

LEHRBUCH

Frank Slomka · Michael Glaß

Grundlagen der Rechnerarchitektur

Von der Schaltung zum Prozessor



Springer Vieweg

Grundlagen der Rechnerarchitektur

Frank Slomka · Michael Glaß

Grundlagen der Rechnerarchitektur

Von der Schaltung zum Prozessor



Springer Vieweg

Frank Slomka
Institut für Eingebettete Systeme
Universität Ulm
Ulm, Deutschland

Michael Glaß
Institut für Eingebettete Systeme
Universität Ulm
Ulm, Deutschland

ISBN 978-3-658-36658-2 ISBN 978-3-658-36659-9 (eBook)
<https://doi.org/10.1007/978-3-658-36659-9>

Die Deutsche Nationalbibliothek verzeichnetet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über ► <http://dnb.d-nb.de> abrufbar.

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2023

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Aus Gründen der besseren Lesbarkeit verwenden wir in diesem Buch überwiegend das generische Maskulinum. Dies impliziert immer beide Formen, schließt also die weibliche Form mit ein.

Planung/Lektorat: Leonardo Milla
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Liebe Leser,

Das Schreiben dieses Buches war ein Abenteuer. Ursprünglich sollte es als Skript zu einer einführenden Vorlesung in die technische Informatik und die Rechnerarchitektur an der Universität Ulm dienen. Es zeigte sich jedoch schnell, dass das Gebiet tiefgründiger ist als bei der Übernahme einer bestehenden Vorlesung im Jahr 2015 vermutet. Dieser Erkenntnis entsprang der Wunsch, ein Skript zu erstellen, das mehr ist als eine Aneinanderreihung technischer Konzepte. Seit der Berufung von Michael Glaß im Jahr 2016 an die Universität Ulm, halten beide Autoren im jährlichen Wechsel die Vorlesung und überarbeiteten das von Frank Slomka erstellte Skript zu dem vorliegenden Lehrbuch.

Die Autoren haben in vielen einführenden Lehrveranstaltungen die Erfahrung gemacht, dass viele Studenten der Informatik ein Studium in der Annahme beginnen, in ihrem späteren Berufsleben Anwendungen zu programmieren. Als angehende Informatiker sind sie daher oft an Programmiersprachen und Softwaretechnik interessiert. Dabei unterschätzen sie jedoch den Einfluss der technischen Realisierung von Rechnern auf das Gesamtsystem aus Hardware und Software. Als Beispiel seien hier parallele und spezialisierte Architekturen wie Grafikkarten genannt, die den derzeitigen Siegeszeug der künstlichen Intelligenz oder der komplexen Bildverarbeitung erst ermöglicht haben. Die leichte Verfügbarkeit leistungsfähiger programmierbarer Logikschaltungen erlaubt darüber hinaus jedem Informatiker, Ingenieur oder Physiker seine eigene integrierte Schaltung mit vollwertigem Prozessor und peripherer Logik selbst zu bauen oder zu programmieren. Zwar unterstützen moderne Entwurfswerkzeuge den Algorithmenentwickler bei der Umsetzung in Hardware, trotzdem erfordert die Auswahl der Algorithmen und die Bedienung dieser Werkzeuge ein grundlegendes Verständnis der Prinzipien und Konzepte des digitalen Schaltungsentwurfs und der Rechnerarchitektur. Diese Grundlagen soll das vorliegende Buch vermitteln.

Darüber hinaus stieg in den vergangenen 40 Jahren der Anteil der Computer in anderen technischen Feldern sprunghaft an. Es gibt kein Flugzeug und kein Kraftfahrzeug, das nicht mehrere Rechner an Bord hat; keine Telekommunikation und kein Fernsehen ohne Computerkomponenten. Viele Programmierer, Informatiker und Ingenieure arbeiten in Deutschland in klassischen Industrien wie dem Maschinen- und Fahrzeugbau. In diesen Branchen spielen Computer, die in Maschinen und Fahrzeuge eingebettet sind, mittlerweile eine entscheidende Rolle in der Wertschöpfung. Allerdings muss bei diesen eingebetteten Rechnern die Software häufig aus Kostengründen an die zur Verfügung stehende Hardware angepasst werden. Wissen über die Technologie und Architektur eines Rechensystems ist daher für jeden Informatiker unerlässlich, insbesondere, wenn er später in der Fahrzeug- oder Maschinenbauindustrie tätig sein wird.

Das Buch ist insbesondere zur Begleitung einer Vorlesung in den ersten Semestern geeignet. Oft werden Stoffgebiete ausführlicher behandelt, als dies in der Vorlesung möglich ist. Normalerweise sind Studenten angehalten, sich derartige Zusatzinformationen selbst zu erarbeiten. Dieser Weg wird mit dem Werk bewusst nicht gegangen. Im Gegenteil: Die Ausführlichkeit soll zeigen, wie intensiv sich ein Stoffgebiet durchdringen lässt. Dies soll dazu anregen, weiterführende Vorlesungen – insbesondere in

späteren Semestern – durch eine eigene, tiefer gehende Literaturarbeit zu ergänzen und aktiv zu studieren.

Die Autoren danken den Mitarbeitern des Instituts für Eingebettete Systeme/Echtzeitsysteme und zahlreichen Studenten vergangener Lehrveranstaltungen für ihre konstruktive Kritik. Nicht alle können genannt werden. Besonderer Dank gilt den langjährigen Betreuern der Vorlesung Nico Rösner und Marcel Rieß sowie den Studenten Nadine Gerstenlauer, Julian Bestler und Alexander Janus. Ganz besonders sei an dieser Stelle Frau Barbara Porada, unserer langjährigen Sekretärin, gedankt. Frau Porada hat sich intensiv bemüht für Material, das nicht von den Autoren stammt, die erforderlichen Rechte einzuholen.

Frank Slomka
Michael Glaß

Einleitung

Das Internet ist heute in aller Munde. Wenn man mittlerweile über Computertechnik spricht, fallen den meisten schillernde und bunte Internet-start-up-Unternehmen ein, die Software für Kommunikation, soziale Interaktion und Unterhaltung anbieten. Dabei ist uns oftmals nicht bewusst, was diese digitale Umwälzung im Grunde erst ermöglicht. Die Integration einer millionenfachen Anzahl von elektronischen Schaltern auf einem einzigen, nur wenige Quadratzentimeter großen Siliziumstück und deren geschickte Anordnung zu vollständigen Rechenmaschinen führte zu einer technischen Revolution. Grundlage aller heutigen digitalen Rechner, vom Smartphone oder dem Personal Computer bis hin zu Supercomputern ist die monolithische Herstellung von Transistoren auf einem Chip. Ausgehend vom elektronischen Rechnen führt dieses Buch in die Rechnerarchitektur ein. Der Schwerpunkt liegt dabei auf den technischen Fundamenten.

Beginnend bei unterschiedlichen Zahlensystemen und der Möglichkeit, mit diesen Rechenmaschinen zu bauen, werden die mathematischen Grundlagen zur Konstruktion digitaler Rechner eingeführt. Die Realisierung dieser Konzepte in einer Maschine erfordert jedoch eine Einführung in die Schaltungstechnik und die Herstellung integrierter Schaltungen. Dies scheint insbesondere notwendig, da die zugrunde liegenden Herstellungsverfahren stets Einfluss auf die Struktur und Architektur des Rechners haben. Sind die technischen Komponenten zum Aufbau logischer Schaltungen, die Gatter erst einmal definiert, wird vom elektrischen Verhalten dieser Bausteine abstrahiert. Es folgt die Beschreibung der grundlegenden Elemente zur Konstruktion von Rechenanlagen. Mit diesen lassen sich dann logische Schaltungen zum Speichern und Rechnen konstruieren. Dabei geht es um einfache arithmetische Operationen und um Bausteine, die den Datenfluss zwischen diesen beeinflussen. Mithilfe arithmetischer Strukturen und den Speichern wird anschließend die Architektur einer frei programmierbaren Rechenmaschine zum Rechnen und Verarbeiten von Daten, wie wir sie heute kennen, konstruiert. Zuletzt werden diese Rechner dann programmiert. Dies erfolgt direkt auf der Maschinenebene, wobei die Schnittstelle zu höheren Programmiersprachen ebenfalls beleuchtet wird. Die Idee dieses Buches ist, zunächst, den Bau von Rechenmaschinen zu motivieren, um dann auf einem festen Fundament sowohl mathematischer Methoden als auch technischer Grundlagen integrierter Schaltungen in die systematische Konstruktion von Computern einzuführen.

Die detaillierte Beschreibung der grundlegenden technischen Basistechniken scheinen dem Informatiker zunehmend fremd zu sein. In den vergangenen 40 Jahren wurde häufig argumentiert, Informatiker in Deutschland würden keine Computer bauen. Es reiche daher, sie mit der Architektur eines Rechners insoweit vertraut zu machen, wie es zur Programmierung dieser Maschinen notwendig sei. Seit etwas mehr als 20 Jahren gibt es leistungsfähige programmierbare Hardware, die es jedem Informatiker erlaubt, zeitkritische Anwendungen direkt in einer logischen Schaltung zu realisieren. Andere rechenintensive Probleme werden mittlerweile auf speziellen Grafikprozessoren gerechnet. Der automatische Entwurf digitaler Schaltungen und die Implementierung dieser mit programmierbarer Logik lassen die technische Informatik wieder zu ihren Wurzeln zurückkehren. Und es ist

notwendiger denn je, Studenten der Informatik und angrenzender Disziplinen wie der Elektrotechnik, der Informationssystemtechnik, der Nachrichtentechnik oder der Softwaretechnik gründlich in diesen Techniken auszubilden. Auch wenn der Schwerpunkt dieses Buches auf den Grundlagen der digitalen Eigenschaften von Rechnerarchitekturen liegt, so kann es den interessierten Leser motivieren, sich weiterführend mit den Schaltungstechniken und Herstellungstechnologien zu beschäftigen.

Inhaltsverzeichnis

1	Historische Entwicklung von Rechenmaschinen	1
1.1	Mechanische Rechner	3
1.2	Elektrische Rechner	11
1.3	Erste elektronische Rechner.....	15
1.3.1	Röhrencomputer	15
1.3.2	Halbleitercomputer	18
1.4	Rechner in der Datenverarbeitung	20
1.4.1	Großrechner von IBM.....	21
1.4.2	Minicomputer von DEC	24
1.5	Die Zahlenfresser: Supercomputer	25
1.6	Mikrocomputer	29
1.7	Eingebettete Computer	32
1.8	Halbleiter und die Entwicklung der Rechenleistung.....	36
	Weiterführende Literatur.....	39
2	Arithmetik: Zahlen und Rechnen	41
2.1	Natürliche Zahlen	43
2.1.1	Darstellung der Zahlen	44
2.1.1.1	Additives Zahlensystem	45
2.1.1.2	Polyadisches Zahlensystem	47
2.1.2	Zahlensysteme.....	50
2.1.3	Arithmetik der natürlichen Zahlen im polyadischen Zahlensystem	55
2.1.4	Codierung von natürlichen Zahlen im Rechner	68
2.1.4.1	Ziffernreihenfolge und Wertebereich.....	68
2.1.4.2	Rechnen im Dezimalsystem.....	70
2.1.4.3	Rechnen im Dualsystem	76
2.2	Ganze Zahlen	79
2.2.1	Darstellung	79
2.2.2	Arithmetik der ganzen Zahlen.....	79
2.2.3	Codierung der ganzen Zahlen im Rechner	81
2.2.3.1	Vorzeichenbehaftete Zahlen	82
2.2.3.2	Dualzahlen im 1-Komplement.....	83
2.2.3.3	Dualzahlen im 2-Komplement.....	86
2.3	Reelle Zahlen.....	89
2.3.1	Dezimalbrüche.....	91
2.3.1.1	Dezimalbruchentwicklung	92
2.3.1.2	Abkürzende Schreibweise von Dezimalbrüchen	94
2.3.2	Kettenbrüche	96
2.3.3	Dezimal- und Kettenbrüche irrationaler Zahlen	98
2.3.4	Maschinenzahlen	104
2.3.4.1	Darstellung	104
2.3.4.2	Numerik	107
2.3.4.3	Binäre Gleitkommazahlen.....	112
2.3.4.4	Binäre Festkommazahlen	120

2.4	Übertragung und alphanumerische Zeichendarstellung	125
	Weiterführende Literatur.....	133
3	Mathematische Grundlagen digitaler Schaltungen	135
3.1	Algebraische Betrachtung digitaler Schaltungen.....	137
3.2	Boolesche Algebra	139
3.2.1	Mathematische und strukturelle Grundlagen	139
3.2.2	Definition und grundlegende Eigenschaften boolescher Algebren.....	144
3.2.3	Rechnen in der booleschen Algebra.....	149
3.2.4	Ausdrücke und Funktionen	152
3.2.5	Axiomatisierung der booleschen Algebra.....	158
3.3	Schaltalgebra und Schaltfunktionen	160
3.3.1	Modellierung von Telefonnetzen.....	160
3.3.2	Formale Definition und Eigenschaften.....	162
3.3.3	Anwendung von Schaltfunktionen	165
3.3.4	Minimierung von Schaltfunktionen.....	168
3.3.4.1	Grafische Minimierung.....	172
3.3.4.2	Algebraische Minimierung.....	180
	Weiterführende Literatur.....	186
4	Digitale Schaltungen	187
4.1	Schaltungstechnik	189
4.1.1	Elektrotechnische Grundlagen	190
4.1.1.1	Spannung und Strom.....	190
4.1.1.2	Elektrische Netzwerke	193
4.1.1.3	Logische Signale	197
4.1.2	Festkörperelektronik	198
4.1.2.1	Schalter in Festkörpern	199
4.1.2.2	Halbleiter	201
4.1.2.3	Halbleitergrenzflächen.....	208
4.1.2.4	Der Transistor	212
4.1.2.5	Der Planarprozess zur Herstellung integrierter Schaltungen.....	217
4.2	Logische Gatter in MOS-Technik	220
4.2.1	NMOS- und PMOS-Inverter.....	222
4.2.2	CMOS-Inverter	225
4.2.3	CMOS-NAND- und -NOR-Gatter	234
4.3	Flipflops und Bitspeicher	237
4.3.1	Flipflops	237
4.3.1.1	Rückgekoppelte Schaltungen	237
4.3.1.2	RS-Flipflop	239
4.3.1.3	D-Flipflop	247
4.3.2	Transistorzellen	249
4.4	Speichermatrizen	256
4.5	Logikfelder	261
4.5.1	Standardzellen.....	266
4.5.2	Maskenprogrammierbare Logik.....	269
4.5.2.1	Gatterfelder (Gate-Arrays).....	269
4.5.2.2	Logisches Feld (PLA).....	271

4.5.3	Speicherprogrammierbare Logik.....	277
4.5.3.1	PLAs aus Speichern	279
4.5.3.2	Speicherprogrammierbare Gatterfelder (FPGA).....	281
	Weiterführende Literatur.....	286
5	Elemente der Rechnerarchitektur.....	287
5.1	Kombinatorische Schaltungen	289
5.1.1	Codierer- und Decodierschaltungen.....	289
5.1.2	Datenverteiler.....	292
5.2	Sequenzielle Schaltungen.....	294
5.2.1	Schieberegister	294
5.2.2	Zähler	296
5.2.3	Steuerwerke	297
5.2.3.1	Automaten.....	298
5.2.3.2	Maskenprogrammierbares Steuerwerk.....	301
5.2.3.3	Speicherprogrammierbares Steuerwerk.....	306
5.3	Arithmetische Schaltungen	309
5.3.1	Addierwerk	309
5.3.1.1	Serielles Addierwerk (Ripple-carry-Addierer)	309
5.3.1.2	Paralleles Addierwerk („carry look-ahead adder“)	311
5.3.1.3	Teilparalleles Addierwerk („carry-select adder“)	313
5.3.2	Addier- und Subtraktionswerk	314
5.3.3	Arithmetisch-logische Einheit (ALU)	315
5.4	Speicher.....	317
5.4.1	Stapel	317
5.4.2	Warteschlange	319
5.4.3	Speicher mit direktem Zugriff.....	320
5.4.4	Assoziativspeicher.....	325
	Weiterführende Literatur.....	329
6	Rechnerarchitektur	331
6.1	Architektur des Befehlssatzes	337
6.1.1	Befehl und Programm	337
6.1.2	Die Architektur und ihre Auswirkung auf den Befehl.....	343
6.1.3	Befehlsgruppen.....	348
6.1.4	Operandenzugriff und Adressierung	353
6.1.4.1	Inhärente Adressierung	355
6.1.4.2	Implizite oder unmittelbare Adressierung.....	357
6.1.4.3	Direkte oder absolute Adressierung	357
6.1.4.4	Indirekte Adressierung.....	362
6.1.4.5	Indizierte Adressierung	363
6.1.4.6	Relative Adressierung.....	366
6.1.5	Befehlssatz.....	367
6.1.6	Datentypen	370
6.2	Speichermodell	374
6.2.1	Adressraumverschränkung	380
6.2.2	Seitenverwaltung	382
6.2.3	Segmentverwaltung.....	385

6.2.4	Caches.....	387
6.2.5	Virtueller Speicher.....	396
6.3	Mikroarchitektur	398
6.3.1	Befehlsausführung	398
6.3.2	Steuerwerk	400
6.3.3	Rechenwerk.....	405
6.3.3.1	Grundelemente und Struktur.....	405
6.3.3.2	Vektorielles Rechenwerk.....	407
6.3.3.3	Skalares Rechenwerk	407
6.3.4	Mikroarchitektur typischer Maschinen.....	421
6.3.4.1	Stapelmaschine und ihre Befehlsformate	422
6.3.4.2	Akkumulatormaschine und ihre Befehlsformate.....	426
6.3.4.3	Registermaschine und ihre Befehlsformate.....	428
6.3.4.4	Universelle Registermaschine und ihre Befehlsformate	432
	Weiterführende Literatur.....	437
7	Programmieren von Maschinen	439
7.1	Programmieren in Hochsprache.....	441
7.2	Programme in C.....	443
7.3	Programme in Maschinensprache.....	447
7.4	Assembler einer universellen Rechenmaschine	454
7.4.1	Programmierschnittstelle des MIPS.....	457
7.4.1.1	Register.....	457
7.4.1.2	Adressierungsarten des MIPS.....	461
7.4.1.3	Befehle	464
7.4.2	Assembler-Anweisungen	470
7.5	C-Schablonen für eine universelle Registermaschine.....	478
7.5.1	Grundblockgraph	478
7.5.2	Kontrollstrukturen.....	480
7.5.2.1	Verzweigungen	480
7.5.2.2	Schleifen.....	483
7.5.3	Datenstrukturen	485
7.5.4	Unterprogramme	490
7.6	Fibonacci in Assembler.....	498
	Theoreme	508
	Definitionen.....	509
	Bildquellenverzeichnis	512
	Literatur.....	515
	Stichwortverzeichnis	517

Abkürzungsverzeichnis

AGC	Apollo Guidance Computer	ESP	elektronisches Stabilitätsprogramm
BASIC	Beginner's All-purpose Symbolic Instruction Code	FiFo	first in, first out
BCPL	Basic Combined Programming Language	FPGA	field-programmable gate array
CAD	Computer-aided Design	GaAs	Galliumarsenid
CAS	column access select/strobe	GAL	generic array logic
CAS	column access select/strobe	GALS	generic array logics
CDC	Control Data Corporation	GFLOPS	giga floating-point operations per second
CERN	Conseil européen pour la recherche nucléaire	HP	Hewlett-Packard
CISC	complex instruction set computer	IAS	Institute for Advanced Study
CPLD	complex programmable logic device	IBM	International Business Machines
DEC	Digital Equipment Corporation	IBM-PC	IBM Personal Computer
DECT	Digital Enhanced Cordless Telecommunications	KNF	kanonische Normalform
DKNF	disjunktive kanonische Normalform	LiFo	last in, first out
DNF	disjunktive Normalform	LTE	Long Term Evolution
DRAM	dynamic random-access memory	MFLOPS	mega floating-point operations per second
DSP	digital signal processor	MIMD	multiple instruction, multiple data
ECL	emitter-coupled logic	MIPS	microprocessor without interlocked pipeline stages
ECU	electronic control unit	MIT	Massachusetts Institute of Technology
EEPROM	electrically erasable programmable read-only memory	MOS	metal-oxide semiconductor
EPLD	electrically programmable logic device	MOS-FET	Metal-oxide-semiconductor-Feldeffekttransistor
EPROM	erasable programmable read-only memory	MS-DOS	Microsoft Disk Operating System
ESC	Electronic Stability Control	NASA	National Aeronautics and Space Administration
		NB	New B

NEC	National Electrical Code	ROM	read-only memory
NOR	not or	SEAC	Standard Eastern Automatic Computer
NTDS	Naval Tactical Data System	SIMD	single instruction, multiple data
PAL	programmable array logic	SoC	system on a chip
PALs	programmable array logics	SRAM	static random-access memory
PC	Personal Computer	SSEC	Selective Sequence Electronic Calculator
PDP	Programmed Data Processor	TFLOPS	tera floating-point operations per second
PDP-1	Programmed Data Processor 1	TLB	transaction look-ahead buffer
PFLOPS	peta floating-point operations per second	TRADIC	Transistorized Airborne Digital Computer
PIA	programmable interconnect array	TTL	Transistor-Transistor-Logik
PLA	programmable logic array	TX-0	Transistorized Experimental computer zero
PLD	programmable logic device	UMTS	Universal Mobile Telecommunications System
PLDs	programmable logic devices	UNIVAC	Universal Automatic Computer
PROM	programmable read-only memory	UNIX	Uniplexed Information and Computing Service
RAM	random-access memory	VAX-Architektur	Virtual Address eXtension-Architektur
RAS	row access select/strobe		
RISC	reduced instruction set computer	VMS	Virtual Memory System



Historische Entwicklung von Rechenmaschinen

Inhaltsverzeichnis

- 1.1 Mechanische Rechner – 3
 - 1.2 Elektrische Rechner – 11
 - 1.3 Erste elektronische Rechner – 15
 - 1.4 Rechner in der Datenverarbeitung – 20
 - 1.5 Die Zahlenfresser: Supercomputer – 25
 - 1.6 Mikrocomputer – 29
 - 1.7 Eingebettete Computer – 32
 - 1.8 Halbleiter und die Entwicklung der Rechenleistung – 36
- Weiterführende Literatur – 39

instrumentum arithmeticum, „eine Maschine, so ich eine Lebendige Rechenbank nenne, dieweil dadurch zuwege gebracht wird, dass alle Zahlen sich selbst rechnen“.
Gottfried Wilhelm Leibniz

Das Kapitel gibt einen Überblick über die historische Entwicklung der Rechenmaschinen. Am Ende soll der Leser in der Lage sein, die wichtigsten Meilensteine vom mechanischen Rechner über die elektronische Datenverarbeitung bis hin zu den numerisch-wissenschaftlichen Supercomputern einordnen zu können. Da die technologische Realisierung eines Computers einen großen Einfluss auf seine Architektur hat, werden die einzelnen Maschinen ins Verhältnis zur technischen Entwicklung der jeweiligen Zeit gesetzt. Ziel ist es zu verstehen, warum eine bestimmte Implementierung gewählt wurde, insbesondere da manche Entscheidungen rückblickend merkwürdig wirken.

Rechnen ist ein mühsames Geschäft. Jeder kennt noch aus der Schule das beschleichende Gefühl beim Anblick der zahlreichen Übungsaufgaben in seinem Schulbuch. Vor allem bei großen Zahlen sind stets gleichbleibende Rechenschritte immer wieder auszuführen. Eine Unkonzentriertheit des Rechnenden bei der 9. Stelle einer 20-stelligen Multiplikationsaufgabe reicht, um ein falsches Ergebnis zu erhalten. Daher träumten die Menschen seit dem Altertum davon, das Rechnen zu mechanisieren. So sollten die als langwierig und stumpfsinnig empfundenen Rechenschritte automatisiert und von einer Maschine ausgeführt werden. Die folgende Einführung gibt einen kurzen historischen Überblick über die Entwicklung automatischer Rechner bis hin zu unseren modernen Universalcomputern. Denn aus den ersten einfachen Maschinen haben sich Geräte entwickelt, die mehr können als im klassischen Sinne zu rechnen. Inzwischen sind diese universellen Rechenmaschinen in der Lage, jegliche Form von Daten zu bearbeiten und abzuspeichern. Von Social Media über die Steuerung und Regelung von Fahrzeugen und die Automatisierung der Produktion jeglicher Art von Produkten sind elektronische Computer alltäglich. Dabei ist es noch keine 100 Jahre her, dass man gar nicht wusste, wie eine universelle Rechenmaschine aussiehen sollte. Es gab zwar einzelne Konzepte; deren Praxisrelevanz und Alltagstauglichkeit war jedoch noch nicht bewiesen. Erst die rasante technische Entwicklung der Elektronik und später der Mikroelektronik auf der einen als auch eine theoretische Durchdringung der Anforderungen auf der anderen Seite ermöglichten den Bau von Universalcomputern, wie wir sie heute kennen. Die folgende Übersicht über die Entwicklung der Rechenmaschinen ist keineswegs vollständig, soll jedoch ein Gefühl dafür geben, dass nicht alle etablierten Ideen und

Methoden zur technischen Realisierung von Computern oder deren Rechnerarchitektur selbstverständlich sind, sondern erst mühsam erarbeitet werden mussten. Die Rechentechnik ist ein schönes Beispiel für einen evolutionären Prozess.

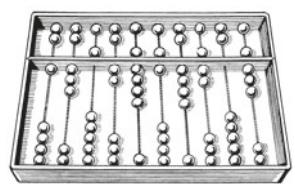
1.1 Mechanische Rechner

Das mühevolle Rechnen wurde zunächst mechanisch automatisiert. Waren diese Versuche am Anfang sehr einfach, aber keineswegs primitiv, so entwickelten sich mit dem Aufkommen der ersten Pendeluhrn bereits ausgefeilte Konzepte zur Implementierung von Rechenmaschinen. Dies war hauptsächlich auf die durch die Uhrmacher entwickelte Feinmechanik zurückzuführen. Allerdings scheiterte die Entwicklung eines vollständig und frei programmierbaren Computers an der Zuverlässigkeit der eingesetzten technischen Komponenten. Obgleich es einfach war, ein mechanisches Berechnungsschema für eine Stelle aufzubauen und mithilfe von Überträgen zu skalieren, so stellte die mechanische Reibung eine natürliche Grenze für die Größe der Zahlendarstellung dar: Im Moment des Überlaufs von der höchsten zur niedrigsten darstellbaren Zahl mussten alle Überträge gleichzeitig durchgeführt werden. Mit steigender Anzahl an Stellen musste in diesem Schritt immer mehr Kraft aufgewendet werden. Hinzu kommen die schiere Größe eines mechanischen Rechners und der Verschleiß der zahlreichen Komponenten. Durch mechanische Hemmungen konnte es dann schnell passieren, dass das komplexe Gebilde Computer schon nach wenigen Rechenschritten streikte. Leider erfordert gerade die Automatisierung des Rechnens eine zuverlässig reproduzierbare Durchführung aller einzelnen Berechnungen. Die technische Unzuverlässigkeit früher mechanischer Rechner verhinderte deren produktiven Einsatz. Erst die im Maschinenbau am Ausgang des 19. Jahrhunderts erreichten Verbesserungen im Zuge der Industrialisierung machten es möglich, einfache mechanische Tischrechner und Registrierkassen zu vermarkten.

Die erste bekannte Rechenmaschine ist der Abakus (lat. „abacus“, Brett, Tafel). Der Abakus ist eine sumerische Erfindung und gelangte aus dem Orient zu den Griechen und Römern und in den Fernen Osten. Die chinesische Adaption des Abakus, der „Suan-Pan“ (Abb. 1.1) wurde später auch in Japan genutzt. Dort ist er unter dem Namen „Soroban“ heute noch in Gebrauch. Der klassische Abakus wurde mit dem Auftreten der ersten mechanischen Rechenmaschinen im Europa des 16. Jahrhunderts abgelöst. Er unterstützt die Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Darüber hinaus kann man mit einem Abakus auch Wur-

1100 v. Chr. bis 1960

Abakus



Pearson Scott Foresman, Wikimedia Commons

Abb. 1.1 Abakus

zeln ziehen. Der Abakus besteht aus einem Rahmen mit mehreren Stangen. Auf diesen sind kleine Kugelchen aufgefädelt. Jede Stange stellt metaphorisch eine Stelle im Zahlensystem dar, während die Position der Kugeln die jeweilige Ziffer repräsentiert. Das manuelle Rechnen – insbesondere die dezimale Addition und Subtraktion – lässt sich mit dieser Rechenmaschine mechanisieren, sodass mit viel Übung nicht mehr im Kopf, sondern mit den Händen gerechnet werden kann.

► Beispiel 1.1.1 – Addition mit dem Abakus

Soll eine Addition mit den Zahlen 8 und 31 durchgeführt werden, so ist die zuletzt genannte Zahl initial mit den Kugelchen auf den Stangen einzustellen. Dazu muss man wissen, dass der vorhandene Steg die 5 unteren Kugeln mit der Wertigkeit 1 von den 2 oberen der Wertigkeit 5 trennt. Die Stangen weisen dabei stets eine 10fach höhere Wertigkeit auf als die jeweils rechts davon gelegenen. Bei Darstellung einer Zahl gelten indes die entsprechenden Kugeln, die zum Steg geschoben sind. Für die Repräsentation der Zahl 31 rückt man folglich ganz rechts 1 Kugel nach oben zum Steg sowie links davon 3 Kugeln. Soll die Zahl 8 addiert werden, rückt man auf der rechten Seite eine Kugel der Wertigkeit 5 nach unten sowie 3 untere zum Steg. Liest man schlussendlich die Zahl ab, so ergibt sich als Wert eine 39. ◀

► Beispiel 1.1.2 – Wurzelziehen mit dem Abakus

Die Quadratwurzel der Zahl 25 wird mit dem Abakus berechnet, indem nacheinander von 1 beginnend die ungeraden Zahlen abgezogen werden, so lange bis der Wert 0 erreicht worden ist. Die Quadratwurzel entspricht genau der Anzahl der getätigten Subtraktionsschritte. Die jeweiligen Subtraktionen erfolgen hierbei durch sukzessives Einstellen des Abakus. Zur Berechnung der Quadratwurzel von 25, geht man wie folgt vor: Man rechnet $25 - 1 = 24$, dann $24 - 3 = 21$, $21 - 5 = 16$, $16 - 7 = 9$ sowie $9 - 9 = 0$. In Summe musste 5-mal subtrahiert werden, was als Schlussfolgerung bedeutet, dass die Quadratwurzel von 25 genau 5 sein muss. ◀



Samuel Freeman, Wikimedia Commons

Abb. 1.2 John Napier

Der schottische Mathematiker Lord John Napier (1550–1617, □ Abb. 1.2) erarbeitete im Jahr 1614 die Grundlagen der Logarithmen unabhängig vom Schweizer Jost Bürgi (1552–1632). 1617 beschrieb der Schotte ein mechanisches Verfahren zur Multiplikation und Division. Dazu verwendete er Rechenstäbchen (□ Abb. 1.3). Es gab genau 8 verschiedene Stäbchen: eines für jedes Vielfache von 2, 3, 4, 5, 6, 7, 8 und 9. Jedes der Rechenstäbchen bestand aus 9 Feldern, die senkrecht untereinander angeordnet waren. Im 2. bis 9. Feld waren die Zahlen jeweils durch einen Diagonalstrich getrennt. Beim Rechenstäbchen für die Vielfachen von 2 steht oben im ersten Feld eine 2. Danach folgen die Vielfachen von 2: Im 2. Feld steht 0/4 wegen $2 \cdot 2 = 4$, im 3. Feld steht 0/6 wegen $3 \cdot 2 = 6$ usw. Die vordere Ziffer stellt dabei den Übertrag dar; so findet sich im Rechenstäbchen für das Vielfache von 2 im 9. Feld der Eintrag 1/8

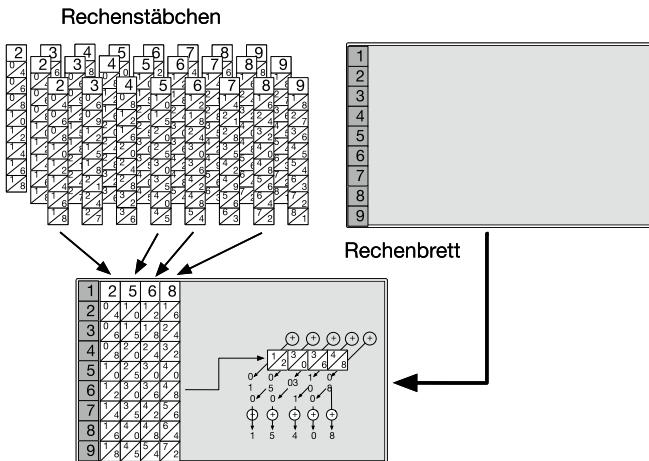


Abb. 1.3 Rechnen mit den Napierschen Rechenstäbchen

wegen $2 \cdot 9 = 18$. Zusätzlich zu den Rechenstäbchen wurde ein mit Zeilennummern 1 bis 9 nummeriertes Rechenbrett benötigt, auf dem sich diese anordnen ließen. Mit den Rechenstäbchen konnten die grundlegenden Rechenoperationen einfach bewerkstelligt werden.

► Beispiel 1.1.3 – Napierische Rechenstäbchen – Multiplikation

Soll wie in Abb. 1.3 die Zahl 2568 mit 6 multipliziert werden, sind die entsprechenden Rechenstäbchen für die Ziffern 2, 5, 6 und 8 derart auf dem Brett zu positionieren, dass sich die höchstenwertige Ziffer der darzustellenden Zahl ganz links befindet. Da der Multiplikator den Wert 6 hat, wählt man auf dem Brett die entsprechende Zeile aus und addiert die Diagonalen der Rechenstäbchen. Ist eine Summe größer als 9, addiert man den entsprechenden Übertrag auf die nebenstehende Stelle dazu. Ausgehend von Zeile 6 liest man also von links nach rechts die Felder 1/2, 3/0, 3/6 sowie 4/8 ab. Für die jeweiligen Stellen der gesuchten Ergebniszahl führt man nun die diagonalen Additionen von rechts nach links durch. Die niedrigstwertige Ziffer bildet somit 8, danach folgen $4 + 6 = 0$ (mit einem Übertrag von 1), $1 + 3 + 0 = 4$ sowie $3 + 2 = 5$ und 1. Unsere Rechnung ergibt somit 15.408. Würde der Multiplikator mehr als eine Stelle umfassen, käme man wie bei der Schulmultiplikation mit Schieben und Addieren auf das erwartete Ergebnis. ◀

Rechenuhr

1



Konrad Melpinger, Wikimedia Commons

Abb. 1.4 Wilhelm Schickard



Unbekannter Autor, Wikimedia Commons

Abb. 1.5 Blaise Pascal



Christoph Bernhard Francke, Wikimedia Commons

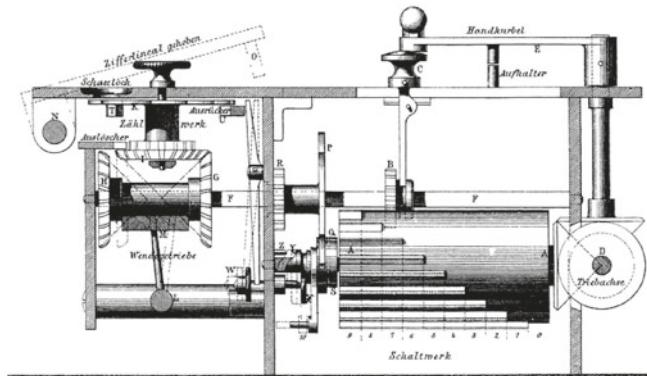
Abb. 1.6 Gottfried Wilhelm Leibniz

Im Jahr 1623 baute der Tübinger Gelehrte und Astronom Wilhelm Schickard (1552–1635, □ Abb. 1.4) eine erste mechanische Maschine für astronomische Berechnungen. Die Rechenuhr beherrschte die Addition und Subtraktion 6-stelliger Zahlen. Die Multiplikation und Division wurden mit einem Rechenwerk auf Grundlage der napierschen Rechenstäbe durchgeführt. In der schickardschen Maschine wurden die Stäbe durch drehende Zylinder ersetzt. Schickards Rechenuhr konnte nur mit ganzen Zahlen rechnen. Die Maschine nutzte 10 Zahnräder, und es kam jeweils 1 Achse für jede Dezimalstelle zum Einsatz. Mithilfe eines Übertragungszahnrades war es möglich, einen Übertrag zu bilden. Erstmals in der Geschichte des maschinellen Rechnens konnte das Ergebnis direkt an der Maschine abgelesen werden.

1642 konstruierte der spätere Philosoph, Mathematiker und Physiker Blaise Pascal (1623–1662, □ Abb. 1.5) mit 19 Jahren eine mechanische Rechenmaschine für seinen Vater, einen königlichen Steuereintreiber. Dieser Rechner bestand aus metallischen Wähl scheiben und galt bis zur Entdeckung von Schickards Unterlagen als die erste Rechenanlage überhaupt. Pascal hat die Maschine im Laufe seines Lebens fortwährend verbessert, und er baute bis zu fünfzig verschiedene und immer komplexere Varianten.

Die Staffelwalzmaschine ist ein zählendes Rechenwerk, das mithilfe der von Gottfried Wilhelm Leibniz (1646–1716, □ Abb. 1.6) erfundenen Staffelwalze arbeitet (□ Abb. 1.7). Eine Staffelwalze ist ein Zylinder mit zumeist 9 unterschiedlich langen Zähnen. Mit diesen lassen sich die Zahlen codieren. Dies geschieht mittels eines auf einer Achse verschiebbaren Lesezahnrades. Der 2. Zahn beginnt nun 2 Breiten des Lesezahnrades vom linken Rand und reicht dann wieder bis zum rechten Rand der Walze. Beim 3. Zahn sind es nun 3 Breiten des Lesezahnrades usw. Platziert man das Lesezahnrad ganz rechts und dreht die Walze 1-mal, dreht sich das Lesezahnrad 10-mal. Befindet sich das Lesezahnrad ganz links, dreht es sich nicht. Ist es jedoch in der Mitte der Walze platziert, dreht es sich 5-mal. Durch die Anordnung des Lesezahnrades auf seiner Achse kann also eine Zahl codiert werden. Auf der Achse des Lesezahnrades wird ein Zählwerk angebracht, welches den Speicher für eine Ziffer einer Zahl darstellt. Für jede Stelle einer zu codierenden Ziffer benötigt man daher ein Lesezahnrad. Ist eine Addition durchzuführen, wird durch Verschieben der Lesezahnräder die 1. Zahl dargestellt. Anschließend arretiert man die Zahnräder und dreht die Walze 1-mal. Nach dem Drehen steht dann diese Zahl in den angeschlossenen Zählern. Nun codiert man die 2. Zahl, arretiert die Maschine erneut und dreht ein 2. Mal die Walze. Am Ende steht in den angeschlossenen Zählern die Summe der beiden Zahlen. Durch Verwen-

1.1 · Mechanische Rechner



Franz Reuleaux, <http://dingler.culture.hu-berlin.de>

Abb. 1.7 Konstruktion einer Rechenmaschine mit Staffelwalze

dung des 10-Komplements und der Abbildung der Multiplikation auf die Addition lassen sich so alle vier Grundrechenarten mit einer Staffelwalzmaschine durchführen.

Eine in hohen Stückzahlen produzierte Staffelwalzmaschine war die Curta (Abb. 1.8). Diese wurde von dem österreichischen Büromaschinenmechaniker Kurt Herzstark (1902–1988) während seiner Haft im Konzentrationslager Buchenwald entwickelt und nach dem Krieg von 1948 bis 1970 gebaut. Das Besondere an der Curta war, dass sie während des Rechenvorgangs in der Hand gehalten werden konnte. Die Curta besaß zunächst 11 Stellen, das spätere Modell Curta II stellte eine Erweiterung auf 15 Ziffern dar. Das Hauptbedienfeld der Rechenmaschine ist das Eingabewerk zum Einstellen der Zahlen. Mithilfe einer Kurbel auf der Oberseite kann die Maschine sowohl subtrahieren als auch addieren. Das Ergebnis nach einer getätigten Operation wird in das Ergebniswerk, das sich ebenfalls oben auf dem Gerät befindet, übernommen. Zusätzlich existiert ein Rundenzähler, welcher registriert, für welche Dezimalstelle die Kurbel wie oft gedreht wurde. Dieser Zähler wird für die Durchführung von Multiplikation und Division mit der Curta benötigt. Darüber hinaus hat das Gerät einen Löschhebel, um sowohl den Rundenzähler als auch das Ergebniswerk auf 0 zurückzusetzen.



Mathias Schneider,
www.curta-schweiz.ch

Abb. 1.8 Curta 1b



Thomas Dewell Scott,
commons.wikimedia.org

Abb. 1.9 Charles Babbage

Rechenschieber

Der in Cambridge, England lehrende Mathematikprofessor Charles Babbage (1792–1871, □ Abb. 1.9) gilt als Vater der modernen programmgesteuerten Rechner. Zur Zeit Babbages ermittelte man die Werte mathematischer Funktionen mithilfe von Tabellen. Diese waren allerdings oft fehlerhaft. Babbage träumte daher davon, auf die Tabellen verzichten und die Funktionswerte mit einer Maschine berechnen zu können. Zunächst entwarf er die Differenzenmaschine. Dabei griff er eine Idee des Franzosen Gaspard Monge (1755–1839) auf: Dieser hatte komplexe Berechnungen in mehrere einfache Additions- und Subtraktionsritte zerlegt. Trainierte Rechenknechte konnten einen derartig zerlegten Algorithmus schnell ausführen. Der Mathematiker selbst beschränkte sich wieder auf das Lösen von Problemen, anstatt viel Zeit mit mechanischen Rechnungen zu verbringen. Dieses manuelle Rechnen übertrug Babbage auf seine Differenzenmaschine. Dabei sollte die Maschine sowohl die Berechnungen übernehmen als auch gleich die erforderlichen Funktionstabellen ausdrucken. Im Anschluss an die von ihm gebaute Differenzenmaschine entwarf Babbage die Analytical Engine. Leider konnte diese Maschine mit den mechanischen Mitteln seiner Zeit nicht gebaut werden. Das Konzept wies jedoch alle Merkmale eines modernen Rechners auf: Babbage sah bereits einen Speicher für 1000 Zahlen vor. Diese sollten jeweils 50 Stellen besitzen. Die Maschine hatte ein Rechenwerk und ein davon getrenntes Steuerwerk. Die Funktion des Steuerwerks, also das Weiterschalten der Befehlsketten, erfolgte dabei in Abhängigkeit von den Ergebnissen des Rechenwerks. Darüber hinaus war eine Ein- und Ausgabeeinheit für Lochkarten vorgesehen, um Resultate dauerhaft zu speichern und automatisch wieder einlesen zu können.

Die ursprüngliche Idee des Rechenschiebers geht ebenfalls auf den schottischen Mathematiker John Napier zurück. Dieser erfand neben den Rechenstäben die Logarithmen (von griech. „logos“, Verständnis, Lehre, Verhältnis und „arithmo“, Zahl, zusammen „Verhältniszahl“), um die Multiplikation auf die Addition zurückzuführen. Zu diesem Zweck legte Napier Logarithmentafeln an. Zu jeder Zahl wurde in einem umfangreichen Tabellenwerk der jeweilige Logarithmus verzeichnet. Durch Nachschauen in diesen Tafeln ließ sich der Logarithmus eines Multiplikanden leicht bestimmen. Durch Addition der so ermittelten Logarithmen und durch erneutes Nachschlagen wurde dann anschließend das Produkt der Zahlen ermittelt. Im Jahr 1620 markierte der Londoner Mathematiker Edward Gunter (1581–1626) ein Lineal mit einer logarithmischen Skala. Mithilfe eines Stechzirkels ließen sich so Längen abmessen. Wurden diese addiert, konnten so Multiplikationen und Divisionen durchgeführt werden.

Im Jahr 1622 ersetzte der Geistliche William Oughtred (1574–1660) den Stechzirkel durch ein zweites Lineal mit logarithmischer Skala und erfand so den ersten Rechenschieber. Diese Vorrichtung wurde 1654 durch Seth Partridge (1603–1686) weiterentwickelt. Mithilfe einer frei beweglichen Zunge mit einer weiteren logarithmischen Skala ließ sich die Handhabung des Rechenschiebers erneut vereinfachen. Insbesondere da die beiden ursprünglich getrennten Rechenlineale nun zu einem kompakten Gerät vereinigt wurden.

Isaac Newton (1643–1727, □ Abb. 1.10) erhöhte die Genauigkeit des Rechenschiebers durch die Einführung eines frei beweglichen Läufers. Allerdings wurde ein funktionierendes Gerät auf der Grundlage von Newtons Idee erst 1775, 48 Jahre nach dem Tode Newtons, durch John Robertson (1712–1776) gebaut. Die heutige Form des Rechenschiebers für die Grundrechenarten Addition, Subtraktion, Multiplikation und Division wurde 1850 eingeführt. Mit dem Rechenschieber kann jede reelle Zahl mit einer anderen reellen Zahl multipliziert werden. Dabei wird die Gleitkommazahl normiert, sodass sie immer die Form $1,xxxx \cdot 10^y$ hat. Den Wert y muss man sich merken. Das Problem $1,xxxx \cdot 1,yyyy$ kann dann mit dem Rechenschieber durch die Berechnung mit einer Festkommazahl gelöst werden.

Bedingt durch die weitere Möglichkeit, trigonometrische Funktionen auf einer Skala darzustellen, wurde der Rechenschieber auch in der Navigation und bei der Artillerie zur Geschossbahnberechnung eingesetzt. Da die große Stärke eines Rechenschiebers die numerische Berechnung von Dezimalzahlen ist, wurde dieser selten für kaufmännische Aufgabenstellungen angewendet. Daher war der Rechenschieber in der Mitte des 20. Jahrhunderts das Symbol für einen Physiker oder Ingenieur, so wie das Stethoskop heute noch mit einem Arzt assoziiert wird. Auch die ersten elektronischen Rechner wurden mithilfe von Rechenschiebern dimensioniert. Allerdings führte die Einführung von naturwissenschaftlichen Taschenrechnern (ab 1972 mit dem HP35) schnell zur Einstellung der Produktion von Rechenschiebern, da ein Taschenrechner die gleichen Aufgaben wie ein Rechenschieber mit einer wesentlich höheren Genauigkeit erfüllen konnte und schlussendlich preiswerter und handlicher war.

Der in □ Abb. 1.11 dargestellte Rechenschieber besitzt die beiden grundlegenden Skalen C und D. Da Rechenschieber sehr umfangreiche Funktionen inkl. der trigonometrischen unterstützen, hat der Rechenschieber noch weitere Skalen: die Skalen T, K, A, L, S. In der □ Abb. 1.11 ist zu erkennen, dass ein Rechenschieber aus drei Bauteilen besteht: dem Körper als grundlegende Trägerstruktur, der Zunge als in den Körper verschiebbar eingelagertes Element und dem mit einer vertikalen sowie mittig platzierten Markierung versehenen durchsichti-



Godfrey Kneller, Barrington Bramley,
Wikimedia Commons

□ Abb. 1.10 Isaac Newton

Skalen

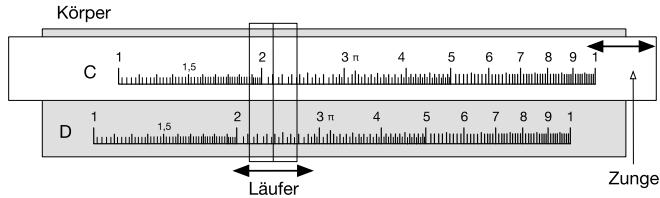


Abb. 1.11 Einfacher Rechenschieber

gen Läufer. Dieser kann über Körper und Zunge frei bewegt werden. Sowohl Körper als auch Zunge sind mit einer identischen logarithmischen Skala ausgestattet. Wichtig ist, dass Läufer und Zunge unabhängig voneinander bewegt werden können. Bei der Fertigung eines Rechenschiebers kommt es darauf an, die Skalen sehr genau und fein zu arbeiten. Gleichermaßen gilt für die Markierung auf dem Läufer. Mit der Verarbeitungsqualität der Skalen ist die mögliche Rechengenauigkeit der Maschine verknüpft. In der Vergangenheit wurden auch große Rechenzyylinder mit logarithmischen Skalen gefertigt, um so die Genauigkeit zu erhöhen.

Der Bauingenieur Konrad Zuse (1910–1995) arbeitete als Statiker im Flugzeugbau und erkannte, dass sich viele formal gut verstandene statische Berechnungen effektiv automatisieren lassen. Er kündigte daher bei den Henschel-Werken, um eine Rechenmaschine zu entwickeln. In seiner Werkstatt baute er im Jahr 1937 die Zuse 1, abgekürzt Z1. Der Rechner war mechanisch aus Schaltblechen aufgebaut, wobei sich Zuse bewusst gegen elektromechanische Relais entschied. Die Schaltglieder wurden mit einem alten Staubsaugermotor angetrieben. Die Z1 wies bereits viele Merkmale heutiger Computer auf: Steuer- und Rechenwerk waren wie bei Babbage getrennt ausgeführt und der Rechner arbeitete binär. Auch besaß die Maschine eine Ein- und Ausgabe mithilfe eines Lochkartenlesers. Obwohl der Rechner kompakter konturiert war als eine elektromechanische Implementierung, ließ die Zuverlässigkeit zu wünschen übrig, da sich die mechanischen Schaltglieder häufig verhakten.

Computer

Es ist heute nicht mehr vorstellbar, dass sowohl die Berechnungen zum Bau der Atombombe als auch die Bahnberechnungen für die ersten Raumflüge mithilfe mechanischer Rechenmaschinen durchgeführt wurden. Diese Geräte beherrschten einzig die Grundrechenarten, und ein Algorithmus wurde „programmiert“, indem die notwendigen Operationen gemäß des Datenflusses in einem Raum mit einzelnen Rechenmaschinen abgebildet wurden. Vor jeder Rechenmaschine saß ein Computer (Abb. 1.12), wie man damals die Angestellten nannte, die diese Maschinen gemäß der Anweisungen der



Courtesy, NASA/JPL-Caltech

Abb. 1.12 Computer am JPL der NASA in den 1950er-Jahren

Wissenschaftler und Ingenieure bedienten. Da diese Berechnungen hohe Konzentration erforderten und sehr gleichförmig abließen, beschäftigte die National Aeronautics and Space Administration (NASA) zu dieser Zeit hauptsächlich weibliche Computer. Diese Frauen wurden unter der Bezeichnung Rocket Girls bekannt und hatten sehr gute Universitätsabschlüsse in Mathematik. Trotz der gleichförmigen Arbeit waren sie motiviert, sich mit der Technik weiterzubilden: Mit der Umstellung auf die elektronische Datenverarbeitung wurden aus den weiblichen Computern Programmierer. Daher sieht man auf den historischen Aufnahmen von frühen Rechnern ausschließlich weibliche Bediener. Der Begriff Computer hat sich somit von einer Bezeichnung eines Berufs oder einer Tätigkeit zu einer Beschreibung einer automatischen Datenverarbeitungsmaschine gewandelt.

1.2 Elektrische Rechner

Nach Einführung der ersten elektromechanischen Bauelemente in der Nachrichtentechnik, den Relais, wurden diese bald auch als Schalter in Rechnern eingebaut. Insbesondere waren diese Bauteile in der Anfangszeit der Computertechnik zuverlässiger als Röhren.

Das Zeitalter der Datenverarbeitung begann mit der Tabelliermaschine. Im ausgehenden 19. Jahrhundert wuchsen die Vereinigten Staaten von Amerika rasant und breiteten sich immer weiter nach Westen aus. Um das Land regierbar zu halten, versuchte die amerikanische Regierung, alle zehn Jahre ei-

1890–1950

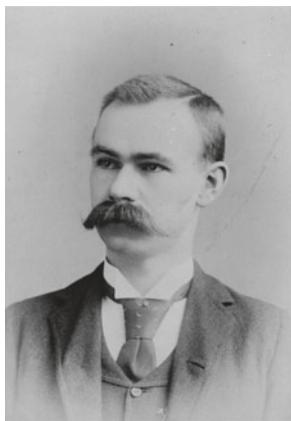
Tabelliermaschine

ne Volkszählung durchzuführen. Bei der Zählung 1880 zeigte sich, dass die inzwischen auftretende Fülle an Daten manuell nicht mehr ausgewertet werden konnte. Herman Hollerith (1860–1926, □ Abb. 1.13), ein Mitarbeiter der 1880er-Volkszählung, suchte in den folgenden Jahren nach einer Lösung des Problems und reichte eine Reihe von Patenten zur automatischen Datenverarbeitung ein. Er hatte beobachtet, dass ein Schaffner die mehrmalige Benutzung von Fahrkarten ausschließen konnte, wenn er die bereits kontrollierten Fahrscheine nach einem bestimmten Schema lochte. Durch diese Lochung codierten die Zugbegleiter verschiedene Eigenschaften einer Person, beispielsweise die Hautfarbe oder das Geschlecht.

Ausgehend von dieser Beobachtung entwickelte Hollerith ein Lochkartenzählverfahren. Die Eigenschaften eines Objektes oder einer Person wurden mithilfe einer Lochkarte codiert und dann automatisch verarbeitet. Dazu entwickelte Hollerith eine Tabelliermaschine genannte Lochkartenzählanlage. Wie bei einer mechanischen Rechenmaschine wurden die Daten mithilfe von Zähluhren dargestellt. Jedem zu zählenden Merkmal wurden mehrere Zählwerke bestehend aus jeweils mehreren Ziffern zugeordnet und mit einem Zahnrad zur Bildung des Übertrags ausgestattet. Die Tabelliermaschine liest dann einen Stapel von Lochkarten ein, und die Löcher betätigen das jeweilige Zählwerk. Anschließend werden die Karten in einen Korb ausgegeben. Ausgehend von dieser einfachen Idee erweiterte Hollerith die Maschine: Durch ein Relais war es möglich, die jeweilige Lochkarte in Abhängigkeit der codierten Daten in unterschiedliche Ausgangskästen auszugeben. Hierfür konnten die Zählwerke elektrische Impulse erzeugen und damit den entsprechenden elektromechanischen Schalter ansprechen. Ordnete man nun verschiedene Tabelliermaschinen gemäß eines gewünschten Datenflusses an, ließ sich ein Algorithmus implementieren. Ähnlich wie mit menschlichen Computern konnte so ein komplexeres Programm realisiert werden.

Hollerith erkannte auch, dass Zählen alleine das Datenverarbeitungsproblem nicht löst. Daher erweiterte er die Tabelliermaschine mit einem Additionswerk, sodass spätere Ausführungen des Gerätes bereits kleine Programme selbstständig ausführen konnten. Da die Vorrichtung auch einfache Rechnungen durchführen konnte, setzten sich diese Maschinen schnell im Bereich der Buchhaltung durch: Die automatische Datenverarbeitung war geboren. Die von Hollerith 1896 gegründete Firma zur Vermarktung dieses Prinzips wurde im Jahr 1923, nach mehreren Fusionen und Umbenennungen zur International Business Machines Corporation, der weltbekann-ten IBM.

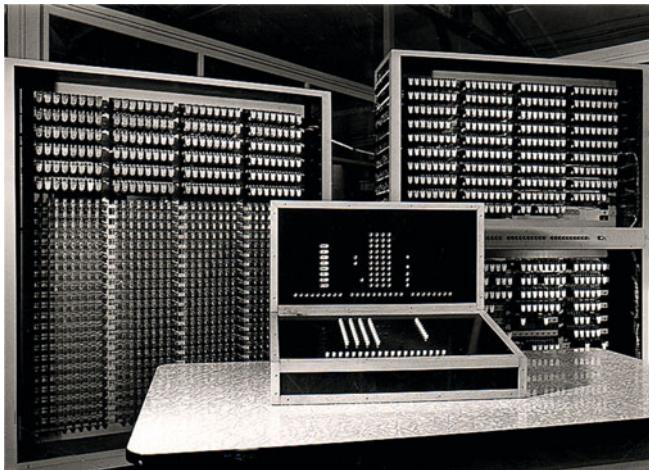
Lochkartenzählverfahren



Charles Milton Bell, 1888, Library of Congress Prints

□ Abb. 1.13 Herman Hollerith

Gründung von IBM



Zuse KG, Horst Zuse

Abb. 1.14 Nachbau der Z3 in den 1960er-Jahren

Mit der Z3 (Abb. 1.14) schuf Konrad Zuse (Abb. 1.15) 1941 in Berlin den ersten universell programmierbaren digitalen Computer. Wie später gezeigt werden konnte, war die Maschine bereits Turing-mächtig, wenn die Lochstreifen zusammengeklebt werden und man von einer begrenzten Rechengenauigkeit ausgeht. Im Gegensatz zur Z1, die sich als unzuverlässig erwiesen hatte, setzte Zuse nun vollständig auf elektromechanische Relais. Der Rechner bestand insgesamt aus 2200 dieser Schalter, davon 600 allein für das Rechenwerk und 1600 für den 200 Kilobyte (kB) großen Speicher. Die mit binärer Gleitkommaarithmetik arbeitende Maschine war jedoch nicht als Turing-mächtiger Computer konzipiert, da Zuse diesen Grundsatz noch nicht kannte. Die Z3 implementierte bereits viele heute verwendete Prinzipien der Rechnerarchitektur. So war es möglich, während der Berechnungen mit dem Benutzer zu interagieren, und es gab eine Ein-/Ausgabe-Schnittstelle sowie verschiedene Lochstreifenleser. Auf den Lochstreifen waren die Rechenprogramme gespeichert, während die Operanden über eine Tastatur eingegeben werden konnten. Der Rechner arbeitete in Fließbandverarbeitung und implementierte bereits das Mikroprogrammprinzip. Der Takt der Maschine von 5,3 Hz wurde von einem rotierenden Elektromotor erzeugt. Die Z3 konnte addieren, subtrahieren, multiplizieren und dividieren. Darüber hinaus konnte Zuse mit der Maschine auch Wurzeln ziehen. Einzig Sprungbefehle fehlten noch, um einen universellen Rechner zu implementieren.



Horst Zuse, Horst Zuse

Abb. 1.15 Konrad Zuse

Mark 1**1**

Mitten im Zweiten Weltkrieg, in den Jahren 1943 und 1944, entwickelte der Professor für angewandte Mathematik, Howard Aiken (1900–1973) an der Harvard University zusammen mit mehreren Ingenieuren und Grace Hopper den elektromechanischen Rechner Mark 1 (Automatic Sequence Controlled Computer, ASCC). Die Maschine wurde von International Business Machines (IBM) gebaut, wog etwa 5 t und bestand aus ungefähr 70.000 Teilen. Sie verarbeitete 10-stellige Dezimalzahlen und führte eine Addition oder Subtraktion in 0,3 s aus. Für eine Multiplikation wurden 6 s benötigt, eine Division dagegen wurde in 11 s durchgeführt. Das Programm der Maschine war auf einem 24-stelligen Lochstreifen gespeichert, und die Datenausgabe erfolgte auf einer Schreibmaschine. Eingesetzt wurde der Computer für ballistische Berechnungen bei der US-Navy. Außerdem führte John von Neumann (1903–1957) im Jahr 1944 Analysen zur Konstruktion der über Nagasaki abgeworfenen Plutoniumbombe im Rahmen des Manhattan-Projektes auf dem Rechner durch. Die Anlage war nach der von Konrad Zuse konstruierten Z3 die zweite existierende Turing-mächtige Maschine der Welt. Die Mark 1 und die Z3 ähnelten sich sehr, da beide Rechner auf den Prinzipien von Charles Babbage aufbauten. Bedingt durch den Zweiten Weltkrieg wussten jedoch Zuse und Aiken nichts von den jeweiligen Bemühungen des anderen.

Zuse Z4

Von 1942 bis 1945 baute Konrad Zuse dann die Z4. Da Zuse während des Krieges aufgrund mangelnder Förderung nicht weiter praktisch arbeiten konnte, konzipierte er erstmals eine Programmiertechnik: das Plankalkül. Zuse verstand unter Programmieren die Erstellung eines Rechenplans und nannte die Z4 ein Planfertigungsgerät. Da die Z4 bereits das Konzept der symbolischen Adressen implementierte, konnte die Programmierung der Maschine innerhalb von 3 h erlernt werden. Der aus 2200 Relais aufgebaute Prozessor arbeitete mit zwei Registern für Gleitkommazahlen und besaß einen Speicher von 64 Zahlen. Der Rechner konnte zunächst nur linear ablaufende Programme ausführen. Allerdings hatte Zuse die Idee, die Maschine mit mehreren Lochkartenlesegeräten auszustatten, von denen er auch zwei einbaute. Da die Lesegeräte vom Programm aus gesteuert werden konnten, ließen sich damit und mit den später eingeführten Sprungbefehlen *UP*, *UP'*, *FIN*, *FIN'* Unterprogrammaufrufe realisieren. Mit dem Befehl *UP* war es möglich, einfach zu springen. Der Befehl *UP'* dagegen implementierte einen bedingten Sprung in Abhängigkeit vom Inhalt eines Registers. *FIN* war somit ein einfacher Rücksprungbefehl, während *FIN'* einen bedingten Rücksprung durchführte. Im Jahr 1950 war die Z4 der einzige funktionierende Computer in Europa. Da der Rechner an der ETH Zürich etwas früher als die Universal Automatic Com-

puter (UNIVAC) in den USA aufgebaut wurde, gilt er als die erste kommerzielle Rechenanlage der Geschichte.

1.3 Erste elektronische Rechner

Da mechanische oder elektromechanische Rechner langsam arbeiteten, versuchte man früh, die bekannten Prinzipien mit elektronischen Schaltungen zu implementieren. In der goldenen Zeit des Radios waren dies zunächst elektronische Vakuumröhren, später dann Dioden und in den 1950er-Jahren die ersten Computer auf Transistorbasis.

1.3.1 Röhrencomputer

Im Zweiten Weltkrieg benutzte die deutsche Wehrmacht eine Maschine zum Verschlüsseln von Nachrichten: die Enigma (Abb. 1.16). Hierbei handelte es sich um eine Rotschlüsselmaschine, die mit mehreren hintereinandergeschalteten, von Buchstabe zu Buchstabe wechselnden sowie jeweils durch einen Rotor ausgeführten monoalphabetischen Substitutionen arbeitete. Dadurch verwendete die Maschine zur Codierung ein einziges festes Schlüsselalphabet. Die Deutschen waren damals der Ansicht, dass die Komplexität der Verschlüsselung so hoch sei, dass ein Mensch eine verschlüsselte Nachricht nicht in vertretbarer Zeit entschlüsseln könne. Zudem änderte man jeden Tag um Mitternacht den Schlüssel, sodass die Entschlüsselung einer Mitteilung nicht dazu führen sollte, dass alle Botschaften decodiert werden können. Während des Krieges gelangten die Alliierten an mehrere Enigmas und erbeuteten auch Codebücher der Wehrmacht. Nahe der Kleinstadt Bletchley betrieb die britische Regierung eine Dechiffrierstation. Diese Station-X genannte Einrichtung wurde später unter dem Namen Bletchley Park bekannt. In einem alten Herrenhaus versammelte damals das Militär die besten britischen Mathematiker. In Bletchley Park arbeitete u. a. Alan Turing (1912–1954) an der Entschlüsselung der deutschen Militärnachrichten. Im Rahmen dieser Tätigkeit gelangen ihm wesentliche Einsichten zur Mathematik von Rechenanlagen.



United States Government, Wikimedia Commons

Abb. 1.16 Enigma

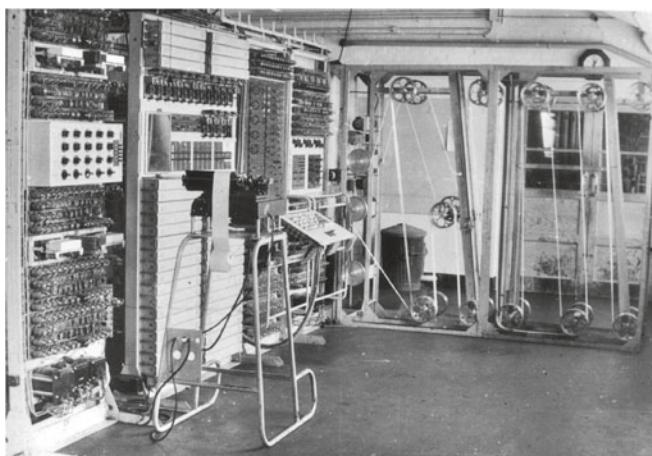
Turing-Bombe

1

Alan Turing konstruierte zusammen mit Gordon Welchman (1906–1985) eine Vorrichtung, die es ermöglichte, den Schlüsselraum der Enigma mechanisch zu durchsuchen. Dazu bildete die Maschine die Motorwalzen der Verschlüsselungsmaschine nach. Damit konnten die Mathematiker schließlich systematisch verschiedene Schlüssel ausprobieren, da sie annahmen, dass in jedem verschlüsselten Text bekannte Worte oder Zeichensequenzen auftreten müssen. Die deutschen Militärfunker beendeten zu dieser Zeit ihre Nachrichten häufig mit der Phrase „Heil Hitler“. Mit diesem Wissen war die Schlüsselsuche erfolgreich. Die von Alan Turing konstruierte Maschine nannte man Turing-Bombe und später nach den Verbesserungen durch Welchman Turing-Welchman-Bombe.

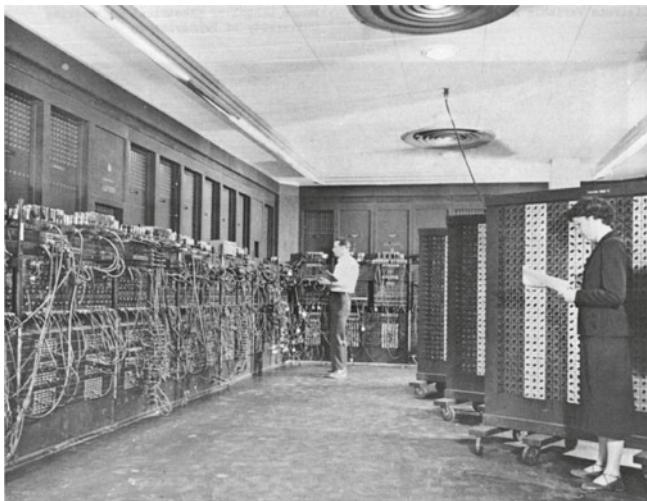
Allerdings führten die Deutschen während des Krieges ein neues Gerät ein, die Lorenz-Schlüsselmaschine. Sie diente der automatischen Verschlüsselung leitungsgebundener Fernschreiber. Um auch die Nachrichten der Lorenz-Schlüsselmaschine knacken zu können, konstruierte der ebenfalls in Bletchley Park arbeitende englische Mathematiker Max Neumann (1897–1984) auf der Grundlage der Turing-Welchman-Bomben einen elektronischen Computer.

Zwischen 1943 und 1946 wurden in Bletchley Park bis zu zehn Maschinen dieses Colossus (Abb. 1.17) genannten Typs eingesetzt. Dadurch konnten die Alliierten verschlüsselte Nachrichten der Deutschen erfolgreich abhören. Bletchley Park trug daher in hohem Maß zum Ausgang des Krieges bei. Ein Colossus bestand aus 1500 Elektronenröhren und benötigte etwa 4,5 kW elektrische Leistung. Diese Maschine konnte um die 100 logische Operationen in 200 µs abarbeiten und



Jack Good, Donald Michie and Geoffrey Timms, 1945, Wikimedia Commons

Abb. 1.17 Colossus 10 in Bletchley Park



US Army, Wikimedia Commons

■ Abb. 1.18 Der erste elektronische Rechner: ENIAC

besaß bereits fotoelektrische Lochkartenleser. Ein Computer im heutigen Sinne war Colossus jedoch nicht: Im Gegensatz zur Z4 war Colossus nicht frei programmierbar, sondern ausschließlich auf das Problem des Knackens von Schlüsseln spezialisiert.

Der erste elektronische, Turing-mächtige und frei programmierbare Computer war der 1946 vorgestellte ENIAC (Electronic Numerical Integrator and Computer) (■ Abb. 1.18). Dieser erste elektronische Universalrechner wurde ab 1942 im Auftrag der United States Army an der University of Pennsylvania entwickelt und gebaut. Dort wurde der Rechner zur Berechnung ballistischer Flugbahnen eingesetzt. Er kostete etwa 500.000 US\$. Die Maschine bestand aus 174.468 Elektronenröhren und wog etwa 27 t. Mit einer elektrischen Leistungsaufnahme von 174 kW benötigte der ENIAC etwa 0,2 ms für die Addition und Subtraktion 10-stelliger Dezimalzahlen. Eine Multiplikation dauerte 2,8 ms, eine Division etwa 24 ms. Das Ergebnis einer Berechnung der Quadratwurzel lag nach etwa 300 ms vor.

ENIAC

Wie schon in Bletchley Park zeigte sich jedoch, dass Elektronenröhren für den Betrieb eines elektronischen Computers nicht zuverlässig genug sind. Denn durch die hohe Anzahl an Röhren war fast immer eine kaputt. Deren Zuverlässigkeit wurde erhöht, indem der Rechner einfach durchlief. Man verzichte darauf den Rechner zum Feierabend aus- und morgens wieder einzuschalten. Es hatte sich nämlich gezeigt, dass die Röhren durch die unterschiedlichen Temperaturen in verschiedenen Betriebszuständen stark belastet wurden und infolge-

dessen schnell durchbrannten. Eine solche Maßnahme hatte man bereits in Bletchley Park eingeführt. Zusätzlich ließ sich die Standzeit der Maschine erhöhen, indem die Röhren mit nur 10 % ihrer Nennleistung arbeiteten. Da ständig Bauteile ausfielen, besaß der ENIAC bereits Testschaltungen, um den Betrieb zu überwachen und Fehler zu melden; insbesondere, da die Betreiber etwaige Ausfälle zunächst nicht bemerkten und dies dann zu falschen Ergebnissen bei den Rechnungen führte. Programmiert wurde der ENIAC, indem man die vorhandenen Register und Operationen von Hand verdrahtete. Auch implementierte der ENIAC zunächst keine Sprungbefehle. Diese wurden erst später durch Umwandlung einiger Datenleitungen in Signalleitungen eingeführt.

Am Institute for Advanced Study in Princeton (IAS) schlug John von Neumann (Abb. 1.19) vor, einen Rechner zu bauen. Dafür hatte er 1945 in einem Aufsatz einen Entwurf einer universellen Rechenmaschine beschrieben. Aufbauend auf den theoretischen Konzepten Alan Turings und der technischen Implementierung des ENIAC gelang es von Neumann, die Mittel für den Bau des IAS-Computers 1945 einzubringen. Die Maschine – zunächst MANIAC-0 (Mathematical and Numerical Integrator and Computer) genannt – wurde 1951 zur Berechnung thermonuklearer Vorgänge bei der Entwicklung der ersten Wasserstoffbombe eingesetzt. Später nutzte man den Computer für Monte-Carlo-Simulationen und für Wettervorhersagen.

IBMs Selective Sequence Electronic Calculator (SSEC) ging bereits im Jahr 1948 in Betrieb. Diese im IBM-Hauptquartier in Manhattan und vom Watson Scientific Computing Laboratory gebaute Maschine bestand aus Röhren und elektromechanischen Relais. Der Rechner wurde von dem Astronomen Wallace John Eckart (1902–1971) konzipiert und berechnete für die NASA die Mondpositionen. Mit diesen Daten wurden später die Flugbahnen der Mondraketen geplant.



LANL, LANL.Gov

Abb. 1.19 John von Neumann



NIST, Wikimedia Commons

Abb. 1.20 Samuel Alexander

1.3.2 Halbleitercomputer

Der erste Computer, der mithilfe von zuverlässigen Halbleiterbauelementen implementiert wurde, war der am National Bureau of Standards entworfene Standards Eastern Automatic Computer (SEAC). Der von Samuel Alexander (1910–1967, Abb. 1.20) gebaute Rechner war einer der ersten nach dem Konzept von John von Neumann implementierten Maschinen und wurde in einer Logik mit bis zu 16.000 Germaniumdioden realisiert. Daneben arbeiteten im SEAC 1600 Elektronenröhren. Der SEAC war die erste Rechenmaschine mit einem internen Programmspeicher. Der Rechner wurde zu Trainingszwe-

1.3 · Erste elektronische Rechner

cken entwickelt, und in den 14 Jahren seines Betriebs schrieb man für ihn bereits die ersten Assembler- und Compilerprogramme.

Im Jahr 1955 versuchte man am Massachusetts Institute of Technology (MIT) Lincoln Laboratory einen Rechner mit Transistoren zu bauen (Abb. 1.21). Der Transistorized Experimental computer zero (TX-0) konnte 83.000 Operationen pro Sekunde verarbeiten und verfügte über einen Arbeitsspeicher aus Magnetkernen. Dieser Speicher hatte eine Kapazität von 65.536 Worten zu 18 bit. Bereits 1957 verwendete man eine 10-Zoll-Kathodenstrahlröhre als einen 512×512 Pixel auflösenden elektronischen Monitor. Bei dieser Maschine ließen sich Programme schon auf Magnetbändern speichern und die Berechnungsergebnisse auf einem angeschlossenen Drucker ausgeben. Da es sich um einen experimentellen Computer handelte, implementierte man auf dem Rechner einfache textbasierte Spiele wie Schach und Tic-Tac-Toe. Der TX-0 wurde zu wesentlichen Teilen vom Elektroingenieur Ken Olsen (1926–2011) entwickelt und war der direkte Vorgänger des später von DEC, einer von Olsen 1957 gegründeten Computerfirma gebauten Minicomputers Programmed Data Processor 1 (PDP-1).

Gleichzeitig entwickelten die Bell-Laboratorien für die amerikanische Luftwaffe einen Computer, der aus etwa 10.000 Germaniumdioden und 700 Transistoren bestand. Bei einer elektrischen Leistung von 100 W konnte der Transistorized Airborne Digital Computer (TRADIC) genannte Rechner eine

TX-0

TRADIC



U.S. Army Photo, Number 163-12-62, Wikimedia Commons

Abb. 1.21 Die Entwicklung einer Speicherzelle: von der ENIAC bis zum Transistor

Million Operationen pro Sekunde verarbeiten. Die Maschine wurde im Jahr 1955 fertiggestellt.

1.4 Rechner in der Datenverarbeitung

Ab 1952

In den Jahren der ersten elektronischen Computer erfolgte die Datenverarbeitung bei Volkszählungen, Banken oder in der Lohnbuchhaltung mit Tabelliermaschinen. Die Hersteller dieser Maschinen entwarfen und bauten später elektronische Datenverarbeitungsanlagen. Diese Großrechner oder Mainframe-Computer hielten dann auch schnell Einzug in die Zentralen großer Banken oder Industriekonzerne. Bei der Verwaltung von Konten oder in der Lohnbuchhaltung mussten sehr viele Datensätze abgearbeitet werden. Die damals zur Verfügung stehenden Rechenleistungen führten bei der Lohnabrechnung zu Rechenzeiten von mehreren Tagen. Ein damaliger Mainframe musste daher über viele Stunden zuverlässig die ganzen Rechenaufträge im Stapelbetrieb abarbeiten. Die Rechner waren für einen hohen Datendurchsatz optimiert und die Ressource Rechenzeit wurde mithilfe eines Zeitscheibenverfahrens aufgeteilt. Anders als bei experimentellen Computern zu Forschungszwecken konnte man sich einen Stillstand der Maschine nicht erlauben. Da in den frühen 1960er-Jahren noch keine hochintegrierten Schaltungen verfügbar waren, ließen sich die Rechner im laufenden Betrieb warten.

UNIVAC

Zwei ehemalige ENIAC-Entwickler John Eckert (1919–1995) und John Mauchly (1907–1980) waren 1951 der Meinung, dass Computer auch in der Datenverarbeitung eingesetzt werden können, also deren Einsatz nicht nur auf mathematisch-technische Anwendungen beschränkt ist. Mit dem Universal Automatic Calculator (UNIVAC) bauten sie für das US Census Bureau nach der Zuse Z4 den zweiten kommerziellen Rechner der Welt. Die Maschine bestand aus 5200 Röhren, wog 13 t und konnte 1905 Operationen pro Sekunde verarbeiten. Dabei benötigte sie 125 kW elektrische Leistung. Die Maschine hatte einen 1000 Worte umfassenden Hauptspeicher. Dieser Speicher war aus Quecksilberrohren aufgebaut, in denen durch jeweils einen sendenden und einen empfangenden Piezokristall Wellen im Ultraschallbereich im Quecksilber erzeugt wurden. Durch verschiedene Schwingmuster konnten nun unterschiedliche Wörter codiert und durch Rückführung eines Musters konnte dieses als umlaufende akustische Welle länger gespeichert werden. 1952 wurde der Rechner für eine Hochrechnung bei der Präsidentschaftswahl der Vereinigten Staaten von Amerika eingesetzt. In Deutschland wurde 1956 ein kommerzielles Rechenzentrum mit einer UNIVAC aufgebaut. Damals kostete eine Stunde Rechenzeit 1470 Mark.

1.4.1 Großrechner von IBM

Die Computerfirma IBM entwickelte sich aus der 1896 von Herman Hollerith (1860–1929) gegründeten Firma Tabulating Machine Company. Mit den Tabelliermaschinen ist das Unternehmen schon vor Einführung von elektronischen Computern in der Datenverarbeitung geschäftlich aktiv. Für Lochkartenmaschinen hatte das seit 1923 unter dem Namen IBM firmierende Unternehmen Anfang des 20. Jahrhunderts eine Monopolstellung. Ab 1950 wurde IBM auch Anbieter von Rechnern für die elektronische Datenverarbeitung und stieg zeitweise zum größten und führenden Unternehmen für Computerhardware auf.

Bei den 700/7000er-Computern von IBM handelte es sich um eine erste kommerzielle Baureihe von Rechnern zur elektronischen Datenverarbeitung. Waren die ersten Maschinen in den 1950er-Jahren noch aus Röhren gebaut, wurden nachfolgende Generationen bereits in Transistorstechnik gefertigt. Um universell einsetzbar zu sein, war der Rechner für verschiedene Anwendungen frei programmierbar. Da die Kunden aber abweichend leistungsfähige Computer nachfragten, entwickelte die IBM unterschiedliche Typen dieser Baureihe und bot diese zu differenzierten Preisen an. Allerdings hatte jede dieser Maschinen noch einen eigenen, zu den anderen Anlagen verschiedenen Befehlssatz.

Im Jahr 1952 kam mit der IBM 701 der erste Rechner von IBM auf den Markt, der das wissenschaftliche Rechnen unterstützte. Die IBM 701 war ein nach der Architektur von John von Neumann entwickelter Rechner, der aus 72 Williams-Röhren bestand. Diese Bauteile waren als elektronische Speicher bereits in den 1940er-Jahren erfunden worden. Abgeleitet von der braunschen Röhre wurden die Informationen in den Leuchtpunkten eines phosphoreszierenden Schirms gespeichert. Eine der Williams-Röhren der IBM 701 konnte 1024 Bit verarbeiten. Die Befehle des Rechners hatten eine Wortbreite von 18 bit, Zahlen wurden im 18-Bit- oder 36-Bit-Festkommaformat berechnet. Die Maschine hatte zwei Register und arbeitete mit einer Taktperiode von $12\ \mu\text{s}$. Die 701 war zudem der erste kommerzielle Computer, der neben den Lochkartenlesern und -stanzern mit Magnetbandlaufwerken ausgestattet war. Auf dem Rechner wurde der erste Fortran-Compiler geschrieben. Dies war die zweite Implementierung eines Compilers, nachdem bereits zuvor ein Übersetzungsprogramm auf einer Maschine des Lawrence Livermore National Laboratory der University of California lief. Von dem Rechner wurden 19 Exemplare für Rüstungsunternehmen gebaut.

Die IBM 702 (Abb. 1.22) war ein Rechner für kaufmännische Anwendungen und sollte 1953 der UNIVAC direkte

IBM

700/7000er-Serie

701

702



Martin H. Weik, Wikimedia Commons

■ Abb. 1.22 IBM 702 zur elektronischen Datenverarbeitung

Konkurrenz machen. Sie hatte eine geringere Leistung als die IBM 701 und ihr Speicher bestand aus Röhren.

704

Die IBM 704 war der erste in Serie produzierte Rechner mit Gleitkommaarithmetik und kam 1954 auf den Markt. Der von Gene Amdahl (1922–2015) gebaute Computer arbeitete erstmals mit Magnetkernspeichern, die die Williams-Röhren ersetzten. Die Maschine verarbeitete 40.000 Befehle pro Sekunde. Bis 1960 wurden 123 Computer dieser Reihe verkauft. Für die IBM 704 stellte der Hersteller den Programmierern einen Fortran- und einen Lisp-Compiler zur Verfügung.

7030

Mit der IBM 7030 vollzog der Computerriese im Jahr 1960 den Übergang zur Transistortechnik. Die IBM 7030 war ein Dezimalrechner, der 27.000 Operationen pro Sekunde ausführen konnte, einen Speicher von bis zu 9000 Worten besaß und zu einem Preis von etwa 813.000 US\$ verkauft wurde.

Ein Programm, das auf einer IBM 701 geschrieben wurde, war inkompatibel zu einer IBM 702. Wenn also ein Unternehmen wuchs und nach einigen Jahren einen leistungsfähigeren Rechner benötigte, konnten die alten Programme nicht mehr genutzt werden. Da die Firmen zu dieser Zeit nicht auf Standardprogramme zurückgreifen konnten, entwickelte jedes seine eigenen Anwendungen. Irgendwann war die Software so aufwendig, dass die Erstellung dieser teurer war als die Anschaffung einer neuen Maschine. Der kommerzielle Erfolg der IBM 700/7000-Serie und die Vielfalt von Anwendern mit ihren zahllosen Programmen führten zu einer Softwarekrise. Inzwischen war viel Aufwand in die Entwicklung der Anwenderprogramme geflossen, und diese Arbeit musste mit der Anschaffung einer neuen, besseren Hardware immer wieder erbracht werden.

360-Serie

Aus diesem Grund investierte IBM Anfang der 1960er-Jahre etwa 5 Mrd. US\$, um eine universell einsetzbare Rechnerfamilie zu bauen. Dabei sollten Programme zwischen den Maschinen ausgetauscht werden können. Alle Rechner der

1.4 · Rechner in der Datenverarbeitung

Familie IBM System/360 (Abb. 1.23) waren daher binär-kompatibel. Zusätzlich konnten alle für die Baureihe entwickelten Peripheriegeräte wie Drucker und Magnetbandspeicher an jeden Rechner angeschlossen werden. Die erste Maschine kam 1964 auf den Markt. Es gab Rechner mit 32-Bit- und mit 64-Bit-Gleitkommaarithmetik. Alle Register der IBM-360-Familie waren universell und die Computer verwendeten erstmals die indizierte Adressierung. Damit konnte ein 16 MB großer Adressraum im Hauptspeicher angesprochen werden. Das machte die Programme unabhängig vom physischen Speicher und beliebig in diesem verschiebbar. Spätere Exemplare der 360-Familie hatten dann sogar einen vollständig virtuellen Hauptspeicher. Im Gegensatz zur 6-Bit-Zeichencodierung auf älteren Computern wurde nun das erste Mal eine Codierung im 8-Bit-Format eingeführt. Die Mikroprogramme der 360er-Architektur waren auf Lochkarten gespeichert. Diese bestanden aus Metall und waren fest im Mikroprogrammspeicher eingebaut, denn günstige Halbleiterspeicher waren zu der Zeit noch nicht verfügbar. Auf den Maschinen lief das Betriebssystem OS/360, dem man nachsagte, es habe konstant 1000 Fehler: Für jeden alten Fehler, der entfernt wurde, kam ein neuer hinzu. Bis 1968 wurden von der IBM-360-Familie 14.000 Maschinen verkauft. Der erste Rechner von 1964, das Modell 30, konnte 34.500 Befehle pro Sekunde abarbeiten und hatte einen Speicher von 8 kB in der kleinsten bis hin zu 64 kB in der größten Ausführung und entsprach somit in seiner Speicheraus-



Lothar Schaack, Wikimedia Commons

Abb. 1.23 Elektronische Datenverarbeitung mit der IBM 360

stattung den ersten PCs in den 1980er-Jahren. Die IBM-360-Familie stellt den Übergang von mit diskreten Transistoren aufgebauten Maschinen zu Rechnern mit integrierten Schaltungen dar.

1.4.2 Minicomputer von DEC

Die von Ken Olsen (1926–2011) gegründete Firma Digital Equipment Corporation (DEC) begann im Jahr 1957, Komponenten für Rechenanlagen zu produzieren. Auf Grundlage dieser Bauteile entwickelte DEC im Jahr 1959 einen ersten Rechner: die Programmed Data Processor 1 (PDP-1). In den 1950er-Jahren nannte die IBM ihre Produkte Computer. Um sich von diesem Produktnamen zu unterscheiden, wurde von DEC die Bezeichnung Programmed Data Processor oder Programmable Data Processor gewählt. Die Geschäftsidee des Unternehmens war, günstigere Rechner zu verkaufen als die damals von IBM angebotenen Mainframe-Computer. Der Programmable Data Processor kostete mit etwa 100.000 US\$ nur ein Zehntel verglichen mit den etwa 1.000.000 US\$, die die Kunden für die Mainframes von IBM zu zahlen hatten. Die PDP-1 basierte zu großen Teilen auf der von Ken Olsen gebauten TX-0. Nachfolger der PDP-1 war die PDP-4. PDPs von DEC wurden später auch Minicomputer genannt, um sie von den damals üblichen Großcomputern abzugrenzen. Wegen des günstigen Preises waren diese Rechner in Forschungseinrichtungen beliebt. Der Erfolg der Minicomputer führte dazu, dass DEC bis weit in die 1980er-Jahre nach IBM der zweitgrößte Computerhersteller der Welt war. Bedingt durch proprietäre Software, die nur auf die Produkte von DEC zugeschnitten war, und die zunehmende Leistungsfähigkeit der auf der 80×86 -Architektur von Intel basierenden Personal Computer, wurde DEC 1998 Teil des Personal Computer (PC)-Herstellers Compaq. Diese Firma wurde schließlich von Hewlett-Packard (HP) übernommen.

PDP-7

Mit der PDP-7 brachte DEC im Jahr 1964 einen Mini-Rechner mit 18 Bit Wortbreite und einem zur PDP-4 kompatiblen Befehlssatz auf den Markt. Sie kostete nur 72.000 Dollar. Aufgrund dieses niedrigen Preises ist die PDP-7 ein Meilenstein in der Geschichte der Computer. Nun konnten sich auch kleine Forschungseinrichtungen oder einzelne Institute einen Rechner anschaffen. So schrieben der Mathematiker Dennis Ritchie (1941–2011) und der Informatiker Kenneth Thompson (*1943) auf einer PDP-7 in einer Forschungseinrichtung der Telefonfirma (AT&T) das Betriebssystem Multics. Das Projekt Multics scheiterte am Ende, und Ritchie und Thom-

son entwickelten daraufhin auf dieser Basis das Betriebssystem Uniplexed Information and Computing Service (UNIX).

Zwischen 1970 und 1990 wurde von DEC die PDP-11 hergestellt und vertrieben. Der Computer hatte eine Wortbreite von 16 Bit, einen adressierbaren Hauptspeicher von 128 kB und war wegen eines universellen Buskonzeptes leicht erweiter- und aufrüstbar. In der Forschung und an naturwissenschaftlichen Instituten war die Maschine sehr beliebt. Im Gegensatz zu älteren Architekturen kannte die PDP-11 keine gesonderten Befehle für die Ein- und Ausgabe. Die Ansteuerung der Peripherie erfolgte direkt über den Speicher („memory mapping“) und den universellen Bus. Auf einer PDP-11 erweiterte Dennis Ritchie die Sprache B, die selbst aus der am MIT entwickelten Programmiersprache Basic Combined Programming Language (BCPL) hervorging, um Datentypen. Mit dieser zunächst New B (NB) genannten Sprache wurde ab 1972 UNIX auf einer PDP-11 reimplementiert. Diese Programmiersprache wurde später unter dem Namen C standardisiert und gilt als grundlegend für die hardwarenahe Programmierung. Die Sprache C wird vor allem bei der Implementierung von Betriebssystemen und in eingebetteten Systemen eingesetzt. Bemerkenswert ist, dass heute noch Kernreaktoren in von General Electric gebauten Kraftwerken mit der PDP-11 betrieben werden.

PDP-11

Mit der Virtual Address eXtension-Architektur (VAX-Architektur) wurde 1977 erstmals das Konzept der virtuellen Adressen in Minicomputern eingeführt. Geplant war zunächst ein Entwurf, der die PDP-11 um einen virtuellen Adressraum erweitert. Allerdings handelte es sich bei der später kommerziell verkauften Maschine um eine komplette Neuentwicklung. Die mit dem Betriebssystem Virtual Memory System (VMS) ausgelieferten Maschinen konnten in einem 32-Bit-Adressraum bis zu 4 GB adressieren. Zunächst basierten die Maschinen auf einem „Complex-instruction-set-computer (CISC)-Mikroprozessor“ der Firma Motorola (68K). Später entwickelte DEC jedoch eigene „Reduced-instruction-set-computer (RISC)-Prozessoren“, die in den MicroVAX genannten Systemen zum Einsatz kamen. Diese mit der Alpha-Prozessorarchitektur ausgestatteten Rechner wurden häufig als Computer-aided Design (CAD)-Workstations für den Entwurf integrierter Schaltungen eingesetzt. Noch heute ist die VAX-Architektur im Bordcomputer der Kampfflugzeuge F-15 Eagle und F-18 Hornet als eingebettetes System im Einsatz.

VAX

1.5 Die Zahlenfresser: Supercomputer

Die elektronische Datenverarbeitung zeigte, dass es möglich ist, große Rechenanlagen wirtschaftlich zu betreiben. Aller-

Ab 1964

dings waren kommerzielle Maschinen zu dieser Zeit für kaufmännische Aufgaben optimiert. Zunächst setzte man klassische Mainframe-Computer auch für wissenschaftliche Anwendungen ein. Es zeigte sich aber sehr schnell, dass die Maschinen mit dem numerischen wissenschaftlichen Rechnen schlecht ausgenutzt wurden und die einzelnen Arbeitsaufträge (Rechenjobs) viel Rechenzeit verschlangen. Insbesondere die in der Numerik notwendigen Operationen auf Gleitkommazahlen erforderten andere Rechnerarchitekturen als die bereits in kaufmännischen Computern implementierten. Schon während der Entwicklung der ersten Atombomben im Manhattan-Projekt und später bei der Durchführung von Rechnungen in den Raketen- und Raumfahrtprogrammen zeigte sich, dass zur Simulation naturwissenschaftlicher Vorgänge Maschinen benötigt wurden, die in kurzer Zeit große Mengen an Gleitkommaoperationen („mega floating-point operations per second“ [MFLOPS]) ausführen konnten.

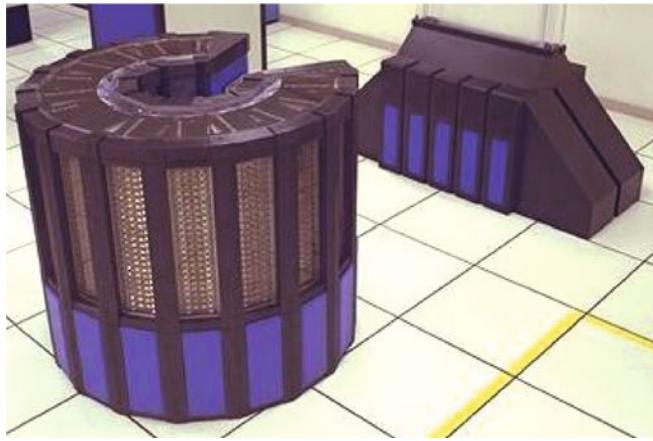
Der erste Rechner, der die Bezeichnung Supercomputer oder Zahlenknacker (engl. „number cruncher“) trug, war die von der amerikanischen Firma Control Data Corporation (CDC) entwickelte CDC 6600. Die Firma CDC baute mit einem Team von Wissenschaftlern, die im Zweiten Weltkrieg deutsche und japanische Verschlüsselungen geknackt hatten, die ersten Computer für die amerikanische Marine (United States Navy): das Navy Tactical Data System (NTDS). Chefentwickler des Marinesystems war damals Seymour Cray (1925–1996). Cray hatte Anfang der 1960er-Jahre die Idee, die bis dahin üblichen Germaniumtransistoren durch erste im Planarprozess gefertigte Siliziumtransistoren zu ersetzen: Das Ergebnis war die CDC 6600. Die erste Maschine wurde an den Conseil européen pour la recherche nucléaire (CERN) (offiziell: European Organization for Nuclear Research) geliefert und wurde zur Analyse der Elementarteilchenbahnen in Nebelkammern eingesetzt. Sie war etwa 3-mal schneller als gewöhnliche Mainframe-Computer zur Datenverarbeitung und wurde neben dem CERN auch in amerikanischen Nukleareinrichtungen verwendet. Nachfolger der CDC 6600 war die CDC 7600; aus 4 Exemplaren dieser Baureihe entstand die CDC 8600, der erste Vektorrechner der Geschichte. Um einen derartigen Rechner zu konstruieren, machte sich Seymour Cray die Beobachtung zunutze, dass in numerischen Anwendungen der Naturwissenschaften derselbe Befehl immer wieder auf unterschiedlichen Daten operierte. Damit konnte eine Operation gleichzeitig auf voneinander unabhängigen Zahlen angewendet werden („single instruction, multiple data“ [SIMD]). Mit mehreren parallel aufgebauten Rechenwerken konnte dann ein Datenvektor schnell abgearbeitet werden.

Nachdem Seymour Cray die CDCs mit einem Team von über 60 Ingenieuren entwickelt hatte, verließ er die Firma und gründete die Cray Research Inc. Der erste von diesem neuen Unternehmen gebaute und vermarktete Supercomputer war die Cray-1. Im Jahr 1976 kaufte das Los Alamos National Laboratory erstmals eine derartige Maschine und setzte den Rechner in der Atomwaffenforschung ein. Die Cray-1 war ein Vektorrechner mit einer Leistung von 160 MFLOPS. Eine Cray-1 kostete damals etwa 8,8 Mio. Dollar. Die Maschine wog 5,5 t und hatte eine elektrische Leistungsaufnahme von 115 kW. Von der Cray-1 wurden etwas mehr als 80 Exemplare weltweit verkauft. Nachfolger der Cray-1 war die Cray X-MP, die 1982 auf den Markt kam. Die Logikgatter in den Maschinen von Cray wurden im Gegensatz zu den Rechnern von CDC, die mit Metal-oxide-semiconductor (MOS)-Transistoren arbeiteten, in ECL (ECL)-Bipolartechnik („emitter-coupled logic“) gefertigt. Diese Logikfamilie war in ihrer Schaltgeschwindigkeit den damals in Transistor-Transistor-Logik (TTL)-Technik gefertigten Mainframes und den in MOS-Technik aufgebauten CDCs weit überlegen. Die ECL-Logikgatter arbeiteten nach dem Prinzip von Differenzverstärkerstufen. Dadurch wurden harte und abrupte Schaltflanken vermieden. Dieses Prinzip und der hohe Strom, der von Bipolartransistoren getrieben werden kann, machten die Schaltungen schnell, führten aber gleichzeitig zu einer starken Leistungsaufnahme. Hinzu kommt, dass in ECL gebaute Computer wesentlich teurer in der Anschaffung sind.

Cray-Supercomputer

1985 kam die Cray-2 (Abb. 1.24) auf den Markt. Die Cray-2 hatte eine Rechengeschwindigkeit von 1,9 GFLOPS („giga floating-point operations per second“). Neu an der Maschine war, dass die Logikchips auf den Platinen in einer speziellen 3-D-Technik angeordnet waren, also Chips direkt auf anderen, die im gleichen Gehäuse angebracht wurden. Das verkürzte die Signalwege stark. Eine andere Innovation der Maschine war, die Kühlung nicht mit Rohrleitungen im Rechner auszuführen, sondern die Platinen so aufzubauen, dass die Chips in der Kühlflüssigkeit baden konnten. Dadurch wurden die Schaltungen komplett umströmt. Die Cray-2 war der schnellste Computer ihrer Zeit. Ihre Rechenleistung entspricht in etwa der eines iPad 2 von Apple aus dem Jahr 2012. Seymour Cray war Anfang der 1990er-Jahre der Meinung, dass schnelle Computer in Zukunft nur in der Halbleitertechnik Galliumarsenid (GaAs) gefertigt werden können, da in dieser Halbleiterlegierung die Elektronenbeweglichkeit wesentlich größer ist als in Silizium. Ende des 20. Jahrhunderts wurden vorwiegend Hochfrequenzschaltungen mit Halbleitern auf GaAs-Basis hergestellt. Allerdings irrte Cray: Mikroprozessoren in Standardsiliziumtechnik wurden immer günstiger und schnel-

Cray-2



NASA, Wikimedia Commons

■ Abb. 1.24 Cray 2 Number Cruncher

ler. Daher lohnte es sich irgendwann nicht mehr Supercomputer aus Spezialkomponenten aufzubauen, da die Kosten für eine Halbleiterfertigung mit den abnehmenden Strukturgrößen rasant angestiegen waren. Es war einfach unwirtschaftlich schnelle Spezialschaltungen für geringe Stückzahlen zu fertigen, um lediglich ein wenig schneller zu sein, als vielleicht die nächste Generation an Siliziumprozessoren.

Multiprozessorrechner

Hatte die Cray-2 noch 8 parallel arbeitende Prozessoren, begannen japanische Firmen Ende der 1980er-Jahre, Supercomputer (■ Abb. 1.25) aus vielen Standardmikroprozessoren aufzubauen. Die National Electrical Code (NEC) SX-3/44R kam 1989 auf den Markt und war 1990 der schnellste Computer der Welt. Er bestand aus 4 Prozessoren. Bereits im Jahr 1994 brachte Fujitsu eine Maschine mit einer Rechenleistung von 1,7 GFLOPS auf den Markt, und dieser Rechner war aus 166 einzelnen Vektorprozessoren hergestellt. 1996 verbaute Hitachi 2048 Prozessoren in der Hitachi SR2201 und erreichte damit eine Rechengeschwindigkeit von 600 GFLOPS. Inzwischen setzte sich eine andere Architektur auf dem Markt für Supercomputer durch: Programmed Data Processor 1 („multiple instruction, multiple data“ [MIMD]). Auch Cray selbst entwickelte Mehrprozessormaschinen: Die Cray T3E mit über 2000 Prozessoren kam im Jahr 1995 auf den Markt. 1998 erreichte eine Cray T3E mehr als 1 TFLOPS („tera floating-point operations per second“). 1999 startete IBM ein Projekt zur Entwicklung von Supercomputern, die die magische Grenze von 1 PFLOPS („peta floating-point operations per second“) durchstoßen sollte. 2004 schaffte man mit einem aus 1024 Prozessoren bestehenden Rechner 70,72 TFLOPS und im



NASA, Wikimedia Commons

■ Abb. 1.25 Columbia Supercomputer in der NASA Advanced Supercomputing Facility

Jahr 2008 gelang es einer auf IBM-Cell-Prozessoren basierenden Maschine am Los Alamos Laboratory, 1 PFLOPS zu erreichen. Die von IBM in diesem Projekt entwickelte skalierbare Architektur wurde Blue Gene/L genannt. Inzwischen (Stand Ende 2019) erreicht der schnellste Computer der Welt, der US-amerikanische IBM Summit, eine Rechengeschwindigkeit von 148,6 PFLOPS. Damit berechnet ein derartiger Supercomputer in 1 h die Menge an Rechenoperationen, für die herkömmliche Desktop-Rechner etwa ein Vierteljahrhundert benötigen würden. Diese gewaltige Maschine besteht aus rund 9200 Prozessoren, deren jeweils 22 Kerne mit einer Taktfrequenz von 1 GHz arbeiten. Der Rechner liegt preislich im mittleren 3-stelligen Millionenbereich und benötigt dabei 10 MW elektrische Leistung.

1.6 Mikrocomputer

Im Auftrag eines japanischen Herstellers für Tischcomputer entwickelte die Firma Intel im Jahr 1971 eine universell programmierbare integrierte Schaltung. In den frühen 1970er-Jahren wurden Tischrechner mithilfe spezieller monolithischer Schaltungen aufgebaut und waren aufgrund der festen Verdrahtung dieser Bausteine sehr unflexibel. Wollte man einen solchen Rechner modifizieren, musste wieder eine neue Schaltung entworfen und gebaut werden. Würde man jedoch eine komplette frei programmierbare Recheneinheit auf einem ein-

Ab 1972

zigen Chip integrieren, wäre es möglich, die gleiche Hardware für verschiedene Produkte einzusetzen. Lediglich die Programmierung müsste dann geändert werden. Da der japanische Tischcomputerhersteller den Prozessor am Ende der Entwicklung nicht abnahm, vermarktete Intel diesen ersten Mikroprozessor selbst. Beim Intel 4004 handelte es sich um eine vollständige Rechnerzentraleinheit auf einem Chip. Intel nannte die integrierte CPU (central processing Unit) Mikroprozessor. Kurz nach dem 4004 führte Intel eine 8-Bit-Variante, den 8008, ein. Da ein Rechner mit 8 Bit nur einen Adressraum von 64 kB unterstützen kann, folgte dem 8008 schnell der 8086, zunächst mit einem auf 8 Bit gemultiplexten 16-Bit-Adressbus und später mit 16 parallelen Adressleitungen. Zu der Zeit standen Logikschaltungen nicht im Fokus der Firma Intel. Um den ersten integrierten Prozessor zu vermarkten, setzte der Intel-Vertriebschef auf eine damals neue Strategie: Intels Vertriebsingenieure verschenkten bei ihren Besuchen in den Entwicklungsabteilungen der Elektronikindustrie Entwicklungsplatinen mit den neuen Chips. Die dortigen Ingenieure spielten mit diesen Platinen und programmierten eigene Anwendungen. Schnell erkannten sie den Nutzen und entwickelten neue Mikroprozessorschaltungen. Neben den Mikroprozessoren von Intel waren bald zahlreiche andere Chips am Markt verfügbar: die damals oft verwendeten Prozessoren Z80 von Zilog oder der von Rockwell Semiconductors gefertigte 6502 (Apple II, Commodore C64). Auf der Grundlage der von Motorola eingeführten Prozessorfamilie 68000 wurden zunächst Apples Lisa und später der erste Macintosh-Computer gebaut.

In den 1970er-Jahren verwendeten viele Bastler Mikroprozessorchipsätze, um selbst Computer zu bauen. Für diese Computer wurden in Hobbyelektronikzeitschriften Schalt- und Baupläne veröffentlicht. Bald begannen einige dieser Bastler die Bausätze ihrer Computer zu vertreiben. Diese enthielten zunächst nur die Platine des Rechners und alle benötigten Bauenteile. Man musste den Rechner dann noch selbst zusammenlöten.

Apple I

Der erste vollständig bestückt verkauft Einplatinencomputer für Amateure, war der von Steve Wozniak (*1950) entwickelte Apple I. Zusammen mit seinem Freund Steve Jobs (1955–2011) fertigte er die Geräte 1976 in einer Garage und verkaufte die Rechner an Elektronikhändler. Diese fügten Tastatur, Netzteil und Bildschirm hinzu und fertig war ein Computer. Einen solchen eigenen Rechner konnte man dann zu Hause als Hobby programmieren. Der Apple I bestand aus einem 6502-Prozessor, einem 4 kB großen Hauptspeicher, und es konnte ein Kassettenrekorder als Massenspeicher angeschlossen werden. Über das Magnetband ließ sich ein Beginner's

All-purpose Symbolic Instruction Code (BASIC)-Interpreter in den Speicher der Maschine laden. Dadurch konnten sogar Schüler einfache Programme in einer Hochsprache schreiben. Während die Firma Commodore mit den auf dem Mikroprozessor 6502 basierenden Heimcomputern, dem VC-20 und dem C64, Kinderzimmer ausrüstete, entwickelte Apple den Apple II (Abb. 1.26). Durch diese etwa 2000 Dollar teuren Geräte hielt der Computer Einzug auf den Schreibtisch in den klassischen Unternehmen. Geschäftsleute nutzten die von Apple zur Verfügung gestellten Programme zur Tabellenkalkulation und Textverarbeitung. Der Apple II war der letzte, ausschließlich von einer einzigen Person entwickelte kommerzielle Mikrocomputer. Im Gegensatz zum Apple I wurde der Rechner nicht als Bausatz, sondern als vollständiges Gerät angeboten. Gegen Mitte des Jahres 1980 wurde als Nachfolger des Apple II der Apple III vorgestellt. Bedingt durch den hohen Preis, die Inkompatibilität zum Apple II und seine technischen Unzulänglichkeiten verkaufte sich der Apple III schlecht. Dadurch entstand oberhalb des Marktsegments für den Apple II eine Lücke.

Diese Lücke wurde vom IBM Personal Computer (IBM-PC) (Personal Computer) besetzt. Nachdem sich Mitte der 1970er-Jahre ein Markt für Spiel- und Heimcomputer entwickelt hatte, war der große Computerhersteller IBM durch diese Entwicklung gezwungen, schnell ein eigenes Produkt anzubieten. Mit dem IBM-PC auf Grundlage des Intel 8086-Mikroprozessors und dem von Microsoft bereitgestellten Betriebssystem (Microsoft Disk Operating System [MS-DOS]) erhielten die zunächst von kleinen dynamischen Unternehmen wie Apple hergestellten Computer den Ritterschlag, da jetzt selbst der Marktführer im Computergeschäft in das PC-Geschäft einstieg.

1983 wurde mit Apples Lisa der erste PC mit grafischer Benutzeroberfläche und 1984 der günstigere Macintosh auf den Markt gebracht. Maus und grafische Fenstersysteme eroberten die Computerwelt, und 1989 war mit der Version Windows 3.1 von Microsoft auch der PC fensterfähig. Da die Architektur von IBM offen gelegt war und Microsoft MS-DOS und Windows an jeden interessierten Hardwarehersteller verkaufte, setzte sich die Intel 8086-Architektur als Quasistandard im Bereich der Personal Computer durch. Dem 8086 folgten die Prozessoren 80186, 80286, 80386 und 80486. In den 1990er-Jahren wurde diese Architektur in Pentium umbenannt, da man eine reine Zahlenfolge markenrechtlich nicht schützen konnte. Im Gegensatz zur Masse der PC-Hersteller setzte Apple in den Mac-Rechnern die Motorola 68000-Architektur bis hin zum 68040 ein. In den 1990er-Jahren wurde dann als Gegengewicht zur 8086 CISC-Architektur („complex instructi-

Apple II und III



Maury Markowitz, Wikimedia Commons

Abb. 1.26 Apple II

IBMs PC

Apple Macintosh

on set computer“) gemeinsam mit Motorola und IBM eine Power-PC genannte RISC-Architektur („reduced instruction set computer“) entwickelt. Die Ära der Power-PC-Prozessoren bei Apple endete im Jahr 2007 mit dem Wechsel zu Intel-Bausteinen.

1.7 Eingebettete Computer

Der technologische Fortschritt in der Halbleiterfertigung ermöglichte Mikrocomputer, die als Desktop-Computer konzipiert waren und nur noch wenig Platz auf einem Schreibtisch benötigten. Mit dem Aufkommen der Mikroprozessoren konnten Ingenieure Anwendungen realisieren, die früher aus aufwendigen mechanischen Konstruktionen bestanden oder für die komplexe analoge Schaltungen notwendig waren. Da sich Software über die Lebensdauer eines Produkts leichter ändern lässt als mechanische oder analoge Hardware, hielten Computer auf der Grundlage von integrierten Prozessoren Einzug in viele Arten von Fahrzeugen: als Bordcomputer in Flugzeugen zur Fluglageregelung und Flugführung, als Steuerung für Verbrennungsmotoren im Kraftfahrzeug oder zur Kontrolle riesiger SchiffsDiesel. Systeme wie das Antiblockiersystem (ABS) oder die elektronische Fahrdynamikregelung (elektronisches Stabilitätsprogramm [ESP] oder „electronic stability control“ [ESC]) erhöhten zudem die Sicherheit von Automobilen. Computer, die in derartige Produkte des Maschinenbaus integriert werden und die man als normaler Anwender nicht wahrnimmt, werden eingebettetes System genannt. Zwar wäre die Bezeichnung eingebetteter Computer passender, der Begriff eingebettetes System hat sich aber für diese Art der Rechner durchgesetzt. Airbag und elektrische Fensterheber, Zentralverriegelung und computergesteuerte Klimaanlagen führen dazu, dass in manchen Fahrzeugen heute bis zu 100 elektronische Rechner verbaut sind. Solch einen Rechner bezeichnet man als Steuergerät oder auch „electronic control unit“ (ECU). Mittlerweile gehen über 80 % der gefertigten Mikroprozessoren oder Mikrocontroller in den Markt für elektronische Steuerungen. Aber auch Telekommunikationssysteme, das Handy und schnurlose Telefone, ja selbst die Telefonvermittlungsstellen sind solche eingebettete Systeme.



NASA, Archive.org

Abb. 1.27 Apollo Missions Simulator

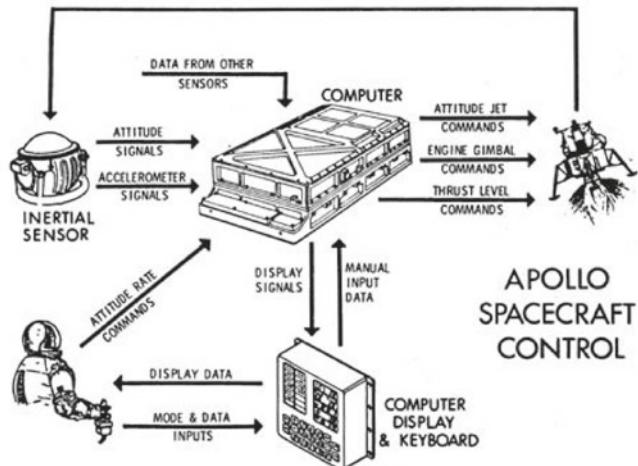
Das Mondlandeprogramm (Abb. 1.27) der amerikanischen NASA war eine enorme technische Herausforderung. Allerdings ging es in dem Programm nicht nur darum, starke Raketen zu bauen. Bereits kurz nach dem Beginn des Apollo-Programms begann man Anfang der 1960er-Jahre, auf IBM-Mainframes die Mondlandung zu simulieren. Dabei stellte man fest, dass alle Astronauten beim Versuch, eine Mondfähre zu landen, scheiterten. Als Konsequenz aus diesen Untersuchungen wurde ein Bordrechner für die Raumschiffe entwickelt. Die Astronauten sollten beim Mondflug und bei der Landung von einem Computer unterstützt werden. Dieser Rechner sollte die Navigation und Flugführung mit manueller Unterstützung der Piloten durchführen. Das Konzept sah vor, die Astronauten von Routineaufgaben zu entlasten und komplexe Flugmanöver rechnergestützt durchzuführen (Abb. 1.28).

Der Computer der Apollo-Missionen wurde aus 7000 „Not-or (NOR)-Gates“ in integrierter Transistorlogik aufgebaut. Der Rechner hatte eine Wortbreite von 15 bit und die Bestimmung der Flugbahn erfolgte in Integer-Arithmetik. Die Flugprogramme waren in einem fest verdrahteten „32-kB-Magnetkern-read-only-memory (ROM)-Speicher“ abgelegt und für die Berechnungen standen 4 kB „random-access memory“ (RAM) als Speicher zur Verfügung. Der Taktzyklus betrug 12 µs. Der Apollo Guidance Computer (AGC) (Abb. 1.29) hatte in etwa die Leistungsfähigkeit der 1980er-Jahre Personal Computer mit 16-bit-Mikroprozessoren. Der Rechner ist insofern bemerkenswert, als dass er erstmals unter strikten Gewichts- und Energieanforderungen gebaut wurde. Der AGC



NASA, NASA.gov

Abb. 1.28 Start von Apollo 11



unbekannt, CalTech.edu

■ Abb. 1.29 Apollo-Navigationsrechner

war auch der erste eingebettete Computer, der die grundlegenden Merkmale heutiger eingebetteter Systeme aufweist: Er hatte bereits ein Betriebssystem mit einem prioritätsbasierten Ablaufplanungsalgorithmus (Scheduling). Die Programme des AGC wurden am Boden auf einem Mainframe-Computer geschrieben und zyklengetreu auf dem Großrechner emuliert. Die Astronauten konnten so bereits im Training alle Flugmanöver in einem Simulator üben. Dieses Vorgehen nennt man eine Hardware-in-the-Loop-Simulation, die heute in der Luftfahrt- und Automobilindustrie zum Stand der Technik bei der Entwicklung eingebetteter Software gehört. Nach dem Apollo-Programm wurde der AGC von der NASA zu Versuchszwecken in ein Kampfflugzeug vom Typ F-8 Crusader eingebaut. Der Rechner ist damit der Urahn aller heutigen Flugzeugbordcomputer und in seinem grundlegenden technischen Aufbau und seiner Hardware-/Softwarearchitektur der Vorgänger fast aller in der Automobilindustrie eingesetzten ECUs.

Mikrocontroller

Ein vollständiger Mikrocomputer benötigt neben der CPU auch Speicher und Peripheriechips. Da eingebettete Computer möglichst klein und kompakt aufgebaut werden, entwickelte sich mit zunehmender Integrationsdichte eine eigene Klasse von Halbleiterprodukten, die Mikrocontroller. Mit Zunahme der Transistordichte verfolgte man in der Chipindustrie zwei unterschiedliche Wege: Zum einen wurden die von Chipgeneration zu Chipgeneration mehr zur Verfügung stehenden Transistoren genutzt, um den Adressraum und die Wortbreite der Prozessoren (8, 16, 32 bit) zu vergrößern und somit immer

leistungsfähigere Prozessoren zu bauen. Zum anderen konnte man sich aber für manche Anwendungen auf Geräte in 8-Bit-Architektur beschränken. Bei Verwendung bekannter 8-Bit-Mikroprozessoren in Kombination mit leistungsfähiger Elektronik zur Ein- und Ausgabe und mit integrierten Speichern zum Ablegen von Daten und Programmen konnten komplettete Rechner auf einem Chip angeboten werden. Diese Chips werden Mikrocontroller genannt. Mithilfe derartiger Bausteine lässt sich ein Steuergerät auf einer Platine aufbauen: Neben dem Computer können analoge Elektronik oder spezielle Schnittstellen zur Ansteuerung der jeweiligen Maschinen platziert werden. Mikrocontroller können heute umfangreiche E/A-Schnittstellen vom Analog-Digital-Wandler bis zu Ethernet- und WLAN-Schnittstellen bieten. Inzwischen gibt es eine große Vielfalt dieser Bausteine am Markt und spezielle Produkte für die Steuerelektronik in Raketen, Flugzeugen oder Autos; Mikrocontroller für weiße Ware (Waschmaschinen, Kühlschränke usw.). Es gibt auch spezielle Mikrocontroller zur Ansteuerung von Festplatten. Immer wenn man eine Maschine oder ein anderes physikalisch-technisches System steuern oder regeln möchte, kann man sich am Markt einen günstigen Mikrocontroller beschaffen, eine einfache Platine mit wenigen Komponenten entwickeln und die Funktion dieses eingebetteten Systems in Software realisieren. Gängige Mikrocontrollerfamilien sind Infineons Tricore-Familie, die PIC-Mikrocontroller und zahlreiche auf der ARM-Architektur basierende „Ein-Chip-Systeme“ (System on a Chip).

Mittlerweile werden nicht nur Ein-Chip-Computer gebaut, sondern ganze Systeme auf einem Mikrochip integriert. Neben dem Mikroprozessor und den Peripheriebausteinen für den Rechner findet man heute Hardware zum Codieren oder Verschlüsseln, Analog-Digital-Wandler, Basisbandschaltungen der Telekommunikation und komplettete Kommunikations-schnittstellen für WLAN und Bluetooth auf einem einzigen Chip. Mittlerweile sind eingebettete Systeme nicht nur mit einem Prozessorkern, sondern mit mehreren Prozessoren und Signalverarbeitungsprozessoren („digital signal processor“ [DSP]) ausgestattet. Solche integrierten Systeme nennt man „System on a Chip“ (SoC). Beispiele sind der A11-Chip im iPhone oder der Cell-Prozessor in der Playstation von Sony. Weniger bekannt sind komplettete Chipsets für Universal Mobile Telecommunications System (UMTS), Long Term Evolution (LTE) und Digital Enhanced Cordless Telecommunications (DECT), um nur ein paar bekannte Mobilfunkstandards zu nennen.

System on a Chip (SoC)

1.8 Halbleiter und die Entwicklung der Rechenleistung



TI, Courtesy of Texas Instruments

Abb. 1.30 Jack Kilby

Die Schrift ist seit etwa 4000 Jahren bekannt. Belegbare Nachweise der ersten Mathematik gehen auf das alte Ägypten zurück. Dort wurden Pyramiden gebaut und nach jeder Nilflut mussten die Feldgrößen für die Bauern neu bestimmt werden. Schon im Altertum träumten die Menschen davon, die Bahn der Sterne zu berechnen, um Vorhersagen über das Erscheinen von Himmelsphänomenen machen zu können. Erste mechanische Rechenmaschinen tauchten bereits in der Antike auf. Aber erst mit dem Aufkommen der modernen Naturwissenschaft zurzeit Newtons beschäftigten sich Mathematiker, Physiker und später Ingenieure mit dem Bau universeller Rechner. Die Industrialisierung und das damit zusammenhängende Bevölkerungswachstum führten zur mechanischen Datenverarbeitung. Allerdings waren mechanische Rechner groß und fehleranfällig. Zwar ermöglichte die inzwischen entwickelte Feinmechanik in den 1930er-Jahren die Herstellung kompakter mechanischer Rechenmaschinen wie der Curta, doch blieb bis Anfang der 1970er-Jahre der Rechenschieber das Werkzeug der Naturwissenschaftler und Ingenieure. Die steigenden Ansprüche an die Numerik in Wissenschaft und Technik und der Bedarf, Daten vieler Menschen automatisch zu verarbeiten, führten zu intensiven Bemühungen, immer leistungsfähigere Rechenmaschinen zu bauen. Allerdings zeigte sich schnell, dass der Bau einer Rechenmaschine zeit- und kostenaufwendig ist und jede Anwendung unterschiedliche Anforderungen an den Computer stellt. Nur universell programmierbare Rechner lösen das Problem: Die gleiche Hardware kann durch Programme, also Software, leicht an unterschiedliche Aufgabenstellungen angepasst werden. Der systematische Entwurf, die Konstruktion und die Programmierung sind eine erst ca. 80 Jahre junge Disziplin, und erste Informatikstudiengänge sind in den 1970er-Jahren aufgebaut worden. Doch warum entwickelte sich der Bau von Computern derartig schnell? Neben dem Bedarf an immer besseren und leistungsfähigeren Rechnern ist der Grund in der Halbleiterindustrie zu finden. Mit Einführung des Silizium-Planarprozesses wurde die Entwicklung der notwendigen Chips skalierbar.

Halbleiter

Die moderne Elektronik begann mit der Erfindung und dem Bau von Funkanlagen. Bereits in den ersten Geräten wurden Festkörper zur Erzeugung und Verstärkung elektromagnetischer Wellen verbaut. Allerdings war das Prinzip dieser Halbleiter genannten Kristalle nicht verstanden, und der erfolgreiche Betrieb einer derartigen Anlage glich einem Glücksspiel. Mit einem Stromabnehmer musste ein Operator eine bestimmte Position auf dem jeweiligen Stück Material finden, an

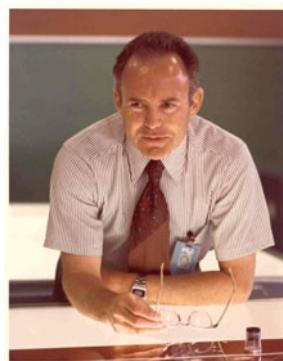
der dieser exakt die zur Verstärkung oder Gleichrichtung notwendigen Eigenschaften hatte. Daher setzte sich in der kommerziellen Nutzung von Funkgeräten zunächst die Röhrentechnik durch. Die Röhre als Bauelement war physikalisch verstanden und konnte exakt so hergestellt werden, wie dies für die jeweilige Anwendung notwendig war. Parallel wurden jedoch grundlegende Forschungen zu Halbleitern betrieben.

Bereits in den 1930er-Jahren wurden der Feldeffekttransistor postuliert und in vielen Forschungslaboren intensiv an seiner Realisierung gearbeitet. Durch die Forschung an den grundlegenden physikalischen Eigenschaften der Grenzflächen von Halbleitern verstanden die Physiker immer besser, wie der elektrische Strom in Festkörpern fließt. Mit Röntgenstrukturuntersuchungen konnte das Verhalten der Festkörper ermittelt, modelliert und berechnet werden. Messungen zur Leitfähigkeit spezieller Halbleiterschichten führten im Jahr 1947 zur Entdeckung des Transistoreffekts. Nun war es möglich, Bipolartransistoren zu bauen und zu vermarkten. Der Feldeffekttransistor war damit zwar noch nicht gefunden, durch den erfolgreichen Verkauf von Halbleiterbauelementen entstand aber eine eigene Industrie. Durch die Einnahmen standen wesentlich mehr Mittel zur Erforschung der Halbleitertechnologie zur Verfügung, und bedingt durch die kommerzielle Konkurrenz investierten die Unternehmen viel in diese Untersuchungen. In den 1960er-Jahren entstanden die ersten Feldeffekttransistoren. Mit den seit 1957 auftretenden integrierten Schaltungen, die unabhängig voneinander von Jack Kilby (1923–2005, □ Abb. 1.30) bei Texas Instruments und vom Intel-Gründer Robert Noyce (1927–1990), damals bei Fairchild Semiconductors, erfunden wurden, konnten dann sogar mehrere Transistoren monolithisch in einem Bauelement realisiert werden.

Den endgültigen Durchbruch zum kleinen Universalcomputer brachte der Anfang der 1960er-Jahre eingeführte Planarprozess. Diese heute noch in der Halbleitertechnik eingesetzte Technologie vereinfachte den Bau integrierter Schaltungen und hatte den Vorteil, skalierbar zu sein. Die grundlegende Technik, eine Schaltung zu integrieren, ändert sich durch Verkleinerungsschritte nur unmerklich. Im Jahr 1965 gelangte der amerikanische Chemiker und Physiker Gordon Moore (*1929, □ Abb. 1.31) auf der Grundlage des Planarprozesses und durch bei der Herstellung gewonnene Daten zu der Erkenntnis, dass sich die Anzahl auf einem Chip integrierbarer Transistoren etwa alle 18 Monate verdoppeln würde. Mehr Schalter bedeuten mehr parallele Rechenoperationen und damit leistungsfähigere Maschinen (□ Abb. 1.32). Gleichzeitig werden kleinere Schaltungen schneller und können mit höheren Taktfrequenzen betrieben werden. Erst diese als zweite in-

Transistoren

Mooresches Gesetz



CHF photographer, Intel Corporation

□ Abb. 1.31 Gordon E. Moore

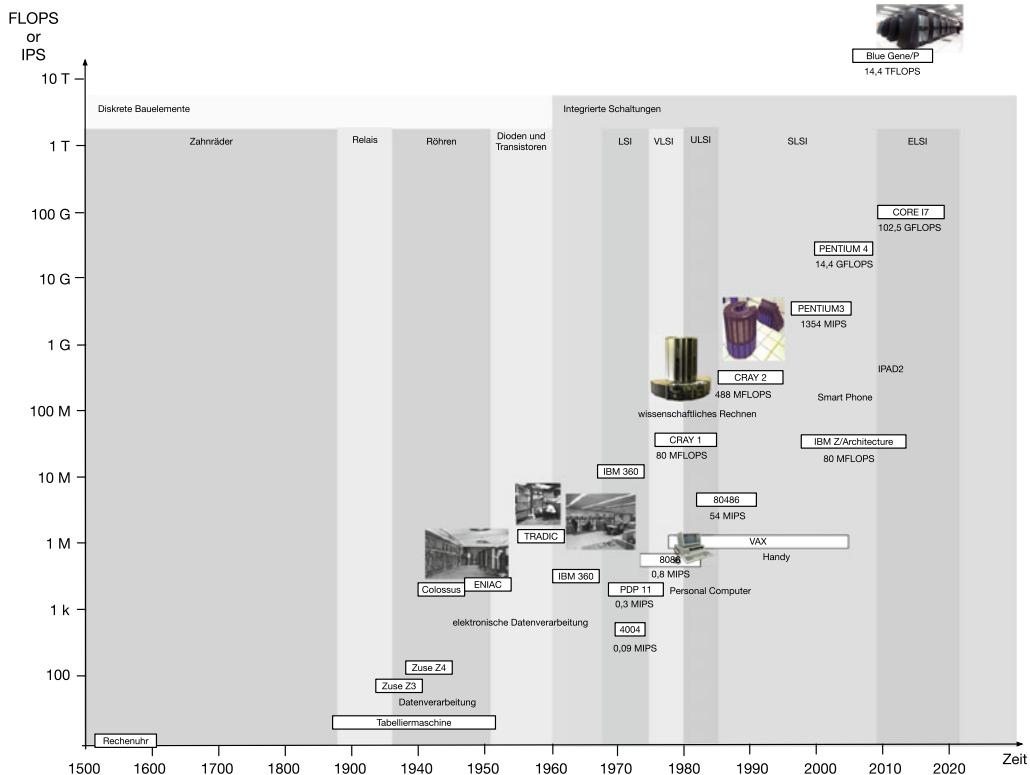


Abb. 1.32 Entwicklung der Rechenleistung

dustrielle Revolution bezeichnete Entwicklung der Mikro- bis zur heutigen Nanoelektronik ermöglicht es, den Taschenrechner oder später das Smartphone zu bauen. Diese elektronischen Geräte, die bezüglich ihrer Rechenleistung jeden Supercomputer aus den 1990er-Jahren weit zurücklassen, wären ohne die Mikroelektronik mit ihren integrierten Schaltungen gar nicht möglich.

?

Übungsaufgaben

Aufgabe 1.1 Erklären Sie am Beispiel einer einfachen Addition die Funktionsweise der Rechenmaschine von Schickard.

Aufgabe 1.2 Was ist ein Rechenschieber? Auf welcher mathematischen Grundlage beruht sein Prinzip?

Aufgabe 1.3 Welche Rolle spielt der Lochstreifen in der elektronischen Datenverarbeitung, und worauf ist sein Prinzip zurückzuführen?

Aufgabe 1.4 Welche Nachteile haben die mechanischen Rechenmaschinen?

Aufgabe 1.5 Welche Rolle spielt der Transistor in der Entwicklungsgeschichte des Computers?

Aufgabe 1.6 Was ist der Unterschied zwischen einem Rechner für kaufmännische und einem für wissenschaftliche Anwendungen? Warum hat man in der Vergangenheit Rechenmaschinen für unterschiedliche Anwendungen gebaut?

Aufgabe 1.7 Was ist das mooresche Gesetz, und welche Bedeutung hat es in der Rechnerarchitektur?

Weiterführende Literatur

- [Her02] Raul Rojas (Herausgeber). *The First Computers: History and Architectures*. MIT Press, 2002. ISBN: 9780262681377.
- [Cer03] Paul E. Ceruzzi. *History of Modern Computing*. MIT Press, 2003. ISBN: 9780262532037.
- [Den09] Robert Dennhardt. *Die Flipflop-Legende und das Digitale. Eine Vorgesichte des Digitalcomputers vom Unterbrecherkontakt zur Röhrenelektronik 1837–1945*. Kulturverlag Kadmos, 2009. ISBN: 9783865990747.
- [Cer12] Paul E. Ceruzzi. *Computing: A Concise History*. The MIT Press, 2012. ISBN: 9780262517676.
- [Dys14] G. Dyson. *Turings Kathedrale: Die Ursprünge des digitalen Zeitalters*. Propyläen Verlag, Auflage: 3. (30. September 2014). ISBN: 978-3549074534. ► <https://books.google.de/books?id=XH0XBAAQBAJ>
- [Mal13] Michael S. Malone. *Der Mikroprozessor: Eine ungewöhnliche Biographie*. Springer, Softcover reprint of the original 1st ed. 1996 (21. Januar 2013). ISBN: 9783662065280.
- [ZBZ10] K. Zuse, F.L. Bauer und H. Zemanek. *Der Computer – Mein Lebenswerk*. Springer, Auflage: 5. unveränd. Aufl. 2010 (17. Mai 2010). ISBN: 9783642120961. ► <https://books.google.de/books?id=cMRNElgwLhUC>



Arithmetik: Zahlen und Rechnen

Inhaltsverzeichnis

- 2.1 Natürliche Zahlen – 43
 - 2.2 Ganze Zahlen – 79
 - 2.3 Reelle Zahlen – 89
 - 2.4 Übertragung und alphanumerische Zeichendarstellung – 125
- Weiterführende Literatur – 133

Mathematik ist das Alphabet, mit dessen Hilfe Gott das Universum gebaut hat.
Galileo Galilei

2

Was sind Zahlen, und wie wird mit ihnen gerechnet? Das Ziel des Kapitels ist es, zu verstehen, wie numerische Berechnungen im Computer durchgeführt werden und wie Ziffern und Symbole gespeichert werden. Insbesondere das duale Zahlensystem ist dabei von Bedeutung, da in heutigen Rechnern Zeichen binär codiert werden. Es ist wichtig, sich zu merken, dass die Subtraktion auf die Addition zurückgeführt wird. Mit einer geschickten Zahldarstellung ist es möglich, negative und positive Ziffernfolgen effizient zu speichern. Nachdem die grundlegenden Prinzipien der Darstellung ganzer Zahlen eingeführt wurden, wird noch erläutert, wie Computer reelle Zahlen approximieren können. Neben der reinen Codierung der Zahlen geht das Kapitel auf die Umrechnung zwischen unterschiedlichen Zahlensystemen ein.

Die Menschen hatten früh das Bedürfnis, mühsames Rechnen von Maschinen automatisch ausführen zu lassen. Dabei bestimmte die jeweils zur Verfügung stehende Technologie, seien es Zahnräder, elektrische Relais, Röhren, Transistoren oder integrierte Schaltungen, wie und für welche Einsatzgebiete welche Automatisierung möglich war. Selbst mit unzureichender Mechanik versuchte man immer wieder, Rechenmaschinen zu bauen. Grundlegend zum Verständnis eines Computers ist die Darstellung von Zahlen. Von der Codierung dieser hängen die Art und Komplexität des Rechnens und damit die Struktur einer Rechenmaschine ab. Daher sind zunächst die Zahldarstellung, die Zahlensysteme und deren Handhabung zu betrachten. Dafür wird auf den aus der Schule bekannten Grundlagen erst die Addition im dezimalen Stellenwertsystem rekapituliert. Aus dieser lässt sich mit einer geeigneten Codierung direkt die Subtraktion ableiten. Basierend auf diesen Prinzipien werden das duale Stellenwertsystem eingeführt und gezeigt, dass Rechnen in diesem Zahlensystem analog zu den bekannten Verfahren aus dem Dezimalsystem ist. Die hohe Bedeutung des binären Stellenwertsystems ergibt sich daraus, dass heute alle Computer in diesem System arbeiten. Im Anschluss an das reine Rechnen führt das Kapitel noch in die gängige Codierung alphanumerischer Zeichen zur Darstellung von Dezimalziffern und Buchstaben ein. Damit lassen sich beliebige Daten in einer elektronischen Maschine verarbeiten, und die Grundlagen zum Verständnis eines modernen Rechners sind gelegt.

2.1 Natürliche Zahlen

Mit der Domestizierung von Schafen und Rindern fingen die Menschen nachweislich an zu zählen. Den königlichen Hirten wurden am Morgen oder im Frühjahr Tiere aus dem Stall zum Weiden anvertraut. Am Abend oder im Herbst mussten dann alle Schafe oder Rinder wieder nach Hause gebracht werden. Auf dieser Grundlage entwickelte sich eine Zahlschrift. Denn die Wachen oder Verwalter des Königs mussten sich über einen längeren Zeitraum merken, wie viele Tiere sie dem jeweiligen Hirten anvertraut hatten.

Es ist zu vermuten, dass bereits früh in der Geschichte des Menschen der Wunsch aufkam, Teile der Jagdbeute gegen gesammelte Früchte oder Waffen zu tauschen. Daraus entwickelte sich dann der Handel mit Waren. Um diesen betreiben zu können, ist es notwendig, zu einer vorhandenen Menge einer Art von Rohstoffen oder Produkten eine weitere hinzuzufügen oder abzuziehen. Dieser Rechnen genannte Vorgang entwickelte sich zunächst einmal auf Basis von Mengen mit abzählbaren Elementen. Dazu wird ein Ausdruck für eine Anzahl benötigt, die Zahl. Mithilfe der natürlichen Zahlen ist es möglich, Schafe, Rinder, Getreidesäcke oder Steine zu zählen und dann zu rechnen.

Um natürlich zu rechnen, muss klar festgelegt werden, welches der kleinste Wert ist: Mit welchem Symbol wird mit dem Zählen begonnen. Ebenso sind die Nachfolgebeziehungen und Relationen der einzelnen Zeichen untereinander festzulegen. Eine weitverbreitete Definition der natürlichen Zahlen sind die 5 Axiome des italienischen Mathematikers Giuseppe Peano (1858–1932, □ Abb. 2.1).



Unbekannt, Wikimedia Commons

□ Abb. 2.1 Giuseppe Peano

Definition 2.1.1 – Natürliche Zahlen

Axiome des Peano:

1. $0 \in \mathbb{N}$
0 ist eine natürliche Zahl.
2. $\forall n \in \mathbb{N} : n' \in \mathbb{N}$
Jede natürliche Zahl hat einen Nachfolger in den natürlichen Zahlen.
3. $\forall n \in \mathbb{N} : n' \neq 0$
0 ist kein Nachfolger einer natürlichen Zahl.
4. $\forall n, m \in \mathbb{N} \wedge n' = m' : n = m$
Zwei Zahlen mit gleichem Nachfolger sind gleich.
5. $\forall X \wedge 0 \in X \wedge (\forall n \in \mathbb{N} : n \in X \wedge n' \in X) : \mathbb{N} \subseteq X$
Die natürlichen Zahlen sind Teilmenge einer Menge X , wenn X die 0 und für jedes $n \in \mathbb{N}$ sowohl n als auch den entsprechenden Nachfolger n' enthält.

Die Grundregeln des Rechnens, das Kommutativgesetz, das Assoziativgesetz, das Distributivgesetz und die daraus abgeleiteten Eigenschaften der Addition sind dem Leser bereits aus der Schule geläufig. Es soll an dieser Stelle nur interessieren, wie die Summe zweier natürlicher Zahlen gebildet wird. Mathematisch leitet sich diese Regel aus den Axiomen des Peano ab:

$$n + 0 = n \quad (2.1)$$

$$n + m' = (n + m)' \quad (2.2)$$

Hinzu kommt die Definition der 1 als Nachfolger der 0:

$$1 = 0' \quad (2.3)$$

Daraus ergeben sich insbesondere $n + 1 = n + 0' = (n + 0)' = n'$ und damit, dass sich der Nachfolger einer natürlichen Zahl durch Addition mit 1 berechnen lässt, was wiederum unserer natürlichen Vorstellung des Zählens von Waren entspricht.

Angenommen, ein Bauer möchte einen Teil seines Landes an einen Kaufmann aus der Stadt verkaufen, und der Preis beträgt 500 Rinder. Wenn der Landmann bereits 200 Tiere besitzt, kann er berechnen, wie viel Vieh er in Zukunft insgesamt versorgen muss. Allerdings verlangt der König vom Bauern Steuern: Von den neu erhaltenen 500 Rindern sind 100 an die Steuerbeamten abzugeben. Sollen die Tiere auf dem Seeweg in die Hauptstadt gelangen, und das zur Verfügung stehende Schiff hat eine Kapazität von 15 Rindern, so müssen die Steuereintreiber berechnen, wie viele Reisen ein Fahrzeug machen muss oder wie viele Segler ggf. für den Transport zu beschlagnahmen sind. Spätestens mit planvoller Landwirtschaft spielten Zählen und Rechnen im Leben der Menschen eine bedeutende Rolle. Um die wachsende Bevölkerung zu ernähren, mussten Felder und Vorratsspeicher angelegt werden. Der König festigte seinen Machtanspruch durch den Bau von Tempeln und monumentalen Gräbern und forderte daher Steuern von den Bauern. Eine solche Kultur führte zwangsläufig eine Schrift und eine Zahlschrift ein.

2.1.1 Darstellung der Zahlen

Die uns heute geläufigen Zahlschriften oder besser Zahlensysteme sind nicht natürlich. Das heutige Rechnen baut auf einer langen Entwicklung auf. Ausgehend vom Zählen wurden primitive Zahlensysteme immer komplexer, von der einfachen Strichliste über die Anforderungen der Kaufleute und Banken

bis hin zu abstrakten mathematischen Verfahren zur Berechnung multidimensionaler Vorgänge in der Natur.

2.1.1.1 Additives Zahlensystem

Die ersten Zahlschriften entwickelten sich für die natürlichen Zahlen; eben die Symbole, die die Anzahl von Tieren oder Gegenständen beschreiben. Eine Zahlschrift besteht aus einer Folge von Zahlzeichen. Da die Anzahl dieser begrenzt ist, entsteht ein unendlich großer Zahlenraum durch Aneinanderreihung der Zeichen. Eine Folge von Zahlzeichen nennt man „Zahlzeichenreihe“ oder „Zahlzeichenfolge“. Die einfachste Form einer Zahlschrift ist eine Strichliste, wobei dabei noch nicht von einem System gesprochen werden kann. Für jede Anzahl von Objekten gibt man eine Folge von Strichen an: für eine Eins 1. Strich und für eine Fünf 5 Striche. Allerdings wird diese Darstellung für große Zahlen sehr schnell unübersichtlich. Eine Verbesserung ist es, nach 4 Strichen mit dem 5. Strich die vorangegangenen 4 Striche durchzustreichen. Mit Einführung dieser Regel ist bereits ein frühes Zahlensystem eingeführt, dass es ermöglicht, Zahlen in 5er-Blöcken mit je 5 Elementen zu gruppieren. Weltweit haben sich unterschiedliche Formen der Strichliste entwickelt. Bei einer wird beispielsweise mit 5 Strichen ein durchgestrichenes Quadrat gezeichnet. In diesem Fall bedeuten dann 1 Strich wieder eine 1, 2 rechtwinklig angeordnete Striche eine 2, ein nach unten offenes Quadrat eine 3, das Quadrat eine 4 und ein mit Querstrich versehenes Quadrat eine 5. Eine weitere bekannte Form ist das strichweise Aufbauen von bestimmten Schriftzeichen.

Zahlschrift

Ein urindianisches Volk, die Zapoteken, die im heutigen Mexiko lebten (Blütezeit 3. bis 9. Jahrhundert), entwickelten schon um 1500 v. Chr. eine Zahlschrift. Dabei wurden die Zahlen von 1 bis 4 durch Punkte und die 5 durch einen Strich dargestellt. Vier Punkte über einem Strich bedeuteten also die 9.

Indianische Zahlschriften

Ägyptische Zahlen		1081086	
	Zahlwerte		
1		Gespann	
10		Schlinge	
100		Lilie	
1000		Finger	
10000		Kaulquappe	
100000		Gott	
1000000			
			1 · 1000000
			0 · 100000
			8 · 10000
			1 · 1000
			0 · 100
			8 · 10
			6 · 1

Abb. 2.2 Altägyptisches Zahlensystem

Hieroglyphen

Das Volk der Mayas entwickelte das Zahlensystem der Zapoteken zu einem Stellenwertsystem weiter.

Eine der ältesten Hochkulturen der Antike, das ägyptische Pharaonenreich, entwickelte zur Verwaltung des Staates eine der ersten Schriften der Menschheitsgeschichte. Bekannt ist uns diese in deren kultischer Anpassung: den noch heute an und in vielen Tempeln und Gräbern von der Zivilisation der Pharaonen kündenden Hieroglyphen. Teil dieser Schrift ist auch eine Zahlschrift. Im alten Ägypten wurde einem Zahlzeichen die Bedeutung einer Potenz zur Basis 10 gegeben. Dadurch entwickelte sich ein erweitertes additives Zahlensystem.

Das Zahlensystem der ägyptischen Hochkultur kannte jede Zeichen für die 1, 10, 100, 1000, 10.000, 100.000 und 1.000.000. Die 1 wurde mit einem Strich codiert, die 10 stellte ein Gespann dar, die 100 wurde mit dem Symbol Schlinge und die 1000 mit einer Wasserlilie beschrieben. Offenbar konnten sich die Ägypter unter den dargestellten Symbolen bestimmte Mengen von Gegenständen ihres Alltags vorstellen. Die Zahl 10.000 wurde mit einem Finger und 100.000 mittels einer Kaulquappe dargestellt. Im Reich des alten Ägyptens war 1.000.000 die größte bekannte Zahl. Da sich die Menschen zu dieser Zeit in Ägypten keine höheren Zahlen vorstellen konnten, war die 1. Million unerreicht göttlich und wurde durch einen Gott symbolisiert.

Die Zahl 1.081.086 wie in (Abb. 2.2) ließ sich folglich mit 1 Zeichen für Gott, 8 Zeichen für Finger, 1 Zeichen für Wasserlilie, 8 Zeichen für Gespann und 6 Zählstrichen darstellen. Ein heute noch gebräuchliches additives Zahlensystem ist das römische. Es unterscheidet sich von den bisher vorgestellten Zahlensystemen insofern, dass Zahlen durch Subtraktion von Ziffern zu beschreiben sind, indem ein Symbol vor ein anderes Zeichen geschrieben wird (Abb. 2.3). Die römische Zahl-

Römische Zahlen

Abb. 2.3 Verschiedene additive Zahlensysteme

schrift ist also mit ergänzender subtraktiver Regel ausgestaltet. Im Gegensatz zu den einfachen additiven Zahlensystemen beinhaltet das lateinische Ziffernsystem Zeichen für die Fünf (V), die Zehn (X) und deren Vielfache (L, C, D, M). Allerdings kannten die Römer keine Ziffer für die Null. Zwar konnte die Null in verbalisierter Form als das Nichts (lat. „nihil“) in ihrer Zählweise oder das Nichtvorhandensein von etwas als „nicht etwas“ (lat. „nullum“) bezeichnet werden. Die Zahlschrift selbst hatte aber kein Zeichen für die Null, insbesondere da sich alle 10er-Potenzen durch eigene Symbole einzeln darstellen lassen. Ebenso war den römischen Gelehrten die Schreibweise von Brüchen bekannt. Additive Zahlensysteme haben den Nachteil, dass Rechnen nur umständlich möglich ist, da Ziffern nicht einzeln behandelt werden können. Für das automatische Verarbeiten von Zahlen kommen diese daher nicht infrage.

In einem derartigen additiven Zahlensystem ist die Addition einfach. Werden neue Steuern der Schatzkammer eines Königs hinzugefügt, müssen die Schreiber nur zusätzliche Zeichen an die alte, bereits vorhandene Zeichenfolge anfügen. Auch beim Subtrahieren müssen dann ausschließlich Ziffern gelöscht werden. Selbst in der römischen Zahlschrift kann dies im Prinzip als Kombination aus Anfügen und Löschen geschehen. Multiplikation und Division dagegen sind in einem additiven Zahlensystem schlecht auszuführen.

2.1.1.2 Polyadisches Zahlensystem

Im präkolumbianischen Amerika entwickelte sich die Hochkultur der Maya. Diese Indianer zählten in 4 5er-Blöcken. Es wird vermutet, dass die frühen Maya ihre Finger und Zehen zum Zählen verwendeten. Mithilfe von zwei Zeichen, einem Punkt und einem Strich ließen sich die Zahlen von 1 bis 19 einfach wie in einem additiven Zahlensystem codieren. Da die Maya bereits die 0 kannten und für dieses Nichts ein eigenes Symbol einführten, konnte spaltenweise untereinander geschriebenen Zahlzeichen eine Bedeutung gegeben werden. Dabei wird im Grunde jeder Zeile ein Vielfaches von 20 zugewiesen, wobei es 3 Positionen mit zusätzlicher Multiplikation des Wertes 18 gibt. Dies geht ebenso wie die Zahl 20 auf den Maya-Kalender zurück, der das Sonnenjahr „Haab“ in 18 Monate mit jeweils 20 Tagen aufteilt. Daraus ergeben sich die Werte für die jeweiligen Stellen wie in □ Tab. 2.1 gezeigt.

Zahlen der Maya

■ Tab. 2.1 Polyadisches Zahlensystem – Werte

Position			Wert
1	20^0		= 1
2	20^1		= 20
3	$20^1 \cdot 18$		= 360
4	$20^1 \cdot 18 \cdot 20$	$= 20^2 \cdot 18$	= 7200
5	$20^1 \cdot 18 \cdot 20^2$	$= 20^3 \cdot 18$	= 144.000

Ein derartiges System heißt polyadisches Zahlensystem. In der Kultur der Maya war ein quadratisches Zeichenmuster göttlich und daher anzustreben. Wie bei den alten Ägyptern hatte die Schrift dieses uramerikanischen Volkes eine tiefen kultischen Bedeutung. In diesen religiös geprägten Gemeinschaften wurde daher wie selbstverständlich ein dem kultischen Stellenwert angemessenes Schriftbild gewählt.

Betrachten wir nachfolgend die Maya-Zahl mit dem Dezimalwert 1.081.086 in (Abb. 2.4). Die 2. Zeile von unten stand für die 20, die 3. Zeile für die Stelle 360, die 4. für die Stelle 7200 und die 5. Stelle für die 144.000. Für diese Kalenderperioden kannten die Maya auch eigene Zeichen, sodass bis auf die Tatsache, dass die Maya im Gegensatz zu den Ägyptern die Zahlen zur Basis 20 darstellten, das ägyptische dem indianischen Maya-Zahlensystem sehr ähnlich war.

Ziffern und Ziffernfolge

Bei einer Zahlschrift wie der der Maya bekommt jeder Zeile eine Bedeutung zu. In den meisten modernen Zahlschriften werden die Zahlzeichen nebeneinander in einer Reihe geschrie-

Maya Zahlen		1081086	
•	Zahlwerte		$7 \times 144000 = 1008000$
..			$10 \times 7200 = 72000$
...			$3 \times 360 = 1080$
....	• 0		$0 \times 20 = 0$
.....	— 1		$6 \times 1 = 6$
.....	— 5		<u><u>1081086</u></u>

Abb. 2.4 Zahlensystem der Maya

ben. Dabei besitzt jede Position oder Stelle in der jeweiligen Reihe eine eigene Bedeutung: den Stellenwert. Die Symbole, die man in einer Stelle verwenden kann, nennt man Zahlzeichen oder Ziffer. Die Zahl ergibt sich damit als Reihe oder Folge von Ziffern. Wie bereits eingeführt, haben die Symbole oder Ziffern eine unterschiedliche Wertigkeit, den Ziffernwert. Da nun der eigentliche Wert einer Ziffer in einer Ziffernfolge sowohl von ihrem Ziffernwert als auch vom Stellenwert abhängt, nennt man ein solches polyadisches (griech. „polus“, viel[wertige]) Zahlensystem Stellenwertsystem.

Die Menge der Symbole in einem Stellenwertsystem ist begrenzt. Diese Anzahl b ist elementar für die Zahlendarstellung, denn sie definiert die sogenannte Basis b der Zahl. Werden 5 Zahlzeichen definiert, ist die Basis $b = 5$, bei 10 Zeichen ergibt sich die Basis $b = 10$ und bei 16 Zeichen die Basis $b = 16$. Die Zahl eines Symbols in einer Spalte bestimmt sich aus der Multiplikation des Ziffernwertes selbst mit dem Stellenwert. Es erfolgt also eine Art Gewichtung der Ziffer mit dem Wert der Stelle. Der Stellenwert ist dabei immer ein festgelegtes Vielfaches der Basis b . Diese Festlegung des Zahlwortes reicht allerdings nicht aus. Das Problem wird mit folgendem Beispiel erläutert:

► Beispiel 2.1.1 – Interpretation der Ziffernfolge

Angenommen, eine alte Hochkultur verwendet die 5 Symbole $\Sigma_5 := \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}\}$ als Zahlzeichen. Die Basis des Systems wäre dann $b = 5$. Eine Zahl in dieser Zahlschrift könnte also wie folgt aussehen: $x_{b5} = \mathcal{B}\mathcal{E}\mathcal{A}\mathcal{C}$. Doch nun stellt sich die Frage, wie die einzelnen Ziffern gewichtet werden. Zunächst einmal macht es Sinn, die unterschiedlichen Vielfachen der Basis nicht beliebig zu streuen, sich also die natürliche Reihenfolge der Stellen im Wort zunutze zu machen. Dann blieben zwei Möglichkeiten, das Zahlwort zu interpretieren:

- $x_{b10} = \mathcal{B} + \mathcal{E} \cdot 5 + \mathcal{A} \cdot 10 + \mathcal{C} \cdot 15 = 82$
- $x'_{b10} = \mathcal{B} \cdot 15 + \mathcal{E} \cdot 10 + \mathcal{A} \cdot 5 + \mathcal{C} = 88$

Die angenommene Reihenfolge der Stellen und damit die entstehende Stellenwertigkeit sind von entscheidender Bedeutung, um das Zahlwort korrekt zu interpretieren. ◀

Das Problem, jeder Stelle im Zahlwort eine eindeutige Wertigkeit zuzuordnen, lässt sich wie folgt lösen: Jede Stelle wird mit genau einem Vielfachen der Basis in Abhängigkeit der Position der Stelle in der Ziffernfolge verknüpft. Dabei gibt es insbesondere eine niedrigstwertige Stelle und eine höchstwertige Stelle im Zahlwort. Die niedrigstwertige Stelle ist die Stelle, bei der keine Vervielfachung auftritt, und die höchstwertige Stelle ist die Stelle, mit der die maximale Vervielfachung mit der Basis auftritt. Die Vervielfachung einer Stelle ergibt sich dann durch ihre relative Position, beispielsweise zur niedrigstwertigen Stel-

Wertigkeit der Stellen

le. Legt man nun fest, ob die Reihe mit der niedrigstwertigen oder höchstwertigen Stelle beginnt, lässt sich das Zahlwort eindeutig interpretieren.

2

Definition 2.1.2 – Zahl als Ziffernfolge

Eine Zahl x ist durch eine Folge von Ziffern $a_\infty \dots a_1 a_0$ (bzw. $a_0 a_1 \dots a_\infty$) wie folgt definiert:

$$x = \sum_{i=0}^{\infty} a_i \cdot b^i \quad (2.4)$$

Dabei ist i die Position der jeweiligen Ziffer und bestimmt deren Stellenwert b^i durch Potenzierung der Basis b mit der Stelle i .

Mit dieser Definition können nun beliebig große natürliche Zahlen gebildet werden. Immer dann, wenn eine höhere Zahl als die gerade noch darstellbare benötigt wird, wird dem Zahlwort eine weitere Stelle hinzugefügt.

► Beispiel 2.1.2 – Darstellung einer Zahl

Die Zahl $x_{b25} = FDCX$ in einem Zahlensystem zur Basis $b = 25$ mit der Symbolmenge Σ_{b25} , wobei $A_{b25} := 0_{b10}$ gilt, und der höchstwertigen Stelle links ergibt:

$$\Sigma_{b25} := \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, V, W, X, Y, Z\}$$

$$x_{b10} = X \cdot 25^0 + C \cdot 25^1 + D \cdot 25^2 + F \cdot 25^3 = 80.072$$

**2.1.2 Zahlensysteme**

Aus heutiger Sicht ist es leicht ersichtlich, dass im additiven Zahlensystem schwer zu rechnen ist. Zwar sind die römischen Zahlen historisch bedingt in Europa stark verbreitet, aber bereits Leonardo Fibonacci (1170–1240) erkannte die Nachteile dieses Systems und die Überlegenheit der indischen Ziffern. Nach verschiedenen Untersuchungen gab der deutsche Rechenmeister Adam Riese (1492–1559) dem aus der indischen Zahlschrift abgeleiteten arabischen Stellenwertsystem den Vorrang beim Verfassen seines zweiten Rechenbuchs.

Definition 2.1.3 – Dezimalsystem

Das Dezimalsystem ist ein Stellenwertsystem zur Basis 10. Die Ziffern werden mit den Symbolen $\Sigma_{b=10} := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ repräsentiert. Die niedrigstwertige Ziffer steht rechts.

Es ist möglich, Zahlen eines beliebigen Zahlensystems in das uns geläufige Dezimalsystem wie folgt umzuwandeln:

Definition 2.1.4 – Bestimmung der Dezimalzahl aus einer Zahl zur Basis b

Ein Zahlwort eines Stellenwertsystems zur Basis b lässt sich in ein Zahlwort des Dezimalsystems umwandeln. Das Zahlwort im Dezimalsystem ist:

$$x_{b=10} = \sum_{i=0}^{\infty} a_i \cdot b^i \quad (2.5)$$

Vergleicht man die Definition zur Umwandlung eines Zahlwortes einer beliebigen Basis b in ein Zahlwort zur Basis $b = 10$ mit Definition 2.1.2, fällt auf, dass die erforderliche Berechnung quasi identisch ist. Dies liegt daran, dass wir für Zahlen im Normalfall Zahlwörter zur Basis $b = 10$ verwenden.

Wir sind mit dem dezimalen Zahlensystem aufgewachsen und haben dessen Verwendung in der Schule geübt. Dies folgt höchstwahrscheinlich daraus, dass wir 10 Finger besitzen und natürlicherweise zunächst mithilfe dieser gezählt wird. Die Beispiele der altamerikanischen Hochkulturen wie die Zapoteken oder die Maya zeigen jedoch, dass unterschiedliche Kulturen verschiedene Zahlensysteme wählen können, je nachdem, auf welcher grundlegenden Annahme das Zählen beruht oder welche kulturellen oder kalendarischen Einteilungen als göttlich angesehen werden.

In den 1950er-Jahren wurden zunächst Rechner auf der Grundlage des Dezimalsystems gebaut, da sich mithilfe von Röhren leicht elektronische Bauteile mit vielen, insbesondere mit 10 Zuständen realisieren lassen. Heutige Computer, deren technologische Basis Transistoren, also Schalter sind, arbeiten allerdings im binären oder dualen Zahlensystem:

Definition 2.1.5 – Dualsystem

Das Dualsystem ist ein Stellenwertsystem zur Basis 2. Die Ziffern werden mit den Symbolen $\Sigma_{b2} := \{0, 1\}$ dargestellt. Die Ziffer mit dem niedrigsten Wert steht rechts oder links.

Die Ziffern einer Binärzahl bezeichnet man häufig als Bits (von engl. „binary digits“). Wie bereits aus der Definition ersichtlich, kann die niedrigstwertige Ziffer – auch least significant Bit (LSB) genannt – entweder auf der linken oder rechten Seite des Zahlwortes stehen. Gerade im Dualsystem finden sich beide Konventionen in unseren Rechnern wieder.

► Beispiel 2.1.3 – Dualzahl

Die 8-stellige Zahl $x_{b2} = 1011\ 1011$ (LSB rechts) hat das dezimale Zahlwort:

$$\begin{aligned}x_{b10} &= \sum_{i=0}^7 a_i \cdot 2^i \\&= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 \\&= 1 + 2 + 8 + 16 + 32 + 128 = 187\end{aligned}$$



Neben dem binären Zahlensystem wird in der technischen Informatik häufig das hexadezimale mit der Basis 16 verwendet. Dies liegt vor allem daran, dass 16 als 2^4 ein Vielfaches der dualen Basis ist und damit als kompakte Darstellung von Zeichen eines Rechners genutzt werden kann: Eine Binärzahl mit 4 Ziffern codiert dabei genau $2^4 = 16$ verschiedene Zustände, beispielsweise die Zahlen 0, 1, …, 15. Dies entspricht genau der Anzahl von Zahlen, die sich auch durch 16 Symbole und damit einer einzigen Ziffer im Hexadezimalsystem darstellen lassen. Für 8 Ziffern im Dualsystem mit 256 darstellbaren Zahlen benötigt man lediglich 2 Ziffern im Hexadezimalsystem. Listings in hexadezimaler Schreibweise sind wesentlich einfacher zu lesen als endlose Folgen aus Nullen und Einsen. Die hexadezimale Codierung eignet sich in der Informatik daher gut zur Darstellung von Speicherinhalten.

Definition 2.1.6 – Hexadezimalsystem

Das Hexadezimalsystem ist ein Stellenwertsystem zur Basis 16. Die Ziffern werden mit den Symbolen $\Sigma_{b16} := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ dargestellt. Die niedrigstwertige Ziffer steht rechts.

► **Beispiel 2.1.4 – Hexadezimalsystem**

Die 4-stellige Zahl $x_{b16} = A0F4$ hat den dezimalen Zahlenwert:

$$\begin{aligned}x_{b10} &= \sum_{i=0}^3 a_i \cdot 16^i = 4 \cdot 16^0 + 15 \cdot 16^1 + 0 \cdot 16^2 + 10 \cdot 16^3 \\&= 4 + 240 + 40.960 = 41.204\end{aligned}$$



Da wir uns an einen natürlichen Umgang mit Zahlen zur Basis 10 gewöhnt haben, Rechner technisch bedingt jedoch beispielsweise Dual- oder Hexadezimalzahlen verwenden, ist es in der Informatik notwendig, Zahlen aus unterschiedlichen Stellenwertssystemen in andere gebräuchliche Stellenwertssysteme umzuwandeln. Besonders häufig sind dabei die Umwandlung und Rückumwandlung von Dezimalzahlen in Dualzahlen und von Dualzahlen in Hexadezimalzahlen.

Konvertierung in andere Zahlensysteme

Definition 2.1.7 – Bestimmung einer Zahl zur Basis b aus einer Dezimalzahl

Jede Dezimalzahl x_{b10} lässt sich in eine Zahl mit einer anderen Basis b wie folgt umrechnen:

$$a_i = \begin{cases} x_{b10} \bmod b & i = 0 \\ \frac{x_{b10} - \sum_{k=0}^{k=(i-1)} a_k \cdot b^k}{b^i} \bmod b & i > 0 \end{cases} \quad (2.6)$$

Diese Vorschrift berechnet die Stellen a_i der Zahl zur Basis b einzeln.

Zur Umrechnung von Dezimalzahlen in Zahlen zur Basis b , insbesondere für die manuelle Berechnung, kann auch nachfolgender Algorithmus verwendet werden. Dieser teilt die Dezimalzahl immer wieder durch die Basis b , und aus dem jeweils verbleibenden Rest ergibt sich die Stelle a_i der gesuchten Zahl zur Basis b :

$$a_i = x_i \bmod b \quad (2.7)$$

$$x_i = \left\lfloor \frac{x_{i-1}}{b} \right\rfloor \text{ mit } x_0 = x_{b10} \quad (2.8)$$

Hierbei macht man sich zunutze, dass uns die wiederholte Division zur Berechnung der Zahlen x_i im Zehnersystem ebenso wie die Restberechnung zur Bestimmung der Ziffer a_i wohlvertraut sind. Zu beachten ist, dass x_i hierbei nicht das i -te Symbol der Zahl, sondern die zum i -ten Mal geteilte Dezimalzahl

darstellt. Anhand von zwei Beispielen soll dieses Vorgehen ver deutlicht werden:

► Beispiel 2.1.5 – Umrechnung einer Dezimalzahl in eine Dualzahl

Die zur Dezimalzahl $x_{b10} = 41$ äquivalente Dualzahl ergibt sich wie folgt:

$$\begin{array}{r} i \quad x_i \quad b \quad a_i \\ \hline 0 \quad 41 \quad \div 2 = 20 \text{ Rest } 1 \\ 1 \quad 20 \quad \div 2 = 10 \text{ Rest } 0 \\ 2 \quad 10 \quad \div 2 = 5 \text{ Rest } 0 \\ 3 \quad 5 \quad \div 2 = 2 \text{ Rest } 1 \\ 4 \quad 2 \quad \div 2 = 1 \text{ Rest } 0 \\ 5 \quad 1 \quad \div 2 = 0 \text{ Rest } 1 \end{array}$$

Das resultierende binäre Zahlwort lautet $x_{b2} = 101001$. Die Probe verifiziert unsere Umrechnung:

$$\sum_{i=0}^5 a_i \cdot 2^i = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 = 41$$



► Beispiel 2.1.6 – Umrechnung einer Dezimalzahl in eine Hexadezimalzahl

Die zur Dezimalzahl $x_{b10} = 45.054$ äquivalente Hexadezimalzahl ergibt sich wie folgt:

$$\begin{array}{r} i \quad x_i \quad b \quad a_i \quad \Sigma_{b16} \\ \hline 0 \quad 45.054 \quad \div 16 = 2815 \text{ Rest } 14 \quad E \\ 1 \quad 2815 \quad \div 16 = 175 \text{ Rest } 15 \quad F \\ 2 \quad 175 \quad \div 16 = 10 \text{ Rest } 15 \quad F \\ 3 \quad 10 \quad \div 16 = 0 \text{ Rest } 10 \quad A \end{array}$$

Das resultierende hexadezimale Zahlwort lautet $x_{b16} = AFFE$. Eine Probe zeigt die Korrektheit der Umrechnung:

$$\begin{aligned} \sum_{i=0}^3 a_i \cdot 16^i &= E \cdot 16^0 + F \cdot 16^1 + F \cdot 16^2 + A \cdot 16^3 \\ &= 14 \cdot 16^0 + 15 \cdot 16^1 + 15 \cdot 16^2 + 10 \cdot 16^3 \\ &= 45.054 \end{aligned}$$



Gerade, wenn Zahlen verschiedener Basen im selben Kontext vorkommen, bietet sich mitunter eine kompaktere Schreibweise an, bei der man statt $x_{b16} = AFFE$ oder $x_{b2} = 101001$ die Basis direkt an die Zahl $AFFE_{16}$ oder 101001_2 notiert. In diesem Buch werden beide Schreibweisen Verwendung finden. Da

wir uns an Zahlen zur Basis 10 gewöhnt haben, sollen diese ohne extra annotierte Basis als Zahlen zur Basis 10 interpretiert werden, d. h., 41.204 ist gleichbedeutend mit 41.204_{10} .

2.1.3 Arithmetik der natürlichen Zahlen im polyadischen Zahlensystem

Neben dem reinen Schreiben der Zahlen hat das Rechnen mit ihnen eine enorme Bedeutung in Wirtschaft, Wissenschaft und Technik. Da in additiven Zahlensystemen schlecht gerechnet werden kann, haben sich heute weltweit die Stellenwertssysteme durchgesetzt. Auf den Ziffern und Zahlworten werden im Folgenden Rechenoperationen definiert.

Bei der Addition (lat. „addere“ = hinzufügen) werden die zwei Zahlen x und y zusammengezählt. Die Operanden der Addition heißen Summanden. Das Ergebnis der Operation ist die Summe s . Am besten kann man die Addition durch Verschieben von Zahlenstrahlen veranschaulichen: Man verschiebt den Zahlenstrahl des zweiten Summanden an das Ende des Zahlenstrahls des ersten Summanden. Das Ende des Zahlenstrahls des zweiten Summanden zeigt nun auf die Summe der beiden Summanden.

Addition

In einem Stellenwertsystem lässt sich die Summe s – genauer gesagt, lassen sich ihre Ziffern s_i an den jeweiligen Stellen i durch Addition der Ziffern der Summanden x_i und y_i bestimmen. Zählt man zwei Ziffern x_i und y_i an Stelle i zusammen, kann die Summe s_i größer als die Basis b des Zahlensystems werden. In diesem Fall muss ein Übertrag c (engl. „carry“) gebildet und zur nächsthöheren Stelle $i+1$ addiert werden. Folglich wird die Summe s_i nicht nur aus den Ziffern x_i und y_i , sondern auch aus dem Übertrag c_{i-1} der nächstniedrigeren Stelle $i-1$ gebildet. Lediglich für die niedrigstwertige Stelle $i=0$ muss kein Übertrag c_{-1} berücksichtigt werden ($c_{-1}=0$).¹ Aus diesen Überlegungen folgt die Berechnung einer Stelle des Zahlwortes, des Übertrags und der jeweiligen Summe:

Definition 2.1.8 – Berechnung des Übertrags

Der Übertrag oder Carry wird wie folgt bestimmt:

$$c_i = \begin{cases} 0 & x_i + y_i + c_{i-1} < b \\ 1 & \text{sonst} \end{cases} \quad (2.9)$$

¹ In späteren Abschnitten wird uns der Übertrag c_{-1} unter der Bezeichnung Carry-in c_{in} wieder begegnen, da dieser für die Subtraktion in Addierwerken verwendet wird.

Operand	x	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
Operand	y	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
Carry	c	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1
Summe	s	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1

■ Abb. 2.5 Addition zweier Zahlen in einem Stellenwertsystem

Definition 2.1.9 – Berechnung der Summe

Die Summe zweier Ziffern eines Zahlwertes berechnet sich wie folgt:

$$s_i = \begin{cases} x_i + y_i + c_{i-1} \bmod b & i > 0 \\ x_0 + y_0 & \text{sonst} \end{cases} \quad (2.10)$$

Definiert man $c_{-1} = 0$, vereinfacht sich die Berechnung zu

$$s_i = x_i + y_i + c_{i-1} \quad (2.11)$$

Da es natürlich auch noch auf der höchstwertigen Stelle der Summanden zu einem Übertrag kommen kann, muss für deren Summe immer eine Stelle mehr vorgesehen werden als für die Zahlwörter der Summanden (■ Abb. 2.5).

Im Dezimalsystem ist die Basis 10. Somit werden die einzelnen Ziffern einer n -stelligen Zahl ($i = 0$ bis $i = (n - 1)$) des Ergebnisses und des Übertrags mit $c_{-1} = 0$ wie folgt bestimmt:

$$c_i = \begin{cases} 0 & x_i + y_i + c_{i-1} < 10 \\ 1 & \text{sonst} \end{cases}$$

$$s_i = x_i + y_i + c_{i-1} \bmod 10$$

► Beispiel 2.1.7 – Addition im Dezimalsystem

Die Summe aus 12.454 und 2346 soll berechnet werden:

$$\begin{aligned} c_{-1} &= 0 \Rightarrow s_0 = (4 + 6 + 0) \bmod 10 = 0 \Rightarrow c_0 = 1 \\ c_0 &= 1 \Rightarrow s_1 = (5 + 4 + 1) \bmod 10 = 0 \Rightarrow c_1 = 1 \\ c_1 &= 1 \Rightarrow s_2 = (4 + 3 + 1) \bmod 10 = 8 \Rightarrow c_2 = 0 \\ c_2 &= 0 \Rightarrow s_3 = (2 + 2 + 0) \bmod 10 = 4 \Rightarrow c_3 = 0 \\ c_3 &= 0 \Rightarrow s_4 = (1 + 0 + 0) \bmod 10 = 1 \Rightarrow c_4 = 0 \end{aligned}$$

Aus der Schule sind wir eine wesentlich kompaktere Schreibweise gewohnt, bei der man die beiden Summanden untereinander schreibt. Dabei notiert man dann in einer dritten Zeile den jewei-

ligen Übertrag. Darunter – meistens unter einem Strich – wird schließlich die Summe als Ergebnis vermerkt. Für das Beispiel mit Basis $b = 10$ schreiben wir:

$$\begin{array}{r} x_{b10} & 12454 \\ y_{b10} & 02346 \\ \hline c_{b10} & 000110 \\ s_{b10} & 014800 \end{array}$$

Im Binärsystem ist die Basis 2. Somit werden die einzelnen Ziffern des Ergebnisses und des Übertrags mit $c_{-1} = 0$ mit

$$c_i = \begin{cases} 0 & x_i + y_i + c_{i-1} < 2 \\ 1 & \text{sonst} \end{cases}$$

$$s_i = x_i + y_i + c_{i-1} \bmod 2$$

dargestellt.



► Beispiel 2.1.8 – Addition im Dualsystem

Die Summe aus 101101_2 und 1101_2 berechnet sich dann wie folgt:

$$\begin{aligned} c_{-1} = 0 &\Rightarrow s_0 = (1 + 1 + 0) \bmod 2 = 0 \Rightarrow c_0 = 1 \\ c_0 = 1 &\Rightarrow s_1 = (0 + 0 + 1) \bmod 2 = 1 \Rightarrow c_1 = 0 \\ c_1 = 0 &\Rightarrow s_2 = (1 + 1 + 0) \bmod 2 = 0 \Rightarrow c_2 = 1 \\ c_2 = 1 &\Rightarrow s_3 = (1 + 1 + 1) \bmod 2 = 1 \Rightarrow c_3 = 1 \\ c_3 = 1 &\Rightarrow s_4 = (0 + 0 + 1) \bmod 2 = 1 \Rightarrow c_4 = 0 \\ c_4 = 0 &\Rightarrow s_5 = (1 + 0 + 0) \bmod 2 = 1 \Rightarrow c_5 = 0 \end{aligned}$$

Oder wieder in der uns gewohnten kompakteren Schreibweise:

$$\begin{array}{r} x_{b2} & 101101 \\ y_{b2} & 001101 \\ \hline c_{b2} & 0011010 \\ s_{b2} & 0111010 \end{array}$$

Eine Probe ist jetzt mit den bereits definierten Umwandlungen durchführbar. Zunächst werden 101101_2 und 1101_2 jeweils in eine Dezimalzahl umgewandelt:

$$\begin{aligned} 101101_2 &= \sum_{i=0}^5 a_i \cdot 2^i = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 \\ &= 1 + 4 + 8 + 32 = 45_{10} \\ 1101_2 &= \sum_{i=0}^3 a_i \cdot 2^i = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \\ &= 1 + 4 + 8 = 13_{10} \end{aligned}$$

Die beiden Dezimalzahlen werden anschließend addiert,

$$\begin{array}{r} x_{b10} \ 45 \\ y_{b10} \ 13 \\ \hline c_{b10} \ 00 \\ s_{b10} \ 58 \end{array}$$

2

und das Ergebnis dieser Addition wird wieder in das Binärsystem umgewandelt:

$$\begin{array}{r} s_{b2} : \\ s_0 = 58 \div 2 = 29 \text{ Rest } 0 \quad 0 \\ s_1 = 29 \div 2 = 14 \text{ Rest } 1 \quad 1 \\ s_2 = 14 \div 2 = \quad 7 \text{ Rest } 0 \quad 0 \\ s_3 = \quad 7 \div 2 = \quad 3 \text{ Rest } 1 \quad 1 \\ s_4 = \quad 3 \div 2 = \quad 1 \text{ Rest } 1 \quad 1 \\ s_5 = \quad 1 \div 2 = \quad 0 \text{ Rest } 1 \quad 1 \end{array}$$



Die Beispielrechnungen zeigen, dass die Addition in einem Stellenwertsystem mit einer anderen Basis als 10 zwar ungewohnt ist, aber ebenso funktioniert wie die in der Schule erlernte Methode zur schriftlichen Addition in der Basis 10.

Subtraktion

Die Subtraktion als Umkehroperation der Addition berechnet die Differenz zwischen zwei Zahlen. In der Mathematik leitet sich die Subtraktion aus der Annahme ab, dass für zwei gegebene Zahlen x und y immer eine Zahl – die Differenz d – existiert, sodass gilt:

$$y + d = x \tag{2.12}$$

Unter Verwendung des Zeichens – lässt sich die Subtraktion dann wie gewohnt notieren:

$$d = x - y \tag{2.13}$$

Dabei nennt man x den Minuenden und y den Subtrahenden. In einem Stellenwertsystem lässt sich diese, wie schon bei der Addition gezeigt, durch Subtraktion der einzelnen Stellen inkl. Übertrag realisieren:

Definition 2.1.10 – Berechnung des Übertrags

Seien x der Minuend und y der Subtrahend. Dann gilt für den Übertrag c_i an der Stelle i :

$$c_i = \begin{cases} 1 & x_i < y_i \\ 0 & \text{sonst} \end{cases} \tag{2.14}$$

Ein Übertrag von 1 an der Stelle i bedeutet, dass der Subtrahend größer als der Minuend ist und für die Berechnung der

Differenz eine Einheit der nächsthöheren Stelle $i + 1$ benötigt wird.

Definition 2.1.11 – Berechnung der Differenz

Die Ziffern d_i der Differenz zwischen Minuend x und Subtrahend y lassen sich wie folgt berechnen:

$$d_i = \begin{cases} (x_i + b - c_{i-1}) - y_i & x_i < y_i \\ (x_i - c_{i-1}) - y_i & \text{sonst} \end{cases} \quad (2.15)$$

Dabei gilt $c_{-1} = 0$.

Ist der Minuend kleiner als der Subtrahend ($x_i < y_i$), wird vor der Berechnung der Differenz die Basis b zum Minuenden hinzugefügt. Dadurch wird der Minuend immer größer gleich dem Subtrahenden, und die Differenz kann gebildet werden. Dieses Hinzufügen der Basis bedeutet jedoch, dass dem Minuenden ein Wert weniger auf der nächsthöheren Stelle zur Verfügung steht. Genau dies wird durch den vorher definierten Übertrag realisiert, der vom Minuenden an der um 1 größeren Stelle abgezogen wird ($x_i - c_{i-1}$).

► Beispiel 2.1.9 – Subtraktion im Dezimalsystem

Betrachten wir wieder die beiden Dezimalzahlen $x_{b10} = 12.454$ und $y_{b10} = 2346$. Jetzt soll aber y von x abgezogen werden:

$$\begin{aligned} c_{-1} = 0 &\Rightarrow d_0 = (4 + 10 - 0) - 6 = 8 \Rightarrow c_0 = 1 \\ c_0 = 1 &\Rightarrow d_1 = (5 - 1) - 4 = 0 \Rightarrow c_1 = 0 \\ c_1 = 0 &\Rightarrow d_2 = (4 - 0) - 3 = 1 \Rightarrow c_2 = 0 \\ c_2 = 0 &\Rightarrow d_3 = (2 - 0) - 2 = 0 \Rightarrow c_3 = 0 \\ c_3 = 0 &\Rightarrow d_4 = (1 - 0) - 0 = 1 \Rightarrow c_4 = 0 \end{aligned}$$

In der bereits eingeführten kompakteren Schreibweise wird wieder der an Stelle i auftretende Übertrag unter die Ziffern der Stelle $i+1$ notiert und entsprechend vom Minuenden abgezogen:

$$\begin{array}{r} x_{b10} \quad 12454 \\ y_{b10} \quad 02346 \\ \hline c_{b10} \quad 000010 \\ \hline d_{b10} \quad 10108 \end{array}$$

► Beispiel 2.1.10 – Subtraktion im Dualsystem

Gegeben sind wieder die beiden Zahlen $x_{b10} = 12.454$ und $y_{b10} = 2346$. Diesmal soll die Subtraktion aber im binären Stellenwertsystem erfolgen. Durch Anwendung der bekannten Umrechnungs-

regeln erhalten wir $x = 12.454_{10} = 11000010100110_2$ und $y = 2346_{10} = 10010010010_2$. Die Differenz ergibt sich dann wie folgt:

2

$$\begin{aligned}
 c_{-1} = 0 &\Rightarrow d_0 = (0 - 0) - 0 &= 0 \Rightarrow c_0 = 0 \\
 c_0 = 0 &\Rightarrow d_1 = (1 - 0) - 1 &= 0 \Rightarrow c_1 = 0 \\
 c_1 = 0 &\Rightarrow d_2 = (1 - 0) - 0 &= 1 \Rightarrow c_2 = 0 \\
 c_2 = 0 &\Rightarrow d_3 = (0 - 0) - 0 &= 0 \Rightarrow c_3 = 0 \\
 c_3 = 0 &\Rightarrow d_4 = (0 + 2 - 0) - 1 &= 1 \Rightarrow c_4 = 1 \\
 c_4 = 1 &\Rightarrow d_5 = (1 - 1) - 0 &= 0 \Rightarrow c_5 = 0 \\
 c_5 = 0 &\Rightarrow d_6 = (0 - 0) - 0 &= 0 \Rightarrow c_6 = 0 \\
 c_6 = 0 &\Rightarrow d_7 = (1 - 0) - 1 &= 0 \Rightarrow c_7 = 0 \\
 c_7 = 0 &\Rightarrow d_8 = (0 - 0) - 0 &= 0 \Rightarrow c_8 = 0 \\
 c_8 = 0 &\Rightarrow d_9 = (0 - 0) - 0 &= 0 \Rightarrow c_9 = 0 \\
 c_9 = 0 &\Rightarrow d_{10} = (0 + 2 - 0) - 1 = 1 \Rightarrow c_{10} = 1 \\
 c_{10} = 1 &\Rightarrow d_{11} = (0 + 2 - 1) - 0 = 1 \Rightarrow c_{11} = 1 \\
 c_{11} = 1 &\Rightarrow d_{12} = (1 - 1) - 0 &= 0 \Rightarrow c_{12} = 0 \\
 c_{12} = 0 &\Rightarrow d_{13} = (1 - 0) - 0 &= 1 \Rightarrow c_{13} = 0
 \end{aligned}$$

Oder in kompakterer Schreibweise:

$$\begin{array}{r}
 x_{b2} \quad 11000010100110 \\
 y_{b2} \quad 00010010010010 \\
 \hline
 c_{b2} \quad 001100000100000 \\
 \hline
 d_{b2} \quad 10110000010100
 \end{array}$$

Die Probe kann unter Verwendung der bekannten Umrechnungsregeln erfolgen, und die Differenz $d = 10110000010100_2 = 10.108_{10}$ ist das uns bereits aus dem vorherigen Beispiel bekannte korrekte Ergebnis. ◀

Soll in einem Rechner die Subtraktion mit der in der Schule eingeführten Methode bestimmt werden, müssten zwei unterschiedliche Rechenwerke implementiert werden; eines für die Addition und eines für die Subtraktion. Je nach darstellbarem Zahlenbereich sind die jeweiligen Rechenwerke aufwendig herzustellen. Um die Konstruktion der Rechenmaschine zu vereinfachen, kann die Subtraktion auf die Addition zurückgeführt werden. Dadurch lässt sich der Aufwand zum Bau des Rechenwerks nahezu halbieren. Insbesondere in der Frühzeit automatischer Rechenmaschinen erhöhte dieses Vorgehen die Zuverlässigkeit einer Rechenanlage, denn weniger Komponenten bedeuten letztendlich auch weniger Teile, die ausfallen können.

Eine Möglichkeit, die Subtraktion durch eine Addition zu ersetzen, ergibt sich durch Umkehr des Vorzeichens des Subtrahenden:

$$d = x - y = x + (-y) = s \tag{2.16}$$

Mit dieser Idee lässt sich dann das im Rechner bereits vorhandene Rechenwerk zur Bildung der Summe s (Addition) auch für die Berechnung der Differenz d (Subtraktion) nutzen. Mit

Subtraktion durch Addition

der aus der Schule bekannten Mathematik wirkt diese Umformung im ersten Moment wie ein Taschenspielertrick: Für die Summe aus 5 und -2 müssen das Vorzeichen gesondert betrachtet und aus der Addition wieder eine Subtraktion gemacht werden, um die 3 als korrektes Ergebnis zu erhalten.

Um die Subtraktion mithilfe einer Addition realisieren zu können, wird also eine spezielle Art zur Darstellung negativer Zahlen notwendig. Eine etablierte Möglichkeit bietet die sogenannte Komplementdarstellung (lat. „complementum“ = Vervollständigung[smittel]). Die Idee der Komplementdarstellung lässt sich an einem Beispiel erklären: Addiert man zur 5_{10} die Basis $b = 10$, so erhält man 15_{10} . Der Wert an der Stelle $i = 0$ ist wieder 5_{10} , und es entsteht ein Übertrag an Stelle $i = 1$, den wir ignorieren. Addiert man zur 5_{10} die Basis abzüglich 2_{10} , also $10_{10} - 2_{10} = 8_{10}$, erhält man $5_{10} + 8_{10} = 13_{10}$. Ignoriert man den Übertrag, so ist das Ergebnis 3_{10} und damit das von uns erwartete Resultat der Berechnung $5_{10} - 2_{10} = 3_{10}$. Die Addition mit 8_{10} unter Vernachlässigung des Übertrags hat also denselben Effekt wie eine Subtraktion mit 2_{10} oder eine Addition mit -2_{10} .

Diese Beobachtung lässt sich nun zum sogenannten b-Komplement verallgemeinern:

Definition 2.1.12 – b-Komplement

In einem Stellenwertsystem zur Basis b ist das b-Komplement z' einer n -stelligen Zahl z gegeben durch:

$$z + z' = b^n \quad (2.17)$$

Insbesondere ist daher z' eine vorzeichenlose Darstellung für $-z$, die für eine normale Addition geeignet ist.

► Beispiel 2.1.11 – 10-Komplement

Das b-Komplement einer Dezimalzahl – das 10-Komplement – ist deren Differenz zur nächsthöheren Zehnerpotenz. Für die 4-stellige Zahl $y = 2346$ ist diese $10^4 = 10.000$. Das Komplement y' erfüllt $2346 + y' = 10.000$ und ist daher $y' = 10.000 - 2346 = 7654$.



► Beispiel 2.1.12 – Subtraktion im 10-Komplement

Berechnung der Differenz der Dezimalzahlen $x_{b10} = 5454$ und $y_{b10} = 2346$ einmal durch Subtraktion und einmal durch Addition mit Komplementbildung. Zuerst erfolgt die Berechnung der Differenz durch Subtraktion:

$$\begin{array}{r} x_{b10} \quad 05454 \\ y_{b10} \quad 02346 \\ c_{b10} \quad 00010 \\ \hline d_{b10} \quad 03108 \end{array}$$

2

Aus dem vorangegangenen Beispiel ist bekannt, dass das 4-stellige Komplement von $y_{b10} = 2346$ gleich $y'_{b10} = 7654$ ist. Nun erfolgt die Berechnung der Differenz durch Addition des Minuenden mit dem Komplement des Subtrahenden:

$$\begin{array}{r} x_{b10} \quad 05454 \\ y'_{b10} \quad 07654 \\ c_{b10} \quad 11100 \\ \hline s_{b10} \quad 13108 \end{array}$$

Im 4-stelligen Zahlenraum wird die 5. Stelle – der Übertrag von 1 – nicht angegeben. Die Addition mit dem 10-Komplement liefert also das korrekte Ergebnis 3108. ◀

Die gezeigte Berechnung der Subtraktion als Addition mit dem Komplement verlagert allerdings das Problem: Um das Komplement zu bilden ist weiterhin eine Subtraktion erforderlich. Man betrachte noch einmal das Beispiel zur Komplementbildung der Dezimalzahl 2346_{10} :

$$\begin{array}{r} 10^4 \quad 10000 \\ y_{b10} \quad 02346 \\ c_{b10} \quad 11110 \\ \hline y'_{b10} \quad 07654 \end{array}$$

Es wird also eine vollständige Subtraktion mit Überträgen benötigt. Dieser Umstand soll aber beim maschinellen Rechnen gerade vermieden werden. Es wurde bereits festgestellt, dass die Addition einer n -steligen Zahl mit dem Vielfachen b^n der Basis b wieder die ursprüngliche Zahl unter Vernachlässigung des Übertrags auf Stelle $n+1$ ergibt. Aus dieser einen Addition lassen sich jedoch zwei machen: Man addiert zur ursprünglichen Zahl erst das Vielfache der Basis abzüglich 1, d. h. $b^n - 1$, und summiert auf das Ergebnis nochmals eine 1. Will man nun eine Subtraktion durchführen, muss das Komplement nicht als b -Komplement, sondern als sogenanntes $(b-1)$ -Komplement gebildet werden:

Definition 2.1.13 – $(b-1)$ -Komplement

In einem Stellenwertsystem zur Basis b ist das $(b-1)$ -Komplement z' einer n -steligen Zahl z gegeben durch:

$$z + z' = b^n - 1 \tag{2.18}$$

Auf den ersten Blick erfordert natürlich auch die Berechnung des (b-1)-Komplements eine Subtraktion. Allerdings hat dieses eine Eigenschaft, die für die Berechnung von großem Nutzen ist: $b^n - 1$ ist eine n -stellige Zahl, bei der jede Ziffer den größtmöglichen Wert der jeweiligen Basis hat. Bei der Subtraktion können also keine Überträge anfallen. Für das bekannte Beispiel der Dezimalzahl 2346_{10} berechnet sich das (b-1)-Komplement – das 9-Komplement – wie folgt:

$$\begin{array}{r} 10^4 - 1 \ 9999 \\ y_{b10} \ 2346 \\ c_{b10} \ 0000 \\ \hline y'_{b10} \ 7653 \end{array}$$

Beim maschinellen Rechnen lässt sich daher eine einfachere Subtraktionseinheit ohne Überträge verwenden. Betrachtet man die Berechnung einmal genauer, wird klar, dass ohne Überträge jede Ziffer einzeln verarbeitet werden kann. Mehr noch, das (b-1)-Komplement jedes Symbols ist von vornherein gegeben und kann quasi statisch hinterlegt werden.

Für das Dezimalsystem mit dem Alphabet $\Sigma := 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ gilt dann:

$$\begin{array}{ll} \sigma_0 = 0 \Rightarrow \sigma'_0 = 9 & \sigma_5 = 5 \Rightarrow \sigma'_1 = 4 \\ \sigma_1 = 1 \Rightarrow \sigma'_1 = 8 & \sigma_6 = 6 \Rightarrow \sigma'_2 = 3 \\ \sigma_2 = 2 \Rightarrow \sigma'_2 = 7 & \sigma_7 = 7 \Rightarrow \sigma'_3 = 2 \\ \sigma_3 = 3 \Rightarrow \sigma'_3 = 6 & \sigma_8 = 8 \Rightarrow \sigma'_0 = 1 \\ \sigma_4 = 4 \Rightarrow \sigma'_0 = 5 & \sigma_9 = 9 \Rightarrow \sigma'_1 = 0 \end{array}$$

Durch diesen Trick ist für die Bildung des (b-1)-Komplements keine Subtraktionseinheit, sondern lediglich eine statische Umwandlung der einzelnen Ziffern notwendig.

Unter Verwendung des (b-1)-Komplements des Subtrahenden lässt sich eine Subtraktion auf eine Addition mit dem (b-1)-Komplement und eine weitere Addition mit 1 zurückführen.

► Beispiel 2.1.13 – Subtraktion im 9-Komplement

Für das bereits bekannte Beispiel der Differenz der Dezimalzahlen $x_{b10} = 5454$ und $y_{b10} = 2346$ sollen nun die Addition und das (b-1)-Komplement verwendet werden. Das 9-Komplement des Subtrahenden 2346 ist $y'_{b10} = 7653$. Nun erfolgt die Berechnung der Differenz durch Addition des Minuenden mit dem 9-Komplement des Subtrahenden:

$$\begin{array}{r} x_{b10} \ 05454 \\ y'_{b10} \ 07653 \\ c_{b10} \ 11100 \\ \hline s_{b10} \ 13107 \end{array}$$

Nun muss zum 4-stelligen Ergebnis die noch fehlende 1 hinzugefügt werden:

$$\begin{array}{r} s_{b10} \ 3107 \\ \quad \quad \quad 1 \\ \hline s_{b10} \ 3108 \end{array}$$

Die Addition mit dem 9-Komplement und der 1 liefert also das korrekte Ergebnis 3108. ◀

2

Sowohl bei der Addition als auch bei der Subtraktion wurde der eingehende Übertrag c_{-1} an der Stelle $i = 0$ nicht betrachtet oder konnte fest auf $c_{-1} = 0$ gesetzt werden. Dieser auch Carry-in $c_{\text{in}} = c_{-1}$ genannte Übertrag kann im Fall der Subtraktion, also der Addition mit dem (b-1)-Komplement, jedoch auf $c_{\text{in}} = c_{-1} = 1$ gesetzt werden, was genau einer Addition des Ergebnisses mit einer 1 entspricht. Dieses Vorgehen spart die vorher gezeigte zweite Addition ein und führt zu einer effizienten Abbildung der Subtraktion auf eine Addition.

► Beispiel 2.1.14 – Subtraktion im 9-Komplement mit Übertrag

Die Berechnung der Differenz der Dezimalzahlen $x_{b10} = 5454$ und $y_{b10} = 2346$ unter Verwendung des 9-Komplements und des Carry-in $c_{\text{in}} = c_{-1} = 1$ vereinfacht sich wie folgt:

$$\begin{array}{r} x_{b10} \ 05454 \\ y'_{b10} \ 07653 \\ c_{b10} \ 11101 \\ \hline s_{b10} \ 13108 \end{array}$$

◀

Multiplikation

Bei der Multiplikation wird eine Zahl vervielfacht (lat. „multiplicare“ = vervielfachen). Die Multiplikation ist eine vereinfachende Schreibweise für das wiederholte Addieren einer Zahl:

$$y + y + \cdots + y + y = \sum_{i=1}^x y = x \cdot y$$

Es wird also x -mal y addiert. Dabei nennt man x und y Faktoren – ebenso gängig sind die Bezeichnungen Multiplikator für x und Multiplikand für y – und das Ergebnis der Multiplikation ist das Produkt. Die Multiplikation direkt mit einer x -maligen Addition des Multiplikanden durchzuführen wird für große Werte von x jedoch sehr schnell ineffizient. Auch für die Multiplikation lässt sich daher das Stellenwertsystem ausnutzen:

Definition 2.1.14 – Berechnung des Produkts

Bei der Multiplikation in einem Stellenwertsystem werden jede Ziffer des Multiplikators einzeln mit dem Multiplikanden und

ihrer Potenz b^i multipliziert und die Ergebnisse der einzelnen Multiplikationen am Schluss aufsummiert:

$$x \cdot y = \left(\sum_{i=0}^{n-1} x_i \cdot b^i \right) \cdot y = \sum_{i=0}^{n-1} x_i \cdot b^i \cdot y = \sum_{i=0}^{n-1} (x_i \cdot y) \cdot b^i \quad (2.19)$$

Dabei ergibt sich $x = \sum_{i=0}^{n-1} x_i \cdot b^i$ aus der Definition des Stellenwertsystems, und y als Konstante kann in die Summe hineingezogen werden.

Für die n Ziffern des Multiplikators wird pro Ziffer eine Multiplikation des Multiplikanden mit einer 1-stelligen Zahl sowie mit der Basis b^i notwendig. Die Multiplikation mit einer 1-stelligen Zahl ist insbesondere für kleine Basen wie $b = 2$ einfach zu realisieren, da als Multiplikatoren nur 0 und 1 beachtet werden müssen. Die Multiplikation mit der Basis b^i ist – wie aus dem Dezimalsystem bekannt – trivial, da der Ziffernfolge nur i Nullen angehängt werden müssen. Die anschließende Addition der n Teilergebnisse lässt sich mithilfe mehrere Additionen berechnen.

► Beispiel 2.1.15 – Multiplikation im Dezimalsystem

Gegeben sind die Dezimalzahlen $x = 2346$ und $y = 12.454$. Das Produkt $x \cdot y$ ergibt sich zu:

$$\begin{aligned} 2346 \cdot 12.454 &= (2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 6) \cdot 12.454 \\ &= 2 \cdot 1000 \cdot 12.454 + 3 \cdot 100 \cdot 12.454 + 4 \\ &\quad \cdot 10 \cdot 12.454 + 6 \cdot 12.454 \\ &= 1000 \cdot (2 \cdot 12.454) + 100 \cdot (3 \cdot 12.454) \\ &\quad + 10 \cdot (4 \cdot 12.454) + 6 \cdot 12.454 \\ &= 1000 \cdot 24.908 + 100 \cdot 37.362 \\ &\quad + 10 \cdot 49.816 + 74.724 \\ &= 29.217.084 \end{aligned}$$

Der Vollständigkeit halber sind auch die Umformungen entsprechend der Definition gezeigt. Tatsächlich berechnet werden muss lediglich die letzte Zeile. Anschließend werden die Teilergebnisse entsprechend ihrer Potenzen erweitert und dann addiert:

$$\begin{array}{r} t_{b10} \ 24908000 \\ u_{b10} \ 03736200 \\ v_{b10} \ 00498160 \\ w_{b10} \ 00074724 \\ c_{b10} \ 02221000 \\ \hline s_{b10} \ 29217084 \end{array}$$



► Beispiel 2.1.16 – Multiplikation im Binärsystem

Gegeben sind die Dualzahlen $x = 1011_2$ und $y = 1001_2$. Das Produkt $x \cdot y$ wird wie folgt berechnet:

$$\begin{aligned} 1011_2 \cdot 1001_2 &= (1_2 \cdot 1000_2 + 0_2 \cdot 100_2 + 1_2 \cdot 10_2 + 1_2) \cdot 1001_2 \\ &= 1_2 \cdot 1000_2 \cdot 1001_2 + 0_2 \cdot 100_2 \cdot 1001_2 \\ &\quad + 1_2 \cdot 10_2 \cdot 1001_2 + 1_2 \cdot 1001_2 \\ &= 1000_2 \cdot (1_2 \cdot 1001_2) + 100_2 \cdot (0_2 \cdot 1001_2) \\ &\quad + 10_2 \cdot (1_2 \cdot 1001_2) + 1_2 \cdot 1001_2 \\ &= 1000_2 \cdot 1001_2 + 100_2 \cdot 0_2 + 10_2 \cdot 1001_2 + 1001_2 \end{aligned}$$

Nun werden die Teilergebnisse entsprechend ihrer Potenzen erweitert und dann addiert:

$$\begin{array}{r} t_{b2} \ 1001000 \\ u_{b2} \ 0000000 \\ v_{b2} \ 0010010 \\ w_{b2} \ 0001001 \\ \hline c_{b2} \ 0110000 \\ s_{b10} \ 1100011 \end{array}$$

Für die Probe können die Dualzahlen in die entsprechenden Dezimalzahlen umgewandelt werden, d. h., $x = 1011_2 = 11_{10}$ und $y = 1001_2 = 9_{10}$. Die Umwandlung des Produkts $1100011_2 = 99_{10}$ bestätigt dann das erwartete Ergebnis. ◀

Division

Die Division ist die Umkehroperation der Multiplikation. Division (lat. „divisio“ = Teilung) ist die Teilung einer Zahl, die als Vielfaches einer anderen Zahl aufgefasst werden kann. Bei der Division wird die Zahl x , der Dividend (lat. für das zu Teilende), durch den Divisor (lat. für der Teiler) y unterteilt. Das Ergebnis der Teilung ist der Quotient. Anders ausgedrückt, wird die Frage gestellt, wie oft y in x passt. Es ergibt sich für den Quotienten q , der die Gleichung

$$q \cdot y = x \tag{2.20}$$

erfüllen soll:

$$q = x \cdot y^{-1} = \frac{x}{y} = x \div y \tag{2.21}$$

Zu beachten ist, dass alle Quotienten mit dem Dividenden 0 undefiniert sind, man also durch null nicht teilen kann. Soll der Quotient wieder ein Element der Menge der natürlichen Zahlen sein, dann bleibt bei bestimmten Kombinationen aus Dividend und Divisor ein sogenannter Rest r übrig. Bei der

ganzzahligen Division von $x \div y$ erfüllen Quotient q und Rest r folgende Bedingung:

$$x = q \cdot y + r, \quad 0 \leq r \leq y \quad (2.22)$$

Definition 2.1.15 – Berechnung der Division

Für die Division existiert ein einfaches schriftliches Vorgehen: In der Ziffernfolge des Dividenden sucht man von den höherwertigen Ziffern des Divisors ausgehend die erste Ziffernkombination als Zahl, die groß genug ist, um durch den Divisor geteilt werden zu können. Für diese Zahl bestimmt man beispielsweise durch Probieren den ganzzahligen Quotienten. Dieser Quotient ist gleichzeitig die höchswertige Ziffer des Ergebnisses. Mit diesem Quotienten wird der Divisor nun multipliziert, und das Resultat wird unter die bei diesem Schritt verwendete Ziffernfolge geschrieben. Die Differenz der beiden Zahlen ist dann gleich dem Rest der ganzzahligen Division. An diesen Rest holt man die nächste Ziffer des Dividenden nach unten und wiederholt den beschriebenen Vorgang, wodurch das Ergebnis Ziffer für Ziffer bestimmt wird. Das Vorgehen wird etliche Male fortgesetzt, bis die verbleibende Zahl nicht mehr durch den Divisor geteilt werden kann. Ist diese Zahl größer null, so handelt es sich um den Rest der Division insgesamt.

► Beispiel 2.1.17 – Division im Dezimalsystem

Man betrachte die beiden Zahlen $x = 414$ und $y = 23$. Die führende 4 des Dividenden kann nicht durch 23 geteilt, 41 kann dagegen einmal durch 23 geteilt werden. Die erste Ziffer des Ergebnisses ist also 1. $1 \cdot 23$ ergibt 23, und die Differenz zwischen 41 und 23 ist der Rest 18. Achtzehn lässt sich nicht durch 23 teilen, die 4 wird neben die 18 geschrieben und die entstehende Ziffernfolge 184 durch 23 geteilt. Das ergibt 8, und 8 multipliziert mit 23 ergibt wieder 18, und es bleibt ein Rest von 0. Das Ergebnis ist daher wie erwartet 18.

$$414 : 23 = 18$$

$$\begin{array}{r} 23 \\ \underline{184} \\ 184 \\ \hline 0 \end{array}$$



2.1.4 Codierung von natürlichen Zahlen im Rechner

Wie bereits diskutiert, sind in einem Stellenwertsystem beliebig große Zahlen denkbar. In der Praxis und bei der Herstellung einer Rechenmaschine ist die Anzahl der Stellen oder Ziffern begrenzt. Im Folgenden sollen die Zahlen technisch in einem Computer dargestellt werden. Mit diesen maschinellen Realisierungen soll die Maschine dann natürlich auch rechnen. Im ersten Kapitel wurde gezeigt, dass das automatische Rechnen eine wesentliche Motivation bei der Entwicklung des heutigen Computers als universelle Symbol verarbeitende Multimedia-maschine war. Am Ende sind digitale Audio- und Videoströme intern im Rechner nur eine Folge von Zahlen. Die Codierung dieser hat also eine enorme Bedeutung beim Bau von Computern.

2.1.4.1 Ziffernreihenfolge und Wertebereich

Neben der begrenzten Anzahl der Stellen ist es in einem Rechner notwendig, die Ziffernreihenfolge und damit die Reihenfolge der Wertigkeiten einzelner Ziffern festzulegen. Im Alltag sind wir es gewohnt, die niedrigstwertige Stelle auf der rechten Seite des Zahlwortes zu schreiben. Dies ist jedoch nicht zwingend. Man kann sich beispielsweise vorstellen das niedrigstwertige Symbol auf der linken Seite des Zahlwortes anzunehmen. Und in der Tat gibt es Computer, die sowohl mit der einen als auch der anderen Interpretation arbeiten.

Definition 2.1.16 – Ziffernreihenfolge im Zahlwort

Eine Zahl in einem Stellenwertsystem besteht aus den Ziffern $a_i \in \mathbb{N}$ und der Basis $b^i \in \mathbb{N}$. Eine n -stellige Zahl, bei der die niedrigstwertige Ziffer rechts steht, wird wie folgt codiert:

$$x = a_{n-1}a_{n-2}\dots a_1a_0 \quad (2.23)$$

Eine n -stellige Zahl, bei der die niedrigstwertige Ziffer links steht, wird wie folgt codiert:

$$x = a_0a_1\dots a_{n-2}a_{n-1} \quad (2.24)$$

Durch die Begrenzung der Stellen bei der Codierung von (natürlichen) Zahlen ergibt sich ein eingeschränkter Wertebereich. Beim Rechnen mit einer begrenzten Anzahl an Wertigkeiten kann es passieren, dass das Ergebnis außerhalb des zur Verfü-

gung stehenden Wertebereichs liegt – es kommt zu einem Überlauf o (engl. „overflow“).

► **Beispiel 2.1.18**

Die Anzahl der codierten Stellen einer Dezimalzahl sei auf $n = 1$ begrenzt. Die Summe aus 8 und 4 berechnet sich dann zu:

$$\begin{array}{r} x_{b10} \quad 8 \\ y_{b10} \quad 4 \\ \hline c_{b10} \quad 10 \\ s_{b10} \quad 2 \end{array}$$



Durch Begrenzung auf eine Stelle wird der Wertebereich von $[0, 9]$ verlassen, und das korrekte Ergebnis 12 würde falsch als 2 dargestellt.

Bei einer Addition von natürlichen Zahlen zeigt ein Übertrag an der höchstwertigen Stelle also einen Überlauf des Wertebereichs an. Bei der Subtraktion durch Addition mit dem Komplement ist dieser Überlauf hin zur korrekten Differenz jedoch beabsichtigt:

► **Beispiel 2.1.19**

Die Anzahl der codierten Stellen einer Dezimalzahl sei auf $n = 1$ begrenzt. Die Differenz aus 8 und 4 berechnet sich bei Verwendung des 10-Komplements $10 - 4 = 6$ von 4 zu:

$$\begin{array}{r} x_{b10} \quad 8 \\ y_{b10} \quad 6 \\ \hline c_{b10} \quad 10 \\ s_{b10} \quad 4 \end{array}$$

Das korrekte Ergebnis $8 - 4 = 4$ wird erhalten, und der Übertrag an der höchstwertigen Stelle wird korrekt ignoriert. Man betrachte nun die Differenz aus 4 und 8 unter Verwendung des 10-Komplements $10 - 8 = 2$ von 8:

$$\begin{array}{r} x_{b10} \quad 4 \\ y_{b10} \quad 2 \\ \hline c_{b10} \quad 00 \\ s_{b10} \quad 6 \end{array}$$

Natürlich wird bei der Berechnung $4 - 8 = -4$ der Wertebereich $[0, 9]$ der natürlichen Zahlen verlassen, und das berechnete Ergebnis 6 ist wie erwartet falsch. Der fehlende Übertrag an der höchstwertigen Stelle kann im Falle der Subtraktion unter Verwendung des Komplements verwendet werden, um ein Verlassen des Wertebereichs zu erkennen. ◀

Es zeigt sich, dass beim Rechnen mit begrenzten Stellen je nach Operation und je nach verwendeter Codierung ein Verlassen des Wertebereichs einer gesonderten Behandlung bedarf.

2.1.4.2 Rechnen im Dezimalsystem

Die ersten Rechenmaschinen arbeiteten im Dezimalsystem. Dieses Zahlensystem war den Wissenschaftlern zur Zeit der Aufklärung geläufig und bei der mechanischen Realisierung einer Ziffer waren die jeweils notwendigen 10 Symbole durch 10 Zustände noch praktikabel realisierbar. Auch einige der ersten elektronischen Rechenmaschinen arbeiteten im Dezimalsystem.

Um auf den eingeführten Zahlworten rechnen zu können, muss man sich darauf einigen, worauf die Rechnung auszuführen ist. Um das festzulegen, benötigt man die Operanden, also die zwei Zahlen, aus denen etwas Neues berechnet werden soll, und die Operation selbst. Als Operationen sollen die vier Grundrechenarten der Arithmetik diskutiert werden, und die Operanden sollen zunächst einmal natürliche Zahlen in Dezimalschreibweise sein. Im Folgenden gilt die Konvention, dass $x, y \in \mathbb{N}$ die Operanden und die Zeichen $\{+, -, \cdot, /\}$ die Operationen sind. x und y liegen als Zahlworte $x = x_{n-1} \dots x_1 x_0$ und $y = y_{n-1} \dots y_1 y_0$ in einem Stellenwertsystem zur Basis $b = 10$ vor.

Die vorher eingeführten Definitionen für die Addition, Subtraktion, Multiplikation und Division pro Ziffer eignen sich sehr gut für das schriftliche Rechnen durch einen Menschen. Möchte man jedoch einer Maschine beispielsweise das Addieren beibringen, ist es nicht von Vorteil, eine Stelle der Summe durch Addition der zwei Ziffern der Operanden und des Übertrags mit einer anschließenden ganzzahligen Division oder der Modulo-Operation zur Basis zu berechnen. Es werden also technisch sinnvoll umsetzbare Varianten der Rechenoperationen benötigt.

Am besten kann man die Addition der natürlichen Zahlen im Dezimalsystem durch Verschieben des Zahlenstrahls eines Summanden veranschaulichen. Man verschiebt zuerst den Zahlenstrahl des zweiten Summanden an die Position des Wertes des ersten Summanden auf dem ersten Zahlenstrahl. Der Wert des zweiten Summanden auf dem zweiten Zahlenstrahl steht nun auf dem Ergebnis der Addition auf dem ersten Zahlenstrahl. Damit ist bereits eine erste Addiermaschine gebaut, bei der die Berechnung durch eine Verschiebung und anschließendes Ablesen technisch umgesetzt wird.

Der Rechenschieber funktioniert nach einem ähnlichen Prinzip, allerdings sind die beiden Skalen auf diesem logarithmisch. Dadurch lässt sich die Multiplikation aus einer Addition logarithmierter Zahlen ableiten und kann dann technisch ebenfalls auf ein Verschieben und Ablesen des Rechenstabs zurückgeführt werden.

Die Subtraktion kann mit einem Rechenstab ebenfalls durchgeführt werden. Der Wert des Subtrahenden auf dem

Addition mit dem Rechenstab

Subtraktion mit dem Rechenstab

zweiten wird an die Markierung des Minuenden auf dem ersten Zahlenstrahl geschoben. Am Ursprung des zweiten kann man das Ergebnis auf dem ersten Zahlenstrahl ablesen. Es ist ebenfalls möglich, die Subtraktion mit dem (b-1)-Komplement auf die Addition abzubilden (Abb. 2.6).

Die Addition lässt sich gemäß ihrer Definition auf das Zählen zurückführen. Dabei erhöht man so lange den Stand eines Zählwerks, bis dieses den ersten Summanden anzeigt. Dann zählt man mit dem Zählwerk erneut und zwar um die Zählschritte des Wertes des zweiten Operanden. In der Anzeige steht nun folglich die Summe der beiden Zahlen. Ein Zählwerk für eine Ziffer im Dezimalsystem muss 10 unterschiedliche Schritte anzeigen, um die Symbole des Dezimalsystems darstellen zu können. Oder besser, das Zählwerk benötigt 10 Zustände, um eine Ziffer zu repräsentieren.

Ein Zählwerk kann aus einem Zahnrad bestehen, das eine Drehung auf eine 10-stellige Anzeige abbildet. Dabei sollte eine volle Umdrehung alle 10 Werte einer Ziffer darstellen. Um eine Rechenmaschine zu bauen, muss dieses Zahnrad angetrieben werden. Der Antrieb wird mit einer Walze mit 9 Zähnen implementiert. Wenn nun das Zahnrad des Zählwerks an diese Walze gekoppelt ist, würde 1 Umdrehung dieser zu 10 Umdrehungen des Zahnrades und somit des angeschlossenen Zählers führen. Führt man daraufhin die Zähne der Walze unterschiedlich lang aus, kann man durch vertikales Verschieben des Zahnrades alle 10 Ziffern des Dezimalsystems codieren. Eine derartig gestaltete Walze heißt Staffelwalze.

Abb. 2.7 zeigt eine einfache Rechenmaschine, die auf dem Prinzip der von Leibniz erfundenen Staffelwalze beruht. Zur Veranschaulichung sind die Walze und die Zählwerke im linken Teil der Abbildung ausgerollt. Man muss sich die wirkliche

Staffelwalzmaschine

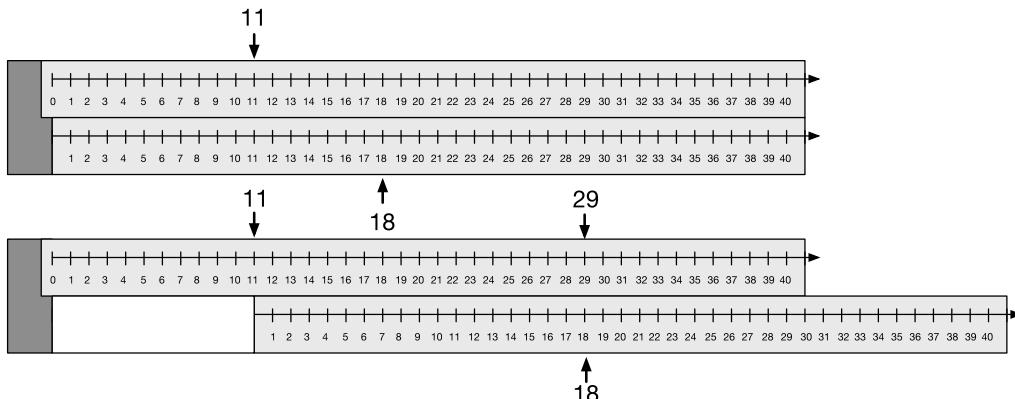


Abb. 2.6 Addition mit dem Rechenstab

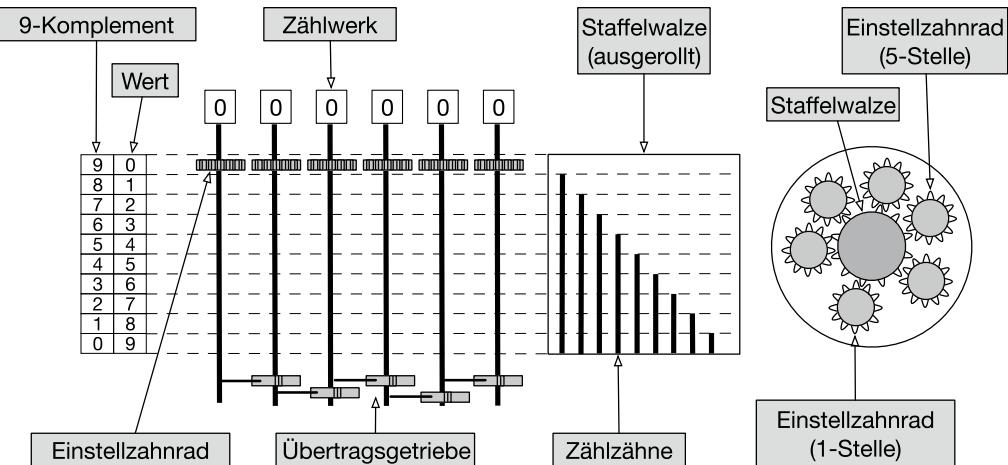


Abb. 2.7 Aufbau einer Staffelwalzmaschine

Anordnung wie im rechten Abschnitt der Zeichnung vorstellen. Deutlich erkennt man auf der Staffelwalze die 9 verschiedenen langen Zähne, um die Ziffern 1, 2, …, 8, 9 zu repräsentieren. Für die 0 existiert naturgemäß kein Zahn. Jedes Zählwerk ist an eine sich drehende Achse angeschlossen. Auf dieser kann nun das Zählnetz verschoben werden, um unterschiedliche Werte einzustellen zu können. Zum Rechnen müssen die Zahnräder die Achse mit drehen. Jede dieser Wellen besitzt noch eine Mechanik zur Berechnung des Übertrags. Dafür ist nur ein Zahn oder ein kleiner Hebel notwendig, der die benachbarte Achse und damit die jeweils nächste Ziffer nach einer Umdrehung um 1 weiter zählt. Mithilfe einer Anzeige kann man nun ein Symbol durch Verschieben eines Zahnrades einstellen. Bleibt noch die linke Seite der Einstellanzeige zu erklären. Diese wird für die Subtraktion benötigt und später beschrieben.

► Beispiel 2.1.20 – Addition mit der Staffelwalze

Eine Addition lässt sich mit der Staffelwalze einfach durchführen: Zunächst stellt man mithilfe der verschiebbaren Zahnräder den ersten Summanden ein. In dem Beispiel in der Abb. 2.8 sind dies die Ziffern 789309. Alle Zählwerke stehen initial bei 0. Dann dreht man die Staffelwalze 1-mal. Das Zählwerk zeigt nun die eingestellte Zahl 789.309 an. Dann kann der zweite Summand 4246 mithilfe der verschiebbaren Zahnräder eingestellt werden, und die Staffelwalze wird erneut 1-mal gedreht. Im Zählwerk steht dann das Ergebnis der Additionsoperation (793.555). ◀

Die Subtraktion wird in Rechenmaschinen auf die Addition von Komplementzahlen zurückgeführt. Das hat den Vorteil, dass nur eine Recheneinheit implementiert werden muss und diese dann durch die unterschiedlichen Codierungen der Zah-

2.1 · Natürliche Zahlen

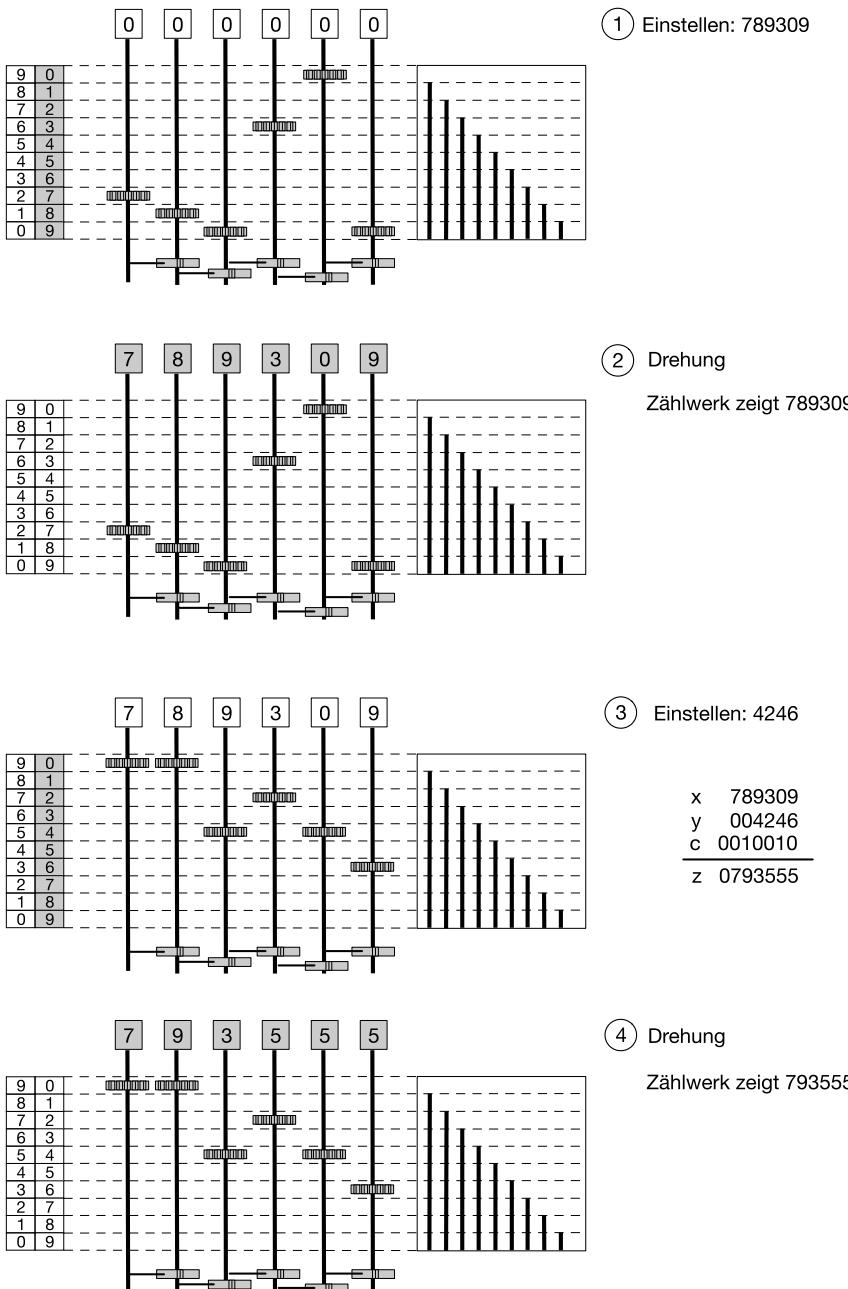


Abb. 2.8 Addition mit der Staffelwalze

len effizient genutzt wird. Durch eine geschickte Anordnung der Eingabeschalter oder eine automatische Codierung kann die Darstellung der Komplementzahlen automatisch in der Rechenmaschine erfolgen.

► Beispiel 2.1.21 – Addition mit der Staffelwalze

Die Abb. 2.9 zeigt, wie man mit einer Staffelwalzmaschine eine Subtraktion durchführt. Möchte man von 789.309 die Zahl 4246 abziehen, stellt man zunächst mit den Schiebereglern den Minuenden ein: 789.309. Anschließend dreht man die Staffelwalze 1-mal. Im Zählwerk steht nun 789.309. Im Gegensatz zu der Addition, bei der die Zahl an der rechten Hauptskala eingegeben wird, stellt man nun den Minuenden 4246 an der linken Komplementärskala ein. Dies bewirkt also, dass als zweiter Summand die Komplementärzahl 995.753 eingestellt ist. Nach einer Drehung zeigt das Zählwerk nun 785.062. Das entspricht $789.309 + 995.753 = 1.785.062$, wobei der Übertrag an der höchstwertigen Stelle (1.) 785.062 verloren geht. Bei der Subtraktion mittels 9-Komplement wird dieser Übertrag an der höchstwertigen Stelle jedoch sowieso gestrichen. Für das endgültige Ergebnis muss bei Nutzung des 9-Komplements noch 1 zu 785.062 addiert werden, sodass man das erwartete Ergebnis 785.063 erhält. ◀

Beispiel einer Staffelwalze:

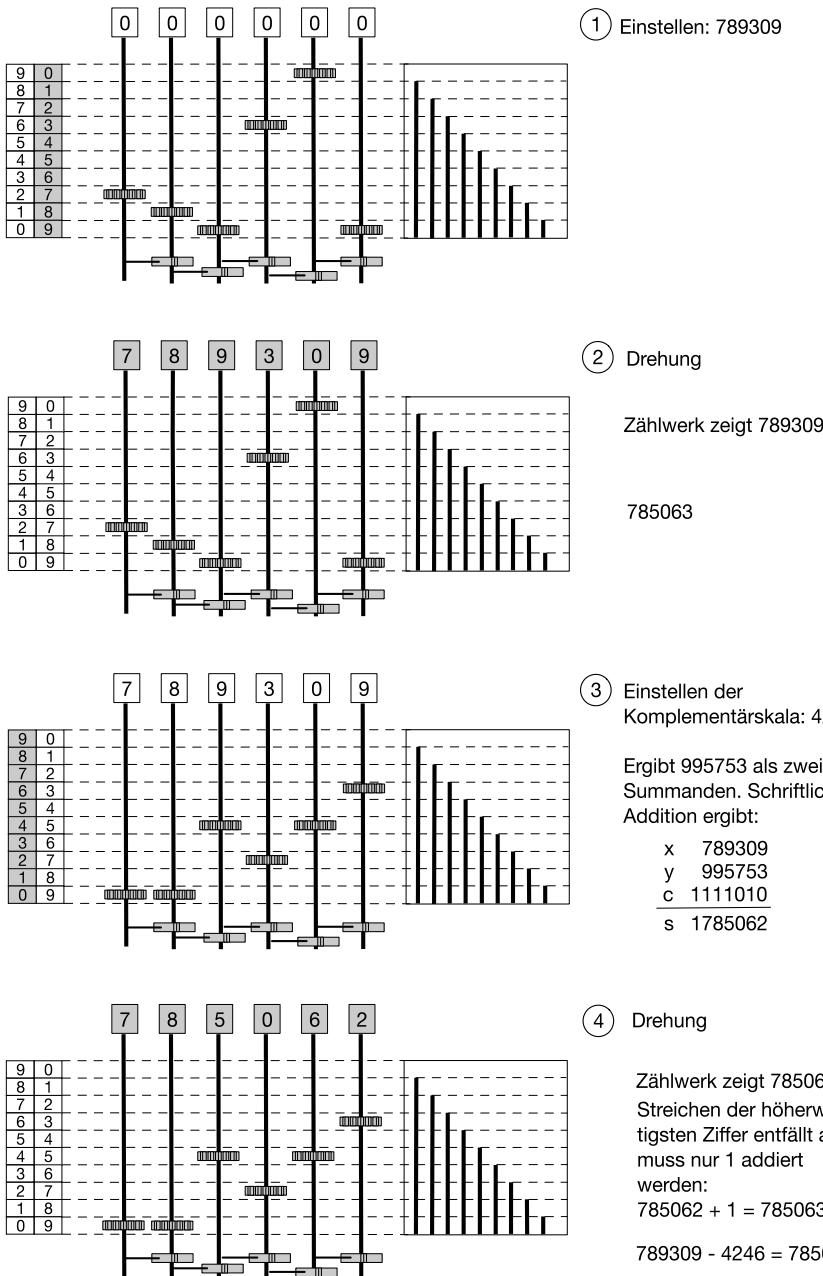
Die Curta

Eine nahezu in Vollendung realisierte technische Implementierung des gezeigten Prinzips der Staffelwalze ist die Curta. Von 1947 bis 1972 erfreute sich die Curta einer großen Beliebtheit. So wurden bis zum Aufkommen der ersten Taschenrechner etwa 140.000 dieser kleinen Doppelstaffelwalzmaschinen produziert.

► Beispiel 2.1.22 – Subtraktion mit der Curta II

Zunächst ist die Curta in ihre Grundeinstellung zurückzusetzen. In allen Stellwerken muss also eine 0 stehen, um den Zustand der Rechenbereitschaft zu erreichen. Dieser Grundzustand wird bei der Curta hergestellt, indem der Wagen angehoben, der Hebel einmal gedreht und anschließend auf diesen wieder abgesenkt werden. Soll etwa 42 von 126 subtrahiert werden, stellt man zuerst den Minuenden ein, also 126. Die niedrigstwertige Ziffer steht dabei an der Stelle ganz rechts, die höchstwertige entsprechend links. Wenn man nun den Hebel einmal komplett rundherum dreht, so kann man auf der Oberseite die eingestellte 126 ablesen. Im nächsten Schritt wird die Zahl eingegeben, die von der vorherig eingegebenen abgezogen werden soll. In dem Fallbeispiel soll die Zahl 42 subtrahiert werden. Wenn nun am Hebel gedreht würde, stünde als Ergebnis 168 auf der Oberseite, da mit der Staffelwalze die beiden Werte addiert würden. Bei der Subtraktion jedoch wird der Hebel der Curta ein Stück aus dem Gehäuse herausgezogen und erst dann wird gedreht. Nach dem Absenken des Hebels wird anschließend das Resultat 84 abgelesen. ◀

2.1 · Natürliche Zahlen

**Abb. 2.9** Subtraktion mit der Staffelwalze

► Beispiel 2.1.23 – Multiplikation mit der Curta II

Ist beispielsweise 76 mit 4 zu multiplizieren, ist die Maschine erst einmal wieder in den Grundzustand zurückzuversetzen. Zunächst wird die 76 eingestellt. Die höchstwertige Ziffer der Zahl steht dabei an der Stelle ganz links, die niedrigstwertige wieder rechts. Wenn nun der Hebel ein ganzes Mal gedreht wird, kann auf der Oberseite die 76 abgelesen werden. Im nächsten Schritt wird die Zahl eingegeben, mit der die vorher eingegebene multipliziert werden soll, also die Zahl 4. Das geschieht, indem der Hebel 4-mal gedreht wird. Das Resultat 304 kann anschließend abgelesen werden. Eine andere Möglichkeit ist, zunächst eine Drehung des Hebels durchzuführen. Dies entspricht dem Wert der niedrigstwertigen Ziffer. Dann hebt man den Rundwagen an, stellt ihn um eine Position weiter und dreht den Hebel 7-mal. Dies entspricht dem Wert der höchstwertigen Stelle. Das Ergebnis ist diesmal auch wieder 304. Mit diesem Trick lassen sich auch Multiplikationsrechnungen mit sehr hohen Zahlen schnell bewerkstelligen. ◀

► Beispiel 2.1.24 – Division mit der Curta II

Auch die Division lässt mit der Curta einfach durchführen: Man stellt zuerst den Divisor ein und dreht dann so lange, bis der Dividend angezeigt wird. Das Ergebnis kann im Anschluss auf dem Umdrehungszähler abgelesen werden. ◀

Die Curta II kann also für Rechnungen mit allen bekannten Grundrechenarten herangezogen werden. Dadurch beherrschte dieses faszinierende Räderwerk schätzungsweise sechs Jahrzehnte lang sämtliche Rechenbüros und wurde erst nach Einführung der ersten elektronischen Taschenrechner langsam vom Markt verdrängt.

2.1.4.3 Rechnen im Dualsystem

Bei der Entwicklung elektronischer Rechner stellte man schnell fest, dass eine Zustandscodierung der Ziffern im Dezimalsystem aufwendig zu bauen ist. Elektrisch lassen sich aber mit Schaltern effizient die zwei Zustände an und aus abbilden. Technisch ist es also geschickt, vom Dezimal- in das Dualsystem zu wechseln. Zwei unterschiedliche Zustände stellen die kleinste mögliche Zahl unterscheidbarer Zustände dar. Technisch ist ein Bauelement, dass nur zwei Zustände unterscheidet, einfach zu bauen, entweder als Relais oder elektronisch mit einer Röhre oder mit Transistoren. Später wird gezeigt, wie sich mithilfe der Planartechnik effektiv integrierte Schaltungen mit Tausenden mittlerweile Milliarden Transistoren auf kleinstem Raum herstellen lassen.

In Rechenanlagen werden zunächst einmal zwei unterschiedliche Darstellungen von Binärcodes betrachtet. Zum einen können alle möglichen Symbole des Dezimalsystems mithilfe von 4 Bit codiert werden. Diese Darstellung war in der

Anfangszeit der kaufmännischen Großrechner üblich, da sie einen natürlichen Übergang von den ersten dezimal codierten Rechnern zu den Binärechnern bedeutet. Eine dualcodierte Ziffer wird BCD-Code genannt („binary coded decimal“). Kaufmännische Anwendungen ließen sich so leicht implementieren und bereits etablierte Peripheriegeräte wie Fernschreiber oder andere elektromechanische Maschinen konnten direkt genutzt werden. In der heutigen Praxis hat sich jedoch die Möglichkeit, Zahlen binär zu codieren, durchgesetzt: die Dualzahl in einem Positionssystem zur Basis 2.

Um eine Dezimalziffer im Rechner darzustellen, werden also immer 4 Bit oder ein „Nibble“ (Halbbyte) benötigt. In 1 Byte können somit 2 Ziffern gespeichert werden. Die nicht genutzten Codes oder „Don't-cares“ des Nibbles nennt man Pseudotetrade (Tetrad = Vierheit). Der BCD-Code wird auch als 8–4–2–1-Code bezeichnet. Dabei steht die jeweilige Ziffer in der Bezeichnung für die Zahl, die durch Setzen des entsprechenden Bits im BCD erzeugt wird. Das Vorzeichen wird in einem separaten Bit außerhalb gespeichert. Alternativ kann dieses durch eine der nicht genutzten Pseudotetraden codiert werden. Der BCD-Code erfordert einen hohen schaltungstechnischen Aufwand. Neben dem enormen Platzbedarf sind BCD-Rechner langsam. Dies führte auf der Grundlage effizient implementierter Gleitkommazahlen bei naturwissenschaftlichen Anwendungen zur Entwicklung der Number Cruncher, da die für kaufmännische Programme entworfenen Mainframes für Aufgaben aus der Wissenschaft unzureichend waren. Mit dem Aufkommen integrierter numerischer Coprozessoren Mitte der 1980er-Jahre und der Integration dieser Schaltungen in Mikroprozessoren seit den 1990er-Jahren spielt die BCD-Codierung kaum noch eine Rolle in der Computertechnik. Allerdings wird das Zeitsignal DCF77, das über einen Langwellensender bei Frankfurt am Main ausgesendet wird, mithilfe von BCD codiert. Auch viele Displays verwenden den BCD-Code zur Darstellung der verschiedenen Ziffern. In dem Fall rechnet der Rechner intern mit Dual- oder Binärzahlen, und eine Decodierlogik erzeugt dann den von der Anzeige darzustellenden Code. In BCD werden die Ziffern 0 bis 9 mittels Dualziffern codiert (☞ Tab. 2.2):

BCD-Code

BCD-Code

Tab. 2.2 BCD-Codierung

0₍₁₀₎	0000_(BCD)	0₍₁₆₎	
1₍₁₀₎	0001_(BCD)	1₍₁₆₎	
2₍₁₀₎	0010_(BCD)	2₍₁₆₎	
3₍₁₀₎	0011_(BCD)	3₍₁₆₎	
4₍₁₀₎	0100_(BCD)	4₍₁₆₎	
5₍₁₀₎	0101_(BCD)	5₍₁₆₎	
6₍₁₀₎	0110_(BCD)	6₍₁₆₎	
7₍₁₀₎	0111_(BCD)	7₍₁₆₎	
8₍₁₀₎	1000_(BCD)	8₍₁₆₎	
9₍₁₀₎	1001_(BCD)	9₍₁₆₎	
10₍₁₀₎	1010_(BCD)	A₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)
11₍₁₀₎	1011_(BCD)	B₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)
12₍₁₀₎	1100_(BCD)	C₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)
13₍₁₀₎	1101_(BCD)	D₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)
14₍₁₀₎	1110_(BCD)	E₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)
15₍₁₀₎	1111_(BCD)	F₍₁₆₎	Pseudotetradie (nicht genutzt, Don't-care)

Bit

In heutigen Rechenanlagen werden nicht nur Zahlen, sondern jegliche Art von Daten oder Befehlen binär codiert. Eine besondere Klasse der Binärcodes sind die Binär- oder Dualzahlen. Im Rechner werden die Stellen einer Dualzahl durch ein Bit („binary digit“) codiert. Die Darstellung von Binärzahlen im Rechner ist dabei für die natürlichen Zahlen äquivalent zur Codierung im Stellenwertsystem der Dualzahlen.

2.2 Ganze Zahlen

Arithmetische Berechnungen beschränken sich nicht nur auf die natürlichen Zahlen. Ist beispielsweise der Subtrahend größer als der Minuend, ist das Ergebnis einer Subtraktion negativ. In der praktischen Anwendung ist es also zwingend, Rechner zu bauen, die nicht nur mit natürlichen, sondern mit ganzen Zahlen rechnen.

2.2.1 Darstellung

Die einfachste und auch naive Variante, ganze Zahlen im Rechner zu codieren, ist, ein Bit für das Vorzeichen zu opfern. Ist beispielsweise die linke oder rechte Ziffer eines binären Zahlwortes auf 1 gesetzt, handelt es sich um eine negative Zahl, ist es 0, um eine positive. Dieses Vorgehen stellt sicher, dass alte Programme, die nur natürliche Zahlen verwenden, kompatibel sind zu neuen Algorithmen, die auch negative Zahlen zulassen.

Definition 2.2.1 – Zahlwort für ganze Zahlen

Ein Stellenwertsystem besteht aus den Ziffern $a_i \in \mathbb{N}$, der Basis $b^i \in \mathbb{N}$ und einem Vorzeichen $\{+, -\}$. Neben den einzelnen Ziffern wird für die ganzen Zahlen also noch das Vorzeichen angegeben. Eine n -stellige Zahl, bei der die niedrigstwertige Ziffer rechts steht, wird wie folgt codiert:

$$(\pm)a_{n-1}a_{n-2}\dots a_0 \quad (2.25)$$

Entsprechend gilt:

$$x = (\pm) \sum_{i=0}^{n-1} a_i \cdot b^i \quad (2.26)$$

Dabei ist es Konvention, das Vorzeichen (+) bei positiven Zahlen wegzulassen.

2.2.2 Arithmetik der ganzen Zahlen

Zur Diskussion der Arithmetik der ganzen Zahlen unterscheidet man zwischen der eigentlichen Zahl ($-z$) und ihrem Betrag ($|-z| = z$).

Addition

Die Addition ganzer Zahlen muss das Vorzeichen beider Operanden berücksichtigen. Für die Berechnung der Summe $s = (\pm x) + (\pm y)$ sind die beiden Fälle gleicher und verschiedener Vorzeichen zu unterscheiden:

2

1. Fall: gleiche Vorzeichen

Sind die Vorzeichen der Summanden gleich, d. h. $x + y$ oder $(-x) + (-y)$, werden die Beträge der beiden Zahlen addiert, und das Ergebnis übernimmt das Vorzeichen der Operanden.

2. Fall: verschiedene Vorzeichen

Bei unterschiedlichen Vorzeichen, d. h. $x + (-y)$ oder $(-x) + y$, wird für das Ergebnis zuerst die Differenz der Beträge der beiden Summanden gebildet. Hierfür wird der kleinere Betrag vom größeren Betrag durch Subtraktion abgezogen. Das so erhaltene Ergebnis übernimmt das Vorzeichen des Summanden mit dem größeren Betrag.

► Beispiel 2.2.1

Man betrachte die Addition der ganzen Zahlen -10 und 7 . Zuerst berechnet man die Differenz der Beträge $|-10| = 10$ und $|7| = 7$ durch Subtraktion $10 - 7 = 3$. Der Summand -10 hat den größeren Betrag, sodass das Ergebnis dessen Vorzeichen übernimmt und wie erwartet -3 ist. ◀

Subtraktion

Für die Subtraktion ganzer Zahlen, also $d = (\pm x) - (\pm y)$, mit x als Minuenden und y als Subtrahenden müssen ebenfalls zwei Fälle unterschieden werden:

1. Fall: gleiche Vorzeichen

Sind die Vorzeichen von Minuend und Subtrahend gleich, also $x - y$ oder $(-x) - (-y)$, so lässt sich die Berechnung der Differenz d auf die vorher dargestellte Addition ganzer Zahlen mit verschiedenen Vorzeichen zurückführen: Es gilt $d = x - y = x + (-y)$ bzw. $d = (-x) - (-y) = (-x) + y$.

2. Fall: verschiedene Vorzeichen

Bei unterschiedlichen Vorzeichen, also $x - (-y)$ oder $(-x) - y$, wird für das Ergebnis zuerst die Summe der Beträge von Minuend und Subtrahend gebildet. Das so erhaltene Ergebnis übernimmt das Vorzeichen des Minuenden.

► Beispiel 2.2.2

Man betrachte die Subtraktion der ganzen Zahl 7 von -10 . Zuerst berechnet man die Summe der Beträge $|-10| = 10$ und $|7| = 7$ und erhält $10 + 7 = 17$. Das Ergebnis übernimmt das Vorzeichen des Minuenden und lautet wie erwartet -17 . ◀

Multiplikation und Division werden genauso durchgeführt wie bei den natürlichen Zahlen, allerdings wird zunächst mit dem Betrag der jeweiligen Operanden gerechnet. Das so errechnete Ergebnis erhält ein positives Vorzeichen, genau dann, wenn beide Zahlen das gleiche Vorzeichen (++ oder --) aufweisen. Andernfalls, also bei verschiedenen Vorzeichen der Operanden (+- oder -+), ist das Resultat negativ.

Auch bei der Verwendung ganzer Zahlen in einer Rechenmaschine mit einer beschränkten Anzahl von Stellen muss das Verlassen des Wertebereichs berücksichtigt werden. Dies kann im Falle der Addition und Subtraktion nur dann auftreten, wenn beide Operanden das gleiche Vorzeichen haben. Wenn bei einer Rechnung ein Übertrag entsteht, so wird der Wertebereich entweder nach oben (beide Operanden positiv) oder nach unten (beide Operanden negativ) verlassen. Bei der Multiplikation ist mehr Vorsicht geboten: Hier muss in jedem Fall geprüft werden, ob bei wiederholter Addition der Betrag den Wertebereich verlässt.

Multiplikation und Division

2.2.3 Codierung der ganzen Zahlen im Rechner

Die gängigste Zahlendarstellung sind die natürlichen Zahlen. Mit den negativen Zahlen ergibt sich der Zahlenraum der ganzen Zahlen. Das Mitführen des Vorzeichens sowie seine Berücksichtigung beim Rechnen mit mechanischen Rechnern ist automatisiert nicht zu bewerkstelligen. Im Dual- oder Binärsystem kann das Vorzeichen dagegen durch ein reserviertes Bit ausgedrückt werden. Konvention dabei ist, dass eine führende 1 immer für eine negative Zahl und eine vorangestellte 0 ausschließlich für eine positive Zahl stehen. Ganze Zahlen können in einem Computer auf unterschiedliche Art dargestellt werden. Dies kann so weit gehen, dass die Codierung der Zahlen die Architektur eines Rechners wesentlich bestimmt. Manche Arten der Codierungen eignen sich besonders gut, um effizient arithmetische Operationen zu implementieren, andere wiederum, um Daten störungsfrei zu übertragen. Die Herausforderung bei der Darstellung ganzer Zahlen besteht darin, positive und negative Zahlen platzsparend zu speichern und grundlegende Rechenoperationen einfach auszuführen. Leider ist es bei der Implementierung zunächst einmal nicht möglich, aus dem Weglassen des Vorzeichens einen Nutzen zu ziehen. Das folgt aus der Notwendigkeit, bei Rechenaufgaben immer beide Vorzeichen durch die Maschine prüfen zu müssen. Bei der in einem späteren Abschnitt gezeigten binären Darstellung von Zahlen kann durch eine geschickte Codierung Speicherplatz eingespart werden.

2.2.3.1 Vorzeichenbehaftete Zahlen

Eine einfache Möglichkeit zur binären Codierung ganzer Zahlen ist neben den eigentlichen Stellen für das Zahlwort noch ein weiteres Bit zur Darstellung des Vorzeichens zu speichern. Eine derartige Zahl nennt man vorzeichenbehaftete Binärzahl (signed Binary, signed Integer). Die Implementierung ist trivial, da sich die bereits besprochenen Rechenregeln für ganze Zahlen einfach auf binäre Maschinenzahlen übertragen lassen. Für positive Zahlen scheint dieses Speichern des Vorzeichens überflüssig. Um beispielsweise den Bereich 0 bis 255 darzustellen, werden 8 Bit benötigt. Um nun den Bereich –255 bis 255 darzustellen, werden 9 anstatt 8 Bit verwendet: Man stellt das negative Vorzeichen durch eine 1 in dem 9. Bit dar und das entsprechend positive Vorzeichen durch eine 0. Ein erst auf den zweiten Blick ersichtliches Problem ist, dass es bei dieser Codierung zwei Darstellungen für die Null gibt: eine positive Null $x_{b2} = 00000000$ und eine negative Null mit $x_{b2} = 100000000$. Eine doppelte Codierung ist nicht effizient und kann zu Missverständnissen führen. Mit einer vorzeichenbehafteten Binärzahl mit n Bit lässt sich folglich ein Wertebereich von $[-2^{n-1} - 1, 2^{n-1} - 1]$ darstellen. Der Term $n - 1$ im Exponent ergibt sich durch das Vorzeichenbit, und es muss jeweils noch die (doppelte) Null abgezogen werden.

► Beispiel 2.2.3 – Vorzeichenbehaftete Binärzahl

Betrachtet werden die ganzen Zahlen –15 bis 15. Um die Zahlen von 0 bis 15 codieren zu können, werden 4 Bit oder 1 Nibble benötigt. Für das Vorzeichen selbst ist ein weiteres Bit vorzusehen. Ist der Wert des zusätzlichen Bits 1, ist die dazugehörige Zahl negativ. Die Abb. 2.10 zeigt alle möglichen Werte in binärer Form (Basis $b = 2$). ◀

Werden Zahlen in einem Rechner vorzeichenbehaftet codiert, besteht die Notwendigkeit, alle Fallunterscheidungen zur Berechnung der Addition und Subtraktion auch in die Maschine einzubauen. Das führt zu einer aufwendigen Implementierung. Daher ist eine Zahlendarstellung notwendig, die positive und negative Zahlen binär codieren kann, ohne dass derartig komplexe Schaltungen in der Rechenmaschine aufzubauen sind. An dieser Stelle erinnere man sich an die bereits eingeführte Berechnung von Differenzen unter Verwendung des b-Komplements oder (b-1)-Komplements. Im Folgenden sollen solche Komplementärzahlen als weitere Möglichkeit der Codierung ganzer Zahlen im Rechner diskutiert werden.

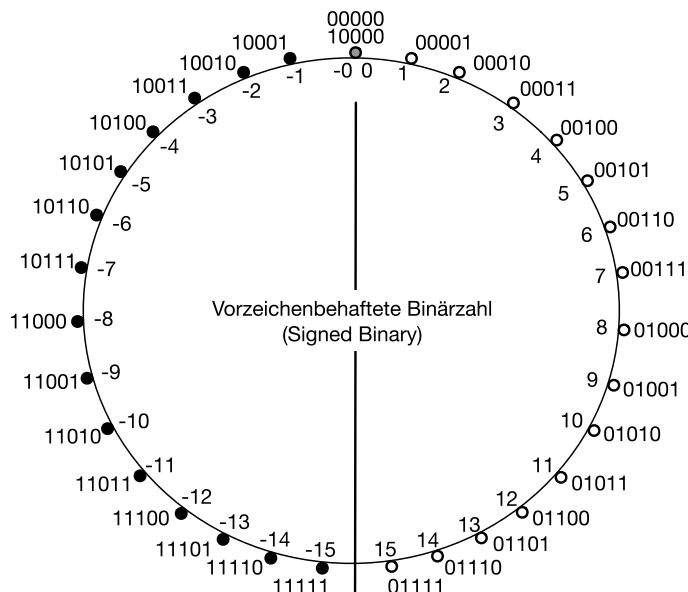


Abb. 2.10 Codierung einer vorzeichenbehafteten Binärzahl

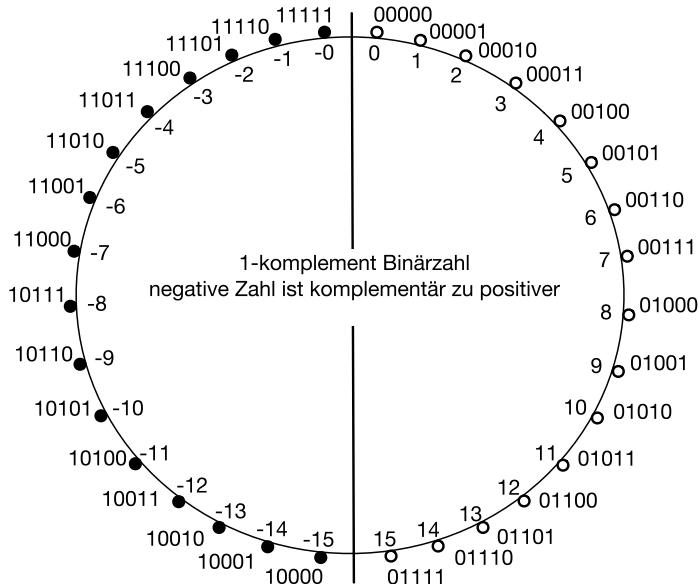
2.2.3.2 Dualzahlen im 1-Komplement

Entsprechend zur Komplementdarstellung im Dezimalsystem ist es möglich, eine derartige Zahl auch im binären Zahlensystem zu definieren. Im 1-Komplement ([b-1]-Komplement) werden die negativen Zahlen durch die zu der positiven Zahl komplementäre duale Zahl beschrieben. Bei der komplementären Zahl ist jede Ziffer der positiven Zahl invertiert, d. h., aus jeder 1 wird eine 0 und umgekehrt. Dies sorgt dafür, dass auch bei Zahlen im 1-Komplement die führende Ziffer anzeigt, ob die Zahl positiv (0) oder negativ (1) ist.

► Beispiel 2.2.4 – 1-Komplement Binärzahl

Betrachtet werden die ganzen Zahlen -15 bis 15 . Die negative Zahl ist die komplementäre Ziffernfolge der positiven. Negative Zahlen sind immer durch eine führende 1 gekennzeichnet. Die Abb. 2.11 zeigt die vollständige Codierung. ◀

Im Vergleich zu vorzeichenbehafteten Binärzahlen können Zahlen im 1-Komplement keinen größeren Wertebereich darstellen. Das heißt, mit n bit lässt sich ebenfalls ein Wertebereich von $[-2^{n-1} - 1, 2^{n-1} - 1]$ ausdrücken. So können weiterhin beispielsweise mit einer 5-stelligen Zahl 15 positive und 15 negative Zahlen sowie 2-mal die 0 codiert werden. Ein Vorteil der Komplementdarstellung liegt jedoch in der Arithmetik ganzer Zahlen, insbesondere in der Abbildung der Subtraktion auf die



■ Abb. 2.11 1-Komplement-Darstellung einer 5-stelligen Binärzahl

Addition. Zudem ist die Berechnung der Komplementärzahl durch ein ziffern- oder bitweises Invertieren zu realisieren. Allerdings erfordert das Problem der doppelten Null (z. B. 00000 und 111111) beim Rechnen im 1-Komplement besondere Aufmerksamkeit.

► Beispiel 2.2.5 – Addition/Subtraktion im 1-Komplement

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 5$ und $y_{b10} = -4$. Die entsprechenden Binärzahlen im 1-Komplement sind $x_{b2} = 00101$ und $y_{b2} = 11011$. Das korrekte Ergebnis sollte $s_{b10} = 1$ oder $s_{b2} = 00001$ sein. Führt man die Berechnung jedoch im binären Stellenwertsystem durch, erhält man:

$$\begin{array}{r} x \quad 00101 \\ y \quad 11011 \\ c \quad 111110 \\ s \quad 100000 \end{array}$$



Offensichtlich ist das Ergebnis falsch. Das Problem tritt im 1-Komplement immer dann auf, wenn die Operanden unterschiedliche Vorzeichen (höchstwertige Stellen) haben und quasi eine Subtraktion im (b-1)-Komplement stattfindet. Wie bereits im Abschnitt zum (b-1)-Komplement diskutiert, kann es

hierbei zu einem beabsichtigten Überlauf kommen, der einem Nulldurchgang entspricht. Beim Rechnen mit dem 1-Komplement muss in diesem Fall jedoch die doppelte 0 beachtet werden, da sonst ein Zählschritt fehlt. Der Fehler beim Nulldurchgang ist folglich zu korrigieren, indem der Überlauf zum Zwischenergebnis addiert wird.

► **Beispiel 2.2.6 – Addition/Subtraktion im 1-Komplement mit Null-durchgangskorrektur**

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 5$ und $y_{b10} = -4$. Die entsprechenden Binärzahlen im 1-Komplement sind $x_{b2} = 00101$ für 5 und $y_{b2} = 11011$ für -4. Die Addition wird im binären Stellenwertsystem berechnet. Das Ergebnis sollte $s_{b10} = 1$ oder $s_{b2} = 00001$ sein:

x	00101
y	11011
c	111110
z	100000
Korrektur des Überlaufs	
z'	00000
o	00001
s	00001



► **Beispiel 2.2.7 – Addition/Subtraktion im 1-Komplement ohne Null-durchgang**

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 2$ und $y_{b10} = -4$ mit entsprechenden Binärzahlen im 1-Komplement $x_{b2} = 00010$ und $y_{b2} = 11011$. Das korrekte Ergebnis sollte $s_{b10} = -2$ oder $s_{b2} = 11101$ sein.

x	00010
y	11011
c	00100
z	011101
Korrektur des Überlaufs	
z'	011101
o	00000
s	11101



Bei vorzeichenbehafteten Zahlen konnte der Überlauf an der höchstwertigen Stelle genutzt werden, um ein Verlassen des Wertebereichs zu erkennen. Wie gerade bei der Korrektur des Nulldurchgangs diskutiert, ist ein Überlauf (engl. „overflow“)

im 1-Komplement Teil regulärer und korrekter Berechnungen innerhalb einer Zahlenmenge. Bei der Verwendung der Komplementdarstellung ist eine Betrachtung der höchstwertigen Stelle der Operanden und des Ergebnisses erforderlich, um ein Verlassen des Wertebereichs zu erkennen: Ist die höchstwertige Stelle, also das Vorzeichen beider Eingaben gleich und unterscheidet sich die höchstwertige Stelle des Resultats hier von, so wurde der gültige Wertebereich verlassen.

► Beispiel 2.2.8 – Addition/Subtraktion im 1-Komplement

Gegeben seien die beiden Dezimalzahlen $x_{b10} = -7$ und $y_{b10} = -9$. Die entsprechenden Binärzahlen im 1-Komplement sind $x_{b2} = 11000$ und $y_{b2} = 10110$. Das korrekte Ergebnis sollte $s_{b10} = -16$ und damit außerhalb des Wertebereichs von $[-15, 15]$ sein:

$$\begin{array}{r} x \quad 11000 \\ y \quad 10110 \\ \hline c \quad 100000 \\ s \quad 101110 \end{array}$$

Das erhaltene Ergebnis $s_{b2} = 01110$ entspricht dabei der 14, und das Verlassen des Wertebereichs wird erkannt, da aus zwei negativen Operanden ein positives Resultat folgte. ◀

2.2.3.3 Dualzahlen im 2-Komplement

Eine weitere Variante der Komplementdarstellung im Binärsystem ist das 2-Komplement (b-Komplement). Eine negative Zahl entspricht hierbei der invertierten positiven Zahl plus 1. Wie im 1-Komplement zeigt die führende Ziffer das Vorzeichen der Zahl (positiv [0] oder vielmehr negativ [1]). Durch die Addition der 1 auf die invertierte Zahl entfällt automatisch die doppelte 0 und damit auch die im 1-Komplement diskutierte Problematik des zu beachtenden Nulldurchgangs. Durch den Wegfall der zweiten 0 erweitert sich der Wertebereich einer n -bit-Zahl im 2-Komplement um eine Zahl im negativen Bereich und ist damit $[-2^{n-1}, 2^{n-1} - 1]$. Die darstellbare Zahlenmenge ist im 2-Komplement nicht symmetrisch, d. h., dass es für die kleinste negative Zahl keine entsprechende positive Zahl gibt.

► Beispiel 2.2.9 – 5-stelliges 2-Komplement

Betrachtet werden die ganzen Zahlen $x_{b10} = -16$ bis $y_{b10} = 15$. Die Abb. 2.12 zeigt die 2-Komplement-Codierung mit 5 binären Stellen. ◀

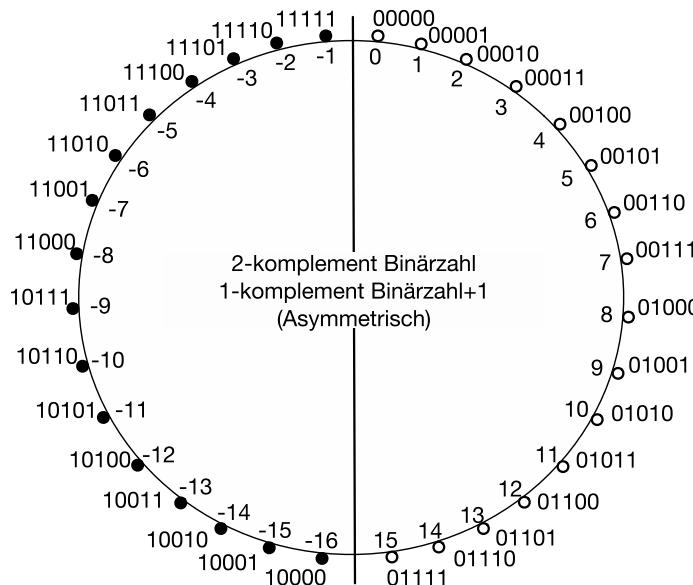


Abb. 2.12 2-Komplement-Darstellung einer 5-stelligen Binärzahl

Die Addition im 2-Komplementsystem wird wie im Dezimal- oder im Binärsystem gerechnet.

► Beispiel 2.2.10 – Addition im 2-Komplement

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 5$ und $y_{b10} = 4$. Die entsprechenden Binärzahlen im 2-Komplement sind $x_{b2} = 00101$ und $y_{b2} = 00100$. Die Addition wird im binären Stellenwertsystem berechnet. Das Ergebnis sollte die positive Zahl $s_{b10} = 9$ oder $s_{b2} = 01001$ sein.

$$\begin{array}{r} x \ 00101 \\ y \ 00100 \\ \hline c \ 01000 \\ s \ 01001 \end{array}$$



Die Subtraktion wird im 2-Komplement-System auf die Addition zurückgeführt, indem einfach eine negative Zahl addiert wird.

Subtraktion

► Beispiel 2.2.11 – Subtraktion im 2-Komplement

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 5$ und $y_{b10} = -4$. Die entsprechenden Binärzahlen im 2-Komplement sind $x_{b2} = 00101$ und $y_{b2} = 11100$. Die Addition wird im binären Stellenwertsystem berechnet. Das Ergebnis sollte $s_{b10} = 1$ oder $s_{b2} = 00001$ sein:

$$\begin{array}{r} x \quad 00101 \\ y \quad -11100 \\ c \quad \underline{111000} \\ s \quad 100001 \end{array}$$

Unter gewohnter Streichung des Übertrags ergibt sich wie erwartet $s_{b2} = 00001$. ◀

Multiplikation

Es sei an die Multiplikation im Dezimalsystem erinnert. Dabei kann man eine schrittweise Multiplikation für jede einzelne Stelle durchführen und am Ende die Teilergebnisse addieren. Im 2-Komplement geht man ebenso vor. Das Verfahren nennt man Multiplikation durch Addieren. Dabei wird jede Stelle des Multiplikanden mit dem anderen Operanden multipliziert und entsprechend nach links verschoben. Die dabei entstehenden Teilergebnisse werden dann addiert. Der genaue Algorithmus wurde bereits in ► Abschn. 2.1.3 für einfache Dualzahlen beschrieben und kann direkt auf die 2-Komplement-Zahlen übertragen werden.

► Beispiel 2.2.12 – Multiplikation im 2-Komplement

Gegeben sind die beiden Zahlen $x = 11101_{b2} = -3_{b10}$ und $y = 00100_{b2} = 4_{b10}$. $s = x \cdot y$ wird wie folgt berechnet:

$$\begin{array}{r} s_{b2} = 11101 \cdot 00100 \\ s_1 \quad 00000 \\ s_2 \quad 00000 \\ s_3 \quad 11101 \\ s_4 \quad 00000 \\ s_5 \quad 00000 \\ \hline s \quad 001110100 \end{array}$$

Das Ergebnis ist $s'_{b2} = 1110100$. Die 2 führenden Stellen werden gestrichen, und man erhält $s = 10100_{b2} = -12_{b10}$. ◀

Division

Im Binärsystem wird eine Division wie im Dezimalsystem durchgeführt. Durch den beschränkten Wertebereich ist zu prüfen, ob der Divisor einmal im Dividenden unterzubringen ist. Ist das der Fall, wird der Divisor von dem entsprechenden Teil des Dividenden subtrahiert. Ist das Ergebnis nicht teilbar, wird eine weitere Stelle des Dividenden nach unten gezogen. Dieses Vorgehen wiederholt sich so lange, bis das Resultat erneut geteilt werden kann. Dann wird wieder subtrahiert, bis das Endergebnis feststeht.

► Beispiel 2.2.13

Man betrachte die Zahlen $x_{b10} = 414$ und $y_{b10} = 23$ und entsprechend $x_{b2} = 110011110$ und $y_{b2} = 10111$. Der Quotient aus x und y errechnet sich dann wie folgt:

$$\begin{array}{r}
 (25) \quad 110011110 : 10111 = 10010 \\
 - \quad (23) \quad 10111 \\
 \quad \quad (c) \quad 00110 \\
 \quad \quad (2) \quad 00010 \\
 \quad \quad (5) \quad 00101 \\
 - \quad (0) \quad 00000 \\
 \quad \quad (c) \quad 00101 \\
 \quad (11) \quad 01011 \\
 - \quad (0) \quad 00000 \\
 \quad \quad (c) \quad 01011 \\
 \quad (23) \quad 10111 \\
 - \quad (23) \quad 10111 \\
 \quad \quad (c) \quad 00000 \\
 \quad (0) \quad 00000 \\
 - \quad (0) \quad 00000 \\
 \quad \quad (c) \quad 00000 \\
 (0) \quad 00000
 \end{array}$$

Mit $10010_{b2} = 18_{b10}$ erhält man das erwartete Ergebnis. ◀

2.3 Reelle Zahlen

Teilt man eine natürliche Zahl durch eine andere, reicht die Menge der natürlichen Zahlen nicht aus, das Ergebnis darzustellen. Diese Zahlen der Brüche natürlicher und ganzer Zahlen heißen Bruchzahlen oder rationale (lat. „ratio“, Verhältnis) Zahlen.

Neben den rationalen gibt es Zahlen, die nicht auf das Verhältnis zweier ganzer Zahlen zurückgeführt werden können. Angenommen, der bereits erwähnte antike König möchte die von den Untertanen eingenommenen Steuern in einem Grabmal anlegen. Eine beliebte Form eines Grabmals war in der Antike die Pyramide. Die Größe und Grundfläche eines solchen Bauwerks ergeben sich aus der Anzahl aufeinander geschichteter Steinquader. Rechteckige Quader haben ein festes Verhältnis aus Höhe und Breite zur Länge. Wollte der König eine Pyramide mit glatten Flächen errichten, war es erforderlich, die aus den rechtwinkligen Steinen erzeugten Stufen zu füllen. Dadurch wurden die Seiten des Grabmals geneigte Dreiecke. Soll die Seitenfläche der Grabstätte zusätzlich noch verkalkt werden, wäre es sinnvoll, die Fläche dieser zu bestimmen. Die Grundseite eines solchen Dreiecks ist gleich der Seitenlänge der Pyramide. Wie groß sind aber die zwei langen Seiten, die Kanten bis zur Spitze des Bauwerks? Dazu muss zunächst die Länge der Diagonalen der quadratisch angenomme-

Rationale Zahl

Irrationale Zahlen

nen Pyramide bestimmt werden. Diese Linie teilt die Grundfläche des Grabes in zwei rechtwinklige Dreiecke. Die Höhe, die Hälfte der Diagonalen des Bauwerks und die am Ende gesuchte Kantenlänge bilden wiederum ein rechtwinkliges Dreieck. Damit kann die Länge der Seite der Pyramide durch Berechnung zweier Aufgaben mit rechtwinkligen Dreiecken vorgenommen werden. Die Diagonale eines rechtwinkligen Dreiecks kann aus den Maßen des Quaders mit dem Satz des Pythagoras bestimmt werden. Angenommen, ein rechtwinkliges Dreieck hat die Kantenlänge 1 Längeneinheit, dann berechnet sich die Länge der Hypotenuse mit $1^2 + 1^2 = 2 = d^2$. Das Ergebnis dieser Rechnung kann nicht mit einem Bruch ganzer Zahlen dargestellt werden. Die entstehende Zahl $\sqrt{2}$ heißt irrational, ist also eine Ziffernfolge, die nicht zur Menge der rationalen Zahlen gehört.

Neben der Lösung quadratischer Gleichungen zeigt sich, dass die Nullstellen nicht konstanter Polynome, das sind Polynome vom Grad größer null, alle irrational sind. Daher nennt man Zahlen, die aus diesen Nullstellen hervorgehen, algebraische Zahlen. Es zeigt sich, dass es auch irrationale Zahlen gibt, die nicht algebraisch sind. Diese Zahlen heißen transzendente Zahlen. Die bekanntesten transzententen Zahlen sind die eulersche Zahl e und das Kreisverhältnis, die Kreiszahl π . Für beide Zahlen existiert kein Polynom, für das sich eine Nullstelle finden lässt.

Reelle Zahl

Die rationalen und die irrationalen Zahlen bilden zusammen die reellen Zahlen. Erst dieser Zahlenbegriff erlaubt Längen-, Flächen- und Volumenberechnungen von Objekten der realen Welt. Die rationalen Zahlen lassen sich immer als Bruch darstellen und die irrationalen Zahlen durch Angabe ihrer analytischen Konstruktion. Dieses Vorgehen ist oft nicht praktikabel. Zum einen existiert für viele Probleme aus Naturwissenschaft und Technik keine allgemeine analytische oder geschlossen zu berechnende Lösung. Zum anderen sollen die in den vorangegangenen Kapiteln entwickelten Verfahren zur numerischen Behandlung ganzer Zahlen auch auf reelle Zahlen angewendet werden. Der Grund ist einfach: Auf diese Weise lassen sich einmal entwickelte und aufgebauten Rechenwerke ebenso für reellwertige Probleme nutzen. Allerdings kann ein Rechner, sei es ein menschlicher oder ein maschineller, immer nur eine numerische Approximation des Rechenergebnisses erzeugen, da die Anzahl der Stellen begrenzt ist. Zahlen wie e und π besitzen jedoch unendlich viele Stellen.

Nachfolgend wird nun eine mögliche Darstellung reeller Zahlen als Ziffernfolge unter Verwendung sogenannter Dezimalbrüche eingeführt. Daran anschließend werden Kettenbrüche als alternative Darstellung diskutiert sowie die Entwicklung ausgewählter reeller Zahlen als Dezimal- und Kettenbrü-

che vorgestellt. Abschließend wird die Darstellung von reellen Zahlen und deren Operationen im Rechner als sogenannte Maschinenzahlen vorgestellt.

2.3.1 Dezimalbrüche

Dezimal- oder Zehnerbrüche sind eine Untermenge rationaler Zahlen der Form $\frac{x}{y}$, bei denen der Nenner y eine Potenz von 10 mit einer natürlichen Zahl als Exponent ist.

Dezimalbrüche sind insbesondere deshalb interessant, weil sie eine direkte Schreibweise als Dezimalzahl, genauer gesagt als Ziffernfolge einer solchen erlauben. Wie eine natürliche Zahl wird ein Dezimalbruch als eine Folge von Zeichen geschrieben, deren Wertigkeit durch die Stellung der Ziffer gegeben ist. Die Bruchstellen des Dezimalbruchs werden dabei durch Stellen mit einer Wertigkeit der Basis und einem negativen Exponenten dargestellt:

Definition 2.3.1 – Bruchstellen des Dezimalbruchs als Ziffernfolge

Die Ziffernfolge der Bruchstellen eines Dezimalbruchs ist:

$$0, d_{-1} d_{-2} \dots d_{-(m-1)} d_{-m}$$

Der Wert der Zahl mit der Wertigkeit der einzelnen Stellen ist gegeben durch:

$$x = 0 + \sum_{i=1}^m d_{-i} \cdot b^{-i} \quad (2.27)$$

► **Beispiel 2.3.1 – Bruchstellen eines Dezimalbruchs**

Die rationale Zahl $\frac{17}{100}$ ist ein Dezimalbruch. Die Bruchstellen der Zahl können direkt in der Form $1 \cdot 10^{-1}$ und $7 \cdot 10^{-2}$ geschrieben werden:

$$\frac{17}{100} = 0 + 1 \cdot 10^{-1} + 7 \cdot 10^{-2} = 0,17 \quad (2.28)$$

In der Definition ist bereits implizit ein sogenanntes Dezimaltrennzeichen, in diesem Fall das Komma, enthalten. Dieses ist notwendig, wenn ein Dezimalbruch nicht nur Bruchstellen, sondern auch einen ganzzahligen Anteil hat. Im kontinentalen Europa ist dieses Zeichen ein Komma, im englischsprachigen

Raum ein Punkt. Beide Schreibweisen gehen auf John Napier zurück, der in seinen Logarithmentafeln zunächst das Komma und dann den Punkt verwendete. Da sich die Logarithmentafeln großer Beliebtheit erfreuten und weitverbreitet waren, setzte sich die Trennzeichen Komma oder Punkt gegenüber anderen Möglichkeiten durch. Mithilfe des Dezimaltrennzeichens lässt sich eine Darstellung als Ziffernfolge für alle Dezimalbrüche definieren. Dabei stehen Stellen mit positiven Exponenten der Basis zur linken und mit negativen zur rechten Seite des Trennzeichens.

Definition 2.3.2 – Dezimalbruch als Ziffernfolge

Die Ziffernfolge $d_{n-1} \dots d_2 d_1 d_0, d_{-1} d_{-2} \dots d_{-(m-1)} d_{-m}$ eines Dezimalbruchs besteht aus n Zeichen vor und m Zeichen nach dem Trennzeichen. Der Wert der Zahl mit der Wertigkeit der einzelnen Stellen ist gegeben durch:

$$x = \sum_{i=0}^{n-1} d_i \cdot b^i + \sum_{i=1}^m d_{-i} \cdot b^{-i} \quad (2.29)$$

► Beispiel 2.3.2 – Ziffernfolge eines Dezimalbruchs

Die rationale Zahl $\frac{342}{100}$ ist ein Dezimalbruch. Die Stellen der Zahl können in der Form $3 \cdot 10^0$, $4 \cdot 10^{-1}$ und $2 \cdot 10^{-2}$ geschrieben werden:

$$\frac{342}{100} = 0 + 3 \cdot 10^0 + 1 \cdot 10^{-1} + 7 \cdot 10^{-2} = 3,42 \quad (2.30)$$



2.3.1.1 Dezimalbruchentwicklung

Dezimalbrüche lassen sich, direkt in eine Darstellung als Ziffernfolge überführen. Diese ist zudem endlich, sodass bei ausreichend Stellen eine exakte Darstellung des Dezimalbruchs als Ziffernfolge erfolgen kann. Solch eine endliche und exakte Darstellung als Dezimalbruch existiert ebenfalls für rationale Zahlen, deren Nenner in gekürzter Form lediglich die Primfaktoren 2 oder 5 aufweisen.

► Beispiel 2.3.3 – Endlicher Dezimalbruch

Die rationale Zahl $\frac{2}{8}$ hat gekürzt zu $\frac{1}{4}$ einen Nenner mit dem Primteiler 2, sodass eine endliche und exakte Darstellung als Dezimalbruch erzielt werden kann:

$$\frac{2}{8} \stackrel{\div 2}{=} \frac{1}{4} \stackrel{\cdot 25}{=} \frac{25}{100} = 0 + 2 \cdot 10^{-1} + 5 \cdot 10^{-2} = 0,25 \quad (2.31)$$



Aus den obigen Definitionen erschließt sich intuitiv, dass nicht jede rationale Zahl und folglich auch nicht jede reelle Zahl ein Dezimalbruch ist. Allerdings lässt sich jeder reellen Zahl eine unendliche Ziffernfolge analog zur Ziffernfolge eines Dezimalbruchs zuordnen. Die Umwandlung einer reellen Zahl in diese Ziffernfolge bezeichnet man als Dezimalbruchentwicklung. Typischerweise wird bei der Dezimalbruchentwicklung nur eine endliche Menge von Ziffern betrachtet, sodass ein Dezimalbruch entsteht, der die reelle Zahl annähert.

Bei Dezimalbrüchen selbst erzeugt eine Dezimalbruchentwicklung ab einer (endlichen) Stelle nur noch den Wert 0, und man spricht von einer endlichen Dezimalbruchentwicklung. Rationale Zahlen im Allgemeinen haben unendliche Dezimalbruchentwicklungen, die entstehenden Ziffern weisen jedoch eine Periode auf. Man spricht von einer periodischen Dezimalbruchentwicklung. Die Wiederholung kann direkt nach dem Komma auftreten oder an beliebiger Stelle nach dem Komma. Eine – wenn auch nicht immer effiziente – Möglichkeit der periodischen Dezimalbruchentwicklung ist mittels Division nach Schulmethode.

Für irrationale Zahlen entsteht bei der Dezimalbruchentwicklung eine unendliche, nicht periodische Ziffernfolge, sodass eine Darstellung als endliche Ziffernfolge eines Dezimalbruchs immer eine Näherung bleibt.

► Beispiel 2.3.4

Der Bruch $\frac{88}{3456}$ kann durch einfache schriftliche Division nach der Schulmethode in einen Dezimalbruch umgewandelt werden:

$$\begin{array}{r}
 8\ 8 \qquad : 3456 = 0,0254629629\dots \\
 -\ 0 \\
 \hline
 8\ 80 \\
 -\ 0 \\
 \hline
 8\ 800 \\
 -6\ 912 \\
 \hline
 1\ 8880 \\
 -1\ 7280 \\
 \hline
 1\ 6000 \\
 -1\ 3824 \\
 \hline
 21760 \\
 -20736 \\
 \hline
 \dots
 \end{array}$$

Das Beispiel zeigt, dass der resultierende Dezimalbruch periodisch mit der Vorperiode 0254 und der Periode 629 und damit einer Periodenlänge von 3 ist. Eine geläufige Darstellung der Periode ist die Verwendung des Überstrichs, am Beispiel $0,\overline{0254629}$. ◀

2.3.1.2 Abkürzende Schreibweise von Dezimalbrüchen

Wird eine sehr große oder eine sehr kleine Zahl als Ziffernfolge eines Dezimalbruchs dargestellt, kann es erforderlich sein, sehr viele Ziffern vor oder nach dem Komma schreiben zu müssen. Besonders umständlich und unübersichtlich wird es, wenn beispielsweise viele Nullen vor (große Zahlen) oder nach dem Komma (kleine Zahlen) dargestellt werden müssen. Der Aufwand, eine entsprechend große oder kleine Zahl aufzuschreiben, lässt sich mithilfe der wissenschaftlichen Exponentialschreibweise reduzieren. Die grundlegende Idee ist hierbei, die eigentlichen Ziffern und die Position des Dezimaltrennzeichens getrennt anzugeben.

Theorem 2.3.1 – Dezimalbruch als Bruchanteil mit Exponent

Sei x die Ziffernfolge eines Dezimalbruchs mit $n + m$ Stellen und dem Dezimaltrennzeichen zwischen den n Stellen für den ganzzahligen Anteil und den m Stellen für den Bruchanteil. Es gilt:

$$x = \pm \sum_{i=0}^{n-1} d_i \cdot b^i + \sum_{i=1}^m d_{-i} \cdot b^{-i} = \pm \left(\sum_{j=1}^{n+m} d_{-j} \cdot b^{-j} \right) \cdot b^n \quad (2.32)$$

Jedes x lässt sich also als Bruchanteil $\sum_{j=1}^{n+m} d_{-j} \cdot b^{-j}$ darstellen, wobei das Dezimaltrennzeichen durch Multiplikation mit b^n und damit der Basis b und der Anzahl der ganzen Stellen n wieder an die korrekte Stelle verschoben wird.

► Beispiel 2.3.5 – Dezimalbruch als Bruchanteil mit Exponent

Die Zahl $x = 027,82$ mit $n = 3$ und $m = 2$ kann als Bruchanteil mit Exponent geschrieben werden als $x = 0,02783 \cdot 10^3$. ◀

Die eingeführte Schreibweise sieht auf den ersten Blick wie eine triviale Umformung aus, die das angesprochene Ziel einer Einsparung bei sehr großen oder sehr kleinen Zahlen noch nicht erfüllen kann. So würde bei $n = 3$ und $m = 2$ und einem $x = 000,01$ die Darstellung $x = 0,00001 \cdot 10^3$ keine Vorteile bringen. Beiden Darstellungen sind gleich, da das Dezimaltrennzeichen an einer fest definierten Stelle steht: entweder zwischen ganzzahligem Anteil und Bruchanteil oder vor den $n + m$ Ziffern bei Darstellung als Bruchanteil und Exponent. Die bereits erwähnte Exponentialschreibweise erlaubt es, das

Dezimaltrennzeichen durch Änderung des Exponenten an eine beliebige und damit besser geeignete Stelle zu platzieren:

Definition 2.3.3 – Exponentialschreibweise

Sei x eine beliebige Ziffernfolge mit $n+m$ Stellen, so kann diese in Exponentialschreibweise dargestellt werden:

$$x = \pm \sum_{i=0}^{n-1} d_i \cdot b^i + \sum_{i=1}^m d_{-i} \cdot b^{-i} \stackrel{!}{=} \pm a \cdot b^e \quad (2.33)$$

Die Exponentialschreibweise besteht aus der sogenannten Mantisse a und der Basis b hoch dem Exponenten e . Die Mantisse a ist die Ziffernfolge eines Dezimalbruchs und kann das Dezimaltrennzeichen im Allgemeinen an beliebiger Stelle beinhalten. Die Zahl $e \in \mathbb{Z}$ heißt Exponent und setzt bzw. korrigiert die Position des Dezimaltrenzeichens der Mantisse auf dessen korrekte Position in der Zahl x .

Für die Exponentialschreibweise hat sich eine sogenannte Normalisierung der Mantisse bewährt, bei der man den Exponenten so wählt, dass die Mantisse immer im selben Wertebereich liegt. Aus der Schule ist eine Normalisierung der Mantisse auf $1 \leq a < 10$ (allgemein $1 \leq a < b$) sowohl aus der Physik als auch beispielsweise von vielen Taschenrechnern bekannt.

► Beispiel 2.3.6 – Exponentialschreibweise mit Normalisierung

Die Zahl $x = 0,000000000013$ kann mit der Mantisse $a = 1,3$ und dem Exponenten $e = -12$ geschrieben werden: $x = 1,3 \cdot 10^{-12}$. Die Zahl 1 Mio. kann mit $x = 1,0 \cdot 10^6$ ausgedrückt werden. ◀

Wie aus dem Beispiel ersichtlich, erlaubt die 22 Exponentialschreibweise, insbesondere mithilfe der Normalisierung, sehr große wie auch sehr kleine Zahlen kompakt darzustellen und ggf. sogar die Genauigkeit zu erhöhen. Da die letztendliche Position des Dezimaltrenzeichens in der Exponentialschreibweise nicht fest ist, ist sie eine Darstellung der sogenannten Gleitkommazahlen. Auf deren Vor- und Nachteile sowie deren Arithmetik wird im übernächsten Abschnitt zu Maschinenzahlen noch einmal detaillierter eingegangen.

Bei transzendenten Zahlen wie e oder π bewirkt die Schreibweise mit Mantisse, Basis und Exponent nur eine Verschiebung des Dezimaltrenzeichens. Da jede Ziffer dieser Zahlen unterschiedlich ist, tritt keine Vereinfachung auf.

2.3.2 Kettenbrüche

Bereits im ausgehenden 16. Jahrhundert suchte man nach einer Möglichkeit, insbesondere irrationale Zahlen wie $\sqrt{2}$ elegant darzustellen und zu berechnen. Die italienischen Mathematiker Rafael Bombelli und Pietro Cadaldi veröffentlichten die ersten Arbeiten zu sogenannten Kettenbrüchen. Der niederländische Astronom Christiaan Huygens berechnete die Übersetzungswerte der Zahnräder eines mechanischen Modells des Sonnensystems mit dieser Methode. Kettenbrüche können als alternative Darstellung beispielsweise zu den bisher diskutierten Dezimalbrüchen angesehen werden. Sie eignen sich insbesondere zur kompakten Darstellung und Näherung reeller Zahlen, werden jedoch weniger zum Rechnen mit ihnen selbst verwendet.

Definition 2.3.4 – Kettenbruch

Ein Kettenbruch besitzt die Form:

$$h + \cfrac{i}{j + \cfrac{k}{l + \cfrac{m}{\dots}}} \quad (2.34)$$

Die grundlegende Form entspricht dabei der häufig aus der Schule bereits bekannten Darstellung als gemischter Bruch, wobei der Nenner des Bruchanteils wieder ein gemischter Bruch ist. Dieses Schema lässt sich immer wieder fortsetzen, sodass sich jede rationale Zahl als endlicher und jede reelle Zahl als unendlicher Kettenbruch darstellen lassen.

Eine für die Darstellung häufig verwendete Form ist der reguläre Kettenbruch:

Definition 2.3.5 – Regulärer Kettenbruch

Bei einem regulären Kettenbruch sind alle Zähler der Bruchanteile 1:

$$h + \cfrac{1}{j + \cfrac{1}{l + \cfrac{1}{\dots}}} \quad (2.35)$$

Bei einem regulären Kettenbruch muss nur die Folge der ganzzahligen Anteile h, j, \dots der gemischten Brüche zur Darstellung bzw. Speicherung herangezogen werden.

► **Beispiel 2.3.7 – Regulärer Kettenbruch**

Gegeben sei die rationale Zahl $\frac{64}{26}$. Die Darstellung als regulärer Kettenbruch ist dabei:

$$\frac{64}{26} = 2 + \frac{1}{2 + \frac{1}{6}}$$

Eine kompakte Darstellung wäre hierbei [2; 2, 6], wobei das Semikolon häufig verwendet wird, um den ganzzahligen Anteil (2) der reellen Zahl von den Teilnennern (2, 6) des Bruchanteils zu unterscheiden. ◀

Zur Umwandlung von reellen Zahlen in einen regulären Kettenbruch, der sogenannten Kettenbruchzerlegung, wird ein Rechenverfahren benötigt. Bereits im Altertum beschrieb der griechische Gelehrte Euklid ein Verfahren zur Bestimmung eines grundlegenden Maßes für zwei unterschiedliche Strecken. Gesucht ist dabei eine Zahl, deren Vielfaches beide Distanzen darstellen kann. Das Problem entspricht der Suche nach dem größten gemeinsamen Teiler. Die Methode zur Bestimmung des größten gemeinsamen Teilers und seine modernen Varianten heißen euklidischer Algorithmus. Es zeigt sich, dass die bei der Anwendung des euklidischen Algorithmus auftretenden Quotienten bei der Division mit Rest auch die Teilnenner des gesuchten regulären Kettenbruchs sind.

Definition 2.3.6 – Euklidischer Algorithmus

Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler $ggT(a, b) = f_e(a, b)$ mit der rekursiven Funktion

$$f_e(a, b) = \begin{cases} a & b = 0 \\ f_e(b, a \bmod b) & b \neq 0 \end{cases}$$

Auf eine ausführliche Einführung und den Beweis des euklidischen Algorithmus soll an dieser Stelle verzichtet werden. Die Grundidee des Algorithmus ist, so lange iterativ die jeweils kürzere Strecke von der längeren Strecke abzuziehen, bis die kürzere Strecke 0 ist. Die verbleibende längere Strecke ist dann der gesuchte größte gemeinsame Teiler. In der oben gezeigten modernen Variante wird die Subtraktion durch eine Division mit Rest unter Verwendung der Modulo-Operation ersetzt.

Für die Kettenbruchzerlegung relevant ist dabei folgende Überlegung zur Zerlegung einer Zahl bei Division mit Rest:

$$a = \left\lfloor \frac{a}{b} \right\rfloor \cdot b + (a \bmod b) \quad (2.36)$$

Der Term $\lfloor \frac{a}{b} \rfloor$ ist dabei jeweils der Teilnenner des regulären Kettenbruchs und entspricht den Quotienten, die bei der Anwendung des euklidischen Algorithmus auftreten.

2

► Beispiel 2.3.8 – Kettenbruchzerlegung mittels euklidischem Algorithmus

Gegeben sei die bereits bekannte rationale Zahl $\frac{64}{26}$. Zuerst betrachte man die entstehenden iterativen Aufrufe des euklidischen Algorithmus für $f_e(64, 26)$:

$$f_e(64, 26) \rightarrow f_e(26, 12) \rightarrow f_e(12, 2) \rightarrow f_e(2, 0) \rightarrow ggT = 2$$

In ausführlicher Darstellung werden die Quotienten (fett markiert) bei der Division mit Rest der einzelnen Iterationen sichtbar:

$$64 \text{ mod } 26 : 64 = 2 \cdot 26 + 12$$

$$26 \text{ mod } 12 : 26 = 2 \cdot 12 + 2$$

$$12 \text{ mod } 2 : 12 = 6 \cdot 2 + 0$$

Stellt man das Vorgehen schrittweise als gemischte Brüche dar, wird die Zerlegung noch deutlicher:

$$\begin{aligned} \frac{64}{26} &= 2 + \frac{12}{26} = 2 + \frac{1}{\frac{26}{12}} \\ &= 2 + \frac{1}{2 + \frac{2}{12}} = 2 + \frac{1}{2 + \frac{1}{\frac{12}{2}}} \\ &= 2 + \frac{1}{2 + \frac{1}{6}} \end{aligned}$$



Mithilfe des euklidischen Algorithmus kann also der jeweilige Kettenbruch einer reellen Zahl berechnet werden. Dabei ist zu prüfen, ob das Verfahren abbricht, da es im Fall transzgender Zahlen nicht konvergiert.

Ähnlich wie bei den Ziffernfolgen der Dezimalbrüche kann es auch bei Kettenbrüchen zu unendlichen Folgen kommen: Die Lösungen quadratischer Gleichungen ergeben oft einen unendlichen periodischen Kettenbruch. Es gibt aber auch irrationale Zahlen mit unendlichen, nicht periodischen Kettenbrüchen. Die Kreiszahl und die eulersche Zahl sind Beispiele für solche Zahlen.

2.3.3 Dezimal- und Kettenbrüche irrationaler Zahlen

Die Dezimalbruchdarstellung irrationaler Zahlen als Ergebnis des Ziehens einer Wurzel lässt sich mit Kettenbrüchen berechnen.

Theorem 2.3.2 – Kettenbruchdarstellung $\sqrt{2}$

Der Kettenbruch der irrationalen Zahl $\sqrt{2}$ ist

$$\sqrt{2} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{\ddots}}}}$$

Beweis

$$\begin{aligned}\sqrt{2} &= 1 + \sqrt{2} - 1 = 1 + (\sqrt{2} - 1) = 1 + \frac{(\sqrt{2} - 1)(1 + \sqrt{2})}{1 + \sqrt{2}} \\ &= 1 + \frac{(2 + \sqrt{2}) - (\sqrt{2} - 1)}{1 + \sqrt{2}} = 1 + \frac{1}{1 + \sqrt{2}}\end{aligned}$$

Setzt man $\sqrt{2} = 1 + \frac{1}{1 + \sqrt{2}}$ immer wieder ein:

$$\sqrt{2} = 1 + \frac{1}{1 + 1 + \frac{1}{1 + \sqrt{2}}} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{\ddots}}}}$$

■

Damit ergibt sich der unendliche Dezimalbruch der Wurzel aus 2 zu $\sqrt{2} = 1,414213562373095$. Die einzelnen Näherungen des Kettenbruchs sind in □ Tab. 2.3 aufgelistet.

Die irrationalen Zahlen sind entweder algebraisch oder transzendent. Bedingt durch die Definition der algebraischen Zahlen kann die Dezimalbruchdarstellung eines solchen Zahlwertes durch eine Nullstellensuche des dazugehörigen Poly-

□ Tab. 2.3 Näherungen der Wurzel aus 2

Näherungsbruch	Dezimalbruch
$\frac{1}{1}$	1
$\frac{3}{2}$	1,5
$\frac{7}{5}$	1,4
$\frac{17}{12}$	1, 416̄6
$\frac{41}{29}$	1, 413793103448276
$\frac{99}{70}$	1, 414285714

noms gefunden werden. Numerisch lassen sich diese Nullstellen mithilfe eines Iterationsalgoritmus bestimmen:

2

Theorem 2.3.3 – Nullstellensuche mit dem Newton-Verfahren

Die Nullstellen einer Funktion lassen sich mit

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

iterativ bestimmen.

Beweis

Sei $f(x)$ eine beliebige Funktion und sei x_0 eine beliebige Näherung von $f(x) = 0$. Dann ist $f'(x_0)$ die Tangente im Punkt x_0 . Diese Tangente schneidet die x -Achse bei x_1 . Mit der Gerade $f(x_0)(x_1 - x_0) + f(x_0)$ folgt dann:

$$\begin{aligned} f(x_0) + f'(x_0)(x_1 - x_0) &= 0 \\ f(x_0) + f'(x_0)x_1 + f'(x_0)x_0 &= 0 \\ x_1 &= \frac{f'(x_0)x_0 - f(x_0)}{f'(x_0)} \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

oder

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

■

Dieses iterative Verfahren wurde zuerst 1671 von Isaac Newton zur Lösung algebraischer Gleichungen beschrieben und 1690 von Joseph Raphson (1648–1715) mithilfe der heute bekannten Iterationsgleichung formalisiert. Deshalb wird diese Methode auch *Newton-Raphson-Verfahren* genannt (Abb. 2.13).

Theorem 2.3.4 – Wurzelziehen durch Nullstellensuche

Die Dezimalbruchdarstellung der $\sqrt[n]{x}$ kann näherungsweise bestimmt werden mit

$$x_{i+1} = \frac{(n-1)x_i^n + a}{nx^{n-1}}$$

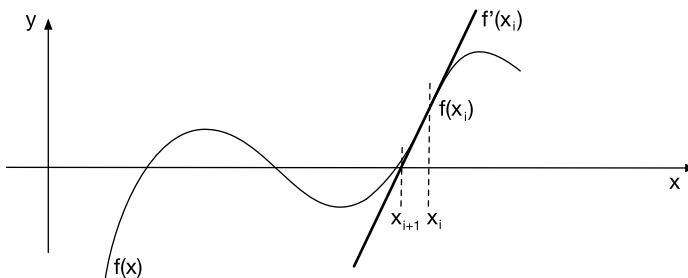


Abb. 2.13 Newton-Verfahren

Beweis

Sei $x^n = a$ eine beliebige Wurzel. Dann gilt $f(x) = x^n - a = 0$. Unter Anwendung des Newton-Raphson-Verfahrens erhält man

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^n - a}{nx_i^{n-1}} = \frac{nx_i x_i^n - x_i^n + a}{nx_i^{n-1}} \\ &= \frac{nx_i^n - x_i^n + a}{nx_i^{n-1}} = \frac{(n-1)x_i^n + a}{nx_i^{n-1}} \end{aligned}$$

■

Für $n = 2$ geht der Algorithmus zur Bestimmung des Dezimalbruchs einer Wurzel in das Heron-Verfahren über. Das Heron-Verfahren ist ein geometrisches Näherungsverfahren zur Berechnung einer Quadratwurzel und ist bereits für die Babylonier unter ihrem König Hammurapi I. (1750 v. Chr.) nachgewiesen. Der Grieche Heron von Alexandria veröffentlichte die Methode in seinem Buch Metrica und wurde so zum Namensgeber.

Das Heron-Verfahren basiert auf der Flächenberechnung eines quadratischen Rechtecks. Da sich die Fläche aus dem Quadrat der Kantenlänge a , $A = a \cdot a = a^2$ ergibt und die Wurzel die Umkehroperation der Quadratfunktion ist, entspricht die Berechnung der Kantenlänge a der Operation Wurzelziehen. Das Heron-Verfahren beruht darauf, dass ein gegebenes Rechteck der Fläche $A = a \cdot b$ iterativ in ein Quadrat gleicher Fläche umgewandelt werden kann. Die Wurzel einer Quadratzahl lässt sich so geometrisch bestimmen: Zunächst wird ein Rechteck mit der ersten Kantenlänge a_0 der Quadratzahl und der zweiten Kantenlänge $b_0 = 1$ gewählt. Durch Addition der beiden Kantenlängen und durch Teilen des Ergebnisses durch 2 wird im nächsten Schritt eine neue Kantenlänge a berechnet: $a_1 = \frac{a_0+b_0}{2}$. Die neue Kantenlänge von b_1 für ein flächengleiches Rechteck kann dann mit $b_1 = \frac{A}{a_1}$ ermittelt werden. Durch iteratives Anwenden der beiden Gleichungen $a_i = \frac{a_{i-1}+b_{i-1}}{2}$

und $b_i = \frac{a}{a_i}$ ist es möglich, eine Näherungslösung der Wurzel einer Quadratzahl zu berechnen.

2

Theorem 2.3.5 – Ableitung des Heron-Verfahrens

Die Dezimalbruchdarstellung der $\sqrt[3]{x}$ kann näherungsweise bestimmt werden mit

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{a}{x_i} \right)$$

Beweis

Sei $x^2 = a$ eine beliebige Wurzel. Dann gilt $f(x) = x^2 - a = 0$. Unter Anwendung des Newton-Raphson-Verfahrens erhält man das bereits im Altertum bekannte Heron-Verfahren

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{2x_i^2 - x_i^2 + a}{2x_i} \\ &= \frac{x_i^2 + a}{2} = \frac{1}{2}(x_i + \frac{a}{x_i}) \end{aligned}$$

■

► Beispiel 2.3.9 – Wurzel aus 16

Mit dem Rechner soll die Wurzel aus 16 mit dem Heron-Verfahren bestimmt werden: Zunächst wird ein Rechteck mit den Kantenlängen $a = 16$ und $b = 1$ gewählt. Dessen Fläche ist offensichtlich 16. Unter der Annahme, dass der Taschenrechner neben der Multiplikation auch eine Taste zur Division und eine Dezimalstelle besitzt, sind die von einem menschlichen Rechner zu drückenden Tasten in der rechten Spalte der Tabelle angegeben. Allerdings muss sich der Rechner Zwischenergebnisse notieren.

i	$a_i = \frac{a_{i-1} + b_{i-1}}{2}$	$b_i = \frac{a}{a_i}$
1	$a_1 = \frac{16+1}{2} = 8,5$	$b_1 = \frac{16}{8,5} = 1,882$
2	$a_2 = \frac{8,5+1,882}{2} = 5,191$	$b_2 = \frac{16}{5,191} = 3,082$
3	$a_3 = \frac{5,191+3,082}{2} = 4,137$	$b_3 = \frac{16}{4,137} = 3,867$
4	$a_4 = \frac{4,137+3,867}{2} = 4,002$	$b_4 = \frac{16}{4,002} = 3,998$
5	$a_5 = \frac{4,002+3,998}{2} = 4$	$b_5 = \frac{16}{4} = 4$

◀

Um die Betrachtung zu komplettieren, sei am Ende noch auf zwei wichtige transzendenten Zahlen eingegangen: die eulersche Zahl e und die Kreiszahl π .

Die eulersche Zahl kann ebenfalls durch Kettenbrüche approximiert werden. Die Berechnung ist aufwendig. So gelang Euler selbst nur eine Näherung von 23 Ziffern. Erst im Jahr 1949 schaffte es John von Neumann auf dem ersten digitalen Universalrechner, der ENIAC, die transzendenten Zahl e mit mehr als 1000 Stellen zu approximieren.

► Beispiel 2.3.10 – Dezimalbruch der eulerschen Zahl

Die eulersche Zahl ist definiert durch den Grenzwert der Reihe

$$\begin{aligned} e &= \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \dots \\ &= \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots \\ &= \sum_{k=0}^{\infty} \frac{1}{k!} \end{aligned}$$

Der unendliche Dezimalbruch des Grenzwertes ist

$$\begin{aligned} e = & 2,71828182845904523536028747135266249 \\ & 77572470936999595749669 \dots \end{aligned}$$



Im Gegensatz zur eulerschen Zahl hat man für π noch keine geschlossene Darstellung gefunden. Damit hebt gerade die Kreiszahl ein Problem der Darstellung reeller Zahlen hervor: Die Ziffernfolge einer irrationalen Zahl müsste unendlich viele Ziffern hinter dem Trennzeichen aufweisen, um die Zahl exakt darstellen zu können. Das bedeutet, Zahlworte in Trennzeichenschreibweise von irrationalen Zahlen können immer nur Näherungen dieser Zahl sein. Im Ergebnis können somit nur rationale Zahlen in einem Positionssystem mit festgelegter Anzahl von Stellen exakt dargestellt werden.

► Beispiel 2.3.11 – Darstellung der Kreiszahl π

Die Kreiszahl π ist definiert als das Verhältnis des Umfangs U eines Kreises zu seinem Durchmesser d : $\pi := \frac{U}{d}$. Die Kreiszahl kann ebenfalls durch einen Kettenbruch angenähert werden:

$$\begin{aligned} \pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{\ddots}}}}} \end{aligned}$$

Damit ergibt sich π in Dezimalbruchschreibweise:

$$\pi = 3,14159265359 \dots \blacktriangleleft$$

► **Beispiel 2.3.12**

Die rationale Zahl $\frac{1}{3} = 0,33333\dots$ kann im Gegensatz zur Zahl $\frac{1}{4} = 0,25$ nicht exakt in einem Positionssystem zur Basis 10 dargestellt werden. ◀

2.3.4 Maschinenzahlen

2.3.4.1 Darstellung

Die beschränkte Darstellung einer reellen Zahl in einem Rechner heißt Maschinenzahl. Eine reelle Maschinenzahl besteht aus der Basis oder Grundzahl des Zahlensystems $b \in N$, dem Exponenten e , der Mantisse $f = 0, d_1d_2\dots d_m = \pm(\sum_{j=1}^{\infty} d_j b^{-j})$. Diese Maschinenzahl ist normalisiert: für $x \neq 0 : d_1 \neq 0$, und es gilt $b^{-1} \leq |f| < 1$. Die Anzahl der Stellen der Mantisse und des Exponenten ist beschränkt. Die Mantisse besitzt m Stellen und der Exponent ist auf eine Zahl zwischen $r \in N$ und $R \in N$ begrenzt: $r \leq e \leq R$.

Definition 2.3.7 – Gleitkommazahl, Datentyp Float

Eine Gleitkommazahl wird als

$$x = \pm f \cdot b^e = \pm \left(\sum_{j=1}^m d_j b^{-j} \right) \cdot b^e \quad (2.37)$$

beschrieben, mit r dem minimalen und R dem maximalen Exponenten. Damit kann die Maschinenzahl in Gleitkommadarstellung mit einer Mantisse der Länge m mit der Struktur

$$\mathbb{M}(b, m, r, R) \quad (2.38)$$

charakterisiert werden.

Eine Gleitkommazahl ist die Realisierung einer halblogarithmischen Zahlendarstellung in einem Rechner. Bei der Abbildung einer reellen Zahl auf eine Gleitkommazahl handelt es sich um eine näherungsweise Beschreibung. Die reelle Zahl wird also approximiert. Bedingt durch die endliche Anzahl von Stellen des Exponenten existieren eine kleinste und eine größte Gleitkommazahl, die im Rechner gespeichert und dargestellt werden können:

► **Beispiel 2.3.13 – Gleitkommazahl**

Die Zahl $x = 0,452$ kann als Gleitkommazahl zur Basis 10 als $x = 4,52 \cdot 10^{-1}$ geschrieben werden. Aus $x = -0,000452$ wird $x = (-1) \cdot 4,452 \cdot 10^{-4}$ und aus $x = 45.200.000$ wird $x = 4,52 \cdot 10^7$.

◀

Theorem 2.3.6 – Kleinstes darstellbare Gleitkommazahl

Die zu einer Basis b mit Mantissenlänge m und minimalem Exponenten r in Gleitkommadarstellung abbildbare Zahl ist $x_{\min} = b^{r-1}$.

Beweis

$$\begin{aligned} x_{\min} &= 0,100 \cdots 0 \cdot b^r \\ &= (0 + 1 \cdot b^{-1} + 0 \cdot b^{-2} \cdots + 0 \cdot b^{-m}) \cdot b^r \end{aligned} \quad (2.39)$$

$$= b^{-1} \cdot b^r = b^{r-1} \quad (2.40)$$

■

Theorem 2.3.7 – Größtes darstellbare Gleitkommazahl

Die zu einer Basis b mit Mantissenlänge m und maximalem Exponenten R in Gleitkommadarstellung abbildbare Zahl ist $x_{MAX} = (1 - b^{1-m})b^R$.

Beweis

Sei $x_{MAX} = (0, a_1 a_2 a_3 \cdots a_m) \cdot b^R$ eine Gleitkommazahl mit einer m -steligen Mantisse und einem maximalen Exponenten $R \in \mathbb{N}$. Dann ist die größte Zahl durch $a_1 = a_2 = \cdots = a_i = \cdots a_m = b - 1$ gegeben. Einsetzen in (2.37) ergibt

$$x_{MAX} = \left(0 + \sum_{j=1}^m d_j b^{-j} \right) \cdot b^R \quad (2.41)$$

Da alle a_i gleich sind, gilt:

$$x_{MAX} = \left(\sum_{j=1}^m a_i b^{-j} \right) \cdot b^R \quad (2.42)$$

oder nach ausklammern

$$x_{MAX} = \left[a_i \cdot \sum_{j=1}^m b^{-j} \right] \cdot b^R \quad (2.43)$$

Dieser Ausdruck kann durch Abzug von a_i und Anpassen der Grenzen der Summe mithilfe der geometrischen Reihe gelöst werden:

$$x_{MAX} = \left[a_i \cdot \sum_{j=0}^m b^{-j} - a_i \right] \cdot b^R = \left[a_i \cdot \frac{1 - b^{-1-m}}{1 - b^{-1}} - a_i \right] \cdot b^R$$

Nach Erweitern des Bruches mit $\frac{b}{b}$ folgt

$$\begin{aligned} x_{MAX} &= \left[a_i \cdot \sum_{j=0}^m b^{-j} - a_i \right] \cdot b^R \\ &= \left[a_i \cdot \frac{b - b^{-m}}{b - 1} - a_i \right] \cdot b^R \\ &= \left[(b - 1) \cdot \frac{b - b^{-m}}{b - 1} - (b - 1) \right] \cdot b^R \\ &= [b - b^{-m} \cdot (-b + 1)] \cdot b^R \\ &= [1 - b^{-m}] \cdot b^R \end{aligned}$$

■

Maschinengenauigkeit

Die Maschinengenauigkeit einer Gleitkommazahl ist durch die Approximation der reellen Zahl auf die Maschinenzahl festgelegt. Allgemein kann eine beliebige reelle Zahl durch eine Reduktionsabbildung genannte Funktion approximiert werden:

$$fl(x) = \mathbb{R} \rightarrow \mathbb{M}(b, m, r, R) \quad (2.44)$$

In der Regel kann diese Funktion durch Runden realisiert werden.

Definition 2.3.8 – Standardrundung

Sei $x \in \mathbb{R}$ und $x \in [x_{MIN}, x_{MAX}]$ und ist x definiert mit

$$\pm \left(\sum_{j=1}^m d_j b^{-j} \right) \cdot b^e, \text{ dann ist die Standardrundung}$$

$$x = \begin{cases} \pm \left(\sum_{j=1}^m d_j b^{-j} \right) \cdot b^e & \text{falls } d_{m+1} < \frac{b}{2} \\ \pm \left(\sum_{j=1}^m d_j b^{-j} + b^{-m} \right) \cdot b^e & \text{falls } d_{m+1} \geq \frac{b}{2} \end{cases} \quad (2.45)$$

Die letzte Ziffer der Mantisse wird um 1 erhöht, wenn die nächste Stelle größer als $\frac{b}{2}$ ist.

Jede durch Rundung gewonnene Maschinenzahl besitzt einen Rundungsfehler.

Definition 2.3.9 – Rundungsfehler

Seien $x \in \mathbb{R}$ eine reelle Zahl und $f(x) = \mathbb{R} \rightarrow \mathbb{M}(b, m, r, R)$ die Reduktionsabbildung, dann ist der Rundungsfehler durch

$$\epsilon = \left| \frac{f(x) - x}{x} \right| \leq \frac{\frac{b}{2} \cdot b^{-m}}{b^{-1} \cdot b^e} = \frac{b^{1-m}}{2} \quad (2.46)$$

gegeben. ϵ wird relative Maschinengenauigkeit genannt.

Die Reduktionsabbildung kann dann auch als

$$f(x) = x \cdot (1 + \epsilon) \quad (2.47)$$

geschrieben werden.

2.3.4.2 Numerik

Die erste Maschine, die es erlaubte mit reellen Zahlen zurechnen, ist der Rechenschieber. Vor der Erfindung der integrierten Schaltungen in Halbleitertechnik war er das Standardwerkzeug der Ingenieure. Die Abb. 2.14 zeigt einen einfachen Rechenschieber mit den beiden Skalen C und D . Die weiteren Skalen T, K, A, L, S werden erst einmal nicht betrachtet. In der Abbildung ist zu erkennen, dass ein Rechenschieber aus drei Bauteilen besteht: dem Körper als grundlegende Trägerstruktur, der Zunge als in den Körper verschiebbar eingelagertes Element und dem mit einer Markierung versehenen durchsichtigen Läufer. Der Läufer kann über Körper und Zunge frei bewegt werden. Sowohl Körper als auch Zunge sind mit einer identischen logarithmischen Skala ausgestattet. Wichtig ist, dass Läufer und Zunge unabhängig voneinander bewegt werden können. Bei der Fertigung eines Rechenschiebers kommt es darauf an, die Skalen sehr genau und fein zu arbeiten. Gleiches gilt für die Markierung auf dem Läufer. Mit der Genauigkeit der Skalen ist direkt die mögliche Rechengenauigkeit der Maschine verknüpft. In der Vergangenheit wurden auch große Rechenzylinder mit logarithmischen Skalen gefertigt.

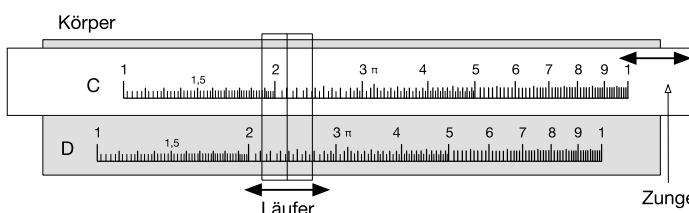


Abb. 2.14 Einfacher Rechenschieber

Logarithmen

2

tigt, um so die Genauigkeit der in den folgenden Abschnitten beschriebenen Verfahren zu erhöhen.

Mithilfe von Logarithmen lassen sich die aufwendigeren arithmetischen Operationen wie die Multiplikation, die Division, das Potenzieren und das Radizieren („Wurzel ziehen“) wieder auf einfache Rechenoperationen herunterbrechen. Wie bei den ganzen Zahlen können diese Operationen einfach auf die Addition zurückgeführt werden.

Der Rechenschieber nutzt die Reduktion der Multiplikation auf die Addition durch Logarithmen aus. In der Schule haben wir gelernt, dass der Logarithmus die Umkehrfunktion des Potenzierens ist. Eine positive Zahl x aus den reellen Zahlen zu einer Basis b entspricht gerade dem Wert y mit $y = b^x$. Dies löst das Gleichungssystem $x = \log_b(y)$. Daher ist $\log_{10}(3, 5) \approx 0,544$.

$$\log_x(a \cdot b) = \log_x a + \log_x b \quad (2.48)$$

$$\log_x\left(\frac{a}{b}\right) = \log_x a - \log_x b \quad (2.49)$$

$$\log_x(a^b) = b \log_x a \quad (2.50)$$

$$\log_x(\sqrt[b]{a}) = \frac{1}{b} \log_x a \quad (2.51)$$

Diese Regeln lassen sich zur Vereinfachung von Rechenoperationen nutzen, wenn es möglich ist, den Logarithmus einer Zahl zu bestimmen. Als es noch keine Taschenrechner gab, verwendete man dazu Logarithmentafeln. In der Regel waren diese 3-, 4- oder 5-stellig. Je mehr Stellen eine Logarithmentafel hatte, umso genauer, aber auch umfangreicher war sie. Werden Logarithmentafeln für den Logarithmus zur Basis 10 erstellt, kann eine schöne Eigenschaft der 10er-Logarithmen ausgenutzt werden: Ein Logarithmus besteht immer aus einer Kennzahl und einer Mantisse. Dabei sind die Kennzahl die Ziffernfolge vor dem Komma und die Mantisse die Folge der Ziffern nach dem Komma. Da

$$\log_{10}(10) = 1 \quad (2.52)$$

$$\log_{10}(100) = 2 \quad (2.53)$$

$$\log_{10}(10^n) = n \quad (2.54)$$

gilt, ist die Kennzahl eines 10er-Logarithmus leicht zu bestimmen:

$$\log_{10}(z_0, z_1 z_2 \dots z_n) = 0, l_1 l_2 \dots l_n \quad (2.55)$$

$$\log_{10}(z_0 z_1, z_2 \dots z_n) = 1, l_1 l_2 \dots l_n \quad (2.56)$$

$$\log_{10}(z_0 z_1 z_2, \dots z_n) = 2, l_1 l_2 \dots l_n \quad (2.57)$$

$$\log_{10}(z_0 z_1 z_2 \dots z_n) = n, l_1 l_2 \dots l_n \quad (2.58)$$

Im Ergebnis dieser Betrachtung ist es bei 10er-Logarithmentafeln ausreichend, lediglich den Zahlenbereich von 1,00 bis 9,99 aufzulisten. Die Logarithmen aller weiteren Zahlen können dann wie oben beschrieben bestimmt werden.

Wie wird nun der Logarithmus mithilfe einer Logarithmentafel bestimmt? Eine Zahl, für die man diesen finden möchte, heißt Numerus. Soll also der Logarithmus des Numerus bestimmt werden, schaut man in der Logarithmentafel unter der in der linken Spalte unter N gegebenen ($n - 1$)-stößigen Ziffernfolge nach. Dabei ergibt die 1. Spalte die ersten 2 Ziffern des Logarithmus. Die n -te Stelle des Numerus ist dann die von 0 bis 9 durchnummerierte Zeile. In der entsprechenden Spalte, die dem Wert der n -ten Ziffer des Numerus entspricht, findet man dann die letzten 2 Stellen des Logarithmus (Abb. 2.15).

Für das Rechnen mit dem Rechenschieber und dem Logarithmus sind vier wichtige Regeln einzuhalten:

- Der Logarithmus eines Produkts ergibt sich aus der Addition der Logarithmen der jeweiligen Faktoren.
- Der Logarithmus eines Bruches hat denselben Wert wie die Subtraktion von Zähler und Nenner.
- Der Logarithmus einer Potenz entspricht der Multiplikation des Logarithmus der Grundzahl mit dem Exponenten.

\log_{10}	3,237																		
		$\log_{10} 3000 — \log_{10} 4000$																	
N										0	1	2	3	4	5	6	7	8	9
315			83	84	86	87	89	90	91	93	94	95							
316			97	98	00	01	01	04	05	06	08	09							
317	50		11	12	13	15	16	17	19	20	22	23							
318			24	26	27	28	30	31	32	34	35	37							
319			38	39	41	42	43	45	46	47	49	50							
320			51	53	54	56	57	58	60	61	62	64							
321			65	66	68	69	70	72	73	75	76	77							
322			79	80	81	83	84	85	87	88	89	91							
323			92	93	95	96	97	99	00	01	03	04							
324	51		05	07	08	09	11	12	13	15	16	17							
325			19	20	22	23	24	26	27	28	30	31							
326			32	34	35	36	38	39	40	41	43	44							
327			45	47	48	49	51	52	53	55	56	57							
328			59	60	61	63	64	65	67	68	69	71							
329			72	73	75	76	77	79	80	81	83	84							

Abb. 2.15 Übersicht über Logarithmen

- Der Logarithmus einer n -ten Wurzel entspricht der Multiplikation des Logarithmus der Diskriminante mit dem Kehrwert von n .

Formal entspricht dies den folgenden Gesetzen:

$$\log_b(u \cdot v) = \log_b u + \log_b v \quad (2.59)$$

$$\log_b\left(\frac{u}{v}\right) = \log_b u - \log_b v \quad (2.60)$$

$$\log_b(u^v) = v \log_b u \quad (2.61)$$

$$\log_b(\sqrt[v]{u}) = \frac{1}{v} \log_b u \quad (2.62)$$

Soweit kein anderer Wert als Basis explizit angegeben ist, ist der dekadische Logarithmus, also der Logarithmus zur Basis $b = 10$, beim Rechnen mit dem Rechenschieber Konvention.

Durch Anwendung der Logarithmengesetze kann die Multiplikation zweier Zahlen auf die Addition der Nachkommastellen der logarithmierten Faktoren reduziert werden. Möchte man den Stellenwert genau ausrechnen, so bedarf es einer Überschlagsrechnung, da der Rechenschieber nicht die gesamte Größenordnung einer Zahl anzeigt. Wenn man nur die Gesamtlänge der Strecken der aneinander gereihten Nachkommastellen (Mantissen) misst, erhält man dasselbe Ergebnis wie bei der vollständigen Berücksichtigung der Logarithmen. Diese Erkenntnis wurde mit dem Rechenschieber umgesetzt. Die Mantissen sind in einer nicht linearen Skala angegeben. Bedingt durch $\log_{10}(1) = 0$ wird die Skala mit 1 beginnend indiziert.

Um zu verstehen, wie mit einem Rechenschieber multipliziert wird, sollen nur die C -Skala und die D -Skala betrachtet werden. Die unterschiedlichen Bezeichner für die beiden Skalen wurden gewählt, damit man klar die beiden Multiplikanden voneinander unterscheiden kann. Der Rechenschieber kann nun wie ein Rechenstab bei der Addition verwendet werden. Dabei wird die 1 der C -Skala nach rechts geschoben. Man bewegt also die Zunge so lange zur rechten Seite, bis die 1 der C -Skala auf dem Wert des ersten Multiplikanden auf der D -Skala steht. Schiebt man nun den Läufer ebenfalls nach rechts und zwar genauso, dass die Markierung des Läufers auf den Zahlenwert des zweiten Multiplikanden auf der C -Skala zeigt, dann kann man das Ergebnis der Multiplikation auf der D -Skala des Läufers ablesen.

► **Beispiel 2.3.14 – Multiplikation mit dem Rechenschieber**

Das Ergebnis von $2 \cdot \pi$ wird wie folgt bestimmt: Gemäß 2.59 muss mit $u = 2$ und $v = \pi$ gelten, dass $\log_{10}(2 \cdot \pi) = \log_{10} 2 + \log_{10} 3$ ist. Aus dem Produkt wird so eine Addition zweier Summanden, deren Werte jeweils mit der logarithmischen Skala des Rechenschiebers grafisch ermittelt werden können (Abb. 2.16). Dazu schiebt man den mit dem Wert 1 versehenen Anfangsstrich der Zungeneinteilung unter die Ziffer u (hier: 2) der Skala des Lineals und bewegt den Läufer nun so weit, bis der vertikale mittig angeordnete Läuferstrich über der Ziffer v (hier: π) der Zungenskala steht. Auf der festen Skala darunter kann man nun das Ergebnis durch genaues Hinsehen ablesen. Auch das Produkt von mehrstelligen Zahlen lässt sich mit einem kleinen Trick bewerkstelligen. Hierzu skaliert man vorab die entsprechenden Faktoren, sodass diese auf der Skala ablesbar sind. Beispielsweise entspricht das Produkt $250 \cdot 50$ der Multiplikation der beiden Faktoren $2,5 \cdot 10^2$ und $5,0 \cdot 10^1$, weshalb man $u = 2,5$ sowie $v = 5,0$ setzt und das abgelesene Ergebnis auf dem Rechenschieber mit $10^2 \cdot 10^1 = 10^{2+1} = 10^3$ multipliziert. ◀

Es kann allerdings vorkommen, wenn man einen Wert auf der D -Skala mithilfe der Zunge einstellen möchte und die Zunge zu diesem Zweck nach rechts verschiebt, dass man zwar den zweiten Wert mit dem Läufer auf der C -Skala einstellen kann, der Läufer nun aber das Ende der D -Skala überschritten hat und somit das Ergebnis auf der D -Skala nicht abgelesen werden kann. In einem solchen Fall vertauscht man die Seiten: Man schiebt die Zunge nach links, bis die 10 der C -Skala auf dem gewünschten Wert des ersten Multiplikanden auf der D -Skala steht, bewegt wieder den Läufer auf den zweiten Multiplikanden auf der C -Skala und liest das Ergebnis auf der D -Skala ab.

► **Beispiel 2.3.15**

In einem zweiten Beispiel soll die Aufgabe $25 \cdot 5$ gerechnet werden. Verschiebt man die Zunge mit der C -Skala, sodass die linke 1 auf der 5 der D -Skala steht, kann man das Ergebnis nicht mehr durch Verschieben des Läufers nach links auf der D -Skala ablesen (Abb. 2.17). Als Erstes muss die 25 normiert werden, und zwar so, dass man einen Wert zwischen 1 und 10 erhält. Bei der 25 ist dies die 2,5, nachdem man 25 durch 10 geteilt hat. Den Tei-

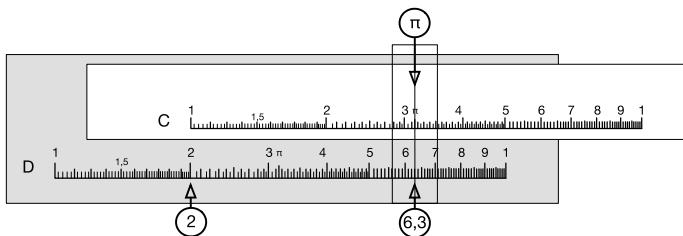


Abb. 2.16 Multiplikation $2 \cdot \pi$ mit einem Rechenschieber

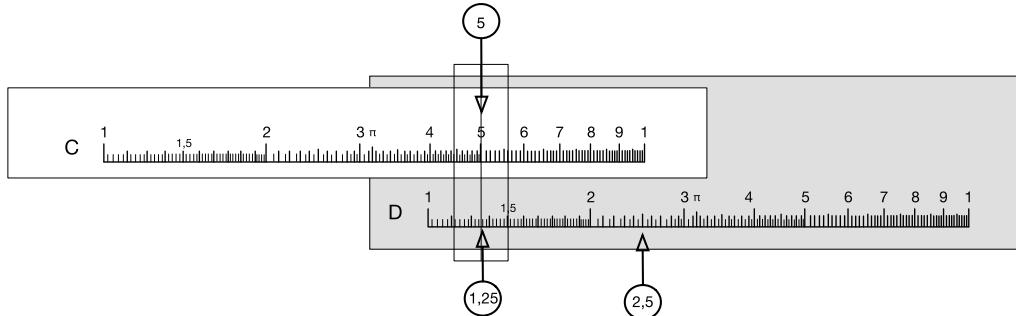


Abb. 2.17 Multiplikation $5 \cdot 25$ mit einem Rechenschieber

ler 10 muss man sich merken. Nun verschiebt man die rechte 1 der C-Skala auf die 2,5. Anschließend wird der Läufer nach links geschoben, bis die Markierung des Läufers auf der 5 der C-Skala steht. Auf der D-Skala des Läufers liest man nun 1,25 ab und erhält mit Multiplikation mit der gemerkten 10 den richtigen Wert 125. ◀

► Beispiel 2.3.16 – Berechnung der Quadratzahlen mit dem Rechenschieber

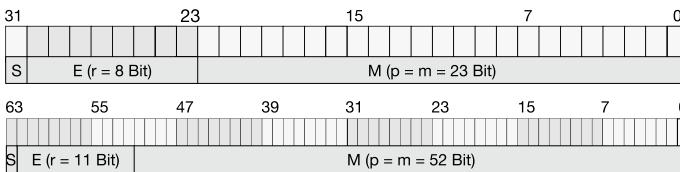
Für die Berechnung von Quadratzahlen verwendet man die Skalen A und B , die die Quadratskala zur Grundskala D auf dem Körper, respektive zur Grundskala C auf der Zunge oben abbilden. Hierzu macht man sich die Gesetzmäßigkeit der Multiplikation eines Logarithmus mit einer natürlichen Zahl $\log x^n = n \cdot \log x$ zunutze. Prinzipiell liest man anhand des Läuferstrichs auf der A -Skala die quadrierte Zahl ab, nachdem die zu potenzierende Zahl als Wert auf der D -Skala eingestellt worden ist. Soll exemplarisch 3^2 ermittelt werden, so schiebt man den Läufer so weit, bis der vertikale und mittig angeordnete Läuferstrich über der Ziffer 3 steht. Sofern dieser Schritt vollzogen wurde, kann man auf der D -Skala bereits den gewünschten Wert (hier: 9) ableSEN. Nach selbigem Prinzip ließe sich auch eine Kubikzahl, sprich eine Zahl zur 3. Potenz, darstellen, indem anstelle der A -Skala die K -Skala verwendet wird. ◀

2.3.4.3 Binäre Gleitkommazahlen

Binäre Gleitkommazahl

IEEE Std 754

In modernen Rechnern wird die binäre Darstellung einer Gleitkommazahl verwendet. Die Basis der Gleitkommazahl ist also $b = 2$. Um den Datenaustausch und die Programmportabilität zu erhöhen, sind Gleitkommazahlen seit Mitte der 1980er-Jahre standardisiert. Ein weitverbreiteter Standard ist der *IEEE Std 754–2008* (*IEEE* = Institute of Electrical and Electronics Engineers, ausgesprochen [ai tripəl iː]). In diesem Standard werden sowohl das Format der zu speichernden Zahlenfolge als auch die Codierung spezieller Zahlen definiert. Auch wurde dort geregelt, wie Zahlen nach der Berechnung gerun-



■ Abb. 2.18 Gleitkommaformat für einfache und doppelte Genauigkeit

det werden sollen, sodass es möglich wird, dass unterschiedliche Rechner zu den gleichen Ergebnissen kommen. Das binäre Format, die Repräsentation der Gleitkommazahl im Rechner, ist im IEEE Std 754 definiert zu $S\ E[r]\ M[p]$ (■ Abb. 2.18). Dabei werden das Vorzeichen S (Sign) in 1 Bit gespeichert, der Exponent E in r bit und die Mantisse M in p Bit. Im IEEE Std 754–2008 ist $p = 23$ bei einfacher Genauigkeit (single) und $p = 52$ bei doppelter Genauigkeit (single extended). Neben diesen beiden Formaten können noch 128-Bit-Zahlworte in doppelter erweiterter Genauigkeit (double extended) gespeichert werden. Der Standard erlaubt die Darstellung von Zahlen zur Basis $b = 2$ und $b = 10$, wobei in gängigen Computern $b = 2$ ist. Für das Vorzeichen wird 1 Bit benötigt: $S = 0$ für positive und $S = 1$ für negative Zahlen. Bei einer Wortgröße von 32 im Format für die einfache Genauigkeit bleiben dann 8 Bit zur Codierung des Exponenten. Bei doppelter Genauigkeit im 64-Bit-Format stehen 52 Bit für die Mantisse und 11 Bit für den Exponenten zur Verfügung. Der IEEE-Standard definiert insgesamt 4 binäre (16, 32, 64 und 128 Bit) und 3 dezimale (32, 64 und 128 Bit) Datenformate. Auf die dezimalen Formate wird im Folgenden nicht mehr eingegangen.

Oft werden Gleitkommazahlen normalisiert. Das bedeutet, der Exponent wird immer so gewählt, dass die Mantisse im Bereich $0 \leq z \leq x$ liegt. Gängige Formatierungen sind $x = 1$ oder $x = 10$.

$$x = (-1)^S \cdot M \cdot 2^E \quad (2.63)$$

Mit dieser Codierung lassen sich im 32-Bit-Format einfacher Genauigkeit $2 \cdot 10^{-38}$ bis $2 \cdot 10^{38}$ in einem Rechner darstellen. Wird der Exponent so groß, dass die Bits nicht ausreichen, um die Zahl abzubilden, erhält man einen Überlauf. Mit Gleitkommazahlen lassen sich sowohl sehr große als auch sehr kleine Zahlen darstellen. Allerdings ist die Anzahl von Stellen in einem Rechner immer begrenzt, sodass eben nicht unendlich viele Zahlen ausgedrückt werden können. Da dies auch für kleine Zahlen gilt, sind in der Gleitkommaarithmetik ein Über- und ein Unterlauf vorgesehen. Beide Ereignisse signalisieren eine Ausnahme im Rechner. Im Format der doppelten Genauigkeit

lassen sich Zahlen zwischen $2 \cdot 10^{-308}$ und $2 \cdot 10^{308}$ darstellen. Bedingt durch die Erhöhung der Speicherstellen in der Mantisse um 3 Bit ergibt sich bei doppelter Genauigkeit nicht nur ein größerer Wertebereich, sondern die Auflösung der Zahl wird ebenfalls erhöht. Im IEEE Std 754–2008 kann diese noch weiter gesteigert werden: Normalisiert man die Binärzahlen der Mantisse, hat jede darzustellende Zahl eine führende 1. Normalisieren bedeutet also, die Mantisse so zu wählen, dass die entsprechende binäre Codierung keine führende 0 besitzt. Da man nun weiß, dass die erste Ziffer der Mantisse eine 1 ist, kann man darauf verzichten, diese zu speichern. Dadurch kann man bei einfacher Genauigkeit mit einer Mantisse von 24 Bit und bei doppelter Genauigkeit mit einer Mantisse von 53 Bit rechnen.

$$x = (-1)^S \cdot (1 + 0, m) \cdot 2^e \quad (2.64)$$

$$\text{oder } x = (-1)^S \cdot 1, M \cdot 2^e \quad (2.65)$$

Eine häufige Aufgabe in einem Rechner ist es, Zahlen zu sortieren, diese also gemäß ihrer Größe zu ordnen. Negative Zahlen sollen immer eine 1 im 1. Bit stehen haben, sodass sie leicht erkannt werden können. Codiert man z. B. negative Exponenten in der Gleitkommandarstellung und möchte man aus Kostengründen eine Integer-Einheit zum Vergleichen verwenden, so scheinen Zahlen mit negativen Exponenten größer als Zahlen mit positiven Exponenten zu sein. Dieser Nachteil ist dramatisch, da die mathematischen Rechenoperationen der Gleitkommaarithmetik häufig einen Vergleich der Exponenten mit anschließender Sortierung benötigen.

► Beispiel 2.3.17

In einem Rechner sollen die Zahlen $x = -0,75$ und $y = 1,75$ sortiert werden. Zunächst wird $x = -0,75$ umgewandelt:

$$-0,75_{B10} = \frac{-3_{B10}}{4_{B10}} = \frac{-3_{B10}}{2_{B10}^{2_{B10}}} = -3 \cdot 2^{-2_{B10}}$$

oder

$$-0,75_{B10} = -11_{B2} \cdot 2^{-2_{B10}} = -0,11_{B2} \cdot 2^0$$

In normalisierter Gleitpunktdarstellung erhält man dann

$$-0,75_{B10} = -1,1_{B2} \cdot 2^{-1}$$

oder

$$-0,75_{B10} = (-1)_{B2}^1 \cdot (1,10000000000000000000000000000000_{B2}) \cdot 2^{11111111_{B2}}$$

Im IEEE-Format wird dann das Bitmuster

1 11111111 1000000000000000000000000

gespeichert. Nun wird $y = 1,75$ erstellt:

$$1,75_{B10} = \frac{7_{B10}}{4_{B10}} = \frac{7_{B10}}{2^2_{B10}} = 7 \cdot 2^2_{B10}$$

oder

$$1,75_{B10} = 111_{B2} \cdot 2^{2_{B10}} = 1,11_{B2} \cdot 2^{-2}$$

oder

Im IEEE-Format wird dann das Bitmuster

1 11111110 1100000000000000000000000000

gespeichert. Vergleicht man die beiden Exponenten 11111111 und 11111110, ergibt ein Vergleich in der kostengünstigen Integer-Arithmetik, dass die Zahl x größer als die Zahl y ist. Das ist offensichtlich falsch. ◀

Das Problem kann gelöst werden, indem man eine Verschiebekonstante (Bias) einführt, die immer implizit mitgeführt wird. Die Verschiebekonstante wird zum Exponenten addiert. Für Zahlen mit einfacher Genauigkeit führt man als Verschiebekonstante 127, für Zahlen mit doppelter Genauigkeit 1023 ein:

$$x = (-1)^S \cdot 1, M \cdot 2^{e+B} \quad (2.66)$$

$$\text{oder } x \equiv (-1)^S \cdot 1, M \cdot 2^{e+127} \quad (2.67)$$

$$\text{oder } x = (-1)^S \cdot 1, M \cdot 2^E \quad (2.68)$$

mit $E = e + B$ und $B = b^r - 1$ oder im binären IEEE Std 754-2008-Format $B = 2^{r-1}$. Die Summe aus der führenden 1 und der Mantisse $m = 1$, M wird zur besseren Unterscheidung Signifikant genannt.

► Beispiel 2.3.18

In einem Rechner sollen die Zahlen $x = -0,75$ und $y = 3,75$ sortiert werden. Mithilfe der Verschiebekonstanten ergeben sich folgende Gleichungen:

$$-0,75_{B10} = -1,1_{B2} \cdot 2^{-1_{B10} + 127_{B10}}$$

$$3,75_{B10} = 1,111_{B2} \cdot 2^{1B10+127_{B10}}$$

oder

$$3,75_{B10} = (-1)_{B2}^0 \cdot (1,11000000000000000000000000000000_{B2}) \cdot 2^{10000000B2}$$

Im IEEE-Format werden dann die Bitmuster

$$1\ 01111110\ 10000000000000000000000000000000 \text{ für } x = -0,7$$

und

$$0\ 10000000\ 11100000000000000000000000000000 \text{ für } x = 3,75$$

gespeichert. Mithilfe der Verschiebekonstanten (Bias) werden die Exponenten so dargestellt, dass mit einer Ganzzahlarithmetik ein Vergleich auf den Exponenten vorgenommen werden kann und kleine Zahlen mit negativem Exponenten auch als kleine Zahlen erkannt werden. ◀

Um die Implementierung der Arithmetik von Gleitkommazahlen in einer Rechenmaschine zu erleichtern, definiert man noch einige ausgezeichnete besondere Zahlen. Oft ergeben Berechnungen keine numerische Lösung, beispielsweise wenn durch 0 geteilt wird oder die Wurzel aus einer negativen Zahl gezogen wird. Damit die Hardware derartige Ergebnisse dem Programmierer signalisieren kann, wurde das Format „not a number“ (NaN, keine Zahl) eingeführt. Man unterscheidet dabei noch zwischen einer anzeigenden und einer stillen Darstellung („quiet not a number“). Die anzeigende Signalisierung dient der Kommunikation zwischen Hard- und Software, die stille wird intern im Rechenwerk verwendet. NaN wird angezeigt, indem alle Bits des Exponenten E auf 1 gesetzt werden. Ist dazu noch Bit 23 der Mantisse M gesetzt, handelt es sich um ein anzeigendes Signal, ist Bit 23 0, handelt es sich um ein stilles. Alle anderen Bits von M können beliebig gesetzt werden, dürfen aber in ihrer Summe nicht null sein.

$$NaN = X\ 11111111\ 1XX\ XXXX\ XXXX\ XXXX\ XXXX\ XXXX \quad (2.69)$$

$$= X\ 11111111\ 0XX\ XXXX\ XXXX\ XXXX\ XXXX\ XXXX \quad (2.70)$$

Unendlich

Durch dieses Vorgehen ist bei einfacher Genauigkeit die höchste anzeigbare Zahl um eine Zweierpotenz reduziert. Sind alle Bits von E auf 1 gesetzt und ist die Mantisse $M = 0$, definiert die Gleitkommazahl den Wert unendlich. Mit dem Vorzeichen lassen sich dann sogar $-\infty$ und $+\infty$ darstellen:

$$-\infty = 11111111\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad (2.71)$$

$$+\infty = 0\ 11111111\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad (2.72)$$

Null

Neben unendlich sollte sich auch die Null eindeutig darstellen lassen. Bei der Null wird das Vorzeichenbit ignoriert, und alle anderen Stellen des Formats sind auf 0 gesetzt:

$$0 = X\ 00000000\ 000\ 0000\ 0000\ 0000\ 0000\ 0000 \quad (2.73)$$

Mithilfe dieser Konventionen können dann Zahlen in einem festgelegten Zahlenbereich dargestellt werden. Die größte dar-

stellbare Zahl in einfacher Genauigkeit wäre dann:

0 11111110 1111 1111 1111 1111 11111111

$$\text{also } 3,3554431 \cdot 2^{254-127} = 3,3554431 \cdot 2^{127} = 5,708991 \cdot 10^{38}$$

Die kleinste positiv darstellbare Zahl, die nicht null ist, ergibt sich aus dem Bitmuster:

0 00000001 000 0000 0000 0000 0000 0000

Bedingt durch die Normalisierung entspricht dies:

$$\begin{aligned} 0 00000001 1000 0000 0000 0000 0000 &= 1,6777216 \cdot 2^{1-127} \\ &= 1,6777216 \cdot 2^{-126} \\ &= 1,427 \cdot 10^{-38} \end{aligned}$$

Würde man auf die Normalisierung verzichten, wäre die kleinste von null verschiedene Zahl:

$$\begin{aligned} 0 00000000 0000 0000 0000 0000 0000 0001 &= \\ 1,6777216 \cdot 2^{-150} &= 2,3945 \cdot 10^{-45} \end{aligned}$$

Aus diesem Grund ist es möglich, der Hardware zu signalisieren, dass eine denormalisierte Zahl betrachtet wird, die führende 1 also als 0 zu interpretieren ist. Da sowohl für die Mantisse M als auch für den Exponenten E zur Darstellung der Null $M = 0$ und $E = 0$ sind, kann die Normalisierung im Fall von $M \neq 0$ und $E = 0$ abgeschaltet werden. Zahlen, bei denen im Gleitkommaformat der Exponent $E = 0$ und deren Mantisse ungleich null sind, heißen denormalisierte Zahlen. Das Gleitkommaformat entspricht dann:

X 00000000 XXX XXXX XXXX XXXX XXXX XXXX

Das Rechnen mit Gleitkommazahlen wird Gleitkommaarithmetik genannt. Gegenüber dem Arbeiten mit den ganzen Zahlen weist die Gleitkommaarithmetik einige Besonderheiten auf. Die Mantisse und der Exponent werden getrennt betrachtet. Das bedeutet, dass bei einer Berechnung die Exponenten der zwei Operanden gleich sein müssen. Nur so sind die einzelnen Ziffern der beiden Mantissen vergleichbar und können wie ganze Zahlen behandelt werden.

Um aber in Gleitkommaarithmetik möglichst exakt rechnen zu können, definiert der IEEE-Standard 754 zusätzliche Bits. Diese heißen Guard-Bit, Round-Bit und Sticky-Bit und werden dem IEEE-Datenrahmen hinzugefügt. Im Guard- und im Round-Bit werden Ziffern gespeichert, die, wenn man ausschließlich auf den genormten Datenformaten rechnen wür-

de, verloren gingen, und das Sticky-Bit zeigt an, dass es rechts vom Round-Bit noch weitere Stellen gibt, die von 0 verschoben sind.

Addition und Subtraktion erfolgen mithilfe der gleichen Algorithmen unter Zuhilfenahme der jeweils ganzzahligen Operation. Bei der Gleitkommaaddition müssen in einem ersten Schritt die beiden Exponenten der Operanden verglichen werden. Sind diese ungleich, müssen das Komma der Mantisse der Zahl mit dem kleineren Exponenten so lange nach links verschoben werden und der Exponent so lange erhöht werden, bis die Exponenten beider Zahlen gleich sind. Das Linksschieben des Kommas ergibt sich, da die Mantisse selbst nach rechts verschoben wird.

► Beispiel 2.3.19 – Normieren des Exponenten

Gegeben sind die Operanden $x_{B10} = 0,5_{B10}$ und $y_{B10} = -0,4375_{B10}$. In Gleitkommadarstellung werden diese zu $x_G = 1_{B2} \cdot 2^{-1_{B10}}$ und $y_G = -0,110_{B2} \cdot 2^{-2_{B10}}$. Verschiebt man die Mantisse von y_G um eine Stelle nach rechts, erhält man $y'_G = -0,110_{B2} \cdot 2^{-1_{B10}}$. Dabei kann die ganz rechte 0 im Guard-Bit der Arithmetikeinheit gespeichert werden. ◀

Nach der Normierung der Exponenten lassen sich die beiden Signifikanten addieren. Die Mantisse des Ergebnisses muss anschließend wieder normiert werden. Dies geschieht durch Verschieben dieser nach rechts oder nach links bei gleichzeitigem Verringern und Erhöhen des jeweiligen Exponenten – und zwar so lange, bis die entstandene Gleitkommazahl vor dem Komma 1-stellig ist. Dabei muss in jedem Schritt überprüft werden, ob der resultierende Exponent größer oder kleiner als der maximale oder der minimale im Rechner darstellbare Exponent ist. Ist dies der Fall, wird ein Über- oder Unterlauf signalisiert.

► Beispiel 2.3.20 – Addition

Die beiden Signifikanten werden addiert: $z_G = 0,1_{B2} + (-0,110_{B2}) = 0,001$. Damit ist das Ergebnis in Gleitkommadarstellung zunächst $z_G = 0,001_{B2} 2^{-1_{B10}}$ oder renormiert $z_G = 1,000_{B2} 2^{-4_{B10}}$, da das Komma um 3 Stellen nach rechts und der Signifikant 3 Stellen nach links verschoben werden. ◀

Die Multiplikation zweier Gleitkommazahlen besteht aus der Multiplikation der beiden Signifikanten und der Addition der beiden Exponenten. Bedingt durch die Darstellung des Exponenten unter Annahme einer Verschiebekonstanten ist es erforderlich, nach dem Bilden der Summe noch die jeweilige Verschiebekonstante abzuziehen, um das korrekte Ergebnis zu erhalten. Die Multiplikation der Signifikanten und die Addi-

tion der Exponenten einschließlich der Subtraktion der Verschiebekonstante wird in Ganzzahl-Arithmetik realisiert. Allerdings ist bei der Exponenten-Operation immer zu prüfen, ob ein Über- oder Unterlauf stattfindet. Nach der Berechnung muss die entstandene Gleitkommazahl normiert werden. Auch nach dieser Operation ist zu prüfen, ob ein Über- oder ein Unterlauf stattfindet.

► Beispiel 2.3.21 – Multiplikation von binären Gleitkommazahlen

Gegeben sind die beiden Operanden $x = -0,75$ und $y = 3,75$.

$$-0,75_{B10} = (-1)_{B2}^1 \cdot (1,10000000000000000000000000000000_{B2}) \cdot 2^{01111110_{B2}}$$

$$3,75_{B10} = (-1)_{B2}^0 \cdot (1,11100000000000000000000000000000_{B2}) \cdot 2^{10000000_{B2}}$$

Die Multiplikation von 1,1000 mit 1,1110 ergibt

$$\begin{array}{r} 1 , 1 0 0 0 \cdot 1 , 1 1 0 \\ \hline 0 0 0 0 \quad 0 0 0 0 \\ 0 0 0 0 \quad 0 0 0 0 \\ 0 0 0 0 \quad 0 0 0 0 \\ 1 1 1 0 0 0 \\ \hline 1 , 1 1 0 0 0 \\ \hline 0 1 1 0 , 1 0 1 \end{array} \quad 21$$

Durch Addition der Exponenten 01111110 und 10000000 erhält man

$$\begin{array}{r} x \quad 0111\ 1110 = 126 \\ y \quad 1000\ 0000 = 128 \\ \hline c \quad 0\ 0000\ 0000 \\ \hline 0\ 1111\ 1110 = 254 \\ 1000\ 0001 = -127 \text{ Abziehen der Verschiebekonstante, da} \\ c \quad 1\ 0000\ 0000 \text{ diese im Ergebnis zweimal vorhanden ist.} \\ \hline 1\ 0111\ 1111 = 127 \end{array}$$



Das Ergebnis 127_{B10} ist binär noch mit 8 Stellen darstellbar. Somit tritt kein Überlauf auf, und der Exponent ist $127_{B10} = 01111111_{B2}$. Das Ergebnis ist also für den Signifikanten 010, 101_{B2} und für den Exponenten $e = 01111111_{B2} = 124$. Das Vorzeichen wird negativ, und es ergibt sich die folgende Darstellung:

$$(-1)_{B2}^0 \cdot (010, 101_{B2}) \cdot 2^{01111111_{B2}} \quad (2.74)$$

Dieses Ergebnis muss noch normiert werden,

$$(-1)_{B2}^0 \cdot (1, 0101_{B2}) \cdot 2^{10000000_{B2}} \quad (2.75)$$

und man erhält 1 10000000 01010000000000000000000000000000. Die Probe ergibt

$$2 \quad 0,75 = \frac{3}{4} \text{ und } 3,75 = \frac{15}{4} \quad (2.76)$$

$$\frac{3}{4} \cdot \frac{15}{4} = \frac{45}{16} = \frac{45}{2^4} = 45 \cdot 2^{-4} \quad (2.77)$$

Binär ist die $45_{B10} = 101101$ oder $45_{B10} = 1,01101_{B2} \cdot 2^5$, da $1,01101_{B2} \cdot 2^5 = 101101,0_{B2}$. Damit entspricht dies:

$$\frac{3}{4} \cdot \frac{15}{4} = \frac{45}{16} = \frac{45}{2^4} = 1,01101_{B2} \cdot 2^5 \cdot 2^{-4} = 1,01101_{B2} \cdot 2^1 \quad (2.78)$$

Das lässt sich umrechnen,

$$\begin{aligned} &= 1,01101000000000000000000000000000_{B2} \cdot 2^{1_{B10}} \\ &= 1,01101000000000000000000000000000_{B2} \cdot 2^{1+127_{B10}} \\ &= 1,01101000000000000000000000000000_{B2} \cdot 2^{128_{B10}} \\ &= 1,01101000000000000000000000000000_{B2} \cdot 2^{10000000_{B2}} \end{aligned}$$

und dann im Format einer Gleitkommazahl darstellen:

$$-0,75_{b10} \cdot 3,75_{b10} := 1\ 1000000\ 01101000000000000000000000000000 \quad (2.79)$$

2.3.4.4 Binäre Festkommazahlen

Die Näherung einer reellen Zahl in Gleitkommadarstellung erfordert die Speicherung der Mantisse und des Exponenten, da jede reelle Maschinenzahl einen anderen Exponenten aufweist. Dadurch muss bei Gleitkommarechenwerken die Arithmetik der Mantisse und des Exponenten berücksichtigt werden. Dies führt zu aufwendigen Implementierungen des Rechenwerks. Nicht in jedem Fall ist ein derartig hoher Aufwand gerechtfertigt. Komplexe Gleitkommaeinheiten kosten Geld, Energie und Rechenzeit. Insbesondere in kostengünstigen Systemen werden daher oft Rechenwerke für ganze Zahlen implementiert. Ein berühmtes Beispiel ist der Navigationsrechner des Apollo-Raumschiffs. Da es beim Flug zum Erdtrabanten auf jedes Gramm ankommt und Computer zu dieser Zeit große Maschinen waren, begnügte man sich beim Bau der Hardware mit einem 15-Bit-Rechenwerk für ganze Zahlen. Die Navigation zwischen Erde und Mond macht es aber erforderlich, mit reellen Zahlen zu rechnen. Der Apollo-Navigationsrechner arbeitete daher mit Festkommazahlen und es war nicht notwen-

dig, einen Exponenten zu speichern und mit diesen zu nutzen. Auch heute verwenden viele kostengünstige eingebettete Systeme Festkommazahlen.

Festkommaformate bilden positive Festkommazahlen in Binär- und beliebige Festkommazahlen in 2-Komplement-Darstellung ab. Die Formate für natürliche (natural Binary, unsigned Integer) und ganze Zahlen (signed Integer) sind streng genommen Sonderfälle der jeweiligen Festkommazahlen.

Eine Festkommazahl wird in einem festen Bitformat gespeichert. Die Formate $UQ_{m,n}$ oder $Q_{m,n}$ charakterisieren eine solche Struktur eindeutig. Da Festkommazahlen ausschließlich rationale Zahlen darstellen können, entspricht Q dem mathematischen Symbol der rationalen Zahlenmenge. Der Bezeichner UQ kennzeichnet eine vorzeichenlose Festkommazahl und das Zeichen Q eine vorzeichenbehaftete Nummer in 2-Komplement-Darstellung. Die Anzahl der Stellen einer Festkommazahl ist dann $N = a + b$. Häufig findet man auch Formate der Art Q_n . In diesem Fall wird von 32- oder 16-Bit-Wörtern zum Speichern ausgegangen und nur die Länge des jeweiligen Bruchteils spezifiziert.

Man erinnere sich an die Definition der reellen Zahlenmenge und vergegenwärtige sich, dass eine Festkommazahl nur eine rationale Zahl darstellen kann. Die algebraischen und die transzendenten Zahlen müssen daher im Rechner approximiert werden. Die kann mit einer Kettenbruchdarstellung erfolgen.

Definition 2.3.10 – Festkommazahl $UQ(m, n)$

Eine positive Festkommazahl mit m Stellen für den ganzzahligen Anteil und n Stellen für den gebrochenen Teil wird wie folgt dargestellt: $b_m b_{m-1} \dots b_0, b_{-1} b_{-2} \dots b_n$. Der Wert ergibt sich aus

$$x = \frac{1}{2^n} \sum_{i=0}^{N-1} d_i \cdot 2^i \quad (2.80)$$

In dieser Darstellung stellt $UQ_{m,n}$ einen Teilbereich der realen Zahlen dar. Dieser Teilbereich ist $P := \{p \in \mathbb{Z}, p/2^n | 0 \leq p \leq 2^{m+n-1}\}$ oder $P := \{p \in \mathbb{Z}, p/2^n | 0 \leq p \leq 2^{N-1}\}$.

Eine solche Darstellung („unsigned fixed point rational“) kann den Zahlenbereich $2^N - \frac{1}{2^n} = 2^m - 2^{-n}$ darstellen.

► Beispiel 2.3.22

Die Wurzel von 2 ($x = \sqrt{2} = 1,41421356237\dots$) kann im Format $UQ(5, 2)$ mit folgendem Dezimalbruch angenähert werden:

$$x_{UQ(5,2)} = 00001,41$$



Im Festkommaformat sind die natürlichen Zahlen als Sonderfall enthalten. So spezifizieren die Formate $UQ_{15,0}$, $UQ_{32,0}$ oder UQ_0 jeweils natürliche Zahlen im vorzeichenlosen Format (natural Binary, unsigned Integer).

Festkommazahlen können wie die natürlichen Zahlen auch mit einem Vorzeichen versehen werden. Die einfachste Möglichkeit ist, einfach ein Bit zum Speichern des Vorzeichens zu reservieren. Wie bei den ganzen Zahlen ist es am effizientesten, das 2-Komplement-Format zu nutzen und wieder anzugeben, wo das Komma stehen soll. Dieses Vorgehen führt zu einer vorzeichenbehafteten Festkommazahl („signed two complement fixed point rational“).

Definition 2.3.11 – Vorzeichenerweiterte Festkommazahl im Format $Q(m, n)$

Ein N bit langes Speicherwort, mit m ganzzahligen und n gebrochenen Bits kann als Bruch in 2-Komplement-Darstellung interpretiert werden:

$$x = \pm \left(\frac{1}{2^n} \right) \cdot \left(-2^{N-1} d_{N-1} + \sum_{i=0}^{N-2} d_i \cdot 2^i \right) \quad (2.81)$$

In dieser Darstellung stellt $Q_{m,n}$ einen Teilbereich der realen Zahlen dar. Dieser Teilbereich ist $P := \{p \in \mathbb{Z}, -p/2^{n+m} | 0 \leq p \leq 2^{m+n-1}\}$ oder $P := \{p \in \mathbb{Z}, -p/2^N | 0 \leq p \leq 2^{N-1}\}$.

Da eine Festkommazahl nur eine Näherung einer reellen Zahl ist, ist es erforderlich, die Abweichungen des Formats zu quantifizieren. Zum einen hat eine solche Darstellung eine gewisse Auflösung, die beschreibt, in welchen Zwischenschritten sie dargestellt werden kann. Die Differenz aus der größten und der kleinsten darstellbaren Nummer heißt Darstellungsbereich, und die Abweichung der dargestellten von der realen Zahl ist die Genauigkeit. Daneben beschreibt das Verhältnis der Vorfür den Nachkommastellen den Dynamikbereich des Festkommamformats.

Definition 2.3.12 – Auflösung

Die kleinste darstellbare Zahl größer als null eines Festkommaformats Q heißt Auflösung („resolution“) $R(Q)$:

$$R(Q) = \frac{1}{2^n}$$

Definition 2.3.13 – Darstellungsbereich

Der Zahlenbereich („range“), der von einem Festkommaformat dargestellt werden kann, ist definiert durch:

$$Q_R = q_{max+} - q_{max-}$$

Definition 2.3.14 – Genauigkeit

Die maximale Differenz zwischen dem reellen Zahlenwert x und seiner Repräsentation im Festkommaformat $Q_{m,n}$ wird Genauigkeit („accuracy“) genannt:

$$A(Q) = \frac{R(Q)}{2} \quad (2.82)$$

Definition 2.3.15 – Dynamikbereich

Die Dynamik eines Festkommaformats ist das Verhältnis der Anzahl der Vor- zu den Nachkommastellen. Für positive Festkommazahlen ist dies

$$D(UQ) = \frac{2^m}{2^{-n}} = 2^{m+n} = 2^{N-1} \quad (2.83)$$

und für vorzeichenbehaftete

$$D(Q) = \frac{2^m - 2^n}{2^{-n}} = 2^{m+n} - 1 = 2^N - 1 \quad (2.84)$$

► Beispiel 2.3.23

Das Format Q(13,2) deckt $N = m + n = 13 + 2$ Bit im Zahlenbereich $-2^13 = 8192$ bis $2^13 - \frac{1}{4} = 8191,75$ ab. Die Auflösung ist 0,25 und die Genauigkeit ist $\frac{1}{8}$. ◀

Beim Rechnen mit reellen Zahlen muss zusätzlich die Stellung des Kommas berücksichtigt werden. Ansonsten kann die Arithmetik der ganzen Zahlen übernommen werden. Dies gilt aber nur bei idealen mathematischen Zahlen mit unendlich vielen Stellen.

Addition und Subtraktion können wie bei den natürlichen oder den ganzen Zahlen durchgeführt werden. Dabei ist lediglich darauf zu achten, dass bei dem Minuenden und dem Subtrahenden das Komma jeweils an der gleichen Stelle steht. Die Zahlen werden also so geschrieben, dass die jeweiligen Trennzeichen untereinander liegen. Das Komma lässt sich mithilfe der Exponentialschreibweise beliebig verschieben.

Definition 2.3.16 – Festkommaaddition

Seien x eine Festkommazahl im Format $Q_x(a, b)$ und y eine Zahl im Format $Q_y(c, d)$, dann ist unter der Voraussetzung $a = c$ und $b = d$ die Addition definiert durch:

$$e = x + y \quad (2.85)$$

Da bei der Addition ein Übertrag entstehen kann, muss das Ergebnis im Format $Q_e(a+1, b)$ dargestellt werden.

Definition 2.3.17 – Festkommasubtraktion

Seien x eine Festkommazahl im Format $Q_x(a, b)$ und y eine Zahl im Format $Q_y(c, d)$, dann ist unter der Voraussetzung $a = c$ und $b = d$ die Subtraktion definiert durch:

$$e = x - y \quad (2.86)$$

Da die Subtraktion durch eine Addition im 2-Komplement berechnet wird, kann wieder ein Übertrag entstehen, und daher muss das Ergebnis im Format $Q_e(a+1, b)$ dargestellt werden.

Definition 2.3.18 – Festkommamultiplikation

Seien x eine Festkommazahl im Format $Q_x(a, b)$ und y eine Zahl im Format $Q_y(c, d)$, dann ist unter der Voraussetzung $a = c$ und $b = d$ die Multiplikation definiert durch:

$$e = x \cdot y \quad (2.87)$$

Je nachdem, ob die Zahlen vorzeichenbehaftet sind, ist das Ergebnis im Format $UQ_e(a + c, b + d)$ oder $Q_e(a + c + 1, b + d)$ darzustellen.

Definition 2.3.19 – Festkommadivision

Seien x eine Festkommazahl im Format $Q_x(a, b)$ und y eine Zahl im Format $Q_y(c, d)$, dann ist unter der Voraussetzung $a = c$ und $b = d$ die Division definiert durch:

$$e = \frac{x}{y} \quad (2.88)$$

Je nachdem, ob die Zahlen vorzeichenbehaftet sind, muss das Ergebnis im Format $UQ_e(a + c, \lceil \log_2 2^{c+a} - 2^{a+c} \rceil)$ oder $Q_e(a + c + 1, b + d)$ dargestellt werden.

► Beispiel 2.3.24

Gegeben seien die beiden Dezimalzahlen $x_{b10} = 124,54$ und $y_{b10} = 23,46$. Die beiden Zahlen werden wie folgt addiert:

$$\begin{array}{r} x_{b10} \ 124,54 \\ y_{b10} \ 023,46 \\ \hline c \ \ \ \ 00110 \\ \hline s_{b10} \ 148,00 \end{array}$$



2.4 Übertragung und alphanumerische Zeichendarstellung

Historisch begann die Entwicklung des Computers mit der Idee des automatisierten Rechnens. Daher behandelten die letzten Abschnitte hauptsächlich die Codierung von Zahlen. Günstige elektronische Komponenten und der Wunsch, Daten auch automatisch und nicht nur manuell zu bearbeiten, führten zur elektronischen Datenverarbeitung. Damit ein Computer nicht ausschließlich Zahlen verarbeiten kann, müssen für den Rech-

ner technisch beliebige Symbole codiert werden. Aufgrund der Einfachheit der binären Arithmetik setzte sich das Binärsystem durch, und alle anderen alphanumerischen Zeichen werden aus diesem Grund ebenfalls binär dargestellt. Die Symbole werden mithilfe von Tabellen eindeutig festgelegt. Jede Spalte steht darin für ein Zeichen und die jeweilige dazugehörige Codierung. Die Zeilen listen dann die unterschiedlichen Darstellungen.

Die Zeichencodierung stellt die Schnittstelle zwischen dem Benutzer, den Ein- und Ausgabegeräten und der eigentlichen Maschine dar. Peripheriegeräte verarbeiten die in verschiedenen Tabellen oder Standards codierten Informationen. Um in der Maschine Berechnungen mit Zahlen durchzuführen, müssen die bei der Eingabe erzeugten Zeichenfolgen in binäre Ziffernfolgen umgewandelt werden. Dies geschieht automatisch mit Decodierschaltungen (Decoder) oder für die Ausgabe mit Codiererschaltungen (Coder). Die in Computern verwendeten Zeichencodes gehen direkt auf die Anfänge der maschinellen Umwandlung von Zeichen zurück. Aus dem für die Telegrafie optimierten Morse-Code entwickelten sich zunächst mit der Einführung von Fernschreibern der Baudot- und der Murray-Code. Da in der Anfangszeit des Computers diese Geräte als Ein-/Ausgabegeräte genutzt wurden, setzten sich diese Codes auch in der elektronischen Datenverarbeitung durch.

1-aus-n-Code

Beim 1-aus-n-Code wird immer nur 1 Bit des Codewortes auf 1 gesetzt. Diese Codierung wird auch One-hot-Codierung genannt. Zum Speichern eines 1-aus-n-Codes wird sehr viel Speicherplatz benötigt. Allerdings hat der Code einige Vorteile. Bei der Übertragung von Daten können mit dem 1-aus-n-Code leicht Fehler erkannt werden, da eine korrekte Übertragung nur dann vorliegt, wenn ein einziges Bit gesetzt ist. Diese Annahme ist möglich, da es sehr unwahrscheinlich ist, dass ein Fehler bei der Übertragung dazu führt, dass ein gewollt gesetztes Bit gelöscht, dafür aber ein anderes Bit gesetzt wird. Implementiert man die jeweiligen Zustände einer Zustandsmaschine mit einem 1-aus-n-Code, wird zwar ein großer Zustandsspeicher benötigt, die Übergangslogik ist aber effizient und schnell. Die □ Tab. 2.4 zeigt ein Beispiel für einen 16-Bit-1-aus-n-Code.

EBCDIC

Der EBCDIC („extended binary coded decimal interchange code“) ist ein Zeichenformat, das aus mehreren unterschiedlichen Zeichenformatfamilien besteht. So kam der EBCDIC auf diversen Großrechensystemen von IBM und Fujitsu zum Einsatz. EBCDIC ist eine 8-Bit-Erweiterung des 4-Bit-BCD-Codes und erlaubt die Codierung alphanumerischer Zeichen. Der Code wurde 1963 mit dem IBM System/360 eingeführt.

Der 8-4-2-1-Code ist nicht der einzige bekannte BCD-Code. Gebräuchlich sind u. a. der 2-4-2-1-Code (Aiken-Code) oder der Gray-Code (2.4).

Hex-Wert	Gray-Code	Aiken-Code	Binärzahl	1-aus-n-Code
$0_{(16)}$	$0000_{(Gray)}$	$0000_{(Aiken)}$	$0000_{(2)}$	0000000000000001
$1_{(16)}$	$0001_{(Gray)}$	$0001_{(Aiken)}$	$0001_{(2)}$	0000000000000010
$2_{(16)}$	$0011_{(Gray)}$	$0010_{(Aiken)}$	$0010_{(2)}$	00000000000000100
$3_{(16)}$	$0010_{(Gray)}$	$0011_{(Aiken)}$	$0011_{(2)}$	00000000000001000
$4_{(16)}$	$0110_{(Gray)}$	$0100_{(Aiken)}$	$0100_{(2)}$	00000000000010000
$5_{(16)}$	$0111_{(Gray)}$	$1011_{(Aiken)}$	$0101_{(2)}$	0000000000100000
$6_{(16)}$	$0101_{(Gray)}$	$1100_{(Aiken)}$	$0110_{(2)}$	0000000001000000
$7_{(16)}$	$0101_{(Gray)}$	$1101_{(Aiken)}$	$0111_{(2)}$	0000000010000000
$8_{(16)}$	$1100_{(Gray)}$	$1110_{(Aiken)}$	$1000_{(2)}$	0000000100000000
$9_{(16)}$	$1101_{(Gray)}$	$1111_{(Aiken)}$	$1001_{(2)}$	0000001000000000
$A_{(16)}$	$1111_{(Gray)}$	nicht definiert	$1010_{(2)}$	0000010000000000
$B_{(16)}$	$1110_{(Gray)}$	nicht definiert	$1011_{(2)}$	0000100000000000
$C_{(16)}$	$1010_{(Gray)}$	nicht definiert	$1100_{(2)}$	0001000000000000
$D_{(16)}$	$1011_{(Gray)}$	nicht definiert	$1101_{(2)}$	0010000000000000
$E_{(16)}$	$1001_{(Gray)}$	nicht definiert	$1110_{(2)}$	0100000000000000
$F_{(16)}$	$1000_{(Gray)}$	nicht definiert	$1111_{(2)}$	1000000000000000

Der Gray-Code zeichnet sich dadurch aus, dass sich von Codewort zu Codewort immer nur 1 Bit verändert. Daher ist er besonders zur analogen Übertragung von Daten geeignet. Aufgrund seiner Eigenschaften ist er gut geeignet, digitale Signale auf parallelen Leitungen fehlerfrei zu übertragen. Bedingt durch seine Konstruktion gleicht der Code automatisch verschiedene Laufzeiten auf unterschiedlichen Übertragungswegen aus. Somit werden Streuungen der physikalischen Parameter von Leitungen und Bauteilen ausgeglichen.

Gray-Code

ASCII-Code**2**

Zur etwa selben Zeit, zu der der EBCDIC entstand, entwickelte die American Standards Association (ASA) einen Standard zur Codierung von Zeichen für Fernschreiber. Im Jahr 1963 veröffentlichte die ASA den ASCII-Code (American Standard Code for Information Interchange). Ein ASCII-Rahmen umfasst im Gegensatz zum EBCDIC nur 7 bit. ASCII codiert 128 Zeichen, 95 druckbare und 33 nicht druckbare Sonderzeichen zur Systemsteuerung. Die Steuerzeichen enthalten Kommandos für den Zeilenvorschub, Protokoll und Bestätigungszeichen, Satz- und Datensatztrennzeichen. Die druckbaren Zeichen entsprechen dem Zeichensatz einer Schreibmaschine. Er umfasst die Symbole des lateinischen Alphabets sowohl in Groß- und Kleinschreibung als auch die zehn arabischen Ziffern und die Satzzeichen. Das 8. Bit des ASCII wurde oft zur Paritätsprüfung zur Fehlersicherung bei der Datenübertragung eingesetzt. Länderspezifische 8-Bit-Erweiterungen von ASCII codieren Umlaute oder Sonderzeichen anderer Sprachen (Beispiel: DIN 66303).

Die □ Tab. 2.5 zeigt die Codierung des ASCII-Formats in der typischen Form. Die binäre Codeinformation ist hexadezimal dargestellt. Dabei codiert jede Zeile durchgehend die vier niedrigstwertigen, während jede Spalte fortlaufend die 4 höchstwertigen Bits beschreibt, dies aber ohne Verwendung des höchstwertigen Bits. Die Spaltencodierung von ASCII endet daher mit 7.

Unicode

Mit der zunehmenden Computerisierung und der daraus entstandenen Verschiebung der Textverarbeitung von der Schreibmaschine zum Computer entstand der Bedarf, mehrere unterschiedliche Sprachen in einem Dokument darstellen zu können. Da es umständlich schien, in diesem zwischen verschiedenen Zeichensätzen umzuschalten, wurde auf der Grundlage eines 16-Bit-Binärformats ein universeller Code entwickelt (Unicode). Ziel war es, alle heute aktiv benutzten Zeichen unterschiedlicher Kulturen in einem Zeichensatz darzustellen. Der Entwurf für eine solche universelle Codetabelle entstand 1988 in den Laboren des Kopiermaschinenherstellers Xerox. Im Jahr 1991 wurde die erste Version des Unicode-Standards veröffentlicht. Diese enthielt zunächst nur die europäischen, nahöstlichen und indischen Schriftzeichen. Inzwischen beinhaltet Unicode neben den unterschiedlichen fernöstlichen Schriften selbst die altägyptischen Hieroglyphen, die persische Keilschrift, Runenschriften und die Gebärdenschrift. Damit ist es möglich, fast alle jemals erfundenen Zeichen der menschlichen Sprache in einem Dokument oder auf einer Webseite darzustellen. Unicode erreicht das Ziel, die unterschiedlichen Schriftsysteme zu codieren, durch die Einführung von sechs verschiedenen Ebenen. In der mehrsprachigen Basisebene werden die Zeichen der heute verwendeten Sprachen definiert.

Eine ergänzende Ebene dient der Codierung historischer Schriften und spezieller Zeichensätze (z. B. von grafischen Ideogrammen [Emoji]). In der idiografischen Stufe werden seltene, nicht mehr im Gebrauch befindliche fernöstliche Schriftsysteme codiert. Hinzu kommt eine letzte definierte Unterteilung für spezielle Verwendungen und zur Beschreibung von Kontrollzeichen zur Sprachmarkierung. Die letzten beiden Ebenen sind für die private Nutzung reserviert.

?

Übungsaufgaben

Aufgabe 2.1 Welche Möglichkeiten zur binären Darstellung einer reellen Zahl kennen Sie? Erörtern Sie kurz die jeweiligen Vor- und Nachteile.

■ Tab. 2.5 Codierung des ASCII-Formats

Co-de	0 ...	1 ...	2 ...	3 ...	4 ...	5 ...	6 ...	7 ...
... 0	NUL	DLE	SP	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
... A	LF	SUB	*	:	J	Z	j	z
... B	VT	ESC	+	;	K	[k	{
... C	FF	FS	,	<	L	\	l	
... D	CR	GS	-	=	M]	m	}
... E	SO	RS	.	>	N	^	n	~
... F	SI	US	/	?	O	–	o	DEL

Aufgabe 2.2 Die Entscheidung, ob die Stelle mit dem geringsten Wert links oder rechts steht, ist prinzipiell willkürlich. Daher muss für ein Zahlensystem stets auch festgelegt werden in welcher Reihenfolge die Ziffern gelesen werden müssen. Hierfür wird häufig das least significant Bit (LSB) oder das most significant Bit (MSB) angegeben. Das LSB beschreibt die Stelle, die den geringsten Beitrag zum Wert der Zahl liefert. Analog dazu beschreibt das MSB die Stelle, die den größten Beitrag zum Wert der Zahl liefert. Die Werte der restlichen Stellen ergeben sich dann aus der Reihenfolge.

Bei archäologischen Ausgrabungen wurden Steintafeln einer Hochkultur mit den Symbolen

- I) \$ ~ *#
- II) # ~~ \$

gefunden. Archäologen konnten entziffern, dass es sich bei den Symbolen um ein Zahlensystem handeln muss. Dieses Zahlensystem enthält (nur) die vier Symbole *, #, ~ und \$, die bereits in ihrer Wertigkeit aufsteigend sortiert sind. Es ist bekannt, dass diese Hochkultur eine Null kannte.

- a) Welchen Dezimalwert haben die Zahlen I und II, wenn das LSB rechts steht?
- b) Wie lautet der Dezimalwert für die Zahlen I und II, wenn die niedrigste Stelle links steht?

Aufgabe 2.3 Neben unterschiedlichen Festlegungen für die Stellenwertigkeit einer Zahl gibt es auch für das Ablegen der Bytes im Speicher verschiedene Möglichkeiten. Je nach Rechnerarchitektur ist dies nicht immer einheitlich festgelegt.

In der Praxis haben sich zwei Bytereihenfolgen etabliert. Bei *big-endian* ist das niedrigstwertige Byte, wie gewohnt, ganz rechts, bei *little-endian* ganz links.

Sie sollen im Folgenden die Hex-Zahlen

- I) BEEF₁₆
- II) FF11₁₆

im Speicher ablegen.

- a) Welche Möglichkeiten gibt es, die Zahlen I und II im Speicher abzulegen? Geben Sie für beide Möglichkeiten den resultierenden Bitstream an.
- b) Gehen Sie davon aus, dass Sie sich für die falsche Alternative entschieden haben. Welchen Wert (im Dezimalsystem) haben Sie anstelle der beiden Zahlen im Speicher abgelegt?

Aufgabe 2.4 Berechnen Sie nach der aus der Schule bekannten Methode. Geben Sie dazu alle Schritte, das Ergebnis und Überträge nachvollziehbar an.

- a) $1037_{10} + 3802_{10}$
- b) $1010110_2 + 1010001_2$
- c) $1101001_2 + 11110_2$

Aufgabe 2.5 Berechnen Sie jeweils den angegebenen Ausdruck.

- a) Diese Zahlen seien positiv/vorzeichenlos.
Berechnen Sie $1101001_2 + 11110_2$.
- b) Diese Zahlen seien vorzeichenbehaftet.
Berechnen Sie $0101001_2 + 1011110_2$.
- c) Diese Zahlen seien im (b-1)-Komplement.
Berechnen Sie $0011110_2 + 1101001_2$.
- d) Diese Zahlen seien im b-Komplement.
Berechnen Sie $1101001_2 + 0011110_2$.

Aufgabe 2.6 Wandeln Sie die folgenden Zahlen in das angegebene System um. Geben Sie den Rechenweg an.

- a) 11000111_2 in das Dezimalsystem,
- b) 1065_7 in das Dezimalsystem,
- c) 1944_{10} in das Dualsystem,
- d) 1535_{10} in das Hexadezimalsystem,
- e) 227_{16} in das Oktalsystem,
- f) 10010001101_2 in das Oktalsystem,
- g) $10101111111110110100000001111_2$ in das Hexadezimalsystem,
- h) 5742_9 in das System zur Basis 3.

Aufgabe 2.7 Warum kann ein Computer die Menge der reellen Zahlen nur näherungsweise (approximiert) darstellen?

Aufgabe 2.8 Wandeln Sie die folgenden rationalen Zahlen in das Zielsystem um. Geben Sie den Rechenweg an.

- a) $10,625_{10}$ ins Dualsystem,
- b) $101101,1101_2$ ins Dezimalsystem.

Aufgabe 2.9 Geben Sie den vollständigen Rechenweg an. Sie können davon ausgehen, dass alle Binärzahlen in dieser Aufgabe vorzeichenlos sind.

- a) Berechnen Sie $10111011_2 \cdot 1001101_2$.
- b) Berechnen Sie $10011010_2 \cdot 111001_2$.
- c) Berechnen Sie $10011010_2 \div 10010_2$.
- d) Berechnen Sie $011101100001_2 \div 10110_2$.

Aufgabe 2.10 Binär codierte Dezimalzahlen (BCD) können in zwei Formaten dargestellt werden. Wandeln Sie die nachfolgenden Dezimalzahlen unter der Verwendung von „BCD packed“ um. Bei diesem Format wird jede Dezimalzahl als 4-bit-Zahl dargestellt.

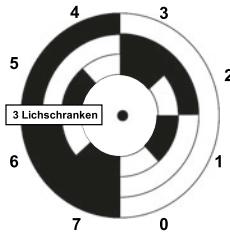


Abb. 2.19 Binäre Codierscheibe

- Erklären Sie den Begriff der Pseudotetrade.
- Wandeln Sie 377_{10} in BCD.
- Berechnen Sie die Addition von 17_{10} und 13_{10} in BCD.
- Berechnen Sie die Addition von 110_{10} und 99_{10} in BCD.
- Multiplizieren Sie 3_{10} mit 4_{10} in BCD.

Aufgabe 2.11 Ein Einsatzgebiet des Gray-Codes sind Sensor- bzw. Codierscheiben. Die Abb. 2.19 zeigt einen beispielhaften schematischen Aufbau für das Auslesen der Windrichtungen.

- In Abb. 2.20 ist die Codierscheibe für eine normale binäre Codierung dargestellt. Zeichnen Sie eine äquivalente Codierscheibe für eine Gray-Codierung und geben Sie den dazugehörigen Code an.
- Welche Vorteile bietet die Gray-Codierung gegenüber der binären Codierung? Beschreiben Sie diese Vorteile am Beispiel der Codierscheibe zur Windrichtungsmessung.
- Welche Genauigkeit hat der Windrichtungssensor bei der gegebenen Codierung, und wie könnte man diese erhöhen?

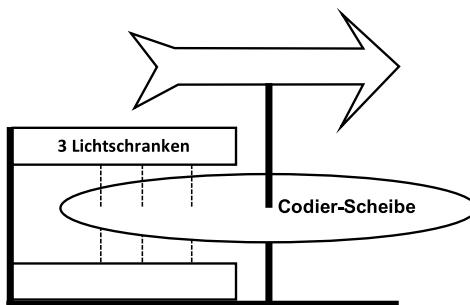


Abb. 2.20 Schematischer Aufbau des Sensors zur Messung der Windrichtung

- d) Durch eine Erweiterung des Systems könnte auch die Windgeschwindigkeit gemessen werden. Welche Erweiterungen sind hier notwendig?

Aufgabe 2.12 Wie viele Bits benötigt ein 1-aus-n-Code zur Darstellung einer Ziffer der folgende Basen?

- a) Basis $b = 16$
- b) Basis $b = 8$
- c) Basis $b = 4$
- d) Basis $b = 12$

Aufgabe 2.13 Stellen Sie die folgenden Zahlen jeweils als vorzeichenbehaftete Dualzahl, Dualzahl im b-Komplement und Dualzahl im $(b-1)$ -Komplement dar.

- a) -861_{10}
- b) 765_8
- c) -210_3

Aufgabe 2.14 Geben Sie für die folgenden Zahlen jeweils den 1-aus-n-Code an, und wandeln Sie diese in hexadezimal ($b = 16$), dezimal ($b = 10$), oktal ($b = 8$), binär ($b = 2$) und „unpacked BCD“ um.

- a) 12_3
- b) 12_4
- c) 12_7

Aufgabe 2.15 Stellen Sie die folgenden Zahlen jeweils als vorzeichenbehaftete Dualzahl, Dualzahl im b-Komplement und Dualzahl im $(b-1)$ -Komplement dar.

- a) -861_{10}
- b) 765_8
- c) -210_3

Weiterführende Literatur

- [Ifr10] Georges Ifrah. *Universalgeschichte der Zahlen*. Seghers, 2010. ISBN: 978-3942048316.
- [Dei08] Oliver Deiser. *Reelle Zahlen: Das klassische Kontinuum und die natürlichen Folgen*. Springer, Auflage: 2 (2. Juni 2008). ISBN: 978-3540793755.
- [Iwa14] Sebastian Iwanowski. *Diskrete Mathematik mit Grundlagen*. Springer Vieweg, Auflage: 2014 (2. Oktober 2014). ISBN: 978-3658071301.
- [Wit12] Kurt-Ulrich Witt. *Algebraische Grundlagen der Informatik: Strukturen -- Zahlen -- Verschlüsselung -- Codierung*. Vieweg+Teubner Verlag, Auflage: 3 (20. September 2012). ISBN: 978-3834801203.



Mathematische Grundlagen digitaler Schaltungen

Inhaltsverzeichnis

- 3.1 Algebraische Betrachtung digitaler Schaltungen – 137
 - 3.2 Boolesche Algebra – 139
 - 3.3 Schaltalgebra und Schaltfunktionen – 160
- Weiterführende Literatur – 186

Es gibt nichts Praktischeres als eine gute Theorie.
Kurt Lewin

Ressourcen sind begrenzt. Einfache mechanische Rechenmaschinen waren in ihrem Aufbau so komplex, dass es nicht möglich war, mit sehr großen Zahlen zu rechnen. Erst die elektronischen Rechner ermöglichen den Bau leistungsstarker Maschinen. Sehr große Zahlen benötigen immer wieder die gleichartigen oder besser die fast gleichen Schaltungen. Dies führt zu komplexen und regelmäßigen Strukturen und der Frage, wie sich diese optimieren lassen. Der Leser lernt in diesem Kapitel die Grundlagen der booleschen Algebra kennen. Aus dieser wird dann die Schaltalgebra abgeleitet und zur Minimierung logischer Funktionen genutzt. Aus den Schaltfunktionen lassen sich anschließend mithilfe digitaler Schaltungen grundlegende Elemente eines Rechners zusammenbauen. Nach der Behandlung der zur Beschreibung einer logischen Funktion notwendigen Strukturen wird das Problem der Schaltungsmiminierung mit einfachen grafischen und algorithmischen Verfahren gelöst. Am Ende des Kapitels sollte der Leser in der Lage sein, den mathematischen Aufbau von Computern zu erklären und die zugrunde liegenden Schaltfunktionen selbstständig zu minimieren. Denn eine kleine Schaltung mit wenigen Bauelementen bedeutet auch eine zuverlässige Schaltung. Er sollte die Zusammenhänge zwischen der booleschen und der Schaltalgebra erläutern können und eine Idee entwickeln, warum moderne Logiksynthese beim Entwurf heutiger Rechner eine große Rolle spielt.

Die unterschiedlichen Beispiele früher Computer zeigen, dass die Wahl der zu verwendenden Basis eines Zahlensystems sowohl von der Bedienung als auch von der zur Verfügung stehenden Technologie abhängt. Die Gewohnheit, im Dezimalsystem zu rechnen, und die Mechanik der Staffelwalzmaschine legen es nahe, eine Maschine ebenfalls in diesem System zu betreiben. In der Anfangszeit des Computers hat man dies auch oft so gemacht. Allerdings zeigt sich schnell, dass die Komplexität einer Maschine mit der Anzahl unterschiedlich darzustellender Zeichen für eine Ziffer steigt. Daraus folgt eine aufwendige Implementierung von Grundbausteinen zum Rechnen. Beim Entwurf von Rechenmaschinen ist man daher bemüht, die Menge der eingesetzten Bauelemente auf wenige Grundbauteile zu reduzieren. Da die Anzahl der verwendeten Zeichen im Binär- oder Dualsystem die kleinstmögliche unterscheidbare Zahl von Zeichen darstellt, diese zwei Zustände einfach zu realisieren sind und die Arithmetik dieselbe wie im Dezimalsystem ist, rechnen moderne Computer binär. Die ungewohnte Handhabung ist bei einer vollautomatischen Maschine irrelevant, und die Darstellung der Zeichen im gewohnten dezimalen Zahlen-

system kann anschließend durch die Ein- und Ausgabegeräte übernommen werden. Der Vorteil des Binärsystems ist, dass sich seine zwei Zustände physisch durch einfache Schalter realisieren lassen. Diese konzeptionelle Reduktion ermöglicht es, die Werte analoger physikalischer Größen, die diese Zustände repräsentieren, großzüig zu trennen. Dadurch haben externe Störungen und Bauteiltoleranzen einen geringen Einfluss auf die Rechenergebnisse. Im folgenden Kapitel wird die binäre Logik als Grundlage der Implementierung von Recheneinheiten eingeführt. Dazu wird zunächst die boolesche Algebra hergeleitet, Funktionen auf ihr definiert und gezeigt, wie mit diesem mathematischen Fundament die Schaltalgebra abgeleitet werden kann. Mit dieser werden dann die Rechenoperationen in binärer Logik realisiert und diese für den Entwurf digitaler Schaltungen eingeführt. Der rein konstruktiven Verwendung der Schaltalgebra schließt sich die Minimierung von Schaltfunktionen an. Ziel ist es, die für den Bau eines Rechners notwendige Anzahl von Schaltern so weit wie möglich zu reduzieren, um Kosten zu sparen.

3.1 Algebraische Betrachtung digitaler Schaltungen

Es spielt für eine Maschine keine Rolle, zu welcher Basis das Stellenwertsystem gewählt wird. Wir Menschen sind das dezimale gewohnt, aber die mathematischen Grundlagen der Arithmetik sind unabhängig von der Basis des verwendeten Zahlensystems. Die Wahlfreiheit dieser ermöglicht bei der Konstruktion einer Rechenmaschine das Zahlensystem mit dem geringsten Aufwand bei der Implementierung der Maschine zu wählen. Das Ziel, einen kostengünstigen, leistungsfähigen und zuverlässigen Rechner zu bauen, wird am besten erreicht, wenn die Bauelemente möglichst einfach sind. Mit modernen Halbleiterprozessen können Schalter in großer Zahl und preiswert hergestellt werden. Derartige Transistoren eignen sich hervorragend zur Realisierung einer binären, dualen oder zweiwertigen Logik. Um die Anzahl der zu implementierenden Schalter möglichst gering zu halten, ist es notwendig, die für eine Rechenoperation notwendige Schaltung zu minimieren. Eine derartige Minimierung stellt ein Optimierungsproblem dar, dass wiederum mit mathematischen Methoden gelöst werden kann. Aus diesem Grund wird eine zweiwertige Algebra benötigt, die in diesem Kapitel eingeführt wird.

Dieser Sachverhalt führt zu einem interessanten Zusammenhang: Wissenschaftler möchten Rechner bauen, um langweilige Rechenoperationen zu automatisieren. Da die Konstruktion der Maschinen selbst auch ein mathematisches und

Schaltungsminimierung

Entwurfsautomatisierung

3

rechenintensives Problem ist, können diese Computer dazu verwendet werden, um wiederum bessere Anlagen zu konstruieren. Wie Münchhausen sich an seinem Schopf aus dem Sumpf zieht, verwenden die Konstrukteure von Rechenmaschinen immer bessere Maschinen, um diese zu bauen. Seit Mitte der 1960er-Jahre ist so ein neues Teilgebiet der Informatik und Elektrotechnik entstanden: die Automatisierung des Entwurfs von Computern und von integrierten Schaltungen. In der Industrie wird der Bereich elektronische Entwurfsautomatisierung (Electronic Design-Automation, EDA) genannt. Der Begriff grenzt sich bewusst vom Computer-aided Design (CAD) ab, weil die EDA Algorithmen und Werkzeuge für den Entwurf elektronischer und digitaler Schaltungen bereitstellt, die über das reine Zeichnen von Schaltplänen hinausgehen.

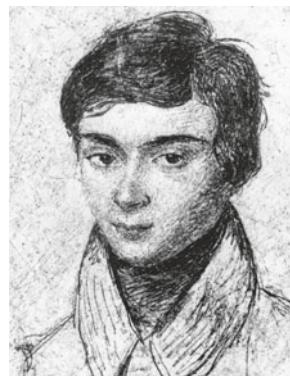
Bereits Konrad Zuse verwendete in seinen Rechenmaschinen elektrische Relais als Schalter. Am MIT (Massachusetts Institute of Technology) wurde in den 1930er-Jahren ein Analogrechner zum Lösen von Differenzialgleichungen entwickelt. In diesem Rechner kamen ebenfalls Relais als elektrische Schalter zum Einsatz. Die Probleme beim Entwurf einer solchen Maschine führten zu der 1937 von Claude Shannon (1916–2001) in seiner Masterarbeit entwickelten Schaltalgebra. Offensichtlich können Rechenmaschinen algebraisch konstruiert werden. Die Schaltalgebra ist heute ein grundlegendes Werkzeug beim Entwurf digitaler Rechner. Als mathematische Methode wird sie eingesetzt, um Computer zu konstruieren und zu optimieren. Sie erlaubt es, mit einer Rechnung zu beweisen, dass eine entworfene Logik genau das tut, was von ihr erwartet wird. In der Sprache des Informatikers wird die Schaltung gegen ihre Spezifikation verifiziert. Damit ist gemeint, einen maschinenlesbaren Schaltplan dahingehend zu überprüfen, ob er auch die Anforderungen einer formalen Beschreibung seiner Funktion erfüllt. Die Methode der Schaltalgebra ist fundamental für die Konstruktion von Rechenmaschinen. Das mathematische Fundament sind algebraische Strukturen. In diesem Kapitel wird die Schaltalgebra aus diesen abgeleitet. Dies soll das Verständnis bei der Anwendung erhöhen. Insbesondere die Äquivalenz zu den anderen Logiken oder Algebren der Informatik ist interessant, da die Schaltalgebra eng mit den formalen Mechanismen der Algorithmen, Datenstrukturen und Programmierung verknüpft ist. Ausgehend von der algebraischen Struktur der booleschen Algebra werden logische Funktionen zur Konstruktion von digitalen Schaltungen eingeführt. Es wird gezeigt, wie mit deren Hilfe die grundlegenden Bausteine von Rechenanlagen im ingeniermäßigen Sinne formal konstruiert werden können. Darüber hinaus erlaubt es der Formalismus, Softwarewerkzeuge zu schreiben, die in der Lage sind, Schaltungen automatisch zu erzeugen. Dieser Logiksyn-

these genannte Schritt ermöglicht es, Rechner aus Millionen von Transistoren zu bauen und diese auf engstem Raum zu integrieren. Das Internet, der Mobilfunk und deren Verknüpfung in Form von Smartphones wären ohne die im Folgenden vorgestellten mathematischen Grundlagen nicht realisierbar.

3.2 Boolesche Algebra

In der Mittelstufe lernt jeder Schüler ein wichtiges Teilgebiet der Mathematik kennen: die Algebra. Dieses klassische oder elementare Algebra genannte Gebiet geht bereits auf den Griechen Diophantos von Alexandria (irgendwann zwischen 100 v. Chr. und 350 n. Chr.) zurück. Das Wort Algebra stammt aus dem Arabischen und bedeutet *al-jabr*, die Vereinigung des Geteilten. In der klassischen Algebra werden algebraische Gleichungen und die Eigenschaften ihrer Lösungen untersucht. Seit Mitte des 19. Jahrhunderts hat sich eine neue Form der Algebra entwickelt. Ausgehend von den Arbeiten von Évariste Galois (1811–1832, □ Abb. 3.1) entstand die Gruppen- und Ringtheorie. Dieses moderne oder abstrakte Algebra genannte Teilgebiet der Mathematik beschreibt allgemein mathematische Strukturen und deren Eigenschaften. Algebren beschäftigen sich mit dem Studium von Symbolen und Regeln, um diese zu verändern. Allgemein ist eine Algebra eine mathematische Struktur, bestehend aus einer Menge von Symbolen und einer oder mehreren Operationen auf diesen. Symbole können Zahlen oder Buchstaben und Operationen beispielsweise Multiplikation und Addition sein. Dabei müssen die Operationen bestimmte Eigenschaften wie die Assoziativität, Kommutativität oder Absorption erfüllen. Im Folgenden werden Strukturen zur Beschreibung der Konstruktion von Rechenmaschinen (Hardware) eingeführt.

Algebra



Adelaide Pauline Chantelot,
Wikimedia Commons

□ Abb. 3.1 Évariste Galois

3.2.1 Mathematische und strukturelle Grundlagen

Dieses Kapitel führt in die grundlegende Mathematik der booleschen Algebra ein. Es werden die grundlegenden mathematischen Begriffe erklärt und angewendet, um das Rüstzeug zur Schaltungsminimierung anzulegen – insbesondere, da in einem einführenden Kurs in die Rechnerarchitektur nicht davon ausgegangen werden kann, dass alle Leser das gleiche mathematische Vorwissen mitbringen. Die Einführung erhebt keinen Anspruch auf formale Strenge, und bezüglich streng formaler und vollständiger Ableitungen sei auf die Fachliteratur der Mathematik verwiesen.

Menge

Eine algebraische Struktur besteht aus einer Menge von Elementen und Operationen zur Verknüpfung dieser. Ausgehend von diesem Begriff lässt sich die boolesche Algebra ableiten. Die Schaltalgebra ist dann nichts weiter als eine spezielle Form der booleschen Algebra. Die Eigenschaften der Schaltalgebra erlauben Rechenregeln, mit denen logische Schaltungen minimiert werden können, aufzustellen. Abgeleitet aus der Aussagenlogik der klassischen Philosophie über die boolesche Algebra bietet die Schaltalgebra ein solides mathematisches Fundament zur Beschreibung und Berechnung digitaler Schaltungen. Dies ist ein spezielles Beispiel der fundierten Anwendung der Mathematik auf konkrete technische Probleme.

Eine Menge ist eine Sammlung von Objekten. In der Mengenlehre werden die Objekte als Elemente bezeichnet. Eine Menge wird in der Mathematik durch zwei geschweifte Klammern {} dargestellt. Natürlich kann eine Menge auch durch einen allgemeinen Bezeichner ausgedrückt werden. So kann die Menge der Dezimalziffern mit D abgekürzt werden, also $D := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Eine besondere Menge ist die leere Menge. Abkürzende Schreibweise der leeren Menge ist $D := \{\}$ oder \emptyset . Die Anzahl von Elementen einer Menge wird Kardinalität der Menge genannt. Sei A die Menge der Kleinbuchstaben des Alphabets, dann ist $|A| = 26$. Sei D die Menge der Dezimalziffern, dann ist $|D| = 10$.

► **Beispiel 3.2.1 Menge**

Beispiele für Mengen sind das Alphabet der kleinen Buchstaben $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$, die Dezimalziffern $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ oder die vier Farben von Spielkarten $\{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$. ◀

Alle Elemente einer Menge aufzählend aufzuschreiben ist oft mühselig, insbesondere wenn eine Menge viele Objekte enthält. Mengen lassen sich aus diesem Grund auch zusammenfassend schreiben. So kann die Menge der Dezimalstellen einfach in folgender Form angegeben werden: $D := \{x \mid x \in \mathbb{N}, x < 10\}$. In Worten bedeutet diese Schreibweise, dass D die Menge aller x ist, für die gilt, dass x Element der natürlichen Zahlen und $x < 10$ ist. In dem Beispiel wurde also eine bereits definierte Menge genutzt, um eine Teilmenge dieser zu definieren. Mathematisch wird dies wie folgt formuliert: $D \subset \mathbb{N}$.

Beliebige Mengen lassen sich zusammenfassen: Eine Tastatur mit Kleinbuchstaben A und Ziffern D bildet die Menge der Tasten $T := A \cup D$. Oder wieder in Worten: T ist die Vereinigung der Mengen A und D . Neben der Operation Vereinigung lassen sich Schnittmengen bilden: Eine Schnittmenge ist eine Menge, die nur Elemente enthält, die in beiden Ausgangsmengen vorhanden sind. Wird die Menge A mit der Menge

Operationen auf Mengen

D geschnitten, $S := A \cap D = \emptyset$, ergibt sich die leere Menge, da kein Element von A auch in B vorkommt. Bezuglich einer gegebenen Grundmenge (oder eines Universums) aus Elementen eines bestimmten Sachverhaltes kann zu jeder Teilmenge der Grundmenge ein Komplement gebildet werden. So sind die natürlichen Zahlen beispielsweise eine Grundmenge bezüglich der Teilmenge der Dezimalziffern. Das Komplement $\bar{D} := \{x \mid x \in \mathbb{N}, x \geq 10\}$ enthält alle Elemente von \mathbb{N} , die nicht in D sind.

Über zwei Mengen A und D kann das Kreuzprodukt gebildet werden. Dieses auch „kartesisches Produkt“ genannte Konstrukt erzeugt eine Menge aller geordneten Paare der gegebenen Mengen. Im gewählten Beispiel ergibt $A \times D := \{(a,0), (a,1), (a,2), (a,3), (a,4), (a,5), (a,6), (a,7), (a,8), (a,9), (b,0), (b,1), (b,2), (b,3), (b,4), (b,5), (b,6), (b,7), (b,8), (b,9), \dots, (z,4), (z,5), (z,6), (z,7), (z,8), (z,9)\}$. Anwenden lässt sich das kartesische Produkt auch auf zwei gleiche und auf mehr als zwei Mengen: $A \times A$ und $A \times A \times A$ oder A^2 und A^3 . Daraus lässt sich die Potenzmenge ableiten: Die Potenzmenge enthält alle möglichen Teilmengen der Menge. Sei Z die Menge der natürlichen Zahlen kleiner als 3, $Z := \{x \mid x \in \mathbb{N}, x < 3\} = \{1, 2\}$, dann ist $2^V = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Eine Relation Λ oder Beziehung zweier Mengen A und B , $A \diamond B$ ist eine Teilmenge aus $A \times B$ oder $A \diamond B := \{x \mid x \subset A \times B\}$.

Relation

► Beispiel 3.2.2 – Relation

Seien die Menge $S := \{\text{Alina, Sina, Tarek}\}$ eine Teilmenge der Studenten einer Universität und die Menge $M := \{\text{Pop, Jazz, Punk, Ska}\}$ eine Teilmenge unterschiedlicher Musikrichtungen, dann lässt sich die Relation \equiv (hört oder mag) definieren: $S \equiv M := \{(\text{Alina, Pop}), (\text{Alina, Ska}), (\text{Sina, Pop}), (\text{Sina, Jazz}), (\text{Tarek, Punk}), (\text{Tarek, Ska})\}$. Elemente einer Relation sind geordnete Paare. Zu jeder Relation Λ gibt es eine inverse Relation Λ^{-1} ; für das Beispiel also $S \equiv^{-1} M := \{(\text{Alina, Jazz}), (\text{Alina, Punk}), (\text{Sina, Punk}), (\text{Sina, Ska}), (\text{Tarek, Pop}), (\text{Tarek, Jazz})\}$. ◀

Eine Funktion f von A nach B , $f : A \rightarrow B$, ist eine Relation oder Beziehung zweier Mengen, die genau einem Element der einen Menge genau ein Element der anderen Menge zuweist.

Funktion

► Beispiel 3.2.3 – Funktion

Angenommen, M sei die Menge $\{\text{Tarek, Harald, Tobias, Peter}\}$ und sei $F := \{\text{Eva, Karin, Alina, Sina}\}$, dann ist „ist verheiratet mit“ im abendländischen Kulturkreis eine Funktion $v : M \rightarrow F := \{(\text{Tarek, Eva}), (\text{Harald, Alina}), (\text{Tobias, Sina}), (\text{Peter, Alina})\}$. Schreibt man $y = f(x)$, dann ist y die Abbildung von x auf y unter f . ◀

Ordnungsrelation

Auf einer beliebigen Menge M lässt sich eine Ordnungsrelation definieren. Eine solche Relation vergleicht und sortiert Elemente gemäß einer vorgegebenen Metrik. Eine Ordnungsrelation ist eine 2-stellige Relation $R \subseteq M \times M$. Eine geordnete Menge ist definiert durch das Tupel (M, R) mit der Menge M und der Ordnungsrelation R . Weniger abstrakt lässt sich R oder \leq schreiben. Für zwei Elemente einer Menge $a \in M$ und $b \in M$ gilt für ein beliebiges Tupel $(a, b) \in R$. Man kann auch schreiben aRb oder unter Verwendung des Symbols $\leq: a \leq b$.

► Beispiel 3.2.4 – Ordnungsrelation

Es ist offensichtlich, dass für die Menge $D := \{x \mid x \in \mathbb{N}, x < 10\}$ die Ordnungsrelation $R := \{(0,0), (0,1), (0,2), (0,3), \dots, (0,9), (1,1), (1,2), \dots, (2,2), (2,3), \dots, (2,9), \dots, (8,9), (9,9)\}$ gilt, oder anders geschrieben $0 \leq 0, 0 \leq 1, \dots, 8 \leq 9, 9 \leq 9$. ◀

Eine besondere Rolle bei der Definition einer booleschen Algebra spielt die Halbordnung.

Definition 3.2.1 – Halbordnung

Eine Halbordnung ist eine reflexive, antisymmetrische und transitive Ordnungsrelation:

Reflexivität: $x \leq x$

Antisymmetrie: $x \leq y \wedge y \leq x \rightarrow x = y$

Transitivität: $x \leq y \wedge y \leq z \rightarrow x \leq z$

Reflexivität bedeutet, dass jedes Element in Relation zu sich selbst steht, antisymmetrisch heißt, dass es keine zwei Elemente gibt, für die die Umkehrung der Ordnungsrelation gilt, und transitiv meint, dass sich die dritte Ordnungsrelation zwischen drei Elementen einer Menge automatisch aus den zwei Ordnungsrelationen ableiten lässt. Formal schreibt man für diese Eigenschaften einer Halbordnung: Die Menge $D := \{x \mid x \in \mathbb{N}, x < 10\}$ ist eine solche Halbordnung.

Definition 3.2.2 – Größte untere Grenze, Infimum

Ein Element einer Halbordnung P ist eine untere Grenze u der Elemente $a, b \in P$, wenn gilt $u \leq a$ und $u \leq b$. Ein Element $i \in P$ ist die größte untere Grenze (Infimum) der Elemente $a, b \in P$, wenn es die folgenden zwei Eigenschaften besitzt:

1. $i \leq a$ und $i \leq b$,
2. $\forall u \in P$ gilt, dass, wenn $u \leq a$ und $u \leq b$, das Infimum $i \geq u$ ist. Für $a \leq b$ schreibt man auch $a \cdot b$.

► Beispiel 3.2.5 – Infimum

Gegeben sei die Menge $D := \{x \mid x \in \mathbb{N}, x < 10\}$. Für die beiden Elemente $a = 2$ und $b = 5$ sind die Elemente $\{0, 1, 2\}$ untere Schranken u , da gilt $0 \leq 2$ und $0 \leq 5$, $1 \leq 2$ und $1 \leq 5$, $2 \leq 2$ und $2 \leq 5$. Damit sind $\{0, 1, 2\}$ untere Schranken von D . Das Infimum ist $i = 2$, da $0 \leq 2$, $1 \leq 2$ und $2 \leq 2$. Das Infimum $i = 2$ von $(2, 5)$ ist die größte untere Schranke aus $\{0, 1, 2\}$. ◀

Definition 3.2.3 – Kleinste obere Grenze, Supremum

Ein Element einer Halbordnung P ist eine obere Grenze u der Elemente $a, b \in P$, wenn gilt, $u \geq a$ und $u \geq b$. Ein Element $s \in P$ ist die kleinste obere Grenze (Supremum) der Elemente $a, b \in P$, wenn es die folgenden zwei Eigenschaften besitzt:

1. $s \geq a$ und $s \geq b$,
2. $\forall u \in P$ gilt, dass, wenn $u \geq a$ und $u \geq b$, das Supremum $s \leq u$ ist. Für $a \leq b$ schreibt man auch $a + b$.

► Beispiel 3.2.6 – Supremum

Gegeben sei die Menge $D := \{x \mid x \in \mathbb{N}, x < 10\}$. Für die beiden Elemente $a = 2$ und $b = 5$ sind die Elemente $\{5, 6, 7, 8, 9\}$ obere Schranken o , da gilt $5 \geq 2$ und $5 \geq 5$, $6 \geq 5$ und $6 \geq 2$ usw. Damit sind $\{5, 6, 7, 8, 9\}$ obere Schranken von $(2, 5)$. Das Supremum ist $s = 5$, da $6 \geq 5$, $6 \geq 2$ und $6 \leq 5$. Für $\{7, 8, 9\}$ gilt das Gleiche. Das Supremum $s = 5$ von $(2, 5)$ ist die kleinste obere Schranke aus $\{5, 6, 7, 8, 9\}$. ◀

Ordnungsrelationen lassen sich grafisch visualisieren. In der Informatik werden zur Visualisierung häufig Graphen verwendet. Ein Graph besteht aus Knoten und Kanten, wobei die Kanten die Knoten des Graphen verbinden. Man unterscheidet zwischen ungerichteten und gerichteten Graphen. Bei gerichteten Graphen haben die Kanten eine ausgezeichnete Richtung. Grafisch wird der Knoten durch einen Punkt oder Kreis und die Kante bei ungerichteten Graphen durch eine Verbindungsline und bei gerichteten Graphen durch einen Pfeil dargestellt. Die Abb. 3.2 zeigt einen ungerichteten und Abb. 3.3 einen gerichteten Graphen.

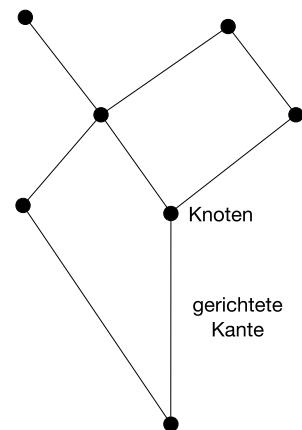


Abb. 3.2 Graph

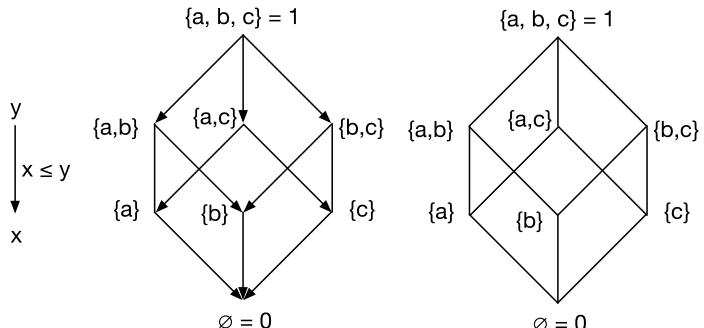


Abb. 3.4 Bildung eines Hasse-Diagramms für Mengen. a) mit gerichteten Kanten b) mit impliziter Ordnung

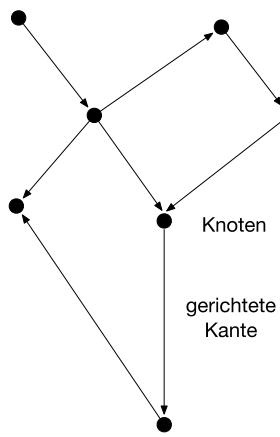


Abb. 3.3 Gerichteter Graph

Die Relation $x \leq y$ kann durch eine gerichtete Kante eines Graphen dargestellt werden. Dabei zeigt der Pfeil auf das kleinere Element. Eine Ordnungsrelation kann somit durch einen Graphen repräsentiert werden. Wenn z. B. $M := \{a, b, c\}$ eine Menge ist, dann ist die Potenzmenge von M die Menge $2^M := \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Eine Ordnungsrelation auf der Potenzmenge lässt sich durch den Betrag der jeweiligen Teilmengen, also die Anzahl der in der Teilmenge vorhandenen Elemente definieren. Die Abb. 3.4a zeigt den diese Ordnungsrelation darstellenden Graphen ohne explizite Kennzeichnung der Knoten. Wenn man nun die Konvention einführt, dass Knoten, die oben im Diagramm stehen, größer sind als Knoten, die unten im Diagramm stehen, und verzichtet man gleichzeitig auf waagerechte gerichtete Kanten, kann man die Ordnungsrelation auch durch einen ungerichteten Graphen darstellen. Dies ist in Abb. 3.4b gezeigt. Einen ungerichteten Graphen, bei dem angenommen wird, dass grafisch weiter oben stehende Knoten größer im Sinne der beschreibenden Ordnungsrelation sind, nennt man Hasse-Diagramm. Ordnungsrelationen lassen sich mit wenig Aufwand mit Hasse-Diagrammen visualisieren.

3.2.2 Definition und grundlegende Eigenschaften boolescher Algebren

Im Jahr 1830 entwickelte der Engländer George Boole (1815–1864) eine mathematische, streng formale Beschreibung für die klassische Logik der Philosophie. Die Idee, mithilfe einer formalen Vorgehensweise das philosophische Denken zu axiomatisieren, war im frühen 19. Jahrhundert neu und überraschte die Mathematiker. Es war damit erstmals möglich, logische

Schlussfolgerungen auf mathematische Berechnungen zurückzuführen. Es entstand eine Algebra des Denkens. Im Folgenden soll daher auf die formale Struktur der booleschen Algebra eingegangen werden. Durch Rechnen lassen sich in den Ingenieurwissenschaften und der Physik komplexe Sachverhalte verstehen. Dies führt zur Vermeidung und Aufdeckung von Zirkelschlüssen. Rechnen erlaubt darüber hinaus quantitative Vorhersagen und damit eine numerische Überprüfung von Überlegungen. Dazu wird die boolesche Algebra als mathematische Struktur, die sich aus allgemeineren Axiomen der Algebra ableiten lässt, aufgefasst.

Ein Verband ist eine Menge mit festgelegten Eigenschaften oder mit anderen Worten eine mathematische Struktur. Allgemein ist ein Verband eine Halbordnung (P^2, \leq) , in der jedes Paar von Elementen (a, b) eine größte untere Schranke (Infimum) und eine kleinste obere Schranke (Supremum) hat. Daraus besitzt jeder Verband ein Infimum, bezeichnet mit $0 \in P$, und ein Supremum, als $1 \in P$ geschrieben. Die Operationen auf die Menge des Verbandes sind so definiert, dass sie assoziativ und kommutativ sind.

Verband

Definition 3.2.4 – Verband

Ein Verband (V, \sqcup, \sqcap) ist eine algebraische Struktur mit einer nicht leeren Menge V und zwei binären Operationen \sqcup (Vereinigung, engl. „join“) und \sqcap (Durchschnitt, engl. „meet“). Die beiden binären Operationen sind assoziativ und kommutativ. Darüber hinaus gelten die Absorptionsgesetze.

$$\text{Assoziativität: } a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \quad \text{und}$$

$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$$

$$\text{Kommutativität: } a \sqcup b = b \sqcup a \quad \text{und}$$

$$a \sqcap b = b \sqcap a$$

$$\text{Absorption: } a \sqcup (a \sqcap b) = a \quad \text{und}$$

$$a \sqcap (a \sqcup b) = a$$

Daraus folgt mit $a \sqcup a = a$ und $a \sqcap a = a$ die Eigenschaft, dass wenn ein Element mit einer Operation mit sich selbst verknüpft wird, das Element unverändert zurückgegeben wird. Ein Verband kann in einen anderen umgewandelt oder transformiert werden, indem man \sqcup und \sqcap , \leq und \geq , 0 und 1 vertauscht. Diese Eigenschaft heißt Dualität.

Ausgehend vom Infimum 0 und Supremum 1 eines Verbandes nennt man ein Element \bar{a} eines beliebigen gegebenen Elementes a komplementäres Element, wenn es die Eigenschaften $a \sqcup \bar{a} = 1$ und $a \sqcap \bar{a} = 0$ hat. Das Komplement wird häufig auch als

Komplementarität

Distributivität

$\neg a$ geschrieben. Ein Verband, für den für jedes Element ein komplementäres Element existiert, nennt man komplementär.

Manche Verbände sind distributiv. In einem solchen Fall handelt es sich um einen distributiven Verband, und es gilt:

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c) \text{ und } a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$

Mit diesen grundlegenden Definitionen kann die boolesche Algebra formuliert werden.

Definition 3.2.5 – Boolesche Algebra

Eine boolesche Algebra ist ein komplementärer, distributiver Verband $(B, \cdot, +, 0, 1)$.

Durch die Rückführung einer booleschen Algebra auf die Struktur des Verbandes ergeben sich automatisch die folgenden Eigenschaften:

Assoziativität:	$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$ oder $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
	$a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$ oder $a + (b + c) = (a + b) + c$

Kommutativität:	$a \sqcup b = b \sqcup a$ oder $a \cdot b = b \cdot a$
-----------------	--

	$a \sqcap b = a \sqcap b$ oder $a + b = a + b$
--	--

Distributivität:	$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$ oder $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
------------------	---

	$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$ oder $a + (b + c) = (a + b) \cdot (a + c)$
--	---

Idempotenz:	$a \sqcup a = a$ oder $a \cdot a = a$
-------------	---------------------------------------

	$a \sqcap a = a$ oder $a + a = a$
--	-----------------------------------

Absorption:	$a \sqcup (a \sqcap b) = a$ oder $a \cdot (a + b) = a$
-------------	--

	$a \sqcap (a \sqcup b) = a$ oder $a + (a \cdot b) = a$
--	--

Die abstrakten Operatoren \sqcap und \sqcup werden im Folgenden bewusst durch die Operatoren \cdot und $+$ ersetzt, um durch die Verwandtschaft zur Schulalgebra die Anwendung zu vereinfachen. Es empfiehlt sich bei Rechnungen oder Beweisführungen die gewohnten algebraischen Operatoren zu verwenden. Zum einen können abkürzende Schreibweisen wie das Weglassen des Punktes beibehalten werden, zum anderen sind die Regeln so ähnlich, dass sich die erlernte Intuition beim Rechnen mit booleschen Ausdrücken nutzen lässt.

Das abstrakte Konzept algebraischer Strukturen ist ein mächtiges mathematisches Werkzeug. Auch die boolesche Algebra ist zunächst einmal völlig abstrakt formuliert und muss für konkrete Anwendungen konkretisiert werden. Dies geschieht durch die entsprechende Festlegung der Elemente der Trägermenge und der konkreten Konstruktion der darauf auszuführenden Operationen. Beispiele sind die Teileralgebra, die Mengenalgebra und die Schaltalgebra.

Theorem 3.2.1 – Teileralgebra*Die algebraische Struktur*

$$(T, \text{gg}T(+), \text{kg}V(\cdot), 0, n \in \mathbb{N} \setminus 0)$$

mit der Menge aller Teiler von n und den Operationen größter gemeinsamer Teiler ($\text{gg}T$) und kleinstes gemeinsames Vielfaches ($\text{kg}V$) ist eine boolesche Algebra.

■ Beweis

Die Teileralgebra erfüllt die Eigenschaften der booleschen Algebra, wenn n aus einfachen Primfaktoren zusammengesetzt ist, also ein Primfaktor nicht mehr als einmal in n enthalten ist. ■

Die Teileralgebra ist ein interessantes Beispiel. Scheint sie für den praktischen Rechnerarchitekten doch keinen wirklichen Nutzen zu haben, so zeigt sie exemplarisch, wie mächtig ein paar wenige strukturelle Regeln sind. Insbesondere, da die Zahlentheorie und Fragestellungen zur Teilbarkeit durch ihre Anwendung in der Kryptografie den Architekten digitaler Schaltungen spätestens dann wieder beschäftigen, wenn Ver- und Entschlüsselmaschinen in Hardware gebaut werden sollen.

► Beispiel 3.2.7 – Teileralgebra

Wählt man $n = 30 = 2 \cdot 3 \cdot 5$ und $T := \{0, 1, 2, 3, 5, 6, 10, 15\}$ als Trägermenge, dann ist $(T, \text{gg}T, \text{kg}V, 0, 30)$ eine boolesche Algebra. Die Menge aller Teiler kann mit der Operation „ist Teiler von“ ebenfalls als Halbordnung aufgefasst werden. Die entsprechende Ordnungsrelation zeigt die □ Abb. 3.5a. ◀

Das Beispiel zeigt, welche Eigenschaften eine Zahlenmenge erfüllen muss, damit sich die Operationen der Teileralgebra nutzen lassen. Aber nicht alle Mengen natürlicher Zahlen erzeugen eine solche Struktur.

► Beispiel 3.2.8 – Gegenbeispiel

Die Struktur $(T, \text{gg}T, \text{kg}V, 0, 24)$ mit $T \in \{0, 1, 2, 3, 4, 6, 8, 12\}$ ist keine boolesche Algebra, da $n = 24 = 2 \cdot 2 \cdot 2 \cdot 3$.

Begründung: Der Primfaktor 2 kommt bei der Zerlegung der 24 mehr als einmal vor. Des Weiteren ist die Teileralgebra eine Submenge der booleschen Algebra, und damit ist die Struktur weder eine boolesche Algebra noch eine Teileralgebra. ◀

Eine der grundlegenden mathematischen Strukturen ist die Mengenalgebra. Diese kann direkt genutzt werden, die für den Entwurf digitaler Schaltungen notwendige Schaltalgebra formal abzuleiten.

Theorem 3.2.2 – Mengenalgebra

Sei M eine beliebige Menge mit

1. $A + B = A \cup B$
2. $A \cdot B = A \cap B$
3. $\overline{A} = B \notin A$

Wenn das Infimum die leere Menge und das Supremum M sind, dann ist $(2^M, \cap, \cup, \emptyset, M)$ eine boolesche Algebra, die Mengenalgebra heißt.

■ Beweis

Es lässt sich leicht überprüfen, dass die Mengenalgebra die Eigenschaften einer booleschen Algebra nach 3.2.2 erfüllt. ■

Ein wesentlicher Aspekt der Mengenalgebra ist die Definition von Ordnungsrelationen. Diese erlauben es, Abbildungen auf Mengen und somit Funktionen zu definieren. Damit lassen sich Elemente der einen Menge eindeutig einer anderen Menge zuordnen. Die Mengenalgebra bildet das Fundament der Schaltalgebra. Beim Bau von Rechenmaschinen geht es genau darum, Werte am Eingang einer Schaltung mit dieser in eindeutige Ausgaben umzuwandeln.

► Beispiel 3.2.9 – Ordnungsrelation der Potenzmenge

In der Mengenalgebra kann eine Ordnungsrelation über die Teilmenge \subseteq definiert werden. Sei $M := \{a, b, c\}$ eine Menge, dann stellt die Potenzmenge $2^M = 2^{a, b, c}$ eine Ordnungsrelation dar. Für das Beispiel ist diese in □ Abb. 3.5b dargestellt. ◀

Mit diesen fundamentalen Gesetzen kann die Schaltalgebra direkt aus der booleschen Algebra abgeleitet und mithilfe der Mengenalgebra begründet werden. Im Vorgriff auf Abschn. 3.3 sei dies im Folgenden exemplarisch gezeigt.

► Beispiel 3.2.10 – Ordnungsrelation der Trägermenge {0,1}

Auf Vektoren der Menge $M := \{0, 1\}$ lässt sich eine Ordnungsrelation definieren: $(0, 1^n, \leq)$. Die Relation \leq ist dabei komponentenweise beschrieben. Ein Vektor ist genau dann kleiner gleich eines anderen Vektors, wenn gilt $(a_1, \dots, a_n) \leq (b_1, \dots, b_n)$. Die □ Abb. 3.5c zeigt die entsprechende Ordnungsrelation für die Menge $(0, 1^3, \leq)$. ◀

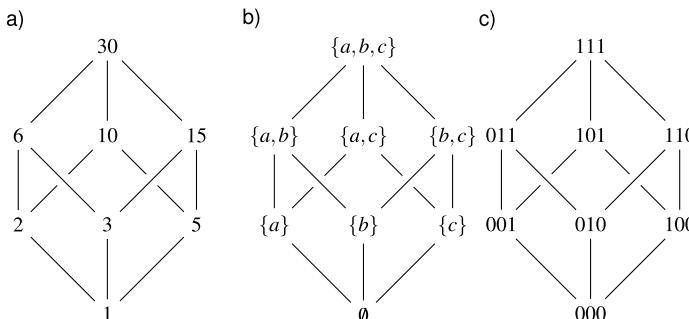


Abb. 3.5 Hasse-Diagramme verschiedener Mengen

3.2.3 Rechnen in der booleschen Algebra

Boolesche Algebren besitzen einige bemerkenswerte Eigenschaften, um algebraische Berechnungen zu ermöglichen und im Falle der Schaltalgebra die Konstruktion und Optimierung logischer Schaltungen systematisch und formal durchzuführen. Da eine boolesche Algebra ein komplementärer, distributiver Verband ist, existiert offenbar eine Ordnungsrelation: Es gilt $\emptyset \leq \{a\} \leq \{a, b\}$ und $\emptyset \leq \{b\} \leq \{a, b\}$. Dieser Zusammenhang offenbart sich auch in der Ableitung der Schaltalgebra aus der Mengenalgebra. Dadurch lässt sich später der Aufwand an notwendigen Schaltern in einer Implementierung von Hardware schon auf der Ebene der logischen Konstruktion von Rechenfunktionen beziffern. Damit sind die Ordnungsrelationen der Schlüssel zur Minimierung digitaler Funktionen und Grundlage der Reduktion von Kosten im Entwurf integrierter Schaltungen. Die wichtigsten Rechenregeln sind die Sätze von De Morgan. Um diese beweisen zu können, sind zunächst ein paar wenige Vorüberlegungen durchzuführen.

Theorem 3.2.3 – Existenz eines einzigen Komplements

In einer booleschen Algebra existiert für ein gegebenes Element der Trägermenge nur ein einziges komplementäres Element.

Beweis

Angenommen \bar{a} und b sind komplementär zu a . Wenn gilt $\bar{a} = b$, existiert in einer booleschen Algebra zu jedem Element nur ein komplementäres Element. Wenn \bar{a} und wenn b ein Komplement von a sind, dann gilt $a \cdot \bar{a} = 0$, $a + \bar{a} = 1$ und $a \cdot b = 0$, $a + b = 1$. Damit kann man zeigen, dass es in einer booleschen Algebra zu jedem Element nur ein komplementäres Element gibt:

$$\begin{aligned} b &= b \cdot 1 = b(a + \bar{a}) = ba + b\bar{a} = 0 + b\bar{a} = a\bar{a} + b\bar{a} \\ &= \bar{a}(b + a) = \bar{a} \cdot 1 = \bar{a} \end{aligned} \quad (3.1)$$

■

3

Dieser Satz wirkt unscheinbar, ermöglicht jedoch erst die spätere Definition von Schaltvariablen, also mathematischen Objekten wie die Stellung eines technischen Schalters, der genau zwei Zustände annehmen kann.

Theorem 3.2.4 – Selbstinverse Abbildung

$$\bar{\bar{a}} = a$$

■

■ Beweis

In einem komplementären algebraischen Verband gilt $a \sqcup \bar{a} = 0$ oder $a \cdot \bar{a} = 0$ und $a \sqcap \bar{a} = 1$ oder $a + \bar{a} = 1$. Kommutativität ergibt $\bar{a} \cdot a = 0$ und $\bar{a} + a = 1$.

■

Die Abbildung eines invertierten Objektes auf sich selbst zurück ist der zweite Schritt auf dem Weg zur Schaltalgebra. Auch diese Gesetzmäßigkeit ist fundamental bei der Einführung einer Trägermenge, die lediglich mit zwei Zuständen auskommt. Sie entspricht im Wesentlichen der Multiplikation zweier negativer Zahlen in der aus der Schule bekannten Arithmetik.

Theorem 3.2.5 – Absorption des Komplements

In einer booleschen Algebra gilt:

$$a + \bar{a}b = a + b \quad (3.2)$$

$$a(\bar{a} + b) = ab \quad (3.3)$$

■

■ Beweis

Mit dem Distributivgesetz $a + bc = (a + b)(a + c)$ gilt $a + \bar{a} = (\bar{a} + \bar{a})(a + b) = (a + b)$. Mit $a\bar{a} = 0$ gilt $a(\bar{a} + a) = a\bar{a} + ab = ab$.

■

Dieser Satz ist die Grundlage der Minimierung logischer oder digitaler Schaltungen. Zusammen mit den beiden vorangegangenen Sätzen kann durch geschickte Negation eine logische Funktion so umgewandelt werden, dass komplementäre Paare im Sinne der Absorptionsregel auftreten. Das Streichen einer Variablen, ohne dass dies eine Folge auf die durchzuführende

Operation hat, führt direkt zum Einsparen von Schaltern. Mit Hilfe einer Ordnungsrelation in der booleschen Algebra kann schließlich der Aufwand einer Schaltung bestimmt werden:

Theorem 3.2.6 – Halbordnung in der booleschen Algebra

In einer booleschen Algebra gilt:

$$x \leq y \Leftrightarrow x\bar{y} = 0 \quad (3.4)$$

$$x \leq y \Leftrightarrow \bar{x} + y = 1 \quad (3.5)$$

■ Beweis

$x \leq y \Leftrightarrow x\bar{y} \leq y\bar{y} \Leftrightarrow x\bar{y} \leq 0 \Leftrightarrow x\bar{y} = 0$. Nach dem Dualitätsprinzip können die Multiplikation in $x\bar{y} = 0$ in eine Addition und die 0 in eine 1 umgewandelt werden. Es folgt $x \leq y \Leftrightarrow \bar{x} + y = 1$.

■

Der Nutzen der Absorption wird sofort klar, wenn es eine Möglichkeit gibt, boolesche Formeln durch Negation in kürzere oder einfacher zu reduzierende Gleichungen umzuwandeln. Daher sind die Sätze des englischen Mathematikers Augustus De Morgan (1806–1871, □ Abb. 3.6) von grundlegender Bedeutung für die Schaltungsminimierung. Die fundamentale Rolle De Morgans wird unterstrichen durch die Tatsache, dass er und George Boole heute allgemein als Begründer der formalen Logik gelten.

Theorem 3.2.7 – De Morgansches Gesetz

$$\overline{(a+b)} = \bar{a}\bar{b} \quad (3.6)$$

$$\overline{(ab)} = \bar{a} + \bar{b} \quad (3.7)$$



Sophia Elizabeth De Morgan,
Wikimedia Commons

□ Abb. 3.6 Augustus De Morgan

■ Beweis

Wenn $\overline{(a+b)} = \bar{a} \cdot \bar{b}$, gilt auch $\overline{(a+b)} \leq \bar{a} \cdot \bar{b}$. Erweitert man mit $(a+b)$, erhält man $\overline{(a+b)} \cdot (a+b) \leq \bar{a} \cdot \bar{b} \cdot (a+b)$, und daraus folgt $\bar{a} \cdot \bar{b} \cdot (a+b) = 0$. Da $\bar{a} \cdot a = 0$ und $\bar{a} \cdot b \leq 0$, ist der Ausdruck wahr. Wendet man das Dualitätsprinzip an, wird aus $\overline{(a+b)} = \bar{a} \cdot \bar{b}$ sofort $\overline{(a \cdot b)} = \bar{a} + \bar{b}$.

■

Um diese Formeln und Funktionen geht es im Folgenden. Haben wir einfache logische Gleichungen schon in den obigen Sätzen verwendet, sollen diese nun formal hergeleitet und beschrieben werden.

3.2.4 Ausdrücke und Funktionen

Rechenanlagen bilden eine Menge von Eingangsdaten auf eine Menge von Ausgangsdaten ab. Im einfachsten Fall ist diese Abbildung eine Funktion. Funktionen werden in Form von Formeln aufgeschrieben. Formal ist zwischen den Funktionen und den sie beschreibenden Formeln strikt zu trennen, da es unendlich viele Formeln gibt, die die gleiche Funktion beschreiben. Aus diesem Grund werden zunächst boolesche Formeln eingeführt. Diese Formeln bilden die Grundlage zur Beschreibung boolescher Funktionen. Die booleschen Funktionen können dann genutzt werden, um das gewünschte Ausgangsverhalten von logischen Schaltungen zu beschreiben.

Sei eine boolesche Algebra $(B, \cdot, +, 0, 1)$ gegeben, dann kann eine boolesche Funktion über n boolesche Variablen gebildet werden. Mit anderen Worten: $x_i \in B, i = 1, \dots, n$ seien boolesche Variablen, die alle Werte aus der Trägermenge B annehmen können; dann wird ein boolescher Ausdruck aus n x_1 bis x_n booleschen Variablen gebildet. Boolesche Ausdrücke werden dann rekursiv definiert.

Definition 3.2.6 – Boolesche Ausdrücke

Ist $(B, \cdot, +, 0, 1)$ eine boolesche Algebra, dann ist ein boolescher Ausdruck F gegeben durch:

- Die Elemente von F sind boolesche Formeln.
- Die Symbole oder Variablen x_1, \dots, x_n von F sind boolesche Formeln.
- Sind g und h boolesche Formeln, dann sind
 - $(g) \cdot (h)$
 - $(g) + (h)$
 - (\bar{g})
 ebenfalls boolesche Formeln.
- Eine Zeichenfolge (String) ist ein boolescher Ausdruck F genau dann, wenn die oben genannten Regeln unendlich oft auf die Zeichenfolge angewandt werden können.

Ein boolescher Ausdruck ist also eine Zeichenkette, die gewissen Regeln folgt. Diese Folgen von Zeichen haben zunächst einmal noch keine Bedeutung. Diese wird ihnen im Folgenden eingehaucht:

Definition 3.2.7 – Boolesche Funktion

Für boolesche Funktionen gilt:

1. Für jedes Element $b \in B$ ist die Funktion f , gegeben durch $f(x_1, \dots, x_n) = b \forall (x_1, \dots, x_n) \in B$ eine n -stellige boolesche Funktion.
2. Für jede Variable x_i aus $\{x_1, \dots, x_n\}$ ist $f(x_1, \dots, x_n) = x_i \forall (x_1, \dots, x_n) \in B$ eine boolesche Funktion. Diese Funktion heißt Projektion.
3. Wenn g und h n -stellige boolesche Funktionen sind, dann sind auch $g \cdot h$, $g + h$ und \bar{g} definiert durch:

$$(g \cdot h)(x_1, \dots, x_n) = g(x_1, \dots, x_n) \cdot h(x_1, \dots, x_n)$$

$$(g + h)(x_1, \dots, x_n) = g(x_1, \dots, x_n) + h(x_1, \dots, x_n)$$

$$(\bar{g})(x_1, \dots, x_n) = \overline{(g(x_1, \dots, x_n))}$$

für alle $(x_1, \dots, x_n) \in B$ n -stellige boolesche Formeln.

4. Nur Funktionen, die aus dem Anwenden der oben genannten Regeln abgeleitet werden, sind boolesche Funktionen.

Eine boolesche Formel ist also eine Folge von Zeichen oder besser eine Zeichenkette (engl. „string“). Die Elemente der Zeichenfolge heißen Literale (lat. „littera“, Buchstabe). In der mathematischen Logik sind diese atomar, also nicht weiter zerlegbar. Eine wichtige Rolle spielen die Disjunktions- oder Konjunktionsterme. Diese Moleküle der booleschen Formeln verknüpfen mehrere logische Variablen mit jeweils einer der logischen Operationen. Die aus Literalen aufgebauten Zeichenketten heißen auch Klausel oder Monom. Im Folgenden soll unter einem Literal ein Zeichen, dass eine Variable in einem booleschen Ausdruck darstellt, verstanden werden.

► Beispiel 3.2.11

Sei $B := \{0, a, b, 1\}$ und seien x_1, x_2 boolesche Variablen, dann sind $F = ((x_1 \cdot a) + (x_1 \cdot b)) \cdot x_2 = (x_1 \cdot (a + b)) \cdot x_2$ boolesche Formeln. ◀

Boolesche Ausdrücke sind Zeichenfolgen mit festgelegten Eigenschaften. Ihre Funktionen von n Variablen sind gegeben durch die Auswertung der booleschen Ausdrücke. Eine boolesche Funktion mit n Variablen erzeugt eine Wahrheitstabelle mit B^n Kombinationen. Jede dieser Verknüpfungen der Variablen erzeugt genau ein Bild in der Trägermenge B : $f(x) : B^n \mapsto B$. Die booleschen Funktionen $f(x) : \{0, 1\}^n \mapsto \{0, 1\}$, die über die Schaltalgebra definiert sind, nennt man Schaltfunktionen.

Wahrheitstabelle**3**

Eine alternative Darstellung oder Repräsentation eines booleschen Ausdrucks ist die Wahrheitstabelle. Eine Wahrheitstabelle besteht aus n Eingangsspalten und m Ausgangsspalten für jeweils n boolesche Eingangsvariablen und m boolesche Ausgangsvariablen. Da in der Wahrheitstabelle für alle möglichen Eingangskombinationen Ausgangswerte angegeben werden müssen, besteht die Tabelle aus 2^n Zeilen. Jede Wahrheitstabelle beschreibt eine boolesche Funktion eindeutig und ist damit äquivalent zu einer booleschen Formel. Beim Entwurf digitaler Schaltungen lassen sich Wahrheitstabellen direkt in boolesche Funktionen überführen und diese anschließend minimieren.

Da es viele boolesche Formeln gibt, die dieselbe logische Funktion beschreiben, ist streng zwischen den beiden Begriffen zu unterscheiden, insbesondere da nur $|B|^{2^n}$ boolesche Funktionen für n -stellige boolesche Formeln existieren. Diese Tatsache scheint auf den ersten Blick trivial, aber sie stellt den Entwickler digitaler Schaltungen vor ein Problem: Da jede logische Gleichung eine unterschiedliche Anzahl von Variablen enthalten kann, sollte ein Schaltungsentwickler die Formel aus der Menge aller gleichen Ausdrücke verwenden, die die wenigensten logischen Variablen und Verknüpfungen besitzt. Nach dem Aufstellen eines booleschen Ausdrucks kann sich der Informatiker zunächst nicht sicher sein, die minimale Formel gefunden zu haben. Daher muss eine Logikminimierung durchgeführt werden. In der Praxis heißt diese auch Logiksynthese. Die Programme zur Minimierung führen neben dieser Funktion noch weitere Berechnungen und Abbildungen durch und gelangen so zu einer effizienten Schaltung auf einer vorgegebenen Zieltechnologie.

Von einer Disjunktion spricht man, wenn Variablen mit der Operation $+$, von einer Konjunktion, wenn die Literale mit der Operation \cdot verknüpft werden. Jeder boolesche Ausdruck kann normiert werden. Eine solche normierte Darstellung wird Normalform genannt. Unterschieden werden zwei Arten von Normalformen: zum einen die disjunktive Normalform (DNF) und zum anderen die kanonische Normalform (KNF).

Eine einfache Näherung an die logischen Normalformen erfolgt über Beispiele. Verknüpft man Disjunktionen ($+$) von Konjunktionstermen (\cdot), entsteht die disjunktive Normalform.

► Beispiel 3.2.12

Eine boolesche Funktion in der disjunktiven Normalform besteht aus Disjunktionen ($+$) von Konjunktionstermen $x_1 \cdot x_2 \cdot x_3$: $f(x_1, x_2, x_3) = \bar{x}_2 \cdot x_3 + \bar{x}_1 \cdot x_2 + x_1 \cdot x_2 \cdot \bar{x}_3$. Dabei muss nicht jede Variable in jedem Term auftreten. ◀

Disjuktive Normalform

Natürlich lässt sich der Vorgang umdrehen: Die Verknüpfung von Disjunktionstermen (\cdot) mit Konjunktionen ($+$) führt zur konjunktiven Normalform.

Konjunktive Normalform

► Beispiel 3.2.13

Eine boolesche Funktion in der konjunktiven Normalform besteht aus Konjunktionen (\cdot) von Disjunktionstermen der Form $x_1 + x_2 + x_3$; $f(x_1, x_2, x_3) = (x_1 + \bar{x}_2) \cdot (\bar{x}_1 + x_3) \cdot (\bar{x}_1 + x_2)$. Dabei muss nicht jede Variable in jedem Term auftreten. ◀

Diese exemplarische Herleitung wird mit den folgenden Definitionen präzisiert. Zunächst muss der erste nicht atomare Baustein einer logischen Formel eingeführt werden:

Definition 3.2.8 – Vollkonjunktion

Eine Anzahl von logischen Variablen ist eine Vollkonjunktion, wenn alle Variablen einer booleschen Funktion konjunktiv sind, also durch ein UND verknüpft werden. Eine Vollkonjunktion wird auch „Minterm“ genannt.

Mithilfe des Bausteins Minterm wird nun die kanonische Normalform abgeleitet. Diese normierte Funktion hat die Eigenschaft, dass jede Variable oder deren Komplement in der Formel in jedem Minterm zu stehen hat. Diese Eigenschaft zu fordern ist auf den ersten Blick umständlich und aufwendig. Sie hat aber den Vorteil, dass boolesche Ausdrücke mit einem einheitlichen Vorgehen ohne nachzudenken gebildet oder von einem Programm erzeugt oder synthetisiert werden können. Eine solche Beschreibung ist kanonisch (lat. „canonicus“, regelmäßig), und die Funktion heißt dann disjunktive oder konjunktive kanonische Normalform. Die Terme $f(0, \dots, 0, 0)$, $f(0, \dots, 0, 1)$, \dots , $f(1, \dots, 1, 1)$ nennt man „Diskriminante“ (lat. „discriminare“ = unterscheiden) und die elementaren Produkte $\bar{x}_1 \dots \bar{x}_{n-1} \bar{x}_n$, $\bar{x}_1 \dots \bar{x}_{n-1} x_n$, \dots , $x_1 \dots x_{n-1} x_n$ die „Minterme“ bzw. „Maxterme“.

Definition 3.2.9 – Disjunktive kanonische Normalform (DKNF)

Die disjunktive kanonische Normalform ist ein boolescher Ausdruck, der alle Minterme, für die die boolesche Funktion $f(x_1, \dots, x_n) = 1$ ist, disjunktiv (ODER) verknüpft:

$$f(x_1, \dots, x_n) = \sum_j^m \prod_j^n (\neg x_{i,j}) \quad (3.8)$$

Aus Gründen der Symmetrie kann das Vorgehen auch umgedreht werden.

3

Definition 3.2.10 – Volldisjunktion

Eine Anzahl von logischen Variablen ist eine Volldisjunktion, wenn alle Variablen einer booleschen Funktion disjunktiv sind, also durch ein ODER verknüpft werden. Eine Volldisjunktion wird auch „Maxterm“ genannt.

Die Maxterme werden nun konjunktiv verknüpft, und es entsteht die konjunktive kanonische Normalform.

Definition 3.2.11 – Konjunktive kanonische Normalform (KKNF)

Die konjunktive kanonische Normalform ist ein boolescher Ausdruck, der alle Maxterme, für die die Funktion $f(x_1, \dots, x_n) = 1$ ist, konjunktiv (UND) verknüpft:

$$f(x_1, \dots, x_n) = \prod_j^m \sum_j^n (\neg)x_{i,j} \quad (3.9)$$

Shannon-Zerlegung

Eine fundamentale Eigenschaft boolescher Funktionen wird durch die Shannon-Zerlegung oder den Satz von Shannon beschrieben. Die Shannon-Zerlegung wird zwar Shannon zugeschrieben, war jedoch schon Boole bekannt. Ausgehend von der Schreibweise

$$f_{\bar{x}_1} = f|_{x_1=0} = f(0, x_2, \dots, x_n)$$

$$f_{x_1} = f|_{x_1=1} = f(1, x_2, \dots, x_n)$$

werden die Funktionen $f_{\bar{x}_1}(x_2, \dots, x_n)$ als der positive und $f_{x_1}(x_2, \dots, x_n)$ als der negative Kofaktor von f bezogen auf x_1 bezeichnet. Positive und negative Kofaktoren einer booleschen Funktion können für jede Variable x_1, \dots, x_n angegeben werden.

Theorem 3.2.8 – Shannon-Zerlegung

Wenn $f : B^n \mapsto B$ eine boolesche Funktion ist, dann gilt für die disjunktive Normalform

$$\begin{aligned} f(x_1, \dots, x_i, \dots, x_n) = & \bar{x}_i \cdot f(x_1, \dots, 0, \dots, x_n) + \\ & x_i \cdot f(x_1, \dots, 1, \dots, x_n) \end{aligned}$$

sowie analog für die konjunktive Normalform

$$\begin{aligned} f(x_1, \dots, x_i, \dots, x_n) = & [\bar{x}_i + f(x_1, \dots, 0, \dots, x_n)] \cdot \\ & [x_i + f(x_1, \dots, 1, \dots, x_n)] \end{aligned}$$

für alle $(x_1, \dots, x_n) \in B$.

Abkürzend lässt sich sich für die disjunktive Normalform

$$f(x) = \bar{x}_i \cdot f_{i,0}(x) + x_i \cdot f_{i,1}(x) \quad (3.10)$$

sowie für die konjunktive Normalform

$$f(x) = [\bar{x}_i + f_{i,0}(x)] \cdot [x_i + f_{i,1}(x)] \quad (3.11)$$

schreiben.

■ Beweis

Da in einer booleschen Algebra $a + \neg a = 1$ gilt, ist $f(x) = 1$, wenn $f_{i,0}(x) = 1$ oder $f_{i,1}(x) = 1$ ist. Wenn $x_i = 0$ ist, ist $f(x) = f_{i,0}(x)$, und wenn $x_i = 1$, ist $f_{i,1}(x)$. ■

Der Satz von Shannon wird in zahlreichen Algorithmen zur Optimierung, Synthese und Verifikation logischer Schaltungen eingesetzt. Er erlaubt es, komplexe boolesche Funktionen durch Zerlegung zu vereinfachen. Mithilfe rekursiver Algorithmen kann das Problem dann nach dem Prinzip „teile und herrsche“ immer weiter reduziert werden. Damit lassen sich große Formeln, wie sie beispielsweise zur Beschreibung von 64-Bit-Rechenwerken entstehen, durch Zerlegung minimieren.

Der Satz von Shannon kann dazu genutzt werden, aus jeder beliebigen logischen Formel eine kanonische Normalform abzuleiten:

$$\begin{aligned} f(x_1, \dots, x_{n-1}, x_n) = & f(0, \dots, 0, 0) \bar{x}_1 \dots \bar{x}_{n-1} \bar{x}_n + f(0, \dots, 0, 1) \\ & \bar{x}_1 \dots \bar{x}_{n-1} x_n + \dots + f(1, \dots, 1, 1) x_1 \dots x_{n-1} x_n \end{aligned}$$

Als Ergebnis erhält man wieder eine Normalform, in der in jedem Disjunktions- oder Konjunktionsterm alle Variablen der Funktion vorkommen.

► Beispiel 3.2.14

Sei $f(x_1, x_2) = x_1 x_2 + a \bar{x}_2$ eine boolesche Funktion auf der Trägermenge $B := \{0, a, b, 1\}$. Die disjunktive kanonische Normalform ist $f = f(0, 0) \cdot \bar{x}_1 \bar{x}_2 + f(0, 1) \cdot \bar{x}_1 x_2 + f(1, 0) \cdot x_1 \bar{x}_2 + f(1, 1) \cdot x_1 x_2$. Mit den Diskriminanten $f(0, 0) = 0 \cdot 0 + a \cdot 1 = a$, $f(0, 1) = 0 \cdot 1 + a \cdot 0 = 0$, $f(1, 0) = 1 \cdot 0 + a \cdot 1 = a$, $f(1, 1) = 1 \cdot 1 + a \cdot 0 = 1$ folgt für die disjunktive kanonische Normalform $f = a \cdot \bar{x}_1 \bar{x}_2 + 0 \cdot \bar{x}_1 x_2 + a \cdot x_1 \bar{x}_2 + 1 \cdot x_1 x_2$. ◀

Auch der umgekehrte Fall, die Ableitung der konjunktiv kanonischen Normalform mithilfe des Satzes von Shannon ist möglich:

► Beispiel 3.2.15

Sei $f(x_1, x_2) = x_1 x_2 + a \bar{x}_2$ eine boolesche Funktion auf der Trägermenge $B := \{0, a, b, 1\}$, dann ist $f = [a + x_1 + x_2][0 + \bar{x}_1 + \bar{x}_2][a + \bar{x}_1 + x_2][1 + \bar{x}_1 + \bar{x}_2]$ eine konjunktive kanonische Normalform von $f(x_1, x_2)$. Die Faktoren 0 und 1 sind die Deskriptoren und die Disjunktionen $(x_1 + x_2)$, $(x_1 + \bar{x}_2)$, $(\bar{x}_1 + x_2)$, $(\bar{x}_1 + \bar{x}_2)$ die Maxterme. ◀

Maximale Anzahl n -stelliger Funktionen

Offensichtlich kann jede boolesche Funktion mit jeweils einer charakteristischen kanonischen disjunktiven Normalform und einer charakteristischen konjunktiven kanonischen Normalform ausgedrückt werden. Angenommen, die Trägermenge einer booleschen Algebra hat n Elemente, dann hat eine kanonische Normalform 2^n verschiedene Diskriminanten, und jede dieser Diskriminanten kann n Werte annehmen. Dann ist die Anzahl aller möglichen booleschen Funktionen $n^{2^n} = |B|^{2^n}$. Eine 4-wertige boolesche Algebra hat demnach $4^{2^4} = 2^{32} = 4.294.967.296$ verschiedene boolesche Funktionen. Diese Eigenschaft erscheint unbedeutend, ermöglicht aber das systematische Auffinden und Bezeichnen oft zu verwendender Logikbausteine. Insbesondere kann auch die Anzahl der möglichen Realisierungen oder besser des Lösungsraumes einer Logikminimierung abgeschätzt werden.

3.2.5 Axiomatisierung der booleschen Algebra

Die auf algebraischen Strukturen aufbauende boolesche Algebra ermöglicht die Überführung der Logik in einen streng formalen mathematischen Formalismus und ein Schlussfolgern mittels Rechnen. Die Realisierung logischer Formeln mit Schaltungen erfordert allerdings weitere Vereinfachungen, um den Schaltungsaufwand zu reduzieren. Um die boolesche Algebra einfach zugänglich zu machen, ist es notwendig diese zunächst zu axiomatisieren. Die erste Axiomatisierung einer

booleschen Algebra gelang dem italienischen Mathematiker Giuseppe Peano (1858–1932) Ende des 19. Jahrhunderts.

Theorem 3.2.9 – Peano-Axiome

Erfüllt die algebraische Struktur $(B, \cdot, +, 0, 1)$ die folgenden Axiome, ist sie eine boolesche Algebra:

- Kommutativität: $(P1) \quad a \cdot b = b \cdot a$
 $(P1') \quad a + b = b + a$
- Assoziativität: $(P2) \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$
 $(P2') \quad (a + b) + c = a + (b + c)$
- Idempotenz: $(P3) \quad (a \cdot a) = a$
 $(P3') \quad (a + a) = a$
- Distributivität: $(P4) \quad a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
 $(P4') \quad a + (b \cdot c) = (a + b) \cdot (a + c)$
- Neutralität: $(P5) \quad a \cdot 1 = a$
 $(P5') \quad a + 0 = a$
- Extremität: $(P6) \quad a \cdot 0 = 0$
 $(P6') \quad a + 1 = 1$
- Doppelnegation: $(P7) \quad \bar{\bar{a}} = a$
- De Morgan: $(P8) \quad \overline{a \cdot b} = \bar{a} + \bar{b}$
 $(P8') \quad \overline{a + b} = \bar{a} \cdot \bar{b}$
- Komplementarität: $(P9) \quad a \cdot \bar{a} = 0$
 $(P9') \quad a + \bar{a} = 1$
- Dualität: $(P10) \quad \bar{0} = 1$
 $(P10') \quad \bar{1} = 0$
- Absorption: $(P11) \quad a + (a \cdot b) = a$
 $(P11') \quad a \cdot (a + b) = a$

■ Beweis

Der Beweis ergibt sich direkt aus der Herleitung der booleschen Algebra aus einem algebraischen Verband und den Sätzen aus ▶ Abschn. 3.2.2. ■

Ein einfacherer Axiomensatz sind die heute hauptsächlich genutzten Axiome des amerikanischen Mathematikers Edward Huntington (1874–1952).

Definition 3.2.12 – Boolesche Algebra nach Huntington

Eine boolesche Algebra ist eine algebraische Struktur $(B, +, \cdot, 0, 1)$, die die folgenden Axiome erfüllt:

- Kommutativität: (H1) $a + b = b + a$
(H1') $a \cdot b = b \cdot a$
- Distributivität: (H2) $a + (b \cdot c) = (a + b) \cdot (a + c)$
(H2') $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- Identität: (H3) $a + 0 = a$
(H3') $a \cdot 1 = a$
- Komplementarität: (H4) $a + \bar{a} = 1$
(H4') $a \cdot \bar{a} = 0$

Zu jedem Element $a \in B$ existiert also ein Element $\bar{a} \in B$. An dieser Stelle sei daran erinnert, dass 0 die größte untere Schranke und 1 die kleinste obere Schranke der Trägermenge B sind.

3.3 Schaltalgebra und Schaltfunktionen

3.3.1 Modellierung von Telefonnetzen

Hatte schon die Telegrafie dafür gesorgt, dass die Welt zusammenrückte, explodierte mit der Erfindung des Telefons der Bedarf an komplexen Leitungsnetzen. Neben der reinen Sprachübertragung mussten die einzelnen Teilnehmer auch verbunden werden. Dies geschah zunächst einmal manuell mithilfe von elektrischen Steckbrettern, den Switchboards. Mit zunehmender Anzahl an Teilnehmern musste die Vermittlung automatisiert werden. Den Nachrichtentechnikern stellte sich die Frage, wie Telefonapparate effizient und automatisch verbunden werden konnten. Um diese Fragestellung zu lösen, entwickelte Claude E. Shannon (1916–2001) ein schaltalgebraisches System. Grundlage seiner Masterarbeit war die Annahme, dass Telefone und deren Anschlüsse mit Relais verbunden sind. Ein solches Bauelement ist ein stromgesteuerter, elektromagneti-

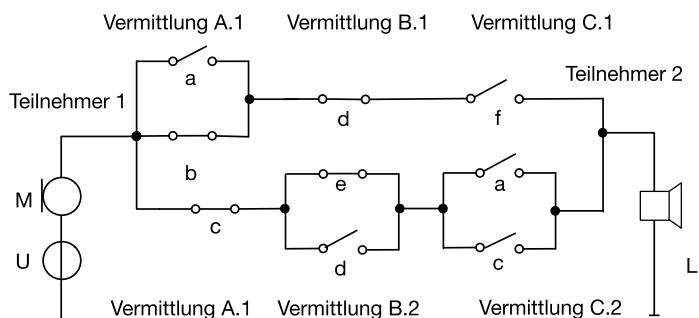
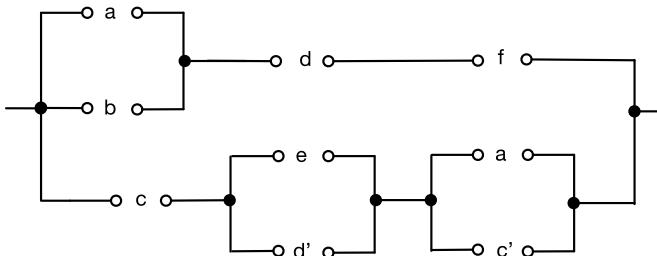


Abb. 3.7 Telefonschaltnetz



■ Abb. 3.8 Telefonschaltnetz (vereinfacht)

scher Schalter. Shannon modellierte in seiner Arbeit eine Kaskade von Vermittlungsschaltern. Jedes Relais ließ sich mit einer Variablen in einer booleschen Funktion beschreiben. Da die elektromagnetischen Schalter entweder geschlossen oder geöffnet sind, kann eine boolesche Algebra auf zwei Zustände reduziert werden. Eine boolesche Algebra mit der Trägermenge $B := \{0, 1\}$ heißt Schaltalgebra. Diese erlaubt eine einfache Manipulation logischer Schaltfunktionen und ermöglicht so die kostenoptimale Konstruktion von Telefonnetzen und Rechenanlagen.

Schaltalgebra

Die ■ Abb. 3.7 zeigt eine einfache Variante der Problemstellung in den frühen Telefonnetzen. Zwei Telefonapparate sind über mehrere Leitungen und Vermittlungsstellen erreichbar. Rechts im Bild wird der Schaltkreis über eine Spannungsquelle mit elektrischer Energie versorgt. Über ein Mikrofon kann ein Teilnehmer mit einem weiteren kommunizieren, wenn ein Weg durch das Netz freigeschaltet ist. In einem echten Telefonnetz ist die Verbindung bidirektional. Dieses technische Detail wurde der Einfachheit halber weggelassen. Auch wird in einem realen Telefonnetz die elektrische Energie von der Vermittlungsstelle bereitgestellt. Die Frage, die sich Claude Shannon stellte, ist nun, welches ist die beste Schalterstellung, um beide Teilnehmer zu verbinden. Shannon hat zunächst das Problem abstrahiert und die Signalquelle oder das Mikrofon und den Lautsprecher als Signalsenke weggelassen. Die Schalter werden ebenfalls nur über Variablen modelliert. ■ Abb. 3.8 zeigt eine vereinfachte Darstellung.

Um nun mit mathematischen Methoden eine zu finden, nutzte Shannon eine boolesche Algebra mit der einfachen Trägermenge $B := \{0, 1\}$. Der Wert 1 einer Variablen stand dabei für einen geschlossenen, das Zeichen 0 für einen geöffneten Schalter. Da Shannon eine algebraische Lösung anstrebte, benötigte er noch die Operationen + und . Hinzukam der Begriff des Inverters oder des „NICHT“, um auszudrücken, dass ein Schalter geöffnet ist, wenn die entsprechende Variable 1 ist.

Dies wurde mit einem Strich neben der Variablen gekennzeichnet. Also a' ist geschlossen, wenn $a = 0$ ist, und geöffnet, wenn $a = 1$ ist (Abb. 3.9).

Shannon hat die Schaltalgebra unter der Annahme von Widerstandsnetzwerken abgeleitet, indem er davon ausgeht, dass eine Variable für einen entsprechenden elektrischen Widerstand steht. Dabei geht er von 2 diskreten Werten aus. Ist die Variable 1, ist die Annahme, der Widerstand ist unendlich, und hat die Variable den Zustand 0, ist der Widerstand auch null und der Schaltkreis geschlossen. Zwei in Reihe geschaltete Relais realisieren daher die Multiplikation (\cdot), besser gesagt ein UND, und zwei parallel angeordnete Schalter das Plus (+) oder das ODER. Offensichtlich lassen sich geschachtelte Schalterstellungen mit logischen Begriffen wie UND (\wedge) und ODER (\vee) beschreiben. Zwar hatte Shannon die Schaltalgebra aus der Netzwerktheorie abgeleitet, in seiner Arbeit beschreibt er aber bereits die Analogie zur booleschen Algebra, basierend auf Huntingtons Axiomen. Die Shannon-Zerlegung wird mithilfe einer Taylor-Entwicklung für zwei Schaltzustände hergeleitet und direkt zur Minimierung von Schaltfunktionen genutzt. Formal ergibt sich dann eine boolesche Algebra mit der Trägermenge $B := \{0, 1\}$.

$$\begin{array}{l} \text{3} \\ \text{---} \\ \circ a \circ \circ b \circ = \circ a \cdot b \circ \\ \text{---} \\ \begin{array}{c} \circ b \\ \square a \end{array} = \circ a + b \circ \end{array}$$

Abb. 3.9 Schaltalgebra

3.3.2 Formale Definition und Eigenschaften

In einem System aus Schaltern ist jeder Schalter eine Variable und kann die 2 Zustände $Z := \{0, 1\}$ annehmen. Damit lässt sich eine Algebra definieren, die Schaltalgebra heißt.

Definition 3.3.1 – Schaltalgebra

Eine algebraische Struktur $B := (\{0, 1\}, \cdot, +, 0, 1)$ wird Schaltalgebra genannt.

Theorem 3.3.1 – Die Schaltalgebra ist eine boolesche Algebra

Die Schaltalgebra ist eine boolesche Algebra mit der Trägermenge $T := \{0, 1\}$ und den Operationen $\cdot = \text{UND}$ und $+ = \text{ODER}$.

■ Beweis

Sei $B := (\{0, 1\}, \cdot, +, 0, 1)$ eine Mengenalgebra $B := (\{0, 1\}, \cap, \cup, 0, 1)$. Ausgehend von der auf den Vektoren einer booleschen Algebra definierten Ordnungsrelation gilt, dass die leere Menge

dem Nullelement entspricht ($\emptyset = 0$) und eine Teilmenge mit 2 Elementen von 2^M dem Einselement ($a, b = 1$). Es folgt daher:

\cdot (UND, \wedge)			
a	b	$a \cap b$	c
0	0	0	0
0	1	0	$\inf\{\emptyset\} = 0$
1	0	0	$\inf\{\emptyset\} = 0$
1	1	1	1

$+$ (ODER, \vee)			
a	b	$a \cup b$	c
0	0	0	$\sup\{0\} = 0$
0	1	{0,1}	$\sup\{0,1\} = 1$
1	0	{0,1}	$\sup\{0,1\} = 1$
1	1	{0,1}	$\sup\{0,1\} = 1$

Wenn eine Mengenalgebra die Axiome einer booleschen Algebra erfüllt, dann ist die Schaltalgebra ebenfalls eine boolesche Algebra. ■

Formal entsprechen die Schnittmenge der Mengenalgebra dem UND in der Schaltalgebra und die Vereinigungsmenge dem ODER. Die Mengen- und die Schaltalgebra haben die gleiche Gestalt. In der Mathematik sagt man, sie sind isomorph. Offensichtlich können die Operationen \cdot und $+$ beliebig durch andere Operationen ersetzt werden: \cdot entspricht \cap sowie \wedge und $+$ entspricht \cup und \vee . Um diesem Umstand Rechnung zu tragen, findet man in der Literatur häufig die Konvention, die aus der klassischen Algebra gewohnten Operationen \cdot und $+$ durch die Operationen \sqcap und \sqcup zu ersetzen. Um die gewohnte Intuition beim Rechnen und den Vorteil abkürzender Schreibweisen bei der Multiplikation zu nutzen, wird im Folgenden mit \cdot und $+$ geschrieben. Also $\cdot = \wedge$ oder UND und $+ = \vee$ oder ODER. Die Aussage, dass jede boolesche Algebra zu einer Mengenalgebra isomorph ist, ist allgemeingültig und wurde von dem amerikanischen Mathematiker Marshall H. Stone (1903–1989) formuliert und bewiesen. Da die Schaltalgebra eine boolesche Algebra ist, lassen sich alle Definitionen und Sätze, die bereits für die boolesche Algebra eingeführt wurden, auch in der Schaltalgebra anwenden. Insbesondere das Komplement $a \cdot \bar{a} = 0$ und die Sätze von De Morgan sind für die optimale Konstruktion logischer Schaltungen unverzichtbar. Moderne Logiksynthesewerkzeuge optimieren die Logik und erzeugen die Schaltnetze direkt aus Hochsprachen. Sie sind ohne das mathematische Fundament der algebraischen Strukturen und der booleschen Algebra undenkbar. Zusammen mit der Axiomatisierung nach Huntington und den de Morganschen Gesetzen ergibt sich ein mächtiges Werkzeug zur Beschreibung von beliebigen Schalterkonfigurationen. Diese können algebraisch oder heuristisch vereinfacht werden. So lassen sich äquivalente Schaltungen mit weniger Schaltern und damit geringeren Kosten konstruieren.

Die Grundoperationen der Schaltalgebra UND und ODER, so wie sie eingeführt wurden, sind binäre boolesche Ausdrücke, d. h., zwei Eingänge werden auf einen Ausgang abgebildet. In

Äquivalenz der Mengen- und der Schaltalgebra

der zweielementigen Schaltalgebra existieren aber $|B|^{2^n} = 2^2 = 2^4 = 16$ verschiedene binäre boolesche Ausdrücke:

Kontradiktion		$F_0 = 0$
Konjunktion	UND	$F_1 = a \cdot b$
Inhibition		$F_2 = \bar{a} \cdot b$ oder $F_3 = a \cdot \bar{b}$
Identität		$\forall b \wedge a = 1 : F_4 = a$ oder $\forall a \wedge b = 1 : F_5 = b$
Antivalenz	XOR	$F_6 = a\bar{b} + b\bar{a}$
Disjunktion	ODER	$F_7 = a + b$
Peirce-Funktion	NOR	$F_8 = \overline{a + b}$
Äquivalenz	XNOR	$F_9 = \overline{a\bar{b} + b\bar{a}}$
Negation	NICHT	$F_{10} = \bar{a}$ oder $F_{11} = \bar{b}$
Implikation		$F_{12} = \bar{a} + b$ oder $F_{13} = a + \bar{b}$
Sheffer-Funktion	NAND	$F_{14} = \overline{a \cdot b}$
Tautologie		$F_{15} = 1$

Offensichtlich gibt es 2 Funktionen, die eine Konstante erzeugen, F_0 und F_{15} , 4 unäre Funktionen mit 1 Variablen F_2 , F_3 und F_{10} , F_{11} und 10 binäre Operationen F_1 , F_4 , F_5 , F_6 , F_7 , F_8 , F_9 , F_{12} , F_{13} und F_{14} . Da die booleschen Ausdrücke unterschiedlich leicht zu implementieren sind, spielen nicht alle bei der technischen Realisierung die gleiche Rolle. Während z. B. die Tautologie und die Kontradiktion trivial zu realisieren sind, indem der Ausgang im ersten Fall mit der Spannungsversorgung und im zweiten mit der Masse verbunden wird, sind die Inhibition und Implikation in moderner Halbleitertechnik nur schwer aufzubauen.

Logische Funktionen in Rechnern wie ein Telefonnetz zu realisieren hat einen gewichtigen Nachteil: Die Pegel auf den Leitungen werden nicht verstärkt. In landgebundenen Telefonnetzen kann dies durch ausreichend hohe Spannungen kompensiert werden, zumal über die Kabel analoge Signale übertragen werden. In einem Computer finden aber häufig Zustandswechsel statt, sodass hohe elektrische Potenziale beim Schalten zu großen Energieverlusten führen. Aus diesem Grund sollten die logischen Funktionen in einem Rechner nicht durch Schalter realisiert werden, sondern besser durch Verstärker. In jeder Logikstufe wird das Signal verstärkt. Dadurch werden die digitalen Maschinen robust gegen Fehler oder Signaleinstreuungen. Logische Schalter, die aus Verstärkern bestehen, heißen Gatter. Diese Bausteine können in moderner Halbleitertechnik als Peirce- oder Sheffer-Funktion einfach realisiert werden. Mithilfe der folgenden zwei Sätze wird formal gezeigt, dass sich jede Schaltfunktion ausschließlich durch verstärkende NOR- oder NAND-Gatter darstellen lässt.

Theorem 3.3.2 – NOR

Jede Schaltfunktion kann mit der Peirce-Funktion NOR dargestellt werden.

■ **Beweis**

Auf jede Schaltfunktion kann die selbstinverse Abbildung angewendet werden: $f(x_1, \dots, x_n) = \overline{\sum_j^n \prod_j^n (\neg)x_{i,j}}$. Unter Anwendung des Satzes von De Morgan folgt: $f(x_1, \dots, x_n) = \overline{\sum_j^m \sum_j^n (\neg)x_{i,j}}$. ■

Theorem 3.3.3 – NAND

Jede Schaltfunktion kann ausschließlich mit der Sheffer-Funktion (NAND) dargestellt werden.

■ **Beweis**

Anwendung des Satzes von De Morgan auf die komplementäre Schaltfunktion $\overline{f(x_1, \dots, x_n)}$ ergibt: $\overline{f(x_1, \dots, x_n)} = \overline{\sum_j^m \prod_j^n (\neg)x_{i,j}} = \prod_j^m \overline{\prod_j^n (\neg)x_{i,j}}$. Wird dieser Ausdruck negiert, folgt: $f(x_1, \dots, x_n) = \overline{\prod_j^m \overline{\prod_j^n (\neg)x_{i,j}}}$. ■

3.3.3 Anwendung von Schaltfunktionen

Mithilfe der Grundoperationen der booleschen Algebra lassen sich nicht nur logische Ausdrücke formulieren, sondern auch komplexe mehrwertige Funktionen. In der booleschen Algebra wurden bereits boolesche Ausdrücke und boolesche Funktionen eingeführt. Diese können in der Schaltalgebra übernommen werden.

► **Beispiel 3.3.1 – Subtraktion im 9-Komplement**

Die Addition jeder einzelnen Stelle einer Zahl im binären Stellenwertsystem besteht aus zwei 3-wertigen booleschen Funktionen für jede Stelle, dem Ergebnis der Addition, der Summe $s_n(a, b, c_{n-1})$ und dem Übertrag (engl. „carry“) für die nächste Stelle $c_n(a, b, c_{n-1})$ auf $B := \{0, 1\}$. ◀

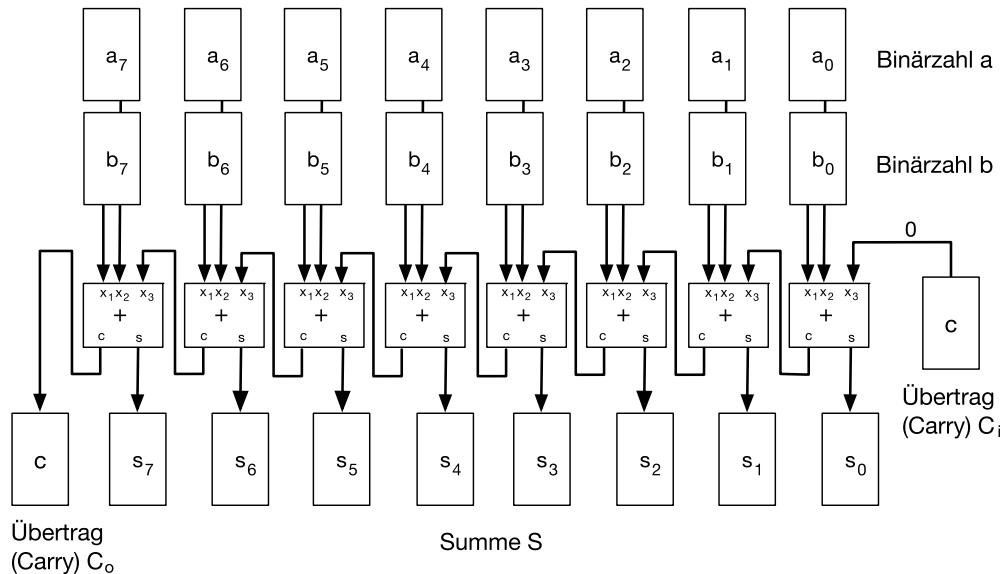


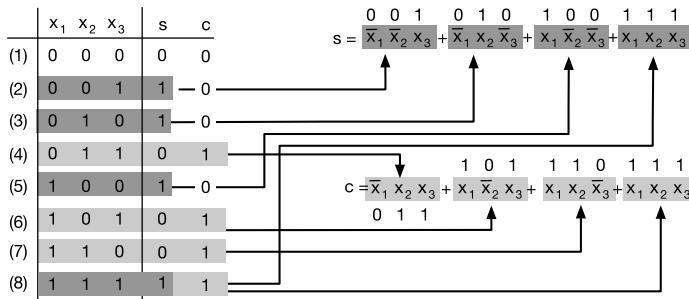
Abb. 3.10 Schema einer Addition zweier 8-Bit-Zahlen

Die Abb. 3.10 zeigt eine logische Schaltung, mit der zwei 8-stellige binäre Zahlen (a, b) addiert werden. Die Addition ist dabei abstrakt mit einem $+$ gekennzeichnet und repräsentiert eine wie auch immer gestaltete logische Schaltung, die im Folgenden Addierwerk heißt. Das Ergebnis des Addierwerks sind zum einen die Summe s der beiden Zahlen und zum anderen der Übertrag c . Das Addierwerk summiert jede Stelle von a und $b(x_1, x_2)$ einzeln und berücksichtigt dabei den Übertrag der vorhergehenden Stelle (x_3). Es erzeugt dann die Summe s und den Übertrag c für die nächste Stelle. Das Addierwerk hat also die Eingänge x_1, x_2 und x_3 , wobei x_3 meist als c_{in} bezeichnet wird, und die Ausgänge s und c oder c_{out} . Das Addierwerk, auch Volladdierer genannt, kann mit zwei Schaltfunktionen realisiert werden: eine Schaltfunktion für den Übertrag und eine für die Summe (Abb. 3.11).

Es ist einfach, für alle Eingangskombinationen des Addierwerks eine Wahrheitstabelle aufzustellen. Mithilfe des aus der Schule bekannten Algorithmus zur Addition von Zahlen kann dann bestimmt werden, in welcher Zeile der Wahrheitstabelle s bzw. c wahr oder 1 ist. Für s sind dies die Zeilen (2),(3),(5) und (8) und für den Übertrag c die Zeilen (4),(6),(7) und (8). Ein boolescher Ausdruck für c ist schnell gefunden; es ist jede Kombination zweier Variablen zu betrachten. Der Fall, dass alle möglichen Zustände der Schaltfunktion $c = 1$ sind, ist

a	a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀	b ₁₀	a	2	2	6
b	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀		b	0	9	7
c	c ₈	c ₇	c ₆	c ₅	c ₄	c ₃	c ₂	c ₁	c ₀	c	1	1	
s	s ₈	s ₇	s ₆	s ₅	s ₄	s ₃	s ₂	s ₁	s ₀	s	3	2	3

a	1	1	1	0	0	0	1	0	b ₂	a	1	1	1
b	0	1	1	0	0	0	0	1		b	0	1	1
c	1	1	1	0	0	0	0	0		c	1	1	1
s	1	0	1	0	0	0	0	1		s	1	0	1

Abb. 3.11 Addition zweier Zahlen im Dezimal- und im Binärsystem**Abb. 3.12** Ableitung der disjunktiven kanonischen Normalform

durch die ODER Verknüpfung aller Terme abgedeckt, da es ausreichend ist, wenn nur ein Term 1 wird (**Abb. 3.12**).

Dazu muss nur für jede Zeile, in der die logische Funktion 1 wird, ein entsprechender Konjunktionsterm, eine Vollkonjugation oder Minterm gebildet werden. Variablen, die den Wert 0 annehmen, sind negiert. Die Minterme werden dann mithilfe von Disjunktionen zu einer disjunktiven kanonischen Normalform (DKNF).

DKNF

► Beispiel 3.3.2

Die disjunktive kanonische Normalform lässt sich direkt in der Wertetabelle ablesen:

$$s(x_1, x_2, x_3) = \overline{x}_1 \overline{x}_2 x_3 + \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 x_3 \quad (3.12)$$



► Beispiel 3.3.3

Der Übertrag lässt sich auch direkt als UND aus der Tabelle ableiten. Es kann also auf die Bildung einer kanonischen Normalform verzichtet werden.

$$c(x_1, x_2, x_3) = x_2 x_3 + x_1 x_3 + x_1 x_2 + x_1 x_2 x_3 \quad (3.13)$$

Die disjunktive kanonische Normalform kann alternativ berechnet werden: Auf die Schaltfunktion wird einfach die Shannon-Zerlegung angewandt: $f(0, 0, 0) = 0, f(0, 0, 1) = 0, f(0, 1, 0) = 0, f(0, 1, 1) = 1, f(1, 0, 0) = 0, f(1, 0, 1) = 1, f(1, 1, 0) = 1, f(1, 1, 1) = 1$. Es sei daran erinnert, dass $f(0, 0, 0) = c(0, 0, 0) = x_1 x_2 x + x_1 x_3 x + x_2 x_3 x = 00 + 00 + 00 = 0$ und $f(1, 1, 0) = c(1, 1, 0) = 11 + 10 + 10 = 1$ ist. Die disjunktive kanonische Normalform ist:

$$c(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \quad (3.14)$$



3.3.4 Minimierung von Schaltfunktionen

Die Ableitung der Schaltfunktionen für einen binären Addierer zeigt, dass eine Schaltfunktion mit booleschen Formeln unterschiedlicher Länge dargestellt werden kann. Die disjunktive kanonische oder die konjunktive kanonische Normalform ist bedingt durch das Auftreten jeder Variablen in jedem Min- oder Maxterm nicht minimal in Bezug auf die Anzahl der auftretenden Literale.

Offensichtlich kann eine kanonische Normalform minimiert werden, insofern, dass eine Schaltfunktion mit weniger Literalen existiert. Minimale Schaltfunktionen sind für die Implementierung von Rechenanlagen von großer Bedeutung, da das Einsparen von Konjunktionen und Disjunktionen den Schaltungsaufwand reduziert. Die Minimierung einer Schaltfunktion beruht auf dem Umstand, dass in einer booleschen Algebra $x + \bar{x} = 1$ ist, d. h., es gilt $x_1 x_2 + x_1 \bar{x}_2 = x_1$. In einer disjunktiven kanonischen Normalform sind also Paare von Variable und Negation dieser in den Konjunktionen zu suchen. Mithilfe algebraischer Umformungen lassen sich dann entsprechende Dopplungen, die $x + \bar{x} = 1$ erfüllen, finden. Allerdings ist es rechnerisch sehr aufwendig, und die notwendigen Umstellungen können leicht übersehen werden.

► Beispiel 3.3.4

Für die ODER-Funktion kann direkt die disjunktive kanonische Normalform aus der Wahrheitstabelle abgelesen werden:

$$y(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2 + x_1 x_2 \quad (3.15)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

ODER-Funktion

Ausklammern ergibt:

$$y(x_1, x_2) = \bar{x}_1 x_2 + x_1 (\bar{x}_2 + x_2) = \bar{x}_1 x_2 + x_1 \quad (3.16)$$

Da für x_1 die Absorptionsregel gilt, $x_1 = x_1 + x_1 x_2$, kann y erweitert werden:

$$y(x_1, x_2) = \bar{x}_1 x_2 + x_1 + x_1 x_2 = x_2 (\bar{x}_1 + x_1) + x_1 = x_2 + x_1 \quad (3.17)$$



Das Problem der Minimierung von Schaltfunktionen lässt sich mit einem Graphen veranschaulichen. Ein solcher aus Graph stellt alle möglichen Belegungen der Variablen aller denkbaren booleschen Funktion dar.

Die Schaltalgebra ist eine boolesche Algebra mit der Trägermenge $B := \{0, 1\}$ und den Operationen $\{+, \cdot\}$ oder $\{\text{ODER}, \text{UND}\}$. Die Variablen eines booleschen Raumes können daher zwei Zustände annehmen. Der boolesche Raum der Schaltalgebra lässt sich aufgrund der geringen Anzahl von Elementen der Trägermenge mithilfe eines gerichteten Graphen darstellen. Die Knoten repräsentieren dabei immer genau eine mögliche Belegung der einzelnen Variablen. Bei $n = 4$ Variablen sind dies 2^4 Knoten, bei $n = 42$ Variablen 2^{42} Knoten. Zwischen zwei Knoten existiert eine Kante, genau dann, wenn sich die Belegung der Variablen in nur einer der Variablen unterscheidet. Die Anzahl unterschiedlich belegter Variablen wurde nach einem Pionier der algebraischen Codierungstheorie, dem amerikanischen Mathematiker Richard Wesley Hamming (1915–1998) Hamming-Abstand oder Hamming-Distanz genannt.

Boolescher Raum

Definition 3.3.2 – Hamming-Distanz

Gegeben sei ein Alphabet \sum : Sind $x, y \in \sum$ zwei n Zeichen lange Worte, dann ist die Hamming-Distanz oder der Hamming-Abstand:

$$\delta(x, y) := |\forall (i, j) \in \mathbb{N} : x_i \neq y_i| \quad (3.18)$$

► Beispiel 3.3.5 – Hamming-Distanz

Seien $x = 1001$ und $y = 0101$ boolesche Terme. Die Hamming-Distanz ist dann 2, da sich die beiden Terme in 2 Variablen unterscheiden: $uu01$. ◀

Im Folgenden werden die Graphen einfacher boolescher Räume diskutiert.

Definition 3.3.3 – Boolescher Raum der Trägermenge $B := \{0, 1\}$

Ein boolescher Raum der Trägermenge $B := \{0, 1\}$ ist ein Graph $G(V, E)$, in dem Knoten alle möglichen Kombinationen der Schaltvariablen darstellen und die Kanten Knoten mit Hamming-Abstand $\delta(v_i, v_j) = 1$ verbinden:

$$V := \{v \in B^n = \{0, 1\}^n\} \quad (3.19)$$

$$E := \{e \in \{0, 1\}^n \times \{0, 1\}^n \mid \forall i, j \in \mathbb{N} : \delta(v_i, v_j) = 1\} \quad (3.20)$$

Für $n = 0$ besteht der boolesche Raum der Schaltalgebra aus lediglich 1 Knoten. Die Dimension $|B^0| = 1$. Da eine boolesche Variable mindestens 2 Zustände annehmen kann, dies aber in einem 1-dimensionalen Raum nicht darzustellen ist, repräsentiert der B^0 keine boolesche Variable, und Funktionen der Schaltalgebra sind im B^0 nicht darstellbar.

Ein boolescher Raum der Dimension B^1 repräsentiert in der Schaltalgebra mit der Trägermenge $B := \{0, 1\}$ zwei mögliche Zustände, entspricht somit einer logischen Variablen. Jeder Knoten des Graphen ist gleich einem Zustand dieser Variablen. Da die Hamming-Distanz in dem Fall 1 ist, kann zwischen den beiden Knoten eine Kante eingezeichnet werden. Es entsteht eine Hantel (Abb. 3.13).

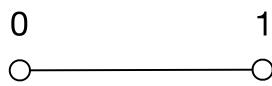


Abb. 3.13 B^1

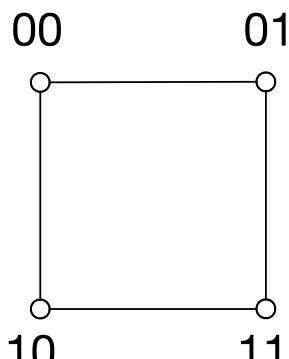


Abb. 3.14 B^2

Im B^2 sind 2 Variablen zu betrachten. Daher sind 4 Zustände darstellbar. Aus der Hantel wird ein Quadrat. Jeder Knoten des Raumes ist mit seinem Nachbarn verbunden und lediglich die Hamming-Distanz der zwei diagonalen Elemente ist größer als 1 (Abb. 3.14).

Beim B^3 verdoppelt sich wieder die Anzahl der Zustände. Aus 4 Knoten des Quadrates werden 8 Knoten eines Würfels.

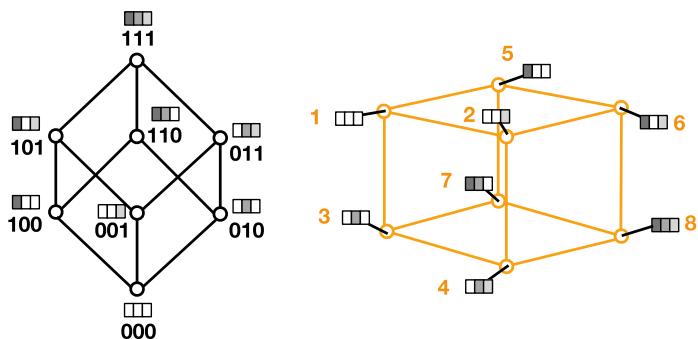


Abb. 3.15 Herleitung des B^3

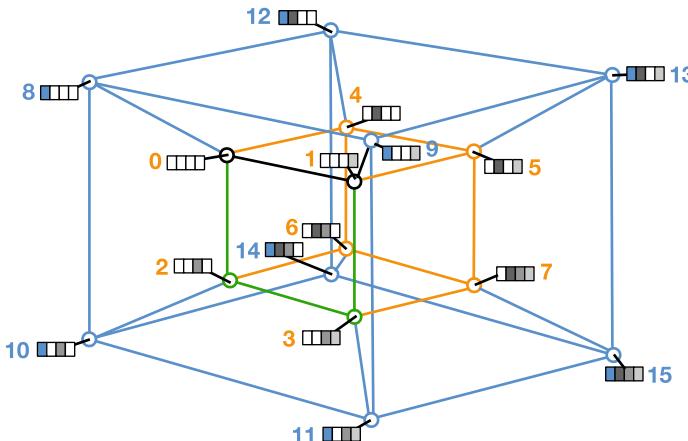


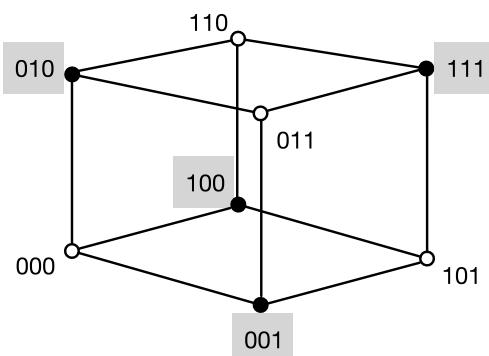
Abb. 3.16 B^4

Und wieder fehlen diagonale Kanten, da gegenüberliegende Knoten des Graphen eine Hamming-Distanz größer als 1 haben (Abb. 3.15).

Gerade noch als Graph darstellbar ist der B^4 (Abb. 3.16). Wieder verdoppelt sich die Anzahl der Knoten. Grafisch folgt ein Würfel in einem Würfel. Zwar kann man sich denken, wie es weitergeht – aus 2 verknüpften Würfeln im B^4 werden 4 ineinander geschachtelte Würfel im B^5 und 8 Würfel im B^6 . Wie eine inverse Matrjoschka oder Matroschka sind die verknüpften Würfel ineinander verschachtelt. Wird die Darstellung umgedreht, d. h., steht für den B^3 ein äußerer Würfel, ist der entstehende Graph höherdimensionaler Räume fast dem Bild unendlich ineinander geschachtelter Puppen gleich, nur dass in jedem Schritt eine Verdopplung der geschachtelten Würfel entsteht. Es lässt sich erahnen, wie unübersichtlich und komplex boolesche Räume von 64- oder 128-Bit-Registern heutiger Computer sind und wie aufwendig eine Minimierung von Schaltfunktionen wird.

Um eine Schaltfunktion in einem booleschen Raum darzustellen, wird ein Graph der entsprechenden Dimension gebildet, eine Funktion mit 3 Variablen wird im B^3 und eine Schaltfunktion mit n Variablen im B^n visualisiert. Jeder Min- oder Maxterm, je nachdem, welche Darstellungsform der Schaltfunktion gewählt wird, wird dann im Graphen dargestellt. Dafür kann der entsprechende Knoten markiert und schwarz gefärbt werden. Die Abb. 3.17 zeigt dies am Beispiel eines Addierers für eine Binärstelle. Oder in anderen Worten: Im Graphen des booleschen Raumes werden die Knoten markiert, für die die Selektoren nach der Shannon-Zerlegung 1 sind.

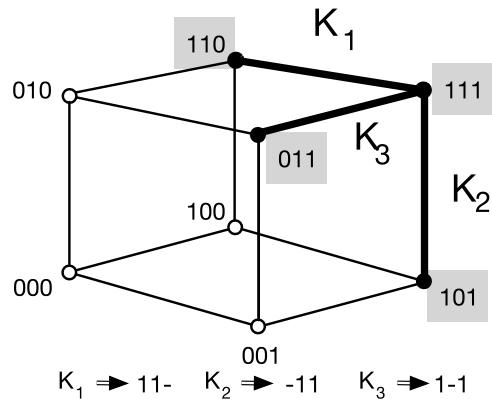
Schaltfunktion der Summe:



3

$$S = \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3$$

Schaltfunktion des Übertrages:



$$C = x_1 x_2 + x_2 x_3 + x_1 x_3$$

$$K_1 \Rightarrow 11- \quad K_2 \Rightarrow -11 \quad K_3 \Rightarrow 1-1$$

Abb. 3.17 Addierwerk für eine Stelle als Schaltfunktion im B^3

$$c(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3$$

Boolesche Räume und somit die Schaltfunktionen in diesen lassen sich unterschiedlich darstellen. Die Knoten können dabei mit den Selektoren der Shannon-Zerlegung, mit den Mintermen der disjunktiven kanonischen Normalform oder einfach einer Bitrepräsentation eines Minterms beschriftet sein. In der Literatur findet man oft die Darstellung mit Bitmustern.

3.3.4.1 Grafische Minimierung

Der Vorteil der booleschen Räume ist leicht zu erkennen. Durch, dass alle Knoten mit der Hamming-Distanz 1 benachbart sind, markieren die Kanten des Graphen $x + \bar{x} = 1$ Kandidaten zur Minimierung. Sind also Min- oder Maxterme einer Funktion im booleschen Raum mit einer Kante verknüpft, kann die entsprechende Variable gemäß der Absorptionsgesetze der booleschen Algebra eliminiert werden. Benachbarte Knoten werden zusammengefasst. In der Abb. 3.17 ist dies durch eine markierte Kante gezeigt. Diese Markierung bedeutet, dass die Schaltfunktion 1 wird, egal ob die entsprechende Variable 0 oder 1 ist. Jede markierte Kante stellt also einen neuen Minterm dar, der die beiden verbundenen Minterme ersetzt. Für den Übertrag des Volladdierers ergibt sich somit als minimale Schaltfunktion

$$c = x_1 x_2 + x_1 x_3 + x_2 x_3 \quad (3.21)$$

Die Schaltfunktion für die Summe lässt sich nicht mehr minimieren.

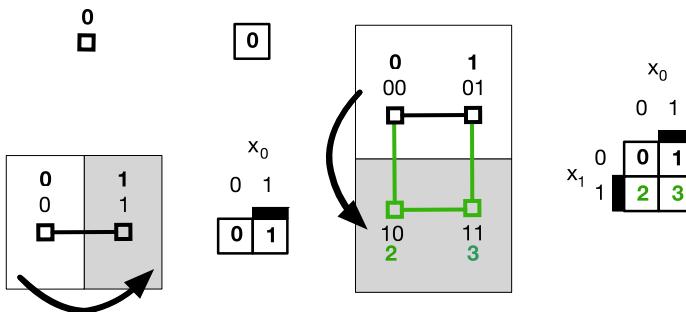


Abb. 3.18 Ableitung der Karnaugh-Veitch-Tafel

Boolesche Räume sind aufwendig zu zeichnen, insbesondere für mehr als 2 Dimensionen. Es ist jedoch möglich einen booleschen Raum in die Ebene zu projizieren: Wie die Projektion von einem 3-dimensionalen Globus auf 2-dimensionales Papier nennt man die so entstandenen Darstellungen boolesche oder logische Karten. Die Abb. 3.18a–b zeigt die Konstruktion für boolesche Räume mit weniger als 3 Variablen: Für jeden Knoten des Graphen wird ein Feld vorgesehen. Aus der Hantel werden 2 nebeneinanderliegende Quadrate, die die unterschiedlichen Zustände der Variablen x_0 zeigen. Das entspricht einem Spiegeln des 0-Zustands an einer virtuellen Kante nach rechts. Denkt man sich eine 2. virtuelle Kante und spiegelt die Tafel ein 2. Mal, erhält man eine logische Karte für 2 Variablen. Die Codierung an den Kanten der Tafel ergibt sich automatisch durch die Spiegelung und ist in der Abb. 3.18c gezeigt.

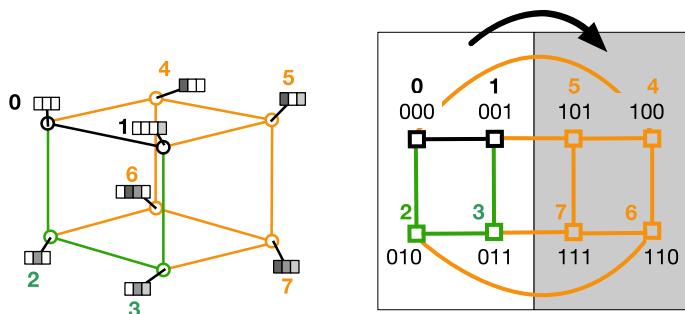
Boolesche Karten

Derartig konstruierte boolesche oder logische Karten heißen Karnaugh-Veitch-Diagramm. Die angelsächsische Literatur bezeichnet diese Tafeln oft verkürzt als Karnaugh-Diagramm (Karnaugh Map). Das Karnaugh-Veitch-Diagramm wurde von dem amerikanischen Informatiker Edward W. Veitch (1924–2013) 1952 vorgeschlagen und vom Physiker Maurice Karnaugh (1924) 1953 verfeinert. Karnaugh-Veitch-Diagramme werden in der deutschen Literatur häufig auch Karnaugh-Tafel genannt.

Karnaugh-Veitch-Tafel

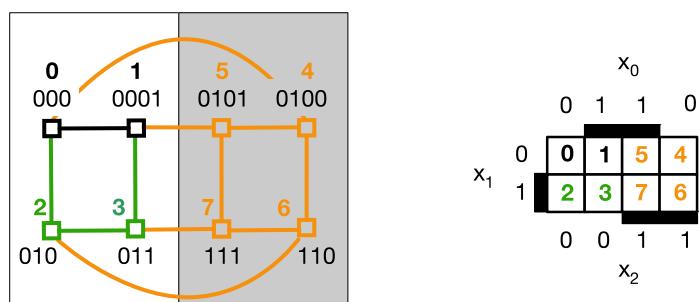
Die Abb. 3.19 zeigt, wie aus einem Würfel ein 2-dimensionaler Graph konstruiert wird. Dazu sind alle Knoten durchzunummerieren. Dabei entspricht die Nummer dem jeweiligen zu betrachtenden Bitmuster. Dieses wird im booleschen Raum codiert und annotiert den Graphen am entsprechenden Knoten. Wird das Quadrat des B^2 an einer virtuellen Geraden nach rechts gespiegelt und werden die Knoten des entstehenden neuen Quadrates links mit dem Quadrat des B^2 verbunden und zusätzlich noch zwei Kanten zwi-

3

Abb. 3.19 Flache Darstellung des B^3

schen den entstehenden Endknoten eingefügt, entsteht eine 2-dimensionale Abbildung, Abb. 3.19b des B^3 . Der neu entstandene Graph ist noch zu beschriften. Dafür wird den ursprünglichen Bitmustern des B^2 eine 0 vorangestellt und den gespiegelten Bitmustern eine 1. Der Leser vergewissere sich, dass in dem neu konstruierten Graphen nur Knoten zwischen Knoten der Hamming-Distanz 1 eingezeichnet sind. Von dem so entstandenen Bild kann abstrahiert werden. Dabei wird jeder Knoten ein neues Quadrat, und durch Kanten verbundene Knoten sind benachbart. Dann können die Kanten gestrichen werden, und es entsteht die in der Abb. 3.20 gezeigte Karnaugh-Tafel. Bedingt durch die grafische Konstruktion des ebenen Graphen ergibt sich die richtige Beschriftung der Variablen automatisch.

Es ist offensichtlich, dass benachbarte Min- oder Maxterme in der Karnaugh-Tafel zusammengefasst werden können. Würden in der gezeigten Tafel Feld 3 und 7 mit einer 1 belegt, könnte die Variable x_2 eliminiert werden. Das Gleiche gilt für eine entsprechende Belegung in den Feldern 2 und 6. Zwar sind die Felder in der Karnaugh-Tafel nicht benachbart, aber

Abb. 3.20 Ableitung der Karnaugh-Veitch-Tafel im B^3

im flachen Graphen existiert eine Kante. Demnach kann man sich die Karnaugh-Tafel wie einen aufgeschnittenen, flach geklopften Torus vorstellen. Ähnlich wie bei einer Weltkarte in Mercator-Projektion müssen der obere und der untere und der linke und der rechte Rand jeweils als benachbart angesehen werden. Der flache Graph auf der linken Seite der Abb. 3.20 liefert die Begründung für das Vorgehen.

Der boolesche Raum für 4 Variablen ist bereits komplex und im 3-dimensionalen aufwendig zu zeichnen: Es handelt sich um 2 ineinander geschachtelte Würfel, die über ihre Eckkanten verbunden sind. Die boolesche Karte wird wieder durch die Spiegelung an einer virtuellen Kante abgeleitet (Abb. 3.21). Der 2-dimensionale B^3 wird dabei nach unten gespiegelt, zwischen den Knoten 2, 3, 7 und 6 und den Knoten 10, 11, 15 und 14. Die Abb. 3.22 zeigt das Ergebnis.

Wird die Karnaugh-Tafel für 3 Variablen ebenfalls nach unten gespiegelt, entsteht eine Karte für 4 Veränderliche. Bei der Verwendung der Tafel ist auf die Variablenbelegung an den Kanten zu achten. Diese soll immer die Hamming-Distanz 1 darstellen, und zwar so, wie sie durch den booleschen Raum vorgegeben ist. An der oberen und linken Kante befinden sich die beiden Eins-Terme in der Mitte und an den verbleibenden Rändern die beiden Nullterme. An der unteren und der rechten Kante stehen beide Eins-Terme unten oder rechts beieinander, die beiden Nullterme oben oder unten beieinander.

In der Karnaugh-Tafel werden die Minterme der disjunktiven kanonischen Normalform mit 1 bezeichnet, also mit dem

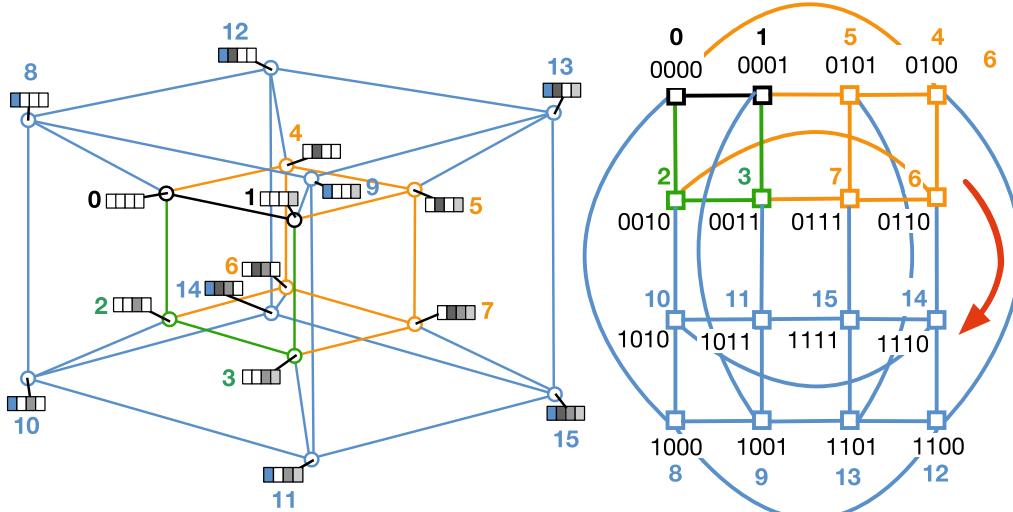
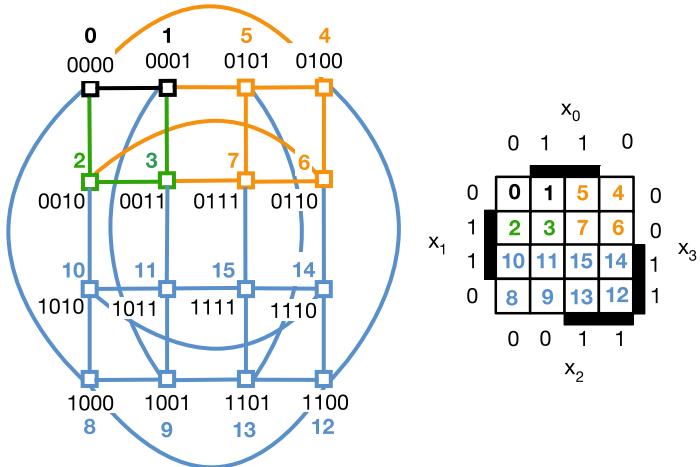


Abb. 3.21 Flache Darstellung des B^4

Abb. 3.22 Ableitung der Karnaugh-Veitch-Tafel im B^4

Wert, den der Minterm bei der entsprechenden Variablenbelegung annimmt. Die Minimierung der Schaltfunktion erfolgt durch das Zusammenfassen benachbarter Terme. Die Nachbarschaft ergibt sich aus dem flachen Graphen des entsprechenden booleschen Raumes. Dabei dürfen immer nur so viele Felder zusammengefasst werden, dass ihre Verknüpfung eine Potenz von 2 ist. Es ist leicht zu erkennen, dass für die Summe des Addierers keine Zusammenfassungen möglich sind. Bei der Minimierung von Mintermen dürfen beliebige Muster zusammengefasst werden, solange die Summe der ausgewählten und zusammengefassten Felder immer eine Potenz von 2^n ergibt und sich in diesen genau der Wert von n Variablen ändert. Darüber hinaus gehen die Karnaugh-Tafeln über ihre Grenzen weiter. Daher dürfen an den Rändern benachbarte Terme ebenfalls zusammengefasst werden.

► Beispiel 3.3.6

Gegeben sei die disjunktive kanonische Normalform der logischen Operation ODER:

$$f(x_0, x_1) = \overline{x_0}x_1 + x_0\overline{x_1} + x_0x_1.$$

Durch die in der Abb. 3.23 gezeigte grafische Minimierung findet man:

$$f(x_0, x_1) = x_0 + x_1$$

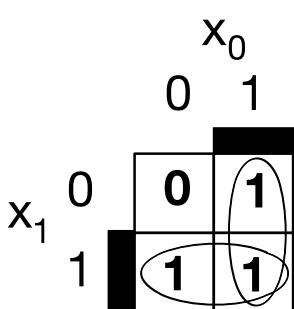


Abb. 3.23 Karnaugh-Tafel des logischen ODER

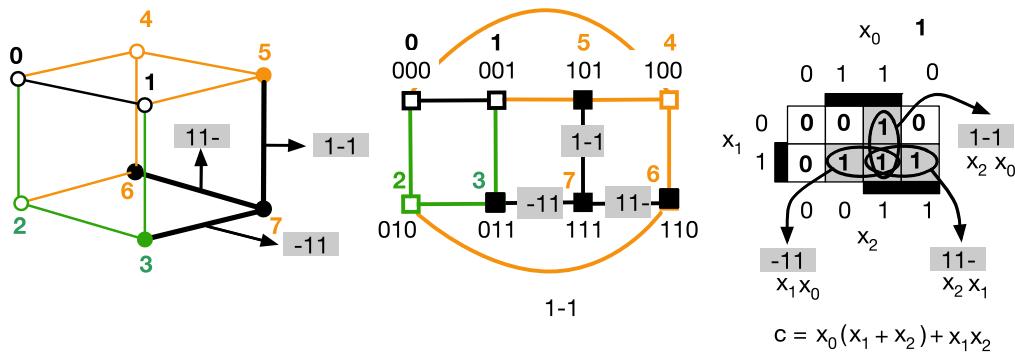


Abb. 3.24 Minimierung der Schaltfunktion Carry

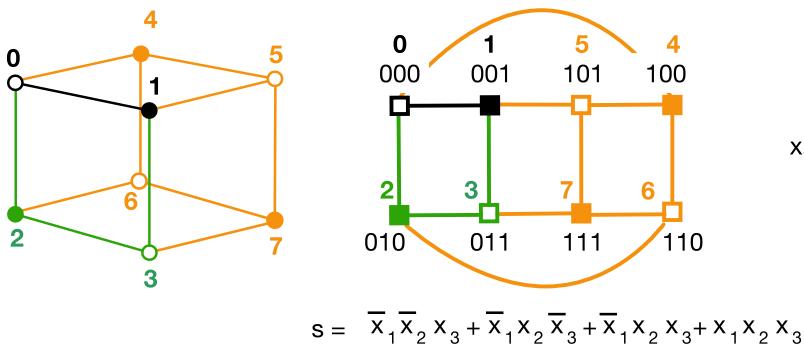


Abb. 3.25 Minimierung der Schaltfunktion Summe

Beispiel 3.3.7

Gegeben sei das bereits bekannte binäre Addierwerk für eine Stelle. Die Abb. 3.24 und 3.25 zeigen die Schaltfunktion der Summe und des Übertrags jeweils im 3-dimensionalen und flachen B^3 und der dazu gehörigen Karnaugh-Tafel. Im booleschen Raum sind die Minterme der Schaltfunktion mit den gefüllten Knoten markiert. In der Karnaugh-Tafel steht für jeden Minterm eine 1 und jeden nicht vorkommenden Term eine 0. Benachbarte Terme lassen sich zusammenfassen. Die jeweilige Variable kann beliebig belegt werden. In der Karnaugh-Tafel werden benachbarte Einsen zusammengefasst, und zwar immer als gerade Zahl, also 2, 4 usw. Terme. Für die Schaltfunktion der Summe können keine weiteren Terme zusammengefasst werden, da dafür keine Variablenbelegungen mit der Hamming-Distanz 1 existieren. Für die Schaltfunktion des Übertrags (Carry) lassen sich die Terme 3, 5, 6 und 7 zusammenfassen. ▲

Die Schaltfunktionsminimierung mithilfe der Karnaugh-Tafel kann wie folgt zusammengefasst werden:

Don't-care

- Erzeuge die Karnaugh-Tafel. Dabei werden für n Variablen 2^n Felder benötigt. Jede Kante der Karte repräsentiert eine Variable der Schaltfunktion. Diese werden so beschriftet, dass alle Felder eindeutig einem Minterm zugeordnet werden, also so, dass alle Variablenkombinationen aus Variable und negierter Variablen in der Tafel dargestellt werden können.
- Jeder Minterm wird mit 1 beschriftet. Die Kombinationen von Variablen und negierten Variablen, die für die Schaltfunktion eine 1 ergeben, werden markiert.
- Markierte benachbarte Felder werden zusammengefasst, und zwar so, dass möglichst große 2^n Rechtecke entstehen.
- Die minimierte Schaltfunktion wird konstruiert, indem Terme der Form $x + \bar{x} = 1$ eliminiert werden.

Zur Minimierung von Schaltfunktionen ist vorausgesetzt worden, dass für jede Kombination von Eingangsvariablen für eine Schaltfunktion spezifiziert wurde, ob die Funktion auf 0 oder 1 abbildet. Dies ist aber nicht immer der Fall. Allerdings ist das kein Nachteil, da die nicht spezifizierten Teile zur weiteren Minimierung genutzt werden können. Sollte es egal sein, welchen Wert die Schaltfunktion für eine bestimmte Eingangsbelegung annimmt, wird für den entsprechenden Max- oder Minterm ein „X“ in der Wahrheitstabelle angegeben oder vielmehr spezifiziert. Diese Variablenbelegung heißt Don't-care.

► Beispiel 3.3.8

Bei der Umwandlung von Dezimalzahlen mithilfe des BCD-Codes in Binärzahlen sind für bestimmte Bitmuster keine entsprechenden Dezimalsymbole definiert. Das folgt daraus, dass mit 4 Bit nur 10 Ziffern zu codieren sind. Ist ein Codierer zu bauen, der BCD-Ziffern in das entsprechende 9-Komplement umwandelt, ergibt sich die Wahrheitstabelle aus □ Tab. 3.1. Stellt man nun die disjunktive kanonische Normalform für jede Variable des Ausgangs, also für y_1, y_2, y_3, y_4 auf, erhält man die 4 Schaltfunktionen:

$$\begin{aligned}y_1(x_1, x_2, x_3, x_4) &= \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 \\y_2(x_1, x_2, x_3, x_4) &= \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 x_4 + \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 x_2 \bar{x}_3 x_4 \\y_3(x_1, x_2, x_3, x_4) &= \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 x_4 + \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 x_2 \bar{x}_3 x_4 \\y_4(x_1, x_2, x_3, x_4) &= \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 x_2 x_3 \bar{x}_4 + x_1 \bar{x}_2 \bar{x}_3 \bar{x}_4\end{aligned}$$

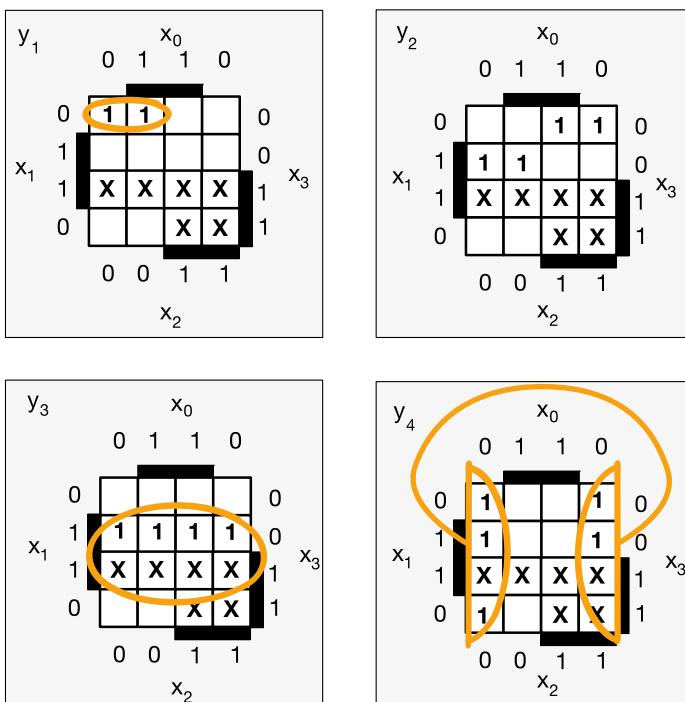
Im nächsten Schritt sind vier Karnaugh-Tafeln zu erstellen. Am einfachsten ist es dabei, zunächst die Don't-cares mithilfe eines X zu markieren, da diese Markierungen in allen vier Tafeln gleich sind. Nun können die jeweiligen Minterme eingetragen und zusammengefasst werden. In □ Abb. 3.26 sind die vier KV-Diagramme dargestellt. Für den BCD-9-Komplement-Codierer ergeben sich die folgenden minimalen Schaltfunktionen:

$$y_1 = \bar{x}_1 \bar{x}_2 \bar{x}_3, \quad y_2 = x_1 \bar{x}_2 + \bar{x}_1 x_2, \quad y_3 = x_3, \quad y_4 = \bar{x}_4$$

Tab. 3.1 Wahrheitstabelle des 9BCD

BCD		9BCD		BCD		9BCD	
x		y		x		y	
0	0000	9	1001	8	1000	1	0001
1	0001	8	1000	9	1001	0	0000
2	0010	7	0111	X	1010	X	XXXX
3	0011	6	0110	X	1011	X	XXXX
4	0100	5	0101	X	1100	X	XXXX
5	0101	4	0100	X	1101	X	XXXX
6	0110	3	0011	X	1110	X	XXXX
7	0111	2	0010	X	1111	X	XXXX

In der **Tab. 3.1** wurde als Zeichen für ein Don't-care ein X gewählt. ▶

**Abb. 3.26** Minimierung für Schaltfunktionen für den BCD-Codierer

Quine-McCluskey-Algorithmus

Primimplikanten

3.3.4.2 Algebraische Minimierung

Die grafische Methode, mithilfe der Karnaugh-Tafel Schaltfunktionen zu minimieren, stößt spätestens bei Funktionen mit 6 Variablen an ihre Grenze. Da Verfahren, die mit Binärmustern arbeiten und viele gleichartige Schritte erfordern, für den Menschen eintönig und langweilig sind, ist es wünschenswert, Logik nicht von Hand zu minimieren, sondern mit entsprechenden Programmen auf einem Rechner. Die bisher vorgestellten Ideen müssen also in algorithmische Form gebracht werden.

Das von einem Rechner auszuführende Verfahren der Primimplikanten wurde vom amerikanischen Philosophen und Logiker Willard V. O. Quine (1908–2000) und dem ebenfalls amerikanischen Physiker und Informatiker Edward f. McCluskey (1929–2016) im Jahr 1956 während ihrer gemeinsamen Zeit am Massachusetts Institute of Technology (MIT) entwickelt und wird daher auch Quine-McCluskey-Verfahren oder Quine-Algorithmus genannt.

Der Algorithmus von Quine-McCluskey versucht die Komplexität des Problems der Logikminimierung zu reduzieren. Um Terme beim späteren Ablauf des Verfahrens wieder finden zu können, sind zunächst alle Terme der Schaltfunktion zu nummerieren. Anschließend sollen die Min- oder Maxterme betrachtet werden. Diese werden zu Gruppen zusammengefasst, und zwar werden alle die Terme in einer solchen Gruppe zusammengefasst, die die gleiche Anzahl an positiven Literalen haben. In anderen Worten, eine Gruppe besteht aus Termen mit derselben Anzahl von Nullen und Einsen. Somit entstehen Gruppen von Termen mit keiner, mit einer, mit zwei, bis zu n Einsen. Gemäß ihrer Gewichtung der Anzahl der Einsen werden diese Gruppen sortiert, um dann eine niedrige mit einer jeweils höheren Gruppe zu verknüpfen. Diese Zusammenfassung entspricht der Absorptionsregel. Alle Literale, die sich bei einer Verknüpfung auslöschen, werden mit einem Don't-care markiert. Wurden zwei Gruppen verknüpft, entsteht dadurch eine neue Gruppe, die idealerweise reduzierte Terme enthält. Mit den aus allen Verknüpfungen neu entstandenen Gruppen wird das Verfahren so lange wiederholt, bis keine Reduktion mehr möglich ist.

Die so entstandenen maximal reduzierten Terme heißen Primimplikanten. Es ist bei dem Verfahren nicht zwingend, dass am Ende immer eine Gruppe von Primimplikanten entsteht. Vielmehr sind die Primimplikanten auf mehrere verschiedene Gruppen aufgeteilt, sodass für die minimale Schaltfunktion die besten oder minimalsten Primimplikanten ausgewählt werden müssen. Dazu ist zu untersuchen, welche der Minterme der Schaltfunktion von einem Primimplikanten abgedeckt werden. Ein Primimplikant soll dabei möglichst viele Terme,

die im Ergebnis eins werden, überdecken. Die Anzahl der abgedeckten Minterme ist daher ein Maß für die Qualität eines Primimplikanten. Je mehr Terme der Schaltfunktion ein Primimplikant abdeckt, umso weniger Primimplikanten werden dann für die minimierte Schaltfunktion benötigt. Die Überdeckung der Primimplikanten kann in eine Tabelle eingetragen werden: Dazu werden alle Minterme in den Spalten der Tabelle aufgelistet und die einzelnen Primimplikanten zeilenweise ge-listet. An dieser Stelle können noch die Nummern der Terme, aus denen der jeweilige Primimplikant hervorgegangen ist, ge-listet werden. Wenn ein Primimplikant einen Minterm abdeckt, wird die Stelle in der Tabelle markiert. Wird ein Minterm nur von einem einzigen Primimplikanten überstrichen, so ist dieser zwingend in der resultierenden Schaltfunktion anzugeben. Bei Mintermen, die von mehreren Primimplikanten abgedeckt werden, wird der Primimplikant gewählt, der am meisten der noch abzudeckenden Minterme berücksichtigt. Decken zwei Primimplikanten die gleiche Anzahl von Mintermen ab, kann frei gewählt werden.

► Beispiel 3.3.9

Gegeben sei die Schaltfunktion:

$$\begin{aligned}y = & \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 \bar{x}_2 x_3 x_4 + \\& \bar{x}_1 x_2 x_3 \bar{x}_4 x_1 + x_1 \bar{x}_2 \bar{x}_3 x_4 + \bar{x}_1 x_2 x_3 x_4 \\& + x_1 \bar{x}_2 x_3 x_4 + x_1 x_2 \bar{x}_3 x_4 + x_1 x_2 x_3 x_4\end{aligned}$$

Die Minterme werden, um Schreibarbeit zu sparen, binär codiert:

$$y = P_0 + P_1 + P_2 + P_3 + P_4 + P_6 + P_7 + P_9 + P_{11} + P_{13} + P_{15}$$

Im ersten Schritt werden Gruppen gebildet. In der Gruppe 0 befinden sich die Minterme, die keine 1 besitzen, (G_0), in der zweiten Gruppe die Minterme, die eine 1 haben, (G_1), bis hin zur Gruppe mit vier 1 im Minterm (G_4). Per Definition ist die Hamming-Distanz zwischen der Gruppe G_{i-1} und der Gruppe G_i immer 1. Man kann jetzt also alle Minterme einer Gruppe G_{i-1} mit den Mintermen der Gruppe G_i vergleichen. Unterscheiden sich zwei Minterme bei diesem Vergleich in lediglich einer Stelle voneinander, kann man die dazugehörige Variable eliminieren. In der Abb. 3.27 ist das durch einen Bindestrich dargestellt. Nicht alle Kombinationen von Mintermen bei diesem direkten Vergleich unterscheiden sich in einer einzigen Stelle. Diese brauchen nicht weiter betrachtet werden und sind in der Abb. 3.27 mit $H > 1$ markiert. Kombinationen von Mintermen, die sich reduzieren lassen, werden in der Abb. 3.27 grau dargestellt. Die Verknüpfung jeweils zweier Gruppen ergibt nach einem Rechenschritt eine neue Gruppe. Die Gruppen G_{10} und G_{11} der ersten Stufe erzeugen die Gruppe G_{20} für den zweiten Schritt, die Gruppen G_{12} und G_{13} bringen die Gruppe G_{21} hervor usw. Mit diesen neuen Gruppen wird die geschilderte Operation erneut durchgeführt. Dieses Verfahren wird so lange wiederholt, bis keine weiteren Vereinfachun-

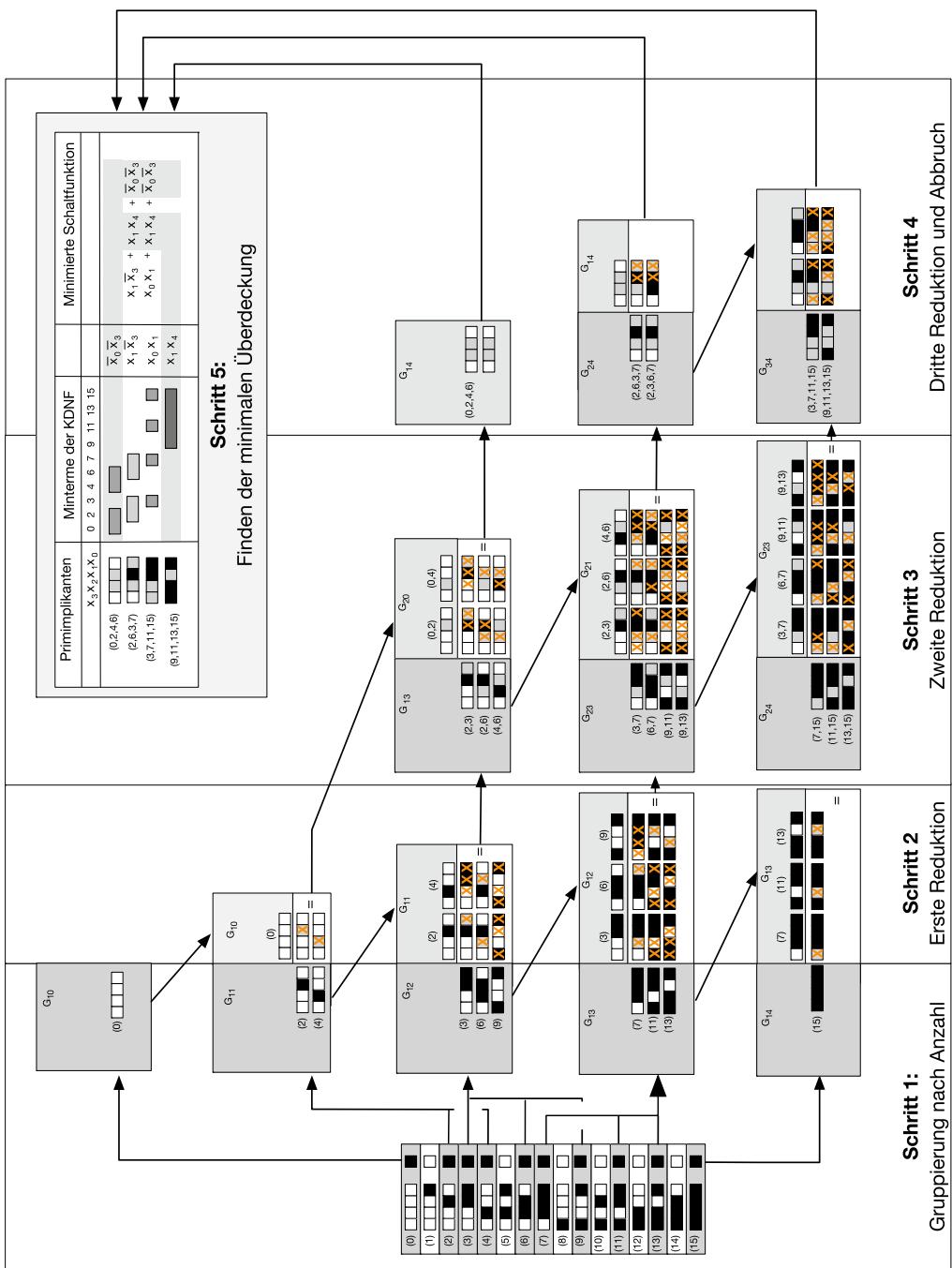


Abb. 3.27 Verfahren der Primimplikanten nach Quine-McCluskey

gen möglich sind. In dem Beispiel sind dies zwei Schritte. Im Ergebnis erhält man vier verschiedene Terme, die Primimplikanten. Im letzten Schritt ist dann zu überprüfen, welcher dieser Implikanten welchen Minterm erzeugt. Führt man während des Verfahrens die Nummern der ursprünglichen Minterme mit, kann man sie direkt der dafür vorgesehenen Datenstruktur entnehmen. Jetzt wird überprüft, ob alle Primimplikanten benötigt werden, um die gegebene Schaltfunktion darzustellen. In dem Beispiel müssen der erste und der letzte Primimplikant zwingend in der minimierten Schaltfunktion auftauchen, da die Minterme P_0 und P_4 nur vom ersten und die Minterme P_9 und P_{13} nur vom letzten Primimplikanten erzeugt werden. In der Abb. 3.27 ist dies extra kenntlich gemacht. Die beiden verbleibenden Primimplikanten sind frei wählbar, sodass am Ende zwei minimale Schaltfunktionen gefunden werden:

$$y_1 = x_1x_4 + \bar{x}_1\bar{x}_4 + \bar{x}_1x_3 \text{ und } y_2 = x_1x_4 + \bar{x}_1\bar{x}_4 + x_3x_4$$



⑦ Übungsaufgaben

Aufgabe 3.1 Zeigen Sie, dass die folgenden booleschen Funktionen äquivalent sind. Verwenden Sie hierfür die Gesetze der booleschen Algebra. Geben Sie jeweils die verwendeten Axiome an.

- a) $(x_1 \cdot \bar{x}_2) + (\bar{x}_1 \cdot x_2) = \overline{(x_1 \cdot \overline{(x_2 \cdot x_2)}) \cdot \overline{((x_1 \cdot x_1) \cdot x_2)}}$
- b) $(x_1 \cdot x_2) + (\bar{x}_1 \cdot x_3) + (\bar{x}_2 \cdot \bar{x}_3) = \overline{(\bar{x}_1 + \bar{x}_2)} \cdot (x_1 + \bar{x}_3) \cdot (x_2 + x_3)$

Aufgabe 3.2 Minimieren Sie die angegebenen Funktionen so weit wie möglich. Geben Sie den vollständigen Rechenweg an, und referieren Sie die benutzten Axiome.

- a) $f(x_1) = (x_1 + 0) \cdot (0 + 1) \cdot 1$
- b) $i(x_1, x_2) = \overline{((x_1x_2)(x_1x_2))}$
- c) $j(x_1, x_2, x_3, x_4) = x_1 + x_2x_4 + x_3 + x_1 \cdot x_4$
- d) $g(x_1, x_2) = \overline{x_1 \cdot \bar{x}_2 \cdot x_1}$
- e) $h(x_1, x_2, x_3, x_4) = (x_1 \cdot x_2) + (x_1 \cdot x_3) + x_1 \cdot (x_2 + x_3 \cdot x_4) + x_1$
- f) $k(x_1, x_2, x_3) = ((x_1 + x_3 \cdot (x_2 + x_3)) \cdot 1) \cdot 1$

Aufgabe 3.3 Wie bereits bekannt ist, bilden ODER und NICHT, UND und NICHT, NAND sowie NOR eine vollständige Basis. Das bedeutet, dass jede beliebige boolesche Funktion nur mithilfe der Funktionen einer Basis dargestellt werden kann.

- a) Stellen Sie den booleschen Ausdruck $x_1 \oplus x_2$ nur unter Verwendung von NAND bzw. NOR dar. Geben Sie die benutzten Axiome an.
- b) Stellen Sie folgenden booleschen Ausdruck

$$(x_1 \cdot \bar{x}_2) + (\bar{x}_2 \cdot \bar{x}_3) + (x_3 \cdot \bar{x}_0) + (x_0 \cdot \bar{x}_1)$$

nur unter Verwendung von NAND dar. Geben Sie die benutzten Axiome an.

Aufgabe 3.4 Wie für jede andere mathematische Struktur gelten auch für die boolesche Algebra Rechenregeln und Gesetze. Überprüfen Sie mithilfe einer Wertetabelle, ob die de morganischen Gesetze gelten. Beweisen Sie die Gesetze anschließend mithilfe der booleschen Algebra.

- a) $\overline{x_1 + x_2} = \overline{x_1} \cdot \overline{x_2}$
- b) $\overline{x_1 \cdot x_2} = \overline{x_1} + \overline{x_2}$

Aufgabe 3.5 Sie haben folgende Funktion gegeben:

$$f(x_2, x_1, x_0) = \begin{cases} 1 & \text{falls der Dezimalwert von } (x_2x_1x_0) \text{ mod } 2 = 0 \\ 0 & \text{sonst} \end{cases}$$

wobei $(x_2x_1x_0)$ eine vorzeichenlose Dualzahl mit x_0 als niedrigstwertige Zahl (LSB) ist.

- a) Stellen Sie die Wahrheitstabelle von $f(x_2, x_1, x_0)$ auf.
- b) Stellen Sie die DKNF und KKNF von $f(x_2, x_1, x_0)$ auf.
- c) Minimieren Sie die DKNF von $f(x_2, x_1, x_0)$, und geben Sie die dazu benutzten Axiome an.

Aufgabe 3.6 Formen Sie die folgenden booleschen Ausdrücke jeweils in ihre DKNF um.

- a) $f(x_1, x_2, x_3) = x_2$
- b) $f(x_1, x_2, x_3) = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_3$

Aufgabe 3.7 Gegeben ist die Funktion $f(x_1, x_2, x_3, x_4)$ mit $(x_1x_2x_3x_4)$ als vorzeichenlose Dualzahl.

$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{falls } (x_1x_2x_3x_4)_{10} \text{ mod } 4 = 1 \\ 1 & \text{falls die Quersumme von } (x_1x_2x_3x_4)_{10} = 6 \text{ ist} \\ 1 & \text{falls } (x_1x_2x_3x_4)_{10} \text{ mod } 2 = 0 \\ 1 & \text{falls } (x_1x_2x_3x_4)_{10} = 3 \\ 0 & \text{sonst} \end{cases}$$

- a) Geben Sie die Wertetabelle der Funktion $f(x_1, x_2, x_3, x_4)$ an.
- b) Stellen Sie die DKNF der Funktion $f(x)$ auf.
- c) Stellen Sie die KKNF der Funktion $f(x)$ auf.
- d) Minimieren Sie die Funktion $f(x)$ mittels KV.
- e) Vergleichen Sie die kanonischen Normalformen der Funktion $f(x_1, x_2, x_3, x_4)$ mit der minimierten Lösung. Nennen Sie drei Vorteile der Minimierung.

Aufgabe 3.8 Gegeben ist die Wahrheitstabelle mit den drei Variablen x_1, x_2 und x_3 und den Funktionen $f(x_1, x_2, x_3)$ und $g(x_1, x_2, x_3)$:

x_3	x_2	x_1	$f(x_1, x_2, x_3)$	$g(x_1, x_2, x_3)$
0	0	0	1	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

- a) Bilden Sie die DKNF und die KKNF für die Funktionen $f(x_1, x_2, x_3)$ und $g(x_1, x_2, x_3)$.
- b) Bilden Sie die DKNF und die KKNF für die Funktionen $f(x_1, x_2, x_3)$ und $g(x_1, x_2, x_3)$.
- c) Bestimmen Sie mithilfe eines KV-Diagramms die vollständig minimierte DNF für die Funktion $f(x_1, x_2, x_3)$.
- d) Bestimmen Sie mithilfe eines KV-Diagramms die vollständig minimierte KNF für die Funktion $g(x_1, x_2, x_3)$.

Aufgabe 3.9 Führen Sie für folgende DKNF jeweils eine Shannon-Zerlegung durch.

- a) $f(x_1, x_2) = x_1 \cdot x_2 + x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$
- b) $f(x_1, x_2, x_3) = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot x_3$
- c) $f(x_1, x_2, x_3) = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$

Aufgabe 3.10 Führen Sie für folgende boolesche Funktionen eine Shannon-Erweiterung nach allen Literalen durch, um die jeweilige DKNF zu bestimmen.

- a) $f(x_1, x_2, x_3) = x_1 \cdot x_2 + \overline{x_3}$
- b) $f(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot x_4$

Aufgabe 3.11 Minimieren Sie die angegebenen DKNF mithilfe eines Karnaugh-Veitch-Diagramms (verwenden Sie dazu die Vorlage auf der letzten Seite). Überprüfen Sie anschließend mithilfe der Shannon-Zerlegung ihr Ergebnis, indem Sie die Ursprungs-DKNF wiederherstellen.

- a) $f(x_0, x_1, x_2) = x_0 \cdot \overline{x_1} \cdot \overline{x_2} + \overline{x_0} \cdot x_1 \cdot \overline{x_2} + x_0 \cdot x_1 \cdot \overline{x_2} + \overline{x_0} \cdot x_1 \cdot x_2$
- b)

$$\begin{aligned} g(x_0, x_1, x_2, x_3) &= x_0 \cdot \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + \overline{x_0} \cdot \overline{x_1} \cdot x_2 \cdot \overline{x_3} + \overline{x_0} \cdot x_1 \cdot \overline{x_2} \cdot \overline{x_3} \\ &\quad + \overline{x_0} \cdot x_1 \cdot x_2 \cdot \overline{x_3} + \overline{x_0} \cdot x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_0} \cdot x_1 \cdot x_2 \cdot x_3 \\ &\quad + x_0 \cdot \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_0} \cdot \overline{x_1} \cdot x_2 \cdot x_3 \end{aligned}$$

Weiterführende Literatur

- [Gaj96] Daniel D. Gajski. *Principles of Digital Design*. Pearson, 1996. ISBN: 9780133011449.
- [Pau99] Christoph Scholl Paul Molitor. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. Stuttgart, Leipzig: Teubner Verlag, Springer Fachmedien, 1999.
- [Bro03] Frank Markham Brown. *Boolean Reasoning: The Logic of Boolean Equations*. 2. Aufl. Dover Books on Mathematics, 2003. ISBN: 9780486427850.
- [Max08] Clive Maxfield. *Bebop to the Boolean Boogie: An Unconventional Guide to Electronics*. 3. Aufl. Newnes, 2008. ISBN: 9781856175074.
- [RS97] Curtis T. McMullen Robert K. Brayton Gary D. Hachtel und Alberto L. SangiovanniVincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984, 8th reprinting 1997. ISBN: 9781461297840.



Digitale Schaltungen

Inhaltsverzeichnis

- 4.1 Schaltungstechnik – 189
 - 4.2 Logische Gatter in MOS-Technik – 220
 - 4.3 Flipflops und Bitspeicher – 237
 - 4.4 Speichermatrizen – 256
 - 4.5 Logikfelder – 261
- Weiterführende Literatur – 286

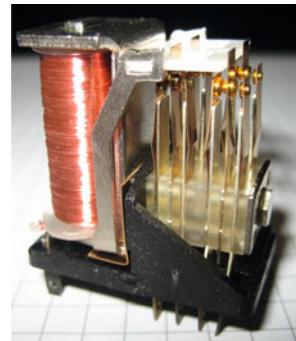
*Digital ist besser
Tocotronic*

Um einen Rechner zu bauen, spielen Kenntnisse der Mechanik keine Rolle mehr. Heutige Computer sind elektronische Geräte. Um zu verstehen, wie ein elektronischer Rechner funktioniert, werden nach einer knappen Wiederholung der elementaren physikalischen Grundlagen der Elektrotechnik die Bauelemente Diode und Transistor vorgestellt. Ziel des Kapitels ist es, eine Vorstellung zu vermitteln, wie elektronische Schalter arbeiten und gefertigt werden können. Aus diesen werden dann logische Schaltungen abgeleitet. Mit diesen Gattern lassen sich die abstrakten mathematischen Schaltfunktionen aus ▶ Kap. 3 realisieren. Gatter sind dann Grundlage von Speicherbausteinen und Flipflops. Der Leser sollte nach dem Durcharbeiten des Kapitels in der Lage sein, die Funktion digitaler Schaltungen mit einfachen Modellen zu berechnen. Er sollte die heute am meisten verbreitete Schaltungstechnik CMOS kennen und die elektrische Arbeitsweise logischer Gatter erläutern können. Mithilfe dieser binären Schaltung kann er Flipflops und andere Speicherelemente konstruieren und deren Funktion beschreiben. Das Kapitel legt die elektrotechnischen Grundlagen der digitalen Schaltungen und bildet die Basis zum technischen Verständnis der heutigen Rechnerarchitektur.

Die Schaltalgebra erlaubt das Aufstellen beliebiger Schaltfunktionen. Mit diesen lassen sich mathematische Operationen beschreiben, die die Grundlage für den Bau eines Rechners bilden. In den Telefonnetzen Shannons wurden die Schaltfunktionen durch eine Kaskade von Schaltern realisiert. Diese konnten manuell oder automatisch betätigt werden. Gesteuert schaltende Bauelemente hießen zu dieser Zeit Relais (alfrz. „relaier“ = zurücklassen, frz. „relais“ = Station für den Pferdewechsel) und waren seit Mitte des 19. Jahrhunderts aus der drahtgebundenen Telegrafie, in der sie die Signale nach etwa 30 km verstärkten, bekannt. Ein Relais bestand im Eingangskreis aus einer Drahtwicklung mit Eisenkern als Elektromagnet und einem metallischen Taster, der einen Ausgangstromkreis schließen konnte (► Abb. 4.1). Floss durch die Spule ein Strom, wurde ein Magnetfeld aufgebaut, und dieses wirkte auf den Schalter. Wurde dieser angezogen, öffnete sich der Ausgangsschaltkreis, wurde er abgestoßen, schloss er. Gegebenenfalls konnte der Schalter durch eine Feder in seine Ausgangsstellung gebracht werden. Ausgehend von den Relais und der aus der booleschen Algebra hervorgegangenen Schaltalgebra bot sich zur Realisierung von Computern eine zweiwertige Logik an. Da elektromechanische Schalter in den 1930er-Jahren für den Bau von Telefon- und Telegraphennetzen weitverbreitet waren, wurden die frühen Rechner in Relaistechnik gebaut.

Die von Konrad Zuse 1941 gebaute Z3 als erster Digitalrechner ist ein Beispiel hierfür. Diese zweiwertige, digitale (lat. „*digitus*“ = Finger; mit Fingern wird gezählt, abzählbar) Logik erlaubte den Bau beliebiger Rechenmaschinen auf der Grundlage zunächst elektromechanischer und später elektronischer Bauelemente.

Elektromechanische Komponenten sind leichter ansteuerbar als rein mechanische wie Zahnräder oder die Staffelwalze und benötigen viel weniger Raum. Elektronische Bauelemente wiederum haben keine Mechanik und arbeiten rein elektrisch. Mechanische Bausteine haben den Nachteil eines hohen Verschleißes. Insbesondere durch Reibung entsteht Abrieb, und dieser kann mit der Zeit zum Ausfall des Bauelementes führen. Zwar können auch elektronische Schaltungen altern und verschleißen, jedoch ist deren Zuverlässigkeit millionenfach größer als die von elektromechanischen Bauelementen. Insbesondere der Bau von Rechnern mit seinen vielen gleichartigen Schaltern erfordert für den sicheren Betrieb und eine hohe Verfügbarkeit zuverlässige und wartungsarme Bauelemente mit langer Lebensdauer. Aus diesem Grund sind heutige Computer aus elektronischen Komponenten aufgebaut. Zum besseren Verständnis moderner Anlagen führt das folgende Kapitel in einfache elektrotechnische und elektronische Grundlagen ein.



redhat, Wikimedia, Wikimedia Commons

Abb. 4.1 Relais offen

4.1 Schaltungstechnik

Elektrische Naturphänomene bilden die physikalische Grundlage vieler heute gängiger Maschinen. Von der Waschmaschine über die Elektrolokomotive bis zum Telefon beruhen diese Erfindungen auf dem Wechselspiel zwischen Spannung und Strom. Insbesondere die Elektronik spielt in unserem Alltag eine inzwischen fast beängstigend große Rolle. Viele der in diesem Kapitel eingeführten Grundlagen sollten dem Leser aus der Schule bekannt sein und sind zum technischen Verständnis von heutigen Rechenmaschinen unabdingbar. Um in den weiteren Abschnitten von den gleichen Voraussetzungen ausgehen zu können, werden diese im Folgenden wiederholt und so weit ausgebaut, dass der Informatiker versteht, wie die von ihm genutzte Hardware funktioniert. Fundamentales Wissen zur Physik der Elektronik und der Herstellung integrierter Schaltungen ist wesentlicher Bestandteil für den Bau der Schaltkreise von Computern. Aber auch für den Anschluss beliebiger Geräte an einen Rechner benötigen Softwareingenieure dieses Grundgerüst an elektrotechnischen Kenntnissen.

4.1.1 Elektrotechnische Grundlagen

Zunächst werden noch einmal die Begriffe Ladung, Spannung und Strom rekapituliert und in Zusammenhang gebracht. Aus diesen Bezeichnungen und dem ohmschen Gesetz leitet sich ein einfaches mathematisches Konzept zur Beschreibung elektrischer und elektronischer Schaltungen ab. Dieses Modell ist physikalisch tiefer gehend als die reine, durch Logikgatter aufgebaute digitale Schaltung. Sie ermöglicht jedoch ein Verständnis der in einem Rechner auftretenden zeitlichen Verzögerungen und erlaubt es später, grundlegende Rechnerarchitekturen technisch zu begründen.

4.1.1.1 Spannung und Strom

Elektrizität

Elektrische Phänomene (griech. „élektron“ = Bernstein) waren bereits im Altertum bekannt. So werden erste Experimente zur Elektrizität dem griechischen Astronomen Thales von Milet (um 623/24 v. Chr. bis zwischen 548/44 v. Chr.) zugeschrieben: Bekannt war damals, dass ein Bernstein eine anziehende Kraft auf Vogelfedern ausübt, wenn vorher mit einem Stück Stoff oder einem Fell an diesem gerieben wurde. Im elisabethanischen England begründete der Arzt William Gilbert (1544–1603) die moderne Elektrizitätslehre. Er beschrieb als Erster den Unterschied zwischen magnetischen und elektrischen Erscheinungen und untersuchte systematisch die Ladung unterschiedlicher Materialien. Gilbert fand heraus, dass Elektrizität nicht nur auf Bernstein vorkam, sondern auch auf anderen Stoffen nachweisbar ist. Mithilfe des von ihm erfundenen Elektroskops konnte Gilbert zeigen, dass zwischen den elektrischen Ladungen eine Kraft wirkt. Später fand Otto von Guericke (1602–1682) heraus, dass es zwei unterschiedliche Wirkungen gibt, eine abstoßende und eine anziehende. Dies führte zu der Erkenntnis des Vorhandenseins zweier Arten von Ladung, einer positiven und einer negativen.

Ladung

Das Elektroskop weist experimentell nach, dass sich gleichartige Ladungen mit einer bestimmten Kraft abstoßen. Diese kann gemessen werden. Es zeigt sich darüber hinaus, dass unterschiedliche Ladungen einander mit dieser bestimmten Kraft anziehen. Diese verschiedenen Ladungen sind die positive (+) und die negative (−) Ladung. Charles Augustin de Coulomb (1736–1806) berechnete als Erster die zwischen elektrischen Ladungen auftretende Kraft. Der als coulombsches Gesetz berühmt gewordene Satz besagt, dass die Kraft proportional zur Menge der jeweiligen Ladungen und umgekehrt proportional zum Abstand dieser ist. Heute ist bekannt, dass kleine Elementarteilchen Ladung tragen. Die kleinste in der Natur vorkommende negative Ladung hat das Elektron, die minimal

mögliche positive haben das Positron und das Proton. Ladungen sind ursächlich für elektromagnetische Erscheinungen wie den Magnetismus und den Stromfluss. Ladungen lassen sich speichern und können so Schaltzustände in Computern repräsentieren.

Die Kraft, die Ladungen aufeinander ausüben, kann berechnet und gemessen werden. Für theoretische Betrachtungen ist es lästig, immer die Ladungsmengen und den Abstand dieser zueinander zu betrachten. Der schottische Physiker James Clerk Maxwell (1831–1879, □ Abb. 4.2) begründete eine auf dem Begriff des elektrischen Feldes aufbauende Theorie des Elektromagnetismus. Teilt man die auf ein Teilchen wirkende Kraft durch die Trägerladung dieses Teilchens, lässt sich jedem Punkt im Raum eine elektrische Feldstärke genannte Eigenschaft zuweisen:

$$E = \frac{F}{Q} \quad (4.1)$$

Mithilfe dieser auf die Menge der Ladung normierten Kraft vereinfachen sich die Berechnungen der elektrischen Wechselwirkung zwischen geladenen Teilchen. Mit dem Feldbegriff gelang es Maxwell auf der Basis von lediglich vier grundlegenden Gleichungen alle elektromagnetischen Phänomene in einer einheitlichen Theorie zusammenzufassen und auch die elektromagnetischen Wellen vorherzusagen.

Zum Verständnis elektronischer Schaltungen und zur einfachen mathematischen Modellierung ist ein aus dem elektrischen Feld abgeleiteter Begriff von zentraler Bedeutung: die elektrische Spannung. In einem elektrischen Feld ist die Feldstärke umgekehrt proportional zum Quadrat des Abstands. Verbundene Punkte im Raum um eine Ladung mit der gleichen elektrischen Feldstärke heißen Äquipotenziallinien. Eine kleine elektrische Punktladung mit im Vergleich zur Ursache des elektrischen Feldes vernachlässigbarer elektrischer Ladung kann im elektrischen Feld bewegt werden.

Auf diese Ladung wirkt eine Kraft, und gemäß der Definition der Arbeit

$$W_{AB} = \int_A^B F \, ds = \int_A^B E q \, ds \quad (4.2)$$

und

$$U_{AB} = \frac{W_{AB}}{q} \quad (4.3)$$

kann die elektrische Spannung mit

Elektrisches Feld

Elektrische Spannung



G. J. Stodart, nach einer Photographie von F. Of Greenock, Wikimedia Commons

□ Abb. 4.2 James Clerk Maxwell

$$U_{AB} = \int_A^B E \, ds \quad (4.4)$$

definiert werden. Die elektrische Spannung ist somit eine Potenzialdifferenz im elektrischen Feld und übt auf Ladungsträger eine anziehende bzw. abstoßende Kraft aus (Abb. 4.3).

Angenommen, der Weg zwischen den Punkten A und B ist ein geschlossener Umlauf. Ähnlich wie im Gravitationsfeld für Massen ist die aufgewendete Energie für eine bewegte Ladung auf einer Äquipotenziallinie null. Oder mit anderen Worten: Wird eine kleine positive Probeladung in einem elektrischen Feld einer sehr viel größeren Ladung in Richtung zur Quelle des Feldes verschoben, muss die Abstoßungskraft mit einer entsprechenden Gegenkraft überwunden werden. Es ist daher Arbeit aufzuwenden. Wirkt keine Kraft mehr auf die Probeladung, wird diese abgestoßen, und die gleiche Energie wird wieder freigesetzt. Im Ergebnis ist also die gesamte verrichtete Arbeit null. Daher gilt

$$\oint E \, ds = 0 \quad (4.5)$$

Elektrischer Strom

Wirkt auf Ladungsträger eine Kraft, bewegen sich diese. Eine durch ein elektrisches Feld oder alternativ eine elektrische Spannung hervorgerufene Kraft bewirkt einen Transport von Ladungsträgern. Elektrischer Strom ist also ein stetiger Fluss bewegter Ladungsträger. Die Bewegung von Ladungsträgern kann durch einen zeitlich konstanten Ladungsfluss durch eine Fläche beschrieben und die Stromstärke berechnet werden:

$$I = \frac{\Delta q}{\Delta t} \quad (4.6)$$

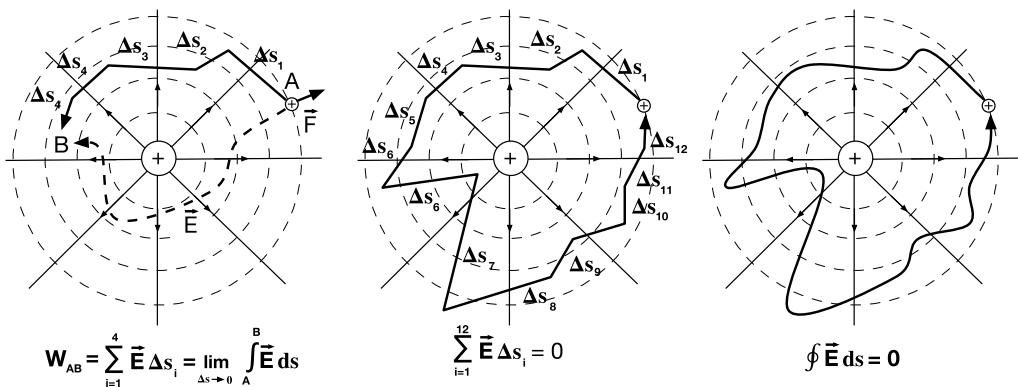


Abb. 4.3 Bewegen einer Ladung im elektrischen Feld

Da aber der Augenblickswert der Stromstärke interessiert, ist ein Grenzübergang durchzuführen:

$$I = \lim_{\Delta t \rightarrow 0} \frac{\Delta q}{\Delta t} = \frac{dq}{dt} \quad (4.7)$$

Befinden sich in einem Körper freie Ladungsträger, ist dieser in der Lage, den elektrischen Strom zu leiten. Legt man an den Enden dieses Leiters eine elektrische Spannung an, bildet sich in diesem ein elektrisches Feld. Die Elektronen bewegen sich dann in Richtung der positiven Ladung der Spannungsquelle: Es fließt ein elektrischer Strom.

4.1.1.2 Elektrische Netzwerke

In den von Shannon ursprünglich betrachteten Telefonsystemen wurden ein Mikrofon und ein Lautsprecher über ein Netzwerk aus elektrischen Leitern und Schaltern verbunden. Um das elektrische und nicht nur das logische Verhalten wie in der Schaltalgebra zu analysieren, wird ein einfaches physikalisches Modell benötigt. Dies könnte zunächst einmal auf der Grundlage der maxwellschen Differenzialgleichungen erfolgen. Diese sind jedoch allgemein formuliert und müssen unter bestimmten Randbedingungen gelöst werden. In diesen ist dabei die Geometrie der Bauelemente zu berücksichtigen, und es entstehen aufwendig zu lösende Gleichungssysteme. In der Schaltungstechnik abstrahiert man daher von den Feldgleichungen Maxwells. Dazu werden für jeden einzeln zu beobachtenden natürlichen Effekt ein ideales Bauelement eingeführt und das Strom-Spannungs-Verhalten dieses Netwerkelementes aus den maxwellschen oder anderen physikalischen Beziehungen abgeleitet. Eine Schaltung wird dann als topologisches Netz aus diesen Elementen beschrieben. Ein solches Netzwerk ist ein aus Knoten und Kanten bestehender Graph. Die Kanten des Graphen werden von den unterschiedlichen idealen elektrischen Netwerkelementen gebildet und die Knoten stellen Verbindungen zwischen mehr als zwei Elementen her.

Ideale Netwerkelemente mit einem definierten Strom-Spannungs-Verhalten sind der elektrische Widerstand, die Kapazität (Kondensator) und die Induktivität (Spule). Um einen Stromkreis, also eine Schaltung zu modellieren, wird dann noch eine ideale Spannungs- oder Stromquelle benötigt. Stromquellen entstehen in der Natur, wenn elektrische Ströme durch chemischen Ladungstransport hervorgerufen werden. Mit diesen idealen Netwerkelementen können technische Schaltungen, wie z. B. das klassische Hochspannungsnetz oder die Elektrik eines Hauses modelliert und berechnet werden.

Fließt durch einen stofflichen Körper ein Strom, kann dieser nicht ungehindert strömen. Zunächst einmal fließen die Ladungsträger in Richtung des elektrischen Feldes, und je höher

**Netzwerk-
elemente**

Elektrischer Widerstand

die elektrische Spannung ist, desto größer ist der Stromfluss. Bedingt durch die brownsche Wärmebewegung der den Körper bildenden Atome werden die fließenden Elektronen immer wieder abgelenkt. Offensichtlich wird dem elektrischen Strom ein Widerstand entgegengesetzt. Dieser ist umso stärker, je länger der von den Ladungsträgern zurückzulegende Weg durch den Körper ist. Der elektrische Widerstand ist daher proportional zur Länge l des Leiters. Dagegen verringert sich die Hemmung der Elektronen mit zunehmendem Leitungsquerschnitt: Je größer also die von den Ladungsträgern zu durchströmende Fläche A ist, desto stärker ist der den Körper durchfließende Strom. Daraus folgt:

$$I \simeq \frac{A}{l} \cdot U \quad (4.8)$$

Dieser Zusammenhang lässt sich leicht experimentell bestätigen. Es zeigt sich bei systematischen Versuchen, dass verschiedene Stoffe dem elektrischen Strom einen unterschiedlichen Widerstand entgegensetzen. Durch die Einführung einer Materialkonstanten, der elektrischen Leitfähigkeit κ , lässt sich das nach dem Physiker Georg Simon Ohm (1789–1854) benannte Gesetz formulieren:

$$R = \rho \frac{l}{A} \quad (4.9)$$

Aus dieser Gleichung kann direkt das ideale Netzwerkelement elektrischer Widerstand abgeleitet werden (Abb. 4.4). Mit $R = \rho \frac{e}{A}$ und $\rho = \frac{1}{\kappa}$, dem spezifischen elektrischen Widerstand des Materials folgt:

$$I = \frac{U}{R} \quad (4.10)$$

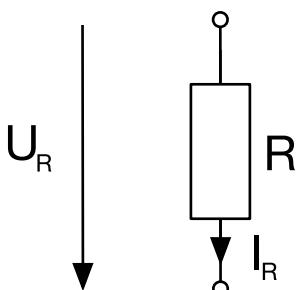


Abb. 4.4 Widerstand

Ladungen lassen sich speichern. Bereits William Gilbert war in der Lage, elektrische Ladung mithilfe seines Elektroskops festzuhalten. Und auch der mit einem Fell elektrisierte Bernstein speichert Ladung. Im 18. Jahrhundert konnte man elektrische Ladung mit Elektrisiermaschinen erzeugen. Allerdings konnte ein Elektroskop nur geringe Ladungsmengen aufnehmen. Bedingt durch die Analogie des elektrischen Flusses zum Fluss von Wasser versuchte man daher Ladung in wassergefüllten Flaschen zu speichern. Dazu wurde in ein Glasgefäß eine metallische Kette gehängt und diese mit einer Elektrisiermaschine mit Ladungsträgern versorgt. Allerdings ließ sich kein speichernder Effekt beobachten oder messen. Bei einigen dieser Versuche unterlief dem Naturforscher Pieter van Musschenbroek (1692–1761) in Leiden ein Fehler: Die zunächst auf einer isolierenden Unterlage stehende Flasche wurde versehentlich

auf einen leitenden Unterbau gestellt, und van Musschenbroek erhielt einen heftigen Stromschlag bei Berührung der Kette, und dies, obwohl die Elektrisiermaschine bereits abgetrennt war.

Die spätere Physik konnte zeigen, dass um Ladungen zu speichern zwei elektrische Leiter durch ein isolierendes Medium getrennt sein müssen und dass die Speicherkapazität nicht vom Volumen des Gefäßes abhängt, sondern nur von der Fläche der einander gegenüberliegenden und voneinander isolierten Leiter. Pieter van Musschenbroek hatte mit der Leidener Flasche ein heute Zylinderkondensator (lat. „condensare“ = verdichten) genanntes Bauelement entdeckt. Die Ladung, die ein solcher Kondensator speichern kann, ist proportional zur angelegten Spannung U und zur Kapazität C des Kondensators:

$$q = C \cdot U \quad (4.11)$$

Die Strom-Spannungs-Charakteristik des idealen Netzwerk-elementes Kapazität (Abb. 4.5) berechnet sich dann über den Stromfluss durch den Kondensator:

$$I = \frac{dq}{dt} = C \frac{dU}{dt} \quad (4.12)$$

In modernen integrierten Schaltungen zur Realisierung von Rechnern hat der Kondensator eine wichtige Bedeutung: Eine gespeicherte Ladung lässt sich als eine 1 und keine Ladung als logische 0 interpretieren. Damit können mit Kondensatoren Zahlen gespeichert werden. Dieses Speichern ist neben dem Rechnen eine grundlegende Voraussetzung zum Bau von Computern.

Der englische Physiker Michael Faraday (1791–1867, Abb. 4.6) konnte experimentell nachweisen, dass ein strom-durchflossener Leiter in einem Magnetfeld rotiert, sich also um den Magneten herum bewegt. Dieses elektrische Rotation genannte Phänomen führte zum Nachweis der elektrischen Induktion, bei der durch Annäherung eines Magnetfeldes an einen Draht ein Strom induziert wird. Die Induktion tritt verstärkt in zu Spulen gewickelten Kabeln auf. Integrierte Siliziumschaltungen haben keine nennenswerten Induktivitäten. Lediglich die Bonddrähte zur Verbindung des Chips mit dem Gehäuse verhalten sich bei hohen Schaltfrequenzen stark induktiv. Auf eine Herleitung der Strom-Spannungs-Charakteristik aus den maxwellschen Gleichungen soll daher an dieser Stelle verzichtet werden. Symmetrisch zur Strom-Spannungs-Charakteristik der Kapazität ergibt sich für das ideale Netzwerkelement Spule mit der gegebenen Induktivität L die Gleichung

Kondensator

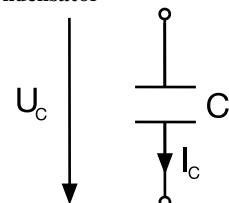
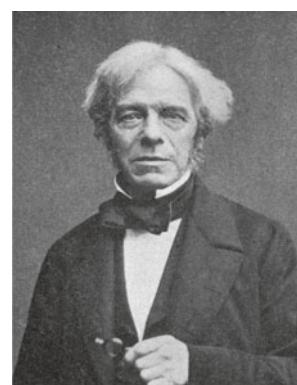


Abb. 4.5 Kapazität

Elektrische Kapazität



Probably albumen carte-de-visite by John Watkins, Wikimedia Commons

Abb. 4.6 Michael Faraday

$$I = L \frac{dI}{dt} \quad (4.13)$$

Die Induktivität L leitet sich dann aus der Geometrie der jeweiligen zu betrachtenden Leiteranordnung ab (Abb. 4.7).

Bei der Berechnung von Schaltungen ist eine Spannungsversorgung anzunehmen. Im Idealfall kann diese unendlich viel Energie in unendlich kurzer Zeit bereitstellen, d. h., zwischen zwei Knoten eines elektrischen Netzwerks liegt immer die Spannung U_0 an, und zwar egal wie groß die daran angeschlossene Last ist. Es ist offensichtlich, dass sich reale Spannungsquellen nicht so verhalten. Entzieht man einem Generator oder einer Batterie Energie, dann muss diese bereitgestellt oder erzeugt werden. Diese Arbeit steht aber nicht sofort zur Verfügung und kann nicht beliebig groß sein. Hinzu kommt, dass, wenn Energie transportiert wird, auch ein Strom vorhanden sein muss. Wenn aber ein Strom fließt, fungieren Teile der Quelle als Widerstand. Ein derartiges Verhalten einer technischen Spannungsquelle kann durch eine ideale Spannungsquelle (Abb. 4.8) zusammen mit einem idealen Widerstand modelliert werden. Zur Betrachtung der Eigenschaften logischer Schaltungen ist es allerdings ausreichend anzunehmen, die Versorgung mit Energie ist verlustlos.

Das Äquivalent zur idealen Spannungsquelle ist die ideale Stromquelle (Abb. 4.9). Eine ideale Stromquelle stellt zwischen zwei Knoten eines elektrischen Netzwerks immer einen konstanten Strom bereit. Reale Stromquellen besitzen ebenfalls einen Innenwiderstand, dieser ist nun aber parallel zur Stromquelle geschaltet. Ideale Spannungsquellen lassen sich in ideale Stromquellen umrechnen und umgekehrt.

Es sei daran erinnert, dass auf einem geschlossenen Weg in einem elektrischen Feld keine Arbeit verrichtet wird. Der Graph eines Netzwerks, bestehend aus Quellen und Elementen, ist zyklisch. Es finden sich daher Maschen in dem Graphen. Knoten in dem Netz haben ein unterschiedliches elektrisches Potenzial, zwischen zwei Knoten fällt also eine Spannung ab. In einem solchen Netzwerk gelten die folgenden, nach dem deutschen Physiker Robert Kirchhoff (Abb. 4.10) benannten, Regeln (Abb. 4.11).

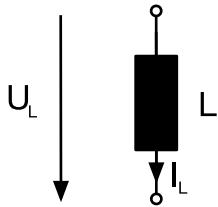


Abb. 4.7 Induktivität

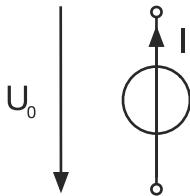


Abb. 4.8 Ideale Spannungsquelle

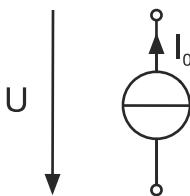


Abb. 4.9 Ideale Stromquelle

Theorem 4.1.1 – Satz von Kirchhoff

Die Summe aller Ströme in einem Knoten und die Summe aller Spannungen in einem elektrischen Netzwerk sind null:

$$\sum_{k=1}^n I_k = 0 \text{ und } \sum_{k=1}^n U_k = 0 \quad (4.14)$$

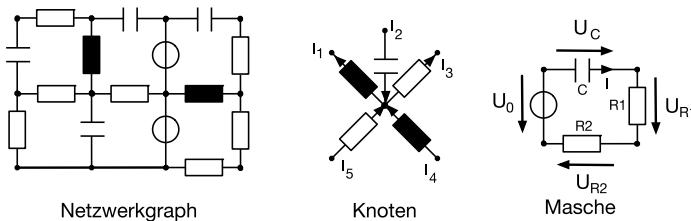


Abb. 4.11 Elektrisches Netzwerk mit Knoten und Maschen

Beweis

Der erste Satz leitet sich aus der ersten maxwellschen Gleichung ab und entspricht physikalisch der Erhaltung der Ladung. In einem Knoten eines elektrischen Netzwerks werden keine Ladungen gespeichert. Ladungen, die hineinfließen, müssen auch herausfließen. Der zweite Satz folgt direkt aus dem Induktionsgesetz der zweiten maxwellschen Gleichung: Die Summe der Potenzialdifferenzen bei einem Umlauf im elektrischen Feld ist null.



unbekannt, Wikimedia Commons

Abb. 4.10 Robert Kirchhoff

4.1.1.3 Logische Signale

Verändert sich eine physikalische Größe mit der Zeit, spricht man von einem Signal. Kann dieses zu allen Zeitpunkten beliebige Werte wie z. B. Druck, Temperatur, Spannung oder Strom annehmen, ist es ein analoges Signal. Digitale Signale wechseln ihre Wertigkeit dagegen nur zu bestimmten Zeiten und nehmen ausschließlich eine begrenzte Anzahl von Werten oder Zuständen an. Mit elektrischen oder elektronischen Schaltungen erzeugte Signale sind analog. Allerdings werden die in der digitalen Schaltung vorliegenden kontinuierlichen Signale von dieser selbst diskret interpretiert. Der Strom oder die Spannung auf einer Leitung in einem Rechner kann beliebige Werte annehmen, die Schalter am Ende der Übertragung verarbeiten dieses analoge Signal jedoch immer digital. Dazu wird festgelegt, dass ein Potenzial über einem definierten Wert den Zustand 1 und eine Spannung unter einem bestimmten Wert den Zustand 0 besitzt. Dadurch bekommt jede Spannung in einem Rechner zwei interpretierte Zustände und die abzählbare digitale Zahlenmenge wird durch mehrere parallele Leitungen dargestellt. Ein derartiges Signal kann zu beliebigen Zeiten verändert werden. Ein analoges Signal, das sich zu jedem Zeitpunkt ändern kann, aber nur zunächst einmal zwei diskrete Zustände (0 und 1) annimmt, soll im Folgenden „logisches Signal“ oder zu Ehren von Edward Boole „boolesches Signal“ heißen.

Definition 4.1.1 – Logisches Signal

Eine zeitlich veränderliche Größe oder Variable, die zu jedem beliebigen Zeitpunkt ausschließlich Werte zwischen 0 und 1 annehmen kann heißt logisches Signal.

Digitales Signal

4

Die Abb. 4.12 zeigt eine analoge elektrische Spannung, die logisch interpretiert wird. Zunächst einmal wird eine Grenzspannung U_b eingeführt. Ist die Spannung des analogen Signals größer als U_b , wird diese als logische 1 interpretiert, ist die Spannung kleiner, als logische 0. Das logische Signal wechselt dann seinen zeitabhängigen Zustand zu den Zeitpunkten t_1 , t_2 , t_3 , t_4 und t_5 . Ändern sich die Eingangsvariablen einer Schaltfunktion über die Zeit, ist das Ergebnis der Funktion nicht nur von den logischen Werten der Variablen abhängig, sondern auch von der Zeit selbst. Aus der Schaltfunktion $y = f(x_1, x_2, \dots, x_n)$ wird ein logisches Gatter $y(t) = f(t, x_1, x_2, \dots, x_n)$.

4.1.2 Festkörperelektronik

Mit dem Aufkommen der drahtlosen Nachrichtentechnik nach dem experimentellen Nachweis elektromagnetischer Wellen stieg der Bedarf an Bauelementen, die den Fluss von Elektronen regeln. Diese Bauteile beruhten zunächst auf makroskopischen elektronischen Effekten. Bekannt ist die Elektronenröhre zur Modulation und Verstärkung elektrischer Ströme. Da es sich bei der Röhre um Glaskolben, in denen ein Vakuum herrschte, handelte, waren diese mechanisch sehr empfindlich. Schon am Anfang des 20. Jahrhunderts begannen Phy-

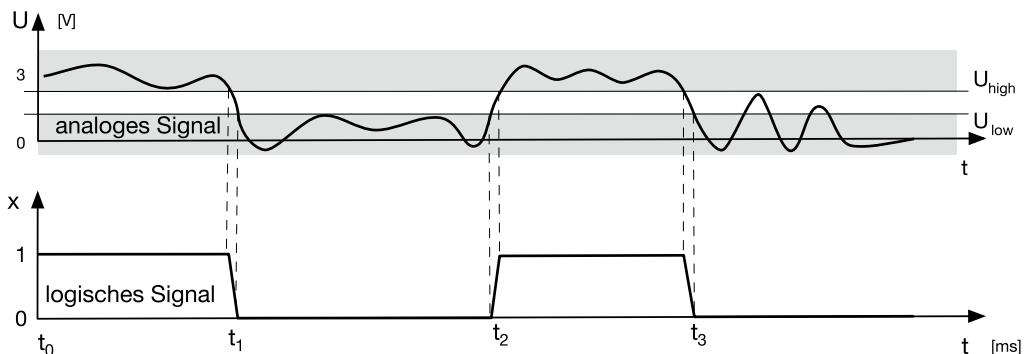


Abb. 4.12 Bildung eines logischen oder booleschen Signals

siker und Elektrotechniker nach physikalischen Effekten zur Verstärkung von Signalen in Festkörpern zu suchen. Dadurch ist es heute möglich, elektronische Schaltungen monolithisch integriert in einem Halbleiter zu implementieren. Die Grundlagen dieser Technik werden im folgenden Kapitel qualitativ vorgestellt.

4.1.2.1 Schalter in Festkörpern

Automatisch in Abhängigkeit von einem elektrischen Strom arbeitende elektrische Schalter sind Relais. Ein solches Bauteil nutzt die elektrisch-magnetische Wechselwirkung. Durch den Stromfluss einer Spule im primären Stromkreis wird ein Magnetfeld erzeugt, und ein metallischer Schalter des sekundären Stromkreises wird durch das Feld angezogen und so geschlossen. Aufgrund der Massenträgheit der Mechanik sind derartige Bauelemente langsam. Zusätzlich sind Relais groß, und verbrauchen viel Strom. Insbesondere in Rechnern müssen sehr viele gleichartige Schalter verbaut werden. Daher sind mit Relais realisierte Rechenanlagen voluminös und schwer. Bedingt durch ihre mechanische Konstruktion haben Relais nicht die erforderliche Zuverlässigkeit, und so ist die Wahrscheinlichkeit groß, dass im Betrieb immer wieder einzelne Relais ausfallen.

Moderne Rechner arbeiten nicht mehr mit elektromechanischen Schaltern, sondern mit elektronischen. Das sind Bauelemente, die die elektronischen Eigenschaften von Festkörpern nutzen. Diese entstehen durch die Wechselwirkung der Ladungsträger mit dem Kristall des Festkörpers. In diesen Bauelementen fließt zwar immer noch ein Strom, die logischen Zustände werden aber nicht wie im Relais mithilfe eines Magnetfeldes dargestellt. Vielmehr dienen elektrische Ladungen, die in speziellen Strukturen der elektronischen Bauelemente festgehalten werden, als Repräsentation der Zahlen. Da im Gegensatz zu einem elektromechanischen Baustein keine Massen mehr bewegt werden, schalten elektronische Bauteile schneller und verbrauchen dabei weniger Energie. Auch sind in Festkörpern integriert hergestellte Schalter um Größenordnungen zuverlässiger als Relais, da sie alle auf einem einzigen Kristall aufgebracht sind. Daher sind sie unempfindlich gegen mechanische Störungen.¹

Der österreich-ungarische Physiker Julius Edgar Lilienfeld (1882–1963, □ Abb. 4.13) beschrieb bereits im Jahr 1925 die Konstruktion eines Feldeffekttransistors. Das entsprechende Prinzip wurde 1934 von dem deutschen Physiker Oskar Heil

Relais

Elektrischer Schalter



UNBEKANNT, Wikimedia Commons

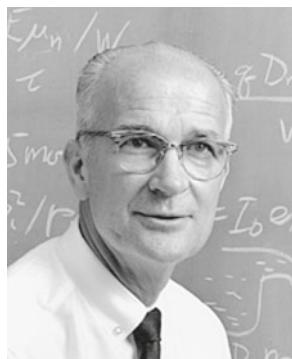
□ Abb. 4.13 Julius Lilienfeld

Halbleiterphysik

¹ So können sich beispielsweise keine Käfer oder andere Insekten mehr zwischen den Relais verfangen und so zu einem Bug führen. Was nicht heißt, dass es in modernen Rechnern keine Bugs gibt.

zum Patent angemeldet. Weder Lilienfeld noch Heil bauten aber einen funktionsfähigen Transistor. Als Konrad Zuse seine ersten Rechenanlagen konstruierte, waren daher monolithisch (altgriech. „monólithos“ = einheitlicher Stein, in diesem Zusammenhang ein Einkristall) integrierte elektronische Schalter technisch noch nicht realisierbar. Zur Zeit von Zuse wurden in der Physik aber intensive Anstrengungen zum Verständnis der Eigenschaften von Festkörpern unternommen. Im Jahr 1945 wurde an den Bell Laboratories eine Abteilung Festkörperphysik mit einer Forschungsgruppe Halbleiter gegründet. Ziel war es, neuartige Bauelemente für die erwartete hohe Nachfrage nach nachrichtentechnischen Geräten zu entwickeln. Dazu versuchte man unter der Leitung des Physikers William Bradford Shockley (1910–1989, □ Abb. 4.14) die elektronischen Eigenschaften von Halbleitern theoretisch und experimentell zu verstehen. Ziel war es, Bausteine nach dem Vorbild der Patente Lilienfelds und Heil zu bauen. Dabei konnten sie auf Arbeiten des amerikanischen Physikers Karl Lark-Horovitz (1892–1958) zurückgreifen. Horovitz hatte mit seiner Forschungsgruppe Spitzendioden zum Einsatz in Radar-anlagen entwickelt. Diese wurden daher bereits industriell gefertigt und waren leicht zu beschaffen. Mit den Spitzendioden führten die Physiker um Shockley zahlreiche Experimente zum Verständnis von Halbleitern durch. Bei einem dieser Versuche entdeckten die Physiker Walter Houser Brattain (1902–1987) und John Bardeen (1908–1991) 1947, dass die Schaltung nicht nur den Stromfluss richtete, sondern auch ein Verstärkungseffekt erzielt wurde. Der erste Bipolartransistor (Transistor = „transfer resistor“) war gefunden oder besser entdeckt.

Transistoren



Chuck Painter,

□ Abb. 4.14 William Shockley

Mit dem Transistor wurde ein zuverlässiges elektronisches Bauelement gefunden, das in vielen Anwendungen der Nachrichtentechnik die unzuverlässigen Vakuumröhren ablöste. Ausgehend von den Bemühungen in den 1950er-Jahren, Relais durch Röhren zu ersetzen, bot sich nun an, einen Rechner direkt mit Transistoren aufzubauen. Mittlerweile ist es üblich, Milliarden von Transistoren in einem einzigen Kristall monolithisch zu integrieren. Da alle Transistoren einer derartigen Schaltung gleichzeitig unter denselben Bedingungen gefertigt werden, zeichnen sich diese integrierten Schaltungen durch eine hohe Zuverlässigkeit aus.

Die technische Grundlage moderner Rechenanlagen ist der Bau logischer Funktionen mithilfe von Halbleiterbauelementen. Da die Halbleiterschaltungstechnik die Eigenschaften von Ladungsträgern, speziell von Elektronen nutzt, heißt dieses Gebiet der Elektrotechnik Halbleiterelektronik – im Gegensatz zur Elektronik, welche auch Bauelemente kennt, die nicht aus Halbleitern bestehen (z. B. Vakuumröhren). Inzwischen sind die Schalter in den integrierten Schaltungen mikrosko-

pisch klein, sodass von der Mikroelektronik gesprochen wird. Bei minimalen Strukturgrößen im Nanometerbereich findet sich in den letzten Jahren auch häufig der Begriff Nanoelektronik. Die mikroelektronische Schaltungstechnik beruht auf elektronischen Festkörperbauteilen. Deren Funktion lässt sich sehr gut mithilfe der Festkörperphysik an den Grenzflächen unterschiedlicher Materialien beschreiben. Aus grundlegenden Annahmen dieser Physik lassen sich die Eigenschaften der konstruierten Bauteile berechnen. Im folgenden Kapitel sollen die notwendigen Grundlagen qualitativ und möglichst anschaulich dargelegt werden. Für physikalische Details und quantitative Berechnungen sei auf einführende Literatur zur Festkörperphysik und Halbleiterelektronik verwiesen.

4.1.2.2 Halbleiter

In der Elektrotechnik wird zwischen Leitern und Isolatoren unterschieden. Erstere transportieren Ladungsträger und leiten so den Energie- oder Informationsfluss, Letztere verhindern die Übertragung der Elektrizität. In der Technik spielen sowohl leitende als auch isolierende Werkstoffe eine große Rolle, entsteht doch der unmittelbare Nutzen elektrischer Geräte erst aus dem Zusammenspiel der Materialien. Typische Leiter sind Metalle wie Kupfer, Eisen oder Aluminium. Bekannt für ihre isolierende Eigenschaft sind Holz, Glas oder Kunststoffe. Neben Leitern und Isolatoren gibt es noch eine weitere Klasse von Materialien: die Halbleiter. Eher zufällig entdeckte 1874 der deutsche Elektrotechniker und spätere Nobelpreisträger Ferdinand Braun (1850–1912) den Gleichrichtereffekt. Er beobachte, dass bestimmte Materialien den elektrischen Strom nur in eine Richtung passieren ließen. Aus dieser Entdeckung leitete sich später die in der Funktechnik bedeutende Spitzendiode ab. Ausgehend von diesen Arbeiten begann man im frühen 20. Jahrhundert die Leitfähigkeit unterschiedlicher Materialien systematisch zu erforschen. Dabei zeigte sich, dass der Gleichrichtereffekt an den Grenzflächen der Stoffe entstand. Während Leiter und Isolatoren leicht von einander zu unterscheiden sind, ist die Unterscheidung zwischen Leitern und Halbleitern nicht offensichtlich. Fließt bei einem Leiter der elektrische Strom besser, je kälter der Festkörper ist, ist dieser Effekt bei Halbleitern genau umgekehrt. Je wärmer ein Halbleiter ist, umso reibungsloser leitet er. In den folgenden Abschnitten soll gezeigt werden, warum das so ist.

Materie aus der Gruppe der Leiter oder Halbleiter kristallisiert, d. h., in festem Zustand bilden die Atome des Materials einen Kristall. Diese bestehen aus regelmäßig angeordneten Atomen: einem Kristallgitter. Die Atome selbst sind aus einem

Atome und
Elektronen

4**Bohrsches Atommodell****Atommodell**

Kern aus Protonen und Neutronen und einer Hülle aus Elektronen aufgebaut. Träger der für den Stromfluss erforderlichen Ladung sind die Elektronen. Der Atomkern kann in den weiteren Betrachtungen als positiv geladener Atomrumpf aufgefasst werden. Die an diesen gebundenen, herumflitzenden Elektronen bilden eine Art einhüllende Wolke. Insbesondere ist die Elektronenhülle für die chemischen Eigenschaften verantwortlich und bestimmt die Bindungseigenschaften der Atome in einem bestimmten Material.

Anfang des 20. Jahrhunderts stellte man sich ein Atom wie ein kleines Planetensystem vor. Um den Kern kreisen die Elektronen auf festen Bahnen. Allerdings ergab sich in diesem Modell ein Widerspruch zur bekannten Physik, denn beschleunigte Ladungen müssen Energie abgeben oder aufnehmen. Auf einen Ladungsträger auf einer Kreisbahn wirkt die Zentripetalbeschleunigung. Der dänische Physiker Nils Bohr (1885–1962) postulierte daher, dass sich Elektronen auf vorgegebenen Bahnen konstanter Energie um den Atomkern bewegen. Dieses Postulat ergab sich zu der damaligen Zeit aus der Beobachtung fester Spektrallinien. Das heißt, die Atome absorbieren nur Licht spezieller Wellenlänge.

Mit diesem einfachen Modell eines Atoms, bestehend aus einem positiv geladenen Kern, umgeben von in Bewegung befindlichen Elektronen, lassen sich die grundlegenden elektronischen Eigenschaften eines Kristalls beschreiben. Zunächst einmal bewegen sich die Elektronen auf Kreisbahnen um den Kern. Der positive Kern zieht die negativ geladenen Elektronen an, und ihre kinetische Energie sorgt dafür, dass sie auf der Kreisbahn bleiben. Ein Bahnwechsel eines Elektrons kann nur durch eine feste, gequantele Energiemenge erfolgen. Die Elektronen bewegen sich also auf festen vorgegebenen Schalen. Im Gegensatz zu Körpern in einem Sonnensystem können Elektronen damit nicht auf beliebigen Bahnen kreisen. Das bohrsche Atommodell postuliert diese festen, für Elektronen zwingenden Energieniveaus, erklärt diese jedoch nicht. Eine Erklärung lieferte einer der Begründer der modernen Quantenmechanik. Der österreichische Physiker Erwin Schrödinger (1887–1961) postulierte, dass unter der Annahme von Materiewellen Beugungseffekte in den Atomen zu berücksichtigen sind. Ein klassisches mechanisches Modell zur Beschreibung der unterschiedlichen Elektronenschalen muss also versagen. Nimmt man zunächst an, dass ein Elektron eine Materielle Welle bildet, und erlaubt man auf geschlossenen Kreisbahnen keine Amplitudensprünge, kann die berühmte Schrödinger-Gleichung aus einer stehenden Welle um den Atomkern abgeleitet werden. Da Schwingungen mit dieser Eigenschaft nur auf bestimmten Vielfachen der Wellenlänge möglich sind, bewegen sich Elektronen auf festen, gequantelten

Bahnens, oder in anderen Worten, sie können bloß bestimmte Energieniveaus um den Kern einnehmen.

Die bisherigen Betrachtungen gingen von einem Atom im Vakuum aus. Kristalle bestehen jedoch aus vielen Abermillionen Atomen. Werden Atome zusammengebracht, bilden die Elektronenschalen breite Bänder aus (Abb. 4.15). Kann man bei wenigen Atomen noch getrennte Energieniveaus beobachten, verschmieren diese mit zunehmender Anzahl der Atome. Innerhalb dieser Bänder können die Elektronen beliebige Energien annehmen. Die Vorstellung des Bändermodells in Metallen geht auf Felix Bloch (1905–1983) und Hans Bethe (1906–2005) zurück. Von besonderer Bedeutung ist das Valenzband. Bei einem Atom ist es das höchste Energieniveau, in dem sich noch Elektronen befinden. Bei vielen Elementen ist es das erste Niveau, das nicht vollständig mit Ladungsträgern besetzt ist, obwohl die Anzahl der Protonen im Kern dies zulassen würde. Ausnahme sind die Edelgase; bei diesen ist das Valenzband voll. Elemente mit nicht vollständig besetztem Valenzband gehen leicht Bindungen ein. Atome binden Elektronen anteilig und können so die Energie im Kristall oder einer anderen Verbindung minimieren. Dieser Effekt führt zu chemischen Reaktionen zwischen den Elementen oder eben zur Kristallbildung. Daher stammt auch der Name des Bandes: „Valenzband“ (lat. „valentia“ = Kraft).

Um elektrischen Strom leiten zu können, sind in einem Kristall freie Ladungsträger notwendig. Bei Metallen sind nicht alle Elektronen im Valenzband an der Bindung beteiligt, sodass freie Teilchen übrig bleiben. Diese stehen für den Stromfluss zur Verfügung. Bei einem Isolator sind alle Elektronen im Kristallgitter gebunden. Führt man dem Kristall jedoch Ener-

Bändermodell

Leiter

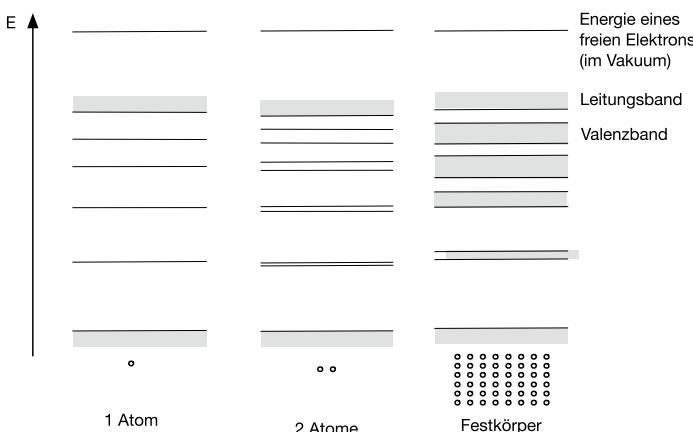


Abb. 4.15 Bändermodell

4

Isolatoren**Halbleiter****Siliziumkristall**

gie zu, z. B. durch Erwärmung und in Form von Licht, können diese aus der Bindung der Atome gelöst werden. Dazu wird eine bestimmte Arbeit benötigt. Oberhalb des Valenzbandes existiert somit ein weiteres Energieband, das Leitungsband der freien Elektronen. Je größer die Energiedifferenz zwischen dem Valenz- und dem Leitungsband ist, umso stärker isoliert das Material. Umgekehrt gilt, wenn die Lücke kleiner ist, desto einfacher lassen sich freie Ladungsträger erzeugen und umso besser fließt der Strom beim Anlegen einer Spannung. Bei Leitern wie z. B. Eisen, Kupfer oder Aluminium überlappen sich das Leitungs- und das Valenzband. Es stehen jederzeit genügend Ladungsträger für den elektrischen Stromfluss zur Verfügung. Isolatoren zeichnen sich dadurch aus, dass die Bandlücke zwischen Valenz- und Leitungsband mindestens 8 eV groß ist (Abb. 4.16).

Für den Bau von Rechnern ist eine weitere Art Werkstoffe interessant, die Halbleiter. Diese besitzen eine Bandlücke, die kleiner ist als 2,5 eV. Das in der Halbleitertechnik wichtige Silizium hat eine Energiedifferenz von 1,12 eV und Germanium, ein anderer Halbleiter, von 0,66 eV. Diese Bandlücken lassen sich im Experiment ausmessen und theoretisch vorhersagen. Wie bereits erwähnt, steigt in einem Metall der elektrische Widerstand mit zunehmender Temperatur, da die freien Elektronen durch die stärkere Wärmebewegung der Atomkerne gestreut werden. In einem Halbleiter sinkt dieser ohmsche Widerstand, da durch die Wärme dem Kristall Energie zugeführt wird und so Ladungsträger aus dem Valenzband in das Leitungsband gehoben werden und anschließend den Stromfluss bilden.

Die Abb. 4.17 zeigt beispielhaft die 2-dimensionale Struktur eines Siliziumkristalls. Die positiv geladenen Atomrumpfe sind als Kreise dargestellt, die äußere Schale der Elektronenhülle als gestrichelter Kreis. Ein Siliziumatom hat 4 Valenz-

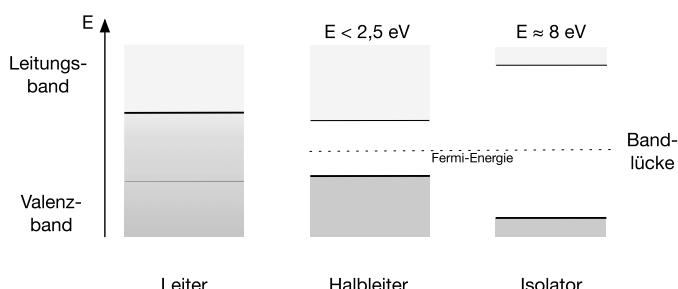


Abb. 4.16 Leiter, Halbleiter und Isolatoren im Bändermodell

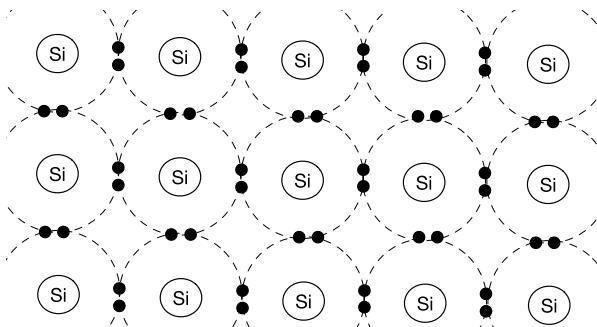


Abb. 4.17 Siliziumkristall

elektronen. Mit jeweils 4 Nachbaratomen können im Kristallgitter die Valenzelektronen zwischen den Atomen geteilt werden, sodass jedes Siliziumatom das Valenzband wie ein Edelgas mit 8 Elektronen voll besetzen kann. Über die gemeinsamen Elektronen zweier Siliziumatome wird die Bindung hergestellt, und die Energie im Festkörper wird minimiert. Im Siliziumkristall sind bei Raumtemperatur alle Valenzelektronen gebunden, das Leitungsband ist leer und elektrischer Strom kann bei Anlegen einer Spannung nicht fließen.

Durch Verunreinigung mit Fremdatomen kann die Leitfähigkeit eines Halbleiters verbessert werden. Bringt man beispielsweise Phosphor in den Siliziumkristall ein, stehen anschließend mehr freie Teilchen zur Leitung des Stroms zur Verfügung. Im Gegensatz zu Silizium besitzt Phosphor 5 Elektronen im Valenzband, und wenn jeweils 4 davon die Bindung mit dem Silizium eingehen, bleibt am Ende ein freier Ladungsträger übrig (Abb. 4.18). Das 5. Elektron ist nun lose an den Phosphor gebunden. Um es zu befreien, ist eine Energie von 0,45 eV nötig. Im Bändermodell ergibt sich dadurch ein weiteres Energieniveau, das „Donatorniveau“ heißt.

Im Gegensatz dazu würde eine Verunreinigung mit Bor, das nur 3 Valenzelektronen besitzt, dazu führen, dass in der Bindung ein Elektron fehlt (Abb. 4.19). Derartige Fehlstellen heißen Löcher, und Bindungselektronen können leicht mit geringer Energie in das Loch springen. Da ein Loch so scheinbar durch den Kristall wandert, heißt diese Art des Stromflusses Löcherleitung. Ein wenig Überlegung führt zu der Einsicht, dass freie Elektronen durch ein angelegtes Feld leichter bewegt werden, als wenn erst Elektronen aus einer Bindung entfernt werden müssen, um die Lücke im Valenzband eines anderen

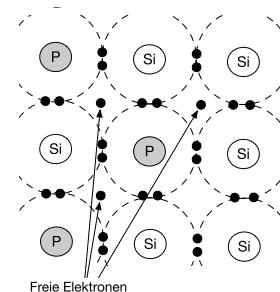


Abb. 4.18 Siliziumkristall n-dotiert

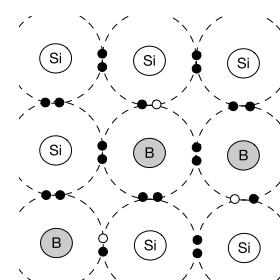


Abb. 4.19 Siliziumkristall p-dotiert

Atoms zu besetzen. Diese Eigenschaft der zwei unterschiedlichen Ladungsträger kann gemessen werden und heißt Ladungsträgerbeweglichkeit. Die Beweglichkeit beeinflusst unmittelbar die Leitfähigkeit des Halbleiters und negativ dotierte Halbleiter leiten den elektrischen Strom besser. Die unterschiedliche Beweglichkeit der Ladungsträger in einem Festkörper hat direkten Einfluss auf den Entwurf integrierter Schaltungen und somit auf die Konstruktion von Rechnern.

Einen Siliziumkristall gezielt mit ausgewählten Elementen zu verunreinigen, wird Dotieren genannt. In der Halbleiterfabrik wird dieses in speziellen Öfen vorgenommen. In solch einem Ofen wird Silizium bis auf 1200 °C erwärmt und einer mit dem Dotierelement gesättigten Gasatmosphäre ausgesetzt. Die Fremdatome diffundieren dann in den Halbleiter ein und können durch die nahe am Schmelzpunkt des Siliziums gehaltene Wärme in den Kristall eingebaut werden. Mit den Prozessparametern Dauer und Temperatur kann einfach bestimmt werden, wie tief die Diffusion erfolgt. Daher lassen sich mithilfe des Dotierens die elektronischen Eigenschaften eines Halbleiters genau festlegen.

Die elektronischen Eigenschaften von Halbleitern lassen sich theoretisch mithilfe der Ladungsträgerdichte gut verstehen und sogar berechnen. Die Verteilung von Ladungsträgern auf das Valenz- oder Leitungsband kann mit einer Verteilungsfunktion beschrieben werden. Im Wesentlichen geht es darum, eine Verteilungsdichte von Ladungen zu bestimmen. In der klassischen Physik oder in der Thermodynamik wird die Energieverteilung von Teilchen durch die Boltzmann-Funktion beschrieben. Dabei ist die Verteilung abhängig von der Temperatur T und der Boltzmann-Konstante k . Für sehr kleine Teilchen wie die Elektronen ist diese Betrachtung nicht mehr zulässig. Man muss die Energieverteilung in einem beliebigen Raum quantenmechanisch herleiten [Gerthsen, 16. Auflage S. 873 ff]. Das Ergebnis einer solchen Betrachtung ist die Fermi-Verteilung.

Ladungsträgerkonzentration

v

Definition 4.1.2 – Fermi-Verteilung im thermodynamischen Gleichgewicht

$$F(E) = \frac{1}{1 + e^{\frac{E-E_f}{kT}}} \quad (4.15)$$

Eine Besonderheit ist die Fermi-Energie E_f . Angenommen, ein quantenmechanisches System besteht aus N Elektronen im Grundzustand, dann ist die Fermi-Energie die Energie des höchsten besetzten Energieniveaus [Kittel, Festkörperphysik, 15. Auflage S. 153]. Um Halbleiterbauelemente zu verstehen, ist es daher erforderlich, die Fermi-Energie in einem Halbleiterkristall zu berechnen. Dies geschieht unter der Annahme, dass die Anzahl freier Ladungsträger im Leitungsband gleich der Zahl der Fehlstellen im Valenzband ist. Mit anderen Worten, durch Energiezufuhr brechen Elektronen aus der Kristallbildung und gelangen in das Leitungsband. Dadurch hinterlassen sie ein Loch. Die Konzentration dieser und der Elektronen im Valenz- oder Leitungsband muss also gleich sein. Damit kann unter der Annahme, dass bei Normaltemperaturen die Energie der Bandlücke des Halbleiterkristalls immer sehr viel größer als kT ist, mithilfe der Fermi-Verteilung und dem Einsetzen der Bandlückenenergie in diese die Fermi-Energie im Halbleiter berechnet werden. Durch die vorher getroffenen Überlegungen gilt:

$$F(E_f) = \frac{1}{1 + e^{\frac{E_f - E_f}{kT}}} = \frac{1}{2} \quad (4.16)$$

Die Verteilung der Elektronen im Valenz- und Leitungsband ergibt sich im Modell mithilfe der Fermi-Funktion. Bei einem undotierten Halbleiter sind alle Elektronen im Valenzband und keine freien Ladungsträger im Leitungsband vorhanden. Statistisch können aber in Abhängigkeit von der Temperatur Teilchen in das Leitungsband gelangen. Um den Zusammenhang der Verteilung mit dem Bändermodell zu verdeutlichen, sind Abszisse ($x = E$) und Ordinate ($y(x) = F(E)$) in der Abbildung vertauscht (Abb. 4.20).

Beträgt die Temperatur 0 K, ist die Wahrscheinlichkeit, Elektronen im Leitungsband zu finden, null. Die Verteilungsfunktion ist entsprechend ein Rechteck: Oberhalb der Mitte der Bandkante ist die Wahrscheinlichkeit 0, unterhalb 1. Wird die Temperatur erhöht, steigt die Wahrscheinlichkeit, Elektronen im Leitungsband anzutreffen. Wie bereits ausgeführt, ist die Fermi-Energie in einem undotierten Halbleiter immer exakt in der Mitte der Bandlücke. Dotiert man einen Halbleiterkristall, verändert sich die Fermi-Energie. Bei einem negativ dotierten Halbleiter ist die Wahrscheinlichkeit, Ladungsträger im Leitungsband zu finden, höher, das Fermi-Niveau verschiebt sich Richtung des Leitungsbandes (Abb. 4.21 und 4.22). In der Abbildung ist das mit E_{f-} bezeichnet. Im umgekehrten Fall der positiven Dotierung verschiebt sich das Fermi-Niveau in Richtung Valenzband. Das Fermi-Niveau gibt die Energie zwischen den Leitungsbändern an, in denen hypothetisch

Fermi-Energie

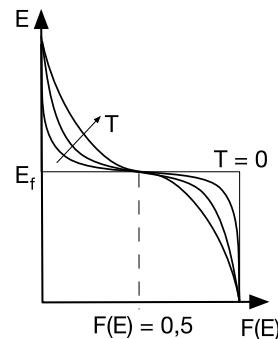


Abb. 4.20 Fermi-Verteilung

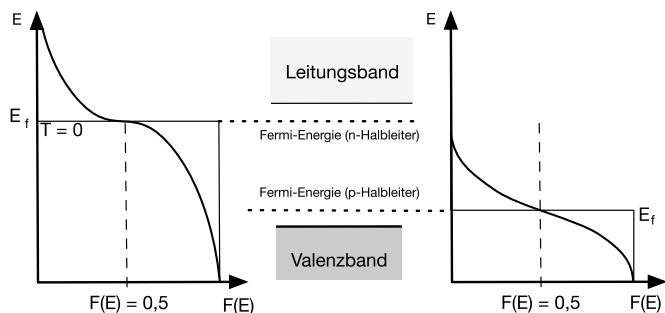


Abb. 4.21 Fermi-Distribution of Doping

tisch die gleiche Anzahl von Löchern und Elektronen vorhanden ist.

4.1.2.3 Halbleitergrenzflächen

Die grundlegende Struktur der meisten Halbleiterbauelemente ist der p-n-Übergang. Dieser entsteht durch Dotierung eines Siliziumkristalls mit Phosphor und mit Bor, sodass positiv und negativ geladener Kristall direkt aneinandergrenzen. Dabei ist bei der Herstellung darauf zu achten, dass der Übergang zwischen den beiden unterschiedlich dotierten Kristallbereichen abrupt ist. Die Abb. 4.23 zeigt ein 2-dimensionales Modell eines solchen steilen p-n-Übergangs mit metallurgischer Grenzfläche. Nach der Herstellung finden physikalische Ausgleichsprozesse statt: Die Elektronen des negativ dotierten Festkörpers driften in den p-dotierten Kristall. Umgekehrt diffundieren die Löcher in den p-dotierten Teil. Abhängig von der mittleren Diffusionslänge, einer physikalischen Eigenschaft der Ladungsträger, rekombinieren Löcher und Elektronen. In einem schmalen Bereich entlang der Grenzfläche zwischen den Materialien bleiben vereinfacht gesagt Boratome ohne Löcher und Phosphoratome ohne freie Elektronen zurück. Statistisch

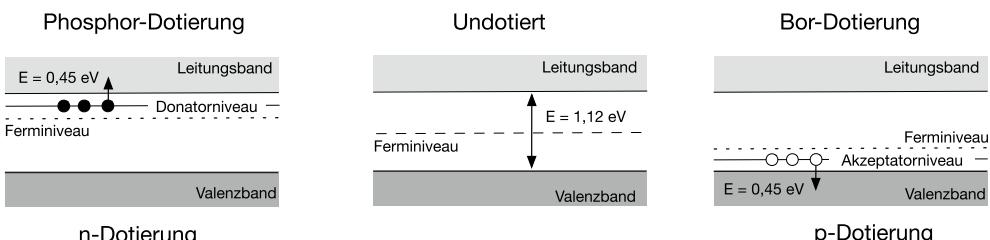


Abb. 4.22 Dotiertes Silizium im Bändermodell

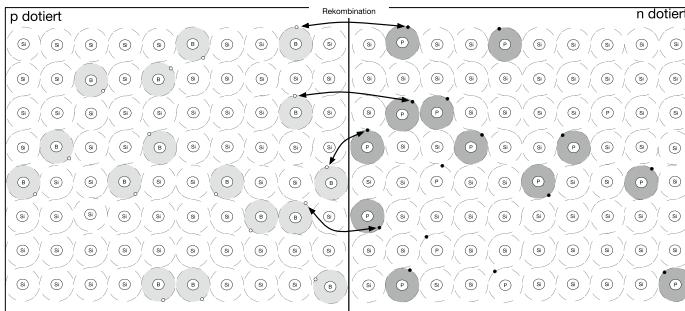


Abb. 4.23 Rekombination an der Grenzfläche des p-n-Übergangs

bedeutet dies, dass in einer Grenzzone im p-dotierten Kristall negativ geladene Ionen zurückbleiben, während im negativ dotierten Teil positive Atomkerne angetroffen werden. Durch diesen Ausgleichsprozess an der Grenzfläche entsteht eine Raumladungszone. Diese Zone wird auch Verarmungszone genannt, weil dort keine freien Ladungsträger anzutreffen sind.

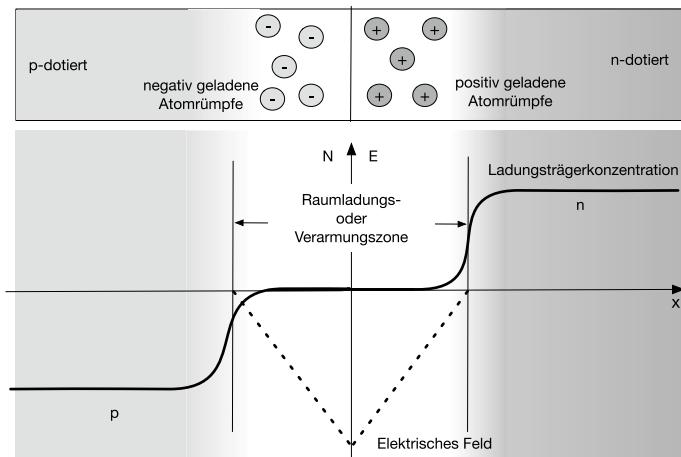
Der p-dotierte Kristall besitzt eine bei der Herstellung festgelegte Konzentration an Donatoren. Die Anzahl der freien Ladungsträger nimmt dann zur Grenzfläche hin rapide ab. Nach einem kurzen Stück im n-dotierten Kristall steigt die Anzahl von Ladungsträgern wieder an. Diesmal sind die Elektronen die Träger der Ladung. Die in der Raumladungszone vorhandenen negativ und positiv geladenen Atomrumpfe erzeugen ein elektrisches Feld (Abb. 4.24). Solange an diese Struktur keine Spannung angelegt wird, fließt auch kein Strom.

Mit der Fermi-Verteilung kann die Anzahl der Ladungsträger im Leitungsband im jeweiligen Teil des Kristalls bestimmt werden. Diese ist wiederum proportional zum Strom, der fließt, wenn ein äußeres elektrisches Feld angelegt wird. Solange dieser Strom und damit auch die Änderung der Ladungsträger bezüglich eines festen Weges null sind, ist die Fermi-Energie im zusammengefügten Festkörper mit dem p-n-Übergang konstant. Die Konsequenz aus einer über den Kristall konstanten Fermi-Energie ist das Verbiegen des Leitungs- und Valenzbandes. Die Abb. 4.25 zeigt einen p-n-Übergang ohne extern angelegte Spannung. Ladungsträger, die von einem Pol des p-n-Übergangs zum anderen wandern wollen, müssen also eine Potenzialdifferenz im Leitungsband überwinden.

Wird eine positive Spannung $+U_{pn}$ an den p-dotierten Körper angelegt, wandern Elektronen aus der Spannungsquelle in den n-dotierten Teil oder werden aus dem p-dotierten Festkörper abgezogen. Dadurch ändert sich die Ladungsträger-

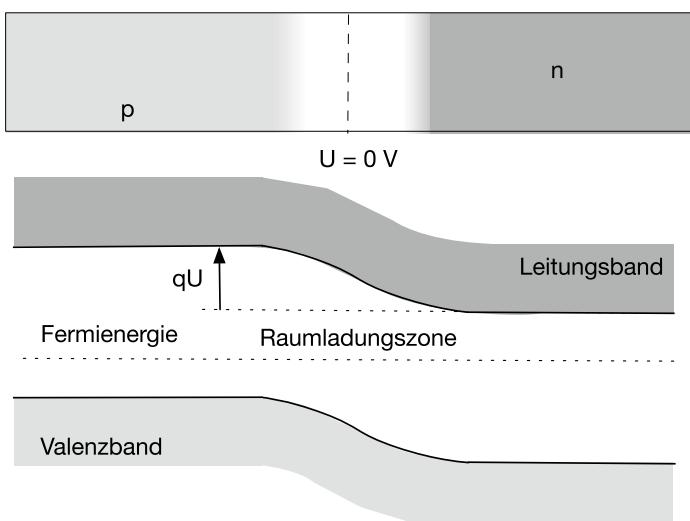
Raumladungszone

p-n-Übergang



■ Abb. 4.24 Entstehung der Raumladungszone

dichte in den beiden unterschiedlichen Seiten des Kristalls. Die Fermi-Energie im n-dotierten Material erhöht sich, die Fermi-Energie im p-dotierten Teil reduziert sich (■ Abb. 4.26). Zwischen den beiden Energien entsteht eine Spannungsdifferenz, die proportional zur angelegten Spannung $+U_{pn}$ ist. Die Potenzialbarriere im Leitungsband verringert sich. Durch Anlegen einer positiven äußeren Spannung werden die Bänder des p-n-Übergangs so verschoben, dass die Barriere im Leitungs-



■ Abb. 4.25 Spannungsloser p-n-Übergang

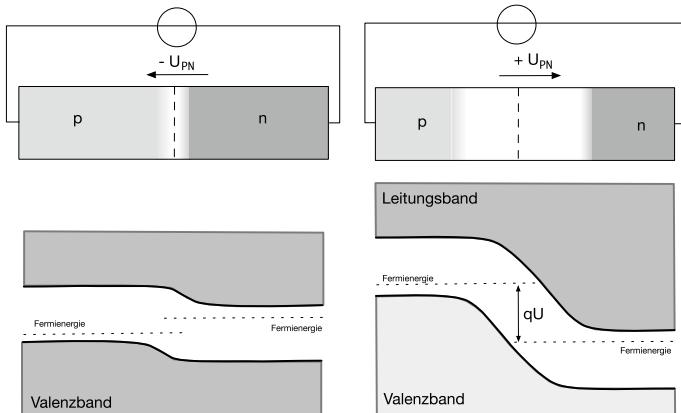


Abb. 4.26 Positive und negative Spannung am p-n-Übergang

band verschwindet und ungehindert Strom fließt. Anders ausgedrückt: Durch die Bereitstellung von Ladungsträgern an beiden Enden des p-n-Übergangs gelangen Ladungsträger in die Raumladungszone und reduzieren dort die durch die Atomrumpfe hervorgerufenen Raumladungen. Dadurch wird der Bereich um die Grenze des Übergangs leitfähig.

Wird dagegen eine negative Spannung an den Halbleiterübergang angelegt, werden Elektronen der n-dotierten Schicht und Löcher der p-dotierten abgesaugt, und die Raumladungszone vergrößert sich. Die Fermi-Energie im p-leitenden Kristall wird größer und die im n-leitenden Teil kleiner. Die Potenzialdifferenz im Leitungsband wird also größer. Elektronen von der n-Zone müssen nun ein hohes Potenzial überwinden. Da die Ladungsträger auf jeder Seite abgesaugt werden, werden thermisch erzeugte Ladungsträgerpaare getrennt, und Elektronen fließen über den n-leitenden Kristall ab, Löcher über den p-leitenden. Zurück bleiben die jeweils negativ oder positiv geladenen Atomrumpfe in der größer werdenden Raumladungszone.

Der p-n-Übergang verhält sich elektrisch wie ein Ventil. Bei positiv angelegter Spannung kann Strom fließen; bei negativer ist kein Stromfluss möglich. Durch die Kombination von einem p-leitenden mit einem n-leitenden Halbleiter wurde ein steuerbarer Widerstand gebaut. In der einen Richtung verhält er sich wie ein Isolator mit einem sehr hohen elektrischen Widerstand, in der anderen wie ein guter Leiter mit einem kleinen ohmschen Widerstand. Wie bereits geschildert, lieferte dieser leicht zu messende Effekt im 19. Jahrhundert die ersten Hinweise auf die Existenz von Halbleitern.

Diode

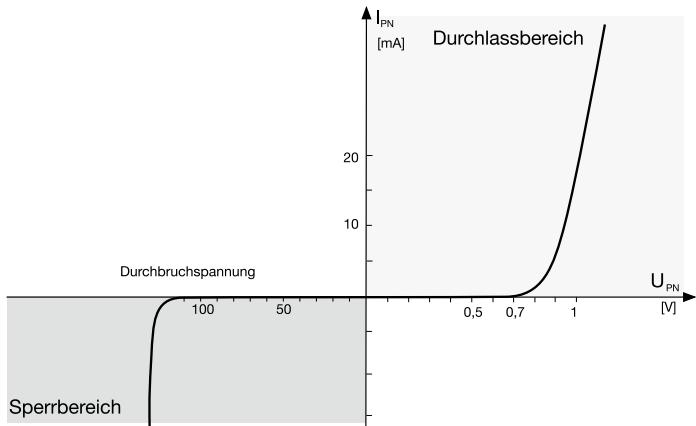


Abb. 4.27 Strom-Spannungs-Kennlinie eines p-n-Übergangs

Das Verhalten elektrischer und elektronischer Bauteile lässt sich anschaulich mithilfe einer Strom-Spannungs-Kennlinie darstellen. Dabei wird die elektrische Stromstärke als Funktion der angelegten Spannung gezeichnet. Die Stromspannung oder U-I-Kennlinie eines p-n-Übergangs zeigt (Abb. 4.27). Etwa bei 0,7 V steigt der Strom in Abhängigkeit von der Spannung exponentiell an. Bereits bei 1 V fließt ein Strom von 15 mA. Im Sperrbereich dagegen müssen erst über 100 V Spannung anliegen, um einen Stromfluss zu ermöglichen. Ab etwa 120 V bricht der p-n-Übergang durch. In diesem Moment ist die zugeführte Energie so groß, dass die Ladungsträger durch die Potenzialbarriere tunneln können und durch Stoßionisation Valenzelektronen aus dem Kristallverbund reißen. Dadurch erhöht sich die Anzahl der zur Verfügung stehenden Elektronen, und ein Stromfluss wird möglich. In der Regel werden Bauelemente auf der Grundlage des p-n-Übergangs dann zerstört. Der p-n-Übergang realisiert eine Diode. Die Diode als Bauelement existierte schon vor der heute gebräuchlichen Siliziumdiode, u. a. auch als Röhre.

4.1.2.4 Der Transistor

Verstärker

Prinzipiell kann mithilfe der Diode bereits eine logische Schaltung realisiert werden. Allerdings wird eine solche Anordnung das Signal nicht verstärken. Um über einen längeren Signalweg durch elektrische Verluste den Signalwert nicht zu verlieren, ist es notwendig, diesen zu stabilisieren. Um also einen möglichst hohen Störabstand zwischen der logischen 1 und der logischen 0 zu realisieren, sind verstärkende Bauelemente notwendig. Gesucht ist daher ein Bauelement, mit dem die Signale in der Rechenanlage verstärkt werden können. Ein solcher

Baustein, realisiert in einem Halbleiterfestkörper mit verstärkender Eigenschaft, ist der Transistor.

Inzwischen gibt es eine Vielzahl verschiedener Typen von Transistoren. Gemein ist allen Bauformen, dass der Stromfluss zwischen zwei Anschlüssen über einen dritten Anschluss durch einen Strom oder eine Spannung gesteuert werden kann. Im Wesentlichen unterscheidet man zwischen zwei Typen von Transistoren, den Bipolar- und den unipolaren Transistoren. Letztere heißen auch Feldeffekttransistoren. Im Bipolartransistor wird der Stromfluss zwischen den Emitter und Kollektor genannten Anschlüssen durch einen Strom am dritten Anschluss, der Basis, verändert. Beim Feldeffekttransistor dagegen steuert eine Spannung am Gate die Leitfähigkeit des Halbleiters zwischen Source und Drain.

Moderne Mikroprozessoren werden mithilfe eines speziellen Herstellungsverfahrens gebaut. Dieses Verfahren heißt Planarprozess und wurde 1958 von dem amerikanischen Physiker Jean Horni (1924–1997) bei Fairchild Semiconductors entwickelt. Auf der Grundlage dieses Herstellungsprozesses baute der Physiker Robert N. Noyce (1927–1990) 1959 die monolithisch integrierte Schaltung. Unabhängig von Noyce hatte bereits Jack Kilby (1923–2005) 1958 eine erste Schaltung in einem Kristall zusammengebaut. Allerdings integrierte Kilby nicht die Verdrahtung der Bauelemente auf dem Chip, sodass erst der Planarprozess die gleichzeitige und monolithische Herstellung vieler gleicher Bauelemente auf einem Siliziumkristall und deren Verschaltung ermöglichte. Das vorherrschende Bauelement in modernen, monolithisch integrierten Schaltungen ist der Feldeffekttransistor, genauer der „Metal-oxide-semiconductor-Feldeffekttransistor“ (MOS-FET), da er einfach zu realisieren ist. Bedingt durch seinen Aufbau speichert er Ladungen. Daher ist er besonders gut zur Verwendung in digitalen Schaltungen wie Mikroprozessoren geeignet.

Ein MOS-FET besteht aus zwei Diffusionswannen, die entweder mit Donatoren oder Akzeptoren verunreinigt werden. Dabei werden die Störatome im Kristall hoch konzentriert. In diesem Zusammenhang spricht man von einer n+- und einer p+-Diffusion. Zwischen diesen beiden Diffusionswannen ist ein Metallstreifen als Gate angeordnet. Dieser liegt aber nicht direkt auf dem Siliziumkristall, sondern zwischen dem Silizium und dem Gate befindet sich eine dünne Isolationsschicht. Diese besteht aus oxidiertem Silizium oder besser Glas (SiO_2). Auch wenn bei modernen MOS-Transistoren das Gate inzwischen aus polykristallinem Silizium und das Gate-Oxid aus anderen Materialien hergestellt werden, ist das Funktionsprinzip gleich. Ein MOS-Transistor kann auf zwei verschiedene Arten gefertigt werden: zum einen mit elektrisch negativ dotierten Wannen in einem positiv leitenden Kristallsubstrat, zum ande-

Transistortypen

Integrierte Schaltungen

MOS-FET

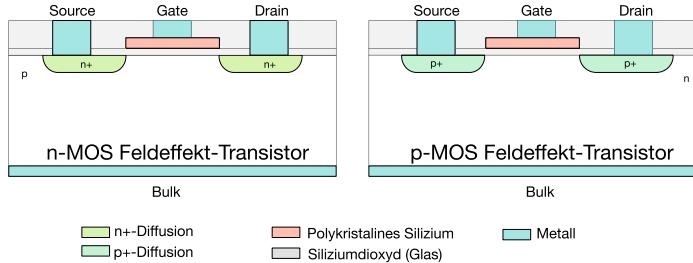


Abb. 4.28 Struktureller Aufbau der MOS-Feldeffekttransistoren

ren mit elektrisch positiv dotierten Wannen in einem negativ leitenden Kristallsubstrat. Im ersten Fall leitet der Transistor Elektronen, im zweiten die Löcher. Es wird also zwischen einem NMOS- und einem PMOS-Transistor unterschieden. Die Abb. 4.28 zeigt den schematischen Aufbau der beiden Varianten und Abb. 4.29–4.30 zeigen die jeweiligen Schaltsymbole. Da beide Transistoren auf dem gleichen Bauprinzip beruhen, soll exemplarisch die Funktion des NMOS-Transistors diskutiert werden.

Die in der Abb. 4.28 dargestellte MOS-Struktur besteht aus den Diffusionswannen, dem isolierenden SiO_2 und einem Gate aus Polysilizium. Das ist Silizium, das direkt auf die Oxidationsschicht abgeschieden wird und dadurch keine Einkristallstruktur annimmt. Gate, Source und Drain werden dann mit einer Metallschicht an andere Schaltungsteile oder die Umgebung des Chips angeschlossen. Das Gate-Oxid ist eine dünne Glasschicht. Die Dicke des Gate-Oxids beeinflusst die elektronischen Eigenschaften des Transistors wesentlich. Das Gate-Oxid muss daher im Herstellungsprozess sehr genau abgeschieden werden. Die anderen dickeren Glasschichten dienen der Isolation. Eine solche integrierte Schaltung besteht im Wesentlichen aus leitenden und isolierenden Strukturen, wobei die Leitfähigkeit der Halbleiter durch die Dotierung konstruktiv beeinflusst werden kann.

Bis auf geringe Leckströme kann zwischen dem Gate, der Source und dem Drain kein Strom fließen. Diese Eigenschaft ermöglicht die Herstellung besonders leistungsschwächer Schaltungen. Ein Stromfluss ist aufgrund der ohmschen Verluste immer mit einer Wärmeentwicklung im Kristall verbunden. Diese Energie muss von der externen Spannungsversorgung bereitgestellt und im Betrieb auch vom Chip wieder abgeführt werden. Moderne Prozessoren mit Millionen von Transistoren sind ohne die MOS-Technik auf engem Raum nicht realisierbar, da die Wärme, die durch die elektrischen Verluste bedingt wird,

MOS-FET-Struktur

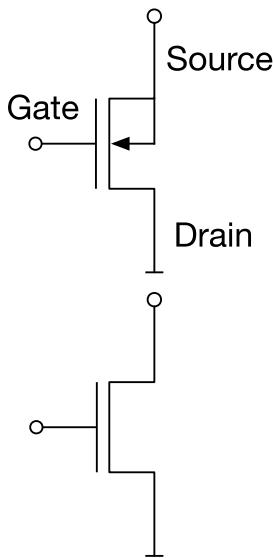


Abb. 4.29 Schaltzeichen NMOS-Transistor

Wärmeentwicklung

nicht abfließen kann d. h. über längere Zeit würde die Schaltung dann zerstört.

Im MOS-Transistor finden sich zwei p-n-Übergänge, einmal die Source-Diffusion zum Substrat (engl. „bulk“) und die Drain-Diffusion ebenfalls in Richtung der Kristallscheibe. Das Schaltzeichen für einen MOS-Transistor deutet dies durch einen kleinen Pfeil an. Entsprechend der Polung sind die Richtungen der Dioden bei NMOS- und PMOS-Transistoren vertauscht. Da das elektrisch korrekte Schaltsymbol aufwendig zu zeichnen ist, gibt es eine alternative Darstellung. Der NMOS-Transistor wird ohne Bulk-Anschluss, der logisch keine Funktion hat, dargestellt und das Gate des PMOS-Transistors erhält ein Negationssymbol.

Die MOS-Struktur ist ein Kondensator und hat kapazitive Eigenschaften. Das aus polykristallinem Silizium oder in früheren Schaltungstechniken aus aufgedampftem Aluminium bestehende Gate ist die eine elektrisch leitende Platte, das Gate-Oxid aus Glas ist die Isolationsschicht und das Substrat ist die zweite gegenüberliegende Platte des Kondensators (Abb. 4.31). Lädt man das Gate auf, werden zum Ausgleich Elektronen oder Löcher aus dem Substrat unter das Gate gezogen. Eine leitende Schicht entsteht dadurch knapp unter dem Gate-Oxid. Dieser Kanal kann mit Ladungsträgern aus der Source des Transistors gefüllt werden, während das Drain genannte Gebiet Ladungsträger aus diesem abzieht. Voraussetzung für diesen Prozess ist, dass zwischen Source und Drain eine Spannung anliegt, die den Stromfluss auslöst. Mit dem Feld des MOS-Kondensators kann so der Strom zwischen Source und Drain, genauer der Strom im Kanal genannten Gebiet der MOS-Struktur reguliert oder moduliert werden.

Interessant wird das Verhalten der Halbleiterstruktur, wenn zwischen Source und Drain eine Spannung angelegt wird. Man kann im Modell noch eine weitere Spannung annehmen: die Spannung zwischen Substrat und Source. In der Praxis ist diese null, da die Source auf das gleiche Potenzial wie der Bulk gelegt wird. Diese Beschaltung zeigt auch das Schaltzeichen des MOS-Transistors. Das qualitative Verhalten des MOS-Transistors wird mithilfe seines Kennlinienfeldes beschrieben.

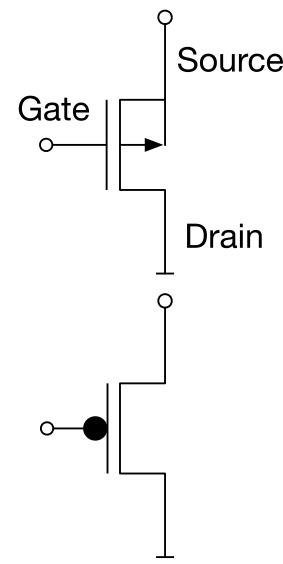


Abb. 4.30 Schaltzeichen PMOS-Transistor

Stromfluss

Kennlinie

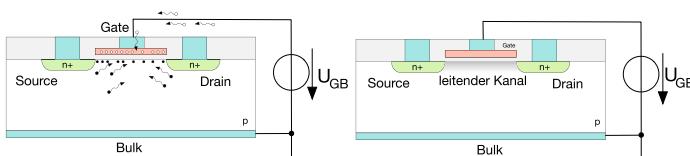


Abb. 4.31 Ausbildung eines leitenden Kanals in der MOS-Struktur

Bedingt durch die drei Anschlüsse des Transistors ist die Strom-Spannungs-Kennlinie keine einzelne Funktion, sondern besteht aus mehreren Kurvenscharen. Eine Kennlinie stellt dabei immer den Zusammenhang zwischen der Source-Drain-Spannung dar und die verschiedenen Scharen zeigen den Einfluss der Spannung zwischen Gate und Source. Liegt eine Spannung zwischen Source und Substrat an, entsteht zwischen dem Diffusionsgebiet eine Raumladungszone. Diese ist in **Abb. 4.32** mit einer gepunkteten Fläche dargestellt. Zwischen Source und Substrat ist der p-n-Übergang in Durchlassrichtung gepolt und zwischen Drain und Substrat in Sperrrichtung. Bedingt durch den Spannungsabfall zwischen Source und Drain ist die Spannung zwischen Gate und Source und zwischen Drain und Source unterschiedlich. Dadurch ist der leitfähige Kanal unter dem Gate nicht über die gesamte Länge des Gate gleich tief. Solange der Einfluss des geladenen Gate bis zum Drain-Anschluss reicht, hat die Strom-Spannungs-Kennlinie einen quadratischen Verlauf. Ab einem gewissen Punkt schnürt der Kanal vor der Drain-Diffusion ab. Der Stromfluss bleibt zwar erhalten, weil die Ladungsträger immer noch von Source zu Drain driftieren, er kann aber nicht mehr gesteigert werden. Der Transistor geht in die Sättigung. Eine steigende Spannung zwischen Source und Drain erhöht den Strom zwischen Source und Drain dann nicht mehr signifikant.

MOS-Transistoren sind in der Lage, einen Strom zu schalten. Mithilfe der Spannung zwischen Gate und Source kann der Transistor ein- und ausgeschaltet werden. Ein NMOS-Transistor ist dann eingeschaltet, wenn an seinem Gate eine

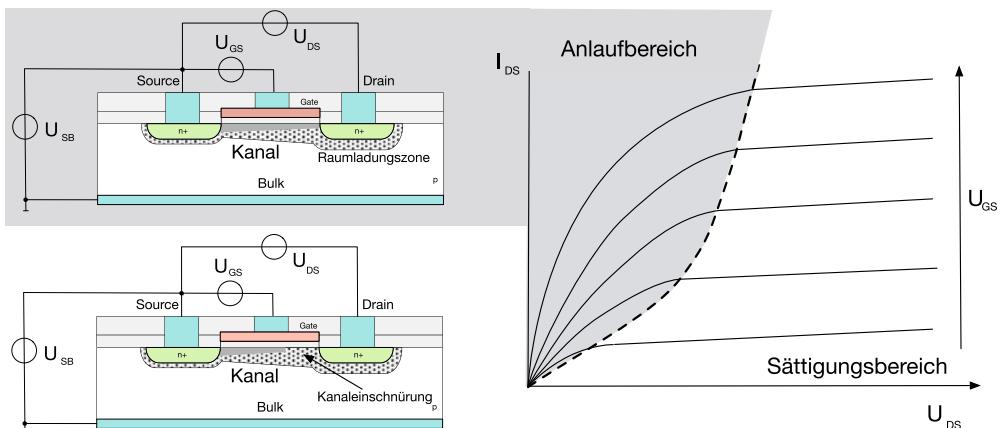


Abb. 4.32 Entstehung des Kennlinienfeldes eines NMOS-Transistors

positive Spannung anliegt, und ein PMOS-Transistor ist eingeschaltet, wenn eine negative Spannung das Gate auflädt.

4.1.2.5 Der Planarprozess zur Herstellung integrierter Schaltungen

Der Planarprozess zur Herstellung monolithischer Schaltungen gründet auf einem Siliziumeinkristall. Dieser wird aus einem Impfkristall gezüchtet: Dazu wird das aus oxidiertem Silizium (Quarzsand) gewonnene Silizium geschmolzen und gereinigt. Im Anschluss wird dann ein kleiner Kristall in eine Schmelze aus reinem Silizium getaucht und langsam herausgezogen. Dabei entsteht eine Säule Silizium mit einheitlicher Kristallstruktur. Nachdem der Kristall noch einmal gereinigt wurde, wird er in Scheiben geschnitten. Auf jedem dieser Wafer (engl. für „dünner Keks“) können Hunderte von gleichartigen Schaltungen hergestellt werden. Dabei wird nur die Oberfläche der Wafer genutzt. Im ersten Schritt wird diese in einer Sauerstoffatmosphäre oxidiert. Anschließend wird die Struktur der Schaltung mithilfe eines fotolithografischen Verfahrens erzeugt. Für jede Ebene wird Fotolack auf den Wafer aufgebracht und dieser dann durch eine Maske belichtet. Dadurch ist es möglich, das Siliziumdioxid selektiv zu ätzen und in einem weiteren Schritt an den geätzten Stellen durch Diffusion mit Fremdstoffen gezielt die Oberfläche zu bearbeiten. Jede Diffusions- oder Verdrahtungsebene der Schaltung erfordert einen derartigen Prozessschritt. Nachdem alle Ebenen gefertigt wurden, wird der Wafer noch einmal mit einer Oxidschicht passiviert und dann in die einzelnen Chips aufgebrochen. In einem Prozessablauf entstehen so auf mehreren Siliziumwafers jeweils Hunderte integrierter Schaltungen (Abb. 4.33). Das erklärt – obwohl die Chipfertigung hohe Kosten und Investitionen erfordert – den niedrigen Preis einzelner Chips. Zwar sind die vielen Prozessschritte aufwendig und die Reinräume der Fabrik teuer, aber in jedem Durchlauf werden Millionen von Einzelschaltungen gleichzeitig gefertigt. In den letzten Dekaden der Halbleiterfertigung ist die Größe der Siliziumwafer durch verbesserte Reinraumtechniken und Herstellungsverfahren kontinuierlich gestiegen. Dadurch war es möglich, trotz hoher Investitionssummen in Fabs, so werden die Chipfabriken im Fachjargon genannt, die Kosten für einzelne Schaltungen weiter zu senken.

Am Beispiel eines NMOS-Transistors soll der wechselseitige Prozess aus Belichten, Entwickeln, Ätzen und Diffundieren genauer beleuchtet werden (Abb. 4.34). Selbstverständlich können auf diese Art viele weitere Transistoren gleichzeitig gefertigt werden. Zunächst wird der Siliziumwafer oxidiert (1). Dazu wird er in einem Ofen bei hohen Temperaturen einer Sau-

Planarprozess

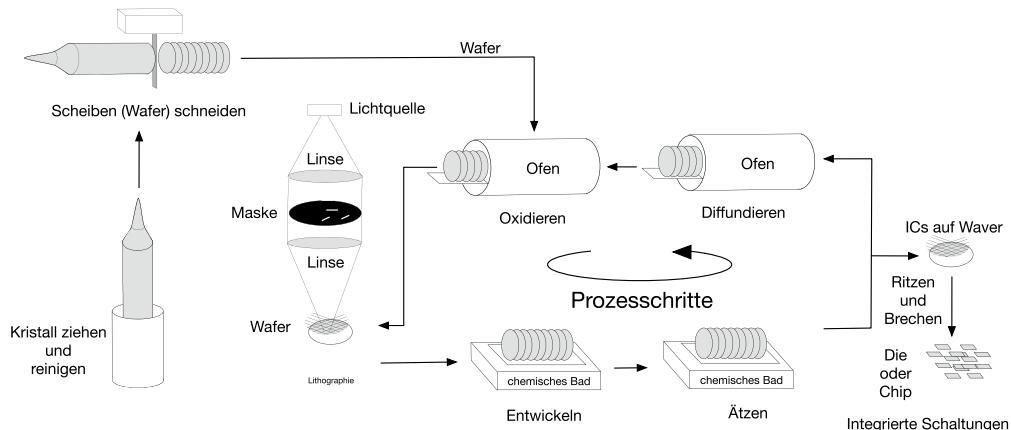


Abb. 4.33 Halbleiterfertigung

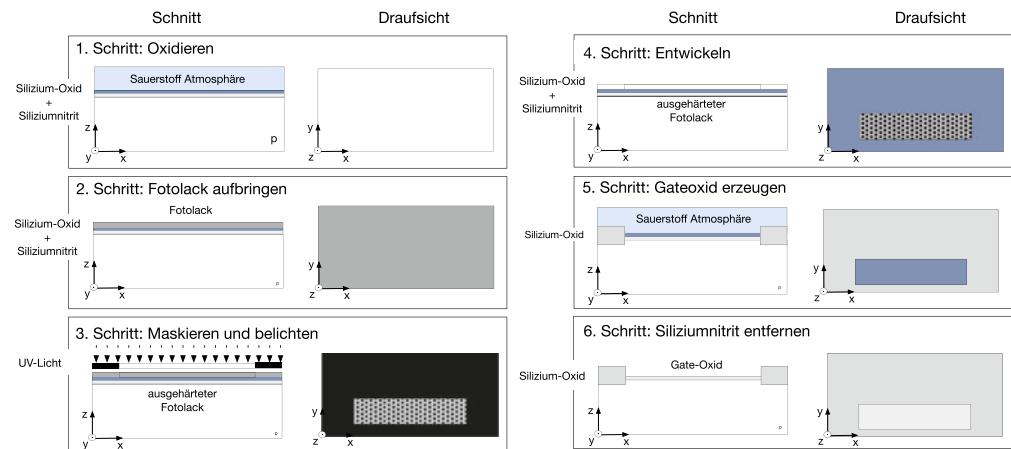


Abb. 4.34 Herstellung des Gate-Oxids

erstoffatmosphäre ausgesetzt. Die Sauerstoffatome dringen in das Silizium ein und reagieren im Kristall. Die Eindringtiefe ist abhängig von der Dauer der Oxidation und der Temperatur. Auf die gleiche Art wird auf der dünnen Siliziumoxidschicht eine Siliziumnitritschicht aufgebracht. Mithilfe von Fotolack kann die Siliziumoxidschicht strukturiert werden: Eine Maske deckt einzelne Bereiche des Wafers ab, und anschließend wird der Lack an den freien Stellen belichtet. Belichteter Fotolack wird hart und maskiert die Siliziumoxidschicht. Nach der fotochemischen Entwicklung bleibt also nur an den belichteten Stellen Lack stehen (4). Im Anschluss wird der Wafer erneut einer Sauerstoffatmosphäre ausgesetzt (5). Dadurch wächst an den Stellen, an denen sich kein Fotolack befindet, das Siliziumoxid. Diese Schicht dient der Isolation der Bau-

elemente gegenüber den späteren Verdrahtungsebenen. Statt des ausgehärteten Fotolacks bildet nun an speziellen, durch die Belichtung festgelegten Stellen eine dünne Siliziumoxidschicht das Gate der herzustellenden Transistoren (5). In einem naschemischen Verfahren wird dann der Fotolack entfernt (6).

Auf die Gate-Oxidschicht wird im nächsten Schritt (Abb. 4.35) Silizium aufgebracht (7). Da Siliziumoxid keine Einkristallstruktur mehr besitzt, entsteht auf dem Wafer eine polymorphe Schicht des Halbleiters. Diese besteht aus lauter kleinen, nicht einkristallinen Siliziumkristallen und heißt daher Polysilizium. Es bildet den Leiter des Gate des späteren Transistors und wird ggf. noch mit Fremdatomen dotiert, um die Leitfähigkeit des Gate zu erhöhen. Nach dem erneuten Aufbringen von Fotolack wird wieder durch eine Maske belichtet (8). Diesmal wird das zukünftige Gate mit abgedeckt. Nach dem anschließenden Ätzen des Polysiliziums und des Siliziumoxids an allen unbedeckten Stellen (9) können Source und Drain diffundiert werden (10). Dazu werden in den Siliziumkristall Fremdatome eingebracht, und zwar nur an den Stellen, die nicht durch Polysilizium oder Oxid bedeckt sind. Durch den Diffusionsprozess wird gleichzeitig die Leitfähigkeit des maskierenden Polysilizium-Gate erhöht. Der so entstandene Transistor justiert sich am Gate selbst. Da die elektrischen Eigenschaften des Bauteils von der exakten Ausrichtung der Diffusionsbereiche und des Gate abhängen, ist damit sichergestellt, dass wenn die Masken nicht perfekt aufeinanderliegen, die MOS-Struktur immer die richtige Geometrie besitzt, selbst wenn es gegenüber anderen Bauteilen ggf. verschoben ist. Diese Selbstjustierung erhöht die Ausbeute und macht die wirtschaftliche Fertigung monolithisch integrierter Schaltungen erst möglich.

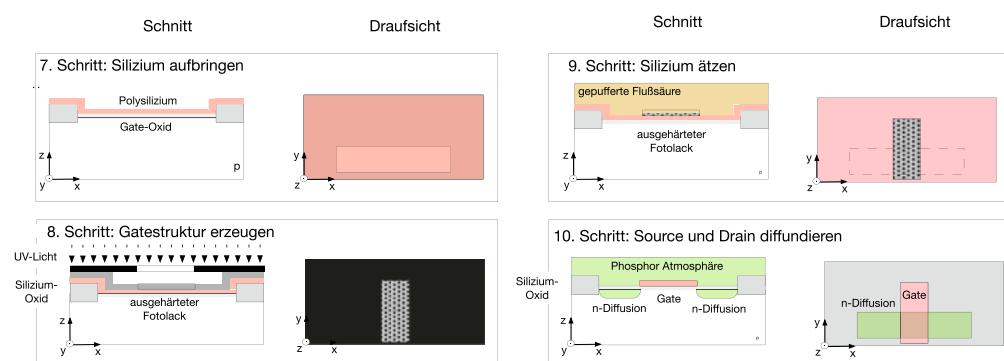


Abb. 4.35 Diffusion des Transistors

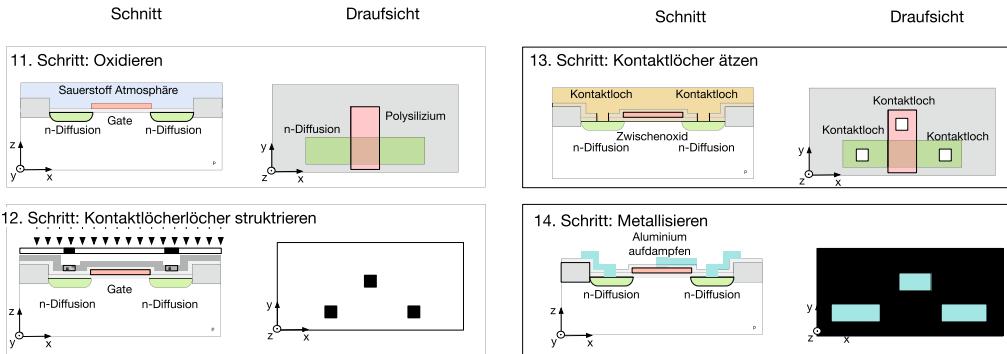


Abb. 4.36 Anschlüsse des Transistors

Zum Schluss (Abb. 4.36) wird wieder eine Siliziumoxid-schicht aufgebracht (11), in die dann nach einem weiteren fotolithografischen Schritt Kontakte zu Gate, Source und Drain geätzt werden können (13). Im letzten Arbeitsablauf wird mit Hilfe der Lithografie die Verdrahtung der Anschlüsse durch Aufdampfen eines Metalls wie Aluminium vorgenommen (14). Nach der Passivierung der Schaltung durch eine erneute Siliziumoxidschicht ist der Transistor fertig. Moderne Halbleiter-schaltungen enthalten inzwischen 4–8 solche Verdrahtungs-ebenen.

Die Abb. 4.37 zeigt den Schnitt durch einen fertigen, im selbstjustierenden Planarprozess hergestellten Transistor. Daneben sind noch einmal alle Masken, die zur fotolithografischen Strukturierung benötigt werden, in Draufsicht dargestellt. Da das Layout integrierter Schaltungen mit CAD-Werkzeugen erstellt wird und diese Werkzeuge es erlauben, eine Ebene halbdurchlässig zu zeigen, werden beim Entwurf häufig alle Strukturen gleichzeitig betrachtet. Um sie unterscheiden zu können, besitzt jede Maske eine eigene Farbe. Diffusionsbereiche sind grün (hellgrün für n-leitend und dunkelgrün für p-leitend), das Polysilizium hellrot, Metall blau und bei einer zweiten Metallschicht gelb dargestellt. Kontaktlöcher sind dann entweder in Weiß mit kreuzenden Linien oder in Schwarz gezeichnet. Diese Farbdarstellung kann je nach Prozess, CAD-Werkzeug oder Halbleiterunternehmen unterschiedlich ausfallen.

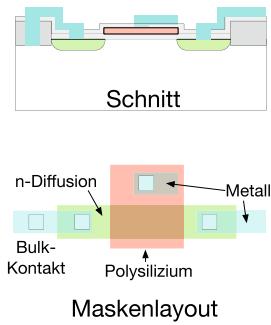


Abb. 4.37 NMOS-Transistor-Layout

4.2 Logische Gatter in MOS-Technik

Mit dem Planarprozess ist es möglich, viele gleichartige Transistoren auf einem kleinen Siliziumchip herzustellen. Wesentliches Merkmal dieses Vorgehens ist die Skalierbarkeit. Die

immer gleiche Grundstruktur kann von Technologieschritt zu Technologieschritt verkleinert werden. Dabei steht der Begriff Technologie für die Summe der einzelnen Parameter des jeweiligen Planarprozesses. Zu diesen gehören neben den elektronischen Eigenschaften, die durch die Diffusionsdauer und -temperatur festgelegt werden, auch die Strukturgrößen wie beispielsweise die Breite und Länge eines Transistor-Gate. Bei der Verkleinerung einer Struktur verändert diese sich nicht, dafür werden andere Parameter des Prozesses angepasst. Alleine durch die Fähigkeit, kleinere Strukturgrößen zu fertigen, kann ein Technologieschritt erfolgen. Bereits entworfene Masken, also Schaltungsstrukturen, können dann wiederverwendet werden, lediglich die Masken werden verkleinert oder nach unten skaliert. Wurden die ersten Mikroprozessoren in $15\text{ }\mu\text{m}$ PMOS-Technologie gefertigt, werden inzwischen Schaltungen im 24 nm-Bereich strukturiert, und 7 nm sind keine Seltenheit.² Zwar werden die Prozesse weiter verfeinert, die Masken und Lithografietechniken werden ausgefeilter, die Reinräume in den Fabriken reiner und die Prozessschritte immer feiner. Das Grundprinzip, eine Schaltung in MOS-Technik zu erstellen, hat sich fast nicht geändert. Daher wird im folgenden Kapitel gezeigt, wie sich die Grundschatungen der Digitaltechnik mithilfe von MOS-Transistoren im CMOS-Prozess aufbauen lassen.

Grundlegendes Element aller digitalen Schaltungen ist das logische Gatter. Diese sind aus Transistoren oder anderen Schaltern aufgebaute Schaltfunktionen. In der Regel werden einfache 2-wertige Gatter wie NICHT, UND, ODER, NAND, NOR konstruiert, um daraus komplexere boolesche Funktionen zu realisieren. Dabei kann man sich zunutze machen, dass sowohl die NAND- als auch die NOR-Funktion jeweils eine elementare Basis der Algebra sind. Das heißt in Konsequenz, dass mit einem in Planartechnik aufgebauten NAND-Gatter alle Schaltfunktionen der Schaltalgebra realisiert werden können. Die Logikgatter werden dann in einer Schaltungsbibliothek abgelegt. Das Gatter ist somit eine Abstraktion der Transistorschaltung. Zu seinen Parametern gehören neben der reinen logischen Funktion auch die Schaltzeiten der elektronischen Bauteile. Mit dem Bauelement Gatter kann der Schaltungsentwurf einer digitalen Schaltung von der Transistorebene auf die logische Ebene der Schaltalgebra gehoben werden. Allerdings nicht vollständig, denn neben der logischen Funktion sind bei der Beschaltung der Gatter die jeweiligen Laufzeiten zu berücksichtigen. Neben den Basiselementen stehen daneben fertige Gatter komplexerer logischer Funktionen zur

Logische Gatter

² Leider veralten heute solche Aussagen schneller, als man sie geschrieben hat.

Bauteilebibliothek**4**

Verfügung: Neben dem XOR- und dem NXOR-Gatter, die mit Tricks auf der Layoutebene sehr kompakt sein können, ist dies das programmierbare logische Array („programmable logic array“, PLA), mit dem sich beliebige logische Funktionen bereits auf der Ebene des Layouts einfach realisieren lassen.

Die wenigsten Entwickler digitaler Schaltungen müssen Transistorschaltungen layouten. In der Regel werden fertige Gatter, deren elektrische Modelle und Layouts in einer Datenbank abgelegt. Diese Schaltung- oder kurz Bibliothek genannten maschinenlesbaren Formate beinhalten alle in einen speziellen Halbleiterprozess gefertigten Bauelemente. Für jeden Prozess existiert somit eine eigene Bauteilebibliothek. Mit grafischen CAD-Werkzeugen (Computer-aided Design) können Schaltungsentwickler auf die Bauteile zugreifen und eine Schaltung entwerfen. Mittlerweile ist dieses Vorgehen jedoch selten geworden, die Schaltungen werden textuell in einer Hardwarebeschreibungssprache beschrieben und dann mit Synthesewerkzeugen automatisch erzeugt. In dem Fall greifen die Synthesewerkzeuge auf die Gatterbibliothek zu, und der Ingenieur realisiert den physischen Aufbau der Schaltung nicht einmal mehr.

4.2.1 NMOS- und PMOS-Inverter

Mithilfe der MOS-Transistoren lassen sich logische Gatter einfach realisieren. Die technisch am unkompliziertesten zu bauende boolesche Funktion ist die einwertige Negation. Das dazugehörige Gatter ist das NICHT (engl. NOT). Um ein NICHT mithilfe eines Transistors zu realisieren, wird eine Schaltung benötigt, die eine positive Eingangsspannung $U_A(t)$ auf Masse zieht, also eine logische 0 erzeugt. Im Gegensatz dazu, liegen 0 V am Eingang des Gatters an, soll die Spannung am Ausgang auf die Versorgungsspannung V_{DD} angehoben werden. Merkmal der Schaltung ist, dass zwischen diesen beiden Zuständen beliebig hin- und hergeschaltet werden kann. Ein solcher Inverter kann mit einem Schalter und einem Widerstand R_P realisiert werden. Dazu wird der Schalter in Reihe mit diesem geschaltet. Zwischen dem Widerstand R_P und dem Schalter wird dann der Ausgang angeschlossen. Der obere Anschluss des Widerstandes wird nun an die Versorgungsspannung geklemmt und der Schalter an die Masse. Bei einem Relais kann dann ein Stromfluss am Eingang der das Magnetfeld erzeugenden Spule den Schalter schließen. In dem Fall wird der Ausgang auf das gleiche elektrische Potenzial wie die Masse gelegt, das sind per definitionem 0 V. Über den Widerstand fließt nun ein Strom $I = \frac{V_{DD}}{R_P}$ zur Masse, und nach dem ohmschen Gesetz fällt dann über diesem die Versorgungsspannung ab. Der Widerstand be-

grenzt den Strom von der Spannungsquelle zur Masse. Wäre er nicht vorhanden, würde ein Kurzschluss die Schaltung zerstören. Öffnet man den Schalter, kann über den Widerstand ein Strom zum Ausgang fließen. Dieser sorgt dafür, dass das Potenzial an diesem angehoben wird. Wenn dieses den Wert der Versorgungsspannung erreicht hat, fließt kein Strom mehr über den Widerstand, und der Ausgang $U_E(t) = V_{DD}$ ist logisch 1. Schaltungstechnisch lässt sich das mit einem Kondensator realisieren. Dieser lädt sich dann so lange auf, bis die Versorgungsspannung erreicht ist.

Im nächsten Schritt werden in der Halbleitertechnik das Relais, also die Spule und der Schalter durch einen Transistor ersetzt. Seit den 1970er-Jahren werden dafür MOS-Transistoren eingesetzt. Diese haben den Vorteil, dass sie sich durch eine Spannungsänderung am Gate schalten lassen und dass sie gleichzeitig ein Kondensator am Ausgang des Vorgängergatters realisieren. In der digitalen Schaltungstechnik unterscheidet man zwischen zwei Varianten: erstens eine Inverterschaltung auf der Grundlage eines NMOS-Transistors und zweitens dieselbe Schaltungsstruktur mit einem PMOS-Transistor. Bedingt durch die unterschiedlichen elektronischen Eigenschaften der beiden Transistorarten ist die PMOS-Schaltung im Verhältnis zum NMOS-Inverter gespiegelt. Die Abb. 4.38 zeigt die gebräuchlichen logischen Schaltsymbole für ein NICHT-Gatter. Das international genormte logische Symbol ist in Abb. 4.38a zu sehen, das veraltete Zeichen in Abb. 4.38b. Die Darstellung des NICHT-Gatters als Dreieck mit einem schwarzen Punkt kommt daher, dass die verwendete Transistorschaltung in der analogen Schaltungstechnik einem Verstärker entspricht, gezeichnet als liegendes Dreieck. Der Punkt symbolisiert die Umkehrung, das NICHT. Tatsächlich verstärkt dieses Gatter die Eingangsspannung. Egal, welchen Wert diese oberhalb der das logische Signal charakterisierenden Grenzspannung aufweist, direkt am Ausgang liegt dann die Versorgungsspannung an. Wenn also zwei Gatter über eine längere Leitung verbunden werden und über diese Verluste entstehen, hebt das Gatter die Ausgangsspannung auf die Versorgungsspannung und stellt den ursprünglichen Logikpegel wieder her. Oft ist es üblich, in logischen Schaltungen Inverter noch mit dem Verstärkersymbol zu bezeichnen. Insbesondere dann, wenn es nicht auf die logische Funktion der Negation ankommt, sondern wenn elektrische Signale z. B. zum Treiben von Verbindungen verstärkt werden sollen.

Inverterschaltung

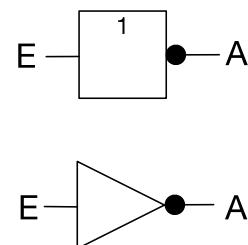


Abb. 4.38 Schaltsymbol eines Inverters

4

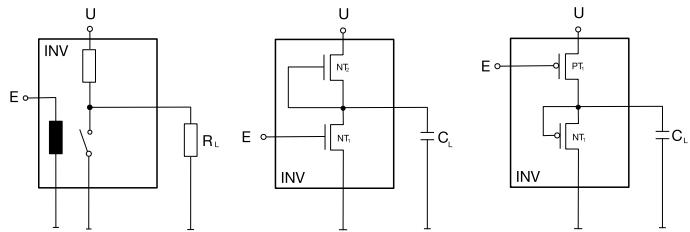


Abb. 4.39 Inverter mit einem Relais, in NMOS- und PMOS-Technik

MOS-Inverter

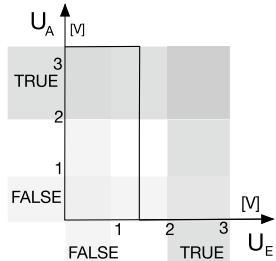


Abb. 4.40 Ideale Ausgangs-Kennlinie eines Inverters

Wird ein Inverter im Planarprozess als integrierte Schaltung realisiert, muss diese ein wenig verändert werden. Der elektrische Widerstand einer elektrischen Leitung ist proportional zur Länge dieser und umgekehrt proportional zum Leitungsquerschnitt und zur Leitfähigkeit des verwendeten Materials:

$$R = \rho \frac{l}{A} \quad (4.17)$$

Wird also die Leitfähigkeit ρ erhöht, sinkt der Widerstand, und es kann mehr Strom fließen. Das Gleiche gilt, wenn der Leitungsquerschnitt A vergrößert wird. Umgekehrt steigt der Widerstand, wenn die Leitung verlängert wird. Sollen also Widerstände zusammen mit Transistoren auf einem Chip integriert werden, besteht die Möglichkeit, die drei genannten Parameter entsprechend einzustellen. Da man Verbindungen auf einem Halbleiter nicht beliebig schmal machen kann, gibt es in jeder Halbleitertechnologie einen fest vorgegebenen Leitungsquerschnitt, der nicht unterschritten werden darf. Die Leitfähigkeit der Schaltungsstrukturen kann nur dann abgewandelt werden, wenn man zusätzliche Prozessschritte einführt, und zwar für jeden gewünschten Widerstandswert. Soll dies aus Kostengründen unterbleiben, kann ausschließlich die Länge der Leitung verändert werden. Allerdings ist die Leitfähigkeit des dotierten Siliziums so hoch, dass Widerstände auf dem Chip sehr lang sind, um einen realistischen Wert anzunehmen. Dies verbraucht viel Fläche. Mit einem Trick lässt sich dieses Problem umgehen: Man verwendet die ohnehin vorhandenen Transistoren als Widerstand. Es handelt sich im Prinzip um steuerbare Widerstände – daher auch der Name Transistor. Kann auf die Steuerbarkeit des Widerstands verzichtet werden, schaltet man in der NMOS-Technik das Gate an den Ausgang. Der Transistor hat dann einen festgelegten Widerstand, wenn der Schalttransistor durchgeschaltet ist. Die Abb. 4.39 zeigt jeweils

einen Inverter in NMOS- und PMOS-Technik. Die Kennlinie, also das Verhalten der Ausgangsspannung als Funktion der Eingangsspannung, ist im Idealfall rechteckig (Abb. 4.40). In der Praxis ist dies jedoch nicht der Fall, und im Folgenden wird diskutiert, warum das so ist.

4.2.2 CMOS-Inverter

Die im vorherigen Abschnitt eingeführten Inverterschaltungen haben einen gravierenden Nachteil. In einem der beiden Schaltzustände fließt immer ein Strom, d. h., die Leistungsaufnahme und dadurch der Energieverbrauch der Schaltung sind hoch. Bereits Anfang der 1980er-Jahre wurden Transistordichten auf den Chips erreicht, bei denen es nicht mehr möglich war, die entstehende Verlustleistung und die daraus resultierende Wärme abzuführen. Um die Integrationsdichte in der Mikroelektronik weiterhin zu erhöhen, war es erforderlich, die Leistungsaufnahme der logischen Gatter zu reduzieren. Dies erreichten die Chipentwickler durch eine zeitliche Begrenzung des Stromflusses. In der neuen Technik fließt ein Strom nur dann, wenn von einem Zustand in den anderen geschaltet wird. Wird der als Widerstand betriebene Transistor im NMOS-Inverter durch einen PMOS-Transistor und im PMOS-Inverter durch einen NMOS-Transistor ersetzt und verbindet man die Eingänge der beiden unterschiedlichen Transistoren miteinander, sperrt der PMOS-Transistor, wenn der NMOS-Transistor leitet. In dem Fall fließt ein Strom nur, wenn von einer logischen 0 auf eine 1 oder von einer 1 auf eine 0 umgeschaltet wird. Diese Schaltungsstruktur implementiert einen Inverter in CMOS-Technik („complementary metal-oxide semiconductor“).

Die Integration von PMOS- und NMOS-Transistoren in einem Siliziumkristall erfordert zusätzliche Prozessschritte. Der Querschnitt durch einen CMOS-Inverter ist in Abb. 4.41 gezeigt. In einem ersten Schritt muss in das positiv dotierte Silizium eine negativ dotierte Wanne eingebracht werden. In diese wird anschließend der PMOS-Transistor integriert. In einem weiteren Schritt wird dann der NMOS-Transistor hergestellt. In der Abbildung ist über dem Querschnitt des Inverters (Abb. 4.44b) eine entsprechende Draufsicht (Abb. 4.44a) gezeigt. Um den gezeigten Schnitt zu erhalten, muss auf der Ebene des Kristalls eine Struktur erzeugt werden. Ähnlich wie bei Leiterplatten wird dafür ein flaches Schema oder Layout benötigt. Dieses beschreibt die Geometrie der Masken, mit denen die Schaltung erzeugt wird. In der Abbildung ist die für den gezeigten Querschnitt notwendige geometrische Struktur der verschiedenen Masken dargestellt. Dabei stellen jede Far-

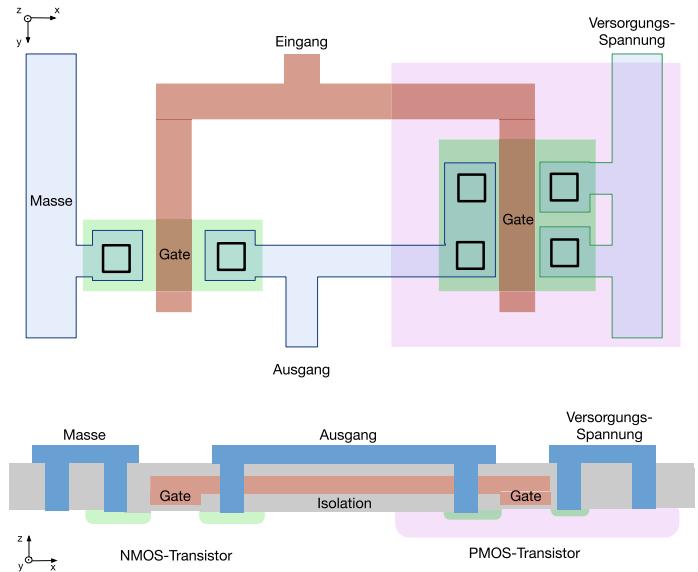


Abb. 4.41 Struktur eines CMOS-Inverters

be eine eigene Schaltungsebene und diese wiederum den jeweiligen Prozessschritt in der Fertigung dar. Für jede dieser Schaltungsebenen muss für die Herstellung der Schaltung eine Fotomaske entworfen und hergestellt werden. An dieser Stelle sei auf eine Besonderheit im Layout von CMOS-Gattern hingewiesen. Da die Beweglichkeit von Löchern merklich schlechter ist als die von Elektronen und somit die Leitfähigkeit des negativ dotierten Siliziums größer ist als die des positiv dotierten, ist der PMOS-Transistor breiter. Dadurch hat der Kanal des Transistors einen höheren Leitungsquerschnitt, und die geringere Leitfähigkeit der positiven Diffusion wird kompensiert.

Das in der Abb. 4.41 gezeigte Layout ist nicht sehr kompakt. Es lässt sich vereinfachen, wenn man die Diffusionsgebiete der Transistoren um 90° dreht. Die Struktur wird dadurch schmäler, und es können mehr Gatter auf einem Chip integriert werden. Abb. 4.49 zeigt das gewünschte Ergebnis. Durch die Drehung der Diffusionsgebiete im Layout entsteht eine sehr einfache Struktur. Wenn alle Logikgatter dabei die gleiche Breite aufweisen, können die einzelnen Gatter leicht hintereinander angeordnet und verbunden werden. Insbesondere die Spannungsversorgung und die Masseleitung müssen über eine solche Gatterreihe nur gerade durchgezogen werden. Diese regelmäßige Struktur erleichtert den Entwurf der Masken und damit der Schaltung.

Im Gegensatz zu der rein logischen Betrachtung der Schaltfunktionen muss beim Entwurf realer Schaltungen das elektrische Verhalten der Gatter berücksichtigt werden. Die ideale Kennlinie eines Inverters lässt sich mit realen elektronischen Bauelementen nicht realisieren. In einer digitalen Schaltung hat ein Transistor idealerweise zwei Zustände: *an* und *aus*. Ist er ausgeschaltet, ist der Widerstand des Kanals sehr hoch, und es kann kein Strom fließen. In einem einfachen Modell ist die Annahme eines unendlichen großen Widerstandes zulässig. Ist der Transistor eingeschaltet, hat der Kanal einen, wenn auch geringen, elektrischen Widerstand. Um den Zustand des Inverters zu speichern, wird angenommen, dass am Ausgang ein Kondensator angeschlossen ist. Dadurch wird erreicht, dass wirklich nur während des Umschaltvorgangs des Gatters ein elektrischer Strom fließt.

Wenn vor dem Zeitpunkt t_0 an den Eingängen der beiden Transistoren die Versorgungsspannung U_0 anliegt, dann sind der NMOS-Transistor durchgeschaltet und der PMOS-Transistor gesperrt. Der Kondensator am Ausgang ist entladen, die Spannung U_C ist 0 V. Schalten nun beide Transistoren zum Zeitpunkt t_0 , weil die Eingänge auf 0 V gezogen werden, fließt ein Strom. Während des Umschaltvorgangs sind beide Transistoren kurz leitend. Dieser Querstrom wird nun im einfachen Modell vernachlässigt. Es wird angenommen, beide Transistoren schalten in unendlich kurzer Zeit. Damit ergibt sich die links in der Abb. 4.42 gezeigte Schaltung. Über den Kanalwiderstand R_P des PMOS-Transistors wird der Kondensator über den Stromfluss durch den Widerstand R_P aufgeladen. Der Strom i_C ist in dem Fall gleich dem Strom i_R . Es gelten $i_R + i_C = 0$ und mit der kirchhoffschen Knotenregel $i_R = -i_C$. Da mit der kirchhoffschen Maschenregel $U_0 - U_R - U_C = 0$ ist, ist die Spannung über dem Widerstand $U_R = U_0 - U_C$. Daraus folgt mit $i_C(t) = C \frac{dU_C(t)}{dt}$:

Elektrisches Modell des CMOS-Inverters

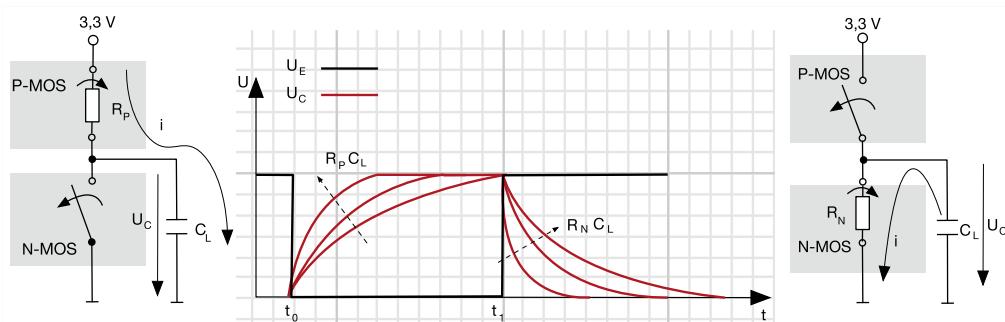


Abb. 4.42 Schaltverhalten eines CMOS Inverters

$$\frac{U_0 - U_C(t)}{R_P} = -C_L \frac{dU_C(t)}{dt} \quad (4.18)$$

Da der zeitliche Verlauf der Spannung $U_C(t)$ gesucht ist, werden die Variablen der Differenzialgleichung getrennt:

$$\frac{dU_C(t)}{U_0 - U_C(t)} = -\frac{1}{R_P C_L} dt \quad (4.19)$$

Mit einem Trick lässt sich diese besondere Form einer inhomogenen Differenzialgleichung 2. Ordnung lösen. Da U_0 konstant ist, gilt $d(U_0 - U_C(t)) = dU_C(t)$. Dann kann einfach vom Startzeitpunkt 0 bis t integriert werden:

$$\int_0^t \frac{dU_C(t')}{U_0 - U_C(t')} dt' = \int_0^t -\frac{1}{R_P C_L} dt' \quad (4.20)$$

Die Lösung ist:

$$[\ln(U_0 - U_C(t'))]_0^t = -\left[\frac{1}{R_P C_L} t' \right]_0^t \quad (4.21)$$

$$\ln(U_0 - U_C(t)) - \ln(U_0 - U_C(0)) = -\frac{1}{R_P C_L} t \quad (4.22)$$

Zum Zeitpunkt $t = 0$ ist der Kondensator nicht geladen, also ist $U_C(0) = 0$ V. Mit dieser Anfangsbedingung ergibt sich

$$\ln(U_0 - U_C(t)) - \ln(U_0) = -\frac{1}{R_P C_L} t \quad (4.23)$$

oder

$$\ln\left(\frac{U_0 - U_C(t)}{U_0}\right) = -\frac{1}{R_P C_L} t \quad (4.24)$$

Auflösen nach $U_C(t)$:

$$\frac{U_0 - U_C(t)}{U_0} = e^{-\frac{1}{R_P C_L} t} \quad (4.25)$$

oder

$$U_C(t) = U_0(1 - e^{-\frac{1}{R_P C_L} t}) \quad (4.26)$$

Beim Einschalten des PMOS-Transistors steigt die Spannung am Kondensator mit einer negativen Exponentialfunktion langsam an. Die Anstiegszeit ist von der Zeitkonstanten $\tau = RC$ abhängig. Das heißt, wenn der Kanalwiderstand R oder

die Lastkapazität des Inverters erhöht wird, dauert das Laden des Kondensators länger. Umgekehrt ist es möglich, durch Reduktion des Kanalwiderstands oder Verringerung der Lastkapazität die Schaltzeit eines logischen Gatters zu reduzieren.

Der inverse Fall, das Entladen des Lastkondensators über den NMOS-Transistor, ist etwas einfacher zu rechnen:

$$U_R - U_C(t) = 0 \quad (4.27)$$

$$i_R R_N - U_C = 0 \quad (4.28)$$

$$R_N C_L \frac{dU_C(t)}{dt} - U_C(t) = 0 \quad (4.29)$$

Die Gleichung lässt sich ebenfalls durch das Trennen der Variablen lösen:

$$\frac{dU_C(t)}{U_C(t)} = -\frac{R_N C_L}{d} t \quad (4.30)$$

und

$$\int_0^t \frac{dU_C(t')}{U_C(t')} dt' = - \int_0^t \frac{1}{R_N C_L} dt' \quad (4.31)$$

Als Ergebnis folgt:

$$[\ln(U_C(t'))]_0^t = \left[-\frac{1}{R_N C_L} t' \right]_0^t \quad (4.32)$$

$$\ln(U_C(t) - U_C(0)) = -\frac{1}{R_P C_L} t \quad (4.33)$$

Da beim Entladen der Kondensator zum Zeitpunkt des Einschaltens des NMOS-Transistors $U_C(0) = U_{C_0}$ ist, gilt

$$\ln(U_C(t) - U_{C_0}) = -\frac{1}{R_P C_L} t \quad (4.34)$$

Nach $U_C(t)$ aufgelöst ist die Lösung

$$U_C(t) = U_{C_0} - e^{-\frac{1}{R_P C_L} t} \quad (4.35)$$

Die beiden überschlagsmäßigen Rechnungen zeigen, dass eine Inverterschaltung nicht sofort umschaltet. Vielmehr ergibt sich eine gewisse Verzögerung. Diese ist asymmetrisch, wenn der Kanalwiderstand des NMOS-Transistors ungleich dem Widerstand des PMOS-Transistors ist. Bei der Herstellung einer integrierten Schaltung treten trotz aller Bemühungen Fertigungstoleranzen auf, d. h., die elektronischen Eigenschaften

der auf einem Chip platzierten Transistoren unterscheiden sich geringfügig. Die Eingangs-Ausgangs-Spannungskennlinie eines CMOS-Inverters ist daher nicht ideal und variiert auch von Gatter zu Gatter.

Die Abb. 4.43 zeigt unterschiedliche Kennlinien. Um Störungen im Verlauf des logischen Signals zu vermeiden, reduziert man den Spannungsbereich der Wahrheitswerte und lässt eine große Lücke zwischen diesen. Die Grenzspannung zwischen der 0 und der 1 wird aufgefächert. Dadurch entsteht ein Spannungsbereich mit einem undefinierten Zustand. Erst wenn das analoge Signal die Spannung U_{high} überschreitet, wird der Eingang als logische 1 (*true*) interpretiert, und wenn die Eingangsspannung U_{low} unterschreitet, liegt eine logische 0 (*false*) vor. Durch dieses Vorgehen wird der Signal-Stör-Abstand der Schaltung vergrößert, und die Rechner werden unempfindlicher gegen Fertigungstoleranzen und äußere Störungen.

Die Anwendung dieser Technik führt zu einem undefinierten Bereich im logischen Signal. Wird das logische vom elektrischen Verhalten gedanklich getrennt – der Zustandswechsel einer Schaltung erfolgt erst nach dem Überschreiten von U_{high} bzw. dem Unterschreiten von U_{low} – ergibt sich das in der Abb. 4.44 gezeigte logische Signal. Aus der Festlegung des Signal-Stör-Abstands können die im Logikentwurf zu berücksichtigenden Schaltzeiten eines Gatters abgeleitet werden. Bedingt durch die elektrischen Eigenschaften der Schaltung ist eine Verzögerungszeit bei einem Pegelwechsel von 0 auf 1 zu berücksichtigen, die Zeit t_r („rise“). Bei einem Wechsel von 1 auf 0 heißt die Verzögerungszeit t_f („fall“). Die auch oft verwendete allgemeine Verzögerungszeit eines Gatters t_d (Delay) berechnet sich aus dem Mittelwert von t_r und t_f :

$$t_d = \frac{t_r + t_f}{2} \quad (4.36)$$

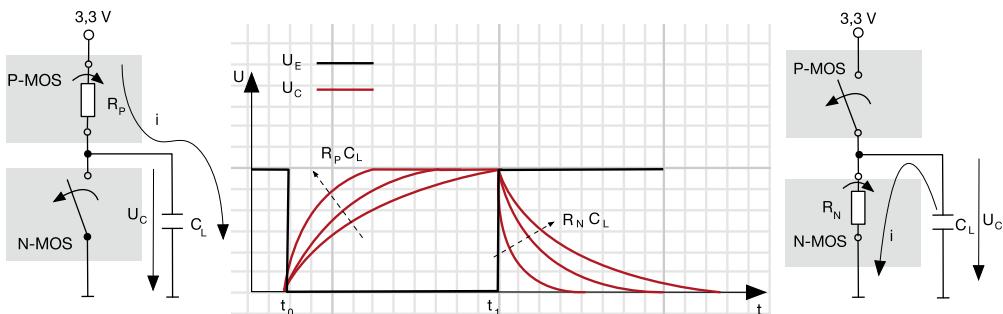


Abb. 4.44 Einfaches elektrisches Modell eines CMOS-Inverters

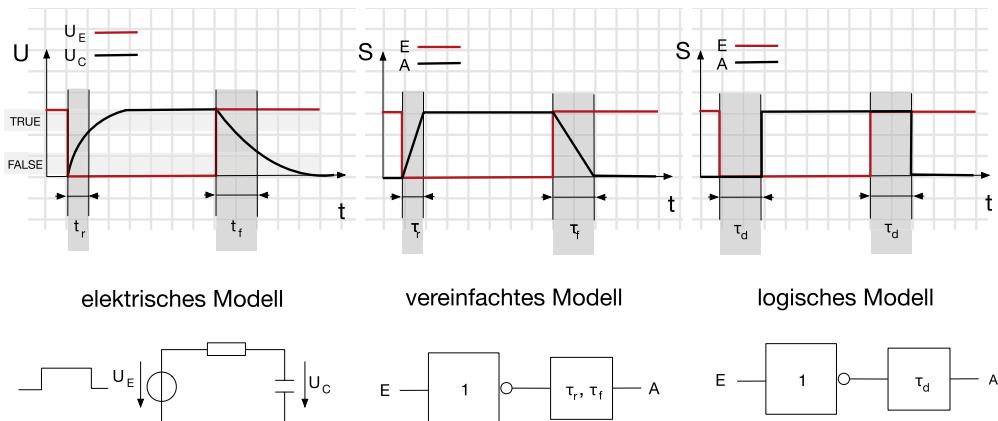


Abb. 4.45 Zeitmodell der Gatter

Die Abb. 4.45 zeigt die Definition von t_r und t_f . Erst wenn die Grenzspannung U_{high} überschritten ist, liegt am Ausgang des Gatters die logische 1 an. Umgekehrt wird diese erst mit Unterschreiten von U_{low} zur 0. Die Zeit vom Umschalten des Eingangssignals bis zum Erreichen von U_{high} bzw. U_{low} ist t_r bzw. t_f . Das elektrische Schaltmodell kann somit vereinfacht werden. Eine einfache logische Modellierung ist, die Signalflanken als Rampen anzunehmen und für die steigende und die fallende Flanke des Gatters unterschiedliche Steigungen anzunehmen. Diese können direkt aus dem elektrischen Modell ermittelt werden. In den digitalen Schaltungen sind diese Verzögerungen durch ein zusätzliches Verzögerungsglied modelliert. Alternativ lassen sich die Zeiten direkt an das Gatter schreiben. Die Modellierung über ein zusätzliches Verzögerungsglied ermöglicht in manchen Fällen eine detaillierte Diskussion des logischen Schaltverhaltens in Abhängigkeit von der Zeit. Das logische Modell verwendet lediglich die einfache Verzögerung eines Gatters. In diesem wird nur die Signalverzögerung eines logischen Signals modelliert, nicht aber sein Verhalten bei steigender oder fallender Flanke. Jede dieser verschiedenen Beschreibungen wird in unterschiedlichen Stadien des Entwurfs eingesetzt. Es muss aber immer sorgfältig geprüft werden, ob ein Modell für die zugrunde gelegte Fragestellung gültig ist.

Im vorherigen Abschnitt wurde gezeigt, dass die Zeit für das Erreichen eines Ein-Pegels und die Zeit bis zum Aus-Pegel proportional zum Produkt aus Kanalwiderstand und Kapazität am Ausgang sind. Je größer die Lastkapazität ist, umso länger ist also die jeweilige Schaltzeit. Diese kann verkürzt werden, wenn der Kanalwiderstand eines Transistors redu-

Fan-in und Fan-out digitaler Gatter

ziert wird, z. B. wenn der Kanal des Transistors breiter ausgelegt wird. Doch Vorsicht, dadurch steigt die Eingangskapazität, und vorhergehende Gatter schalten dann langsamer. In der Regel wird eine Schaltung so ausgelegt, dass ein Gatter mehrere nachfolgende Gatter treiben kann. Als Entwickler von Logikschaltungen muss man die Eingangskapazitäten nicht berücksichtigen. Stattdessen definiert eine einfache Zahl am Eingang der FAN_{IN} und am Ausgang der FAN_{OUT} die Fähigkeit, weitere Logikgatter elektrisch zu treiben. In modernen CMOS-Bibliotheken ist der FAN_{IN} meist 1, während der FAN_{OUT} 50 ist. Damit kann ein Gatter 50 Nachfolger treiben. Spezifiziert ist dann als Verzögerungszeit immer die maximale Verzögerung, also wenn wirklich 50 weitere Gatter angeschlossen sind. In der Regel schalten die Gatter schneller, nämlich dann, wenn die spezifizierte Last unterschritten wird. Die Abb. 4.46 zeigt, wie ein elektrisches Ersatzmodell für eine beliebige logische Schaltung erzeugt wird. Auf der Grundlage eines solchen elektrischen Modells können dann für eine Logikfamilie FAN_{IN} und FAN_{OUT} berechnet werden. Beim Entwurf logischer Schaltungen sind Gatter durch sechs verschiedene Eigenschaften spezifiziert:

- die logische Funktion und die Anzahl der Eingänge,
- die Zeit der steigenden Flanke,
- die Zeit der fallenden Flanke,
- die aus der steigenden respektive fallenden Flanke berechneten Schaltverzögerungen,

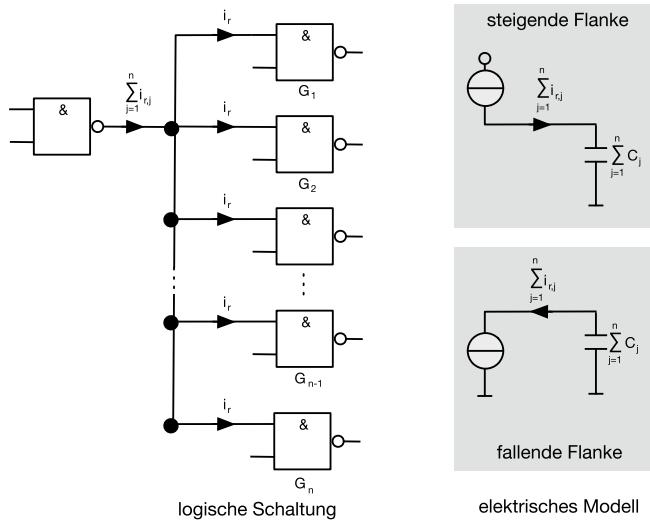


Abb. 4.46 Elektrisches Verhalten von CMOS-Gattern

- den FAN_{IN} , um anzugeben, welche Last ein Gatter erzeugt,
- den FAN_{OUT} , um anzugeben, wie viele weitere Gatter ein Gatter treiben kann.

Werden logische Gatter zu einer logischen Schaltung verbunden, sind einige Besonderheiten zu berücksichtigen. Jede der unterschiedlichen Schaltungen besteht aus mehreren MOS-Transistoren. Jedes der Gates der Transistoren entspricht einem Kondensator, der Ladungen speichert. Elektrisch lässt sich der Eingang eines Gatters als Kapazität beschreiben. Wird dieser an mehrere Transistoren angeschlossen, entspricht das einer Parallelschaltung dieser Kondensatoren. Die jeweiligen Kapazitäten addieren sich also. Es kann dann mehr Ladung gespeichert werden, und die Schaltzeit verringert sich.

Bei der Analyse der Inverterschaltung wurde eine einfache Differenzialgleichung aufgestellt, und es wurde gezeigt, dass sich Gatter nicht ideal verhalten. Auch das Modell eines Transistors als Schalter aufgefasst ist eine grobe Näherung. Aus diesem Grund werden in der Elektrotechnik komplexe Ersatzschaltbilder zur genauen Schaltungsanalyse auf der Schaltungsebene verwendet. Aus diesen Ersatzschaltungen entstehen aufwendige Differenzialgleichungssysteme, die das Verhalten einer Schaltung beschreiben. Da sich auf integrierten Schaltungen Ströme und Spannungen nicht messen lassen, wird zur Bestimmung der elektrischen Eigenschaften eine Schaltungssimulation auf Rechnern durchgeführt.³ Offensichtlich kann eine Transistororschaltung nicht nur durch seinen Schaltplan dargestellt werden, sondern auch durch ein mathematisches Modell. Selbiges gilt für die Gatterschaltung auf der logischen Ebene. Beschrieben wird das logische Verhalten dieser Schaltungen mithilfe von Schaltfunktionen. Offensichtlich unterscheidet sich der Entwurf einer Schaltung nicht nur auf unterschiedlichen Ebenen oder Schichten, sondern auch durch verschiedene Sichten. Ein Schaltplan stellt die Struktur einer Schaltung dar, während mathematische Modelle das Verhalten beschreiben, also die Verhaltenssicht darstellen. Daher wird der Entwurf von Computern in unterschiedliche Schichten von unten nach oben, Prozess-, Transistor-, Gatter-, Register-Transfer- und algorithmische Ebene und die beiden Sichten Struktur (Schaltplan) und Verhalten (mathematisches Modell) aufgeteilt.

³ Man kann sich das klar machen, indem man sich bewusst wird, dass Messspitzen elektrische Kapazitäten sind. Im Vergleich zur Nanoelektronik ist selbst eine mikroskopische Messspitze gigantisch. Sie verfälscht also das zu messende Verhalten in einem unüberwindlichen Maßstab.

4.2.3 CMOS-NAND- und -NOR-Gatter

NAND-Gatter

Die aus einem NMOS und einem PMOS bestehende Inverterschaltung lässt sich leicht zu einem weiteren grundlegenden Gatter erweitern. Wird ein zweiter NMOS-Transistor (PT_2) in Reihe zu dem ersten NMOS-Transistor (NT_1) geschaltet, kann der Ausgang nur die logische 0 annehmen, wenn an E_1 und an E_2 eine logische 1 anliegt. Es wird also ein NAND-Gatter gebildet. Allerdings ist in der CMOS-Schaltungstechnik zu berücksichtigen, dass, wenn nur einer der beiden Eingänge auf Masse liegt, der Ausgang auf die Versorgungsspannung gezogen werden muss. Sind beispielsweise der NMOS-Transistor NT_1 eingeschaltet (E_1 liegt auf der Versorgungsspannung = logische 1) und der NMOS-Transistor NT_2 ausgeschaltet (E_1 liegt auf Masse = logische 0), dann wäre der PMOS-Transistor PT_1 ausgeschaltet, und C_L könnte nicht geladen werden. Es wird somit ein weiterer PMOS-Transistor PT_2 parallel zu PT_1 benötigt, der in diesem Fall eingeschaltet ist und die Lastkapazität auf die Versorgungsspannung zieht.

Die entsprechende Schaltung und das international genormte logische Schaltzeichen sind in Abb. 4.47 gezeigt. Das Layout, die Vorlage für die Herstellungsmaske des Inverters, ist sehr leicht zu erweitern. Beide Diffusionsgebiete werden verlängert und mit einem weiteren Gate ausgestattet, sodass sich im Bereich des negativ dotierten NMOS-Transistors eine Reihenschaltung zweier Transistoren ergibt. Wenn dann im Teil des PMOS-Transistors die Versorgungsspannung zwischen den beiden Gates angeordnet wird und der obere sowie der untere Anschluss jeweils mit dem Ausgang A des Gatters verbunden werden, erhält man das Layout eines NAND-Gatters (Abb. 4.49).

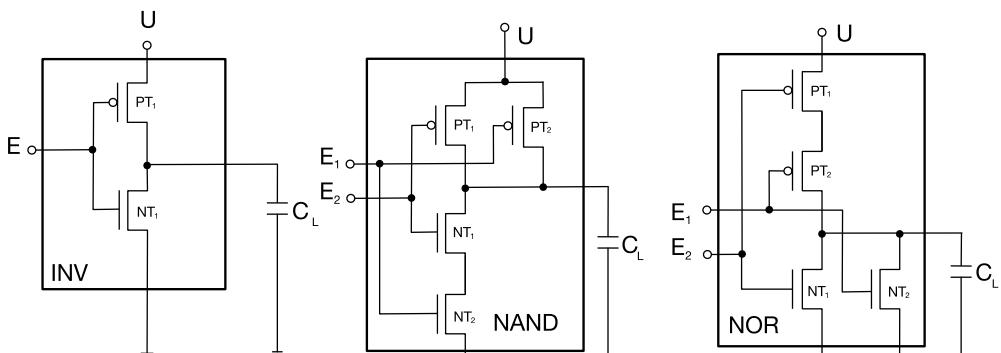


Abb. 4.47 Gatter in CMOS-Schaltungstechnik

Die schaltungstechnische Realisierung eines NOR-Gatters ist komplementär zu der Implementierung eines NAND-Gatters. Um ein ODER zu erhalten, müssen die beiden NMOS-Transistoren NT_1 und NT_2 parallel geschaltet werden. In dem Fall ist der Ausgang des Gatters auf Masse gelegt, wenn nur einer der beiden NMOS-Transistoren durchgeschaltet ist. Das bedeutet, dass der Ausgang ausschließlich dann auf die Versorgungsspannung gezogen werden darf, wenn beide PMOS-Transistoren eingeschaltet sind. Die Transistoren PT_1 und PT_2 sind daher in Reihe zu schalten. Die Abb. 4.48 und 4.49 zeigen die jeweilige Schaltung, das Schaltzeichen für das logische Gatter und das dazugehörige Maskenlayout.

Logische Gatter, die in der CMOS-Schaltungstechnik aufgebaut werden, folgen einer strengen komplementären Symmetrie. Um ein NAND zu bauen, werden die beiden NMOS-Transistoren wie ein UND geschaltet, um ein NOR zu realisieren, werden sie wie ein ODER verschaltet. Die PMOS-Transistoren sind dann komplementär zu verdrahten. Aus dieser Beobachtung kann eine allgemeine Regel für die logische Verschaltung der PMOS-Transistoren und der NMOS-Transistoren abgeleitet werden. Sei $f(x_1, x_2, \dots, x_n)$ eine beliebige Schaltfunktion.

$$f_P(E_1, E_2, \dots, E_3) = f(E_1, E_2, \dots, E_3) \quad (4.37)$$

$$f_N(E_1, E_2, \dots, E_3) = \overline{f(E_1, E_2, \dots, E_3)} \quad (4.38)$$

NOR-Gatter

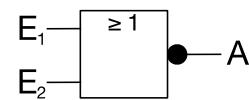


Abb. 4.48 Schaltsymbol NOR

CMOS-Gatter beliebiger Schaltfunktionen

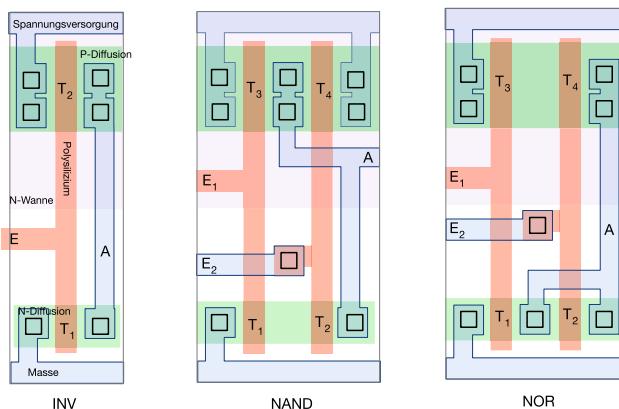


Abb. 4.49 NAND und NOR in CMOS

Definition 4.2.1 – CMOS-Logikgatter

Sei $f(x_1, x_2, \dots, x_n)$ eine beliebige Schaltfunktion, die in CMOS-Schaltungstechnik umgesetzt werden soll. Die Verschaltungen $f_P(E_1, E_2, \dots, E_3)$ der PMOS- und $f_N(E_1, E_2, \dots, E_3)$ der NMOS-Transistoren werden wie folgt bestimmt: PMOS-Schaltung:

$$f_P(E_1, E_2, \dots, E_3) = f(x_1, x_2, \dots, x_n) \quad (4.39)$$

NMOS-Schaltung:

$$f_N(E_1, E_2, \dots, E_3) = \overline{f(x_1, x_2, \dots, x_n)} \quad (4.40)$$

Dabei sind x_i die logischen Variablen und E_i die, die jeweilige logische Variable implementierenden Eingänge des Gatters.

► Beispiel 4.2.1 – NAND-Gatter

Die Schaltfunktion des NAND ist $f(x_1, x_2) = \overline{x_1 x_2}$. Dann gilt:
PMOS-Schaltung:

$$f_P(E_1, E_2, \dots, E_3) = \overline{x_1 x_2} = \overline{x_1} + \overline{x_2}$$

NMOS-Schaltung:

$$f_N(E_1, E_2, \dots, E_3) = \overline{\overline{x_1 x_2}} = x_1 x_2$$

**► Beispiel 4.2.2 – XOR-Gatter**

Die Schaltfunktion $f(x_1, x_2) = x_1 \overline{x_2} + \overline{x_1} x_2$ ist die disjunktive kanonische Normalform der booleschen Funktion XOR. In CMOS-Schaltungstechnik lässt sie sich wie folgt implementieren:
PMOS-Schaltung:

$$f_P(E_1, E_2, \dots, E_3) = x_1 \overline{x_2} + \overline{x_1} x_2$$

NMOS-Schaltung:

$$f_N(E_1, E_2, \dots, E_3) = \overline{\overline{x_1} \overline{x_2}} + \overline{x_1} \overline{x_2}$$

$$f_N(E_1, E_2, \dots, E_3) = \overline{x_1 \overline{x_2}} \cdot \overline{\overline{x_1} x_2}$$

$$f_N(E_1, E_2, \dots, E_3) = (\overline{x}_1 + x_2) \cdot (x_1 + \overline{x}_2)$$

$$f_N(E_1, E_2, \dots, E_3) = x_1 x_2 + \overline{x}_1 \overline{x}_2$$



4.3 Flipflops und Bitspeicher

Mit den logischen Gattern ist es möglich, beliebige Schaltfunktionen zu realisieren. Damit sind die technischen Grundlagen zur Umsetzung der in ▶ Kap. 3 beschriebenen mathematischen Verfahren in konkrete Hardware gelegt. Wenn man sich allerdings an die Rechnungen aus ▶ Kap. 2 erinnert, müssen beim Rechnen Ziffern oder sogar ganze Zahlen gespeichert werden. Neben der reinen logischen Funktion werden demnach noch Speicherelemente benötigt. Im folgenden Abschnitt des Kapitels werden Schaltungen beschrieben, um 1 Bit zu speichern. In späteren Kapiteln werden diese Strukturen dann erweitert, um ganze Zahlen, also mehrere Bits abzulegen.

4.3.1 Flipflops

Neben kombinatorischen Schaltungen zur Realisierung von Schaltfunktionen ist es notwendig, Zahlen in einem Rechner zu speichern. Jede Zahl besteht aus einzelnen Ziffern, und im Binärsystem kann jede Ziffer 2 Zustände einnehmen. Es wird also eine Schaltung benötigt, um zwei binäre Zustände sicher über die Zeit zu speichern. Ist eine derartige Schaltung gefunden, können größere Zahlen durch eine Parallelschaltung dieser 1-Bit-Speicherschaltungen realisiert werden. Neben den einfachen Schaltfunktionen existiert noch eine weitere Klasse digitaler oder logischer Schaltungen, die rückgekoppelten Schaltungen oder auch Schaltwerke.

Die englischen Physiker William Henry Eccles (1875–1966) und Frank Wilfred Jordan (1882–1941) experimentierten im frühen 20. Jahrhundert mit Schaltungen zur drahtlosen Kommunikation. Bei der Suche nach einer Zählschaltung erfanden sie 1918 die zunächst nach ihnen benannte Eccles-Jordan-Schaltung, die heute bistabile Kippstufe genannt wird. Die Schaltung von Eccles und Jordan arbeitete mit über Kondensatoren und Widerstände rückgekoppelten Röhrenverstärkern. An dieser Stelle sei noch einmal daran erinnert, dass ein RC-Glied eine Verzögerungsschaltung mit der Zeitkonstanten RC darstellt.

4.3.1.1 Rückgekoppelte Schaltungen

Eine bistabile Kippstufe heißt in der Digitaltechnik Flipflop. Da ein MOS-Transistor gleichzeitig einen Verstärker darstellt und der Eingang des Transistors eine Kapazität ist, sollte sich ein Flipflop auch mit Gattern in MOS-Technik realisieren lassen. Indem zwei Inverter rückgekoppelt werden, entsteht eine einfache Flipflopschaltung. Dabei bilden die Kanalwiderstän-

de der Transistoren das R und die Gate-Kapazität das C des RC-Gliedes. Diese Schaltung kann zwar Zustände speichern, aber wie kann sie gesetzt oder zurückgesetzt werden? Dazu müsste es einen Setz- und einen Rücksetzeingang geben. Um dies zu realisieren, werden 2-wertige Gatter benötigt. Daher sollen im Folgenden einmal die Eigenschaften rückgekoppelter, 1- und 2-wertiger Gatter der logischen Basis *NICHT*, *UND* und *ODER* diskutiert werden.

Zunächst einmal wird ein rückgekoppelter Inverter betrachtet. Das in CMOS-Technik aufgebaute Gatter ist nicht ideal. Das Ausgangssignal tritt also verzögert auf. Wird der Ausgang einer Inverterschaltung direkt auf den Eingang der gleichen Schaltung rückgekoppelt und ändert der Eingang seinen Zustand, dann muss auch der Eingang verspätet seinen Zustand wechseln. Dieses Verhalten wird ohne Ende bis in alle Ewigkeit fortgesetzt. Es entsteht eine Schwingung. Und in der Tat ist die Inverterschaltung nichts anderes als ein rückgekoppelter Verstärker. Unter bestimmten Parametern seiner Beschaltung fängt ein solcher Verstärker ebenfalls an zu schwingen: Die Schaltung heißt dann Oszillator. In der Digitaltechnik ist es nicht ratsam, einen einzelnen Inverter rückzukoppeln, da dies, insbesondere in CMOS-Technik, zur Zerstörung führen kann. Es ist aber möglich, ab mindestens drei Invertern eine beliebige ungerade Zahl dieser Gatter rückzukoppeln. Die entstehende Schaltung heißt Ringoszillator und kann zur Erzeugung von Taktsignalen verwendet werden (Abb. 4.50).

Was passiert nun bei 2-wertigen Logikgattern? Als erstes 2-wertiges Gatter wird das *UND* betrachtet. Für die Diskussion wird das Modell eines idealen Gatters, das an seinem Ausgang mit einem Verzögerungselement verschaltet ist, gewählt. Dadurch kann neben dem Eingang und dem Ausgang noch der logische Signalverlauf eines Zwischenzustands betrachtet

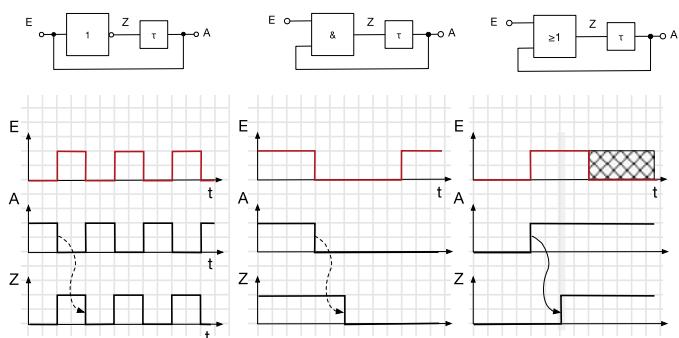


Abb. 4.50 Rückgekoppelte Gatter

werden. Wird ein *UND* rückgekoppelt und ist der Ausgang des UND-Gatters zu Beginn der Betrachtung 1, dann gibt das *UND*-Gatter eine 0 am Eingang einfach mit der Zeitverzögerung durch. Den Eingang am Anfang auf 0 zu legen ist sinnlos, da der Ausgang immer erst 0 ist und egal was am Eingang anliegt, ständig nur 0 ausgegeben wird. Offensichtlich setzt ein rückgekoppeltes *UND* lediglich seinen Ausgang auf 0, wenn der Eingang von 1 auf 0 wechselt. Alle anderen Anregungsmuster bewirken nichts.

Bei einem rückgekoppelten *ODER* sollen zunächst einmal sowohl der geschlossene als auch der offene Eingang 0 sein. Wechselt nun der offene Eingang für kurze Zeit auf 1, erzeugt die Schaltung am Ausgang einen kurzen Puls. Das liegt daran, dass, wenn der offene Eingang vor Ablauf der Verzögerung τ durch das Gatter wieder 0 wird, auch der Ausgang zu 0 wird und das 1-Signal nicht gehalten werden kann. Wenn aber das Eingangssignal etwas länger als die Verzögerungszeit des Gatters am Eingang anliegt, wechselt der Ausgang auf 1 und bleibt dann auch auf diesem Pegel. Danach spielt es keine Rolle mehr, welche Pegel am Eingang gesetzt werden. Der Ausgang bleibt immer eingeschaltet, also auf 1. Mit dieser Schaltung lässt sich somit 1 Bit speichern. Da es ab jetzt egal ist, welche Pegel am offenen Eingang anliegen, kann das rückgekoppelte *ODER* nicht zurückgesetzt werden. Das Gatter lässt sich somit nicht mehr löschen, es sei denn, die Spannungsversorgung wird unterbrochen. Da ein rückgekoppeltes *ODER* gesetzt, aber nicht zurückgesetzt und ein rückgekoppeltes *UND* nur zurückgesetzt werden können, liegt es nahe, beide Schaltungen miteinander zu kombinieren.

4.3.1.2 RS-Flipflop

Werden ein ODER und ein UND-Gatter kreuzweise rückgekoppelt, entsteht ein Flipflop, das über das ODER-Gatter gesetzt und über das UND-Gatter zurückgesetzt werden kann. Diese Schaltung heißt RS-Flipflop für Rücksetz-Setz-Flipflop (Abb. 4.51 und 4.52). Die Funktion des RS-Flipflops soll kurz diskutiert werden: Zunächst sind der Eingang E_1 des ODER-Gatters 0 und der Eingang E_2 des UND-Gatters 1. Ein länger als τ dauernder Impuls an E_1 setzt den Ausgang A_1 . Wenn dann E_2 auf 0 geht, wird A_1 zurückgesetzt. A_1 kann später, wenn E_2 nach einiger Zeit wieder gesetzt wurde, erneut 0 werden und 1 Bit speichern. Diese Schaltung ist somit ein Flipflop, das mit einem positiven Signal an dem einen Eingang gesetzt (*set*) und mit einem negativen Impuls am anderen Eingang zurückgesetzt (*reset*) wird. Leider ist dieser Aufbau unsymmetrisch: Es werden ein logisch positives Signal zum Setzen und ein logisch negativer Impuls zum Zurücksetzen benötigt.

RS-Flipflop

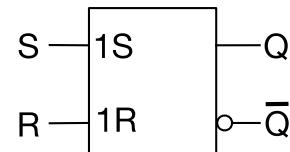
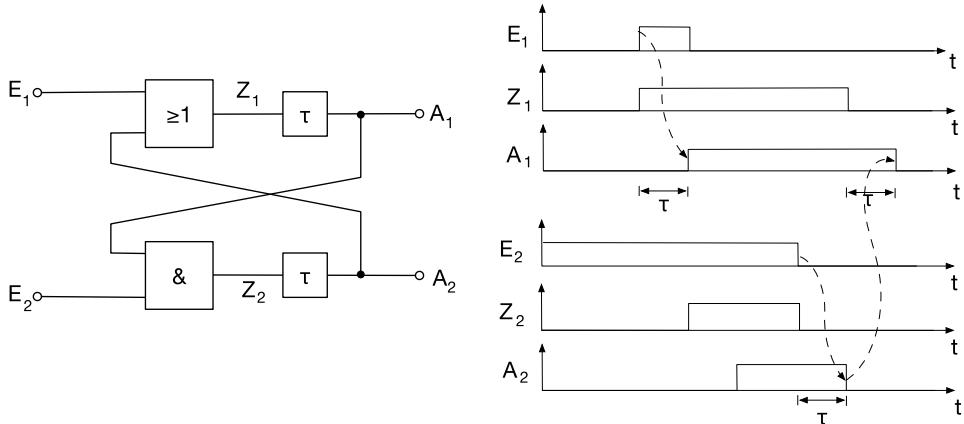


Abb. 4.51 Schaltsymbol eines RS-Flipflops

**Abb. 4.52** RS-Flipflop

tigt. Im Folgenden sollen nun der Eingang E_1 S und der Eingang E_2 R heißen. Auch die Ausgänge werden umbenannt: Aus A_1 wird \bar{Q} und aus A_2 wird Q .

Wird einfach der E_2 - bzw. R-Eingang negiert,

$$A_1 = E_1 + A_2 \quad (4.41)$$

$$\bar{Q} = S + Q \quad (4.42)$$

$$A_2 = \overline{E_2} \cdot A_1 \quad (4.43)$$

$$Q = \overline{R} \cdot S \quad (4.44)$$

folgt dann:

$$\overline{A_1} = \overline{E_1 + A_2} \quad (4.45)$$

$$\overline{A_1} = \overline{E_1 + A_2} \quad (4.46)$$

Mit dem Satz von DeMorgan

$$A_2 = \overline{E_2} \cdot \overline{A_1} = \overline{E_2 + A_1} \quad (4.47)$$

$$Q = \overline{R} \cdot \overline{S} = \overline{R + \overline{Q}} \quad (4.48)$$

wird eine neue Schaltung erzeugt. Ein RS-Flipflop, das mit positiven Pegeln gesetzt und rückgesetzt werden kann, lässt sich mittels zwei rückgekoppelten NOR-Gattern aufbauen. Aus

4.3 · Flipflops und Bitspeicher

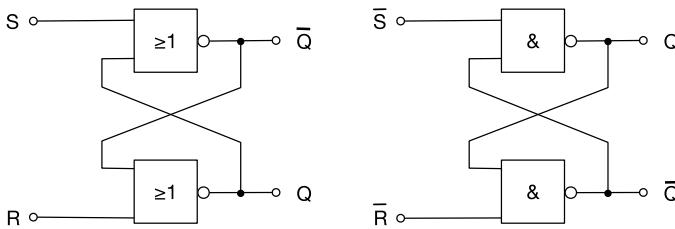


Abb. 4.53 Schaltungen des RS-Flipflops

Symmetriegründen ist zu erwarten, dass sich ein Flipflop dann auch mit zwei *NAND*-Gattern bauen lässt. Beide Schaltungen sind in der Abb. 4.53 gezeigt, und deren Funktion und Eigenschaften werden im Folgenden diskutiert.

Das logische Diagramm in Abb. 4.54 zeigt die Signalverläufe für das idealisierte Zeitverhalten einer digitalen Schaltung. Es wird eine konstante Verzögerungszeit für jedes der beiden Gatter des Flipflops angenommen. Diese Annahme ist in monolithisch hergestellten Schaltungen erlaubt, da alle Transistoren denselben Fertigungsschritten gleichzeitig ausgesetzt sind. Fertigungstoleranzen sind daher sehr gering und sind an dieser Stelle nicht zu berücksichtigen. Zum Beginn der Diskussion sei angenommen, dass der Setz- und der Rücksetzeingang des Flipflops 0 sind und kein Bit im Flipflop gespeichert ist, somit $Q = 0$ und $\bar{Q} = 1$ gelten. Bei einem 1-Puls,

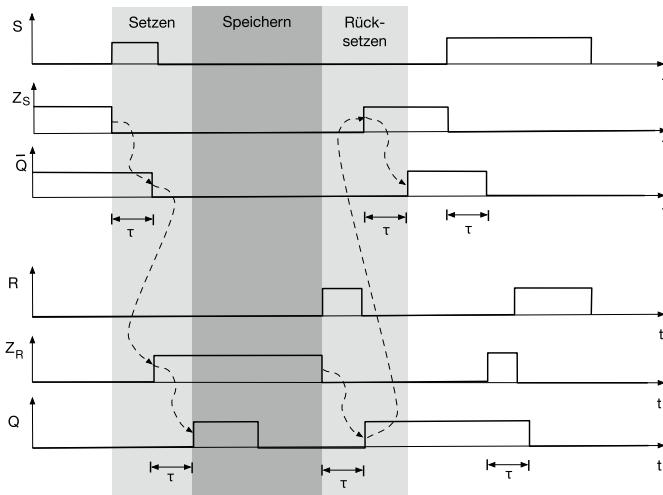


Abb. 4.54 Signalverläufe des NOR-RS-Flipflops

der länger als die Gatterverzögerungszeit ist, wechseln $Q = 1$ und $\bar{Q} = 0$, und zwar \bar{Q} nach 1 Verzögerungszeit und Q nach 2 Verzögerungszeiten, da dieser Wert erst durch beide Gatter läuft. Solange beide Eingänge des Flipflops 0 sind, speichert die Schaltung das Signal des Setzeingangs. Erst wenn R mit einem Rücksetzpuls belegt wird, wechseln die beiden Ausgänge wieder ihren Zustand. Der Ablauf ist diesmal spiegelbildlich. \bar{Q} benötigt diesmal zwei Gatterverzögerungen, um den Zustand zu ändern. Offensichtlich speichert das RS-Flipflop einen Wert, solange beide Eingänge 0 sind. Was passiert aber, wenn beide 1 sind? Dieser Zustand heißt verbotener Zustand, da das Flipflop bei einem Verlassen dieses Zustands zerstört werden kann. Im Modell einer einfachen, konstanten Verzögerungszeit ist dieser Effekt nicht zu verstehen, daher sollen zunächst einmal ein aus NAND-Gattern aufgebautes RS-Flipflop diskutiert und das elektrische Verhalten der Schaltung später analysiert werden.

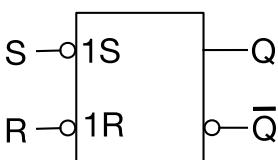


Abb. 4.55 Schaltsymbol eines aus NAND-Gattern aufgebauten RS-Flipflops

Ein aus NAND-Gattern aufgebautes RS-Flipflop verhält sich spiegelbildlich zu einem aus NOR-Gattern entworfenen Flipflop. Zu Beginn der Diskussion sind die Eingänge S und R auf logisch 1-Pegel. Diesmal sind aber der Ausgang $Q = 1$ und der Ausgang $\bar{Q} = 0$. Somit wird durch einen negativen Impuls am S-Eingang ein inverses Bit gespeichert, und zwar so lange, bis ein invertiertes Signal am R-Eingang das Flipflop wieder invers zurücksetzt. Verbotener Zustand bei dieser Schaltung ist nun, wenn an beiden Eingängen gleichzeitig eine 0 anliegt. Die gespiegelte Eigenschaft eines aus NAND-Gattern aufgebauten RS-Flipflops wird durch ein entsprechend anderes Schaltsymbol dargestellt (Abb. 4.55). Auch das RS-Flipflop aus NAND-Gattern lässt sich aus der ersten Schaltung des Flipflops mathematisch ableiten:

$$A_1 = E_1 + A_2 \quad (4.49)$$

$$A_2 = \overline{E_2} \cdot A_1 \quad (4.50)$$

kann auch wie folgt umgeformt werden:

$$\overline{A_1} = \overline{E_1 + A_2} = \overline{E_1} \cdot \overline{A_2} \quad (4.51)$$

$$A_1 = \overline{\overline{E_1} \cdot \overline{A_2}} \quad (4.52)$$

und

$$A_2 = \overline{E_2} \cdot \overline{A_1} \quad (4.53)$$

$$\overline{A_2} = \overline{\overline{E_2} \cdot \overline{A_1}} \quad (4.54)$$

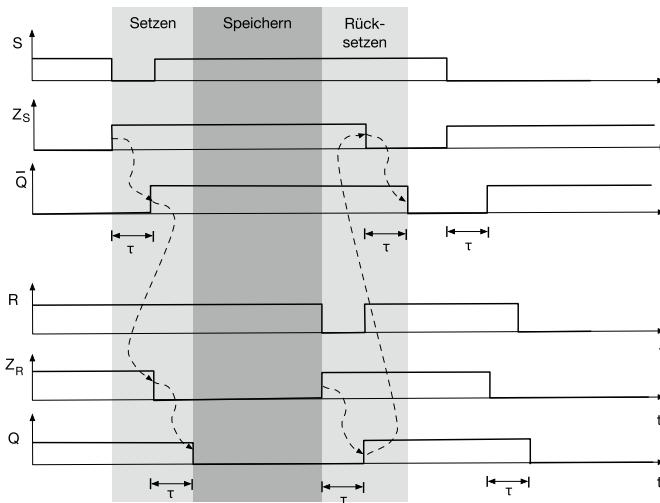


Abb. 4.56 Signalverläufe des NAND RS-Flipflop

Die NOR-Gatter können also durch NAND-Gatter ersetzt werden. In dem Fall ändern sich die Signalpegel am Eingang. Im Speicherzustand sind beide Eingänge 1, und ein negativer Impuls an einem der beiden Eingänge führt zu dem Zustandswechsel (Abb. 4.56).

Warum sind der Zustand $S = R = 1$ bei einem NOR-RS-Flipflop und der Zustand $S = R = 0$ bei einem NAND-RS-Flipflop auf logischer Ebene verbotene Zustände? In der Diskussion des Signalverlaufs zeigt sich logisch kein bösartiges Verhalten. Je nachdem, an welchem der beiden Eingänge des Flipflops der Pegel wechselt, wird das Flipflop gesetzt oder zurückgesetzt. Ein Puls an dem jeweils anderen Eingang hat dann keine Wirkung mehr. Bei der Ableitung der Schaltzeiten aus den elektrischen Eigenschaften der Inverterschaltung wurde die Verzögerung der Schaltung durch eine negative Exponentialfunktion beschrieben. Im Verzögerungsmodell nimmt man an, dass der Signalpegel nicht unendlich steil auf den neuen Wert springt. Wird der Signalwechsel mit einer langsam ansteigenden Geraden modelliert, zeigt sich, dass das Flipflop beim Übergang aus dem Verbotenen in einen anderen Zustand zwischen den Logikpegeln hängen bleibt. Nach kurzer Zeit wechselt es dann aber unvorhergesehen in einen stabilen Zustand. Daher nennt man den Zwischenzustand metastabil. Wenn also beide Eingänge gleichzeitig einen Signalwechsel durchführen und sich das Flipflop im verbotenen Zustand befindet, ist es möglich, dass eines der Gatter noch nicht vollständig durchgeschaltet ist. Wenn dann ein Wechsel des anderen Pfades eben-

Metastabiler Zustand

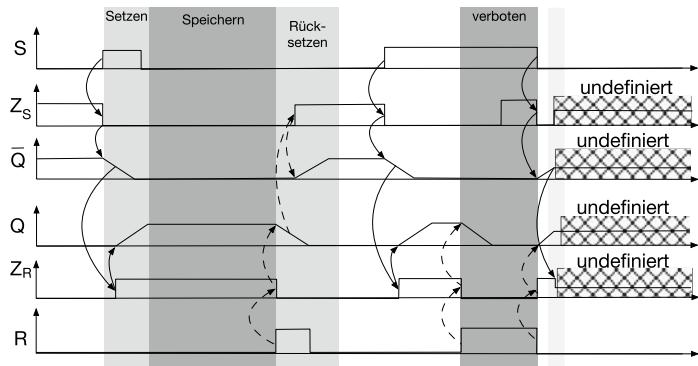


Abb. 4.57 Metastabiler Zustand des RS-Flipflops

falls an diesem Gatter eintrifft, kann dieses kurz zwischen den beiden Signalpegeln 1 und 0 verharren (Abb. 4.57). Je nach Fertigungstoleranzen der Schaltung kippt es anschließend erneut in einen stabilen Zustand. Da nicht vorhersehbar ist, in welchen Zustand das Flipflop fällt, ist es möglich, dass beide Ausgänge den gleichen Wert annehmen. In dem Fall würde die Schaltung anfangen zu schwingen. Schaltungstechnisch ist der metastabile Zustand des Flipflops kritisch, da hohe Ausgleichsströme in den Gattern zur Zerstörung derselben führen können.

Die Abb. 4.58a verdeutlicht den Wechsel in den metastabilen Zustand auf der Schaltungsebene. Wenn an einem Eingang eines *NAND*-Gatters eine 1 anlegt, kann dieses Gatter im Modell zu einem Inverter vereinfacht werden. In dem Fall, dass $Q = \bar{Q} = 1$ ist, wäre das bei dem *NAND*-RS-Flipflop der Fall. Werden die beiden Übertragungskennlinien der zwei Inverter in einem Spannungsdiagramm eingetragen, dann ist die Kennlinie von Inverter 2 um 90° zu drehen und anschließend zu spiegeln. Es ergibt sich das in der Abb. 4.58b gezeigte Diagramm. In diesem sind zwei stabile und der metastabile Zustand der Flipflopschaltung zu erkennen. Es sei noch einmal daran erinnert, dass logisch alles in bester Ordnung ist. Wechseln beide Eingänge gleichzeitig ihren Pegel, ändern auch die beiden Ausgänge ihren Wert (Abb. 4.58c). Wenn beide Eingänge wie die Ausgänge vor dem Zustandswechsel jeweils gleich waren, dann sind sie es nach dem Wechsel ebenfalls zur selben Zeit. Wenn jetzt ein Eingang des Flipflops wechselt, fällt das Flipflop logisch wieder in einen erlaubten Zustand. Es ist nun einzuwenden, dass es sehr unwahrscheinlich ist, wenn beide Eingänge einer solchen Schaltung auf 0 sind, dass dann ausgerechnet die anderen beiden Eingänge einen Zustandswechsel ausführen. Das mag für ein Flipflop auch zutreffen. In einem

4.3 · Flipflops und Bitspeicher

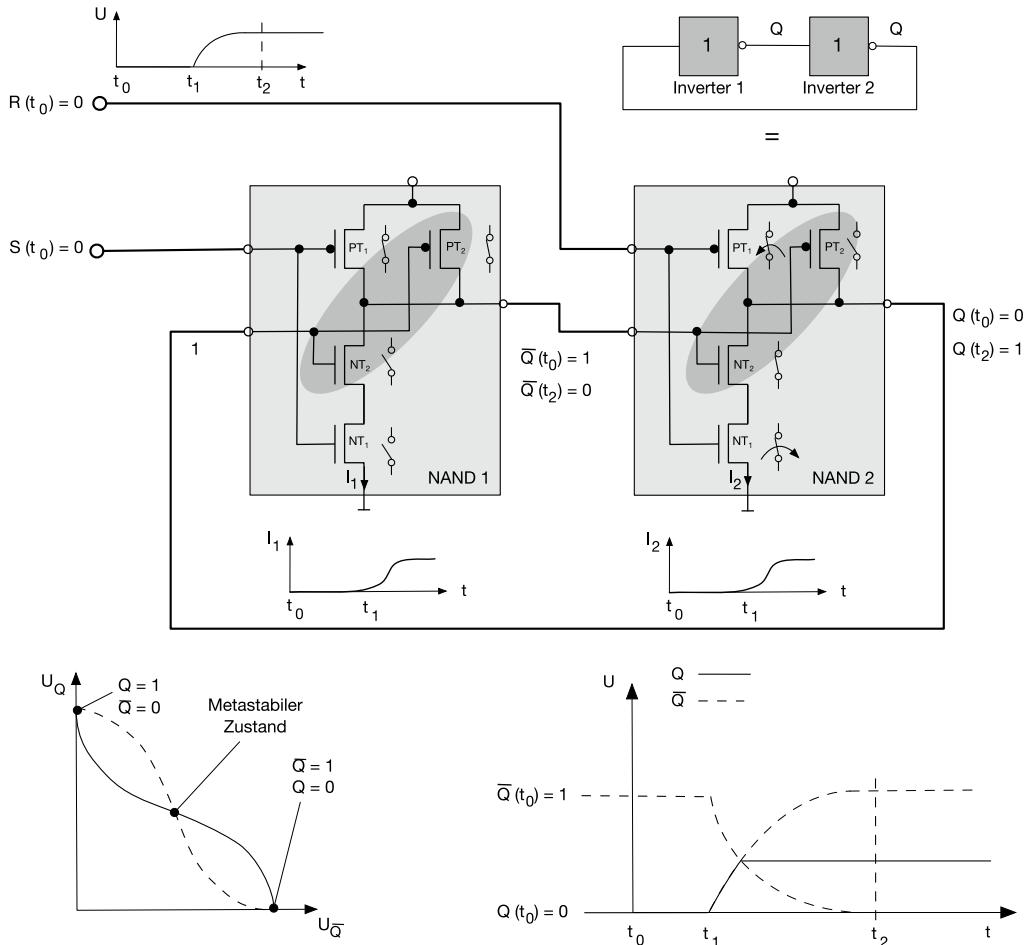


Abb. 4.58 Entstehung des metastabilen Zustands

Mikroprozessor sind aber Hunderttausende, wenn nicht sogar Millionen Flipflops verbaut. Jedes dieser Flipflops wird durch unterschiedliche Schaltfunktionen angesteuert. Die Laufzeiten der Signale durch diese Schaltungen sind verschieden, je nachdem wie viele Gatter verbaut wurden und wie diese verdrahtet sind. Bei Millionen von Gattern, Leitungen und Flipflops ist die Wahrscheinlichkeit, dass zwei Signalwechsel gleichzeitig passieren, nicht mehr gering. Im Gegenteil, bei Abermillionen Schaltvorgängen während des Ablaufs eines Algorithmus ist anzunehmen, dass metastabile Zustände häufig sind. Da aber ein länger andauernder metastabiler Zustand in der rückgekoppelten Schaltung eines Flipflops die Zerstörung derselben zur Folge haben könnte und ein zerstörtes Flipflop den ganzen Prozessor zerstört, wird ein RS-Flipflop so spezifi-

Taktgesteuertes RS-Flipflop

Master-Slave-Flipflop

ziert, dass eine 0 an beiden Eingängen für ein NAND-Flipflop und eine 1 an beiden Eingängen für ein NOR-Flipflop verboten sind. Letztendlich ist dies ein festgelegtes Entwurfsprinzip. Wendet man dieses an, muss man sich als Entwickler logischer Schaltungen keine Gedanken mehr um das elektrische Verhalten der Schaltung machen.

Offensichtlich möchte man sich im Entwurf digitaler Schaltungen nicht auf die Entwickler verlassen. Daher wurde die Schaltung des RS-Flipflops so erweitert, dass metastabile Zustände nicht mehr entstehen können. Ein RS-Flipflop kann mit einer Torschaltung erweitert werden. Ein solches RS-Flipflop nennt man taktgesteuertes RS-Flipflop. Die Torschaltung besteht aus zwei *UND*-Gattern, und über ein spezielles Signal lassen sich nun die Eingänge des RS-Flipflops sperren. Während das Signal *C* (wie Clock oder Takt aber oft auch *E* wie Enable) aktiv ist, also auf logisch 1 ist, kann das Flipflop Werte übernehmen. Damit kann die Wahrscheinlichkeit, dass am Eingang des RS-Flipflops ein verbotenes Signalpaar anliegt, reduziert werden. Allerdings kann in der Zeit eines aktiven Enable-Signals dieses mit einem nachfolgenden Übergang in einen bistabilen Zustand immer noch erfolgen.

Die Wahrscheinlichkeit in einen metastabilen Zustand überzugehen, lässt sich weiter reduzieren. Dazu werden zwei taktgesteuerte RS-Flipflops hintereinandergeschaltet und einer der beiden Enable-Eingänge wird invertiert angeschlossen. Durch dieses Schaltungsprinzip ist das Flipflop nur für sehr kurze Zeiten transparent, denn zuerst wird das erste durchgeschaltet, während das zweite noch verriegelt ist, und wenn das zweite Flipflop neue Werte annehmen kann, sperrt das erste. Ein derartiges Flipflop reagiert nicht mehr auf Pegel am Takteingang, sondern auf Flanken. Daher ist dieses Flipflop flankengesteuert. Je nachdem, wie die Anschlüsse des Takteingangs verdrahtet werden, reagiert dieses nach dem Master-Slave-Prinzip erstellte Flipflop auf eine steigende oder eine fallende Flanke des Taktsignals (Abb. 4.59).

Das flankengesteuerte RS-Flipflop nach dem Master-Slave-Prinzip hat einen Nachteil: Zwar wird mit diesem erfolgreich die Wahrscheinlichkeit gesenkt, dass das Flipflop in den verbotenen Zustand übergeht und somit metastabil wird, aber dieser Zustand kann anschließend nicht genutzt werden. Dieses Problem lässt sich beheben, wenn die Ausgänge des Master-Slave-RS-Flipflops über *UND*-Gatter auf die Eingänge zurückgeschaltet werden.

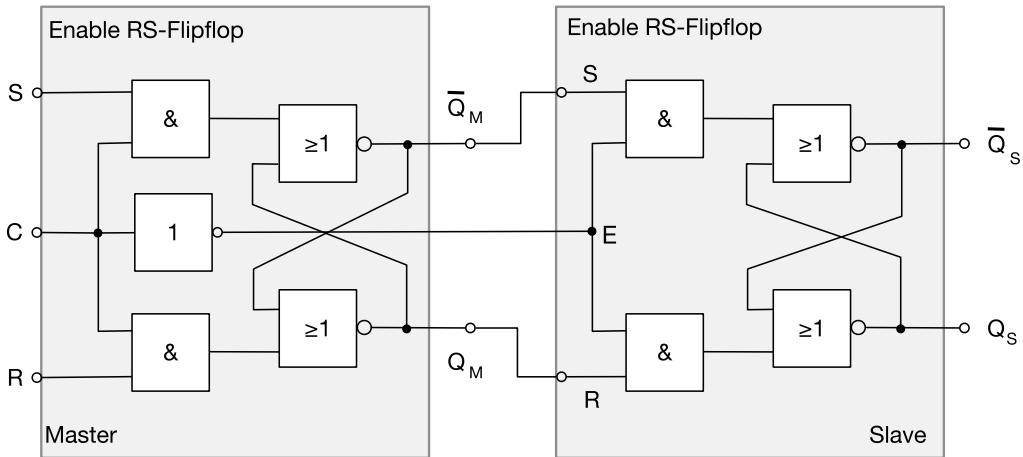


Abb. 4.59 CRS-Flipflop

Da die beiden Ausgänge des Master-Slave-Flipflops immer komplementär sind, wird jeweils nur ein Eingang des Flipflops freigeschaltet. Die so entstandene Konstruktion heißt JK-Flipflop (Abb. 4.60 und 4.61). JK-Flipflops wechseln bei einem konstanten periodischen Eingangssignal immer ihren Zustand mit der halben Frequenz. Sie wurden daher vorzugsweise in Frequenzteilern und Zählschaltungen eingesetzt. Durch bestimmte Eingangsbelegungen von J und K können mithilfe eines JK-Flipflops sowohl ein RS- als auch verschiedene Typen des im nächsten Abschnitt zu diskutierenden D-Flipflops realisiert werden. In modernen Digitalschaltungen ist das JK-Flipflop anzutreffen. Die Universalität dieser Schaltung ist nicht notwendig, und die im Folgenden diskutierten D-Flipflops sind wesentlich flächengünstiger zu implementieren.

4.3.1.3 D-Flipflop

Wird die Schaltung eines getakteten RS-Flipflops mit einem Inverter erweitert, entsteht ein D-Latch (oder Data- bzw. Delay-Latch für Signalspeicher). Dabei wird vor der Torschaltung des Reset ein Inverter oder *NICHT* geschaltet. Dieser invertiert das Set-Signal, sodass das Set- und Reset-Signal immer komplementär sind. Durch die Torschaltung ist das Flipflop in den kurzen Zeiten des Signalwechsels der Set-Leitung vor undefinierten Zuständen geschützt (Abb. 4.62 und 4.63).

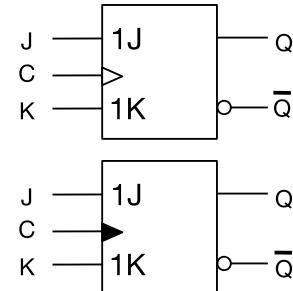
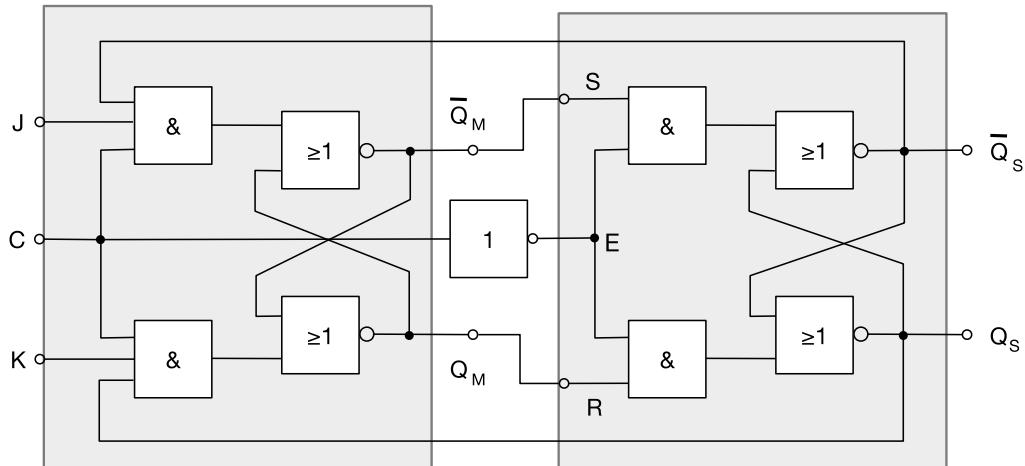


Abb. 4.60 Schaltzeichen JK-Flipflop

D-Latch



4

Abb. 4.61 JK-Flipflop

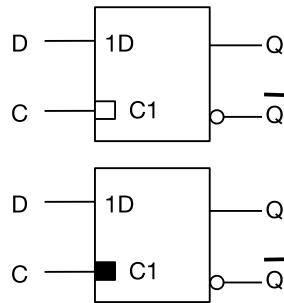


Abb. 4.62 D-Latch

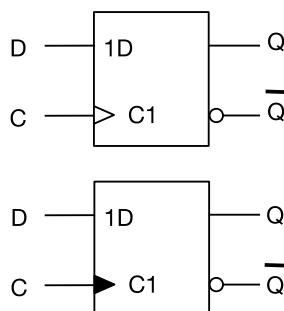


Abb. 4.64 D-Flipflop

Die Torschaltung und der Inverter zusammen schützen das Flipflop vor Signalbelegungen, die zu bistabilen oder undefinierbaren Zuständen führen. Der Eingang des D-Flipflops wird mit D bezeichnet. Erst wenn der Eingang C auf 1 liegt, übernimmt das Flipflop den Zustand auf der D -Leitung. Je nachdem ob D gesetzt oder nicht gesetzt ist, wird das D-Flipflop selbst gesetzt oder zurückgesetzt. Allerdings reagiert das Flipflop auf jede Pegeländerung des Eingangs D , während C wahr ist. Daher sollte D während der gesamten Zeit in der C gesetzt ist, auf 1 oder 0 sein. Ändert der Eingang D seinen Pegel, während C eingeschaltet ist, kann es zur Übernahme falscher Werte kommen. Die zeitlichen Bedingungen dafür hängen von den Verzögerungszeiten der einzelnen Gatter ab. Bei der Verwendung von D-Flipflops ist also darauf zu achten, dass während des Taktpulses immer ein konstanter Wert am D -Eingang anliegt (Abb. 4.64 und 4.65).

Die Zeit, in der das Eingangssignal an einem D-Latch stabil anliegen muss, kann verkürzt werden. Dazu wird hinter dieses ein zweites Latch geschaltet. Das zweite Latch wird dabei von einem invertierten Takt angesteuert. In der angelsächsischen Literatur heißen das pegelgesteuerte D-Flipflop D-Latch und das Master-Slave-D-Flipflop D-Flipflop. Da bei dieser Schaltung das Eingangssignal nur während der kurzen Flanke des Taktsignals übernommen wird, nennt man das D-Master-Slave-Flipflop auch „flankengesteuertes D-Flipflop“. In der Elektronik heißen Signale, die ein bestimmtes Verhalten auslösen, Trigger (engl. für auslösen) oder Trigger-Signal. Daher spricht man auch von pegel- und flankengetriggerten D-Flipflops. Im Signalverlauf des D-Flipflops ist deutlich zu

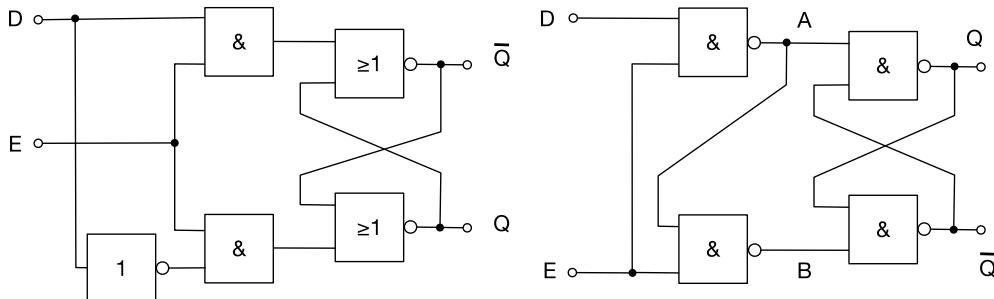


Abb. 4.63 D-Latch

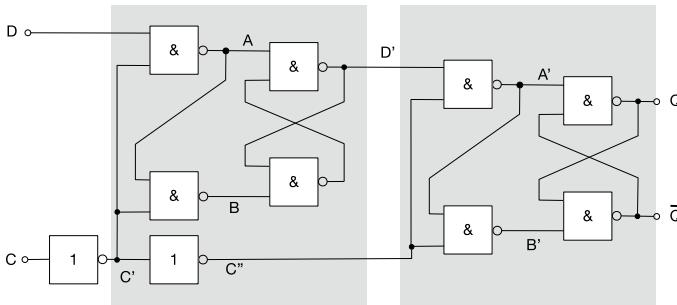
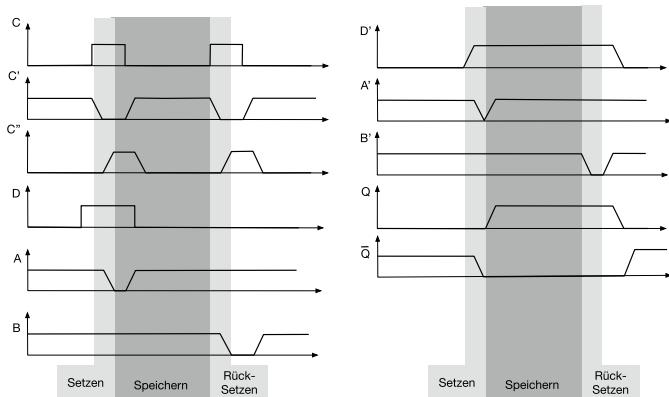


Abb. 4.65 D-Flipflop

erkennen, dass die Zeit, in der D aktiv sein muss, wesentlich verkürzt wurde (Abb. 4.66). In realen Schaltungen sind die Verzögerungen kürzer als in der Abbildung gezeigt, da die Gatter schneller schalten. Werden die Verzögerungszeiten der einzelnen Gatter kleiner, verringert sich auch die Zeit, in der D aktiv sein muss. In praktischen Implementierungen wird der Wert von D tatsächlich in der kurzen Zeit der steigenden Flanke von C übernommen. Natürlich gibt es sowohl positiv als auch negativ getriggerte Flipflops. Erstere übernehmen den Eingangswert, wenn am Takteingang eine positive oder steigende Flanke anliegt; Letztere speichern das Eingangssignal bei einer fallenden Taktflanke.

4.3.2 Transistorzellen

Noch bis Anfang der 1970er-Jahre standen den Rechnerarchitekten nur teure Techniken wie beispielsweise der Magnetkernspeicher zur Verfügung. Diese Speicher waren aufwen-



■ Abb. 4.66 Signalverlauf des D-Flipflops

dig herzustellen, hatten eine enorme Baugröße und benötigten viel elektrische Energie. Zu der Zeit waren Halbleiterspeicher bedingt durch die geometrischen Strukturen und die unzuverlässigen Fertigungsprozesse ausgesprochen teuer. Insbesondere waren die verwendeten Wafer noch klein, und auch die einzelnen Chips besaßen keine große Fläche, um die Ausbeute bei der Fertigung wirtschaftlich zu halten. Mit Einführung des Planarprozesses und der Herstellung skalierbarer integrierter Schaltungen fielen die Preise für Halbleiterspeicherbausteine. Diese Entwicklung sorgte dafür, dass Halbleiterspeicher magnetische auf der Ebene der Programmspeicher bis in die 1980er-Jahre vollständig ablösten. Ein ähnlicher Prozess ist zurzeit bei den Massenspeichern, die noch immer magnetisch dominieren (Festplatten und Bandlaufwerke), zu beobachten. Durch die kostengünstige Herstellung von Flashspeichern und die hohe Nachfrage nach diesen für mobile Endgeräte ist auch der Markt für Massenspeicher auf dem Weg zu einem Halbleitermarkt. Da heute nur noch die Halbleiterspeicher als Programm- und Datenspeicher eingesetzt werden, sollen ausschließlich diese im Folgenden beschrieben werden.

Flipflops lassen sich vom Prinzip her auch nutzen, um den Hauptspeicher eines Rechners aufzubauen. Allerdings ist der Flächenverbrauch eines flankengesteuerten D-Master-Slave-Flipflops für einen großvolumigen Speicher viel zu hoch. In der Halbleiterindustrie hat man früh kompakte und flächenoptimale Technologien entwickelt. Im Folgenden sollen daher einige wichtige Schaltungstechniken eingeführt werden. Technisch unterscheidet man verschiedene Speichervarianten. Dies sind zum einen flüchtige und zum anderen nicht flüchtige Speicher. Bei einem flüchtigen Halbleiterspeicher geht der Inhalt

verloren, wenn die Stromversorgung unterbrochen wird. Ein nicht flüchtiger Speicher behält seine Daten auch ohne eine aktive Energiezufuhr.

Der Leser mag nun einwenden, dass flüchtige Speicher unbrauchbar sind – insbesondere, da man Rechenergebnisse archivieren möchte und ein Stromausfall mit anschließendem Verlust des Speicherinhaltes investierte Rechenzeit vergeudet. Berechnungen müssen in einem solchen Fall wiederholt werden und aufwendige Verfahren würden nie zu Ende gerechnet werden – ein allgegenwärtiges Problem in der Anfangszeit des maschinellen Rechnens. Leider ist es so, dass nicht flüchtige Speicher teuer sind und sich flüchtige günstig herstellen lassen. Dies gilt mit einer Ausnahme: Verzichtet man darauf, Daten überschreiben zu können, dann kann dieser ausgesprochen wirtschaftlich gefertigt werden. Neben den flüchtigen und nicht flüchtigen Speichern unterscheidet man daher noch zwischen wiederbeschreibbaren und den nicht wiederbeschreibbaren oder Nur-lese-Speichern. Ersteres nennt man einen RAM (Speicher mit wahlfreiem Zugriff, engl. „random-access memory“) und Zweiteres einen ROM (engl. „read-only memory“). Ein ROM ist nicht flüchtig, aber günstig herzustellen. Allerdings wird die günstige Produktion durch ausschließliches Lesen des Speichers erkauft.

Neben den dynamischen flüchtigen Speichern und den fest verdrahteten Speichern gibt es noch Speicher, die bei Wegfall der Versorgungsspannung den Inhalt speichern, aber trotzdem programmierbar sind (PROM). Dabei wird zwischen löschenbaren, z. B. durch ultraviolettes Licht löschenbaren (EPROM) und elektrisch löschenbaren Speichern (EEPROM) unterschieden. EEPROMs können byteweise oder blockweise beschrieben oder gelesen werden. Die zweite Variante wird Flash-EEPROM genannt. EEPROMs haben verglichen mit RAM-Bausteinen den Nachteil, dass sie nicht beliebig oft neu programmiert werden können, sondern nur eine begrenzte Anzahl von Programmierungsvorgängen zulassen. Ist ein wahlweises Schreiben und Lesen gewünscht, hat man die Wahl zwischen mehreren Speichertechnologien. Zum einen kann ein Bit magnetisch oder elektrisch gespeichert werden. Magnetische Speicher sind der Ringkernspeicher, das Magnetband und die Festplatte. Magnetband und Festplatte sind günstige Massenspeicher und sollen zunächst nicht besprochen werden.

Grundelement eines Speichers ist die Speicherzelle zum Ablegen eines einzigen Bits. Die Zelle eines Halbleiterspeichers in MOS-Technik kann einfach erzeugt werden, indem zwei Inverterschaltungen über Kreuz verschaltet werden, d. h., der Ausgang des ersten Inverters ist mit dem Eingang des zweiten Inverters verbunden und umgekehrt (Abb. 4.67). Ein Bit wird in der Speicherzelle gespeichert, indem der Wert ei-

SRAM-Zelle

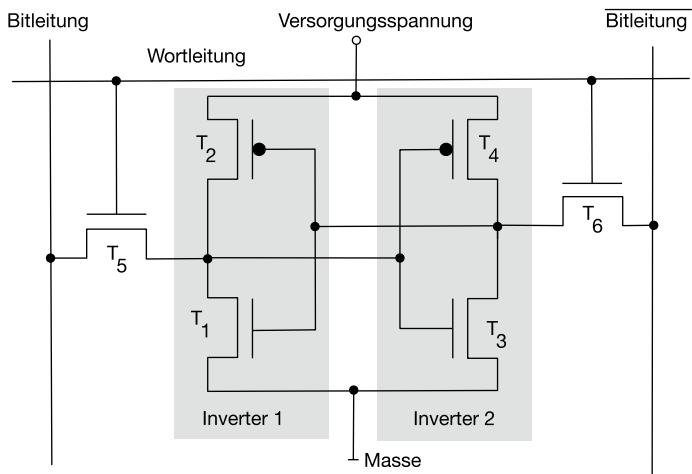


Abb. 4.67 6-T-SRAM-Zelle

ner Bitleitung oder besser einer Schreib-/Leseleitung von der Schaltung gelesen wird. Zu diesem Zweck befindet sich am Eingang der jeweiligen Inverter ein Transistor, der als Tor dient. Ein weiterer Transistor bei dieser Anordnung ist dabei mit einer invertierten Schreib-/Leseleitung verbunden. Dadurch ist es möglich, metastabile Zustände, die die „Static-random-access-memory (SRAM)-Zelle“ zerstören würden, zu vermeiden. Wird die Zelle über die Wortleitung angesprochen, sind die Schreib- oder Lesetransistoren (T_5 und T_6) durchgeschaltet. Das Bit von der Bitleitung oder vielmehr das inverse Bit der negierten Schreib- und Leseleitung werden in die Speicherschaltung übernommen. Das Lesen der Zelle funktioniert ähnlich. Nur müssen dann von der Speicherzelle die Schreib- und Leseleitung und im anderen Fall die negierte Bitleitung umgeladen werden. Eine solche SRAM-Zelle wird ebenfalls als Flipflop bezeichnet. Das Akronym SRAM steht dabei für engl. „static random-access memory“. Der Unterschied zu den dynamischen Zellen wird bei der späteren Diskussion dieser klar.

Da in einem Halbleiterspeicher möglichst viele Zellen auf einer geringen Chipfläche platziert werden sollen, kann das Layout dieser kompakter ausgeführt werden. Kompakter meint, die Bauelemente sind kleiner als ein im Verhalten gleiches Gatter in CMOS-Technik. Die Idee ist, die beiden PMOS-Transistoren T_2 und T_4 durch hochohmige Widerstände zu ersetzen. Wird im MOS-Prozess eine weitere Lage Polysilizium aufgebracht, kann man diese dafür nutzen, einzelne Widerstände (R_1 und R_2) über den Transistoren zu realisieren. Es sei daran erinnert, dass in einem selbstjustierenden MOS-Prozess

die Polysilizium-Gates ebenfalls dotiert werden und dadurch niederohmig werden. Eine zweite Schicht über den Gates wird erst nach diesem Schritt aufgebracht, ist daher nicht dotiert und daher hochohmig. Durch diesen schaltungs- und prozess-technischen Trick in die Höhe auszuweichen und die Widerstände auf der Transistorschaltung zu integrieren, ist es möglich, ausgesprochen kompakte Zellen zu bauen.

Eine SRAM-Zelle hat mindestens 4 Transistoren. Zwar sind diese integrierten Siliziumschaltungen verglichen mit Magnetkernspeichern ausgesprochen klein, die Fläche eines RAM lässt sich allerdings noch weiter verkleinern. Die Idee zum „Dynamic-random-access-memory (DRAM)-Baustein“ geht auf Robert H. Dennard (geboren 1932), einem bei IBM beschäftigten Elektroingenieur zurück. Im Jahr 1966 verwendete Dennard die Eigenschaft der MOS-Struktur, Ladung zu speichern, um eine Kapazität als Speicherzelle zu nutzen (Abb. 4.68 und 4.69). Um ausreichend Ladungsträger zu erhalten, wird eine große Fläche als Platte eines Kondensators notwendig. Diese kann als zusätzliche Polysiliziumschicht in den Fertigungsprozess integriert und durch das Ätzen von Gräben in den Halbleiter ebenfalls 3-dimensional in die Tiefe gebaut werden. Eine solche Struktur ist flächenminimal, hat aber einen Nachteil. Im Mikro- oder inzwischen Nanometerbereich besteht die gespeicherte Ladungsmenge nur noch aus sehr wenigen Ladungsträgern. Da die Isolation der Struktur im Vergleich zu makroskopischen Größenordnungen ebenfalls geringer ist, verliert eine derartige Zelle vergleichsweise viel Ladung durch Leckströme.

Aus diesem Grund muss der Kondensator regelmäßig neu aufgeladen werden. Der Vorgang heißt Refresh und der Name dynamisch leitet sich aus der dauernden Wiederholung dieses Prinzips ab. Die Schaltung ist also ständig aktiv. Neben dem Kondensator wird nur noch ein Transistor benötigt. Dieser trennt die Speicherzelle von den Bitleitungen und wird geöffnet, um Ladung zu speichern oder zu lesen. Die orthogonal in der Schaltung angeordnete Wortleitung schaltet dabei den Transistor ein. Damit kann 1 Bit in einer Zelle gespeichert werden. In der Fabrikation wird die einfache Schaltung noch weiter optimiert. So gehen der Schaltransistor und die Speicherzelle direkt ineinander über, sodass zwischen den beiden Bauelementen in der Praxis keine Verdrahtung erfolgen muss. Durch eine alternierende Anordnung im Layout lässt sich ein DRAM auf wenig Chipfläche realisieren (Abb. 4.70).

In einem ROM werden die zu speichernden Werte fest eingestellt. Wort- und Bitleitung werden daher in einem ROM direkt miteinander verbunden. In dem Fall ist an der Speicherstelle eine 1 gespeichert. Immer wenn die Wortleitung 1 ist, ist dann die Bitleitung ebenfalls 1. Besteht keine Verbindung, ist

DRAM-Zelle

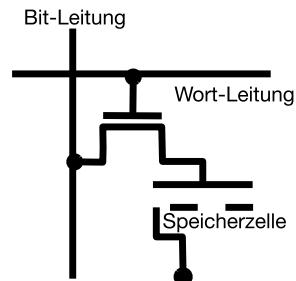


Abb. 4.68 DRAM-Zelle

ROM-Zelle

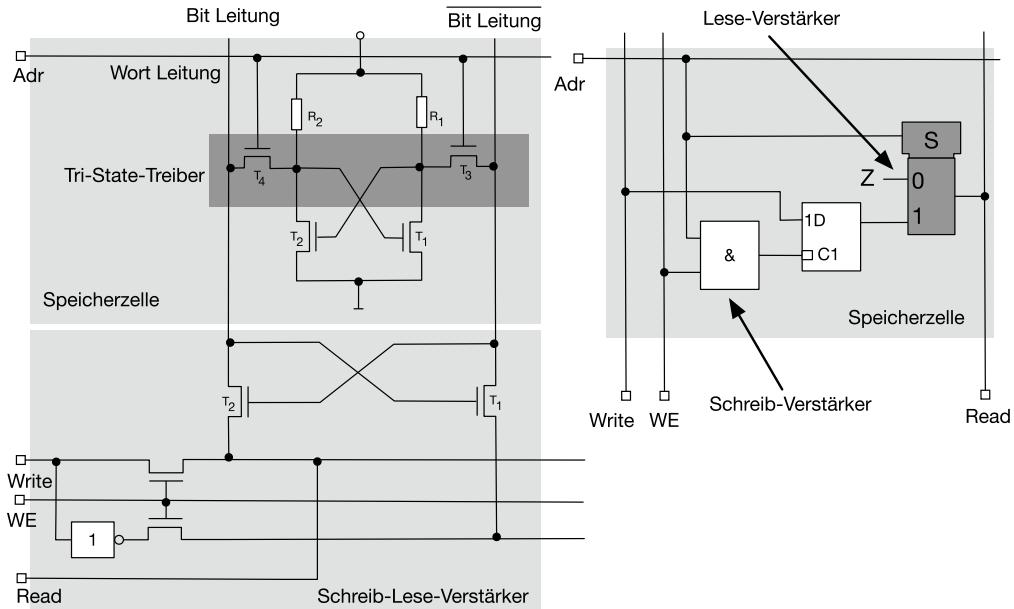


Abb. 4.69 Kompakte SRAM-Zelle mit Schreib-Leseverstärker (einmal pro Spalte)

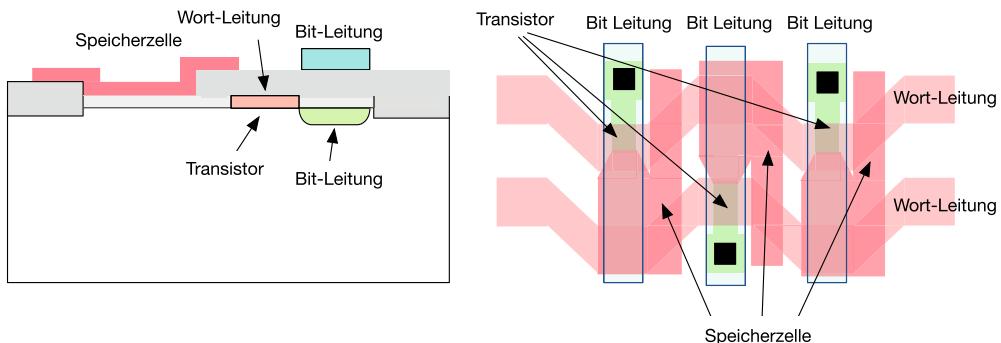


Abb. 4.70 Aufbau und Layout einer DRAM-Zelle

bei anliegender Wortadressierung die entsprechende Zelle 0. Die Verbindung wird in MOS-Schaltungstechnik durch einen Transistor hergestellt. Alternativ kann dafür auch eine Diode verwendet werden. Die Abb. 4.71 zeigt verschiedene Möglichkeiten, eine 1 oder eine 0 in einem ROM zu speichern.

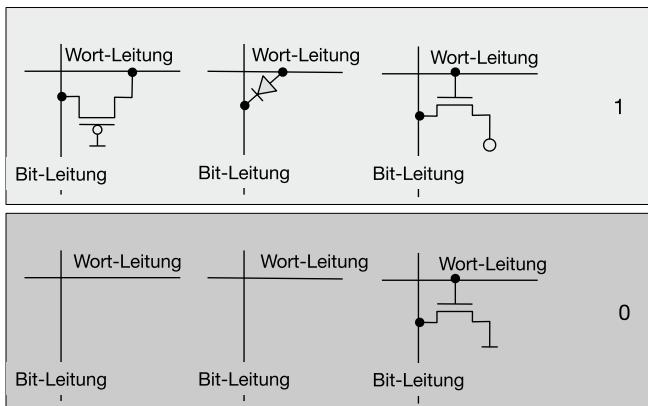


Abb. 4.71 Varianten von ROM-Zellen

Ein elektrisch veränderbares ROM ist in der Abb. 4.72 gezeigt. Die Schaltung einer PROM-Zelle ist dieselbe wie bei einer ROM-Zelle. Nur wird bei einem EEPROM an jedem Kreuzungspunkt der Bitleitung mit der Wortleitung ein Transistor implementiert. Dieser unterscheidet sich aber von bisher eingeführten MOS-Transistorstrukturen. Ein Transistor einer EEPROM-Zelle hat zwei unabhängige und übereinanderliegenden Gates aus Polysilizium. Eines der beiden Gates, das Steuer-Gate ist dabei direkt mit der Wortleitung verbunden. Ist dieses mit positiver Ladung belegt und wird das Diffusionsgebiet unter dem Tunnelbereich mit einer gesetzten Bitleitung verschaltet, können Ladungsträger auf das zweite Gate, das Speicher-Gate tunneln. Das Bauelement ist nun so dimensioniert, dass das Tunneln nur bei hohen Spannungen im Kanal erfolgen kann. Dadurch wird vermieden, dass im Lesebetrieb Ladungsträger vom Speicher-Gate zurück in das Diffusionsgebiet tunneln können. Im Lesemodus liegt am Steuer-Gate keine Spannung an. Lediglich das Speicher-Gate sorgt, wenn es denn programmiert wurde, dafür, dass der Transistor der EEPROM-Zelle leitend ist und so ein Wert ausgelesen oder gespeichert werden kann. Ob Ladung auf dem Speicher-Gate als 0 oder 1 interpretiert wird, hängt dabei von der Schaltung der ROM-Zelle ab, also ob der leitende Transistor eine 1 oder eine 0 darstellt.

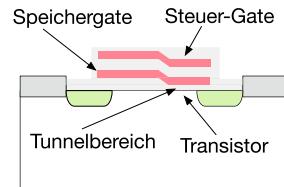


Abb. 4.72 EEPROM-Zelle

4.4 Speichermatrizen

Speicherzellen lassen sich in Blöcken oder Registern anordnen. Dabei ergeben mehrere parallel geschaltete Zellen einen Speicherblock. In der Regel spricht man von einem Register, wenn der Speicherblock direkt in einer logischen Schaltung verbaut wurde und dort Daten temporär speichert. Solche Registerblöcke sind oft aus D-Latches oder D-Flipflops aufgebaut. Ist das Register aus D-Flipflops konstruiert, kann es zur Entkopplung von Daten genutzt werden, beispielsweise wenn das Ergebnis einer Berechnung in das gleiche Register gespeichert wird, in dem einer der Operanden der Operation steht. Durch das Master-Slave-Prinzip sind am Ausgang die Daten gültig, obwohl das Register bereits ein neues Datum liest. Immer wenn an der Taktleitung eine Flanke eines Signals anliegt, wird der Wert an den Datenleitungen für alle Bits des Wortes gleichzeitig übernommen. Dieses Verhalten spiegelt sich im Schaltzeichen wieder. Mithilfe eines Ladesignals kann das Register mit neuen Daten beschrieben werden (Abb. 4.73 und 4.74).

Speichermatrix



Abb. 4.73 Schaltzeichen:
Register

Werden mehrere Register oder allgemein Blöcke aus Speicherzellen untereinander angeordnet, entsteht ein Feld aus Speicherelementen (Abb. 4.75). Dabei kann jedes Register durch eine Auswahlleitung einzeln angesprochen werden. Ist ein Register ausgewählt, kann dieses über die Datenleitung mit Daten beschrieben, oder es kann gelesen werden. Ein Speicherfeld oder eine Speichermatrix kann aus beliebigen Speicherzellen aufgebaut werden. Werden ROM-Zellen verwendet, entsteht eine Nur-lese-Matrix, werden SRAM-Zellen verbaut, ergibt sich ein SRAM-Speicher, und bei Verwendung von DRAM-Zellen ein DRAM-Speicher. Allen derartigen Speichermatrizen ist gemein, dass das Ansprechen eines Speicherblocks durch paralleles Setzen einer Auswahl- oder Adressleitung erfolgt. Die Auswahl der jeweiligen Speicherzelle hat einen großen Einfluss auf die Geschwindigkeit, mit der ein Rechner arbeiten kann. Speichermatrizen sind regelmäßige Strukturen und daher kostengünstig als integrierte Schaltungen zu fertigen. Trotzdem sollen Speicherbausteine immer

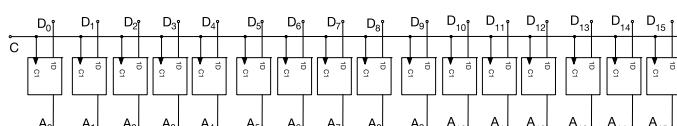


Abb. 4.74 Register

4.4 · Speichermatrizen

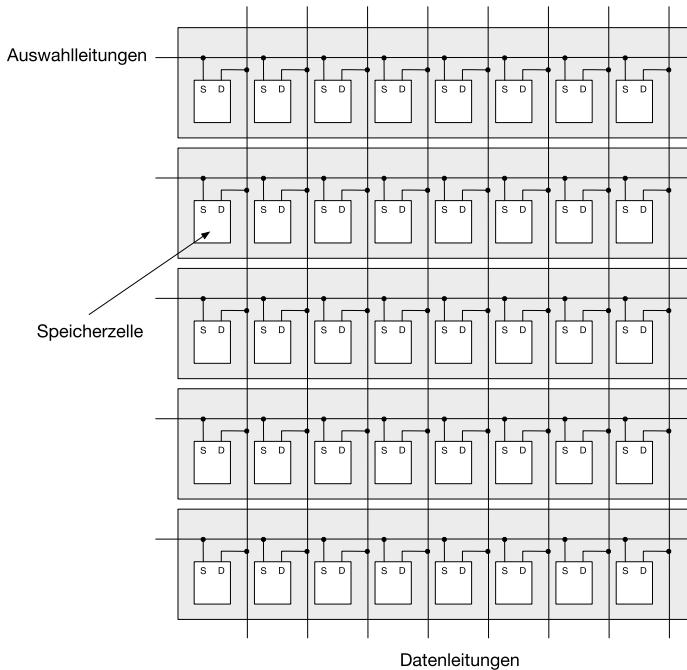


Abb. 4.75 Speichermatrix

günstig sein, da in einem modernen Rechner sehr viel Speicher verbaut und dieser nicht groß genug sein kann.

Der Aufbau eines Nur-lese-Speichers ist einfach. Dabei wird jede 1 durch einen Transistor dargestellt, der die Wort mit der Bitleitung verbindet. Bei einer 0 fehlt der Transistor. Da dieser als Schalter arbeitet, zieht er die Bitleitung auf die Versorgungsspannung, wenn die entsprechende Wortleitung aktiviert ist. Dies kann auch invers implementiert werden. Eine vorgespannte Bitleitung wird auf Masse gezogen, wenn die Wortleitung aktiv ist. In dem Fall ist dann bei einem vorhandenen Transistor eine logische 0 gespeichert. In MOS-Technik gibt es somit 4 verschiedene Varianten, einen ROM-Speicher aufzubauen: je zwei mit NMOS- und zwei mit PMOS-Transistoren. In der Regel werden ROM-Speicher maskenprogrammiert. Maskenprogrammiert bedeutet, dass alle Transistoren implementiert werden, egal ob sie genutzt werden oder nicht. Nur die Transistoren, die eine 1 darstellen, werden dann mit Kontakten ausgestattet und bei Aufbringen der Verdrahtungsebenen mit den Bit- und Wortleitungen verbunden. Dieses Vorgehen hat den Vorteil, dass alle Prozessschritte in der Fertigung gleich sind und ausschließlich die letzte Verdrahtungsebene kundenspezifisch ist. Da die Chipgeometrie nicht proportional zum logischen Aufbau des Speichers ist, wer-

ROM-Matrix

SRAM

den die Zellen über eine Wortleitung und eine zusätzliche Reihenauswahl angesprochen (Abb. 4.76). Diese Aufteilung in Bänke hat noch einen weiteren Vorteil: Durch Verringerung der Leitungslängen erhöht sich die Zugriffsgeschwindigkeit zum Schreiben und Lesen.

Ein Zugriff auf die ROM-Speichermatrix erfolgt ähnlich wie bei einem RAM, mit dem Unterschied, dass die Daten nicht geschrieben werden können. Bei einem ROM können daher die Schreib- und die Leseleitung entfallen. Da die Daten eines ROM von der Schaltung sofort bereitgestellt werden können, sind ROM-Speicher genauso schnell wie SRAM-Speicher. Ein ROM wird eingesetzt, um Programmcode eines Rechners dauerhaft zu speichern. Daher sind ROMs meist in eingebetteten Systemen zu finden. Zwar kommen dort auch häufig Flashspeicher zur Anwendung. Wenn hohe Stückzahlen eines Produkts gefertigt werden und die Kosten eine große Rolle spielen, sind aber ROMs ungeschlagen und werden daher immer noch gerne eingesetzt.

Ein SRAM („static random-access memory“) ist aus vielen SRAM-Speicherzellen aufgebaut. Diese Transistorzellen sind reihen- und spaltenweise auf dem Chip angeordnet. Auch in SRAMs werden die Zellen über eine Wortleitung und eine zusätzliche Reihenauswahl angesprochen. Jede SRAM-Zelle ist mit dieser über eine eigene Bitleitung und eine zu dieser komplementäre Leitung verbunden, sodass die Speicherinhalte immer differenziell geschrieben und gelesen werden können. Dies reduziert die zum Treiben der Verbindungsleitungen notwendigen Ströme und verbessert die Zuverlässigkeit der Schaltung.

Die beiden Schreib- oder Leseleitungen können über einen Transistor kurzgeschlossen werden. Diese PMOS-Transistoren, in jeder Reihe einer werden über ein Signal ϕ_p geschaltet. Solange $\phi_p = 0$ ist, sind die Transistoren leitend, und ein aktiver Pegel auf ϕ_p schaltet den Transistor aus und beendet damit den Kurzschluss zwischen den Leitungen. Mithilfe dieses Schalters lassen sich die Bitleitungen vorspannen. Das bedeutet, nachdem ein Wert aus einer Zelle gelesen wurde, ist entweder die Bitleitung oder die komplementäre Leitung 1, oder in anderen Worten, sie entspricht der Versorgungsspannung. Wird nun $\phi_p = 0$, schaltet der PMOS-Transistor durch und sorgt für einen Ladungsausgleich auf den beiden Leitungen. Diese haben dann eine Spannung, die der halben Versorgungsspannung entspricht. Die Idee hinter dieser Schaltungstechnik ist, dass die Speicherzellen nicht die volle Kapazität der langen Bitleitungen aufladen müssen. Dadurch kann die Geometrie der Speicherzellentransistoren kompakter ausfallen, denn Transistoren, die viel Strom treiben, müssen auch entsprechend breit und damit groß ausgelegt werden. Ist in einer Zelle 1 Bit gespeichert, so werden die Bitleitung ein wenig

4.4 · Speichermatrizen

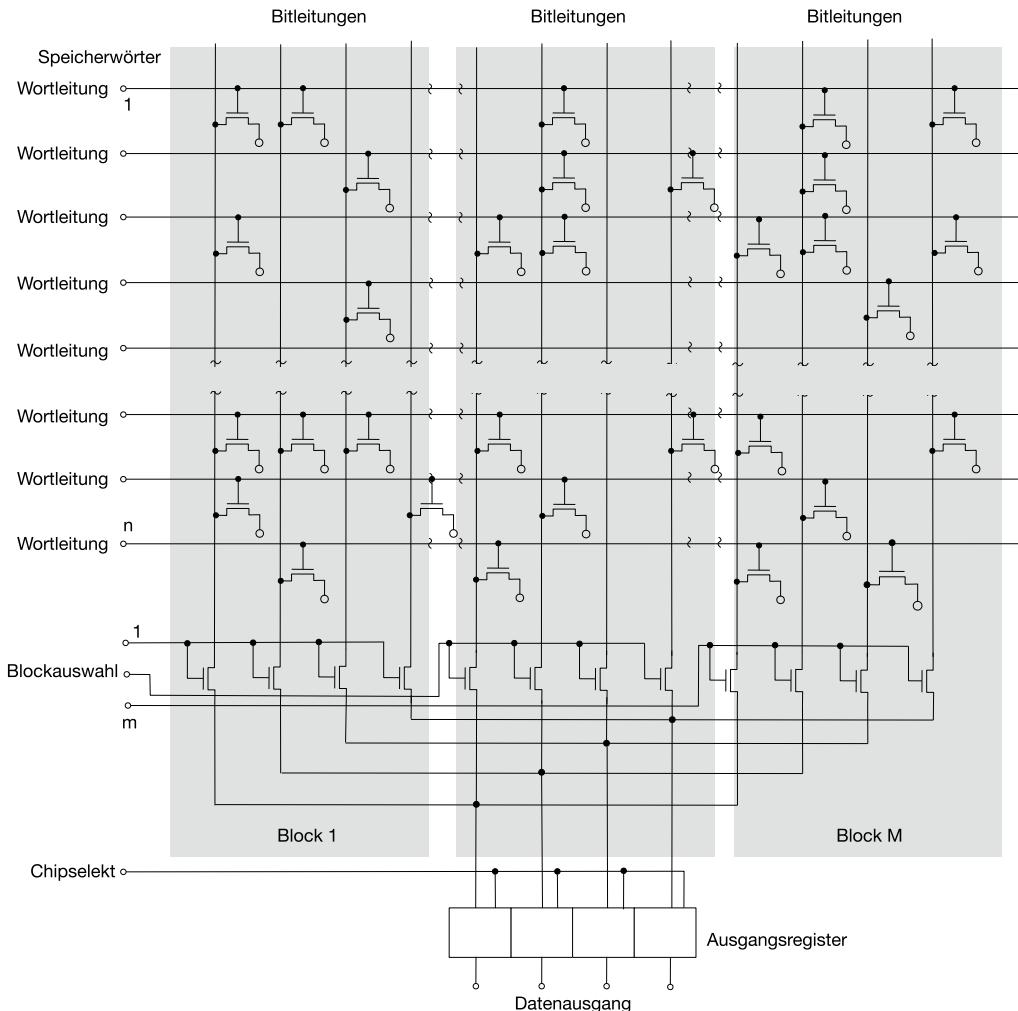


Abb. 4.76 ROM-Speichermatrix

angehoben und die komplementäre Leitung abgesenkt. Hinter der Auswahlschaltung der Reihe befindet sich ein Differenzverstärker als Leseverstärker. Dieser misst die Differenz zwischen der Bitleitung und ihrer komplementären Leitung und verstärkt diese auf den Wert der Spannungsversorgung.

Soll der SRAM-Baustein gelesen werden, wird zunächst eine gültige Adresse an den Adresseingang angelegt. Dieses Wort untergliedert sich zu einem Teil zur Spalten- und einem Abschnitt zur Reihenauswahl. Die Adresse muss vor dem Signal ϕ_{sel} stabil an den Adressleitungen anliegen, da ϕ_{sel} den Lese- oder Schreibvorgang startet. Gleichzeitig mit ϕ_{sel} muss auch der Leitungskurzschluss der Bitleitungen durch ϕ_p aufgehoben

DRAM-Matrix

werden. Kurze Zeit später ist dann die Zeile des Speichers, die gelesen werden soll, ausgewählt, und mit dem Signal ϕ_r werden dann der Leseverstärker aktiviert und damit der Datenausgang gültig. In dem dargestellten Beispiel müssen die einzelnen Zellen sequenziell gelesen und die jeweiligen Bits in einem Ausgangsregister zwischengespeichert werden. Wird jede Spalte mit einem eigenen Leseverstärker ausgestattet, kann der Speicher auch parallel ausgelesen werden. Dazu wird die Spaltenauswahl zu einer Blockauswahl umgebaut, und für jeden n -spaltigen Block müssten n Verstärker implementiert werden.

In einem DRAM-Speicher sind die Speicherzellen in Spalten und Reihen angeordnet. Um die Zugriffsgeschwindigkeit auf einen Speicherbaustein zu optimieren, werden die Speichermatrizen in einem DRAM-Baustein in Blöcken organisiert. Diese Blöcke lassen sich getrennt adressieren. Dieses erfolgt oft sequenziell, um Adressleitungen zu sparen.

Der Zugriff zum Schreiben oder Lesen einer DRAM-Matrix ist ähnlich einem SRAM-Chip (Abb. 4.77). In der Regel müssen zwei getrennte Adressen an den Baustein angelegt werden: zunächst einmal eine Adresse für die Reihenauswahl und später die Adresse für die Zeilenauswahl. Die Signale „row access select/strobe“ (RAS) und „column access select/strobe“ (CAS) starten dann den jeweiligen Lesevorgang. RAS wählt die Wortleitung aus, und dieses bewirkt eine Verschiebung der Spannungspiegel auf der entsprechenden Bitleitung. CAS sorgt anschließend dafür, die Flipflops des dazugehörigen Blocks umzukippen und das Signal auf der Bitleitung zu verstärken. Später liegen dann die Daten am Ausgang des RAM-Bausteins an. Bedingt durch den getrennten Zugriff auf die zwei Adressteile benötigt Lesen eines mit DRAM imple-

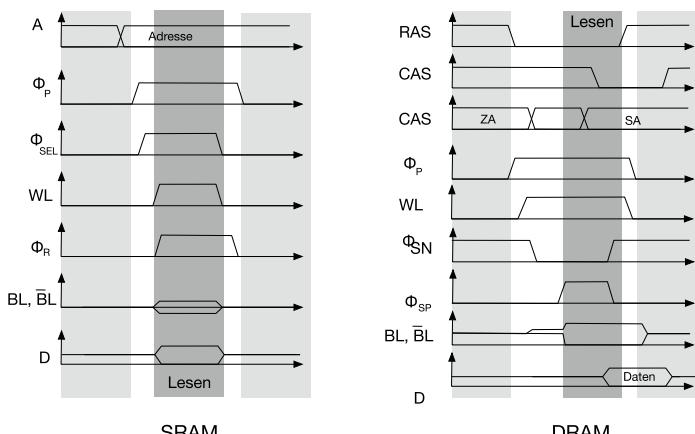


Abb. 4.77 Lesezyklen SRAM und DRAM

mentierten Speichers mehr Zeit als der Zugriff auf den Inhalt eines SRAM. Hinzu kommt ggf. die Verzögerung, wenn der DRAM-Chip gerade einen Refresh-Zyklus hat und aus diesen Gründen auf den Baustein nicht zugegriffen werden kann.

Durch einen schaltbaren Kurzschluss können die Bit- und die komplementäre Bitleitung vorgespannt werden, damit die in den DRAM-Zellen gespeicherte Information ausreichend ist, die jeweilige Leitung in Richtung zu einer 0 oder einer 1 mit wenig Ladung zu kippen. Jede Bitleitung ist in jeder zweiten Zeile mit einer Speicherzelle versehen. Dies gilt versetzt auch für die komplementäre Leitung. Diese Schaltungstechnik sorgt dafür, dass Bitleitung und deren Komplement immer eine klare Spannungsdifferenz nach dem Lesen aufweisen. Die in der Speicherzelle gespeicherte Ladung wird somit differenziell verstärkt. Zusätzlich befindet sich noch vor der Blockauswahl ein einfaches Flipflop. Dieses reagiert auf die Reihenauswahl, die wie ein Set-Signal wirkt, und kippt dann in die durch die beiden Bitleitungen vorgegebene Richtung. Dadurch wird die Bitleitung entweder auf 0 oder auf 1 mit der vollen Spannung gesetzt, je nachdem, ob die Schaltung Ladung enthielt oder nicht. Ist eine der Bitleitungen wieder auf die Versorgungsspannung angehoben, wird die gelesene Speicherzelle erneut aufgeladen (engl. „refresh“). Damit sorgt die Schaltungstechnik mit dem Flipflop am Ende der Leseleitung auch für ein automatisches Wiederbeschreiben des Speichers. Dies ist zwingend notwendig, da das Lesen einer kapazitiven DRAM-Speicherzelle immer ladungszerstörend ist und so ein eigener Schreibzyklus vermieden werden kann. Da die Kondensatoren der Speicherzellen ihre Ladung mit der Zeit verlieren (Leckströme in das Substrat), kommt dieser Technik noch eine weitere Funktion zu. Ein einfaches periodisches Aktivieren der Speicherzellen durch einen Lesevorgang frischt automatisch die Speicherzellen auf. Zu diesem Zweck ist in dem DRAM-Baustein ein Refresh-Controller eingebaut. Dieser erzeugt in regelmäßigen Abständen ein Adresssignal für die entsprechenden Zeilen und Blöcke und liest so periodisch die einzelnen Zellen. Allerdings wird das Ergebnis des Lesens nicht in das Datenregister am Ausgang übernommen. Der Lesevorgang dient lediglich zum Wiederbeschreiben der ihre Ladung verlierenden Speicherzellen (☞ Abb. 4.78).

4.5 Logikfelder

Mithilfe des selbstjustierenden Planarprozesses lassen sich Transistoren günstig im Mikrometerbereich und inzwischen auch im Nanometerbereich herstellen. Die Integration vieler Schalter monolithisch auf einem Siliziumkristall (Chip) wird

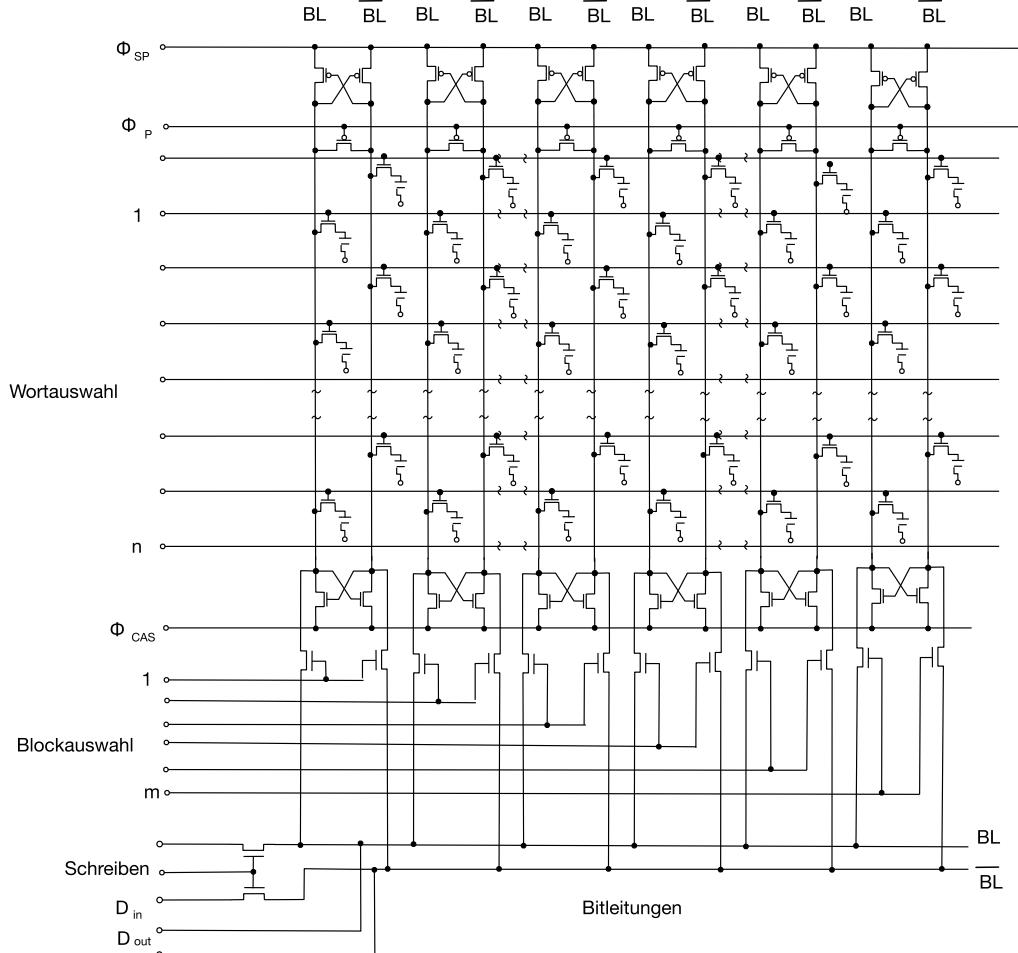


Abb. 4.78 Speichermatrix eines DRAM

„integrierte Schaltung“ (engl. „integrated circuit“, IC) genannt. Inzwischen lassen sich auf einem solchen die genannten Siliziumplättchen Milliarden von Transistoren implementieren. Moderne Computer sind aus den im vorherigen Abschnitt diskutierten Grundschaltungen aufgebaut. Aus technischen sowie Kostengründen werden integrierte Schaltungen in unterschiedlichen Halbleiterprozessen gefertigt. Grundlage ist zwar immer der Planarprozess, jedoch sind für hochintegrierte Halbleiterspeicher (DRAM) andere Prozessschritte notwendig als für reine Logikschaltungen. Auch für diese existieren ebenfalls unterschiedliche Fertigungsverfahren, je nachdem, in welcher Stückzahl die spätere Schaltung produziert wird. So sind Prozessoren für Desktopsysteme und Chips für Smartphones in hochintegrierten Prozessen gefertigt, während Schaltungen für

Spezialanwendungen häufig nur anwendungsspezifisch auf der Basis von standardisierten Logikfeldern „verdrahtet werden“. Insbesondere ein standardisierter und rechnergestützter Entwurf ermöglicht es, Halbleiter flexibel und kostengünstig einzusetzen. Dabei kann je nach erwartetem Produktionsvolumen des späteren Produkts ein anderes Herstellungsverfahren genutzt werden.

Da alle Logikbausteine und Prozessoren in ähnlichen, fast gleichen CMOS-Prozessen hergestellt werden, ist eine Unterscheidung der Halbleiterprodukte nach den jeweiligen Entwurfsverfahren notwendig. Im Laufe der Entwicklung integrierter digitaler Schaltungen haben sich unterschiedliche Entwurfsstile und damit verbundene Schaltungstechniken herausgebildet: zum einen hochintegrierte Speicherchips, die für den Massenmarkt hergestellt werden, und auf der anderen Seite anwendungsspezifische Schaltkreise („application-specific integrated circuits“, ASICs), die in kleinen Stückzahlen gefertigt werden. Lohnt sich bei den hochintegrierten Prozessoren und Speichern die Optimierung selbst einzelner Transistoren im Zusammenspiel mit dem Herstellungsprozess, wird bei den anwendungsspezifischen Schaltungen versucht, den Entwurfsaufwand zu reduzieren. Dazu werden vorgefertigte Bauelemente und Schaltungsteile wie Legobausteine zu einer Schaltung zusammengesetzt. Ein solches Vorgehen erlaubt es darüber hinaus Ingenieuren aus den Anwendungsgebieten, die keine Halbleiterspezialisten sind, hochintegrierte Logikchips zu entwerfen.

ASIC

In diesem Zusammenhang wird zwischen dem vollkundenorientierten („full-custom design“) und dem teilkundenorientierten Entwurf („semi-custom design“) unterschieden. Bei Ersterem entwickelt der Kunde oder das Designteam eines Halbleiterunternehmens die Schaltung vollständig auf allen Abstraktionsebenen. Im teilkundenorientierten Entwurf dagegen beschränken sich die Entwickler auf die Logik und die Verdrahtung der Gatter untereinander. Dabei werden vorgefertigte Elemente aus von den Halbleiterherstellern definierten Bibliotheken zusammengefügt. Da es nicht einfach ist, Millionen von Gattern oder Milliarden von Transistoren richtig auf einem Siliziumkristall zu verbinden, kommt dem eigentlichen Entwurfsablauf eine entscheidende Bedeutung zu. Bedingt durch die hohen Kosten der Chipfertigung muss ein neu entwickelter IC schon bei der ersten Inbetriebnahme funktionieren. In der Industrie ist dieses Vorgehen unter dem Begriff „first time right“ bekannt. Da bereits die Herstellung eines Maskensatzes für die verschiedenen Belichtungsschritte enorm teuer ist, wird versucht, die Funktion der Schaltung mit rechnergestützten Methoden zu validieren.

Entwurfsstile

Abstraktionsebenen

Wurden Anfang der 1970er-Jahre die ersten Mikroprozessoren noch als Transistorschaltung entworfen, um aus dieser Schaltung direkt das Layout für die Herstellungsmasken abzuleiten, zeigte sich sehr bald, dass der Entwurf einer Schaltung in unabhängige Entwurfsschritte aufgeteilt werden muss. Wesentliches Merkmal ist die Einteilung des Entwurfs integrierter Schaltungen in unterschiedliche Abstraktionsebenen. Jede Abstraktionsebene beinhaltet einen Entwurfsschritt. Es wurde bereits zwischen dem Layout und der Schaltung unterschieden. Es wurde auch gezeigt, wie sich logische Gatter aus Transistoren zusammensetzen lassen und wie sich das Entwurfsmodell einer logischen Schaltung von einer Transistorschaltung unterscheidet. Damit sind schon drei verschiedene Entwurfsebenen genannt:

- Auf der untersten Ebene das Layout: Dieses legt die Geometrie der Schaltung fest und definiert die einzelnen Masken des Planarprozesses. Das Layout spezifiziert somit die elektronischen Eigenschaften bestimmter Teile der planaren Fläche eines Siliziumkristalls.
- Die Transistorebene: Transistoren stellen gegenüber dem Layout eine Abstraktion dar. Die Geometrie der Schalter wird nicht mehr gezeichnet, sondern ihre elektrischen Eigenschaften sind entscheidend. Das elektronische Verhalten der Bauelemente ist damit auf dieser Ebene von Bedeutung. Wie bereits gezeigt wurde, lässt sich die Bistabilität eines Flipflops ausschließlich elektrisch erklären.
- Die Ebene der logischen Gatter: Diese sind wiederum durch Transistoren aufgebaut. Im Entwurf kümmert man sich also um das logische Verhalten unter Berücksichtigung einfacher Zeitmodelle.

In den vorherigen Kapiteln wurden auf der Layoutebene und der Transistorebene Logikgatter zusammengebaut. Aus den Logikgattern wurden dann durch Rückkopplung Speicherelemente zusammengefügt. Um nun auch Rechnerarchitekturen aufzubauen zu können, müssen diese Grundelemente zu logischen Feldern integriert werden. Dabei steht in diesem Kapitel noch nicht die Rechnerarchitektur im Vordergrund, sondern Techniken, um viele Gatter auf einem Chip zu fertigen.

Logikfelder (Gate-Fabrics)

Bereits bei den ersten Mikroprozessoren war der Aufwand eine Schaltung zu entwerfen hoch. Bedingt durch die Skalierung der Halbleiterprozesse und die damit verbundene Verdopplung der Transistoranzahl eines Chips alle 18 Monate zeigte sich schnell, dass die Fertigungsmöglichkeiten nur ausgenutzt werden können, wenn der Entwurf standardisiert wird. Insbesondere die Erkenntnis, dass regelmäßige Geometrien im Layout eine höhere Packungsdichte der logischen Gatter ermöglichen, führte schnell zu standardisierten Logik- und Prozessfamilien. Von den Halbleiterunternehmen werden daher

4.5 · Logikfelder

unterschiedliche Familien standardisierter Logikschaltungen oder Logikfelder (Gate-Fabrics) angeboten.

Das Modell eines logischen Gatters ermöglicht es, digitale Schaltungen nur noch auf der Logikebene zu betrachten. Um sich die Abstraktion auch beim Schaltungsentwurf zunutze zu machen, werden Logikgatter im CMOS-Layout standardisiert. Eine solche Standardisierung ermöglicht es, Schaltungen vorzufertigen und deren Funktion erst später durch die Verdrahtung der Zellen festzulegen. Eine Schaltung mithilfe derartiger regulärer Zellen zu entwerfen, heißt Standardzellenentwurf und die entsprechenden Bausteine werden Standardzellen-ICs genannt. Der Standardzellenentwurf profitiert von der gleichmäßigen Struktur des Logikfeldes und ermöglicht es den Schaltungsentwicklern, sich auf den Entwurf der logischen Funktion zu konzentrieren. Letztendlich erlaubt der Standardzellenentwurf eine Arbeitsteilung zwischen Schaltung- und Layoutentwicklung mit genormten Gattern und dem Logikentwurf unter Verwendung eben dieser.

Grundsätzlich kann man zwischen Schaltungen, die vollständig nach den Wünschen der Kunden gefertigt werden, und Schaltungen, die zu einem gewissen Grad vorgefertigt sind, unterscheiden. Diese kundenspezifischen Chips sind maskenprogrammierbar, im Gegensatz zu Bausteinen, bei denen die Funktion durch Bitmuster in einem Speicher, also durch Software programmiert wird. Das Wort Programm stammt aus dem Französischen und bedeutet „schriftliche Bekanntmachung“, abgeleitet von griech. „prόgramma“ , „Vorgeschrriebenes“ oder „Vorschrift“. Ein Programm ist so etwas wie eine Handlungsanweisung. Die Maske für die Verdrahtung erlaubt also die Programmierung eines Logikbausteins. Bei solchen Schaltungen kann der fast gleich gefertigte Chip unterschiedliche Funktionen ausführen, je nachdem wie die letzten Verdrahtungsmasken gewählt werden. In der einfachsten Form eines derartigen maskenprogrammierbaren Bausteins wird ein vollständiger Mikrocontroller integriert, und lediglich der mit dem Prozessor hergestellte Speicher ROM wird maskenprogrammiert. So lassen sich unterschiedliche Einprozessorsysteme durch Programmierung eines universellen Chips herstellen.

Neben den maskenprogrammierbaren integrierten Schaltungen sind aber auch speicherprogrammierbare Logikschaltungen erhältlich. In diesen Bausteinen wird die logische Struktur der Schaltung, die Verdrahtung durch die Programmierung einzelner SRAM-Zellen hergestellt. Natürlich können solche im Feld zu programmierenden Schaltungen („field-programmable gate arrays“ (FPGAs)) auch in anderen Speicher technologien wie einem EPROM oder einem EEPROM gefertigt werden. Und um die Vielfalt zu komplettieren, kann so ein FPGA nach erfolgreichem Integrationstest für hohe

Standardzellen

Maskenprogrammierung

Speicherprogrammierung

Stückzahlen fest in Silizium als maskenprogrammierbarer IC gegossen werden. Im Folgenden werden die wichtigsten auf dem CMOS-Prozess basierenden Schaltungsfamilien vorgestellt.

4.5.1 Standardzellen

Im Layout von CMOS-Gattern fällt auf, dass egal welches Gatter implementiert wird, diese immer wieder die gleiche Struktur aufweisen. Es ist daher naheliegend, die Gatter zu normieren. Dabei werden die Höhe der Zelle, die Höhe und Lage der Wanne und die Höhe und Lage der Anschlüsse für die Versorgung- und Masseleitungen geometrisch immer gleich ausgelegt. Ein Logikfeld oder im Jargon „Gattergrab“ kann dann durch das Aneinanderfügen derartiger geometrischer Strukturen folgen. Ein solches standardisiertes Gatter wird als Gatter- oder Standardzelle bezeichnet. Die Idee ist, die Zellen immer gleich hoch, aber ggf. unterschiedlich breit zu bauen. Wenn alle Gatter die gleichen Abmessungen haben, lassen sie sich beliebig aneinanderreihen und bilden so eine reguläre Struktur logischer Gatter (logic Gate-Fabric). Da sich die Gatter höchstens in ihrer horizontalen Ausdehnung unterscheiden, können sie leicht durch Softwarewerkzeuge zur automatischen Platzierung angeordnet werden. Zusammen mit leistungsfähigen Programmen zum Erzeugen einer optimierten Verdrahtung lassen sich so komplexe integrierte Schaltungen ohne Halbleiterprozess- und Layoutkenntnisse entwerfen. Zu diesem Zweck erstellen Ingenieure eines Halbleiterherstellers das Layout für standardisierte Logikzellen und bestimmen mithilfe von Simulationsprogrammen die elektrischen Eigenschaften dieser Zellen. Die Gatter, ihre Funktion und die schaltungstechnischen Parameter und deren Auswirkung auf das logische Verhalten werden in Bibliotheken, die von den Schaltungsdesignern und den Softwareentwurfswerkzeugen genutzt werden können, abgelegt. Die Abb. 4.79 zeigt beispielhaft zwei optimierte Standardzellen. Bei diesen Zellen wird der vertikale Abstand zwischen den PMOS- und den NMOS-Transistoren vergleichsweise groß gewählt. Damit lassen sich die Gatter ohne viel Aufwand leicht verdrahten.

Eine Zelle bildet dabei einen kleinsten Entwurfsbaustein oder ein Makro. Erst ein Verdrahtungslayout legt dann später fest, welche Funktion mehrere dieser Makros einnehmen. Makros, die bereits fertig in Geometrie, also im Layout festgelegt sind, heißen Hardmakros. In einem Standardzellen-IC ist die Anzahl der verwendeten Hardmakros flexibel. Für den Logikentwurf werden noch Flipflops benötigt. Diese lassen sich aber wieder aus Gattern zusammenbauen. Mehrere zu einem

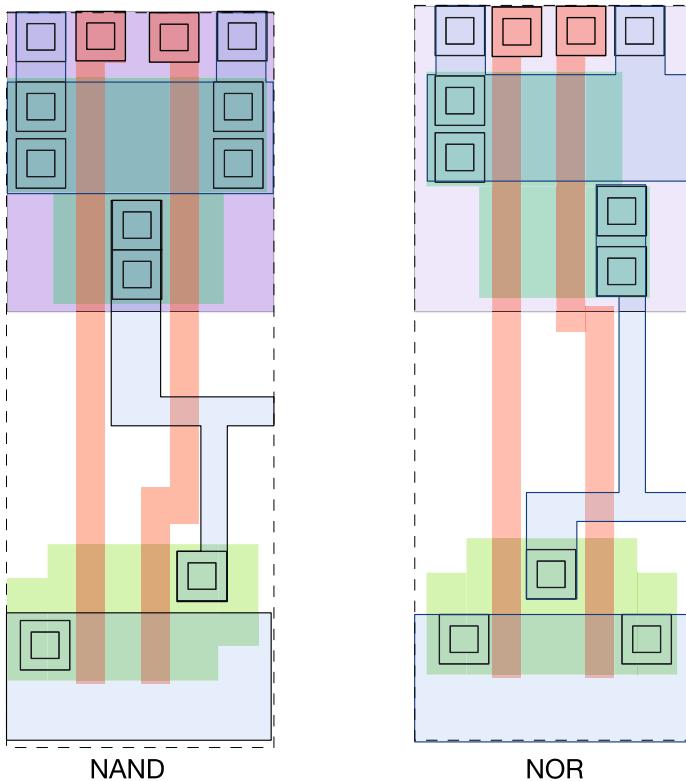


Abb. 4.79 Genormte Zellen für Standardzellenentwürfe

Flipflop verdrahtete Gatter bilden ebenfalls ein Hardmakro, wenn diese Zelle speziell für den Standardzellen-IC entworfen wurde.

Komplexere Schaltungen wie Addierer oder Schieberegister können entweder von den Ingenieuren des Halbleiterherstellers oder kundenspezifisch entworfen werden. Diese immer wieder verwendeten Bausteine können für einen Standardzellen-IC ebenfalls genormt sein und heißen Softmakros.

Fertige Hard- und Softmakros werden in einer Datenbank abgespeichert. Diese wird im Jargon der Halbleiterindustrie Bibliothek (engl. „library“) genannt. In der Schaltungsbibliothek sind alle Gatter mit ihren logischen und elektrischen Eigenschaften abgelegt. Ein Entwurfsingenieur oder heute eher Synthesewerkzeuge können aus den Makros komplexere Logikfunktionen zusammensetzen und die zusammengesetzte Schaltung bezüglich ihres Verhaltens untersuchen. Dies geschieht mit Logik- oder ereignisbasierten Simulatoren bis hin zu Programmen, die elektrische Netzwerke simulieren (Schaltungssimulatoren wie beispielsweise SPICE).

Die Abb. 4.80 zeigt einen aus Hard- und Softmakros zusammengesetzten Standardzellen-IC. Dabei wird der komplette IC in Entsprechung der Kundenwünsche gefertigt, d. h., alle Prozessschritte des Halbleiterprozesses werden kundenspezifisch ausgeführt. Für einen Standardzellen-IC wird lediglich der Entwurfsaufwand reduziert, da dieser durch automatische Werkzeuge auch von unerfahrenen Ingenieuren durchgeführt werden kann. Der Baustein besteht aus gleichmäßigen Reihen von Gattern, die kammförmig an die Spannungsversorgung und die Masseleitungen des Chips angeschlossen sind. Die Verdrahtung erfolgt in einer Ebene über den Zellen. In diesen Zeilen können komplexere Schaltungen zu Makros zusammengefügt werden. Durch die hohe Integrationsdichte findet man heute ebenfalls hochintegrierte Speicherbereiche auf den Bausteinen. Um die Fläche der Gate-Fabrics sind Treiber zum Ansteuern der ganz außen liegenden Pad-Felder angeordnet (I/O, für Input-Output). Die Pads selbst sind große Metallflächen, auf denen die Bonddrähte platziert werden können. Diese verbinden den IC mit den Anschlüssen des Gehäuses.

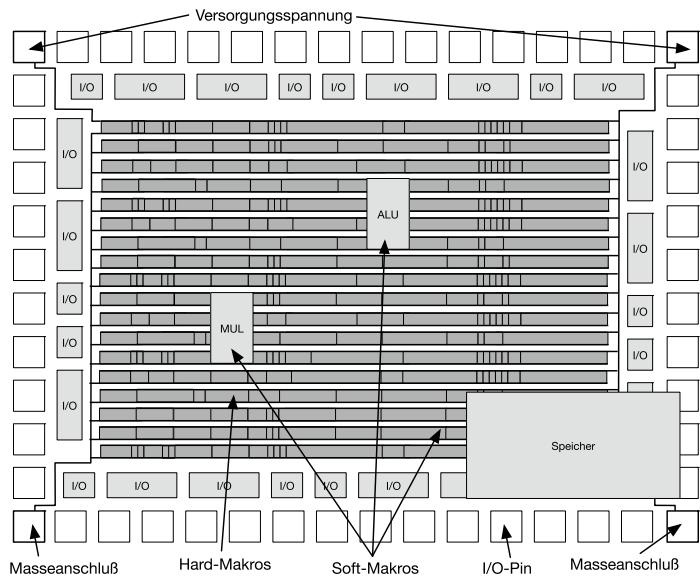


Abb. 4.80 Die Hard- und Softmakros eines Standardzellen-IC

4.5.2 Maskenprogrammierbare Logik

4.5.2.1 Gatterfelder (Gate-Arrays)

Die Technik der Standardzellen ermöglicht es, integrierte Schaltungen auch für Anwendungen, die in geringer Stückzahl produziert werden, einzusetzen. Bei Standardzellenentwürfen sind nur die Zellen für den Entwurf genormt. Ein kunden- oder anwendungsspezifischer IC wird jedoch für jeden Kunden vollständig gefertigt. Das bedeutet, dass alle Prozessschritte des Fertigungsverfahrens anwendungsspezifisch durchgeführt werden müssen. Der Fertigungsaufwand selbst kann nicht reduziert werden, allerdings ist es möglich, den kundenspezifischen Entwurfsaufwand zu verringern, indem die Funktion einer Standardzelle erst durch die Verdrahtung festgelegt wird. Dazu wird die normierte Zelle im Layout ein wenig angepasst. Bei solchen Gate-Array-Schaltungen können die ersten Prozessschritte für alle verschiedenen kundenspezifischen ICs gleich sein, und nur zur Verdrahtung werden unterschiedliche Masken benötigt. Damit lassen sich die Herstellungskosten einer anwendungsspezifischen Schaltung weiter reduzieren. Werden nur die Verdrahtung und Kontaktierung kunden-spezifisch ausgelegt, spricht man von Maskenprogrammierung (Abb. 4.81).

Bei dem NAND-Gatter werden die beiden PMOS-Transistoren parallel geschaltet, indem der Zwischenkontakt zwischen den beiden Transistoren mit dem Ausgang verbunden wird. Dies trifft ebenfalls für den rechten Kontakt der n-Diffusionen zu. Die Leitung für die Versorgungsspannung oder den Masseanschluss wird gesetzt und die jeweiligen Ein-

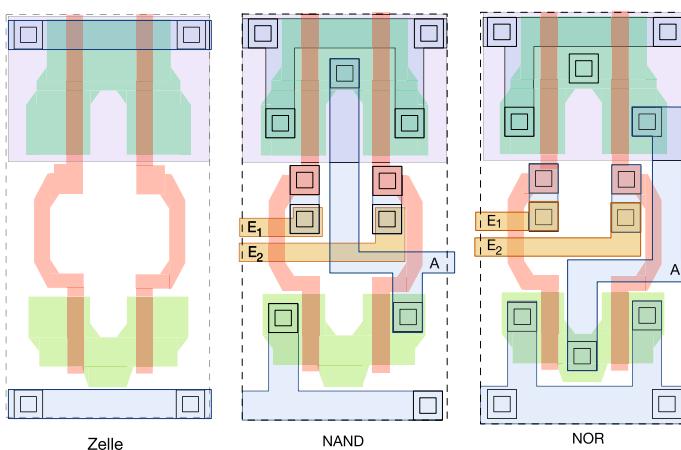


Abb. 4.81 Gate-Array-Zelle

Gate-Array**4****Sea of Gates**

gänge des Gatters werden über eine zweite Metallisierungsebene in die Mitte der Zelle geführt. Dort können sie an die Gates der Transistoren angeschlossen werden.

Beim NOR-Gatter ist die Verdrahtung komplementär. Nun wird der Platz für den Mittelkontakt zwischen den NMOS-Transistoren genutzt, und die PMOS-Transistoren sind in Reihe geschaltet.

Ein Gate-Array besteht aus vielen dieser maskenprogrammierbaren Zellen in Reihe. Jede dieser Reihen wird an die Versorgungsspannung und die Masse des Chips angeschlossen. Der dargestellte Baustein besitzt jeweils zwei Masse- und zwei Versorgungsspannungsanschlüsse. Dies ermöglicht einen höheren Stromfluss bei gleichzeitiger Abnahme der durch die Bonddrähte erzeugten Induktivität. Beide Maßnahmen zusammen erhöhen die möglichen Schaltgeschwindigkeiten. Direkt neben den Bond-Pads des Chips befinden sich spezielle I/O-Schaltungen (Input/Output). Dies sind Verstärker, um die große kapazitive Last der Bond-Pads und der daran angeschlossenen Leitungen treiben zu können. Zwischen den Reihen, in denen die einzelnen programmierbaren Logikzellen angeordnet sind, ist ausreichend Platz vorgesehen, um diese Zellen dann auch verdrahten zu können. Die Zellen und ihre Programmierung bilden die jeweiligen Logikgatter, und diese müssen anschließend noch verdrahtet werden, um die eigentliche logische Schaltung zu realisieren. In einem Halbleiterprozess mit nur zwei Metallisierungsebenen können diese Leitungen nicht über die Zellen geführt werden. Daher ist auch zwischen einzelnen Zellenblöcken in regelmäßigen Abständen ein Durchgang vorgesehen. Gate-Arrays sind nicht immer so strikt wie in der Abb. 4.82 aufgebaut. Da in vielen Schaltungen Speicher benötigt wird, ist es auch bei dieser Schaltungsfamilie möglich, Teile der Chipfläche für Speicherbereiche zu nutzen. Da der Speicher so wie die programmierbaren Logikzellen eine regelmäßige Layoutstruktur aufweist und maskenprogrammiert werden kann, lässt sich auf diese Art eine optimale anwendungsspezifische Schaltung erstellen.

In den ersten Halbleiterprozessen für Gate-Arrays stand zunächst nur eine Metallisierungsebene neben der Polysiliziumschicht zum Verdrahten der Gatter zur Verfügung. Mit zunehmender Verringerung der Strukturgröße wurden auch immer mehr Prozessschritte eingefügt, sodass inzwischen CMOS-Prozesse mit mehr als zwei Metallebenen zur Verfügung stehen. In dem Fall muss kein extra Raum mehr für die Verdrahtung der Gatter reserviert werden. Die durch Programmierung der Polysilizium- und ersten Metallebene entstandenen Logikgatter können jetzt mit Leitungen, die über diesen angeordnet sind, verdrahtet werden. Dadurch lässt sich die Anzahl möglicher logischer Gatter auf einer anwendungsspezifischen inte-

4.5 · Logikfelder

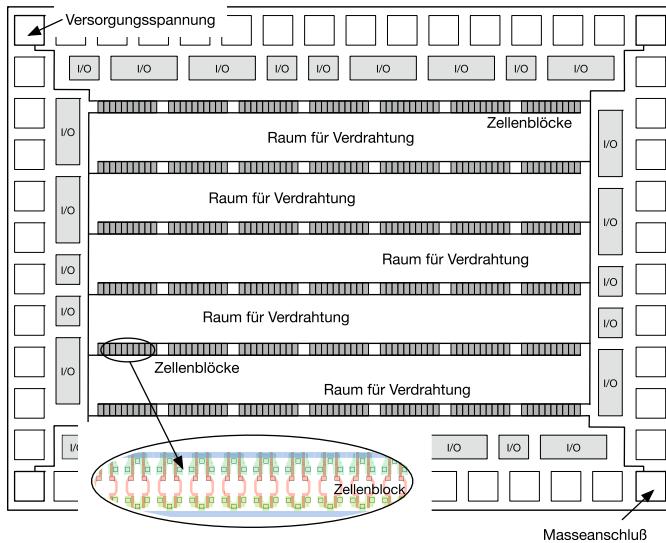


Abb. 4.82 Floorplan eines Gate-Arrays

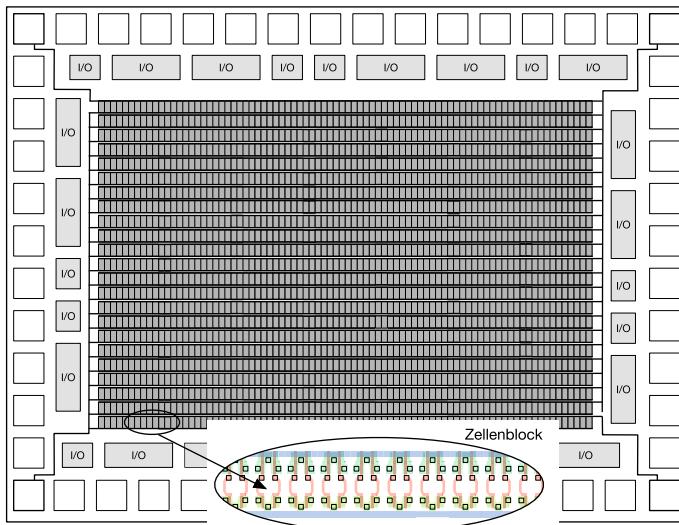


Abb. 4.83 Floorplan eines Sea of Gates

grierten Schaltung wesentlich erhöhen. Die **Abb. 4.83** zeigt eine solche „Sea of Gates“ genannte Technik.

4.5.2.2 Logisches Feld (PLA)

Schaltungen mit einem homogenen und regelmäßigen Layout benötigen weniger Fläche als welche mit einem irregulären.

4

PLA

Insbesondere Speicher lassen sich sehr effizient implementieren, da immer wieder die gleiche Zelle regulär wiederholt wird. Einzig die Ansteuerlogik führt bei diesen Bausteinen zu einer irregulären Struktur.

Insbesondere komplexe, aus Gattern aufgebaute Logikschaltungen führen zu einem irregulären Layout und benötigen daher viel Chipfläche. Komplexe logische Funktionen werden im Allgemeinen durch ihre disjunktive kanonische Normalform beschrieben. Eine solche Formel ist regulär, da jede Variable in jedem Minterm mindestens einmal vorkommt.

Eine reguläre Struktur für die Implementierung kanonischer Normalformen ist eine „programmierbares Logikfeld“ genannte Schaltungsstruktur (engl. „programmable logic array“, PLA). Ein solches „programmable logic array“ wird durch Setzen von Kontaktlöchern maskenprogrammiert. Ein PLA besitzt für jede Eingangsvariable einen Inverter. Für jeden möglichen Minterm kann ein Verstärker angenommen werden. Jeweils eine Leitung für jede Variable und ihr jeweiliges Komplement werden senkrecht über eine Schaltmatrix für die jeweiligen Minterme gezogen. Die Verstärker für die Minterme werden dazu waagerecht übereinander angeordnet. Min- oder Maxtermleitungen werden dann mit den Eingangseingängen an der jeweiligen Stelle verknüpft, sodass sich am Eingang eine Matrix von UND-Schaltungen ergibt. Dadurch wird jeder Minterm einer DKNF in der UND-Matrix berücksichtigt. Die Leitungen der Minterme wiederum werden waagerecht über eine zweite Schaltmatrix gezogen. Senkrecht auf dieser sind die Ausgänge des PLA angeordnet. Die Anzahl dieser und die Zahl der Mintermleitungen bestimmen, wie viele unterschiedliche Schaltfunktionen von einem PLA implementiert werden können. Die Ausgänge werden mit den Mintermen über ODER-Schaltungen verknüpft. Es entsteht am Ausgang eine Matrix von ODER-Gattern, die die disjunktiven Verknüpfungen der DKNF implementieren. In der Abb. 4.84 ist ein Beispiel gezeigt. Dabei sind UND-Verknüpfungen durch Kreuze und ODER-Verknüpfungen durch Kreise markiert. Es sei darauf hingewiesen, dass es sich bei der logischen Darstellung um ein Modell handelt. Im Layout von MOS-Schaltungen werden NAND-NAND- oder NOR-NOR-PLAs konstruiert, da sich diese mithilfe eines „wirred OR“ einfach realisieren lassen.

Ein PLA lässt sich in MOS-Technik nicht wie gewünscht in kompakter Form mit einer UND- und einer ODER-Matrix realisieren. Allerdings kann ein NOR einfach implementiert werden. Durch Anwenden des Satzes von De Morgan kann die DKNF in eine NOR- oder NAND-Variante überführt werden. In NMOS- oder auch in PMOS-Technik ist ein PLA dann einfach zu realisieren. Mehrere parallel geschaltete Schalter

Dynamisches CMOS-PLA

4.5 · Logikfelder

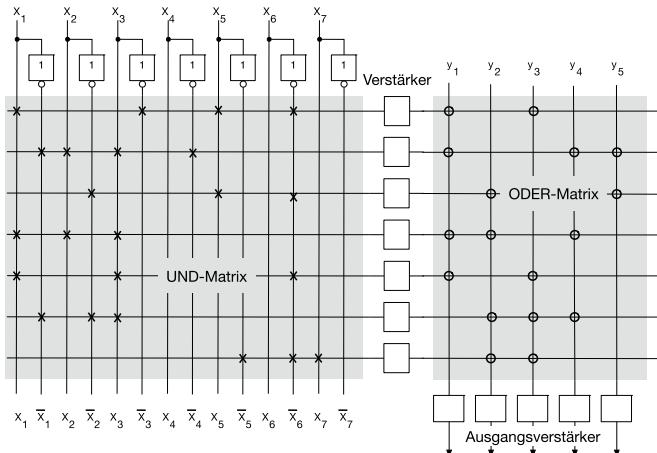


Abb. 4.84 Modell zur Programmierung eines PLA

bilden die ODER-Schaltung, und ein als Lastwiderstand verschalteter Transistor ermöglicht die Negation oder das Laden der Lastkapazität. Dieses Prinzip kann auch in CMOS angewendet werden. Nur betreibt man den zusätzlichen Transistor nicht als Widerstand, sondern als Aufladeschalter. Dazu wird der Schalter mit einem Takt angesteuert. Die Ausgänge des PLA sind dann aber nur kurz nach dem Beenden der Aufladephase gültig. Diese Technik heißt dynamisches CMOS. Zuerst aktiviert der Takt die PMOS-Transistoren. Die Mintermleitungen oder die Ausgangsleitungen werden aufgeladen, dann, nachdem der Takt den PMOS-Transistor ausgeschaltet hat, schalten die NMOS-Transistoren die Schaltfunktion. Durchgeschaltete NMOS-Transistoren werden auf Masse gezogen und stellen eine 0 dar. Wird kein NMOS-Transistor eines Minterms angesteuert, bleibt die Leitung auf der zuvor geladenen Versorgungsspannung und repräsentiert eine 1 (Abb. 4.85).

Dynamische CMOS-Schaltungen erfordern eine zeitsynchrone Taktsteuerung. Mit einem statischen CMOS-PLA kann auf eine Taktansteuerung verzichtet werden. Diese kann mit einem schaltungstechnischen Trick implementiert werden: Die NMOS-Transistoren werden als ODER-Schaltung im Feld angeordnet. Zusammen mit einem PMOS-Transistor am Ausgang der Matrix bilden sie eine NOR-Schaltung. Die komplementären Eingänge sind ebenfalls Eingänge eines auf gleiche Art gebauten NAND-Gatters. Mehrere PMOS-Transistoren bilden die ODER-Matrix und zusammen mit einem NMOS-Transistor am Ausgang eine UND-Matrix. Schaltet man die

Statisches CMOS-PLA

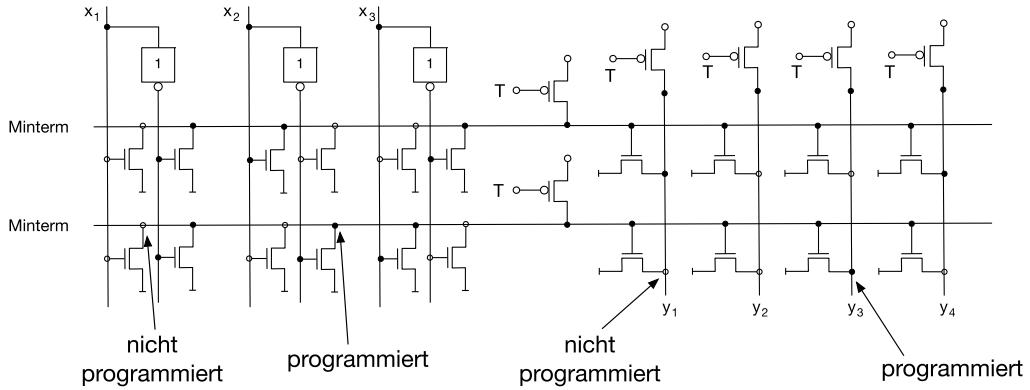


Abb. 4.85 Schaltung eines dynamischen CMOS-PLA

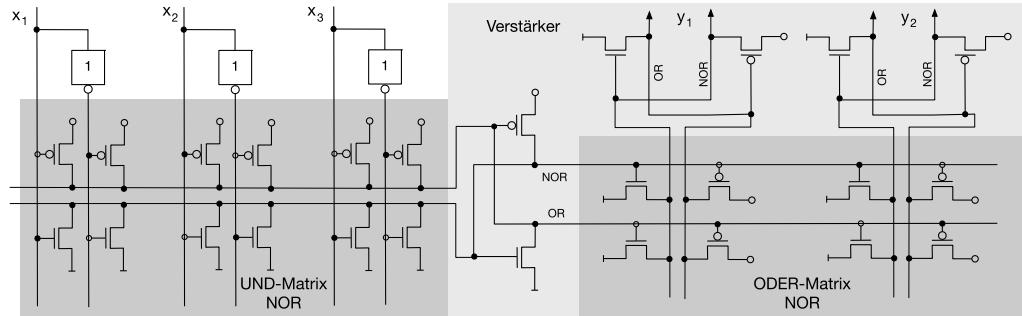


Abb. 4.86 Schaltung eines statischen CMOS-PLA

beiden Strukturen über Kreuz, erhält man ein statisches CMOS-ODER- oder NOR-PLA (Abb. 4.86).

Layout eines CMOS-PLA

Im Layout zeigt sich die reguläre Struktur des CMOS-PLA. Die Abb. 4.87 stellt ein PLA, mit dem sich 5 Eingänge auf 8 Minterme schalten lassen, dar. Die ODER-PLA-Zelle wird in dem Beispiel als NOR-Zelle betrieben, da sich eine UND-Matrix in CMOS nicht so einfach realisieren lässt. Die 4 Ausgänge können dann ebenfalls über eine NOR-Matrix verschaltet werden. In dem vorliegenden Layout wurden zwei unterschiedliche Arten von Zellen verwendet. Obwohl beide mit den Mintermverstärkern eine ODER-Schaltung bilden, wurde die linke Seite des PLA in Hinblick auf die Breite des entstehenden Schaltfeldes minimiert, während die rechte Seite auf die Höhe optimiert wurde. Mit jeweils nur einem Layout einer PLA-Zelle lässt sich keine reguläre Matrixstruktur implementieren. In der Abb. 4.87 werden daher alle Leitungen verwendet. In einem „Full-custom-Layout“ benötigt man aber nur die wirklich genutzten Transistoren und kann dann die nicht genutzten weglassen und ggf. das Layout noch kompaktieren. Ein Vor-

4.5 · Logikfelder

teil von PLAs ist, dass sich logische Funktionen direkt im Layout der integrierten Schaltung realisieren lassen. Bei einfachen Schaltungen, wie sie in der Frühzeit der Mikroelektronik realisiert wurden, war es daher nicht nötig, zunächst einen Logikschaltplan und dann eine Transistorschaltung zu entwerfen. Vielmehr konnten die logischen Funktionen direkt in einem PLA programmiert werden. Programmierung meint also in dem Fall das Setzen und Weglassen eines Transistors im Layout.

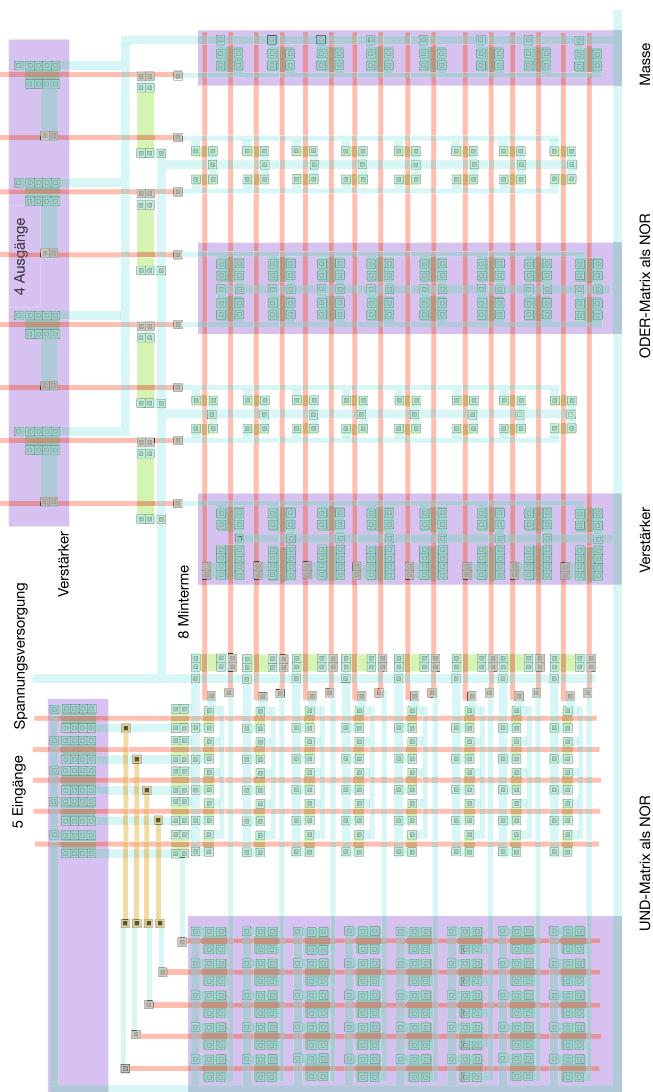


Abb. 4.87 Layout eines statischen CMOS-PLA

► Beispiel 4.5.1 – PLA-Addierer

Die minimierten Schaltfunktionen des Addierers werden zunächst in eine NOR-Schaltfunktion umgewandelt. Diesmal wurde eine andere, oft anzutreffende Schaltskizze verwendet. Bei dieser Variante werden statt der Verstärker die entsprechenden Logikfunktionen am Anfang der Mintermeleitungen eingezeichnet. Das Gleiche gilt für die Ausgänge. Ein Kreuz in der Zeichnung markiert dann immer die Nutzung des entsprechenden Gattereingangs. In der Abb. 4.87 wurde ein optimiertes PLA gewählt, d. h., es wurden nur Leitungen und Verbindungen implementiert, die auch verwendet werden. ◀

► Beispiel 4.5.2 – 1-aus-n-Decodierer

Eine häufig in der Rechnerarchitektur verwendete Schaltung ist der 1-aus-n-Decodierer. Dabei werden von einer n -steligen logischen Zahl 2^n Leitungen eindeutig angesprochen. Jede n -stellige Zahl wählt also genau eine Leitung aus. Für eine mit 3 Stellen codierte Zahl ist daher die Wertetabelle aus Tab. 4.1 zu implementieren. Die Abb. 4.88 zeigt eine dazugehörige Decodierschaltung auf der Grundlage eines PLA. Allerdings wird in dem Fall die ODER-Matrix nicht benötigt. ◀

PLAs lassen sich im Schaltungsentwurf beliebig einsetzen. Durch die regelmäßige Struktur eines PLA ist es möglich, eine Schaltfunktion direkt auf der Layoutebene zu implementieren. Da die Programmierung durch das Setzen von Kontaktlöchern erfolgt und alle Transistoren bereits vorgefertigt sind, kann auf einen Entwurfsschritt wie die Darstellung der Logik mithilfe von Gattern verzichtet werden. Die ersten Werkzeuge zur automatischen Synthese von Logikschaltungen („si-

Tab. 4.1 Wertetabelle 1-aus-n-Decodierer

Wert	Codierte Zahl
00000001	000
00000001	000
00000010	001
00000100	010
00001000	011
00010000	100
00100000	101
01000000	110
10000000	111

4.5 · Logikfelder

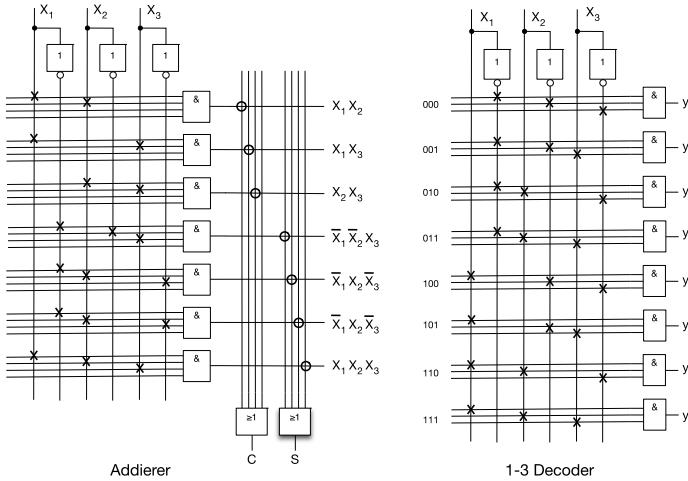


Abb. 4.88 Addierer- und Decoder-Schaltfunktion mit einem PLA

licon compiler“) beruhten auf dieser Eigenschaft von PLAs. Es ließen sich damit rechnergestützt ein Layout und somit die Fertigungsdaten einer Schaltung erzeugen. Nachteil von PLAs ist die Leitungslänge bei großen Schaltfunktionen. Durch den Einsatz von PLAs wird auf die Verstärkereigenschaft der Gatter verzichtet. Damit sind sehr große PLAs unwirtschaftlich, da sie einen hohen Flächenverbrauch und langsame Schaltzeiten aufweisen. Da sich Standardzellenentwürfe gut mit modernen computergestützten Synthese-, Verdrahtungs- und Layoutwerkzeugen erzeugen lassen, werden sie mittlerweile direkt aus Hochsprachen generiert und als Gatterschaltung realisiert. PLAs sind allerdings ein schönes Beispiel dafür, wie komplexe Schaltfunktionen mithilfe eines Maskenlayouts auf der Ebene der Verdrahtung einfach programmiert werden können. Sie sind somit der Prototyp einer direkten Maskenprogrammierung.

4.5.3 Speicherprogrammierbare Logik

Die einfachste Methode, logische Funktionen zu programmieren, ist es, diese durch einen Speicher zu realisieren. Ein Speicher besteht aus einer Speichermatrix und einem Auswahldecoder. Der Auswahldecoder wählt exakt eine Zeilenleitung der Speichermatrix bei Anlegen eines bestimmten Bitmusters aus. Das Bitmuster stellt bei Verwendung des Speichers die Adresse, unter der die gewünschten Daten zu finden sind, dar. Wird mithilfe eines Speichers eine Schaltfunktion realisiert, sind die Adressen als Minterme zu interpretieren und die Ein-

träge in der Speichermatrix als logisches Ergebnis des Minterms. Immer wenn eine Minterm 1 ist, wird in die entsprechende Zeile des Speichers eine 1 eingetragen, ansonsten eine 0. Damit bildet eine Speicherreihe eine disjunktive Normalform und zusammen mit dem Adressdecoder eine Wahrheitstabelle, in der die Adressleitungen die Eingänge repräsentieren und die Dateneinträge die jeweilige Schaltfunktion. Mit einer 16×8 -Schaltmatrix können so bis zu 8 disjunktive kanonische Normalformen mit 4 Variablen und mit 16 Mintermen realisiert werden.

► Beispiel 4.5.3 – Realisierung einer Schaltfunktion mit Speicher

Gegeben sei die Schaltfunktion $y_0(x_0, x_1, x_2) = \overline{x_0}x_1\overline{x_2} + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2}$. Die Minterme dienen direkt als Eingang eines Adressdecoders. In jeder Zeile wird dann im Speicher eine 1 oder eine 0 abgelegt. In diesem Fall steht jeweils in Zeile 3, Zeile 6 und Zeile 7 eine 1 und in allen anderen Zeilen eine 0. Wird ein entsprechendes Mintermmuster an den Adressdecoder des Speichers angelegt, entspricht der Ausgang y_0 dem korrekten Ergebnis der Schaltfunktion. ◀

Aus Abertausenden von logischen Funktionen bestehende Rechenmaschinen sind ein Albtraum, wenn ein Gatter nicht korrekt aufgebaut oder vielmehr die Verbindungen untereinander falsch gesetzt wurden. Daher war man schon früh bemüht, die Hardware eines Rechners so allgemein und einfach wie möglich auszuführen und die eigentliche Anwendung der Maschine zu programmieren. Diese Methode wird Mikroprogrammierung genannt. Insbesondere der in der Vergangenheit teure Speicher für wahlfreien Zugriff führte zum Aufbau von Logik mithilfe von Nur-lese-Speichern. Mit dieser Mikroprogrammierung genannten Technik ist es möglich, komplexe Befehlssätze außerhalb der Elektronik zu realisieren. Durch Austausch von ROM-Speicher konnte der Befehlssatz einer Maschine sogar im Feld oder auch im Betrieb angepasst oder erweitert werden. Der Bedarf nach günstiger und frei gestaltbarer Logik, um aufwendige Logikimplementierungen (Gattergräber) durch integrierte Schaltungen zu ersetzen, führte zu der speicherprogrammierbaren Logik, also Schaltungen, bei denen die Funktion oder die Verdrahtung durch eine änderbare binäre Programmierung festgelegt wird. Diese Programmierung wird entweder in SRAM-Zellen gespeichert oder durch Durchbrennen von Kontakten realisiert. Deshalb wird zwischen mehrmalig programmierbaren und einmalig programmierbaren Bausteinen unterschieden. Derartige Schaltungen eignen sich gut für Anwendungen mit geringer Stückzahl oder zum Erstellen von Prototypen, um neue Konzepte zu erproben (Abb. 4.89).

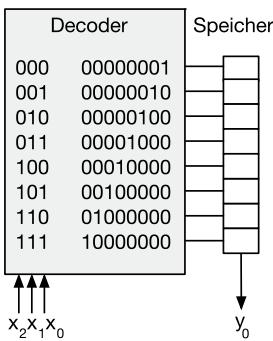


Abb. 4.89 Realisierung einer Schaltfunktion mit einem Speicher

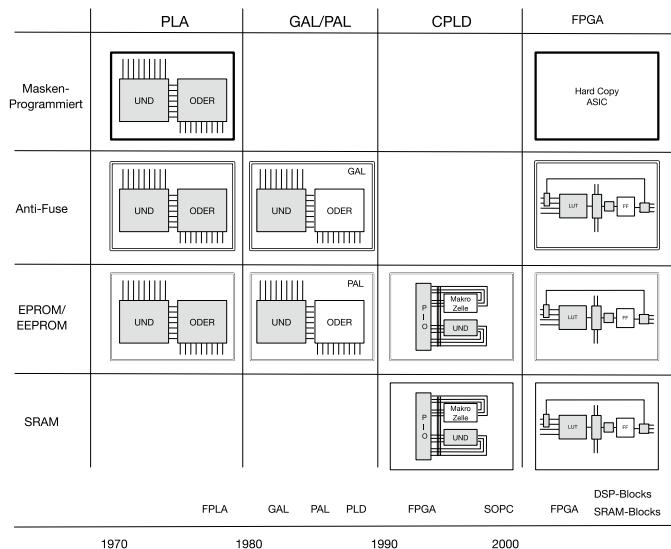


Abb. 4.90 Entwicklung speicherprogrammierbarer Logik

Die Abb. 4.90 gibt eine Übersicht über die Entwicklung programmierbarer Bausteine. Die zunächst einmal rein maskenprogrammierbaren Strukturen einer PLA wurden direkt durch speicherprogrammierbare Schaltungen ersetzt. Dazu mussten lediglich die Verbindungstransistoren durch Speicherzellen ausgetauscht werden. Bei einigen frühen Logikbausteinen war aus Kostengründen nur die UND-Matrix programmierbar. Das änderte sich jedoch schnell mit der zunehmenden Transistordichte. Neben Bausteinen, deren Programmierung über SRAM-Zellen oder EEPROM-Zellen erfolgt, gibt es nur einmal programmierbare Schaltungen, bei denen mit hohen Spannungen Verbindungsstrukturen auf dem Chip durchgebrannt werden. Diese Anti-Fuse-Chips sind dann fest verdrahtet. Dieser Nachteil wird allerdings durch eine höhere Integrationsdichte und somit mehr logischen Gattern und einer höheren Zuverlässigkeit ausgeglichen. Inzwischen können mit FPGAs nicht nur einfache Schaltfunktionen auf dem programmierbaren Chip implementiert werden, sondern komplexe konfigurierbare Mikrocomputersysteme („system on a programmable chip“, SOPC).

4.5.3.1 PLAs aus Speichern

Mithilfe zweier handelsüblicher Speicherbausteine und einem Codierer lässt sich ebenfalls eine PLA herstellen. Dazu fungieren der erste Speicherbaustein als UND-Matrix und der zweite als ODER-Matrix. Der Codierer wird benötigt, um das vor-

Field-programmable logic array

Programmable array logic

PLD

liegende Ergebnis der ersten Schaltfunktion wieder zu codieren, sodass der zweite Speicher Codierte Daten am Eingang bekommt. Bei einem mit integrierten Speichern konstruierten PLA ist dies nicht notwendig: Lässt sich die Schaltung komplett selbst bauen, kann auf den Codierer und den Decodierer des zweiten Speichers verzichtet werden. Damit lässt sich aus zwei Speichermatrizen und einem Decodierer ein feldprogrammierbares Logikfeld aufbauen.

Die Verbindungen in PLAs können statt durch Festlegen einer gewünschten Verdrahtung bei der Herstellung ebenfalls mit durchbrennbaren Brücken in der Schaltung realisiert werden. Derartige Schaltungen können dann vor der eigentlichen Anwendung direkt im Feld programmiert werden. Daher werden sie „field-programmable logic array“ (FPLA) genannt. Das FPLA der früheren Halbleiterfirma Signetics, der Baustein 82S100, bestand aus einer $16 \cdot 48 \cdot 8$ -Schaltmatrix. In der Schaltmatrix dieses Bausteins wurden das UND-Feld mithilfe von Dioden und das ODER-Feld mit Transistoren implementiert.

Wird die Programmierfähigkeit der PLA nur auf die UND-Matrix eingeschränkt, werden die entstehenden Bausteine „programmable array logic“ (PAL) oder „generic array logic“ (GAL) genannt. GALs wurden von der Firma Monolithic Memories Ende der 1970er-Jahre entwickelt. Sie konnten mithilfe der Anti-Fuse-Technik nur einmal programmiert werden. PALs dagegen sind Bausteine, die sich mehrmals programmieren lassen. Dazu wird die EEPROM-Technik genutzt. Ebenfalls erhältlich waren Bausteine, deren Programmierung durch UV-Licht gelöscht werden konnte. PALs wurden in den 1980er-Jahren von der Firma Lattice Semiconductor hergestellt und vertrieben.

„Generic array logics“ (GALs) und „programmable array logics“ (PALs) wurden von „programmable logic devices“ (PLDs) verdrängt. Die Schaltfunktionen in PLDs werden durch frei programmierbare PLAs entweder in Anti-Fuse- oder in EEPROM-Technik realisiert. Gegenüber den ersten PLA-Bausteinen besitzen PLDs jedoch noch programmierbare Rückkopplungen und diverse Ein- und Ausgabeblocks. Ein Beispiel für ein PLD ist die MAX5000-Serie der Halbleiterfirma Altera, die inzwischen von Intel übernommen wurde. MAX5000-Bausteine bestehen aus 32 bis 192 logischen Feldern. Diese PLA-Strukturen werden in Logic-Array-Blöcken gruppiert (LAB). In der Regel befinden sich in einem LAB zwei verschiedene Gruppen von logischen Schaltungen: zum einen ein klassisches PLA und zum anderen ein Feld spezieller Makrozellen. In einer Makrozelle wird das ODER-Feld des PLA durch ein Flipflop ersetzt. Die Mintermleitungen des UND-Feldes steuern dabei unterschiedliche Funktionen des

Flipflops wie den Dateneingang, aber auch Reset-, Set- und Clock-Leitungen an. Der Ausgang des Flipflops wird wieder auf die UND-Matrix zurückgeführt, sodass ebenfalls Schaltwerke implementiert werden können. Das PLA ist mit einer lokalen Verbindungsstruktur sowohl an seinen Ein- als auch den Ausgängen gekoppelt. Die Makrozelle ist darüber hinaus noch an einer weiteren programmierbaren Verbindungsstruktur („programmable interconnect array“ [PIA]) angeschlossen. Jede Logikzelle, bestehend aus PLA und Makrozelle, ist über E/A-Kontrollblöcke (oder engl. „I/O“) mit einem Pin des Chips verbunden. Die PLA-Strukturen werden von außen über 8 bis 20 Eingabeleitungen angesprochen. Ein Baustein wie der MAX5000 wird auch „complex programmable logic device“ (CPLD) oder „electrically programmable logic device“ (EPLD) genannt.

CLPD

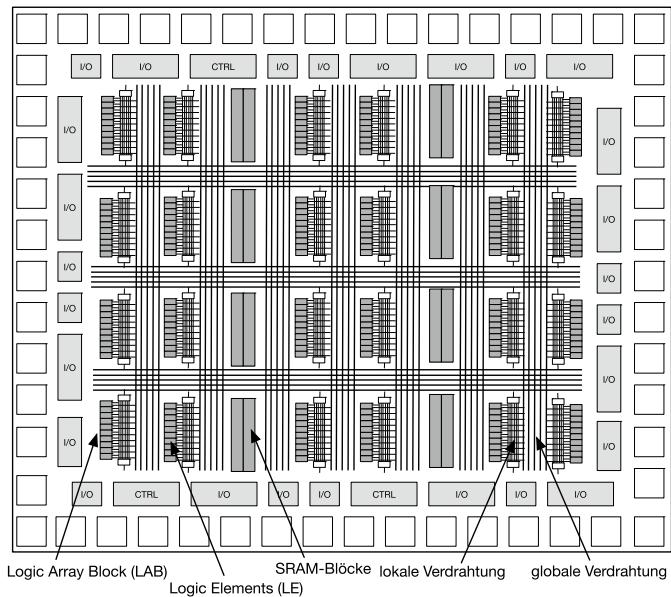
4.5.3.2 Speicherprogrammierbare Gatterfelder (FPGA)

Die zunehmende Integrationsdichte hat Ende der 1980er-Jahre zu einer neuen Form programmierbarer Logikbausteine geführt: den FPGAs. Diese Bausteine sind eine Mischung aus Gate-Array oder Sea-of-Gate-ASICs mit PLDs. Statt aber die Gatterfunktion durch die Verdrahtung von CMOS-Strukturen zu realisieren, wird eine logische Funktion mit einem Speicher programmiert. Dieser kleine Speicher heißt in einem FPGA Look-up-Table (LUT). Damit sind einzelne Gatterfunktionen speicherprogrammierbar. Die LUT bilden zusammen mit Flipflops Zellen, und diese Zellen sind wie in einem Gate-Array in Blöcken angeordnet. Mithilfe einer frei programmierbaren Verdrahtungsstruktur zwischen den Blöcken lassen sich diese freiprogrammierbar verbinden. Verbindungsstrukturen sind dabei hierarchisch aufgebaut, zum einen durch kurze lokale und zum anderen lange globale Verbindungen. Diese Trennung dient der effizienten Nutzung der Chipfläche für die Verdrahtung (Abb. 4.91).

FPGA

Look-up-Table

Ein Logic-Element (LE) oder ein Slice besteht jeweils aus einer Look-up-Table (LUT) und einem Flipflop. Mehrere LEs bilden einen Logic-Array-Block (LAB). Zellen innerhalb der LABs können über kurze, lokale Leitungen verbunden werden. Das Flipflop kann über eine programmierbare Schaltung mit verschiedenen Taktsignalen, Set- und Reset-Signalen angesprochen werden. Neben der LUT und dem Flipflop sind noch eine Übertragslogik sowie einfache Schaltungen zum Synchronisieren der Daten implementiert. Ein- und Ausgänge können sowohl an lokale und globale Verdrahtungskanäle angeschlossen werden als auch in der Zelle vom Ausgang auf den Eingang und vom Eingang direkt auf den Ausgang geführt werden. In-



■ Abb. 4.91 Schematische Darstellung eines FPGA

teressant ist, dass die LUT konfiguriert werden kann und es so möglich ist, die Zelle in unterschiedlichen Zuständen zu betreiben. Einige Hersteller von FPGAs unterscheiden dabei zwischen zwei verschiedenen Modi in denen eine Zelle betrieben werden kann: einem Logik- und einem arithmetischen Modus. Im Logikbetrieb besitzt die LUT 4 Eingänge. Damit können dann 16 Schaltfunktionen implementiert werden. Alle 16 Werte werden auf eine Leitung zum Flipflop geführt. Im arithmetischen Modus wird die große LUT in 4 kleine LUTs mit jeweils 2 Eingängen unterteilt. Dabei können 2 LUTs auf das Flipflop geschaltet werden, und die 2 anderen Tabellen werden mit der Übertragslogik verbunden. Zu bemerken ist noch das XOR-Gatter am Eingang der Zelle. Dieses lässt sich nutzen, um Teile von Addierschaltungen ohne Verwendung der LUTs zu realisieren. Es gibt jedoch auch Hersteller, die eine Zelle (Slices) aus jeweils 4 LUTs und 4 Flipflops aufbauen. Die LUTs können dann über ein mit Multiplexern konfigurierbares Schalt- netz mit den 4 Flipflops der Zelle verbunden werden. Auch die Slices verfügen über Rückkopplungen.

Exemplarisch sollen 2 LUTs mit 2 Eingängen betrachtet werden. Beide werden auf einen Adressdecoder geführt. Wie bei der Realisierung einer Logik mit einem Speicher stellen die Adressen die jeweiligen Minterme der zu realisierenden Schaltfunktion dar. Über ein UND-Gatter kann noch eine separa-

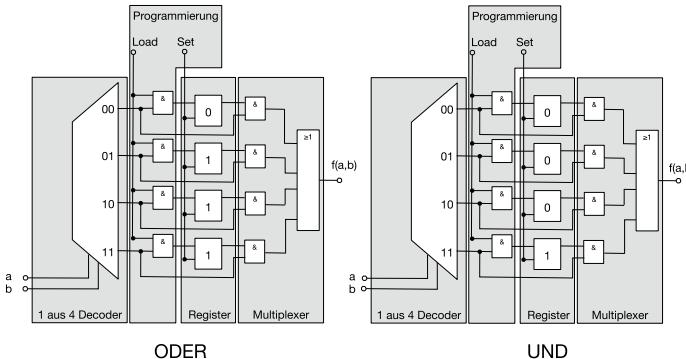


Abb. 4.92 Programmierung der LUT mit einem ODER und einem UND

te Ladelogik an die Speicherzellen angeschlossen werden, um die Flipflops der LUT mit einem Wert zu initialisieren. In der Abb. 4.92 sind exemplarisch eine UND- und eine ODER-Funktion realisiert. Über den Adressdecodierer wird der gewünschte Minterm auf ein ODER-Gatter geschaltet. Zusammen mit den UND-Gattern zur Auswahl der Zeile bildet das ODER-Gatter einen Multiplexer.

Moderne FPGAs können über 100.000 derartiger Zellen aufweisen, sodass es inzwischen möglich ist, komplett Mikrocomputer auf einem FPGA zu realisieren. Damit ist ein speicherprogrammierbares FPGA ausgezeichnet zum Ausprobieren von Schaltungsideen geeignet. FPGAs sind inzwischen günstig zu bekommen und spielen daher im Laboralltag eine große Rolle. Insbesondere anwendungsspezifische Hardware-/Softwaresysteme lassen sich mit FPGAs günstig prototypisch realisieren. Viele bekannte Halbleiterhersteller benutzen FPGAs zum Erproben der zukünftigen „Full-custom-Chips“. Dazu sind Systeme erhältlich, die selbst in einem Schaltschrank (Rack) wieder mehrere FPGAs enthalten. Die Verbindungen zwischen den einzelnen FPGA-Bausteinen lassen sich ebenfalls programmieren, sodass auch große Schaltungen mit FPGAs emuliert werden können.

Übungsaufgaben

Aufgabe 4.1 In Abb. 4.93 ist eine digitale Logik als Transistorschaltung in NMOS-Technologie gegeben. In dieser Schaltung bezeichnen die digitalen Signale x_1 bis x_3 die Eingänge und $f(x_1, x_2, x_3)$ repräsentiert den Ausgang.

- Bestimmen Sie die boolesche Funktion bestehend aus den Variablen x_1 , x_2 und x_3 , die die Transistorschaltung in Abb. 4.93 realisiert.
- Stellen Sie die Wertetabelle der Funktion auf.

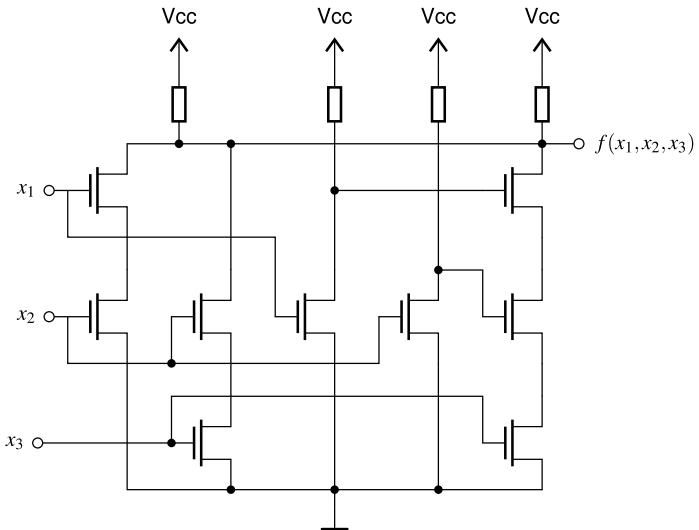


Abb. 4.93 NMOS-Transistorschaltung

- c) Wandeln Sie die Schaltung aus Abb. 4.93 in eine PMOS-Transistorschaltung um.
- d) Worin besteht der Vorteil von CMOS- im Gegensatz zu NMOS- und PMOS-Gattern?

Aufgabe 4.2 Gegeben ist die folgende Funktion:

$$f(x_3, x_2, x_1, x_0) = x_0 + \overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot \overline{x_3}$$

- a) Stellen Sie f nur mit NAND- und NOR-Gattern dar. Die Negationen der eingehenden Variablen stehen Ihnen zur Verfügung.
- b) Setzen Sie f in CMOS um. Benutzen Sie dafür nachfolgende Schaltungsschablone. Dazu stehen Ihnen 6 Schaltungsgruppen zur Verfügung (Abb. 4.94), mit denen jeweils ein NAND oder NOR umgesetzt werden kann. Tragen Sie dazu in die freien grauen Felder jeweils entweder ein P, N, L oder O ein, und ergänzen Sie die fehlenden Verbindungen. Dabei gilt die Bedeutung wie in untenstehender Tabelle angeben. Verbunden werden die Bausteine über die weißen Anschlussstellen. Negationen der Eingabeparameter werden in dieser Implementierung direkt zur Verfügung gestellt.
- c) Beschreiben Sie in eigenen Worten, was der Vorteil von CMOS gegenüber NMOS und PMOS ist.

4.5 · Logikfelder

P	PMOS	Schaltet durch wenn am Gate eine 0 anliegt.
N	NMOS	Schaltet durch wenn am Gate eine 1 anliegt.
L	Leitung	Überbrückt das anliegende Signal. (Der obere Eingang wird direkt auf den unteren Ausgang geschaltet.)
O	Offen	Unbenutzter Bereich.

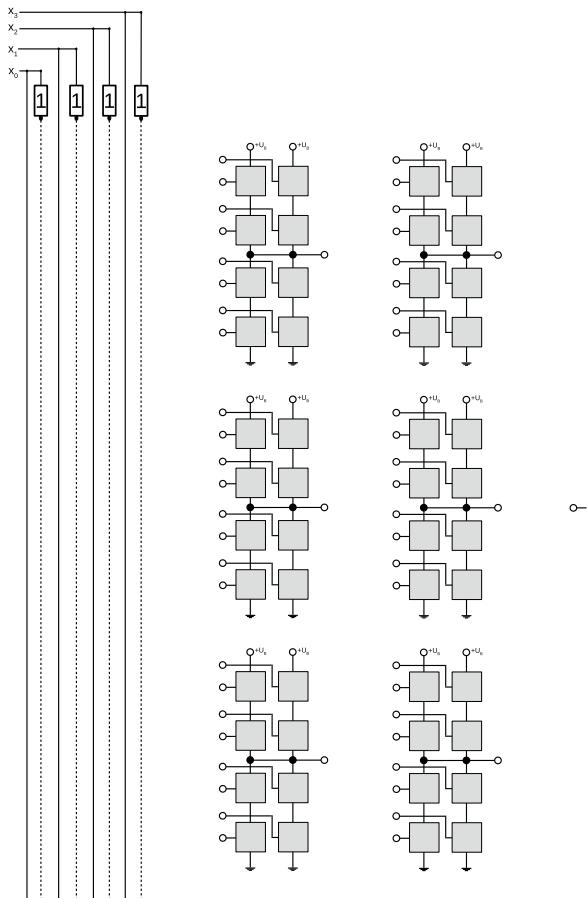


Abb. 4.94 Template für CMOS-Schaltung

Aufgabe 4.3 Gegeben ist ein RS-Flipflop, welches über die folgende Tabelle definiert ist:

R	S	Q_t	\bar{Q}_t
0	0	Q_{t-1}	\bar{Q}_{t-1}
0	1	1	0
1	0	0	1
1	1	X	X

Beschreiben Sie die jeweiligen Zustände der Wahrheitstabelle in Abhängigkeit der Eingänge, und erklären Sie (schriftlich oder grafisch), was der Zustand mit $S = R = 1$ für das RS-Flipflop bedeutet.

Aufgabe 4.4 Zeichnen Sie ein D-Latch aus Gattern ihrer Wahl.

Weiterführende Literatur

- [And78] W.v. Münch Andreas Schlachetzki. *Integrierte Schaltungen*. Vieweg+ Teubner Verlag, 1978. ISBN: 9783662485538.
- [Sch90] Andreas Schlachetzki. *Halbleiter-Elektronik: Grundlagen und moderne Entwicklungen*. Vieweg+Teubner Verlag, 1990. ISBN: 9783519030706.
- [Hof98] Kurt Hoffmann. *VLSI-Entwurf: Modelle und Schaltungen*. 4. Aufl. De Gruyter Oldenbourg, 1998. ISBN: 9783486597554.
- [Ito01] Kio Itoh. *VLSI Memory Chip Design*. Springer-Verlag Berlin Heidelberg, . ISBN: 3540678204.
- [Kit13] Charles Kittel. *Einführung in die Festkörperphysik*. De Gruyter Oldenbourg, 2013. ISBN: 978-3486597554.
- [Vee17] Harry J.M. Veendrick. *Nanometer CMOS ICs: From Basics to ASICs*. Springer International Publishing, 2017. ISBN: 9783319475950.
- [Ulr19] Eberhart Gamm Ulrich Tietze Christoph Schenk. *Halbleiterschaltungstechnik*. Springer Vieweg, Auflage: 16., erw. u. aktualisierte Aufl. 2019 Edition (5. Juli 2019). ISBN: 9783662485538.



Elemente der Rechnerarchitektur

Inhaltsverzeichnis

5.1 Kombinatorische Schaltungen – 289

5.2 Sequenzielle Schaltungen – 294

5.3 Arithmetische Schaltungen – 309

5.4 Speicher – 317

Weiterführende Literatur – 329

*Die höchsten Türme fangen beim Fundament an
Edison*

Aus den digitalen Schaltungen werden in diesem Kapitel Rechenwerke und andere elementare Bauteile der Rechnerarchitektur konstruiert. Dies sind neben unterschiedlichen Varianten von Addierwerken Zähler und Speicher. Am Ende sollte der Leser ein Rechenwerk selbstständig aufbauen, seine Funktion erläutern und die Vor- und Nachteile verschiedener Implementierungen benennen können. Er ist in der Lage, unterschiedliche Speicher und ihre Implementierung im Kontext der Rechnerarchitektur einzuordnen und hat ein tiefes Verständnis grundlegender Komponenten von Rechenanlagen.

Aus den einzelnen digitalen Schaltungen, den logischen Gattern, den Flipflops und Speicherzellen, sollen nun größere Komponenten zusammengebaut werden. Diese Bausteine bilden das Grundgerüst eines Rechners. So wie ein Haus aus Wänden, Decken, Treppen, Fenstern, Türen und einem Dach aufgebaut ist, sind die im Folgenden diskutierten Teile grundlegend für die Architektur eines Computers. Wird angenommen, dass Logikgatter die digitalen Atome eines Rechners sind, dann sind bereits die Flipflops einfache und die Elemente der Rechnerarchitektur komplexe Moleküle.

In einem binären Rechner müssen die codierten Zahlen und Daten in unterschiedliche Formate umgewandelt und verknüpft werden. Dies wird mit den kombinatorischen Schaltungen erreicht. In einer kombinatorischen Schaltung ist das Ergebnis am Ausgang nur von den Daten am Eingang abhängig. Neben diesen bilden die sequenziellen Schaltungen wichtige grundlegende Komponenten eines Rechners. Diese sind meist rückgekoppelte kombinatorische Schaltungen. Das Ergebnis ist daher von den vorherigen Eingaben und den daraus resultierenden Zuständen der Schaltung abhängig. Bei sequenziellen Schaltungen ist also noch die Reihenfolge der Eingaben und Ausgaben zu betrachten. Aus kombinatorischen und sequenziellen Schaltungen können dann sowohl Speicher zum Ablegen der Daten und der Variablen als auch arithmetische und logische Schaltungen für die Rechenoperationen aufgebaut werden.

Neben der Funktion des Rechnens müssen Daten auch transportiert werden. Dabei werden oft mehrere Komponenten gleichzeitig angesprochen, und einer dieser Bausteine ist von der Steuerung zur Kommunikation auszuwählen. Hinzu kommen komplexe Protokolle, die mit ausschließlich kombinatorischen Schaltungen nicht realisiert werden können. Zu diesem Zweck sind also wieder sequenzielle Schaltungen notwendig. Daher sind noch Busse und Busprotokolle als wichtige grundlegende Elemente eines Rechners zu nennen.

Kombinatorische Schaltungen realisieren eine Schaltfunktion oder mehrere Schaltfunktionen in Form eines Schaltnetzes. Sie speichern keine Daten und besitzen im Gegensatz zu Schaltwerken keine Rückkopplungen. Kombinatorische Schaltungen werden häufig auch als kombinatorische Logik bezeichnet. Derartige Komponenten sind einfach zu entwerfen:

- Zunächst werden die Eingangs- und die Ausgangsvariablen der Schaltung festgelegt oder spezifiziert.
- Mithilfe einer Wahrheitstabelle wird dann die Funktion der kombinatorischen Schaltung festgelegt.
- Nach Aufstellen der disjunktiven kanonischen Normalform kann diese Schaltfunktion minimiert werden und anschließend
- mithilfe von Logikgattern implementiert werden.

Schaltnetze

5.1.1 Codierer- und Decodierschaltungen

Die erste einfache Anwendung von Schaltfunktionen ist ein Codierer. Dieser hat mehr Ein- als Ausgänge. Mithilfe eines Codierers können beispielsweise Kontaktstellungen von einer Tastatur in ein kompaktes binäres Wort übersetzt werden. Sie werden aber auch eingesetzt, um Zahlen aus einem Zahlenbereich in einen anderen umzuwandeln. So lassen sich mit einem Codierer beispielsweise BCD-codierte Dezimalzahlen in die 2-Komplement-Darstellung zum binären Rechnen überführen.

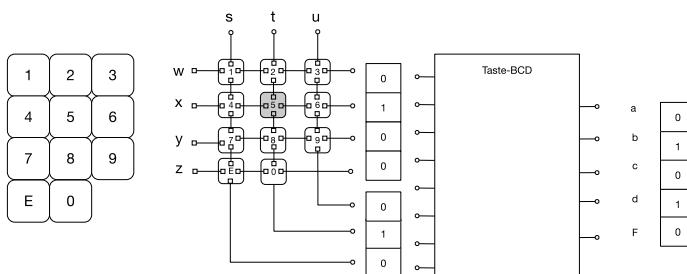


Abb. 5.1 BCD-Tastaturcodierer

► Tab. 5.1 Wahrheitstabelle eines BCD-Tastencodierers

	Eingänge			Ausgänge								
	s	t	u	w	x	y	z	a	b	c	d	e
0	0	1	0	0	0	1		0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1	0
2	0	1	0	1	0	0	0	0	0	1	0	0
3	0	1	1	1	0	0	0	0	0	1	1	0
4	1	0	0	0	1	0	0	0	1	0	0	0
5	0	1	0	0	1	0	0	0	1	0	1	0
6	0	1	1	0	1	0	0	0	1	1	0	0
7	1	0	0	0	0	1	0	0	1	1	1	0
8	0	1	0	0	0	1	0	0	0	0	0	0
9	0	1	1	0	0	1	0	0	0	1	0	1
E	1	0	0	0	0	0	1	0	0	0	0	1

► Beispiel 5.1.1 – Codierung von Tastatureingaben

Eingabe-und Ausgabe

Bei einer Tastatur sind die Tasten oft über einer Matrix von Kontakten angebracht. Wird eine Taste gedrückt, so wird dies in der entsprechenden Zeile und Spalte signalisiert (► Abb. 5.1). Die Tastendrücke sollen dann mithilfe des Codierers in einen BCD-Code übersetzt werden. Die dazugehörige Wertetabelle 5.1 ist im Folgenden ausschnittsweise angedeutet. ◀

Decodierer kehren den Prozess der Codierung um. Aus einer kleinen Menge von Eingängen können wieder einzelne binäre Signale extrahiert werden. Um einen Decodierer zu bauen, wird eine Wertetafel aufgestellt. Mit den Methoden der Schaltalgebra lässt sich dann leicht die entsprechende kombinatorische Schaltung aufbauen.

► Beispiel 5.1.2 – 7-Segment-Anzeige

Alphanumerische Zeichen können mit einer 7-Segment-Anzeige dargestellt werden. Jedes Segment der Anzeige ist Teil einer Dezimalzahl. Sind alle Segmente eingeschaltet, erscheint beispielsweise eine 8. Eine BCD-codierte Zahl kann nun einfach decodiert werden und mithilfe einer 7-Segment-Anzeige dargestellt werden. Die Elemente u , v , w , x , y und z der 7-Segment-Anzeige können einzeln angesprochen werden. Ist die dazugehörige Leitung 1, leuchtet das entsprechende Segment. Damit können die dezimalen Ziffern wie in der ► Abb. 5.2 gezeigt dargestellt werden. Mithilfe der zugehörigen Wahrheitstafel kann dann die Funktion des BCD-

5.1 · Kombinatorische Schaltungen

	Eingänge				Ausgänge						
	a	b	c	d	t	u	v	w	x	y	z
0	0	0	0	0	1	1	1	0	1	1	1
1	0	0	0	1	1	0	0	0	0	0	1
2	0	0	1	0	1	1	0	1	1	0	0
3	0	0	1	1	1	0	1	0	1	1	1
4	0	1	0	0	1	0	1	1	0	0	1
5	0	1	0	1	0	1	1	1	0	1	1
6	0	1	1	0	0	1	1	1	1	1	1
7	0	1	1	1	1	0	0	0	0	1	1
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1

Abb. 5.2 7-Segment-Decodierer

7-Segment-Codierer spezifiziert werden. Das Schaltsymbol eines Codierers ist ein Block mit den jeweiligen Ein- und Ausgängen. Die Eingänge werden mit Ziffern beschriftet. Diese codieren die internen Zustände des Codierers, in denen ausschließlich die dazugehörige Variable 1 ist. Also an Eingang a ist in der Zeile 8 (interner Zustand 8) nur das Bit a aktiv. Die Variablen b,c,d sind in dieser Zeile 0. An den Ausgängen werden die internen Zustände angeschrieben, für die die jeweilige Ausgangsvariable 1 wird. Die Beschriftung an diesen codiert also immer eine Spalte der Ausgangsmatrix der Wahrheitstabelle des Codierers. Die entsprechende Wertetabelle ist in Tab. 5.2 abgebildet. ◀

Tab. 5.2 Wahrheitstabelle eines 7-Segment-Decodierers

	Eingänge	Ausgänge
	abcd	tuvwxyz
0	0000	1110111
1	0001	1000001
2	0010	1101110
3	0011	1101011
4	0100	1011001
5	0101	0111011
6	0110	0111111
7	0111	1100001
8	1000	1111111
9	1001	1111001

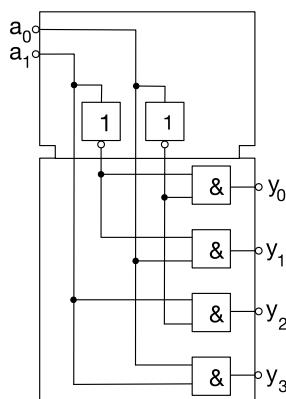


Abb. 5.3 1-n-Decodierer

Multiplexer

Eine besondere Form eines Decodierers ist der 1-n-Decodierer. Beim 1-n-Decodierer werden die Worte am Eingang derart de-codiert, dass immer nur eine einzige Leitung des Bausteins aktiv ist. Der 1-n-Decodierer ermöglicht so gezielt die Auswahl von Daten und codiert beliebige Eingangsworte in eine „One-hot-Codierung“ um. Aufgebaut ist der 1-n-Codierer aus der UND-Matrix eines logischen Feldes (PLA) wie in Abb. 5.3 gezeigt. Eine ODER-Schaltung ist nicht notwendig, da ja zur-zeit immer nur eine der Ausgangsleitungen aktiv sein kann. Der entsprechend umzusetzende Code kann dann direkt im Feld der UND-Matrix codiert werden (Tab. 5.3).

5.1.2 Datenverteiler

Ein Multiplexer ist eine Art Datenwegschalter: Mehrere Datenleitungen des Eingangs lassen sich auf einen Ausgang schalten (Abb. 5.4). Ähnlich wie bei einer Weiche bei der Eisenbahn können die Daten unterschiedlich abzweigen. Die Steuerleitungen a entsprechen einem codierten Auswahlwort; die Datenleitungen x_0 bis x_n können selbst eine beliebige Breite von m parallelen Leitungen besitzen. Multiplexer ermöglichen es, mehrere Datenquellen auf eine Datensenke zu schalten und auszuwählen, von welcher Quelle die Daten angenommen werden sollen.

Tab. 5.3 1-n-Decodierer

5.1 · Kombinatorische Schaltungen

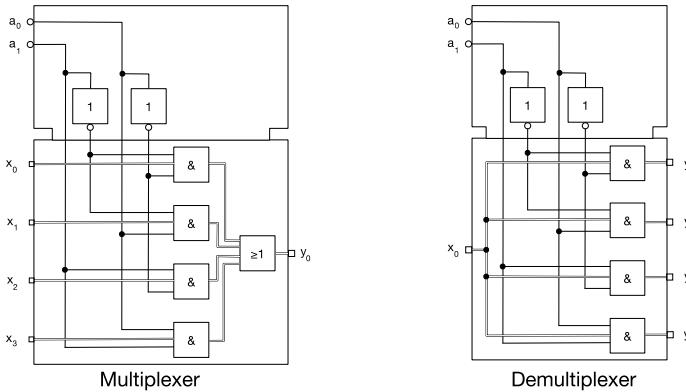


Abb. 5.4 Multiplexer und Demultiplexer für 4 Eingänge bzw. Ausgänge

Im umgekehrten Fall, wenn eine Eingangsleitung auf mehrere Ausgänge zu schalten ist, heißt die Schaltung Demultiplexer. Demultiplexer ermöglichen es, eine Datenquelle auf mehrere Datensenken zu verteilen und auszuwählen, an welche Senke die Daten verschickt werden. Auch ein Demultiplexer entspricht einer Datenweiche.

Barrel-Shifter

Ein Barrel-Shifter ist eine kombinatorische Schaltung zum Verschieben von Bits. Dazu werden mehrere Multiplexer parallel geschaltet. Sollen n Bit um n Stellen verschoben werden, werden Multiplexer mit n Eingängen benötigt. Die Abb. 5.5 zeigt einen Barrel-Shifter für 4 Bit. Die 4 Multiplexer werden so verschaltet, dass wenn der Eingang 0 gewählt wurde, einfach alle Eingänge auf die jeweiligen Ausgänge der Multiplexer ausgegeben werden. Wird dagegen Eingang 1 der Multiplexer aktiviert, werden X_1 auf Y_0 , X_2 auf Y_1 , X_3 auf Y_2 geschaltet. Y_3 benötigt dann einen zusätzlichen Eingang X_4 . Das Prinzip setzt sich fort: Werden durch die Auswahlleitungen s die Multiplexer auf 2 geschaltet, gilt: X_2 auf Y_0 , X_3 auf Y_1 , X_4 auf Y_2 und X_5 auf Y_3 . Mithilfe der Auswahlleitungen kann also entschieden werden, ob um 1, 2, 3 oder 4 bit verschoben werden soll.

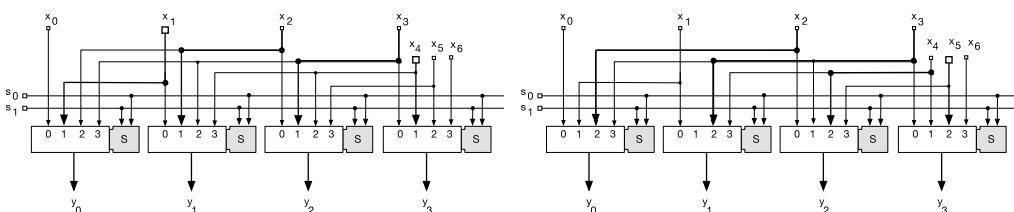
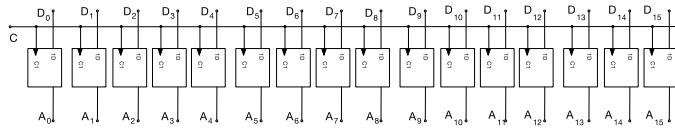


Abb. 5.5 Funktion eines 4-Bit-Barrel-Shifters: Verschieben um 1 und um 2 Stellen



■ Abb. 5.6 Register

5.2 Sequenzielle Schaltungen

Es zeigte sich früh in der Entwicklung digitaler Rechenanlagen, dass nicht nur mathematische Operationen zu berechnen sind, sondern dass Algorithmen sequenziell nacheinander ausführbare Anweisungen benötigen. Ein Beispiel ist das Zählen. Dieses funktioniert nur in einem zeitlichen Kontext, wenn also ein Vorher und ein Nachher definiert sind. Schaltungen, deren Verhalten von der Zeit abhängen, werden sequenzielle Schaltungen (lat. „sequens“, aufeinander folgend) genannt. Synchrone sequenzielle Schaltungen heißen Schaltwerke, insbesondere wenn die Rückkopplung über ein Schaltnetz erfolgt, wie bei den Zählern und den endlichen Zustandsautomaten.

5.2.1 Schieberegister

Am einfachsten lassen sich Daten in einem Rechner mit Registern speichern. Register sind eine spezielle Form des Speichers. Zum einen können mit Registern Berechnungen zeitlich entkoppelt werden, zum anderen werden Register auf der Ebene der Rechnerarchitektur genutzt, um Variablen in einem Prozessor zwischenzuspeichern. Werden mehrere D-Flipflops parallel geschaltet, entsteht ein Register, wodurch sie grundsätzlich SRAMs ähneln. Allerdings enthalten Registersätze eines Prozessors wenig Speicherzellen im Vergleich zu einem SRAM-Baustein. Daher kann bei Registern auf aufwendige Verstärker und Adressdecoder verzichtet werden. Ein Register kann pegel- (Latches) oder flankengetriggert (D-Flipflop) aufgebaut werden. In einem Register können so mehrere Bits gleichzeitig gespeichert werden. In der Rechnerhardware dienen Register dazu, arithmetische und logische Komponenten zu entkoppeln. Die ■ Abb. 5.6 zeigt ein 4-Bit-Register aus vier flankengetriggerten D-Flipflops. Das Register übernimmt das Datum an den Eingängen immer zu einer steigenden Taktflanke.

Bits können mit einem Barrel-Shifter in einem Datum um eine oder mehrere Stellen verschoben werden. Häufig sollen aber Daten nicht parallel über mehrere Leitungen transportiert

werden, sondern seriell auf nur einem Weg, um so den Schaltungsaufwand zu reduzieren. Ein mehrere Bit langes Datum ist also zeitlich zu strecken. Diese Aufgabe erfüllt ein Schieberegister. Die einfachste Form eines Schieberegisters sind mehrere hintereinandergeschaltete D-Flipflops. Bei einer derartigen Schaltung kann mithilfe eines Taktes ein serieller Datenstrom am Eingang *ID* in das Register eingetaktet werden. Nach n Takten sind dann n Bit in das Register geladen. Diese können anschließend über den Ausgang *S* parallel ausgelesen werden. Mit dieser Schaltung lassen sich serielle Daten in ein Datenwort umwandeln.

Das ursprüngliche Problem ist damit aber noch nicht gelöst. Dazu müssen Daten parallel in die Flipflops geschrieben werden können, um sie dann seriell auszulesen. Dies kann mit einem ladbaren Schieberegister erreicht werden. Bei diesem wird vor jedes Flipflop ein Multiplexer geschaltet, sodass jeweils der Ausgang eines vorhergehenden Flipflops mit dem Eingang eines folgenden verbunden ist. Werden die Takteingänge der Flipflops nun mit einem Signal getaktet, werden die einzelnen Bits durch das Register geschoben. Am Ausgang liegt dann ein getakteter serieller Bitstrom an, der dem ursprünglich geladenen Bitmuster entspricht. Alle Multiplexer können durch ein Steuersignal so geschaltet werden, dass bei einer Taktflanke Daten vom parallelen Eingang *SI* gelesen werden. Ist die Steuerleitung $L = 0$, können diese Bits dann nach rechts durchgeschoben werden. Der serielle Datenstrom wird anschließend an *OD* ausgegeben. Über den Eingang *ID* kann gleichzeitig ein serieller Datenstrom eingelesen werden. Werden derartige Schieberegister in Rechnern eingesetzt, ist *ID* = 0, sodass links immer Nullen in das Register eingelesen werden (Abb. 5.7).

Das Prinzip lässt sich verallgemeinern. Werden die Ausgänge niedrigstwertiger Stellen über Multiplexer zurück an den Ausgang einer höherwertigen Stelle geführt, wird die Schieberichtung umgekehrt. Das Register ist also in der Lage, nach links und nach rechts zu schieben. Über die Steuerleitung kann

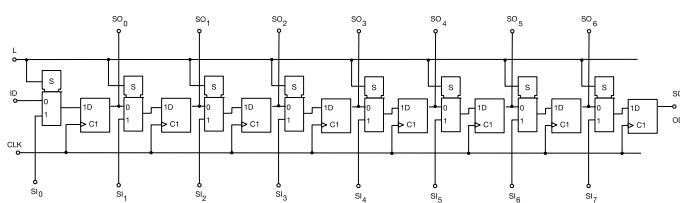


Abb. 5.7 8-Bit-Schieberegister

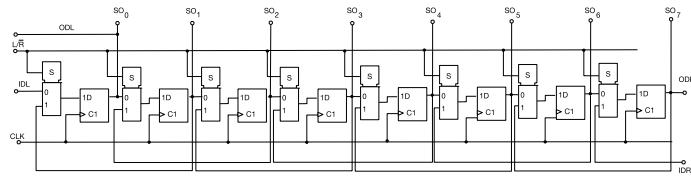


Abb. 5.8 Schieberegister zum Linkss- und Rechtsschieben

5

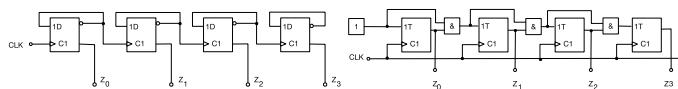


Abb. 5.9 Asynchron- und Synchronzählerschaltungen mit Flipflops

dann die jeweilige Schieberichtung ausgewählt werden. Führt man den Multiplexer als Schaltung mit drei Eingängen aus und erweitert man die Steuerleitung um eine zweite Leitung, kann ein Links-rechts-Schieberegister mit parallelem Ladeeingang aufgebaut werden (Abb. 5.8).

5.2.2 Zähler

Eine elementare mathematische Operation ist das Zählen. Mechanische Rechenanlagen arbeiteten mit Zählwerken, und die Addition und Subtraktion wurden mit diesen direkt durchgeführt. Auch die ersten elektronischen Rechner führten die Operation Addieren mit Zählern aus. Somit waren Zählschaltungen zunächst die elementaren Schaltungen digitaler Computer. In modernen Rechenanlagen sind häufig Variablen um einen festen Wert zu erhöhen. Werden die Flipflops eines Registers einzeln rückgekoppelt, entsteht ein Zählregister. Solche Register können als Zähler oder aber auch als Frequenzteiler eingesetzt werden.

Asynchronzähler

Wird der Signalverlauf aller 4 Stellen dieses Zählers analysiert, zeigt sich, dass das Zählregister nach der 1, der 3, der 7, der 11, der 13 und der 15 bedingt durch die Schaltzeiten der Flipflops falsche Werte annimmt. Nach der 1 folgt für kurze Zeit nicht die 2, sondern die 0; nach der 3 liegt für kurze Zeit am Ausgang die 2 an. Da Laufzeiten Einfluss auf das Zählergebnis haben, heißt diese Schaltung Asynchronzähler (Abb. 5.9, 5.10, 5.11 und 5.12).

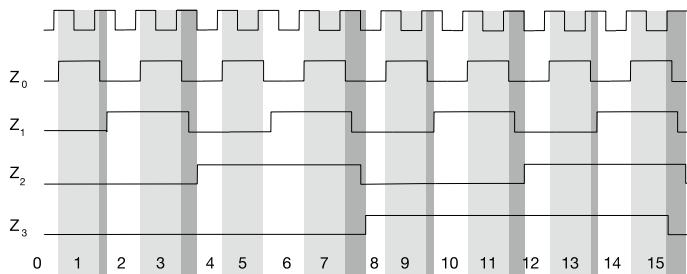


Abb. 5.10 Signalverlauf des Asynchronzählers

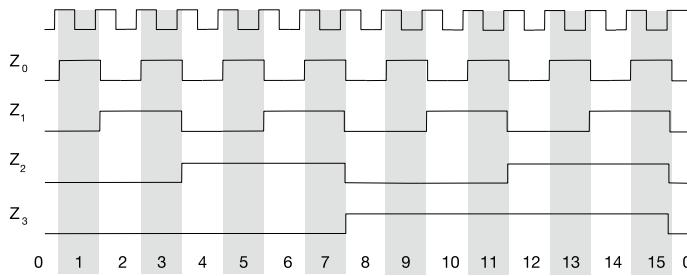


Abb. 5.12 Signalverlauf des Synchronzählers

Das unerwünschte Verhalten des Asynchronzählers kann beseitigt werden, wenn an den Ausgang wieder ein Register angeschlossen wird und diese Flipflops immer dann geladen werden, wenn der Zustand des Zählregisters eingeschwungen und gültig ist. Allerdings erfordert diese Lösung den Aufbau eines weiteren Registers. Da dieses unerwünschte Verhalten durch unterschiedliche Laufzeiten in den Signalwegen entsteht, kann dieses durch eine geeignete Synchronisation ausgeschlossen werden. Bei dieser „Synchronzähler“ genannten Schaltung werden die Ausgänge mehrerer Flipflops über ein UND-Gatter synchronisiert. Am Eingang des 2. Flipflops ist dies noch nicht notwendig, am Eingang des 3. Flipflops werden aber Z_0 und Z_1 synchronisiert. Mit jedem Flip flop müssen mehr niedrigstwertige Ausgänge über ein UND-Gatter synchronisiert werden. Daher steigt der Schaltungsaufwand mit jeder Stelle des Synchronzählers.

Synchronzähler

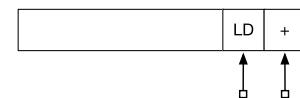


Abb. 5.11 Schaltzeichen
ladbarer Zähler

5.2.3 Steuerwerke

Eine wichtige Aufgabe im Entwurf von Rechenanlagen kommt den Steuerwerken zu. Steuerwerke automatisieren die Steuerung der Recheneinheiten und implementieren Automaten. Ein

Steuerwerk wird als synchrones Schaltwerk realisiert. Im Gegensatz zu den asynchronen Schaltwerken ändern sich Zustandsvektor und Ausgangsvektor synchron zu einem externen Takt. Dadurch kann das Schaltwerk zu diskreten Zeitpunkten in Abhängigkeit vom anliegenden Eingangssignal und dem aktuellen, im Register gespeicherten Zustand einen neuen Ausgangswert berechnen.

5.2.3.1 Automaten

Aus dem täglichen Leben ist der Begriff Automat (lat. „automatus“, selbstständig handelnd) bekannt. Ein Automat erledigt unterschiedliche Dinge ohne Eingriff eines Menschen. Noch vor 50 Jahren waren Fahrkarten für eine Bahnfahrt nur an einem Schalter zu erstehen. Ein Bahnbeamter stellte die Fahrausweise jeweils in Abhängigkeit vom Fahrziel des Fahrgastes aus und kassierte den entsprechenden Fahrpreis. Heute wird das Ziel wie selbstverständlich an einem Automaten ausgewählt, der Automat berechnet selbstständig den Preis und druckt nach der Bezahlung die richtige Fahrkarte. Einfachere Automaten verkaufen Kaugummis, Eis, Kaffee oder Zigaretten. Nach Bezahlung liefern diese Maschinen das gewünschte Produkt selbstständig, also automatisch. Ein Automat muss demnach eine Folge von Schritten in Abhängigkeit von den vorgenommenen Eingaben ausführen und die in jedem Schritt festgelegten Aufgaben erfüllen. In der technischen Informatik soll es reichen, unter einem Automaten eine sequenziell arbeitende Schaltung zu verstehen. Im Unterschied zu den Schieberegistern und Zählern soll ein Automat in Abhängigkeit von weiteren Eingaben und dem jeweiligen Zeitschritt unterschiedliche Ausgaben erzeugen. Um dieses Verhalten formal besser greifen zu können, wird der Begriff „Zeitschritt“ durch den Begriff „Zustand“ ersetzt. Im Gegensatz zu linear ablaufenden Zeitschritten besitzt ein Zustand eine festgelegte Eigenschaft. Diese kann frei gewählt werden, sodass sich im Unterschied zu einer linear zählenden Maschine Sprünge und Rücksprünge realisieren lassen. Der Übergang von einem Zustand in einen anderen wird „Transition“ genannt. Ein Automat soll immer so lange in einem Zustand verharren, bis sich an seinem Eingang eine Änderung ergibt. In manchen Zuständen führt diese dann zu einem Zustandsübergang oder eben zu einer Transition mit einem bestimmten Verhalten am Ausgang.

Automaten werden in Form eines Graphen beschrieben. In einem solchen Graphen sind die Zustände als Knoten und die Zustandsübergänge als Kanten modelliert. Jeder Zustand bekommt dabei eine oder mehrere Eigenschaften oder Attribute zugewiesen. In der Regel hat ein Zustand ein Attribut in Form eines Namens oder einer Nummer. Jede Kante besitzt mindes-

tens zwei Attribute. Ein spezifiziertes Eingangswort, bei dessen Anlegen die Transition ausgelöst wird, und ein Ausgangswort, das als Ergebnis von dem Automaten ausgegeben wird. Ein solcher Automat ist nach dem amerikanischen Mathematiker George H. Mealy (1927–2010) benannt.

Der abgebildete Mealy-Automat besitzt 4 Zustände S_1 bis S_4 . Liegt am Eingang des Automaten eine 1 an und ist der Automat in Zustand S_1 , gibt der Automat eine 1 aus und wechselt in Zustand S_2 . Liegt im Zustand S_1 eine 0 am Eingang an, wird ebenfalls eine 1 ausgegeben. Der Automat wechselt aber in Zustand S_3 . Angenommen nach dem Einschalten befindet sich der Automat in S_1 (dies wird durch die doppelte Linie des Knotens S_1 dargestellt), dann kann der Ablauf wie in □ Tab. 5.4 resultieren.

Mealy- Automat

Definition 5.2.1 – Mealy-Automat

Seien Z eine endliche Menge von Zuständen $Z := z_0, z_1, \dots, z_n$, X ein Eingabevektor oder Eingabealphabet $X := x_0, x_1, \dots, x_m$ und Y ein Ausgabevektor oder Ausgabealphabet $Y := y_0, y_1, \dots, y_0$. Dann ist der Mealy-Automat durch die Übergangsfunktion $\delta: X \times Z \rightarrow Z$ und die Ausgabefunktion $\lambda: X \times Z \rightarrow Y$ definiert.

Ein alternatives Konzept ist der nach dem amerikanischen Chemiker und Mathematiker Edward F. Moore (1925–2003) benannte Moore-Automat. Bei einem Moore-Automaten wird

Moore- Automat

□ Tab. 5.4 Zustandswechsel

Zeit	Zustand	Eingang	Ausgang	Folgezustand
t_0	S_1	0	1	S_3
t_1	S_3	0	1	S_3
t_2	S_3	0	1	S_3
t_3	S_3	1	0	S_4
t_4	S_4	1	1	S_1
t_5	S_1	1	1	S_2
t_6	S_2	1	0	S_3
t_7	S_3	1	0	S_4
t_8	S_4	1	1	S_1

der Ausgang in Abhängigkeit vom Zustand gesetzt. Solange der Automat in einem betreffenden Zustand ist, liegt der spezifizierte Wert am Ausgang an. Ein zum ersten Beispiel äquivalenter Moore-Automat ist in Abb. 5.13 gezeigt. Ein Moore-Automat kann aus einem Mealy-Automaten abgeleitet werden. Dazu müssen neue Zustände eingeführt werden: Immer wenn Transitionen mit unterschiedlichen Ausgaben auf einen Zustand im Mealy-Automaten führen, muss für jeden Zustandsübergang des Mealy-Automaten ein eigener Zustand im Moore-Automaten abgeleitet werden. In der Abbildung ist das durch die Zustände $S2a$ und $S2b$, $S3a$ und $S3b$, $S4a$ und $S4b$ ausgedrückt.

Definition 5.2.2 – Moore-Automat

Seien Z eine endliche Menge von Zuständen $Z := z_0, z_1, \dots, z_n$, X ein Eingabevektor oder Eingabealphabet $X := x_0, x_1, \dots, x_m$ und Y ein Ausgabevektor oder Ausgabealphabet $Y := y_0, y_1, \dots, y_o$. Dann ist der Moore-Automat durch die Übergangsfunktion $\delta: X \times Z \rightarrow Z$ und die Ausgabefunktion $\lambda: Z \rightarrow Y$ definiert.

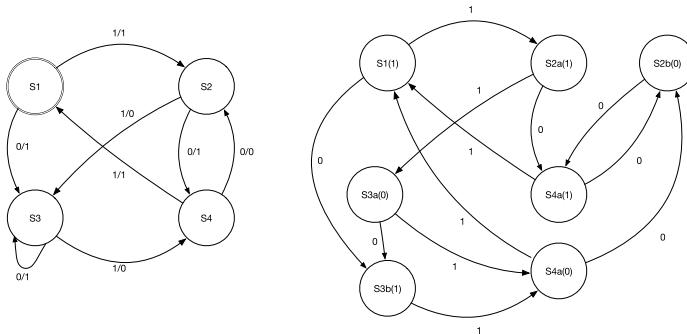


Abb. 5.13 Mealy- und äquivalenter Moore-Automat als Graph

Bei einem Medwedew-Automaten ist der Zustand, in dem sich der Automat befindet, gleichzeitig das Ausgabewort. Als Spezialfall eines Moore-Automaten ist der Medwedew-Automat sehr einfach zu realisieren, da die Ausgangsfunktion nicht implementiert werden muss.

Medwedew-Automat

5.2.3.2 Maskenprogrammierbares Steuerwerk

Als digitale Schaltungen werden Mealy- und Moore-Automaten auch als Zustandsmaschinen oder endliche Automaten (engl. „finite state machine“, Akronym FSM) bezeichnet. Eine Zustandsmaschine koppelt eine Schaltfunktion über ein Zustandsregister zurück. Die Abb. 5.14 zeigt die drei unterschiedlichen Varianten. Das Zustandsregister wird mithilfe von flankengesteuerten D-Flipflops realisiert. Die Übergangs- und Ausgangsfunktionen λ und δ dagegen werden als Schaltnetz implementiert. Die Zustandsmaschine besitzt einen Eingangsvektor x und einen Ausgangsvektor y . Der Eingangsvektor realisiert die Eingangsvariablen, der Ausgangsvektor die Ausgangsvariablen des Automaten. In unserem Fall sind Ein- und Ausgangsvektor und somit die Eingangs- und Ausgangsvariablen binär codiert. Die Schaltfunktionen λ und δ und das Zustandsregister können unterschiedlich angeordnet werden, je nachdem welche Form eines Automaten realisiert werden soll. Besonders einfach wird die Realisierung eines solchen Schaltwerks, wenn die Schaltnetze λ und δ in Form einer PLA aufgebaut werden. Da eine PLA jede beliebige disjunktive kanonische Normalform implementieren kann, kann man dadurch die Schaltnetze als regelmäßige, flächensparende Schaltung realisieren. Alternativ ist es auch möglich, wie bei einer PLA die Schaltwerke mithilfe eines Speichers aufzubauen.

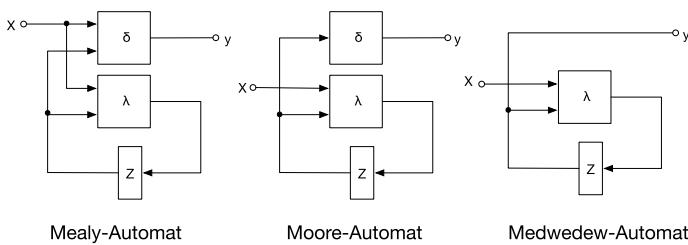


Abb. 5.14 Automaten implementiert mit Schaltnetzen und einem Zustandsregister

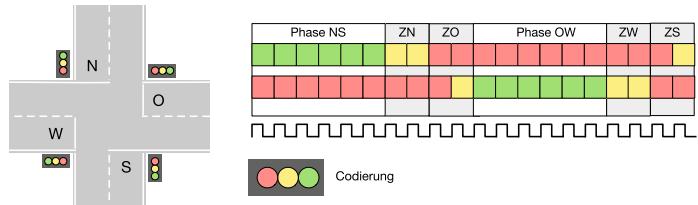


Abb. 5.15 Straßenkreuzung mit Ampelphasen

5

► Beispiel 5.2.1 – Ampelschaltung

Es soll eine elektronische Schaltung für die Steuerung der Ampeln einer einfachen Kreuzung aufgebaut werden (Abb. 5.15). Diese ist mit 4 Ampeln ausgestattet. Jeweils 2 Ampeln steuern paarweise den Verkehr in Nord-Süd- und Süd-Nord-Richtung und 2 Ampeln regeln die Fahrt in Ost-West- oder West-Ost-Richtung. Dabei sollen beide Ampeln Rot anzeigen, wenn die jeweiligen Fahrzeuge anhalten sollen. Sind 2 Ampeln grün, darf die Kreuzung passiert werden. So wie wir es von einer regulären Ampel gewohnt sind, sollen die Ampeln bei einem Wechsel der Phase von Grün auf Rot zunächst auf Gelb schalten und bei einem Wechsel von Rot auf Grün, Gelb und Rot gleichzeitig anzeigen. ◀

Die Ampelphasen bestehen aus zwei Grün-Rot-Phasen, den Phasen NS und OW und den Zwischenphasen ZN, ZO und ZW und ZS. Zunächst ist die Nord-Süd-Verbindung freigegeben, die Nord- und die Südampele zeigen Grün, die West- und die Ostampel Rot. Nach 6 Takschritten soll umgeschaltet werden. Dazu wechseln die Nord- und die Südampele für 2 Takte auf Gelb, und die Fahrt in Ost-West- und West-Ost-Richtung ist durch das Leuchten von Rot weiter gesperrt. In der Phase ZO zeigen alle 4 Ampeln für 1 Takschritt Rot, und die beiden West-Ost-Ampeln schalten für 1 Takt auf Rot-Gelb. In der Phase OW sind nun die Nord-Süd- und die Süd-Nord-Richtung gesperrt und die Ost-West- bzw. West-Ost-Richtung freigegeben. Der Wechsel in den Phasen erfolgt dann gespiegelt in den Phasen ZW und ZS: Zunächst schalten die Ampeln der Ost-West-Richtung auf Rot, und nachdem alle Ampeln für 1 Takt Rot zeigten, wechseln die Ampeln der Nord-Süd-Richtung auf Gelb-Rot.

Die einzelnen Ampelphasen lassen sich mit Zuständen in einem Mealy-Automaten beschreiben (Abb. 5.16). Um Zustände zu sparen, wird ein Zähler, in dem Beispiel Timer genannt, eingeführt. Dieser Zähler kann auf einen festen Wert gesetzt werden und dann gezielt herunterzählen. Der Wert dieses Timers kann vom Automaten abgeprüft werden. In dem Automaten sind die 6 Phasen als Zustände dargestellt. Codiert sind

5.2 · Sequenzielle Schaltungen

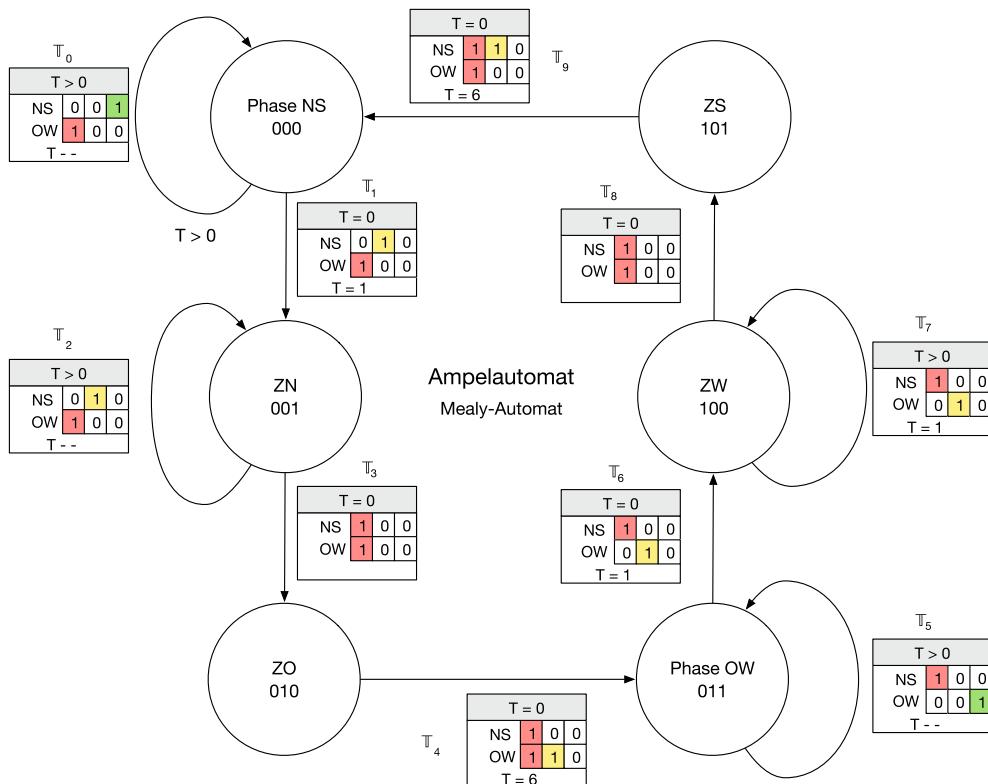


Abb. 5.16 Mealy-Automat der Ampelsteuerung

die Zustände des Automaten von 000 bis 101. Die Schaltung der Ampel erfolgt über die Zustandsübergänge, auch Transitionen genannt. Der Automat hat 9 Transitionen von T_0 bis T_8 . Die Transitionen sind beschriftet. Die Eingangsbedingung ist grau hinterlegt, im Beispiel ist das die Abfrage, ob ein Timer auf 0 zurückgezählt wurde. Ist die Bedingung einer Transition erfüllt, findet ein Zustandswechsel statt, und die Variablen NS, OW für die 4 Ampeln werden gesetzt. Gegebenenfalls wird der Timer noch neu programmiert. Der 1. Zustand ist der Zustand 000 (NS). Für diesen Zustand existieren 2 Transitionen T_0 und T_1 . Ist der Wert des Timers größer als 0, werden die Ampeln in Nord-Süd-Richtung (T_0) auf Grün geschaltet und die in Ost-West-Richtung auf Rot. Der Folgezustand ist wieder 000. Erst wenn der Timer 0 ist, löst Transition T_1 aus, und die Nord-Süd-Ampeln schalten auf Gelb, und der Folgezustand ist 001. Der Timer wird auf einen Zeitschritt gesetzt, sodass die nächste Transition im Zustand 001

5

T_2 ist. Die Nord-Süd-Ampeln bleiben auf Gelb und die Ost-West-Ampeln auf Rot. Erst beim Übergang T_3 in Zustand 010 schalten alle Ampeln auf Rot. Der Timer wird nicht gesetzt, sodass direkt nach dem Zustandswechsel direkt Transition T_4 ausgelöst wird. Dort bleiben alle Ampeln auf Rot, nur die Ampeln in Ost-West-Richtung schalten zusätzlich auf Grün. Der Timer wird dann wieder auf 6 Zeitschritte gesetzt um 6 Tackschritte lang die Transition T_5 auszuführen. Dann sind die Ost-West-Ampeln grün und die Nord-Süd-Ampeln rot, und der Timer wird heruntergezählt. T_5 wechselt schließlich in Zustand 011. Ist der Timer $T = 0$ löst Transition T_6 , und der beschriebene Vorgang findet spiegelbildlich über die Zustände 100, 101 zurück zu Zustand 000 statt.

Für die Ampelsteuerung werden 3 Flipflops zur Speicherung des Zustands benötigt (Abb. 5.17). Die logischen Schaltfunktionen lassen sich mit einer PLA realisieren. Der Vorteil dabei ist, dass die Zustandsübergangsfunktion und die Eingangs- und Ausgangslogik direkt aus dem Automaten heraus codiert werden können. Unmittelbar aus der PLA wird der Timer angesprochen. Der Befehl -- sorgt dafür, dass der Inhalt des Zählers um 1 heruntergezählt wird, und der Befehl LD bedeutet Laden des Bitmusters der an den Timer angeschlossenen Leitungen der PLA. Für jede Transition ist eine waagerechte Leitung vorgesehen. In der Eingangslogik kann dann für jede Transitionsleitung die entsprechende Auslösung eingetragen werden. Die Zustandsleitungen aus den Flipflops codieren jeweils den Zustand, aus dem die Transition heraus

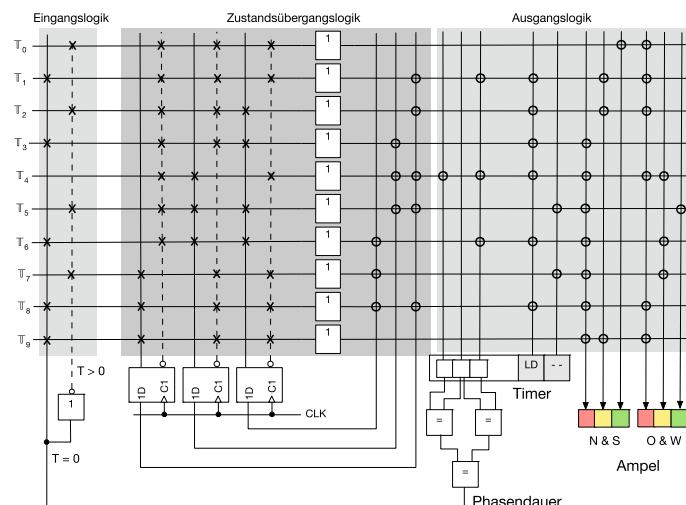


Abb. 5.17 Endliche Zustandsmaschine zur Steuerung der Ampel

folgt. In der Zustandsübergangslogik wird dann auch festgelegt, welcher Folgezustand nach Auslösen der Transition eingenommen werden soll. Die Ausgangslogik codiert die Ausgangsbelegung der jeweiligen Transition äquivalent zum Automaten. Das beschriebene Vorgehen ermöglicht die sofortige Umsetzung eines Automaten ohne Zuhilfenahme von Syntheseschritten und erlaubt den Aufbau einer endlichen Zustandsmaschine direkt im Layout oder mithilfe eines Speichers.

In dem Beispiel wurden die Zustände aufzählend codiert. Es ist jedoch auch denkbar, diese in anderer Form zu codieren. Die jeweiligen Codiermöglichkeiten haben dabei einen Einfluss auf den Flächen- oder den Energieverbrauch der Zustandsmaschine, und sie beeinflussen direkt die mögliche Schaltfrequenz.

Die Codierung von Zahlen nach der minimalen Hamming-Distanz ist bekannt; der Gray-Code ist ein Beispiel. Die Zustände einer Zustandsmaschine lassen sich ebenfalls mit dem Gray-Code codieren. Vorteil einer Zustandscodierung mit minimaler Hamming-Distanz ist das Einsparen von Logik in der Zustandsübergangsfunktion. Zustandscodierung mit dem Gray-Code führt zu einem geringeren Energieverbrauch des Automaten.

Bei der Codierung zur Zustandsminimierung oder der priorisierten Adjazenz wird das Absorptionsgesetz zur Logikminimierung angewendet:

- Haben zwei Zustände den gleichen Folgezustand, soll deren Codierung das Absorptionsgesetz erfüllen.
- Sind ein Zustand Vorgänger und ein anderer Zustand Nachfolger eines weiteren Zustands, sollen die Codierungen des Vorgängers und des Nachfolgers das Absorptionsgesetz erfüllen.
- Zustände, die für die gleiche Eingabe die gleiche Ausgabe erzeugen, sollen in ihrer Codierung das Absorptionsgesetz erfüllen.

Zustände, deren Codierung das Absorptionsgesetz erfüllen, sind in der entsprechenden Karnaugh-Tafel adjazent zueinander. Die Regeln 1 bis 3 können nicht immer eingehalten werden. In einem Graphen können Konflikte der Forderungen auftreten. Im Konfliktfall werden die Regeln mit der geringeren Nummer priorisiert (Abb. 5.18).

Gray-Codierung

Zustandsminimierte Codierung

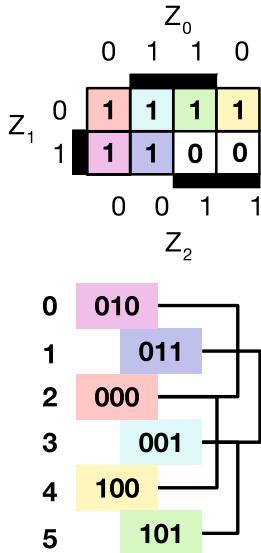


Abb. 5.18 Codierungsbildung des Ampelautomaten mit periodisierter Adjazenz

Bei der One-hot-Codierung werden für n Zustände n Flipflops benötigt, da immer nur eins der Flipflops 1 sein darf. Die One-hot-Codierung von Zustandsmaschinen vereinfacht die Zustandsübergangs- und die Ausgangslogik der Zustandsmaschine, hat jedoch den Nachteil, sehr viele Flipflops zu brauchen. Allerdings ist sie die Methode der Wahl, wenn die Zieltechnologie FPGAs sind. Da FPGAs wenig Logikfunktionen pro Zelle und damit Flipflops verwenden, können die Flipflops genutzt werden, um damit die Anzahl der Logikfunktionen und somit der Look-up-Tables (LUT) zu reduzieren. Auch führt eine Minimierung der Logik zu kurzen Signalpfaden in der Schaltung und ermöglicht so höhere Taktfrequenzen der Zustandsmaschine.

► Beispiel 5.2.2 – Ampelautomat

Die Zustandscodierung des Ampelautomaten kann aufzählend mit dem Gray-Code mit minimaler Hamming-Distanz, priorisierte Adjazenz und mit One-hot-Codierung abgebildet in Tab. 5.5 erfolgen. ◀

5.2.3.3 Speicherprogrammierbares Steuerwerk

Eine PLA ist maskenprogrammierbar, kann aber auch mithilfe von Speichern realisiert werden. Wird in der Zustandsmaschine die PLA durch Speicher ersetzt, kann die Logik des Automaten jederzeit reprogrammiert werden. Das Steuerwerk einer Maschine umzuprogrammieren heißt Mikroprogrammierung. Die Mikroprogrammierung hat einige Vorteile. Zum einen ist es möglich, Fehler im Steuerwerk nachträglich zu beheben, zum anderen lassen sich durch geschickte Erweiterung der Hardware der Zustandsmaschine Teile des Mikroprogramms

Tab. 5.5 Zustandscodierungen für den Ampelautomaten

Zustand	Zählende Codierung	Gray-Codierung	Priorisierte Adjazenz	One-hot-Codierung
NS	000	000	010	000001
ZN	001	001	011	000010
ZO	010	011	000	000100
OW	011	010	001	001000
ZW	100	110	100	010000
ZS	101	100	101	100000

wiederverwenden. Dies spielt u. a. eine Rolle bei der Umsetzung komplexerer Befehlssätze in Prozessoren. Die Mikroprogrammierung hat aber noch einen weiteren Vorteil: Als Hardware teuer war, konnte eine Maschine kostengünstig aus wenigen Komponenten aufgebaut werden. Die Funktion der Maschinenstruktur wurde dann mithilfe eines Mikroprogramms später festgelegt. Da ein Mikroprogramm fest ist, konnte der Speicher für das Mikroprogramm als ROM-Speicher ausgelegt werden. In Zeiten von Magnetkernspeichern und Magnetbändern ein kostengünstiger Weg, da der ROM-Speicher auch aus Kupferplatten mit Löchern bestehen konnte. Mikroprogrammsteuerwerke lassen sich unterschiedlich komplex aufbauen. Wie bereits erwähnt, können die Hardwarestrukturen einer PLA direkt durch Speicher ersetzt werden. Komplexere Mikroprogrammarchitekturen sind der zeigerbasierte Mikroprogrammspeicher, der stackbasierte Mikroprogrammspeicher und der frei programmierbare Mikroprogrammspeicher.

Eine alternative Möglichkeit, einen Einsprung in ein Mikroprogramm durchzuführen, ist die Verwendung eines Assoziativspeichers. Dabei kann das Bitmuster des Eingangsvektors als Einsprung in den Assoziativspeicher genutzt werden. Gleicher gilt für das Erzeugen von Folgezuständen. Mithilfe des Assoziativspeichers kann an jede beliebige Stelle des Mikroprogrammspeichers gesprungen werden, ohne einen Zähler zu benötigen (Abb. 5.19).

Der maskenprogrammierbare Teil der Zustandsmaschine kann durch Speichermatrizen ersetzt werden. Wird zusätzlich noch eine Codierlogik für den Speicher verwendet, kann das Zustandsregister durch einen Zähler implementiert werden. Dieser Zähler kann durch die Zustandsmaschine gesetzt oder hochgezählt werden. Der Wert im Zähler ist eine Referenz oder ein Zeiger („microprogram counter“) in den Spei-

Mikroprogramm im Assoziativspeicher

Mikroprogramm mit Sprüngen

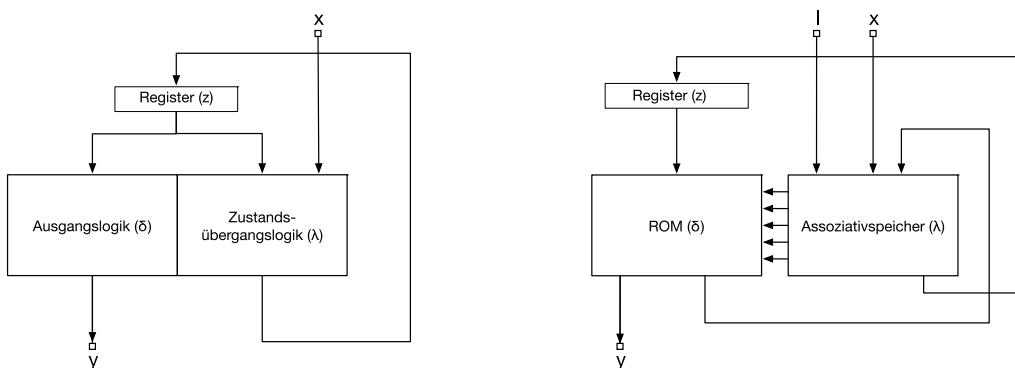


Abb. 5.19 Steuerwerk mit PLA und mit Speicher

Mikroprogramm mit Unterprogrammaufrufen

5

cher. Der Automat selbst kann durch Erhöhen das Mikroprogramm abarbeiten oder aber einen neuen Zustand setzen. Dies geschieht über den dem Zähler vorgesetzten Multiplexer. Dadurch können Sprünge im Mikroprogramm realisiert werden. Der Multiplexer wird an dieser Stelle benötigt, um auch einen externen Einsprung über den Eingang I zu implementieren (Abb. 5.20).

Erweitert man dieses Mikroprogrammwerk um einen Stapspeicher (Stack), können Zählerstände in diesem gespeichert werden. Damit ist es möglich, in einem Mikroprogramm Unterprogramme aufzurufen und aus diesen Unterprogrammen wieder an die Ursprungsstelle zurückzukehren. Dazu wird der Zählerstand in den Stapspeicher geladen. Dieser ermöglicht verschachtelte Unterprogrammaufrufe, da jeder weitere Unterprogrammaufruf einen neuen Wert des Zählers auf dem Stapel speichert. Soll aus einem Unterprogramm zurückgekehrt werden, wird der Zählerstand, der zuletzt auf den Stapel gelegt wurde, aus dem Speicher zurück in den Zähler geschrieben und das Mikroprogramm kann in der übergeordneten Unterroutine fortgesetzt werden. Der Stapspeicher ermöglicht daher eine einfache Verwaltung der Unterprogrammaufrufe.

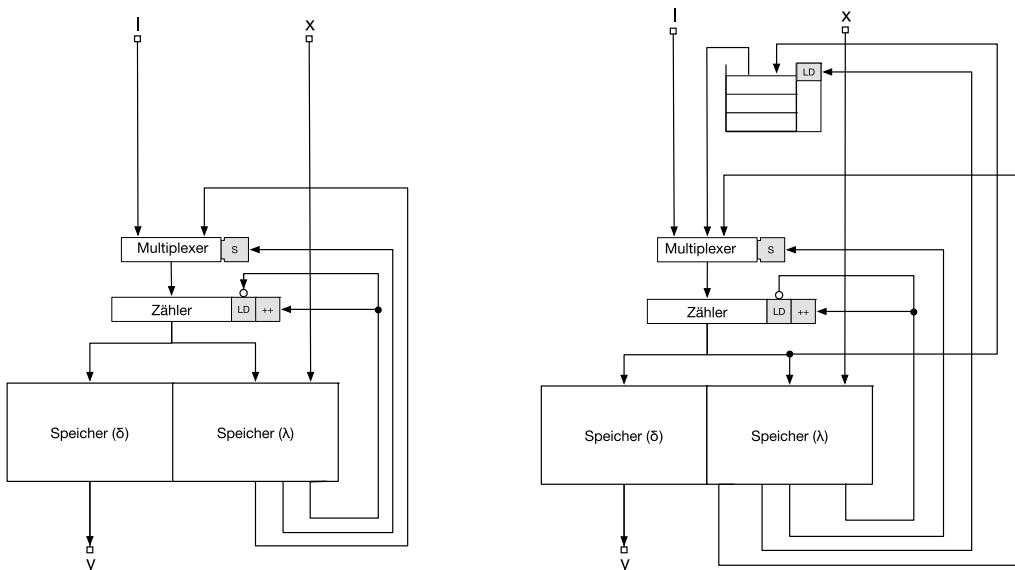


Abb. 5.20 Mikroprogrammierbares Steuerwerk mit Stapspeicher

Das Herz eines Rechners sind die arithmetischen Schaltungen. Jeder Computer muss die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division beherrschen. In der Regel werden dabei Multiplikation und Division mithilfe der Addition algorithmisch implementiert. Vereinzelt gibt es aber auch für die letzten zwei Operationen eigene Rechenschaltungen. Diese sind selten, da die Hardware für Multiplikation und Division hohen Bedarf an Chipfläche hat. Da Zahlendarstellungen im Rechner endlich sind, hat eine Rechenmaschine mit einer endlichen Wortbreite auch nur eine endliche Genauigkeit. Die mathematische Zuverlässigkeit stellt dabei besondere Anforderungen an die arithmetischen Schaltungen. Neben der Numerik spielen auch die wirtschaftliche Implementierung und das Zeitverhalten eine tragende Rolle. Im Laufe dieses Kapitels zeigt sich, dass der Flächenverbrauch und damit die Wirtschaftlichkeit mit der Geschwindigkeit arithmetischer Schaltungen im Konflikt stehen. Schnelle Hardware benötigt viel Fläche, und günstige Schaltungen sind dagegen langsam. Beim Entwurf arithmetischer Schaltungen ist also ein Kompromiss oder Optimum zu finden.

5.3.1 Addierwerk

Die grundlegende Schaltung in der maschinellen Arithmetik ist der Addierer. Sowohl Subtraktions- als auch Multiplikations- und Divisionswerke lassen sich aus Addierern aufbauen.

5.3.1.1 Serielles Addierwerk (Ripple-carry-Addierer)

Die einfachste Methode, einen Addierer zu implementieren, ist die direkte Realisierung der klassischen Schulmethode im binären Zahlensystem. Es wird jede Stelle einer Zahl im dualen Stellenwertsystem einzeln addiert. Dabei kann ggf. ein Übertrag (engl. „carry“) erzeugt werden, wenn die Summe zweier zu addierender Stellen die maximale Zifferngröße überschreitet. Bereits in ▶ Kap. 2 wurde eine Schaltfunktion beschrieben, die die Summe aus zwei Bits der Operanden und einem Übertrag bildet. Eine Schaltung, die diese Schaltfunktion implementiert, wird Volladdierer genannt.

Durch Wiederverwendung bereits implementierter Schaltungen können in der digitalen Schaltungstechnik komplexe Schaltfunktionen effizient und auch kostengünstig realisiert werden. Dies liegt daran, dass eine einfache Schaltung industriell repliziert werden kann. Dies trifft insbesondere in der

Halbleiterindustrie zu, da die Fertigungsprozesse auf die Herstellung vieler gleichartiger Schaltungen optimiert sind. Die in ▶ Kap. 2 entwickelten Schaltfunktionen wirken zunächst einmal komplex. Die minimierten, aus der disjunktiven kanonischen Normalform abgeleiteten Funktionen für die Summe und den Übertrag lauten:

$$s(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3 \quad (5.1)$$

$$c(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \quad (5.2)$$

Die Schaltfunktionen für die Summe und den Übertrag lassen sich umformen:

$$\begin{aligned} s(x_1, x_2, x_3) &= \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3 \\ &= \bar{x}_1 (\bar{x}_2 x_3 + x_2 \bar{x}_3) + x_1 (\bar{x}_2 \bar{x}_3 + x_2 x_3) \\ &= \bar{x}_1 (x_2 \oplus x_3) + x_1 (\bar{x}_2 \oplus \bar{x}_3) = X_1 (\oplus x_2 \oplus x_3) = x_1 \oplus x_2 \oplus x_3 \\ c(x_1, x_2, x_3) &= \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \\ &= x_3 (\bar{x}_1 x_2 + x_1 \bar{x}_2) + x_1 x_2 (\bar{x}_3 + x_3) \\ &= x_3 (x_1 \oplus x_2) + x_1 x_2 \end{aligned}$$

Mit $s(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$ kann eines der XOR-Gatter sowohl zur Berechnung der Summe als auch des Übertrags verwendet werden.

Halbaddierer

Führt man eine Schaltung ein, die mithilfe eines UND- und eines XOR-Gatters die Addition zweier Bits implementiert, erhält man einen Halbaddierer. Zwei dieser Schaltungen hintereinander angeordnet und die jeweiligen Überträge mit einem ODER verknüpft, ergeben den gewünschten Volladdierer. Mit anderen Worten, zwei Bits können mit einem Volladdierer addiert werden. Die Schaltung erzeugt eine Summe und einen Übertrag. Bei n Stellen werden dann n Volladdierer benötigt. Aus dieser Überlegung ergeben sich für jede Stelle die folgenden Schaltfunktionen für Summe und Übertrag des Volladdierers:

$$s(a_i, b_i, c_{i-1}) = a_i \oplus b_i \oplus c_{i-1} \quad (5.3)$$

$$c(a_i, b_i, c_{i-1}) = a_i b_i + c_{i-1} (a_i \oplus b_i) \quad (5.4)$$

Mit einem Halbaddierer $c_h(x_1, x_2) = x_1 x_2$ und $s_h(x_1, x_2) = (x_1 \oplus x_2)$ kann also ein Volladdierer aufgebaut werden. Angenommen, es soll die Summe aus a_i, b_i und c_{i-1} gebildet werden und die Signale am Ausgang des ersten Halbaddierers heißen s' und c' . Dann gilt:

$$s(a_i, b_i, c_{i-1}) = ((a_i \oplus b_i) \oplus c_{i-1}) \quad (5.5)$$

oder

$$s(a_i, b_i, c_{i-1}) = s_{h_1}(a_i, b_i) \oplus c_{i-1} = s_{h_2}(s', c_{i-1}) \quad (5.6)$$

und

$$c(a_i, b_i, c_{i-1}) = a_i b_i + c_{i-1}(a_i, b_i) = c_{h_1}(a_i, b_i) + c_{h_2}(c_{i-1}, c') \quad (5.7)$$

Die Abb. 5.21 zeigt einen aus Halbaddierern aufgebauten 3-Bit-Addierer. Da die Summe einer Stelle i immer erst dann berechnet werden kann, wenn der Übertrag dieser Stelle $i - 1$ bekannt ist, wird diese Schaltung Ripple-Carry-Addierer genannt. Der Übertrag läuft in so einer Schaltung wie eine Welle durch die Stellen; daher der Name. Sind die Operanden a und b jeweils in einem Register gespeichert und soll die Summe s später auch in einem Register gespeichert werden, dann muss der Takt, der die Register ansteuert, die kompletten Durchläufe durch alle Stellen des Addierers berücksichtigen. Ist die Laufzeit einer Stelle gegeben durch $t_i = d_i(XOR) + d_i(UND) + d_i(ODER)$, gilt für die Laufzeit des kompletten Addierers mit n Stellen:

$$t_{rca} = \sum_{i=0}^{n-1} t_i = \sum_{i=0}^{n-1} d_i(XOR) + d_i(UND) + d_i(ODER) \quad (5.8)$$

Deshalb muss die Taktperiode $T \geq t_{rca}$ sein. Die Kette zum Bilden des Übertrags macht die Schaltung langsam. Es stellt sich daher die Frage, ob es nicht schnellere Schaltungen gibt, um die Summe zweier n -stelligen Binärzahlen zu bilden.

5.3.1.2 Paralleles Addierwerk („carry look-ahead adder“)

Eine Analyse der Schaltfunktion zur Bildung des Übertrags ergibt die Substitutionen $a_i b_i = g_i$ und $a_i \oplus b_i = p_i$. Damit vereinfacht sich die Schaltung zu

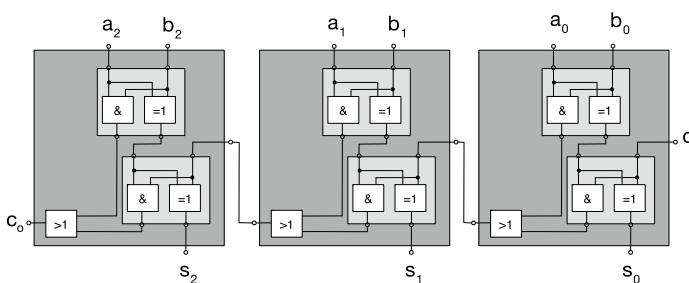


Abb. 5.21 3-Bit-Ripple-carry-Addierer aus Halbaddierern

$$c_{i+1} = g_i + p_i c_i \quad (5.9)$$

Der Ausdruck $g_i = a_i b_i$ berechnet den Übertrag der Stelle i („generate“) und der Ausdruck $p_i = a_i \oplus b_i$ leitet den Übertrag einer niedrigerwertigen Stelle $i - 1$ weiter („propagate“). Die durch Substitution vereinfachte Schaltfunktion kann für jede Stelle der Addition wie folgt umformuliert werden:

$$c_1 = g_0 + p_0 c_0 \quad (5.10)$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_0 p_1 c_0 \quad (5.11)$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2(g_1 + p_1 g_0 + p_0 p_1 c_0) \quad (5.12)$$

$$= g_2 + p_2 g_1 + p_1 p_2 g_0 + p_0 p_1 p_2 c_0 \quad (5.13)$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3(g_2 + p_2 g_1 + p_1 p_2 g_0 + p_0 p_1 p_2 c_0) \quad (5.14)$$

$$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_1 p_2 p_3 g_0 + p_0 p_1 p_2 p_3 c_0 \quad (5.15)$$

5

Wird dieses schrittweise Vorgehen verallgemeinert, lässt sich eine geschlossene Form der Übertragsberechnung für die n -Stelle formulieren:

$$c_n = g_n \sum_{i=0}^n \left(\prod_{j=i}^n p_j \right) g_{i-1} \quad (5.16)$$

Ein solcher Addierer wird carry-look-ahead genannt. Im Gegensatz zu einem Ripple-carry-Addierer werden zur Realisierung eines Carry-look-ahead-Addierers wesentlich mehr Gatter benötigt. Für jede Stelle n müssen n weitere Bauteile verdrahtet werden. Es lässt sich erahnen, welcher Aufwand an Schaltelementen erbracht werden muss, wenn ein 32- oder gar ein 64-Bit-Addierer gebaut werden soll. Es sei erwähnt, dass sich hier ein fundamentales Prinzip beim Entwurf digitaler Schaltungen zeigt: Laufzeit kann gegen Gatter oder Chipfläche getauscht werden. Um die Laufzeit einer Schaltung zu reduzieren und somit die Taktfrequenz eines Rechners zu erhöhen, muss Fläche aufgewendet werden. Laufzeit und Aufwand sind zwei von zahlreichen Parametern zur Beurteilung der Qualität einer Schaltung. Offensichtlich kann die eine Eigenschaft nicht reduziert werden, ohne die andere zu erhöhen.

► Beispiel 5.3.1 – 4-bit-Carry-look-ahead-Addierer

Für einen 4-Bit-Addierer entsteht die in der Abb. 5.22 gezeigte Schaltung. Die Addierschaltung spaltet sich dabei in zwei Teilschaltungen: eine, um die Erzeugungs- und Durchleitungsfunktion zu realisieren, und einen Generator, um die Überträge vorausschauend zu erzeugen („carry look-ahead generator“, CLA). ◀

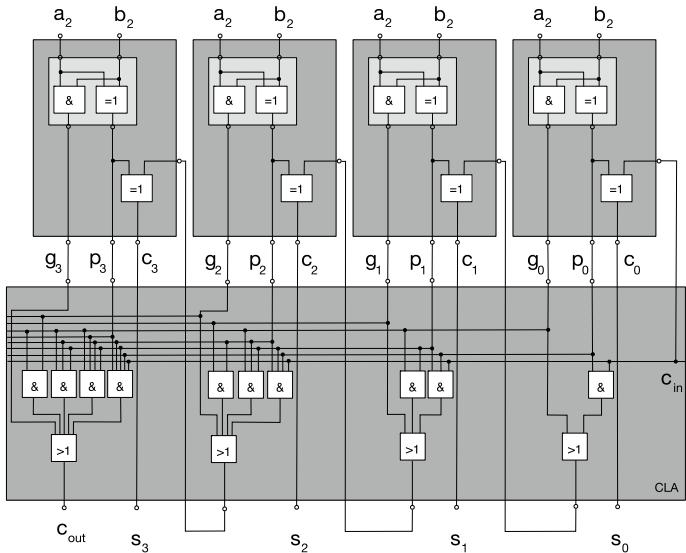


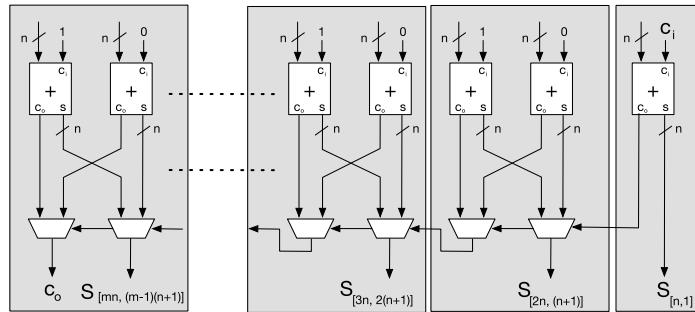
Abb. 5.22 4-Bit-Carry-look-ahead-Addierer

5.3.1.3 Teilparalleles Addierwerk („carry-select adder“)

Durch Kombination der bereits vorgestellten Konzepte lässt sich ein Kompromiss bezüglich des Flächenverbrauches und der Verzögerungszeit finden. Die Idee ist, die Summe zweier Ziffern einmal ohne Berücksichtigung des niedrigstwertigen Übertrags und einmal unter Berücksichtigung dieses Übertrags parallel zu rechnen und erst später festzulegen, welches der beiden Ergebnisse verwendet werden soll. Angenommen, es stehen n Bit breite Carry-look-ahead-Addierer zur Verfügung. Dann kann eine $m \cdot n$ -Bit-Addition mit $2 \cdot m$ Addierern berechnet werden. Dabei wird mithilfe zweier Carry-look-ahead-Addierer die Summe jeder Stelle mit und ohne Übertrag bestimmt. Der Übertrag der vorhergehenden Stelle steuert einen Multiplexer an, und dieser wiederum wählt dann das entsprechende Berechnungsergebnis aus. Wenn ein Übertrag stattgefunden hat, wird die Berechnung mit Übertrag ausgegeben, und wenn es keinen Übertrag gab, entsprechend das Ergebnis ohne Übertrag. Eine solche Schaltung heißt Carry-select-Addierer und ist in der Abb. 5.23 dargestellt.

► Beispiel 5.3.2 – 32-Bit-Carry-select-Addierer

Es soll ein 32-bit-Addierer aufgebaut werden. Dieses Problem kann mit 16 4-Bit-Carry-look-ahead-Addierern gelöst werden. Der Übertrag der jeweils niedrigstwertigen Carry-look-ahead-Addierer



■ Abb. 5.23 Carry-select-Addierer

wird verwendet, um mit einem Multiplexer zu entscheiden, welches der zwei parallel erzeugten Ergebnisse ausgewählt wird. ◀

5.3.2 Addier- und Subtraktionswerk

Eine Subtraktion kann ebenfalls mit einem Addierwerk durchgeführt werden, indem vom Subtrahenden das 2-Komplement gebildet und dieses dann zum Minuenden addiert wird. Ein Addierer kann somit zur Subtraktion genutzt werden, indem ein Operand mithilfe eines Schaltnetzes umcodiert wird: Aus der positiven Zahl wird die entsprechende negative Zahl im 2-Komplement erzeugt. Wird das vor den Addierer vorgeschaltete Schaltnetz mit einer Auswahlleitung o_0 (Operation) versehen, kann entweder die positive Zahl oder die negative Zahl im 2-Komplement an das Addierwerk weitergegeben werden. Die Auswahlleitung schaltet das so entstandene Rechenwerk zwischen Addition und Subtraktion um. Seien a der Minuend und b der Subtrahend, dann ergibt sich für jede Ziffer b_i ein modifizierter, zu b_i komplementärer Eingang b'_i für den Addierer. Mit der Auswahlleitung $o_0 = 0$ für die Addition und $o_0 = 1$ für die Subtraktion ergibt sich die Wahrheitstabelle 5.6.

■ Tab. 5.6 Erweiterung zum Subtrahierwerk

Operation	o_0	b_i	b'_i
+	0	0	0
+	0	1	1
-	1	0	1
-	1	1	0

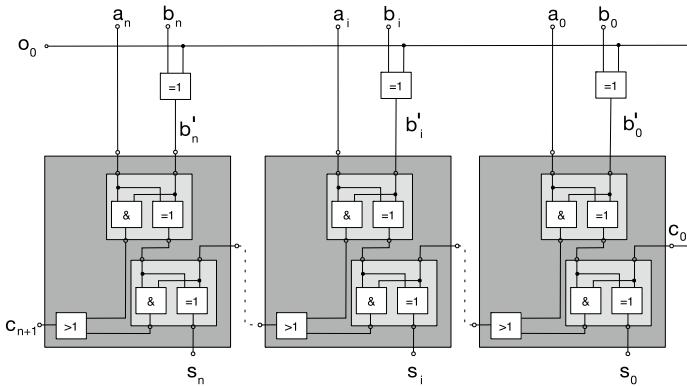


Abb. 5.24 Addier- und Subtraktionswerk

Mithilfe eines vorgeschalteten XOR-Gatters für jede Ziffer kann aus dem Addier- ein Subtraktionswerk gebaut werden. Um mit dem 2-Komplement zu rechnen, muss noch eine 1 addiert werden. Das geschieht durch Anlegen einer 1 an den Übertragseingang c_0 des Addierwerks. Soll addiert werden, muss dort eine 0 anliegen, soll subtrahiert werden, eine 1. Somit kann die Auswahlleitung o_0 direkt an c_0 angeschlossen werden. Selbstverständlich kann als Addierer ein Ripple-Carry-, ein Carry-look-ahead-Addierer oder auch ein Carry-select-Addierer verwendet werden (Abb. 5.24).

5.3.3 Arithmetisch-logische Einheit (ALU)

Einem frei programmierbaren Rechner sollte es möglich sein, neben den byte- oder wortweiten arithmetischen Operationen auch byte- oder wortweise logische Verknüpfungen zu bilden. Das Addier- und Subtrahierwerk soll daher zu einer arithmetisch-logischen Einheit (engl. „arithmetic logic unit“, Akronym ALU) mit den logischen Elementen UND, ODER und Komplement (NICHT) erweitert werden. Eine solche Zusammenführung ist ökonomisch, da in einer arithmetisch-logischen Einheit immer nur eine Operation gleichzeitig ausgeführt werden kann, die unterschiedlichen Verknüpfungen aber Schaltungsteile anderer Operationen mit verwenden können.

Um drei Operationen zu realisieren, wird eine weitere Steuerleitung o_1 eingeführt. Da sich mit 2 Steuerleitungen 4 Funktionen ausführen lassen, wird noch die logische Identität hinzugefügt. Neben den logischen Operationen soll die ALU auch um exakt 1 addieren und subtrahieren können, um so mit der ALU einen Zähler realisieren zu können. Mit einer Leitung A/\bar{L} soll zwischen dem arithmetischen und dem logischen Mo-

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

a_i b_i

Abb. 5.25 Logikminimierung der logischen Erweiterung

dus umgeschaltet werden. Dazu soll der Eingang a des Addier- und Subtraktionswerks wie in Abb. 5.25 gezeigt erweitert werden.

Aus dieser Wertetafel lässt sich folgende Schaltfunktion ableiten: $b'_i = \overline{o_0}(o_1 a_i + o_1 b_i) + o_0(o_0 \overline{a_i} \overline{b_i} + \overline{o_1} a_i b_i)$.

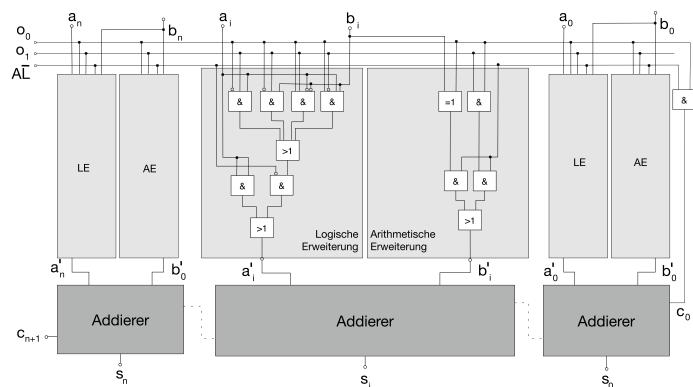
Durch Einführung einer zusätzlichen Steuerleitung kann die ALU im arithmetischen Modus noch um zwei weitere Funktionen erweitert werden: der Addition um 1 und der Subtraktion um 1. Es ergibt sich die Wertetafel in Tab. 5.7.

Die beiden zusätzlichen Schaltungen heißen logische (LE) und arithmetische Erweiterung (AE). Mit ihnen ist es möglich, mit geringem Aufwand den Addierer zu einer vollen ALU zu erweitern (Abb. 5.26).

5

■ Tab. 5.7 Arithmetische und logische Erweiterung des Addierwerks

Modus	A/L Operation	A/L			Modus	A/L Operation	A/L		
		$o_1 o_0$	$a_i b_i$	$a'_i b_i$			$o_1 o_0$	$a_i b_i$	$a'_i b_i$
0	a	00	00	00	1	a+1	00	00	00
0		00	01	00	1	a+1	00	01	00
0		00	10	00	1	a+1	00	10	10
0		00	11	10	1	a+1	00	11	10
0	a & b	01	00	00	1	a+b	01	00	00
0		01	01	00			01	01	01
0		01	10	00			01	10	10
0		01	11	10			01	11	11
0	a b	10	00	00	1	a-b	10	00	00
0		10	01	10			10	01	01
0		10	10	10			10	10	11
0		10	11	10			10	11	10
0	\bar{a}	11	00	10	1	a-1	11	00	01
0		11	01	10			11	01	01
0		11	10	00	1	a-1	11	10	11
0		11	11	10	1	a-1	11	11	11



■ Abb. 5.26 ALU: aufgebaut aus Addierern, logischen und arithmetischen Erweiterungen

5.4 Speicher

5.4.1 Stapel

Ein Stapel oder Stapspeicher ist ein Speicher, der nach dem LiFo-Prinzip arbeitet (engl. „last in, first out“). Das heißt, das Datum, das zuletzt im Speicher abgelegt wurde, wird als Erstes wieder gelesen – genau wie bei einem Bücher- oder Aktenstapel, von dem auch immer nur das Schriftstück entnommen werden kann, das als Letztes auf den Stapel gelegt wurde. Bei einem derartigen Konzept zur Adressierung des Speichers kann nicht auf jeden Eintrag beliebig zugegriffen werden. Die Speicherstellen der Speichermatrix sind also nicht frei adressierbar. Ein Datum wird auf dem Stapel mit dem Signal push (engl. für ablegen) gespeichert und mit pop (engl. für herunterholen) gelesen und dann gleichzeitig gelöscht (Abb. 5.27).

Die Steuerlogik für die Speichermatrix eines Staps kann mit wenig technischem Aufwand mit einem Rechts-/Links-Schieberegister aufgebaut werden. Dieses Schieberegister sollte ein Flipflop mehr aufweisen, als es Zeilenauswahl oder Adressleitungen in der Speichermatrix gibt. Nach einem Clear-Signal werden fast alle Flipflops des Schieberegisters auf null gesetzt. Das mit keiner Adressleitung verbundene Flipflop da-

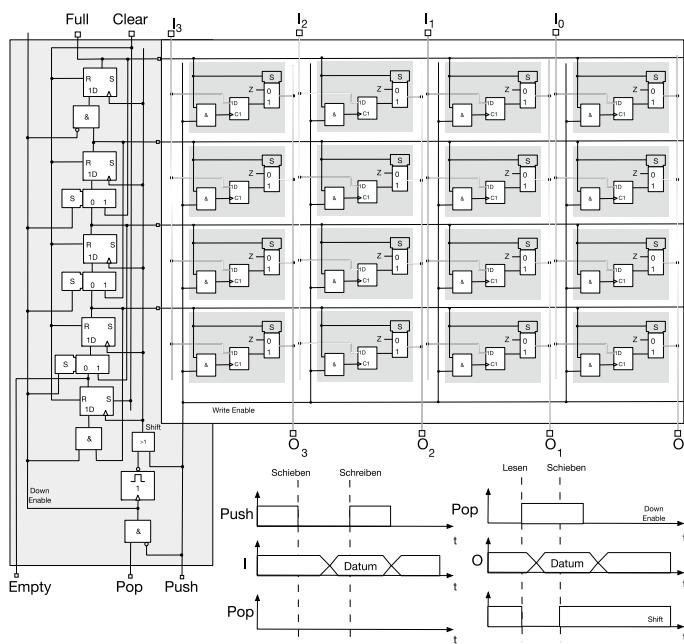


Abb. 5.27 Stapelspeicher mit Schieberegister

gegen wird auf logisch 1 gelegt. Damit zeigt dieses Flipflop an, dass der Speicher leer ist und keine gültigen Daten gelesen werden können. Mit einer fallenden Flanke auf der Push-Leitung soll das FiFo beschrieben werden. Dazu muss erst einmal die 1 aus dem untersten Flipflop nach oben geschoben werden. Danach kann mit der steigenden Flanke auf der Leitung push ein Datum übernommen werden. Push ist also negativ aktiv. Alternativ kann eine Leitung für push/pop vorgesehen werden, dann wird aber noch ein Signal enable benötigt, um anzugeben, dass gültige Daten am Eingang des Stacks anliegen. Auf diese Art wird vor jedem Schreiben zuerst die 1 im Schieberegister nach oben geschoben und das Datum an einen freien Speicherplatz geschrieben, zeigt der Speicher dies mit dem Signal full an. Die 1 im Schieberegister steht also automatisch an der Stelle, an der das nächste Datum gelesen werden kann.

Ein Datum wird auf dem Stapel abgelegt (push), indem zunächst der Zeiger erhöht wird, das Bit im Schieberegister also nach oben geschoben wird. Anschließend wird das Datenwort in den Speicher geschrieben. Die steigende Flanke des Signals push schiebt zunächst das Bit, und mit der fallenden Flanke werden die Daten übernommen. Ist der Stapel voll, wird das letzte Bit als Anzeige des Füllzustands verwendet (Abb. 5.28).

Beim Lesen des Speichers wird zunächst das Datum gelesen. Das Signal pop dient dabei der Umkehr der Richtung des Schieberegisters. Wenn geschoben werden soll, dann soll das Adresszeigerbit wieder nach unten verschoben werden. Um nach dem Lesen das Schieben durchzuführen, wird ein Monoflop eingesetzt. Dieses wird gesetzt, und auf dessen fallender Flanke wird das Bit im Schieberegister um eine Stelle versetzt neu zugewiesen. Nachdem der Zeiger auf die Lesezeile verschoben wurde, ist das Datum in dieser Zelle nicht mehr gültig (Abb. 5.29).

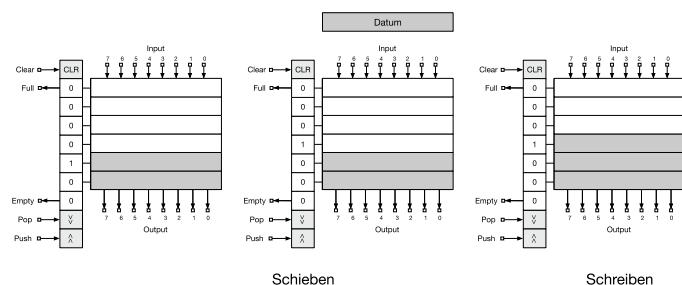


Abb. 5.28 Stackspeicher beschreiben

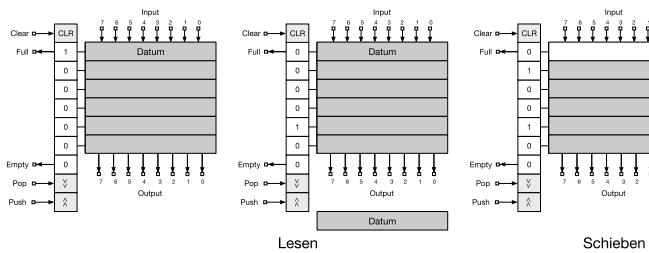


Abb. 5.29 Stapelspeicher auslesen

Statt eines Schieberegisters kann ein Stapel auch mit einem Zähler implementiert werden. Da die Zustände jedoch nun codiert sind, wird noch zusätzlich ein Decoder benötigt, der die Zählerstände wieder in eine One-hot-Codierung umwandelt. In dem Fall vereinfacht sich die Schaltung: Push und pop können durch ein Signal abgebildet werden, und die Logik zur Ansteuerung der Richtung des Schieberegisters entfällt. Es wird nun aber ein Enable-Signal benötigt, um für das Beschreiben des Speichers zu sorgen. Zwar ist die Schaltung einfacher, aber das richtige Zugriffsverhalten, d. h. die korrekte Abfolge der Eingangssignale, muss außerhalb des Stapeles realisiert werden.

5.4.2 Warteschlange

Oft müssen Daten zwischengespeichert werden. Insbesondere in Signal verarbeitenden Systemen, beim Transportieren digitaler Sprache oder Musik werden die Informationen dabei in der Reihenfolge, in der sie ankommen, auch wieder abgegeben. Das erste Datum wird geschrieben (engl. „first in“) und als Erstes gelesen (engl. „first out“). Da ein solcher Speicher einer Schlange auf dem Postamt oder beim Bäcker entspricht, nennt man eine solche Schaltung Warteschlange. Eine solche Warteschlange heißt auch FiFo-Puffer oder -Speicher (Abb. 5.30). Ein solcher Speicher kann ähnlich wie der Stapelespeicher mit Schieberegistern zur einfachen Adressierung der Speicherstellen und einer Speichermatrix realisiert werden. Im Gegensatz zum Stapele werden bei dieser Variante 2 Schieberegister benötigt, die jeweils nur in eine Richtung schieben. Ein Schieberegister repräsentiert also immer einen Zeiger. Mit 2 Schieberegistern werden daher Schreib- und Lesezeiger des Speichers getrennt implementiert.

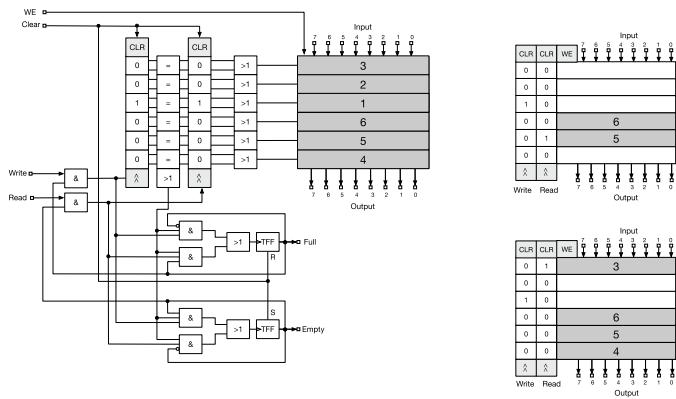


Abb. 5.30 Aufbau eines FiFo-Speichers

Angenommen, das FiFo ist leer. In dem Fall zeigen beide Schieberegister auf die gleiche Speicherzeile. Soll nun ein Datum geschrieben werden, muss zunächst einmal das Bit im Schreibschieberegister weitergeschoben werden. Anschließend können dann die Werte übernommen werden. Die Übernahme der Daten erfolgt bei einer positiven Flanke auf der Leitung WE („write enable“). Da das letzte Flipflop des Schieberegisters mit dem ersten Flipflop verbunden ist, wird die 1 von der letzten zur ersten Stelle weitergeschoben. Erst wenn der Schreib- und der Lesezeiger gleich sind, ist das FiFo voll. Dieser Zustand wird nach außen signalisiert, und dadurch wird der Schreibzeiger blockiert, sodass beim ersten neuen Schreiben nicht weitergeschoben wird. Erst wenn wieder Daten gelesen werden, wird die Verriegelung aufgehoben, und der Zeiger kann ein weiteres Mal beim Schreiben bewegt werden.

Das Lesen der Warteschlange kann nur erfolgen, wenn Daten im FiFo gespeichert sind. Nachdem ein Wort aus dem Speicher gelesen wurde, wird der Lesezeiger weitergeschoben, und das auch so lange, bis beide Zeiger wieder gleich sind. Wird die Gleichheit der Zeiger durch einen lesenden Zugriff erreicht, wird das Signal `empty` gesetzt. Dies bleibt so lange stehen, bis beide Zeiger ungleich sind, wenn also wieder neue Daten in den Speicher geschrieben worden sind.

5.4.3 Speicher mit direktem Zugriff

Die Zugriffsmöglichkeiten auf Stapel und Warteschlange sind begrenzt. Bedingt durch ihre einfache technische Realisierbarkeit stellen diese beiden Arten Daten abzulegen eine kurze

Zeit lang die einzige Möglichkeit dar, Informationen dynamisch zu verwalten. Es zeigte sich schnell, dass die begrenzten Zugriffsmöglichkeiten die Anwendungsprogrammierung eines Rechners erschweren. Aus diesem Grund wurde bereits früh nach effizienten Techniken für einen Speicher mit wahlfreiem Zugriff gesucht. Ein Direktzugriffsspeicher besteht aus mehreren Speichermatrizen und entsprechender Adressdecodierlogik (Abb. 5.31). Diese decodiert die Adresse, die an der Datenschnittstelle des Speichers anliegen muss und vom Prozessor bereitgestellt wird. Die Adresse wird dabei in eine One-hot-Codierung umgewandelt, um jede Zeile der Matrix einzeln anzusprechen. Aus schaltungstechnischer Sicht führen große Speichermatrizen zu langen Laufzeiten beim Lesen und Schreiben. Aus diesem Grund werden Speicher in Bänke unterteilt. Es werden also mehrere Speichermatrizen parallel angeordnet. Zwar wird für die Verwaltung der Bänke Logik benötigt, aber durch die parallele Anordnung der Matrizen lässt sich die Geometrie des Speicherchips ebenfalls optimieren. Eine solche Struktur braucht daher einen Auslesemultiplexer, um die entsprechenden Daten auf den Ausgang zu legen. Dazu wird bei einem SRAM-Speicher die Adresse in einen Spalten- und einen Zeilenanteil aufgespalten. Da SRAM-Zellen relativ groß sind, wird die Fläche des SRAM-Speichers verhältnismäßig klein gehalten. Dementsprechend wird eine geringere Anzahl an Adressworten benötigt als bei einem DRAM. Die Adresswortanzahl kann bei einem DRAM so groß werden, dass nicht genug Eingangspins am Gehäuse für die Adresse zur Verfügung stehen. Daher wird die Adresse in eine Zeilen- und eine Spaltenadresse aufgeteilt und mit einem Zeitmultiplexverfahren an den Baustein angelegt – zunächst die Zeilenadresse und dann die Spaltenadresse. Dieses Zeitmultiplexverfahren wird mithilfe der zwei Steuerleitungen RAS und CAS rea-

SRAM

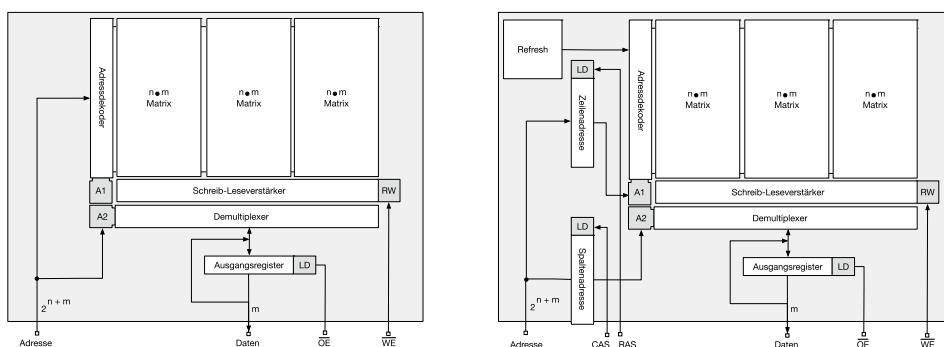


Abb. 5.31 SRAM- und DRAM-Direktzugriffsspeicher

DRAM

lisiert. Über einen Schreib-/Lese-Verstärker können die Daten in den Speicher geschrieben oder aus dem Speicher gelesen werden. Der DRAM-Baustein benötigt zusätzlich noch eine Auffrischlogik („refresh logic“). Bedingt durch die Bauart der Zellen und die bereits vorhandenen Schreib-/Lese-Verstärker reicht es, die DRAM-Zellen zyklisch auszulesen. Dazu müssen lediglich ein eigener Adresszähler und eine kleine Auswahllogik auf dem Speicherchip implementiert werden.

Lesen und Schreiben eines SRAM ist einfach. Wird die an den Baustein angeschlossene Adressleitung gesetzt, kann kurze Zeit später das entsprechende gewünschte Datum am Ausgang der Speichermatrix ausgelesen werden. Beim Schreiben werden zunächst die Adresse und ein gültiges Datenwort am Speicher angelegt, und mithilfe des Signals write enable werden dann die Daten übernommen. Adress- und Datenleitungen können einmal pro Zykluszeit verändert werden, und so können Daten direkt und ohne Zeitverzögerung gelesen und geschrieben werden (Abb. 5.32).

Leseprotokoll

Das Zugriffsprotokoll eines DRAM-Bausteins ist wie folgt spezifiziert: Ein Lesezyklus beginnt mit einer fallenden Flanke auf der RAS-Leitung. Zu diesem Zeitpunkt muss an der Adressleitung eine gültige Adresse anliegen (Zeilenadresse). Nachdem der Speicherbaustein mit der fallenden Flanke von RAS diese gelesen hat, kann eine Spaltenadresse an den Adresseingang angelegt werden. Die Übernahme der Zeilenadresse erfolgt dann ebenfalls mit fallender Flanke, aber der CAS-Leitung. Gleichzeitig wird die Leitung WE gesetzt. Da WE negativ aktiv ist, bedeutet ein Setzen des Signals Lesen. Nach einer festen Latenzzeit, die von der Implementierung und dem Refresh-Protokoll des Speichers abhängt, können die Daten am Ausgang abgegriffen werden. Bedingt durch die Vorspannphase und die möglichen Refresh-Zeiten sind DRAMs im Vergleich zu SRAMs langsam. Da aber häufig Daten im Block transportiert werden müssen, lassen sich diese auch blockweise aus einem DRAM auslesen. Die Datenworte werden in einem

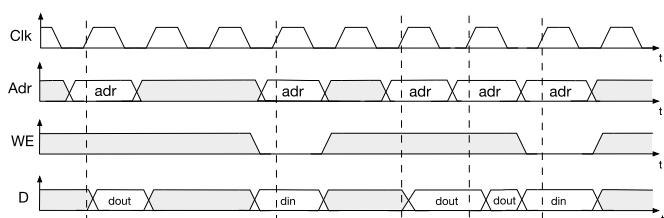


Abb. 5.32 SRAM-Zugriffsprotokoll

Schub (engl. „burst“) vom Speicher zur Verfügung gestellt. Das ist möglich, da der Baustein in der Lesezeit keinen Refresh auf die entsprechenden Zellen ausführen muss und die Vorspannphase für alle Spalten gleichzeitig durchgeführt werden kann. Bei einem blockweisen Auslesen wird an den Adressleitungen die Spaltenadresse hochgezählt, und der Pegel auf der CAS-Leitung wechselt zyklisch. Die Zeilenanwahl ändert sich also nicht. Ausgelesen werden somit nur die Spaltenblöcke. Bei jeder fallenden Flanke von CAS können dann Daten gelesen werden. Es gibt ebenfalls Bausteine, bei denen die Daten sowohl bei einer fallenden als auch einer steigenden Flanke von CAS gelesen werden können. Dazu muss sich das Signal auf der Adressleitung entsprechend schnell ändern (Abb. 5.33).

Der Schreibzyklus eines DRAM-Speichers unterscheidet sich vom Lesezyklus insofern, dass die Leitung WE negativ ist und die Daten mit fallender Flanke von WE von der Datenschnittstelle in den Speicher geschrieben werden. Der Wert der Leitung WE ist dabei unerheblich. Selbstverständlich können die Daten auch schubweise geschrieben werden (Abb. 5.34).

ROM-Speicher werden wie SRAM-Speicher gelesen. Das Beschreiben eines ROM-Speichers entfällt natürlich.

Neben den RAMs und den ROMs spielt inzwischen eine weitere Speichertechnologie eine große Rolle: die Flash-RAMs. Flash-Speicher sind aus EEPROM-Speicherzellen aufgebaut. Eine EEPROM-Zelle muss immer gelöscht werden, bevor sie wieder beschrieben werden kann. Das macht diese Speichertechnik langsam, verglichen mit SRAM oder DRAM-Bausteinen. Auf der anderen Seite kann mit EEPROMs gut ein ROM für einen Programmspeicher ersetzt werden. Dieser wird normalerweise nur einmal geschrieben und dann im

Datenburst

Schreibprotokoll

Flash-Speicher

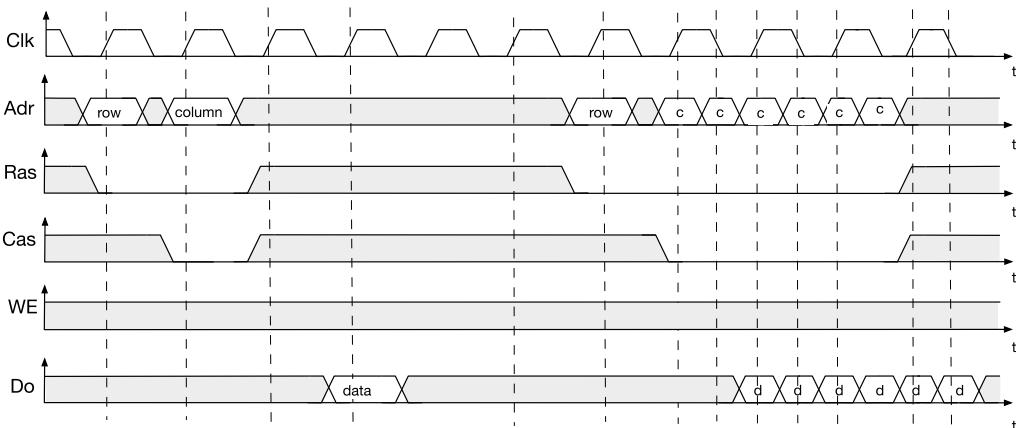


Abb. 5.33 Auslesen von Daten aus einem DRAM

Betrieb nicht mehr verändert. Der Vorteil gegenüber einem ROM ist aber, dass ein Rechner im Feld umprogrammiert werden kann. Typisches Einsatzgebiet für EEPROMs waren daher eingebettete Systeme. Den Geschwindigkeitsnachteil der EEPROMs konnte die Firma Toshiba in den 1980er-Jahren verringern. Diese Technologie ermöglichte ein blockweises Löschen des EEPROM. Im Wesentlichen bedeutet dies, dass Zeit gewonnen wird, da ja nicht immer alle Inhalte zu löschen sind. Ein solcher Speicher heißt Flash-EEPROM oder kurz Flash-Speicher. Chips dieses Typs eignen sich hervorragend um nicht flüchtige Speicher in Festkörpern aufzubauen. Mit zunehmender Integrationsdichte bei fallenden Herstellungskosten verdrängen derartige Bausteine inzwischen die magnetisch basierten Festplatten.

5

Mehrtorspeicher

Auf die Daten in einem Direktzugriffspeicher kann beliebig zugegriffen werden. Die Schnittstelle zu anderen Komponenten bilden die Adress- und die Datenleitungen, das Tor (engl. „port“). Es ist möglich, den Zugriff auf die Speichermatrix über mehrere solcher Tore durchzuführen (Mehrtorspeicher). Dazu müssen weitere Schreib- und Leseleitungen für jedes Tor jeweils eingefügt werden. Hinzu kommt ein eigener Adressdecoder für jedes Tor. Um zu verhindern, dass zwei Tore gleichzeitig in eine Zelle schreiben, wird eine Zugriffsverwaltung (Arbiter, engl. für Schiedsmann oder -richter, lat. für Richter) benötigt. Diese Schaltung bestimmt im Falle eines Konfliktes, welches Tor auf den Speicher zugreifen kann. Der Arbiter prüft für jede Zelle, ob ein solches Problem vorliegt. In Hardware kann dies mit einem XOR-Gatter einfach realisiert werden. Im Falle eines Konfliktes wird dieser nach außen signalisiert und anschließend aufgelöst. Im Beispiel wird

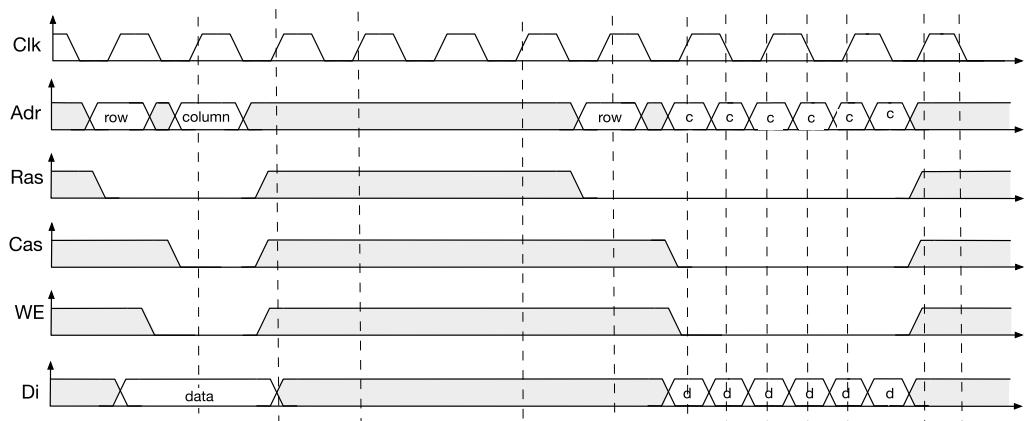


Abb. 5.34 Signalverlauf bei einem DRAM-Schreibzugriff

immer Tor 1 der Vorzug gegeben. Es ist aber auch möglich, die Schaltung so zu erweitern, dass eine Konfliktauflösung gewählt werden kann. Konflikte können in manchen Implementierungen direkt intern gehandhabt werden. Dann ist für jedes Tor ein eigenes Eingangsregister vorzusehen. Im Konfliktfall wird ein Datum dort gespeichert und erst geschrieben, wenn der Zugriff auf die entsprechende Zelle wieder frei ist. Das Lesen der Speicherzelle ist so konfliktfrei möglich. In dem Fall wird einfach der Inhalt auf die Leseleitung beider Ports gegeben (Abb. 5.35).

Das Prinzip lässt sich zu einem N -Tor-Speicher erweitern. Die Anzahl der Tore ist nur durch die zur Verfügung stehenden Ressourcen der Chipfläche oder ggf. der Zugriffsleitungen des Chips beschränkt. Mehrtorspeicher werden häufig in eingebetteten Systemen zum Puffern von schnell anfallenden Daten verwendet, und mit 3-Tor-Speichern lassen sich ebenfalls gut die internen Registerbänke von Prozessoren realisieren. Die Abb. 5.36 zeigt das Schaltsymbol eines 2-Tor-Speichers mit den 2 Adressmultiplexern und den beiden symbolischen Bereichen, auf die gleichzeitig geschrieben und gelesen werden kann. Natürlich besitzt ein derartiger Speicher nur eine Speichermatrix, und beide Ein-/Ausgänge können auf jede der Speicherzellen beliebig zugreifen.

5.4.4 Assoziativspeicher

Beim Assoziativspeicher (engl. „content-addressable memory“, CAM) handelt es sich um einen Speicher, bei dem das Zugriffsprinzip umgekehrt wird. Das bedeutet, statt einer Adresse wird an den Baustein ein Datum angelegt, und der Speicher selbst liefert am Ende eine Adresse. Deshalb wird der Assozia-

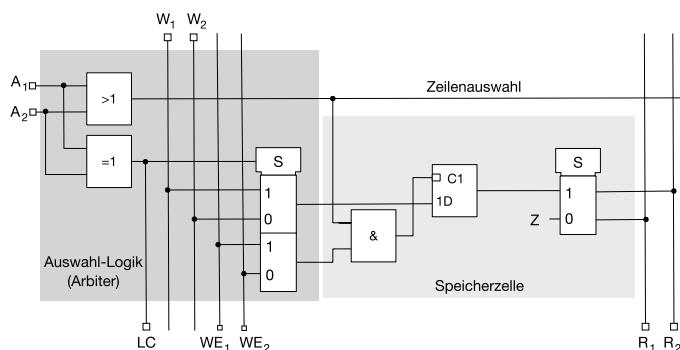


Abb. 5.35 Modell einer Speicherzelle eines 2-Tor-Speichers

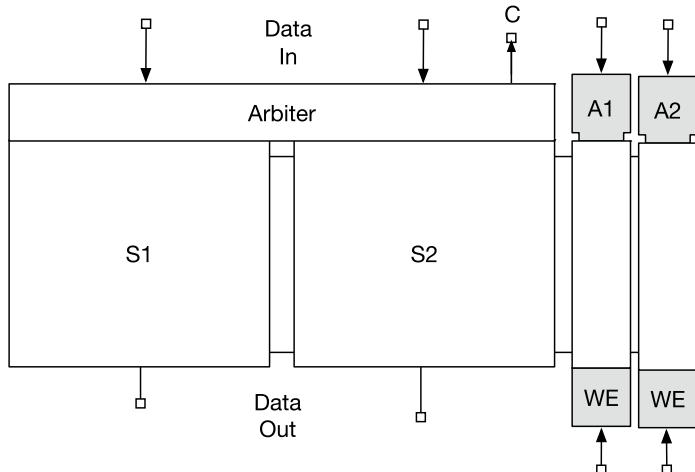


Abb. 5.36 Ein 2-Tor-Speicher mit wahlfreiem Zugriff

tivspeicher häufig auch als inhaltsadressierbarer Speicher bezeichnet. Diese Speicherform wurde 1943 bereits von Konrad Zuse beschrieben. Da die Realisierung sehr aufwendig ist und viele parallele Vergleichsoperationen benötigt werden, wurden Assoziativspeicher erst spät in Halbleitertechnik realisiert. Die Abb. 5.37 zeigt den inneren Aufbau eines Assoziativspeichers.

Jede einzelne Speicherzelle ist mit einem XOR-Gatter verbunden, und der zweite Anschluss aller UND-Gatter einer Speicherreihe wird mit einer Zelle eines Eingangsregisters verschaltet. Die Ausgänge aller UND-Gatter einer Zeile wiederum müssen dann wieder UND-verknüpft werden, um ein Signal auf der Zeilenleitung zu generieren. Über einen Adresscodierer kann anschließend das richtige Adresswort erzeugt werden.

Ein CAM-Speicher besteht aus zwei Speichermatrizen – genau gesagt, einer konventionellen SRAM-Matrix, in der beliebige Daten geschrieben werden können, und einer zweiten Matrix, in der die Adressen abgelegt sind. Wird ein Datum an den Eingang eines CAM angelegt, wird das Bitmuster automatisch von der Hardware mit allen in der Matrix gespeicherten Bitmustern verglichen. Dafür ist für jede Speicherzelle ein XOR-Gatter vorgesehen. Die Ausgänge alle XOR-Gatter einer Zeile werden im Anschluss UND-verknüpft. Auf diese Weise entsteht eine one-hot-codierte Zeilenadresse für die zweite Matrix. In dieser speziellen Speichermatrix können wieder beliebige Datenworte in SRAM-Zellen gespeichert werden. Wenn also alle Bits des Indexes gleich allen gespeicherten Bits sind, wird ein Match signalisiert. Da die Matchleitung als Adresse

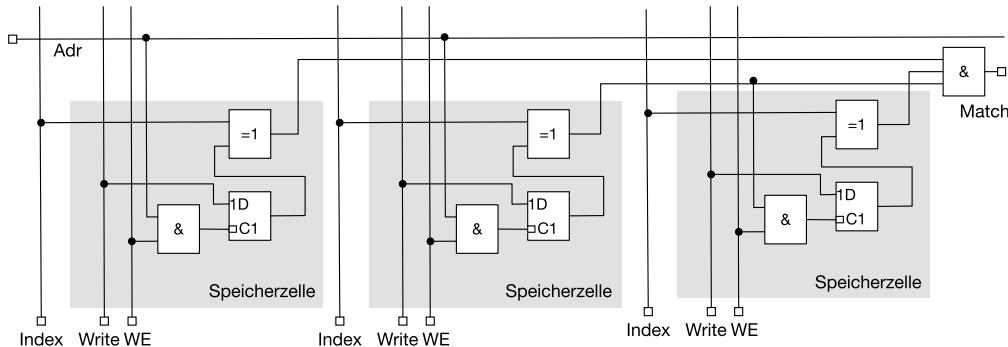


Abb. 5.37 Funktionsmodell eines CAM-Speichers

der zweiten Speichermatrix dient, kann jetzt ein Datum ausgegeben werden, das unter diesem Index zu finden sein soll (Abb. 5.38).

Überall, wo eine Information in ein anderes Datum übersetzt werden soll, kommen assoziativ oder inhaltsadressierbare Speicher zum Einsatz; beispielsweise zur Adressauflösung oder Umsetzung von Netzwerkadressen in physikalische Adressen in Netzwerkroutrern oder in versteckten Zwischenspeichern (Caches).

Übungsaufgaben

Aufgabe 5.1 Nennen Sie 2 Vorteile des Gray-Codes.

Aufgabe 5.2 Die nachfolgende Tabelle zeigt einen Umcodierer für Gray-Code zu 7-Segment-Anzeige. Der Gray-Code ist 4 Bit breit und auf die Dezimalzahlen 01_0 – 91_0 beschränkt. Für alle weiteren Binärkombinationen sollen die Elemente der 7-Segment-Anzeige *nicht* leuchten.

- Stellen Sie die Wahrheitstabelle unter zu Hilfenahme von Tab. 5.2 und Tab. 2.4 auf.
- Geben Sie für die Funktion $y = A(x_0, x_1, x_2, x_3)$ aus der Wahrheitstabelle in alle Primimplikanten nachvollziehbar an.
- Geben Sie die minimierte DNF der Funktion $y = B(x_0, x_1, x_2, x_3)$ aus der Wahrheitstabelle mithilfe des Nelson-Verfahrens nachvollziehbar an.
- Geben Sie die Funktion $y = E(x_0, x_1, x_2, x_3)$ aus der Wahrheitstabelle in disjunktiver kanonischer Normalform (DKNF) an.
- Nennen Sie 2 Vorteile des Gray-Codes.

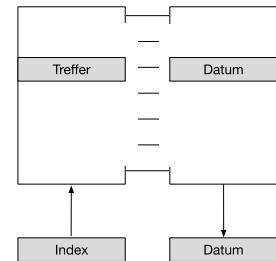


Abb. 5.38 CAM-Speicher

Aufgabe 5.3 Entwerfen Sie einen Dezimal-zu-Aiken-Code-Umcodierer mithilfe der folgenden Wertetafel:

Dezimal <i>d</i>	Binär				Aiken-Code			
	<i>x</i> ₁	<i>x</i> ₂	<i>x</i> ₃	<i>x</i> ₄	<i>y</i> ₁	<i>y</i> ₂	<i>y</i> ₃	<i>y</i> ₄
0	0	0	0	0				
1	0	0	0	1				
2	0	0	1	0	0	0	1	0
3	0	0	1	1				
4	0	1	0	0				
5	0	1	0	1				
6	0	1	1	0				
7	0	1	1	1	1	1	0	1
8	1	0	0	0				
9	1	0	0	1				
10	1	0	1	0				
11	1	0	1	1				
12	1	1	0	0				
13	1	1	0	1				
14	1	1	1	0				
15	1	1	1	1				

- 5
- a) Vervollständigen Sie die Wertetabelle zur Codierung einer binär codierten Dezimalzahl im Aiken-Code. Achten Sie auf die Einschränkungen dieses Codes.
 - b) Geben Sie für y_1, y_2, y_3 und y_4 jeweils die DKNF an.
 - c) Minimieren Sie die Funktionen mithilfe von KV-Diagrammen.
 - d) Auch XOR und UND stellen zusammen eine vollständige Basis dar. Stellen Sie die minimierten Funktionen aus Teilaufgabe c nur unter der Verwendung von XOR und UND dar. Zeichnen Sie die Funktionen anschließend als Gatterschaltung. Verwenden Sie dabei ebenfalls nur XOR- und UND-Gatter.

Aufgabe 5.4 Erstellen Sie ein 4-Bit-serial-in-serial-out-Schieberegister mit ausschließlich D-Flipflops.

Aufgabe 5.5 In dieser Aufgabe erstellen Sie einen Ripple-Carry-Addierer.

- a) Erstellen Sie zunächst eine Wertetabelle für einen Halbaddierer und minimieren Sie dessen Schaltfunktionen, so dass nur 2 Gatter benötigt werden. Skizzieren Sie die Schaltfunktion des Halbaddierers auf Gatterebene.
- b) Skizzieren Sie nun mithilfe des Halbaddierers als Komponente die Schaltfunktion eines Volladdierers.
- c) Zeichnen Sie nun unter Verwendung des Volladdierers als Komponente die Schaltfunktion eines 4-bit-Ripple-Carry-Addierers auf Komponentenebene.
- d) Erweitern Sie den 4-bit-Ripple-Carry-Addierer um die Funktion des Subtrahierens. Der Addierer soll also $a + b$

und $a - b$ berechnen können. Dafür benötigen Sie einen weiteren Eingang, der zwischen beiden Rechenarten unterscheidet.

- e) Welche Vor- und Nachteile hat der Carry-select-Addierer gegenüber dem Ripple-Carry-Addierer?

Weiterführende Literatur

- [Wal80] Klaus Waldschmidt. *Schaltungen der Datenverarbeitung*. Vieweg+ Teubner Verlag, 1980. ISBN: 9783519061083.
- [Sta13a] William Stallings. *Computer Organization and Architecture: Designing for Performance*. 10. Aufl. Piercon Education Ltd., 2013. ISBN: 9781292096858.



Rechnerarchitektur

Inhaltsverzeichnis

6.1 Architektur des Befehlssatzes – 337

6.2 Speichermodell – 374

6.3 Mikroarchitektur – 398

Weiterführende Literatur – 437

Architektur ist Harmonie und Einklang aller Teile, die so erreicht wird, dass nichts weggemommen, zugefügt oder verändert werden könnte, ohne das Ganze zu zerstören.

Leon Battista Alberti

Im vorletzten Kapitel werden die Bausteine der Rechnerarchitektur zusammengefügt. Nach dem Durcharbeiten kann der Leser verschiedene Architekturkonzepte und den Unterschied zwischen Stapel- und Registermaschinen erläutern. Insbesondere die Verknüpfung der technischen Grundlagen mit den Konzepten der Rechnerarchitektur zu verstehen ist das Ziel. Der heutige Aufbau elektronischer Rechner und allgemeine Entwurfsprinzipien wie die Trennung von Rechen- und Steuerwerk können skizziert werden. Darüber hinaus sind dem Leser unterschiedliche Speicherarchitekturen und die Notwendigkeit versteckter Zwischenspeicher (Caches) bekannt. Er beherrscht somit die Grundlagen, die zum Verständnis moderner Betriebssysteme notwendig sind. Aufbauend auf diesen Konzepten wird die Basis des maschinennahen Programmierens gelegt. Leser sollten nach der Lektüre in der Lage sein, Befehlssätze zu verstehen und anzuwenden. Insbesondere die unterschiedlichen Konzepte der Operanden- und Speicheradressierung als Grundlage des Verständnisses der Assembler-Programmierung sind bekannt und können erläutert werden.

Der Aufbau eines Computers unterliegt bestimmten Prinzipien. Im Laufe der Zeit haben sich diese zu festen Merkmalen verfeinert. Dabei wird von der schaltungstechnischen Implementierung abstrahiert. Die Merkmale der Rechnerarchitektur werden zwar stark von der Schaltungs- und Herstellungstechnik beeinflusst, lassen sich aber auch vollständig ohne Kenntnisse dieser beschreiben. Lediglich die Frage nach dem Warum eines Architekturmerkmals führt zurück zur Schaltungstechnik.

Eigenschaften und Verhalten verschiedener Maschinen erlauben es, diese zu Gruppen zusammenzufassen und zu klassifizieren. In der Geschichte der Menschheit prägen stilbildende Bauwerke Epochen. Im gleichen Sinne bestimmen feste Entwurfsmuster oder -stile den Aufbau von Rechnern. Diese Stile korrelieren mit der historischen Abfolge konkret gebauter Computer. Inzwischen existieren viele verschiedene Erscheinungsformen von Rechenmaschinen, die sich je nach Einsatzzweck unterscheiden. Wie in der Baukunst oder Architektur von Bauwerken ist die Erscheinungsform eines Rechners geprägt durch die verwendeten Baumaterialien. Sind dies beim Bau von Gebäuden Holz, Stein, Glas und andere Materialien, ist auch die Konstruktion von Computern beeinflusst durch die zur Verfügung stehende Schaltungstechnik, deren Grundelemente wie Relais, Röhren, Dioden, Bipolar- und Feldeffekt-

transistoren und die dazugehörigen Herstellungstechniken¹. Und ähnlich wie in der Architektur existieren in der maschinellen Datenverarbeitung universelle Baugruppen. Den elementaren Formen Mauern, Bögen, Dächer, Türme und Portale der Baukunst stehen in der Rechnerarchitektur die Teile Befehlstypen, Register- und Speicheradressierung, Stackspeicher und Fließbandverarbeitung – um nur einige Prinzipien zu nennen – gegenüber. Diese Prinzipien sind unabhängig von der Implementierung durch eine spezielle Schaltungstechnik, können jedoch nicht alle mit jeder Schaltungstechnik aufgebaut werden. Wie der italienische Gelehrte und Architekt Leon Battista Alberti (1404–1472) das Wesen der Architektur als

- » Harmonie und Einklang aller Teile, die so erreicht wird, dass nichts weggenommen, zugefügt oder verändert werden könnte, ohne das Ganze zu zerstören.

beschreibt, verstehen die amerikanischen Informatiker John Hennessy (*1952) und David Patterson (*1947) den Entwurf eines Rechners als den Bau einer

- » Maschine mit maximaler Rechenleistung unter Einhaltung der Kostenvorgaben.

Demnach ist die Rechengeschwindigkeit eines Computers zu messen, und die maximale Leistung soll mit minimalem Aufwand erzielt werden. Ein hoher Datendurchsatz oder eine kurze Antwortzeit entsteht demnach durch den ausgewogenen Einsatz der zur Herstellung eines Rechners verwendeten Elemente. Damit finden sich die vom römischen Architekten Vitruv (Marcus Vitruvius Pollio, 1. Jahrhundert v. Chr.) definierten Grundbegriffe der Architektur, „firmitas“ (Festigkeit, Stabilität), „utilitas“ (Nützlichkeit) und „venustas“ (Anmut, Schönheit), in abgewandelter Form auch als Ziele in der Rechnerarchitektur: Ausgewogenheit (Stabilität), Universalität (Nützlichkeit) und Einfachheit (Anmut/Schönheit).

Eine Rechenmaschine ist aus den in ▶ Kap. 5 beschriebenen Komponenten aufgebaut. Die Verschaltung dieser zu einer Maschine, um Algorithmen automatisch abzuarbeiten, ist eine Rechner- oder Maschinenstruktur. Diese Rechnerstruktur kann entweder programmierbar oder aber für einen ganz bestimmten festen Algorithmus aufgebaut sein. Der Begriff „Rechnerarchitektur“ wurde von Gene M. Amdahl (1922–2015), Frederick P. Brooks (*1931) und Gerrit A. Blaauw

1 In der Elektrotechnik hat sich dafür der Begriff Technologie eingebürgert, weil nicht nur die Struktur des Aufbaus, also die Schaltungstechnik, sondern auch das Wissen um ihre Herstellung notwendig sind, um Elektronik zu bauen. Inzwischen ersetzt aber die Rückübersetzung des englischen „technology“ das Wort Technik im Deutschen. Wir meinen mit Technologie immer, dass sich etwas physisch herstellen lässt, klammern Begriffe wie Softwaretechnologie daher bewusst aus.

(1924–2018) eingeführt. Dies geschah bei der Entwicklung der IBM System/360. Die IBM System/360 war dadurch gekennzeichnet, dass es sich um eine Familie von unterschiedlich leistungsfähigen Rechnern handelte, die aber die gleichen Befehle oder besser Befehlssatz aufwiesen. Dadurch konnten Programme, die auf einer Maschine entwickelt wurden, auch auf einem anderen Computer dieser Architektur ablaufen. Somit meint Rechnerarchitektur die typischen Charakteristika eines Rechners aus Sicht des Programmierers und nicht die Details des inneren Aufbaus. In dem Sinne, wie man in der Baukunst („architékton“, altgriech. für Baukunst) Baustile als Architektur bezeichnet, so werden in der Informatik Varianten von unterschiedlichen Befehlssätzen als Architektur wahrgenommen.

6

Definition 6.0.1 – Befehlssatzarchitektur (Amdahl, Brooks, Blaauw)

Die Architektur eines Rechners ist gegeben durch die Eigenschaften und das Verhalten des Computers aus Sicht des Programmierers. Die Beschaffenheit einer Maschine wird durch die dem Anwender zur Verfügung stehenden Befehle, den Befehlssatz, ihrem Bitformat, die Anzahl interner Speicher, der Register und ihre Anordnung, Verwendung und die Anzahl und Art der vorhandenen arithmetischen Operationen bestimmt. Die Reaktion auf Eingaben eines Rechners ergibt sich aus der Summe des Verhaltens einzelner Befehle und ihrer Reihenfolge, dem Programm und den zu den Befehlen gehörenden Möglichkeiten, externe Speicher anzusprechen.

Amdahl et al. unterscheiden zwischen der Architektur und der Implementierung eines Computers. Demnach ist die Realisierung eines Rechners durch

- » den aktuellen Hardwareaufbau, das logische Design und die Organisation der Datenpfade einer bestimmten Ausführung der Architektur definiert.

Amdahl et al. reduzieren den Begriff der Rechnerarchitektur auf die Klassifikation bestimmter Entwürfe. Demgegenüber fassen die deutschen Informatiker Peter Stahlknecht (*1933) und Ulrich Hasenkamp (*1949) die Rechnerarchitektur im Sinne des amerikanischen Elektroingenieurs Harold Stone (*1938) auf:

- » Rechnerarchitektur ist das Wissen um den inneren Aufbau eines Rechners und die Organisation der jeweiligen Arbeitsabläufe.

Da das Verhalten eines Rechners wesentlich durch seinen inneren Aufbau und die Organisation der Arbeitsabläufe bestimmt wird, kann die Definition nach Amdahl et al. erweitert werden:

Definition 6.0.2 – Rechnerarchitektur

Die Architektur ist bestimmt durch die Eigenschaften und das Verhalten des Rechners. Diese werden durch die zur Verfügung stehenden Befehle, ihr Format, die Anzahl interner Speicher, der Register und ihre Anordnung und die Verwendung und die Anzahl und Art der vorhandenen arithmetischen Operationen festgelegt. Das Verhalten eines Rechners wird durch seinen inneren Aufbau und die Organisation der Arbeitsabläufe bestimmt. Die Rechnerarchitektur ermöglicht, Maschinen derart zu klassifizieren, dass Programmierer und Ingenieure ein gemeinsames Verständnis über unterschiedliche Entwurfsstile erlangen, ohne tief in die jeweilig andere Domäne eintauchen zu müssen. Für den Programmierer stellt die Architektur eine abstrakte Beschreibung des Computers bereit, und dem Ingenieur liefert sie eine Vorgabe in Form einer Spezifikation für dessen Aufbau.

In der klassischen Baukunst wird der Begriff Architektur sowohl für die Kunst oder Wissenschaft von der Konstruktion als auch zur Klassifikation von Baustilen verwendet. In dem Sinne ist Rechnerarchitektur sowohl die Lehre vom Bau von Computern als auch ein Klassifikationsmittel für deren Entwurfs- oder Baustil. Der Rechnerarchitekt versucht daher, einfach zu programmierende und leistungsstarke Entwürfe von Rechnern zu gestalten, während der Programmierer auf die Benutzbarkeit der Maschine fokussiert.

Ursprünglich sollten Rechenmaschinen das Rechnen von Hand erleichtern. Im Laufe der Zeit zeigte sich aber, dass universell programmierbare Computer anspruchsvollere Aufgaben erledigen können. Elementar in jeder Maschine ist auch heute die Verknüpfung von Rechenoperationen. Bei einer solchen Berechnung werden Operanden mathematisch verknüpft und das Ergebnis gespeichert. Bei zusammengesetzten Rechenoperationen wie der Multiplikation sind oft mehrere elementare Rechenschritte nacheinander auszuführen. Dabei müssen die Operanden und die Ergebnisse über eine feste Zeit gespeichert werden. Die Verbindung zwischen dem eigentlichen Rechenwerk und dem Speicher ist daher elementar stilbildend für einen Computer. Im Laufe der Zeit entwickelten sich drei unterschiedliche Stile Speicher und Rechenwerk zu verbinden: Von der Stapelmaschine, die die Operanden in Stapspeichern vorhält, über die Akkumulatormaschine, bei der Ergebnisse in einem zentralen Speicher akkumuliert werden, bis hin zur Register- und dann universellen Registermaschine ist die Adressierung der Operanden und des Speichers zentrales Thema der Rechnerarchitektur. Die Art der Verknüp-

Architekturmerkmale

fung von Operanden und das Ansprechen des Speichers beeinflussen sowohl die innere Struktur des Rechners als auch die Befehlsformate und das Verhalten der Befehle. In diesem Zusammenhang wird auch gerne von Rechnerstrukturen und -organisation gesprochen.

Programmieren

Moderne Computer sind programmierbar, d. h., die Funktion der Maschine wird durch ein Programm (griech. „prόgramma“, Vorgeschriebenes, Vorschrift) festgelegt. Damit ist die Maschine selbst in ihrer Struktur universell und eine Folge von Befehlen; eben das Programm legt erst die spätere Funktion des Computers fest. Oft spricht man in diesem Zusammenhang von der Hardware, also der Rechenmaschine und der Software, der Befehlsfolge oder dem Programm, das die eigentliche Funktion bestimmt. So offensichtlich uns diese Unterscheidung heute erscheint, so schwierig war es in den Anfängen der Computertechnik, dieses Prinzip zu entwickeln. Die ersten Maschinen waren fest verdrahtet oder nur auf eine spezielle Funktion ausgelegt. Teure Hardware und lange Umrüstzeiten führten zunächst zu leicht änderbaren Steckverbindungen und dann später zu Programmen und Befehlen. Damit wurde der Computer universell.

Befehlssatz

Der Befehlssatz als das Element einer Rechnerarchitektur ist eine Spezifikation, die die Funktion einer Rechenmaschine aus Sicht des Programmierers definiert. Umgekehrt legt der Befehlssatz fest, wie ein Programmierer seine Programme schreiben muss, damit die Maschine die gewünschte Funktion ausführt. Der Befehlssatz oder besser seine Beschreibung stellt somit eine Art Handbuch eines Computers dar.

Definition 6.0.3 – Befehlssatz

Der Befehlssatz einer Maschine ist die Menge aller Befehle und der dazugehörigen Befehlsworte. Er ist die Schnittstelle zwischen dem Programm (der Software) und dem Rechner (der Hardware).

Damit ist das Programm die Implementierung eines Algorithmus auf einer speziellen Maschine². Im Folgenden werden zunächst die verschiedenen Architekturen des Befehlssatzes diskutiert. Die Anbindung des Speichers ist Thema des zweiten Kapitels. Am Ende beschreiben wir, wie die Struktur eines Rechners im Allgemeinen aufgebaut ist. Mit den Prinzipien der Mikroarchitektur genannten Rechnerstruktur lassen sich dann beliebige Befehlssätze implementieren.

² Im ursprünglichen Sinne der Rechnerarchitektur für eine Familie oder Klasse von Computern mit dem gleichen Befehlssatz.

Ausgehend von einer zentralen Berechnungseinheit zeichnen sich die verschiedenen Architekturstile von Rechnern durch unterschiedliche Varianten im Zugriff auf die Operanden eines Befehls aus. Diese Operandenadressierung wird ergänzt durch Techniken zur Adressierung von Daten in großen Speichern. Insbesondere Speicher mit wahlfreiem Zugriff haben ein komplexes Zeitverhalten. Der im Vergleich zur Berechnung zeitaufwendige Datenzugriff führt zu immer feineren und komplexeren Speichermodellen, um Rechen- und Datenzugriffszeit fein und ausbalanciert aufeinander abzustimmen.

Ziel der Überlegungen ist es, einen universell programmierbaren Rechner zu bauen. Universell programmierbar bedeutet, dass die Funktion der Maschine erst durch eine austauschbare Folge von Anweisungen, das Programm festgelegt wird. Diese Anweisungen werden als Bitmuster in einem Speicher abgelegt und heißen Befehl. Durch das Laden unterschiedlicher Bitmuster oder Befehle in den Speicher einer Maschine kann selbige unterschiedliche Programme ausführen. Verfügte man über eine Maschine, die mehrere Berechnungsschritte automatisch hintereinander ausführt, lässt sich mit dieser ein Algorithmus implementieren (engl. „implementation“, Ausführung, Umsetzung) und automatisch ausführen. Ein Algorithmus ist eine Handlungsvorschrift zur Lösung eines Problems und gliedert sich folglich in mehrere Einzelschritte.

Die grundlegenden Komponenten eines Computers sind der Prozessor (central processing Unit, CPU) zur Bearbeitung des Programms und der Speicher (Memory, M). Die Architektur von Computern wird ganz entscheidend davon beeinflusst, wie Prozessor und Speicher physisch und logisch verbunden werden und wie die Befehle des Prozessors auf den Speicher zugreifen -- man sagt, wie sie den Speicher adressieren. Daneben spielt auch die Auswahl, leider auch Adressierung genannt, der Register durch die Befehle eine Rolle. Zwar sind Register und Speicher in einer Architektur heute klar getrennt. Da dies aber in der Geschichte der Rechentechnik nicht immer der Fall war, vermischen sich oft die Begriffe. Oft ergibt es sich aus dem Kontext, ob mit Adressierung die Register- oder Operanden- oder vielmehr die Speicheradressierung gemeint ist.

6.1.1 Befehl und Programm

Jeder Einzelschritt des Algorithmus muss festgelegt werden. Dies geschieht mit einem Befehl. Ein Befehl ist eine Anweisung, die bestimmte Aktionen in der Hardware eines Compu-

ters auslöst, und eine Spezifikation, wo die Maschine die Argumente dieser Anweisung finden kann. Das Programmieren eines Computers ist somit das Finden einer sinnvollen Reihenfolge von Befehlen, um gezielt einen bestimmten Algorithmus zu implementieren.

Definition 6.1.1 – Befehl und Programm

Ein Programm ist eine Folge von Anweisungen, den Befehlen. Jeder Befehl gliedert sich in eine Operation und den zu der Operation gehörenden Operanden.

Bevor ein Befehl ausgeführt werden kann, ist er aus dem Speicher zu holen. Das immer wiederkehrende Holen und Ausführen eines Befehls und somit der Rhythmus der Befehlausführung heißt Befehlszyklus. Ein solcher Befehlszyklus besitzt mehrere Phasen. Zunächst muss der Befehl geholt werden (engl. „instruction fetch“, IF), als Nächstes muss die Maschine den Befehl decodieren (engl. „instruction decode“, ID). Es folgen die eigentliche Ausführung des Befehls (engl. „execute“, Ex) und das Rückspeichern des Ergebnisses (engl. „write back“, WB). Ausführlicher wird der Befehlszyklus im ▶ Abschn. 6.3 Mikroarchitektur behandelt.

Die Operation oder Anweisung eines Befehls wird durch einen Opcode (Operation-Code) codiert. Der Opcode ist ein eindeutiges Bitmuster. Jedes Bitmuster eines Befehls hat in einer Architektur eine festgelegte Bedeutung. Die Wahl der Codierung erfolgt speichereffizient, d. h., in einem Befehlssatz muss die Codierung nicht der Logik der einzelnen Befehlsklassen folgen, sondern kann davon abweichend komprimiert sein. Da Bitmuster für Menschen schwer zu unterscheiden und zu merken sind, ist jedem Opcode eine Mnemonik (Gedächtnisstütze, vom Griechischen „mnēmoniká“, Gedächtnis) genannte lesbare Abkürzung zugeordnet.

Definition 6.1.2 – Opcode

Ein Opcode oder Operation-Code ist in einem Befehlssatz ein eindeutiges Bitmuster, das einen Befehl und dessen Operandenzugriffe eindeutig spezifiziert.

Definition 6.1.3 – Operation

Die Anweisung oder Operation o bestimmt die Funktion eines Befehls. Ein Befehl kann ein oder mehrere Operationen aufweisen.

Eine Operation spezifiziert die logische Funktion eines Befehls. In einigen Befehlssätzen ist der Opcode identisch mit der Operation. Das ist aber nicht immer der Fall. Unterstützt eine Architektur für einen Befehl mehrere unterschiedliche Datenzugriffe (Adressierungsarten), dann kann eine Operation mehrere Opcodes besitzen. Dabei spezifiziert ein Opcode eine Operation mit einer der zugehörigen Operandenadressierungen.

Die Argumente eines Befehls spezifizieren die Schnittstelle zu den Daten. Über sie wird auf die Datenwörter der Operanden im Speicher zugegriffen. Da unterschiedliche Operationen eine unterschiedliche Anzahl von Operanden haben, hat jeder Befehl eine andere, unterschiedliche Anzahl von Argumenten.

Argumente

Definition 6.1.4 – Argumente eines Befehls

Die Operanden oder Argumente eines Befehls spezifizieren die Daten, auf denen die jeweilige Operation anzuwenden ist, und sind Teil des Befehlswortes.

Definition 6.1.5 – 3-Adress-Code (DAC)

Ein idealer Befehl, der als Modell und grundlegende Schnittstelle zwischen Programm und Maschine genutzt wird, besteht aus der Operation und drei Argumenten: der Zieladresse z und den beiden Quelladressen x und y .

Befehlswort

Um einen Befehl auszuführen, muss sein Bitmuster von der Maschine interpretiert werden. Damit ein Programmierer weiß, welche Bitmuster welche Funktionen des Computers auslösen, muss klar festgelegt sein, welches Bit des Bitmusters welche Funktion hat. Im Allgemeinen werden Bits bestimmter Funktionen zu einem Bitfeld zusammengefasst. Ein aus mehreren solcher Felder zusammengesetztes Bitmuster wird Befehlswort (engl. „instruction word“) genannt. Ein Befehlswort besteht aus einem Feld für den eigentlichen Befehl und Operanden, die durch den Befehl verknüpft werden.

Definition 6.1.6 – Befehlswort

Das Befehlswort ist ein Bitmuster, das einen Befehl codiert. Das Befehlswort ist an die Speicherwortbreite eines Rechners angepasst, kann aber ggf. aus mehreren Speicherwörtern bestehen. Ein Befehlswort besteht mindestens aus einem Bitmuster zur Spezifikation der Operation und ihrer Argumente:

$$\{o, z, x, y\}$$

Die Zahl der Argumente kann je nach Rechnerarchitektur variieren.

Befehlsformat

6

Da die Hardware des Rechners die Befehle interpretieren oder besser decodieren muss, muss von Rechnerarchitekten klar spezifiziert werden, welches Bit eines Befehls welche Funktion in der Maschine steuert. Die Anordnung der Bitfelder in einem Befehlswort heißt Befehlsformat (engl. „instruction format“). Das Format eines Befehlswortes wiederum hängt von der Architektur der Maschine ab – oder umgekehrt, die Befehlsformate eines Befehlssatzes spezifizieren eindeutig die Architektur einer Maschine. Dadurch wird festgelegt, um welchen charakteristischen Typ eines Computers es sich handelt. Ein Computer kann mehrere unterschiedliche Befehlsformate besitzen, da es unterschiedliche Arten von Befehlen mit unterschiedlichen Aufgaben gibt. Aus diesem Grund sind für jede Rechnerarchitektur mehrere eindeutige Befehlsformate festgelegt. Unterschiedliche Rechnerarchitekturen unterscheiden sich wesentlich in den von ihnen verwendeten Befehlsformaten. Man könnte an dieser Stelle einwenden, dass die Bedeutung der Bitfelder und der in ihnen gespeicherten Bitmuster dynamisch festgelegt werden könnte, ähnlich wie in einer natürlichen Sprache. Dies ist aber nicht möglich; zum einen, da eine derartige Zuordnung Ressourcen kostet und diese Ressourcen auf der Ebene der Rechnerarchitektur nicht zur Verfügung stehen. Zum anderen lassen sich noch so ausgeklügelte Datenstrukturen am Ende immer auf elementar spezifizierte Grundbausteine zurückführen. Das Befehlsformat ist somit das Element aller komplexen in Hochsprachen aufgebauten Datenstrukturen.

Definition 6.1.7 – Befehlsformat

Ein Befehlsformat ist ein Bitfeld vorgegebener fester oder variabler Länge. Das Befehlsformat f spezifiziert die Struktur des Befehlswortes.

Ein Befehlssatz hat oft mehrere unterschiedliche Befehle mit unterschiedlichen Befehlsformaten. Die Anzahl der Bits in einem Befehlswort richtet sich nach unterschiedlichen Kriterien. Sind die zur Verfügung stehenden Ressourcen beschränkt, steht also wenig Speicher zur Verfügung oder ist dieser teuer, dann ist es sinnvoll, Befehle mit variabler Wortlänge einzusetzen. Dies hat den Vorteil, dass nur das gerade Notwendige auch gespeichert werden muss. Sind die Ressourcen nicht ent-

scheidend oder soll der zu entwerfende Prozessor einen hohen Datendurchsatz haben, dann ist ein Befehlswort fester Länge zu wählen. Das macht die Struktur der Prozessorhardware einfach und schnell. Derartige Entwürfe verbrauchen weniger elektrische Energie und sind einfacher zu entwerfen.

► Beispiel 6.1.1 – Befehlswort

Die unterschiedlichen Befehlwörter können durch das folgende allgemeine Tupel spezifiziert werden:

$$\{o_4, a_4, s_8, s_8\}_8$$

Diese Struktur bedeutet, dass das Befehlswort ein 4-Bit-Feld zur Speicherung des Opcode, 4 Bit zum Festlegen der Zugriffsart und 2 8-Bit-Speicheradressen als Argumente besitzt. Die Wortbreite des Rechners ist 8 Bit, gekennzeichnet durch die Zahl 8 am Ende der Struktur. Damit belegt ein derartiges Befehlswort 3 Worte. ◀

Die Befehlsformate einer Maschine stellen das Fundament des Befehlssatzes dar. Die grundlegende Architektur eines Computers spiegelt sich direkt in seinen Befehlsformaten wider. Befehlsformate variieren in Befehlsart oder -typ und bezüglich der gewählten Architektur. Um Befehle und die dazugehörigen Formate eindeutig und aussagekräftig zu beschreiben, soll im Folgenden eine textuelle Notation eingeführt werden. Diese Notation ist kompakter als Zeichnungen der jeweiligen Formate und kann schneller und einfacher aufgeschrieben werden. Insbesondere beim Vergleich unterschiedlicher Architekturen und ihrer Formate trägt es zur Übersichtlichkeit bei.

Definition 6.1.8 – Befehlsformat

Sei f_w ein Befehlswort mit w Bit. Dann kann dieses Format in folgende Bitfelder aufgeteilt werden:

- o_i , ein i Bit langer Opcode,
- z_j, x_j und y_j , jeweils j Bit lange Befehlsargumente,
- u_k und v_k , Bitfelder zur Verwaltung der Befehlsformate.

Als Beispiel sollen die Befehlsformate der bekannten „Microprocessor-without-interlocked-pipeline-stages (MIPS)-Architektur“ dienen. Der MIPS besitzt drei Formate, das R-, I- und J-Format:

MIPS-Formate

► Beispiel 6.1.2 – MIPS-R-Format

Ein MIPS-Befehl im R-Format (Registerformat) hat folgende Struktur:

$$f_{32R} = \{o_6, \{r_5\}_3, c_5, o_6\}$$

Ein Befehlswort des R-Formats besteht aus 32 Bit, besitzt einen 6-Bit-Opcode jeweils am Anfang und Ende des Befehlswortes, 3 Registeroperanden zu jeweils 5 Bit und eine 5 Bit lange Konstante.



► Beispiel 6.1.3 – MIPS-I-Format

Ein Befehl im I-Format (Immediate-Format) hat folgende Struktur:

$$f_{32I} = \{o_6, \{r_5\}_2, c_{16}\}$$

Einem 6 bit langen Opcode folgt die Spezifikation zweier Operanden in Form zweier 5 Bit langer Bitfelder. Am Ende eines solchen Befehlswortes steht eine 16 Bit zählende Konstante, die bei bestimmten Befehlen auch als Adresse aufgefasst werden kann (Displacement):

$$f_{32I} = \{o_6, \{r_5\}_2, d_{16}\}$$



► Beispiel 6.1.4 – MIPS-J-Format

Ein Befehl im J-Format („jump format“) hat folgende Struktur:

$$f_{32J} = \{o_6, d_{26}\}$$

Einem 6 Bit langen Opcode folgt eine 26 Bit lange Adresse, das Sprungziel (Displacement): ◀

Eine Folge von Befehlsworten nennt man ein Maschinenprogramm.

Definition 6.1.9 – Maschinenprogramm

Eine Folge von binären Befehlsworten, die sich von einer Maschine interpretieren lassen, heißt Maschinenprogramm. Ein solches Programm steuert die Maschine derart, dass sie den vom Programmierer gewünschten Algorithmus ausführt. Ein Programm implementiert einen Algorithmus. Ein solches Maschinenprogramm heißt oft auch Objektcode.

Binäre Bitfelder sind von Menschen schwer zu lesen. Aus diesem Grund bekommt jeder Befehl einen Namen und eine zu diesem Namen gehörende Abkürzung. Diese Mnemonik genannte Abkürzung wird um einheitliche Schreibweisen zur Codierung des Speicherzugriffs erweitert. Ein aus Mnemonik mit

Operandenzugriffen bestehendes Programm entspricht in jeder seiner Zeilen einem Maschinenbefehl. Eine Folge solcher abstrakter Befehle wird Assembler-Programm genannt. In der Assembler-Sprache lassen sich auch Sprungziele abstrakt über Namen definieren. Ziel dieser Abstraktion ist, eine Sprache zu definieren, die leicht zu lesen ist und doch die typischen Merkmale einer Architektur vollständig darlegt. In Assembler verfasste Programme müssen mit einem anderen speziellen Programm, dem Assembler, in die von der Maschine verstandenen binären Bitfelder übersetzt werden.

Definition 6.1.10 – Assembler-Programm

Ein Assembler-Programm ist eine Folge von leicht lesbaren Befehlwörtern. Die Datenstruktur und die Opcodes werden dabei durch menschenlesbare Zeichenketten ersetzt. Ein Assembler-Programm ist äquivalent zu dem dazugehörigen Maschinenprogramm.

Definition 6.1.11 – Assembler

Ein Assembler ist ein Programm, das ein Assembler-Programm in ein Maschinenprogramm übersetzt. Aus dem Assembler-Code wird also der Objektcode.

Der Begriff Assembler wird oft sowohl für die Sprache bestehend aus unterschiedlichen Mnemoniken als auch für das Übersetzungsprogramm benutzt. Die jeweilige Bedeutung ergibt sich dann immer aus dem Kontext.

6.1.2 Die Architektur und ihre Auswirkung auf den Befehl

Die Anordnung der Argumente eines Befehls ist ein wesentliches, charakteristisches Merkmal eines Computers. Die Anzahl der Argumente in einem Befehl beeinflusst die Struktur der späteren Hardware. Oder andersherum, die Struktur einer Implementierung, die Struktur des Rechenwerks legt die Anzahl der Argumente eines Befehls fest.

Im vorhergehenden Abschnitt haben wir gesehen, dass ein idealer Befehl drei Argumente hat: ein Zieladresse, an der das Ergebnis der Operation gespeichert werden kann, und zwei Quelladressen, deren Inhalte durch die Operation verknüpft werden. Damit wäre alles gesagt. Leider ist die Sache nicht so einfach – zum einen, weil nicht jede Operation zwei Argumente benötigt und es auch Operationen auf ein einzelnes Argu-

ment gibt, zum anderen, weil ein Befehl am Ende von einer realen Maschine verstanden und ausgeführt werden muss. Es gibt technische Einschränkungen, die dazu führten, dass Befehlsätze definiert wurden, deren Befehle keine drei Argumente besitzen. Man vergegenwärtige sich, dass jedes Argument Speicherplatz benötigt. Ist Speicher teuer und daher nicht in ausreichendem Maß vorhanden, muss auf Argumente verzichtet werden. Besitzt ein Befehl kein oder nur ein Argument, kann er folglich effektiv gespeichert werden. In der Vergangenheit war Speicher teuer und schwer³. Daher war es erforderlich, Programme kompakt zu speichern.

Im Folgenden werden die unterschiedlichen, charakteristischen Stile von Rechnern diskutiert. Diese ergeben sich notwendigerweise durch die Anzahl und Anordnung der Operanden im Befehl. Sie legen daher die Struktur des Befehlswortes und die Grundstruktur des Rechenwerks und damit die Struktur der späteren Implementierung fest.

Allgemein kann ein Verfahren für das Bereitstellen von Daten Protokoll (lat. „protocollum“, Klebe oder Leim) genannt werden. Ein Protokoll beschreibt ein Verfahren, das die Daten so überträgt, dass sie nach der Übertragung korrekt beim Empfänger angekommen sind. Im Fall des Speicherzugriffs durch ein Rechenwerk besteht das Protokoll aus der Auswahl der gewünschten Speicherstellen und dem Transport der Daten zum Rechenwerk. Der Teil des Protokolls, der die Auswahl der richtigen Speicherstelle vornimmt, heißt Adressierung (Adresse aus dem Lateinischen oder Französischen, Anschrift). Wesentliches Architekturnmerkmal des Datenflusses ist also die Adressierung des Speichers zum Transport der Rechenoperanden. Wie wir später sehen, ist die Form der Adressierung und somit die Wahl des Transportprotokolls elementar und hat Einfluss auf die später beschriebene Struktur des Rechenwerks und dessen Anbindung an den Speicher. Das Format der Befehle ist ebenfalls stark abhängig von der Adressierung der Argumente eines Befehls.

Die Anordnung der Operanden oder Argumente eines Befehls ist ein architektonisches Merkmal sowohl des Rechenwerks oder der Mikroarchitektur als auch der Befehlssatzarchitektur oder einfach Architektur eines Computers.

Der Zugriff auf Speicherstellen ist abhängig von der verwendeten Speichertechnik. Der wahlfreie Zugriff auf jede beliebige Zeile von Zellen eines Speicherwortes erfordert eine komplexe Auswahllogik. Ist diese bei kleinen Speichern noch übersichtlich und leicht zu realisieren, wächst der Aufwand exponentiell mit der Größe des Adressraumes. Aus diesem Grund wurden die ersten Rechner als Zählmaschinen reali-

³ Magnetkernspeicher.

siert. In Akkumulatoren (lat. „accumulator“, Aufhäufer) werden die Ziffern einer Zahl gespeichert, und Berechnungen erfolgen durch das Abzählen von Taktten gemäß dieser gespeicherten Werte. Eine Zählmaschine ist das elektronische Äquivalent zu mechanischen Rechnern auf der Grundlage von Zahnrädern. Da die Rechenoperationen ausschließlich auf den Akkumulatoren ausgeführt werden, entfällt ein aufwendiger Speicherzugriff; eine Adressierung ist nicht nötig. Elektronische Speicher können dagegen aufwendiger gebaut werden. Zunächst, unter der ausschließlichen Verwendung von Flip-flops in Röhrentechnik, ließen sich große Stapspeicher aufbauen. Bei einem Stapspeicher oder Stack entfällt die Adress-decodierung der Speicherstellen, da das zu schreibende oder zu lesende Datum über ein in einem Schieberegister gespeicherten Bit ausgewählt werden kann. Große Speicher mit komplexer Adressdecodierlogik sind erst mithilfe der Halbleitertechnik kostengünstig herzustellen. Große wahlfreie Speicher waren zunächst teuer und wurden in den frühen Jahren der Computertechnik selten eingesetzt.

Ein Stapel oder Stack ist die einfachste Art, einen Speicher zu bauen, da die jeweilige Auswahl der richtigen Speicherstelle der jeweiligen Operanden eines Befehls sehr wenig Logik benötigt. Aus diesem Grund hatten frühe Rechner einen auf Staps basierenden Operandenspeicher. Der Transport der Daten vom Speicher zur Recheneinheit ist einfach: Die beiden obersten Elemente eines Staps werden direkt an die zwei Eingänge der Recheneinheit angeschlossen und das Ergebnis anschließend wieder auf den Stapel gelegt. Um auf diese Art Rechenoperationen auszuführen, können nun der Stapel gelesen und gleichzeitig der Stapelzeiger für neue Einträge so verschoben werden, dass das Ergebnis die alten Einträge überschreibt. Gedanklich werden so beide Operanden vom Stapel genommen oder gelesen, und das Ergebnis wird dann an die alte Stelle gelegt. Eine Rechenoperation wie die Addition („add“) kann so ohne Angabe von Operanden spezifiziert werden. Der zugehörige Assembler-Code eines Befehls ist nur die Zeichenkette oder Mnemonik der entsprechenden Operation, und das Maschinenwort besteht lediglich aus dem die Operation codierenden Binärmuster. Eine solche Architektur heißt 0-Adress-Maschine oder auch 0-Adress-Computer, weil im Befehlswort keine Operandenadressen anzugeben sind. Die Codierung eines Programms einer solchen 0-Adress-Maschine ist sehr kompakt und daher spechereffizient.

Die Größe von Staps ist naturgemäß begrenzt, zum einen durch die Kosten sie zu bauen, zum anderen aber auch durch die Zugriffszeit. Tiefe Staps haben zwar eine zu der Tiefe des Staps lineare Zugriffszeit. Komplexe Programme erfordern jedoch häufig das Umsortieren von Daten und somit ein stän-

Stapelmaschine

Stapspeicher

diges Umordnen der Reihenfolge auf dem Stapel. Das bedeutet zum einen, dass mehrere Stapel eingesetzt werden müssen, und zum anderen, dass im schlimmsten Fall die Tiefe der jeweiligen Stapel die Ausführungsgeschwindigkeit und somit die Leistung des Rechners begrenzt. Daher ist es notwendig, einen externen Speicher neben den Stapel zu stellen. Dies ist ein Speicher mit wahlfreiem Zugriff: in der Vergangenheit z. B. ein Loch- oder Magnetband, später ein wahlfreier Halbleiterspeicher. Durch die Erweiterung um einen solchen wahlfreien Speicher müssen nun Befehle eingeführt werden, die Daten vom Speicher zum Stapel und zurück transportieren. Diese Befehle benötigen mindestens eine Adresse, an der ein Datum lokalisiert und so gelesen werden kann. Im einfachsten Fall werden diese Daten auf den Stapel gelegt (push) oder vom Stapel in den Speicher geschrieben (pop). Neben dem eigentlichen Opcode (push oder pop) muss nun das Befehlswort noch die Adresse der entsprechenden Speicherstelle erhalten. Eine solche Maschine heißt 1-Adress-Computer.

Note[Akkumulator-Maschine] Es ist nicht zwingend notwendig die Operanden in einem Stapel zu speichern. Der einfachste Fall eines 1-Adress-Computers besitzt nur eine, Register genannte, universelle Speicherstelle. Diese zeichnet sich dadurch aus, dass ihr Eingang auf den Ausgang der Recheneinheit geschaltet ist und ihr Ausgang an deren Eingang. Der zweite Eingang der Recheneinheit ist dann direkt mit dem Speicher verbunden. Die Adresse im Befehlswort zeigt damit direkt auf den zweiten Operanden im wahlfreien Speicher. Mit einer solchen Struktur lassen sich nun Maschinen realisieren, bei denen nicht nur die Transportbefehle auf eine Speicherzelle zeigen, sondern Rechenoperationen direkt auf dem wahlfreien Speicher arbeiten. Die Angabe einer einzigen Adresse in den Befehlsworten ist ausreichend, da der erste Eingang der Recheneinheit immer mit dem Register verbunden ist. In einem derartigen Register lassen sich Rechenergebnisse akkumulieren. Das Register heißt daher Akkumulator und eine derartige 1-Adress-Maschine wird auch Akkumulatormaschine genannt. Gegenüber einer Stapelmaschine benötigt ein solcher Rechner neue Transportbefehle zum Laden und Speichern des Akkumulators – z. B. *acc.la* oder *lda* („load accumulator“) und *sta* („store accumulator“). □ Abb. 6.1 zeigt eine solche Maschine.

Note[2-Adress-Maschine] Wird die Anzahl der Register erhöht, um z. B. die Zugriffszeit auf Operanden zu reduzieren und somit die Rechengeschwindigkeit zu erhöhen, müssen die Register selbst einzeln ansprechbar oder adressierbar sein. Behält man das Konzept des Akkumulators bei, dann wird das Ergebnis einer arithmetischen oder logischen Rechenoperation immer im Akkumulator abgelegt. Das Ergebnis überschreitet dabei immer mindestens einen der beiden Operanden. Bei

6

Akkumulatormaschine

2-Adressmaschine

6.1 · Architektur des Befehlssatzes

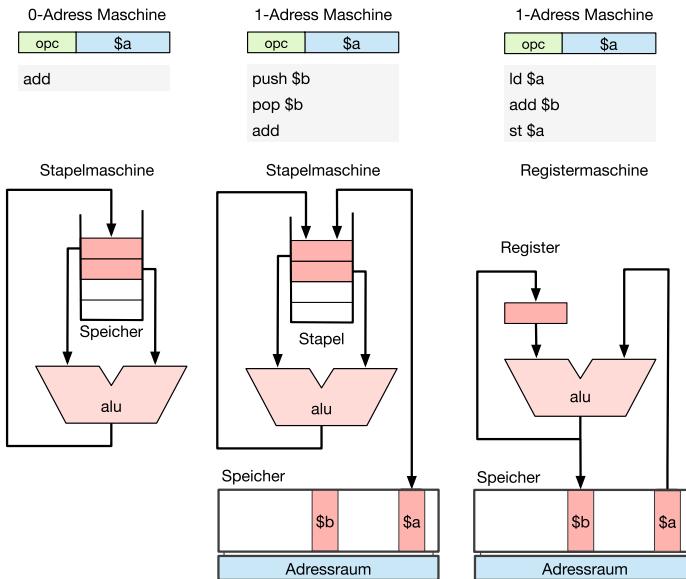
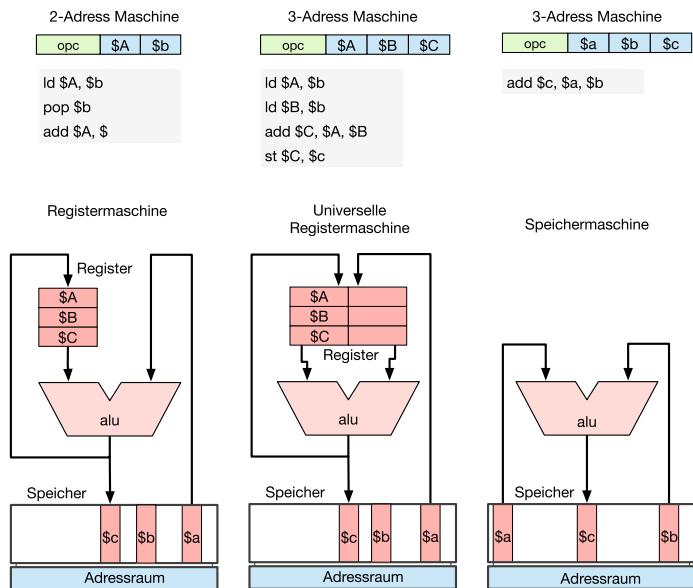


Abb. 6.1 Architektur von 0-Adress- und 1-Adress-Maschinen

einer derartigen Registermaschine kommen die Befehlswörter des Befehlssatzes mit maximal zwei Operandenadressen aus (Abb. 6.2).

Note[3-Adress-Maschine] Da auf der einen Seite die Zugriffszeiten auf den wahlfreien Speicher im Verhältnis zu den Zugriffszeiten auf die Register immer weiter stiegen, auf der anderen Seite aber immer mehr Fläche auf den integrierten Schaltungen der Prozessoren verfügbar wurde, konnten Rechner gebaut werden, bei denen die Recheneinheit ausschließlich auf Register zugreift. Wird gleichzeitig das Befehlsformat um ein weiteres Adressfeld erweitert, entsteht ein 3-Adress-Computer. Bei einer solchen Maschine operieren die arithmetisch-logischen Befehle ausschließlich auf dem Registersatz („register file“), und nur die Transportbefehle (load, store) greifen auf den wahlfreien Hauptspeicher zu. Eine solche Architektur wird daher im angelsächsischen Sprachraum Load-store-Architektur genannt. Eine solche 3-Adress-Maschine erlaubt es direkt einen 3-Adress-Code, wie er z. B. in einem Datenflussgraphen steht, direkt in Assembler oder Maschinensprache umzusetzen. Zwar ist der resultierende Programmcode nicht so kompakt wie bei 1- oder 2-Adress-Maschinen, aber immer kostengünstigere wahlfreie Halbleiter-Speicher haben diesen Aspekt immer unwichtiger werden lassen.

3-Adressmaschine



■ Abb. 6.2 Architektur von 2-Adress- und 3-Adress-Maschinen

Speichermaschine

In den 1960er-Jahren war es noch nicht ersichtlich, dass sich das Verhältnis der Rechenzeit zu der Speicherzugriffszeit zuun- gunsten der Speicherzugriffszeit verändern würde. Aus diesem Grund wurden Maschinen postuliert, die mit einem 3-Adress-Format direkt auf den Speicher zugreifen. Bedingt durch die relativ zur Rechenzeit langen Speicherzugriffe konnten sich derartige Maschinen allerdings nicht durchsetzen.

6.1.3 Befehlsgruppen

Die Rechenoperationen der Recheneinheit entsprechen in einem Befehlssatz jeweils einer Gruppe von Befehlen mit der jeweiligen Operation mit verschiedenen Adressierungsarten. Diese Befehle bilden den Kern eines jeden Computers und gliedern sich in die arithmetischen und die logischen Operationen. Sie bilden also die Gruppe der arithmetisch-logischen Befehle.

Ausgehend von den von der Recheneinheit angebotenen Operationen muss ein Befehlssatz Befehle für die Addition (*add*), die Subtraktion (*sub*) und ggf. die Multiplikation (*mul*) mit der Division (*div*) enthalten.

Daneben bieten Recheneinheiten häufig auch logische Operationen wie UND (*and*), NICHT (*not*), XOR (*xor*) an. Manche Befehlssätze besitzen noch die Befehle NAND und/oder

Arithmetische Befehle

Logische Befehle

NOR. Aus der Schaltalgebra und den De Morganschen Gesetzen folgt, dass in einem Befehlssatz nicht alle möglichen logischen Operationen angeboten werden müssen. Es reicht eine vollständige logische Basis, aus der alle anderen Operationen abgeleitet werden, bereitzustellen. Neben diesen grundlegenden Operationen unterstützen gängige Befehlssätze noch unterschiedliche Schiebe- und Rotationsoperatoren. Der Unterschied zwischen Schieben und Rotieren ist, dass Bits bei einer Schiebeoperation verloren gehen und bei einer Rotation diese Bits auf der anderen Seite eines Registers wieder hinzugefügt werden. Daneben wird bei den Schiebeoperationen zwischen arithmetischen und logischen Schiebebefehlen unterschieden. Beim arithmetischen Rechtsschieben ist es das Ziel, das Vorzeichen des gespeicherten Wertes zu behalten. Daher ist beim Rechtsschieben das neue höherwertige Bit eine Kopie des alten höherwertigen Bits. Beim Linksschieben wird dagegen das niedrigstwertige Bit durch eine 0 ersetzt. Im Gegensatz dazu wird beim logischen Schieben, sowohl beim Rechts- als auch beim Linksschieben jeweils die frei werdende Stelle mit einer 0 aufgefüllt. Da logisches und arithmetisches Linksschieben identisch sind, fehlt in vielen Befehlssätzen die Operation arithmetisches Linksschieben.

Die □ Tab. 6.1 gibt einen Überblick über typische arithmetisch-logische Befehle am Beispiel des Befehlssatzes des Mikroprozessors 6502 und des MIPS.

Der Inhalt von Registern muss aus dem Hauptspeicher geladen oder in diesem abgelegt werden, um mit einer eingeschränkten Anzahl von Registern Programme mit vielen Variablen ausführen zu können. Es muss darüber hinaus auch möglich sein, den Inhalt eines Registers in ein anderes Register zu verschieben. Diese Aufgaben werden von Transportbefehlen ausgeführt. Transportbefehle eines typischen Befehlssatzes sind das Ablegen von Daten auf einem Stapel („push“ und „pop“), die Lade- und Speicherbefehle für die Register („load“ und „store“) und Befehle, die ein Datum von einem Register oder einer Speicherstelle in ein anderes Register oder eine andere Speicherzelle schreiben („move“). Einige Beispiele für Transportbefehle zeigt die folgende □ Tab. 6.2.

Ein Algorithmus zeichnet sich durch bedingte Entscheidungen aus. Damit also ein Programm einen Algorithmus vollständig implementiert, ist es notwendig Befehle einzuführen, die in Abhängigkeit eines vorgegebenen Kriteriums den Programmfluss verzweigen – also den Ablauf des Programms in Abhängigkeit von vorgenommenen Berechnungen ändern. Dies geschieht durch das bedingte Laden des Befehlszählers. Derartige Befehle heißen bedingte Verzweigung („conditional branch“). Typisch für bedingte Verzweigungen ist oft, dass die Bedingung in Abhängigkeit von einem Statusbit (Flagge genannt)

Transportbefehle

Bedingte Sprungbefehle

Tab. 6.1 Assembler-Mnemoniken für die Befehlsklasse der arithmetisch-logischen Befehle

	6502	MIPS	Bedeutung
add	„Addition“	add	„Addition“
sub	„Subtraction“	sub	„Subtraction“
and	„And“	and	„And“
ora	„Or accumulate“	or	„Or“
eor	„Exclusive or (with accumulator)“	xor	„Exclusive or“
		nor	„Not or“
rol	„Rotate left“		Linksrotieren
ror	„Rotate right“		Rechtsrotieren
lsl	„Logical shift right“	srl	„Shift right logical“
		sll	„Shift left logical“
		sra	„Shift right arithmetic“
asl	„Arithmetic shift left“		Arithmetisches Linksschieben
		sllv	„Shift left logical variable“
		srlv	„Shift right logical variable“
			Rechtsschieben der Variable

der Recheneinheit ausgeführt wird. Dazu muss dieses aber zunächst einmal gesetzt werden. Die meisten Rechner verwenden dafür Vergleichsbefehle („compare“). Dabei wird der Inhalt zweier Register oder Speicherstellen verglichen, indem deren Inhalte voneinander abgezogen werden. Ist das Ergebnis einer Subtraktion gleich 0, wird die Flagge Z (engl. „zero flag“) gesetzt. Sind die beiden Werte nicht gleich, resultiert aus einer Subtraktion ein negatives Ergebnis, die Flagge N wird gesetzt.

■ Tab. 6.2 Assembler-Mnemoniken für die Befehlsklasse der Trans-
portbefehle

	6502	MIPS	Bedeutung
LDA	„Load accumula- tor“	ld	„Load“ Lade aus Hauptspeicher
STA	„Store accumula- tor“	st	„Store“ Schreibe in den Hauptspeicher
TAX	„Transfer accumula- tor to <i>X</i> “		Schreibe Inhalt des Akku-mulators in das Indexregister <i>X</i>
		mfhi	„Move from HI“ Bewegt den Inhalt des ausgezeichneten Regis- ters HI in ein beliebiges Register
		mflo	„Move from LO“ Bewegt den Inhalt des ausgezeichneten Regis- ters LO in ein beliebiges Register
		mthi	„Move to HI“ Bewegt den Inhalt eines Register in das ausge- zeichnete Register HI
		mtlo	„Move to LO“ Bewegt den Inhalt eines Register in das ausge- zeichnete Register LO

(kleiner), und ist das Ergebnis positiv, entsteht ein Überlauf (größer), und das Overflowbit wird gesetzt. Dabei ist immer darauf zu achten, welche Speicherstelle oder welches Register der Subtrahend und welches der Minuend der Vergleichsoperation sind. Auf den Flaggen können nun die bedingten Verzweigungen *bcc* („branch carry clear“), *bcs* („branch carry set“), *beq* („branch on equal“), *bne* („branch on not equal“), *bpl* („branch on plus“), *bmi* („branch on minus“) testen und entsprechend verzweigen. Es gibt darüber hinaus Architekturen, die für jede Flagge der Recheneinheit eine bedingte Verzweigung anbieten, und zwar jeweils für den Fall, ob die entsprechende Flagge gesetzt ist oder nicht. Beispielsweise sind in der SPARC-Architektur u. a. die Befehle *bvc* („branch on over-
flow clear“) und *bvs* („branch on overflow set“) definiert. Die ■ Tab. 6.3 zeigt wieder exemplarisch einige bedingte Sprünge und die dazugehörigen Vergleiche.

Tab.6.3 Assembler-Mnemoniken für die Befehlsklasse bedingter Sprung

6502	MIPS	Bedeutung
bcc „Branch on carry clear“		Verzweige, wenn die Flagge Carry nicht gesetzt ist
bcs „Branch on carry set“		Verzweige, wenn die Flagge Carry gesetzt ist
beq „Branch on equal“	beq	„Branch on equal“ Verzweige bei Gleichheit ($Z = 0$)
bne „Branch on not equal“	bne	„Branch on not equal“ Verzweige bei Ungleichheit ($Z = 1$)
tpl „Branch on plus“	bgtz	„Branch on greater“ Verzweige, wenn größer
bmi „Branch on minus“	blez	„Branch on lower or equal“ Verzweige, wenn kleiner
cmp „Compare“	slt	„Set register if lower“ Vergleiche und setze Flaggen oder Register je nach Ergebnis
	slg	„Set register if greater“ Vergleiche und setze Flaggen oder Register je nach Ergebnis

6

Unbedingte Sprungbefehle

Bedingte Verzweigungen gehören zur Gruppe der Kontrollflussbefehle, da sie den Fluss des Programms verändern. Neben den bedingten Verzweigungen sind in vielen Befehlsätzen absolute Sprünge definiert. Ein Grund, derartige Befehle einzuführen, ist das Unterprogramm. Mithilfe der Befehle *jsr* („jump to subroutine“) und *rts* („return from subroutine“) ist es möglich, in einen anderen Teil des Programms zu springen und automatisch zu dieser Sprungstelle zurückzukehren. Der Befehl *jsr* ändert nicht nur den Befehlszähler auf die im Befehl angegebene neue Adresse, sondern er sorgt auch dafür, dass vorher der alte Stand des Befehlszählers gerettet wird. Bei einigen Rechnern wird dazu der Befehlszähler auf dem Stapel abgelegt. Dadurch entsteht automatisch eine Hierarchie von Funktionsaufrufen. Bei anderen Architekturen wird der Befehlszähler in einem dafür extra reservierten Register des Registersatzes gespeichert, und ganz ausgefeilt Hardware schiebt ein gleitendes Fenster über einen sehr großen Registersatz. Wesentlich dabei ist, dass mit Ausführung eines Sprungs die Adresse nach der Herkunftsadresse des Sprungs gespeichert wird, um an diese Stelle zurückzuspringen zu können (► Tab. 6.4).

Tab. 6.4 Assembler-Mnemoniken für die Befehlsklasse Sprung

	6502		MIPS	Bedeutung
<i>jsr</i>	„Jump to subroutine“	<i>jal</i>	„Jump and link“	Springe zu einer neuen Adresse, und speichere den Stand des Befehlszählers
<i>rts</i>	„Return from subroutine“	<i>jr</i>	„Jump register“	Rücksprung zu einer auf dem Stack oder in einem Register liegenden Adresse
<i>jmp</i>	„Jump“	<i>j</i>	„Jump“	Springe zu der angegebenen Adresse

Der Vollständigkeit halber sollen zum Schluss noch die Systembefehle erwähnt werden. Moderne Befehlssätze beinhalten noch Befehle zur Ausnahmebehandlung (engl. „traps“ und „exceptions“) und in älteren und einfachen Befehlssätzen das Sperren oder Freigeben von Unterbrechungen (Interrupts). Daneben gibt es noch Befehle zur Unterstützung der Programmierung von Betriebssystemen, z. B. die Befehle `syscall` und `break`. Befehle zur Abfrage und zum Setzen und Löschen von Statusregistern gehören ebenfalls in diese Befehlsklasse. Insbesondere in modernen Architekturen können so unterschiedliche Betriebszustände (engl. „user“ und „supervisor mode“) verwaltet werden.

Systembefehle

6.1.4 Operandenzugriff und Adressierung

Eigentlich ist das Bereitstellen der Operanden keine schwierige Sache: Entsprechend der Architektur der Maschine und des daraus abgeleiteten Befehlsformats muss einfach nur das Datum aus der im Befehl angegebenen Speicherstelle gelesen werden und mit der zweiten oder einer weiteren, im Befehl spezifizierten Adresse gemäß der gewählten Operation verknüpft werden. Das Ergebnis wird dann an einer ebenfalls im Befehl angegebenen dritten Stelle gespeichert.

Sowohl programmiertechnische als auch technologische Randbedingungen haben dazu geführt, dass sich die Spezifikation einer Operandenadresse als Argument eines Befehls nicht auf die einfache Angabe einer Stellennummer oder Adresse eines homogenen linearen Speichers erschöpft. Leider hängt die Latenz eines Speicherzugriffs quadratisch von der Zahl seiner Stellen ab. Speicher, der in einer bestimmten Technologie gefertigt wird, kann aus diesem Grund ohne strukturelle oder architektonisch-logische Maßnahmen nicht beliebig groß ge-

Randbedingungen

macht werden. Vielmehr muss die vom Speicher hervorgerufene Latenz in einem gesunden Verhältnis zur Ausführungs geschwindigkeit des Prozessors stehen.

Hinzu kommt, dass ein Befehlswort die Adressen der Argumente des Befehls ebenfalls enthalten muss. Im Programmspeicher ist also viel Speicherplatz für die Adressen der Argumente zu reservieren. Bei einem großen Adressraum sind die Bitmuster dieser Adressen ggf. sehr breit und benötigen daher viel Speicherplatz. Die Anzahl der Stellen des Speichers und damit die Anzahl der Adressbits der Argumente begrenzt die Länge des Befehlswortes und damit die Anzahl der von einem Rechner adressierbaren Speicherzellen. Die Größe des Speichers ist dann abhängig von der Spezifikation der Argumente in einem Befehl. Es ist leicht einzusehen, dass Rechnerarchitekten diese beiden Aspekte entkoppeln möchten und die Länge des Befehlswortes nicht von der Anzahl der adressierbaren Speicherstellen abhängig sein soll.

Das Schreiben von Programmen soll einfach und effizient sein. Im Laufe der Zeit wurden daher unterschiedliche Adressierungsarten eingeführt. In den 1960er- bis 1980er-Jahren wurde mit jeder neuen Rechnergeneration eine neue Adressierungsart eingeführt. Ziel war es, möglichst oft genutzte algorithmische Strukturen zu unterstützen und so eine einfache Abbildung der Anwendungsprobleme oder der neu entwickelten Hochsprachen zu ermöglichen.

Abhängig von der Art der Operandenadressierung unterstützt ein Befehlssatz bis zu drei Argumente eines Befehls. Um ein Programm effizient zu speichern, besitzen die Befehle und deren Formate immer nur die zur Ausführung eines Befehls notwendigen Argumente. Einige wenige Befehle benötigen kein Argument, manche nur eins, andere aber zwei und mehr. Durch den geschickten Einsatz spezieller Register, z. B. des Akkulators kann die Anzahl der Argumente eines Befehls auf zwei beschränkt werden.

Um die einzelnen Argumente eines Befehls anzusprechen, ist zwischen Registern und Speicher zu unterscheiden. Die Anzahl der Register einer Maschine ist im Gegensatz zum Speicher begrenzt. Das hat den Vorteil, dass sie wesentlich einfacher zu adressieren sind. Um ein Register eindeutig anzusprechen, sind wesentlich weniger Adressbits notwendig als um eine Stelle im Speicher anzuwählen. Damit ist die Adressierung von Registern viel einfacher als die Adressierung des Speichers. Komplexe Protokolle sind in dem Fall nur bei Speicherzugriffen notwendig, und zwar, um einen möglichst großen Adressraum bei hoher Kompaktheit der Befehle zu ermöglichen. Die meisten Registermaschinen sind als 2-Adress-Maschinen realisiert. Dabei sind ein Argument des Befehls eine kompakte Registeradresse und das andere Argument eine Speicher-

Operanden- adressierung

Register und Speicher

Adresse. Dieses Vorgehen stellt einen Kompromiss zwischen Ausdrucksmächtigkeit und Speichereffizienz eines Befehlssatzes dar. Die Entwicklung von Computern hat gezeigt, dass eine konzeptionelle Trennung von Programmen und Daten bei gleichzeitigem Vorhalten beider Aspekte in einem gemeinsamen Speicher zu effizienten Programmen führt. Um das Programm und seine Daten getrennt verwalten zu können, wurden mehrstufige Adressprotokolle entwickelt. Mehrstufigkeit meint, dass das Argument eines Befehls nur einen Teil der notwendigen Adressinformation enthält und die weiteren Daten zum Finden eines Arguments in Tabellen im Speicher stehen. Unterstützt die Hardware die Verwaltung dieser Tabellen, entstehen komplexe mehrstufige Adressprotokolle. Dabei werden die Daten aus dem Befehl mit Informationen aus den Adresstabellen unterschiedlich verknüpft.

In der Vergangenheit entstand so ein ganzer Zoo unterschiedlicher Adressierungsarten. Je nach Hersteller eines Computers besaßen Befehlssätze unterschiedliche Adressierungsarten. Manche dieser Protokolle sind gleich, manche sich fast ähnlich und manche komplett verschieden. Es kommt vor, dass je nach Hersteller konzeptionell gleiche Verfahren zur Adressierung unterschiedliche Namen und unterschiedliche Verfahren gleiche Namen haben. Das macht es nicht immer einfach, die Literatur zu studieren. Ein Versuch, die grundlegenden Konzepte zu extrahieren, führt auf fünf grundlegende Adressierungsarten: die inhärente Adressierung, also die einem Befehl innewohnende Adressspezifikation, die unmittelbare, die direkte, die indirekte und die relative Adressierung. Jede dieser fünf Verfahren spaltet sich ggf. noch in Unterverfahren auf, je nachdem, ob bestimmte Adressberechnungen vor einem Speicherzugriff oder erst nach diesem erfolgen.

Um das Verhalten eines Befehls in Bezug auf seine Operanden und die Speicheradressierung möglichst kompakt zu formulieren, führen wir die abkürzenden Schreibweisen aus Tab. 6.5 ein.

6.1.4.1 Inhärente Adressierung

Bei der inhärenten Adressierung spezifiziert der Opcode des Befehls direkt den Operandenzugriff. Beispielsweise benötigt der Befehl *push* zum Ablegen von Daten auf dem Stapel kein Argument. Auch die Operation *lda* bezieht sich bezüglich ihres ersten Arguments inhärent auf das Register Akkumulator. Viele klassische Mikroprozessoren mit einem 2-Adress-Befehlssatz nutzen die Technik der inhärenten Adressierung häufig, um das erste Argument direkt als Quelle und Ziel der Operation zu spezifizieren.

Tab.6.5 Abkürzende Schreibweise der Speicheradressierung

Abkürzung	Bedeutung	Abkürzung	Bedeutung
o	Opcode	w	Wortbreite
r	Registeradresse	l	Adressraumlänge 2^l
m	Speicheradresse	O	Befehlssatz
$z = o \ z$	1-Adress-Befehl	R	Registersatz
$z = o \ x$	2-Adress-Befehl	F	FiFo-Stack
$z = o \ x \ y$	3-Adress-Befehl	M	„Random-access memory“
i	Unvollständige Konstante (immediate)	S	SRAM
j	Vollständige Konstante (immediate)	D	DRAM
a	Vollständige Speicheradresse	$s(d_l)w$	„Sign extension“
d	Unvollständige Speicheradresse (Displacement)	$\{o_i \ z_j \ x_k \ y_l\}_w$	Befehlswort der Breite w mit der Operation o und den Argumenten z , x und y
c	Verzweigungsbedingungen	$\{o.a\}_i$	Inhärente Adressierung des Arguments a
u,v	Verwaltungsfelder	$x_k \{x, y\}_l$	Bitlängen und Bitbreiten
l	Befehlswortlänge		

► Beispiel 6.1.5 – Inhärente Adressierung

Ein typisches Befehlswort einer inhärenten Adressierung hat die Form $o.r8_8$. Das bedeutet, das Befehlswort hat eine Länge von 8 Bit und das Feld für den Opcode selbst ist 8 Bit breit. Alternativ könnte man ein Nibble für den Befehl und ein Nibble für die Adressierungsart reservieren. Ein solches Befehlswort sähe wie folgt aus: $\{o4.r4\}_8$. Weitere Beispiele sind:

```

1  ||| push.acc    # push accumulator
2  ||| pop.acc   # pop accumulator
3  ||| pha        # push accumulator
4  ||| pla        # pop accumulator
5  ||| plp        # pop processor status
6  ||| php        # push processor status
7  ||| inx        # increment x-register
8  ||| dex        # decrement x-register

```

Listing 6.1: Inhärente Adressierung

Definition 6.1.12 – Inhärente Adressierung

Der Opcode eines Befehls ist eindeutig und spezifiziert die Adressierungsart. Sei o ein für den Opcode reserviertes Bitfeld der Breite k in einem Befehlswort, dann beschreiben $\{o.z\}_k$ einen Opcode mit inhärenter Adressierung des Arguments z und $\{o.x.y\}_k$ einen Opcode mit inhärenter Adressierung der Argumente x und y .

6.1.4.2 Implizite oder unmittelbare Adressierung

Wird das Argument des Befehls direkt im Speicherwort abgelegt, spricht man von der impliziten Adressierung (immediate). Bei der Programmierung wird häufig zwischen Variablen und Konstanten unterschieden. Befehle, deren Argumente konstant sind, lassen sich mithilfe der impliziten Adressierung kompakt speichern. Bei einer begrenzten Wortbreite des Rechners erstreckt sich das Befehlswort über mehrere Speicherzellen. Das unmittelbar im Befehlswort gespeicherte Argument muss also direkt nach dem Laden des Opcodes durch einen erneuten Speicherzugriff gelesen werden. Das alles muss vor der eigentlichen Ausführung der Operation erfolgen (☞ Abb. 6.3).

Definition 6.1.13 – Unmittelbare Adressierung

Sei c ein k -breites Bitfeld in einem beliebigen Befehlswort der Breite w . Ist $k = w$, definiert c_k direkt ein Argument dieses Befehls. Ist $k \leq w$, definiert $s(c_k)$ ein vorzeichenerweitertes Argument des Befehls.

► Beispiel 6.1.6 – Implizite Adressierung

Der Ausdruck $(\{o_4.a_2.r2\}, c_8)_8$ ist ein Beispiel für ein solches Befehlswort. Dabei handelt es sich um einen 2-Adress-Befehl, bei dem das erste Argument inhärent spezifiziert und das zweite Argument direkt der zu verwendende Operand sind. Die folgenden Ladebefehle veranschaulichen das Laden einer konstanten Zahl in den Akkumulator (☞ Abb. 6.4):

```
1 ||  ld.acc  8    # load accumulator
2 ||  lda     8    # load accumulator
```

Listing 6.2: Implizite Adressierung



6.1.4.3 Direkte oder absolute Adressierung

In Programmen sind aber nicht alle Operanden von Befehlen Konstanten. Im Gegenteil, die meisten Daten, auf die der

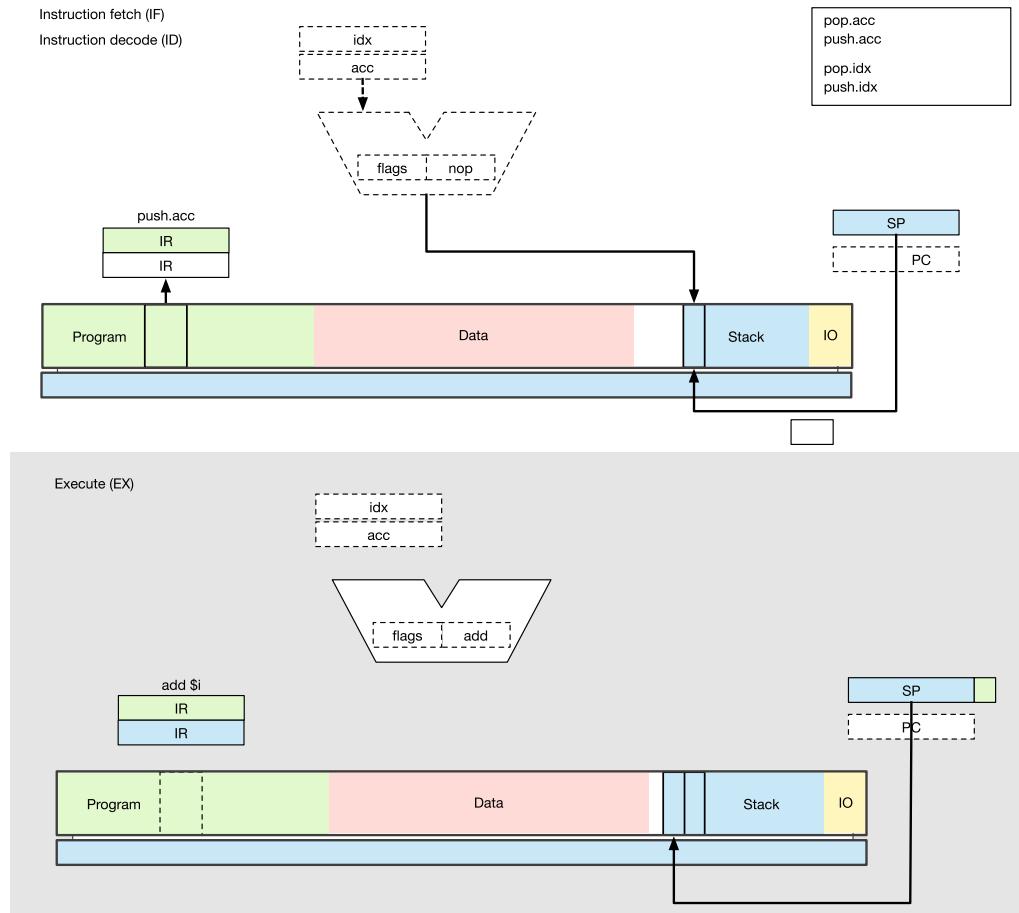


Abb. 6.3 Implizite Adressierung

Prozessor zugreift, sind variabel und ändern sich während der Ausführung – insbesondere wenn noch nicht klar ist, wie viele Variablen während der Programmausführung benötigt werden. Wenn also der Speicher während der Laufzeit alloziert wird, sollte der Adressbereich der dynamisch wachsenden Daten vom statischen Bereich des Programms und seiner Konstanten getrennt werden. Die Variablen stehen dann an einer beliebigen Stelle des Speichers und werden über eine Adresse im Befehl angesprochen. Dies geschieht mit der direkten Adressierung. Diese heißt oft auch absolute Adressierung. Der im Speicherwort abgelegte Wert wird nun als Adresse im Speicher interpretiert. An dieser Stelle befindet sich dann das gesuchte Datum (Abb. 6.5).

6.1 · Architektur des Befehlssatzes

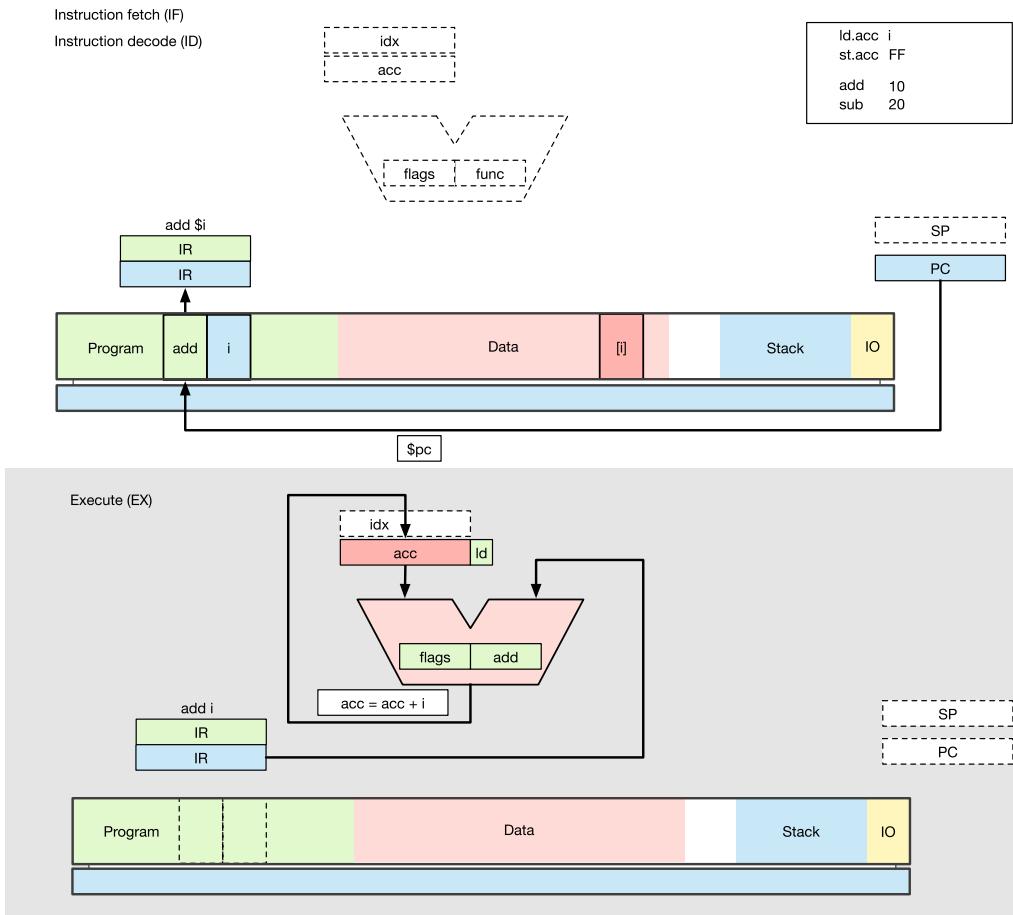


Abb. 6.4 Adressierung von Konstanten

Definition 6.1.14 – Registerdirekte Adressierung

Sei r ein k -breites Bitfeld in einem beliebigen Befehlswort der Breite w , dann definiert $R[r_k]_w$ ein Speicherwort eines Registersatzes der Breite w an der Stelle $R[r_k]_w$ als Argument dieses Befehls.

Beispiel 6.1.17 – Registerdirekte Adressierung

Sei $\{o, z, x, y\}$ das Befehlswort einer 3-Adress-Maschine mit den Argumenten z, x, y und der Operation o . Dann wird in registerdirekter Adressierung wie folgt auf die Argumente zugegriffen: $R[z] = R[x]oR[x]$.

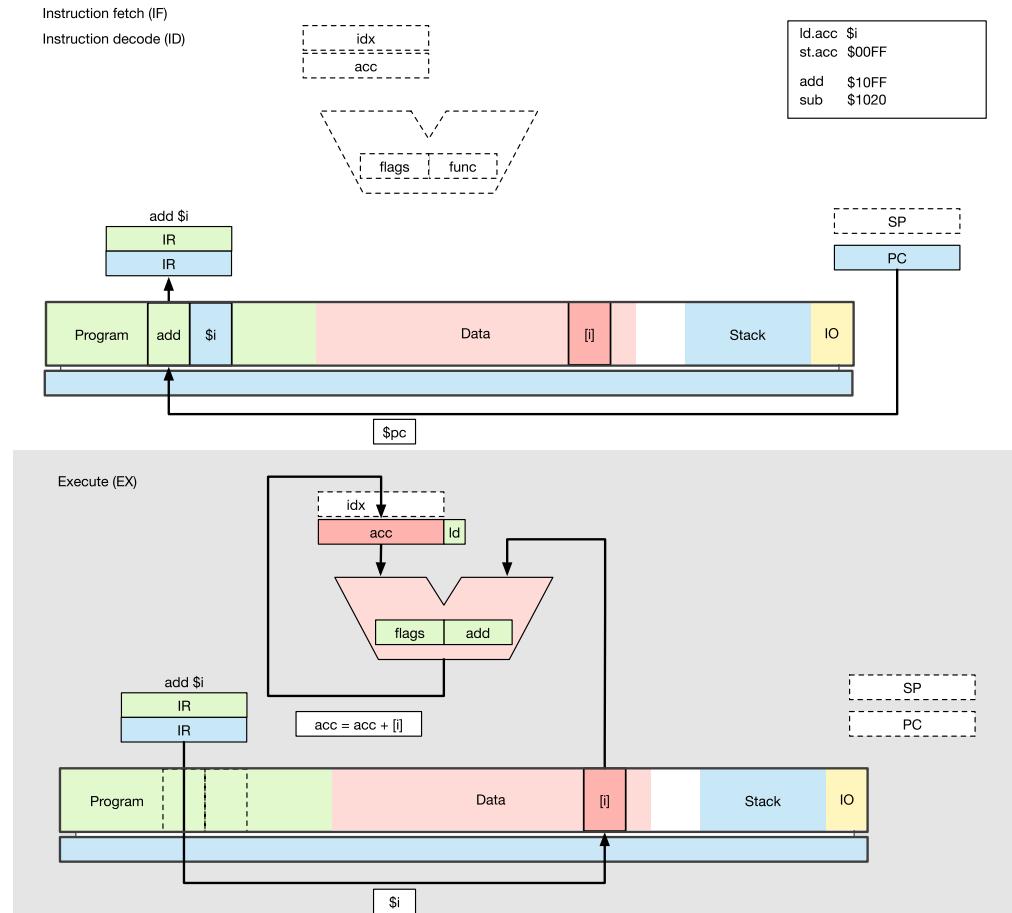


Abb. 6.5 Direkte Adressierung

```
1 ||      add z, x, y
2 ||      sub z, x, y
```

Oft wird in Assembler-Schreibweise der Zugriff auf Registeradressen gesondert gekennzeichnet. In der MIPS-Architektur mit einem Dollarsymbol:

```
1 ||      add $z, $x, $y
2 ||      sub $z, $x, $y
```

Und in der SPARC-Architektur mit Prozentzeichen:

```
1 ||      add %z, %x, %y
2 ||      sub %z, %x, %y
```



Definition 6.1.15 – Speicherdirekte Adressierung

Sei m ein l -breites Bitfeld in einem beliebigen Befehlswort, dann definiert $M[m_l]_w$ ein Speicherwort eines wahlfreien Speichers der Breite w an der Stelle m_l als Argument dieses Befehls.

► Beispiel 6.1.8 – Speicherdirekte Adressierung

Sei $\{o, z, x, y\}$ das Befehlswort einer 3-Adress-Maschine mit den Argumenten z, x, y und der Operation o . Dann wird in speicherdirekter Adressierung wie folgt auf die Argumente zugegriffen: $M[z] = M[x]oM[y]$. ◀

► Beispiel 6.1.9 – Mehrteiliges Befehlswort

Der Befehl $(\{o_4.a_2.r_2\}, d_8)_8$ einer Maschine mit 8 Bit langem Adressraum (255 Speicherstellen) und einem 8 Bit breiten Datenwort muss insgesamt drei-mal auf den Speicher zugreifen, bevor alle Argumente des Befehls zur Verfügung stehen: Zunächst muss der Opcode geladen und decodiert und interpretiert werden, dann muss die Adresse hinter dem Opcode gelesen und erneut an den Speicher angelegt werden. Erst jetzt kann das Datum an dieser Adresse gelesen und zur Berechnung verwendet werden. Die Befehle

```
1 || 1d.acc $0x8      # load accumulator
2 || lda     $0x8      # load accumulator
```

Listing 6.3: Mehrteiliges Befehlswort

laden jeweils ein Datum von der Stelle 0x8 des Speichers in den Akkumulator. Dabei drückt das \$-Zeichen in der Assembler-Notation aus, dass es sich um eine direkte Adressierung handelt, die hexadezimale Zahl anschließend also als Adresse zu interpretieren ist. Im oben genannten Fall ist die Befehlausführung drei-stufig. Dagegen wäre die Ausführung des Befehlswortes $(\{o_4.a_2.r_2\}, d_8)_{16}$ nur zwei-stufig. ◀

Hat ein Befehl die gleiche Wortbreite wie die Speicherstelle, in der er abgelegt ist, stehen nicht alle Bits des Befehlswortes zur Adressierung der Argumente zur Verfügung. Erfolgt der Zugriff auf Register, ist dies nicht schlimm, da die Anzahl der Register häufig so gewählt wird, dass die Argumente des Befehls vollständig in ein Befehlswort passen. Wenn aber eine beliebige Stelle im Speicher adressiert werden soll, ist dies nicht möglich, da Teile des Befehlswortes durch den Opcode belegt sind. Bei vielen Maschinen mit geringer Wortbreite ist die Lösung, dass die Breite des Befehlswortes ein Vielfaches der Breite der Speicherstellen ist. Der Befehl belegt demnach mehr als eine Speicherstelle. In einem solchen Fall ist die Ausführung des Befehls mehrstufig. Zunächst wird der erste Teil des Befehls geladen und der Befehl decodiert. In Abhängigkeit vom Opc-

de oder weiterer Felder im Befehlswort lädt dann die Maschine weitere Teile des Befehls. Die Ausführung benötigt daher mehrere Befehlszyklen. Deren Anzahl ist oft erst nach dem Lesen und Decodieren des Opcodes und seiner Argumente im ersten Teil des Befehlswortes bekannt.

6.1.4.4 Indirekte Adressierung

Häufig verwalten Programme Tabellen. Verweisen die Inhalte dieser Tabellen wiederum auf weitere Speicherstellen, an denen das Datum erst gespeichert ist, vereinfacht die indirekte Adressierung das Schreiben solcher Programme. Bei der indirekten Adressierung findet sich an der im Befehl angegebenen Speicherstelle nicht das zu verarbeitende Datum, sondern eine weitere Adresse, an der dieses im Speicher zu finden ist. Nach dem Laden des Befehls müssen somit noch mindestens zwei weitere Stufen von Speicherzugriffen durchlaufen werden. Die indirekte Adressierung ist eine einfache Möglichkeit, in einem kleinen Adressraum Programme im Speicher kompakt zu halten. Maschinen mit geringer Befehlswortbreite besitzen häufig einen speziellen Speicherbereich, die Zero-Page. Auf den Adressbereich 0x00 bis 0xFF kann dabei mit einem 8-Bit-Argument direkt zugegriffen werden. Legt man nun in diesem Speicher die längeren Zieladressen ab, so kann der Befehl einfach auf diese Adresse verweisen, und die indirekte Adressierung erledigt das Auffinden des Operanden. Da die Zero-Page schnell angesprochen werden kann, verkürzen sich das Programm und seine Ausführungszeit merklich, insbesondere wenn die dort gespeicherten Tabellen häufig geändert werden.

Definition 6.1.16 – Registerindirekte Adressierung

Sei r ein k -breites Bitfeld in einem beliebigen Befehlswort, dann definiert $M[R[r_k]]_w$ ein Speicherwort eines wahlfreien Speichers der Breite w an der Stelle $R[r_k]_w$ als Argument dieses Befehls.

► Beispiel 6.1.10 – Registerindirekte Adressierung

Gegeben sei das Befehlswort eines 2-Adress-Rechners mit 8 Bit Wortlänge und 8 Bit Adressraum: $(\{o_4, r_4\}, m_8)_8$. Da es sich um einen 2-Adress-Befehl handelt, ist das erste Argument das inhalt im Befehl spezifizierte Register, und auf das zweite Argument wird die registerindirekte Adressierung angewendet: $R[r] = R[r] \circ M[R[r] + m_8]$. Zunächst werden also der Inhalt des Registers gelesen, der Wert aus dem Befehlswort zu diesem Wert addiert und das Ergebnis als Adresse an den Speicher angelegt. ◀

Definition 6.1.17 – Speicherindirekte Adressierung

Sei m ein l -langes Bitfeld in einem beliebigen Befehlswort, dann definiert $M[M[m]_w]_w$ ein Argument in einem wahlfreien Speicher der Breite w an der Stelle $M[m]_w$ als Argument dieses Befehls.

Bei der indirekten Adressierung sind noch mehr Befehlszyklen oder Stufen notwendig als bei der direkten Adressierung. Erst muss der Befehl geholt und decodiert werden, dann muss eines seiner Argumente ausgewertet werden, um dann erneut den Inhalt einer Speicherzelle zu laden. Mit diesem Datum als Adresse muss dann wieder eine Zeile des Speichers gelesen werden, bevor das Argument des Befehls bei der Rechenoperation verwendet werden kann (☞ Abb. 6.6).

6.1.4.5 Indizierte Adressierung

Häufig sollen in Programmen immer wieder die gleichen Rechenoperationen mit unterschiedlichen Parametern ausgeführt werden. Häufiges fehlerfreies Wiederholen der immer gleichen Anweisungen macht Computer so hilfreich, da einfache und damit langweilige Operationen von der Maschine immer und immer wieder mit einer großen Geduld abgearbeitet werden. Mithilfe bedingter Sprünge lassen sich zu diesem Zweck Schleifen programmieren. Schleifen haben oft eine Zählvariable, die einfach hoch- oder heruntergezählt wird. Eine solche Lauf- oder Zählvariable wird am besten in einem eigenen Register gespeichert. Ein solches Register heißt Indexregister oder kurz X-Register (manchmal auch IX oder IDX). Für dieses Register sind in einem Befehlssatz häufig die Befehle Dekrement (Herunterzählen) und Inkrement (Hochzählen) definiert. Eine Schleife kann nun mit einem Speicherzugriff relativ zu dem Indexregister implementiert werden, indem der Inhalt des Registers zu einer im Befehl direkt angegebenen Adresse addiert wird. Liegt an dieser Adresse eine Liste, kann mithilfe der Zählvariablen indiziert auf die Liste zugegriffen werden (☞ Abb. 6.7).

Definition 6.1.18 – Indizierte Adressierung

Seien m und r jeweils ein l - oder k -langes Bitfeld in einem beliebigen Befehlswort, dann definiert $M[R[r] + m]$ ein Argument in einem wahlfreien Speicher an der Stelle $M[R[r] + m]$. Das Register r wird häufig Indexregister genannt. In vielen Befehlssätzen wird das Indexregister häufig inhärent adressiert: $M[o.r + m]$.

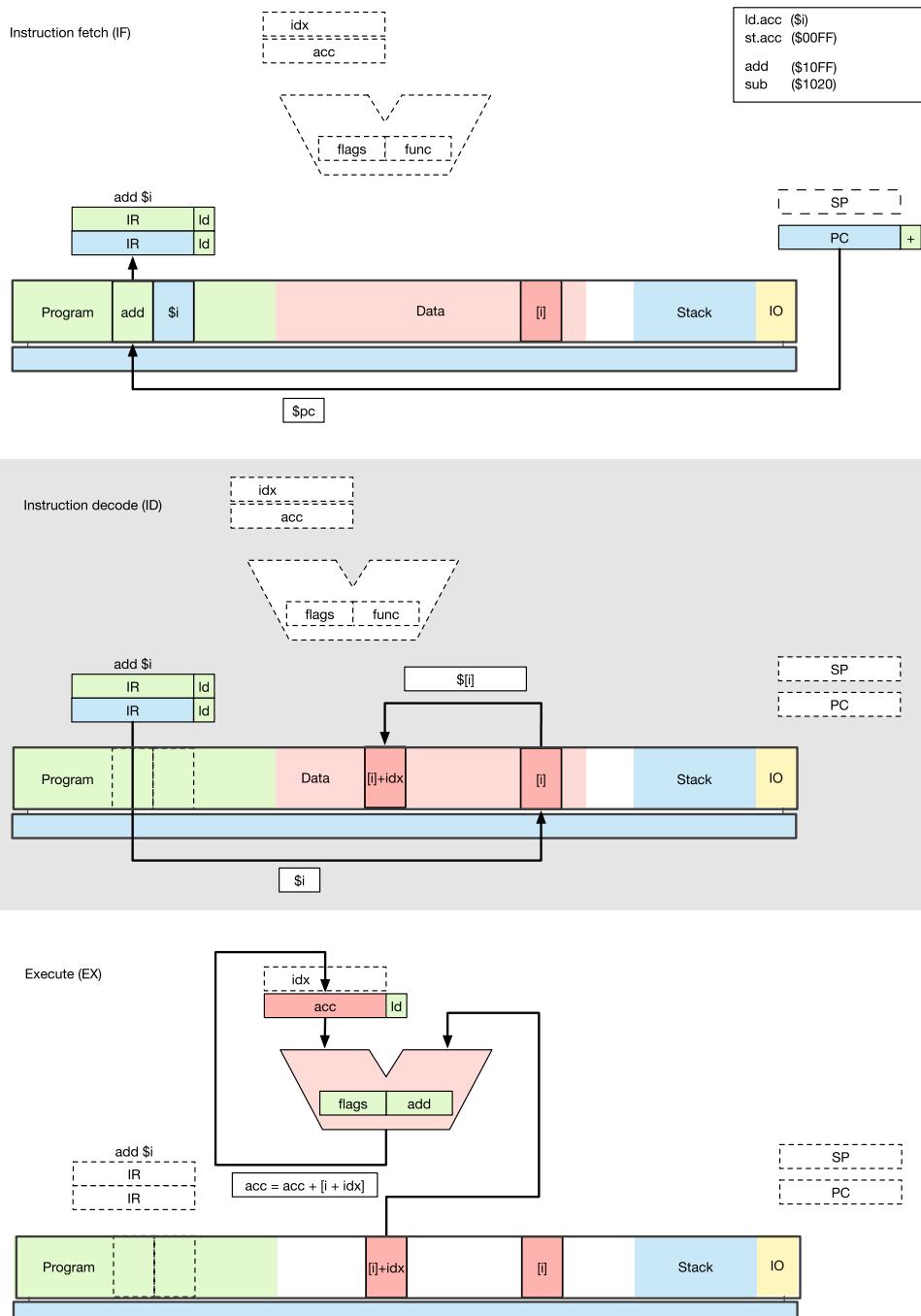


Abb. 6.6 Indirekte Adressierung

6.1 · Architektur des Befehlssatzes

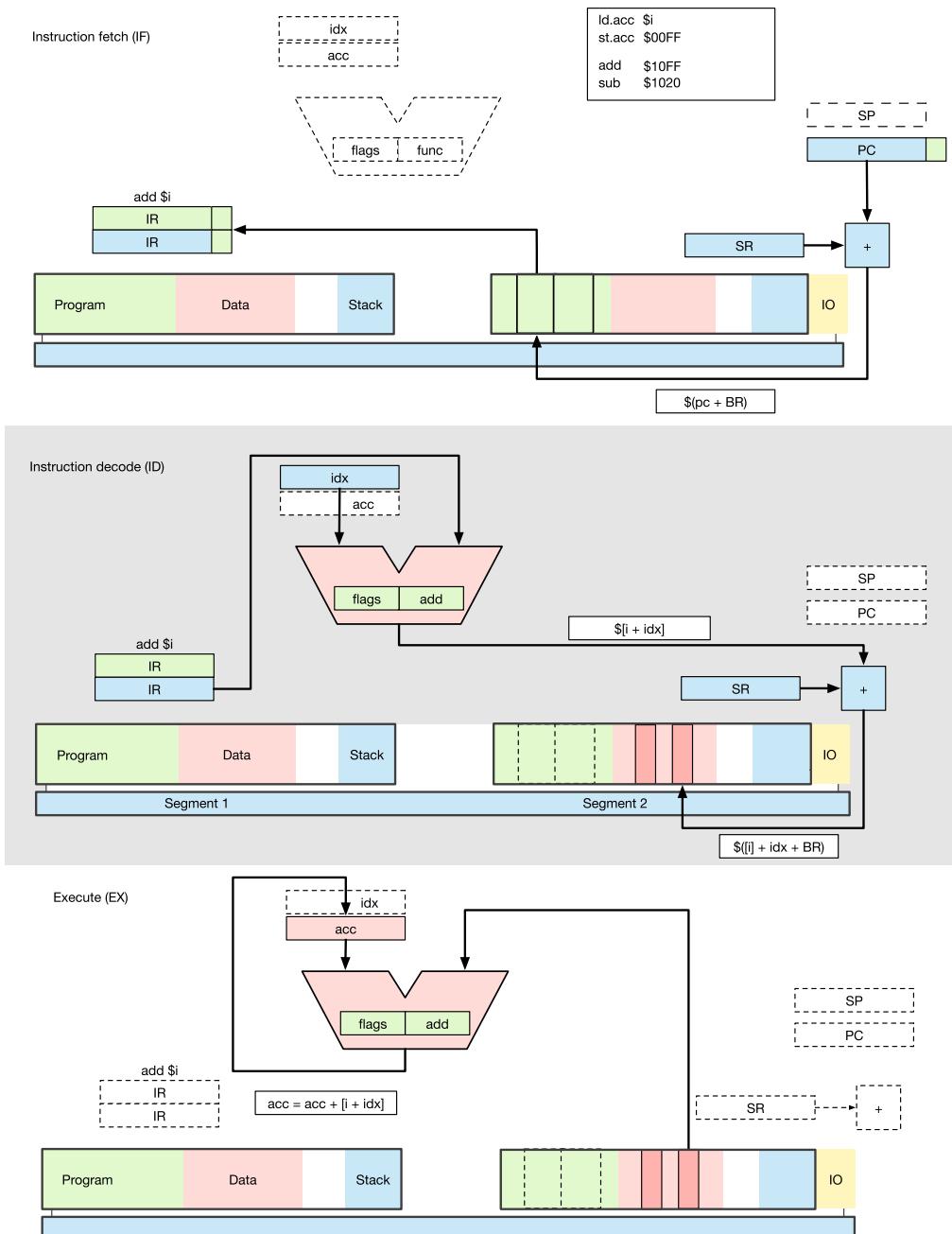


Abb. 6.7 Registerindizierte Adressierung

► Beispiel 6.1.11 – Indizierte Adressierung

Das Format eines Befehls mit indizierter Adressierung könnte wie folgt aussehen: $(\{o_4.a_2.r_2\}, d_8)_{16}$. Dabei wird das Indexregister mit dem 8. Bit spezifiziert. Beispiele für Befehle mit indizierter Adressierung sind:

1	lda \$0x8,x #load accumulator
2	adc \$0x8,x # add \$0x8,x to accumulator and store result

Listing 6.4: Indizierte Adressierung

6

Damit können Laufvariablen von Schleifen direkt im Indexregister implementiert werden. Das Zählen der Laufvariablen erfolgt dann direkt auf dem Indexregister. Viele Rechner besitzen mehrere Indexregister, das X- und das Y-Register. Die Verwendung zweier Indexregister wird am Beispiel der Fibonacci-Reihe in □ Abb. 7.3 demonstriert.

Es gibt sogar Befehlssätze mit automatischer Änderung des Indexregisters nach einem Zugriff. Da dann auf einen oder mehrere explizite Zählbefehle für das Indexregister verzichtet werden kann, lässt sich Programmspeicher einsparen. Nachteilig ist jedoch, dass derartige Indexregister nicht mehr universell eingesetzt werden können.

6.1.4.6 Relative Adressierung

Neben dem Zählen auf einem Index muss bei Schleifen noch geprüft werden, ob die Schleifenbedingung erfüllt ist. Ist das der Fall, wird häufig ein Sprung zu einer neuen Adresse durchgeführt. Bei der Bearbeitung von Sprüngen macht man sich zunutze, dass der Rumpf einer Schleife kurz ist, verglichen mit der Länge des Programms. Es muss also im Adressraum nicht weit gesprungen werden. Dies hat dann kurze Sprungadressen zur Folge. In dem Fall reicht es aus, im Befehl ein kurzes Sprungziel zu definieren und so Programmspeicherplatz zu sparen. Da immer relativ zum aktuellen Programmzähler gesprungen werden muss, ist es erforderlich, das im Befehl spezifizierte Sprungziel auf den Programmzähler zu addieren oder von diesem zu subtrahieren. Die Adressierung erfolgt somit relativ zum Inhalt des Befehlszählers.

Definition 6.1.19 – Registerrelative Adressierung

Seien r ein ausgezeichnetes Register der Adresswortlänge k und d ein Bitfeld der Breite l in einem beliebigen Befehlswort. Ist $l \leq k$, dann spezifiziert $r_k + s(d)$ eine neue Adresse. Die Funktion $s(d)$ heißt Vorzeichenerweiterung („sign extension“).

Es mag verwirren, zwei unterschiedliche Arten einer auf den Inhalt eines Register bezogenen Adresse einzuführen. Die registerdirekte und die registerindirekte Adressierung werden hauptsächlich für Argumente zum Auffinden von Daten eingesetzt, während die relative Adressierung Sprünge bezogen auf einen gegebenen Befehlszählerstand ermöglicht. Erstes ermöglicht das Schreiben von verschiebbaren Programmen, deren Daten immer bezogen zu einer festen Basisadresse abgespeichert sind, Letzteres eine dynamische Änderung der Bezugsadresse während der Ausführung eines Programms.

► Beispiel 6.1.12 – Befehlszählerrelative Adressierung

Gegeben seien die Befehle $(\{o_4.r_4\}, d_8)_8$ und $(\{o_4.r_4\}, d_8)_{16}$. Der Befehl

```
1 ||| bne      $0xFF    # branch if any condition is
|||          notequal
```

Listing 6.5: Befehlszähler-Relative Adressierung

Ist immer relativ zum Befehlszähler auszuführen, also $0xFF + pc$. Im Allgemeinen kann eine relative Adressierung in Bezug zu jedem beliebigen Register erfolgen: $(\{o_6, r_5\}, d_{21})_{32}$. Dieses Befehlsformat besitzt einen 6 Bit langen Opcode, einen 5 Bit langen Zeiger auf ein Register in der Registerdatei und eine 21 Bit lange Adresse (Displacement). ◀

6.1.5 Befehlssatz

In den vorherigen Abschnitten wurden die grundlegenden Arten von Befehlen beschrieben. Jede Operation kann dabei unterschiedlich auf ihre Argumente zugreifen: entweder inhärent, implizit, direkt, indiziert, indirekt oder absolut. Jede Kombination einer Operation mit den jeweiligen Adressierungsarten definiert in einem Befehlssatz einen Befehl. Zudem kann jeder Befehl eine unterschiedliche Anzahl von Argumenten haben. Maschinen werden dann nach der maximalen Anzahl von Argumenten in Befehlen ihres Befehlssatzes klassifiziert. Die drei Aspekte Operation, Anzahl und Anordnung der Operanden und der für jeden Operanden notwendigen Adressierungsart werden im Befehlswort vereint. Befehlssätze können daher viele unterschiedliche Befehlsworte besitzen und bei hinreichend vielen möglichen Adressierungsarten für unterschiedliche Argumente sehr komplex werden. Das Zusammenspiel zwischen Operationstyp, den dadurch notwendigen Befehlsformaten und dem Zugriff auf den Speicher bildet dann den vollständigen Befehlssatz einer Maschine: die Befehlssatzarchitektur oder kurz die Architektur eines Rechners.

Befehlssatzentwurf

Der Entwurf eines Befehlssatzes ist gar nicht so einfach, wie es zunächst scheinen mag. Natürlich lassen sich alle bisher diskutierten Prinzipien in einen Topf werfen und dann beliebig kombinieren. Am Ende des Tages steht aber die Frage, ob dies sinnvoll oder elegant ist. Besser: ob es effizient ist und der Befehlssatz leicht mithilfe einer Mikroarchitektur implementiert werden kann. Die Notwendigkeit eines Befehls wird vom Rechnerarchitekten definiert. Programmierer wünschen kurze und übersichtliche Programme. Die Programme sollen schnell geschrieben und leicht lesbar sein. Das bedeutet, sie sollen in ihrem Aufbau dem Algorithmus ähneln, den sie implementieren. Auf der anderen Seite steht die Maschine. Der am Ende zu bauende Rechner soll leicht zu warten, zuverlässig und schnell sein. Leider stehen diese Anforderungen, die Elektronik des Programms und die Effizienz der Maschine im Widerspruch zueinander. Effiziente und kostengünstige Hardware ist oft unbequem zu programmieren. Eine Maschine, die elegante Programme erlaubt, ist dagegen komplex. Diese Komplexität macht die Maschine teuer, unzuverlässig und erlaubt keine späteren Erweiterungen. Einen eleganten und effizienten Befehlssatz zu entwerfen ist eine Kunst. Dabei sind die notwendigen Befehle und ihre Argumente geschickt den richtigen Befehlsformaten zuzuordnen, und zwar derart, dass die spätere Hardware, die Mikroarchitektur, die Befehle mit wenig Aufwand entschlüsseln und ausführen kann.

Ein Programmierer möchte immer den Datentyp und die Adressierungsart einsetzen, der oder die für sein Problem oder den zu schreibenden Algorithmus angemessen ist. Programmierer wünschen sich also Befehlssätze, die für jeden Befehl jede mögliche Adressierungsart unterstützen. Ein solcher Befehlssatz, der alle Datentypen und alle Adressierungsarten für alle Operationen anbietet, ist bequem und einfach zu nutzen. Dem steht aber der Aufwand bei seiner Implementierung durch die logischen Schaltungen entgegen. Jede Variante eines Befehls benötigt einen eigenen Opcode, und jeder dieser zahlreichen Opcodes muss von der Steuerung decodiert werden. Wenn alle Operationen in Kombination mit allen Adressierungsarten als Befehle angeboten werden, heißt ein Befehlssatz orthogonal.

Definition 6.1.20 – Orthogonaler Befehlssatz

Wenn ein Rechner eine bestimmte Anzahl von Adressierungsarten und Datentypen unterstützt, ist sein Befehlssatz orthogonal, wenn alle Operationen alle Datentypen und Adressierungsarten implementieren.

In der Praxis wird man eine derartige Maschine selten finden, insbesondere da nicht alle Operationen die maximale Anzahl von Argumenten benötigen und nicht jede Adressierungsart bei jeder Operation sinnvoll ist. So werden für Operationen auf dem Stapel keine komplexen Adressierungsprotokolle benötigt. Daher sind die meisten praktisch eingesetzten Rechner partiell orthogonal.

Orthogonale und partiell orthogonale Befehlssätze unterstützen viele Adressierungsarten. Um die Befehle zu codieren, werden daher viele unterschiedliche Befehlsformate genutzt. Wenn viele unterschiedliche Eigenschaften beliebig kombiniert werden, nennt man einen Befehlssatz komplex. Aus diesem Grund heißen Rechner mit derartig komplexen Befehlssätzen „complex instruction set computer“ (CISC)..

Bis in die 1980er-Jahre wurden die Befehlssätze immer komplexer. Dies führte dazu, dass die inzwischen eingeführten Compiler zur Übersetzung von einem in einer Hochsprache geschriebenen Programm nicht mehr alle angebotenen Befehle bei der Übersetzung nutzten. Gleichzeitig wurde der inzwischen eingeführte Halbleiterspeicher immer günstiger, sodass Compiler die Abbildung der Hochsprache in ein Assembler-Programm optimieren konnten. Diese technologischen und konzeptionellen Fortschritte führten dazu, dass immer weniger der mühsam vom Rechnerarchitekten und von den Ingenieuren implementierten Befehle genutzt wurden. Die Einführung neuer Entwurfsverfahren in der Halbleitertechnik führte zusätzlich dazu, dass Rechnerarchitekten begannen, ihre Entwürfe selbst in Silizium zu gießen. Mehrere Forschungsprojekte führten zu einfacheren Befehlssätzen, die optimal von Compilern unterstützt werden konnten. Die Entwicklung zeigte, dass sich mit einem reduzierten Befehlssatz mit wenigen homogenen Befehlsformaten und einer beschränkten Anzahl von Adressierungsarten die Rechengeschwindigkeit einer Maschine um Größenordnungen steigern ließ. Da diese Rechner viele Befehle in kurzer Zeit bearbeiten konnten, fiel es kaum ins Gewicht, dass ein Compiler mehr Befehle zur Übersetzung eines in Hochsprache geschriebenen Programms nutzen musste, als dies ein menschlicher Assembler-Programmierer tun würde. Rechner mit homogenen Befehlsworten und mit beschränkter Anzahl von Adressierungsarten heißen „reduced instruction set computer“ (RISC).

Lässt man die Forderung fallen, dass Programmierer die Anwendungsprogramme in der gleichen Sprache schreiben wie die verwendete Maschine rechnet, ist es möglich, die beiden Anforderungen Eleganz des Programms und Effizienz der Maschine zu vereinen. Das Programm wird nun in einer Hochsprache geschrieben. Ein solches Programm heißt Quellprogramm (engl. „source code“). Ein Programm, genannt

CISC

RISC

Compiler

Compiler, übersetzt dann das Quellprogramm in den Assembler-Code oder das Zielprogramm des entsprechenden Prozessors. Dieser Code wird dann von einem Assembler in die von der Maschine ausführbaren Befehle umcodiert (engl. „object code“). Aus einer einfachen Anweisung in Hochsprache wird dann ggf. eine umfangreiche Folge von Befehlen in der Sprache der Maschine. Da diese Befehlsfolgen für die unterschiedlichen Datentypen wie Schablonen von dem Compiler genutzt werden können, wird also ein Teil der Programmkomplexität von der Hardware in die Software, den Compiler verschoben. Mit dem Aufkommen von günstigen Massenspeichern in Halbleitertechnik löste sich gleichzeitig das Argument, dass Code möglich kompakt zu speichern sei, auf.

Ein nicht zu unterschätzendes Problem in der Computer-technik sind die hohen Kosten, die das Schreiben der Software verursacht. Unternehmen, die viel Geld in die Programmierung von Software investiert haben, möchten diese auch weiter nutzen, wenn eine neue Hardware eingeführt wird. Dies führt dazu, dass sich Hersteller von Computern genötigt sehen, einmal eingeführte Befehle auch in neuen Generationen der Rechner weiter zu unterstützen. Um eine derartige Befehlssatzkompatibilität sicherzustellen, sind Prozessoren auf dem Markt, die selbst und automatisch komplexe CISC-Befehle in Sequenzen von RISC-Befehlen umsetzen und diese dann von einem RISC-Kern ausführen lassen. Durch dieses Vorgehen lassen sich die Vorteile beider Welten in einer Maschine vereinen.

Komplexe Befehlssätze

Trotz der Einführung der RISC werden Befehlssätze immer komplexer. Dies scheint ein Widerspruch, der sich aber auflöst, wenn man der Ursache auf den Grund geht. Vor 50 Jahren bestand die Komplexität des Befehlssatzes aus der Kombination Operation, Datenformat, Befehlsformat oder Argumentanzahl und Adressierungsart. Moderne Befehlssätze besitzen aber neue Funktionen. Die Erweiterungen des Befehlssatzes sind also nicht technischer Natur, sondern ergeben sich aus Funktionserweiterungen. Aufwendige Berechnungen wie die Codierung und Verschlüsselung von Video- oder Audioströmen werden heute direkt von der Hardware durchgeführt. Videostreaming und Internettelefonie sind zeitkritische Anwendungen. Sie werden erst durch die direkte Ausführung einiger Schlüsselfunktionen direkt in Hardware möglich. Die neue Komplexität der Befehlssätze führt somit zu einer breiteren Anwendung der entwickelten Prozessoren.

6.1.6 Datentypen

Jede Form von Daten, die im Speicher in Form von Bitmustern abgelegt werden, muss ebenfalls von der Maschine interpretiert

werden. Dabei ist zwischen Zahlen und Zeichen zu unterscheiden. Bei Zahlen müssen sich Maschine und Programmierer einig sein, welches Bit das höherwertige und welches Bit das niedrigerwertige des Datums ist. Darüber hinaus ist die Zahl als vorzeichenbehaftete oder vorzeichenlose Zahl zu interpretieren. Das Zahlenformat der Bitmuster ist daher eindeutig festzulegen. In der Regel werden diese Zuordnungen vom Rechnerarchitekten vorgenommen. Neben dieser rein funktionell-logischen Sichtweise auf die Daten spielen auch technische Gesichtspunkte beim Speichern eines Datums eine Rolle. Welche Datenwortbreite hat der Rechner? Werden die Datenworte in einer Speicherzelle abgelegt, oder werden mehrere Zeilen benötigt?

Die Zeilenbreite eines Rechners wird als Vielfaches von der Basis der binär Zahlen, also 2 angegeben. Vier Bit werden als Nibble bezeichnet, 8 Bit als ein Byte, 16 bit als Wort (engl. „word“), 32 Bit als Doppelwort (engl. „double word“) und 64 Bit als langes Wort (engl. „long word“). Üblich sind aber auch die Bezeichnungen Byte, half Word, Word, double Word für 8, 16, 32 und 64 Bit breite Daten. Ein Datum kann entweder in Zweierkomplementdarstellung als signed Integer oder aber als einfache Bitfolge ohne Vorzeichen als unsigned Integer gespeichert werden. □ Tab. 6.6 gibt einen Überblick über die Darstellung natürlicher oder ganzer Zahlen im Rechner. Zeichen werden dabei als unsigned Integer in einem Byte gespeichert. Oft nennt man eine binäre Ziffernfolge, die ein Zeichen codiert, ein Character (Char) und eine Folge von Zeichen eine Zeichenkette (engl. „string“).

Nicht unerheblich für den Programmierer und die Maschine ist die Reihenfolge der Bits in einem Datum. Bei einer binären Ziffernfolge muss Einigkeit darüber bestehen, ob das höchstwertige (most significant Bit, MSB) oder das niedrigstwertige Bit (least significant Bit, LSB) ganz rechts oder ganz links im Datenwort stehen. Dies muss vom Rechnerarchitekten eindeutig festgelegt werden, und der Programmierer muss sich beim Schreiben von Programmen streng danach richten. Unglücklicherweise sind sich Rechnerarchitekten über die Lage des niedrigstwertigen Bits nicht einig. Manche Architekturen schreiben das LSB auf die rechte, andere wiederum auf die linke Seite eines Datums. Programmierer müssen, wenn sie das erste Mal ein Programm für eine neue Maschine schreiben, im Handbuch des Prozessors ermitteln, wie ein Datum vom Rechner interpretiert wird. Unterschieden wird daher zwischen LSB 0 und MSB 0. Das bedeutet, dass das Bit mit der Nummer 0 im Datenformat des Prozessors das LSB oder das MSB ist.

Tab. 6.6 Datentypen

Bezeichnung	Größe (Bit)	Name	Vorzeichen	Wertebereich
Char, Byte	8	uint8	Unsigned	[0,255]
Char, Byte	8	int8	Signed	[-128,127]
Word, half Word, short Word	16	uint16	Unsigned	[0,65.535]
Word, half Word, short Word	16	int16	Signed	[-32.768,32.767]
Word, double Word	32	uint32	Unsigned	[0,4.294.967.295]
Word, double Word	32	int32	Signed	[-2.147.483.648,2.147.483.647]
Long Word	64	uint64	Unsigned	[0,18.446.744.073.709.551.615]
Long Word	64	int64	Signed	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]
Octaword	128	uint128	Unsigned	[0,3.40.282.1038]
Octaword	128	int128	Signed	[-1.70.141.1038,1.70.141.1038]

Definition 6.1.21 – MSB 0

Gegeben sei eine binärcodierte Ziffernfolge mit N Bit. Bei der MSB-0-Codierung ist das höchstwertige Bit an der 0-Stelle im Befehlswort gespeichert. Der dezimale Wert der Zahl berechnet sich mit

$$\sum_{i=0}^{N-1} z_i \cdot 2^{N-1-i}$$

Definition 6.1.22 – LSB 0

Gegeben sei eine binärcodierte Ziffernfolge mit N Bit. Bei der LSB-0-Codierung ist das niedrigstwertige Bit an der 0-Stelle im Befehlswort gespeichert. Der dezimale Wert der Zahl berechnet sich mit

$$\sum_{i=0}^{N-1} z_i \cdot 2^i$$

Aber nicht nur die Reihenfolge der Bits im Datum spielt eine Rolle. Beispielsweise könnte in einem Speicher der Breite 8 Bit ein Wort mit 16 Bit abgelegt werden oder in einem 16-Bit-Speicher eine Zahl im 64-Bit-Format. In einem solchen Fall muss die Reihenfolge der Bytes in dem überlangen Datum ebenfalls eindeutig festgelegt werden. Eine solche Spezifikation heißt Bytereihenfolge (engl. „byte order“ oder „endianness“). Es wird zwischen den Formaten little- und big-endian unterschieden.

Endianness

Definition 6.1.23 – Big-endian

Seien M_n ein Speicher mit n Elementen und wahlfreiem Zugriff und $D_m \subset M_n$ ein Datum mit mehreren Elementen, dann ist D_m im Format little-endian (engl. für kleinendig) gespeichert, wenn das niedrigstwertige Byte an der niedrigsten Adresse von D_m lokalisiert ist.

Definition 6.1.24 – Big-endian

Seien M_n ein Speicher mit n Elementen und wahlfreiem Zugriff und $D_m \subset M_n$ ein Datum mit mehreren Elementen, dann ist D_m im Format big-endian (engl. für großendig) gespeichert, wenn das höchstwertige Byte an der niedrigsten Adresse von D_m lokalisiert ist.

Bei der Speicherung eines Datums im Umfang mehrerer Bytes ist die Lage der ersten Adresse der Werte entscheidend. Obwohl die Architektur moderner Rechner von 32-Bit- oder 64 Bit Speicherworten ausgeht, erfolgt die Zählweise der Speicherstellen oft noch byteorientiert. Daher können 4 Byte lange Datenworte beliebig im Speicher stehen. Es ist aber geschickt, die Speicherworte immer an Bytadressen abzulegen, die Vielfache der Adressierungszählweise sind – im Falle einer 32-Bit-Architektur als Vielfache von 4. Manchmal ist dies im Sinne einer effizienten Nutzung des Speichers nicht immer möglich. In einem solchen Fall muss entweder die Hardware, meist aber die Software dafür sorgen, dass das Datum korrekt interpretiert wird. Der Programmierer einer Maschine hat also ggf. die Ausrichtung der Informationen im Speicher zu berücksichtigen. Es wird zwischen ausgerichteten und ungerichteten Daten unterschieden.

Alignment

Definition 6.1.25 – Ausgerichtetes Datum (alignment)

Seien M_n ein Speicher mit n Elementen und wahlfreiem Zugriff und $D_m \subset M_n$ ein Datum mit mehreren Elementen, dann ist der Wert im Speicher ausgerichtet (engl. „alignment“) wenn die Adresse i ein ganzzahliges Vielfaches von n ist, d. h., i hat die Eigenschaft $i \bmod n = 0$.

Datenfelder mit einer Startadresse, die obige Eigenschaft nicht besitzen, sind unausgerichtet (engl. „misalignment“).

6.2 Speichermodell

Bisher unterschieden wir zwischen zwei Arten von Daten- und Programmspeichern, dem Stapspeicher und dem Speicher mit wahlfreiem Zugriff. Der Stapspeicher kam in der Vergangenheit aus technologischen Gründen in den ersten einfachen Rechnern und Taschenrechnern zum Einsatz. Es zeigte sich aber bereits früh, dass in den meisten Anwendungsfällen ein wahlfreier Speicher bequemer zu programmieren ist. In den Fällen, für die ein Stapel gebraucht wird, kann diese Funktion mit wahlfreiem Speicher einfach emuliert werden. Man könnte jetzt denken, der Speicher eines Computers ist nur ein Feld oder eine Reihe von nummerierten Speicherplätzen oder Speicherstellen. In jedem dieser linear organisierten und durch eine eindeutige Nummer adressierbaren Speicherworte liegen dann die gespeicherten Daten. Aus Sicht des Programmierers ist dieses einfache Speichermodell vertretbar. Dies gilt auch bei der Betrachtung der Komplexität von Software. Dabei geht man davon aus, dass die Zugriffszeit einer Stelle im wahlfreien Speicher immer gleich ist. Für einen Rechnerarchitekten ist es nur leider nicht einfach, einen Speicher zu organisieren, der sich exakt so verhält, wie ein Programmierer es erwartet. Das hat im Wesentlichen drei Gründe:

- Leider ist die Zugriffszeit auf die Einträge eines wahlfreien Speichers beeinflusst von der Anzahl der adressierbaren Wörter in der Speichermatrix. Und zwar steigt diese Zeit quadratisch mit der Länge der Leitung. Damit kann wahlfreier Speicher nicht beliebig groß werden, ohne mit erheblichen strukturellen und technologischen Maßnahmen die Zugriffszeit zu reduzieren.
- Programme können größer sein als der zur Verfügung stehende Hauptspeicher. Sie müssen daher in Teile von der Größe des Hauptspeichers aufgeteilt werden.

- Mehrere Programme können im Hauptspeicher stehen. Der Zugriff auf einzelne Anwendungen muss so organisiert werden, dass sich die Software nicht gegenseitig stört.

Der erste Punkt spielt für Computer mit Stapspeichern keine Rolle. Der Zugriff auf den Stapspeicher ist schnell, muss doch immer nur der Wert gelesen werden, der oben auf diesem liegt. Beim Schreiben ist lediglich der Stapelzeiger zu erhöhen, um dann die Daten auf dem Stapel abzulegen. Stapspeicher eignen sich für Programme, die zu groß für den Speicher sind. Gleches gilt, wenn ein Rechner mehrere Anwendungen gleichzeitig bearbeitet muss. Im ersten Fall müsste immer ein kompletter Stapel ein und ausgelagert werden, im zweiten benötigte man für jedes Programm einen eigenen Stapspeicher.

Definition 6.2.1 – Halbleiter-Speicherzelle

Elektronische Schaltung zum Speichern eines Bits.

Definition 6.2.2 – Speichermatrix

Eine Speichermatrix ist ein zweidimensionales Feld aus Speicherzellen, die in mehreren Zeilen und Spalten angeordnet sind.

Definition 6.2.3 – Speicherwort

Eine Zeile in einer Speichermatrix, die mit einem Index ausgewählt wird, heißt Speicherwort, Speicherstelle oder Speicher-element. Wird ein solches Wort angesprochen, kann auf alle Speicherzellen dieser Zeile gleichzeitig zugegriffen werden.

Definition 6.2.4 – Adresse

Ein Index in einem Speicherfeld zum gleichzeitigen Zugriff auf alle Einträge einer Speicherzeile oder eine Speicherreihe heißt Adresse.

Logisch gesehen ist der Kern⁴ oder Hauptspeicher (engl. „main memory“) eines Computers eine logische Liste von Speicherwörtern, auf die indiziert zugegriffen wird. Die Gesamtheit aller Speicherelemente oder Speicherstellen, die mit unterschiedlichen Adressen oder Indizes angesprochen werden können,

⁴ Die Zellen der ersten Hauptspeicher waren mit Magnetkernen implementiert.

wird Adressraum genannt. Sind die Adressen monoton steigend, ist dieser linear.

Definition 6.2.5 – Linearer Adressraum

Eine Liste von Speicherwörtern mit monoton steigenden Indizes bildet einen linearen Adressraum.

Mit den vorangegangenen Definitionen sollte es einfach sein, einen beliebig großen Hauptspeicher mit wahlfreiem Zugriff aufzubauen. Das Prinzip ist simpel und die Größe des Speichers hängt nur von der zur Verfügung stehenden Fläche auf den integrierten Schaltungen und der Anzahl der zur Verfügung stehenden Adressbits zusammen. Das Hinzufügen von 1 Bit zum Adresswort verdoppelt den zur Verfügung stehenden Speicher. Wenn also genug Platz für Speicherzellen vorhanden ist, kann dieser auch genutzt werden. Leider ist Fläche nicht das einzige Entwurfskriterium für einen Speicher. Entscheidend ist auch die Zugriffszeit. Ideal ist es, wenn ein Rechner ein Datum anfordert und dieses möglichst instantan zur Verfügung steht. Dies ist nicht möglich, sodass die Zugriffszeit auf den Speicher verglichen mit der Rechenzeit kurz sein sollte. Einen solchen Speicher nennt man auch einen schnellen Speicher.

In der Anfangszeit der Computer war der Zugriff beschränkt. Es gab zu dieser Zeit noch keine einfachen Röhrenschaltungen, um kompakte logische Verstärker, die Gatter, zu bauen. Eine Adressdecodierung für einen wahlfreien Speicher war sehr aufwendig. Daher waren die ersten elektronischen Rechner als Zählmaschinen (Akkumulatormaschine) und dann später als Stapelmaschinen implementiert. Mit dem Aufkommen der Magnetkernspeicher konnte erstmals ein linearer Speicher, bestehend aus mehreren Zeilen parallel geschalteter Magnetkerne realisiert werden. Allerdings steigt die Zugriffszeit dann mit dem Quadrat der Länge der Bitleitungen. Je größer also ein Adressraum ist, umso länger ist die Zugriffszeit. Eine Verdopplung des Adressraumes bedeutet dann eine Vervierfachung der Zugriffszeit. Der Rechnerarchitekt kann demnach nicht nur den Speicher vergrößern, sondern muss dabei die Auswirkungen auf die Rechengeschwindigkeit berücksichtigen. Bedingt durch den nicht linearen Zusammenhang zwischen Adressraumgröße und Zugriffszeit ist es notwendig, den Hauptspeicher physisch geschickt zu organisieren, und zwar so, dass die Zugriffszeit nicht beliebig lang wird. Aus diesem Grund entspricht die physisch-strukturelle Organisation des Speichers nicht dem einfachen algorithmischen Modell linear angeordneter Speicherzellen.

Definition 6.2.6 – Haupt- oder Kernspeicher

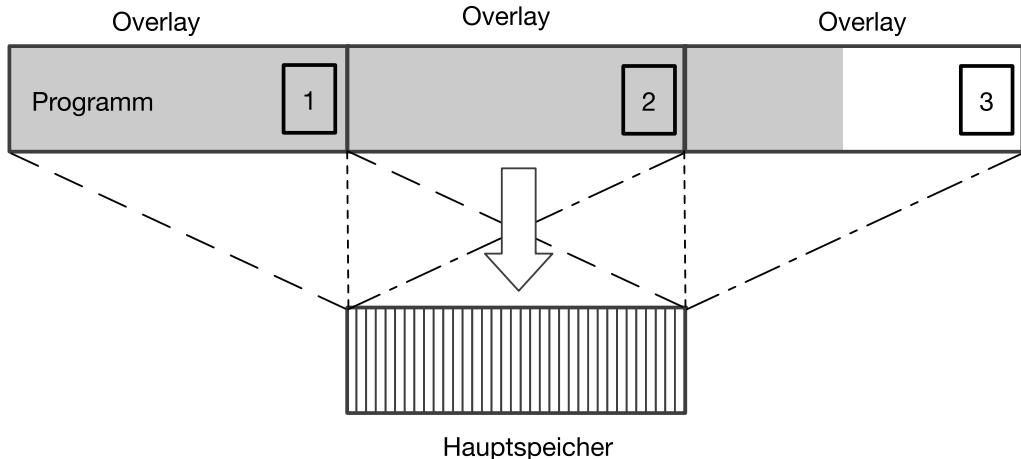
Einen Speicher mit wahlfreiem Zugriff, in dem die gerade auszuführenden Programme mit ihren Daten vorgehalten werden, heißt Haupt- oder Kernspeicher. Der Rechner greift direkt auf den Hauptspeicher zu.

Je größer ein linearer und wahlfreier Adressraum ist, umso länger dauert der Zugriff auf ein Speicherwort. Zwar ist die Zugriffszeit immer gleich, aber dafür steigt diese für alle Speicherworte nicht linear mit der Größe des Speichers. Dies bedeutet, Speicher in einer vorgegebenen Technologie – SRAM oder DRAM – oder eines bestimmten Herstellungsprozesses wird mit jedem frei adressierbaren Wort langsamer. Das Problem, dass der Zugriff auf den Speicher mit zunehmender Größe immer länger wird, kann technologisch durch Einführung neuer Herstellungsprozesse mit kleineren Strukturgrößen und technisch durch Einführung schaltungstechnischer Tricks gelöst werden. Dem Rechnerarchitekten bleibt nur die geschickte Partitionierung des Adressraumes. Eine solche Aufteilung des Speichers ist die Adressraumverschränkung. Diese erlaubt es den Speicherplatz zu verdoppeln, ohne den Adressraum zu vergrößern und so die Zugriffszeit nur leicht anwachsen zu lassen.

Leider wuchs der Speicherbedarf der Programme schneller, als physischer Hauptspeicher kostengünstig gebaut werden konnte. Da mit den Lochstreifen und Magnetbändern günstige Hintergrundspeicher zur Verfügung standen, lagerte man Inhalte, die nicht mehr im Speicher vorgehalten werden konnten, auf diese Datenträger mit linearem Zugriff aus. Für die Programmierer ist es aber unpraktisch, beim Entwurf eines Programms immer in unterschiedlichen Technologien zu denken und je nach Programmteil andere technische Primitive einzusetzen. Das gilt ebenfalls in modernen Computern, in denen die Massenspeicher durch Festplatten und Magnetbänder realisiert werden.

Wenn ein Programm nicht in den Hauptspeicher passt, teilt man es einfach auf und lagert die Teile, die nicht mehr benötigt werden, auf einen externen Massenspeicher aus. Diese Technik heißt Programmüberlappung (engl. „overlay“). Dabei wird ein Programm in mehrere Overlays aufgeteilt (Abb. 6.8), und der Programmierer muss am Ende eines solchen Programmteils entweder manuell oder durch eine kleine Hilfsroutine die nicht mehr benötigten Teile des Programms auslagern und die neuen Teile in den Speicher laden. Funktioniert dieses Prinzip sehr gut mit der ersten Generation der Hardware, bekommt ein Programmierer mit dem Kauf einer neuen Maschine ein Pro-

Overlays



6

Abb. 6.8 Aufteilung eines Programms in Overlays

Seiten

blem: Die manuell auf die zur Verfügung stehenden physischen Ressourcen zugeschnittene Software ist für den neuen Computer umzuschreiben. Selbst wenn nun das Programm komplett in den Speicher passt, sind die Overlays bei der Portierung der Anwendung neu anzupassen. Im schlimmsten Fall müssen Overlays neuer Größe entworfen werden. Um eine transparente Verwaltung des Speichers, die auch bei der Portierung von Software unveränderlich ist, zu erreichen, wurde das Konzept des virtuellen Speichers mit einer Kachel- oder besser Seitenverwaltung (engl. „pages“) entwickelt (Abb. 6.9).

Segmente

Durch die hohe Verfügbarkeit von Halbleiterspeichern wurde es möglich, immer größeren Hauptspeicher zu bauen. Dadurch waren irgendwann mehr Speicherplätze vorhanden, als von den Programmen benötigt wurden. Es entstand der Wunsch, gleichzeitig mehrere Anwendungen im Speicher betriebsbereit vorzuhalten. Dazu muss der Hauptspeicher passend unterteilt werden, sodass es für jede Anwendung wirkt, als hätte sie den Speicher exklusiv zur Verfügung. Dazu ist ein Programm so zu schreiben, dass es an jeder beliebigen Stelle des Hauptspeichers stehen kann. Programmadressen müssen also relativ zu einer virtuellen Startadresse und nicht absolut codiert werden. Um eine solche adressunabhängige Programmierung zu unterstützen, besitzen moderne Maschinen hardwareseitige Mechanismen zur logischen Verwaltung des Speichers. Dabei wird gleich berücksichtigt, dass der Speicher nicht nur aus flüchtigem Halbleiterspeicher besteht, sondern auch nicht flüchtige Medien wie Festplatten als Hintergrundspeicher angeschlossen sind. Ein wichtiges Entwurfskriterium bei der Einführung einer derartigen Speicherverwaltung ist die Anforderung, dass Software Speicher häufig dynamisch anfor-

6.2 · Speichermodell

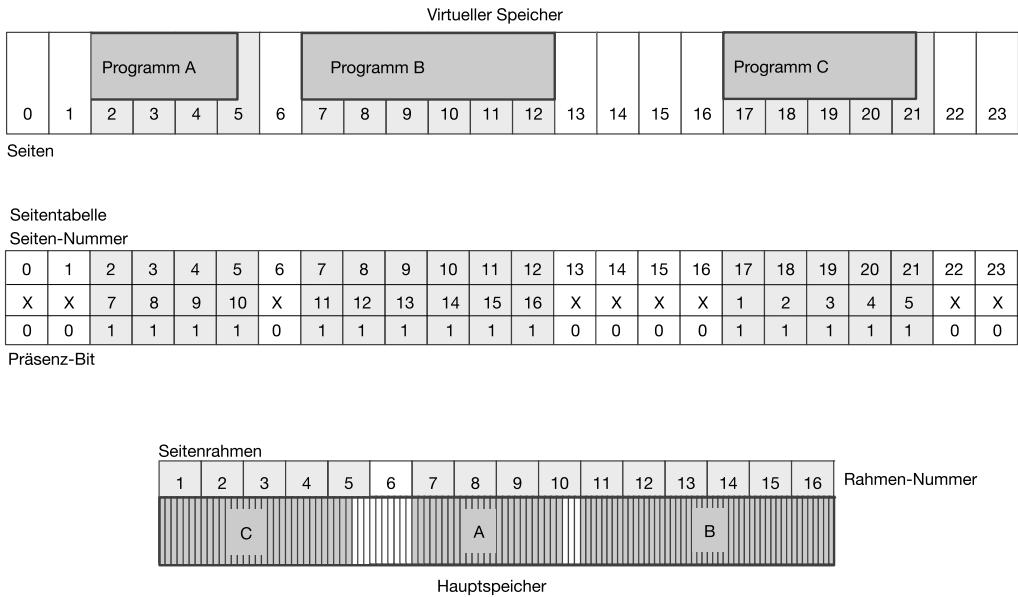


Abb. 6.9 Prinzip der Seitenverwaltung

dert, d.h., es ist zur Zeit der Programmierung gar nicht bekannt, wie viel Speicher das Programm benötigen wird. Insbesondere müssen sowohl der Stapel als auch die Programmdaten des Programms dynamisch verwaltet werden. Um den obigen Anforderungen zu genügen, wird ein einzelnes Programm in Segmente aufgeteilt. Typische Segmente sind das Code- oder Textsegment mit dem Programmcode und statischen Daten, das Datensegment mit den dynamischen Daten und das Stacksegment für den Stapel (Abb. 6.10).

Ein großer Hauptspeicher ist langsam, da ein großer Speicher einen großen Adressraum und daher lange Leitungen besitzt. Es ist daher erforderlich, den Adressraum aufzubrechen

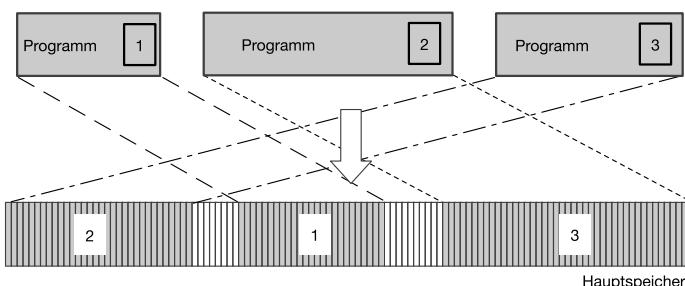


Abb. 6.10 Segmentierung

und in kleinere Teile zu zerstückeln. Das Verschränken des Adressraumes ist nur bis zu einem gewissen Grad nützlich und kann nicht beliebig gesteigert werden, da auch parallele Adressräume über Leitungen angesprochen werden müssen. Hinzu kommt, dass die Daten durch Multiplexer auszuwählen sind, und je mehr Bänke vorhanden sind, umso größer wird die Schaltung des Multiplexers. Die vom Rechner zu generierende Adresse wird dann in immer kleinere Teile zerlegt. Bis zu einer gewissen Größe des Speichers ist auch die Anzahl der vom Betriebssystem zu verwaltenden Programme überschaubar. Mit zunehmender Speichergröße und komplexer werdender Speicherstruktur oder -architektur soll die technische Struktur der Speicherverwaltung vor dem Programmierer, sei es dem Betriebssystemprogrammierer oder dem Anwendungsprogrammierer, verborgen werden. Programme sollten portabel sein, d. h., sie müssen auf Maschinen mit unterschiedlicher Speicherorganisation lauffähig sein. Diese Forderung nach Portabilität, Flexibilität und Reduktion der Komplexität der Hardware aus Sicht des Programmierers führte zur einer Speicherhierarchie mit einem virtuellen Adressraum. In dieser werden Speicher mit unterschiedlichen physischen Eigenschaften so geschickt kombiniert, dass sie für das Programm transparent wirken, also wie ein großer und schneller nicht flüchtiger Speicher. Ziel aller folgenden architektonischen Maßnahmen ist es, dem Programmierer einen linearen, wahlfreien Speicher mit konstanter und geringer Zugriffszeit unter Berücksichtigung der elektrotechnischen Rahmenbedingungen bereitzustellen.

6.2.1 Adressraumverschränkung

In einem Speicher mit wahlfreiem Zugriff sind die Speicherstellen linear angeordnet. Über den Adressdecodierer wird eine Wortleitung angesprochen, und alle an dieser Leitung liegenden Speicherzellen legen ihren Inhalt auf die jeweilige Bitleitung. Das bedeutet, die Wortbreite des Speichers bestimmt die Ausdehnung der Wortleitung und die Anzahl der Adressen die Länge der Bitleitungen. Da die Laufzeit quadratisch von der Leitungslänge abhängt, folgt aus einer Verdopplung des Adressraumes eine Vervierfachung der Ansprechverzögerung. Ein Speicher kann offensichtlich nicht einfach vergrößert werden, ohne die Geschwindigkeit der Zugriffsoperationen negativ zu beeinflussen.

Das Problem kann reduziert werden, wenn man bei der Verdopplung des Adressraumes nicht einfach die Anzahl der Speicherplätze verdoppelt, sondern eine Adresse mehrfach nutzt, also Speicherelemente parallel schaltet und erst in einem zweiten Schritt das richtige Speicherwort mithilfe eines Mul-

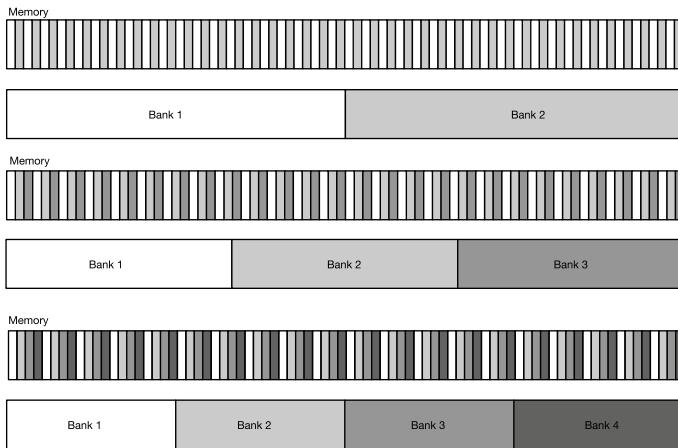


Abb. 6.11 Idee der Adressraumverschränkung

tiplexers auswählt. Der Multiplexer kann durch ein eigenes Feld, den Index im Adresswort gesteuert werden. Eine einzelne Speichermatrix eines so aufgeteilten Speichers heißt Bank, und das Verschränken mehrerer Bänke wird im angelsächsischen Sprachgebrauch Banking oder „interleaving“ genannt. Der Rechner legt beim Banking eine Adresse an und muss ggf. warten, bis Wort- und Bitleitung eingeschwungen sind. Im Anschluss lassen sich aber durch einfaches Erhöhen des Indexes im Befehlswort schnell die unterschiedlichen Bänke ansprechen. Daten werden bei der Adressraumverschränkung also nicht linear im Adressraum abgelegt, sondern verschränkt in den Bänken (Abb. 6.11). Ein unter einer Adresse in mehreren Bänken abgelegtes Datum heißt Speicherblock (engl. „memory line“).

Definition 6.2.7 – Bank

Eine Bank oder besser Speicher- oder Memorybank ist ein linearer Adressraum, in dem jede Adresse 8 Speicherzellen adressiert. An jeder Adresse einer Bank ist daher 1 Byte gespeichert.

Wird der Adressraum durch n Bänke verschränkt, ist die lineare Adresse ein Index, der auf n Byte zeigt. Diese n Byte heißen Speicherwort.

Definition 6.2.8 – Speicherwort im verschränkten Adressraum

Sei n die Anzahl der Bänke eines Speichers, dann ist $n \cdot b$ das Speicherwort (engl. „line“) in einem verschränkten Adressraum.

Die meisten modernen Computer adressieren den Speicher immer noch byteweise. Das bedeutet, der Programmierer kann auf jedes Byte einzeln zugreifen. Die technische Organisation des Speichers macht es jedoch erforderlich, den linearen Adressraum in Bänke zu unterteilen. Daher muss man sich auf der Ebene der Maschine vom Konzept der linearen als einfache, monoton steigenden Folge von Speicherelementen verabschieden. Die Adresse besteht nun aus einem linearen Teil zur Anwahl des Speicherwortes und einem eigenen Index zum Ansprechen der jeweiligen Bank. Das Adresswort eines Prozessors wird in diesem Fall in zwei unabhängige Felder aufgeteilt.

6.2.2 Seitenverwaltung

Durch Verschränken des Adressraumes steigt die Zugriffszeit auf den Speicher bei einer Vergrößerung der Anzahl der Speicherstellen moderat. Das Verfahren ist für einen Rechnerarchitekten von elementarer und für Anwendungsprogrammierer von eher untergeordneter Bedeutung. Autoren von Anwendungssoftware benötigen einen transparenten großen und linearen Speicher. Dabei sollten sich Daten auch permanent speichern lassen. Dies ist nur mit unterschiedlichen Speicher-technologien möglich. Programmierer wiederum wollen sich aber nicht mit verschiedenen Technologien auseinandersetzen und möchten Programme technologieunabhängig schreiben. Daher wurde bereits in den 1950er-Jahren die Seitenverwaltung eingeführt. Diese automatisiert die Technik der Overlays und macht sie so für den Programmierer transparent.

Um den logischen Speicher eines Programms technologie-unabhängig zu machen, trennt man diesen konzeptionell vom physischen. Zu diesem Zweck wird ein virtueller Adressraum eingeführt. Dieser wird in eine feste Anzahl gleich großer Partitionen unterteilt. Eine Partition des virtuellen Adressraumes heißt Kachel oder Seite (engl. „page“) und umfasst mehrere Speicherworte. Eine typische Größe für eine Seite sind etwa 512 Byte bis zu 64 kB. Ein virtueller Adressraum von 64 kB kann in 16 Seiten zu 4 kB unterteilt werden. Im physischen Speicher wird immer nur ein Teil des virtuellen Adressraumes vorgehalten. Im einfachsten Fall ist die Größe einer Seite gleich der Größe des physischen Speichers. Die Größe ist aber in der

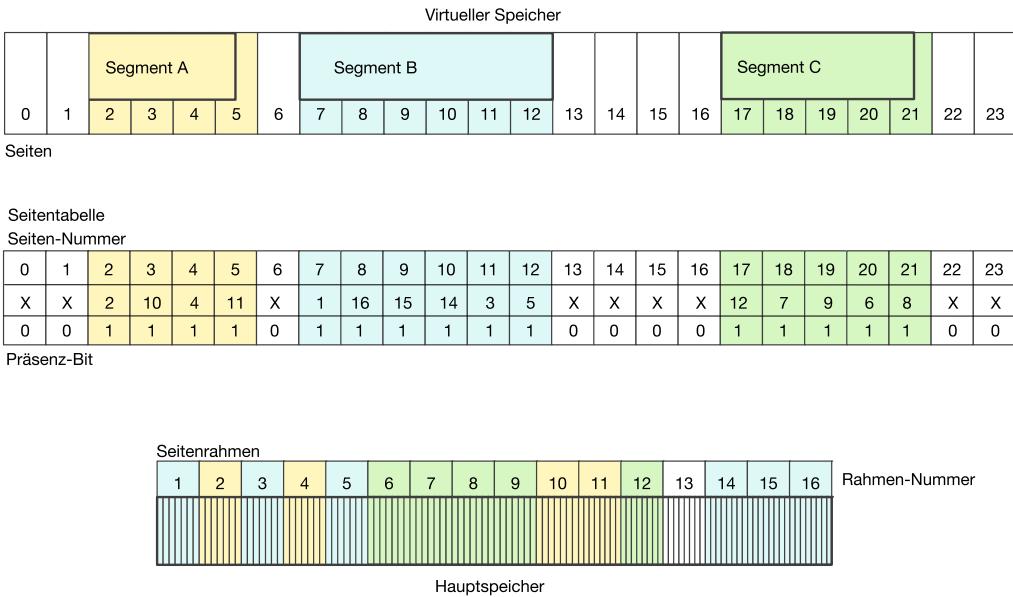


Abb. 6.12 Seitenverwaltung

Praxis unzweckmäßig, insbesondere da mit günstigen Halbleiterspeichern inzwischen auch ein großer physischer Speicher aufgebaut werden kann. Der physische Speicher selbst sollte daher ebenfalls partitioniert sein. Diese Partitionen haben die gleiche Größe wie die Seiten und werden Seitenrahmen (engl. „page frame“) genannt. Wie in Abb. 6.12 gezeigt, ist es nicht notwendig, linear aufsteigende Seiten linear in den Seitenrahmen abzulegen. Vielmehr kann die Zuordnung willkürlich erfolgen. Die Abbildung der Seiten zu den Seitenrahmen geschieht über eine Seitenliste genannte Verwaltungsstruktur.

Die Größe der Seiten und Seitenrahmen richtet sich nach dem physischen Hauptspeicher. Der physische Adressraum ist ein ganzzahliges Vielfaches der Seitengröße. Damit ist der virtuelle Adressraum ebenfalls ein ganzzahliges Vielfaches der Seiten. Dabei sollte die Seite selbst größer sein als ein Speicherwort, um den Aufwand beim Verschieben in nicht flüchtige Massenspeicher zu minimieren.

Definition 6.2.9 – Seiten und Seitenrahmen

Eine Seite oder Kachel ist eine Partition eines virtuellen Adressraumes. Alle Seiten haben die gleiche Größe und passen vollständig in den virtuellen Speicher. Jeder Seite ist eine eindeutige Nummer, die Seitennummer zugeordnet.

Definition 6.2.10 – Seitenrahmen

Ein Seitenrahmen ist eine Partition eines physischen Speichers. Die Seitenrahmen bestehen aus der gleichen Anzahl physischer Speicherworte wie die Seiten. In einem Seitenrahmen kann daher exakt eine Seite des virtuellen Adressraumes gespeichert werden. Seitenrahmen eines physischen Speichers sind mittels Seitenrahmennummern linear durchnummeriert.

Definition 6.2.11 – Virtueller Adressraum

Ein logischer, nicht real vorhandener Adressraum, auf den ein Programm zugreifen kann. Der virtuelle Adressraum besteht aus gleich großen Seiten, und jede Seite besitzt eine eindeutige Seitennummer. Der virtuelle Adressraum wird über die Seitennummer adressiert.

Eine Adresse zum Ansprechen eines Speicherblocks in einem byteorganisierten Speicher besteht nun aus der Blocknummer, einem Index auf das auszuwählende Speicherwort und einen weiteren Index auf das jeweilige Byte.

Das Konzept, einen virtuellen Speicher in Seiten aufzuteilen und diese in Seitenrahmen zu speichern, entspricht einer Verallgemeinerung der Technik der Overlays. Dadurch lässt sich das Ein- und Auslagern von Seiten in den oder vom Hauptspeicher automatisieren. So wirkt der langsame Massenspeicher zusammen mit dem Hauptspeicher auf den Programmierer wie ein großer linearer nicht flüchtiger Speicher. Letzten Endes müssen aber alle Daten des virtuellen Adressraumes auch physisch gespeichert werden. Ohne einen nicht flüchtigen großen und preiswerten Hintergrundspeicher ist dies nicht möglich. Solche Hintergrundspeicher sind Magnetbänder, magnetische Festplatten oder Flash-Halbleiterspeicher. Bedingt durch die jeweilige physische Realisierung sind die großen Hintergrundspeicher technisch anders organisiert als der Hauptspeicher. Das Konzept des virtuellen Speichers verbirgt daher die unterschiedlichen Technologien vor dem Anwendungsprogrammierer.

Definition 6.2.12 – Hintergrundspeicher

Im Hintergrundspeicher werden Programme und Daten permanent gespeichert. Der Rechner greift nicht direkt auf diesen zu.

Werden Ausschnitte von Daten aus einem Hintergrundspeicher in einem Hauptspeicher vorgehalten, muss sichergestellt

werden, dass diese konsistent sind. Daher müssen Daten, die im Hauptspeicher durch den Rechner verändert wurden, automatisch im Hintergrundspeicher aktualisiert werden. Zu diesem Zweck müssen die Seiten des virtuellen Speichers von einem Betriebssystem verwaltet werden. Dieses führt mithilfe der Seitennummern eine Liste, welche Seiten sich im Hauptspeicher und welche sich im Hintergrundspeicher befinden. Möchte der Computer auf eine Seite zugreifen, die nicht im Hauptspeicher ist, muss diese aus dem Hintergrundspeicher in den Hauptspeicher geladen werden. Sind alle Seitenrahmen im Hauptspeicher belegt, dann hat das Betriebssystem zu entscheiden, welcher Seitenrahmen frei gemacht wird. Dazu muss eine entsprechende Seite in den Hintergrundspeicher geschrieben werden, bevor der dazugehörige Seitenrahmen mit der neu aus dem Hintergrundspeicher gelesenen Seite gefüllt wird. Dieser Vorgang wird Seitenauslagerung (engl. „paging“) genannt. Ein Zugriff des Prozessors auf eine nicht im Hauptspeicher vorhandene Seite führt zu einem Fehler und löst eine Unterbrechung (engl. „interrupt“ oder „trap“) aus. Eine solche Unterbrechung wird Seitenfehler (engl. „page fault“) genannt. Der Prozessor unterbricht dann die Programmausführung und startet Routinen des Betriebssystems, die das Auslagern der entsprechenden Seite durchführen.

Bei einem Seitenfehler muss das Betriebssystem entscheiden, welche Seite aus dem Hauptspeicher zurück in den Hintergrundspeicher zu schreiben ist. Dieser Vorgang heißt Seitenersetzung. Diese benötigt einen Algorithmus, um zu entscheiden, welche eine Seite ausgelagert werden kann. Ein solches Programm könnte beispielsweise immer die Seite auswählen, auf die am längsten nicht zugegriffen wurde (engl. „least recently used“, LRU). „Least recently used“ arbeitet nicht immer zuverlässig, minimiert aber die Anzahl der Seitenfehler.

6.2.3 Segmentverwaltung

Mithilfe der Seitenverwaltung ist es möglich, große Programme und deren Daten zu partitionieren und automatisch Teile der Software in einem kostengünstigen Hintergrundspeicher zu sichern. Allerdings fordern die meisten Anwendungen Speicher dynamisch an. Das bedeutet, dass zum Zeitpunkt der Programmerstellung noch gar nicht klar ist, wie viel Speicherplatz ein Programm benötigt. Auch die Größe des benötigten Stapsels ist abhängig vom Ablauf der Software und meist nicht vor der Ausführung vorherzusagen. Hinzu kommt, dass jede Programmausführung zu einer unterschiedlichen dynamischen Speicheranforderung führen kann.

Ein linearer Adressraum ist eindimensional. Dynamische Speicheranforderungen mehrerer Programme sind mehrdimensional. Um eine dynamische Speicherverwaltung durch den Compiler und das Betriebssystem zu unterstützen, wurde das Konzept der Segmente eingeführt. Jedem Programm wird nun ein Segment zugeordnet.

Jedes Programm besteht aus einem statischen und einem dynamischen Teil. Programm und Konstanten liegen am unteren Ende des Speichers. An diesen Teil knüpft direkt das Segment der Variablen an. Dieser Bereich kann also nach oben in den Speicher wachsen. Am obersten Ende des Speichers liegt der Stapel. Dieser kann nach unten wachsen. Der Bereich, in dem sich Stapel und Daten ausbreiten können, heißt Haufen (engl. „heap“). Ist der Heap zu klein, kann der Stapel in den dynamischen Datenbereich wachsen und so Informationen zerstören. Damit entspricht jedes Segment einer Dimension oder einem linearen Adressraum für genau ein Programm. Stellt das Betriebssystem keine Mechanismen zur Speicherverwaltung bereit, muss der Programmierer darauf achten, dass sich Stapel und Daten nicht überschneiden. Es sei noch kurz bemerkt, dass diese Aufteilung des Speichers gefährlich ist. So kann durch ein ungeschütztes Wachsen des Stapels das Datensegment des Programms überschrieben werden. Damit lassen sich über den Stapel Schadsoftware einschleusen und die Programmausführung verändern.

Sollen nun mehrere Programme gleichzeitig im Speicher vorgehalten werden, wird dieser in Segmente aufgeteilt. Jedes dieser Segmente stellt den Speicherbereich für ein Programm bereit, gliedert sich also in die eingangs erwähnten Felder. Diese Segmentierung führt zu neuen Anforderungen an die Hard- und Software. Zum einen muss der Programmcode so erzeugt werden, dass das Programm an jeder beliebigen Stelle des Speichers abgelegt werden kann. Das heißt, alle Adressangaben sind relativ zu einer zum Zeitpunkt der Programmgestaltung unbekannten Basisadresse anzugeben. Ein Programmierer muss in seinem Assembler-Programm daher eine Basisadresse festlegen, und zwar als variable Bezugsadresse, um sich später im Programm auf diese zu beziehen. Der wirkliche Wert der Basisadresse wird entweder von einem dem Assembler nachgeschalteten Übersetzungsprogramm, dem Linker, vorgenommen, oder die Adresse wird vom Betriebssystem gesetzt. Die Hardware muss dann schließlich durch eine spezielle Adressierungsart einen Zugriff zu Speicherworten relativ zu dieser Basisadresse durchführen.

Ein Programm kann vollständig ein Segment ausfüllen oder aber auf mehrere verschiedene aufgeteilt werden. Erstes bildet den Speicher für ein Programm so ab, als wenn es den physischen Speicher exklusiv zur Verfügung hat, und wirkt daher für

den Anwendungsprogrammierer gewohnt. Letzteres hat den Vorteil, dass eine dynamische Speicherverwaltung erleichtert wird.

Definition 6.2.13 – Segment

Unabhängiger Teil eines Programms: Code- oder Textsegment, Daten- und Stapelsegment.

Speziell Prozessoren der Intel-X86-Familie unterstützen die Segmentierung des Speichers mit einer registerrelativen Adressierung und besitzen zu diesem Zweck ein besonderes Register: das Basisregister. In dieses schreibt das Betriebssystem die Startadresse des Segments, das zu diesem Zeitpunkt ausgeführt wird. Zu jeder Adresse des Programms wird vorher der Inhalt des Basisregisters addiert. Neben dem Basisregister existiert häufig noch ein Limitregister. In diesem kann die Länge des Segments hinterlegt werden, sodass entweder das Programm oder die Hardware selbst überprüfen kann, ob die Grenzen der Segmente eingehalten werden.

Eine Architektur mit einem Basisregister kann ggf. mehrere Adressmodi unterstützen, so können sowohl die indizierte, die direkte als auch die indirekte Adressierung relativ zu diesem durchgeführt werden. Das bedeutet, Programme, die ursprünglich für Maschinen ohne Basisregister geschrieben wurden, lassen sich einfach an die Segmentverwaltung anpassen. Um nun mehrere Segmente zu unterstützen, kann das Betriebssystem eine Segmenttabelle anlegen; das Basisregister zeigt dann auf den ersten Eintrag dieser Tabelle. In dieser Segmenttabelle wird neben der Startadresse auch die Länge des jeweiligen Segments abgelegt.

Viele Computerarchitekturen mit komplexem Befehlssatz (CISC) unterstützen verschiedene Techniken der Segmentierung. So war bei Intel das Basisregister elementarer Bestandteil der Speicherverwaltung. Dagegen verfolgten die Entwickler von Prozessoren mit reduziertem Befehlssatz (RISC) oft die Strategie, nur die Verwaltung von Seiten zu unterstützen und die Aufteilung des Adressraumes dem Betriebssystem zu überlassen.

6.2.4 Caches

Da die Zugriffszeit von der Länge der Leitungen und somit von der Größe des Adressraumes abhängt, muss der Speicher virtuell aufgebrochen werden. Die Lösung ist, vor einen großen und daher langsamen, einen kleineren und schnellen Speicher zu schalten und in diesem die häufig gebrauchten Programm-

teile und Daten zu puffern. Da die Zugriffszeit auf einen Speicher sowohl von der Größe als auch von dessen Technologie abhängt, ergibt sich eine Hierarchie aus schnellen und kleinen sowie langsamen, dafür aber großen Speichern. Die Speicherhierarchie spiegelt dem Prozessor einen schnellen und gleichzeitig großen Speicher vor. Der schnellste Speicher ist auch der kleinste; es sind die Register. Diese erlauben einen schnellen Zugriff, da sie auf dem gleichen Die oder Mikrochip wie der Prozessor hergestellt werden und keine aufwendige Verwaltung benötigen. Die Anzahl dieser lokalen Speicherelemente ist aber begrenzt, zum einen durch ihre Anordnung und zum anderen durch den begrenzten Platz auf dem Chip. Daher existiert in einer zweiten Stufe ein schneller externer Speicher zwischen den Registern des Prozessors und dem eigentlichen Hauptspeicher mit seinem großen und daher langsamen Adressraum. Dieser Zwischenspeicher wird mithilfe von Registerzellen oder Flipflops als statischer, wahlfreier Speicher (RAM) realisiert. Dieses Vorgehen hat den Vorteil, dass dadurch ein Speichercontroller zum Auffrischen eines dynamischen Speichers entfällt. Dieser Puffer zwischen Prozessor und Hauptspeicher wird direkt von der Hardware verwaltet, und zwar derart, dass er dem Programmierer vollständig verborgen bleibt. Ein solcher schneller und verborgener Depotspeicher heißt Cache (franz. „cache“, Versteck⁵). Erst der Hauptspeicher selbst wird aus kostengünstigen DRAM-Zellen aufgebaut. Da dieser groß ist, ist der zusätzliche Aufwand für den Speichercontroller vernachlässigbar.

Da SRAM-Zellen groß sind und viel Platz benötigen⁶, werden die Bitleitungen in einem SRAM-Speicher schnell zu lang. In der Praxis wird daher der Cache noch in mehrere unterschiedlich große Bereiche aufgeteilt. Diese heißen Level-1- (L1), Level-2- (L2) und Level-3 (L3)-Cache. Jede dieser drei Stufen ist aus SRAM-Zellen aufgebaut⁷.

Da der Cache eine begrenzte Größe hat, muss sein Inhalt von Zeit zu Zeit ausgetauscht werden. Es macht wenig Sinn,

⁵ In der Ägyptologie ist die Cachetee von Deir el-Bahari bekannt. Dort wurden umgebettete Mumien schon in der Antike vor Grabräubern versteckt.

⁶ Sei F eine Konstante und im Chipentwurf skalierbare Größe, dann sind DRAM-Zellen etwa $6F^2$ Flächeneinheiten groß, während SRAM-Zellen von $80F^2$ bis zu $160F^2$ Flächeneinheiten besitzen. Die Zugriffszeit eines gleich großen SRAM kann dann gegenüber einem DRAM um mehr als 10-mal langsamer sein, wenn man die Zeit für den Refresh der Zellen außer Acht lässt.

⁷ In modernen Mehrkernarchitekturen teilen sich mehrere Kerne den L3-Cache, der dort auf dem Die des Prozessors integriert ist. Der L3-Cache ist daher groß und wird inzwischen als DRAM aufgebaut. Die dazu notwendige Halbleitertechnologie heißt embedded DRAM.

dabei auf einzelne Speicherwörter zuzugreifen, da der Aufwand diese auszuwechseln hoch ist. Es zeigte sich, dass Programme zeitlich und räumlich lokal begrenzt den Hauptspeicher nutzen. Das bedeutet zum einen, dass wenn auf ein Speicherwort zugegriffen wird, die Wahrscheinlichkeit hoch ist, dass auf eins der Speicherwörter benachbarter Adressen ebenfalls zugegriffen wird, und zum anderen, dass wenn auf ein Speicherwort geschrieben oder gelesen wurde, demnächst wieder auf dieses zugegriffen wird. Diese Eigenschaft macht es möglich, einen begrenzten Teil des Hauptspeichers im Cache zu puffern. Aus diesem Grund werden größere, zusammenhängende Partitionen der Daten im Cache vorgehalten. Die bei der Seitenverwaltung verwendeten Seiten sind allerdings zu umfangreich, da schnelle Caches in der Regel kleiner als die Seitenrahmen des Hauptspeichers sind und sich die Größe der Seiten eher an der Latenz und Technologie der Hintergrundspeicher orientiert. Aus diesem Grund werden Inhalte des Hauptspeichers im Cache blockweise gespeichert.

Definition 6.2.14 – Speicherblock

Ein Speicherblock fasst m Speicherworte eines linearen Adressraumes zusammen. Speicherblöcke sind wesentlich kleiner als Pages und Segmente.

Wie wird das kleine Depot durch die Hardware vor dem Programmierer verborgen? Der Cache wird wie der Hauptspeicher selbst an die gleichen Adress-, Dienst- und Datenleitungen des Prozessors angeschlossen. Eine spezielle Schaltung lauscht auf dem Steuerbus und Adressbus des Systems und aktiviert den Cache immer dann, wenn ein Lese- oder Schreibzugriff auf den Hauptspeicher stattfindet. Dieses Verfahren heißt Bus-snooping (engl. „to snoop“, schnüffeln). Der Cache selbst teilt sich in zwei Bereiche, einen Tagspeicher (engl. „tag“, Etikett) und einen Datenspeicher. Im Tagspeicher befindet sich die Blocknummer des im Datenspeicher abgelegten Blocks. Stimmen die Blocknummer aus dem Adresswort des Prozessors und die im Tagspeicher überein, meldet der Cache einen Treffer (engl. „cache hit“). Der Cache ist also als kontinentadressierbarer Speicher (engl. „content addressable memory“, CAM) realisiert und verbirgt sich so vor dem Programmierer. Tag- und Dateneintrag des Cache bilden zusammen mit 2 Steuerbits eine Cachezeile (engl. „cache line“). Die 2 Steuerbits verwalten einen Eintrag. Das 1. Bit signalisiert, dass der Inhalt des Cache mit dem Inhalt des Hauptspeichers übereinstimmt. Nachdem ein Speicherblock in den Cache geschrieben wurde, ist dies gültig (engl. „valid“). Schreibt der Prozessor Daten in den Cache, ohne gleichzeitig den Hauptspeicher zu aktualisieren, setzt er

das Dirty-Bit der Cache-Line. Dies ist das zweite Verwaltungsbitt. Eine Cache-Line beinhaltet also immer einen vollständigen Speicherblock, seine Blocknummer und 2 Steuerbits.

Beim Zugriff auf den Cache wird das Bitmuster des Adresswortes mit den Einträgen in der kontextadressierbaren Speichermatrix verglichen. Zu diesem Zweck verfügt der CAM über eine Reihe von Vergleichern (XOR-Gatter). Bei einem Treffer (engl. „cache hit“) stimmen das Bitmuster der Adressleitungen und der Inhalt des kontextadressierbaren Speichers überein. Das mit dieser Adresse verknüpfte Datum kann nun direkt aus dem Cache geladen werden. Dies geschieht, indem der Ausgang des Vergleiches als Wortleitung des zweiten Speicherblocks fungiert. Da der Cache wesentlich kleiner als der Hauptspeicher ist, kann nicht jeder Speicherzugriff des Prozessors zu einem Treffer führen. Bei einem solchen Cache-Miss muss der Prozessor zunächst warten und kann das Programm nicht weiter ausführen. Die Hardware des Cache lädt dann selbstständig das Datum aus dem Hauptspeicher in den Puffer, sodass es dort bei Folgezugriffen vorliegt. Anschließend signalisiert sie dem Prozessor den Abschluss des Nachladens. Die Kunst besteht nun darin, die Größe des Cache so zu wählen, dass eine möglichst hohe Trefferrate erzielt wird. Leider ist diese aber beschränkt. Experimente zeigten jedoch, dass die Trefferrate stark von der Größe des zu ladenden Datums abhängt. Aus diesem Grund besteht eine Cache-Line nicht aus einem Speicherwort, sondern mehrere Speicherwörter werden zu Blöcken zusammengefasst. Der jeweilige Block wird über den Tagspeicher adressiert. In zahlreichen Experimenten wurde nachgewiesen, dass für gängige Programme ab einer gewissen Speicherblockgröße die Trefferrate im Cache nicht weiter steigt. Es tritt eine Art Sättigung ein. Die maximal zu erreichende Trefferrate beträgt etwa 90 %. Das heißt, die Blockgröße der im Cache abgelegten Speicherblöcke ist so zu wählen, dass eine Rate von ungefähr 90 % erreicht wird. Insbesondere durch Verschränkung und gleichzeitiges Bereitstellen mehrerer Folgeadressen in den Cacheblöcken kann ein Cache kurze Zugriffszeiten bei einer hohen Trefferrate haben. Wenn dann Blöcke nicht im Cache bereitliegen, kann durch schnellen Blocktransfer die Anbindung des Hauptspeichers ebenfalls optimiert werden.

Der Cache ermöglicht dem Prozessor einen schnellen Zugriff auf einen für den Programmierer verborgenen Ausschnitt des Hauptspeichers. Dieser ist nicht linear und wird durch die temporale und räumliche Zugriffsrate des Programms auf bestimmte Speicherblöcke bestimmt. Als Index oder Tag in den Cache dient die Adresse (Abb. 6.13).

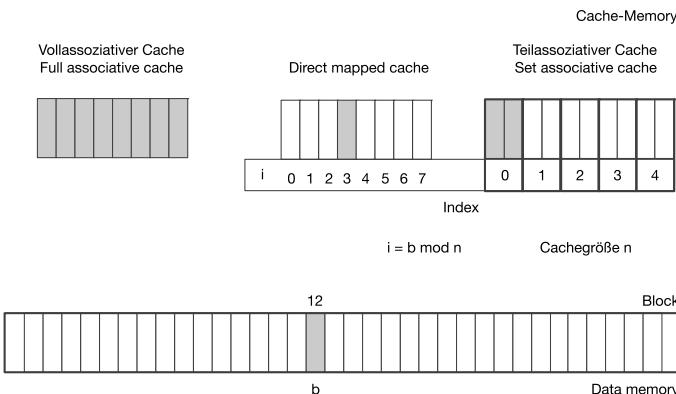


Abb. 6.13 Abbildung des Hauptspeichers durch den Cache

Definition 6.2.15 – Tag

Etikett (engl. „tag“) zur Kennzeichnung einer Cache-Line. Das Tag identifiziert den jeweiligen Speicherblock im nicht linearen Speicher des Cache und ermöglicht einen transparenten Zugriff auf einen versteckten Ausschnitt des gesamten Speichers. Als Tag dient ein Teil des Adresswortes, in der Regel die Blocknummer.

Die einfachste Art, einen Cache zu realisieren, ist den Speicher in Blöcke zu unterteilen und jedem dieser Blöcke eine Nummer zu geben. Da die Größe des Cache durch die Anzahl der in ihm gespeicherten Blöcke bekannt ist, kann ein Block des Speichers an der Stelle $i = b \bmod m$ gespeichert werden, wenn i der Index in den Cache, b die jeweilige Blocknummer und m die Anzahl der im Cache gespeicherten Blöcke sind. Damit ist jeder Blocknummer ein eindeutiger Platz im Cache zugewiesen (engl. „one way“). Der Cache ist somit direkt abbildend (engl. „direct mapping“). In einer Cachezeile werden dann die entsprechende Blocknummer und die dazugehörigen Daten gespeichert. Diese Variante eines Cache ist mit sehr wenig Hardwareaufwand leicht zu implementieren. Insbesondere wird im direkt abbildenden Cache nur ein Vergleicher benötigt, da auf eine Cachezeile immer direkt zugegriffen wird und nur noch überprüft werden muss, ob sich der entsprechende Eintrag auch im Cache befindet. Die Adressierung in den Cache ist der Index. Nur wenn im Tagspeicher eine entsprechende Blocknummer gefunden wurde, wird ein Cachehit signalisiert. Da in einer Cachezeile immer ein vollständiger Block abgelegt ist, muss die Adresse bei einem byteweise organisierten Spei-

cher noch einen Index in das Speicherwort und einen weiteren Index auf das Byte beinhalten. Nachteilig ist, dass durch die feste Zuordnung der Speicherblöcke zu Cachezeilen die Aus- und Einlagerung von Daten sehr unflexibel sind. Insbesondere kann es bei Verwendung eines direkt abbildenden Cache als gemeinsamer Befehls- und Datenspeicher zu häufigen Ein- und Auslagerungen kommen, da die Daten weit entfernt von den Befehlen im Hauptspeicher abgelegt sind. In einer reinen Princeton- oder von Neumann-Architektur kann der Vorteil eines Cache, schnell auf lokale Daten zuzugreifen, dadurch zuничтегемacht werden (Abb. 6.14).

Beim voll assoziativen Cache (engl. „fully associative cache“) wird im Tagspeicher jeweils die komplette Nummer eines Hauptspeicherblocks abgelegt. Die Zuordnung der einzelnen Cachezeilen ist dabei beliebig. Jedem Block ist es erlaubt, an jeder Stelle des Cache zu stehen. Hat der Cache eine Größe von m Blöcken, kann jeder Speicherblock an m Stellen im Cache gespeichert werden, statt nur an einer wie im direkt abbildenden Cache. Es gibt also m Ways einen Hauptspeicherblock für einen schnellen Zugriff zu deponieren. Die Abb. 6.15

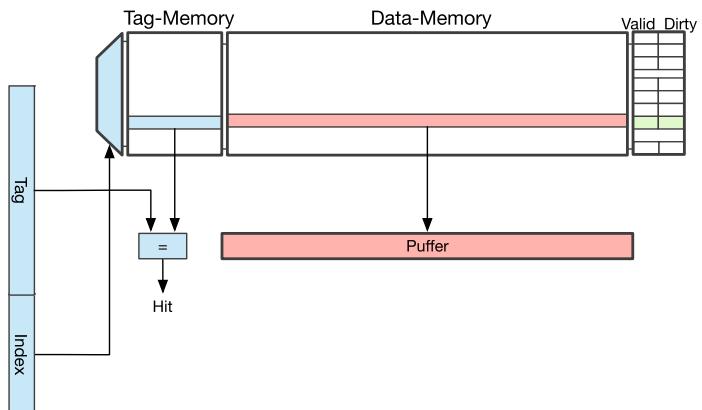


Abb. 6.14 Direkt abbildender Cache

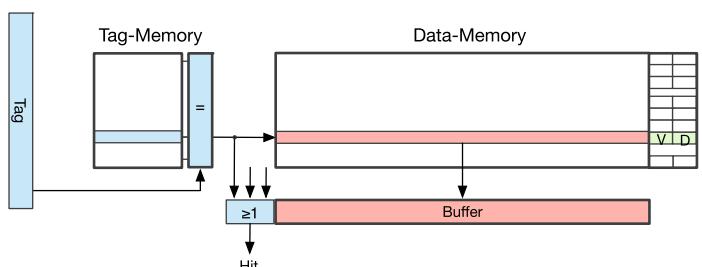


Abb. 6.15 Voll assoziativer Cache

zeigt einen solchen voll assoziativen Cache. Die Blocknummer wird mit allen Einträgen im Tagspeicher verglichen. Liegt ein entsprechendes Datum vor, wird die jeweilige Cachezeile aktiviert. Wird auch nur eine Cachezeile angesprochen, erzeugt das abgebildete ODER-Gatter ein Cachehit genanntes Signal. Ein voll assoziativer Cache ist aufwendig zu implementieren, da für jede Cachezeile ein Vergleicher vorzusehen ist.

Der mengenassoziative Cache ist eine Kombination aus dem direkt abbildenden und dem voll assoziativen Cache. Die Adressen eines mit einem mengenassoziativen Cache ausgestatteten Computers mit byteadressiertem Speicher sind in 4 unterschiedlich große Bitfelder eingeteilt. Das 1. Feld ist ein Index in einen Way genannten Bereich und wählt den Tagspeicher und Datenspeicher aus, der von dieser Adresse angesprochen werden soll. Im 2. Feld der Adresse befindet sich die Blocknummer als Etikett und die beiden verbleibenden Felder adressieren wieder das Speicherwort und das jeweilige Byte. Die Abb. 6.16 zeigt einen 4fach assoziativen Cache. Der Cache ist in 4 Ways aufgeteilt. In dem Tagspeicher des indizierten Way wird nach Übereinstimmung mit der Blocknummer gesucht. Wurde ein Eintrag gefunden, wird über einen Multiplexer mithilfe des Index der entsprechende Speicher ausge-

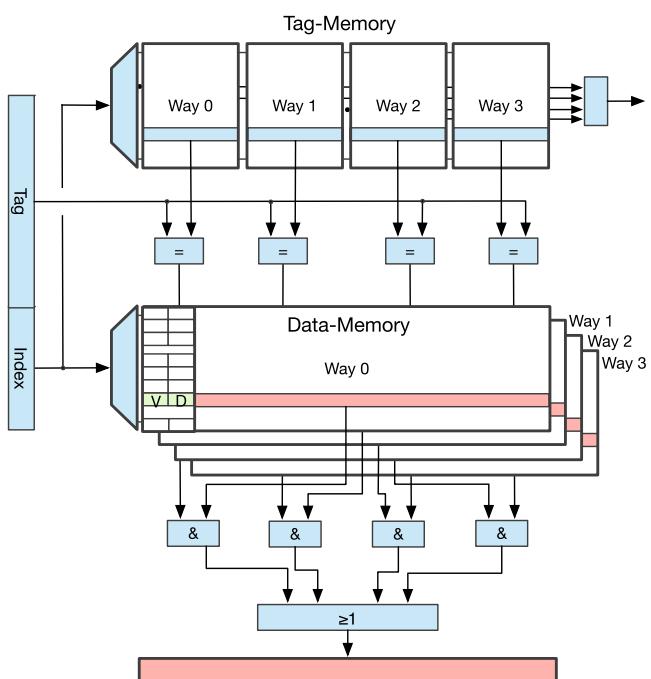


Abb. 6.16 Aufbau eines 4fach assoziativen Cache

wählt. Die Zeile, in der das ermittelte Tag steht, adressiert wiederum den ausgewählten Datenspeicher. Ein Treffer wird wieder über das Hitsignal angezeigt und das gesuchte Datum in einem Puffer hinterlegt. In einem mengenassoziativen Cache, der in n Ways und m Blöcke aufgeteilt ist, kann jeder Hauptspeicherblock auf n verschiedene Weisen zwischengespeichert werden. In dem Fall spricht man von einem „4-way set associative cache“.

Typischerweise sind in einem Block eines Cache 4 Speicherwörter zu jeweils 4 Byte organisiert. Eine typische 32-Bit-Adresse teilt sich dann in 28 Bit für die Blocknummer und jeweils 2 Bit für die Wort- und die Bytenummer auf. Eine Cache-Line gliedert sich dann in 28 Tagbits, die mit dem Tag in der Adresse verglichen werden müssen, 4 Bit zur Adressierung von Wort und Byte und 2 zusätzlichen Steuerbits. Mit den 2 Steuerbits wird der Cacheeintrag verwaltet. Das Valid-Bit zeigt an, dass ein Eintrag gültig ist und nicht von den Daten im Hauptspeicher abweicht. Das Dirty-Bit wird vom Prozessor gesetzt, wenn das Datum im Cache verändert wurde. Den 34 Bit schließen sich dann noch $4 \cdot 4 \cdot 8 = 128$ Bit zum Speichern des Datums an.

Beim Zugriff auf den Cache unterscheidet man mindestens zwischen zwei unterschiedlichen Verfahren: dem Demand-fetching und dem Prefetching.

Beim Demand-fetching wird eine Cachezeile nur dann geladen, wenn versucht wurde, auf einen Hauptspeicherblock zugreifen und dieser nicht im Cache gefunden wurde (engl. „cache miss“). In diesem Fall muss der Prozessor warten und der entsprechende Block aus dem Hauptspeicher geholt werden. Bei vielen Implementierungen wird als Erstes das Speicherwort geladen, auf das der Prozessor zugreifen wollte, damit dieser die Ausführung des Programms sofort fortsetzen kann. Daher werden danach alle folgenden Worte geholt. Speicherworte des Blocks, die vor dem adressierten Speicherwort im Hauptspeicher stehen, sind im Anschluss zu laden. Diese Technik heißt „wrap-around“.

Beim Prefetching wird nicht nur der Block geladen, in dem das durch den Prozessor angeforderte Datum steht, sondern auch ein weiterer Block. Dies kann entweder der Block mit einer um 1 niedrigeren Nummer oder der Block nach dem bereits geholten Block sein. Wird also auf den Block n zugegriffen, lädt das Prefetching entweder den Block mit der Blocknummer $n - 1$ oder den Block mit der Nummer $n + 1$.

Das Lesen auf dem Cache ist relativ einfach zu implementieren. Bei einem Zugriff auf eine Speicheradresse muss der Cachecontroller überprüfen, ob der angesprochene Block bereits im Cache steht. Ist dies der Fall, wird das entsprechende Datum aus dem Cache gelesen. Im Falle eines Cachehits

muss der Block aus dem Hauptspeicher geholt werden. Das Schreiben des Cache ist komplizierter: Zum einen kann das Datum gleichzeitig in den Cache und den Hauptspeicher gespeichert werden (engl. „write-through“), zum anderen kann zunächst auch nur in den Cache geschrieben werden (engl. „copy-back“). In dem Fall wird das Dirty-Bit gesetzt, und der Cache muss ggf. selbst die Synchronität zum Hauptspeicher herstellen.

Beim Write-through geht ein wesentlicher Vorteil des Cache verloren, da der Prozessor das Datum gleichzeitig in den Cache und den Hauptspeicher schreibt. Die Dauer des Schreibvorgangs wird also merklich durch die Zeit des Hauptspeicherzugriffs bestimmt. Die fortwährende Kohärenz (lat. „cohaerentia“, zusammenhängen) zwischen Hauptspeicher und Cache ist ein Vorteil, verglichen mit anderen Verfahren.

Beim Copy-back-Verfahren schreibt der Prozessor das Datum nur in den Cache und setzt das Dirty-Bit, um anzugeben, dass Cache- und Speicherinhalt nicht mehr kohärent sind. Erst wenn ein Block aus dem Cache entfernt wird, muss dieser wieder in den Speicher kopiert werden. Daher der Name des Verfahrens. Bei diesem wird die geringe Zugriffszeit auf den Cache auch beim Schreiben genutzt, allerdings zu dem Preis eines höheren Verwaltungsaufwandes. Auch die Ein- und Auslagerung von Blöcken in den Cache nimmt bei diesem Verfahren mehr Zeit in Anspruch. Insbesondere in Mehrprozessorsystemen ist die nicht vorhandene Kohärenz zwischen dem Hauptspeicher und dem Cache ein Problem, da bei einem Zugriff von mehreren Prozessoren auf ein Speicherwort erst die Kohärenz dieses Speicherwortes geprüft werden muss und dann ggf. ein Copy-back-Verfahren durchzuführen ist.

Ist der Cache vollständig gefüllt und erzeugt ein Zugriff einen Cache-Miss, müssen wie bei der Seitenverwaltung Blöcke ausgelagert werden. Es bieten sich zwei wesentliche Verfahren an: zum einen wieder das LRU-Verfahren („least recently used“), bei dem der Block aus dem Cache entfernt wird, auf den am längsten weder lesend noch schreibend zugegriffen wurde, zum anderen einfach das Zufallsprinzip, bei dem ein Block zufällig ausgewählt wird.

Sollen Cachezeilen, auf die lange nicht zugegriffen wurde, bestimmt werden, muss der Zugriff auf diese durch die Hardware protokolliert werden. Eine einfache Möglichkeit ist ein 3 Bit breites Zählregister, das bei jeder Verwendung um 1 erhöht wird. In dem Fall ist die Cachezeile mit dem niedrigsten Zählerwert die älteste Cachezeile. Alternativ kann die Alterung auch mithilfe eines rechtsschiebenden Schieberegisters verwaltet werden. Dabei entspricht jeder Eintrag im Schieberegister einer Alterungsstufe. Jede Cachezeile hat bei diesem Verfahren eine eigene Nummer. Bei leerem Schieberegister wird bei einem

Zugriff diese Nummer in das Schieberegister geschrieben. Bei einem belegten Schieberegister wird ermittelt, ob die Nummer der entsprechenden Zelle schon im Schieberegister steht. Wenn das der Fall ist, wird diese dem Register entnommen, anschließend gelöscht und wieder vorne in das Register geschrieben. Nummern, auf die selten zugegriffen wird, stehen dann am weitesten rechts im Schieberegister.

Das Verfahren mit dem geringsten Aufwand ist, die zu löschen Zeile durch Zufall auszuwählen. Dieser Einfachheit steht der Nachteil gegenüber, dass die Vergangenheit des Cache unberücksichtigt bleibt und damit der Vorteil der temporalen Lokalität aufgegeben wird.

6

6.2.5 Virtueller Speicher

In einem virtuellen Adressraum werden die Daten über die Zuordnung der Seitennummer zu den Seitenrahmennummern adressiert. Dazu muss eine Seitentabelle im Speicher vorgehalten werden, die diese verwaltet. Bei einem Speicherzugriff wird die virtuelle Adresse in eine physische Adresse umgewandelt. Der Prozessor muss zunächst einmal in der Seitentabelle nachschlagen, welcher Seitenrahmen zu welcher Seite gehört, bevor er auf das eigentliche Datum zugreifen kann. Die Flexibilität des Zugriffs auf den Speicher wird durch eine doppelt so lange Zugriffszeit wie bei einem einfachen Speichermodell erkauft. Um diese Zeit zu minimieren, wandelt die Hardware eine virtuelle Adresse direkt in die physische Adresse um. Die Einheit zur Adressumwandlung in der digitalen Schaltung heißt Memory-Management-Unit.

Die Memory-Management-Unit verwaltet die im Hauptspeicher abgelegte Seitentabelle. An der Schnittstelle zum Prozessor liegt die virtuelle Adresse an. Diese besteht aus zwei Teilen, einem Index in die Seitentabelle und einem direkten Offset, um mehrere Seiten blockweise im Speicher halten zu können. Da die Seitentabelle sehr groß werden kann, erfolgt die Adressumsetzung häufig mehrstufig. Das heißt, eine erste Tabelle verweist auf mehrere geteilte Seitentabellen. Dies hat den Vorteil, dass Seitentabellen ebenfalls auf den Hintergrundspeicher ausgelagert werden können. Die Memory-Management-Unit besitzt zu diesem Zweck ein eigenes Register. In dieses legt das Betriebssystem die Startadresse der ersten Indextabelle der Seitenverwaltung ab. Der erste Eintrag einer in gleich große Teile zerlegten virtuellen Adresse dient als Index in die Seitentabelle. Hier steht nun nicht der Seitenrahmen, sondern die Startadresse einer zweiten Seitentabelle. Auf diese greift dann das zweite Feld der virtuellen Adresse als Index zu. In der Regel ist dieser Prozess vierstufig.

Um den Speicherzugriff nicht unnötig zu verzögern, werden häufig benötigte Einträge der Seitentabelle in einem schnellen, versteckten Zwischenspeicher direkt in der Memory-Management-Unit gehalten. Basierend auf dem bereits bekannten räumlichen und zeitlichen Lokalitätsprinzip von Programm- und Datenzugriffen kann die Memory-Management-Unit bis zu 64 Einträge der Seitentabelle lokal in einem kontinentadressierbaren Speicher vorhalten. Dieses Adressumsetzungsdepot heißt „Transaction-look-ahead-Puffer“ (TLB).

Die Abb. 6.17 skizziert die Umsetzung einer virtuellen in eine physische Adresse, also die Zuordnung einer Seitennummer zu einer Seitenrahmennummer. Die virtuelle Adresse gliedert sich dabei in einen Tag genannten Zeiger. Dieser dient direkt als Etikett des Transaction-look-ahead-Puffers. Ein zweites, kürzeres Feld ist ein Index in diesen Speicher. Da der TLB wieder wie ein Cache in mehrere Sets aufgeteilt ist, kann der Zugriff auf den TLB effizient gestaltet werden. Ein Offset genanntes Feld der virtuellen Adresse wird direkt in die physische Adresse kopiert. Liegt ein gültiger Eintrag im TLB vor, wird das in der entsprechenden Zeile gespeicherte Datum der

TLB

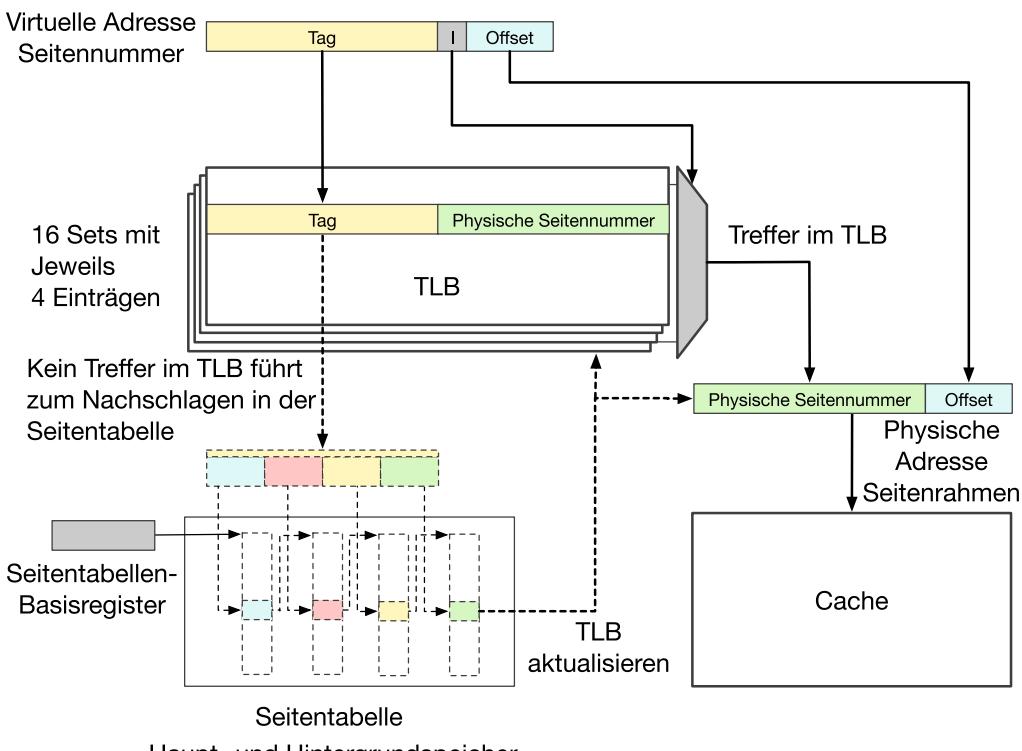


Abb. 6.17 Skizze der Adressumsetzung mit TLB

erste Teil der physischen Adresse. Mit dieser und dem Offset kann nun direkt der Cache adressiert werden. Liegt kein Treffer im TLB vor, dient das Tag der virtuellen Adresse als Index in die im Speicher liegende Seitentabelle. Mit deren Hilfe werden dann der erste Teil der physischen Adresse gebildet und der TLB wie ein Cache aktualisiert. Viele moderne RISC-Prozessoren unterstützen die Seitenverwaltung nicht direkt durch Hardware, sondern lösen eine Unterbrechung der Software aus. Dieses Vorgehen reduziert den schaltungstechnischen Aufwand. Daher ist nur ein TLB zur Adressumsetzung notwendig.

6.3 Mikroarchitektur

Die Mikroarchitektur beschreibt den inneren Aufbau des Rechners, die Struktur des Steuer- und Rechenwerks. Während die Architektur spezifiziert, was die Maschine tun soll, definiert die Mikroarchitektur, wie sie das macht. Verbindendes Element zwischen der Architektur und der Mikroarchitektur ist das Mikroprogramm. Im Gegensatz zur Architektur eines Rechners, die die Schnittstelle und die Abläufe der Maschine gegenüber dem Programmierer spezifiziert, implementiert die Mikroarchitektur diese Spezifikation. Ein Computer mit komplexem Befehlssatz (CISC) besteht aus mindestens zwei Teilen: dem Steuerwerk, um das Verhalten, und dem Rechenwerk (engl. „data path“) um die eigentliche Datenverarbeitung zu implementieren. Bei Rechnern mit reduziertem Befehlssatz (RISC) verwischt diese klare Trennung, da sie als skalierbare Architektur auf die Fließbandverarbeitung optimiert sind oder sogar als superskalare Mikroarchitektur implementiert werden.

6.3.1 Befehlausführung

Auf die einzelnen Befehle eines Programms greift ein Prozessor zu, indem an den Speicher die Adresse der jeweils nächsten Operation angelegt wird. Zu diesem Zweck besitzt die Maschine einen Zeiger, der immer auf die als Nächstes auszuführende Programmadresse zeigt. Dieser Zeiger heißt Befehlszähler (engl. „program counter“, PC). Nach dem Anlegen des Befehlszählers an die Adressleitungen des Speichers kann der Prozessor den Befehl lesen und dann decodieren oder vielmehr entschlüsseln. Je nach Befehl müssen ggf. noch weitere Daten aus dem Speicher geholt werden. Erst dann kann dieser ausgeführt werden. Nach der Ausführung werden der Befehlszähler erhöht und der nächste Befehl gelesen. Die Zeit zwischen An-

legen der Adresse zum Holen eines Befehls und dem erneuten Ansprechen der Adressleitungen heißt Befehlszyklus.

Der Befehlszyklus wiederum gliedert sich in mehrere Phasen: Befehl holen, Befehl decodieren, Operanden laden, Berechnung durchführen und Ergebnis schreiben.

Die erste Phase im Befehlszyklus ist das Holen des Befehls (engl. „instruction fetch“, IF). Wie bereits beschrieben, wird dazu der Inhalt des Befehlszählers über den Adressbus an den Speicher übermittelt. Dieser legt dann ein Datenwort auf den Datenbus, in diesem Fall das zu holende Befehlswort. Je nachdem, aus wie vielen Befehlsworten ein Befehl besteht, kann diese Phase unterschiedlich lang sein. Um die Ausführung eines Programms zu beschleunigen, verfügen auch einfache Prozessoren häufig über einen Befehlspuffer: mehrere Befehlsregister, in die der aus mehreren Worten bestehende Befehl auf einmal geladen werden kann. Die Abb. 6.18 zeigt die Abhängigkeit der Länge der Befehlsphasen vom Befehlsformat.

Nach dem Laden des Befehls in das Befehlsregister ist dieser zu decodieren (engl. „instruction decode“, ID).

Nach der Decodierung des Befehls müssen die Argumente aus dem Speicher geholt werden (engl. „operand fetch“, OF). Dies kann je nach Adressierungsart unterschiedlich ablaufen. Bei einem Befehl mit inhärenter Adressierung ist die Phase, die Operanden zu holen, kurz, da die Daten in Registern stehen. Soll aber indirekt oder indiziert zugegriffen werden, muss erst die Adresse des Arguments ermittelt werden. Dabei werden Registerinhalte mit aus dem Speicher gelesenen Werten verknüpft. Dies geschieht mit der ALU und ist mit der Un-

Befehl holen

Befehl decodieren

Operanden holen

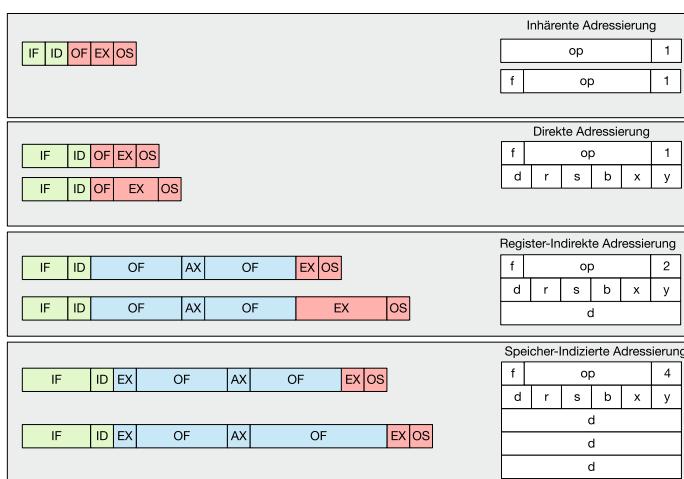


Abb. 6.18 Befehlsphasen

Befehl ausführen**6****Ergebnis schreiben**

terphase AX bezeichnet. In der Abbildung ist diese Adressberechnung während der Phase des Holens von Operanden angedeutet. Je nachdem, welches Befehlsformat die Architektur implementiert, kann ggf. mehrmals auf den Speicher zugegriffen werden.

Nachdem die Operanden geholt wurden und im Prozessor bereitstehen, kann die eigentliche Operation ausgeführt werden. Diese Phase heißt Ausführung (engl. „execute“, EX). Auch dieser Teil der Befehlausführung kann unterschiedlich lang sein. Je nachdem, ob eine Ganzzahladdition oder eine Fließkommamultiplikation durchzuführen ist, unterscheiden sich die Rechenzeiten erheblich.

Nach der Berechnung muss das Ergebnis gespeichert (engl. „Operand store“, OS) werden. Je nach Befehlstyp und Adressierungsart können die Befehlszyklen stark variieren. Eine dieser Variationen ist in □ Abb. 6.18 nur angedeutet. Es gibt Maschinen mit mehreren Tausend verschiedenen Befehlszyklen, je nachdem, welcher Befehl welche Adressierungsart unterstützt und wie viele unterschiedliche Zugriffsprotokolle ausgeführt werden müssen.

Ein programmierbarer Computer mit einem solchen Befehlszyklus kann durch eine universelle Turing-Maschine modelliert werden. Oder allgemeiner: Eine Maschine, die Befehle aus einem Speicher holt und je nach Befehl Daten liest und dann wieder schreibt, implementiert eine universelle Turing-Maschine. Daher können Prinzipien, Algorithmen und theoretisch gewonnene Erkenntnisse zur Berechnung, die mithilfe des Modells der Turing-Maschine gewonnen wurden, auf derartige Rechner angewendet werden.

6.3.2 Steuerwerk

Die Aufgabe des Steuerwerks ist die korrekte Ansteuerung des Rechenwerks. Es sei daran erinnert, dass die Rechnerarchitektur dieses Verhalten beschreibt und spezifiziert und das Steuerwerk der Mikroarchitektur dieses implementiert. Die Schnittstelle des Steuerwerks zum Programmierer sind das Befehlsregister (engl. „instruction register“, IR), der Befehlszähler (engl. program counter, PC) und ggf. der Stapelzeiger (engl. „stack pointer“, SP). Im Befehlsregister steht der jeweils aktuell abzuarbeitende Befehl. Mit dem Rechenwerk ist das Steuerwerk über das Statuswort und das Steuerwort verbunden. Teile des Steuerwortes sind als eigenes Register dem Programmierer lesbar zugänglich. Das Statuswort zeigt den Zustand des Rechenwerks an, und das Steuerwort enthält alle Steuersignale für dessen Ansteuerung. Um eine Funktion im Rechenwerk durchzuführen, wird der Opcode aus dem Befehlsregister ausgewertet.

In Abhängigkeit vom Zustand des Rechenwerks wird dann ein neues Steuerwort erzeugt. Dabei kann das Steuerwerk auch eine Folge von Steuerworten generieren und so ein zeitliches Verhalten in Form eines Algorithmus ausführen.

Das Steuerwerk kann im einfachsten Fall mithilfe eines PLA aufgebaut werden. Dabei wird die entsprechende Transition durch einen aktuellen Zustand, den Opcode und den Status des Rechenwerks ausgewählt. In der ausgewählten Zeile sind dann das neue Steuerwort und der Folgezustand des Steuerwerks angegeben. Das Steuerwerk ist also als Automat realisiert. Steuerwerke können auf verschiedene Arten implementiert werden. Die einfachste Variante ist die bereits diskutierte mit einer PLA und einem Zustandsregister. Wie wir aus Abschn. 5.2.3 wissen, kann ein Automat direkt als logische Schaltung implementiert werden. Es hat sich im Laufe der Entwicklung von Rechnern allerdings gezeigt, dass eine nicht änderbare (harte) Verdrahtung des Steuerwerks unzweckmäßig ist. Der Grund ist, dass im Falle von Fehlern die komplette Schaltung zu ändern ist. Dies ist zeit- und fehleranfällig. Aus diesem Grund wird die PLA häufig mit einem oder mehreren Speichern realisiert. Das hat den Vorteil, dass die Funktion des Steuerwerks jederzeit verändert werden kann. Da das Verändern des Speicherinhaltes der Programmierung der Steuerwerksfunktion entspricht, nennt man den Vorgang Mikroprogrammierung und den Inhalt des Speichers ein Mikroprogramm. Die Unterscheidung von durch den System- oder Anwendungsprogrammierer geschriebenen Programmen ist wichtig. Ist der Speicher, der die PLA implementiert, als Nur-lese-Speicher (ROM) realisiert, spricht man von statischer, ist der Speicher als RAM ausgelegt, von dynamischer Mikroprogrammierung.

Ein Mikroprogramm besteht wie ein Programm aus mehreren Schritten. In jedem Schritt werden ein Steuerwort für die entsprechende Phase im Befehlszyklus und die Folgeadresse für die nächste Transition ausgegeben. Prinzipiell kann ein Mikroprogramm genau so aufgebaut werden wie ein normales Programm. In der Regel werden Mikroprogramme aber einfacher gehalten, da es ihre Ausgabe ist direkt das Rechenwerk anzusteuern. Auch muss auf der Mikroprogrammebene kein Speicher adressiert werden. Somit ist die Eingabe des Mikroprogrammsteuerwerks homogen; es hat somit keine unterschiedlichen Befehslängen. Verschiedene Adressierungsmodi der Rechnerarchitektur können nun als Folge von Mikroprogrammbefehlen im Mikroprogrammsteuerwerk realisiert werden.

Der Befehlszyklus besteht aus mehreren Phasen. Für jede dieser Befehlsphasen muss ein Steuerwort für das Rechenwerk erzeugt werden. Das Mikroprogramm wird in diesem Fall se-

**Vertikale
Mikroprogrammierung**

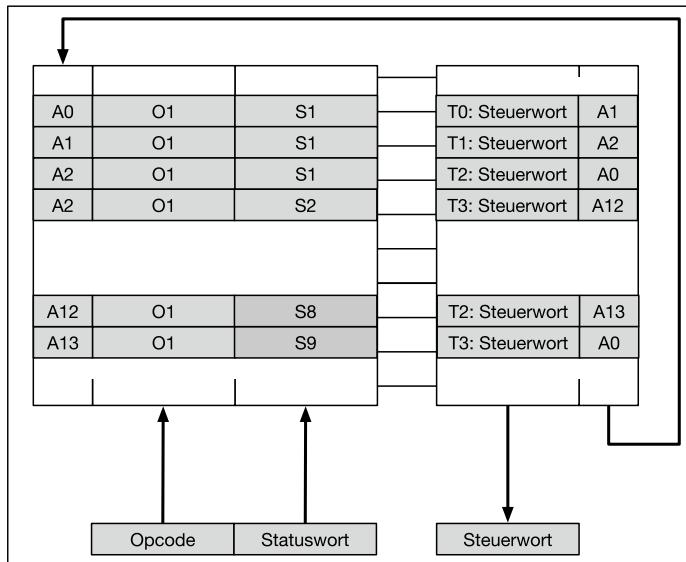


Abb. 6.19 Einfache vertikale Mikroprogrammierung

quenziell abgearbeitet. Ist ein Mikroprogramm in viele sequenzielle Schritte unterteilt, spricht man von vertikaler Mikroprogrammierung. Die beiden Abb. 6.19 und 6.20 zeigen zwei unterschiedliche, vertikal programmierte Mikroprogrammsteuerwerke.

Das einfache Steuerwerk in Abb. 6.19 ist als 0 + 1-Maschine realisiert. Da es sich um ein Steuerwerk handelt, müssen keine Operanden adressiert werden; aus der $n + 1$ -Maschine wird eine 0+1-Maschine. Allerdings muss im Mikroprogramm in jedem Schritt die Folgeadresse gespeichert werden. Dies drückt die +1 aus.

Abb. 6.20 zeigt ein Steuerwerk mit einem Mikrobefehlszähler. Da insbesondere komplexe Adressmodi eine Vielzahl unterschiedlicher und manchmal auch gleicher Schritte benötigen, lässt sich durch Einführung eines Befehlszählers der Mikroprogrammablauf einfacher gestalten. Zwar muss auch in diesem Fall ggf. eine Folgeadresse in der Transitionszeile gespeichert werden, aber durch Einführung eines Stapels kann ein Mikroprogramm geschachtelte Unterprogramme haben. Ein typischer Befehlsablauf beginnt mit einem Einsprung in das Mikroprogramm durch den Opcode. Dafür wird ein Teil des Opcodes als Startadresse des Mikroprogramms interpretiert und in den Mikrobefehlszähler geladen. Das Mikroprogramm wird nun abgearbeitet, bis ggf. eine bestimmte Adressierungsart erkannt wird. In dem Fall werden in ein anderes Mikroprogramm gesprungen und die aktuelle Adresse auf

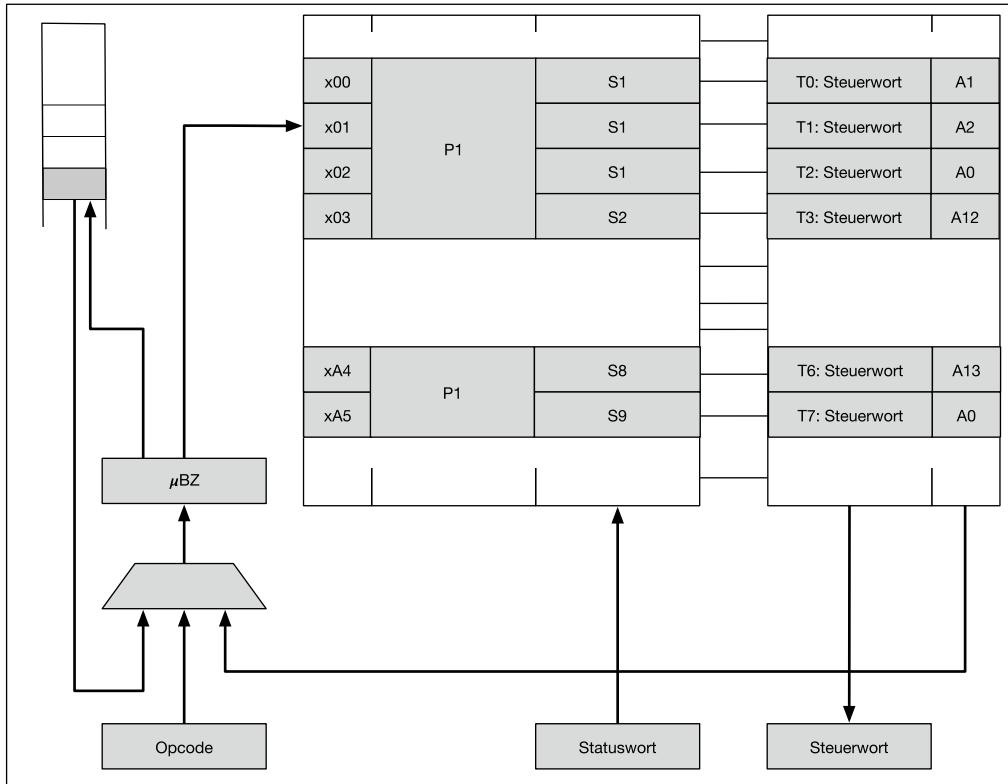


Abb. 6.20 Mikroprogrammsteuerwerk mit Mikroprogrammzähler und Stapel

dem Stapel gespeichert. Nach erfolgreicher Durchführung eines Datentransportes kann dann wieder zum ursprünglichen Mikroprogramm zurückgekehrt werden, und die Ausführung des Befehls wird fortgesetzt. Diese Technik erlaubt es, Adressierungsarten und Rechenanweisungen für das Rechenwerk als getrennte Mikroprogramme auszulegen. Vorteilhaft dabei ist, dass das Mikroprogramm für einen Adressierungsmodus, beispielsweise eine indizierte Adressierung, nur einmal als Mikroprogramm hinterlegt werden muss und dieses Mikroprogramm dann von jedem Befehl aufgerufen werden kann. Ein solcher Aufruf kann sogar mehrmals hintereinander erfolgen, je nachdem wie viele Operanden ein Befehl adressiert und wie viele Datentransfers vom Speicher in die Register des Rechenwerks notwendig sind.

Maschinen, die einen Befehl in einem Befehlszyklus abarbeiten oder die die Befehlssphasen mittels Fließbandverarbeitung aneinanderreihen lassen sich mit jeweils einem Steuerwort pro Befehlszyklus steuern. Das Befehlswort ist dann häufig

**Horizontale
Mikroprogrammierung**

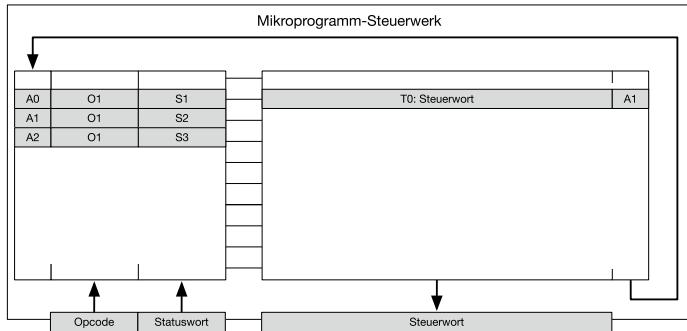


Abb. 6.21 Horizontale Mikroprogrammierung

6

sehr lang. In einem solchen Fall spricht man von einer horizontalen Mikroprogrammierung. Die Abb. 6.21 zeigt ein derartiges Steuerwerk. Natürlich kann ein horizontales Mikroprogrammsteuerwerk auch mit einem Stapel und einem Mikrobefehlszähler ausgestattet sein und so Unterprogrammaufrufe realisieren. In der Praxis sind vertikale und horizontale Mikroprogrammierung in den beschriebenen Formen nicht immer eindeutig zu trennen. Häufig handelt es sich um Mischformen. Bei einer rein skalaren Architektur mit Fließbandverarbeitung kann durch geschickte Wahl der Befehlsformate und der Codierung der Befehle sogar auf ein zentrales Steuerwerk verzichtet werden. Die Ablaufsteuerung erfolgt dann automatisch durch die Reihenfolge der Befehsstufen und läuft immer sequenziell ab. Teile der Befehle werden dann durch das Fließband weitergereicht und immer nur lokal in der entsprechenden Stufe durch eine einfache kombinatorische Schaltung ausgewertet.

Neben dem Mikroprogrammsteuerwerk mit der PLA, einem kleinen Stapel und dem Mikroprogrammzähler gehört zum Steuerwerk der Befehlszähler. Der Befehlszähler ist ein Register. Sein Inhalt zeigt immer auf den als Nächstes auszuführenden Befehl. Der Befehlszähler ist für den Programmierer sichtbar, aber nicht direkt manipulierbar. Zwar kann der Befehlszähler mit bedingten oder unbedingten Sprüngen neu gesetzt werden, direkt veränderbar über das Speichern eines Wertes ist er aber nicht. Bei der Ausführung eines Programms zählt das Steuerwerk den Befehlszähler automatisch hoch, es sei denn, ein Sprungbefehl setzt es neu. Ebenfalls Einfluss auf den Befehlszähler haben Unterbrechungen (Interrupts) und die Ausnahmebehandlung des Prozessors (Exceptions und Traps).

Ein weiteres wichtiges Register im Steuerwerk ist der Stapelzeiger. Wie der Befehlszähler kann er vom Programmierer

nicht direkt, sondern nur indirekt mithilfe spezieller Befehle verändert werden. Der Stapelzeiger zeigt auf einen bestimmten Speicherbereich und simuliert im universellen Speicher mit wahlfreiem Zugriff einen Stapel. Der Stapel wird zum Verwalten von Funktionsaufrufen benötigt, kann vom Programmierer aber auch für andere Aufgaben genutzt werden.

6.3.3 Rechenwerk

Im Rechenwerk befinden sich eine oder mehrere Recheneinheiten. Dies können arithmetisch-logische Schaltungen zum Rechnen mit Integer sein, Multiplikations- und Divisionshardware oder Fließkommaeinheiten. Um die Operanden für die Recheneinheiten bereitzustellen, verfügt das Rechenwerk über mindestens ein Register, in der Regel aber über einen größeren Registersatz. Die Mikroarchitektur des Rechenwerks wird durch die Verschaltung der Rechenwerke mit den Registern bestimmt. Schnittstelle zum Speicher ist oft ein Datenregister zum Puffern der zu lesenden oder zu schreibenden Daten. Das Steuerwerk der Maschine ist über ein Status- und ein Steuerwort mit dem Rechenwerk verbunden. Die Struktur des Rechenwerks bestimmt im Wesentlichen die Leistungsfähigkeit und die Geschwindigkeit, mit der Berechnungen durchgeführt werden. Zugriffskonflikte können durch Multiplexer aufgelöst werden. Diese treten auf, wenn beispielsweise zwei Rechenwerke gleichzeitig in ein Register schreiben oder Operanden zu unterschiedlichen Zeiten am gleichen Eingang eines Rechenwerks anliegen sollen. Diese Multiplexer werden vom Steuerwerk gesteuert. Bei großen Datenwortbreiten sind Multiplexer aufwendig und komplex zu implementieren. Aufwendige Logik ist langsam und benötigt viel Chipfläche. Daher erfolgt der Datentransport zwischen Registern und Recheneinheiten oft über Busse. Mit Tri-State-Treibern können einzelne Komponenten gezielt durch das Steuerwerk von einem Bus getrennt werden. Dadurch ist es möglich, mithilfe der gemeinsamen Leitungen des Busses unterschiedliche Transportwege zwischen Komponenten des Rechenwerks zu schalten.

6.3.3.1 Grundelemente und Struktur

Die Anzahl der Register hat einen hohen Einfluss auf den Datendurchsatz und somit auf die Geschwindigkeit eines Rechenwerks. Register sind schnelle Zwischenspeicher, die auf demselben Chip wie die Recheneinheiten untergebracht sind. Durch ihre lokale Nähe zu den Recheneinheiten ist ein schneller Zugriff auf die Daten gewährleistet. Register werden zu Registersätzen (engl. „register file“) zusammengefasst. Ein Re-

Register

gistersatz wird dazu als wahlfreier Speicher realisiert. Durch die reguläre Struktur des Speichers kann so Chipfläche eingespart werden. Insbesondere in Maschinen mit Fließbandverarbeitung bietet es sich an, den Registersatz als Mehrtorspeicher aufzubauen. Dadurch lassen sich Ressourcenkonflikte automatisch vermeiden. Neben den rein technischen Anforderungen an die Implementierung der Register unterscheidet man noch zwischen sichtbaren und unsichtbaren Registern. Sichtbare Register sind in der Architektur, also im Befehlssatz spezifiziert und beschrieben. Ihre Funktion ist dem Programmierer bekannt, und er kann beim Schreiben seiner Programme gezielt auf diese zugreifen. Unsichtbare Register dagegen werden vom Programmierer nicht wahrgenommen und sind auch in der Architektur nicht beschrieben. Daher kann ein Programmierer diese nicht gezielt manipulieren. Die unsichtbaren Register unterstützen und vereinfachen Mikrobefehlsfolgen des Steuerwerks. Dabei sorgen sie für eine zeitliche Entzerrung von Abläufen.

Die arithmetisch-logische Einheit führt die eigentlichen Rechnungen durch. Dazu wird sie vom Steuerwerk konfiguriert. Recheneinheiten unterscheiden sich durch die Zahlen- oder Datenformate wie Fließkommazahlen und Integer, die sie verarbeiten können, und durch ihre Implementierung. Die Implementierung der Recheneinheiten bestimmt den Datendurchsatz. Neben den aus Abschn. 5.3 bekannten Addierwerken kann ein Datenpfad aus unkonventionellen Einheiten zusammengesetzt sein. Das können Bausteine zur Verschlüsselung oder Codierung von Nachrichten oder Bildschirminhalten sein. Hier sei an Grafikprozessoren oder aber auch Netzwerkprozessoren zum Umcodieren von Internetadressen in physische Adressen der Leitungen erinnert. Die Anordnung der Recheneinheiten bestimmt die Mikroarchitektur des Rechenwerks. Die Entwurfsentscheidungen, die zu treffen sind, sind, ob Recheneinheiten sequenziell oder parallel angeordnet sind oder ob ein Steuerwort mehrere parallele Rechenwerke gleichzeitig ansprechen kann, um mit einem Befehl viele Argumente gleichzeitig zu verknüpfen. Ein derartiges Rechenwerk implementiert eine Vektorarithmetik.

Der Datenfluss im Rechenwerk findet zwischen den Registern und den Rechenwerken statt. Er wird vom Steuerwerk mithilfe des Mikroprogramms gemäß der Spezifikation der Architektur entsprechend der ablaufenden Befehle gesteuert. Die Anzahl und Anordnung der Register und Recheneinheiten bestimmt den Aufwand ihrer Verschaltung. Im Sinne der Universalität des Rechenwerks ist es wünschenswert, jede Recheneinheit mit jedem Register zu verdrahten. Aus dem Grund der Effizienz ist das aber nicht immer möglich. Ein Mikroarchitekt muss daher beim Entwurf des Rechenwerks einen guten Kom-

promiss zwischen Datendurchsatz und Chipfläche finden. Je nach Aufgabe des zu entwerfenden Rechenwerks ist dies ein schweres Optimierungsproblem. Die Struktur der Verschaltung der Komponenten eines Rechenwerks bestimmt im Zusammenspiel mit dem Speicher den Datendurchsatz und somit die Geschwindigkeit des Rechners.

6.3.3.2 Vektorielles Rechenwerk

Eine besondere Form der Recheneinheiten sind die Multiplizier-addier-Einheiten (engl. „multiply and accumulate“, MAC). Die Idee dieser Recheneinheiten ist es, zunächst zwei Zahlen in Hardware direkt miteinander zu multiplizieren und das Ergebnis dieser Multiplikation gleich im Anschluss mit einem dritten Argument zu addieren. Eine MAC-Einheit führt somit die Operation $a = a + (b \cdot c)$ durch.

Baut man ein Rechenwerk aus mehreren parallelen MAC-Recheneinheiten, kann die entstehende Maschine schnell und elegant Vektor- und Matrizenoperationen der linearen Algebra durchführen. Dieses Konzept kommt in der digitalen Signalverarbeitung zum Einsatz, da digitale Filter mathematisch häufig in Matrixform beschrieben werden und der Rechner diese dann effizient verarbeiten kann. Rechner mit einem hochgradig parallelen Rechenwerk und vielen einfachen, aber auch parallel angeordneten Registern werden Vektorrechner genannt. Sie bearbeiten einen Befehl des Programms gleichzeitig auf mehreren unterschiedlichen Argumenten. Eine spezielle Kategorie dieser Vektorrechner ist der digitale Signalprozessor (engl. „digital signal processor“, DSP). In der Rechnerarchitektur werden derartige Maschinen auch SIMD-Computer (engl. „single instruction, multiple data“) genannt. Durch die gleichzeitige Bearbeitung mehrerer Rechenschritte steigt der Durchsatz der Maschine enorm. Allerdings ist dies nur möglich bei Problemen mit einer relativ geringen Datenabhängigkeit zwischen den einzelnen Variablen des Programms. Ein DSP mit einem hohen Datendurchsatz für digitale Filteroperationen hat nicht für jedes beliebige Programm einen hohen Datendurchsatz, sondern wurde extra auf eine hohe Leistungsfähigkeit zur Berechnung von Filtern in der Signalverarbeitung hin optimiert. Neben der SIMD-Anordnung des Datenpfads sind auch die SISD- („single instruction, single data“) und die MIMD-Architektur („multiple instruction, multiple data“) denkbar.

6.3.3.3 Skalares Rechenwerk

Dem Rechenwerk mit Vektorverarbeitung steht die skalare Mikroarchitektur gegenüber. In einem skalaren Rechenwerk werden alle Befehle sequenziell ausgeführt. Eine sequenzielle

Bearbeitung der Befehle führt zu vielen Befehlszyklen. Je nach Art des Befehls hat ein Befehl einen kurzen oder langen Befehlszyklus. Befehle mit langen Befehlszyklen reduzieren den Befehlsdurchsatz.

Die Mikroarchitektur des Rechenwerks hat einen großen Einfluss auf die Ausführungsgeschwindigkeit eines Rechners. Um dies zu verdeutlichen, betrachten wir den Befehlszyklus mehrerer sequenzieller Befehle. Die Abb. 6.22 zeigt die Phasen in den Befehlszyklen unterschiedlicher Rechner. Betrachten wir zunächst einen klassischen von Neumann-Rechner in Princeton-Architektur und den bereits in Abschn. 6.3.1 beschriebenen Befehlszyklus. In diesem Fall haben wir einfach fünf Befehle aneinandergereiht. Aufgrund der Verwendung unterschiedlicher Adressmodi sind die Befehlphasen und daraus resultierend auch die Befehlszyklen unterschiedlich lang. Die Abbildung zeigt noch, auf welche Ressource der Mikroarchitektur welche Phase des Befehlszyklus zugreift. Ausgehend von einer 2-Adress-Architektur wird das Ergebnis einer Rechnung in ein Register geschrieben. Die Phase Operand-Fetch gliedert sich in diesem Fall in zwei Teile: einen Zugriff auf interne Register – in der Abbildung nicht gezeigt – und den zeitintensiven Zugriff auf den Speicher (MEM). Ein Operand-Fetch auf Register und Speicher kann gleichzeitig erfolgen, und für die hier gemachten Betrachtungen ist der Zugriff auf den Speicher zu betrachten. In der Abbildung sind Befehle dargestellt, die ausschließlich die inhärente Adressierung verwenden und daher beim Operand-Fetch nicht den Speicher lesen. Einige Befehle verwenden komplexere Adressierungsmodi, sodass sie mehrmals auf den Speicher zugreifen. Es fällt auf, dass die Ressourcen der Mikroarchitektur nur in einem Bruchteil der Zeit des kompletten Befehlszyklus ausgelastet sind. Dominiert wird der Befehlszyklus bei einigen Befehlen durch den Speicherzugriff. Jede Ressource der Mikroarchitektur, ob Register, ALU oder der Befehlscodierer ist schlecht ausgelastet und arbeitet in durchschnittlichen Programmen kaum. Im Folgenden wird daher untersucht, wie sich die Auslastung einzelner Komponenten des Rechenwerks steigern lässt.

Die Auslastung der Ressourcen der Mikroarchitektur kann durch eine überlappende Befehlausführung (Fließbandverarbeitung, engl. „pipelining“) erhöht werden. Die Idee dabei ist, die Ausführungseinheit eines Rechenwerks vom Speicherzugriff zu trennen. Der Zugriff auf den Speicher erfolgt dann durch einen eigenen Speichermanager. Durch die eigene Hardware für den Zugriff auf den Speicher können Operationen, die ausschließlich Ressourcen des Prozessors nutzen, überlappend zu Operationen mit Speicherzugriff erfolgen. Es ist nun sogar möglich, Teile eines Befehls überlappend mit Teilen eines nachfolgenden Befehls auszuführen. So kann beispielsweise ein Be-

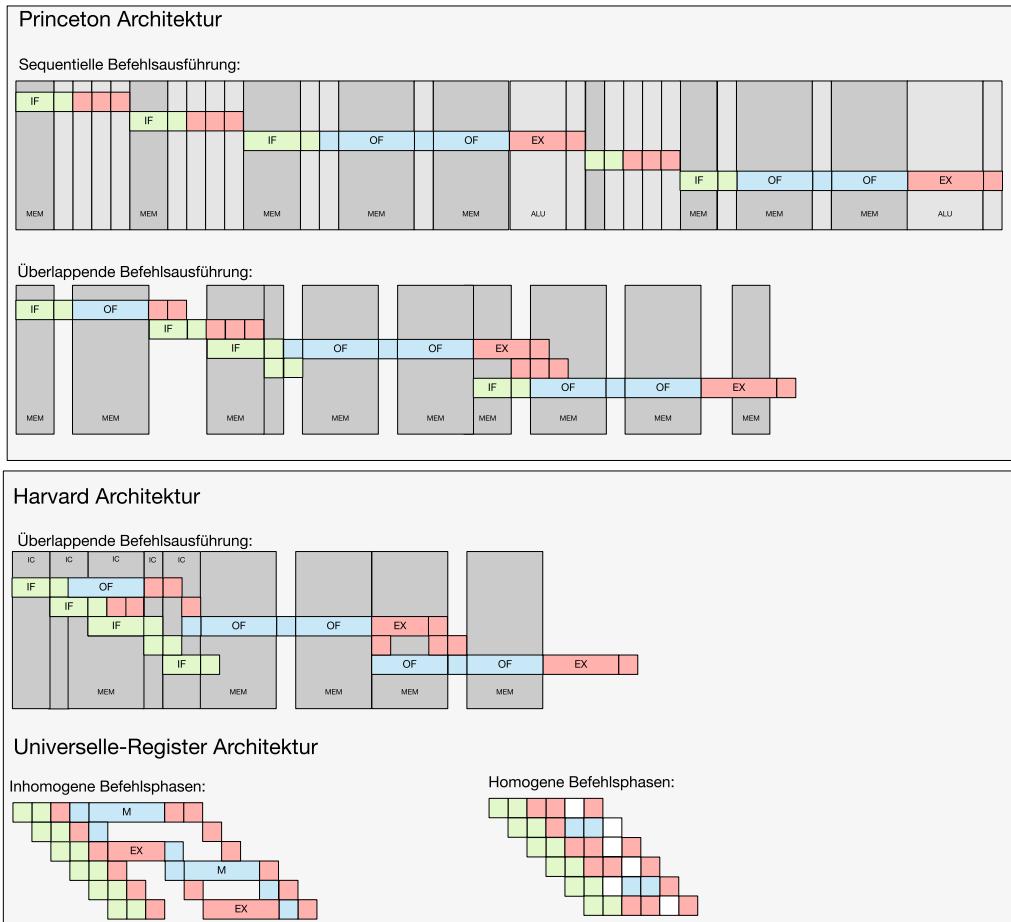


Abb. 6.22 Überlappende Befehlausführung

fehl ausgeführt werden und gleichzeitig ein neuer Befehl aus dem Speicher geladen werden. Man kann in der Abb. 6.22 gut erkennen, wie die Latenz aller fünf Befehle in Summe reduziert wird und die Auslastung der Ressource Speicher (MEM) steigt⁸.

Betrachtet man die überlappende Ausführung der Princeton-Architektur genauer, zeigt sich, dass in unterschiedliche Phasen auf den Speicher zugegriffen wird. Bedingt durch das von Neumann-Prinzip, Programm und Daten im gleichen Speicher abzulegen, konkurrieren zwei eigentlich unabhängige Befehlphasen, der Instruction-Fetch und der Operand-Fetch,

8 Um die Übersicht zu wahren, wurden in der Abbildung der Speicherzugriff exemplarisch gewählt und die gleichzeitige Bearbeitung durch andere Ressourcen wie die ALU weggelassen.

um die Ressource Speicher. Dies wäre bei einer Harvard-Architektur nicht der Fall. Aus diesem Grund werden die beiden Konzepte Princeton- und Harvard-Architektur kombiniert. Im einfachsten Fall einer derartigen Kombination lassen sich in einem Befehlspuffer mehrere Befehle vorhalten und diese blockweise laden. Viel effizienter ist es aber, über einen Befehlscache („instruction cache“, IC) auf den Speicher zuzugreifen. Bei richtiger Dimensionierung des Befehlscache kann der Speicherzugriff beim Holen eines Befehls vom Zugriff auf die Operanden entkoppelt werden. In der Abbildung erkennt man, wie der Grad der Überlappung steigt und die Latenz der Ausführung der fünf Befehle weiter fällt. Ergänzt man den Befehlscache durch einen zweiten Cache für die Daten, erhält man eine Harvard-Architektur auf der Mikroarchitekturebene, ohne die Vorteile der Princeton-Maschine auf der Architekturebene aufzugeben.

Offensichtlich ist der Speicherzugriff die größte Hürde, Ressourcen der Mikroarchitektur effektiv zu nutzen, da sowohl im Kontrollpfad eines Programms beim Holen des Befehls als auch beim Zugriff auf die Daten auf diesen Speicher zugegriffen wird. Den Grad der Überlappung kann man also erhöhen, wenn man den Zugriff auf das Programm und den Zugriff auf Daten konzeptionell trennt. Eine Idee ist, Rechenoperationen ausschließlich auf Registern durchzuführen und dabei nicht mehr auf den Speicher zuzugreifen. Dazu wird eine große Menge von Registern benötigt. Hinzu kommt, dass auf die Register gleichzeitig zugegriffen werden kann. Diese Anforderung erfüllt wird, wenn man die Register mithilfe eines Mehrtorspeichers realisiert. Bei einem Mehrtorspeicher kann auf unterschiedliche Speicherzellen gleichzeitig zugegriffen werden. Durch den Einsatz eines großen Registerspeichers in einer 3-Adress-Architektur können alle arithmetisch-logischen Operationen so gestaltet werden, dass die Phasen Operand-Fetch, Execute und Operand-Store vollständig überlappen. Der Zugriff auf den Speicher erfolgt dann nur noch über Transportbefehle wie load und store. Die Abwicklung des Speicherzugriffs über ausschließlich zwei Befehle hat noch den Vorteil⁹, dass nicht mehr jeder arithmetisch-logische Befehl alle Adressmodi unterstützen muss. Um also den Grad der Überlappung zu erhöhen, trennen wir den Datentransport von der eigentlichen Berechnung. Die so entstandene universelle Registerarchitektur heißt daher auch Load-Store-Architektur. Diese Trennung von Berechnung und Datentransport reduziert den Befehlssatz eines Computers, da jetzt nicht jeder Befehl alle unter-

⁹ In Abhängigkeit von unterschiedlichen Datenworten gibt es natürlich mehrere Load- und auch Store-Befehle. Dieser Aspekt soll aber zunächst einmal vernachlässigt werden.

schiedlichen Adressmodi unterstützt. In □ Abb. 6.22 erkennt man, wie die Entkopplung zwischen Transport und Berechnung den Befehlszyklus drastisch verkürzt. In Kombination mit jeweils einem eigenen Programm- und Datencache kann so der Durchsatz an Befehlen massiv gesteigert werden, insbesondere weil arithmetisch-logische Befehle nur noch auf den Registern arbeiten. Was passiert aber mit den Datentransportbefehlen? Wie fügen sich Laden und Schreiben der Daten aus dem bzw. in den Speicher in den Befehlszyklus ein? Dazu muss eine weitere Befehlphase eingeführt werden: der Speicherzugriff (engl. „memory“, MEM). In dieser Phase erfolgt der Zugriff auf den Speicher, wenn ein Load- oder ein Store-Befehl ausgeführt wird. Diese Phase befindet sich hinter der Ausführungsphase. Dadurch können in dieser Adressberechnungen durch die ALU vorgenommen werden.

Durch Einführung der universellen Registermaschine der Grad der Überlappung gesteigert wird. Betrachtet man □ Abb. 6.22 genauer, fällt auf, dass eine inhomogene Adressberechnung den Überlappungseffekt zerstört. Durch komplexe und langwierige Adressberechnungen entstehen Lücken im Befehlsablauf. Es liegt also nahe, die Adressberechnung und den Speicherzugriff zu homogenisieren. Das lässt sich erreichen, indem auf komplexe Adressmodi verzichtet wird und nur noch wenige Adressierungsarten unterstützt werden. So kann bei einer Beschränkung auf die inhärente und die direkte Registeradressierung bei arithmetisch-logischen Befehlen und die registerrelative Adressierung bei Transportbefehlen die Berechnung der neuen Adresse in der gleichen Zeitspanne wie der Zugriff auf den Datencache durchgeführt werden. Gelingt es beim Entwurf eines Rechners, den Zugriff auf den Programmcache, den Registersatz und den Datencache in der gleichen Zeit wie die Berechnung durch die ALU durchzuführen, erreicht man eine vollständig überlappende und homogene Befehlausführung mit hohem Durchsatz. Die Befehlphasen eines Befehlszyklus sind jetzt Instruction-Fetch (IF), Instruction-Decode (ID), Operand-Fetch (OF), Execute (EX), Memory (MEM) und Operand-Store (OS). Die letzte Phase wird meist Write-Back (WB) genannt, weil in dieser Phase Daten zurück in die Register geschrieben werden. Die Homogenisierung des Befehlszyklus bringt es mit sich, dass nicht bei jedem Befehl jede Phase des Befehlszyklus genutzt wird. So benötigen arithmetisch-logische Befehle die Phase MEM nicht und der Transportbefehl Store benötigt kein Write-Back (WB). Ein Befehlssatz mit homogenen Befehlphasen hat eine Mikroarchitektur, die streng zwischen Programm- und Datenspeicher unterscheidet (Harvard-Architektur). Da ein Universalrechner besser in Princeton-Architektur aufgebaut wird, werden universelle Registermaschinen mit jeweils einem Pro-

grammcache und einem Datencache ausgestattet. Der verbor- gene Cache suggeriert dem Programmierer Herr über einen von Neumann-Rechner zu sein, während die Mikroarchitek- tur von den Vorteilen der Harvard-Architektur profitiert. Eine solche Maschine ist eine gemischte Harvard-Princeton- Architektur.

Um einen Befehlszyklus mit homogenen Befehlsphasen zu erhalten, sollte das Befehlswort selbst ebenfalls homogen sein. Dafür legt sich ein Rechnerarchitekt auf möglichst wenige und einheitliche Befehlsworte fest. In der Regel ist dies ein Format für arithmetisch-logische Befehle, ein Format für Trans- portbefehle und jeweils ein Format für bedingte und unbeding- te Sprünge. Die Abb. 6.23 gibt einen Überblick über einfache Befehlsformate und die dazugehörigen Befehlszyklen. Da- bei gibt es leichte Unterschiede in den konkreten Implemen- tierungen derartiger Maschinen. In den in der Abbildung ge- zeigten Formaten ist f eine Codierung des jeweiligen Formats, o steht für den Opcode, c für Bedingungen („condition“) und x, y und z jeweils für die Adressen der Operandenquell- und - zielregister. Bei einer bedingten Verzweigung wird das Befehlswort mit einem Teil der Zieladresse aufgefüllt (d, Displace- ment). Alternativ kann an dieser Stelle auch eine Konstan- te bei der Konstantenadressierung (immediate) stehen. Skiz- ziert wird eine Auswahl von Formaten der SPARC- und der MIPS-Architektur. Je nach Befehlstyp unterscheiden sich die jeweiligen Formate. Unterschiedliche Implementierungen der Mikroarchitektur können auch unterschiedliche Befehlspha-

6

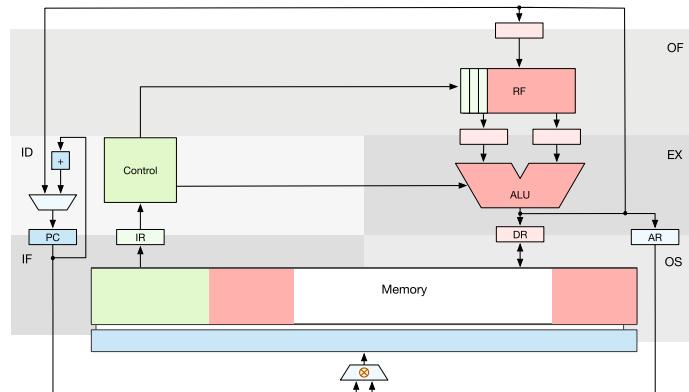
SPARC	MIPS	Befehlsphasen																																																				
Sprung-Format	J-Format	JUMP ① IF ID OF EX M OS BRANCH ② IF ID OF EX M OS																																																				
<table border="1"> <tr> <td>f</td> <td>\$d</td> <td>op</td> <td>\$d</td> </tr> <tr> <td>f 0</td> <td>c o</td> <td></td> <td>\$d</td> </tr> </table>	f	\$d	op	\$d	f 0	c o		\$d	<table border="1"> <tr> <td></td> <td></td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> </tr> </table>																																													
f	\$d	op	\$d																																																			
f 0	c o		\$d																																																			
Transport-Format	I-Format	STORE ③ IF ID OF EX M OS LOAD ④ IF ID OF EX M OS BRANCH ⑤ IF ID OF EX M OS BRANCH ⑥ IF ID OF EX M OS																																																				
<table border="1"> <tr> <td>f</td> <td>\$z</td> <td>o</td> <td>\$d</td> <td>op</td> <td>\$s</td> <td>\$t</td> <td>\$d</td> </tr> <tr> <td>f \$z</td> <td>o i</td> <td></td> <td></td> <td>op \$s</td> <td>\$t i</td> <td></td> <td></td> </tr> </table>	f	\$z	o	\$d	op	\$s	\$t	\$d	f \$z	o i			op \$s	\$t i			<table border="1"> <tr> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </table>																																					
f	\$z	o	\$d	op	\$s	\$t	\$d																																															
f \$z	o i			op \$s	\$t i																																																	
Register-Format		ALU ⑦ IF ID OF EX M OS ALU ⑧ IF ID OF EX M OS																																																				
<table border="1"> <tr> <td>f</td> <td>\$z</td> <td>o</td> <td>\$x</td> <td>0</td> <td>i</td> <td>\$y</td> <td>op</td> <td>\$d</td> <td>\$s</td> <td>\$t</td> <td>sht</td> <td>func</td> </tr> <tr> <td>f \$z</td> <td>o \$x</td> <td>1 i</td> <td></td> </tr> </table>	f	\$z	o	\$x	0	i	\$y	op	\$d	\$s	\$t	sht	func	f \$z	o \$x	1 i											<table border="1"> <tr> <td></td> </tr> <tr> <td></td> </tr> </table>																											
f	\$z	o	\$x	0	i	\$y	op	\$d	\$s	\$t	sht	func																																										
f \$z	o \$x	1 i																																																				

Abb. 6.23 Befehlsformate und -zyklen von universellen Registermaschinen

sen erzeugen. Als Beispiel sollen Sprung- und Verzweigungsbefehle betrachtet werden. Soll beispielsweise die ALU für die Berechnung der relativen Sprungadresse verwendet werden, so ist es notwendig, bei der Befehlsausführung unbedingt die Phasen Operand-Fetch und Execute einzuhalten. Das hat aber den Nachteil, dass im Falle eines Sprungs eine Phase zu viel ausgeführt wird und somit bereits zwei weitere Befehle geladen wurden. Könnte man nun die Adressberechnung durch zusätzliche Hardware in die Operand-Fetch-Phase verlegen, würde sich ein geringfügig höherer Durchsatz ergeben. Bei Millionen von Befehlen macht das bereits einen Unterschied. Auch bei bedingten Verzweigungen lassen sich derartige Optimierungen durchführen: Statt den Vergleich für den Sprung in der Execute-Phase auszuführen, verschiebt man diesen durch zusätzliche Vergleichshardware in die Phase davor. Dies bietet sich insbesondere bei dem MIPS an. In einer Architektur mit homogener Fließbandverarbeitung sind die Formate für die arithmetisch-logischen Befehle am einfachsten. Bis auf die Speicherphase unterstützen sie alle Phasen des Befehlszyklus und müssen lediglich die Operandenregister spezifizieren und die korrekte ALU-Funktion ansprechen.

Eine Maschine mit universellem Registersatz und homogenen Befehlsformaten und Befehlszyklen wird RISC genannt. Durch die Beschränkung auf einige wenige homogene Befehlsformate und eine daraus resultierende gleichförmige Befehlausführung reduziert sich die Komplexität des Befehlssatzes. Dies folgt zwangsläufig aus der Reduzierung der Adressierungsarten und der Begrenzung dieser auf jeweils nur eine Befehlsgruppe. Gegenüber den klassischen CISC-Maschinen erzielt eine RISC-Maschine durch dieses Vorgehen einen wesentlich erhöhten Befehlsdurchsatz.

Die Abb. 6.24 zeigt eine universelle Registermaschine mit den dazugehörigen Befehlsphasen. Um diese in eine Mikroarchitektur für maximale Überlappung oder Fließbandverarbeitung umzuwandeln, muss die Hardware in mehreren Schritten angepasst werden. Zunächst einmal sind die Pfade für die Befehlsdecodierung und die Datenverarbeitung zu trennen. Dies geschieht dadurch, dass Befehle im Befehlscache und Daten im Datencache zwischengespeichert werden. Um diese beiden resultierenden Pfade konzeptionell klar zu trennen, sind der Befehlspfad der Abb. 6.25 waagerecht und der Datenpfad senkrecht gezeichnet. Beide Wege müssen für die Fließbandverarbeitung ausgelegt sein. Im Befehlspfad werden die Phasen Instruction-Fetch und Instruction-Decode angeordnet. Nach der Entschlüsselung des Befehls müssen Teile von diesem durch die Pipeline weitergereicht werden. Zum Beispiel muss die ALU in der Phase Execute richtig konfiguriert werden. Aber auch die Zieladresse für die Phase Write-Back ist über den gesamten



6

Abb. 6.24 Aufbau einer universellen Registermaschine

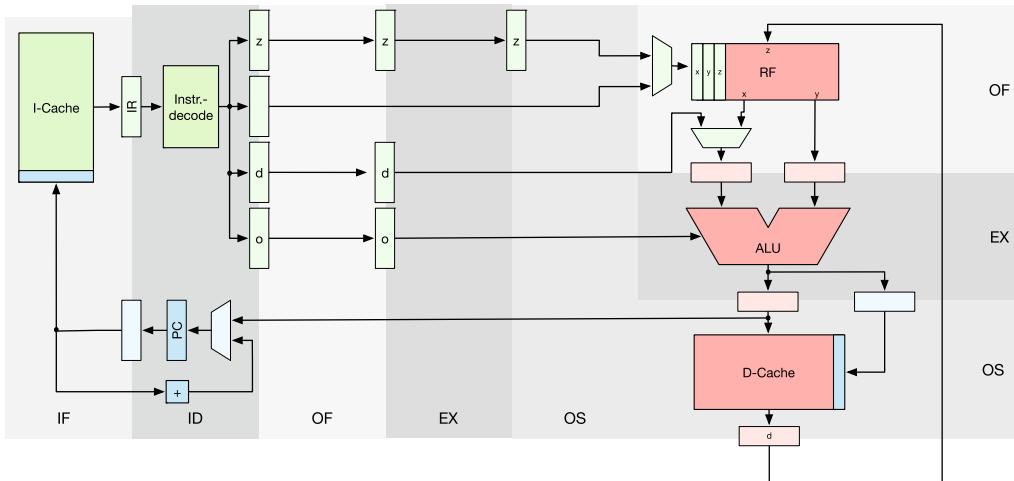


Abb. 6.25 Mikroarchitektur für Fließbandverarbeitung

Zyklus durch die Hardware durchzureichen. Demgegenüber steht die Pipeline für den Datenpfad, beginnend mit der Phase Operand-Fetch aus dem Registersatz über die Phase Execute mit der ALU und den Zugriff auf den Datencache in der Phase MEM. Der letzte Schritt ist dann das Write-Back (Operand-Store) in die Register.

Die Bearbeitung eines Befehls in der in Abb. 6.25 dargestellten Mikroarchitektur läuft wie folgt ab. Zunächst wird die Operation aus dem Befehlscache geholt. Dies ist der Befehl, auf den der Befehlszähler (engl. „program counter“, PC) zeigt. Dieser wird anschließend im Befehlsregister (engl. „instruction register“, IR) gespeichert und in der nächsten Phase (ID) entschlüsselt. Je nach Opcode oder Befehlsformat müssen nun

die Adressen für die Quellregister zum Registersatz weitergeleitet werden. Dazu werden sie in einem für den Programmierer unsichtbaren Register zwischengehalten. Die Zieladresse für das Ergebnis muss dagegen durch die Pipeline weitergereicht werden. Während der Phase OF liegen nun die Quellregisteradressen an der Registerdatei an, und die Inhalte dieser Register werden dann in den dafür vorgesehenen Zwischenregistern gespeichert. Nachdem die ALU den ebenfalls durch die Pipeline durchgereichten Befehl ausgeführt hat, wird das Ergebnis zunächst zwischengespeichert, um anschließend unter Umgehung des Datencache in der WB-Phase in das Zielregister geschrieben werden zu können. Dazu wurde die Registeradresse des Ziels parallel durch die Pipeline geschoben. Der Zugriff auf den Datencache erfolgt nur bei einem Load- oder Store-Befehl. Um dies zu erreichen werden mithilfe der ALU eine Adressberechnung durchgeführt und das Ergebnis anschließend an den Datencache angelegt. Dazu muss der Teil der Adresse aus dem Befehlswort (das Displacement, d) ebenfalls durch die Pipeline geschoben und in den Datenpfad eingespeist werden. Nach dieser Berechnung können der Cache gelesen und ein Datenwort in der Write-Back-Phase in das entsprechende Register geschrieben werden. Im Gegensatz dazu muss bei einem Store-Befehl vor der Adressberechnung der Inhalt eines Registers in der Operand-Fetch-Phase im unsichtbaren Datenregister vor dem Datencache gespeichert werden. Dazu werden die entsprechende Registeradresse an den Registersatz angelegt und alle Zwischenregister und die ALU durchgeschaltet. Anschließend kann diese dann verwendet werden, um die Adresse zum Speichern des Datums zu berechnen.

Mithilfe der überlappenden Verarbeitung von Befehlen können der Durchsatz der Programmausführung und dadurch die Auslastung der Komponenten des Rechenwerks erheblich erhöht werden. Dies gilt insbesondere, wenn die durchzuführenden Berechnungen unabhängig voneinander sind. Dieser Fall ist leider nur theoretisch. In der Praxis zeichnen sich Programme durch ein hohes Maß an Kontroll- und Datenabhängigkeiten aus. Abhängigkeiten im Kontrollfluss sind Schleifen oder Verzweigungen. Dabei ist die Ausführung bestimmter Teilprogramme oder Programmsequenzen abhängig von Bedingungen, die im Programmfluss vor oder auch nach einer Berechnung stattfinden. Datenabhängigkeiten von Rechenoperationen ergeben sich aus dem Zugriff auf gemeinsame Variablen. Der Kontrollfluss eines Programms lässt sich mithilfe eines Kontrollflussgraphen darstellen. In diesem Zusammenhang spricht man auch von einem Basisblockgraphen. Basisblöcke sind Sequenzen von Befehlen, die linear ablaufen, d. h., es handelt sich um Codefragmente, die keine bedingten Verzweigungen oder Sprünge enthalten. Knoten im Kontrollfluss-

graphen sind dann jeweils die Basisblöcke und Kanten modellieren den Kontrollfluss zu den Blöcken. Ein Datenflussgraph beschreibt den Datenfluss in einem Basisblock. Die Knoten sind Operationen und die Kanten die Variablen dieser. Abhängigkeiten ergeben sich dann, wenn ein Befehl Daten schreibt, die eine nachfolgende Anweisung liest. Es ist leicht ersichtlich, dass Operationen, zwischen denen keine Datenabhängigkeiten bestehen, umsortiert werden können, ohne die Semantik des Programms zu verändern. Sowohl der Kontroll- als auch der Datenfluss können zu Konflikten (engl. „hazards“) in der Fließbandverarbeitung führen. Konflikte entstehen, wenn Befehle in unterschiedlichen Phasen auf die gleiche Ressource des Rechenwerks zugreifen wollen. Es wird zwischen Struktur-, Daten- und Steuerkonflikten unterschieden. Konflikte in der überlappenden Befehlsausführung führen immer zu einem Anhalten der Pipeline (engl. „stall“, abreißen oder abwürgen). Allerdings lässt sich durch spezielle Mikroarchitekturstrukturen, den Interlocks, und einer Sprungvorhersage ein solcher Pipeline-Stall vermeiden.

In einer reinen von Neumann- oder Princeton-Architektur führt der Zugriff auf den Speicher beim Holen eines Befehls (IF-Phase) zu einem Strukturkonflikt mit einem Speicherzugriff einer anderen Anweisung in der Phase der Operandenbereitstellung (OF), da in beiden Stufen auf den Speicher zugegriffen wird. Das Gleiche gilt, wenn ausschließlich auf einem Registersatz gearbeitet wird. Gleichzeitiges Lesen und Schreiben in den Phasen OF und OS ist mit einem Ein-torspeicher nicht möglich. Es empfiehlt sich daher bei einer universellen Registermaschine den Speicher für die Register als 3-Tor-Speicher zu implementieren. Ein weiterer Strukturkonflikt entsteht bei Unterbrechungen oder in der Ausnahmebehandlung. Führt beispielsweise eine Division durch 0 zu einer durch die ALU ausgelösten Ausnahme, so muss der Stand des Befehlszählers (PC) in einem Register gerettet werden. Will gerade ein vorangehender Befehl in seiner Write-Back-Phase (OS) Ergebnisse in den Registersatz schreiben, kommt es ebenfalls zu einem Strukturkonflikt. Bei einem 3-Tor-Speicher lässt sich dieser vermeiden, wenn ein Register für die Ausnahmebehandlung und ein weiteres Register für Unterbrechungen reserviert werden und diese ausschließlich zu diesem Zweck genutzt werden.

Um eine arithmetisch-logische Operation fertigzustellen, muss ein Befehl alle fünf Stufen der Pipeline (IF-ID-OF-EX-MEM-OS) durchlaufen. Erst wenn das Ergebnis im Zielregister gespeichert ist, ist die Anweisung aus Sicht des Programmierers semantisch korrekt abgeschlossen. Bei einer Datenabhängigkeit entsteht ein Konflikt, wenn ein auf einen arithmetisch-logischen Befehl folgendes Kommando auf ein

Datum zugreifen möchte, bevor die vorangehende Operation die Phase OS verlassen hat. Eine Datenabhängigkeit zwischen zwei Befehlen, die ein Lesen eines Ergebnisses vor der vollständigen Abarbeitung des Vorgängerbefehls notwendig macht, heißt wahre oder echte Datenabhängigkeit („true data dependency“). Weitere Varianten der Datenabhängigkeit in einer Mikroarchitektur mit Fließbandverarbeitung heißen Gegenabhängigkeit („anti-dependency“) und Ausgabeabhängigkeit („output dependency“).

Eine echte Datenabhängigkeit besteht dann, wenn eine Operation auf ein Datum zugreift, bevor ein Vorgängerbefehl das Ergebnis seiner Berechnungen im Zielregister speichern konnte.

Eine solche echte Datenabhängigkeit und der entsprechende Konflikt sind in Abb. 6.26a dargestellt. Ein solcher Pipelinekonflikt heißt auch Read-after-write (RAW)-Hazard. In einer Pipeline, in der die einzelnen Phasen des Befehlszyklus strikt hintereinander ausgeführt werden und in jeder Phase nur eine Hardwareressource im Rechenwerk zur Verfügung steht, ist der RAW-Hazard der am häufigsten vorkommende Pipelin konflikt. Ein solcher Konflikt kann auf drei unterschiedliche Arten behoben werden:

- Anhalten der Pipeline und warten, bis der Vorgänger-Befehl abgearbeitet wurde.
- Einbau zweier oder mehrerer NOP-Befehle durch den Compiler. In dem Fall muss die Hardware keine Hazards detektieren, und der normale Befehlsfluss bleibt gewahrt. Wenn es möglich ist, können zusätzliche NOP-Befehle durch Umsortieren des Codes vermieden werden. Sind im Programm Anweisungen vorhanden, die keine Datenabhängigkeiten untereinander und zu den zwei betrachteten Befehlen aufweisen, können diese durch den Compiler zwischen die beiden abhängigen Befehle geschoben werden. Der Prozessor kann in diesem Fall ohne die Pipeline anzuhalten die überlappende Ausführung weiter durchführen.
- Einbau einer Hardware, die Datenkonflikte automatisch erkennt und mithilfe von Umgehungsschaltungen selbstständig behebt. Ein solches Vorgehen heißt Interlock oder Pipeline-Interlock.

Datenabhängigkeit

Eine Gegenabhängigkeit ist der umgekehrte Fall zu einer wahren Datenabhängigkeit. Wenn ein vorangestellter Befehl eine Variable oder besser ein Register beschreibt, geschieht dies erst in der letzten Phase des Zyklus. Greift die nachfolgende Anweisung aber vorher auf dieselbe Variable zu – beispielsweise bei einem Store-Befehl, der den Inhalt des Registers in den Speicher schreibt –, dann würde das Lesen vor dem Schreiben erfolgen, und die Reihenfolgesemantik der Operation wäre ver-

Gegenabhängigkeit

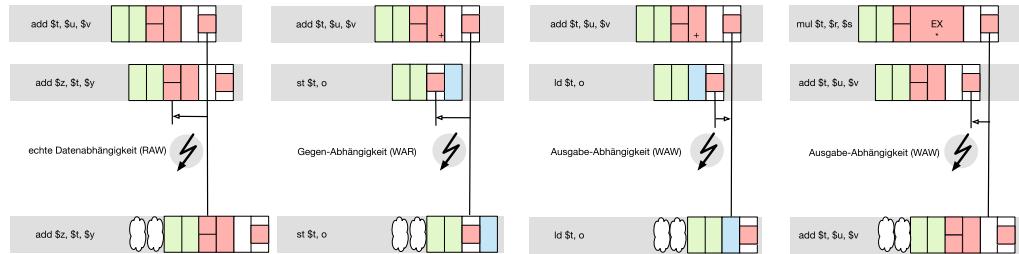


Abb. 6.26 Datenkonflikte bei überlappender Befehlausführung

6

Ausgabeabhängigkeit

letzt. Ein Beispiel zeigt Abb. 6.26b. Eine Gegenabhängigkeit führt zu einem Write-after-read-Hazard in der Fließbandverarbeitung.

Stehen in einem Rechenwerk mehrere Recheneinheiten zur Verfügung, beispielsweise wenn neben einer ALU noch eine Fließkommamultiplikationseinheit vorhanden ist, können Ausgabeabhängigkeiten entstehen. Gegenabhängigkeiten führen zu Write-after-write (WAW)-Hazards. Angenommen, die Befehlsfolge

$$t = r o_1 s$$

$$t = u o_2 v$$

soll ausgeführt werden, und für die Operation o_1 und den Befehl o_2 ist jeweils eine eigene Recheneinheit in der Mikroarchitektur des Rechenwerks implementiert. Angenommen, die Rechenoperation o_1 benötigt die doppelte Ausführungszeit der Anweisung o_2 . In dem Fall würde das Ergebnis des 1. Befehls das Ergebnis der 2. Operation überschreiben. Das Programm würde also nicht mit der korrekten Semantik ausgeführt werden. In der Abb. 6.26d ist ein Beispiel gezeigt. Die früher ausgeföhrte Multiplikation überschreibt das Ergebnis der später begonnenen Addition. In einer klassischen Pipeline mit jeweils einer Ressource für jede Phase des Befehlszyklus kann dieser Konflikt nicht auftreten. In Architekturen mit vielen parallelen Recheneinheiten wird dies jedoch häufig passieren.

Wahre oder echte Datenabhängigkeiten können schaltungstechnisch beseitigt werden. Ein solches Verfahren heißt Interlock. Die Abb. 6.27 zeigt eine Interlock-Hardware. Zunächst einmal sind in den Phasen Execute und Operand-Store jeweils zwei Vergleiche zu implementieren. Mit diesen wird geprüft, ob in diesen Phasen der überlappenden Ausführung gleichzeitig auf Register zugegriffen wird. Im Fall eines RAW sind zwei Fälle denkbar: Der Zugriff auf die im vorherigen Befehl zu verändernde Variable erfolgt direkt nach der Execute-Phase des Vorgängers. In diesem Fall muss die

6.3 · Mikroarchitektur

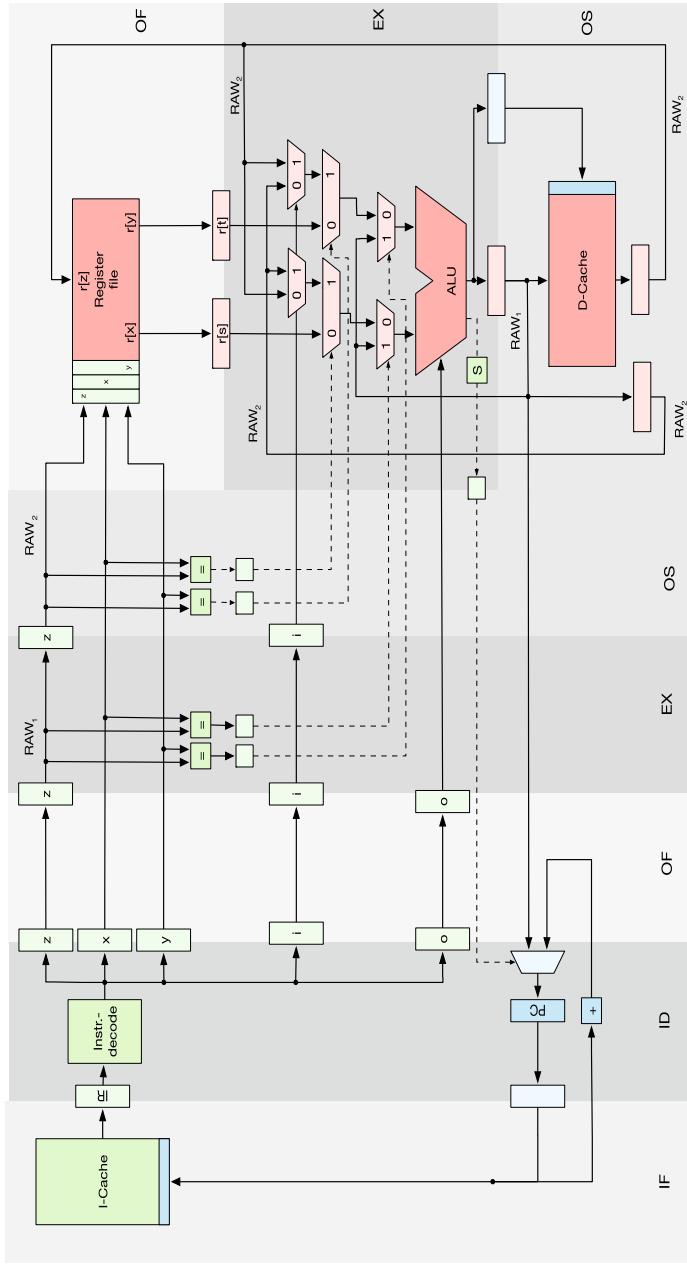


Abb. 6.27 Interlock-Hardware zur Vermeidung von RAW-Hazards

ALU umgangen werden. Im Bild ist dies durch den Pfad RAW_1 gekennzeichnet. RAW_1 bedeutet, dass der aufzulösende Hazard um eine Phase verschoben ist. Im zweiten Fall steht zwischen den beiden Anweisungen noch ein weiterer Befehl, oder

aber der Zugriff erfolgt erst in einer späteren Phase. Ein RAW-Hazard über zwei Phasen ist in der Zeichnung als RAW_1 -Pfad gekennzeichnet. Der RAW_2 -Hazard greift auf den Datenpfad nach dem Datencache zu und leitet den Inhalt des entsprechenden, für den Programmierer unsichtbaren Datenregisters zurück zur ALU. Die Kaskade an Multiplexern vor der ALU regelt den Datenfluss in Abhängigkeit von der von der Hardware festgestellten Datenabhängigkeit. Eine Besonderheit ist dabei ein Load-Befehl vor einer ALU-Operation. In dem Fall kann die Pipeline erst nach dem Lesen des Datencache umgangen werden. Dazu wurden zwei weitere Multiplexer vor die ALU geschaltet, die nur umschalten, wenn ein Load-Befehl vorliegt. Dazu muss das Load ebenfalls durch die Pipeline geschoben werden. Die Beseitigung von RAW-Hazards ist aufwendig. Aus diesem Grund gibt es Implementierungen von Architekturen, die ganz auf eine Interlock-Hardware verzichten. In dem Fall müssen RAW-Hazards durch den Compiler aufgelöst werden, entweder durch Umsortieren der Befehle oder durch Einfügen von Leeranweisungen (NOP). Ein Beispiel hierfür ist die MIPS-Architektur („microprocessor without interlocked pipeline stages“).

Der am schwierigsten aufzulösende Konflikt in einer Pipeline ist der Steuerkonflikt. Steuerkonflikte treten bei bedingten Verzweigungen auf. Wenn ein Prozessor eine Verzweigung vornimmt, dann sind alle bis dahin bereits in die Pipeline geladenen und ausgeführten Anweisungen wertlos und zu verwerfen. Die überlappende Ausführung funktioniert nicht mehr. Steuerkonflikte einer Verzweigung können durch zusätzliche Hardware gelöst werden. Ein Rechner hat zwei Pipelines und führt die beiden Stränge der Verzweigung parallel aus. Erst im Moment der Entscheidung wird auf die richtige Pipeline umgeschaltet. Allerdings stößt diese Technik bei verschachtelten Schleifen (engl. „nested loops“) schnell an ihre Grenze, muss doch für jede Schleife eine eigene Pipeline vorgesehen werden. Da die Anzahl möglicher Pfade durch verschachtelte Schleifen exponentiell ansteigt, hieße das, je nach Programm, entsprechend viele Pipelines zu implementieren. Es ist leicht einzusehen, dass dies nicht wirtschaftlich ist.

Auflösen lässt sich dieser Konflikt durch zwei andere Techniken. Einige Rechnerarchitekturen verschieben durch zusätzliche Hardware und einen geschickten Befehlssatz die Verzweigungsentscheidung direkt in die Instruction-Decode-Phase. In diesem Fall befindet sich nur noch ein weiterer Befehl in der Pipeline. Statt diesen zu verwerfen wird dieser immer ausgeführt. Ist dieses Verhalten der Hardware klar spezifiziert, kann der Compiler wieder den Konflikt lösen, indem Operationen umsortiert werden. Wenn einfach ein Befehl ohne Datenabhängigkeit zur Verzweigung – was nicht unwahrscheinlich ist

– nicht vor dieser ausgeführt wird, sondern erst danach, ändert sich die Semantik des Programms nicht. Ein Steuerkonflikt wird vermieden. Gibt es keinen Befehl, der nach der bedingten Verzweigung noch ausgeführt werden kann, kann der Compiler wieder einen NOP-Befehl einfügen.

Eine Strategie, die insbesondere bei verschachtelten Schleifen effizient ist, ist die Sprungvorhersage (engl. „branch prediction“). Bei der spekulativen Ausführung vertraut man darauf, dass wenn eine bedingte Verzweigung genommen wurde, dies auch beim nächsten Durchlauf der Fall ist. Dies trifft insbesondere auf Schleifen zu. Führt ein Programm eine Schleife aus, ist die Wahrscheinlichkeit hoch, dass es auch im nächsten Durchlauf wieder die Schleife nimmt. Zu diesem Zweck kann die Hardware im Cache ein dafür vorgesehenes Bit verwenden. Immer wenn eine bedingte Verzweigung genommen wurde, wird diese Flagge in der entsprechenden Cache-Line gesetzt, und wenn eine Schleife nicht ausgeführt wird, wieder gelöscht. Je nachdem, ob das Bit wahr ist, wird also der Code der Schleife oder das Programm nach der Schleife durch die Pipeline geschoben. Manche Mikroarchitekturen sehen sogar mehr als 1 Bit zur Sprungvorhersage im Cache vor. Sind z. B. zwei Bit dafür reserviert, kann ein Sprungvorhersageautomat in der Hardware implementiert werden. Jede Kombination der beiden Sprungvorhersage Bits entspricht dann dem Zustand des Automaten. So kann die Kombination 00 bedeuten, dass ein Sprung kaum wahrscheinlich ist, die Kombination 01, dass ein Sprung wenig wahrscheinlich, die Kombination 10, dass er wahrscheinlich, und die Kombination 11, dass er sehr wahrscheinlich ist. Immer wenn ein Sprung ausgeführt wurde, zählt die Hardware die Sprungvorhersage um 1 nach oben, wenn kein Sprung durchgeführt wurde, nach unten. Mit dieser Technik lassen sich gute Sprungvorhersagen und eine spekulative Ausführung für verschachtelte Schleifen implementieren. Die Sprungvorhersage für n Schleifen wird umso genauer, je mehr Vorhersagebits implementiert sind. Im Fall von n Bit sind das dann jeweils 1 Bit und eine Vorhersage für genau eine Schleife.

6.3.4 Mikroarchitektur typischer Maschinen

Nach der Diskussion unterschiedlicher Konzepte zur Mikroarchitektur sollen im Folgenden konkrete Maschinen diskutiert werden. Dabei wurde der Versuch unternommen, die Rechner möglichst detailliert darzustellen. Dazu wurden alle Steuersignale spezifiziert. Auf die Leitungen zwischen Steuerwerk und Komponenten des Rechenwerks wurde aus Gründen der Übersichtlichkeit verzichtet. Allerdings lassen sich die Steu-

erleitungen mithilfe der angegebenen Etiketten in den Zeichnungen leicht zuordnen. Das Steuerwerk der Maschinen wurde angedeutet und auf eine vollständige Beschreibung des Mikroprogramms wurde verzichtet. In der jeweiligen Abbildung wird der Datenfluss (rot) getrennt vom Kontrollfluss (grün) und der Steuerung dargestellt. Grundlagen der Anbindung des Speichers (blau) und des Speicherlayouts der Software unterschiedlicher Maschinen werden ebenfalls besprochen. Die Ein- und Ausgabeeinheiten (gelb) sind zunächst noch dargestellt, werden dann im Verlauf der Diskussion weggelassen, da sich der Leser diese leicht selbst dazudenken kann. Insbesondere bei der universellen Registermaschine ist die zugehörige Fließbandverarbeitung bereits so kompliziert, dass Vereinfachungen vorgenommen werden mussten.

6.3.4.1 Stapelmaschine und ihre Befehlsformate

Ein sehr einfacher, mit wenigen Komponenten zu implementierender Computer ist die Stapelmaschine. Der Grund für die Einfachheit dieses Rechners ist der geringe Aufwand, mit dem der Speicher an die Recheneinheit angeschlossen werden kann. Der Klarheit der Speicherverwaltung und -anbindung steht aber ein komplexes Ausführungsmodell gegenüber. Eine Stapelmaschine implementiert direkt die umgekehrte polnische Notation (UPN) oder polnische Postfixnotation¹⁰. Um komplexe algebraische Ausdrücke in UPN zu formulieren, werden keine Klammern benötigt. Dies reduziert die Anzahl der Zeichen eines Programms und beschleunigt die Eingabe. Stapelmaschinen haben jedoch den Nachteil, dass, um eine Turingvollständige Maschine zu implementieren, mehrere Stapel notwendig sind und die Ausführung komplexer Algorithmen ein häufiges Umsortieren der Einträge der Stapel erfordern. Die umgekehrte polnische Notation ist für die meisten ungewohnt. Geübte Nutzer eines Taschenrechners können mit dieser aber schnell und effizient komplizierte Berechnungen durchführen. Die Einfachheit des Speicherns und des Zugriffs auf Operationen und deren Argumente wird daher mit einem komplexen Berechnungsmodell erkauft.

Eine sehr einfache Maschine kommt mit einem Stapel aus. Dabei werden zunächst die Argumente auf diesen gelegt und anschließend die durchzuführende Operation. Die Logik der Maschine kann jetzt durch eine einfache Verdrahtung der Komponenten aufgebaut werden, wenn der oberste Platz des Stacks direkt mit dem Konfigurationseingang der arithmetisch-logischen Einheit und die beiden folgenden Einträge des Stacks mit den Eingängen der Recheneinheit ver-

¹⁰ Die Präfixnotation beliebiger Ausdrücke geht auf den polnischen Mathematiker Jan Łukasiewicz zurück.

bunden werden. Das Ergebnis einer Operation mit der ALU wird wieder auf den Stapel gelegt. Dazu ist es erforderlich, den Stapel so zu bauen, dass nicht nur ein Eintrag, sondern maximal drei Einträge heruntergenommen oder überschrieben werden können. Um Berechnungen zu automatisieren, sind noch Leitungen notwendig, die den Füllstand des Stacks anzeigen. Der Stack dieser einfachen Mikroarchitektur wird über einen Multiplexer geschrieben. Dieser löst den Zugriffskonflikt zwischen der Recheneinheit und der Eingabeeinheit auf. Um zwischen Argumenten und Operationen unterscheiden zu können, signalisiert der Codierer der Tastatur dem Steuerwerk, um welche Form es sich bei der Eingabe handelt. Das Steuerwerk entscheidet dann, ob lediglich ein Datum auf den Stack gelegt, ob ein Befehl oder ein komplexeres Mikroprogramm beispielsweise zur Multiplikation ausgeführt wird.

Kernelement dieser Maschine ist die arithmetisch-logische Einheit. Ein Befehl oder eine Operation erlaubt es, diese Funktion der ALU zu konfigurieren. Liegt am S-Eingang der ALU ein entsprechender 8-Bit-Code an, werden die 2 Operanden, im Beispiel von Stack 3 kommend, entsprechend verknüpft und das Ergebnis von der ALU ausgegeben. Von der ALU gehen 4 Flaggensignale (*aluFlag*) direkt zum Steuerwerk. Diese zeigen an, ob ein Überlauf (*O*, engl. „overflow“) oder ein Übertrag (*C*, engl. „carry“) bei einer Addition entstanden ist; die Flaggen *Z* und *N* werden bei einer Subtraktion gesetzt. Ist das Ergebnis dieser 0, wird *Z* (engl. „zero“) markiert, ist es negativ, *N* (engl. „negative“).

Die Maschine verfügt über einen Stack, dessen obere beiden Plätze direkt mit der ALU verbunden sind. Daten können auf diesem mithilfe der Signale *push* und *pop* abgelegt oder heruntergenommen werden. Der Stack verfügt über zwei *Push*-Leitungen, um signalisieren zu können, dass ein neuer Eintrag beide Eingangsplätze löschen soll. Dies ist bei allen arithmetischen Operationen mit 2 Argumenten notwendig, denn dabei produzieren zwei Operanden jeweils 1 Ergebnis. Dieses ersetzt somit zwei Einträge auf dem Stack. Neben diesen Steuersignalen wird das Steuerwerk noch informiert, ob der Stack leer ist, ob er voll ist oder ob mehr als ein Element auf dem Stack liegt. Der oberste Eintrag des Stacks kann in dem Beispiel über eine Anzeige ausgegeben werden. Um Daten auf den Stack zu legen, muss der Datenfluss durch einen vorgeschalteten Multiplexer geregelt werden. Daten können von der Eingabe beispielsweise einer Tastatur aus der ALU oder einem zweiten Datenstack kommen (Abb. 6.28).

Die komplexe Ansteuerung des Stacks beim Entfernen der Einträge nach der Durchführung einer Operation und bei der Programmierung beliebiger Algorithmen kann vereinfacht werden. Wenn die Argumente und der Befehl jeweils auf ei-

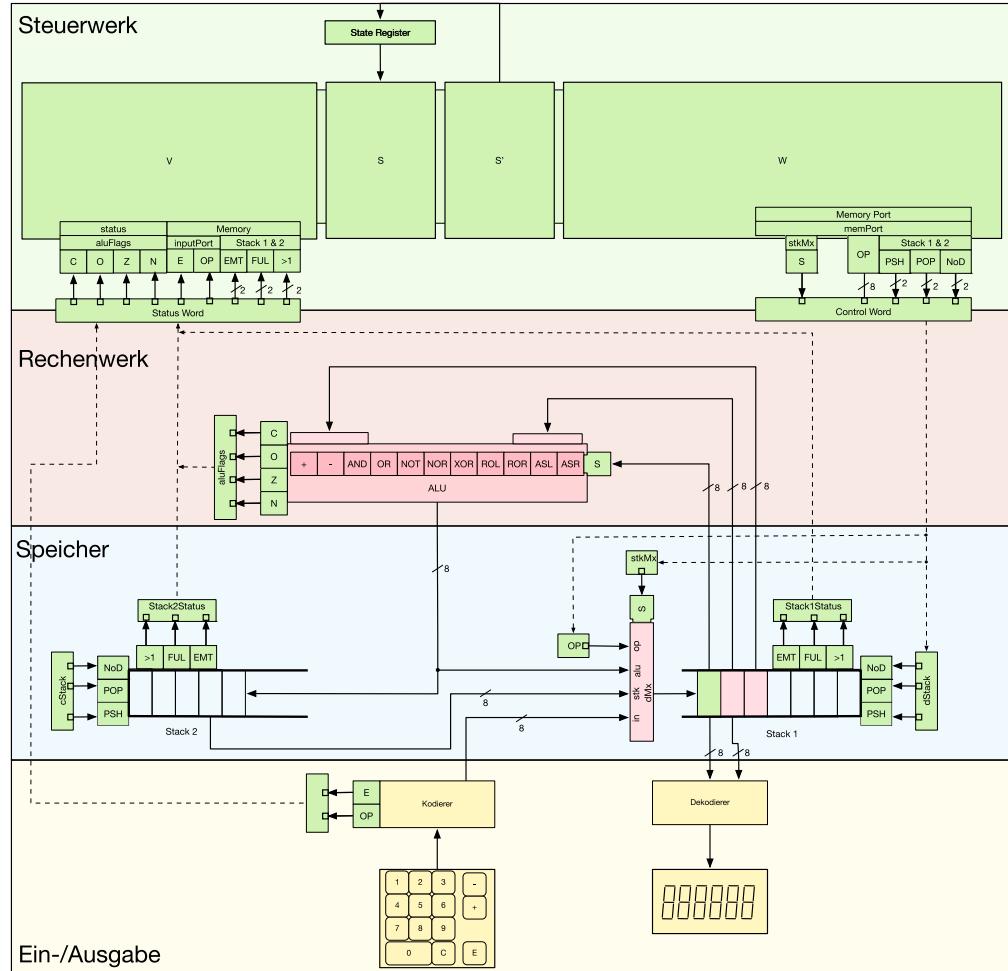


Abb. 6.28 Stapelmaschine mit zwei Stacks

nem eigenen Stapel abgelegt werden, dann müssen nur noch die oberen beiden Stellen des Datenstapels an die Recheneinheit angeschlossen werden. Durch diese Änderung der Mikroarchitektur und die damit erfolgte Trennung von Steuer- und Kontrollfluss ist es bereits möglich, Multiplikation und Division als sequenzielles Programm für eine Stapelmaschine zu implementieren. Insbesondere wenn noch ein zweiter Datenstapel vorgenommen wird, entfällt das häufige Umsortieren. Der Datentransport in diesem Rechnerentwurf wird wieder über Multiplexer gesteuert. Der in der Abb. 6.29 gezeigte Rechner verwendet drei Stapel: zwei als Speicher und einer im Rechenwerk. Die Maschine stellt einen einfachen UPN-Taschenrechner dar.

6.3 · Mikroarchitektur

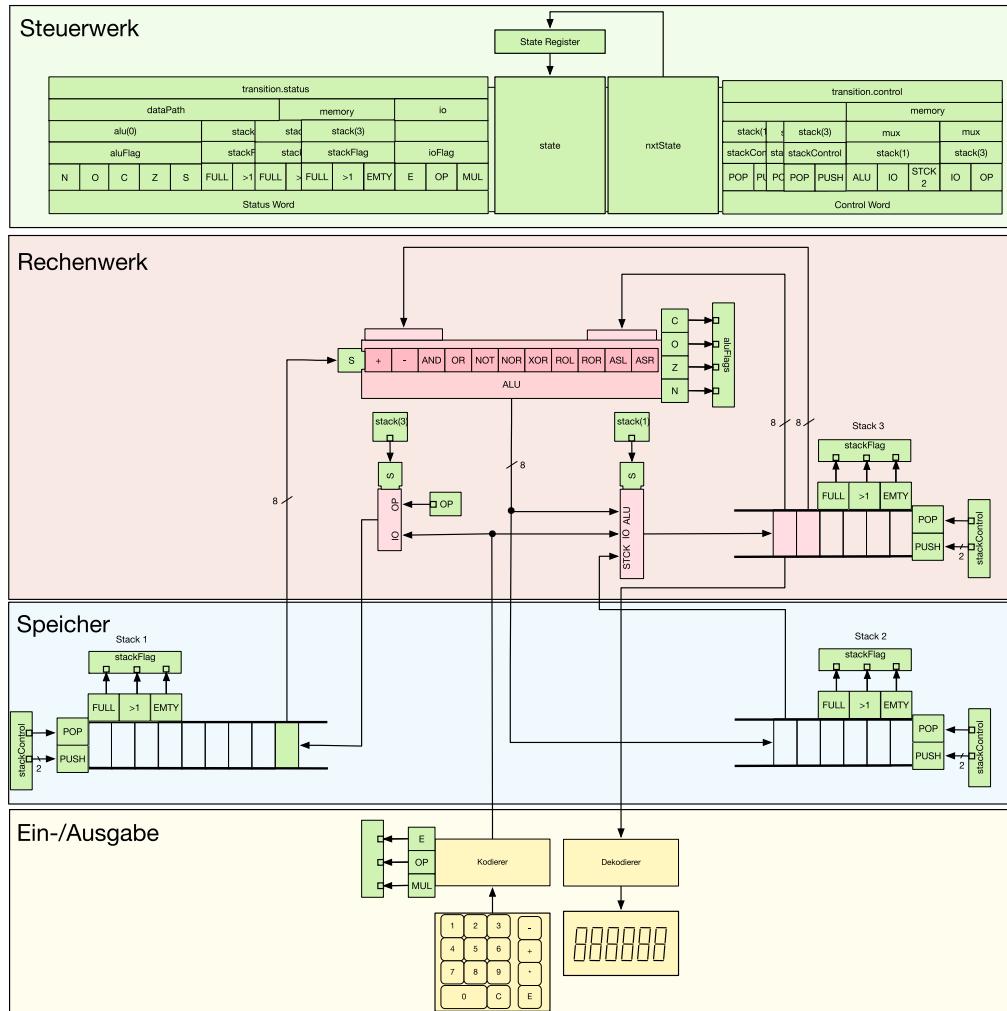


Abb. 6.29 Mikroarchitektur einer Stapelmaschine mit getrenntem Daten- und Kontrollfluss

Bisher wurde nicht unterschieden zwischen Speichern oder Stapeln des Rechenwerks oder einem Hauptspeicher. Bei einer Stapelmaschine mit drei Stapeln lässt sich einer der Datenstapel als Argumentspeicher der ALU interpretieren und Programm- und zweite. Datenstapel als Speicher für das Programm und seine Variablen. Das oberste Element des Befehlstapels ist direkt mit der Auswahlleitung der ALU verbunden. Operationen auf diesem Stapel stellen also ein Programm dar, das in umgekehrter Reihenfolge schrittweise abgearbeitet werden kann. Daten aus der ALU können wahlweise auf dem internen Stapel des Rechenwerks (Stack 3) oder im Stapel Stack

2 abgelegt werden. Dabei fungiert der Stapel Stack 2 als Partner von Stapel Stack 3. Der oberste Eintrag kann von Stapel Stack 2 auf den Stapel Stack 3 gelegt werden. Inhalte von Stapel Stack 3 können vom Steuerwerk über die ALU mit einem NOP auf Stapel Stack 2 gelegt werden. Damit kann die Reihenfolge der Daten auf Stapel Stack 3 beliebig vertauscht werden. Um derartige Operationen, also das Verschieben von Daten, oder auch um eine Sequenz von Befehlen für die Multiplikation zu erzeugen, kann das Steuerwerk eigene Befehle auf den Befehlsstapel Stack 1 legen. Dazu hat der Multiplexer vor dem Stapel einen weiteren Eingang.

Das Steuerwerk liest das Statuswort des Rechenwerks und gibt in Abhängigkeit seines inneren Zustands ein neues Kontrollwort aus. Dafür besitzt es den Eingang *transition.status*. In Abhängigkeit vom aktuellen Zustand *state* wechselt es in einen Folgezustand *nextState*. Der Ausgang des Steuerwerks ist das Steuerwort *transition.control*. Sowohl Statuswort als auch Steuerwort sind in einzelne Bereiche unterteilt. Es wird unterschieden zwischen Signalen, die vom oder zum Rechenwerk (*datapath*) laufen, vom oder zum Speicher (*memory*) oder die die Ein-/Ausgabe (I/O) steuern. Da die Ansteuerung der Stapel immer gleich ist, ist in der Zeichnung jeweils nur eine dargestellt, während die anderen halb verdeckt sind. Die Leitungen vom Steuerwerk zu den Komponenten der Ein- und Ausgabe, zum Speicher oder zum Rechenwerk sind nicht gezeigt. Die jeweiligen Verbindungen sind durch Marken an den entsprechenden Stellen angedeutet.

Die dargestellte Mikroarchitektur implementiert einen Taschenrechner. Dieser besitzt eine einfache alphanumerische Tastatur mit einem Codierer und eine 7-Segment-Anzeige mit entsprechendem Decodierer. Über die Signale *E* (Enter), *OP* (Operation) oder *mul* (Multiplikation) wird das jeweilige Steuerprogramm im Steuerwerk ausgewählt. Da das Programm *mul*, das eine Multiplikation als Mikroprogramm darstellt, mithilfe der Operationen Schieben und Addieren umgesetzt wird, ist es notwendig, dass die Eingabe das Vorliegen dieser Operation getrennt signalisiert.

6.3.4.2 Akkumulatormaschine und ihre Befehlsformate

Komplexe Operationen wie die Multiplikation, die aus mehreren Additionen zusammengesetzt wird, sind mit einer Stapelmaschine algorithmisch aufwendig zu realisieren. Es zeigt sich, dass die Einführung eines speziellen Registers des Akkumulators die Komplexität einer so einfachen Rechenoperation wie der Multiplikation oder Division erheblich reduzieren kann. Dabei fungiert der Akkumulator als Zwischenspei-

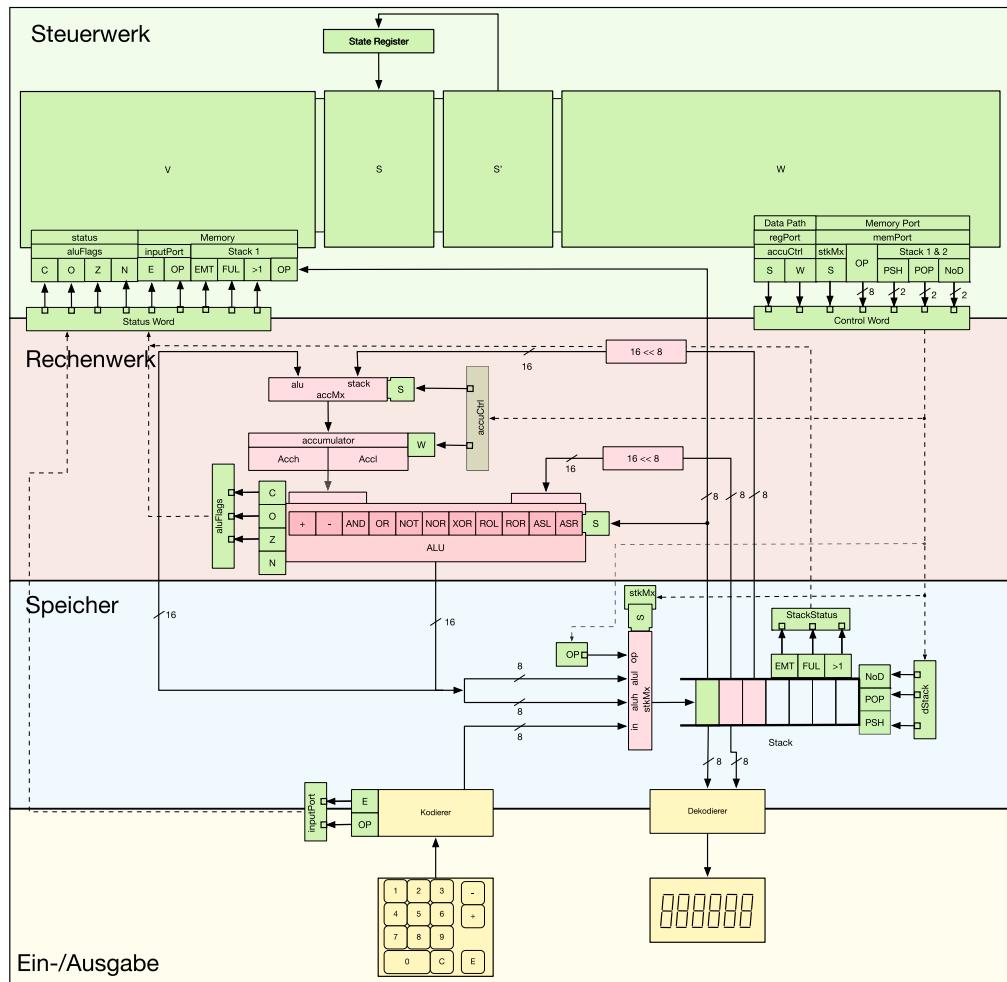


Abb. 6.30 Stapelmaschine mit Akkumulator

cher, der die jeweiligen Berechnungsergebnisse aufnimmt und im nächsten Rechenschritt der ALU gleich wieder zur Verfügung stellt. Dadurch können Speicherzugriffe vermieden werden und Programme laufen effizienter ab. Mit anderen Worten, der Akkumulator muss nicht adressiert werden. Dadurch reduziert sich zum einen der Aufwand an Hardware im Rechenwerk, zum anderen wird auch die Steuerung der Berechnungen vereinfacht (Abb. 6.30).

Die Stapelmaschine wird in ihrer Mikroarchitektur nun so modifiziert, dass die Ergebnisse der arithmetisch-logischen Einheit direkt in ein Register geschrieben werden. Der Akkumulator der gezeigten Maschine arbeitet dabei mit einer

Wortbreite, die dem Doppelten der allgemeinen Wortbreite des Computers entspricht. Dies erleichtert die Implementierung von Multiplikations- und Divisionsalgorithmen. Da der Stapel nur eine Wortbreite von 8 Bit hat, muss ggf. der Inhalt des Akkumulators sequenziell in 2 Stufen auf dem Stapel abgelegt werden. Die Recheneinheit ist dann ebenfalls für 16-Bit-Operanden auszulegen. Aus diesem Grund müssen Werte, die vom Stapel genommen werden, vorzeichenerweitert werden. Wir erinnern uns, das Vorzeichen einer vorzeichenbehafteten Zahl markiert, ob es sich um einen positiven oder einen negativen Wert handelt. Wird das höchste Bit in die zu erweiternden Ziffern geschrieben, bleiben Wert und Vorzeichen der Zahl erhalten. Die Vorzeichenerweiterung ist leicht in Hardware aufzubauen. So muss immer das höchstwertige Bit in die zu definierenden Ziffern kopiert werden. Zusätzlich zum Akkumulator wird noch ein Multiplexer benötigt. Dieser steuert den Datenfluss vom Stapel oder von der Recheneinheit und löst den Zugriffskonflikt auf den Akkumulator auf.

6.3.4.3 Registermaschine und ihre Befehlsformate

Ein Rechenwerk mit Akkumulator kann nun mit einem oder mehreren Stapspeichern verwendet werden. Andererseits ist es aber auch möglich, an den zweiten Eingang der ALU direkt einen Speicher mit wahlfreiem Zugriff anzuschließen.

In der Geschichte des Rechnerbaus zeigte sich schnell, dass die Automatisierung der Datenverarbeitung, also die vollständige automatische Abarbeitung von Algorithmen, zu einem komplexen Steuerwerk führt. Aufwendig müssen bei Änderung des Algorithmus Drähte und Verbindungen in diesem verändert werden. Bei der Stapelmaschine mit einem getrennten Stapel für Operationen kann man schon erkennen, dass dies nicht notwendig ist: Die Befehle selbst können auch im Speicher stehen. Um einen Rechner flexibel programmieren zu können, sind aber mehrere Probleme zu lösen. Zunächst ist die Mikroarchitektur passend zu wählen: Genau wie bei den vorher diskutierten Stapelmaschinen ist zu entscheiden, ob Operationen und Operanden im gleichen Speicher stehen. Danach ist die zeitliche Abfolge der Befehle zu planen. Dazu muss das richtige Ausführungsmodell gewählt werden. Die Frage, Programm und Daten gemeinsam oder getrennt zu speichern, führt zu einem grundlegenden Prinzip programmierbarer Maschinen. Dabei wird sich zeigen, dass performante und flexible Computer sich nicht unbedingt ausschließen.

Am einfachsten ist es, Operationen und Programm konzeptionell zu trennen. Jeweils ein Speicher mit wahlfreiem Zugriff wird für die Befehle und einer für die Operanden vorgesehen. Dies entspricht im Wesentlichen dem Konzept der Stapelma-

schine mit getrenntem Kontrollfluss. Am Ende wird bei dieser Mikroarchitektur nur der Stapelspeicher für Programm und Daten durch wahlfreien Speicher ersetzt, und der Stapel für die Operanden wird zu einem Registersatz (engl. „register file“). Dadurch, dass Programm und Daten konzeptionell getrennt sind, sind auch Steuerwerk und Rechenwerk klar abzugrenzen. Jeder Befehl hat als Argument entweder die Adresse eines Registers oder die Adresse eines Datums aus dem Datenspeicher. Verfügt die Maschine über einen Akkumulator, kann auf die Adresse des Zielregisters im Operanden verzichtet werden. Eine Architektur, in der Programm und Daten in unterschiedlichen Speichern liegen, heißt Harvard-Architektur¹¹.

Definition 6.3.1 – Harvard-Architektur

Bei der Harvard-(Mikro-)Architektur sind Programm und Daten in zwei verschiedenen wahlfreien Speichern abgelegt. Damit kann der Rechner gleichzeitig auf Programm und Daten zugreifen.

Ist das Rechenwerk mit einem Akkumulator zum Zwischen speichern von Ergebnissen ausgestattet, handelt es sich um eine einfache Registermaschine. Im Unterschied zu einer Stapel maschine müssen aber die Befehle im Befehlsspeicher aus gewählt adressiert werden. Dies geschieht mit einem Zeiger auf den aktuell auszuführenden Befehl: dem Befehlszähler (engl. „program counter“). Erweitert man die Steuereinheit noch um einen weiteren Zähler, einen Stapelzeiger oder Stack-Pointer, kann mit wahlfreiem Speicher ein Stapel emuliert werden. Mit den entsprechenden Befehlen im Befehlssatz kann ein Rechner mit wahlfreiem Speicher also die bisher beschriebenen Konzepte in einer Maschine vereinen.

Die Harvard-Architektur hat den Nachteil, dass die beiden Speicher nicht effizient genutzt werden. Oftmals ist nicht klar, wie aufwendig ein Algorithmus ist oder wie viele Befehle ein Programm besitzt. Parallel zum Konzept der Harvard-Architektur hat sich daher die von Neumann-Maschine entwickelt. Bei einer von Neumann-Maschine oder Princeton-Architektur ist das Programm im gleichen Speicher abgelegt wie seine Daten. Dieser kann so effizient genutzt werden. Ein langes Programm mit wenigen Daten kann die gleiche Hardware verwenden wie ein kurzes Programm mit vielen Daten.

Einfache
Registermaschine

Princeton-Architektur

¹¹ Streng genommen müsste es Harvard-Mikroarchitektur heißen. Da sich der Begriff Mikroarchitektur erst Jahrzehnte nach dem Wort Rechnerarchitektur herausbildete, ist diese Ungenauigkeit historisch begründet.

Definition 6.3.2 – Princeton- oder von Neumann-Architektur

Die Princeton-(Mikro-)Architektur hat einen wahlfreien Speicher, in dem Programm und Daten gemeinsam liegen.

von Neumann-Flaschenhals

Allerdings wird dieser Vorteil mit einigen Nachteilen erkauft. Zum einen benutzen Programm und Daten die gleichen Leitungen zum Steuer- und Rechenwerk. Dadurch ergibt sich ein Ressourcenkonflikt, der nur über eine zeitliche Trennung der Zugriffe aufgelöst werden kann. Die Ausführung des Programms verlängert sich. Dieser Effekt wird von Neumann-Flaschenhals genannt, da theoretisch mit einer Harvard-Architektur doppelt so schnell auf den Speicher zugegriffen werden kann.

Zum anderen können bei einem gemeinsamen Speicher Daten jederzeit zu einem Programm werden. Das erlaubt selbst modifizierenden Code. Eine derartige Maschine ist naturgemäß anfällig für externe Manipulationen. Der gemeinsame Speicher ist daher eine strukturelle Sicherheitslücke, die durch Mechanismen des Speicherzugriffsschutzes geschlossen werden muss. Um den Zugriff auf bestimmte Speicherbereiche zu schützen, besitzen moderne Architekturen einen Supervisor- und einen User-Mode. Im User-Mode sind dann das Lesen und Schreiben bestimmter vom Betriebssystem genutzter Adressen verboten.

Princeton-Architektur

Die Abb. 6.31 zeigt die Mikroarchitektur einer Registermaschine in Princeton-Architektur. Im Vergleich zum Computer in Harvard-Architektur sind ein Stapelzeiger (engl. „stack pointer“) und zwei Indexregister hinzugekommen. Mit dem Stapelzeiger kann in dem wahlfreien Speicher ein Stapel simuliert werden. Die Abbildung zeigt zusätzlich ein typisches Speicherlayout einer solchen von Neumann-Architektur. Im niedrigstwertigen Teil des Speichers liegt das Programm in Assembler-Sprache, das Textsegment. Danach kommt das Datensegment mit allen globalen Konstanten. Dem schließt sich ein Bereich mit dynamischen Daten an. Dieser kann wachsen, je nachdem, wie viel Speicher das Programm benötigt. Im oberen Teil des Speichers befinden sich Adressen, deren Datum direkt an die Ein- und Ausgabeeinheiten angeschlossen ist. Das bedeutet, die Inhalte dieser Speicherelemente werden entweder extern gelesen oder können von externen Geräten beschrieben werden. Dieses Vorgehen heißt speicherorientierte Ein-/Ausgabeverwaltung (engl. „memory mapped I/O“). Ein solcher Rechner ist streng genommen keine von Neumann-Maschine, sah doch John von Neumann eigene Befehle für die Ein- und Ausgabe vor. Im Zuge der Rechnerentwicklung hat

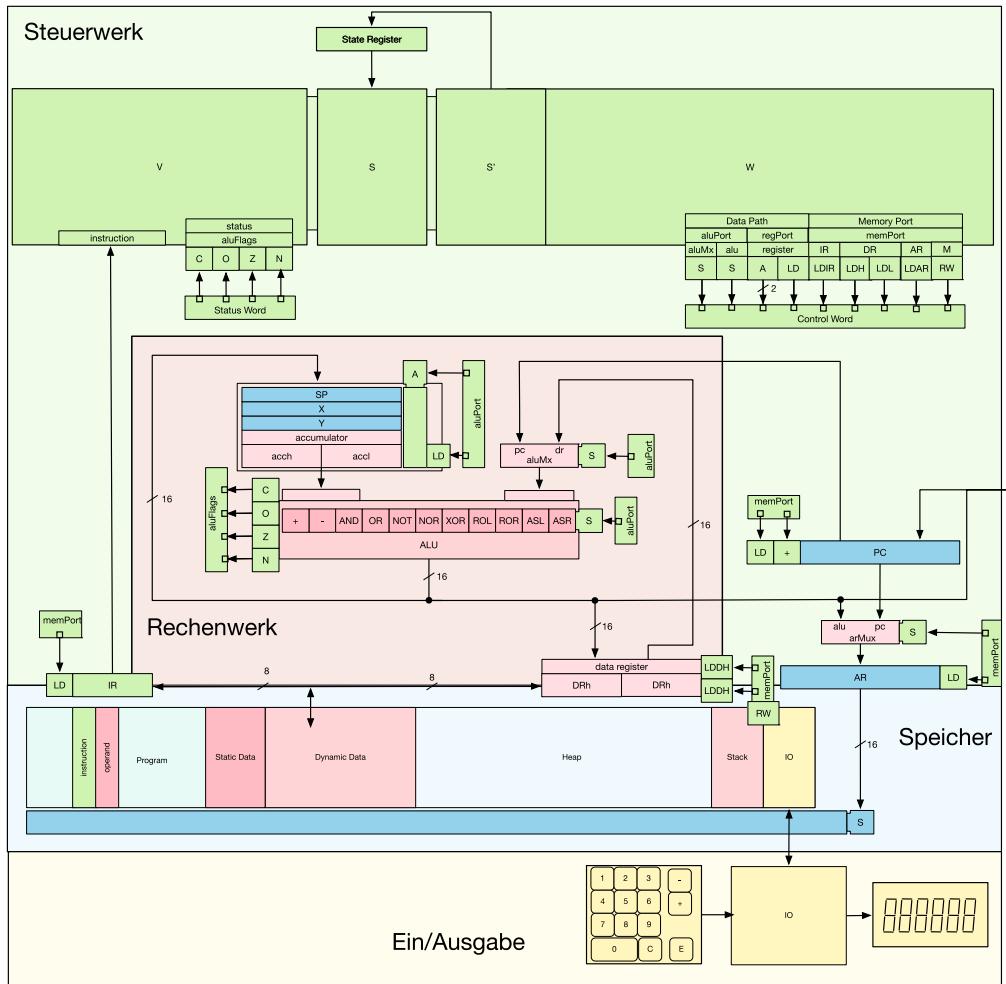


Abb. 6.31 Die Princeton-Architektur mit gemeinsamem Speicher für Programm und Daten

sich jedoch gezeigt, dass diese Aufgabe von Datentransportbefehlen genauso gut erledigt werden kann, da die Ein- und Ausgabe über den Adressraum des Prozessors abgewickelt werden kann. Dem I/O-Speicherbereich folgt der mit dem Stapelzeiger realisierte Stapel. Dieser ist ein Relikt aus der Zeit der Stapelmaschinen. Es hat sich gezeigt, dass Stapel in bestimmten Fällen dem wahlfreien Speicher überlegen sind. So sind beispielsweise Unterprogrammaufrufe mit einem Stapel wesentlich einfacher zu realisieren als ohne. In der klassischen Princeton-Architektur wird der Stapelzeiger über zwei spezielle und gesonderte Befehle angesprochen: *push* und *pop*. Moderne Befehlssätze besitzen diese Befehle nicht mehr und simulieren den Stapel nur noch durch Lesen und Schreiben des Stapelzeigers.

Das Speicherlayout kann je nach Rechner variieren. So kann z. B. der I/O-Bereich auch vor oder nach dem Programm liegen. Dies ist jedem Rechnerarchitekten freigestellt. Wichtig ist aber, dass er sich nicht zwischen dem dynamischen, dem sich in seiner Größe während des Programmlaufs verändernden Segment und dem Stapel befindet. Dieser Bereich, Heap (engl. für Haufen oder Halde) genannt, wird vom Programm variabel genutzt.

Das wesentliche Prinzip einer Registermaschine mit wahlfreiem Speicher, ob nun in Harvard- oder in Princeton-Architektur ist, dass Befehle zunächst einmal ins Steuerwerk geladen werden müssen. Dazu ist eine Anweisung aus dem Speicher zu holen und in einem dafür vorgesehenen Register zu speichern. Dieses heißt Befehlsregister oder Instruktionsregister. Genau wie der Akkumulator sorgt es für eine Entkopplung zwischen schnellem Rechner und langsamem Speicher. Um einen Befehl in das Befehlsregister zu laden, wird zunächst der Inhalt des Befehlszählers an die Adressdecodierlogik des Speichers angelegt. Der Speicher stellt dann das in der gewählten Speicherzelle gespeicherte Datum zur Verfügung, und dieses wird anschließend in das Befehlsregister geschrieben. Der Befehlszähler wird nun hochgezählt und der nächste Befehl geholt.

6.3.4.4 Universelle Registermaschine und ihre Befehlsformate

Durch die Einführung weiterer Register stiegen die Zahl der Operandenadressierungen und damit die Anzahl möglicher Befehle. Insbesondere die Verwaltung des Adressraumes wurde mit größeren Speicherräumen immer komplexer. Aus einer einfachen Rechenoperation einer Stapelmaschine wurde im Laufe der Zeit ein umfangreicher Befehl mit unterschiedlichen Adressierungsarten. Dadurch stieg der Aufwand zur Implementierung des Steuerwerks. Speicher, insbesondere dynamische Halbleiterspeicher, wurden immer leistungsfähiger und größer. Da die Zugriffsprotokolle oder Adressierungsarten komplexer wurden, vervielfachte sich die Anzahl unterschiedlicher Befehle in einem Befehlssatz. Hinzu kam, dass ein großer Adressraum zu immer längeren Zugriffszeiten führte und daher schnelle Puffer notwendig wurden, um die Rechengeschwindigkeit von der Speicherzugriffszeit zu entkoppeln – zum einen zum Programmspeicher, zum anderen aber auch zu den Daten. Bei einer Princeton-Architektur entsteht dabei ein Konflikt: Welcher Zwischenspeicher darf wann auf den gemeinsamen Speicher zugreifen?

Durch die hohe Skalierung der Halbleiterprozesse konnten immer mehr Transistoren auf dem Chip der Rechenma-

schine integriert werden. Dadurch waren mehr Register in der Mikroarchitektur möglich. Eine universelle Registermaschine hat daher eine große frei adressierbare Registerdatei („register file“). Befehle können Register in dieser Datei direkt adressieren, und das Prinzip des 3-Adress-Codes lässt sich so mit kurzen Adressen in den Befehlen sofort umsetzen. Ein weiteres Merkmal einer solchen universellen Registermaschine ist, dass nicht mehr alle Befehle auf den Speicher zugreifen. Der Datentransport wird nur noch mit zwei Befehlen vorgenommen, einem Befehl store zum Speichern und einem Befehl load zum Laden von Daten in die Register. Daher wird eine solche Architektur des Rechenwerks auch Load-Store-Architektur genannt. Fließbandverarbeitung im Zusammenspiel mit einer Load-Store-Architektur führt zu einem Zugriffskonflikt auf die Registerdatei und den Speicher, da gleichzeitig verschiedene Befehlsphasen unterschiedlicher Befehle aktiv sind. Beispielsweise soll ein Ergebnis einer Berechnung in ein Register geschrieben werden, während aber ein anderer Befehl dieses Register bereits ausliest. Besonders problematisch wird es, wenn zwei Befehle gleichzeitig in die Registerdatei schreiben. Dies kann beispielsweise passieren, wenn eine Sprungadresse gesichert werden soll. Diese Konflikte lassen sich durch die Verwendung eines Mehrtorspeichers als Registerdatei lösen. Zusammen mit einigen Konventionen beim Zugriff auf die Register und einem disziplinierten Programmierstil können Zugriffskonflikte gelöst werden. Anders ist dies bei Zugriffen auf den Speicher. Befindet sich beispielsweise ein Store- oder Load-Befehl in einer Phase des Hauptspeicherzugriffs und wird gleichzeitig ein Befehl aus dem Speicher geholt, kommt es automatisch zu einer Kollision. Derartige Kollisionen lassen sich nicht vermeiden, ohne die Fließbandverarbeitung anzuhalten. Verzichtet man jedoch auf die Vorteile der Princeton-Architektur und kehrt wieder zur Harvard-Architektur zurück, lässt sich dieses Problem umgehen. Dies hat allerdings eine schlechte Speicherauslastung zur Folge. Es ist also wünschenswert, die Harvard- mit der Princeton-Architektur zu kombinieren, um die Vorteile beider Stile nutzen zu können. Die Einführung von Caches zur Beschleunigung von Speicherzugriffen macht dies möglich: Für Befehle und für Daten werden zwei getrennte Caches verwendet. Beide werden an den gleichen Hauptspeicher angeschlossen, und es entsteht eine modifizierte Harvard-Architektur.

Lösen lässt sich das Problem der langen Zugriffszeiten auf den Speicher mithilfe schneller Zwischenspeicher. Diese sind für einen Programmierer versteckt und nicht sichtbar. Die Verwaltung erfolgt ausschließlich durch die Hardware und beeinflusst das Ausführungs- und damit das Programmiermodell nicht. Diese Caches sorgen in modernen Mikroarchitek-

turen dafür, dass die klare Trennung von von Neumann- und Harvard-Architektur verwischt.

Die konsequente Einführung eines regelmäßigen Befehlsatzes mit möglichst wenigen homogenen Befehlsworten und der schnelle gepufferte Zugriff mithilfe von Caches auf den Hauptspeicher ermöglichen die Implementierung einer effizienten Fließbandverarbeitung („pipelining“). Dabei werden die einzelnen Befehlssphasen zwar immer noch sequenziell ausgeführt, jedoch werden alle Phasen gleichzeitig bearbeitet. Somit befinden sich unterschiedliche Befehle simultan in verschiedenen Phasen in der Maschine. Einen solchen Rechner mit Fließbandausführung nennt man einen skalaren Computer, im Gegensatz zu einem Vektorrechner, bei dem ein Befehl gleichzeitig und parallel mehrere Rechenwerke ansteuert.

Allerdings wird mit der Trennung des Befehls- vom Datenfluss und der Einführung der Fließbandverarbeitung auch die Mikroarchitektur komplexer. Die Abb. 6.32 versucht dem Rechnung zu tragen, indem der Kontrollfluss waagerecht und der Datenfluss senkrecht angeordnet sind. In der Abbildung sind die einzelnen Stufen der Pipeline grau hinterlegte Balken. Die schwach ausfüllten Register sind, egal ob im Kontrollfluss oder im Datenfluss, für den Programmierer unsichtbar. Die in der Abbildung dargestellte Mikroarchitektur zeigt eine mögliche Implementierung des MIPS. Da diese keine Pipeline-Interlocks besitzt, ist die entsprechende Darstellung noch einigermaßen übersichtlich. In einem typischen MIPS-Befehl sind Adressen zum Ansprechen unterschiedlicher Register vorhanden. Je nachdem, ob es ein Quell- oder ein Zielregister ist, wird auf dieses zu verschiedenen Zeiten zugegriffen. Die Adresse muss also auch im Kontrollpfad durch die Pipeline gereicht werden. Bei Verzweigungen oder arithmetisch-logischen Befehlen sind ggf. noch einfache Berechnungen durchzuführen. So liegen im Kontrollpfad die Vorzeichenerweiterung für das Feld der Konstanten einer Anweisung und eine Multiplikation um 4. Diese wird mithilfe eines Schieberegisters oder Barrel-Shifters vorgenommen und erzeugt das korrekte Alignment von in einem Befehl angegebenen Sprungadressen. Dieses Vorgehen spart Bits im Befehlsformat, da die Adressierung der Worte im Speicher byteweise erfolgt, Befehle aber an Wortgrenzen stehen. Diese Werte werden durch die Pipeline geschoben, um dann in der Execute-Phase von der ALU mit einem Registerinhalt verknüpft zu werden. Dadurch kann die ALU zur Berechnung registerrelativer Adressen verwendet werden. Über diesen Pfad, vom Befehlscache über das Befehlsregister bis zur ALU kann auch der Wert für einen Schiebebefehl zur ALU durchgereicht werden.

Eine Besonderheit stellt noch die Ausnahmbehandlung dar. Wird z. B. eine Ausnahme durch die ALU ausgelöst, muss

6.3 · Mikroarchitektur

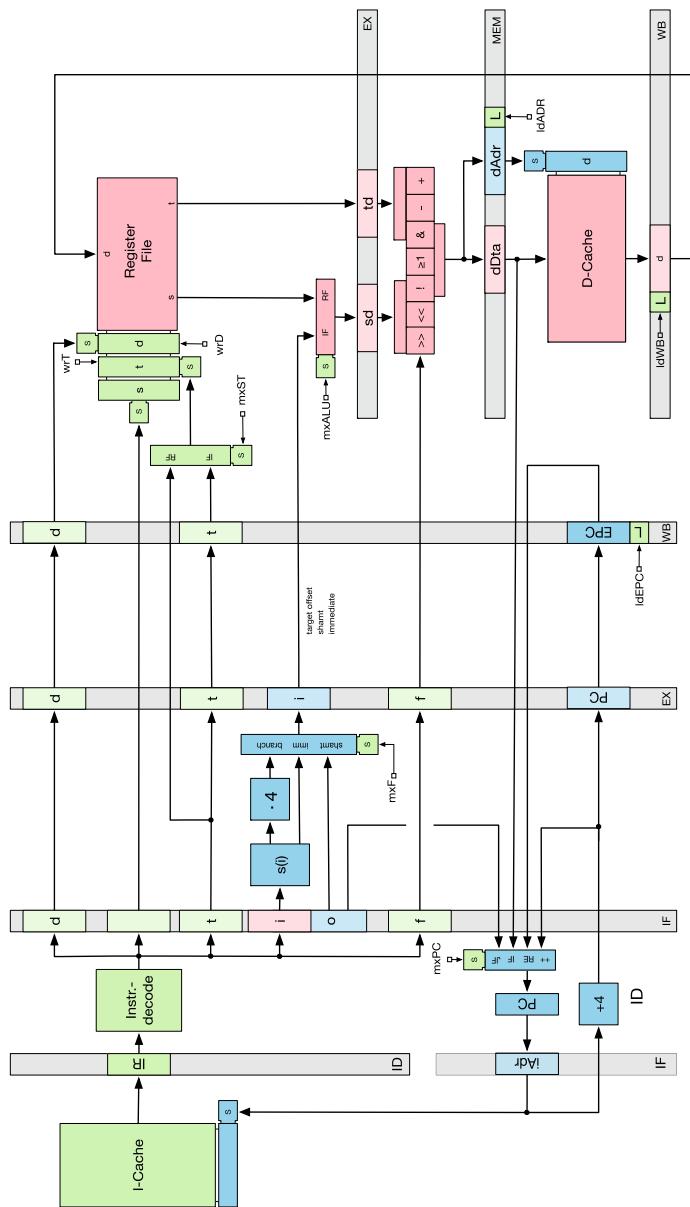


Abb. 6.32 Mikroarchitektur einer universellen Registermaschine am Beispiel des MIPS

der entsprechende Wert des Befehlszählers im EPC-Register gespeichert werden. Dazu ist dieser ebenfalls durch die Pipeline zu reichen. Der Anschluss des Hauptspeichers ist in dieser Darstellung der Mikroarchitektur nicht gezeigt. Dies ist auch nicht nötig, da bei korrekter Implementierung des Programm- oder Datencache diese ein Abbild der relevanten Speicherbe-

reiche sind. Sollte ein Wert nicht im Cache liegen, würde dies dem MIPS signalisiert werden, und bis zum Nachladen dieses Wertes würde die Pipeline anhalten. Die Caches sollten so dimensioniert sein, dass die Trefferrate etwa 90 % ist. In dem Fall kann die Pipeline einfach durchlaufen. Da der MIPS keine Interlock-Hardware besitzt, sind Pipeline-Stalls möglich. Dieses Verhalten muss vom Compiler durch Umsortieren, von Code oder durch das Einfügen von NOP-Befehlen kompensiert werden.

Durch den Wegfall umfangreicher Adressierungen bei den meisten Befehlen reduziert sich der Befehlssatz einer universellen Registermaschine. Daher nennt man diese Rechner auch RISC. RISC zeichnen sich durch universelle Register und einheitliche Befehslängen aus. Allerdings verschärfen derartige Maschinen durch den effizienten Zugriff auf die internen Register die Geschwindigkeitslücke zum Hauptspeicher enorm. Dies wird durch die Caches wieder ausgeglichen. Durch die Kombination der Prinzipien der Princeton- und der Harvard-Architektur kann der Programmfluss wieder vom Datenfluss getrennt werden. Dabei kann der Speicher nach wie vor effizient genutzt werden. Auch das Problem des von Neumann-Flaschenhalses wird reduziert, da die Caches Zugriffe auf den Hauptspeicher selbst verwalten und durch die Puffer diese seltener erfolgen.

?

Übungsaufgaben

Aufgabe 6.1 Über welche grundlegenden Befehlsformate verfügt der MIPS, und wofür werden diese genutzt?

Aufgabe 6.2 Was versteht man unter dem Begriff Pseudobefehl im Bezug auf den MIPS?

Aufgabe 6.3 Mithilfe eines Speicherelementes wie dem Stack auf Basis von LiFo können wir Zahlen einlesen und ausgeben. Um zu rechnen, müssen die Zahlen entsprechend des genutzten Algorithmus umgesetzt auf die Rechenmaschine gespeichert werden. Im Folgenden sind chronologisch ausgeführte Befehle auf einem Stack gelistet.

push 8 → push 5 → push 3 → add → push 4 → push 2 → sub → mul → add → push 20 → sub

- Zeichnen Sie zu jedem Befehl den Stack nach jeder Ausführung eines Befehls.
- Geben Sie den kompletten mathematischen Ausdruck wieder, der damit berechnet wurde.
- Geben Sie nun auf die gleiche Art wie in der vorherigen Teilaufgabe die Befehle an, um den Ausdruck $(2(8-2)) + (10 \cdot 3)$ mithilfe des Stacks zu berechnen.

- ## Weiterführende Literatur
-
- [Dav17] Andrew Waterman David A. Petterson. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN: 9780999249116.
 - [And14] Todd Austin Andrew S. Tanenbaum. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. Pearson Studium ein Imprint von Pearson Deutschland, Auflage: 6. Edition (3. März 2014). ISBN: 9783868942385.
 - [Dav13] John L. Hennessy David A. Petterson. *Computer Organization and Design MIPS Edition: The Hardware-Software Interface*. Morgan Kaufmann, Auflage: 5. Edition (10. Oktober 2013). ISBN: 9780124077263.
 - [Sta13b] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson Education Ltd., Auflage: 10. Edition (10. Oktober 2013). ISBN: 9781292096858.



Programmieren von Maschinen

Inhaltsverzeichnis

- 7.1 Programmieren in Hochsprache – 441
- 7.2 Programme in C – 443
- 7.3 Programme in Maschinensprache – 447
- 7.4 Assembler einer universellen Rechenmaschine – 454
- 7.5 C-Schablonen für eine universelle Registermaschine – 478
- 7.6 Fibonacci in Assembler – 498

*Software ist wie ein Gas, sie füllt jeden erdenklichen Speicher.
frei nach Nathan Myhrvold*

In diesem Kapitel werden die Rechner genutzt. Im Vordergrund stehen einfache Algorithmen und deren programmtechnische Realisierung. Leser sollten am Ende des Kapitels die grundlegenden Konzepte des Programmierens verstanden haben. Dabei liegt der Fokus auf dem Zusammenhang zwischen der Struktur des Rechners, dem Befehlssatz und dem Programm. Leser sollten Teile der Hardware eines Computers mit den grundlegenden Primitiven des Programmierens wie Verzweigungen, Schleifen und Unterprogrammaufrufen verbinden können, insbesondere da moderne, weiterführende Programmiermethoden letztendlich auf diesen Grundkonstruktionen aufbauen. Die Programmierung der Algorithmen erfolgt am Beispiel der MIPS-Architektur.

7

Zur technischen Informatik gehören nicht nur die Technologie, Schaltungstechnik und der logische Aufbau eines Rechners. Die Maschinen müssen auch programmiert werden. Insbesondere sollte ein Rechnerarchitekt ein tiefes Verständnis der maschinennahen Programmierung haben, da er den Befehlssatz entwerfen muss. Und zwar so, dass sowohl Hardware als auch Software effizient sind und reibungslos ineinander greifen. Naturgemäß kann dieses Kapitel nicht alle Aspekte des Programmierens abdecken. An dieser Stelle kommt es darauf an, die grundlegenden Konzepte der maschinennahen Programmierung zu beleuchten und zu verstehen, wie sich Primitive der Hochsprachen auf die Maschine bringen lassen. Das heißt, wir werden uns um die Prinzipien der Programmierung kümmern und Schablonen entwerfen, um später beliebige Anwendungen für Computer programmieren zu können. Mittlerweile hat sich die universelle Registermaschine sowohl im Desktopbereich als auch in der Welt der eingebetteten Rechner durchgesetzt. In diesem Kapitel werden am Beispiel des MIPS die grundlegenden Konzepte der Assembler-Programmierung behandelt und gezeigt, wie ein Compiler diese in Maschinencode umsetzen kann. Beispielhaft sollen Primitive der Hochsprache C in Assembler-Code übersetzt werden. Dazu soll ein einfaches C-Programm als Beispiel dienen. Dieses besteht nur aus wenigen Zeilen, deckt aber alle grundlegenden Aspekte ab. Insbesondere ist C Grundlage der oft verwendeten Programmiersprachen C++, C# oder auch Java. Sie eignet sich daher gut als Ausgangspunkt zum Verständnis der Assembler-Programmierung. Die Sprache selbst entstand im Rahmen von Forschungsarbeiten zu Betriebssystemen. Bedingt durch ihren Ursprung ist sie gut zur systemnahen Programmierung geeignet und wird daher oft in eingebetteten Systemen verwendet. Im Vergleich zu ihren Nachfolgern wie C++ und Java ist sie

auf der einen Seite einfach, da nicht alle bekannten Programmierparadigmen abgedeckt sind, andererseits aber komplex, weil sie viele Freiheiten und syntaktische Tricks zur hardwarenahen Programmierung erlaubt. Dieser, aus Sicht des praktischen Informatikers, Nachteil prädestiniert sie allerdings, um zu zeigen, wie Hochsprache auf die Ebene der Maschine abgebildet wird. Und darum geht es uns am Ende in diesem Kapitel.

7.1 Programmieren in Hochsprache

Sprachen wie Java und C++ werden von der Hardware nicht direkt verstanden. Sie müssen maschinell vom Computer in für die Maschine verständliche Bitfolgen umgewandelt werden. Dies geschieht in mehreren Schritten. Zunächst einmal muss der Quellcode des Programms gelesen werden. Meist ist dies eine nach bestimmten Vorschriften erstellte Textdatei. Diese Regeln, auch Syntax genannt, werden von der verwendeten Programmiersprache definiert. Die Textdatei wird dann mithilfe eines Programms, dem Compiler, in eine weitere Datei übersetzt. In dieser steht der vom Übersetzer erzeugte Assembler-Code. Um diesen zu erhalten, wird das Programm in einzelne Blöcke unterteilt. Jeder dieser Schritte entspricht einem oder mehreren Befehlen und somit einem Befehlszyklus der Zielmaschine. Insbesondere zeichnen sich die Blöcke dahingehend aus, dass sie keine Verzweigungen oder Schleifen mehr enthalten. Jeder Einzelne dieser Blöcke wird dann in elementare Befehlsfolgen zerlegt. Diese Folge von Befehlen hat die Eigenschaft, dass jede dieser Anweisungen einem Befehlszyklus der Maschine entspricht. Das resultierende Assembler-Programm ist daher maschinenspezifisch, d. h., die Instruktionen spiegeln stark die Rechnerarchitektur wider. So ist leicht zu erkennen, ob der Code für eine Akkumulator- oder universelle Registermaschine übersetzt wurde. Auch die Schreibkonventionen des Assembler-Codes sind oft von der verwendeten Maschine abhängig. Dieser Assembler-Code ist aus lesbaren Befehlsmnemoniken zusammengesetzt und kann gut vom Menschen gelesen werden. Die meisten Compiler erlauben es dem Programmierer, den maschinenspezifischen Code zu inspirieren. So kann man sich beispielsweise mit den Eigenarten einer Maschine vertraut machen oder überprüfen, ob der Übersetzer den Quellcode korrekt übersetzt.

Der Computer kann den Assembler-Code nicht direkt verstehen, dazu muss dieser erst in den Maschinen- oder Objektcode umgewandelt werden. Eine Datei mit Maschinencode enthält Zeilen aus einer für die Maschine verständlichen Folge von Nullen und Einsen. Oft werden Objektdateien auch in hexadezimaler Schreibweise dargestellt. Dies erlaubt das Pro-

Compiler

Assembler

grammieren oder besser Patchen (engl. für flicken oder ausbessern) sofort auf der Maschine, selbst wenn diese nur aus einer Platine besteht. Derartige Hex-Dumps (engl. für Halde, also Hex-Müll oder -Abfall) sind schwer durchzuarbeiten, stellen aber oft den einzigen Weg dar, den Speicherinhalt eines Computers direkt zu lesen. Das Programm, das den Assembler-Code in Maschinencode umwandelt, wird Assembler genannt. Auf den ersten Blick ist das verwirrend, heißt häufig der Assembler-Code auch Assembler. Die Bedeutung ergibt sich schließlich aus dem Kontext. Der Assembler übersetzt also das Assembler-Programm in die Bitfolgen und Objektbefehle, die von der Maschine direkt verstanden werden können.

Linker

Der Linker ist ein Hilfsprogramm des Assemblers. In der Regel schreiben Assembler-Programmierer die Programme nicht mit absoluten Adressen, sondern definieren statt dieser Zahlen symbolische Marken. Diese erlauben es, die endgültigen Speicherstellen der Befehle erst zu einem späteren Zeitpunkt festzulegen. Dadurch werden Programme portierbar und sind leichter lesbar. Auch Compiler folgen dieser Regel. Da ein Programmierer häufig gar nicht weiß, in welchem Kontext sein Code eingesetzt wird, würde es viel Arbeit für Anwender einer Software bedeuten, wenn der Softwareentwickler die Adressen des Programms bereits beim Schreiben des Quelltextes festlegen würde. Da die Adressen des Assembler-Codes relativ zu einer symbolisch definierten Marke angelegt werden, kann das Programm durch den Linker im Speicher beliebig verschoben werden. Der Linker kann nun diese Marken in den Assembler-Programmen eindeutig mit einer Adresse verknüpfen und alle Adressen zwischen ihnen berechnen. Dies hört sich nicht nach einem aufwendigen Schritt an. Doch Vorsicht, der Linker sorgt dafür, dass Code wiederverwendet werden kann. So ist es möglich, den Anwendungsprogrammierern umfangreiche Bibliotheken zur Verfügung zu stellen. In diesen muss der Code der Funktionen nicht im Quellcode vorliegen, sondern kann schon fertig übersetzt als speicherverschiebbarer Maschinencode abgelegt werden. Dies hat im Wesentlichen zwei Vorteile: Zum einen ist nicht immer die komplette Software zu kompilieren, wenn ein einzelnes Modul geändert wird, zum anderen kann ein Unternehmen sein Know-how bis zu einem gewissen Grad schützen, indem es Programm als Objektcode bereitstellt, der für Menschen nur schwer zu lesen ist. Die Festlegung der Adressen mehrerer Programme untereinander übernimmt der Linker. Dieser bindet alle Maschinensprogramme zusammen. In der Regel ist auch dieser Code noch im Speicher verschiebbar. Diese Verschiebbarkeit wird durch die relative Adressierung direkt in der Hardware vorgenommen und von der Speicherverwaltung genutzt. Diese Technik erlaubt es dem Betriebssystem, Programme beliebig im Speicher zu ver-

schieben. Lediglich in kleinen eingebetteten Systemen wird der Linker das Programm absolut platzieren und dafür bereits alle Adressen festlegen.

7.2 Programme in C

Die typischen Schablonen zur Übersetzung der Hochsprache C in Assembler-Sprache werden an einem kleinen und einfachen Programm diskutiert. Die Wahl fiel auf die Berechnung einer beliebigen Fibonacci-Zahl. Das Problem ist aus mathematischer Sicht interessant und einfach zu programmieren. Wichtig an dem Beispiel ist, dass die eigentliche Berechnung in einem Unterprogramm stattfindet, das von einem Hauptprogramm aufgerufen wird. Da das Programm zur Berechnung der Fibonacci-Zahlen sowohl iterativ als auch rekursiv geschrieben werden kann, eignet es sich besonders gut, die unterschiedlichen Methoden zur Implementierung von Unterprogrammen zu diskutieren. Im Folgenden werden zwei Verfahren in C programmiert, um die n -te Fibonacci-Zahl zu bestimmen. Die Fibonacci-Zahlen lassen sich am besten rekursiv definieren. Daher wird dieses Verfahren sowohl zur Definition als auch als erstes Unterprogramm genutzt. Der anschließend behandelte iterative Algorithmus dagegen benutzt einfachere Programmkonstrukte, die aber ebenfalls beleuchtet werden sollen. Aus diesem Grund wird das iterative Verfahren zuerst in Assembler-Code übersetzt. Im Laufe der Diskussion werden wir den Code des iterativen Programms kontinuierlich an neue Problemstellungen anpassen, um so am Ende alle Primitive der Sprache C untersucht zu haben.

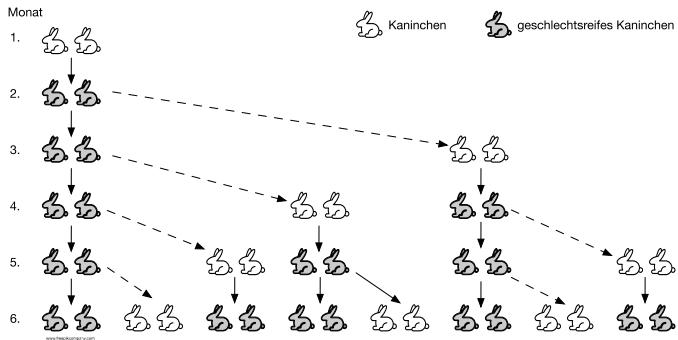
Leonardo da Pisa (1170–1240), genannt Fibonacci, war ein mittelalterlicher Rechenmeister. Auf Reisen nach Konstantinopel, Syrien und Nordafrika lernte er die damals in Europa noch wenig verbreitete Rechentechnik der Araber kennen. Rechneten Europäer zu dieser Zeit im lateinisch-römischen System, bevorzugten die Araber das uns heute geläufige dezimale Stellenwertsystem, das wahrscheinlich aus Indien stammt. Im Jahr 1202 veröffentlichte Fibonacci das Rechenbuch *Liber abaci*, das Buch der Rechenkunst. In diesem Werk formuliert er in Kapitel VII ein Problem, das sich bemerkenswert häufig in der Natur stellt und das viele Bezüge zu Kunst und Kultur aufweist. Dieses war schon in der Antike bekannt, erlangte aber im Mittelalter erst wieder durch Fibonacci Aufmerksamkeit im Abendland.

Angenommen, im ersten Monat leben in einem Hof oder begrenzten Gebiet 1 weibliches und 1 männliches Kaninchen¹.

Fibonacci

Fibonacci-Folge

¹ Das ist wichtig, könnten Kaninchen sonst ab- oder zuwandern.



■ Abb. 7.1 Fibonacci Kaninchen

Der erste Monat im Leben dieser 2 Hasen verläuft ereignislos, benötigen Fibonacci-Tiere exakt 4 Wochen, um geschlechtsreif zu werden. Nach einem weiteren Monat Tragezeit werden dann immer 1 weibliches und 1 männliches Kaninchen geboren (Abb. 7.1). Die Frage, die sich Fibonacci nun stellte, war, wie viele Hasen leben nach einem Jahr in dem Hof, wenn keine Tiere sterben und ihre Reproduktionsrate konstant ist:

Monat 0: 1 Paar

Monat 1: 1 Paar

Monat 2: 2 Paare

Das 2. Paar ist nach dem nächsten Monat noch nicht geschlechtsreif, daher kommen nur 2 weitere Tiere zu den Kaninchen im Hof dazu.

Monat 3: 3 Paare

Im 4. Monat bekommen nun 2 Paare jeweils 1 Paar Junge, 1 Paar ist noch nicht geschlechtsreif.

Monat 4: 5 Paare

Im folgenden Monat bekommen 3 Paare Junge und 2 Paare werden geschlechtsreif. Im 5. Monat leben also

Monat 5: 8 Paare

im Hof. Führt man die Diskussion fort, leben nach 12 Monaten 89 Kaninchenpaare im Hof. Die Kaninchenpopulation folgt der Bildungsvorschrift

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n - 1) + f(n - 2) & \text{sonst} \end{cases} \quad (7.1)$$

wenn die Fibonacci-Folge noch um eine führende 0 erweitert wird, also irgendein Züchter oder Schöpfer zu einem bestimmten Zeitpunkt die Kaninchen in den Hof setzt.

Die Funktion in Listing 7.1 bildet diese Berechnungsvorschrift direkt in der Hochsprache C ab. Zunächst wird die Funktion selbst beschrieben. Die erste Zeichenfolge int spezifiziert, dass der Rückgabewert der Funktion eine ganze Zahl sein soll (Typ Integer), die Zeichenfolge rFib definiert den Namen der Funktion, der die ganze Zahl n als Parameter übergeben wird. In Abhängigkeit von dem in n gespeicherten Wert wird dann die entsprechende Fibonacci-Zahl berechnet. Dazu muss eine Auswahl getroffen werden. In C können ein häufiger Vergleich mit Konstanten und eine darauf aufbauende Verzweigung mit der Primitive „switch-case“ vorgenommen werden. Dies entspricht dem Vorgehen in der Sprache Java oder fast jeder anderen Hochsprache. Das Switch-case kann beliebig viele Auswahlzweige mithilfe von Konstanten definieren. Die Variable in der Switch-Anweisung wird dann mit jeder dieser Konstanten verglichen und der entsprechende Zweig anschließend ausgeführt. In unserem Fall gibt die Funktion eine 0 zurück, wenn der Übergabeparameter ebenfalls eine 0 ist, und eine 1 wenn der Parameter eine 1 ist. In allen anderen Fällen wird die Funktion rFib erneut aufgerufen, und zwar mit den gemäß der Bildungsvorschrift reduzierten Parametern. Eine solche rekursive Lösung des Problems ist aus Sicht des Programmierers einfach und elegant. An den vom Compiler erzeugten Assembler-Code und die Hardware der ausführenden Maschine stellt diese Art der Programmierung aber besondere Herausforderungen.

Rekursiv

```

1 || int rFib(int n) {
2 /* Rekursive Bestimmung der n-ten Fibonacci-Zahl */
3     switch(n) {
4
5         case 0: return 0;
6         case 1: return 1;
7         default: return rFib(n-1)+rFib(n-2);
8
9     } /* switch(n) */
10 } /* rFib(int n) */

```

Listing 7.1: Fibonacci-Zahl

Da die Verwaltung einer rekursiven Berechnung ein wenig Aufwand erfordert, soll zunächst eine einfachere Variante des Fibonacci-Algorithmus (Listing 7.2) betrachtet werden:

Iterativ

```

1  int iFib(int n) {
2  /* Iterative Bestimmung der n-ten Fibonacci-Zahl */
3  int i = 2;
4  int F, n_0 = 0;
5  int n_1 = 1;
6
7  do {
8      if (n<1) F=0;
9      else {
10         F = n_0 + n_1;
11         n_0 = n_1;
12         n_1 = F;
13     } /* if */
14 } while(i++ < n);
15 return F;
16
17 } /* iFib */

```

Listing 7.2: Iterative Bestimmung der n-ten Fibonacci-Zahl

7

Diese iFib genannte Funktion berechnet die n -te Zahl der Fibonacci-Folge iterativ. Einzelne ausführbare Programmteile werden in C mit geschweiften Klammern gekapselt. Nach jeder Kontrollanweisung oder nach der Spezifikation einer Funktion werden die zusammenhängenden Stücke Code durch die geschweiften Klammern zu Blöcken gebündelt. Ein algorithmischer Schritt oder vielmehr eine Zeile Programmcode wird in C mit einem Semikolon begrenzt.

Dieser Funktion wird wieder eine ganze Zahl übergeben. Nach der Initialisierung einiger Hilfsvariablen werden alle n Nummern der Fibonacci-Folge mithilfe einer Schleife berechnet. Diese „Do-while-Schleife“ genannte Programmkonstruktion wird so lange ausgeführt, bis die Index- oder Zählvariable i den Wert des Übergabeparameters erreicht. Ist $n < 1$ oder 0, wird die Fibonacci-Variable F auf 0 gesetzt, andernfalls wird ein Wert für F berechnet. Durch geschickte Initialisierung der Variablen wird für $n = 2$ das korrekte Ergebnis bestimmt. Für größere Werte von n muss eine schrittweise Vertauschung der Operanden vorgenommen werden. Der $n - 2$ -Wert (x) wird zum $n - 1$ -Wert (y) und dieser wiederum zum n -ten Wert (F). Am Ende wird das Ergebnis F zurückgegeben.

While-
Array

Dieses Verfahren kann unter Zuhilfenahme der Datenstruktur Liste (engl. „array“) eleganter programmiert werden. Dazu wird eine Liste mit 3 Einträgen definiert, um die jeweils letzten 3 berechneten Zahlen zu speichern. Auf die Inhalte der einfachen Liste kann dann indiziert zugegriffen werden:

7.3 · Programme in Maschinensprache

```

1 int iFib(int n) {
2 /* Iterative Bestimmung der n-ten Fibunacchi-Zahl */
3 int i=0;
4 int f[3];
5
6 if (n<1) return 0;
7 if (n<2) return 1;
8
9 f[i++] = 0;
10 f[i] = 1;
11
12 while(i < n) {
13     i++;
14     f[2] = f[0] + f[1];
15     f[0] = f[1];
16     f[1] = f[2];
17 } /* while(I < n) */
18 return f[2];
19
20} /* iFib */

```

Listing 7.3: Iterative Bestimmung der n-ten Fibunacchi-Zahl mit While-Array

Natürlich lässt sich der Algorithmus auch mit einer Do-Schleife **Do-Array** formulieren:

```

1 int iFib(int n) {
2 /* Iterative Bestimmung der n-ten Fibunacchi-Zahl */
3 int i=0;
4 int f[3];
5
6 if (n<1) return 0;
7 if (n<2) return 1;
8
9 f[i++] = 0;
10 f[i++] = 1;
11
12 do {
13     f[2] = f[0] + f[1];
14     f[1] = f[2];
15
16 } while(i++ < n);
17
18 return f[2];
19} /* iFib */

```

Listing 7.4: Iterative Bestimmung der n-ten Fibonacchi-Zahl mit Do-Array

7.3 Programme in Maschinensprache

Um die Programmierung eines Computers grundsätzlich zu verstehen, soll die Maschine direkt programmiert werden. Da der Rechner nur binäre Bitfelder interpretiert, diese aber für den Menschen schwer zu lesen sind, wird im Folgenden Assembler-Code geschrieben. Im Assembler gibt es für jeden Be-

fehl, dessen Bitfeld die Maschine ausführen kann, eine für den Programmierer leicht lesbare Mnemonik. Zusammen mit einigen Konventionen zur Beschreibung unterschiedlicher Adressierungsarten lassen sich in Assembler komplexe Algorithmen darstellen und schreiben. Jede Zeile eines solchen Programms entspricht dann exakt einem Befehlszyklus der ausführenden Maschine. Assembler-Programme sind abhängig von der verwendeten Architektur des Rechners. Auch die Konventionen in der Schreibweise des Assembler-Codes unterscheiden sich oft. Daher sind Assembler-Programme maschinenabhängig. Ein Programm von einer Maschine auf einen anderen Computer zu portieren erfordert nicht unerheblichen Aufwand. Zwar muss die Programmlogik selbst nicht geändert werden, da aber unterschiedliche Architekturen und daher auch ihre Assembler die verschiedenen Adressierungsarten verschieden bezeichnen, ist bei der Portierung eines Programms sorgfältig vorzugehen. Hinzu kommt, dass ein bestimmter Modus, der auf der einen Maschine häufig genutzt wird, auf dem Rechner gar nicht vorhanden ist. Das ist auch der Grund, warum Software fast ausschließlich in Hochsprachen wie C oder Java geschrieben wird. Die Portierung auf eine andere Maschine kann dann der Compiler übernehmen. Um besser zu verstehen, welchen Einfluss die Architektur eines Rechners auf die Ausführung eines Programms hat, soll das Problem der Fibonacci-Folge in seiner Grundfunktion und seiner Übersetzung für unterschiedliche Architekturen diskutiert werden.

Stapelmaschine

Zuerst wird die Ausführung der Berechnung einer Fibonacci-Zahl auf einer Stapelmaschine diskutiert. Die beiden obersten Einträge des Stacks werden von der arithmetisch-logischen Einheit verknüpft. Ergebnisse können darüber hinaus mit den zwei Datentransferbefehlen push und pop in einen kleinen wahlfreien Hauptspeicher geschrieben werden. Speicherzellen sollen einfach und direkt adressiert werden. Adressen werden in der verwendeten Notation mit einem Dollarzeichen (\$) gekennzeichnet.

Die folgende Abb. 7.2 gliedert sich in drei Teile. Ganz links ist die Mikroarchitektur des Datenpfads oder Rechenwerks skizziert. Rechts davon ist die aufgerollte Programmausführung gezeigt. Im Bild sehen wir den Inhalt des Stacks und die Veränderungen im Speicher. Das Programm läuft von oben nach unten ab.

► Beispiel 7.3.1 – Fibonacci-Programm mit einer Stapelmaschine

Angenommen, das Programm selbst steht von der Adresse 0x00 beginnend im Hauptspeicher und kann linear von der Maschine abgearbeitet werden. Zunächst wird der Stack mit einer 0 und einer 1 initialisiert. Diese Zahlen stehen an der Adresse 0xA0 und 0xA1 im Speicher. Mit dem Befehl push *adresse* kann ein Datum

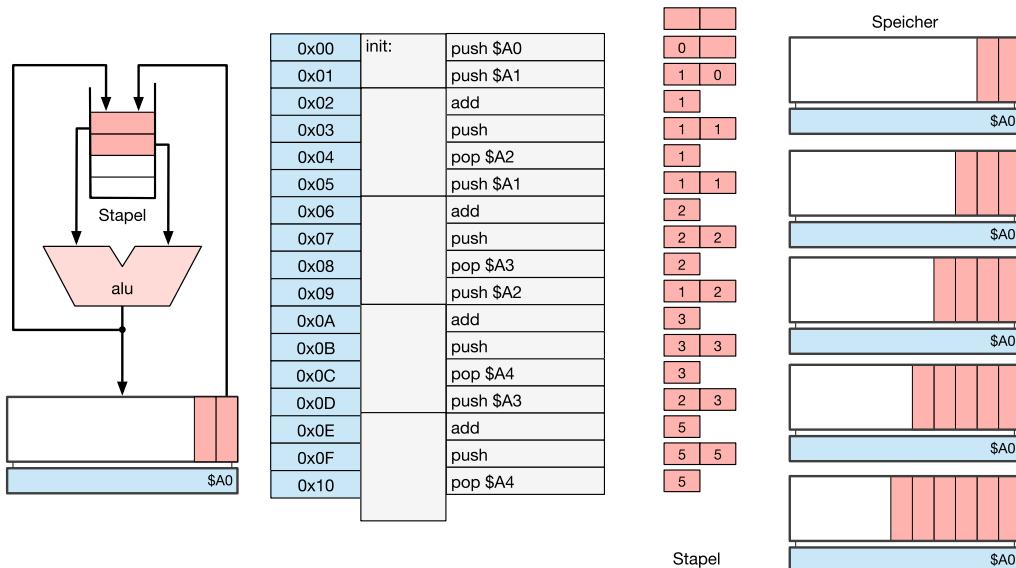


Abb. 7.2 Fibonacci-Folge auf einer Stapelmaschine

aus dem Hauptspeicher auf den Stapel gelegt werden. Nach der Initialisierung des Stacks werden die 0 und die 1 addiert. Das Ergebnis wird dann auf dem Stack dupliziert und anschließend in den Speicher geschrieben. Am Ende wird der Inhalt der vorhergehenden Speicherstelle auf den Stack gelegt und wieder addiert, kopiert und an der nächsten Stelle gespeichert. Es fällt auf, dass sich der wahlfreie Speicher selbst wie ein Stack verhält. Eigentlich könnte man als zweiten auch einen Stack verwenden. Um aber die Fibonacci-Folge zu berechnen, muss in jedem Schritt auf eine Vorgängerstelle zugegriffen werden. Dieses Verhalten ist mit einem Speicher mit wahlfreiem Zugriff einfacher zu realisieren als mit einem Stack. Alternativ könnte unsere Stackmaschine auch einen Befehl zum Vertauschen von Stackeinträgen haben. Dazu müsste nur der vorletzte Eintrag vom Stack genommen werden und dann wieder auf den Stack gelegt werden. In dem Fall könnte auf den Speicher mit wahlfreiem Zugriff verzichtet und stattdessen ein zweiter Stack eingesetzt werden. Unter der Annahme, dass pushA immer etwas auf den Stack der ALU legt und popA ein Datum vom Stack der ALU nimmt und auf dem zweiten Stack speichert, berechnet das folgende Programm die Fibonacci-Folge bis zur dritten Zahl. Der Befehl shft tauscht zwei Einträge auf dem Stack A. pushA 0 und pushB 1 initialisieren den jeweiligen Stack, popA holt einen Wert vom Stack A und schreibt es in Stack B, popB liest Daten vom zweiten Stack und legt diese auf dem ersten Stack ab (Abb. 7.2).

Die Ausführung des Programms mit den entsprechenden Einträgen der zwei Stacks sei dem Leser zur Übung überlassen. ◀

Akkumulatormaschine

Als Nächstes wird das Fibonacci-Programm auf einer 2-Adress-Maschine mit wahlfreiem Speicher ausgeführt.

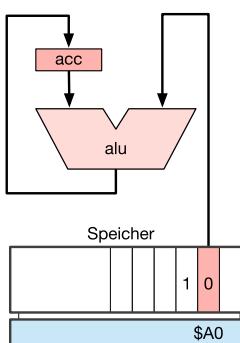
► Beispiel 7.3.2 – Fibonacci-Programm mit einer Akkumulatormaschine

Das Programm zur Berechnung der Fibonacci-Folge kann auf einer Akkumulatormaschine einfacher gestaltet werden. Die Abb. 7.3 zeigt einen solchen Rechner. Das Ergebnis einer Berechnung überschreibt den Inhalt des Registers *acc* direkt. Zum Datentransport stehen zwei Befehle zur Verfügung: Zum einen kann mit *lda \$Adresse* („load accumulator“) der Inhalt der Speicherstelle *Adresse* in den Akkumulator geladen werden, zum anderen wird das Datum des Akkumulators mit *sta \$Adresse* („store accumulator“) an der Stelle *Adresse* gespeichert. Wird der Additionsbefehl noch um einen Operanden erweitert, kann eine Hauptspeicherstelle direkt auf den Akkumulator addiert und dann wieder im Akkumulator abgelegt werden. ▲

7

Befehle für Schleifen

Es fällt bei der Programmierung der Akkumulatormaschine auf, dass sich ein Teil des Programms immer wiederholt. Die Folge der Befehle ist gleich, nur der Zugriff auf die Speicheradressen ändert sich. Es ist daher sinnvoll, Schleifen einzuführen. Diese lassen sich in Assembler mit bedingten Sprüngen programmieren. Die Bedingung ist notwendig, da die Anzahl der Ausführungen der Schleife begrenzt sein soll. Die zwei Befehle *bne* (engl. „branch not equal“) und *beq* (engl. „branch equal“) verzweigen immer dann, wenn die Zero-Flagge (*Z*) des Statuswortes durch die arithmetisch-logische Einheit gesetzt wird. Viele Maschinen nutzen diese Flagge, wenn das Ergebnis einer arithmetischen Operation null ist. Dies kann im Code durch eine Subtraktion zweier Variablen erreicht werden. Um die Programme lesbarer zu machen, besitzen viele Befehlssätze einen oder mehrere Vergleichsbefehle *cmp* (engl. „compare“). Da die Schleifendurchläufe gezählt werden, insbesondere wenn linear auf einen Speicher zugegriffen wird, kann der



0x00	init:	lda \$A0
0x01		add \$A1
0x02		sta \$A2
0x03		add \$A1
0x04		sta \$A3
0x05		add \$A2
0x06		sta \$A4
0x07		add \$A3
0x08		sta \$A5

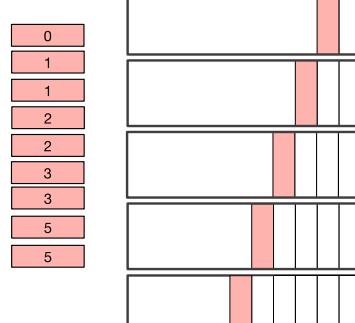


Abb. 7.3 Berechnung der Fibonacci-Folge mit einer Akkumulatormaschine

Zugriff auf solche Datenfelder (engl. „array“) effizient mit der indizierten Adressierung implementiert werden. Diese nutzt in einer Registermaschine eins der Indexregister. Für die Indexregister sind häufig Zählbefehle im Befehlssatz spezifiziert (engl. „increment“ und „decrement“, *inx*, *iny* und *dex*, *dey*). Wird direkt auf dem Indexregister gezählt, kann im Anschluss mit einem *cmp*-Befehl die Prüfung der Schleifenbedingung einfach programmiert werden.

Um Programme im Speicher beliebig verschieben zu können, lassen sich im Assembler Marken (engl. „label“) definieren. Ein Label wird immer durch eine Zeichenkette, bei der das letzte Zeichen ein Doppelpunkt ist, beschrieben. Der Linker ersetzt später diese Zeichenkette durch eine entsprechende Adresse und ordnet die Sprünge auf diese Zeichenkette automatisch den Adressen zu. Außerdem berechnet er alle Adressen zwischen den Marken. **Marken und Sprünge**

► Beispiel 7.3.3 – Fibonacci-Programm mit der Registermaschine

Das Fibonacci-Programm mit Schleifen ist auf der rechten Seite von Abb. 7.4 gezeigt. Der erste Befehl des Programms wird wieder an der Adresse 0x00 in den Speicher gelegt. Alle weiteren Befehle folgen direkt auf diese Adresse. Mit dem Label *fib:* lässt sich dieser Startpunkt symbolisch markieren. Unter der Annahme, 8 Fibonacci-Zahlen zu berechnen, werden die beiden Indexregister mit einer 7 durch den Befehl lade X-Register (*ld.x*) initialisiert. Der Akkumulator wird zu Beginn mit *ld.acc* auf 0 gesetzt. Um die Register zu initialisieren, wird die Immediate-Adressierung genutzt. Die führende 0 der Fibonacci-Zahlen wird nun in den Speicher geschrieben, und zwar an die höchste Adresse eines Feldes, dessen Basis an der Adresse 0x0F liegt. Dies geschieht mit dem Befehl *st.acc* („store accumulator“) unter Anwendung der indizierten Adressierung.

Nun wird das Indexregister um 1 dekrementiert, also um 1 heruntergezählt, und der Akkumulator inkrementiert und hochgezählt.

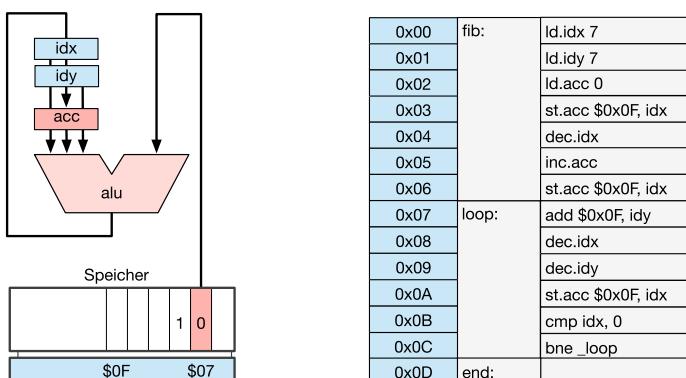


Abb. 7.4 Programm zur Berechnung der Fibonacci-Folge mit Schleife

Im Akkumulator steht jetzt eine 1 und im Indexregister *idx* eine 6. Damit kann die 1 aus dem Akkumulator gleich wieder indiziert im Array gespeichert werden. Gemäß der Definition der Fibonacci-Folge wird die letzte mit der vorletzten Zahl addiert. Da das Indexregister *idy* immer noch auf den höchsten Inhalt des Arrays zeigt, kann dies direkt mit einem *add*-Befehl mit indizierter Adressierung erfolgen. Damit steht die letzte berechnete Zahl im Akkumulator, und das *idy*-Register referiert den vorletzten Eintrag, wenn beide Indexregister von nun an synchron dekrementiert werden. Nach Speichern des Akkumulators im Feld muss am Ende nur noch die Schleifenbedingung überprüft werden. Solange das Indexregister *idx* nicht 0 ist, wird mit der bedingten Verzweigung zum Label *loop* gesprungen. Die Abb. 7.5 und 7.6 zeigen den vollständig abgerollten Ablauf dieses Programms für 8 Fibonacci-Zahlen. ▲

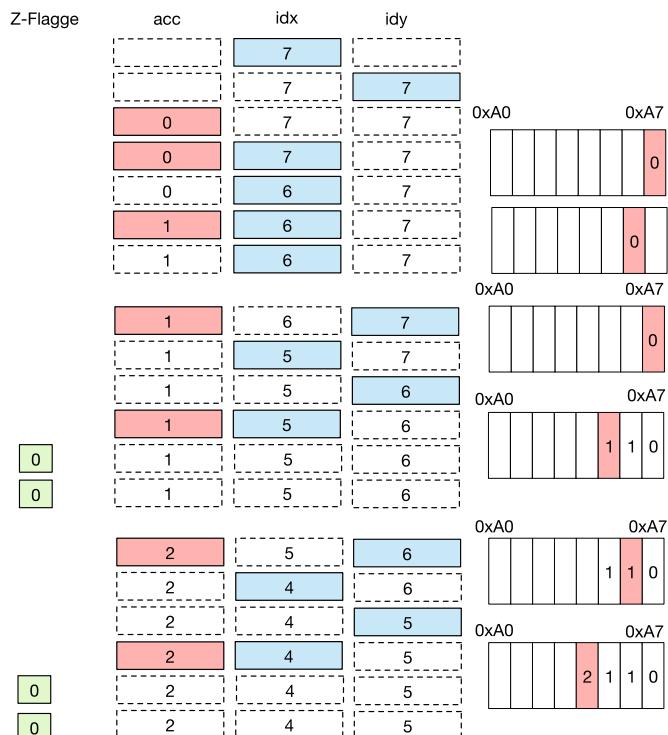


Abb. 7.5 Berechnung der Fibonacci-Folge mit einer Registermaschine (1)

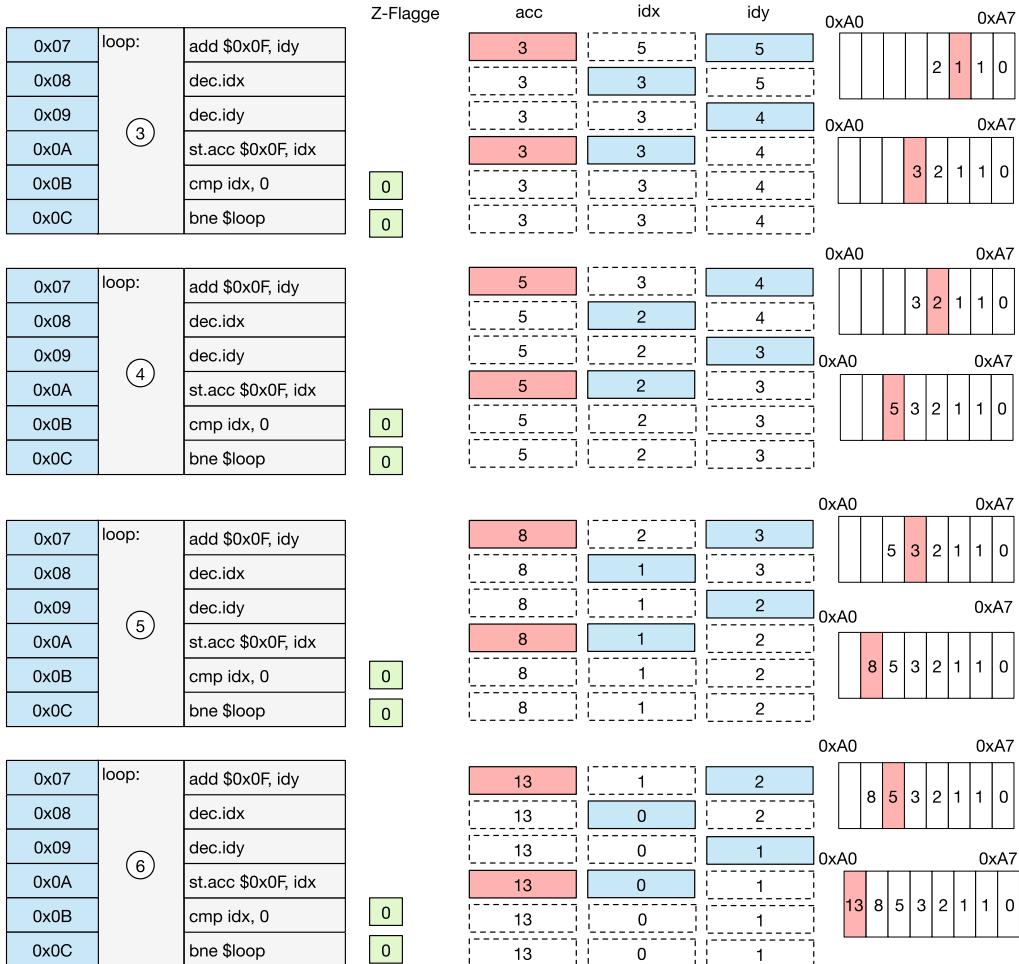


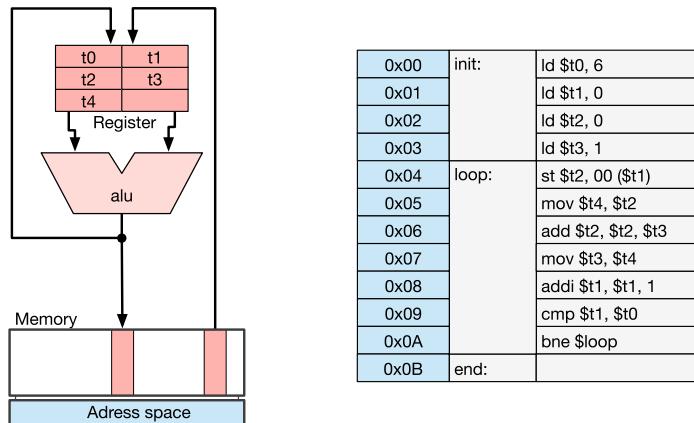
Abb. 7.6 Berechnung der Fibonacci-Folge mit einer Registermaschine (2)

Wie verändert sich das Programm zur Berechnung einer Fibonacci-Zahl, wenn es für universelle Registermaschinen geschrieben ist? Bei einer universellen Registermaschine in Load-Store-Architektur werden alle Operationen auf dem Registersatz durchgeführt. Der Aufbau einer solchen Maschine entspricht der Zeichnung auf der linken Seite von Abb. 7.4.

Load-
Store-
Architektur

► Beispiel 7.3.4 – Fibonacci-Programm mit der universellen Registermaschine

Das Programm zur Berechnung der Fibonacci-Folge einer universellen Registermaschine ähnelt dem einer Registermaschine. Bei der universellen Registermaschine kann nun jedes der Register des Registersatzes als Indexregister zum Speichern der Indexvariablen genutzt werden (Abb. 7.7). In unserem Fall ist es das tem-



■ Abb. 7.7 Mikroarchitektur einer universellen Registermaschine mit interaktivem Fibonacci-Programm

7

poräre Register $\$t_0$. Das Register $\$t_0$ fungiert dabei im Assembler als symbolischer Bezeichner einer bestimmten Registeradresse. Die letzte und die vorletzte Zahl der Folge sind jeweils einem Register zugeordnet. In unserem Fall stehen in Register $\$t_3$ immer die letzte Fibonacci-Zahl und in Register $\$t_2$ immer die vorletzte Zahl. Allerdings wird bei der universellen Registermaschine mit der registerrelativen Adressierung auf das Feld zugegriffen. Die Funktion der Register $\$t_0$ und $\$t_1$ könnte auch ein einziges Register wahrnehmen, wenn das Feld wieder von oben nach unten gefüllt werden würde. Den Ablauf dieses Programms zeigen Abb. 7.8 und 7.9. ◀

7.4 Assembler einer universellen Rechenmaschine

Im vorangegangenen Abschnitt haben wir den Einfluss unterschiedlicher Rechnerarchitekturen auf die Programmierung diskutiert. Im Folgenden soll anhand des Beispiels der Fibonacci-Folge die Übersetzung eines in Hochsprache geschriebenen Algorithmus in den konkreten Assembler eines Prozessors beschrieben werden. Da sich die Programmierkonventionen und Assembler-Sprachen unterschiedlicher Maschinen unterscheiden, legen wir uns auf einen Rechner fest, insbesondere da die Programme maschinenabhängig sind. Die Wahl fiel auf den MIPS („microprocessor without interlocked pipeline stages“). Der MIPS ist eine universelle Registermaschine in Load-Store-Architektur mit gemischter Harvard-Princeton-Architektur. Die Mikroarchitektur der MIPS wurde bereits in Abb. 7.7 skizziert. Der MIPS ist ein in der Lehre häufig verwendeter Prozessor, da sehr viel Literatur zu der Maschine existiert. Daneben hat er den Vorteil, dass er kommerzi-

7.4 · Assembler einer universellen Rechenmaschine

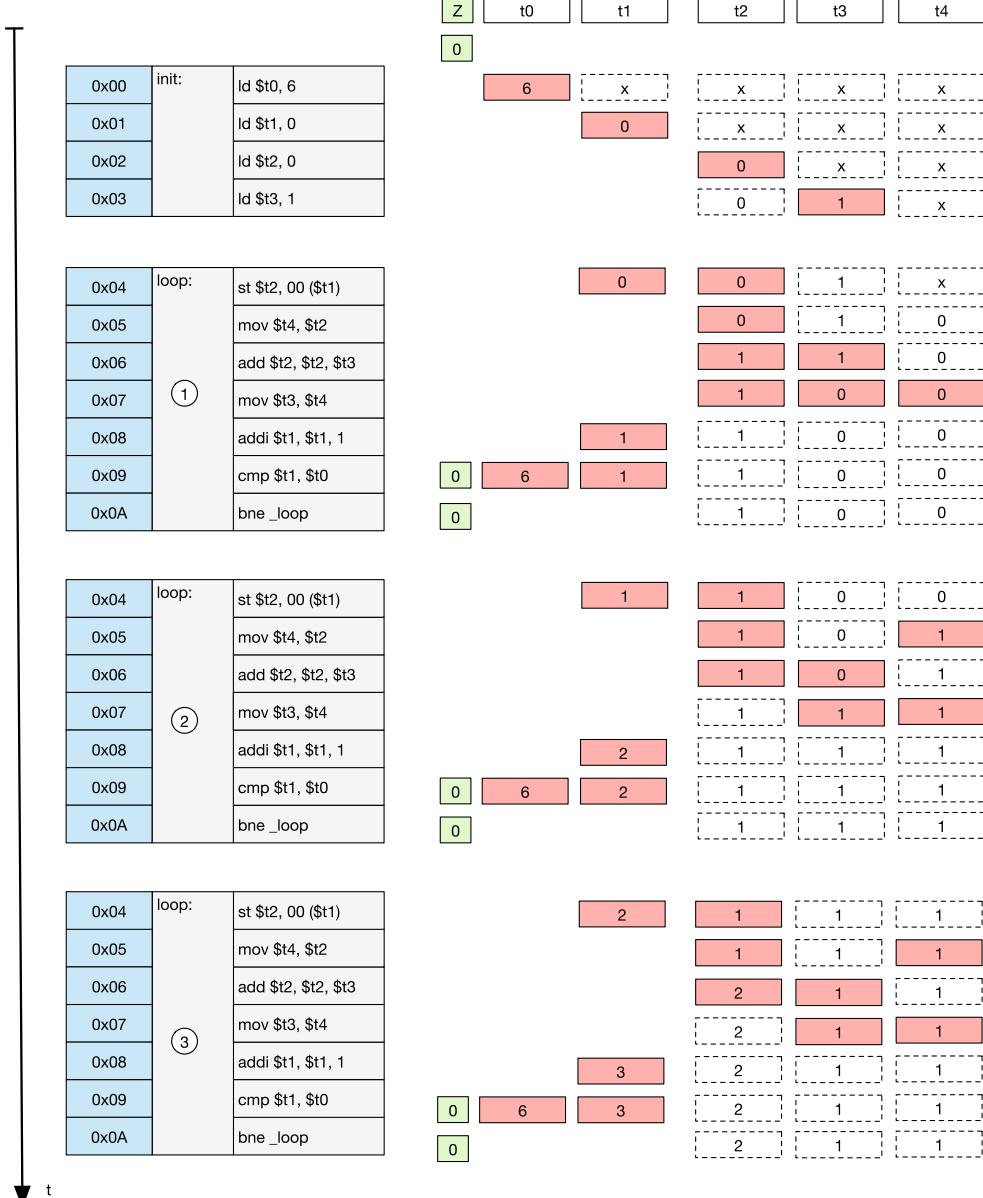


Abb. 7.8 Ablauf des iterativen Fibonacci-Programms auf einer universellen Registermaschine (1)

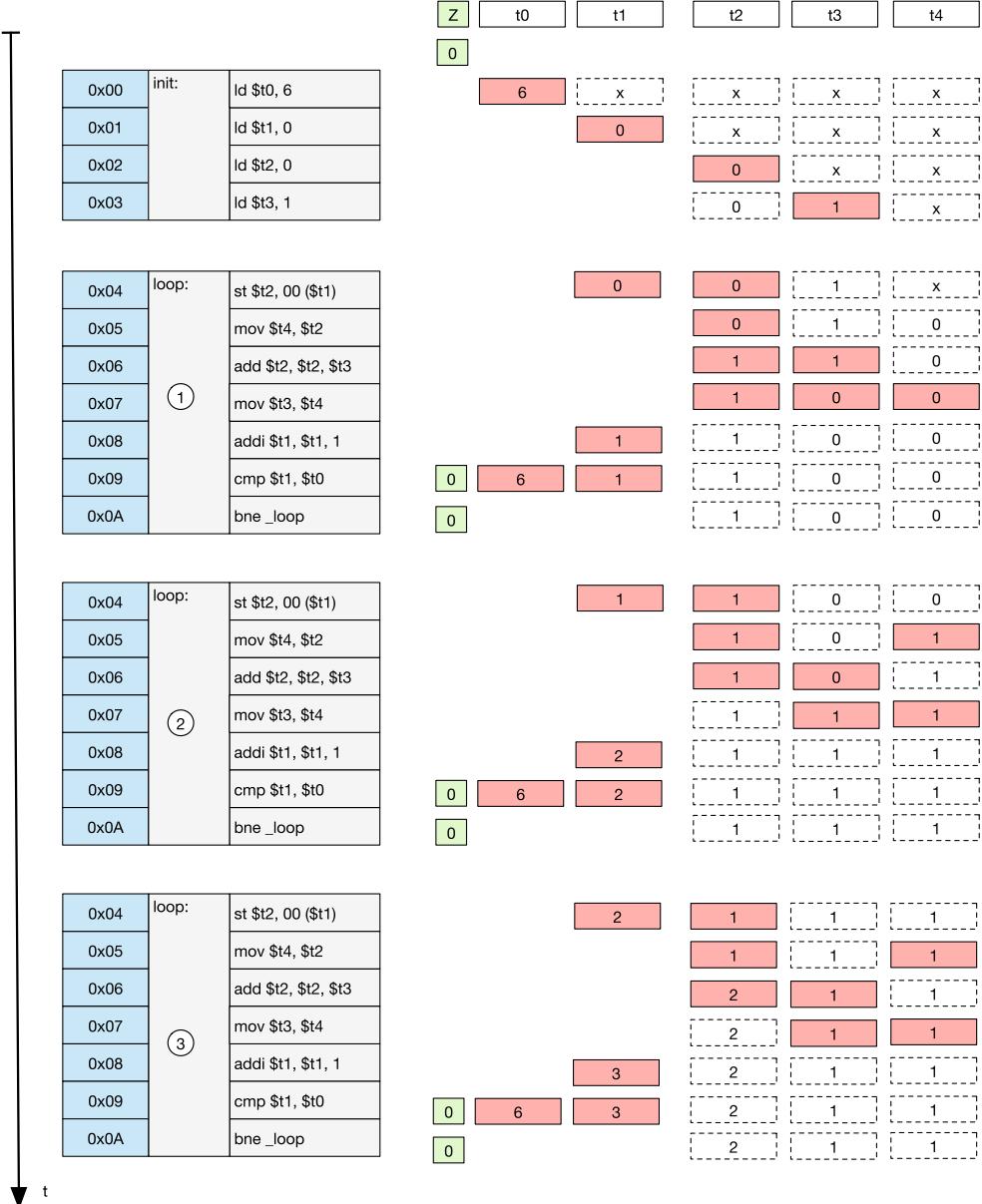


Abb. 7.9 Ablauf des iterativen Fibonacci-Programms auf einer universellen Registermaschine (2)

ell vertrieben wird, also kein Spielzeug ist. Durch die häufige Nutzung des MIPS in der Lehre existieren viele Umgebungen, die den MIPS auf einem PC emulierten. Daher kann jeder Programme für diesen Prozessor schreiben, ohne die MIPS-Hardware selbst erwerben zu müssen. Der MIPS ist eine

typische RISC-Architektur. Diese Prozessoren wurden in den 1980er-Jahren entwickelt und sind in vielen eingebetteten Systemen zu finden.

7.4.1 Programmierschnittstelle des MIPS

Die Mikroarchitektur und wie diese aus logischen Schaltungen aufgebaut wird, sind ausgiebig beschrieben worden. Auch kennen wir die Technologie und Schaltungstechnik, um zu wissen, wie sich ein Prozessor in Silizium gießen lässt. Für die drei wichtigsten Architekturen haben wir ein iteratives Fibonacci-Programm geschrieben und besprochen. Zum Programmieren komplexer Software ist aber mehr nötig als Register, Zählvariablen und ein linearer Speicher. Im Folgenden konzentrieren wir uns auf die Übersetzung grundlegender Primitive der Hochsprache C und führen als erstes Element der Softwaretechnik Unterprogramme ein. Diese ermöglichen eine gezielte Wiederverwendung von Code. Die konkrete Umsetzung von Programmen in unterschiedlichen Speichermodellen und vielleicht sogar in verteilten Systemen sei den Betriebssystemprogrammierern überlassen.

Um den MIPS zu programmieren, sind drei verschiedene Aspekte zu betrachten: Zum einen, welche Befehle können wir verwenden, und wie lauten die dazugehörigen Mnemoniken, und welche Argumente hat eine Anweisung. Die Operanden der meisten Befehle sind Zeiger auf einzelne Register des Prozessors, da der MIPS eine Load-Store-Architektur ist. Daher sind der Registersatz, die Namen seiner Register und die Konventionen zu ihrer Verwendung zu erlernen. Der zweite Aspekt, den wir aufmerksam betrachten müssen, ist der Assembler selbst. Welche Funktionen bietet uns dieser, um ein Programm bequem und effektiv schreiben zu können, und zwar in einer Form, die leicht lesbar ist? Zum Schluss diskutieren wir einzelne Primitive der Hochsprache und wie sich diese mit Assembler-Code implementieren lassen.

Programmierschnittstelle

7.4.1.1 Register

Der Registersatz des MIPS hat 32 Universalregister. Diese Register können beliebig benutzt werden. Um aber MIPS-Code möglichst übersichtlich zu gestalten, geben die MIPS-Referenzhandbücher und -Datenblätter eine Konvention zu ihrer Verwendung vor. Da diese bereits viele Problemstellungen der Programmierung berücksichtigt und Code dadurch leichter zu lesen ist, sollten sich MIPS-Assembler-Programmierer oder Compilerbauer an die vorgegebenen Regeln halten. Genau so soll es daher im Folgenden gehandhabt werden.

Tab. 7.1 Universalregister des MIPS

Name	Adresse	Verwendung
\$zero	\$0	Register enthält immer 0
\$at	\$1	Reserviert für den Assembler
\$v0–\$v1	\$2–\$3	Rückgabewerte von Funktionen
\$a0–\$a3	\$4–\$7	Argumente von Funktionen
\$t0–\$t7	\$8–\$15	Temporäre Register, ggf. Sicherung durch den Caller
\$s0–\$s7	\$16–\$23	Gesicherte Register, ggf. Sicherung durch den Callee
\$t0–\$t9	\$24–\$25	Temporäre Register, ggf. Sicherung durch den Caller
\$k0–\$k1	\$26–\$27	Reserviert für das Betriebssystem
\$gp	\$28	Globaler Zeiger, zeigt auf die MIPS-Zero-Page
\$sp	\$29	Stapelzeiger
\$fp	\$30	Rahmenzeiger
\$ra	\$31	Rücksprungadresse
\$lo		Register der /Divisionshardware Multiplikations-
\$hi		Register der /Divisionshardware Multiplikations-

Register

Die Tab. 7.1 gibt einen Überblick über die allgemein verwendbaren Register des MIPS. Ein besonderes Register ist das *Zero*-Register. In diesem steht immer die Konstante 0. Das *Zero*-Register wird vom Programmierer nicht verändert, damit diese 0 dort immer zur Verfügung steht. Insbesondere bei bedingten Sprüngen oder bei der Initialisierung von Variablen wird dieses Register häufig genutzt. Da der MIPS keinen *Move*-Befehl² besitzt und Datentransport von einem Register in ein anderes Register durch eine Immediate-Addition mit 0 erfolgt, kommt dem *Zero*-Register im MIPS eine besondere Bedeutung zu. Um einem Assembler-Programmierer

² Im ursprünglichen Befehlssatz ist kein *move* vorgesehen, und es gibt keinen Opcode dafür. Allerdings kennt der MIPS-Assembler Pseudobefehle.

das Codieren zahlreicher RISC-Befehlssequenzen zu erleichtern, sind häufige Sequenzen zu Pseudobefehlen zusammengefasst worden. So kann in MIPS-Assembler der *Move*-Befehl direkt im Code genutzt werden. Der Assembler ersetzt diesen anschließend durch eine Folge von Befehlen, die exakt die Semantik eines *Move* implementieren. Allerdings geht der Pseudobefehl *move* davon aus, dass im *Zero*-Register immer eine 0 steht. Hält sich ein Programmierer nicht an die MIPS-Programmierkonventionen und verwendet er aus Bequemlichkeit Pseudobefehle, kann es zu merkwürdigen Effekten kommen. Wenn also das *Zero*-Register anders genutzt wird als vorgesehen, resultiert das in einem unerklärlichen Verhalten des Programms. Die Einführung von Pseudobefehlen führt dazu, dass vom Assembler erzeugter Code Operationen auf Registern durchführen muss. Um das Verhalten des Assemblers für Programmierer transparent zu halten, ist das zweite Register *\$at* für den Assembler reserviert.

Die Register $\$2\text{--} \7 werden zur Implementierung von Unterprogrammen genutzt. Dabei sollen in den Registern $\$v0\text{--} \$v1$ jeweils die Rückgabewerte und in den Registern $\$a_0\text{--} \a_3 die Argumente der Funktion gespeichert sein. Mehr zur Behandlung von Unterprogrammen in Abschn. 7.5.4.

Für temporäre lokale Variablen von Funktionen sind die Register $\$t_0\text{--} \t_7 und $\$t_0\text{--} \t_9 reserviert. Selbiges gilt für die Register $\$s_0\text{--} \s_7 . Die unterschiedliche Namensgebung röhrt daher, dass die Register in der Konvention zur Implementierung von Unterprogrammen verschieden genutzt werden. Mehr dazu im Abschn. 7.5.4.

Die Register $\$k_0\text{--} \k_1 sind für den Kernel eines Betriebssystems vorgesehen. Der Rahmenzeiger und der Stapelzeiger werden in Abschn. 7.5.4 beschrieben. Der globale Zeiger (engl. „global pointer“) referenziert einen speziellen Bereich im Datensegment des MIPS. Es handelt sich um ein Segment von etwa 64 kB Größe. Dieser Bereich dient als kleiner (engl. „small“) Speicher für Verwaltungstabellen und wird wie die Zero-Page eines CISC-Prozessors genutzt. Über den Global-Pointer kann über einfache relative Adressierung auf die Tabellen zugegriffen werden.

Zum Schluss bleiben noch die Register zur Verwaltung des Stapsels. Wie jeder moderne Prozessor simuliert der MIPS einen Stapselspeicher, insbesondere, um geschachtelte Funktionsaufrufe zu ermöglichen, den Stapel- und den Rahmenzeiger sowie das Register *\$ra* (engl. „return address“). In Letzterem wird die Adresse gespeichert, zu der von einem Unterprogramm aus zurückgesprungen werden soll. In einer Architektur mit Fließbandverarbeitung bietet es sich an, einen Platz für die Rücksprungadresse im Registersatz zu reservieren. Dadurch wird dieses Register nur bei Aufruf- und Rücksprung von Unter-

**Register für
Unterprogramme**

**Temporäre
Register**

Kernelregister

Rücksprungadresse

Spezialregister

programmen genutzt. Das verhindert Ressourcenkonflikte bei der Fließbandverarbeitung.

Neben diesen allgemein zu verwendenden Registern sind für den Programmierer noch weitere Register sichtbar. Nicht alle sollen hier beschrieben werden. Diese Register gehören nicht zum allgemeinen Registersatz des MIPS. Das bekannteste dieser Spezialregister ist der Befehlszähler (engl. „program counter“), dessen Inhalt ein Zeiger auf die Adresse des als Nächstes zu ladenden Befehls ist. Darüber hinaus verfügt der MIPS noch über ein spezielles Register, das die Multiplikation von ganzen Zahlen in der MIPS-Hardware unterstützt. Bei der Multiplikation zweier 32-Bit-Werte kann das Resultat 64 Bit groß werden. Da eine solche Zahl nicht in den Universalregistern, die nur 32 Bit breit sind, gespeichert werden kann, steht das Ergebnis der Multiplikation beim MIPS in einem speziellen 32-Bit-Register. Für den Programmierer ist dieses Register als zwei 32-Bit-Stellen sichtbar: zum einen das Register *HI* und zum anderen das Register *LO*. In dem Register *HI* steht die höherwertige Hälfte des Ergebnisses, in dem Register *LO* die niedrigerwertige. Mit zwei speziellen Befehlen können die Inhalte dieser beiden Register in Universalregister verschoben werden. Die Hardware arbeitet im Gegensatz dazu nahtlos auf den vollständigen 64 Bit des *HI – LO*-Registers.

Alle Ausnahmebehandlungen (engl. „traps“ und „exceptions“) werden in der MIPS-Architektur an einen Coprozessor ausgelagert. Dieser Coprozessor verwaltet den MIPS. Zahlreiche Register des Coprozessors CP0 dienen der Speicherverwaltung. Sie konfigurieren den TLB und den virtuellen Speicher (Tab. 7.2). Im Folgenden beschränken wir uns auf ein paar wenige Register mit zentraler Bedeutung: zum einen das Statusregister, das den aktuellen Prozessorstatus anzeigt, und zum anderen das Konfigurationsregister, das den Prozessor konfiguriert. In diesem Register kann der MIPS z. B. zwischen big- und little-endian umgeschaltet werden. Im *Cause*-Register legt die Hardware den Grund der jeweiligen Ausnahme ab. Durch das Lesen dieses Registers kann der Programmierer entscheiden, welche Ausnahmebehandlung in Software erfolgen soll. Das EPC-Register (engl. „exception program counter“) zeigt auf eine Adresse, an der der jeweilige Code zur Ausnahmebehandlung stehen sollte. Dafür ist der Programmierer verantwortlich. Dieser Code muss prüfen, welche Ausnahme vorliegt, um dann die jeweils richtige Funktion aufzurufen. Das letzte der hier diskutierten Coprozessorregister ist das *Count*-Register. Dieses Register kann so konfiguriert werden, dass es unterschiedliche Teiler des Prozessortaktes zählt. Damit kann ein Timer realisiert werden. Die CP0-Prozessorregister sind stark von der jeweiligen Implementierung des MIPS abhängig. Zur genaueren Diskussion dieser Register sei auf das Referenz-

Tab. 7.2 Wichtige CP0-Register des MIPS

Symbolischer Name	Verwendung
Statusregister	Zeigt den Status des Prozessors und der Speicherverwaltung an
Configuration-Register \$0	Konfiguration des MIPS durch den Programmierer
Cause-Register	Verwaltung der Ausnahmebehandlung
EPC-Register	Dieses Register enthält eine Adresse, die auf den Code zur Ausnahmebehandlung zeigt
Count-Register	Ein Register, das Taktzyklen zählt und als Timer verwendet werden kann

handbuch (engl. „programme reference handbook“) des MIPS verwiesen. Dieses Handbuch sollte ein Programmierer für die jeweils von ihm verwendete Architektur griffbereit auf seinem Schreibtisch liegen haben, um ggf. Details der Maschine nachschlagen zu können. Schreibt man das erste Mal Programme für einen Prozessor, ist dieses Handbuch das wichtigste Dokument, dass gelesen werden sollte.

7.4.1.2 Adressierungsarten des MIPS

Die MIPS-Architektur unterstützt 3 verschiedene Befehlsformate und 4 unterschiedliche Adressierungsarten: die direkte Registeradressierung, die unmittelbare Adressierung, die befehlzählrelative und die registerrelative Adressierung. Die Abb. 7.10 gibt einen Überblick, wie die jeweils unterschiedlichen Adressen aus den Informationen des Befehlswortes berechnet werden.

Die einfachste Adressierungsart des MIPS ist die Registeradressierung. Diese ergibt sich direkt aus dem Registerformat (R-Format), einem der drei MIPS-Befehlsformate. Befehle im R-Format sind hauptsächlich die arithmetisch-logischen. Jede Anweisung dieses Formats adressiert 3 Registeradressen und verknüpft diese. Konvention im MIPS-Assembler ist, dass das erste im Befehlswort angegebene Register das Zielregister zum Speichern des Ergebnisses ist und sich die zwei folgenden Adressen auf die Argumente der Operation beziehen. Das Beispiel an Marke *ex0* zeigt das Vorgehen im Fall des arithmetischen Befehls *add*. Die Register *\$t1* und *\$t2* sind die Argumente, die addiert werden sollen, und in *\$t1* wird die Summe nach der Ausführung des Befehls gespeichert.

Registeradressierung

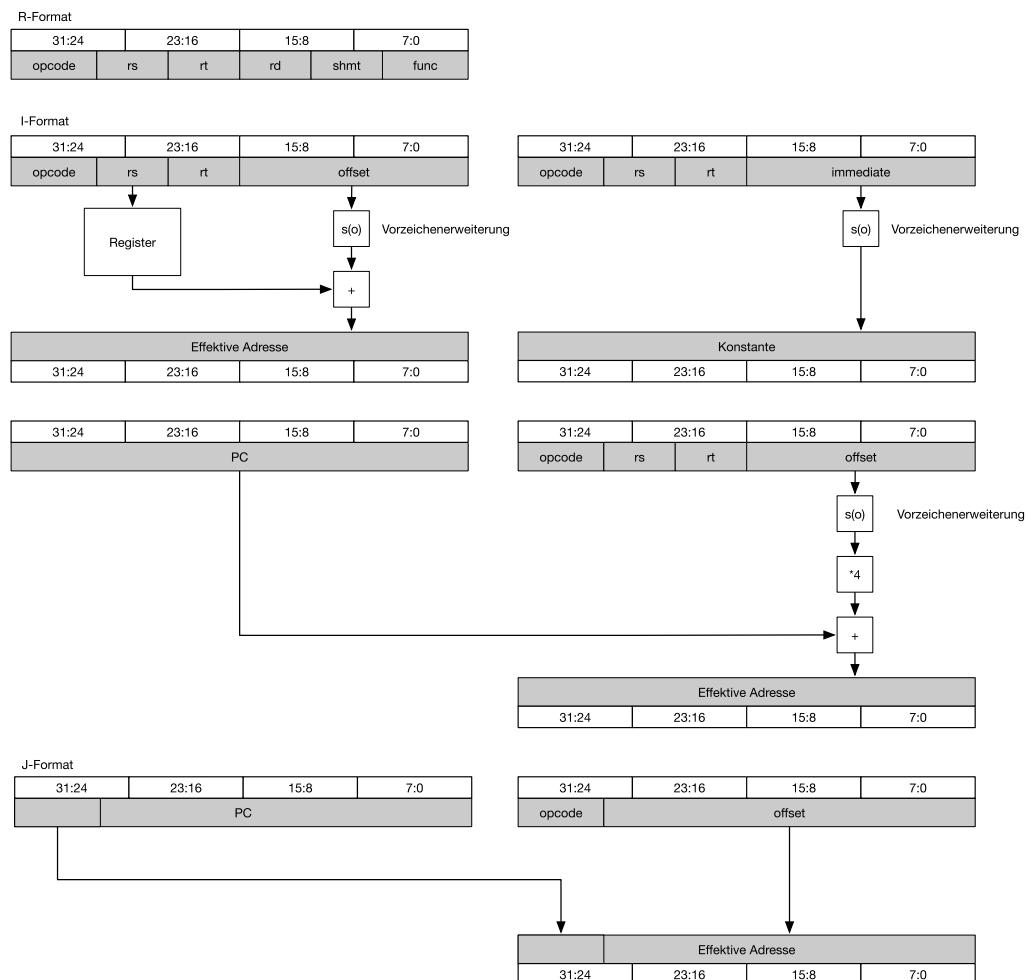


Abb. 7.10 Adressberechnung im MIPS

```
1 || ex0:      add      $t1, $t2, $t3      # $t1 = $t2+$t3
```

Immediate- Adressierung

Eine arithmetische Operation kann auch eine Konstante auf den Inhalt eines Registers addieren oder subtrahieren und das Ergebnis in einem dritten Register abspeichern. Das Beispiel an Marke *ex1* ist eine derartige Immediate-Operation. In dem Fall sollen 4 auf den Inhalt von Register \$t2 addiert und das Ergebnis in Register \$t1 abgelegt werden. In Abb. 7.10, 2. Zeile links kann man erkennen, dass beim I-Format nicht alle 32 Bit für die Konstante (immediate) zur Verfügung stehen. Daher ist eine Vorzeichenerweiterung durchzuführen. Das bedeutet, das höchstwertige Bit der Konstanten, Bit 15 wird gelesen, und dann kopiert die Hardware dieses Bit in die Bits 31–16. Auf die-

se Weise entsteht ein neuer 32-Bit-Wert, mit dem der MIPS anschließend rechnen kann. Die Vorzeichenerweiterung erlaubt es auch, negative Konstanten zu behandeln. Solche negative Werte als Konstante zeigt das Beispiel an Marke ex2.

```
1 || ex1:    addi    $t1, $t2,  4      # $t1 = $t2+4
2 || ex2:    addi    $t1, $t2, -4     # $t1 = $t2-4
3 || ex3:    addiu   $t1, $t2,  4      # $t1 = $t2+4
```

Neben den arithmetisch-logischen Operationen als reine 3-Adress-Befehle und den Befehlen auf Konstanten sind im Befehlssatz des MIPS noch bedingte und unbedingte Sprünge definiert. Bedingte Sprünge verwenden ebenfalls das I-Format. Dabei werden die unteren 16 Bit als Adresse (engl. „displacement“) interpretiert. Da auch dieser Modus durch die Vorzeichenerweiterung der Hardware unterstützt wird, kann im Programm mit einer bedingten Verzweigung vor- und zurückgesprungen werden. Das auf 32 Bit vorzeichenerweiterte Argument des Sprungs wird anschließend mit 4 multipliziert; in der Hardware kann dies direkt mit einer Schiebeoperation um 2 Bit erfolgen. Das Ergebnis wird dann auf den Inhalt des Befehlszählers addiert oder subtrahiert. Ein bedingter Sprung kann damit einen Bereich von $2^{16} \cdot 4 = 2^{19} = 262.144\text{ Byte} = 256\text{ kB}$ vor und hinter der Adresse des Sprungbefehls erreichen. Die Adressierung eines bedingten Sprungs erfolgt immer relativ zum Inhalt des Befehlszählers.

```
1 || ex4:    beq    $t1, $t2, ex5      # if($t1 == $t2) ex5
```

Neben den bedingten erlaubt der MIPS unbedingte Sprünge; zum einen, um einen größeren Adressraum zu unterstützen, und zum anderen, um Unterprogramme zu realisieren. Durch die einheitliche Länge der Befehlsworte kann in der MIPS-Architektur nicht die komplette Sprungadresse im Befehlswort gespeichert werden. Bei einem Sprungbefehl sind immer die obersten 6 Bit des Befehlswortes für den Opcode reserviert. Diese fehlen aber in der Adresse des Sprungziels. Anders als bei bedingten Sprüngen wird bei unbedingten nicht der Stand des Befehlszählers zum Inhalt des Befehlswortes addiert. Statt dessen werden die obersten 6 Bit des Befehlszählers in die 6 bit breite Lücke der Befehlswortadresse einkopiert. Dadurch können die fehlenden Bits von der Hardware ohne eine aufwendige arithmetische Operation ergänzt werden. Diese Art der Adressierung haben wir bis jetzt noch nicht behandelt. Sie wird pseudodirekte Adressierung genannt, und die Beispiele an den Mar-

Befehlszählerrelative Adressierung

Pseudodirekte Adressierung

ken *ex5* und *ex6* zeigen die Verwendung im Assembler-Code. Unbedingte Sprünge im MIPS verwenden das J-Format der drei Befehlsformate.

1	ex5:	j	ex0
2	ex6:	jal	ex1

Registerrelative Adressie- rung

Zum Schluss müssen noch die für eine universelle Registerarchitektur typischen *Load*- und *Store*-Befehle diskutiert werden. Beide Befehle nutzen das I-Format und haben damit eine 16-Bit-Displacement-Adresse zur Verfügung. Die Berechnung der effektiven Adresse ist ebenfalls in Abb. 7.10 gezeigt. Dazu wird der Inhalt des im Befehl spezifizierten Registers zu dem vorzeichenverweiterten Argument des Befehlswortes addiert. Das Speichern und Laden eines Datums können daher relativ zu der in einem Register gespeicherten Adresse erfolgen. Damit kann indiziert auf einzelne Speicherbereiche zugegriffen werden. Eine registerrelative Adressierung nutzt auch der *jr*-Befehl. In dem Fall wird der Befehlszähler auf einen Wert aus dem Register \$31, der Rücksprungadresse eines Unterprogrammaufrufs gesetzt.

1	ex7:	lw	\$t1, 4 (\$t3)
2	ex8:	sw	\$t1, 4 (\$t3)
3	ex9:	jr	# return

Arithmetische Befehle

7.4.1.3 Befehle

Die Tab. 7.3 fasst alle arithmetischen Befehle des MIPS zusammen. Dies sind Anweisungen zur Addition, Subtraktion, Multiplikation und Division vorzeichenbehafteter und vorzeichenloser Zahlen. Dabei stehen in den Registern \$s und \$t immer die Argumente des Befehls, und das Ergebnis wird im Register \$d gespeichert. Zwei spezielle Additionsbefehle ermöglichen zu einer in einem Register gespeicherten Variablen eine Konstante zu addieren. Da in dem dafür verwendeten I-Befehlsformat des MIPS nicht genügend Platz für eine 32-bit-Konstante ist, wird das Datum des Befehls um das Vorzeichen erweitert. Dazu werden die fehlenden Ziffern eines 32-bit-Wortes mit dem Inhalt des höherwertigen Bits des Befehlsarguments aufgefüllt. Bis auf diese zwei Konstantenadditionen verwenden die arithmetischen Befehle das R-Format der MIPS-Formate.

Tab. 7.3 Arithmetische Befehle der MIPS

Befehl	Name	Bedeutung
add \$d, \$s, \$t	„Addition“	Vorzeichenbehaftete Addition zweier Register
addu \$d, \$s, \$t	„Addition unsigned“	Addition zweier Register
addi \$d, \$s, i	$\$d = \$s + i$ „Addition immediate“	Vorzeichenbehaftete Addition einer Konstanten
addiu \$d, \$s, i	„Addition immediate unsigned“	Addition einer Konstanten
div \$s, \$t	„Division“	Vorzeichenbehaftete Division zweier Register, Ergebnis legt in HI und LO
divu \$s, \$t	„Division unsigned“	Division zweier Register, Ergebnis legt in HI und LO
mult \$s, \$t	„Multiplication“	Vorzeichenbehaftete Multiplikation zweier Register, Ergebnis legt in HI und LO
multu \$s, \$t	„Multiplication unsigned“	Multiplikation zweier Register, Ergebnis legt in HI und LO
sub \$d, \$s, \$t	„Subtraction“	Vorzeichenbehaftete Subtraktion zweier Register
subu \$d, \$s, \$t	„Subtraction unsigned“	Subtraktion zweier Register

Befehle für die logischen Verknüpfungen UND, ODER, NOR und XOR vervollständigen die Gruppe der arithmetisch-logischen Operationen (Tab. 7.4). Hinzu kommen Anweisungen zum Links- und Rechtsschieben. Einige logische Operationen arbeiten ausschließlich auf Registern, andere zusätzlich auf Konstanten (sll und sllv, engl. „shift logical left“ und „shift logical left variable“). Je nach Befehl liegt dieser dann im R- oder I-Format vor. Die Schiebebefehle schieben Bits um n Bit nach rechts oder links. Das Argument kann entweder als Konstante direkt im Befehl stehen, es liegt das I-Format vor, oder die Operation adressiert ein Register, dann handelt es sich um einen R-Format-Befehl. Beim Linksschieben wird das niedrigstwertige Bit immer mit einer 0 aufgefüllt. Für das Rechtsschieben sind zwei Varianten zu unterscheiden: Das arithme-

Logische Befehle

Tab. 7.4 Logische Befehle des MIPS

Befehl	Funktion	Bedeutung
and \$d, \$s, \$t	$\$d = \$s \& \$t$	UND-Verknüpfung aller Bits des Inhalts zweier Register
andi \$d, \$s, i	$\$d = \$s \& i$	UND-Verknüpfung mit einer Konstanten
nor \$d, \$s, \$t	$\$d = \$s \parallel \$t$	ODER-Verknüpfung aller Bits des Inhalts zweier Register
or \$d, \$s, \$t	$\$d = \$s \parallel \$t$	ODER-Verknüpfung aller Bits des Inhalts zweier Register
ori \$d, \$s, i	$\$d = \$s \parallel i$	ODER-Verknüpfung mit einer Konstanten
sll \$d, \$s, n	$\$d = \$s \gg n$	Logisches Linksschieben um n Bits
sllv \$d, \$s, \$t	$\$d = \$s \gg \$t$	Logisches Linksschieben um \$s Bits
sra \$d, \$s, n	$\$d = \$s \gg \$n$	Arithmetisches Rechtsschieben um n Bits
sraw \$d, \$s, \$t	$\$d = \$s \gg \$t$	Arithmetisches Rechtsschieben um \$t Bits
srl \$d, \$s, n	$\$d = \$s \gg n$	Logisches Rechtsschieben um n Bits
srlv \$d, \$s, \$t	$\$d = \$s \gg \$t$	Logisches Rechtsschieben um \$s Bits
xor \$d, \$s, \$t	$\$d = \$s XOR \$t$	XOR-Verknüpfung aller Bits des Inhalts zweier Register
xori \$d, \$s, i	$\$d = \$s XOR i$	XOR-Verknüpfung mit einer Konstanten

tische Schieben ersetzt das höchstwertige Bit entsprechend des Vorzeichens der Variablen. Steht dort eine 0, wird an die Stelle wieder eine 0 geschrieben, ansonsten eine 1. Schiebebefehle können mit einem konstanten Eintrag im Befehlswort oder ein Register (engl. „shift word right arithmetic variable“, sarv) spezifizieren, um wie viele Bits geschoben wird. Beim logischen Rechtsschieben wird unabhängig vom Vorzeichen des Arguments das höchstwertige Bit immer durch eine 0 ersetzt.

Der MIPS ist eine Load-Store-Architektur. Aus diesem Grund kann der Datentransport von den Registern zum Speicher und vom Speicher zu den Registern ausschließlich mit den Befehlen *load* und *store* erfolgen. Die MIPS-Architektur

Transportbefehle

spezifiziert unterschiedliche Load- und Store-Befehle. Neben dem Laden oder Speichern eines ganzen 32-Bit-Wortes ist es auch möglich, nur 1 Byte oder ein Halbwort aus dem Speicher zu lesen oder in den Speicher zu schreiben. Eine Besonderheit des MIPS ist ein Register zum Speichern der Ergebnisse von ganzzahligen Multiplikationen und Divisionen. Dieses Register umfasst 64 Bit und kann als 2 virtuelle Register angesprochen werden. Dabei werden der obere Teil des Registers *HI* (engl. „high“) und der untere *LO* (engl. „low“) genannt. Mit vier speziellen Befehlen können Daten inhärent aus diesem Register in andere Register transportiert werden. Alternativ kann auch der Inhalt eines beliebigen Registers in *HI* oder *LO* gespeichert werden (Tab. 7.5).

Die Tab. 7.6 fasst Befehle für bedingte Sprünge zusammen. Ein bedingter Sprungbefehl wird im MIPS mithilfe des

Bedingte Sprungbefehle

■ Tab. 7.5 Transportbefehle des MIPS

Befehl	Name	Bedeutung
lb \$t, i(\$s)	„Load byte“	Lade Byte vorzeichenbehaftet
lbu \$t, i(\$s)	„Load byte unsigned“	Lade Byte
lh \$t, i(\$s)	„Load half word“	Lade halbes Wort vorzeichenbehaftet
lhu \$t, i(\$s)	„Load half word unsigned“	Lade halbes Wort
lw \$t, i(\$s)	„Load word“	Lade Wort
sb \$t, i(\$s)	„Store byte“	Speichere Byte
sh \$t, i(\$s)	„Store half word“	Speichere halbes Wort
sw \$t, i(\$s)	„Store word“	Speichere Wort
mfhi \$d	„Move from HI“	Kopiert den Inhalt des HI-Registers in das Register „store byte“
mflo \$d	„Move from LO“	Kopiert den Inhalt des LO-Registers in das Register \$d
mthi \$s	„Move to HI“	Kopiert den Inhalt des Registers \$s in HI
mtlo \$s	„Move to LO“	Kopiert den Inhalt des Registers \$s in LO

■ Tab. 7.6 Bedingte Sprungbefehle des MIPS

Befehl und Argumente	Name	Bedeutung
<i>beq \$s, \$t, o(pc)</i>	„Branch on equal“	Springe, wenn Register $\$s = \t
<i>bgtz \$s, o(pc)</i>	„Branch on greater than zero“	Springe, wenn Register $rs > 0$
<i>blez \$s, o(pc)</i>	„Branch on less or equal to zero“	Springe, wenn Register $rs \leq 0$
<i>bne \$s, \$t, o(pc)</i>	„Branch on not equal“	Springe, wenn Register $\$s \neq \t

7

Vergleichsbefehle**Sprungbefehle**

I-Formats codiert. Dabei werden zwei Register auf Gleichheit geprüft, und je nach Ergebnis wird der Programmfluss verzweigt oder nicht. Im hinteren Teil des Befehls steht ein Adressversatz oder Adressoffset. Dieser Wert wird in der Hardware zunächst vorzeichenerweitert und dann zum Befehlszähler hinzugeaddiert. Bedingte Sprünge werden in der MIPS-Architektur also immer relativ zum Inhalt des Befehlszählers durchgeführt.

Die bedingten Sprungbefehle des MIPS reichen nicht aus, um alle notwendigen Sprünge zu realisieren, da mit diesen nur auf Gleichheit und größer gleich oder kleiner gleich 0 getestet werden kann. Es fehlen noch Tests für die Relation kleiner und ob Konstanten bestimmte Bedingungen erfüllen. Die folgenden Befehle unterstützen diese Varianten. Allerdings werden für diese Anweisungen keine bedingten Sprünge implementiert. Vielmehr wird bei positiver Prüfung ein Register auf den Wert 1 gesetzt, wenn die entsprechende Bedingung wahr ist (Tab. 7.7).

Die letzte Gruppe der MIPS-Befehlsarchitektur sind Sprünge. Für diese wird ein eigenes Befehlsformat definiert, um möglichst viele Ziffern für die Sprungadresse bereitzustellen zu können. Durch die Vereinheitlichung aller Befehlsformate auf 32 Bit und um die Fließbandverarbeitung optimal zu unterstützen, fehlen jedoch einige Bits zu einer vollständigen MIPS-Adresse. Diese Bits werden für den Opcode der Anweisung benötigt. Aus diesem Grund wird der Inhalt des Befehlswortes mit dem Befehlszähler konkateniert. Das verhindert zwar Sprünge durch den vollständigen Adressraum, erlaubt jedoch ausreichend weite Sprünge. Ist die Sprungreichweite nicht groß genug, sind ggf. mehrere Sprunganweisungen kaskadiert zu programmieren. Neben dem reinen Sprungbefehl sind die Anweisungen *jl* (engl. „jump and link“) und *jr* (engl. „jump return“) von großer Bedeutung. Diese beiden Befehle erlauben

Tab. 7.7 Vergleichsbefehle des MIPS

Befehl	Name	Bedeutung
slt \$d, \$s, \$t	„Set on less than“	Setzt das Register \$d auf 1, wenn der Inhalt \$t kleiner als \$s ist
sltu \$d, \$s, \$t	„Set on less than unsigned“	Setzt das Register \$d auf 1, wenn der Inhalt \$t kleiner als \$s ist. Die Integer werden als nicht vorzeichenbehaftet interpretiert
slti \$d, \$s, i	„Set on less than immediate“	Setzt das Register \$d auf 1, wenn der Inhalt \$t kleiner als die vorzeichenerweiterte Konstante i ist
sltiu \$d, \$s, \$t	„Set on less than immediate unsigned“	Setzt das Register \$d auf 1, wenn der Inhalt \$t kleiner als die vorzeichenerweiterte Konstante i ist. Die Integer werden als nicht vorzeichenbehaftet interpretiert

Tab. 7.8 Sprungbefehle des MIPS

Befehl	Name	Bedeutung
j o	„Jump“	Springe zu der mit dem Befehlszähler konkatenierten Adresse
jal o	„Jump and link“	Springe zu der mit dem Befehlszähler konkatenierten Adresse, und speichere den Stand des Befehlszählers in \$ra
jalr d s	„Jump and link register“	Springe zu der mit s konkatenierten Adresse, und speichere d + 4 in \$ra
jr s	„Jump register“	Springe zu der Adresse in s
j o	„Jump“	Springe zu der mit dem Befehlszähler konkatenierten Adresse
jal o	„Jump and link“	Springe zu der mit dem Befehlszähler konkatenierten Adresse, und speichere den Stand des Befehlszählers in \$ra
jalr d s	„Jump and link register“	Springe zu der mit s konkatenierten Adresse, und speichere d + 4 in \$ra
jr s	„Jump register“	Springe zu der Adresse in s

Tab. 7.9 MARS/SPIM-syscall-Codes

Aus-nahme	Bezeichnung	Ausnahm	Bezeichnung
1	„Print integer“	9	„Allocate memory“
2	„Print float“	10	„Exit“
3	„Print double“	11	„Print character“
4	„Print string“	12	„Read character“
5	„Read integer“	13	„Open file“
6	„Read float“	14	„Read file“
7	„Read double“	15	„Write to file“
8	„Read string“	16	„Close file“

Syscall

es, eine Adresse während eines Sprungs im Register \$ra des Registersatzes zu speichern und genau an diese Stelle im Speicher zurückzuspringen (Tab. 7.8).

Der Befehl *syscall* erlaubt es dem Programmierer eine Softwareausnahme auszulösen. Dazu werden in Register \$v0 ein Wert gespeichert und anschließend *syscall* aufgerufen. Je nach Parameter wird dann die entsprechende Ausnahmebehandlung ausgeführt. Bedingt durch die Konventionen der MIPS-Architektur legt der Assembler bereits automatisch Ausnahmebehandlungsprogramme und die notwendige Tabelle zur Ausnahmebehandlung an. Die folgenden Codes werden in den Emulatoren SPIM und MARS verwendet (Tab. 7.9).

7.4.2 Assembler-Anweisungen

Eine Datei, die von einem MIPS-Assembler gelesen und in Maschinencode übersetzt werden kann, besteht aus mehreren Zeilen. Jede entspricht exakt einem Befehlszyklus des MIPS. Daher steht dort jeweils nur ein Befehl. Die Adresse der jeweiligen Zeile einer Anweisung wird vom Assembler oder genauer gesagt dem Linker automatisch berechnet. Sie wird beim Programmieren daher weggelassen. Da aber ggf. in bestimmte Zeilen gesprungen werden soll, müssen diese gekennzeichnet werden. Dazu erlaubt der MIPS-Assembler die Definition von Marken. Eine solche Marke ist eine Zeichenkette, immer gefolgt von einem Doppelpunkt. Dieser Markierung schließt sich dann die Befehlsmnemonik mit ihren jeweiligen Argumenten.

ten an. Das Ende der Zeile kann noch mit einem Kommentar ergänzt werden.

```
1 || label: <Memnmonic> <arg1><arg2><arg3> # <comment>
2 || label: <Memnmonic> <arg1>,<arg2>,<arg3> # <comment>
```

Assembler-Anweisungen (engl. „directive“) sind Kommandos an den Assembler. Sie steuern die Übersetzung des Assembler-Programms in das entsprechende Maschinenprogramm. Da die Direktive nur vom Assembler genutzt werden, wird für sie selbst kein Maschinencode erzeugt. Das spätere Maschinenprogramm ist also frei von diesen Anweisungen.

Direktive

Mithilfe der Direktiven kann festgelegt werden, an welche Stelle Assembler und Linker den Code in den Speicher schreiben. Bekanntlich wird dieser in mindestens drei feste Bereiche oder Segmente unterteilt. Im 1. Teil befindet sich das Programm. Im MIPS-Assembler ist dies das *Text*-Segment. Zeilen in der Assembler-Datei, die mit der Direktive *.text* beginnen, markieren diesen Bereich des Speicherlayouts. Die Direktive *.data* bezeichnet leicht einsehbar das Datensegment. Das 3. Segment, der Stapel wird dynamisch über den Stapelzeiger *\$sp* angesprochen. Die Tab. 7.10 fasst alle wichtigen Assembler-Anweisungen zusammen. Insbesondere die Direktiven *.byte*, *.halfword* und *.word* sind zu beachten, lassen sich mit diesen doch die globalen Variablen des Programms spezifizieren und der Speicher für diese reservieren. Neben diesen leicht zu verwendenden Assembler-Anweisungen gibt es eine Reihe spezieller Direktiven (Tab. 7.11).

Segmente

In C wird der Eintrittspunkt in ein Programm mit der Funktion *main* spezifiziert. Dazu benötigt jeder Prozessor eine kleine Start-Firmware. Diese ruft nach erfolgreichem Start des Prozessors die Funktion *main* auf. Manchmal reicht es sogar, die Funktion *main* an die Startadresse des Prozessors zu legen. *Main* ist das Hauptprogramm, und aus diesem können dann weitere Unterprogramme aufgerufen werden. Die Programmierung von Unterprogrammen beschreibt Abschn. 7.5.4. Zunächst einmal müssen wir nur wissen, dass es möglich ist, Unterprogramme zu schreiben. Wird ein C-Programm aus einem Betriebssystem heraus aufgerufen, dann ist die Routine *main* nichts anderes als ein Unterprogramm des Betriebssystems. Das folgende C-Programm (7.5) berechnet n Zahlen der Fibonacci-Folge und verwendet dazu zwei Unterprogramme: zum einen die Funktion *rFib*, die eine Fibonacci-Zahl rekursiv berechnet, und zum anderen die Funktion *iFib*, die ei-

Tab. 7.10 Wichtige Direktiven des MIPS-Assemblers

MIPS-Direktive	Bedeutung
.global sym	Definition eines globalen Symbols, das von anderen Dateien aus referenziert werden kann
.extern sym n	Externer, in anderen Dateien sichtbarer <i>n</i> -Byte-Speicherplatz
.space n	Reserviert <i>n</i> Plätze im entsprechenden Segment
.align n	Ausrichten eines Datums an 4-B-Grenzen
.kdata <adr>	Datensegment des Kernels, ggf. an der Adresse <adr>
.ktext <adr>	Textsegment des Kernels, ggf. an der Adresse <adr>
.data <adr>	Datensegment des Nutzer, ggf. an der Adresse <adr>
.text <adr>	Textsegment des Nutzers, ggf. an der Adresse <adr>
.ascii str	Nicht terminierender String
.asciiz str	0-terminierender String
.byte b1, . . . , bn	Speichern von <i>n</i> Bytes
.half h1, . . . , hn	Speichern von <i>n</i> 16-bit-Halbworten
.word w1, . . . , wn	Speichern von <i>n</i> 32-bit-Worten
.double d1, . . . , dn	Speichern von <i>n</i> Double-Werten
.float f1, . . . , fn	Speichern von <i>n</i> Float-Werten

ne Fibonacci-Zahl iterativ bestimmt³. Zunächst einmal wird dem Programm bekannt gegeben, welche Funktionen der C-Bibliotheken verwendet werden – in diesem Fall die Bibliothek *stdio* (engl. „standard input/output“). In dieser findet sich

3 Diese Programmierung der Fibonacci-Folge ist ineffizient, da in den beiden Unterprogrammen die komplette Folge bis zum Schleifenindex kalkuliert und dann verworfen wird. Eigentlich würde es ausreichen, im Fibonacci-Unterprogramm jede berechnete Zahl auszugeben und dafür das Unterprogramm gleich mit dem Argument *n* aufzurufen. In diesem Kapitel kommt es uns jedoch nicht darauf an, die Fibonacci-Folge effizient zu programmieren, sondern darauf, anhand dieser die wesentlichen Aspekte der Assembler-Programmierung zu diskutieren. Das Beispiel mahnt allerdings, sich beim Schreiben von Programmen ggf. zu vergegenwärtigen, dass ineffizienter Code auch durch das blinde Verwenden von Bibliotheken entstehen kann.

Tab. 7.11 Weitere Direktiven des MIPS-Assemblers

MIPS-Direktive	Bedeutung
.ent str	Extern sichtbarer symbolischer Bezeichner. Markiert den Beginn einer Funktion
.end str	Extern sichtbarer symbolischer Bezeichner. Markiert das Ende einer Funktion
.frame n, <reg>	Rahmendefinition, reserviert n B auf dem Stack und spezifiziert das Rücksprungregister
.rdata <reg>	ROM-Datensegment
.sdata <reg>	Kleines Datensegment, relativ zum \$gp
.bss <reg>	Zero-Page des MIPS („block started by storage“)
.sbss <reg>	„Small“ bss, relativ zum \$gp adressierbar

das Unterprogramm *printf*. Mit diesem können Zeichenfolgen auf dem Bildschirm ausgegeben werden. In einer Schleife wird dann eine Ausgabezeile erzeugt. In dieser Zeile befinden sich jeweils das rekursive und das iterative Ergebnis der Berechnung einer konkreten Fibonacci-Zahl. Die obere Grenze der Schleife wird dem Programm *main* übergeben. Dieses wird im Folgenden schrittweise in Assembler-Code übersetzt. Dabei konzentrieren wir uns zunächst auf das Speicherlayout und die durch den Assembler vorgenommenen Initialisierungen. Daran anschließend diskutieren wir die Assembler-Implementierung der beiden Fibonacci-Unterprogramme.

```

1  #include <stdio.h>
2  int main(int n)
3  {
4      int i;
5      printf("Berechnung_von_n_Fibonacci-Zahlen_\%d:\n", n);
6      for(i; i<20; i++) {
7          printf("Rekursiv:_\%5d", rFib(i));
8          printf("_|_Iterativ:_\%5d\n", iFib(i));
9      } /* for(i; i<20; i++) */
10     return 0;
11 }
```

Listing 7.5: Berechnung von n Fibonacci-Zahlen

Datensegment

Das folgende Textsegment spezifiziert die globalen Variablen x , y , z und i . Die Variablen x , y , z werden für das Fibonacci-Programm nicht benötigt, verdeutlichen jedoch das Prinzip. Mithilfe der Direktive `.asciiz` werden die Zeichenketten für die Ausgabe des Programms spezifiziert. Um das Ende einer Kette eindeutig zu markieren, wird dort ein *Null-Zeichen* angehängt.

```

1  .data
2  x:.byte 2    # Variable as Byte, initialized with 2
3  y:.half 3    # Variable as Half Word, initialized with 3
4  z:.word 7    # Variable as Word, initialized with 7
5  i:.word 0    # Variable as Word, initialized with 0
6  str0:.asciiz  "Berechnung_von_n_Fibonacci-Zahlen_\%d:"
7  str1:.asciiz  "Rekursiv:_\%5d"
8  str2:.asciiz  "Iterativ:_\%5d"
```

Listing 7.6: Daten-Segment

Der Hauptspeicher wird zunächst in zwei Segmente aufgeteilt. Das Segment *Data* reserviert den Speicher für die globalen Variablen des Programms – in diesem Fall nur die eine Variable *int i*. Für diese werden im Beispiel ein Speicherwort mit der Direktive `.word` reserviert und dieser Speicher mit 0 initialisiert. Weitere mögliche Datentypen sind `.byte` und `.halfword`. Dem Datensegment folgt das Textsegment. Sein Beginn wird mit der Direktive `.text` angegeben und beinhaltet das eigentliche Programm. Im MIPS-Assembler ist es auch möglich, die Segmente an fest vorgegebene Adressen zu legen und richtig im Speicher auszurichten (`.align`). In der Regel übernimmt der Assembler, wenn man sich an alle Konventionen hält, die Ausrichtung automatisch. Die folgenden Anweisungen exportieren die Funktion *main*, um aus anderen Assembler-Dateien zugreifen zu können. Die Routine *main* selber ist ein Unterprogramm und sollte auch so programmiert werden. Diesen Teil diskutieren wir später. Anschließend folgt eine Schleife, die ausführlich in Abschn. 7.5.2.2 beschrieben wird. In dieser Schleife werden dann die Unterprogramme zur Ausgabe und zur Berechnung der Fibonacci-Zahl aufgerufen. Zum Schluss ist wieder Code auszuführen, wie jedes Unterprogramm beschlossen wird. Die Anweisung `.ent_main` zeigt dem Assembler das Ende der Datei an.

```

1 .data
2 i:.word 0
3 .text
4 .globl main
5 .ent main
6
7 main:
8     # main itself is a subroutine. We need some code
9     # to support something called the stack frame
10    for:
11        slti      $t0, $s0, 10          # $t0 = (i++ < 10)
12        beq      $t0, $zero, endFor   # for\index{for} ($t0) {
13        # basic block to print to the results and to call
14        # the subroutines
15        addiu    $t1, $zero, 1         # i++
16        j for                  # }
17    endFor:
18        # main itself is a subroutine. We need some code
19        # to support something called the stack frame
20 .ent main

```

Listing 7.7: Text-Segment

In manchen Fällen tritt bestimmter Code immer wieder auf.

Sind die Codefragmente hinreichend klein, soll nicht jedes Mal ein Unterprogrammaufruf durchgeführt werden. Das reduziert Code und spart Speicherplatz, da für einen solchen Aufruf ein Stapelrahmen reserviert werden muss. Um Codeschnipsel, die aus einigen wenigen Befehlen bestehen, nicht immer neu schreiben zu müssen, können in Assembler-Programmen und in Quellcode von Hochsprachen Makros genutzt werden. Ein Makro definiert ein Codefragment, und der Compiler oder Assembler kopiert dieses in einer Post-Processing-Phase an die richtige Stelle in den Code. Im Folgenden ist ein kurzes Makro für den Pseudobefehl *la* („load address“) gezeigt. Makros können Parameter haben. Diesen Assembler-Variablen wird im MIPS-Assembler immer ein Prozentzeichen vorangestellt.

Makros

```

1 #the following is MIPS assembler
2 #Note that SPIM and MARS does
3 #not support such macros, but pseudo instructions!
4 #define    la(%r,%adr)           # to know how it works
5     lui t0, %hi(adr)           # a macro is defined in MIPS
6     ori t0, t0, %lo(adr)       # la is a pseudo instruction
7     lw   r, 0(t0)

```

Listing 7.8: Assembler Makros

**SPIM-
Makros**

MIPS-Programme lassen sich mit Emulatoren auf Desktop-Computern ausführen. Die bekanntesten sind SPIM und MARS. Normalerweise unterstützen diese Emulatoren keine Makros, dafür aber Pseudoinstruktionen. Im SPIM-Emulator dagegen ist es möglich, Makros zu verwenden. Die dort genutzte Syntax ist aber verschieden von der regulären Syntax der MIPS-Assembler.

```

1  #SPIM-Macro
2  .macro la(%r,%adr)
3      lui t0, %hi(%adr)
4      ori t0, t0, %lo(%adr)
5      lw %r, 0(t0)
6  .end_macro

```

Listing 7.9: SPIM-Macro

Den MIPS in Assembler zu programmieren kann sehr mühsam sein. Als RISC-Architektur beschränkt sich der MIPS-Befehlssatz auf die notwendigsten Befehle. Für einen Compiler stellt das kein Problem dar, kann dieser doch längere Befehlssequenzen automatisch erzeugen. Für Assembler-Programmierer ist es jedoch aufwendig und fehleranfällig, immer wieder die gleichen Codeschnipsel programmieren zu müssen. Abhilfe schaffen die Assembler-Makros, mit denen sich längere wiederkehrende Schnipsel definieren lassen. Für einige viel genutzte Sequenzen stellt der MIPS bereits Makros zur Verfügung. In der MIPS-Architektur werden diese Schnipsel Pseudobefehle genannt. Für sie ist zwar eine Mnemonik definiert, diese entspricht aber einer Folge von echten⁴ MIPS-Befehlen und besitzt selbst kein Maschinenbefehlswort. Die folgende Tabelle gibt eine Übersicht über häufig verwendete MIPS-Pseudocodes. Diese ist nicht erschöpfend und je nach Assembler können weitere Pseudobefehle zur Verfügung stehen (Tab. 7.12).

⁴ Ein echter Befehl ist einer, für den ein eigener Opcode spezifiziert ist und der daher direkt von der Maschine verstanden wird.

Tab. 7.12 Pseudobefehle des MIPS-Assemblers

Befehl	Name	Bedeutung
subu \$d, \$t	„Subtract unsigned“	Subtrahiert eine Konstante von \$t, und schreibt das Ergebnis in \$d
abs \$d, \$t	„Absolute“	Berechnet den absoluten Wert von \$t, und speichert diesen in \$t
neg \$d, \$t	„Negation“	Negiert den Wert in \$t, und speichert diesen in \$d
not \$d, \$t	„Not“	Bitweises NICHT auf den Inhalt von \$t
ror \$d, \$t, n	„Rotate right“	Rechtsweises Rotieren der Bits in \$t um n bit
rol \$d, \$t, n	„Rotate left“	Linksweises Rotieren der Bits in \$t um n bit
ble \$t, i(\$s)	„Branch on less than or equal“	Bedingter Sprung für die Relation kleiner gleich
blt \$t, i(\$s)	„Branch less than“	Bedingter Sprung für die Relation kleiner
bgt \$t, i(\$s)	„Branch greater than“	Bedingter Sprung für die Relation größer
bge \$t, i(\$s)	„Branch on equal than or greater“	Bedingter Sprung für die Relation größer gleich
li \$t, i	„Load immediate“	Lädt eine Konstante in das spezifizierte Register
la \$d	„Load address“	Lädt eine Adresse in das spezifizierte Register
move \$d \$s	„Move“	Kopiert den Inhalt des Registers \$s in \$d
sge \$s	„Set greater or equal than“	Kopiert den Inhalt des Registers \$s in LO
sgt \$s	„Set greater than“	Kopiert den Inhalt des Registers \$s in LO

```

1 || #Define subu (%d,%s,%c)
2 || adiu d, s, -c

```

Das Resultat des Programmierens ist eine Datei, aus der der MIPS-Assembler ein Maschinenprogramm erzeugt. Dabei unterstützen Marken und Direktiven die Steuerung des Assemblers durch den Programmierer. In dieser Datei können auch leere Zeilen oder Zeilen mit ausschließlich Kommentaren stehen. Derartige Zeilen heißen Nullzeilen und können genutzt werden, ein Programm übersichtlich zu strukturieren.

7.5 C-Schablonen für eine universelle Registermaschine

Um sich nicht immer neu überlegen zu müssen, wie Elemente oder Primitive einer Hochsprache übersetzt werden sollen, werden im folgenden Kapitel die wichtigsten Elemente der Sprache C und ihre Umsetzung in MIPS-Assembler-Code diskutiert.

7.5.1 Grundblockgraph

Grundblockgraph

Hochsprachenschablonen lassen sich kategorisieren. Jedes Quellprogramm hat eine bestimmte Struktur, die korrekt zu übersetzen ist. Im Compilerbau hat sich das Prinzip des Grundblockgraphen etabliert. Der Grundblock- oder auch Kontrollflussgraph beschreibt oder modelliert den Kontrollfluss, den Ablauf eines Programms. Dabei werden bestimmte Sprachelemente der Hochsprache in bedingte Verzweigungen oder Schleifen mit Bedingungen übersetzt. Im Folgenden wird daher zwischen Programmschablonen für Kontrollstrukturen, also dem Graphen und seinen Grundblöcken und den Teilen mit verzweigungslosen Befehlsfolgen unterschieden. Der Code innerhalb eines Grundblocks kann mit einem Datenflussgraph beschrieben werden. Für die Kontrollstrukturen, aber auch für übergeordnete Datenstrukturen sind komplexe Schablonen notwendig. Ergänzt wird das Modell noch durch Unterprogramme. Dabei wird unterschieden zwischen Unterprogrammen, die rekursiv aufgerufen werden können, also wieder andere Unterprogramme beinhalten können (engl. „non-leaf subroutine“), und Routinen, die dies nicht können (engl. „leaf subroutine“). In einem Grundblockgraphen sind Unterprogramme als Untergraphen dargestellt. Unterprogramme haben in den Hochsprachen unterschiedliche Namen, manchmal auch

mit verschiedener Semantik. In der Sprache C heißen Unterprogramme Funktionen, in Pascal Prozeduren.

Definition 7.5.1 – Grundblock

Ein Grundblock ist eine Folge von Anweisungen zwischen zwei Verzweigungen. Im Grundblock selber stehen keine Befehle für bedingte oder unbedingte Sprünge.

Definition 7.5.2 – Steuerblock

Ein Steuerblock ist ein Grundblock zum Berechnen einer Verzweigungsbedingung.

Definition 7.5.3 – Basisblock

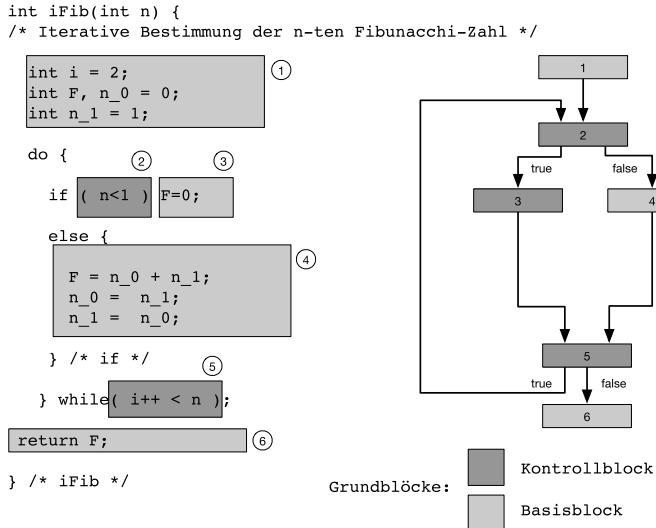
Ein Basisblock ist ein Grundblock, der kein Steuerblock ist.

Definition 7.5.4 – Grundblockgraph

Ein Grundblockgraph ist ein gerichteter bipartiter Graph aus Steuer- und Basisblöcken. Die Kanten des Grundblockgraphen beschreiben den Kontrollfluss des Programms.

► Beispiel 7.5.1 – Grundblockgraph des interaktiven Fibonacci-Programms

Die Abb. 7.11 zeigt den Grundblockgraphen des interaktiven Programms zur Berechnung der Fibonacci-Folge. Jeder Teil des Quellcodes, der nicht verzweigt, bildet einen Grundblock. Die Grundblöcke 2 und 5 sind jeweils Steuerblöcke, da dort Verzweigungsbedingungen ausgerechnet werden, bei den Grundblöcken 1, 3, 4, 6 handelt es sich um Basisblöcke. In dem Programm kann einmal in einer If-Bedingung verzweigt und mit einer While-Schleife iteriert werden. Die entsprechenden Kanten sind abhängig von der jeweiligen Bedingung mit true und false beschriftet, je nachdem, welchem Pfad der Kontrollfluss folgen soll. ◀



■ Abb. 7.11 Grundblockgraph für das Fibonacci-Programm

7.5.2 Kontrollstrukturen

7.5.2.1 Verzweigungen

In der Programmiersprache C sind drei unterschiedliche Verzweigungen definiert. Zum einen die Abfrage

if(Steuerblock){Basisblock},

bei der der Basisblock nur ausgeführt wird, wenn die Bedingung des Steuerblocks erfüllt ist. Dann die Anweisung

if(Steuerblock){BasisblockTrue} else {BasisblockFalse}

bei dem zwei unterschiedliche Basisblöcke ablaufen, je nachdem, ob die Bedingung des Steuerblocks *wahr* oder *falsch* ist. Das dritte Element für Verzweigungen in der Sprache C ist die Auswahl „case-select“. Dabei kann aus einer im Programm statischen, aber beliebigen Auswahl ein Zweig ausgewählt werden. Das rekursive Fibonacci-Unterprogramm nutzt diese Primitive, um die mathematische Definition der Fibonacci-Folge direkt zu implementieren. Alle drei Hochsprachenelemente können mit bedingten und unbedingten Sprüngen implementiert werden.

Der Assembler-Code für das If-Element ist sehr einfach. Wenn die Bedingung des Steuerblocks nicht erfüllt ist, wird einfach über den Code des Basisblocks hinweggesprungen. Der zu der Schablone gehörende Grundblockgraph wird durch die

Label *if:* und *endIf:* dargestellt. Insbesondere das Label *endIf:* ist von großer Bedeutung, fließen doch an dieser Stelle die zwei verschiedenen Kontrollflüsse wieder zusammen. Vor der Verzweigung wird zunächst einmal auf Gleichheit geprüft. Dies geschieht, indem die zu vergleichenden Variablen in zwei Register geschrieben werden – in diesem Fall die Register $\$t_1$ und $\$t_2$.

```
1 || if:    beq    $t1, $t2, endIf    # if($t1 == $t2) {
2 ||     . . .
3 || endIf:           # TRUE basic block
# } /* if */
```

Soll auf Ungleichheit geprüft werden, ist lediglich der Befehl für die bedingte Verzweigung auszutauschen. Über den Code des Basisblocks wird mit *beq* hinweggesprungen, wenn die Bedingung erfüllt ist, wenn also beide Register $\$t_1$ und $\$t_2$ den gleichen Inhalt haben.

```
1 || if:    bne    $t1, $t2, endIf    # if($t1 != $t2) {
2 ||     . . .
3 || endIf:           # TRUE basic block
# } /* if\index{if} */
```

In der MIPS-Architektur vergleichen die Befehle *bne* und *beq* den Inhalt zweier Register direkt. Ist dieser gleich, wird gesprungen, ist er ungleich, wird die Ausführung des Programms fortgesetzt. Soll die Bedingung eine Relation sein, ist es notwendig, andere Befehle zu nutzen. Soll auf größer oder kleiner gleich 0 getestet werden, ist dies mit den Befehlen *bgtz* oder *blez* möglich. Sollen jedoch zwei Variablen verglichen werden, ist es notwendig, vor der Verzweigung den Befehl *slt* oder *sltu* durchzuführen. Diese Befehle schreiben eine 1 in das Zielregister, wenn die Relation erfüllt ist. Durch einen anschließenden Vergleich mit dem *Zero*-Register, also auf die 0 kann dann verzweigt werden. Der dafür notwendige Code ist im Folgenden exemplarisch angegeben. Je nach zu erfüllender Relation müssen die Inhalte der Register entsprechend abgeändert werden.

Verzweigungen

```
1 || if:    sltu    $t3, $t1, $t2      # $t1 < $t2
2 ||     bne    $t3, $zero, endIf    # if(true) {
3 ||     . . .
4 || endIf:           # TRUE basic block
# } /* if-else */
```

Die Primitive if-else benötigt einen weiteren Assembler-Befehl. Die jeweiligen Zweige des Grundblockgraphen werden wieder durch Labels markiert. Im Falle, dass die Bedingung nicht erfüllt ist, wird in dieser Schablone nicht zum Ende des Basisblocks gesprungen, sondern zu der else.-Marke. Dort kann dann der Code eines weiteren Basisblocks stehen. Wird jedoch der If-Zweig ausgeführt, darf der Else-Zweig nicht betreten

werden. Nachdem der If-Zweig durchlaufen wurde, muss daher vor der Else-Marke zum Ende der Verzweigung gesprungen werden. Dies geschieht mit dem Befehl *jump* Adresse. Die durch Label erzeugte Struktur entspricht exakt der des zum Quellprogramm gehörenden Grundblockgraphen. Auch für diese Schablone werden im Folgenden drei unterschiedliche Varianten angegeben, je nachdem, wie die Verzweigungsrelation definiert ist:

```

7
1 | if:      beq    $t1, $t2, else      # if($t1 == $t2) {
2 | . . .
3 | j      endIf                      # }
4 | else:                                # else {
5 | . . .
6 | endIf:                               # } /* if-else */
```



```

1 | if:      bne    $t1, $t2, else      # if($t1 != $t2) {
2 | . . .
3 | j      endIf                      # }
4 | else:                                # else {
5 | . . .
6 | endIf:                               # } /* if-else */
```



```

1 | if:      sltu   $t3, $t1, $t2      # $t1 < $t2
2 | bne    $t3, $zero, else           # if(true) {
3 | . . .
4 | j      endIf                      # }
5 | else:                                # else {
6 | . . .
7 | endIf:                               # } /* if-else */
```

Case-select

Die Semantik des Case-select-Sprachelementes ist etwas komplexer als die bisher diskutierten Verzweigungen. Im Wesentlichen handelt es sich aber um die kompakte Schreibweise mehrerer If-Anweisungen. Allerdings mit einem Unterschied: Die Zustände oder Bedingungen, auf die geprüft wird, dürfen nur Konstanten und nicht wie in den If-Verzweigungen Variablen sein. Und noch eine Besonderheit weißt das Case-select auf: Wird ein Basisblock nicht ausdrücklich durch die Anweisung *break* beendet, wird die nächste Bedingung ebenfalls überprüft. Damit entspricht das Case-select einer Folge von If-Anweisungen und ist nicht identisch mit einer If-else-Struktur. Im Folgenden wird das Case-select einmal mit zugehörigen Break-Anweisungen beschrieben und ein anderes Mal ohne. Mithilfe der Addition mit einer Konstanten wird in das Register $\$0$ die entsprechende Konstante des Select-Zweiges geschrieben. Anschließend wird wieder bedingt verzweigt, und zwar bei Ungleichheit der Bedingung wird zur nächsten Marke gesprungen.

```

1 case:      # assume select variable is $s0
2 sel_0:     addiu   $t0,$zero,0
3          bne    $s0,$t0,sel_1      # ($s0 == 0)
4          . . .
5 sel_1:     addiu   $t0,$zero,1
6          bne    $s0,$t0,sel_2      # ($s0 == 1)
7          . . .
8          addiu   $t0,$zero,n
9 sel_n:     bne    $s0,$t0,default   # ($s0 == n)
10         . . .
11 default:  . . .
12 break:

```

Listing 7.10: Case-Select

Im Gegensatz zur vorangegangenen Schablone wird bei der Verwendung des Schlüsselwortes *break* am Ende des Basisblocks wieder ein Sprung eingefügt. Damit wird dann über die folgenden Select-Abschnitte zum letzten Abschnitt der Struktur gesprungen, und zwar an die Stelle, wo die unterschiedlichen Pfade des Grundblockgraphen zusammenkommen. Um diese Semantik zu implementieren, wird die Marke *break*: eingeführt.

```

1 case:      # assume select variable is $s0
2 sel_0:     addiu   $t0,$zero,0
3          bne    $s0,$t0,sel_1      # ($s0 == 0)
4          . . .
5          j      break            # break
6 sel_1:     addiu   $t0,$zero,1
7          bne    $s0,$t0,sel_2      # ($s0 == 1)
8          . . .
9          j      break            # break
10         . . .
11         addiu   $t0,$zero,n
12 sel_n:     bne    $s0,$t0,default   # ($s0 == n)
13         . . .
14         j      break            # break
15 default:  . . .
16 break:

```

Listing 7.11: Case-Select mit Break

7.5.2.2 Schleifen

Die Semantik der strukturiertesten Do-while-Schleife in C ist einfach: Zunächst wird der durch *do* geklammerte Basisblock ausgeführt. Mit dem Schlüsselwort *while* kann dann eine Bedingung am Ende des Blocks überprüft werden. Ist diese wahr, wird der Basisblock erneut bearbeitet. Im Wesentlichen muss nur die Bedingung im Steuerblock programmiert werden. Die Begrenzungen des Basisblocks werden wieder durch Marken angegeben. Die bedingte Verzweigung springt zu der Marke *do*, wenn die Bedingung der While-Schleife erfüllt ist. Im Fol-

genden findet sich wieder die Diskussion für unterschiedliche Bedingungen und Relationen:

```

1 || do:                                # do {
2 || . . .                               # basic block
3 || while: beq      $t0,$t1,do    # } while ($t0 == $t1)

1 || do:                                # do {
2 || . . .                               # basic block
3 || while: bne      $t0,$t1,do    # } while ($t0 != $t1)

1 || do:                                # do {
2 || . . .                               # basic block
3 || while: sltu     $t2,$t0,$t1
4 ||          beq      $t2,$zero,do# } while ($t0 < $t1)

```

Bei der While-do-Schleife wird die umgekehrte Semantik in C spezifiziert. Dabei wird zunächst die Bedingung überprüft und erst dann der Basisblock ausgeführt. Wenn die Bedingung erfüllt ist, wird der Code im Block durchlaufen, wenn die Bedingung ungültig ist, wird die Folge von Anweisungen ignoriert. Diese Semantik kann durch eine einfache Anpassung der Do-while-Schleife in Assembler implementiert werden. Dazu müssen lediglich die Labels vertauscht werden und die bedingte Verzweigung entsprechend des spezifizierten Verhaltens auf den Befehl *beq* geändert werden. Um die Auswertung des Steuerblocks an den Anfang zusetzen, wird mit einem Sprungbefehl der Basisblock übersprungen. Die Diskussion anderer Verzweigungsrelationen sei dem Leser zur Übung überlassen.

```

1 || while: j      do          # while\index{while} ($t0 == $t1) {
2 || . . .                               # basic block
3 || do:       beq $t0,$t1,while  # } do;

```

Das verbleibende Schleifenelement der Sprache C ist die klassische Zählschleife for:

```

1 || for(i; i < 10; i++) {
2 || /* basic block */
3 ||

```

Listing 7.12: Zählschleife

Diese Schleife wird so lange durchlaufen, bis die Zähl- oder Indexvariable *i* kleiner einer vorgegebenen Zahl ist. Dabei kann die Bedingung vor dem Erhöhen der Laufvariablen wie im Beispiel (*i*++) oder danach (+ + *i*) erfolgen. In MIPS-Assembler kann die Zählschleife unterschiedlich implementiert werden. Beide Varianten werden im Folgenden beschrieben. Dabei wird angenommen, dass die Laufvariable im temporären Register

`$t2` gespeichert ist:

```

1  || for:
2    ||| slti   $t0,$t1,10      # $t0 = (i++ < 10)
3    ||| beq    $t0,$zero,endFor# for ($t0) {
4    ||| ...
5    ||| addiu  $t1,$zero,1      # i++
6    ||| j      for             # }
7  || endFor:

```

Listing 7.13: For-Schleife

Auch die Schablone für die Zählschleife lässt sich ähnlich der Do-while-Schleife implementieren. Dabei wird der auszuführende Basisblock zunächst wieder übersprungen:

```

1  ||| j forEnd
2  || for:                      # for ($t0 < $t1) {
3  ||| ...
4  || forEnd: slti   $t0,$t1,10  # $t0 = (i++ < 0)
5  ||| addiu  $t0,$zero,1       # i++
6  ||| beq    $t0,$t1,for       # }

```

Listing 7.14: Alternative For-Schleife

7.5.3 Datenstrukturen

Die Byte-Reihenfolge (Endianness) kann beim MIPS konfiguriert werden. Dabei wird zwischen der Konfiguration *MIPSel* für little-endian und *MIPSeb* für big-endian unterschieden. Hochsprachencompiler wie beispielsweise C-Compiler sorgen für die Erzeugung des richtigen Codes. Bei der Programmierung in Assembler muss der Programmierer selbst darauf achten, wie er längere Folgen von Bytes speichert. Dies ist insbesondere wichtig, wenn Felder oder Strukturen einer Hochsprache implementiert werden sollen. Der MIPS kann über Bit 15 in seinem Konfigurationsregister in den entsprechenden Modus gesetzt werden. Das Feld heißt *BE*, und wenn es wahr ist, arbeitet der Rechner im big-endian.

Im Datensegment `.data` werden die globalen Variablen eines Programms spezifiziert. Diese können unterschiedliche Datentypen besitzen. Auf der Ebene der Maschine im MIPS-Assembler trifft die Bezeichnung Datentyp nicht zu. Vielmehr handelt es sich um Speicherwörter. Der MIPS ist eine 32-Bit-Architektur. Sowohl Befehlwörter als auch Speicherwörter liegen als 32-Bit-Felder vor. Traditionell ist der Speicher aber byteadressierbar. Das heißt, eine Speicheradresse bezeichnet immer ein Byte. Beim Laden und Speichern kann der MIPS auf Wörter (*lw*, *sw*), Halbwörter (*lh*, *sh*) und Bytes (*lb*, *sb*) zugreifen. Je nach Konfiguration der Endianness wird beim La-

Variablen

den oder Speichern von Halbwörtern oder Bytes in andere Segmente eines Speicherwortes geschrieben oder gelesen. Die Abb. 7.12 und 7.13 geben eine entsprechende Übersicht. Das bedeutet auch, dass der Programmierer im Datensegment des Speichers Variablen als Wort, Halbwort oder Byte spezifizieren kann. Im folgenden Beispiel werden die Veränderlichen *x*, *y* und *z* jeweils als Byte, Halbwort und Wort definiert. Der Assembler legt dann bei der Übersetzung des Codes in Maschinensprache die jeweilige Adresse fest und ersetzt alle Marken, die auf *x*, *y* oder *z* verweisen, durch diese.

```

1 .data
2 X:.byte    2           # Variable\index{Variable} as Byte,
3   initialized with 2
4 y:.half     3           # Variable\index{Variable} as Half Word,
5   initialized with 3
6 z:.word     7           # Variable\index{Variable} as Word,
7   initialized with 7
8 str0:.asciiz "Hello_world" # adding zero to detect end
9 str1:.ascii "Hello_world"  # without end mark

```

Listing 7.15: Definieren von Variablen

a		Adresse							
0:7		Byte							
a	a+1	Adresse							
15:8	7:0	Halbes Wort							
a	a+1	a+2	a+3	Adresse					
15:8	7:0	15:8	7:0		Wort und Fließkommazahl einfache Genauigkeit				
a	a+1	a+2	a+3	a+4	a+5	a+6	a+7	Adresse	
63:57	55:48	47:40	39:32	31:24	23:16	15:8	7:0	Doppel Wort	
								und Fließkommazahl doppelte Genauigkeit	

Abb. 7.12 MIPS-big-endian-Datenstruktur

a		Adresse							
0:7		Byte							
a	a+1	Adresse							
7:0	15:8	Halbes Wort							
a	a+1	a+2	a+3	Adresse					
7:0	15:8	23:16	31:24		Wort und Fließkommazahl einfache Genauigkeit				
a	a+1	a+2	a+3	a+4	a+5	a+6	a+7	Adresse	
7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:57	Doppel Wort	
								und Fließkommazahl doppelte Genauigkeit	

Abb. 7.13 MIPS-little-endian-Datenstruktur

Hochsprachen zeichnen sich dadurch aus, dass sie neben den strukturierten Schleifen zusätzlich komplexe Datentypen definieren. Diese erlauben es, beliebige Datenstrukturen mithilfe symbolischer Bezeichner aufzubauen. Am häufigsten stößt man in C-Programmen auf Felder (engl. „array“) und Strukturen (engl. „struct“).

Felder sind eine lineare Folge von Speicherplätzen, auf die indiziert zugegriffen wird. In der Sprache C lassen sich Felder mithilfe eckiger Klammern spezifizieren; die Definition `int iFeld[10]` reserviert 10 Speicherplätze für Ganzzahlen, die über den Bezeichner `iFeld` angesprochen werden. Der Zugriff erfolgt indiziert. Das folgende kleine C-Schnipsel gibt den Inhalt des Feldes `iFeld` auf dem Bildschirm aus.

```

1 | int i = 0;
2 | int iF0[10] = {0,1,2,3,4,5,6,7,8,9};
3 | int iF1[10] = {9,8,7,6,5,4,3,2,1,0};
4 |
5 | for (i, i++, i<10) {
6 |     printf("Inhalt_von_iFeld:_%i\n", iF0[i];
7 |     printf("Inhalt_von_iFeld:_%i\n", iF1[i];
8 | } /* for (i, i++, i<10) */

```

Listing 7.16: Array

Im MIPS-Assembler können Felder unterschiedlich implementiert werden. Zum einen ist es möglich, einfach einer Marke den Typ *word* zu geben und im Anschluss die Größe des Speicherbereichs anzugeben. Die Direktive `0 : 10` reserviert 10 Speicherplätze, die jeweils mit 0 initialisiert werden. Mit `.space 10` wird einfach ein Speicherbereich von 10 Plätzen reserviert. Die Initialisierung muss vom Programmierer mit einem Codefragment selbst vorgenommen werden. Es ist aber auch erlaubt, mit der `.word`-Direktive den Speicher direkt zu initialisieren. Ein Beispiel ist die Spezifikation von `iF1`. Soll Speicherplatz gespart werden, beispielsweise wenn `uint8`-Variablen in einem Feld gespeichert werden sollen, kann ein Array auch byteweise angelegt werden. Dabei muss der Assembler-Programmierer oder Compilerbauer auf die eingestellte Endianess des Prozessors achten. Am Ende bleibt noch die Frage, wie Zeichenketten (engl. „strings“) zu speichern sind. Eigentlich kann dies auch mit einem Feld von Bytes erledigt werden, da Zeichenketten nichts weiter als eine Folge von Bytes sind. Der MIPS-Assembler bietet dafür zwei spezielle Direktiven an, `ascii` und `asciiz`. Diese unterscheiden sich nur dahingehend, dass `asciiz` an den spezifizierten String noch das Endzeichen `\0` anhängt. Mithilfe dieses Zeichens können Programme immer überprüfen, ob das Ende einer Zeichenkette erreicht ist. Alternativ kann der Programmierer auch die Länge des Arrays definieren und im Datensegment speichern. Die folgenden Co-

Datenstrukturen

Felder

defragmente geben einen Überblick zur Implementierung von Feldern in MIPS-Assembler.

```

1  .data
2  i      .word 0
3  IF0a   .word 0:10
4  IF0b   .space 0:10
5  IF0c   .space 10
6  IF1    .word 9,8,7,6,5,4,3,2,1,0
7
8  IF1el  .byte 9,8,7,6,5,4,3,2,1,0    # little endian
9  iF1eb  .byte 0,1,2,3,4,5,6,7,8,9    # big endian
10
11 str0: .asciiz "Hello_world"        # adding zero to detect end
12 str1: .ascii  "Hello_world"        # without end mark

```

Listing 7.17: Definition eines Felds

7

Zugriff auf Felder

Index

Das obige Codeschnipsel beschreibt ein Feld mit 10 Einträgen vom Typ Integer. Im ersten Moment könnte man denken, dass dies lediglich die Abbildung auf einen linearen Speicher erfordert. In modernen RISC-Architekturen wie dem MIPS ist es jedoch nicht so einfach. Wir erinnern uns, dass der Speicher dieser Architektur zwar in 32-Bit-Worte aufgeteilt ist, jedoch byteweise adressiert wird. Um also im MIPS ein Feld zu implementieren, muss der Index immer mit 4 multipliziert werden.

Um eine solche Datenstruktur der Sprache C oder einer anderen Programmiersprache in MIPS-Assembler zu übersetzen, sind zwei Dinge notwendig. Zum einen muss für jedes Feld eine Basisadresse festgelegt werden. Dies wird im Assembler-Quelltext wieder mit einer Marke erreicht. Der Zugriff auf das Feld kann nun nicht mehr direkt erfolgen, sondern der Index des Elementes muss erst in eine Adresse umgewandelt werden. Zu diesem Zweck definieren wir uns im folgenden Code drei Makros für den Assembler. Das erste Makro berechnet den Index für einen Zugriff auf ein Wortfeld. Das zweite Makro benutzt dieses Makro zur Berechnung des Indexes und liest die entsprechende Variable aus dem Feld. Das dritte Makro nutzt auch das Indexberechnungsmakro, um einen Wert im Array zu speichern.

7.5 · C-Schablonen für eine universelle Registermaschine

```

1  #define cix(%i,%ar)          # compute index
2      la $t0, ar             # set base adress
3
4      sll $t1, i, 2          # compute index
5      adiu $t1, $t1, $t0     # access do index i
6
7  #define lwIx(%v,%I,%a)      # load variable from array at index
8      cix(i,a)              # compute index
9      lw v, 0($t1)
10
11 #define swIx(%v,%I,%a)      # store at variable from array at
12      cix(i,a)              # index
13      sw v, 0($t1)
14
15 iArray:.space   10
16 .text
17 .globl main
18 .ent main
19 main:
20 for:
21     # initialize array ar:0,1,2,3,4,5,6,7,8,9
22     li $s0, 0               # i=0
23     slti $t0, $s0,10        # for (i=0, i<10, i++)
24     beq $t0, $zero, endFor # {
25     swIx($s0,$s0,iArray)   # write to array
26     addiu $s0, $zero, 1      # i++
27     j for                  # }
28 endFor: jr
29 .ent main

```

Listing 7.18: Definieren von Feldern

Als Letztes soll noch eine einfache Struktur in C betrachtet werden. Strukturen erlauben es, mehrere Variablen in einem Container zusammenzufassen, um dann strukturiert auf diese zuzugreifen. Man könnte sie auch für die Vorläufer von Klassen halten. Die Struktur *myStruct* besteht aus zwei Integer-Variablen, einem Zeichen, zwei 15-bit-Ganzzahlen, einer Fließkommazahl mit doppelter Genauigkeit und einem Integer.

Strukturen

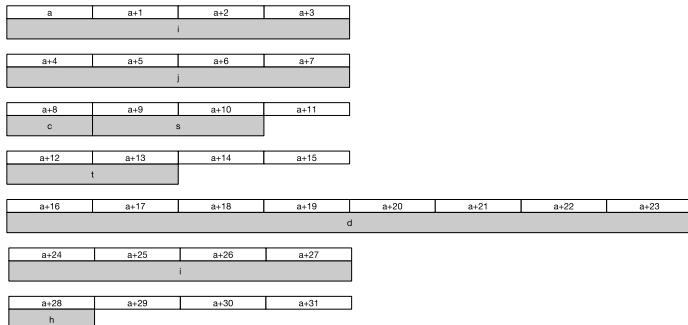
```

1 struct myStruct {
2     int i,j;
3     char c;
4     double d;
5     short s,t;
6     int h;
7 } /* struct */

```

Listing 7.19: Strukture

Eigentlich ist die Übersetzung einer Struktur eine stumpfe Angelegenheit. Man definiert nur eine Variable für jeden Eintrag in die Struktur und codiert den Namen der Struktur mit in den Bezeichner. Um dann ggf. mithilfe eines Zeigers auf diese Struktur zugreifen zu können, wird eine entsprechende Marke spezifiziert. Eine Besonderheit sei noch erwähnt: Würde man



■ Abb. 7.14 Speicherlayout der C-Struktur *myStruct*

für jedes Element der Struktur ein Wort reservieren, ist zwar der Speicherzugriff einfach, es würde aber Speicherplatz verschenkt werden. Aus diesem Grund kann man mit der Direktive *.align 2* Datenelemente an Wortgrenzen platzieren – in einer 32-Bit-MIPS-Architektur also immer an Adressen, die Vielfache von 4 sind. Die Direktiven *.word*, *.half*, *.float* und *.word* führen so ein Alignment automatisch aus. Daher schalten wir mit *.align 0* diese Automatik ab und fügen ein Halbwort gleich hinter einem der Bits ein. Diese Fehlausrichtung (engl. „misalignment“) müssen wir beim Laden der Variablen im späteren Assembler-Code berücksichtigen. Wenn wir die Ausrichtung alleine dem Compiler überlassen, wäre dies selbstverständlich nicht nötig. Die Abb. 7.14 gibt einen Überblick über das so entstandene Layout des Speichers.

```

1 .data
2 a:                      # automatic alignment
3 myStruct:
4 myStruct_i: .word
5 myStruct_j: .word
6 .align 0                 # automatic alignment off
7 myStruct_c: .byte
8 myStruct_s: .halfword
9 .align 2                 # automatic alignment on word boundaries
10 myStruct_t: .halfword
11 myStruct_f: .float
12 myStruct_h: .byte

```

Listing 7.20: Erstellen von Strukturen

7.5.4 Unterprogramme

Im Laufe der Entwicklung von Rechnern wurden die Kosten für die Erstellung von Software zu einem bestimmenden Faktor. War in der Anfangszeit der Computer selbst teuer und

verschlang der Betrieb der Maschine sehr viel Geld, änderte sich dies durch die immer günstiger werdenden Herstellungskosten. Auf der anderen Seite wurden mehr und mehr Prozesse in den Unternehmen durch Programme, also Software, unterstützt. Einmal geschriebene Anwendungen sollten daher wiederverwendet werden. Ein Schritt auf dem Weg war die Einführung der Rechnerarchitektur, also die Möglichkeit, mit einem Befehlssatz Rechner unterschiedlicher Leistungsfähigkeit anzubieten. Auf der Ebene der Software spielt als weiterer wesentlicher Aspekt die Wiederverwendung die entscheidende Rolle. Viele Probleme des Softwareengineering drehen sich darum, wie Programme so strukturiert werden können, dass möglichst wenig Code neu geschrieben werden muss und dieser so oft wie möglich wiederverwendet werden kann. Der erste Schritt auf diesem Weg sind Unterprogrammroutinen. Diese Funktionen oder Prozeduren können aus einem laufenden Programm angesprungen werden. Wie die Definition der Fibonacci-Zahlen nahelegt, ist es erforderlich, dass sich diese Routinen auch selbst aufrufen können. Die Fähigkeit zur Rekursion ist ein wesentlicher Aspekt bei der Implementierung von C-Funktionen.

Um Unterprogramme möglichst einfach wiederverwenden zu können, sind sie so zu programmieren, dass der Programmierer, der eine bereits geschriebene Funktion nutzen möchte, diese einfach nur in sein Programm einbindet. Dazu sollte er die Schnittstelle der Funktion, den Funktionsaufruf kennen und eine grobe Vorstellung vom Verhalten der Funktion besitzen. Funktionen wie die Multiplikation oder das Ziehen einer Wurzel erfüllen diese Eigenschaft auf sehr einfache Art. Bei komplexeren Funktionen ist häufig eine Funktionsbeschreibung notwendig. Aus der Forderung der leichten Wiederverwendbarkeit leitet sich daher ab, dass die Aufgabe, die eine Funktion implementiert, möglichst einfach sein sollte. Daher soll jede Funktion selbst mit wenigen Codezeilen auskommen. Kapseln bedeutet somit auch, dass ein Programm in möglichst viele und kleine Unterprogramme mit vorher klar festgelegtem Verhalten zerlegt werden muss. Dieses Teile-und-herrsche-Prinzip ist ein wesentlicher Punkt des Softwareengineering. Damit eine Funktion Verhalten vor dem Programmierer verbergen kann, ist es notwendig, bestimmte Variablen nur lokal in der Funktion selbst zu kennen. Das heißt, wenn wir in Assembler-Code das Prinzip eines Unterprogramms implementieren wollen, muss die Schablone lokale Variablen unterstützen. Durch die Forderung nach einer klar definierten Schnittstelle müssen mit dem Aufruf einer Funktion auch die Werte dieser Unterprogrammparameter übergeben und zurückgegeben werden.

Kapselung

Funktionsaufruf in C

Bei der Definition der Fibonacci-Folge haben wir bereits ein Unterprogramm eingeführt. Dieses soll von jedem Programmierer genutzt werden können, um die n -te Fibonacci-Zahl zu berechnen. Dazu wird der Funktion die Konstante n übergeben. Für das Unterprogramm ist n keine Konstante, sondern eine Variable. Ebenso kann das aufrufende Programm ein bereits vorher berechnetes Datum übergeben. Dieser oder mehrere Übergabeparameter spezifizieren zusammen mit dem Namen und dem Rückgabeparameter die Schnittstelle der Funktion. In C kann ein Funktionsaufruf beispielsweise so aussehen:

```
1 || int fib(int n)
```

Listing 7.21: Funktionsspezifikation

Der Funktion *fib* (für Fibonacci) wird eine Variable vom Typ einer ganzen Zahl übergeben (engl. „integer“, *int*). Nachdem die Ausführung des Unterprogramms *fib* beendet wurde, gibt die Unterroutine die n -te Fibonacci-Zahl zurück. Das Beispiel verdeutlicht die Kapselung durch das Unterprogramm. Der die Funktion verwendende Programmierer muss nur wissen, dass dieses die n -te Fibonacci-Zahl berechnet. Es interessiert ihn nicht, wie sie das tut – ob nun rekursiv oder iterativ, spielt an dieser Stelle zunächst einmal keine Rolle. Benutzt werden kann die C-Funktion *fib* u. a. mit folgendem Aufruf:

```
1 || int z;
2 || z = fib(5);
```

Listing 7.22: Funktionsaufruf

Zunächst einmal ist im übergeordneten Programm Speicherplatz für die Fibonacci-Zahl zu reservieren. Dieser ist vom Typ Integer. Im Beispiel soll nun die 5. Fibonacci-Zahl berechnet werden.

Aus Sicht des C-Programmierers ist alles einfach. Der Programmierer spezifiziert die Funktion, implementiert sie und kann sie dann im weiteren Verlauf der Programmerstellung wiederverwenden. Doch wie kann ein solches Unterprogramm mit lokalen Variablen und Übergabeparametern in Assembler geschrieben werden? Dazu wird eine Datenstruktur benötigt, die dem Lokalitätsprinzip Rechnung trägt und es gleichzeitig erlaubt, sich die Aufrufreihenfolge zu merken. Diese Struktur ist der Stapel. Ein Stapel funktioniert nach dem Prinzip, dass ein Element oder Datum, das zuerst auf den Stapel gelegt wurde, als Letztes wieder heruntergenommen wird. Dieses FiFo-Verhalten entspricht exakt der gewünschten Funktion von geschachtelten Unterprogrammaufrufen. Der erste Aufruf definiert den untersten Stapeleintrag und jeder weitere wird

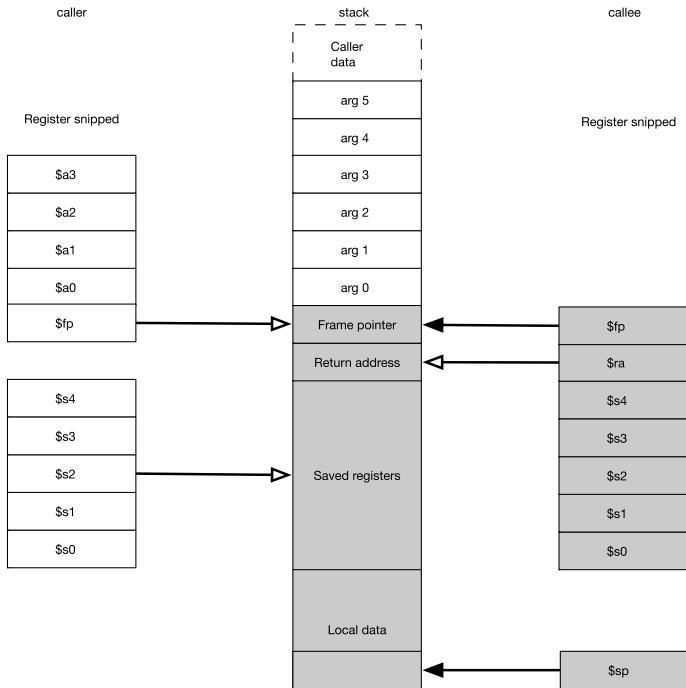
darüber gelegt. Um Funktionen rekursiv aufrufen zu können, muss daher auf dem Stapel Speicherplatz für die Übergabeparameter, die lokalen Variablen und die Rückgabewerte reserviert werden. Immer dann, wenn ein Unterprogramm verlassen wird, werden diese lokalen Parameter automatisch aus dem Speicher entfernt.

Übergabeparameter, lokale Variablen und Rückgabewerte bilden zusammen den Stapel- oder Unterprogrammrahmen (engl. „stack frame“). Um indiziert oder registerrelativ auf diesen zugreifen zu können, ist es notwendig, sich eine Referenz zu merken. Diese heißt Rahmenzeiger (engl. „frame pointer“). In der MIPS-Architektur wird der Stack im Hauptspeicher simuliert. Dazu wird ein Stapelzeiger (engl. „stack pointer“) genanntes Register verwendet. Die in diesem gespeicherte Adresse zeigt immer auf das obere Ende des Staps. Damit kann lokaler Speicher eines Unterprogramms auch lokal auf dem Stapel wachsen und wieder freigegeben werden. Da sich der Inhalt des Stapelzeigers dynamisch mit dem Unterprogrammlauf ändert, aber der Zugriff auf alle anderen Variablen zu jeder Zeit gesichert sein muss, zeigt der Rahmenzeiger immer auf den Anfang des für das jeweils aktive Unterprogramm gültigen Stapelrahmens. Der Rahmenzeiger markiert damit den Beginn des gerade aktiv genutzten Unterprogrammrahmens.

Prinzipiell steht es jedem Programmierer frei, den Stapelrahmen nach Belieben zu gestalten. Für die meisten Rechnerarchitekturen existieren jedoch Konventionen zur Gestaltung des Stapelrahmens.

► Beispiel 7.5.2 – Unterprogrammaufruf des MIPS

Die Abb. 7.15 zeigt beispielhaft einen Stapelrahmen, wie er in einem Assembler-Programm des MIPS vorkommen kann. Die Programmierkonventionen des MIPS legen fest, dass Übergabeparameter eines Unterprogramms in den Registern \$a0–\$a3 abgelegt werden. Diesen symbolischen Namen sind im MIPS-Assembler die Register \$4–\$7 zugeordnet. Sollten mehr als 5 Argumente – daher der symbolische Name der Register – übergeben werden, erfolgt die Übergabe über den Stapel. Die MIPS-Konvention besagt nun, dass die Funktionsargumente, die nicht in diesen Registern stehen, im Bereich der aufrufenden Funktion („Caller“) durch diese gespeichert werden. Die beiden nächsten Einträge auf dem Stapel sind dann für den alten, bereits in den Registern gespeicherten Rahmenzeiger und die Rücksprungadresse des Unterprogramms reserviert. Anschließend wird vom Unterprogramm der Rahmenzeiger auf den neuen Bereich gesetzt. In der MIPS-Architektur sind 5 Register für lokale Variablen reserviert. Die symbolischen Namen sind \$s0–\$s4 und entsprechen den Registern \$16–\$23. Diese Register heißen Saved-Register, und immer wenn ein Unterprogramm (der „Callee“) eines davon nutzen möchte, muss es vorher den Inhalt auf dem Stapel sichern. ◀



■ Abb. 7.15 Definition eines Unterprogrammrahmens in der MIPS-Architektur

Stack-Frame

Das im Beispiel beschriebene Vorgehen deckt aber nur eine Variante zur Implementierung eines Stack-Frames ab. Die Abb. 7.16 zeigt zwei weitere Möglichkeiten, die sich in der Literatur zum MIPS finden. Wurden im vorher beschriebenen Stapelrahmen alle Argumente einer Funktion auf dem Stapel gespeichert, wird in den beiden alternativen Ansätzen darauf verzichtet. In beiden Fällen werden nur Übergabeparameter abgelegt, die nicht in Registern gehalten werden können. In Abb. 7.16a wird aber trotzdem ein Bereich auf dem Stapel reserviert, um die Argumente ggf. doch noch dort speichern zu können; in Abb. 7.16b wird darauf verzichtet.

► Beispiel 7.5.3 – Stack-Frame in MIPS-Assembler

Angenommen ein Unterprogramm Namens *func* ist durch folgende Schnittstelle spezifiziert:

```
1 || int func(int a, int b, int c, int d, int e)
```

Die 5 Variablen *a*, *b*, *c*, *d* und *e* befinden sich anfangs in temporären Registern ($\$t_0 - \t_5). Nehmen wir weiterhin an, die Implementierung des Unterprogramms verwendet alle Saved-Register und dazu noch alle temporären Register. Das Setzen des Stack-Frame im Unterprogramm erfolgt dann mit folgendem Code:

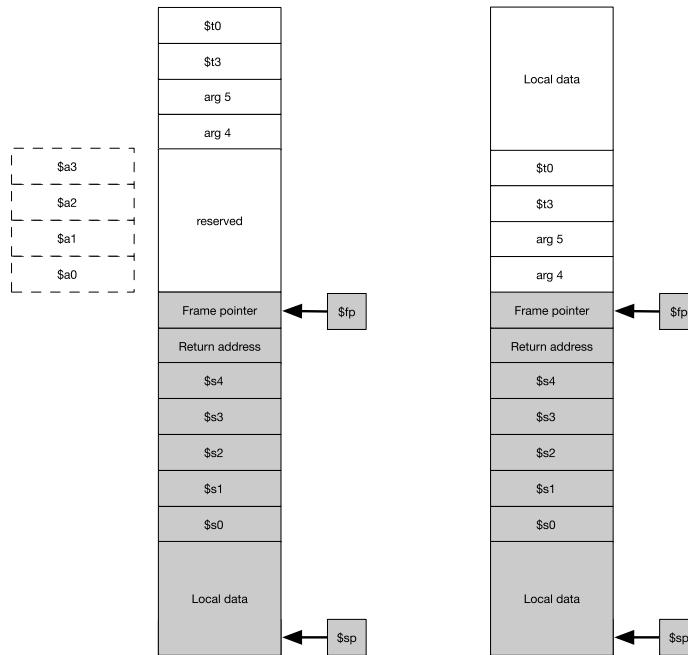


Abb. 7.16 Varianten des MIPS Unterprogrammrahmens

```

1  func:
2      subu    $sp, $sp, 28          # callee stack frame
3      sw     $ra, 24($sp)        # reserve 7*4 bytes
4      sw     $fp, 20($sp)        # save return address
5      sw     $s0, 16($sp)        # save old frame pointer
6      sw     $s1, 12($sp)        # save 5 saved register
7      sw     $s2, 8($sp)
8      sw     $s3, 4($sp)
9      sw     $s4, 0($sp)
10     adiu   $fp, $sp, $zero     # as callee
11     adiu   $fp, $sp, $zero     # set frame pointer

```

Listing 7.23: Stack-Frame

Wenn die Argumente, die bereits in Registern stehen, nicht auf dem Stapel gespeichert werden, müssen nur zwei Parameter in den Stapel geschrieben werden. Wenn die temporären Register nicht zu sichern sind, wird das Unterprogramm mit

```

1  #set argument register
2  addiu   $a0, $t0, $zero       # int a
3  addiu   $a1, $t1, $zero       # int b
4  addiu   $a2, $t2, $zero       # int c
5  addiu   $a3, $t3, $zero       # int d
6
7  #store with argument on stack
8  subu   $sp, $sp, 4            # reserve 4 bytes
9  sw     $t5, 0($sp)           # int e
10    jal    func

```

Listing 7.24: Argumente

aufgerufen. Zunächst werden 28 Byte auf dem Stack reserviert; jeweils 4 Byte für jeden Registereintrag. Dazu wird vom Stapelzeiger die Konstante 28 abgezogen. Dann werden zunächst die Rücksprungadresse und der alte Rahmenzeiger auf dem Stapel abgelegt. Da alle Saved-Register verwendet werden sollen, werden alle auf dem Stapel gespeichert. Um die temporären Register muss sich die aufgerufene Funktion (Callee) nicht kümmern. Gleichtes gilt für die Argumente: Diese müssen von der aufrufenden Funktion (Caller) auf den Stack gelegt werden. Am Ende wird dann der Rahmenzeiger neu gesetzt. Beim Rücksprung aus dem Unterprogramm muss der Rahmen wieder zurückgesetzt werden.

```

1  return:
2    addiu $v0, $t0, $zero      # return $t0
3
4    lw    $ra, 24($sp)        # reset return address
5    lw    $fp, 20($sp)        # reset frame pointer
6    lw    $s0, 16($sp)        # free 5 saved register
7    lw    $s1, 12($sp)        # as callee
8    lw    $s2, 8($sp)
9    lw    $s3, 4($sp)
10   lw    $s4, 0($sp)
11   Addiu $sp, $sp, 28       # free stack
12   jr

```

Listing 7.25: Stack-Frame

Dazu werden alle Register vom Stack wiederhergestellt, nachdem die Rückgabewerte in den Registern \$v0-\$v1 oder \$2-\$3 gesetzt wurden. Wie beim Speichern erfolgt das Laden der Register relativ zum Rahmenzeiger. Am Ende bleiben nur den Stapelzeiger zurückzusetzen und so den Speicher wieder frei zu geben. Mithilfe des Befehls *jr* („jump return“) wird die Ausführung an das aufrufende Programm zurückgegeben. ◀

Gemäß der Konventionen zur Assembler-Programmierung des MIPS muss die aufrufende Funktion – der Caller – die Argumente des Funktionsaufrufs ggf. auf dem Stapel speichern. Sollen temporäre Register über den Unterprogrammaufruf hinaus erhalten bleiben, so ist dies ebenfalls die Aufgabe des Caller. Der Rahmenzeiger und die Saved-Register dagegen werden vom aufgerufenen Unterprogramm (Callee) gesichert und wiederhergestellt. Die Programmskizze in Listing 7.26 fasst den für einen rekursiven Unterprogrammaufruf (engl. „non-leaf subroutine“) notwendigen Assembler-Code noch einmal zusammen.

Manche Unterprogramme rufen keine weiteren auf. Eine anspruchslose Berechnung, beispielsweise eine Multiplikation, wäre ein solches simples Unterprogramm. Diese Blatt-Unterprogramme (engl. „vleaf subroutine“) kommen mit einem wesentlich einfacheren Stapelrahmen aus. So müssen nur die Saved-Register gesichert werden. Eine Veränderung des Rahmenzeigers und ein Sichern der Rücksprungadresse sind bei diesen Unterprogrammen nicht notwendig (Listing 7.27).

7.5 · C-Schablonen für eine universelle Registermaschine

```

1 func:      # callee stack frame
2   subu    $sp, $sp, n          # reserve n bytes
3   sw      $ra, n-4*m($sp)    # save return address
4   sw      $fp, n-4*(m-1)($sp) # save old frame pointer
5   sw      $s0, n-4*(m-2)($sp) # save m callee register
6   .
7   adiu    $fp, $sp, $zero     # as callee
8   .           # set frame pointer
9
10  # function body: any other code
11  subu    $sp, $sp, n          # reserve n bytes on
12  .           # stack
13  sw      $t0, n-4($sp)       # store m temp register
14  .           # as caller
15  sw      $a0, n-4*m($sp)    # store arguments
16  .
17  jal     func               # call me again
18
19  lw      $t0, n-4($sp)       # restore temporaries
20  .           # as caller
21  lw      $a0, n-4*m($sp)    # restore arguments
22  adiu    $sp, $sp, n          # free stack
23
24  addu   $v0, $reg, $zero     # set return values
25  .
26
27 return: # stack frame and return code
28   lw      $ra, n-4*m($sp)    # restore return address
29   lw      $fp, n-4*(m-1)($sp) # restore frame pointer
30   lw      $s0, n-4*(m-2)($sp) # restore saved register
31   .
32   addiu  $sp, $sp, n          # free stack space
33   jr      # return from subroutine

```

Listing 7.26: Rekursiven Unterprogrammaufruf

```

1 func:      # callee stack frame (leaf)
2   subu    $sp, $sp, n          # reserve n bytes
3   sw      $s0, n-4*(m-2)($sp) # save m callee register
4   .
5
6   # function body: any other code
7
8   addu   v0, $reg, $zero      # set return values
9   .
10
11 return: # stack frame and return code
12   lw      $s0, n-4*(m-2)($sp) # restore saved register
13   .
14   addiu  $sp, $sp, n          # free stack space
15   jr      # return leaf subroutine

```

Listing 7.27: Programmskizze eines Blatt-Unterprogramms

7.6 Fibonacci in Assembler

Mit den in den vorangegangenen Abschnitten spezifizierten Assembler-Schablonen können die zwei in C programmierten Funktionen zur Berechnung der Fibonacci-Zahlen in für den MIPS geschriebenen Assembler-Code übersetzt werden.

```

1 iFib:      # callee stack frame (leaf)
2    subu $sp, $sp, 16      # reserve 16 stack bytes
3    sw $s0, 3*4($sp)      # store needed registers
4    sw $s1, 2*4($sp)      # align to MIPS word
5    sw $s2, 1*4($sp)
6    sw $s3, 0*4($sp)

7
8    # initialize local variables
9    addi $s0, $zero, 0      # int F = 0;
10   addi $s1, $zero, 0      # int x = 0;
11   addi $s2, $zero, 1      # int y = 1;
12   addi $s3, $zero, 2      # int i = 2;

13
14    # compute the nth Fibonacci number
15 do:          # do {
16 if:
17     slti $t1, %a0, 1      # n<1
18     bne $t1, $zero, _else  # if (n<1) {
19     add $s0, $zero, $zero  # F=0;
20     j endIf               # }
21 else:        # else {
22     add $s0, $s1, $s2      # F = x + y;
23     add $s1, $s2, $zero    # x = y;
24     add $s2, $s0, $zero    # y = F;} /* else */
25 endIf:
26 while:
27     sltu $t1, $s3, $a0     # i < n
28     addi $s3, $s3, 1        # i++
29     bne $t1, $zero, do      # } while (i < n)

30
31 return: # stack frame and return code
32     add  $v, $s0, $zero    # set return value
33     lw   $s0, 3*4($sp)    # restore needed registers
34     lw   $s1, 2*4($sp)    # align to MIPS word
35     lw   $s2, 1*4($sp)
36     lw   $s3, 0*4($sp)
37     addiu $sp, $sp, 16     # free stack
38     jr   $r21, $r21         # return to caller

```

Listing 7.28: Iteratives Fibonacci Unterprogramm

Iteratives Fibonacci-Unterprogramm

Zunächst wird der Stapelrahmen (Stack-Frame) des Unterprogramms *iFib* gesetzt. Da das Unterprogramm die Register $\$s_0 - \s_3 nutzt, sind diese erst einmal auf den Stapel zu legen. Dafür sind 16 Byte zu reservieren. Jeweils 4 Byte für die 4 32-Bit-Register. Rahmenzeiger und Rücksprungadresse sind nicht zu sichern, da es sich um ein Blatt-Unterprogramm handelt. Das Unterprogramm *iFib* ruft keine weiteren Unterprogramme auf. Gemäß des Quellprogramms werden vier lokale Variablen verwendet. Diese werden zunächst mit den spezifizierten Werten initialisiert. Die Berechnung der Fibonacci-Folge bis zu der n -ten Zahl erfolgt dann iterativ mit der Schleife.

blone für die Do-while-Schleife. Diese beiden Fälle werden mit der If-else-Schablone implementiert. Am Ende werden die Fibonacci-Zahl im Rückgaberegister gespeichert und der Stapelrahmen zurückgesetzt.

Das rekursive Fibonacci-Unterprogramm heißt *rFib*. Im Gegensatz zu der interaktiven Version handelt es sich nicht um ein Blatt-Unterprogramm. Die Rücksprungadresse und der Rahmenzeiger sind also zu speichern. Das Case-select kann direkt mit der entsprechenden Schablone implementiert werden. Lediglich im Default-Zweig müssen zwei Aufrufe der Funktion *rFib* programmiert werden. Das Unterprogramm ruft sich also selbst auf. Dazu wird *n* vor dem jeweiligen Aufruf um 1 heruntergezählt. Die beiden Rückgabewerte werden dann am Ende addiert und gleich im Rückgaberegister gespeichert. Der folgende Code implementiert das rekursive Programm zur Berechnung der *n*-ten Fibonacci-Zahl:

Rekursives Fibonacci-Unterprogramm

```

1 rFib:      # callee stack frame
2         subu    $sp, $sp, 12          # reserve 12 bytes
3         sw     $ra, 2*4($sp)        # save return address
4         sw     $fp, 1*4($sp)        # save caller frame pointer
5         sw     $s0, 0*4($sp)        # we use $s0
6         addiu   $fp, $sp, $zero    # set frame pointer
7
8         # compute the nth Fibonacci\index{Fibonacci} number
9         case:      # we select on n=$a0
10        sel_0:
11           addiu   $t0, $zero, 0
12           bne    $a0, $t0, sel_1    # (n == 0)
13           addiu   $v0, $zero, 0
14           j       return        # return 0
15           # we do not return directly
16           # to safe stack frame code
17        sel_1:
18           addiu   $t0, $zero, 1
19           bne    $a0, $t0, _break   # (n == 1)
20           addiu   $v0, $zero, 1
21           j       return        # return 1
22
23        default:   # n > 1
24           subu    $a0, $a0, 1
25           jal     rFib            # rFib(n-1)
26           add    $s0, $v0, $zero    # save return value
27           subu    $a0, $a0, 1
28           jal     rFib            # rFib(n-2)
29           add    $v0, $v0, $s0    # rFib(n-2) + rFib(n-1)
30
31        break:
32
33        return:    # stack frame and return code
34           lw     $ra, 2*4($sp)    # restore return address
35           lw     $fp, 1*4($sp)    # restore frame pointer
36           lw     $s0, 0*4($sp)    # we use $s0
37           addiu  $sp, $sp, 12    # free 16 byte
38           jr      $r31           # return

```

Listing 7.29: Rekursives Fibonacci Unterprogramm

Um das Programm zum Berechnen der Fibonacci-Folge vollständig in den MIPS-Assembler zu übersetzen, benötigen wir noch ein Unterprogramm *printf*. Das im Listing 7.30 gezeigte Unterprogramm ist lediglich eine Skizze, um das prinzipielle Vorgehen zu veranschaulichen. Viele Regeln zur Formatierung der Zeichenketten der originalen C-Bibliotheksfunktion sind nicht implementiert. Das Unterprogramm geht davon aus, dass alle Argumente auf dem Stapel liegen. Der Grund ist, dass die Anzahl der Funktionsparameter unklar ist, da diese von der Zahl der auszugebenden Variablen abhängt. Weiterhin gehen wir davon aus, dass das erste Argument ein Zeiger auf den entsprechenden String ist und das zweite Argument die Anzahl aller weiteren Argumente angibt. Dies wird dann dazu genutzt, im jeweiligen Fall immer das richtige Argument auszugeben. Nach der Initialisierung prüft das Unterprogramm die übergebene Zeichenkette auf Formatierungsanweisungen. Je nachdem, ob ein Tabulator vorliegt oder eine Zahl auszugeben ist, wird die Art des Arguments ausgewertet und dieses dann mit Hilfe eines *syscall* auf den Bildschirm geschrieben. Tabulatoren werden gleich entsprechend ihrer Spezifikation als Character dargestellt. Am Ende fehlt noch die Ausgabe aller Zeichen der übergebenen Zeichenkette. Dies erfolgt nach einer Abfrage, die Formatierungszeichen erkennt. Im Anschluss werden das Zeichen ausgegeben, der Zeiger auf die Zeichenkette erhöht und das nächste Zeichen dahingehend überprüft, ob es sich um ein Formatierungszeichen oder ein weiteres auszugebendes Zeichen handelt.

7.6 · Fibonacci in Assembler

```

1 .text          # begin code segment
2 .globl main    # for gcc/ld linking
3 .ent printf    # for gdb debugging info
4
5 printf: # callee stack frame (leaf)
6     subu $sp, $sp, 8      # reserve 4 stack bytes
7     sw   $s0, 8($sp)      # store needed registers
8     sw   $s1, 4($sp)      # store needed registers
9     # this code is just a draft
10    # it does NOT fulfill the whole printf spec
11    # we assume that a pointer to the string is given
12    # by the first argument locates in $a0. Then we
13    # assume all additional variables in the following
14    # arguments. To support register-relative addressing
15    # we assume all arguments are on the stack, if
16    # we had more than 4 arguments
17     subu $sp, $sp, 8      # reserve 4 stack bytes
18
19     # initialize local variables
20     add  $s0, $a0, $zero # start address of string is $s0
21     add  $s1, $a1, $zero # number of arguments
22     add  $t6, $t6, $zero # number of characters printed
23     sll  $s1, 2           # multiply number of arguments
24     neg  $s1, $s1         # arguments are on the stack
25     # negative number of arguments
26     # we start with the first argument
27     addiu $t7, $fp, $s1   # $t7 pointer on arguments
28     # the first argument is a pointer
29     # to the string, the next one
30     # is the first variable.
31
32 print: # if (str++ = \) lineFormat
33     addiu $t0, $zero, 92 # $t0 = \
34     bne  $s0, $t0, lineF
35     # if (++str = %) numberFormat
36     addiu $t0, $zero, 37 # $t0 = %
37     bne  $s0, $t0, numberF
38     # each others are characters of the string
39     # therefore we print until formatting instruction
40     lw   $a0, $s0          # get character
41     li   $v0, 11           # print character
42     Syscall
43     addiu $s0, $s0, 4      # str++
44     j    break
45
46 numberF: # assume select variable is $s0
47     # this is the actual character
48     addiu $s0, $s0, 4      # str ++ (entrance from %)
49
50 int:   addiu $t0, $zero, 105 # $t0 = i
51     bne $s0, $t0, float
52     lw $a0, ($s1)          # load current argument
53     li $v0, 1               # to print
54     Syscall                # print integer

```

```

55      addiu   $s0, $s0, 4      # str++
56      addiu   $s1, $s1, -4     # arg++
57      j       break
58
59 float:   addiu $t0, $zero, 102    # $t0 = f
60      bne $s0, $t0, double
61      lw   $a0, ($s1)          # load current argument
62      li   $v0, 2              # print float
63      Syscall
64      addiu $s0, $s0, 4      # str++
65      addiu $s1, $s1, -4     # arg++
66      j       break
67
68 double:  addiu $t0, $zero, 100   # $t0 = d
69      bne $s0, $t0, cr
70      lw   $a0, ($s1)          # load current argument
71      li   $v0, 3              # print double
72      syscall
73      addiu $s0, $s0, 4      # str++
74      addiu $s1, $s1, -4     # arg++
75      j       break
76
77 default: j       return
78
79 lineF:   # assume select variable is $s0
80      # this is the actual character
81      addiu $s0, $s0, 4      # str ++ (entrance from \)
82      # next character is a tabulator. It should be printed
83      # directly
84      addiu $a0, $s0, $zero
85      li   $v0, 3              # print CR
86      syscall
87      addiu $s0, $s0, 4      # str++
88
89 break:   add $t6, $t6, $zero    # we print one character
90      j       print
91
92 return:  # stack frame and return code
93      add $v, $t6, $zero      # return number of
94      sw   $s0, 8($sp)         # printed characters
95      sw   $s1, 4($sp)         # clear stack frame
96      addiu $sp, $sp, 8        # free stack
97      jr   $ra                  # return to caller
98 ent.    printf

```

Listing 7.30: Programmskizze printf

Nachdem wir alle Sprachelemente des Assemblers kennengelernt haben und die notwendigen Unterprogramme für unser Fibonacci-Programm geschrieben sind, können wir das fehlende Hauptprogramm aufbauen. Zunächst einmal wird das Datensegment erstellt. Wir benötigen drei Zeichenfolgen (Strings) und spezifizieren diese. Zusätzlich brauchen wir noch die globale Variable *i*. Eigentlich müssten wir diese gar nicht anlegen, da wir *main* als Unterprogramm des Betriebssystems auffassen wollen. Daher könnte *i* lokal in dieser Unterroutine und auch dort nur in einem Register vorgehalten werden. Das wäre aber semantisch nicht korrekt, da ja *i* eine globale Variable

in *main* ist, die alle Unterprogramme kennen sollten. Nachdem die globalen Variablen festgelegt wurden, wird wieder ein Stack-Frame reserviert. Damit ist *main* ein beliebig aufrufbares Unterprogramm. In die Schleife wird nun der Code zur Berechnung der jeweiligen Fibonacci-Zahl und zum Ausgeben dieser hinzugefügt. Gemäß der von uns festgelegten Konvention des *printf*-Unterprogramms legen wir alle Argumente auf den Stapel. Damit kann das Unterprogramm indiziert auf alle auszudruckenden Parameter zugreifen. Zwar ist das in diesem Fall jeweils nur eine Variable, aber die Anzahl der Zeichen in den Strings ist unterschiedlich. Im Wesentlichen handelt es sich immer um den gleichen Code. Zunächst werden die Argumente in die entsprechenden Register geschrieben, und dann wird in das Unterprogramm verzweigt. Lediglich in der Anzahl und Art der Verwaltung der Funktionsparameter unterscheiden sich die drei verwendeten Unterprogramme.

```

1  .data
2  intro:  .asciiz "Berechnung von n Fibonacci-Zahlen \%d:\n"
3  l1:     .asciiz "Rekursiv: \%d"
4  l2:     .ascii  "Iterativ: \%d\n"
5  i:      .word   0
6  .text      # begin code segment
7  .globl main    # for gcc/ld linking
8  .ent main     # for gdb debugging info
9
10 main:   #now we know how to create a stack frame
11    subu   $sp, $sp, 12          # reserve 12 bytes
12    sw     $ra, 12($sp)        # save return address
13    sw     $fp, 8(m-1)($sp)    # save old frame pointer
14    sw     $s0, 4(m-2)($sp)    # save m callee register
15    addiu  $fp, $sp, $zero    # set frame pointer
16    li     $t1, i              # init i
17
18    #print intro
19    add   $s0, $a0, $zero      # store argument
20    la    $a0, intro
21    addiu $a1, $zero, 39
22    add   $a3, $s0, $zero      # new argument to print
23    lw    $a0, 4($fp)         # read n from stack
24    subu $sp, $sp, 12          # put arguments on the stack
25    sw    $a2, 8(sp)
26    sw    $a1, 4(sp)
27    sw    $a0, 0(sp)
28    jal   printf
29    addiu $sp, $sp, 12          # free stack
30    addiu $t1, $zero, 14
31 for:
32    slti  $t0, $s0, 40          # $t0 = (i++ < 10)
33    beq  $t0, $zero, endFor    # for ($t0) {
34
35    #compute the two Fibonacci variants
36    #and print each
37    add $a0, $s0, $zero        # call Fibonacci with n
38    jal rFib
39    la    $a0, 11

```

```

40      addiu   $a1, $zero, 14
41      addiu   $a2, $v0, $zero          # return from rFib is argument
42      subu   $sp, $sp, 12            # put arguments on the stack
43      sw     $a2, 8($sp)
44      sw     $a1, 4($sp)
45      sw     $a0, 0($sp)
46      jal printf
47      addiu   $sp, $sp, 12          # free stack
48
49      add $a0, $s0, $zero          # call Fibonacci
with n
50      jal iFib
51      la    $a0, 12
52      addiu   $a1, $zero, 15
53      addiu   $a2, $v0, $zero          # return from rFib is argument
54      subu   $sp, $sp, 12            # put arguments on the stack
55      sw     $a2, 8($sp)
56      sw     $a1, 4($sp)
57      sw     $a0, 0($sp)
58      jal printf
59      addiu   $sp, $sp, 12          # free stack
60
61      addiu   $t1, $t1, 4          # i++
62      j      for                  # }
63 endFor:
64
65 return: # stack frame and return code
66      lw    $ra, 12($sp)          # restore return address
67      lw    $fp, 8($sp)           # restore frame pointer
68      lw    $s0, 4($sp)           # restore saved register
69      addiu   $sp, $sp, 12          # free stack space
70      jr                    # return from subroutine
71
72 .ent main

```

Listing 7.31: Programmskizze zur Berechnung von n Fibonacci-Zahlen

?

Übungsaufgaben

Aufgabe 7.1 Erklären Sie mit eigenen Worten, was ein Compiler und Assembler sind.

Aufgabe 7.2 Erstellen Sie ein Unterprogramm „fakiterativ“ in C-Code, welches die Fakultät einer Zahl n iterativ über eine Schleife berechnet und zurückgibt.

Aufgabe 7.3 Erstellen Sie ein Unterprogramm „fakrekursiv“ in C-Code, welches die Fakultät einer Zahl n rekursiv berechnet und zurückgibt.

Aufgabe 7.4 In dieser Aufgabe werden Sie sich mit der binären Codierung der MIPS-Befehle befassen und anschließend ein eigenes Assembler-Programm schreiben.

- Schauen Sie sich den folgenden Binärcode an. In diesem finden Sie binär codierte MIPS-Assembler-Befehle. Geben Sie zu jeder Zeile des binär codierten Codes den entsprechenden MIPS-Assembler-Befehl an, und kommentieren Sie dessen Funktion.

7.6 · Fibonacci in Assembler

1	0010	0000	0000	1000	0000	0000	0000	0000	0000
2	0001	0001	0000	0101	0000	0000	0000	0000	0011
3	0000	0001	0010	0110	0100	1000	0010	0000	
4	0010	0001	0000	1000	0000	0000	0000	0001	
5	0000	1000	0001	0000	0000	0000	0000	1111	

Listing 7.32: Binärkode

- b) Schreiben Sie nun ein Programm in MIPS-Assembler-Code, das 2 Zahlen einliest, diese multipliziert und das Ergebnis ausgibt.

Serviceteil

Theoreme – 508

Definitionen – 509

Bildquellenverzeichnis – 512

Literatur – 515

Stichwortverzeichnis – 517

Theoreme

2. Kapitel	Arithmetik: Zahlen und Rechnen
2.3.1	Dezimalbruch als Bruchanteil mit Exponent
2.3.2	Kettenbruchdarstellung $\sqrt{2}$
2.3.3	Nullstellensuche mit dem Newton-Verfahren
2.3.4	Wurzelziehen durch Nullstellensuche
2.3.5	Ableitung des Heron-Verfahrens
2.3.6	Kleinste darstellbare Gleitkommazahl
2.3.7	Größte darstellbare Gleitkommazahl
3. Kapitel	Mathematische Grundlagen digitaler Schaltungen
3.2.1	Teileralgebra
3.2.2	Mengenalgebra
3.2.3	Existenz eines einzigen Komplements
3.2.4	Selbstinverse Abbildung
3.2.5	Absorption des Komplements
3.2.6	Halbordnung in der booleschen Algebra
3.2.7	De Morgansches Gesetz
3.2.8	Shannon-Zerlegung
3.2.9	Peano-Axiome
3.3.1	Die Schaltalgebra ist eine boolesche Algebra
3.3.2	NOR
3.3.3	NAND
4. Kapitel	Digitale Schaltungen
4.1.1	Satz von Kirchhoff

Definitionen

2. Kapitel Arithmetik: Zahlen und Rechnen

- 2.1.1 Natürliche Zahlen
- 2.1.2 Zahl als Ziffernfolge
- 2.1.3 Dezimalsystem
- 2.1.4 Bestimmung der Dezimalzahl aus einer Zahl zur Basis b
- 2.1.5 Dualsystem
- 2.1.6 Hexadezimalsystem
- 2.1.7 Bestimmung einer Zahl zur Basis b aus einer Dezimalzahl
- 2.1.8 Berechnung des Übertrags
- 2.1.9 Berechnung der Summe
- 2.1.10 Berechnung des Übertrags
- 2.1.11 Berechnung der Differenz
- 2.1.12 b-Komplement
- 2.1.13 (b-1)-Komplement
- 2.1.14 Berechnung des Produkts
- 2.1.15 Berechnung der Division
- 2.1.16 Ziffernreihenfolge im Zahlwort
- 2.2.1 Zahlwort für ganze Zahlen
- 2.3.1 Bruchstellen des Dezimalbruchs als Ziffernfolge
- 2.3.2 Dezimalbruch als Ziffernfolge
- 2.3.3 Exponentialschreibweise
- 2.3.4 Kettenbruch
- 2.3.5 Regulärer Kettenbruch
- 2.3.6 Euklidischer Algorithmus
- 2.3.7 Gleitkommazahl, Datentyp Float
- 2.3.8 Standardrundung
- 2.3.9 Rundungsfehler
- 2.3.10 Festkommazahl $UQ(m, n)$
- 2.3.11 Vorzeichenerweiterter Festkommazahl im Format $Q(m, n)$
- 2.3.12 Auflösung
- 2.3.13 Darstellungsbereich
- 2.3.14 Genauigkeit
- 2.3.15 Dynamikbereich
- 2.3.16 Festkommaaddition
- 2.3.17 Festkommasubtraktion
- 2.3.18 Festkommamultiplikation
- 2.3.19 Festkommadivision

3. Kapitel Mathematische Grundlagen digitaler Schaltungen

- 3.2.1 Halbordnung
- 3.2.2 Größte untere Grenze, Infimum
- 3.2.3 Kleinste obere Grenze, Supremum
- 3.2.4 Verband
- 3.2.5 Boolesche Algebra
- 3.2.6 Boolesche Ausdrücke
- 3.2.7 Boolesche Funktion
- 3.2.8 Vollkonjunktion
- 3.2.9 Disjunktive kanonische Normalform (DKNF)
- 3.2.10 Volldisjunktion
- 3.2.11 Konjunktive kanonische Normalform (KKNF)
- 3.2.12 Boolesche Algebra nach Huntington
- 3.3.1 Schaltalgebra
- 3.3.2 Hamming-Distanz
- 3.3.3 Boolescher Raum der Trägermenge $B := \{0, 1\}$

4. Kapitel Digitale Schaltungen

- 4.1.1 Logisches Signal
- 4.1.2 Fermi-Verteilung im thermodynamischen Gleichgewicht
- 4.2.1 CMOS-Logikgatter

5. Kapitel Elemente der Rechnerarchitektur

- 5.2.1 Mealy-Automat
- 5.2.2 Moore-Automat

6. Kapitel Rechnerarchitektur

- 6.0.1 Befehlsatzarchitektur (Amdahl, Brooks, Blaauw)
- 6.0.2 Rechnerarchitektur
- 6.0.3 Befehlssatz
- 6.1.1 Befehl und Programm
- 6.1.2 Opcode
- 6.1.3 Operation
- 6.1.4 Argumente eines Befehls
- 6.1.5 3-Adress-Code (DAC)
- 6.1.6 Befehlswort
- 6.1.7 Befehlsformat
- 6.1.8 Befehlsformat
- 6.1.9 Maschinenprogramm
- 6.1.10 Assembler-Programm
- 6.1.11 Assembler
- 6.1.12 Inhärente Adressierung
- 6.1.13 Unmittelbare Adressierung
- 6.1.14 Registerdirekte Adressierung
- 6.1.15 Speicherdirekte Adressierung
- 6.1.16 Registerindirekte Adressierung
- 6.1.17 Speicherindirekte Adressierung
- 6.1.18 Indizierte Adressierung
- 6.1.19 Registerrelative Adressierung
- 6.1.20 Orthogonaler Befehlssatz

- 6.1.21 MSB 0
- 6.1.22 LSB 0
- 6.1.23 Big-endian
- 6.1.24 Big-endian
- 6.1.25 Ausgerichtetes Datum
- 6.2.1 Halbleiterspeicherzelle
- 6.2.2 Speichermatrix
- 6.2.3 Speicherwort
- 6.2.4 Adresse
- 6.2.5 Linearer Adressraum
- 6.2.6 Haupt- oder Kernspeicher
- 6.2.7 Bank
- 6.2.8 Speicherwort im verschränkten Adressraum
- 6.2.9 Seiten und Seitenrahmen
- 6.2.10 Seitenrahmen
- 6.2.11 Virtueller Adressraum
- 6.2.12 Hintergrundspeicher
- 6.2.13 Segment
- 6.2.14 Speicherblock
- 6.2.15 Tag
- 6.3.1 Harvard-Architektur
- 6.3.2 Princeton- oder von Neumann-Architektur

7. Kapitel Programmieren von Maschinen

- 7.5.1 Grundblock
- 7.5.2 Steuerblock
- 7.5.3 Basisblock
- 7.5.4 Grundblockgraph

Bildquellenverzeichnis

Die in diesem Buch verwendeten Bilder werden nach deutschem Urheberrecht für Zitate §51 UrhG (Stand: 1. Januar 2008) genutzt. Alle genutzten Werke weisen (am Werk selbst) den Autor oder Rechteinhaber aus, gefolgt von der Angabe der Quelle des Bildes. Ergänzt werden diese Informationen im Rahmen dieses Bildquellenverzeichnisses um den ursprünglichen Titel, die Lizenz und den letzten Zugriff auf die Quelle, soweit diese vorhanden oder verfügbar sind. Wir haben intensiv darauf geachtet nur Bilder aus gemeinfreier Quelle zu verwenden. An dieser Stelle sei noch einmal ausdrücklich Horst Zuse und Matthias Schneider für die unentgeltliche Bereitstellung ihrer Bilder gedankt. Bilder, für die keine Lizenz zu bekommen war, haben wir aus dem Buch entfernt. Trotz größter Sorgfalt bei der Auswahl der Bilder und der Überprüfung der Rechte ist nicht auszuschließen, nicht alle Rechteinhaber ausreichend gewürdigt zu haben. Sollte jemand seine Bildrechte verletzt sehen, bitten wir um eine kurze Mail mit Angabe der Bildnummer und einem Nachweis über die bestehenden Bildrechte an frank.slomka@uni-ulm.de.

- Abb. 1.1 Abakus, Pearson Scott Foresman, Wikimedia Commons, Abacus 1 (PSF), Public Domain, 4. November 2020
- Abb. 1.2 John Napier, Samuel Freeman, Wikimedia Commons, John Napier, Public Domain, 3. November 2017
- Abb. 1.4 Wilhelm Schickard, Konrad Melperger, Wikimedia Commons, Wilhelm Schickard, Public Domain, 3. November 2017
- Abb. 1.5 Blaise Pascal, Unbekannter Autor, Wikimedia Commons, Blaise Pascal, Public Domain, 4. November 2020
- Abb. 1.7 Konstruktion einer Rechenmaschine mit Staffelwalze, Franz Reuleaux, ► <http://dingler.culture.hu-berlin.de>, Thomas'sche Rechenmaschine mit Staffelwalze, unbekannt, 4. November 2020
- Abb. 1.6 Gottfried Wilhelm Leibniz, Christoph Bernhard Francke, Wikimedia Commons, Gottfried Wilhelm Leibniz, Public Domain, 3. November 2017
- Abb. 1.8 Curta 1b, Mathias Schneider, ► www.curta-schweiz.ch, Curta 1b, Mathias Schneider, Lizenz erteilt, 11.10.2021
- Abb. 1.9 Charles Babbage, Thomas Dewell Scott, commons.wikimedia.org, Charles Babbage, Public Domain, 4 November 2020
- Abb. 1.10 Isaac Newton, Godfrey Kneller, Barrington Bramley, Wikimedia Commons, Isaac Newton, Public Domain, 4. November 2020
- Abb. 1.12 Computer am JPL der NASA in den 1950er Jahren, Courtesy, NASA/JPL-Caltech, When computers were human, Public Domain, 3. November 2017

- Abb. 1.13 Herman Hollerith, Charles Milton Bell, 1888, Library of Congress Prints, Herman Hollerith, Erste Veröffentlichung vor dem 1.1.1923; commons.wikimedia.org/wiki/ Commons:Licensing, 3. November 2017
- Abb. 1.14 Nachbau der Z3 in den 1960er Jahren, Zuse KG, Horst Zuse, Nachbau der Z3 im Jahr 1961 (Zuse KG), Verwertung für Grundlagen der Rechnerarchitektur mit freundlicher Genehmigung von Horst Zuse, 09. März 2022
- Abb. 1.15 Konrad Zuse, Horst Zuse, Horst Zuse, Konrad Zuse, Verwertung für Grundlagen der Rechnerarchitektur mit freundlicher Genehmigung von Horst Zuse, 3. November 2017
- Abb. 1.16 Enigma, United States Government, Wikimedia Commons, Enigma, Public Domain, 3. November 2017
- Abb. 1.17 Colossus 10 in Bletchley Park, Jack Good, Donald Michie and Geoffrey Timms, 1945, Wikimedia Commons, Wartime photo of Colossus 10, Public Domain, 10. November 2020
- Abb. 1.18 Der erste elektronische Rechner: ENIAC, US Army, Wikimedia Commons, ENIAC, Public Domain, 3. November 2017
- Abb. 1.19 John von Neumann, LANL, LANL.Gov, John von Neumann, ► www.lanl.gov/resources/web-policies/copyright-legal.php; Unless otherwise indicated, this information has been authored by an employee or employees of the Los Alamos National Security, LLC (LANS), operator of the Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this information. The public may copy and use this information without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor LANS makes any warranty, express or implied, or assumes any liability or responsibility for the use of this information. United States Department of Energy, Los Alamos National Laboratory, 10 November 2020
- Abb. 1.20 Samuel Alexander, NIST, Wikimedia Commons, Samuel Alexander, Public Domain, NIST, US Federal Government, specifically an employee of the National Institute of Standards and Technology, under the terms of Title 17, Chapter 1, Section 105 of the US Code., 3. November 2017
- Abb. 1.21 Die Entwicklung einer Speicherzelle: Von der ENIAC bis zum Transistor, U.S. Army Photo, Number 163-12-62, Wikimedia Commons, Women holding parts of the first four Army computers, Public Domain, United States Federal Government, 3. November 2017
- Abb. 1.22 IBM 702 zur Elektronischen Datenverarbeitung, Martin H. Weik, Wikimedia Commons, IBM 702, Public Domain, 3. November 2017
- Abb. 1.23 Elektronische Datenverarbeitung mit der IBM 360, Lothar Schaack, Wikimedia Commons, IBM 360 in der Landwirtschaftsbehörde der USA, Public Domain, US Government, 10. November 2020

- Abb. 1.24 Cray 2 Number Cruncher, NASA, Wikimedia Commons, Cray2, Public Domain, US Federal Government, 3. November 2017
- Abb. 1.25 Columbia Supercomputer in der NASA Advanced Supercomputing Facility, NASA, Wikimedia Commons, Columbia Supercomputer – NASA Advanced Supercomputing Facility, Public Domain, US Federal Government, 3. November 2017
- Abb. 1.26 Apple II, Maury Markowitz, Wikimedia Commons, Apple II, Copyrighted Free Use, 11. November 2020
- Abb. 1.28 Start von Apollo 11, NASA, NASA.gov, Start von Apollo 11, Public Domain, 3. November 2017
- Abb. 1.27 Apollo Missions Simulator, NASA, Archive.org, Apollo Missions Simulator, NASA, US Government, 3. November 2017
- Abb. 1.29 Apollo Navigationsrechner, unbekannt, CalTech.edu, Apollo Navigationsrechner, Public Domain, 3. November 2017
- Abb. 1.30 Jack Kilby, TI, Courtesy of Texas Instruments, Jack Kilby, Zur Verfügung gestellt unter den gegebenen Bedingungen, 3. November 2017
- Abb. 1.31 Gordon E. Moore, CHF photographer, Intel Corporation, Gordon E. Moore, Media assets are free for editorial broadcast, print, online and radio use; they are restricted for use for other purposes. Any use of image assets must credit â€œIntel Corporationâ€ as the copyright holder., 11. November 2020
- Abb. 2.1 Giuseppe Peano, Unbekannt, Wikimedia Commons, Giuseppe Peano, Public Domain, 5. November 2017
- Abb. 3.1 Evariste Galois, Adelaide Pauline Chantelot, Wikimedia Commons, Public Domain
- Abb. 3.6 Augustus De Morgan, Sophia Elizabeth De Morgan, Wikimedia Commons, Public Domain
- Abb. 4.1 Relais offen, redhat, Wikimedia, Wikimedia Commons, Public Domain, 05.11.2017
- Abb. 4.2 James Clerk Maxwell, G. J. Stodart, nach einer Photographie von F. Of Greenock, Wikimedia Commons, Public Domain, 6.12.2017
- Abb. 4.6 Michael Faraday, Probably albumen carte-de-visite by John Watkins, Wikimedia Commons, Public Domain, 6.12.2017
- Abb. 4.10 Robert Kirchhoff, unbekannt, Wikimedia Commons, Public Domain, 6.12.2017
- Abb. 4.13 Julius Lilienfeld, UNBEKANNT, Wikimedia Commons, Julius Lilienfeld, Public Domain
- Abb. 4.14 William Schockley, Chuck Painter, Frei unter Nennung des Autors, Creative Commons Attribution 3.0 Unported

Literatur

- [Her02] W.v. Münch Andreas Schlachetzki. *Integrierte Schaltungen*. Vieweg+Teubner Verlag, 1978. ISBN: 9783662485538.
- [Cer03] Klaus Waldschmidt. *Schaltungen der Datenverarbeitung*. Vieweg+Teubner Verlag, 1980. ISBN: 9783519061083.
- [Den09] Andreas Schlachetzki. *Halbleiter-Elektronik: Grundlagen und moderne Entwicklungen*. Vieweg+Teubner Verlag, 1990. ISBN: 9783519030706.
- [Cer12] Daniel D. Gajski. *Principles of Digital Design*. Pearson, 1996. ISBN: 9780133011449.
- [Dys14] Kurt Hoffmann. *VLSI-Entwurf: Modelle und Schaltungen*. 4. Aufl. De Gruyter Oldenbourg, 1998. ISBN: 9783486597554.
- [Mal13] Christoph Scholl Paul Molitor. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. Stuttgart, Leipzig: Teubner Verlag, Springer Fachmedien, 1999.
- [ZBZ10] Kio Itoh. *VLSI Memory Chip Design*. Springer-Verlag Berlin Heidelberg, 2001. ISBN: 3540678204.
- [Her02] Raul Rojas (Herausgeber). *The First Computers: History and Architectures*. MIT Press, 2002. ISBN: 9780262681377.
- [Bro03] Frank Markham Brown. *Boolean Reasoning: The Logic of Boolean Equations*. 2. Aufl. Dover Books on Mathematics, 2003. ISBN: 9780486427850.
- [Cer03] Paul E. Ceruzzi. *History of Modern Computing*. MIT Press, 2003. ISBN: 9780262532037.
- [FB08] Thomas Frey und Martin Bossert. *Signal- und Systemtheorie*. 2. Auflage. Wiesbaden: Vieweg+Teubner, 2008.
- [Max08] Clive Maxfield. *Bebop to the Boolean Boogie: An Unconventional Guide to Electronics*. 3. Aufl. Newnes, 2008. ISBN: 9781856175074.
- [Den09] Robert Dennhardt. *Die Flipflop-Legende und das Digitale. Eine Vorgeschichte des Digitalcomputers vom Unterbrecherkontakt zur Röhrelelektronik 1837–1945*. Kulturverlag Kadmos, 2009. ISBN: 9783865990747.
- [Ifr10] Georges Ifrah. *Universalgeschichte der Zahlen*. Seghers, 2010. ISBN: 978- 3942048316.
- [Cer12] Paul E. Ceruzzi. *Computing: A Concise History*. The MIT Press, 2012. ISBN: 9780262517676.
- [Kit13] Charles Kittel. *Einführung in die Festkörperphysik*. De Gruyter Oldenbourg, 2013. ISBN: 978-3486597554.
- [Sta13a] William Stallings. *Computer Organization and Architecture: Designing for Performance*. 10. Aufl. Piercon Education Ltd., 2013. ISBN: 9781292096858.
- [Dav17] Andrew Waterman David A. Pettersson. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN: 9780999249116.
- [Vee17] Harry J.M. Veendrick. *Nanometer CMOS ICs: From Basics to ASICs*. Springer International Publishing, 2017. ISBN: 9783319475950.
- [And14] Todd Austin Andrew S. Tanenbaum. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. Pearson Studium ein Imprint von Pearson Deutschland, Auflage: 6. Edition (3. März 2014). ISBN: 9783868942385.
- [Dav13] John L. Hennessy David A. Pettersson. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Mor-

- gan Kaufmann, Auflage: 5. Edition (10. Oktober 2013). ISBN: 9780124077263.
- [Dei08] Oliver Deiser. *Reelle Zahlen: Das klassische Kontinuum und die natürlichen Folgen*. Springer, Auflage: 2 (2. Juni 2008). ISBN: 978-3540793755.
- [Dys14] G. Dyson. *Turings Kathedrale: Die Ursprünge des digitalen Zeitalters*. Propyläen Verlag, Auflage: 3. (30. September 2014). ISBN: 978-3549074534. URL: ► <https://books.google.de/books?id=XH0XBAAQBAJ>.
- [Iwa14] Sebastian Iwanowski. *Diskrete Mathematik mit Grundlagen*. Springer Vieweg, Auflage: 2014 (2. Oktober 2014). ISBN: 978-3658071301.
- [Mal13] Michael S. Malone. *Der Mikroprozessor: Eine ungewöhnliche Biographie*. Springer, Softcover reprint of the original 1st ed. 1996 (21. Januar 2013). ISBN: 9783662065280.
- [RS97] Curtis T. McMullen Robert K. Brayton Gary D. Hachtel und Alberto L. SangiovanniVincenzelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984, 8th re-printing 1997. ISBN: 9781461297840.
- [Sta13b] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson Education Ltd., Auflage: 10. Edition (10. Oktober 2013). ISBN: 9781292096858.
- [Ulr19] Eberhart Gamm Ulrich Tietze Christoph Schenk. Halbleiterschaltungstechnik. Springer Vieweg, Auflage: 16., erw. u. aktualisierte Aufl. 2019 Edition (5. Juli 2019). ISBN: 9783662485538.
- [Wit12] Kurt-Ulrich Witt. *Algebraische Grundlagen der Informatik: Strukturen -- Zahlen -- Verschlüsselung -- Codierung*. Vieweg+Teubner Verlag, Auflage: 3 (20. September 2012). ISBN: 978-3834801203.
- [ZBZ10] K. Zuse, F.L. Bauer und H. Zemanek. *Der Computer -- Mein Lebenswerk*. Springer, Auflage: 5. unveränd. Aufl. 2010 (17. Mai 2010). ISBN: 9783642120961. URL: ► <https://books.google.de/books?id=cMRNElgwLhUC>.

Stichwortverzeichnis

Ätzen, 217, 219, 253

A

Abakus, 3, 4

Addierer, 168, 267, 309, 312–316, 328

Addierwerk, 166, 177, 314

Addition, 4, 6, 8, 14, 17, 38, 42, 44, 47, 55, 58, 60–65, 69–72, 80–82, 85–88, 101, 108–111, 118, 119, 124, 132, 139, 151, 165, 166, 296, 309, 310, 312–314, 316, 345, 348, 423, 465, 482

Adresse, 259, 260, 321, 322, 325, 326, 342, 346, 352, 353, 358, 361–363, 366, 367, 373, 374, 380–382, 384, 386, 387, 390, 391, 393, 394, 396–399, 402, 411, 415, 429, 434, 442, 448, 451, 458–460, 463, 464, 469, 470, 472, 477, 486, 488, 493

Addressierungsart, 354, 356, 367–369, 386, 399, 400, 402, 461

Adressraum, 23, 25, 30, 34, 354, 361, 362, 366, 376, 377, 379, 381–384, 386, 396, 431, 432, 463

Adressraumverschränkung, 381

AGC, 33, 34

Aiken-Code, 127

Algorithmus, 8, 10, 12, 53, 88, 97, 98, 101, 166, 180, 245, 333, 336–338, 342, 349, 368, 401, 428, 429, 436, 443, 447, 454

ALU, 315, 316, 399, 408, 409, 411, 413–416, 418–420, 423, 425–428, 434, 449

American Standard Code for Information

Interchange, 128

Apple, 27, 30–32

Arithmetik, 70, 79, 83, 95, 116, 120, 124, 126, 136, 137, 309

ASCII-Code, 128

Assembler, 343, 347, 350–353, 370, 386, 442, 447, 448, 450, 451, 454, 457, 459, 461, 470, 471, 473–476, 484–487, 492, 504

Assembler-Programm, 343, 369, 386, 441, 442, 493

Assoziativspeicher, 307, 326

Atommodell, 202

Auflösung, 114, 123, 124

Automat, 298–301, 303, 308, 401

B

Bändermodell, 205, 207

Babbage, Charles, 8, 14

Bank, 381

BASIC, 31

Basisblock, 479, 480, 482–485

Basisregister, 387

BCD-Code, 77, 290

BCPL, 25

Bedingter Sprung, 477

Befehlsformat, 340, 347, 367, 370, 400, 414, 468

Befehlphasen, 399, 401, 403, 408, 411–413, 433, 434

Befehlssatz, 24, 278, 334, 336, 338, 340, 348, 349, 354, 355, 363, 367–369, 387, 398, 410, 411, 420, 429, 436, 440, 451, 458, 463, 491

Befehlswort, 339–342, 345, 346, 354, 356, 357, 359, 361, 362, 367, 372, 381, 403, 412, 415, 461, 463, 466

Befehlszähler, 352, 367, 398, 400, 404, 414, 429, 432, 460, 464, 468, 469

Befehlszyklus, 338, 399–401, 403, 408, 411–413, 417, 418, 441, 448, 470

Big-endian, 373

Binärzahl, 52, 82, 127

Bipolartransistor, 27, 37

Boole, George, 144, 151

Boolesche Algebra, 158, 159, 184

Boolesche Ausdrücke, 152

C

C-Programm, 440, 471

Cache, 388–398, 410, 415, 421, 436

CAD, 25, 220

CAM, 326, 390

Carry, 55, 312, 313, 352

CAS, 260, 321–323

CDC, 26, 27

CERN, 26

CISC, 25, 31, 369, 370, 387, 398, 413, 459

CMOS, 273, 274, 284

Code, 126–128, 132, 292, 339, 370, 421, 423, 436, 441–443, 446, 450, 457, 459–461, 471, 472, 474, 475, 478, 480, 481, 484, 488, 491, 499, 503

Codierung, 15, 23, 42, 52, 68, 69, 74, 78, 81, 82, 112–114, 125, 126, 128, 129, 132, 133, 173, 290, 292, 305, 306, 321, 328, 338, 342, 345, 370, 372, 404, 406, 412, 504

Computer, 2, 3, 10, 11, 13–15, 17–36, 42, 68, 81, 120, 125, 128, 131, 136, 138, 164, 171, 189, 262, 309, 335, 336, 340, 363, 369, 375, 376, 378, 382, 385, 398, 400, 422, 428, 430, 434, 440, 441, 448, 490

Coprozessor, 460

CPLD, 281

Curta, 7, 74, 76

D

D-Latch, 247, 248, 286

DEC, 24, 25

Decoder, 319

DECT, 35

Demultiplexer, 293

Dezimalbruch, 91–93, 99, 103, 122

Dezimalsystem, 42, 51, 56, 63, 65, 70, 71, 76, 83, 88, 136
 Dezimalzahl, 51, 53, 54, 57, 61–63, 69, 91, 132, 328
 Diffusion, 206, 217, 226
 Digital signal processor, 35
 Direktive, 471–474, 487, 490
 Division, 6, 7, 9, 14, 17, 47, 53, 66, 67, 70, 76, 81, 88, 93, 97, 98, 102, 125, 309, 348, 416, 424, 426, 464, 465
 DKNF, 167, 272
 DNF, 154
 Dotierung, 207, 208, 214
 Drain, 214–216, 219, 220
 DRAM, 253, 256, 260–262, 321–323, 377, 388
 DSP, 35
 Dualzahl, 54, 77, 78, 133, 184
 Dynamik, 123

E

EBCDIC, 126
 ECL, 27
 ECU, 32, 34
 EEPROM, 251, 255, 265
 Elektron, 190, 202, 205
 Enigma, 16
 EPLD, 281
 EPROM, 251, 265
 ESC, 32
 ESP, 32
 European Organization for Nuclear Research, 26
 Exponentialschreibweise, 94, 95, 124

F

Faraday, Michael, 195
 Feldeffekttransistor, 37, 213
 Feldstärke, 191
 Festkommazahl, 9, 121, 122, 124, 125
 Fibonacci, 50, 443, 444, 492
 FiFo, 318–320, 492
 Fließbandverarbeitung, 13, 398, 403, 404, 413, 416, 417, 422, 433, 434, 459, 468
 Fließbandverarbeitung, 406, 413
 Flipflop, 237, 241–248, 252, 261, 267, 280–282, 295, 297, 317, 318, 320
 FPGA, 265, 281–283, 306
 Funktion, 8, 35, 100, 106, 138, 141, 150, 152–158, 161, 167, 169, 171, 172, 178, 183–185, 198, 201, 212, 214, 215, 221–223, 225, 232, 236, 237, 239, 241, 261, 263, 265, 266, 269, 278, 281, 283, 288–290, 307, 327, 328, 336–340, 366, 374, 400, 401, 406, 423, 445, 446, 454, 459, 460, 466, 471, 474, 491–494, 496, 499
 Funktionsaufruf, 491, 492

G

GaAs, 27
 GAL, 280
 Ganze Zahlen, 81
 Gate, 213–217, 219, 223, 224, 234, 266, 269, 270, 281
 Gatter, 198, 221–223, 225–227, 230–235, 238, 239, 241–245, 248, 249, 263, 264, 266, 267, 270, 277, 278, 312, 328
 Genauigkeit, 9, 10, 95, 107, 108, 113–117, 123, 124, 133, 489
 GFLOPS, 27, 28
 Gleitkommaarithmetik, 13, 113, 114, 117
 Gleitkommazahl, 9, 104–106, 112, 113, 116, 118–120
 Graph, 143, 169–171, 173–175, 196, 479
 Gray-Code, 126, 127, 305, 306, 327
 Grenzfläche, 208, 209
 Grundblock, 479
 Grundblockgraph, 479, 480

H

Halbleiter, 36, 199–201, 204, 206, 207, 211, 214, 224, 253, 263
 Halbleitertechnik, 27, 37, 107, 164, 204, 223, 326, 345, 369, 370
 Halbordnung, 142, 143, 145, 147
 Hamming-Distanz, 169–171, 174, 175, 177, 305
 Harvard, 14
 Hauptprogramm, 443, 471, 502
 Hauptspeicher, 23, 25, 30, 250, 347, 349, 375–379, 384, 385, 388–390, 392, 394–396, 433, 434, 436, 448, 449, 474, 493
 Hexadezimalzahl, 54
 Hintergrundspeicher, 377, 378, 384, 385, 389, 396
 HP, 24
 Huntington, Edward, 159

I

IAS, 18
 IBM, 14, 18, 21–24, 28, 29, 31, 32, 334
 IBM-PC, 31
 If, 352
 Induktivität, 193, 195, 196
 Infimum, 142, 143, 145, 148
 Institute for Advanced Study, 18
 Interlock, 417, 454
 Inverter, 222–225, 237, 238, 244, 247, 248, 252
 Isolator, 203, 211

K

Kapazität, 19, 44, 193, 195, 231, 233, 237, 253, 258
 Kapselung, 492
 Karnaugh-Tafel, 173–175, 177, 305
 Kennlinie, 212, 216, 227, 244
 Kernel, 459

- Kettenbruch, 96--99, 103
 Kilby, Jack, 37, 213
 KNF, 154
 Komplement, 61, 62, 69, 141, 145, 149, 155, 163, 261, 272, 315
 Kondensator, 195, 215, 223, 227--229, 253
 Konflikt, 309, 417, 418, 420
 Kristall, 199, 200, 203--206, 208, 209, 211, 213, 214, 217
- L**
- Ladung, 190--195, 202, 233, 253, 255, 261
 Latch, 248
 Layout, 220, 225, 226, 234, 252, 253, 264, 266, 269, 271, 272, 274, 275, 277, 305, 490
 leaf, 478, 496
 Leiter, 195, 201, 211, 219
 LiFo, 317, 436
 Little-endian, 373, 485
 Loch, 205
 LTE, 35
- M**
- Mainframe, 24, 27, 77
 Maschinenprogramm, 342, 343, 471, 478
 Maschinenzahl, 104, 106, 120
 Maxterm, 168
 Maxwell, James Clerk, 191
 Mealy-Automat, 299
 Medwedew-Automat, 301
 Menge, 29, 43, 49, 66, 89, 90, 93, 131, 136, 139--145, 147, 148, 152, 154, 162, 190, 191, 290, 299, 300, 336, 410
 Mengenalgebra, 146, 148, 162, 163
 MFLOPS, 26, 27
 Mikrocomputer, 34, 283
 Mikroprogramm, 307, 308, 401--403, 423, 426
 Mikroprogrammierung, 278, 306, 307, 401, 404
 Mikroprozessor, 30, 31, 35, 245
 MIMD, 28
 Minimierung, 137, 149, 150, 154, 162, 168, 169, 171, 176, 178, 306
 Minterm, 155, 167, 172, 176--178, 181, 183, 272, 278, 283
 MIPS, 341, 412, 413, 420, 434--436, 440, 454, 456--461, 463, 464, 466--468, 470, 471, 473--476, 478, 481, 484, 485, 487, 488, 490, 493, 494, 496, 498, 500, 504, 505
 MIT, 19, 25
 Moore, Gordon, 37
 Moore-Automat, 300
 MOS, 27, 213--215, 221, 222, 233, 251--255, 257, 272
 MOS-FET, 213
 MOS-Transistor, 237
 MS-DOS, 31
- Multiplexer, 292, 293, 295, 296, 308, 313, 314, 380, 381, 393, 405, 420, 423, 424, 426, 428
 Multiplikation, 3, 4, 6--9, 14, 17, 47, 49, 64, 65, 70, 81, 88, 94, 102, 108--112, 118, 119, 125, 139, 150, 151, 162, 163, 309, 335, 348, 407, 418, 423, 424, 426, 434, 460, 464, 465, 491
- N**
- NAND, 164, 183, 184, 235, 236, 284, 348
 NAND-Gatter, 164, 221, 234, 243, 269
 Napier, John, 4, 8, 92
 NASA, 11, 18, 33, 34
 NB, 25
 NEC, 28
 Netzwerkelement, 194, 195
 Newton, Isaac, 100
 NICHT, 164, 477
 NMOS, 234, 284
 non-leaf, 478, 496
 NOR, 33, 164, 165, 183, 221, 235, 243, 246, 270, 272--274, 276, 284, 465
 Normalisierung, 95, 117
 Noyce, Robert, 37
 NTDS, 26
- O**
- ODER, 164, 183
 Opcode, 338, 339, 342, 346, 355--357, 361, 362, 368, 400, 401, 414, 458, 463, 468, 476
 Operandenadressierung, 337, 354
 Operation, 7, 26, 55, 69, 70, 101, 119, 140, 145, 147, 151, 154, 176, 181, 256, 296, 315, 338, 339, 343, 345, 348, 349, 353, 355--357, 359, 361, 367, 369, 398, 400, 407, 414, 416--418, 423, 426, 450, 461--463, 465
 Ordnungsrelation, 142, 144, 147, 148, 151, 162
 Oxid, 219
- P**
- PAL, 280
 Pascal, Blaise, 6
 PC, 24, 31, 32, 456
 PDP, 24, 25
 PDP-1, 19
 PFLOPS, 28, 29
 PIA, 281
 Pipelinekonflikt, 417
 PLA, 272--277, 279--281, 292, 301, 304, 306, 307, 401, 404
 PLD, 280
 PMOS, 234, 284
 Prefetching, 394
 Primimplikant, 180, 181, 183
 Programm, 12, 14, 33, 155, 265, 334, 336--339, 342, 343, 349, 354, 355, 362, 369, 371, 375, 377--380,

384--387, 390, 401, 407, 409, 410, 417, 418, 421, 424--426, 428--432, 441--443, 448--450, 453, 457, 463, 471--473, 478--480, 491, 492, 496, 499, 500, 505
PROM, 251, 255

Q

Quadratzahl, 101, 102

R

Rahmenzeiger, 459, 493, 496, 498, 499
RAM, 33, 251, 253, 258, 260, 323, 388, 401
RAS, 260, 321, 322
Rationale Zahlen, 93
Rechenschieber, 9, 36, 107--111
Rechenstäbchen, 4, 5
Rechenuhr, 6
Rechenwerk, 6, 8, 10, 13, 60, 116, 120, 314, 335, 344, 398, 400, 401, 403, 405--407, 417, 418, 426, 428, 429
Rechnerarchitektur, 3, 130, 139, 264, 276, 288, 294, 332--336, 340, 400, 401, 407, 429, 441
Reduktionsabbildung, 106, 107
Register, 18, 21, 23, 256, 294--298, 311, 334, 335, 337, 346, 347, 349--355, 361--363, 366, 367, 396, 400, 403--406, 408, 410, 411, 415--418, 427, 432--434, 436, 451, 453, 454, 457--462, 464, 466--470, 481, 482, 484, 493, 495, 496, 498, 502, 503
Registerindirekte Adressierung, 367
Registermaschine, 347, 411, 413, 416, 422, 430, 432, 433, 435, 440, 441, 451--456
Relais, 10, 12, 14, 76, 138, 160--162, 188, 199, 200, 222
Relation, 141, 142, 144, 148, 468, 477, 481
RISC, 25, 32, 369, 370, 387, 398, 413, 436, 457, 459, 476, 488
ROM, 33, 251, 253--255, 257, 258, 265, 278, 323, 324
Rundungsfehler, 107

S

Schaltalgebra, 137, 138, 140, 147--149, 153, 162--165, 169, 170, 188, 193, 221, 290, 349
Schalter, 11, 12, 37, 137, 138, 161, 162, 164, 188, 197, 199, 200, 222, 223, 233, 257, 261, 264, 272, 273, 298
Schaltplan, 138, 233
Schaltung, 29, 37, 137, 138, 148, 151, 154, 166, 193, 197, 200, 212--215, 217, 220--227, 229, 230, 232--234, 237--240, 242--248, 252, 253, 255, 256, 258, 261--265, 267, 269--271, 275--277, 280, 281, 283, 284, 288--290, 293, 295--298, 301--303, 306, 309--313, 319, 324, 325, 375, 380, 389, 396, 401, 404
Schaltwerk, 298
Schickard, Wilhelm, 6
Schieberegister, 267, 295, 317--320, 328, 345, 395, 396
SEAC, 18
Segment, 290, 386, 432, 459, 474
7-Segment-Anzeige, 327

Segmentierung, 386, 387
Seitenauslagerung, 385
Seitenrahmen, 383--385, 389, 396
Seitenverwaltung, 378, 382, 389, 395, 396, 398
Shannon-Zerlegung, 156, 168, 171
Signal, 164, 197, 198, 212, 223, 230, 239, 242, 246, 258--260, 295, 317--320, 323, 326
Silizium, 204--206, 213, 215, 217--219, 225, 266, 369, 457
SIMD, 26, 407
SoC, 35
Source, 213--216, 219, 220
Spannung, 189--198, 205, 209--213, 215--217, 222, 227, 228, 230, 255, 261
Spannungsquelle, 161, 196, 209, 223
Speicher, 6, 8, 14, 19--23, 25, 31, 33, 34, 130, 131, 249--251, 257, 260, 265, 270, 272, 277, 278, 280--282, 288, 306--308, 317--327, 334, 335, 337--340, 344--348, 354, 355, 358, 361--363, 370, 373--380, 382--389, 391--393, 395--401, 403, 405--411, 416, 417, 422, 424--426, 428--434, 436, 442, 449--451, 459, 466, 467, 470--472, 474, 485, 487, 488, 493, 496
Speicherbefehle, 349
Speicherblock, 256, 381, 389, 391, 392
Speicherfeld, 256, 375
Speicherhierarchie, 380, 388
Speichermatrix, 256, 277, 278, 317, 319, 322, 324--327, 375, 381, 390
Speicherstruktur, 380
Speicherwort, 357--359, 361, 362, 380, 382, 384, 389, 392--395, 474
Speicherzelle, 251--253, 256, 261, 325, 326, 346, 349, 363, 371, 432
Sprung, 14, 366, 367, 413, 421, 463, 477, 483
Sprungvorhersage, 416, 421
Spule, 188, 195, 199, 222, 223
SRAM, 252, 253, 256, 258--261, 265, 278, 279, 294, 321--323, 326, 377, 388
SSEC, 18
Stack, 345, 353, 430, 436, 473, 493, 496
Staffelwalzmaschine, 6, 7, 74, 136
Stapel, 12, 308, 317--320, 345, 346, 349, 352, 355, 369, 374, 375, 379, 386, 403--405, 422--426, 428--432, 448, 449, 471, 492--496, 498, 500
Stapelmaschine, 346, 422, 424--429, 432, 448, 449
Stapelzeiger, 345, 375, 400, 405, 429--431, 459, 471, 493, 496
Stellenwertsystem, 42, 46, 49--52, 55, 58, 59, 61, 62, 64, 68, 70, 78, 79, 84, 85, 87, 88, 137, 165, 309
Steuerblock, 479, 483
Steuerwerk, 8, 298, 306, 400--402, 404--406, 421--423, 426, 428, 429, 432
Strom, 37, 190, 192--197, 199, 201, 203--206, 209, 211--216, 222--225, 227, 258
Stromfluss, 270
Stromquelle, 196

Summe, 4–6, 44, 55–57, 60, 61, 65, 69–71, 80, 106, 115, 116, 118, 165, 166, 172, 176, 177, 196, 221, 309–311, 313, 334, 409, 461
Supercomputer, 26–29, 38
Supremum, 143, 145, 148

T

Tag, 15, 389–391, 394, 397, 398
TFLOPS, 28
TLB, 397, 398, 460
TRADIC, 19
Transistor, 39, 200, 214, 216, 219, 220, 223–225, 227, 234, 252–255, 257, 258, 273
Transportbefehle, 346, 347, 349, 410, 412, 467
Trennzeichen, 92, 103, 124
TTL, 27
Turing, Alan, 15, 16
TX-0, 19, 24

U

Überlauf, 69, 85, 119, 351, 423
UMTS, 35
UND, 164, 183, 328
UNIVAC, 15, 20, 21
UNIX, 25
Unterprogramm, 308, 443, 459, 471–474, 492–496, 498–500, 502–504

V

Variable, 153–156, 161, 162, 168, 170, 172, 174, 177, 178, 181, 236, 272, 291, 417, 418, 445, 474, 488, 489, 492, 502
VAX-Architektur, 25
VMS, 25
Volldisjunktion, 156
Vollkonjunktion, 155

W

Wahrheitstabelle, 153, 154, 166, 168, 178, 184, 185, 286, 289–291, 327
Warteschlange, 319, 320
Wertebereich, 69, 81–83, 86, 88, 95
Widerstand, 162, 194, 196, 204, 211, 222–225, 227, 229
Wurzel, 98, 99, 101, 102, 110, 116, 122, 491

Z

Zähler, 7, 96, 109, 296, 302, 307, 308, 315, 319
Zahlenstrahl, 55, 70, 71
Zelle, 251–254, 258, 266, 267, 269, 270, 272, 281, 282, 306, 318, 324–326, 396
Ziffernfolge, 49, 50, 65, 67, 83, 90–95, 103, 108, 109, 371, 372
Zuse Z4, 20
Zuse, Konrad, 10, 13, 14, 138, 189, 200, 326