCD}}$ od ,

Dezimal-ziffer	Stellenwertigkeit						
	$*10^0$	$*10^1$	$*10^2$	$*10^3$	$*10^4$	$*10^5$	$*10^6$
0	0	00	000	0000	00000	00000	000000
1	1	0A	064	03E8	02710	186A0	0F4240
2	2	14	0C8	07D0	04E20	30D40	1E8480
3	3	1E	12C	0BB8	07530	493E0	2DC6C0
4	4	28	190	0FA0	09C40	61A80	3D0900
5	5	32	1F4	1388	0C350	7A120	4C4B40
6	6	3C	258	1770	0EA60	927C0	5B8D80
7	7	46	2BC	1B58	11170	AAE60	6ACFC0
8	8	5D	320	1F40	13880	C3500	7A1200
9	9	5A	384	2328	15F90	DBBA0	895440
	Äquivalenter hexadazimaler Summand						

Abbildung 3.46: Konvertierungstabelle für Dezimalzahlen in Hexadezimalzahlen

wobei die BCD-Addition verfügbar sein muß.
Nach dem gleichen Prinzip lassen sich auch Dezimalzahlen in Hexadezimalzahlen wie folgt umwandeln.

1. Lies aus der Tabelle (Abb. 3.46) für jede Dezimalziffer (Zeile) mit ihrer Stellenwertigkeit (Spalte) den äquivalenten hexadezimalen Summanden.
2. Addiere alle Summanden hexadezimal. Die Summe ist die gesuchte Hexadezimalzahl.

Dieser Algorithmus eignet sich auch zur Umwandlung von BCD-Zahlen in Dualzahlen. Anstelle der hexadezimalen Addition kann die Addition von Dualzahlen benutzt werden, wenn jede Hexadezimalziffer wie üblich durch 4 Bits codiert wird.

Zahlenbeispiel: 61836_{10}

=	60000	EA60
+	1000	3E8
+	800	320
+	30	1E
+	6	6
		$F18C_{16}$ = 1111 0001 1000 1100

3.4.2 Divisionsmethode

Die Divisionsmethode ergibt sich durch Gleichsetzung von

$$A = (\cdots(B_n b + B_{n-1})b + \cdots + B_2)b + B_1 .$$

Bei der sukzessiven ganzzahligen Division durch b stellen die Reste die gesuchten Ziffern $B_1, B_2, \ldots, B_n$ dar. Der Algorithmus lautet

for j :=1 to n do
„1.“ $[(B_i)_a := \text{Rest}(A/(b)_a), ; A := A/(b)_a]$ „im Quellsystem“
„2.“ $(B_i)_b := (B_i)_a$ od „Ziffernumcodierung“

Dabei erfolgt die Division und die Restbildung im Quellsystem. Das Trennzeichen „`,;`“ zwischen den Operationen bedeutet, daß die Operationen nacheinander oder gleichzeitig (synchron) durchgeführt werden können. Im allgemeinen ist im zweiten Schritt einer Ziffernumcodierung notwendig, insbesondere wenn $b > a$ ist.

Beispiel: Dezimalzahl → Dualzahl

$$\left.\begin{array}{rcrcc} 196/2 & = & 98 & R & 0 \\ 98/2 & = & 49 & R & 0 \\ 49/2 & = & 24 & R & 1 \\ 24/2 & = & 12 & R & 0 \\ 12/2 & = & 6 & R & 0 \\ 6/2 & = & 3 & R & 0 \\ 3/2 & = & 1 & R & 1 \\ 1/2 & = & 0 & R & 1 \end{array}\right\} \texttt{lloo oloo}$$

Beispiel: Dualzahl → Dezimalzahl

$$\left.\begin{array}{rcr} \texttt{lloo oloo / lolo} & = & \texttt{lo oll R ollo} \\ \texttt{l ooll / lolo} & = & \texttt{l R lool} \\ \texttt{l / lolo} & = & \texttt{o R oool} \end{array}\right\} 196 \qquad (3.71)$$

Die Divisionsmethode wird in Rechenanlagen kaum verwendet, da sie die zeitaufwendige Division voraussetzt. Sie eignet sich hauptsächlich für die Umwandlung von Dezimalzahlen in andere Zahlensysteme durch den Menschen.

3.4.3 Multiplikationsmethode

Die Multiplikationsmethode ergibt sich durch Gleichsetzung von

$$(\cdots(A_m * a + A_{m-1}) * a + \cdots + A_2) * a + A_1 \quad = \quad B\,.$$

Daraus folgt der Algorithmus

$B := 0;$
for $i := m$ to 1 do
„1.“ $(A_i)_b := (A_i)_a$ „Ziffernumcodierung“
„2.“ $B := B * (a)_b + (A_i)_b$ „Rechnung im Zielsystem“
od

Im 1. Schritt ist im allgemeinen eine Ziffernumcodierung notwendig, insbesondere wenn $b < a$ ist. Die Multiplikation mit der Basis a und die Addition der Ziffer A_i wird dann im Zielsystem durchgeführt. Die gesuchte Zahl B ergibt sich als Ganzes nach Ausführung des Algorithmus.

Beispiel: Dezimalzahl 196 → Dualzahl

$$(\overbrace{\texttt{oool}}^{1} * \texttt{lolo} + \overbrace{\texttt{lool}}^{9}) * \texttt{lolo} + \overbrace{\texttt{ollo}}^{6} = \texttt{lloo oloo}$$

Beispiel: Dualzahl `lloo oloo` → Dezimalzahl

```
      l      l       o       o       o       l       o       o
(((((((1*2  + 1)*2  + 0)*2  + 0)*2  + 0)*2  + 1)*2  + 0)*2  + 0  = 196
```

Aus dem obigen allgemeinen Algorithmus ergibt sich speziell für die Umwandlung von Dualzahlen in BCD-Zahlen:

boole $A(m:1), B(4*n:1)$;
$B := 0$;
for $i := m$ to 1 do
$B := B + B + A_i$ od „+ BCD-Addition, $B + B = 2B$“ .

Und speziell für die Umwandlung von BCD-Zahlen in Dualzahlen ergibt sich folgender Algorithmus:

boole$A(4*m:1), B(n:1)$;
$B := 0$;
for $i := m$ to 1 do

$$B := \underbrace{(\text{shl}B) + (3\text{shl}B)}_{= B * \texttt{lolo}} + \overbrace{A\langle i\rangle}^{A(4*i:1+4*(i-1))}$$ od „Dualaddition“

Die Multiplikation mit $10 = 2 + 8$ läßt sich dabei durch zwei Linksschifts implementieren.

6 Rechnerentwurf

6.1 Einige Entwurfsaspekte

Anwender und Entwerfer eines Rechnersystems. Der *Anwender* benutzt Anwendungsprogramme, von denen er bestimmte Leistungsmerkmale erwartet. Dazu zählen insbesondere (1) eine angenehme *Bedienbarkeit* unter einer modernen Bedienoberfläche, (2) eine angemessene *Funktionalität* (nicht zu viele und nicht zu wenige Operationen, die aus „erzeugenden" Operationen durch Kombination gebildet oder maßgeschneidert werden können) und (3) eine geringe *Bearbeitungszeit* (Laufzeit, Reaktionszeit). Der reine Anwender ist weder an der Software-Implementierung noch an der Hardware interessiert, solange seine Bedürfnisse durch das Hardware/Software-System erfüllt werden. Es ist klar, daß bestimmte Leistungsmerkmale sich nur auf einer qualitativ hochwertigen Hardware-Plattform und einem geeigneten Betriebssystem realisieren lassen.

Der *Anwendungsprogrammierer* stellt entsprechende Anforderungen an die Programmentwicklungsumgebung. Sie muß so leistungsfähig sein, daß er sein Programm möglichst schnell entwickeln, dokumentieren und ändern kann. Hier muß eine gute Wahl der Programmiersprache, der Compiler und sonstigen Tools getroffen werden, wobei auch die Frage nach dem Betriebssystems und der eigentlichen Maschine beantwortet werden muß. Der Anwendungsprogrammierer weiß, daß bestimmte Maschinenmerkmale (wie Adreßraum, Graphikansteuerung) bei der Programmierung berücksichtigt werden müssen bzw. bis in die Anwendung durchschlagen. Insofern ist der Anwendungsprogrammierer teilweise gezwungen, sich um Maschinendetails zu kümmern, mit denen sich ansonsten nur der Systemprogrammierer auseinandersetzt. Moderne Compiler und Betriebssysteme stellen aber dem Anwendungsprogrammierer ausreichend leistungsfähige Kommandos und Bibliotheken zur Verfügung, so daß er die Maschine nicht direkt ansteuern muß.

Der *Systemprogrammierer* entwickelt Systemprogramme (Betriebssysteme, Compiler, Debugger, Editoren, Laufzeitsysteme). Er benutzt maschinennahe Sprachen wie „C" und die Assembler-Maschinensprache. Er muß sich mit den Feinheiten des Betriebssystems und den

besonderen Hardware-Eigenschaften auf der Maschinenebene auskennen.

Der *Entwerfer des Rechnersystems* (Rechnerarchitekt im weiteren Sinne) legt die grobe Gliederung des Rechners (die *Rechnerstruktur*) in Prozessoren, Bussysteme, Hauptspeicher, Cachespeicher, Plattenspeicher, Grafikinterface und sonstige Schnittstellen fest. Dabei spielen Fragen der Zuverlässigkeit, des Durchsatzes, der Zugriffszeiten, der Bandbreiten, der maximalen Kapazitäten, der Erweiterbarkeit usw. eine besondere Rolle.

Der *Entwerfer der Rechnerarchitektur* (Rechnerarchitekt im engeren Sinne) legt den Maschinenbefehlssatz und die durch ihn angesprochenen Objekte (wie Register, Speicher, Ein-/Ausgabe-Schnittstelle, Interruptsystem) fest. Weiterhin setzt er die grobe Gliederung des Rechnersystems in eine feine um. Auf dieser Ebene können Hardware-Beschreibungssprachen eingesetzt werden, um die Funktionalität und die „Register-Transfer“-Architektur beschreiben zu können.

Der *Logik-Entwerfer* entwickelt mit Hilfe der Design-Tools aus diesen Beschreibungen die entsprechenden integrierten Schaltkreise. Dabei kommen mehr und mehr Syntheseprogramme zum Einsatz, die aus einer algorithmischen oder Register-Transferbeschreibung eine Logik-Beschreibung generieren.

Der *Rechnerbauer* schließlich fügt die Komponenten entsprechend der definierten Rechnerstruktur zu einem Ganzen zusammen.

Formulierung der Entwurfsaufgabe. Bevor ein System oder eine Funktionseinheit implementiert bzw. realisiert wird, sollte zunächst ein *Plan* (Lösungskonzept) aufgestellt werden, der den Implementierungsweg aufzeigt (Strukturdiagramme, Blockschaltbilder, Graphen, Ablaufpläne, Relationen, Datenorganisation und dergl.). Um nicht unnötige Arbeit in einen Plan zu verschwenden, der vielleicht der Aufgabe nicht gerecht wird, muß die Aufgabenstellung möglichst vollständig formuliert werden. Eine Software-Implementierungsaufgabe könnte z. B. nach folgendem (vereinfachendem) Schema aufgestellt werden:

Gesucht:

a) *Implementierung eines Programms, das die Nebenbedingungen erfüllt.*

b) *Benutzerhandbuch*

c) *Dokumentation über die Planung und Implementierung*

Forderungen/Nebenbedingungen:

a) Erfüllung der expliziten Forderungen des Anwenders (gegeben)

b) Erfüllung der impliziten Forderungen des Anwenders (zu erfragen oder zu erraten)

c) Die Laufzeit für bestimmte Funktionen muß unterhalb einer bestimmten Schranke liegen (gegeben oder vorzugeben).

d) Es soll die Entwicklungsumgebung für die Programmiersprache X auf dem Rechner Y mit dem Betriebssystem Z verwendet werden (gegeben).

e) Das Produkt soll möglichst gut wartbar sein (präzisieren).

f) Die Kosten sollen kleiner als k sein (Kosten präzisieren: Entwicklungs-, Herstellungs-, Wartungs-, Betriebs-, Änderungskosten).

Häufig werden Software-Implementierungsaufgaben nicht vollständig genug formuliert, so daß der Entwerfer/Implementierer z. B. die Erstellung der Dokumentation „vergißt". Oft sind auch die Nebenbedingungen unvollständig definiert, so daß u. U. Produkte entwickelt werden, die keine akzeptable Qualität besitzen. Deshalb sollten die Forderungen/Nebenbedingungen in Abstimmung mit dem Auftraggeber möglichst präzise formuliert werden, damit der Implementierer nicht falsche Wege einschlägt.

Eine typische *Hardware-Entwicklungsaufgabe* könnte etwa wie folgt lauten:

Gesucht:

a) Integrierte Schaltung

b) Datenblatt

c) Entwurfsunterlagen

Forderungen/Nebenbedingungen:

a) Applikationen für den Einsatz der Integrierten Schaltung in Umgebungen

b) Funktionsweise

c) Zeitverhalten

d) Anzahl der Anschlüsse

e) Stromverbrauch

f) Kosten

g) Die Logikkomponenten und erlaubten Zusammenschaltungen

h) Entwicklungswerkzeuge.

Auch bei dieser Entwurfsaufgabe müssen die Forderungen/Nebenbedingungen in der Regel präzisiert (geschätzt, ermittelt oder iterativ verbessert) werden, bevor mit der eigentlichen Entwurfsarbeit begonnen werden sollte.

Man sollte zwischen „Entwurfsaufgaben“ und „Optimierungsaufgaben“ unterscheiden. Eine Optimierungsaufgabe ist mehr als eine Entwurfsaufgabe, weil bei ihr zusätzlich ein Optimierungskriterium (Zielfunktion) ein Optimum annehmen soll. Deshalb kann man sich bei einer Optimierungsaufgabe nicht mit *einer* möglichen Lösung zufrieden geben, sondern muß *mehrere* Entwürfe machen, um sich daraus den besten aussuchen zu können.

Die am häufigsten benutzten, gegenläufigen Optimierungskriterien sind der Aufwand oder die Bearbeitungszeit:

A) Aufwand $\rightarrow$ *Minimum, Bearbeitungszeit* $<$ *Schranke*
B) Aufwand $<$ *Schranke, Bearbeitungszeit* $\rightarrow$ *Minimum.*

Vorgehensweisen beim Rechnerentwurf. Der Entwurf eines Rechners findet auf verschiedenen Ebenen statt und besitzt eine große Zahl von Freiheitsgraden, insbesondere auf den höheren Ebenen. Er erfordert vom Entwerfer ein hohes Maß an Erfahrung und Kreativität.

Dabei müssen im Hinblick auf die Funktionsweise die Vorgaben von „oben“ (Spezifikation) und die Vorgaben von „unten“ (Bausteine, Objekte) berücksichtigt werden:

Spezifikation $\rightarrow$ *Gestaltungsraum* $\rightarrow$ *Bausteine* .

Bei der *Top-Down-Methode* geht man schrittweise von oben nach unten vor, indem man den Entwurf immer mehr in Richtung auf die vorgegebenen Komponenten verfeinert. Bei der *Bottom-Up-Methode* geht man schrittweise von unten nach oben vor, indem man die vorgegebenen Komponenten zu komplexeren Komponenten in Richtung auf das Gesamtsystem zusammensetzt. Ein guter Entwerfer kann sowohl top-down als auch bottom-up vorgehen, wenn er die Vorgaben auf der anderen Seite immer im Auge behält. Die Top-Down-Methode hat den Vorteil, daß man in den ersten Verfeinerungsschritten künstliche (abstrakte, virtuelle) Komponenten benutzen kann, die man erst später auf die konkreten Komponenten abbildet. Dabei kann sich allerdings herausstellen, daß die Abbildung zu ineffizient wird. Die Bottom-Up-Methode hat den Vorteil, daß die Komponenten schrittweise komplexer werden und konkret bleiben, so daß man sie sich leichter vorstellen und auch testen kann. Dabei kann es allerdings vorkommen, daß man komplexe Komponenten konstruiert, die sich schlecht zum Zusammenbau des Gesamtsystems eignen.

Um zu einem guten Entwurf zu kommen, sollte man deshalb mehrere Entwürfe nach verschie-

denen Methoden machen und anschließend Optimierungen vornehmen.

Man kann auch eine gemischte Top-Down/Bottom-Up-Entwurfmethode anwenden, bei der sich Bottom-Up-Schritte und Top-Down-Schritte abwechseln, so daß sie sich schließlich in der Mitte treffen.

Zum Entwurf eines Steueroperationssystems. Wir wollen als Beispiel den Entwurfsvorgang für ein Steueroperationssystem betrachten, bei dem eine bestimmte Funktionsweise realisiert werden soll.

Zuerst wird die Funktionsweise spezifiziert, wobei beliebige Formalismen benutzt werden können. So können beispielsweise logische Aussagen oder parallele Algorithmen mit beliebigen Objekten und Operatoren benutzt werden. Diese Spezifikation wird dann von dem Entwerfer oder einem automatischen Entwurfssystem weiter verarbeitet. Der gewonnene Entwurf muß anschließend gegen die Spezifikation geprüft (verifiziert) werden.

Wir wollen vereinfachend annehmen, daß das System zunächst in ein Operationswerk und ein Steuerwerk aufgespalten werden soll. Dabei gibt es zwei Möglichkeiten: entweder man beginnt mit der Festlegung des Operationswerks und legt anschließend das Steuerwerk fest oder umgekehrt. Dabei können die beiden Werke nicht unabhängig voneinander entworfen werden. Änderungen des Operationswerks ziehen Änderungen des Steuerwerks – und sei es nur eine Programmänderung – nach sich und umgekehrt. In der Regel ist es günstiger, mit dem Entwurf des Operationswerks zu beginnen und danach das Steuerwerk zu entwerfen. Denn die Funktionsweise des Steuerwerks und damit des gesamten Systems läßt sich ja durch einen anderen Ablauf, der in einem Mikroprogramm, einem PLA oder in der Verdrahtung steckt, verhältnismäßig leicht verändern. Beim Entwurf muß man die Spezifikation im Auge behalten bzw. aus ihr direkt die Mikrooperationen für das Operationswerk ableiten. Das ist z. B. möglich, wenn die Spezifikation algorithmisch in einer Hardware-Beschreibungssprache formuliert wurde. Um einen guten Entwurf zu erhalten, wird man abwechselnd Änderungen im Operationswerk und im Steuerwerk vornehmen müssen. Wenn das System die gewünschte Funktionsweise realisiert, müssen die Nebenbedingungen geprüft werden. Arbeitet das System zu langsam, dann müssen das Steuerwerk und/oder das Operationswerk aufwendiger („breiter“) gestaltet werden. Im Steuerwerk und/oder im Operationswerk müssen dann mehr Abfragen bzw. Mikrooperationen parallel durchgeführt werden. Ist dagegen das entworfene System zu teuer, dann müssen das Steuerwerk und/oder das Operationswerk vereinfacht werden, indem die Abfragen und/oder Mikrooperationen sequentialisiert werden. Man versucht dann, die Datenwege zu minimieren (Mehrfachnutzung von Datenwegen/Bussen) und teure Schaltungen (wie Addierer, Multiplizierer etc.) nur sparsam einzusetzen und mehrfach zu nutzen. Die Freiheitsgrade beim Entwurfsvorgang sind mehr oder weniger beschränkt, insbesondere durch die vorgegebene Bausteinbibliothek und vorgebenene Grundstrukturen. (Das Steuerwerk soll z. B. ein Matrix-Steuerwerk oder ein Mikroprogrammsteuerwerk sein; das Rechenwerk soll n Register und ein oder zwei ALUs ohne Multiplizierer besitzen.)

Wenn ein universelles Rechenwerk entworfen werden soll, dann sollte man versuchen, eine möglichst einfache, klare Registerstruktur zu wählen, weil dadurch die Übersichtlichkeit, Überprüfbarkeit und Ansteuerung sehr vereinfacht werden. Wenn das Operationswerk klar strukturiert ist, dann läßt es sich auch durch klar strukturierte Mikrobefehle/Befehle ansteuern. Insofern hängen diese Strukturen gegenseitig voneinander ab.

Zum Entwurf eines Mikroprogramm-Steuerwerks. Beim Entwurf eines MP-Steuerwerks ist es zweckmäßig, zuerst die Größe des MP-Speichers festzulegen. Die Anzahl der Worte läßt sich durch eine Abschätzung der maximalen Länge des Mikroprogramms festlegen. Die Breite des Mikrobefehls ist schwieriger festzulegen, da sie von den parallel auszuführenden Mikrooperationen, ihrer Codierung und Anordnung in Unterfeldern des Mikrobefehls abhängt.

Definition der Mikrobefehle. Bei der Definition der Mikrobefehle im Entwurfsvorgang gibt es zwei Vorgehensweisen: (1) Vom horizontalen zum vertikalen Mikrobefehl und (2) vom vertikalen zum horizontalen Mikrobefehl.

Zu (1): Der Ausgangspunkt ist ein „reines“ MP-Steuerwerk mit einem horizontal codierten Mikrobefehl. Alle Mikrooperationen und Abfragen sind parallel ausführbar. Um den Aufwand des Steuerwerks zu reduzieren, werden anschließend die verschiedenen Methoden zur Einsparung von Speicherplatz angewendet. Diese Vorgehensweise hat den Vorteil, daß man mit einer klaren, übersichtlichen Struktur und mit minimaler Verarbeitungszeit beginnt.

Zu (2): Der Ausgangspunkt ist der vertikale Mikrobefehl. Die Mikrooperationen im Operationswerk und die Abfragen sind nur sequentiell durchführbar. Um die Verarbeitungsgeschwindigkeit zu erhöhen, werden schrittweise mehr Mikrooperationen und Abfragen parallel zugelassen. Diese Vorgehensweise hat den Vorteil, daß man mit geringem Aufwand beginnt. Bei der Parallelisierung verschafft man sich über die Häufigkeit und Verträglichkeit der kombinierten Mikrooperationen Klarheit, so daß keine Mikrooperationen/Mikrobefehle realisiert und codiert werden, die nur selten benutzt werden oder sich nicht vertragen.

Neben den zwei oben geschilderten Vorgehensweisen besteht auch die Möglichkeit, von einem Mikrobefehl auszugehen, der etwa doppelt so breit wie die Datenwegbreite ist, weil dann in einer Hälfte des Mikrobefehls eine Konstante (Direktoperand) untergebracht werden kann, die direkt auf den Datenweg geschaltet werden kann. Wählt man einen zu breiten Mikrobefehl, so wird die verfügbare Parallelität im Mittel nur schlecht ausgenutzt; wählt man ihn zu schmal, dann benötigt man zu viele Schritte zur Durchführung der gewünschten Operationen. Man muß auch beachten, daß ein „breites“ Operationswerk auch einen breiten Steuerbefehl erfordert und umgekehrt. Je billiger die Speicher werden, desto mehr kann man auf die starke Codierung des Mikrobefehls verzichten, wodurch auch die sonst notwendigen Decodierer und Demultiplexer entfallen. Das Multiplexen der Eingangssignale wird man aber nicht umgehen können (außer bei der Verwendung von PLAs), denn der Speicheraufwand wächst ja exponentiell mit der

Anzahl der Eingangssignale.

Zum Entwurf der Maschinenbefehle. Die Maschinenbefehle enthalten verschiedene Informationsanteile, die *Programmadreßinformation*, die *Datenadreßinformation*, die *Recheninformation* und die *Steuerinformation*. Die Programmadreßinformation, die vom Programmadreßrechenwerk ausgewertet wird, dient zur Berechnung der Adresse des nächsten auszuführenden Befehls. Bei normalen Befehlen wird diese Adresse im Befehlszähler gehalten und einfach um Eins hochgezählt, so daß diese Information im Befehlscode enthalten ist. Wenn Sprünge ausgeführt werden sollen, muß sich aus der Programmadreßinformation das Sprungziel berechnen lassen. Die Programmadreßinformation beinhaltet Sprungadreßkonstanten, Adressen von Programmadreßregistern und die Programmadressierungsart. Je nach Adressierungsart werden die Inhalte der Programmadreßregister (dazu zählen der Programmzähler und das Programmbasisregister) mit den Sprungadreßkonstanten (dazu zählen absolute oder relative Sprungadressen) unterschiedlich verknüpft. Bei bedingten Sprüngen hängt das Sprungziel zusätzlich von dem aktuellen Wert der ausgewählten Bedingung ab, die auch zu der Programmadreßinformation gezählt werden kann.

Die *Datenadreßinformation* beinhaltet Adressen von Datenadreßregistern (z. B. Indexregister, Datenbasisregister), Datenadreßkonstanten (z. B. Offsets) und die Datenadressierungsart. Die *Recheninformation* beinhaltet Adressen von Rechenregistern, Rechenkonstanten und den Rechenoperationscode. Die *Steuerinformation* beinhaltet den Operationscode des Befehls und weitere Steuerdaten zur Steuerung der sonstigen Einheiten des Systems.

An dieser Aufzählung ist klar geworden, daß alle diese Informationsanteile nicht parallel im Befehl untergebracht werden können. Es müssen also zwangsläufig Befehlstypen eingeführt werden, die bestimmte „Informationsmischungen“ charakterisieren. Typische Einteilungen bestehen aus unbedingten und bedingten Sprungbefehlen, reinen Rechenbefehlen und LOAD/STORE-Befehlen. Bei den sogenannten CISC(complex instruction set)-Befehlen gibt es auch Befehle mit variabler Wortlänge und direkten Operationen auf Speicherzellen, während die sogenannten RISC(reduced instruction set)-Befehle konstante Länge besitzen und keine Operationen auf den Speicherzellen zulassen, d. h. die Operanden müssen explizit durch LOAD/STORE-Befehle zwischen den Speicherzellen und den Registern hin und her transportiert werden.

Die Frage nach der Einteilung der Befehle in Befehlstypen und Felder ist ein Optimierungsproblem, das zusammen mit der Gestaltung des Operationswerkes erfolgen muß. Dabei sollten u. a. folgende Punkte berücksichtigt werden:

1. *Anzahl der Register.* Sie sollte einerseits möglichst klein sein, damit Adressierungsbits eingespart werden können und der Maschinenstatus nicht zu groß wird (beim Prozeßwechsel), andererseits sollte sie möglichst groß sein, damit möglichst viele Zwischenergebnisse und Parameter gehalten werden können, wodurch sich die Anzahl der LOAD/STORE-Befehle reduziert. Meist wird eine Registerzahl von 16 oder 32 gewählt.

2. *Anzahl der Registersätze und Operationen.* Aus Gründen der logischen Übersichtlichkeit, des Schutzes, der Erweiterbarkeit und der Unterstützung von Parallelarbeit sollte man für jeden Datentyp (spezifische Operationen auf Registern) ein separates Rechenwerk vorsehen (vergl. Abschn. 4.4). Die Rechenwerke können dann getrennt entworfen und optimiert werden, wobei die Registersätze als einfache 2-Port-RAMs konstruiert werden können. Dabei entsteht aber das Problem der zeitlichen Synchronisation (die Operationen dauern unterschiedlich lange, z. B. durch Pipelining, Ausnahmebehandlung) und des Datenaustauschs (einschließlich der Datentypkonvertierung) zwischen Registersätzen, der dann über den Speicher erfolgen muß.

   Wenn man den Hardware-Aufwand minimieren will, kann man die extreme andere Alternative wählen, nämlich ein Rechenwerk für alle Datentypen. Um die steigenden Anforderungen nach Parallelarbeit zu erfüllen, muß man hardwaremäßig aber einen aufwendigen Multiport-Registerspeicher vorsehen, an den die ALUs für die verschiedenen Datentypen angeschlossen sind, z. B. ALU1 und ALU2 für logische und Integer-Operationen, ALU3 für Gleitkommaoperationen, ALU4 für graphische Operationen (Pixel Add usw.), ALU5 Programmadreßberechnungen, ALU6 für Datenadreßberechnungen. In der Praxis wählt man meist einen Mittelweg zwischen den beiden Extremen (n Datentypen $= n$ Rechenwerke) und (n Datentypen $= 1$ Rechenwerk).

3. *Länge der Adresse.* Die Länge der (meist virtuellen) Adresse sollte für die Applikationen ausreichend groß sein, so daß der Programmierer nicht gezwungen ist, sein Programme und Daten künstlich zu segmentieren. Eine Adreßlänge von 32 reicht meist aus, für größere Mehrprogramm- und Multiprozessorsysteme wird man aber auf Adreßlängen von 48 oder 64 übergehen müssen.

4. *Feste/variable Befehlswortlänge.* Eine feste Befehlswortlänge hat den Vorteil der einfacheren hardwaremäßigen Decodierung und schnelleren Ausführbarkeit. Bei variabler Befehlswortlänge können die häufigen Befehle kürzer codiert werden, die dann weniger Speicherplatz benötigen und den Speicherbus weniger belasten. Befehle, die mehr Information tragen sollen, z. B. Befehle mit mehreren Speicheradressen oder langen Konstanten, können sich über mehrere Worte erstrecken. Dadurch wird natürlich der Aufwand zum Holen des Befehls größer, kann aber kleiner gehalten werden, als wenn dafür eine wirkungsgleiche Befehlssequenz benutzt wird. Außerdem erhöht sich der Decodieraufwand, das Steuerwerk wird komplexer und Hardware-Beschleunigungsmaßnahmen wie Pipelining lassen sich schwieriger realisieren. Dafür hat der Rechnerarchitekt mehr Freiheit bei der Definition von softwareorientierten Maschinensprachen.

5. *Anzahl der Befehlstypen.* Wenn die Anzahl der Befehlstypen und Adressierungsarten klein ist, dann ergeben sich folgende Vorteile: (1) Der Maschinenprogrammierer/Übersetzerbauer kann den Maschinenbefehlssatz leicht überblicken, (2) Der Operationscode wird kurz und die Decodierung wird einfach, (3) Hardware-Steuerwerke können benutzt werden, (4) Hardware-Beschleunigungsmaßnahmen lassen sich leichter realisieren und (5) Befehle mit einer geringen Häufigkeit belasten nicht die Architektur und die

Realisierung. Es sprechen aber auch Gründe für eine große Anzahl von Befehlen. Wenn eine größere Funktionalität und Leistung bereit gestellt werden soll, z.B. wenn Befehle und Rechenwerke für höhere Datentypen und für Parallelarbeit von der Hardware unterstützt werden sollen, dann muß die Anzahl der Befehle zwangsläufig größer werden. Das ist ein Grund dafür, weshalb sich der Wunsch nach sehr wenigen Befehlen nicht durchhalten läßt. In der Tat findet man bei modernen sogenannten RISC-Rechnern eine sehr große Anzahl von Befehlen, die z. B. Gleitkommaverarbeitung und Graphikverarbeitung unterstützen. Allerdings sollten zu der Befehlsliste nicht solche Befehle hinzugefügt werden, die leicht nachgebildet werden können oder eine sehr geringen Einfluß auf die Verbesserung der Gesamtleistung haben. Dazu zählen insbesondere Befehle mit einem geringen Produkt aus *Häufigkeit* mal *Befehlsausführungszeit = Zeitanteil*. (Die alleinige Betrachtung der Häufigkeit, wie oft argumentiert wird, genügt nicht!)

Aus diesen Betrachtungen sollte klar geworden sein, daß es statistischer Untersuchungen, einer großen Erfahrung und einer gewissen Kreativität bedarf, um die Rechnerarchitektur, d. h. insbesondere die Maschinenbefehle zusammen mit den Operationswerk, so zu gestalten, daß ein übersichtliches, programmierfreundliches und leistungsfähiges System mit akzeptablem Aufwand entsteht.

Charakterisierung von Entwurfsaufgaben. Im folgenden wollen wir einige typische Entwurfsaufgaben charakterisieren, wobei die Betrachtungen sich auf die Erfüllung der Forderung „Funktionsweise“ beschränken sollen. Bevor ein System entworfen werden kann, muß die gewünschte Funktionsweise bzw. die Menge der Funktionsweisen (im Falle der Progammierbarkeit z. B. durch Angabe der Befehlsliste und Objektmenge) festgelegt (spezifiziert) werden. Eine große Schwierigkeit besteht darin, die Funktionsweise(n) exakt zu spezifizieren, und zwar hauptsächlich aus zwei Gründen: (1) Der Anwender oder Entwerfer weiß zu Beginn oft selbst nicht genau, was er am Ende benötigt. Er kann zu Beginn nur die ihm am wichtigsten erscheinenden Teilfunktionen/Operationen spezifizieren. Aber er ist nicht in der Lage, insbesondere bei komplexen Systemen, das Zusammenspiel und die Auswirkungen der Teilfunktionen zu überblicken. Erst im Laufe der Entwicklung, nachdem gewisse Implementierungsentscheidungen getroffen wurden, gelingt es ihm, seine Anforderungen zu präzisieren, die normalerweise zu einer Revision der schon getroffenen Entscheidungen führen. (2) Die Werkzeuge zur Spezifikation und zur Entwurfsunterstützung werden vom Anwender oft als zu allgemein, zu speziell oder zu formal empfunden. Meist müssen auch verschiedenartige Werkzeuge benutzt werden, die nicht aufeinander abgestimmt sind. Allerdings werden diese Argumente manchmal nur als Vorwand benutzt, um sich nicht der Mühe des Erlernes neuer Methoden und Sprachen unterziehen zu müssen.

Im folgenden werden einige typische Entwurfsaufgaben beschrieben, wobei die Systeme vereinfachend als strenge Interpretationshierarchien (vergl. Abschnitt 5.4) mit konstanten und variablen Komponenten modelliert werden. *Variable* (gesuchte) Komponenten werden durch einen Stern gekennzeichnet, während *konstante* (gegebene) Teile nicht weiter gekennzeichnet werden.

1. **Entwurf einer speziellen Hardware-Funktionseinheit**

   $[Funktionsweise]$ $Verdrahtung^{*} \rightarrow Hardwarekomponenten$

   Gesucht ist eine Verdrahtung von Hardwarekomponenten, so daß die geforderte Funktionsweise realisiert wird. (In der Verdrahtung soll auch die Steuerung der Hardwarekomponenten stecken, deshalb könnte man anstelle von „Verdrahtung“ auch allgemeiner „Steuerung und Verdrahtung“ sagen.)

2. **Festlegung einer Funktionsweise durch ein Programm** (Implementierung einer Maschine)

   $[Funktionsweise]$ $Programm^{*} \rightarrow Prozessor$

   Gesucht ist ein Programm auf einem Prozessor, so daß die geforderte Funktionsweise realisiert wird.

   (Das Programm ist in der Regel ein Maschinenprogramm eines Rechners. Es kann aber auch ein Mikroprogramm einer Mikromaschine sein oder ein Programm in einer höheren Sprache, die vom Prozessor ausgeführt/interpretiert wird.)

   Anwendungsbeispiele: Programmierte Rechner/Prozessoren, die damit eine bestimmte Funktionsweise/Anwendung realisieren; Anwendungsprogramme, spezielle Steuerungen, Rechner mit Betriebssystem, programmierte Automaten.

3. **Entwurf eines „festverdrahteten“ Rechners**

   $Maschinenprogramm^{**} \rightarrow [Maschinenbefehle]$ $Verdrahtung^{*}$
   $\rightarrow Hardwarekomponenten$

   Gesucht ist ein programmierbarer Rechner (*Verdrahtung* $\rightarrow$ *Hardwarekomponenten*), der einen Maschinenprogrammspeicher besitzt und den gegebenen Maschinenbefehlssatz ausführen kann. (Die beiden Sterne kennzeichnen Komponenten, die variabel bleiben sollen.)

   Anwendungsbeispiele: Schnelle universelle und spezielle Rechner/Prozessoren.

4. **Entwurf eines mikroprogrammierbaren Rechners**

   $Maschinenprogramm^{**} \rightarrow Mikroprogramm^{**}$
   $\rightarrow [Mikrobefehle^{*}]$ $Mikromaschine^{*}$

   Gesucht sind Mikrobefehle und eine *Mikromaschine* $=$ (*Verdrahtung* $\rightarrow$ *Hardwarekomponenten*), so daß Mikroprogramme ausgeführt werden können, die Maschinenprogramme interpretieren können. D. h. die Mikromaschine muß einen Mikropogrammspeicher und eine Maschinenprogrammspeicher besitzen.

   Anwendungsbeispiel: Entwurf eines mikroprogrammierbaren Rechners für spezielle oder universelle Anwendungen.

5. **Emulation, Festlegung einer Rechnerarchitektur durch Mikroprogramme**

   $Maschinenprogramm^{**} \rightarrow [Maschinenbefehle]$ $Mikroprogramm^{*}$
   $\rightarrow$ *mikroprogrammierbarer Rechner*

Gesucht ist ein Interpreter-Mikroprogramm, das einen gegebenen Maschinenbefehlssatz interpretiert.

Anwendungsbeispiele: Realisierung von Rechnerarchitekturen innerhalb der durch die Hardware vorgegebenen Grenzen; Realisierung von Interpretern für Zwischensprachen.

6. **Entwurf eines mikroprogrammierten Rechners für eine gegebene Architektur**

   Maschinenprogramm** $\rightarrow$ [*Maschinenbefehle*] *Mikroprogramm**
   $\rightarrow$ *mikroprogrammierbarer Rechner**

   Gesucht ist ein mikroprogrammierbarer Rechner und ein Interpreter-Mikroprogramm, das einen gegebenen Maschinenbefehlssatz interpretiert.

   Anwendungsbeispiele: Realisierung kompatibler Rechner mit gleicher Architektur (Maschinenbefehlssatz) bei unterschiedlicher Hardware, Technologie und Verarbeitungsleistung, wodurch bereits entwickelte Systemprogramme weiter verwendet werden können; Realisierung neuartiger Rechnerarchitekturen und Interpreter für Zwischensprachen, die eine spezielle Hardware erfordern.

Entwurfsphasen für einen mikroprogrammierbaren Rechner. Der Entwurf eines mikroprogrammierbaren Rechners kann etwa in folgende Phasen eingeteilt werden:

A Implementierung der Architektur auf dem mikroprogrammierbaren Rechner

- A1 Architektur (Maschinenbefehlssatz und Objektmenge, Programmiermodell) festlegen
- A2 Interpreter-Mikroprogramm schreiben
- A3 Mikroprogramm überprüfen
- A4 Mikroprogramm laden
- A5 Rechner testen

B Entwurf und Realisierung des mikroprogrammierbaren Rechners

- B1 Mikromaschinenarchitektur festlegen
- B2 Hardwareentwurf mit den gegebenen Hardwarekomponenten durchführen
- B3 Hardwareentwurf überprüfen
- B4 Hardware aufbauen
- B5 Mikroprogrammierbaren Rechner testen

Einige Phasen können gleichzeitig, andere müssen hintereinander durchgeführt werden. Die A-Phasen müssen ebenso wie die B-Phasen hintereinander durchgeführt werden. Die Durchführung der Phase A2 setzt die Durchführung der Phase B1 voraus und die Durchführung der Phase A4 die Phase B5. Ansonsten können die Phasen gleichzeitig durchgeführt werden. Insbesondere ist die Reihenfolge der Definitionsphasen A1, B1 nicht festgelegt. Es muß

nur gewährleistet sein, daß sich die Architektur auf der Mikromaschine realisieren läßt. Die folgende Beziehung stellt die Abhängigkeiten zusammenfassend dar, wobei gleichzeitige Durchführbarkeit durch „[... , ...]“ und sequentielle Durchführung durch „ ; “ gekennzeichnet ist:

[A1, B1]; [(A2; A3), (B2; B3; B4; B5)]; A4; A5

Die Fälle A3, A5, B3, B5 dienen zur Verifikation der Spezifikationen A1, B1. Falls die Spezifikationen nicht verifiziert werden können, müssen vorausgegangene Entwurfsentscheidungen revidiert werden.

Rechnertypen. Je nachdem, in welchem Umfang Rechner programmierbar sind, können vier Rechnertypen unterschieden werden:

1. Rechner mit *konstanter* Maschinensprache
2. Rechner mit *variabler* (programmierbarer) Maschinensprache. (Die Festlegung erfolgt z. B. durch das Mikroprogramm eines mikroprogrammierbaren Rechners oder durch den PLA-Code eines „verdrahteten“ Rechners.)
3. Rechner mit *variablen* (programmierbaren) Mikrooperationen
4. Rechner mit *variabler* Struktur.

Im Laufe der Rechner-Historie wurden bisher die ersten beiden Rechnertypen entwickelt. In Zukunft könnte die Programmierbarkeit weiter ausgedehnt werden, so daß dadurch die neuartigen Rechnertypen 3. und 4. entstehen würden. Den 3. Rechnertyp möchte ich als *adaptierbaren* Rechner bezeichnen, weil er in bezug auf die Operationen besser an die Anwendung angepaßt werden kann (z. B. durch Programmierung der ALU-Funktionen). Voraussetzungen sind entsprechende Technologien (programmierbare Bausteine) und Übersetzungstechniken. Es ist zu vermuten, daß die Bedeutung adaptierbarer Rechner erkannt und mit dem Voranschreiten der Technologie ermöglicht wird.

Es ist noch ein weiterer, flexibler 4. Rechnertyp vorstellbar, bei dem nicht nur die internen Operationen, sondern auch die Speicherstruktur und die Verbindungswege programmierbar sind. Solche Rechner sollen *konfigurierbare* Rechner heißen.

6.2 Entwurf eines Beispiel-Rechners

In den folgenden Abschnitten wird der Beipiel-Rechner DINATOS entworfen. Zuerst wird im nächsten Abschnitt die Architektur auf der Verhaltensebene definiert. Anschließend werden verschiedene Implementierungen diskutiert. 8Im Abschnitt 6.2.2 werden direkte

Hardware-Realisierungen (das Steuerwerk ist ein Hardware-Steuerwerk) besprochen. Zum Vergleich soll der Beispiel-Rechner auf einem mikroprogrammierbaren Rechner durch Emulation/Interpretation implementiert werden. Dazu wird vorab der mikroprogramierbare Rechner PIRI entworfen, der dann zur Emulation der Architektur benutzt wird.

6.2.1 DINATOS-Architektur

Die Architektur eines Rechners wird üblicherweise durch die Wirkung der Maschinenbefehle auf die Objekte (Speicherzellen, Register, Datenträger, Ein-/Ausgangssignale, Teilfunktionseinheiten) beschrieben. Diejenigen Objekte, die zur Beschreibung der Funktionsweise eines Rechners genügen, sollen als *(Modell-)Objektmenge* bezeichnet werden. Die Objekte müssen nicht unbedingt konkret vorhanden sein, sie können auch virtuell (durch Hardware oder Software interpretiert) vorliegen. Zur Objektmenge zählen also keine zusätzlichen Hilfsregister oder dergleichen, die erst bei der Realisierung/Implementierung dazu kommen. In diesem Zusammenhang ist auch der Begriff *Programmiermodell* (auch *Registermodell*) zu erwähnen, mit dem im wesentlichen die Register gemeint sind, auf die der Programmierer mit Hilfe der Maschinenbefehle Zugriff hat. Der Begriff *Objektmenge* ist umfassender, da er auch die sonstigen Objekte, wie Hauptspeicher und Ein-/Ausgabe-Schnittstelle, mit einschließt.

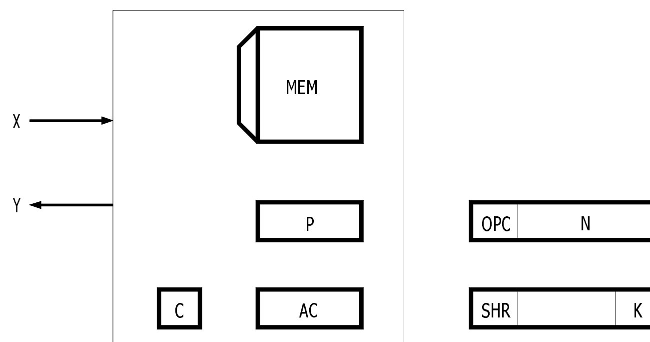


Abbildung 6.1: Objektmenge und Befehle

Die Objektmenge des Beispiel-Rechners (Abb. 6.1) besteht aus dem Hauptspeicher `MEM`, dem Befehlszähler `P`, dem Akkumulator `AC`, dem Carry-Bit `C`, den Eingangssignalen `X` und den Ausgangssignalen `Y`. Die DINATOS-Architektur (Wirkung der Maschinenbefehle auf die Objektmenge) läßt sich durch ein HDL-Programm beschreiben/spezifizieren (Abb. 6.2). Es dient dazu, dem Leser die Funktionsweise in prägnanter Form hinreichend genau zu erklären oder als Spezifikation im Entwurfsprozeß. Entsprechend der obigen Definition der Objektmenge werden in der Beschreibung keine unnötigen Hilfsobjekte verwendet, die erst bei der Realisierung bzw. Emulation (Abschnitte 6.2.2 bzw. 6.2.3.2) hinzugefügt werden.

```
 1 unit COMPUTER'DINATOS;                     "Architekturbeschreibung"
 2 input X[32]; output Y[32];                 "Eingangs-/Ausgangssignale"
 3 boole MEM('FFFF FFFF':0,31:0),             "Hauptspeicher"
 4       AC[32],                              "Akkumulator"
 5       P[32],                               "Befehlszaehler"
 6       C;                                   "Condition Flag"
 7
 8 macro <OPC> ::= MEM(P,31:24) mend          "OPC=Befehlscode"
 9 macro <N>   ::= MEM(P,27:0)  mend          "N=Adresse,Konstante"
10 macro <K>   ::= MEM(P,4:0)   mend          "K=Schiftzahl"
11 macro <MN>  ::= MEM(<N>)     mend          "Inhalt der Speicherzelle N"
12 macro <MMN> ::= MEM(<MN>)    mend          "Inhalt vom Inhalt N"
13
14 perm Y==MEM('FFFF FFFF') pend              "Ausgabespeicherzelle"
15
16 loop MEM('FFFF FFFE'):=X lend              "Eingabespeicherzelle"
17
18 loop                                       "Interpretation der Befehle"
19    case <OPC> of
20    ?0:  "NOP"                      P:=P+1  "No Operation"
21    ?1:  "NOT"   AC:= ~AC          ;P:=P+1  "Negation"
22    ?2:  "SHR K" AC:= K shr AC     ;P:=P+1  "Schift um K Stellen"
23    ?3:  "LDA'N" AC:= <N>          ;P:=P+1  "Lade Konstante"
24    ?4:  "LDA N" AC:= <MN>         ;P:=P+1  "Lade Inhalt Speicherzelle"
25    ?5:  "LDA*N" AC:= <MMN>        ;P:=P+1  "Lade Inhalt vom Inhalt"
26    ?6:  "STA N" <MN>:= AC         ;P:=P+1  "Speichere nach Adr. N"
27    ?7:  "STA*N" <MMN>:= AC        ;P:=P+1  "Speichere indirekt"
28    ?8:  "AND N" AC:= AC.<MN>      ;P:=P+1  "Und"
29    ?9:  "ADD N" AC:= AC+<MN>      ;P:=P+1  "Addition"
30    ?10: "MUL N" AC:= AC*<MN>      ;P:=P+1  "Multiplikation"
31    ?11: "IF>'N" C:= AC > <N>      ;P:=P+1  "Akkuinhalt > Konstante"
32    ?12: "IF='N" C:= AC = <N>      ;P:=P+1  "Akkuinhalt = Konstante"
33    ?13: "GO' N" P:= <N>                    "Sprung nach N"
34    ?14  "GO  N" P:= <MN>                   "Indirekter Sprung"
35    ?15: "DO' N" if ~C then P:= P+<N>+1 "Bedingter Sprung"
36                       else P:= P+1 fi
37    esac
38 lend
39 uend
```

Abbildung 6.2: Architektur des Rechners DINATOS

Die Objektmenge von DINATOS wird in den Programmzeilen 2-6 festgelegt. Die Makros (Zeilen 8-12) definieren die Aufteilung des 32-Bit-Maschinenbefehls MEM(P) in den Operationscode OPC (8 Bit) und die Adresse/Konstante N (24 Bit) oder die Schiftzahl (5 Bit). Das Makro <MN> bezeichnet die Speicherzelle MEM(<N>) und das Makro <MMN> die indirekt adressierte Speicherzelle MEM(MEM(<N>)). Die Speicherzelle mit der Adresse 'FFFFFFFF' dient zur Ausgabe, sie ist permanent mit den Ausgangssignalen Y verbunden (Zeile 14). Die Speicherzelle mit der Adresse 'FFFFFFFE' dient zur Eingabe, die X-Werte werden wiederholt in diese Speicherzelle eingeschrieben (Zeile 16). (Das genaue zeitliche Ein-/Ausgabeverhalten ergibt sich erst bei einer Realisierung.) Die Wirkung der Befehle wird durch die Interpretationsschleife (Zeilen 18-38) beschrieben. Wenn z.B. der Operationscode des durch P adressierten Befehls gleich 9 ist, dann wird zu dem Akkumulatorinhalt der Inhalt der Speicherzelle N addiert und danach der Befehlszähler um 1 erhöht. Die Operationscodes 13, 14, 15 kennzeichnen Sprungbefehle, die nur den Befehlszähler P verändern.

6.2.2 Direkte Hardware-Steuerung

Für die DINATOS-Architektur werden verschiedene Implementierungsentwürfe auf der Register-Transfer-Ebene gemacht, wobei zunächst die Interpretation der Befehle direkt durch ein Hardware-Steuerwerk erfolgen soll.

Jeder Entwurf besteht aus einem Steuerwerk und einem Operationswerk (Hauptspeicher, Rechenwerk, Adreßrechenwerk, Register usw.). Die Hardware-Steuerwerke werden durch Zustandsdiagramme oder HDL-Abläufe charakterisiert, deren Umsetzung in entsprechende Steuerwerksrealisierungen als bekannt vorausgesetzt (Abschn. 4.5.2) wird.

Die Objektmenge von DINATOS besteht aus den Objekten X, Y, MEM, AC, PC und C (Abschnitt 6.2.1). Sie wird jetzt um Hilfsobjekte erweitert, die bei der Implementierung zusätzlich benötigt werden. Die Objektmenge zusammen mit den Hilfsobjekten soll *Implementierungsobjektmenge* heißen.

Wir wollen drei verschiedene Implementierungsentwürfe betrachten. Im Entwurf 1a werden folgende Hilfsobjekte benutzt:

BC[32]	Hilfsakkumulator
BR[32]	Befehlsregister
AR[32]	Adreßregister

Auf der Implementierungsobjektmenge wird nun der Steuerablauf in Form eines Zustandsdiagramms (Abb. 6.3) oder eines HDL-Programms entwickelt. Aus dem Zustandsdiagramm können die benötigten Mikrooperationen entnommen werden:

```
AC      <- ~AC, K shr AC, N, MEM(AR), AC.BC, AC+BC
AC_BC   <- AC**BC
BC      <- MEM(AR)
```

```
C       <- (AC>BC), (AC=BC)
BR      <- MEM(P)
P       <- P+1, N, P+N, MEM(AR)
AR      <- N, MEM(AR)
MEM(AR):= AC
```

Aus den Mikrooperationen ergeben sich wiederum die benötigten ALU-Funktionen `(NOT, K shr, UND, +, **)`, die Vergleichsfunktionen (>, =), die Adreßberechnungen `(P+1, P+N)`, die Speicheradressierung (mit `AR, P)`, das Auslesen des Speicherinhalts nach `AC`, `BC` oder `AR`, und das Schreiben von `AC` in den Speicher. Zusätzlich sind eine Reihe von direkten Verbindungen zwischen den Registern zu realisieren. Um den Implementierungsaufwand zu minimieren, kann anschließend versucht werden, direkte Verbindungen durch indirekte Verbindungen zu ersetzen, Busse und Funktionseinheiten mehrfach zu nutzen, und zu aufwendige Funktionen oder Adreßberechnungen in eine Folge einfacherer Operationen zu zerlegen.

Die eben benutzte Methode könnte man als Top-Down-Methode bezeichnen, weil zuerst der Ablauf definiert wird, ohne sich Einschränkungen in Bezug auf die Realisierbarkeit aufzuerlegen. Diese Vorgehensweise führt leicht zu einer zu aufwendigen Realisierung, so daß eine anschließende Optimierung mit dem Ziel der Aufwandsminimierung stattfinden muß. Bei der Bottom-Up-Methode geht man umgekehrt vor. Zuerst werden die ALU- und Vergleichsfunktionen, die Adreßberechnungen, die Speicheransteuerung und die Verbindungen zwischen den Komponenten definiert. Dadurch ergeben sich die möglichen Mikrooperationen, die dann beim anschließenden Entwurf des Steuerablaufs benutzt werden können. Dabei werden eventuell weitere Mikrooperationen benötigt, oder einige der definierten werden nicht benutzt oder erweisen sich als unzweckmäßig. Die Mikrooperationen und der Steuerablauf müssen also zusammen optimiert werden, damit das geforderte Preis/Leistungsverhältnis erreicht werden kann.

Die hier vorgestellten Implementierungsentwürfe stellen nur prinzipielle Lösungen auf der Register-Transfer-Ebene dar; für eine echte Hardware-Realisierung wären noch weitere Umsetzungsschritte erforderlich.

Im Entwurf 1a wird ein Hauptspeicher mit einer Zugriffszeit von $t_{rw} = 1$ Taktzyklus vorausgesetzt. Der Steuerablauf (Zustandsdiagramm Abb. 6-3) schaltet ebenfalls mit dem Takt von Zustand zu Zustand. Im Zustand 1 wird der Befehl geholt und der Programmzähler schon um Eins erhöht. Im Zustand 2 findet die Decodierung des Operationscodes statt, der im Befehlsregister steht. Bei den einfachen Befehlen `NOP, NOT,` usw. werden gleichzeitig die notwendigen Mikrooperationen veranlaßt, so daß sofort zum Zustand 1 zurückgekehrt werden kann. Die Befehle `LDA, LDA*` usw. erfordern weitere Zustände. Da das Zustandsdiagramm einfach ist, wird es am besten durch ein Hardware-Steuerwerk, z.B. mit Hilfe eines PLAs realisiert.

Es stellt sich nun die Frage, ob nicht eine zeitoptimalere Lösung gefunden werden kann und wie

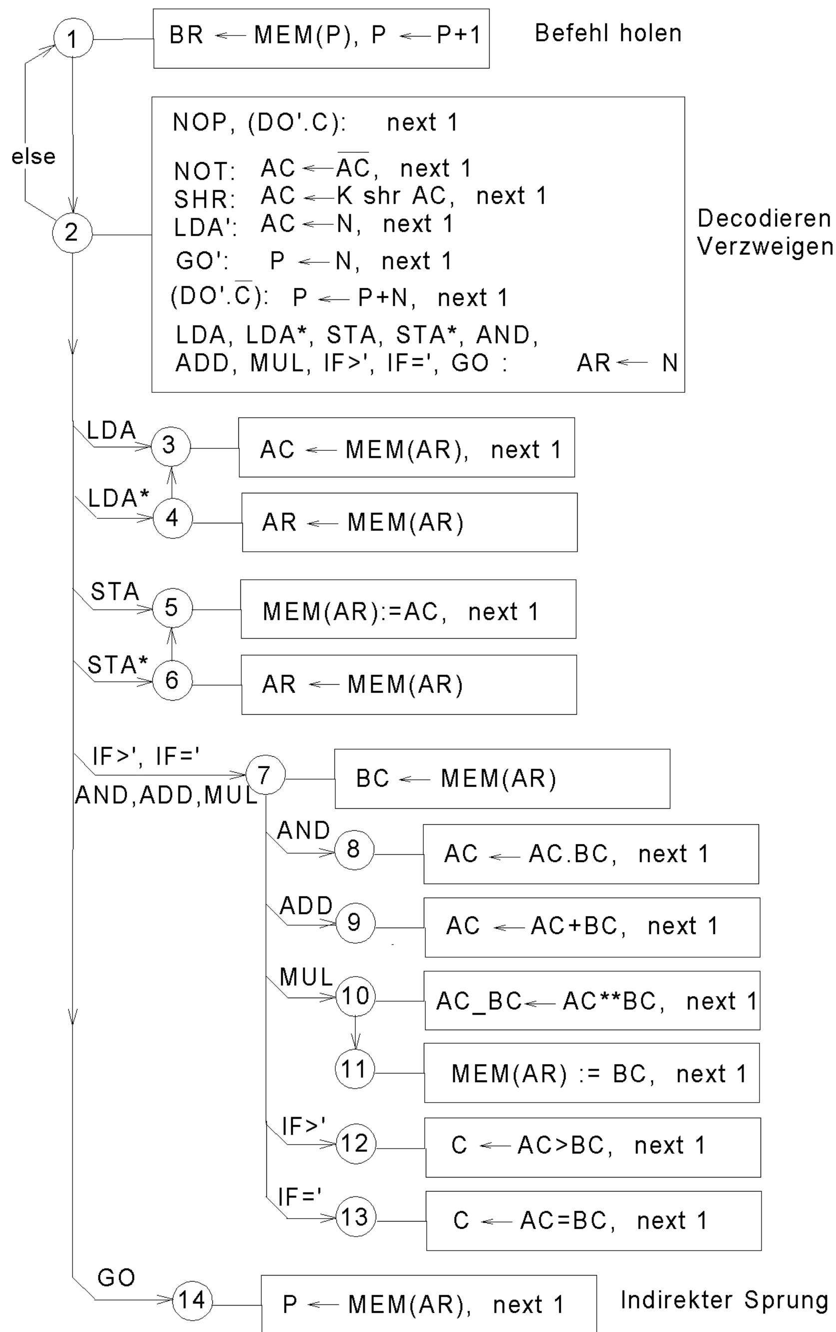


Abbildung 6.3: Zustandsdiagramm für den Entwurf 1a

sich eine Erhöhung der Speicherzugriffszeit auswirken würde. Für eine angenommene Häufigkeitsverteilung H für die Befehle (Abb. 6.4) ergibt sich eine mittlere Zeit von 3,2 Zyklen pro

	NOP NOT SHR LDA' IF	GO'	LDA STA	LDA* STA*	AND ADD	MUL	GO	DO' C=l	DO' C=o	Mittelwert für H
Speicher-zugriffe	1	1	2	3	2	3	2	1	1	
Zyklen bei Version 1a ($t_{rw}=1$)	2	2	3	4	4	5	3	2	2	3,2
Zyklen bei Version 1b ($t_{rw}=2$)	3	3	5	7	6	8	5	3	3	5,1
Zyklen bei Version 1c ($t_{rw}=1$)	1	2	2	3	3	4	3	1	2	2,3
Zyklen bei Version 2 ($t_{rw}=2$)	2	3	4	6	5	7	5	2	3	4,15
Angenommene Häufigkeit H[%]	10	4	40	10	15	9	4	4	4	

Abbildung 6.4: Ausführungszeiten der Maschinenbefehle

Befehl. Das ergibt 31 MIPS (million instructions per second) bei einer Taktfrequenz von 100 MHz. Im Entwurf 1b soll die Speicherzugriffszeit auf $t_{rw} = 2$ Takte erhöht werden. Die mittlere Ausführungszeit erhöht sich dann auf 5,1 Zyklen pro Befehl, wenn der Steueralgorithmus bis auf zusätzliche Wartezyklen beibehalten wird.
Die Abb. 6.5 zeigt einen weiteren Entwurf 1c in Form eines synchronen HDL-Programms. Dabei beträgt die Speicherzugriffszeit $t_{rw} = 1$ Taktzyklus, wie im Entwurf 1a. Der Ablauf wurde aber zeitlich optimiert, indem das Holen des nächsten Befehls (macro <Get>) parallel zur Decodierung und Ausführung durchgeführt wird (Überlappung der Holphase mit den anderen Phasen). Dadurch kann in den meisten Fällen ein Zyklus eingespart werden, so daß die mittlere effektive Ausführungszeit nur noch 2,3 Zyklen beträgt. An diesem Beispiel wird deutlich, daß sich eine Optimierung des Steuerablaufs durch Maßnahmen wie Überlappung und Pipelining durchaus lohnt. Solche Optimierungen lassen sich in Hardware- Steuerwerken leichter als in Mikroprogrammsteuerwerken realisieren.

Ein weiterer Entwurf ist in den Abb. 6.6 und 6.7 dargestellt, bei der eine Speicherzugriffszeit von 2 Takten angenommen wurde. Gegenüber dem Entwurf 1b wurde er aber zeitlich optimiert, wobei der nächste Befehl schon im Vorgriff geholt wird. Die Zustände 0 und 1 dienen zum Initialisieren. Im Zustand 2 wird der Befehl i decodiert und gleichzeitig wird schon begonnen, den nächsten Befehl zu lesen. Wenn der Zustand 3a (oder 3b oder 3c) verlassen wird, dann steht schon der nächste Befehl zur Verfügung (im Zustand 2, 4 bis 12). Eine Sonderbehandlung erfordern die Sprungbefehle (Abb. 6.7). Dabei wird im Zustand 2 anstelle des Befehls i+1 der Befehl an der Sprungadresse geholt. Bei dem Befehl DO' mit der Bedingung C=1 wird normal der nächste Befehl i+1 geholt, während mit C=o ein Sprung erfolgt. In den Abbildungen sind die Befehle LDA*, AND, IF>', IF=' und MUL nicht dargestellt, da ihre Abläufe sich einfach konstruieren lassen. Die mittlere Anzahl der Zyklen pro Befehl beträgt 4,4. Durch

```
 1 unit DINATOS'SYNCHRON;               "Synchrone Realisierung"
 2 input X[32]; output Y[32];
 3 boole MEM('FFFF FFFF':0)[32];        "Hauptspeicher"
 4 register AC[32],                     "Akkumulator"
 5          P[32],                      "Befehlsz\"ahler"
 6          C,                          "Condition Flag"
 7          BC[32],                     "Hilfsakku"
 8          BR[32],                     "Befehlsregister"
 9          AR[32];                     "Adressregister"
10
11 macro <OPC> ::= BR(31:28)     mend  "Befehlscode"
12 macro <N>   ::= BR(27:0)      mend  "Adresse, Konstante"
13 macro <K>   ::= oooo_BR(4:0)  mend  "Schiftzahl"
14 macro <Get> ::= BR<-MEM(P), P<-P+1, mend  "Befehl holen"
15
16 on clock
17 Y==MEM('FFFF FFFF'),                 "Eingabespeicherzelle"
18 asyn MEM('FFFF FFFE'):=X aend,       "Ausgabespeicherzelle
19
20 [1]  BR<-MEM(0), P<-1                "Init. Befehl 0 holen"
21 [2]  case <OPC> of
22       ?0: "NOP"                        <Get> next 2
23       ?1: "NOT"    AC <- ~AC,          <Get> next 2
24       ?2: "SHR K"  AC <- <K> shr AC,   <Get> next 2
25       ?3: "LDA'N"  AC <- <N>,          <Get> next 2
26       ?4: "LDA N"  AR <- <N>,          <Get> next 3
27       ?5: "LDA*N"  AR <- <N>,          <Get> next 4
28       ?6: "STA N"  AR <- <N>,          <Get> next 5
29       ?7: "STA*N"  AR <- <N>,          <Get> next 6
30       ?8: "AND N"  AR <- <N>,                next 7
31       ?9: "ADD N"  AR <- <N>,                next 9
32      ?10: "MUL N"  AR <- <N>,                next 11
33      ?11: "IF>'N"  C  <- AC > <N>,     <Get> next 2
34      ?12: "IF='N"  C  <- AC = <N>,     <Get> next 2
35      ?13: "GO' N"  P  <- <N>,                next 14
36      ?14: "GO  N"  AR <- <N>,                next 15
37      ?15: "DO' N"  if C then           <Get> next 2
38                    else P<-P+1+<N>           next 14 fi
39
40 [3]  "LDA N"  AC <- MEM(AR),                 next 2
41 [4]  "LDA*N"  AR <- MEM(AR),                 next 3
42 [5]  "STA N"  asyn MEM(AR):=AC aend,         next 2
43 [6]  "STA*N"  AR <- MEM(AR),                 next 5
44 [7]  "AND N"  BC <- MEM(AR),                 next 8
45 [8]           AC <- AC.BC,             <Get> next 2
46 [9]  "ADD N"  BC <- MEM(AR),                 next 10
47 [10]          AC <- AC+BC,             <Get> next 2
48 [11] "MUL N"  BC <- MEM(AR),                 next 12
49 [12]          AC_BC <- AC**BC,         <Get> next 13
50 [13]          asyn MEM(AR):=AC aend,         next 2
51 [14] "GO' N, DO' N"                    <Get> next 2
52 [15] "GO  N"  P <- MEM(AR),                  next 14
53
54 noc uend
```

Abbildung 6.5: Synchrone Realisierung mit Überlappung

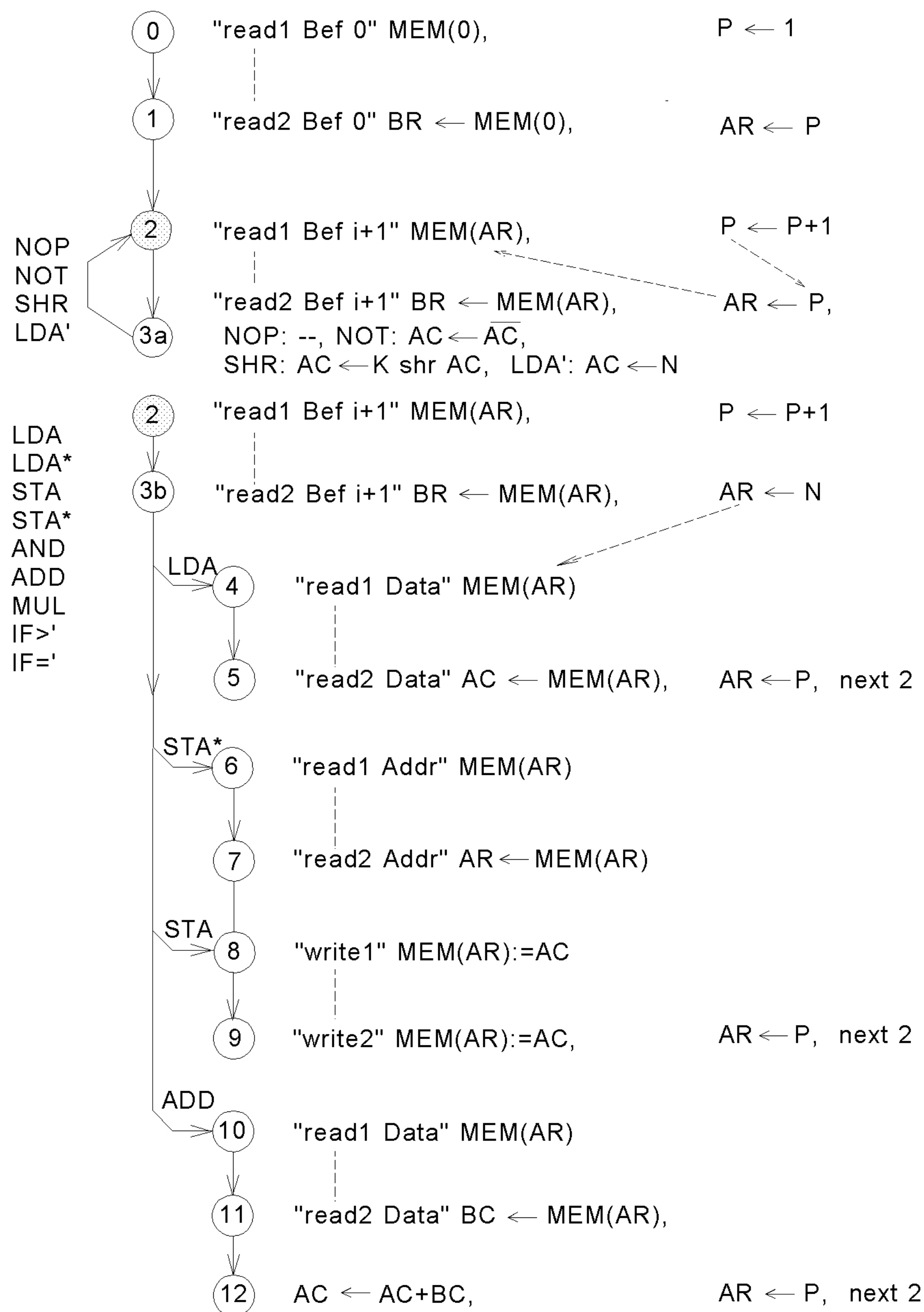


Abbildung 6.6: Zustandsdiagramm für den Entwurf 2

Anwendung des Überlappungsprinzips konnte also knapp ein Zyklus gegenüber dem Entwurf 1b eingespart werden.

```
 [2] if GO' then     AR<-N,        P<-N,    next 13 fi

[13] "read1 Bef N"   BR<-MEM(AR), P<-P+1, next 14

[14] "read2 Bef N"   BR<-MEM(AR), AR<-P,  next 2
```

```
 [2] if GO then AR<-N, next 15 fi

[15] "read1 Addr"    AR<-MEM(AR), next 16

[16] "read2 Addr"    AR<-MEM(AR), P<-MEM(AR), next 17

[17] "read1 Bef N"   BR<-MEM(AR), P<-P+1, next 18

[18] "read2 Bef N"   BR<-MEM(AR), AR<-P,  next 2
```

```
 [2] if DO'.(C=l) then
     "read1 Bef i+1" BR<-MEM(AR), P<-P+1, next 3c fi,
     if DO'.(C=o) then AR<-P+N,   P<-P+N, next 19 fi,

[3c] "read2 Bef i+1" BR<-MEM(AR), next 2

[19] "read1 Bef N"   BR<-MEM(AR), P<-P+1, next 20

[20] "read2 Bef N"   BR<-MEM(AR), AR<-P,  next 2
```

Abbildung 6.7: Ergänzung zu Abb. 6-6, Sprungbefehle

Bei der vorgegebenen Befehlsliste und Architektur DINATOS sind im Mittel ca. 2,2 Speicherzugriffe pro Befehl notwendig. Um eine kleinere Bearbeitungszeit zu erreichen, kann man die Speicherbandbreite erhöhen (z.B. durch Cache-Speicher, breitere Busse), so daß mehrere Befehle parallel (überlappend, Pipelineverarbeitung) ausgeführt werden können. Ein andere Maßnahme zur Leistungssteigerung wäre die Veränderung der Architektur, z.B. durch den Einsatz eines schnellen Mehrport-Registerspeichers, wodurch die Anzahl der Speicherzugriffe verringert würde.

6.2.3 Mikroprogrammierbarer Rechner PIRI

Im nächsten Abschnitt wird der mikroprogrammierbare Rechner PIRI entworfen, der zur Emulation der DINATOS-Architektur dienen soll und darüber hinaus zur Emulation einer Vielzahl von 32-Bit-Rechnern geeignet ist.

Einen mikroprogrammierbaren Rechner kann man auch als RISC-Rechner mit Harvard-Architektur auffassen, wenn man den Mikroprogrammspeicher als Maschinenprogrammspeicher benutzt. Wir verwenden deshalb anstelle des Begriffs *mikroprogrammierbarer Rechner* auch den Begriff *Mikromaschine*, um zum Ausdruck zu bringen, daß ein mikroprogrammierbarer Rechner auch als RISC-Rechner eingesetzt werden kann. Insofern kann man den folgenden Entwurf der Mikromaschine PIRI auch als Entwurf eines RISC-Rechners auffassen.

6.2.3.1 Entwurf der Mikromaschine PIRI

Zunächst wurden grundsätzliche Entscheidungen getroffen: Die Breite des Datenbusses, Speichers und der Register soll 32 Bit betragen. Der Mikrobefehl soll 64 Bit breit sein, um eine 32-Bit-Konstante und die verschiedenen Felder gut unterbringen zu können.

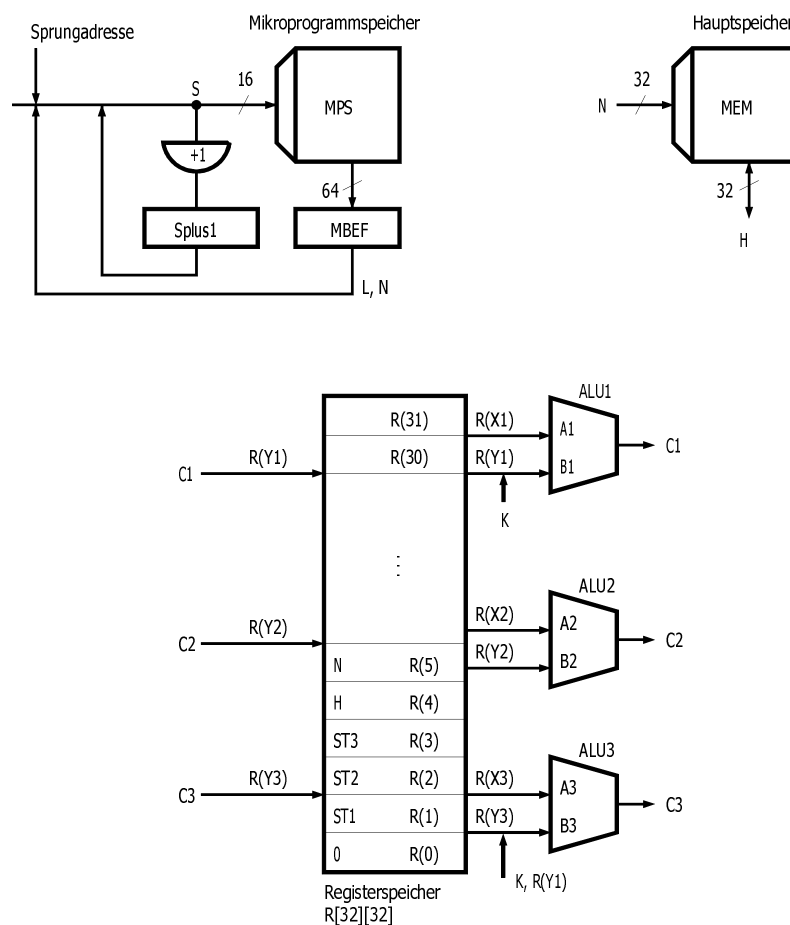


Abbildung 6.8: Die Mikromaschine PIRI

Wie groß sollte die Anzahl der Register sein? Um den Mikrobefehl möglichst kurz zu machen,

sollte sie möglichst klein sein, weil die Registeradresse mehrfach im Mikrobefehl untergebracht werden muß. Andererseits darf sie nicht zu klein sein, um möglichst alle Register der zu emulierenden Architektur (Stackpointer, Indexregister, Basisregister, Unterprogrammparameter usf.) im Registerspeicher unterbringen zu können.

Um mehrere Operationen parallel durchführen zu können, wurden drei Rechenwerke (ALU, Arithmetic and Logic Unit) vorgesehen, die gleichzeitig auf dem Registerspeicher operieren. Deshalb ist er hardwaremäßig als Multiportspeicher mit 6-fachem Lese- und 3-fachem Schreibzugriff zu realisieren. Der hardwaremäßige Aufwand steigt mit jeder zusätzlichen ALU erheblich an, so daß die Anzahl begrenzt werden muß. Außerdem sinkt die mittlere Auslastung der ALUs mit ihrer Anzahl, so daß normalerweise nicht mehr als zwei oder drei ALUs realisiert werden sollten.

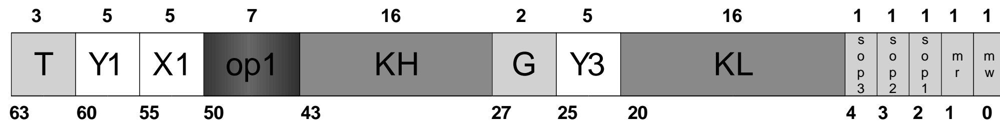
Ein wichtiges Entwurfsziel war eine möglichst homogene Objektmenge, die u.a. folgende Vorteile bietet:

- Übersichtliche Hardware-Struktur,
- Programmierfreundlichkeit durch wenige, klar gegliederte Mikrobefehlstypen.

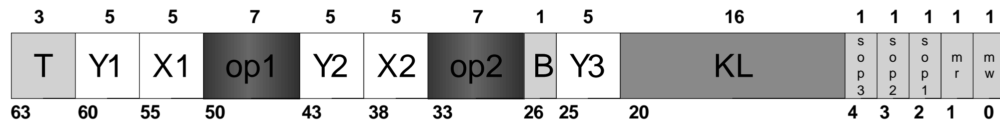
Um dieses Ziel zu erreichen, wurden die Register mit Sonderfunktionen (Folgeadreßregister, Statusregister, Adreßregister, Datenregister) in das Registerfile integriert und gleichartige ALUs verwendet.

Man hätte auch eine ALU mit andersartigen Operationen, insbesondere Gleitkommaoperationen, vorsehen können. Dadurch kommt man zu inhomogenen Strukturen und die Hardware-Steuerung wird komplizierter, wenn die ALU-Operationen mehrere Takte zu Ausführung benötigen. Ein weitergehendes Konzept besteht darin, für jeden Datentyp ein separates Rechenwerk mit getrennten Registerspeichern vorzusehen.

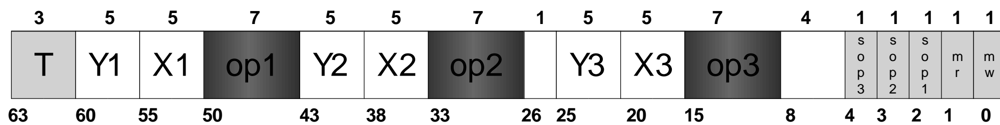
Es stellt sich auch die Frage, welche Mikrooperationen mit Hilfe der ALU-Funktionen realisiert werden sollen. Aus Sicht des Programmierers werden möglichst universelle Transportoperationen, logische Operationen und arithmetische Operationen, auch unter Verwendnung von Konstanten, benötigt. Die zu realisierenden Operationen können aus den am häufigsten benutzten Datentypen abgeleitet werden. Es sind normalerweise Wahrheitswerte (1 Bit), BCD-Ziffern (4 Bits, Nibbles), Zeichen (7, 8, 16 Bits), natürliche Zahlen und Zweikomplementzahlen mit 8, 16, 32, 64 Bits, Gleitkommazahlen mit 32, 64 und mehr Bits. Diese Datentypen treten nicht nur als Skalare, sondern auch als Vektoren auf. Da nicht alle wünschenswerten Operationen realisiert werden können, muß man sich auf eine mehr oder weniger sinnvolle Untermenge beschränken. Dabei werden bevorzugt solche Operationen gewählt, die möglichst vielseitig benutzt und schlecht durch eine Folge anderer Operationen nachgebildet werden können. Die in der vorliegenden Mikromaschine gewählten ALU-Funktionen und Mikrooperationen stellen eine mögliche Auswahl dar, die noch optimiert und erweitert werden kann. Z.B.



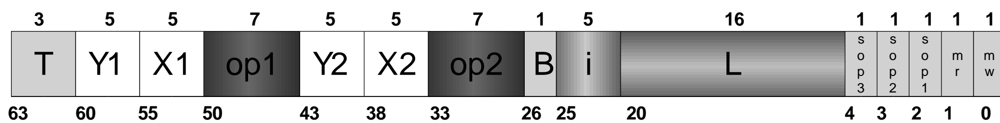

```
T=1: G=0: R(Y1) <- op1(Y1,X1),    R(Y3) <- KH_KL
     G>0: R(Y1) <- op1(KH_KL,X1), R(Y3) <- R(Y1)
```



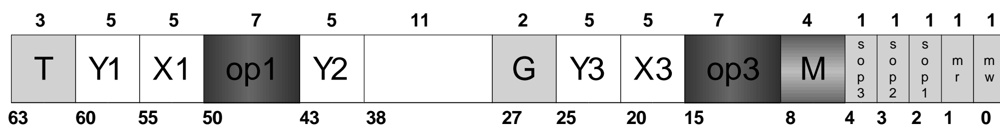

```
T=2: B=0: R(Y1) <- op1(Y1,X1), R(Y2) <- op2(Y2,X2), R(Y3) <- KL
     B>0: R(Y1) <- op1(KL,X1), R(Y2) <- op2(Y2,X2), R(Y3) <- R(Y1)
```




```
T=3: R(Y1) <- op1(Y1,X1), R(Y2) <- op2(Y2,X2), R(Y3) <- op3(Y3,X3)
```




```
T=4,5,6: R(Y1) <- op1(Y1,X1), R(Y2) <- op2(Y2,X2),
T=4:            if ST1(i)=B   then next L
T=5:            if ST2(i)=B   then next L
T=6:            if ST3(i)=B   then next L
```




```
T=7: R(Y1) <- op1(Y1,X1), R(Y3) <- op3(Y3,X3),
       G=0:     next M_R(Y2,31:24)_0000
       G=1:     next M_R(Y2,23:16)_0000
       G=2:     next M_R(Y2,15:8)_0000
       G=3:     next M_R(Y2,7:0)_0000
```

Abbildung 6.9: Mikrobefehle

könnte man Bitfeld-Transport-Operationen, Addition von BCD-Zahlen, Multiplikation und Division, Gleitkommaoperationen und Vektoroperationen hinzufügen. Dabei erhöht sich nicht unwesentlich der Realisierungsaufwand und die Komplexität der Mikrobefehle. Insbesondere verletzt man mit Operationen, die eine größere Anzahl von Takten benötigen, ein Prinzip der Mikroprogrammierung, nämlich die Ausführbarkeit in sehr wenigen (meist einem) Takten. Denn ein Mikrobefehl soll sich gerade aus solchen elementaren Mikrooperationen zusammensetzen, die es erlauben, durch ein kurzes Mikroprogramm die gewünschte Operation zu erzeugen.

Die entworfene Mikromaschine PIRI (Abb. 6-8) besitzt 32 Register `R(0)` bis `R(31)` zu je 32 Bit, einen Hauptspeicher `MEM` mit maximal $2^{32}*$ 32 Bit und einen Mikroprogrammspeicher `MPS` mit maximal $2^{32}*$ 64 Bit. Das Register `R(0)` besitzt beim Lesen konstant den Wert Null. Wird die Registeradresse 0 beim Schreiben benutzt, so wird die zugehörige Mikrooperation nicht ausgeführt. Die Register `R(1:3)` dienen als Statusregister `ST1, ST2, ST3`. Das Register `R(4)` enthält die normale Folgeadresse `Splus1` des nächsten Mikrobefehls, die immer dann benutzt wird, wenn kein Sprung erfolgt.

Das Register `R(5)=H` dient als Datenregister und das Register `R(6)=N` als Adreßregister des Hauptspeichers. Durch Angabe der Registeradressen `Y1,X1,Y2,X2,Y3,X3` können über eine Zugriffslogik des Registerspeichers maximal 6 Registerinhalte parallel ausgewählt und auf die Eingänge `B1,A1` und `B2,A2` und `B3,A3` der drei ALUs geschaltet werden. Die drei ALUs erzeugen die Ergebnisse `C1,C2,C3`, die in die Register `R(Y1), R(Y2)` und `R(Y3)` gleichzeitig eingeschrieben werden können. Außerdem werden von den ALUs bestimmte Bedingungen (z.B. Carry, Overflow, s. HDL-Beschreibung) erzeugt, die in den Statusregistern bei Bedarf (durch Angabe der Bits `sop1, sop2, sop3` im Mikrobefehl) zwischengespeichert und durch bedingte Mikrobefehlssprünge abgefragt werden können.

Das Mikroprogrammsteuerwerk ist vom Typ D Moore (s. Abschn. 5.6.3). Der Mikroprogrammspeicher wird mit der Folgeadresse `S` adressiert. Der nächste Mikrobefehl `MPS(S)` wird mit dem Takt in das Mikrobefehlsregister `MBEF` übernommen. Je nach Befehlstyp und Bedingung ist `S` die normale Folgeadresse `Splus1` oder die Sprungadresse `L` oder eine Mehrfachverzweigungsadresse. Die normale Folgeadresse wird immer vorausberechnet `(Splus1 ← S+1)` und steht im Register `Splus1` zur Auswahl bereit.

Abb. 6-9 zeigt die Mikrobefehle aufgeteilt in die einzelnen Felder und in symbolischer Form. Der Mikrobefehlstyp ist durch `T[3]` bestimmt. Bei den Typen 1, 2 und 7 werden zusätzlich die Felder `G` bzw. `H` ausgewertet, so daß Untertypen gebildet werden. Die ALU-Operationen werden durch `op1[5]`, `op2[5]` und `op3[5]` festgelegt. Das Feld `K[32]` bzw. `K[16]` stellt eine Rechenkonstante dar. `L[16]` ist eine Sprungadresse und `N[4]` bestimmt den Anfang einer Seite.

Typ 1. Bei `G=0` wird eine ALU-Operation ausgeführt und eine Konstante in ein Register geladen. Bei `G>0` wird eine ALU-Operation mit einer Konstanten und ein Registertransport ausge-

```
 1 unit   PIRI;                        "Mikromaschinenarchitektur"
 2 boole  MEM('FFFF FFFF':0)[32];      "Hauptspeicher"
 3        R[32][32],                   "Registerspeicher"
 4        MPS('FFFF':0)[64];           "Mikroprogrammspeicher"
 5
 6 register
 7        MBEF(63:0);                  "Mikrobefehl"
 8
 9 signal S[32],                       "Folgeadresse"
10        (A1,A2,A3,B1,B2,B3)[32],     "ALU-Eingaenge"
11        (CC1,CC2,CC3)[33],           "ALU-Ausgaenge mit Carry"
12        hop3                         "Hilfssignal fuer op3"
13
14 equal  ST1    = R(1),               "Statusregister1"
15        ST2    = R(2),               "Statusregister2"
16        ST3    = R(3),               "Statusregister3"
17
18        H      = R(4),               "Datenregister Hauptspeicher"
19        N      = R(5),               "Adressregister Hauptspeicher"
20        C1     = CC1[32],            "ALU-Ausgaenge ohne Carry"
21        C2     = CC2[32],
22        C3     = CC3[32],
23        T[3]   = MBEF(63:61),        "Mikrobefehlstyp"
24        Y1[5]  = MBEF(60:56),        "Registeradresse Y1"
25        X1[5]  = MBEF(55:51),        "Registeradresse X1"
26        op1[7] = MBEF(50:44),        "Operation ALU1"
27        Y2[5]  = MBEF(43:39),        "Registeradresse Y2"
28        G[2]   = MBEF(27:26),        "Erweiterter Operationscode"
29        K[16]  = MBEF(43:28),        "Konstante High"
          KL[16] = MBEF(20:5),         "Konstante Low"
30        X2[5]  = MBEF(38:34),        "Registeradresse X2"
31        op2[7] = MBEF(33:27),        "Operation ALU2"
32        B      = MBEF(26),           "Erweiterter Operationscode"
33        X3[5]  = MBEF(20:16),        "Registeradresse X3"
34        op3[7] = MBEF(15:9),         "Operation ALU3"
35        sop1   = MBEF(2),            "Statusoperation 1"
36        sop2   = MBEF(3),            "Statusoperation 2"
37        sop3   = MBEF(4),            "Statusoperation 3"
38        mr     = MBEF(1),            "Memory Read"
          mw     = MBEF(0),            "Memory Write"
39        i[5]   = MBEF(25:21),        "Bedingungsindex"
40        L      = KL,                 "Sprungadresse"
41        M      = MBEF(8:5);          "Seitenadresse"
          Y3[5]  = MBEF(25:21);        "Registeradresse Y3"
42
43 on clock
44 "*** Mikroprogrammsteuerwerk ***"
45 MBEF      <- MPS(S),                "Mikrobefehl lesen"
46 Splus1    <- S+1,                   "normale Folgeadresse berechn."
47 case T of
48 ?1,2,3:  S==Splus1                  "Kein Sprung"
49 ?4:      if ST1(i)=H then S==L else S==Splus1 "bed. Sprung"
50 ?5:      if ST2(i)=H then S==L else S==Splus1 "bed. Sprung"
51 ?6:      if ST3(i)=H then S==L else S==Splus1 "bed. Sprung"
```

Abbildung 6.10: Register-Transfer-Architektur der Mikromaschine PIRI

```
 52 ?7:      case G of                "256-fach-Verzweigung"
 53          ?0: S==N_R(Y2,31:24)_oooo
 54          ?1: S==N_R(Y2,23:16)_oooo
 55          ?2: S==N_R(Y2,15:8)_oooo
 56          ?3: S==N_R(Y2,7:0)_oooo
 57          esac
 58 esac,
 59
 60 "*** Hauptspeicherzugriff ***"
 61 "read"    if mr then H <- MEM(N) fi,
 62 "write"   if mw then asyn MEM(N):= H aend fi,
 63
 64 "ALU1-Operation"
 65    A1==R(X1),
 66    B1==R(Y1) . ((T=1) . (G=0) v (T=2) . (B=0) v (T>=3)),
 67    B1==KH_KL. (T=1) . (G=1),
 68    B1== '0000'_ KL. (T=2) . (B=1),
 69    if Y1>0 then R(Y1) <- C1 fi,
 70 "ALU2-Operation"
 71  case T of ?2,3,4,5,6 :
 72     A2==R(X2),  B2==R(Y2),
 73     if Y2>0 then R(Y2) <- C2 fi
 74  esac
 75  "ALU3-Operation"
 76    A3==R(X3),
 77  case T then
 78    ?1:   B3==KH_KL. (G=0),
 79          B3==R(Y1) . (G=1),
 80          hop3==3                  "B3 -> C3 durchschalten"
 81    ?2:   B3=='0000'_KL. (B=0),
 82          B3==R(Y1) . (B=1),
 83          hop3=3                   "B3 -> C3 durchschalten"
 84    ?3,7: A3==R(X3),
 85          B3==R(Y3),
 86          hop3==op3                "Operation 3"
 87    ?1,2,3,7: if Y3>0 then R(Y3) <- C3
 88  esac,
 89  R(0)<-0,                         "R0=0"
 90
 91 "*** Operationen ALU1 ***"
 92 case op1 of
 93 "Konstante Laden"
 94    ?0:  C1==  0
 95    ?1:  C1==  oooo oooo oooo oooo oooo oooo ooo_X1    "0..31"
 96    ?2:  C1==  llll llll llll llll llll llll lll_X1    "-1..-32"
 97
 98 "Monadische Operationen"
 99    ?3:  C1==  B1
100    ?4:  C1==  A1
101    ?5:  C1==  ~A1
102    ?6:  C1==  -A1                 "Zweikomplement"
103    ?7:  C1==  o_A1[31]
104    ?8:  C1==  l_A1[31]
105    ?9: CC1==  A1(31)_A1 + 1
106   ?10: CC1==  A1(31)_A1 - 1
107
```

Abbildung 6.10: Register-Transfer-Architektur der Mikromaschine PIRI

```
108 "Dyadische Operationen"
109   ?11:  C1==  B1 . A1             "R(Y1)<-R(Y1).R(X1)"
110   ?12:  C1==  B1 v A1
111   ?13:  C1==  B1 exor A1
112   ?14: CC1==  B1(31)_B1 + A1(31)_A1    "erweitert um Vorzeichen"
113   ?15: CC1==  B1(31)_B1 - A1(31)_A1
114   ?16: CC1==  B1(31)_B1 + A1(31)_A1 + 1
115   ?17: CC1==  B1(31)_B1 + A1(31)_A1 + ST(10)
116   ?18: CC1==  A1(31)_A1 + ST1(10) "fuer Mehrwort-Addition"
117
118 "Schiftoperationen"
119   ?19:  C1==  shr A1              "R(Y1)<- shr R(X1)"
120   ?20:  C1==  shl A1
121   ?21:  C1==  A1(0)_B1(31:1)
122   ?22:  C1==  B1(30:0)_A1(31)
123   ?23:  C1==  ST1(10) inshr A1    "Carry nachziehen"
124   ?24:  C1==  ST1(1) inshl A1     "A1(0) old nachziehen"
125   ?25:  C1==  ST1(2) inshr A1     "Sign old nachziehen"
126   ?26:  C1==  A1(0)_B1(31:1)      "zum Normalisieren"
127   ?27:  C1==  A1(31) inshr A1     "arithmetischer Shift"
128   ?28:  C1==  X1 shr B1           "R(Y1)<- X1 shr R(Y1)"
129   ?29:  C1==  X1 shl B1           "R(Y1)<- X1 shl R(Y1)"
130   ?30:  C1==  X1 cir B1
131   ?31:  C1==  X1 cil B1
132 esac,
133
134 "*** Operationen ALU2 ***"
135 "wie ALU1, ersetze op1 => op2, A1 => A2, B1 => B2,
136  C1 => C2, CC1 => CC2, ST1 => ST2, X1 => X2"
137
138 "*** Operationen ALU3 ***"
139 "wie ALU1, ersetze op1 => op3, A1 => A3, B1 => B3,
140  C1 => C3, CC1 => CC3, ST2 => ST3, X1 => X3"
141
142 "*** Statusoperationen ALU1 ***"
143 if sop1 then ST1(0) <- 0,
144              ST1(1) <- A1(0),
145              ST1(2) <- A1(31),    "Sign"
146              ST1(3) <- B1(0),
147              ST1(4) <- B1(31),
148              ST1(5) <- B1 > A1,
149              ST1(6) <- B1 = A1,
150              ST1(7) <- B1 /= A1,
151              ST1(8) <- C1(0),
152              ST1(9) <- C1(31),
153              ST1(10)<- CC1(32),   "Carry"
154              ST1(11)<- CC1(32) /= CC1(31),   "Overflow"
155              ST1(12)<- C1 = 0,
156              ST1(13)<- C1 /= 0
157 fi,
158
159 "*** Statusoperationen ALU2 ***"
160 "wie Statusoperationen ALU1, ersetze A1 => A2, B1 => B2,
161 C1 => C2, CC1 => CC2, ST1 => ST2"
162 if sop2 then ... fi,
163
164 "*** Statusoperationen ALU3 ***"
165 "wie Statusoperationen ALU1, ersetze A1 => A3, B1 => B3,
166 C1 => C3, CC1 => CC3, ST1 => ST3"
167 if sop3 then ... fi,
168
169 noc uend\"{y}
```

Abbildung 6.10: Register-Transfer-Architektur der Mikromaschine PIRI

führt.
Typ 2. Bei `H=0` werden zwei ALU-Operationen ausgeführt und eine Konstante in ein Register geladen. Bei `H>0` wird eine ALU-Operation mit einer Konstanten, eine normale ALU-Operation und ein Registertransport ausgeführt.

Typ 3. Drei ALU-Operationen werden durchgeführt.

Typ 4, 5, 6. Zwei ALU-Operationen werden durchgeführt. Ein Bit aus einem der drei Statusregister wird ausgewählt und auf `H=o` oder `H=l` getestet. Wenn die Bedingung erfüllt ist, wird ein Sprung nach `L` ausgeführt.

Typ 7. Zwei ALU-Operationen werden durchgeführt. Zusätzlich wird in Abhängigkeit von `G` ein beliebiges Byte aus einem Register ausgewählt, das zusammen mit der Seitenadresse N die nächste Mikroprogrammadresse festlegt. Dadurch wird eine 256-fache Verzweigung durchgeführt, wobei die Adressen einen Abstand von 16 aufweisen.

Zusätzlich kann in jedem Mikrobefehl ein Hauptspeicherzugriff erfolgen und Statusoperationen können aktiviert werden. Durch `rw=lo` (Lesen) wird der Inhalt der Hauptspeicherzelle mit der Adresse `N` gelesen und in das Datenregister `H` übernommen. Durch `rw=ol` (Schreiben) wird der Inhalt des Datenregisters in den Hauptspeicher geschrieben. Vor dem Lesen muß `N` geladen worden sein, vor dem Schreiben `N` und `H`. Die Statusoperationen (Speichern von Bedingungen) werden nur dann ausgeführt, wenn die entsprechenden Steuerbits `sop1, sop2, sop3` gesetzt worden sind.

Die Wirkung der Mikrobefehle ist im einzelnen aus der Register-Transfer-Architekturbeschreibung (Abb. 6-10) zu entnehmen. Zeilen 47-57: Die Folgeadresse `S` wird aus den Alternativen `Splus1`, `L` oder `N_Registerbyte_oooo` ausgewählt. Die darauf folgende Adresse `S+1` wird zusätzlich berechnet und in `Splus1` gespeichert, damit sie im nächsten Schritt benutzt werden kann. Zeilen 60-62: Der Hauptspeicherzugriff dauert einen Takt. Wenn der Haupspeicherzugriff länger dauern würde, müßten Wartezustände in das Mikroprogramm eingebaut werden, wobei vom Prinzip der Überlappung (vergl. Abschnitt 6.2.2, Entwurf 1c) Gebrauch gemacht werden könnte.

Um die HDL-Beschreibung kurz zu halten, ist die Modellierung an einigen Stellen absichtlich unvollständig. Insbesondere werden die Registerzuweisungskonflikte (mehrfache Benutzung derselben Zielregisteradresse) nicht aufgelöst. Sie können durch eine hardwaremäßige Priorisierung oder durch softwaremäßige Überprüfung ausgeschlossen werden.

Zeile 65: Die Registeradresse `Y3` entspricht dem Feld `Y2` für den Typ 0, ansonsten dem Feld `MBEF(26:22)`. Zeilen 67-76: Auf die ALU-Eingänge können jeweils zwei beliebige Registerinhalte geschaltet werden. Auf den ALU-Eingang `B1` kann auch die Konstante `K` geschaltet werden. Zeilen 78-89: Das Register `R(0)` besitzt konstant den Wert Null. Wenn es als Zielregister benutzt wird, dann wird diese Operation nicht ausgeführt. Die ALU-Ausgänge

C1, C2, C3 werden meistens in die Zielregister R(Y1), R(Y2), R(Y3) transportiert. Bei den Typen 1 und 2 wird entweder eine Konstante oder der Registerinhalt R(Y1) in das Register R(Y3) transportiert. Zeilen 91-132: Hier werden die ALU1-Operationen definiert. Für op1=1,2 wird X1 als 5-Bit-Konstante interpretiert. Die Additionen und Subtraktionen (op1=10,11,15-19) werden mit Erweiterung um eine Vorzeichenstelle (sogenannte Schutzstelle) durchgeführt; dadurch läßt sich die Überlaufbedingung an ungleichen Vorzeichenbits einfach erkennen. Für op1=18,19 wird der Übertrag ST1(10) aus einer vorhergehenden Addition berücksichtigt, um eine Mehrwort-Addition zu realisieren. Für op1=24 bis 27 werden spezielle Schifts ausgeführt, die in der HDL-Beschreibung kommentiert sind. Für op1=28 bis 31 werden Schifts auf einem Register mit einer beliebigen Anzahl von Schritten ausgeführt. Zeilen 134-140: Die Operationen der ALU2 und ALU3 sind entsprechend denen der ALU1 definiert. Zeilen 142-157: Wenn das Bit sop1 im Mikrobefehl gesetzt ist, dann werden die aufgeführten Bedingungen im Statusregister ST1 für spätere Abfragen zwischengespeichert. Dabei werden sowohl Bedingungen aus den ALU-Eingängen A1, B1 als auch auf den ALU-Ausgängen C1, CC1 berechnet. Zeilen 159-167: Die Statusoperationen für die ALU2 und ALU3 sind entsprechend definiert.

Die hier vorgestellte Mikromaschine zeichnet sich dadurch aus, daß auf allen Registern die gleichen Mikrooperationen ausgeführt werden können, mit Ausnahme der Register ST1/2/3, Splus1, H und N, auf denen zusätzliche Mikrooperationen erklärt sind. Zuweisungskonflikte können entstehen, wenn diese Register durch die normalen und die zusätzlichen Mikrooperationen benutzt werden. Eine Priorisierung in der Hardware (im HDL-Programm nicht codiert) sollte dafür sorgen, daß die normalen Mikrooperationen vorrangig ausgeführt werden. Dadurch ist es z.B. möglich, den Splus1 mit Hilfe einer ALU-Operation zu berechnen oder das Statusregister zu setzen. Mikrounterprogrammsprünge lassen sich durch Retten des Inhalts von Splus1 realisieren.

Die vergleichsweise regelmäßige Struktur dieser Mikromaschine bietet gegenüber Strukturen mit einer Vielzahl von Spezialregistern und speziellen Mikrooperationen die folgenden vorteilhaften Eigenschaften. Sie ist (1) übersichtlich, (2) programmierfreundlich, da sie wenig Befehlstypen besitzt, (3) erweiterbar hinsichtlich ihrer Leistungfähigkeit ohne eine Änderung ihrer prinzipiellen Struktur und (4) geeignet zur Emulation einer großen Klasse von Rechnerarchitekturen.

Durch die Verwendung programmierbarer ALUs könnte die Anpassungsfähigkeit an die zu emulierende Architektur bzw. an die gewünschten anwendungsbezogenen Operationen (bei Verwendung als RISC-Rechner) weiter gesteigert werden.

Diese Mikromaschine kann als RISC-Rechner mit *Harvard-Architektur* (getrennte Daten- und Programmspeicher) aufgefaßt werden. Der Mikroprogrammspeicher fungiert dann als Maschinenprogrammspeicher und die Mikrobefehle werden als Maschinenbefehle aufgefaßt. In dem Programmspeicher steht dann nicht mehr ein Mikroprogramm zur Interpretation der Maschinenbefehle, sondern das auszuführende Programm, das direkt von der Hardware

interpretiert wird. Durch den Wegfall dieser Interpretationsebene kann die Ausführungszeit reduziert werden.

Wenn der Mikroprogrammspeicher als Programmspeicher eines RISC- Rechners dienen soll, muß er ladbar sein. Zu diesem Zweck könnte er zusätzlich mit einem Adreßregister und einem Datenregister versehen werden oder in den Adreßraum des Hauptspeichers gelegt werden.

6.2.3.2 Emulation von DINATOS auf PIRI

Die im Abschnitt 6.2.1 definierte DINATOS-Architektur soll mit der Mikromaschine PIRI realisiert werden. Zuerst muß überlegt werden, ob und wie die DINATOS-Objektmenge auf die PIRI-Objektmenge abgebildet werden kann. Da die PIRI-Objektmenge eine Obermenge der DINATOS-Objektmenge ist, läßt sich eine Abbildung leicht finden:

```
          DINATOS                 PIRI

Hauptspeicher     MEM     =  MEM
Akkumulator       AC      =  R(6)
Programmzaehler   P       =  R(7)
Condition Flag    C       =  R(8)
Eingangssignale   X       =  MEM('FFFF FFFF')
Ausgangssignale   Y       =  MEM('FFFF FFFE') .
```

Es soll nicht unerwähnt bleiben, daß die Abbildung der Eingangs- und Ausgangssignale auf die Mikromaschine in den meisten Fällen zusätzliche Hardware erfordert. In unserem Fall muß `X` mit dem Eingang der Speicherzelle `MEM('FFFF FFFF')` und `Y` mit dem Ausgang der Speicherzelle `MEM('FFFF FFFE')` verbunden werden. Zur Implementierung werden weiterhin folgende Hilfsregister benutzt:

```
Hilfsakkumulator          BC  =  R(9)
Zaehler                   J   =  R(10)
Daten/Befehlsregister     H   =  R(4)
Adressregister            N   =  R(5)
```

Das Mikroprogramm zur Interpretation/Emulation der DINATOS-Maschinenbefehle zeigt Abb. 6.11. Es ist in symbolischer Form als synchrones Mikroprogramm in Anlehnung an HDL geschrieben, um es verständlich zu halten. Die Übersetzung in die binäre Form kann anhand der im Abschnitt 6.2.1 definierten Codierung der Mikrobefehle erfolgen.

Betrachten wir nun das Mikroprogramm. Zustand 0: Das Adreßregister N des Hauptspeichers wird mit dem Befehlszähler `P` geladen. `P` wurde vorher hardwaremäßig oder durch einen vorgeschalteten Mikrobefehl zu Null initialisiert, damit am Anfang der Befehl an der Stelle `0` geholt werden kann. Außerdem wird dieser Zustand durch den Befehl `NOP` benutzt. Zustand `1`: Der adressierte Befehl wird aus dem Hauptspeicher in das `H`-Register gelesen und `N`

```
NOP  [0]   N<-P                              Initialisieren
[1]  N<-'00FF FFFF', H<-MEM(N)                  Befehl lesen
[2]  N<-N.H,  next H(31:24)_oooo, P<-P+1          case OPC
NOT  [10]  AC<-~AC,  next 1, N<-P
SHR  [20]  next 1_N(7:0)_oooo, N<-P
           [1010]  AC<- 1 shr AC, next 1
           [1020]  AC<- 2 shr AC, next 1
            ...
           [11F0]  AC<-31 shr AC, next 1
LDA' [30]  AC<-N, next 1, N<-P
LDA  [40]  H<-MEM(N)                           Datum lesen
     [41]  AC<-H, next 1, N<-P
LDA* [50]  H<-MEM(N)                            Adr. lesen
     [51]  N<-H
     [52]  H<-MEM(N)                           Datum lesen
     [53]  AC<-H, next 1, N<-P
STA  [60]  H<-AC,
     [61]  MEM(N):=H, next 1, N<-P       Datum schreiben
STA* [70]  H<-MEM(N)                            Adr. lesen
     [71]  N<-H, H<-AC
     [72]  MEM(N):=H, next 1, N<-P       Datum schreiben
AND  [80]  H<-MEM(N)
     [81]  AC<-AC.H, next 1, N<-P
ADD  [90]  H<-MEM(N)
     [91]  AC<-AC+H, next 1, N<-P
MUL  [A0]  BC<-AC, AC<-0, J<-'00000002'
     [A1]  H<-MEM(N), ST1(1)<-BC(0)     Multiplikatorbit
     [A2]  if ST1(1) then next A4
     [A3]  AC<-shr AC, ST2(1)<-AC(0)     Merke shift out
     [A4]  AC<-AC+H,   ST1(10)<-Carry(AC+H)
     [A5]  AC<-ST(10) inshr AC, ST2(1)<-AC(0)
     [A6]  BC<-ST2(1) inshr BC, J<-shl J, ST2(7)<-(J/=0)
     [A7]  ST1(1)<-BC(0), if ST2(7) then next A2
     [A8]  H<-AC, AC<-BC
     [A9]  MEM(N):=H
     [AA]  next 1, N<-P
IF>' [B0]  C<-1, ST1(5)<-(AC>N)
     [B1]  if ST1(5) then next 1, N<-P
     [B2]  C<-0, next 1
IF=' [C0]  C<-1, ST1(6)<-(AC=N)
     [C1]  if ST1(6) then next 1, N<-P
     [C2]  C<-0, next 1
GO'  [D0]  P<-N, next 1
GO   [E0]  H<-MEM(N)                   Sprungadresse lesen
     [E1]  P<-H, next 1
DO'  [F0]  ST1(7)<-(C/=0)
     [F1]  if ST1(7) then next 1, N<-P
     [F2]  N<-P+N, P<-P+N, next 1
```

Abbildung 6.11: Mikroprogramm, das die DINATOS-Architektur interpretiert

wird mit einer Maske geladen, die zum Ausblenden der Adresse dienen soll. Zustand `2`: Die 24-Bit-Adresse wird ausgeblendet, der Programmzähler wird um Eins erhöht, und es wird eine 256-fache Verzweigung zu den Zuständen `00, 10, ... F0` (hexadezimal) in Abhängigkeit vom Operationscode, der in `H(31:24)` steht, durchgeführt. Dadurch wird der Befehlscode in einem Schritt entschlüsselt. Liegt der Befehlscode `NOT` vor, dann wird der Akkumulatorinhalt im Zustand `10` negiert und zurück zum Zustand `1` gesprungen, um den nächsten Befehl zu holen. Liegt der Befehlscode `SHR` vor, dann wird im Zustand `20` nochmals in Abhängigkeit von der Anzahl der Schiftschritte `N(7:0)` zu den Zuständen `1010`, `1020`, ...

verzweigt, in denen der verlangte Schift durchgeführt wird. Die Ausführung des Befehls LDA′ besteht im Laden der Konstanten N in den Akkumulator (Zustand 30). – Die Ausführung des LDA-Befehls besteht aus dem Lesen der Speicherzelle MEM(N) in das Datenregister H und dem anschließenden Transport in den Akkumulator. – Zur Ausführung des LDA*-Befehls (Zustände 50-53) muß der Speicher zweimal gelesen werden. Der STA-Befehl erfordert zuerst den Transport H ← AC und anschließend das Schreiben in den Speicher. – Die Multiplikation MUL benutzt den Hilfsakkumulator BC. Der Multiplikator AC wird zuerst nach BC transportiert und das niedrigstwertige Bit (Multiplikatorbit) wird in ST1(1) gemerkt. Dann wird 32 mal der Multiplikationsschritt durchgeführt. Wenn das abgefragte Multiplikatorbit o ist, dann wird AC_BC nur nach rechts geschoben, und wenn es 1 ist, dann wird vorher noch der Multiplikand MEM(N) zum Akkumulator addiert. Zum Schluß wird der höherwertige Teil des Produkts in die Speicherzelle MEM(N) gebracht und der niederwertige Teil in den Akkumulator. Um die Anzahl der Multiplikationsschritte zu zählen, wird J mit lo geladen und solange nach links geschoben, bis J gleich Null wird. – Der Befehl IF>′ bewirkt einen Vergleich des Akkumulatorinhalts mit der Konstanten N und setzt gegebenenfalls C auf 1. – Bei dem Sprungbefehl G0′ wird der Befehlszähler mit N und bei G0 mit dem Inhalt von MEM(N) geladen. Die Bedingung C wird durch den DO′-Befehl abgefragt und bewirkt gegebenenfalls eine Erhöhung des Befehlszählers um N.

Die benutzte Methode zur Decodierung des Maschinenbefehlscodes ermöglicht nur Verzweigungen an Seitenanfänge k mal 16, wobei k=OPC der Wert eines beliebigen Bytes/Operationscodes im Befehl (der im Register R(Y2) steht) entspricht. Da die Anzahl der auszuführenden Mikrobefehle in Abhängigkeit vom Befehlscode stark variiert, ist die Größe einer Seite meist zu groß oder zu klein. Der freie Speicherplatz läßt sich aber nutzen, indem längere Mikrobefehlsfolgen die freigelassenen Speicherplätze benutzen.

Um an beliebig wählbare Adressen verzweigen zu können, könnte eine indirekte Decodierung über eine Sprungtabelle gewählt werden, die durch die Mikromaschine PIRI allerdings nicht unterstützt wird. Dabei wird zuerst in Abhängigkeit vom Befehlscode OPC=i an die i-te Stelle einer zusammenhängenden Tabelle gesprungen. In der Tabelle stehen Sprungziele zu beliebig definierten Stellen im Mikroprogrammspeicher, an denen die auszuführenden Mikroprogrammteile beginnen. Dem Vorteil des Verzweigens an beliebig wählbare Sprungadressen steht der Nachteil der zeitaufwendigeren zweischrittigen Decodierung gegenüber. Durch Verwendung einer zusätzlichen hardwaremäßig realisierten Sprungtabelle (PLA oder ROM) läßt sich bei freier Wahl der Sprungadressen die Decodierzeit reduzieren, wenn die Sprungtabelle rechtzeitig und parallel zur Ausführung der Mikrobefehle aktiviert wird. Bei Maschinenbefehlen mit variabler Wortlänge, die Teile des Befehlscodes in noch zu holenden Worten enthalten, ist man grundsätzlich auf eine sequentielle Decodierung angewiesen, so daß dadurch ein erhöhter Zeitaufwand entsteht.

Wenn das Mikroprogramm Teile enthält, die sich mehrfach wiederholen, könnten diese als Mikrounterprogramme implementiert werden, um Speicherplatz zu sparen. Dazu muß die Fortsetzungsadresse in ein spezielles Register geschrieben werden, bevor zum Unterprogramm ver-

zweigt wird. Nach Ausführung muß die gespeicherte Fortsetzungsadresse als aktuelle Folgeadresse ausgewählt werden, damit das Mikroprogramm regulär weitermachen kann. Da die Organisation des Unterprogrammsprungs und -rücksprungs einen zusätzlichen Zeitaufwand erfordert, kann auf die Unterprogrammtechnik auf der Mikroprogrammebene verzichtet werden. Deshalb wird die Unterprogrammtechnik in der Mikromachine PIRI auch nicht hardwaremäßig unterstüzt.

6.2.4 Bewertung der Lösungen

Die Anzahl der benötigten Ausführungszyklen bei der Emulation kann leicht aus dem Mikroprogramm entnommen werden. So benötigen `LDA'` drei, `LDA` vier, `LDA*` sechs, `GO'` drei und `GO` vier Zyklen. Dabei wurde die Zugriffszeit des Hauptspeichers und Mikroprogrammspeichers mit einem Zyklus angenommen. Die mittlere Ausführungszeit eines Maschinenbefehls beträgt etwa 4 Zyklen, wenn man die Multiplikation nicht berücksichtigt. Damit ist die mikroprogrammierte Lösung gegenüber der reinen Hardware-Lösung (Abschnitt 6.2.2) etwa um den Faktor 1,5 bis 2 mal langsamer. Dem steht der Vorteil der Flexibilität gegenüber, denn durch verschiedene Mikroprogramme lassen sich verschiedene Architekturen emulieren. Dieser Vorteil kommt dann zum Tragen, wenn die Architektur komplizierter ist, insbesondere, wenn Zwischensprachen für höhere Programmiersprachen interpretiert werden sollen.

Die mikroprogrammierte Lösung schneidet bei der Bewertung der Leistung günstiger ab, wenn die Zugriffszeit für den Mikroprogrammspeicher kleiner oder wesentlich kleiner als die des Hauptspeichers ist. Denn dann fällt die etwas höhere Anzahl von Interpretationsschritten gegenüber der reinen Hardware-Lösung kaum ins Gewicht. Ein weiterer Vorteil der mikroprogrammierten Lösung ist die Reduzierung der Speicherbandbreite für die Befehle. Denn höhere Leistungen erfordern breitere Befehle zur Steuerung parallel arbeitender Rechenwerke. Dieser erhöhte Steuerungsaufwand läßt sich auf die Mikroprogrammebene verlagern, wobei gleichzeitig die Maschinenbefehle in Richtung auf die Applikation/höhere Maschinensprache angehoben werden, wodurch sie kompakter und leistungsfähiger werden.

Will man eine besonders hohe Leistung erzielen, dann kann man auch die Applikation komplett als Mikroprogramm schreiben bzw. dahin übersetzen. Dabei entfällt die Maschinensprache als Zwischenschicht und der damit verbundene Interpretationsaufwand. Dadurch entsteht aber ein hoher Schreibaufwand bzw. Übersetzungsaufwand. Das Mikroprogramm übernimmt dann die Funktion des Maschinenprogramms und kann dann auch so bezeichnet werden. Man kannn solche (Mikro-)Maschinenbefehle auch als RISC-Befehle bezeichnen. Diese Ähnlichkeiten erkennt man auch, wenn man die Befehlsstrukturen von RISC-Befehlen (s. Kapitel 7) und Mikrobefehlen miteinander vergleicht.

Um noch höhere Leistungen in einem Rechner zu erzielen, muß man eine noch höhere Anzahl von Rechenwerken vorsehen, die für bestimmte Datentypen optimiert sein können. Dadurch entsteht aber ein höherer Steuerungsaufwand, d.h. die Befehle müssen breiter werden. Solche breiten Befehle werden als VLIW (very long instruction word) bezeichnet. Um die vielen

Rechenwerke optimal auszunutzen, muß ein hoher Übersetzungsaufwand betrieben werden. Sicherlich gibt es eine bestimmte Schranke, oberhalb derer sich der erhöhte Hardware- und Steuerungsaufwand nicht mehr auszahlt (oberhalb von ca. 64 bis 128 Bit VLIW-Befehlsbreite), weil dann die mittlere Auslastung zu schlecht wird. Eine entsprechende Aussage gilt für die Mikrobefehle, deren Breite nicht mehr als ca. 128 bis 256 Bit betragen sollte.

In diesem Zusammenhang ist auch der Begriff *Superskalar* (s. Abschnitt 7.2) zu erwähnen. Damit sind Maschinenbefehle gemeint, die mehr als eine Operation pro Zyklus ausführen können. Das ist an sich nichts Neues. In der Realisierung werden parallel arbeitende Rechenwerke benötigt, wie wir sie z.B. in der PIRI-Maschine benutzt haben. Auch die früheren und heutigen Hochleistungsrechner, z.B. die CDC 6600, besaßen bzw. besitzen eine größere Anzahl parallel arbeitender Rechenwerke/Funktionseinheiten. Dabei besteht auch die Möglichkeit, mehrere kurze Befehle zu einem *Superbefehl* zusammenzufassen und in einem Maschinenwort unterzubringen (z.B. vier 15-Bit-Befehle in einem 60-Bit-Maschinenwort bei der CDC 6600).

Literaturverzeichnis

[Anc] Anceau, F.: **The Architecture of Microprocessors.** Addison-Wesley, Workingham 1986

[Anl] Anlauff, H., Bierhals, H., Funk, P., Meinen, P.: *Registertransfersprache PHPL Sprachbeschreibung.* TUM-INFO-7720, Juni 1977, TU München

[Bae] Bähring, H.: **Mikrorechner-Systeme.** Springer, Berlin 1991

[Bar] Barron, D. W.: **Assembler und Lader.** Hanser, München 1970

[Bar] Barnes, G. et al.: *The Illiac IV Computer.* IEEE Trans. on Comp., Vol. C–17, No. 8, Aug. 1968

[Bar] Bartee, T. C.: *Computer Design of Multiple Logical Networks.* IEEE Trans. on Comp., Vol. EC–10, March 1960, pp. 21 – 30

[Bau-1] Bauer, F. L., Heinhold, J., Samelson, K., Sauer, R.: **Moderne Rechenanlagen.** Teubner, Stuttgart 1964

[Bau-2] Bauer, F. L., Goos, G.: **Informatik I.** Springer, Berlin 1971

[BaW] Baugh, C., Wooley, B.: *A Two's Complement Parallel Array Multiplication Algorithm.* IEEE Trans. on Computers, Vol. C–22, No. 12, Dec. 73

[Bel] Bell, G., Newell, A.: **Computer Structures: Readings and Examples.** McGraw-Hill, New York 1971

[Bla] Blaauw, G. A.: **Digital Systems Implementation.** Prentice Hall, Englewood Cliffs 1976

[Bln] Blankenship, P. E.: *Comments on ‚A Two's Complement Parallel Array Multiplication Algorithm'.* IEEE Trans. on Computers, Vol. C–23, No. 12, Dec. 74

[Bod] Bode. A. (Hrsg.): **RISC-Architekturen.** BI-Wiss.-Verl., Mannheim 1990

[Boo] Booth, A. D., Booth, K. H.: **Automatic Digital Calculators.** Academic Press Inc., New York 1956

[Bouk] Bouknight, W. et al.: *The Illiac IV System.* Proc. of the IEEE, Vol. 60, No. 4, April 1972

[Boul] Boulaye, G. G.: **Microprogramming.** Carl Hanser, München 1975 und Macmillan Press, London

[Bur] Burroughs: *B6700 Information Processing Systems Reference Manual.* Burroughs Corporation Detroit, Michigan 48232, 1970

[Bur] Burks, A. W., Goldstine, H. H., Neumann, J. von: *Preliminary Discussion of the Logical Design of an Electronic Computing.* Instrument. In [Bel, S. 92 – 119]

[Cas] Casaglia, F.: *Nanoprogramming vs. Microprogramming.* Computer, Vol. 9, No. 1, Jan. 1976

[Chr] Chroust, G.: **Mikroprogrammierung und Rechnerentwurf.** Oldenbourg, München 1989

[Chu-1] Chu, Y.: **Introduction to Computer Organization.** Prentice-Hall, Englewood Cliffs 1970

[Chu-2] Chu, Y.: **Digital Computer Design Fundamentals.** McGraw-Hill, New York 1962

[Chu-3] Chu, Y.: **Computer Organization and Microprogramming.** Prentice Hall, Englewood Cliffs 1972

[Cla] Clapp, J. A.: *The Application of Microprogramming Technology.* National Techn. Inf. Services AD 724718, 1971, pp. 7 – 11

[Dat] Data General Corporation: *Microprogramming with the ECLIPSE Computer.* WCS Feature, Technical Reference, No. 014–000050, 1974

[Dec] DEC: *Microcomputers and Memories.* Digital Equipment Corp., 1981

[Dol] Dolotta, T. A., McCluskey, E. J.: *The Coding of Internal States of Sequential Circuits.* IEEE Trans. Comp., Vol. EC–13, pp. 549 – 562, Oct. 1964

[Dul] Duley, J. R., Dietmayer, D. L.: *A Digital System Design Language (DDL).* IEEE Trans. on Computers, Vol. C–17, pp. 850 – 861, Sept. 1968

[Dür] Dürr, D.: *SL3 – Eine Systemprogrammiersprache auf Algol 68-Basis als Grundsprache für die Prozeßrechnerlinie AEG 80.* Angewandte Informatik, H. 9 (1975)

[Ell] Ellis, J.R.: **Bulldog: A Compiler for VLIW Architectures.** MIT Press, Cambridge Mass. 1985

[Fli] Flik, Th., Liebig, H.: **Mikroprozessortechnik.** Springer, Berlin 1990

[Gar] Gardner, P. L.: *Functional Memory and its Microprogramming Implications.* IEEE Trans. Comp., Vol. C–20, No. 7, July 1971

[Grd] Gardill, R., Händler, W., Hessing, H., Klar, R., Spies, P. P.: *ERES – Eine nichtprozedurale Rechnerentwurfssprache mit präziser Zeitbeschreibung.* Bericht IMMD, Universität Erlangen

[Gil] Giloi, W., Liebig, H.: **Logischer Entwurf digitaler Systeme.** Springer, Berlin 1980, 2. Auflage

[Glu-1] Glushkov, V. M.: *Automata Theory and formal Microprogramm Transformations.* Kibernetika, Vol. 1, No. 5, pp. 1 – 9, 1965

[Glu-2] Glushkov, Zeitlin, Justschenko: **Algebra, Sprachen, Programmierung.** Oldenbourg, München/Wien 1980

[Gon] Gonzales, R.: *A Multilayer Iterative Computer.* Trans. IEEE, EC-12, No. 5, pp. 781 – 790

[Gri] Grieshaber, R., Ulrich, G., Wendt, S.: *Petri-Netze zur Modellierung von Schaltwerksystemen.* Nachrichtentechn. Fachberichte Bd. 49, 1974, S. 150 – 155, VDE-Verlag

[Haa] Haas, G.: *Grundlagen und Bauelemente Elektronischer Ziffernrechenmaschinen.* Philips Technische Bibliothek, Eindhoven 1961

[Hal] Hallin, T. G., Flynn, M. J.: *Pipelining of Arithmetic Functions.* IEEE Trans. on Computers, Aug. 1972

[Har] Hartenstein, R. W., Liell, Schaaf, Weber: *CHARLES – A Register Transfer Language for Hardware Design and Specification.* Bericht 45/81, Univ. Kaiserslautern, Fachbereich Informatik, Dez. 1981

[Has] Hashizume, B., Johnson, W. N.: *The LSI-11/23 Control Store Microarchitecture.* IEEE 1979

[HDL] *Hardware Description Languages.* Computer, Vol. 7, No. 12, Dec. 1974

[Hil] Hill, F. J., Peterson, G. R.: **Digital Systems: Hardware Organization and Design.** John Wiley, New York 1973

[Hin] Hintz, R. G., Tate, D. P.: *Control Data STAR-100 Processor Design.* COMPCON '72 Digest, pp. 1 – 4

[Hof-1] Hoffmann, R.: *Towards Microprogrammable Machines based on a regularly structured processing unit.* In: *Firmware, Microprogramming and Restructurable Hardware,* Chroust, G. and Mühlbacher, J. G., Editors. IFIP Linz. North Holland Amsterdam 1980

[Hof-2] Hoffmann, R.: *The Hardware Description and Programming Language HDL (Preliminary Def.).* Bericht 75-04, FB 20, Techn. Unversität Berlin 1975

[Hof-3] Hoffmann, R.: *Die Hardware-Beschreibungssprache HDL.* Bericht MPR 2/81, FG Mikroprogrammierung, FB20, TH Darmstadt Dez. 1981

[Hof-4] Hoffmann, R.: *Grundlegende Algorithmen des Rechenwerks.* In: *Funktioneller Entwurf kleiner bis mittlerer Digitalrechner, Brennpunkt Kybernetik.* Technische Universität Berlin 1971

[Hof-5] Hoffmann, R.: *Algorithmen mit booleschen Matrizen in der Schaltwerkstheorie.* Dissertation, Technische Universität Berlin 1974

[Hof-6] Hoffmann, R.: **Rechenwerke und Mikroprogrammierung.** Oldenbourg, München 1978, 1983

[Hol] Holland, J.: *A Universal Computer Capable of Executing an Arbitrary of Subprograms Simultaneously.* Proc. East Joint Comp. Conf. 1959, pp. 108 – 113

[Hor] Horn, K.: *Verfahren zum Top-Down-Entwurf digitaler Systeme mit horizontalem Mikroprogrammwerk.* Dissertation, Technische Universität Berlin FB 20,1978

[Hrs] Horster, P., Manstetten, D., Pelzer, H.: **RISC.** Hüthig, Heidelberg 1987

[Hus] Husson, S. S.: **Mircoprogramming: Principles and Practices.** Prentice Hall, Englewood Cliffs 1970

[Iee] IEEE: **Standard VHDL Language Reference Manual.** New York 1988

[Int] Interdata: *Model 80 Micro-Instruction Reference Manual.* Publ. No. 29–282RO1. 1973

[Ito] Ito, T.: *A Theory of Formal Microprogramms.* Editors: Boulaye, G., Mermet, J. Int. Advanced Summer Inst. on Microprogramming, pp. 257 – 280, 1971.

[Ive] Iverson, K.: **A Programming Language.** John Wiley, New York 1962

[Jae] Jaeger, R.: *Microprogramming: A General Design Tool.* Computer Design, Aug. 1974, pp. 150 – 157

[Jan] Janow, J. I.: *Probleme der Kybernetik.* Bd. 1, S. 87 – 143, Akademie-Verlag, Berlin 1967

[Jes] Jessen, E.: **Architektur digitaler Rechenanlagen.** Springer, Berlin 1975

[Jon] Jones, H. L. et al.: *An annotated Bibliography on Microprogramming.* 5. Annual Workshop on Microprogramming IEEE Comp. Society, Dec. 1973, pp. 51 – 60

[Jum] Jump, J. R.: *Asynchronous Control Arrays.* IEEE Trans. on Computers, Vol. C–23, No. 10, Oct. 1974

[Kan] Kane, G., Heinrich, J.: **MIPS RISC Architecture.** Prentice Hall, Englewood Cliffs, N.J. 1992

[Kat] Katevenis, M.G.H.: **Reduced Instruction Set Computer Architectures.** MIT Press, Cambridge Mass. 1984, 1986

[Kie] Kieáling, I., Lowes, M., Paulik, A.: **Genaue Rechnerarithmetik-Intervallrechnung und Programmierung mit PASCAL-SC.** Teubner, Stuttgart, 1988

[Knu] Knudson, M. J.: *PMSL, An Interactive Language for System Level Description and Analysis of Computer Structures.* Techn. Report AD 762 513, Carnegie-Mellon University, Dept. Comp., 1972

[Laz] Lazov, V.: *Hardware-Realisierung Strukturierter Mikroprogramme.* Dissertation, TU Berlin, FB 20, 1976

[Lie] Liebig, H., Flik, Th.: **Rechnerorganisation.** Springer, Berlin 1993

[Lja] Ljapunov, A. A.: *Probleme der Kybernetik.* Bd. 1, S. 1 – 22, 53 – 86, Akademie Verlag Berlin 1962

[LSI] *LSI-11 WCS User's Guide.* Digital Equipment Corp., Maynard, Mass. 1978

[Mal] Mallach, E. G.: *Emulator Architecture.* Computer, Aug. 1975, Vol. 8, pp. 24 – 31

[Mea] Mealy, G.H.: *A Method on Synthesizing Sequential Circuits.* Bell Syst. Techn. J. 34 (1955), pp. 1045-1048

[Men] Menche, R.T., Schmitt, A.S.: *REGLAN Language Reference Manual.* Institutsbericht RO 89/6, TH Darmstadt FB19, 1992

[Mer] Mermet, J.: *Definition du langage CASSANDRE.* These Doctor Ingenieur, IMAG, Grenoble, 1970

[Mic] Microdata Corporation: *Microprogramming Handbook.* Santa Ana, California 1971

[Mil] Miller, R. E.: *A Comparison of Some Theoretical Models of Parallel Computation.* IEEE Trans. on Computers, Vol. C–22, No. 8, Aug. 1973

[Moo] Moore, E.F.: *Gedanken-Experiments on Sequential Machines.* Automata Studies, Annals of Math. Studies 34, Princeton Univ. Press, 1956

[Mor] Morris, R., Miller, J.: *Designing with TTL Integrated Circuits.* Texas Instruments Electr. Series, McGraw Hill, New York 1971

[Mot] Motorola Ltd.: *MC88110 Second Generation RISC Microprocessor User's Manual* 1993

[Mue] Müller-Schloer, C., Schnitter, E. (Hrsg.): **RISC-Workstation-Architekturen.** Springer, Berlin 1991

[Opl] Opler, A.: *Fourth Generation Software.* Datamation, 13, 1, Jan. 1967, pp. 22 – 24

[Pat] Patil, S. S.: *Coordination of asynchronous events.* PH. D. dissertation, Dep. Elec. Eng., Massachusetts Institute of Technology, Cambridge 1970

[Pil-1] Piloty, R.: *RTS I, Registertransfersprache.* 3. Auflage, Institut für Nachrichtenverarbeitung, TH Darmstadt 1969

[Pil-2] Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F., Skelly, P.: *CONLAN-Report.* Lecture Notes in Computer Science, Springer, Berlin 1983

[Poe] Poel, W. L. van der: *The Essential Types of Operations in an Automatic Computer.* Nachrichtentechn. Fachberichte NTF, Band 4, S. 144 – 145, 1955

[Pos] Pospelov, D. A.: **Rechnersysteme.** BSB B. G. Teubner Verlagsgesellschaft, Leipzig 1975

[Ram] Ramamoorthy, C. V., Tsuchiya, M.: *A High-Level Language for Horizontal Microprogramming.* IEEE Trans. On Computers, Vol. C–23, No. 8, Aug. 1974

[Rhy] Rhyne, V. T.: *A Simple Postcorrection for Nonrestoring Division.* IEEE Trans. on Computers, Feb. 1971

[Ric] Richards, R. K.: *Arithmetic Operations in Digital Computers.* Van Nostrand, New York 1955

[Rob] Robertson, J. E.: *Two's Complement Multiplication in Binary Parallel Digital Computers.* IRE Trans. on Electronic Computers, Sept. 1955, pp. 119 – 120

[Rud] Rudolph, J.: *A production implementation of an associative array processor-STARAN.* Proc. Fall Joint Computer Conf., 1972, pp. 229 – 241

[Sal] Salisbury, Alan B.: **Microprogrammable Computer Architectures.** Elsevier New York/Oxford/Amsterdam 1976, pp.149 – 152

[Sch] Schünemann, C.: *Mikro- und Pico-Programmspeicher.* In: Hasselmeier und Spruth: *Rechnerstrukturen.* Oldenbourg, München 1974, S. 36 – 74

[Spa] Spaniol, O.: **Arithmetik in Rechenanlagen.** Teubner Studienbücher Informatik

[Spe] Speiser, A. P.: **Digitale Rechenanlagen.** Springer, Berlin 1980, 2. Auflage

[Sta] Stabler, E. P.: *Microprogramm Transformations.* IEEE Trans. on Computers, Vol. C–19, No. 10, Oct. 1970

[Sta] Starke, P.H.: **Abstrakte Automaten.** VEB Deutscher Verlag der Wissenschaften, Berlin 1969

[Ste-1] Stein, M. L., Munro, W. D.: **Introduction to Machine Arithmetic.** Addison-Wesley, Reading, Mass. 1971

[Ste-2] Stein, L. M., Munro, W. D.: *On Complement Division.* Comm. of the ACM, April 1971

[Str] Straßer, W.: *Schnelle Kurven- und Flächendarstellung auf grafischen Sichtgeräten.* Dissertation, Technische Universität Berlin 1974

[Stü] Stürz, H., Cimander, W.: **Automaten und Anwendung in der digitalen Schaltungstechnik.** VEB Verlag Technik, Berlin 1974

[Sun-1] Catenzaro, B.J. (Editor): **The SPARC Technical Papers.** Springer, New York 1991

[Sun-2] Sun Microsystems Comp. Corp.: *The SuperSPARC Microprocessor.* Technical White Paper, Mountain View 1992

[Tho] Thornton, J.E.: **Design of a Computer, The Control Data 6600.** Scott, Foresman and Company, Glenview Illinois 1970

[Tuc] Tucker, A. B., Flynn, M. J.: *Dynamic Microprogramming: Processor Organization and Programming.* Comm. ACM, April 1971, Vol. 14, No. 4

[Wal] Wallace, C. S.: *A Suggestion for a fast Multiplier.* IEEE-EC 13 (1964) 14 – 17

[Wat] Watson, W. J.: *The TI ASC – A highly modular and flexible supercomputer architecture.* Proc. Fall Joint Comp. Conf. 1972, pp. 221 – 228

[Wen-1] Wendt, S.: **Entwurf komplexer Schaltwerke.** Springer, Berlin 1974

[Wen-2] Wendt, S.: **Nichtphysikalische Grundlagen der Informationstechnik.** Springer, Berlin 1989

[Wen-3] Wendt, S.: *Zur Systematik von Mikroprogrammwerkstrukturen.* Elektron. Rechenanlagen 13 (1971), 1

[Wil-1] Wilkes, M. V.: *The Best Way to Design an Automatic Calculating Machine.* Manchester Univ. Computer Inaugural Conf. 1951

[Wil-2] Wilkes, M. V., Stringer, J. B.: *Microprogramming and the design of the control circuits in an electronic digital computer.* Proc. Cambridge Phil. Soc., Vol. 49, pp. 230 – 238, April 1953

[Zem] Zemanek, H.: **Das geistige Umfeld der Informationstechnik.** Springer, Berlin 1992

Index

Addierer mit
Übertragsweiterleitung 66
Übertragsvorausberechnung 67
Addition 66
Doppelwort- 84
im 1-Komplement 82
im 2-Komplement 79
serielle 72
von BCD-Zahlen 84
von Gleitkommazahlen 88
VON NEUMANNsche 70
von Vorzeichenzahlen 77
Adressierung
indirekte **??**
relative **??**
ALU
Architektur **??**
Funktionelle- **??**
Register-Transfer- **??**
Außen- **??**
Innen- **??**
arithmetische Erweiterung 18
Ausführungszeit 141, 150, 166, **??**, **??**, **??**
Automaten 49

Bearbeitungsdauer **??**
Belegungsschema **??**
Bedingung **??**
Befehlspuffer **??**
Befehlstypen **??**, 140
binärcodierte Dezimalzahl 20
boole 36
BOOTH-Algorithmus 100
BURKS-GOLDSTINE-VON-NEUMANN-Methode 96
Bottom-Up-Methode 136, 148
Branch-Target-Cache **??**
Bypass **??**

Cache **??**
Sprungziel- **??**
-Kohärenz **??**, **??**
Carry → Übertrag 80, 82
-Generator 70
-Save-Addierer 74
Carry-Look-Ahead 67
Codierung **??**, **??**
analytische **??**
horizontale **??**
vertikale **??**

Datenabhängigkeit **??**
Datentyprechenwerk **??**
Decodiermatrix **??**
Delayed Branch **??**, **??**
Delayed Instruction **??**
Delayed Slot **??**
Demultiplexen **??**
Direct-Mapping-Cache **??**
Division 111
im 2-Komplement 117
mit Rückstellen 112
ohne Rückstellen 113
parallele 124

von BCD-Zahlen 124
von Gleitkommazahlen 127
von Vorzeichenzahlen 113
Dualmatrix **??**
dualer Wert 16

Einskomplementzahl 19
Einserkorrektur 82
Emulation **??**, **??**, 163
Entwurfsaufgabe 134

Firmware **??**
Folgeadresse **??**
Forwarding **??**
Functional Memory **??**
Funktionelle-Architektur **??**
Funktioneller Entwurf 32
Funktionseinheit **??**

Gleitkommazahl 22
Gliederung **??**

Halbaddierer 66
Hardware-Algorithmus 65
Hardware-Beschreibungssprache 31, 32
Hardware-Steuerwerk **??**
Harvard-Architektur 162
HDL 31
horizontale Codierung **??**
History-Buffer **??**

Indirekte Adressierung **??**
Interpretationshierarchie **??**
Interpretationschema **??**
Interpreter **??**
Interpreterprogramm **??**
Interpretierbares Programm **??**
interpretieren **??**

JK-Flipflop-Steuerwerk **??**

Keller-Rechenwerk **??**
Komplement 18
Eins- 19
Neun- 21
Zehn- 21
Zwei- 17
Komposition **??**
Konsistenz **??**
Konflikte **??**
Konvertierung 128

Latenzzeit **??**
Logik-Ebene 31

Matrix-Steuerwerk **??**
MEALY-Automat **??**
MEALY-Mikroprogramm-Steuerwerk **??**
Memory Managment Unit MMU **??**, **??**
Mikroalgorithmus 65
Mikrobefehl **??**, **??**, 138, 154
symbolischer 154
horizontaler **??**
vertikaler **??**
-Operationswerk **??**
-Steuerwerk **??**
Mikrocode **??**
Mikroentscheidung
Mikromaschine 154
Mikrooperation **??**, **??**
asynchrone **??**
erzeugende **??**
synchrone **??**
Mikroprogramm **??**, **??**
abstraktes 54
paralleles 55
reguläres **??**
synchrones 55
Transformation **??**
Übersetzung **??**
Mikroprogramm-Analysator
Mikroprogrammierbarer Rechner **??**, **??**
Mikroprogrammierung **??**
Mikroprogramm-Steuerwerk **??**
der Data General
der IBM 360/40
der Interdata M80
der Microdata 1600

Entwurf eines **??**
Gliederung **??**
MEALY- **??**
MOORE- **??**
Minimierung des Speichers **??**
MOORE-Automat **??**
Multiplexen 41, **??**
Multiplikation 90
Doppelwort- 107
im 2-Komplement 94
nach BOOTH 100, 106
nach ROBERTSON 98, 100
parallele 103
serien-parallele 92
von BCD-Zahlen 109
von Gleitkommazahlen 110
von Vorzeichenzahlen 93

Nanooperation **??**, **??**
Nanoprogramm-Steuerwerk **??**
Normalisieren 24

Objektmenge 145
Implementierungs- 147
Operationswerk **??**
Operatoren in HDL 40
Overflow → Überlauf 79, 82

Picooperation **??**
Pipeline
lineare **??**
multifunktionale **??**
unifunktionale **??**
Pipelining **??**, **??**, **??**, **??**
arithmetisches **??**
Mikrobefehls- **??**
Befehls- **??**
Precise Exception **??**, **??**
Programmiermodell 145
Prozessor **??**
PLA **??**, **??**, **??**

Rechenwerk **??**
Rechnerentwurf 133, 136, 141, **??**, **??**
Rechnertypen 144
Redundanz **??**
horizontale **??**
relative **??**
vertikale **??**, **??**
register 36
Register-Transfer 31
-Architektur **??**, 158, 161
-Ebene 31
-Fenster **??**, **??**
ROBERTSON-Methode
Rundung 29

Schaltfunktion **??**
Schaltmatrix **??**, **??**
Schichtenmodell **??**
Schritt-Steuerwerk **??**
Schutzstelle 81
Scoreboard **??**, **??**
Semantische Lücke **??**, **??**
Signed Integer → Zweikomplementzahl
Snooping **??**, **??**
Software-Maschine **??**
Sprungziel-Cache **??**
Stack-Rechenwerk **??**
Steueralgorithmus 65, **??**, **??**
Steuerbefehl **??**
Steueroperationssystem **??**, 137
Steuerwerk **??**
Hardware- **??**
Mikroprogramm- **??**, **??**, **??**
Subtraktion 77
Superpipeline **??**, **??**
Superskalar **??**
Superbefehl 167

Target Instruction Cache **??**
Top-Down-Methode 136, 148
Transformation von Automaten **??**
Translation Look Aside Buffer **??**

Übergangstabelle
disjunktive **??**
konjunktive **??**

Überlauf 29
Übertrag 67
Underflow → Unterlauf
Unterlauf 29

Verbindungsnetz **??**, **??**
Verlagerung **??**
vertikale Codierung **??**
Verweildauer **??**
Virtuelle Adressierung **??**
Virtuelle Maschine **??**, **??**
VLIW 166, **??**
Volladdierer 66
Vorzeichenzahl 17

Write-Back **??**
Write-Through **??**

Zahl
- abstrakte 15
- b-näre 15
- konkrete 15
- Wert einer 15

Zahlendarstellung 15
Zahlenkonvertierung 128
Zeitanteil 141, **??**
Zugriffsschema **??**
Zweikomplementzahl 17