

Thomas Flik

Mikroprozessortechnik und Rechnerstrukturen

Thomas Flik

Mikroprozessortechnik und Rechnerstrukturen

7., neu bearbeitete Auflage

Unter Mitwirkung von H. Liebig und M. Menge

Mit 303 Abbildungen



Springer

Dr.-Ing. Thomas Flik
Technische Universität Berlin
Institut für Technische Informatik und Mikroelektronik
Fakultät IV, Elektrotechnik und Informatik
Franklinstraße 28/29
10587 Berlin
flik@cs.tu-berlin.de

Die 6. Auflage ist unter dem Titel „Mikroprozessortechnik“ erschienen

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <<http://dnb.ddb.de>> abrufbar.

ISBN 3-540-22270-7 Springer Berlin Heidelberg New York

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgegesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media

springer.de

© Springer-Verlag Berlin Heidelberg 1976, 1984, 1990, 1994, 1998, 2001, and 2005
Printed in Germany

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

Sollte in diesem Werk direkt oder indirekt auf Gesetze, Vorschriften oder Richtlinien (z.B. DIN, VDI, VDE) Bezug genommen oder aus ihnen zitiert worden sein, so kann der Verlag keine Gewähr für die Richtigkeit, Vollständigkeit oder Aktualität übernehmen. Es empfiehlt sich, gegebenenfalls für die eigenen Arbeiten die vollständigen Vorschriften oder Richtlinien in der jeweils gültigen Fassung hinzuzuziehen.

Umschlag-Entwurf: design & production, Heidelberg

Satz: Camera ready Druckvorlage des Autors

Gedruckt auf säurefreiem Papier 7/3020 Rw 5 4 3 2 1 0

Für Ingrid

Vorwort

Das vorliegende Buch baut auf dem in sechs Auflagen erschienenen Standardwerk *Mikroprozessortechnik* auf, jetzt um den Zusatz *Rechnerstrukturen* im Titel ergänzt, womit den inhaltlichen Erweiterungen dieses Buches über die verschiedenen Auflagen hinweg Rechnung getragen wird.

Für die siebente Auflage wurden die ersten vier Kapitel, die das Buch als Lehrbuch charakterisieren, in ihrem Aufbau beibehalten, jedoch im Hinblick auf die modernen, für Parallelarbeit ausgelegten Prozessoren überarbeitet und ergänzt. Umfangreichere Überarbeitungen und Erweiterungen gibt es in den restlichen vier Kapiteln. Hier werden insbesondere die technischen Neuerungen bei heutigen Rechnerstrukturen berücksichtigt, so z.B. die bei den rechnerinternen und peripheren Datenwegen zu beobachtenden Tendenzen: einerseits Verschnellerung paralleler Verbindungen durch Übertragung mehrerer Daten pro Taktschritt, andererseits weg von Bussen und paralleler Übertragung, hin zu Punkt-zu-Punkt-Verbindungen und zur seriellen Übertragung. Wie in der fünften Auflage begonnen, wurde der Schwerpunkt des Buches weiter in Richtung Nachschlagewerk verlagert, was sich im Text durch Kursivschreibung der Begriffe und in einem umfangreichen Sachverzeichnis niederschlägt.

Einen festen Stellenwert nehmen nach wie vor die Prinzipien der Rechnerorganisation mit der für die Mikroprozessor-/Mikrocontrollertechnik charakteristischen Detailtreue auf der Baustein- und Baugruppenebene ein. Darunter fällt auch die für diese Technik notwendige Kombination von Hardware und Software. Außerdem wird der Programmierung dieser Systeme besondere Bedeutung beigemessen, und zwar in der Symbiose von Assembler und C, wie sie heute in der Praxis der Einbettung von Mikroprozessor- und Mikrocontrollersystemen üblich ist.

Die mit vielen Bildern und Tabellen versehene Präsentation des Stoffes soll es Studierenden der einschlägigen Fachrichtungen sowie Entwicklungingenieuren und Anwendern in den verschiedenen technischen Disziplinen ermöglichen, sich in die CISC- und RISC-Mikroprozessortechnik einschließlich ihrer Assembler- und maschinennahen Programmierung einzuarbeiten. Stoffauswahl und Darstellung lehnt sich zwar an die erfolgreichen Mikroprozessorsysteme mit Motorola-, Intel- und SPARC-Prozessoren an, ist aber dennoch nicht an einen bestimmten Maschinentyp gebunden und somit universell. Von Vorteil sind Kenntnisse im Logischen Entwurf digitaler Systeme sowie in höheren Programmiersprachen, insbesondere in C.

Kapitel 1 gibt eine Einführung in die Arbeitsweise, den *Aufbau und die Assemblerprogrammierung von Mikroprozessorsystemen*. Dieses Kapitel kann von Lesern übersprungen werden, denen die traditionellen Grundlagen der Rechnerorganisation und der Assemblerprogrammierung bekannt sind. Lesern, die mit der RISC-Technologie nicht so vertraut sind, sei dennoch die Lektüre von Abschnitt 1.4 empfohlen, da die Kenntnis dieses Abschnitts insbesondere für das Verständnis der Abschnitte 2.2 sowie der heute in diesem Zusammenhang besonders wichtigen Abschnitte 2.3 (Moderne Prozessorarchitekturen mit parallel arbeitenden Funktionseinheiten) und 3.4 (RISC-spezifische Programmierung) vorausgesetzt wird. *Kapitel 2*, mit *Der Mikroprozessor* überschrieben, behandelt die Architekturmerkmale der von CISC- und RISC-Techniken geprägten Mikroprozessoren von heute. Es bildet gleichzeitig die Grundlage für *Kapitel 3*, der *Assemblerprogrammierung mit C-Entsprechungen*, in dem wichtige Programmierungstechniken heutiger Mikroprozessoren-/controller beschrieben werden, wobei zu den exemplarisch gewählten Assembler-Programmbeispielen ihre jeweiligen C-Entsprechungen mit angegeben sind. *Kapitel 4*, die *Maschinennahe Programmierung in C*, ist Programmierungstechniken gewidmet, die im Zusammenwirken von Hoch- und Assemblersprache eine effiziente Programmierung auf hoher Abstraktionsebene erlauben, auch unter Einbeziehung des Betriebssystems.

Kapitel 5 beschreibt *Busse und Systemstrukturen* mit Schwerpunkt auf Ein- und Mehrbusstrukturen sowie Ein- und Mehrprozessorsystemen, auch in der Detailliertheit des Signalflusses zwischen Mikroprozessor/-controller und den einzelnen Systemkomponenten. *Kapitel 6* behandelt die *Speicherorganisation*, insbesondere den Speicheraufbau unter Einbeziehung unterschiedlicher Speicherbausteine, die Techniken des schnellen Speicherzugriffs sowie die Organisation von Caches und von Speicherverwaltungseinheiten. *Kapitel 7* behandelt die *Ein-/Ausgabeorganisation und Rechnerkommunikation* mit der Beschreibung der gebräuchlichen Interface-Techniken: hier werden insbesondere wichtige Schnittstellenvereinbarungen sowie verschiedene Interface-Bausteine und deren Einbeziehung in Mikroprozessor- und Mikrocontrollersysteme detailliert beschrieben. Hinzu kommen die Datenfernübertragung und Rechnernetze. *Kapitel 8* befaßt sich schließlich mit *Ein-/Ausgabesteuereinheiten, peripheren Verbindungen und Hintergrundspeichern* und ergänzt somit die Ausführungen von Kapitel 7. Einen Schwerpunkt bilden hier die noch weit verbreiteten parallelen und die moderneren seriellen Verbindungen zwischen Rechner und Peripherie.

Für die Mitwirkung am vorliegenden Buch bedanke ich mich bei Frau Eveline Homberg für das Zeichnen der Bilder, bei Herrn Dr.-Ing. Irenäus Schoppa für seine Mitarbeit an Kapitel 3 sowie beim Springer-Verlag für die gute Zusammenarbeit. Für spezielle Beiträge gilt mein besonderer Dank Herrn Prof. Dr.-Ing. Hans Liebig (Abschnitte 1.4 und 3.4) und Herrn Dr.-Ing. Matthias Menge (Kapitel 4).

Inhaltsverzeichnis

1 Einführung in den Aufbau und die Programmierung von Mikroprozessorsystemen	1
1.1 Informationsdarstellung	4
1.1.1 Informationseinheiten	5
1.1.2 Zeichen (characters)	6
1.1.3 Hexadezimal- und Oktalcode	7
1.1.4 Ganze Zahlen	8
1.1.5 Gleitpunktzahlen	12
1.1.6 Binärcodierte Dezimalziffern (BCD-Zahlen)	17
1.2 Rechnerstruktur (CISC)	17
1.2.1 Übersicht über die Hardwarekomponenten	18
1.2.2 Busorientierte Systemstruktur	22
1.2.3 Mikroprozessor	23
1.2.4 Speicher	30
1.2.5 Ein-/Ausgabeeinheit	34
1.3 Assemblerprogrammierung (CISC)	35
1.3.1 Programmdarstellung	36
1.3.2 Programmübersetzung (Assemblierung)	41
1.3.3 Programmeingabe und Textausgabe	47
1.4 Der RISC-Mikroprozessor (Autor: H. Liebig)	48
1.4.1 Prozessorstruktur	49
1.4.2 Maschinen-/Assemblerprogrammierung	57
2 Der Mikroprozessor	65
2.1 CISC-Programmiermodell	67
2.1.1 Registersatz und Prozessorstatus	68
2.1.2 Datenformate, Datentypen und Datenzugriff	73
2.1.3 Adressierungsarten und Befehlsformate	76
2.1.4 Befehlssatz	85
2.1.5 Unterbrechungssystem und Betriebsarten	97
2.2 RISC-Programmiermodell	104
2.2.1 Registersatz und Prozessorstatus	105
2.2.2 Datenformate, Datentypen und Datenzugriff	110

2.2.3	Adressierungsarten und Befehlsformate	111
2.2.4	Befehlssatz	114
2.2.5	Unterbrechungssystem und Betriebsarten	120
2.3	Moderne Prozessorarchitekturen mit parallel arbeitenden Funktionseinheiten	121
2.3.1	Coprozessorsysteme	123
2.3.2	Superskalare Prozessoren	125
2.3.3	Konflikte durch Abhangigkeiten	128
2.3.4	VLIW-Prozessoren	143
2.3.5	Superpipelining	147
2.3.6	Sprungvorhersage	149
2.3.7	Sprungvermeidung	154
2.3.8	Vielfadige Verarbeitung (multithreading)	157
3	Assemblerprogrammierung mit C-Entsprechungen	161
3.1	Assemblersprache	162
3.1.1	Algorithmendarstellung	162
3.1.2	Assemblersyntax und Assembleranweisungen	163
3.1.3	Feste und verschiebbare Programme	169
3.1.4	Makrobefehle und bedingte Assemblierung	173
3.2	Programmflusteuerung	176
3.2.1	Programmverzweigungen	177
3.2.2	Programmschleifen	183
3.3	Unterprogrammtechniken	189
3.3.1	Unterprogrammanschlu	190
3.3.2	Parameterbergabe	192
3.3.3	Geschachtelte Unterprogramme	197
3.3.4	Modulare Programmierung	202
3.4	RISC-spezifische Programmierung (Autor: H. Liebig)	203
3.4.1	Lade-/Speichere-Problematik	205
3.4.2	Unterprogrammanschlu	208
3.4.3	Programmunterbrechungen	212
4	Maschinennahe Programmierung in C (Autor: M. Menge)	218
4.1	Abstraktion von der Maschine	220
4.1.1	Abstraktion durch das Betriebssystem	220
4.1.2	Abstraktion durch den Ubersetzer	222
4.2	C-Ubersetzer mit Zuschnitt auf den Prozessor	224
4.2.1	Speicher- und Ein-/Ausgabezugriffe	225
4.2.2	Datentypen	227

4.2.3 Interrupts	228
4.2.4 Nutzung prozessorspezifischer Merkmale	230
4.2.5 Portabilität	231
4.3 Standard-C mit Spezialisierung durch Assemblereinbindungen	235
4.3.1 Inline-Assembler	236
4.3.2 Assemblermodule	238
4.3.3 Startup-Code	243
4.4 C-Programmierung mit Betriebssystemunterstützung	245
4.4.1 Kommerzielle und frei verfügbare Betriebssysteme	245
4.4.2 Entwurf eines einfachen Betriebssystems	251
5 Busse und Systemstrukturen	260
5.1 Systemaufbau und Systemstrukturen	261
5.1.1 Einkarten- und Ein-chip-Systeme	261
5.1.2 Mehrkartensysteme	262
5.1.3 Ein- und Mehrbussysteme	265
5.1.4 Bus- und Übertragungsmerkmale	270
5.1.5 PC-Busse (ISA, EISA, PCI, PCI-X, PCI-Express)	279
5.1.6 PC-Strukturen mit Bridges, Hubs und Switches	287
5.1.7 Mehrprozessorstrukturen (UMA, NUMA)	294
5.1.8 Industrielle Busse (Multibus II, VMEbus)	299
5.2 Adressierung der Systemkomponenten	301
5.2.1 Isolierte und speicherbezogene Adressierung	302
5.2.2 Karten-, Block- und Bausteinanwahl	303
5.2.3 Byte-, Halbwort- und Wortanwahl	305
5.2.4 Big-endian- und Little-endian-Byteanordnung	308
5.2.5 Busankopplung	314
5.3 Datentransportsteuerung	316
5.3.1 Asynchroner und synchroner Bus	316
5.3.2 Schreib- und Lesezyklen	319
5.3.3 Blockbuszyklus	322
5.4 Busarbitration	324
5.4.1 Buszuteilung	326
5.4.2 Systemstrukturen	327
5.5 Interruptsystem und Systemsteuersignale	331
5.5.1 Codierte Interruptanforderungen	332
5.5.2 Uncodierte Interruptanforderungen	336
5.5.3 Besondere Unterbrechungssignale	343
5.6 Der PCI-Local-Bus und sein Nachfolger PCI-X	345
5.6.1 Motivation und technische Daten	345

5.6.2 Bussignale	348
5.6.3 Busoperationen	352
5.6.4 PCI-X-Bus	366
6 Speicherorganisation	371
6.1 Speicheraufbau und Speicherzugriff	371
6.1.1 Funktionsweise von SRAMs und DRAMs.	372
6.1.2 Aufbau einer Speichereinheit	379
6.1.3 Verschränken von Speicherbänken.	383
6.1.4 Überlappen von Buszyklen.	385
6.1.5 Blockbuszyklen aus Prozessor- und Speichersicht.	390
6.1.6 Blockzugriffstechniken bei DRAMs und SRAMs	394
6.2 Caches	404
6.2.1 Systemstrukturen	404
6.2.2 Laden des Cache.	410
6.2.3 Cache-Strukturen	412
6.2.4 Aktualisierungsstrategien und Datenkohärenz	417
6.2.5 Virtuelle und reale Cache-Adressierung.	428
6.3 Speicherverwaltungseinheiten	435
6.3.1 Segmentverwaltung (segmentation)	437
6.3.2 Seitenverwaltung (paging)	442
6.3.3 Mehrstufige Speicherverwaltung	446
6.3.4 Speicherschutz und Speicherstatus.	452
7 Ein-/Ausgabeorganisation und Rechnerkommunikation	455
7.1 Prozessorgesteuerte Ein-/Ausgabe	456
7.1.1 Ein einfacher Interface-Baustein	457
7.1.2 Synchronisationstechniken.	458
7.1.3 Gleichzeitige Bearbeitung mehrerer Ein-/Ausgabevorgänge	464
7.2 Allgemeine Übertragungsmerkmale	466
7.2.1 Punkt-zu-Punkt- und Mehrpunktverbindung	467
7.2.2 Schnittstellen	468
7.2.3 Software- und hardwaregesteuerte Datenübertragung	469
7.2.4 Serielle und parallele Datenübertragung.	470
7.2.5 Synchrone und asynchrone Datenübertragung	470
7.2.6 Simplex-, Halbduplex- und Duplexbetrieb.	471
7.2.7 Übertragungs-, Transfer- und Schrittgeschwindigkeit	472
7.3 Parallele Schnittstellen	472
7.3.1 Universelle Parallel-Schnittstellen	473
7.3.2 Parallel-Interface-Baustein	474

7.3.3 Centronics	481
7.3.4 IEEE 1284: Compatibility-, Nibble- und Byte-Mode.	484
7.3.5 IEEE 1284: Enhanced-Parallel-Port-Mode (EPP)	489
7.3.6 IEEE 1284: Extended-Capability-Port-Mode (ECP)	490
7.4 Serielle Schnittstellen	492
7.4.1 V.24/V.28 (RS-232-C)	494
7.4.2 V.10 (RS-423-A) und V.11 (RS-422-A)	497
7.4.3 RS-485	498
7.4.4 X.21	499
7.5 Asynchron-serielle Datenübertragung	500
7.5.1 Bit- und Zeichensynchronisation	501
7.5.2 Erhöhung der Übertragungssicherheit	502
7.5.3 Asynchron-serieller Interface-Baustein	504
7.6 Synchron-serielle Datenübertragung.	512
7.6.1 Bit- und Zeichensynchronisation	512
7.6.2 BISYNC-, HDLC- und SDLC-Protokoll	513
7.6.3 Synchron-serieller Interface-Baustein	518
7.7 Rechnernetze und Datenfernübertragung	520
7.7.1 Weitverkehrsnetze und lokale Netze	521
7.7.2 Datenfernübertragung.	533
7.7.3 Sicherung der Datenübertragung	538
8 Ein-/Ausgabesteuereinheiten, periphere Verbindungen und Hintergrundspeicher	544
8.1 DMA-Controller und Ein-/Ausgaberechner	545
8.1.1 Direktspeicherzugriff (DMA).	545
8.1.2 DMA-Controller-Baustein	548
8.1.3 Ein-/Ausgabeprozessor	555
8.1.4 Ein-/Ausgaberechner	555
8.2 Periphere Punkt-zu-Punkt-Verbindungen und Peripheriebusse	560
8.2.1 IDE/ATA, ATAPI	562
8.2.2 Serial ATA (SATA)	565
8.2.3 SCSI-Bus	568
8.2.4 Serial Attached SCSI (SAS)	576
8.2.5 FireWire	578
8.2.6 Fibre-Channel	587
8.2.7 Universal Serial Bus	589
8.2.8 IEC-Bus	593
8.3 Hintergrundspeicher	596
8.3.1 Floppy-Disk-Speicher	597

8.3.2 Festplattenspeicher	603
8.3.3 Solid-State-Disk und RAM-Disk	607
8.3.4 Wechselplattenspeicher	608
8.3.5 Optische Plattenspeicher.	608
8.3.6 Magnetooptische Plattenspeicher	613
8.3.7 Magnetbandspeicher	614
Literatur	620
Akronyme	625
Sachverzeichnis	629

1 Einführung in den Aufbau und die Programmierung von Mikroprozessorsystemen

Mikroprozessorsysteme sind universell programmierbare Digitalrechner. Ihre Vorteile liegen in der Miniaturisierung des Systemaufbaus, in den geringen Hardwarekosten und in der Möglichkeit, die Rechnerhardware modular an die Problemstellung anzupassen. Diese Vorteile haben den Mikroprozessorsystemen Anwendungsgebiete geschaffen, die den herkömmlichen Digitalrechnern verschlossen waren oder von Spezialhardware mit hohen Entwicklungs- und Herstellungskosten abgedeckt werden mußte.

Die Mikroprozessortechnik geht zurück auf die Entwicklung der Halbleitertechnologie, die 1948 mit der Erfindung des *Transistors* in den Bell Laboratories ihren Anfang nahm. Sie ermöglichte es, logische Schaltkreise auf Halbleiterplättchen von wenigen mm² Fläche (chips) zu integrieren (integrated circuits, ICs). So gelang es der Firma Fairchild 1959 erstmals, mehrere Transistoren auf einem Chip unterzubringen. Mit fortschreitender Technologie konnten die Integrationsdichte erhöht und die Schaltzeiten verkürzt werden, wodurch sich die Leistungsfähigkeit der Bausteine vergrößerte.

Ende der sechziger Jahre wurden die Logikbausteine in ihrer Funktion immer komplexer, aber gleichzeitig auch immer spezieller, was sie in ihrer Anwendungsbreite mehr und mehr einschränkte. Die amerikanische Firma Datapoint, die sog. intelligente Terminals herstellte, entwickelte 1969 einen einfachen programmierbaren Prozessor zur Terminalsteuerung und beauftragte die beiden Halbleiterfirmen Intel und Texas Instruments, ihn auf einem einzigen Halbleiterchip unterzubringen. Intel gelang zwar die Herstellung des Bausteins; er konnte jedoch wegen zu geringer Verarbeitungsgeschwindigkeit nicht für die ursprünglich geplante Anwendung eingesetzt werden. Intel beschloß daraufhin, diesen Prozessor als programmierbaren Logikbaustein in zwei Versionen mit Datenformaten von 4 und 8 Bit unter den Bezeichnungen Intel 4004 (1971) bzw. Intel 8008 (1972) auf den Markt zu bringen. Damit wurde die Ära der Mikroprozessoren eingeleitet.

In der Folgezeit fanden *4-Bit-* und *8-Bit-Mikroprozessoren* in allen Bereichen der Steuerungs-, der Regelungs- und der Rechentechnik eine weite Verbreitung. Unterstützt wurde diese Entwicklung durch das Erscheinen ganzer Familien von Mikroprozessoren mit einer Vielzahl an Zusatzbausteinen, die den Entwurf von Mikroprozessorsystemen wesentlich erleichterten. Als Schwäche dieser Prozessoren galt jedoch ihre geringe Leistungsfähigkeit bei der Bearbeitung numeri-

scher Probleme. Diesem Nachteil trug man Rechnung, indem man die 8-Bit-Prozessoren um Multiplikations- und Divisionsbefehle sowie um Operationen mit 16-Bit-Operanden erweiterte.

Der erste *16-Bit-Mikroprozessor*, Texas Instruments TMS 9900, kam 1977 auf den Markt und fand mit den nachfolgenden 16-Bit-Mikroprozessoren 8086 von Intel und MC68000 von Motorola in den Jahren 1978 bis 1980 leistungsfähige Konkurrenten. Sie wiesen gegenüber den 8-Bit-Prozessoren eine wesentlich höhere Rechenleistung, aber auch sehr viel komplexere Strukturen auf. Dadurch wurde der Einsatz von Mikroprozessoren in Bereichen der Minicomputer-Anwendungen (personal computer, PC) möglich. Eine weitere Steigerung der Leistungsfähigkeit wurde bei deren Nachfolgemodellen erreicht. Diese sahen zum Teil intern bereits eine 32-Bit-Struktur (MC68000, MC68010) und eine Erweiterung des Adreßraums von üblicherweise 64 Kbyte auf 16 Mbyte vor (MC68000, MC68010, Intel 80286); darüber hinaus wurden Funktionen, wie virtuelle Speicherverwaltung, Direktspeicherzugriff und der Anschluß von Coprozessoren für Gleitpunktarithmetik unterstützt. – Um die Verschiedenheit von Datenbusbreite und interner Verarbeitungsbreite auszudrücken, bezeichnen wird Mikroprozessoren mit externem 16-Bit-Datenbus und interner 32-Bit-Struktur als 16/32-Bit-Prozessoren.

Seit Ende der 1980er Jahre herrschen *32-Bit-Mikroprozessoren*, d.h. Prozessoren mit interner 32-Bit-Verarbeitung vor. Mit ihnen werden Digitalrechner mittlerer und höherer Leistungsfähigkeit aufgebaut, d.h. PCs und Workstations, aber auch Mainframes (Server), d.h. Rechner mit sehr großer Leistungsfähigkeit, die häufig als „symmetrische“ Parallelrechner ausgelegt sind (SMP-Systeme). Man unterscheidet dabei zwei Prozessor-„architekturen“, sog. CISCs (complex instruction set computers) und RISCs (reduced instruction set computers), wobei CISC-Prozessoren – grob gesagt – mit einem umfangreichen Befehlssatz komplexer Befehle und RISC-Prozessoren mit einem demgegenüber reduzierten Befehlssatz elementarer Befehle, aber höherer Parallelität in der Befehlausführung (Fließbandverarbeitung) ausgestattet sind.

Beispiele für 32-Bit-CISCs der ersten Generation sind die Prozessoren Intel i386, i486 und Pentium (letzterer als 64/32-Bit-Prozessor) sowie Motorola MC68020, MC68030 und MC68040. Sie entstanden als Weiterentwicklungen ihrer oben genannten 16-Bit-Vorgänger und sind mit diesen weitgehend kompatibel, so daß Programme von der jeweiligen 16-Bit-Version auf die 32-Bit-Version übertragen werden konnten. Erreicht wurde diese Kompatibilität durch Beibehaltung wesentlicher Merkmale der Prozessorarchitektur. Beispiele für 32-Bit-RISCs der ersten Generation sind der SPARC (Sun Microsystems und andere Hersteller) und die Prozessoren MIPS R3000 (Integrated Device Technology) und Am29000 (Advanced Micro Devices).

Die Erhöhung der Leistungsfähigkeit ergab sich bei den 32-Bit-Prozessoren nicht nur durch die größere Verarbeitungsbreite, sondern auch durch die Erhöhung der

Verarbeitungsgeschwindigkeit und durch Architekturmerkmale, wie sie von den früheren Mainframes bekannt sind. Zu nennen sind: interne Parallelarbeit, interne Pufferung von Befehlen und Operanden (Befehls- und Daten-Caches, heute als L1- und L2-Caches, d.h. als Caches der ersten bzw. zweiten Ebene), Erweiterung des Befehlssatzes und der Adressierungsarten (bei CISCs), Erhöhung der Taktfrequenz als Folge der Fließbandverarbeitung (bei RISCs, inzwischen auch bei CISCs), Unterstützung der virtuellen Speicherverwaltung (memory management units), Coprozessoren, zunächst chip-extern für die Gleitpunktarithmetik, heute chip-intern für diverse Funktionen.

In den 1990er Jahren ging die Entwicklung der 32-Bit-Prozessoren so schnell voran, daß es unmöglich ist, die Entwicklungsstufen mit typischen Prozessoren zu benennen. Andererseits gibt es nur wenige Hochleistungsprozessoren, die eine größere Verbreitung erlangt haben. Bei den CISCs sind das der Pentium-Prozessor (Intel) und sein Konkurrent, der Athlon-Prozessor (AMD), die beide bei PCs eingesetzt werden. Bei den RISCs sind das die PowerPC- und MPC-Prozessoren (IBM, Motorola) und der SPARC-Prozessor (Sun und andere), wie sie bei Macintosh-Computern bzw. bei Unix-Workstations (Sun) zum Einsatz kommen. – Hinzugekommen sind inzwischen die *64-Bit-Mikroprozessoren*, z.B. UltraSPARC (Sun), PowerPC (IBM), Athlon 64, Opteron (beide AMD) und Itanium (Intel).

Diese Prozessoren haben alle eine 32-Bit- bzw. 64-Bit-Internverarbeitung bei in beiden Fällen externem 64-Bit-Datenbus. Der schnelle Zugriff auf einen prozessorexternen Cache erfolgt ggf. über eine zum Prozessorbus zusätzliche Verbindung mit z.B. 128 oder 256 Bit Breite. Charakteristisch für sie ist eine hohe Parallelität in der Befehlsverarbeitung durch Fließbandtechnik mit sehr vielen Stufen und mehreren parallel arbeitenden Fließbändern für unterschiedliche Befehlsgruppen (Superskalarität, Very Long Instruction Word), die spekulative Ausführung von Befehlen bei Programmverzweigungen (branch prediction), das speulative Laden von Daten (data prefetch), der schnelle Zugriff auf Caches der zweiten und ggf. dritten Ebene (L2-/L3-Caches). Hinzu kommt eine sehr hohe Prozessortaktfrequenz, die inzwischen die 3-GHz-Marke überschritten hat und weiter steigt. 64-Bit-Prozessoren zeichnen sich insbesondere durch die bei Servern erforderliche Adressierbarkeit von mehr als 4 Gbyte aus. Diese Bausteinkomplexität drückt sich in bis zu mehreren 100 Millionen Transistorfunktionen aus.

Ein besonderer Schwerpunkt ist derzeit die Entwicklung von Mikroprozessoren mit möglichst geringer Leistungsaufnahme, was allerdings im Widerspruch zu einer hohen Taktfrequenz steht. Dazu werden einerseits die Prozessorstrukturen verkleinert und die Signalspannungen reduziert. Andererseits werden Techniken realisiert, nicht benutzte Funktionsbereiche eines Prozessors vorübergehend in den Ruhezustand zu versetzen. Geringe Leistungsaufnahme führt einerseits zu reduzierter Wärmeentwicklung, was Lüfter überflüssig macht und somit die Geräuschenentwicklung senkt. Andererseits ermöglicht sie längere Arbeitszeiten bei batteriebetriebenen Rechnern.

Trotz dieser stürmischen Entwicklung hin zu Hochleistungsprozessoren sind die weniger leistungsfähigen 8-, 16- und 32-Bit-Prozessoren ein gewichtiger Faktor auf dem Mikroprozessormarkt geblieben, und zwar als kostengünstige Varianten bei Anwendungen mit geringeren Anforderungen an die Rechenleistung. Sie tauchen insbesondere als „Kerne“ von *Mikrocontrollern* auf, d.h. von Steuerungsbausteinen, in die außer der Prozessorfunktion weitere Rechnerfunktionen integriert sind, wie Halbleiterspeicher sowie Ein-/Ausgabefunktionen vielfältiger Art. Sie sind somit quasi Ein-chip-Computer. In dieser Form existiert auch der Motorola-Mikroprozessor MC680x0 weiter, der in der Entwicklungsgeschichte der Hochleistungsprozessoren lange Zeit Konkurrent der Intel-Prozessoren x86 und Pentium war (und in diesem Sektor dann durch den PowerPC-Prozessor ersetzt wurde). Die Mikrocontroller werden im Vergleich zu den Hochleistungsprozessoren mit relativ geringen Taktfrequenzen von 1 bis 33 MHz, teils auch mit bis zu 100 MHz betrieben.

Wir werden uns in diesem Buch mit der Wirkungsweise von Mikroprozessoren sowie dem Systemaufbau mit diesen Prozessoren befassen, wobei wir uns grundsätzlich an 32-Bit-Prozessoren orientieren, die Prozessoren geringerer Verarbeitungsbreite aber nicht aus dem Auge verlieren. So gelten die Erläuterungen meist auch für Mikrocontroller, ohne daß diese extra hervorgehoben sind. – In diesem ersten Kapitel geben wir zunächst einige einführende Erläuterungen: in Abschnitt 1.1 zur Informationsdarstellung, in Abschnitt 1.2 zum Aufbau und zur Arbeitsweise von Mikroprozessorsystemen und in Abschnitt 1.3 zur Programmierung auf der Maschinen- und der Assemblerebene. Als Ausgangspunkt verwenden wir dafür, da leichter verständlich, CISC-Prozessoren. Einen Einblick in die Arbeitsweise und die Maschinen-/Assemblerprogrammierung von RISC-Prozessoren geben wir dann in Abschnitt 1.4.

1.1 Informationsdarstellung

Die Informationsverarbeitung in Mikroprozessorsystemen geschieht im Prinzip durch das Ausführen von Befehlen auf *Operanden* (Rechengrößen). Da Befehle selbst wieder Operanden sein können, z.B. bei der Übersetzung eines Programms (Assemblierung, Compilierung), bezeichnet man Befehle und Operanden gleichermaßen als *Daten*. Ihre Darstellung erfolgt in binärer Form, d.h., die kleinste *Informationseinheit* ist das *Bit* (*binary digit*, Binärziffer). Ein Bit kann zwei Werte annehmen, die mit 0 und 1 bezeichnet werden. Technisch werden diese Werte in unterschiedlicher Weise dargestellt: z.B. durch zwei verschiedene Spannungsspegel auf einer Signalleitung, durch die beiden Übergänge zwischen zwei Signalpegeln, durch den leitenden oder gesperrten Zustand eines Transistors, durch den geladenen oder ungeladenen Zustand eines Kondensators oder durch zwei verschiedene Magnetisierungsrichtungen auf einem magnetisierbaren Informationsträger.

Die Codierung von Daten erfolgt in Informationseinheiten, die aus mehreren Bits bestehen. Die Darstellung von Operanden als Zeichen (Textzeichen), als Binärvektoren (Bitmuster) und als Zahlen unterliegt dabei weitgehend allgemeinen Festlegungen. Die Darstellung von Befehlen ist hingegen prozessorspezifisch.

1.1.1 Informationseinheiten

Die Bitanzahl einer Informationseinheit bestimmt ihr *Datenformat*. Standardformate sind das *Byte* (8 Bit), das *Halbwort* (half word, 16 Bit), das *Wort* (word, 32 Bit) und, unter Einbeziehung der Gleitpunktarithmetik, das *Doppelwort* (double word, 64 Bit). Spezielle Datenformate sind das einzelne Bit, das Halbbyte (nibble, Tetrade, 4 Bit) und das *Bitfeld* (Bitanzahl variabel). – Der Begriff Wort orientiert sich dabei an der Verarbeitungsbreite von 32-Bit-Mikroprozessoren. Bei 16-Bit-Mikroprozessoren wird der Begriff Wort für das 16-Bit-Format verwendet; das 32-Bit-Format wird dann als Doppelwort und das 64-Bit-Format als *Vierfachwort* (quad word) bezeichnet. Allgemein wird der Begriff Wort auch im Zusammenhang mit der Zugriffsbreite eines Speichers (*Speicherwort*) und mit der Codierung von Information (*Codewort*) benutzt.

Bei der Darstellung von Informationseinheiten werden die Bits, mit Null beginnend, von rechts nach links numeriert und ihnen im Hinblick auf Dualzahlen aufsteigende Wertigkeiten zugewiesen (Bild 1-1). Bit 0 wird dementsprechend als niedrigstwertiges Bit (*least significant bit*, LSB), das Bit mit dem höchsten Index als höchstwertiges Bit (*most significant bit*, MSB) bezeichnet.

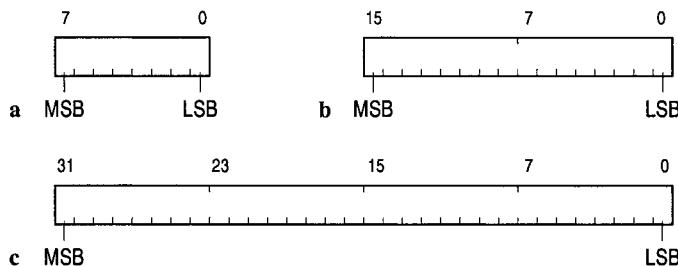


Bild 1-1. Informationseinheiten. a Byte, b Halbwort, c Wort

Zur Angabe der *Speicherkapazität* als Anzahl von Bits, Bytes oder Codewörtern verwendet man in der Informatik in Anlehnung an die Einheitenvorsätze der Physik die Bezeichnungen Kilo (K), Mega (M), Giga (G) und Tera (T), bezogen auf das Dualzahlensystem: $K=2^{10}=1024$, $M=2^{20}=1048\,576$, $G=2^{30}=1\,073\,741\,824$ und $T=2^{40}$. (Bei allen anderen Dimensionsangaben, z. B. von Übertragungsraten, beziehen sich die Bezeichnungen Kilo, Mega und Giga auf das Dezimalzahlsystem. Für Kilo wird dabei die Kurzbezeichnung k verwendet.)

1.1.2 Zeichen (characters)

Die rechnerinterne Darstellung der Schriftzeichen, d.h. der Buchstaben, Ziffern und Sonderzeichen, erfolgt durch *Zeichencodes*. In der Mikroprozessortechnik gebräuchlich ist der *7-Bit-Code* nach IOC/IEC 646 (internationale Referenzversion, IRV), der ursprünglich für die Datenkommunikation entwickelt wurde. Er beschreibt die Codierung von 128 Zeichen (Tabelle 1-1), und zwar von 96 Schriftzeichen und von 32 Zeichen zur Steuerung von Geräten und von Datenübertragungen (Tabelle 1-2). Zwölf der 96 Schriftzeichen sind der nationalen, sprachenspezifischen Nutzung vorbehalten. In der US-amerikanischen Version, dem *ASCII* (American Standard Code for Information Interchange) entsprechen sie der IRV, in der deutschen Referenzversion (DRV) nach DIN 66003 werden acht von ihnen zur Codierung der Umlaute, des Zeichens ß und des Paragraphzeichens § genutzt (vgl. Tabelle 1-1). – Rechnerintern wird den ASCII-Codewörtern wegen des Datenformats Byte ein achtes Bit (MSB) hinzugefügt, teils mit festem Wert, teils als *Paritätsbit* (siehe 7.7.3) oder zur Codeerweiterung (8-Bit-Code nach ISO/IEC 8859).

Ein weniger gebräuchlicher Zeichencode in der Rechentechnik ist der EBCDIC (Extended Binary Coded Dezimal Interchange Code). Er findet als 8-Bit-Code ausschließlich bei Großrechnern, sog. Mainframes Verwendung. In die Mikroprozessortechnik hat er keinen Eingang gefunden; so auch nicht bei PCs. – Zu Zeichencodes siehe vertiefend z.B. [Bohn, Flik 2002].

Tabelle 1-1. 7-Bit-Code nach ISO/IEC 646 bzw. ASCII, ergänzt um die nach DIN 66003 festgelegten zwölf Schriftzeichen (rechts der Schrägstriche)

höherwertige 3 Bits									
binär	0	0	0	0	1	1	1	1	
	0	0	1	1	0	0	1	1	
	0	1	0	1	0	1	0	1	
hex	0	1	2	3	4	5	6	7	
0000	0	NUL	DLE	SP	0	@ / \$	P	` / `	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	# / #	3	C	S	c	s
0100	4	EOT	DC4	\$ / \$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	,	7	G	W	g	w
1000	8	BS	CAN	(8	H	X	h	x
1001	9	HT	EM)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[/ Ä	{ / ä	
1100	C	FF	FS	,	^	L	\ / Ö	/ ö	
1101	D	CR	GS	-	=	M] / Ü	/ ü	
1110	E	SO	RS	.	>	N	^ / ^	~ / ß	
1111	F	SI	US	/	?	O	-	o	DEL

Tabelle 1-2. Bedeutung der Steuerzeichen im ASCII und nach DIN 66003

Hex.	ASCII	Bedeutung (ASCII)	Bedeutung (DIN 66003)
00	NUL	Null	Füllzeichen
01	SOH	Start of Heading	Anfang des Kopfes
02	STX	Start of Text	Anfang des Textes
03	ETX	End of Text	Ende des Textes
04	EOT	End of Transmission	Ende der Übertragung
05	ENQ	Enquiry	Stationsaufforderung
06	ACK	Acknowledge	Positive Rückmeldung
07	BEL	Bell	Klingel
08	BS	Backspace	Rückwärtsschritt
09	HT	Horizontal Tabulation	Horizontal-Tabulator
0A	LF	Line Feed	Zeilenvorschub
0B	VT	Vertical Tabulation	Vertikal-Tabulator
0C	FF	Form Feed	Formularvorschub
0D	CR	Carriage Return	Wagenrücklauf
0E	SO	Shift Out	Dauerumschaltung
0F	SI	Shift In	Rückschaltung
10	DLE	Data Link Escape	Datenübertr.-Umschaltung
11	DC1	Device Control 1	Gerätesteuerung 1
12	DC2	Device Control 2	Gerätesteuerung 2
13	DC3	Device Control 3	Gerätesteuerung 3
14	DC4	Device Control 4	Gerätesteuerung 4
15	NAK	Negative Acknowledge	Negative Rückmeldung
16	SYN	Synchronous Idle	Synchronisierung
17	ETB	End of Transmission Block	Ende des Übertragungsblocks
18	CAN	Cancel	Ungültig
19	EM	End of Medium	Ende der Aufzeichnung
1A	SUB	Substitute	Substitution
1B	ESC	Escape	Umschaltung
1C	FS	File Separator	Hauptgruppen-Trennung
1D	GS	Group Separator	Gruppen-Trennung
1E	RS	Record Separator	Untergruppen-Trennung
1F	US	Unit Separator	Teilgruppen-Trennung
20	SP	Space	Zwischenraum
7F	DEL	Delete	Löschen

1.1.3 Hexadezimal- und Oktalcode

Die Betrachtung binärer Information ist, wenn sie nicht als Zeichencode interpretiert dargestellt wird, für den Menschen ungewohnt und aufgrund der meist großen Binärstellenzahl unübersichtlich. Deshalb wird Binärinformation mikroprozessorextern, z. B. bei der Ausgabe auf einem Drucker, oft in komprimierter Form dargestellt. Am häufigsten wird hier die *hexadezimale Schreibweise* verwendet, bei der jede Bitkombination als Zahl im Zahlensystem mit der Basis 16 angegeben wird. Für die 16 Hexadezimalziffern werden dabei die Dezimalziffern 0 bis 9 und die Buchstaben A bis F (Werte 10 bis 15) verwendet. Zur Umformung einer

Bitkombination in die Hexadezimalschreibweise unterteilt man diese von rechts nach links in eine Folge von 4-Bit-Einheiten und ordnet jeder Einheit die der 4-Bit-Dualzahl entsprechende Hexadezimalziffer zu, z.B.

$$1100\ 1010\ 1111\ 0101 = \text{CAF5}_{16}$$

Eine weitere Möglichkeit der komprimierten Darstellung ist die *oktale Schreibweise*. Bei ihr wird jede Bitkombination als Zahl im Zahlensystem mit der Basis 8 angegeben. Zur Darstellung der acht Ziffern werden die Dezimalziffern 0 bis 7 verwendet; es werden jeweils 3-Bit-Einheiten zusammengefaßt, z.B.

$$1\ 100\ 101\ 011\ 110\ 101 = 145365_8.$$

Tabelle 1-3 zeigt die Zuordnung der Hexadezimal- und der Oktalziffern zu den 4-Bit- bzw. 3-Bit-Binäreinheiten. Anstatt des Begriffs hexadezimal wird gelegentlich auch der Begriff sedezial verwendet.

Tabelle 1-3. Hexadezimalziffern (Sedezialziffern) und Oktalziffern mit ihren Binäräquivalenten

Binär	Hexadezimal	Binär	Oktal
0000	0	000	0
0001	1	001	1
0010	2	010	2
0011	3	011	3
0100	4	100	4
0101	5	101	5
0110	6	110	6
0111	7	111	7
1000	8		
1001	9		
1010	A		
1011	B		
1100	C		
1101	D		
1110	E		
1111	F		

1.1.4 Ganze Zahlen

Der Mensch verwendet für die Darstellung von Zahlen das Stellenwertsystem zur Basis 10 mit den Dezimalziffern 0 bis 9. Der Wert einer n -stelligen Dezimalzahl Z ergibt sich hier als Summe der entsprechend ihren Positionen mit Zehnerpotenzen gewichteten Ziffern a_i zu

$$Z = \sum_{i=0}^{n-1} a_i \cdot 10^i, \text{ z.B.}$$

$$205 = 2 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0.$$

In Rechnern werden, bedingt durch ihre Arbeitsweise mit zweiwertiger Logik, Zahlen binär codiert. Hierfür werden die Ziffernsymbole 0 und 1 verwendet. Gerechnet wird im Stellenwertsystem zur Basis 2 (duales Zahlensystem); in manchen Prozessoren wahlweise auch im Stellenwertsystem zur Basis 10 (dualcodiertes Dezimalsystem).

Vorzeichenlose Dualzahlen

Eine n -stellige *Dualzahl* (*binary number*) ist eine Zahl zur Basis 2 mit Bewertung ihrer Ziffern 0 und 1 (*binary digits, Bits*) durch die Gewichte $2^{n-1}, \dots, 8, 4, 2, 1$ (*Dualcode*). Bei vorzeichenloser Darstellung (*unsigned binary number*) und einer Stellenanzahl von n Bit (Datenformat) kann sie somit die Werte von 0 bis $2^n - 1$ annehmen. Wie bei den Dezimalzahlen ergibt sich der Wert einer n -stelligen Dualzahl Z aus der Summe der gewichteten Ziffern a_i zu

$$Z = \sum_{i=0}^{n-1} a_i \cdot 2^i.$$

Diese Formel beschreibt zugleich die Umrechnung von Dualzahlen in Dezimalzahlen. Dazu ein Beispiel mit Kennzeichnung der Zahlenbasen durch Indizes:

$$11001101_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 205_{10}.$$

Ein gebräuchliches Umrechnungsverfahren für Dezimalzahlen in Dualzahlen ergibt sich aus der Darstellung der obigen Summenformel im Hornerschema:

$$Z = (\dots(a_{n-1} \cdot B + a_{n-2}) \cdot B + \dots + a_1) \cdot B + a_0.$$

Man dividiert die Dezimalzahl durch 2 und notiert den Rest, um dann mit dem Quotienten in gleicher Weise zu verfahren, bis er Null wird. Der jeweilige Rest, der die Werte 0 und 1 annehmen kann, bildet die Dualzahl, beginnend mit der niedrigstwertigen Ziffer a_0 . Für das obige Beispiel ergibt sich:

$$\begin{array}{rcl} 205_{10} \rightarrow & 205 / 2 & = 102 \quad \text{Rest } 1 \\ & 102 / 2 & = 51 \quad \text{Rest } 0 \\ & 51 / 2 & = 25 \quad \text{Rest } 1 \\ & 25 / 2 & = 12 \quad \text{Rest } 1 \\ & 12 / 2 & = 6 \quad \text{Rest } 0 \\ & 6 / 2 & = 3 \quad \text{Rest } 0 \\ & 3 / 2 & = 1 \quad \text{Rest } 1 \\ & 1 / 2 & = 0 \quad \text{Rest } 1 \end{array} \rightarrow 11001101_2.$$

32-Bit-Mikroprozessoren sehen Operationen mit solchen vorzeichenlosen Dualzahlen üblicherweise im Byte-, Halbwort- und Wortformat vor, wobei Zahlen mit geringerer Stellenanzahl durch sog. führende Nullen an die Formate angepaßt werden (*zero extension*). Durch die in der Darstellung begrenzte Bitanzahl n einer Informationseinheit erstreckt sich der Zahlenbereich bei vorzeichenlosen Dualzahlen von 0 bis $2^n - 1$ (Tabelle 1-4). Wird bei einer arithmetischen Operation der Zahlenbereich überschritten, so entsteht zwar wiederum eine Zahl innerhalb der Bereichsgrenzen, ihr Wert ist jedoch nicht korrekt, was vom Mikroprozessor als Bereichsüberschreitung durch Setzen des *Übertragsbits* (*carry bit C*) im sog. Condition-Code-Teil seines Statusregisters angezeigt wird. Bild 1-2a zeigt das Entstehen von Bereichsüberschreitungen am Beispiel von 8-Bit-Dualzahlen im Zahlenring.

Tabelle 1-4. Wertebereiche für vorzeichenlose Dualzahlen und 2-Komplement-Zahlen

Bitanzahl	vorzeichenlose Dualzahlen	2-Komplement-Zahlen	
8	0 bis 255	-128	bis +127
16	0 bis 65535	-32768	bis +32767
32	0 bis $4 \cdot 10^9 - 1$	-2 G	bis +2 G-1
n	0 bis $2^n - 1$	-2^{n-1}	bis $+2^{n-1} - 1$

Vorzeichenbehaftete Dualzahlen

Bei vorzeichenbehafteten Dualzahlen (*signed binary numbers, integers*) wird das höchstwertige Bit zur Codierung des Vorzeichens benutzt (0 positiv, 1 negativ). Gegenüber vorzeichenlosen Dualzahlen stehen damit – bei gleichbleibendem n -Bit-Datenformat – nur noch $n-1$ Bit für die eigentliche Zahl zur Verfügung. Während positive Zahlen aus dem Vorzeichen 0 und der eigentlichen Dualzahl einfach zusammengesetzt werden, kennt man für negative Zahlen drei verschiedene Darstellungen: die Vorzeichen-Betrag-Darstellung (Vorzeichen 1 und eigentliche Dualzahl), die 1-Komplement-Darstellung (Vorzeichen 1 und bitweise invertierte eigentliche Dualzahl) und die 2-Komplement-Darstellung, wie sie in heutigen Rechnern ausschließlich verwendet wird.

Für eine *2-Komplement-Zahl* (*two's complement number*) ergibt sich der Wert Z , gleichgültig ob positiv oder negativ, als Summe der gewichteten Ziffern a_i , wobei das Vorzeichenbit a_{n-1} mit negativem Gewicht einbezogen wird

$$Z = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i, \text{ z.B.}$$

$$10111101_2 = -1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -67_{10}$$

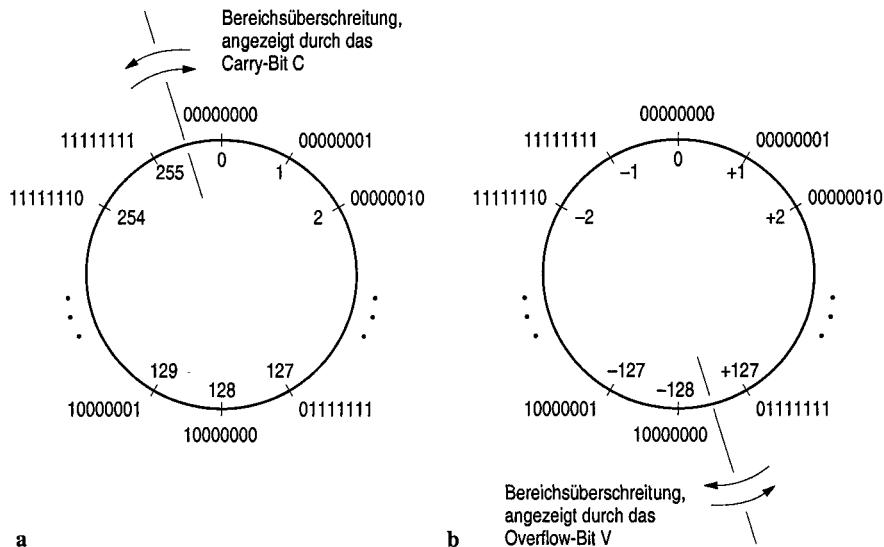


Bild 1-2. Zahlenring, a für vorzeichenlose Dualzahlen, b für 2-Komplement-Zahlen, jeweils in 8-Bit-Zahlendarstellung

Das heißt, bei positivem Vorzeichen $a_{n-1} = 0$ ist der Zahlenwert gleich dem Wert, den die $n-1$ verbleibenden Bits als vorzeichenlose Zahl haben, bei negativem Vorzeichen $a_{n-1} = 1$ ist er gleich dem Wert, den die $n-1$ verbleibenden Bits als vorzeichenlose Zahl haben, jedoch um die Größe des halben Wertebereichs 2^{n-1} in den negativen Zahlenraum verschoben. Oder anders ausgedrückt: Bei n -steligen 2-Komplement-Zahlen ergänzen sich positive und negative Zahlen zu 2^n . Die Umrechnung einer positiven in die entsprechende negative Zahl oder umgekehrt (2-Komplementierung) kann dementsprechend durch Umkehrung aller n Bits und anschließende Addition von Eins erfolgen, z.B.

$$01000011 = +67_{10}$$

$$\begin{array}{rcl} \text{Invertierung:} & 10111100 \\ \text{Addition von 1:} & + \quad \underline{\quad 1 \quad} \\ \text{2-Komplement:} & 10111101 = -67_{10} \end{array}$$

und zurück:

$$\begin{array}{rcl} \text{Invertierung:} & 01000010 \\ \text{Addition von 1:} & + \quad \underline{\quad 1 \quad} \\ \text{2-Komplement:} & 01000011 = +67_{10}. \end{array}$$

32-Bit-Mikroprozessoren sehen Operationen mit 2-Komplement-Zahlen wiederum im Byte-, Halbwort- und Wortformat vor. Zahlen geringerer Stellenzahl werden durch führende Bits gleich dem Vorzeichenbit an diese Formate angepaßt

(*sign extension*). Der Zahlenbereich erstreckt sich von -2^{n-1} bis $+2^{n-1}-1$, wobei die Null als positive Zahl definiert ist (Tabelle 1-4). Eine Bereichüberschreitung, z. B. als Ergebnis einer arithmetischen Operation, führt auf eine nicht korrekte Zahl innerhalb der Bereichsgrenzen (Bild 1-2b). Sie wird vom Mikroprozessor durch das *Überlaufbit* (*overflow bit* V) im Condition-Code-Teil seines Statusregisters angezeigt (sog. *arithmetic overflow*).

1.1.5 Gleitpunktzahlen

Zahlendarstellung. Für das Rechnen mit reellen Zahlen hat sich in der Computertechnik die halblogarithmische Zahlendarstellung mit Vorzeichen, Mantisse und Exponent durchgesetzt. Die entsprechenden Zahlen werden als *Gleitpunktzahlen* (*floating-point numbers*) bezeichnet. Gegenüber den ganzen Zahlen erreicht man mit ihnen, gleiche Bitanzahl für das Datenformat vorausgesetzt, einen wesentlich größeren Wertebereich bei jedoch geringerer Genauigkeit. Ihre Darstellung ist durch die Norm IEEE 754–1985 (DIN IEC 60559 1991) festgelegt. Sie hat die Form

$$Z_{FP} = (-1)^s \cdot (1.f) \cdot 2^{e - bias}$$

und ist in zwei sog. Grundformaten (*basic formats*) gemäß Bild 1-3 codiert: einfach lang mit 32 Bit und doppelt lang mit 64 Bit. Man spricht dabei auch von einfacher und doppelter Genauigkeit der Darstellung (*single/double precision*).

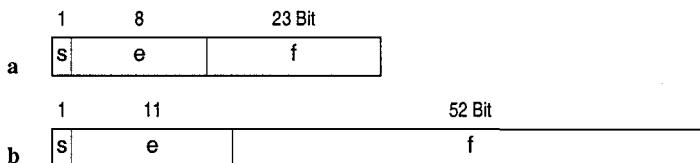


Bild 1-3. Grundformate für Gleitpunktzahlen nach IEEE 754-1985, a einfach lang (32 Bit), b doppelt lang (64 Bit)

s ist das *Vorzeichen* (*sign*) der Gleitpunktzahl (0 positiv, 1 negativ). Die *Mantisse* 1.f (*significand*) wird als gemischte Zahl in *normalisierter Form* angegeben. Sie wird dazu bei gleichzeitigem Vermindern des Exponenten so weit nach links geschoben, bis sie eine Eins an höchster Stelle aufweist (führende Eins). Der Binärpunkt wird rechts von dieser Eins festgelegt. In den Grundformaten gespeichert wird lediglich der Bruch f (*fraction*); die nicht gespeicherte Eins (*integer bit*) wird in der *Gleitpunktrecheneinheit* (*floating-point unit*, FPU) dann automatisch hinzugefügt. Die Mantisse hat gemäß $0 \leq f < 1$ den Wertebereich $1.0 \leq 1.f < 2.0$. Der zunächst vorzeichenbehaftete Exponent E wird im Datenformat als *transformierter Exponent* (*biased exponent*) e codiert, d.h., zu ihm wird, ausgehend von seiner 2-Komplement-Darstellung, eine Konstante (bias: 127 bzw. 1023) addiert, so daß

Tabelle 1-5. Zahlenbereiche und Genauigkeiten für Gleitpunktzahlen einfacher und doppelter Länge (mit $E = e - \text{bias}$)

	Einfache Länge	Doppelte Länge
Datenformat	32 Bit	64 Bit
Mantisse	24 Bit	53 Bit
Größter relativer Fehler	2^{-24}	2^{-53}
Genauigkeit	≈ 7 Dezimalstellen	≈ 16 Dezimalstellen
Transformierter Exponent e	8 Bit	11 Bit
Bias	127	1023
Bereich für E	-126 bis +127	-1022 bis +1023
Kleinste positive Zahl	$2^{-126} \approx 1,2 \cdot 10^{-38}$	$2^{-1022} \approx 2,2 \cdot 10^{-308}$
Größte positive Zahl	$(2 - 2^{-23}) \cdot 2^{127} \approx 3,4 \cdot 10^{38}$	$(2 - 2^{-52}) \cdot 2^{1023} \approx 1,8 \cdot 10^{308}$

sich eine vorzeichenlose Zahl $e = E + \text{bias}$ ergibt. Der transformierte Exponent und die Anordnung von s, f und e in den beiden Datenformaten erlauben es, den Vergleich von Gleitpunktzahlen, genauer von deren Beträgen, auf den Vergleich von Ganzzahlen zurückzuführen. – Tabelle 1-5 zeigt für beide Grundformate die Zahlenbereiche und die Genauigkeiten bei normalisierter Darstellung.

Anmerkung. Für beide Grundformate sieht die Norm je ein erweitertes Format (*extended format*) mit implementierungsabhängigen Breiten und Unterteilungen vor. Sie dienen als Zwischenformate der Gleitpunktrecheneinheit zur Erhöhung der Rechengenauigkeit. Gebräuchlich sind hier 80 Bit als erweitertes doppelt langes Grundformat, unterteilt in 1, 15 und 64 Bit für s, e und $1.f$. Die führende Eins wird hier mit im Format gespeichert.

Zusätzlich zu den normalisierten Zahlen sind in der Norm weitere Zahlen festgelegt: die Null (*zero*), Unendlich (*infinity*) und *unnormalisierte Zahlen* (*denormalized numbers*), jeweils mit positivem und negativem Vorzeichen. Außerdem sieht sie sog. „*Nicht*“zahlen (*not-a-numbers*, NaNs) vor, die gemäß ihrer Bezeichnung nicht zum Zahlenraum gehören. Zur Codierung dieser Zahlen und Nichtzahlen werden der größte Exponentwert und der kleinste Exponentwert als Sonderwerte herangezogen, die dementsprechend für die Darstellung normalisierter Zahlen nicht zur Verfügung stehen (siehe Bereich für E in Tabelle 1-5). Bei den unnormalisierten Zahlen ist zu beachten, daß diese zwar mit $e = 0$ codiert, jedoch mit $e = 1$ ($E = -126$ bzw. -1022) interpretiert werden.

Null:	$v = (-1)^s \cdot 0$	$e = 0, f = 0$
Unnormalisiert:	$v = (-1)^s \cdot (0.f) \cdot 2^{-126 \setminus -1022}$	$e = 0, f \neq 0$
Normalisiert:	$v = (-1)^s \cdot (1.f) \cdot 2^{e-127 \setminus -1023}$	$0 < e < 255 \setminus 2047$
Unendlich:	$v = (-1)^s \cdot \infty$	$e = 255 \setminus 2047, f = 0$
Nichtzahl:	NaN	$e = 255 \setminus 2047, f \neq 0$

Tabelle 1-6. Codierung von Gleitpunktzahlen und Darstellung ihrer Zahlenwerte v. Angabe der Mantisse im Dualcode und des Exponenten E als Dezimalzahl; Schrägstriche trennen die Exponentenangaben für einfache und doppelte Länge

s	e	f	Wert v	Bedeutung
0	11...111	11.....111		Nichtzahlen (NaNs)
:	:	:		
0	11...111	00.....001		
0	11...111	00.....000	$+\infty$	positives Unendlich
0	11...110	11.....111	$+1.11.....111 \cdot 2^{+127+1023}$	positive normalisierte Zahlen
:	:	:	:	
0	00...001	00.....000	$+1.00.....000 \cdot 2^{-126-1022}$	
0	00...000	11.....111	$+0.11.....111 \cdot 2^{-126-1022}$	positive unnormalisierte Zahlen
:	:	:	:	
0	00...000	00.....001	$+0.00.....001 \cdot 2^{-126-1022}$	
0	00...000	00.....000	$+0$	positive Null
1	00...000	00.....000	-0	negative Null
1	00...000	00.....001	$-0.00.....001 \cdot 2^{-126-1022}$	negative unnormalisierte Zahlen
:	:	:	:	
1	00...000	11.....111	$-0.11.....111 \cdot 2^{-126-1022}$	
1	00...001	00.....000	$-1.00.....000 \cdot 2^{-126-1022}$	negative normalisierte Zahlen
:	:	:	:	
1	11...110	11.....111	$-1.11.....111 \cdot 2^{+127+1023}$	
1	11...111	00.....000	$-\infty$	negatives Unendlich
1	11...111	00.....001		Nichtzahlen (NaNs)
:	:	:		
1	11...111	11.....111		

Tabelle 1-6 veranschaulicht die Codierungen und zeigt die Interpretation der verschiedenen Zahlenarten. Bild 1-4 verdeutlicht die durch die begrenzte Stellenanzahl der Mantisse bedingte Diskretisierung des Zahlenraums und die damit verbundene Einschränkung, nämlich daß das Ergebnis einer Gleitpunktoperation gegebenenfalls nicht exakt darstellbar ist und deshalb gerundet werden muß.

Operationen. Die Norm sieht als arithmetische Operationen die Addition, die Subtraktion, die Multiplikation, die Division, die Restbildung, den Vergleich und das Wurzelziehen vor. Hinzu kommen Konvertierungsoperationen zwischen den

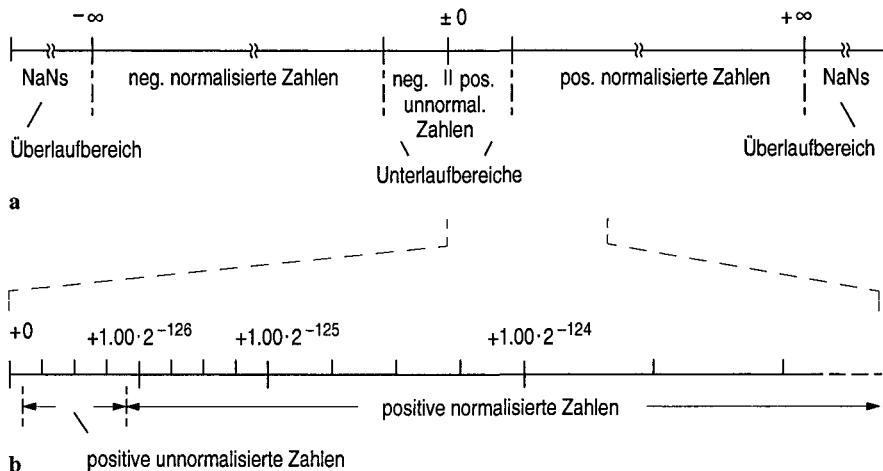


Bild 1-4. Zahlenraum der Gleitpunktzahlen. **a** Gesamter Zahlenraum mit den Überlauf- und Unterlaufbereichen. **b** Ausschnitt der positiven unnormalisierten und normalisierten Zahlen mit Darstellung der Diskretisierung des Zahlenraums, hier für das einfache lange Datenformat, jedoch mit nur 2 Bits für f , d.h. mit 4 Mantissenwerten pro Exponentenwert E

Gleitpunktformaten und Konvertierungsoperationen zwischen Gleitpunktzahlen und 2-Komplement-Zahlen (integer) sowie wenigstens einem Format für BCD-Zeichenketten. Heutige Gleitpunktrecheneinheiten ermöglichen darüber hinaus u.a. trigonometrische und logarithmische Operationen. – Zu Algorithmen von Gleitpunktoperationen siehe z.B. [Goldberg 1990].

Bei der Addition und der Subtraktion zweier Operanden mit ungleichen Exponenten muß die eine Zahl an den Exponenten der anderen angeglichen werden. Hierbei wird der Signifikand des Operanden mit kleinerem Exponenten bei gleichzeitigem Erhöhen des Exponenten so weit nach rechts geschoben, bis die Exponenten gleich sind (unnormalisierte Zahl). – Beispiel: Subtraktion 4,0 minus 1,5 mit vorherigem Angleichen der Exponenten und Normalisieren des Ergebnisses; Darstellung der Mantisse als Signifikand:

s	e	Signifikand		
0	10000001	1.0000....0	=	4,0 normalisierter Minuend
0	01111111	1.1000....0	=	1,5 normalisierter Subtrahend
0	10000001	0.0110....0	=	1,5 unnormalisierter Subtrahend
0	10000001	0.1010....0	=	2,5 Ergebnis
0	10000000	1.0100....0	=	2,5 normalisiertes Ergebnis

Bei der Multiplikation und der Division ist ein Angleichen der Exponenten naturgemäß nicht erforderlich. Diese werden unmittelbar addiert bzw. subtrahiert. Bei allen vier Grundrechenoperationen sind die Vorzeichen der Mantissen jeweils für sich auszuwerten, um das Vorzeichen des Ergebnisses zu erhalten.

Runden. Aufgrund der festen, d.h. begrenzten Stellenzahl des Signifikanden gibt es zwischen zwei benachbarten darstellbaren Zahlen eine beliebige Anzahl nicht-darstellbarer Zahlen. Solche nichtdarstellbaren Zahlen treten bei Gleitpunktoperationen beim Rechtsshift von Signifikanden auf, wenn dabei signifikante Bits aus dem Datenformat hinausgeschoben werden, z.B. beim Erzeugen der unnormalisierten Form eines Subtrahenden oder beim Korrigieren des Ergebnisses nach Auftreten eines Übertrags bei der Addition. Hier ist ein Runden des Ergebnisses erforderlich, wobei sichergestellt sein muß, daß exakte Zahlen ggf. erhalten bleiben müssen (z.B. bei der Multiplikation mit 1).

Als Verfahren mit bester Ausmittelung von *Rundungsfehlern* schlägt die Norm das sog. *korrekte Runden* vor. Bei ihm wird das Ergebnis gleich der nächstliegenden, im Zieldatenformat darstellbaren Zahl, im Zweifelsfall gleich der Zahl mit der geradzahligen Endziffer gesetzt (*round to nearest*). Das Rechenwerk wertet dazu zwei Zusatzbits aus, die als rechenwerksinterne Ergänzung des Signifikanden bei den Gleitpunktoperationen mitgeführt werden. Das sog. *Guard-Bit* speichert das bei einem Rechtsshift zuletzt hinausgeschobene Bit des Signifikanden, das sog. *Sticky-Bit* speichert die davor hinausgeschobenen Bits in oder-verknüpfter Form. Wird das Ergebnis einer Operation vor dem Runden normalisiert, so ist ein drittes Bit, das sog. *Round-Bit* erforderlich. Es ist sozusagen zwischen den beiden vorgenannten Bits angeordnet und nimmt das vorletzte hinausgeschobene Bit des Signifikanden auf, das dementsprechend nicht zur Bildung des Sticky-Bits herangezogen wird [Goldberg 1990].

Drei weitere von der Norm vorgesehene Rundungsverfahren werden unter dem Begriff „*directed roundings*“ zusammengefaßt. Zu ihnen gehören das *Aufrunden* des Ergebnisses in Richtung positiv Unendlich und das *Abrunden* in Richtung negativ Unendlich. Durch den Einsatz beider Verfahren lassen sich Ergebnisse mittels zweier Schranken darstellen, innerhalb deren der korrekte Wert liegt (*Intervalarithmetik*). Beim dritten Verfahren, dem *Runden gegen Null*, werden die das Grundformat überschreitenden Bitpositionen ignoriert.

Gleitpunktrecheneinheiten sind üblicherweise von vornherein auf eine Verarbeitungsbreite von wenigstens 64 Bit ausgelegt, ergänzt um die drei „Rundungsbits“. Das heißt, daß auch einfach lange Operanden grundsätzlich im 64-Bit-Format bearbeitet werden. Häufig arbeiten Gleitpunktrecheneinheiten sogar in einem noch breiteren Format, z.B. von 80 Bit (extended format, siehe oben), womit die Rechengenauigkeit für Operationsfolgen zusätzlich erhöht wird und ein erweiterter Exponentenbereich zur Verfügung steht. Von diesen Verarbeitungsbreiten abhängig sind auch die Datenregister der Recheneinheiten mit z.B. 64 oder 80 Bit ausgelegt. Dementsprechend müssen Quelloperanden beim Laden in diese Register in das gegebenenfalls längere Format, Ergebnisoperanden beim Rückschreiben in den Speicher wieder auf das kürzere Format gebracht werden, was mit einem erneuten Runden verbunden sein kann.

Ergänzende Angaben zur Arithmetik mit Gleitpunktzahlen finden sich in [Bohn, Flik 2002]; für eine vertiefende Betrachtung der Gleitpunktzahlen siehe [Goldberg 1991].

1.1.6 Binärcodierte Dezimalziffern (BCD-Zahlen)

Eine weitere Möglichkeit binärer Zahlendarstellung bietet die ziffernweise Codierung von Dezimalzahlen (*binary coded decimals*, BCD). Am gebräuchlichsten und in der Mikroprozessortechnik ausschließlich verwendet ist der Dualcode (8421-Code). Ein Codewort umfaßt 4 Bit (Tetrade) mit den Gewichten 8, 4, 2 und 1 (Tabelle 1-7).

Tabelle 1-7. BCD-Ziffern

Dezimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Zur Darstellung von ganzen Zahlen werden die Codewörter entsprechend der Ziffernfolge aneinandergereiht, z.B.

$$205_{10} = 0100\ 0000\ 0101_{BCD}$$

Man bezeichnet diese Darstellung in 4-Bit-Einheiten auch als gepackte Darstellung (*packed BCD*). Im Gegensatz dazu spricht man bei einer Dezimalzifferncodierung in einem der Zeichencodes von ungepackter Darstellung (*unpacked BCD*). Sie besteht beim ASCII aus den drei zusätzlichen höherwertigen Bits 011. Beim EBCDIC sind es die vier höherwertigen Bits 1111. – Für eine vertiefende Betrachtung von Informationsdarstellungen siehe z.B. [Hoffmann 1993, Mackenzie 1980].

1.2 Rechnerstruktur (CISC)

Die in diesem und im folgenden Abschnitt 1.3 behandelten Struktur- bzw. Programmieraspekte von Mikroprozessorsystemen basieren, soweit sie prozessor-

spezifisch sind, auf CISC-Prozessoren. Wir betrachten dazu zunächst eine Prozessorstruktur, die der besseren Anschauung halber auf einem 16-Bit-Prozessor basiert. Dementsprechend bezeichnen wir im folgenden – und das als Ausnahme in diesem Buch – das 16-Bit-Datenformat als Wort. In Abschnitt 1.4 beschreiben wir dann RISC-spezifische Struktur- und Programmieraspekte; dabei beziehen wir uns auf einen 32-Bit-Prozessor. Dementsprechend wird dort, wie bei 32-Bit-Prozessoren üblich, das 32-Bit-Datenformat als Wort bezeichnet.

1.2.1 Übersicht über die Hardwarekomponenten

Ein Mikroprozessorsystem besteht aus Hardwarekomponenten mit im Prinzip drei unterschiedlichen Funktionen: dem eigentlichen Mikroprozessor zum Verarbeiten von Rechengrößen durch ein Programm, dem *Hauptspeicher (Arbeitsspeicher)* zum Speichern der Rechengrößen und zum Speichern von Programmen und einer oder mehreren Ein-/Ausgabeeinheiten, über die Daten von den Peripheriegeräten (Ein-/Ausgabegeräten und Hintergrundspeichern) eingelesen und an sie ausgegeben werden. Hinzu kommen Verbindungswege zwischen diesen Komponenten für den Datentransport und den Austausch von Steuersignalen (Bild 1-5).

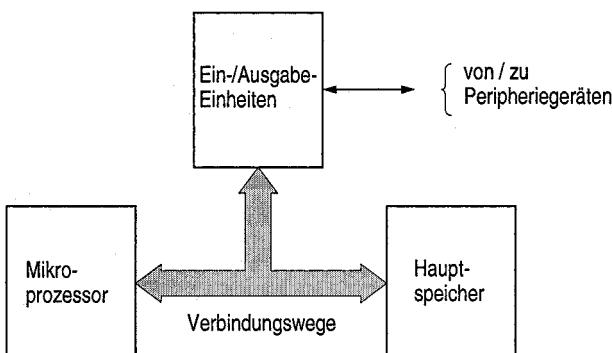


Bild 1-5. Komponenten eines Mikroprozessorsystems

Der Mikroprozessor selbst besteht aus einem steuernden Werk (*Steuerwerk*) und einem ausführenden Werk (*Operationswerk*). Das Steuerwerk übernimmt die Ablaufsteuerung innerhalb des Prozessors sowie die Ansteuerung des Hauptspeichers und der Ein-/Ausgabeeinheiten. Es veranlaßt das Lesen der Befehle aus dem Hauptspeicher, interpretiert sie und steuert ihre Ausführung. Die zu verarbeitenden Operanden werden aus dem Hauptspeicher gelesen oder über eine Ein-/Ausgabeeinheit eingegeben und die Ergebnisse in den Hauptspeicher geschrieben oder über eine Ein-/Ausgabeeinheit ausgegeben. Das Operationswerk übernimmt dabei die Zwischenspeicherung der Operanden und der Ergebnisse und führt die logischen und arithmetischen Operationen mit diesen Operanden aus.

Der Hauptspeicher umfaßt eine Vielzahl einzelner Speicherplätze (*Speicherzellen*) im 8-Bit-, 16-Bit- oder 32-Bit-Format, die man sich aufeinanderfolgend angeordnet denkt. Um einzelne Zellen anwählen zu können, sind die Speicherplätze numeriert, d.h. jeder Speicherplatz ist durch die Angabe einer Zahl eindeutig identifizierbar (adressierbar). Man bezeichnet diese Nummer als *Adresse*. Adressen werden ebenfalls als Dualzahlen dargestellt; mit 16 Bits können z.B. bis zu $2^{16} = 65536 = 64\text{K}$, mit 24 Bits bis zu $2^{24} = 16\text{M}$ und mit 32 Bits bis zu $2^{32} = 4\text{G}$ Adressen dargestellt werden. Durch die *Adresslänge* ist also die Begrenzung des *Adressraums* des Hauptspeichers und damit dessen größtmögliche Kapazität an Speicherzellen vorgegeben.

Der Mikroprozessor selbst besitzt im Vergleich zum Hauptspeicher nur sehr wenige Speicherplätze. Einige von ihnen sind für spezielle Abläufe innerhalb des Prozessors vorgesehen und für die Maschinen- bzw. Assemblerprogrammierung nicht unmittelbar verfügbar. Andere können wie Hauptspeicherzellen explizit angewählt werden und sind somit für den Programmierer „sichtbar“. Auch in anderen Systembausteinen kommen solche einzelnen Speicherplätze vor, die gemeinhin als *Register* bezeichnet werden. Sind mehrere Register zu einem kleinen Speicher zusammengefaßt, so spricht man von einem *Registerspeicher*. Obwohl sich Register und Speicherzellen in ihrer Funktion gleichen, soll die begriffliche Trennung beibehalten werden; damit lassen sich einzelne Speicherzellen von vielen, zu größeren Speichern zusammengefaßten Speicherzellen unterscheiden.

Ein-/Ausgabeeinheiten bilden die Schnittstellen zwischen dem Mikroprozessorsystem und der Peripherie. Sie dienen hauptsächlich zur Anpassung der Datenformate und der Arbeitsgeschwindigkeiten der Übertragungspartner. In ihrer einfachsten Ausführung besteht eine solche Einheit aus einem passiven Schnittstellenbaustein (Interface-Baustein), und die eigentliche Übertragungssteuerung unterliegt dem Prozessor; in komplexeren Ausführungen ist sie zur Steuerung der Übertragung mit einer zusätzlichen Steuereinheit (direct memory access controller) oder einem Ein-/Ausgabeprozessor (i/o processor) ausgestattet. – Auch in den Ein-/Ausgabeeinheiten befinden sich Register oder sehr kleine Registerspeicher, z.B. zum Zwischenspeichern von Information für die Datenübertragung zwischen Prozessor/Speicher einerseits und der Peripherie des Mikroprozessorsystems andererseits.

Zur Speichersymbolik. Wie beschrieben, befinden sich *Speicher* praktisch in allen Systemkomponenten eines Mikroprozessorsystems, von einzelnen Registern mit meist speziellen Funktionen bis hin zu großen Speichern zur Ablage (Schreiben) oder zum Wiedergewinnen (Lesen) von Information, verbunden mit dem bereits geschilderten Auswählen einer Speicherzelle (Addressieren).

Bild 1-6 zeigt eine Zusammenstellung von Symbolen für einige wichtige Speicherarten. Darin kennzeichnen die rechteckigen Felder Speicherzellen der Bit-Anzahl m, entweder einzeln als Register, wie in den Teilbildern a und b, oder zusammengefaßt zu Speichern, wie jeweils rechts in den Teilbildern c, d und f. Die

trapezförmigen Felder links in den Teilbildern c, d und e symbolisieren die Auswahl der Speicherzellen. Es handelt sich dabei um *Decodierer*, die die i.allg. als Dualzahlen verschlüsselten (codierten) Adressen der Bit-Anzahl n entschlüsseln (decodieren), so daß genau eine der 2^n möglichen Adressen eine der 2^m Speicherzellen auswählt.

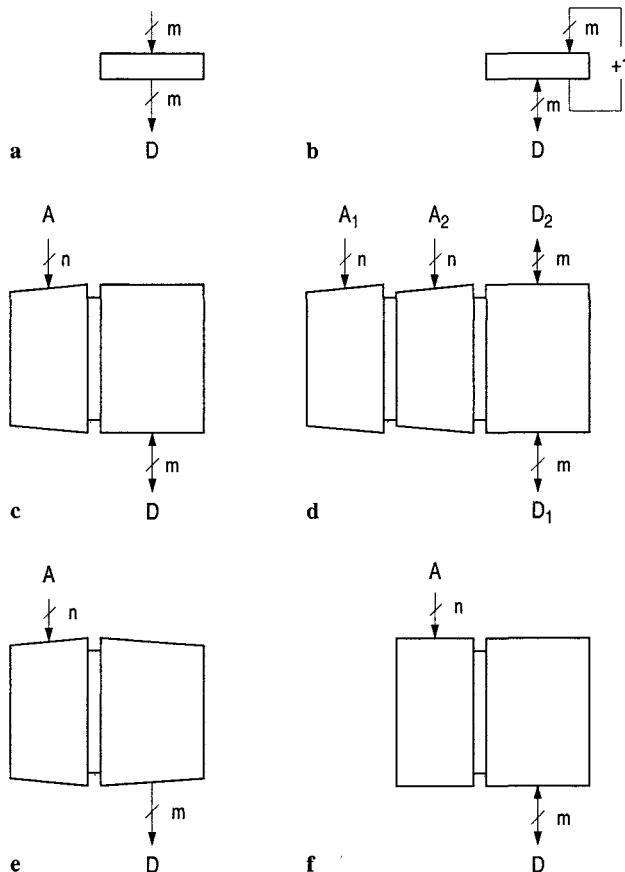


Bild 1-6. Speichersymbolik. **a** Register mit unidirektionalen Schreib-/Leseleitungen, **b** Register mit bidirektonaler Schreib-/Leseleitung und Zählfunktion (Zähler), **c** Speicher mit wahlfreiem Zugriff (RAM), **d** Speicher mit Zweifachzugriff (Zweiport-RAM), **e** Festwertspeicher (ROM), **f** Assoziativspeicher (CAM). D Datenleitungen, A Adresseleitungen, m Datenlänge, n Adresslänge

Schließlich sind noch das trapezförmige Feld rechts in Teilbild e sowie das rechteckige Feld links in Teilbild f zu erklären. Bei ersterem handelt es sich um die Zusammenfassung von Speicherzellen, die nur gelesen werden können, zu einem Speicher unveränderlichen Inhalts. Das in dieser Richtung „durchlaufene“ Trapezfeld bezeichnet man wegen der gegenläufigen Funktion zum Decodierer auch als *Codierer*. Bei zweiterem handelt es sich um Adreßzellen, die wie die mit

ihnen verbundenen Speicherzellen verändert (beschrieben) werden können. Insfern stellt das in dieser Richtung „durchlaufene“ Rechteckfeld einen Decodierer veränderlichen Inhalts dar.

Miteinander verglichen ist den drei Speicherarten in den Teilbildern c, e, und f eine gewisse Systematik zu eigen: Die Felder links enthalten Adressen, die Felder rechts die Speicherzellen. Bei c und e sind die Adressen unveränderlich, bei f sind sie veränderlich; bei e sind die Speicherzellen unveränderlich, bei c und f sind sie veränderlich. Die Funktion hinsichtlich des Adressierungs- und des Lesevorgangs ist bei allen drei Formen identisch: Die am Speicher anliegende Adresse wird gleichzeitig mit allen (links) unveränderlich bzw. veränderlich gespeicherten Adressen verglichen. An der Stelle, an der Übereinstimmung herrscht, wird (rechts) der zugehörige unveränderliche bzw. veränderliche Inhalt der Speicherzelle gelesen. Ein Schreiben von Speicherzellen ist hingegen nur bei c und f möglich; zusätzlich kann bei f eine Speicherzelle *zusammen* mit ihrer Adresse mit Information beschrieben werden, was als Laden bezeichnet wird und weitere, nicht im Symbol berücksichtigte Auswahl- und Schreibvorrichtungen erfordert. Im Falle f ist weiterhin zu beachten, daß die Kapazität dieses Speichers *nicht* von der Adreßlänge abhängt.

Der Speicher in Teilbild c wird wegen des für alle Speicherzellen gleichartigen Zugriffs bzw. wegen seiner Schreib-/Lesemöglichkeit *Speicher mit wahlfreiem Zugriff (random access memory, RAM)* oder – treffender – *Schreib-/Lesespeicher* genannt. In dieser Form kommt er in Mikroprozessorsystemen primär als Hauptspeicher vor (siehe z.B. S. 27, Bild 1-11, rechts). Teilbild d zeigt eine Variante eines RAM, die wegen ihrer beiden unabhängigen Schreib-/Lese-„Tore“ *Zweiport-RAM (dual-port RAM)*, verallgemeinert *Multiport-RAM*, genannt wird. Multiport-RAMs haben geringe Kapazitäten und werden deshalb meistens nur als Registerspeicher in Prozessoren eingesetzt (siehe z.B. S. 27, Bild 1-11, links).

Der Speicher in Teilbild e wird wegen seines fest gespeicherten Inhalts bzw. wegen seiner Nur-Lese-Eigenschaft *Festwertspeicher* oder Nur-Lesespeicher (*read only memory, ROM*) genannt. Auch hier ist im Prinzip die Verallgemeinerung auf Multiport-ROMs möglich. ROMs findet man sowohl prozessorextern zur Speicherung von z.B. unveränderlich bleibenden Programmen als auch prozessorintern zur Speicherung von per se unveränderlichen Mikroprogrammen (siehe z.B. S. 27, Bild 1-11, oben).

Der Speicher in Teilbild f schließlich wird, da Speicherzelle und Adresse zu einer Einheit zusammengefaßt sind, bzw. weil er mittels eines Teils seines Inhalts adressiert wird, *Assoziativspeicher* oder inhaltsadressierbarer Speicher (*content addressable memory, CAM*) genannt. Auf die Problematik des Ladens eines solchen Speichers wird erst im Zusammenhang mit seinem Haupteinsatz als sog. *Cache* (für den Programmierer transparenter, d.h. unsichtbarer Zwischenspeicher) näher eingegangen (siehe 6.2.2). Auch dieser Speichertyp wird sowohl prozessorextern zur Zwischenspeicherung von Ausschnitten von Programmen und

Daten als auch prozessorintern, meist nur zur Zwischenspeicherung von Ausschnitten aus Programmen eingesetzt (siehe z.B. S. 51, Bild 1-24, oben).

1.2.2 Busorientierte Systemstruktur

In Bild 1-5 (S. 18) sind zwischen den einzelnen Komponenten des Mikroprozessorsystems Verbindungswege eingezeichnet, die den Transport von Information symbolisieren, ohne daß damit ihr physikalischer Aufbau festgelegt ist. In Wirklichkeit unterscheidet man drei Arten von Information, für die üblicherweise auch getrennte Verbindungswege aufgebaut werden:

1. *Daten*; dazu zählen die Operanden und die Ergebnisse, in einem weiteren Sinn auch die Befehle,
2. *Adressen* zur Anwahl von Speicherzellen und Registern und
3. *Signale* zur Steuerung des Informationsaustausches zwischen den einzelnen Systemkomponenten.

Um den Aufbau der Verbindungswege möglichst flexibel zu halten, sieht man die Daten-, Adreß- und Steuersignalwege nur je einmal vor und stellt sie allen Komponenten zur Informationsübertragung zur Verfügung. Diese Verbindungswege werden als *Busse* bezeichnet, im einzelnen als *Datenbus*, *Adreßbus* und *Steuerbus*, zusammengefaßt als *Systembus*. Ein Bus ist somit zunächst ein Bündel funktional zusammengehörender Signalleitungen, das mindestens zwei Komponenten eines digitalen Systems für den Informationsaustausch miteinander verbindet. Diese Komponenten können einzelne Register sein, aber auch vollständige Funktionseinheiten, wie Mikroprozessoren, Speicher und Ein-/Ausgabeeinheiten. Die Spezifikation für einen Bus umfaßt darüber hinaus dessen funktionelle und elektrische Eigenschaften.

Ein busorientiertes Mikroprozessorsystem zeigt Bild 1-7. Darin ist der Datenbus durch eine grau unterlegte Doppellinie und der Adreßbus durch eine „leere“ Doppellinie dargestellt. Der Steuerbus ist durch eine einzelne, dicke Linie angegeben. Im folgenden behalten wir dieses Schema bei und verzichten auf eine Extra-Kennzeichnung der Busse in den Bildern.

Gegenüber Bild 1-5 sind in Bild 1-7 mehrere Speichereinheiten und mehrere eigenständige Ein-/Ausgabeeinheiten dargestellt, um die modulare Ausbaufähigkeit eines *busorientierten Mikroprozessorsystems* anzudeuten. In dieser relativ einfachen, aber grundlegenden Systemkonfiguration ist der Mikroprozessor die einzige aktive Komponente (*Master*), d.h., nur er kann das Bussystem steuern. Die Speicher- und die Ein-/Ausgabeeinheiten verhalten sich demgegenüber passiv (*Slaves*). Um Konflikte beim Datentransport zu vermeiden, werden immer nur zwei der Systemkomponenten (ein Sender und ein Empfänger) gleichzeitig auf den gemeinsamen Datenbus geschaltet. – Sind an den Systembus auch Ein-/Aus-

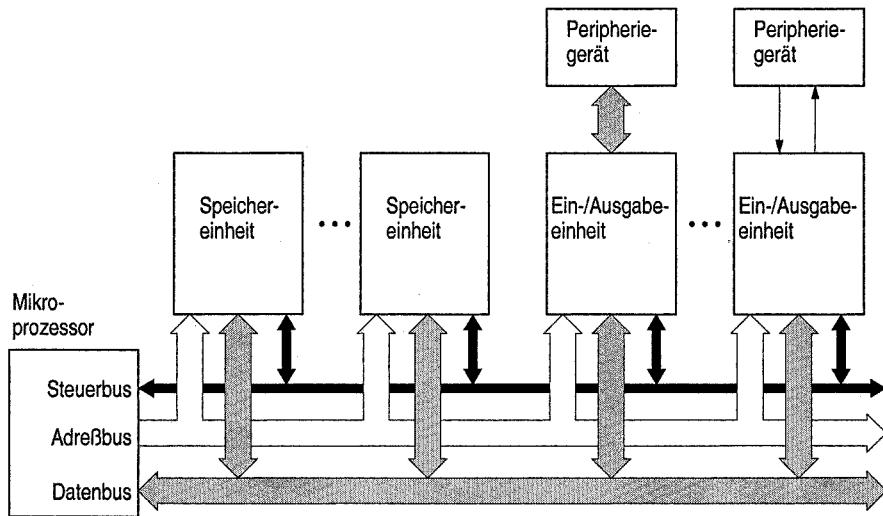


Bild 1-7. Busorientiertes Mikroprozessorsystem

gabeeinheiten mit eigener Übertragungssteuerung angeschlossen, so können auch sie als Master wirken, d.h. die Bussteuerung übernehmen.

Wie in Bild 1-7 zu erkennen ist, erlauben die Datenbusanschlüsse der Systemkomponenten den Datentransport in beiden Richtungen. Solche Signalleitungen bezeichnet man als *bidirektional*. Hingegen werden die Adressbusanschlüsse nur in einer Richtung betrieben; man bezeichnet sie als *unidirektional*. Die meisten Signalleitungen des Steuerbusses sind unidirektional, einige sind bidirektional. Die unidirektionalen Steuerleitungen erlauben in Abhängigkeit von ihrer Funktion den Signalfluss entweder zum Mikroprozessor hinführend oder vom Mikroprozessor herkommend.

Die Vorteile eines busorientierten Mikroprozessorsystems liegen in dessen leichter Erweiterbarkeit bei geringem Aufwand für die gemeinsamen Übertragungswege. Die einzelnen Komponenten des Mikroprozessorsystems werden im folgenden kurz beschrieben.

1.2.3 Mikroprozessor

Die Festlegung der Funktionsweise eines Mikroprozessorsystems liegt in der Vorgabe eines Programms, das im Hauptspeicher steht und vom Mikroprozessor abgearbeitet wird. Die Art der möglichen Operationen des Prozessors ist durch seinen Befehlssatz festgelegt. Die wichtigsten Befehle sind die Befehle für den Transport von Daten, die Befehle für arithmetische und logische Verknüpfungen von Operanden und die Befehle zur Änderung der Abarbeitungsreihenfolge.

Befehlsformate. Für eine zweistellige Operation, d.h. eine Operation, bei der zwei Operanden miteinander zu einem Ergebnis verknüpft werden, sind insgesamt vier Angaben erforderlich:

1. Art der Operation (*Operationscode*),
2. Adresse des ersten Operanden (erste *Quelladresse*),
3. Adresse des zweiten Operanden (zweite Quelladresse),
4. Adresse des Ergebnisses (*Zieladresse*).

Werden diese vier Angaben in einem *Befehl* zusammengefaßt, so entsteht ein *Dreiadreßbefehl* entsprechend Bild 1-8.

Op.-Code	1. Quelladresse	2. Quelladresse	Zieladresse
----------	-----------------	-----------------	-------------

Bild 1-8. Format eines Dreiadreßbefehls

Bei z.B. 8 Bits für den Operationscode und 16 Bits für jede Adresse ergeben sich 56 Bits als Länge eines Befehls. Bezogen auf eine Speicherwortlänge von z.B. 16 Bit würde ein solcher Befehl beim Laden des Programms vier Speicherzellen belegen, wobei in einer Zelle 8 Bits unbenutzt blieben.

Um die Länge eines Befehls zu reduzieren, kann zum einen die Anzahl der Adressen im Befehl verringert werden. Hierzu gibt es zwei Möglichkeiten, die implizite Adressierung und die überdeckte Adressierung, die oft kombiniert angewendet werden. Sie führen zu Einadreß- und Zweiadreßbefehlen. Zum andern kann die Anzahl an Adreßbits reduziert werden, indem der Prozessor mit einem Registerspeicher ausgestattet wird, der als Zwischenspeicher nur eine geringe Speicherkapazität und damit kurze Adressen hat.

Einadreßbefehle. 8-Bit-Mikroprozessoren (heute vorwiegend als Mikrocontroller vorzufinden) besitzen ein spezielles Register, den *Akkumulator*, das bei jeder zweistelligen Operation als Quelle einer der beiden Operanden angesprochen wird. Gleichzeitig wird dieses Register als Ziel für das Ergebnis benutzt, so daß der ursprünglich im Akkumulator gespeicherte Operand damit überschrieben wird. Die Adresse des Akkumulators wird dabei im Befehl nicht explizit angegeben, sondern ist implizit im Operationscode enthalten (*implizite Adressierung*); außerdem fallen durch die Doppelfunktion des Akkumulators zwei Adressen zusammen (*überdeckte Adressierung*). Im Befehl wird neben dem Operationscode lediglich die Quelladresse des zweiten Operanden als Speicheradresse angegeben. Spezielle Lade- und Speicherbefehle ermöglichen den Operandentransport zwischen dem Akkumulator und anderen Registern oder den Speicherzellen. Bild 1-9a zeigt das *Einadreßbefehlsformat* für einen 8-Bit-Mikroprozessor. Bei einem für 8-Bit-Mikroprozessoren üblichen Hauptspeicher mit ausschließlich

byteweisem Zugriff belegt ein solcher Befehl drei aufeinanderfolgende Speicherzellen (Bild 1-9b).

Einadreßbefehlsformate haben den Nachteil, daß zur Durchführung einer zweistelligen Operation oft drei Befehle erforderlich sind:

1. Lade den Akkumulator mit dem Inhalt einer Speicherzelle (erster Operand).
2. Verknüpfe den Inhalt des Akkumulators mit dem Inhalt einer Speicherzelle (zweiter Operand).
3. Speichere den Inhalt des Akkumulators (Ergebnis) in eine Speicherzelle.

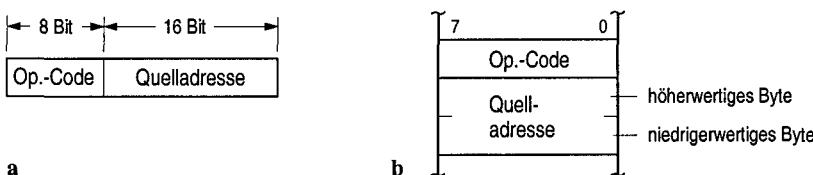


Bild 1-9. Einadreßbefehl eines 8-Bit-Mikroprozessors, **a** allgemeines Format, **b** byteorientierte Darstellung im Speicher

Zweiadreßbefehle. 16- und 32-Bit-Mikroprozessoren haben anstelle des bei 8-Bit-Prozessoren üblichen Akkumulators einen Satz von acht oder 16 allgemein benutzbaren Prozessorregistern, von denen jedes einzelne die Funktion des Akkumulators, aber darüber hinaus auch andere Funktionen übernehmen kann. Zur Anwahl eines Registers wird dabei im Befehl eine *Registeradresse* benötigt, die jedoch viel weniger Bits als eine Speicheradresse erfordert, z.B. bei acht Registern nur drei Bits. Da die Register auf diese Weise allgemein zugänglich sind, bezeichnet man sie auch als allgemeine Register und den Registerspeicher als (allgemeinen) *Registersatz*.

Zweistellige Operationen sehen jetzt die explizite Adressierung beider Operanden vor, behalten jedoch die Überdeckung einer Quelladresse mit der Zieladresse bei. Somit ergibt sich ein Befehlsformat mit zwei expliziten Adreßangaben, die sich wahlweise auf den Registersatz (Registeradressen) oder den Hauptspeicher bzw. andere periphere Einheiten (allg. Speicheradressen) beziehen. Die implizite Adressierung wird bei 16- und 32-Bit-Prozessoren nur für Sonderfälle benutzt, und zwar zur Adressierung einzelner Register, die nicht dem Registerspeicher zugeordnet sind.

Bild 1-10 zeigt ein *Zweiadreßbefehlsformat*. Es umfaßt die beiden expliziten Adreßangaben, wobei (in der hier gewählten vereinfachten Form ohne weitere Adreßmodifikation) jeweils ein Bit (R/\bar{S}) entscheidet, ob es sich um Registeradressen ($R/\bar{S} = 1$) oder um Speicheradressen handelt ($R/\bar{S} = 0$). Die 3-Bit-Registeradressen sind zusammen mit den beiden R/\bar{S} -Bits unmittelbar im ersten Befehlswort angegeben. Die Speicheradressen (hier 16-Bit-Adressen) stehen in

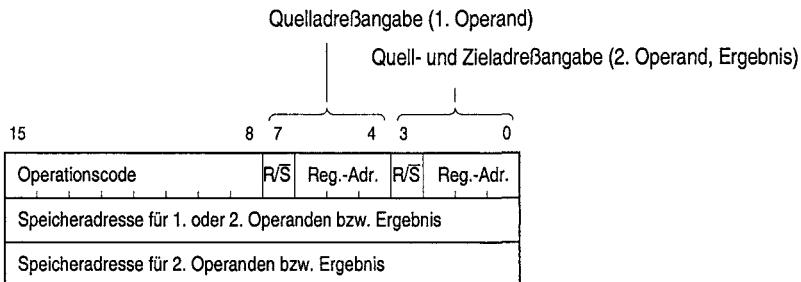


Bild 1-10. Zweiadreßbefehlsformat mit wortorientierter Darstellung im Speicher

den auf das erste Befehlswort folgenden Speicherwörtern. Daraus ergeben sich für dieses Befehlsformat Befehlsdarstellungen als Ein-, Zwei- und Dreiwortbefehle, mit denen sich – bezogen auf die Quell- und Zielangaben – folgende Befehlstypen bilden lassen:

1. Register-Register-Befehle,
2. Register-Speicher-Befehle,
3. Speicher-Register-Befehle,
4. Speicher-Speicher-Befehle.

Anmerkung. Sieht ein Prozessor für die Quell- und die Zieladreßangaben jede Kombination von Register- und Speicheradressen vor, so spricht man auch von *symmetrischer Befehlsstruktur*. Symmetrische Befehle vereinfachen die Programmierung bei nichtkommutativen Operationen, wie z.B. der Subtraktion.

Prozessorstruktur. Um die internen Abläufe eines Mikroprozessors möglichst anschaulich beschreiben zu können, wählen wir die Struktur eines 16-Bit-CISC-Prozessors (Bild 1-11) mit der auch bei 32-Bit-Prozessoren gebräuchlichen 16-Bit-Befehlsstruktur entsprechend Bild 1-10. Zur Vereinfachung beschränken wir uns auf das 16-Bit-Datenformat zur Darstellung von Operanden und Adressen. Der Übergang auf eine 32-Bit-Struktur, auf mehrere Datenformate und auf die Möglichkeiten der Adreßmodifikation erfolgt teilweise in Abschnitt 1.4, generell jedoch in Kapitel 2.

Der Mikroprozessor lässt sich, wie oben erwähnt, in zwei Funktionseinheiten unterteilen, das Operationswerk und das Steuerwerk. Das *Operationswerk* (Bild 1-11, unten) seinerseits besteht aus zwei Teilen: dem Rechenwerk und dem Leitwerk. Das *Rechenwerk* umfasst den Registerspeicher und die *arithmetisch-logische Einheit ALU* (arithmetic and logical unit) mit den beiden Operandenregistern DR1 und DR2 (data registers). Die Datenkommunikation zwischen diesen Komponenten erfolgt über zwei voneinander unabhängige 16-Bit-Datenbusse DB1 und DB2, wovon der letztere mit dem 16-Bit-Datenbus D des prozessorexternen Systembusses verbunden ist.

Die ALU verknüpft Operanden, die zu Beginn der Ausführung eines Befehls in die Operandenregister DR1 und DR2 geladen werden, und erzeugt das Ergebnis sowie Statusinformation (condition code, CC) im *Prozessorstatusregister SR*, z.B. das Carrybit und das Overflowbit (siehe 1.1.4) zur Steuerung von Programmverzweigungen. Quelle und Ziel der Datentransporte sind der (externe)

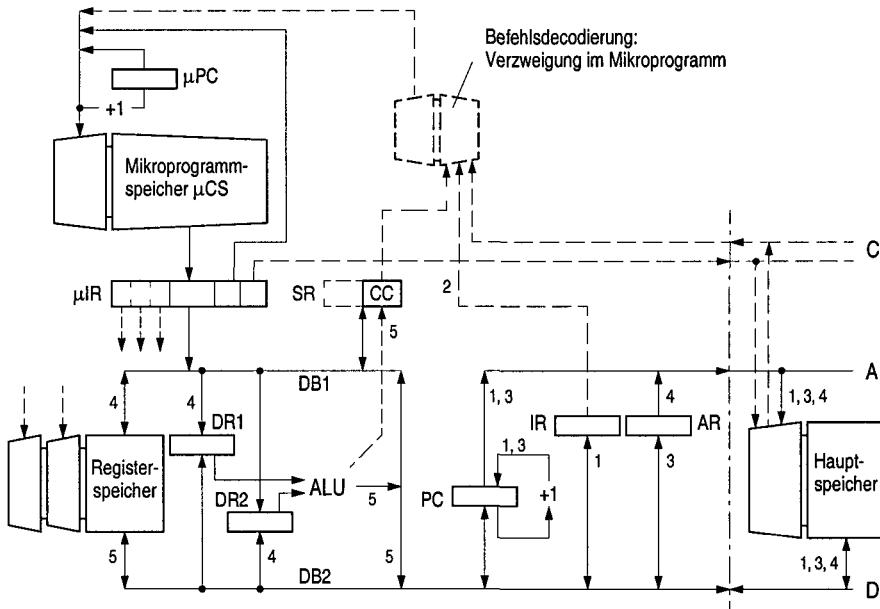


Bild 1-11. Struktur eines 16-Bit-Mikroprozessors (die Ziffern beziehen sich auf Bild 1-14, S. 30). μ PC Mikrobefehlszähler, μ IR Mikrobefehlsregister, PC (Maschinen-)Befehlszähler, IR (Maschinen-)Befehlsregister

Hauptspeicher und der (interne) Registerspeicher. Der Registerspeicher als Zwischenspeicher für Operanden und Ergebnisse erlaubt dem Prozessor aus technischen Gründen einen schnelleren Datenzugriff, als er auf den Hauptspeicher möglich ist. In seiner hier gewählten Eigenschaft als *Zweiport-Speicher* können aus ihm außerdem zwei Operanden gleichzeitig auf die beiden Internbusse gelesen werden.

Das *Leitwerk* umfaßt den Befehlszähler PC (program counter) mit seiner Inkrementierzvorrichtung, das *Befehlsregister* IR (instruction register) zur Speicherung des ersten Befehlswortes des aktuellen Befehls und ein Adreßregister AR zur Speicherung einer Speicheradresse (zweites oder drittes Befehlswort). Die Register PC, IR und AR sind mit dem internen Datenbus DB2 und dem externen Datenbus D verbunden, PC und AR darüber hinaus mit dem externen 16-Bit-Adreßbus A.

Das *Steuerwerk*, genauer Mikroprogrammwerk (Bild 1-11, oben), besteht aus einem Festwertspeicher für die Befehlsdecodierung (Erzeugung der Mikroprogrammstartadressen), einem Mikroprogrammspeicher μ CS (micro control store) mit dem Mikrobefehlszähler μ PC (micro program counter) zur Anwahl der Mikrobefehle und einem Mikrobefehlsregister μ IR (micro instruction register) zur Speicherung des aktuellen Mikrobefehls.

Der *Befehlszähler* PC enthält die aktuelle Befehlsadresse. Sie wird bei aufeinanderfolgenden Befehlen mit jedem Befehlswortzugriff um Eins hochgezählt (hier wortweise Zählung) bzw. bei Programmverzweigungen mit der im Sprungbefehl stehenden Sprungzieladresse geladen. Der Befehlszugriff (instruction fetch) und die Befehlausführung (instruction execution) werden durch das im Mikroprogrammspeicher stehende *Mikroprogramm* als Ablaufschritte vorgegeben. Dazu werden sog. Mikrobefehlssequenzen durchlaufen, deren Startadressen durch den für die Befehlsdecodierung vorhandenen Festwertspeicher erzeugt werden und deren einzelne *Mikrobefehle* durch den Mikrobefehlszähler adressiert und in das Mikrobefehlsregister zur Ausführung geladen werden. Bedingungen zur Bildung der Startadressen dieser Mikrobefehlssequenzen sind je nach Situation neben dem aktuellen Befehlscode aus dem Befehlsregister IR die Bedingungsbits CC des Prozessorstatusregisters und Signale, die auf den Steuerleitungen C (control lines) des Systembusses erscheinen. Der aktuelle Mikrobefehl wirkt mit den in ihm codierten *Mikrooperationen* auf das Operationswerk (Durchschaltung der Datenwege, Vorgabe von ALU- oder Verzweigungsoperationen) und über die Steuerleitungen C auf die prozessorexternen Komponenten.

Befehlszyklus. Der im Mikroprogramm verankerte Ablauf für den Befehlszugriff und die Befehlausführung wird als *Befehlszyklus* bezeichnet. Der Zeittakt für die Ablaufschritte wird von einem Taktgenerator festgelegt und Maschinentakt oder *Prozessortakt* genannt; seine Schrittdauer bezeichnet man als Taktzeit, sein Reziproker Wert ist die Taktfrequenz. Typische Taktfrequenzen von 8-Bit- und 16-Bit-Mikroprozessoren (Mikrocontrollern) liegen zwischen 1 MHz (1 μ s Taktzeit) und 100 MHz (10 ns Taktzeit), heutige 32-Bit-Hochleistungsprozessoren haben die 2-GHz-Marke erreicht und werden sie überschreiten ($\leq 0,5$ ns Taktzeit). Zur begrifflichen Unterscheidung zwischen den für den Programmierer eines Mikroprozessors üblicherweise unzugänglichen Mikrobefehlen einerseits und den für den Programmierer zugänglichen *Maschinenbefehlen* des Mikroprozessors andererseits bezeichnet man den Befehlszyklus auch als *Maschinenbefehlszyklus*.

Bild 1-12 zeigt den Befehlszyklus für eine zweistellige Operation in einer Grobdarstellung. Eine Verfeinerung des Befehlszyklus wollen wir am Beispiel eines Speicher-Register-Befehls, des Additionsbefehls ADD SPADR,R5, vornehmen. In der symbolischen Befehlsschreibweise verwenden wir als suggestive Abkürzungen (*mnenomics*) ADD zur Bezeichnung des Operationscodes und die beiden Adresssymbole SPADR und R5 zur Bezeichnung einer Hauptspeicherzelle und eines Prozessorregisters mit der Adresse 5. Der Befehl hat folgende Wirkung: Der

- 1 Transport des Befehls vom Hauptspeicher in das Befehlsregister.
Erhöhen des Befehlszählers
- 2 Decodieren des Befehls
- 3 Transport des ersten Operanden vom Hauptspeicher oder dem Registerspeicher in das Operationswerk
- 4 Transport des zweiten Operanden vom Hauptspeicher oder dem Registerspeicher in das Operationswerk
- 5 Ausführen der Operation durch Verknüpfen der Operanden
- 6 Transport des Ergebnisses vom Rechenwerk in den Hauptspeicher oder den Registerspeicher

Bild 1-12. Grobdarstellung eines Befehlszyklus

Inhalt der Speicherzelle SPADR (Quelle) wird zum Inhalt des Registers 5 (Quelle) addiert und das Ergebnis in das Register 5 (Ziel) geschrieben; dabei bleibt der Inhalt der Zelle SPADR unverändert.

In Bild 1-13 ist der Additionsbefehl in symbolischer sowie in binärer und hexadezimaler Schreibweise gemäß dem Befehlsformat in Bild 1-10 wiedergegeben. Als Binärkode für ADD wurde das Bitmuster 00000011 und für die symbolische Adresse SPADR der Dezimalwert 16 gewählt. Die symbolische Schreibweise ist für den Menschen gut verständlich, während die binäre Schreibweise der Darstellung im Speicher entspricht und vom Mikroprozessor unmittelbar interpretiert werden kann. Die hexadezimale Schreibweise ist gegenüber der binären Schreibweise übersichtlicher und erspart Schreibarbeit bei der Darstellung von Befehlen und Operanden als Speicherinhalte.

Symbolisch	Binär	Hex.
ADD SPADR, R5	0000 0011 0000 1101 0000 0000 0001 0000	030D 0010

Bild 1-13. Drei verschiedene Schreibweisen eines Additionsbefehls

Bild 1-14 zeigt den Befehlszyklus des Additionsbefehls, aufgeschlüsselt nach den einzelnen Taktstufen und deren Mikrooperationen. Transportoperationen sind durch das Symbol → gekennzeichnet; als Quell- und Zielangaben dienen die Kurzbezeichnungen aus Bild 1-11, in das zur Verdeutlichung des Verarbeitungsablaufs die korrespondierenden Nummern der Taktstufen eingetragen sind. In Bild 1-14 gehen wir davon aus, daß der Zugriff auf eine externe Speicherzelle einen Taktstufe erfordert (in Wirklichkeit erfordert er mehr als einen Taktstufe). Eine einfache, einschrittige ALU-Operation, wie die Addition, benötigt ebenfalls einen Taktstufe; im selben Schritt wird das Ergebnis in den Registerspeicher oder ggf. in den Hauptspeicher geschrieben. Damit ergibt sich für den

1	$PC \rightarrow \text{Hauptspeicher} \rightarrow IR$ $PC+1 \rightarrow PC$	Lesen des ersten Befehlsworts aus dem Hauptspeicher
2	Befehlsdecodierung	Auswerten des Operationscodes und der Adressierungsarten
3	$PC \rightarrow \text{Hauptspeicher} \rightarrow AR$ $PC+1 \rightarrow PC$	Lesen der Adresse des ersten Operanden aus dem Hauptspeicher
4	$AR \rightarrow \text{Hauptspeicher} \rightarrow DR2$ Registerspeicher $\rightarrow DR1$	Lesen der Operanden aus dem Hauptspeicher und dem Registerspeicher
5	$DR2+DR1 \rightarrow \text{Registerspeicher}$ Statusinformation $\rightarrow SR$	Ausführen der Addition, Schreiben des Ergebnisses in den Registerspeicher und der Statusinformation nach SR (CC-Bits)

Bild 1-14. Befehlszyklus des Additionsbefehls als Speicher-Register-Befehl (Abkürzungen siehe Bild 1-11, S. 27). Wortzählung bei der Speicheradressierung durch den Befehlszähler

Additionsbefehl als Speicher-Register-Befehl eine Gesamtausführungszeit von 5 Taktten, während er als Register-Register-Befehl 4 und als Speicher-Speicher-Befehl 7 Takte benötigt. – Die Ausführungszeit eines Befehls hängt nicht nur von der Operandenadressierung ab, sondern auch von der Art der Operation. Sie erhöht sich bei Befehlen mit mehreren Verarbeitungsschritten in der ALU, z. B. beim Multiplikationsbefehl um ca. 16 Takschritte.

1.2.4 Speicher

Als Hauptspeicher für Daten und Programme werden in Mikroprozessorsystemen hauptsächlich Halbleiterspeicher mit wahlfreiem Zugriff eingesetzt. Darüber hinaus werden sog. Massenspeicher als Hintergrundspeicher verwendet, bei denen der Zugriff sequentiell auf jeweils einen gesamten Datenblock erfolgt. Solche Speicher sind z.B. Magnetbandspeicher (Streamer-Tape-Speicher), Magnetfolienspeicher (Floppy-Disk-Speicher), Magnetplattenspeicher (Festplatten- und Wechselplattenspeicher) und optische Plattenspeicher (z.B. CD-ROM-Speicher). – Wir wollen uns im folgenden mit verschiedenen Arten von Halbleiterspeichern befassen, die sich grob in Schreib-/Lesespeicher, Festwertspeicher und Assoziativspeicher einteilen lassen (siehe auch 1.2.1).

Schreib-/Lesespeicher. Ein Schreib-/Lesespeicher (*random access memory, RAM*) kann, wie sein Name sagt, vom Mikroprozessor sowohl beschrieben als auch gelesen werden. Die Speicherung der Information erfolgt hierbei abhängig von der Ausführung des Bausteins in 1-Bit-, 4-Bit-, 8-Bit-, 16-Bit oder 32-Bit-Speicherzellen, die über einen Decodierer adressierbar sind (Bild 1-15a). Eine Steuerlogik ermöglicht über eine Bausteinanwahlleitung die Bausteinaktivierung und über eine Lese-/Schreibleitung die Vorgabe der Datentransportrichtung. Die Anpassung der Datensignale an die Bausteinumgebung erfolgt über einen *bidi-*

rekationalen Datenbustreiber (bus driver, buffer), der über die Steuerlogik in Lese- bzw. Schreibrichtung durchgeschaltet wird (Dreiecksymbole). – Zur Erreichung einer Speicherwortlänge von z.B. 16 oder 32 Bit werden ggf. mehrere solcher Bausteine nebeneinander angeordnet aufgebaut.

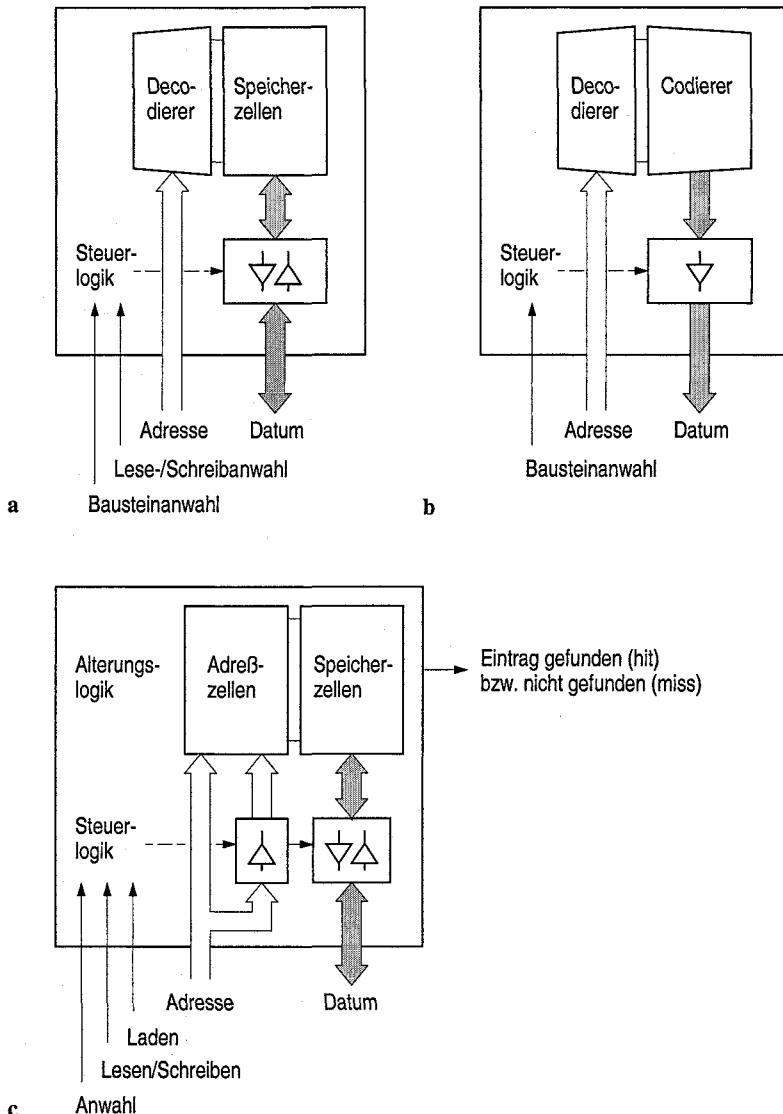


Bild 1-15. Speicherbausteine. **a** Speicher mit wahlfreiem Zugriff (RAM), **b** Festwertspeicher (ROM), **c** Assoziativspeicher (CAM)

Bei Schreib-/Lesespeichern lassen sich zwei Prinzipien der Informationsspeicherung unterscheiden. Bei den *statischen RAMs (SRAMs)* werden die einzelnen Bits in Rückkopplungsspeichern (Flipflops) gespeichert. Der Speichereffekt liegt dabei in den Schaltzuständen zweier Transistoren, von denen jeweils einer leitend und der andere gesperrt ist. Bei den *dynamischen RAMs (DRAMs)* hingegen wird jedes Bit als elektrische Ladung in einem Kondensator gespeichert (Energiespeicher). Da innerhalb der Speicherelemente Ladungsausgleiche stattfinden, müssen die Ladungen ständig aufgefrischt werden, wozu eine zusätzliche Speicheransteuerung erforderlich wird (Refresh-Einrichtung). Ein einzelner Refresh-Vorgang wirkt jeweils auf einen größeren Speicherbereich.

Die vom Anlegen der Adresse bis zur Datenübernahme (Schreiben) bzw. Datenbereitstellung (Lesen) erforderliche Zeit ist die *Zugriffszeit* eines Speichers, die kürzest mögliche Zeit zweier aufeinanderfolgender Schreib- oder Lesezugriffe dessen *Zykluszeit*. Bei SRAMs sind beide Zeiten annähernd gleich; bei DRAMs ist die Zykluszeit wesentlich größer als die Zugriffszeit, da Schreib- und Lesezugriffe speicherintern immer als Lesezugriff mit Rückschreiben ausgeführt werden. Dieser Zyklus schließt einen automatischen Refresh-Vorgang ein. Zu Struktur und Ansteuerung von RAM-Bausteinen siehe auch 6.1.1. – Gegenüber den statischen haben die dynamischen RAMs den Vorteil größerer Speicherkapazität pro Speicherbaustein. Typische Speicherkapazitäten von DRAMs liegen gegenwärtig bei 16, 64 und 256 Mbit, von SRAMs bei 4 und 16 Mbit.

Festwertspeicher. Die in einem Festwertspeicher (*read only memory, ROM*) abgelegte Information kann vom Mikroprozessor zwar gelesen, aber nicht verändert werden. Dementsprechend sind die Speicherzellen, wie Bild 1-15b zeigt, als Codewörter und der Datenbusstreiber als *unidirekionaler* Verstärker ausgebildet. Außerdem entfällt die Lese-/Schreibleitung. Die Speicheradressierung erfolgt wie beim Schreib-/Lesespeicher wahlfrei über einen Decodierer.

Das Festlegen der Binärinformation in einem Festwertspeicher erfolgt vor der Inbetriebnahme des Systems und wird als Programmieren des ROMs bezeichnet. Dabei werden verschiedene Prinzipien angewendet. Bei den einfachen ROMs wird die zu speichernde Information bei der Bausteinherstellung eingebracht. Dazu wird eine Maske verwendet, die die Struktur der Binärinformation enthält. Die einmal eingebrachte Information kann nachträglich nicht mehr verändert werden.

Bei den programmierbaren ROMs (programmable ROMs, PROMs) kann die Information vom Anwender über spezielle Programmiergeräte eingeschrieben werden. Dazu sind die Speicherelemente für die Codewörter programmierbar ausgelegt und können durch Anlegen ihrer Adresse angewählt werden. Durch einen Spannungsstoß werden in Abhängigkeit von den Bits des anliegenden Datenworts Verbindungen in den Speicherelementen des Codeworts entweder zerstört (z.B. durchgebrannt), oder sie bleiben erhalten. Eine andere Programmiermöglichkeit bieten die löschenkbaren PROMs (erasable programmable ROMs, EPROMs), die

ebenfalls vom Anwender programmiert werden können. Bei ihnen wird die Information in der Form elektrischer Ladungen gespeichert, die sich durch Bestrahlung mit ultraviolettem Licht oder durch Anlegen einer Löschspannung (electrically alterable ROMs, EAROMs; electrically erasable ROMs, EEPROMs) wieder löschen lassen. Der Speicher befindet sich danach wieder im programmierbaren Zustand. – Die Speicherkapazität von ROM-Bausteinen liegt derzeit bei 4 Mbit.

ROMs werden aufgrund ihrer Nur-Lese-Eigenschaft zur Speicherung unveränderlicher Information (Programme, konstante Daten) verwendet. RAMs sind dagegen universell verwendbar und werden insbesondere zur Speicherung von veränderbaren Daten (Variablen) benötigt. Beim Ausfall der Versorgungsspannung bleiben die ROM-Inhalte erhalten, während die RAM-Inhalte verlorengehen. Sollen auch die RAM-Inhalte erhalten bleiben, so muß der Ausfall durch eine zusätzliche Spannungsversorgung, z. B. durch Batterien, abgefangen werden.

Assoziativspeicher. Ein Assoziativspeicher (*content addressable memory*, CAM) ist dadurch charakterisiert, daß bei ihm zusätzlich zu den Speicherzellen auch die den Speicherzellen zugeordneten Adreßzellen beschrieben werden können (Bild 1-15c). Darüber hinaus ist bei ihm die Adreßlänge sehr viel größer, als sie gemessen an der Anzahl der Speicherzellen, d.h. der Kapazität des Speichers, erforderlich wäre. Diesen Aspekt, der es ermöglicht, einen großen Adreßraum auf einen kleineren Adreßraum abzubilden, macht man sich zunutze, um den Assoziativspeicher als „schnellen“ Zwischenspeicher für einen Speicher sehr viel größerer Kapazität einzusetzen, z.B. als Zwischenspeicher zwischen Prozessor und Hauptspeicher. Gespeichert werden in diesem Fall Hauptspeicherinhalte und deren Hauptspeicheradressen. Man bezeichnet einen solchen Speicher, da seine Wirkungsweise für den Programmierer verborgen bleibt, als *Cache* (Versteck, unterirdisches Depot).

Bei einem Speicherzugriff wird die Hauptspeicheradresse an den Adreßeingang des Cache gelegt. Die Adreßdecodierung erfolgt durch Vergleich dieser Adresse mit sämtlichen in den Adreßzellen gespeicherten Adressen. Wird eine Übereinstimmung mit einer der gespeicherten Adressen festgestellt, so gilt dies als „Treff“ (Eintrag gefunden, cache hit), d.h., der Speicherzugriff kann über den Cache abgewickelt werden, indem die zugehörige Speicherzelle beschrieben oder gelesen wird. Besteht keine Übereinstimmung mit einer der gespeicherten Adressen, so gilt dies als „Fehlzugriff“ (cache miss) und der Zugriff muß über den Hauptspeicher abgewickelt werden. Üblicherweise wird dabei der im Cache fehlende Hauptspeicherinhalt zusammen mit dessen Adresse in den Cache nachgeladen.

Beim Nachladen tritt das Problem auf, daß eine Entscheidung getroffen werden muß, welcher der bereits vorhandenen Inhalte (Speicherzelle und Adreßzelle) durch den neuen Inhalt überschrieben werden soll. Als Lösungen gibt es diverse Ersetzungsstrategien, die sich in ihrem Hardwareaufwand und davon abhängig in ihrer Güte, die Trefferrate eines Cache zu optimieren, unterscheiden. Gute Strategien setzen die Zellen zueinander in eine Altersbeziehung, z.B. bezüglich des

Zeitpunktes des letzten Zugriffs auf eine Zelle oder bezüglich des Zeitpunktes des Ladens des Inhalts der Zelle. Ersetzt wird beim Nachladen dann jene Zelle mit dem „ältesten Zugriff“ bzw. dem „ältesten Inhalt“. Die Altersangaben der einzelnen Zellen müssen natürlich mit jedem Zugriff bzw. mit jedem Ladevorgang mittels einer Alterungslogik aktualisiert werden.

Assoziativspeicher haben aufgrund ihrer komplexen Hardware für die Adreß-decodierung (u.a. komplette Vergleicher für jede Adreßzelle) üblicherweise relativ geringe Kapazitäten, z.B. von 64, 128 oder 256 Speicherzellen. Diese reichen für den genannten Einsatz dieser Speicher als Zwischenspeicher für Hauptspeicher nicht aus. Größere Kapazitäten werden bei verringelter Wirksamkeit erst dadurch wirtschaftlich realisierbar, daß man den hier beschriebenen sog. „voll“-assoziativen Cache modifiziert und ihn zu einem „teil“-assoziativen Cache vereinfacht. Siehe dazu und zur Cache-Problematik insgesamt die Ausführungen in Abschnitt 6.2.

1.2.5 Ein-/Ausgabeeinheit

Ein-/Ausgabeeinheiten bilden die Schnittstellen zwischen dem Mikroprozessorsystem, d.h. dem Systembus, und den Ein-/Ausgabegeräten und Hintergrundspeichern. Elementarer Bestandteil solcher Einheiten sind Interface-Bausteine (kurz *Interfaces*). Im einfachsten Fall bestehen sie aus einem anwählbaren Datenregister zur Zwischenspeicherung der zu übertragenden Daten. Im allgemeinen übernehmen sie darüber hinaus Steuerfunktionen zur Synchronisation der Datenübertragung. Außer zur Datenübertragung werden sie auch für den Austausch reiner Steuer- und Statusinformation zwischen dem Mikroprozessor und z.B. einem zu steuernden technischen Prozeß eingesetzt.

Für einen Interface-Baustein ergeben sich unterschiedliche Betriebsarten, die durch Laden eines oder mehrerer Steuerregister vorgegeben werden. Die Betriebszustände eines solchen Bausteins werden in einem oder mehreren Statusregistern angezeigt. Sämtliche Register, also Datenregister, Steuerregister und Statusregister, können wie Speicherzellen adressiert, beschrieben und gelesen werden. Das geschieht entweder durch die normalen Speicherbefehle oder durch spezielle Ein-/Ausgabebefehle.

Die Datenübertragung zwischen Mikroprozessor und Interface-Bausteinen erfolgt parallel, vorwiegend im Byteformat, d.h., für jedes Datenbit eines Zeichens steht eine eigene Datenleitung des Systembusses zur Verfügung. Die Datenübertragung zwischen Interface-Baustein und Peripherie wird, angepaßt an die Peripherie, entweder bitparallel oder bitseriell durchgeführt. Den dabei notwendigen verschiedenen Synchronisations- und Übertragungsarten wird durch eine Vielzahl unterschiedlicher Interface-Bausteine und Betriebsarten Rechnung getragen. Im Prinzip unterscheidet man hier einfache Interface-Bausteine für die Übertragung

einzelner Daten, z.B. als Schnittstelle zu einer Tastatur und einer Bildschirmleinheit (serielles Interface) oder zu einem Drucker (serielles oder paralleles Interface), und komplexere Interface-Bausteine mit erweiterter Steuerung für das blockweise Übertragen von Daten, z.B. als Schnittstelle zu einem Floppy-Disk- oder Festplatten-Laufwerk (floppy-disk controller, hard-disk controller) oder zu einem lokalen Rechnernetz (local area network controller).

Interface-Bausteine sind an sich passive Ein-/Ausgabeeinheiten (*Slaves*). Das heißt, der Ablauf einer Datenübertragung wird vom Mikroprozessor gesteuert, indem er z.B. das Starten und Stoppen eines Geräts übernimmt und die zu übertragenden Daten nacheinander in das Interface-Datenregister schreibt (Ausgabe) oder von dort liest (Eingabe). Dabei synchronisiert er sich mit der Peripherie über die Statusinformation des Bausteins. Interface-Bausteine können darüber hinaus durch zusätzliche Steuereinheiten zu aktiven Ein-/Ausgabeeinheiten (*Mastern*) erweitert werden. Diese sind dann in der Lage, die Übertragung von Datenblöcken nach vorheriger Vorgabe von Steuerinformation durch den Mikroprozessor (Initialisierung) selbstständig durchzuführen. Man spricht hierbei von *Direktspeicherzugriff* (*direct memory access*, DMA) und bezeichnet die zusätzliche Steuerseinheit als *DMA-Controller* (DMAC).

In einer zusätzlichen Erweiterung werden zur Durchführung der Ein-/Ausgabe *Ein-/Ausgabeprozessoren* eingesetzt, die sowohl Programme ausführen als auch den Direktspeicherzugriff durchführen können. Sie übernehmen die Steuerung des gesamten Ein-/Ausgabevorgangs einschließlich des Startens und des Stoppons von Geräten. Schließlich werden für die Ein-/Ausgabe universell programmierbare *Ein-/Ausgaberechner* eingesetzt, die ihrerseits mit Interface-Bausteinen und DMA-Controllern ausgestattet sind. – Ausführliche Beschreibungen zu Ein-/Ausgabeeinheiten und Ein-/Ausgabetechniken finden sich in den Kapiteln 7 und 8.

Für eine vertiefende Betrachtung des elektrotechnischen und logischen Aufbaus von Prozessoren, Halbleiterspeichern und Interface-Bausteinen siehe z.B. [Mead, Conway 1980, Klar 1996, Liebig 2003].

1.3 Assemblerprogrammierung (CISC)

Bei der Programmierung von Mikroprozessorsystemen wird zur leichten Handhabung eine symbolische Schreibweise für die Befehlsdarstellung gewählt. In der hardware-nächsten Ebene, der sog. Assemblerebene, entspricht dabei ein symbolischer Befehl gerade einem Maschinenbefehl. Die symbolische Schreibweise wird *AssemblerSprache* genannt. Sie ist durch den Befehlssatz des Mikroprozessors geprägt, jedoch in Symbolik und Befehlsdarstellung (Notation, Syntax) von der Hardware unabhängig. Die Umsetzung eines *Assemblerprogramms* (Assemblercode) in ein *Maschinenprogramm* (*Maschinencode*) übernimmt ein Überset-

zungsprogramm, der *Assembler*. Um die symbolische Programmbeschreibung für den Assembler lesbar zu machen, werden die Zeichen des Programmtextes vom Eingabegerät, z.B. einer Tastatur, im ASCII übertragen. Auf gleiche Weise wird eine vom Assembler erzeugte Programmliste im ASCII an einen Drucker oder an den Bildschirm eines Terminals ausgegeben.

1.3.1 Programmdarstellung

Im folgenden wird die Darstellung von Programmen in der für den Menschen verständlichen Assemblerschreibweise und in der für den Mikroprozessor verarbeitbaren Maschinencodierung betrachtet. Wir gehen dazu von einer einfachen Aufgabenstellung aus und legen einen für die Lösung der Aufgabe ausreichenden Satz von Maschinenbefehlen fest.

Programmieraufgabe. Mit einem Mikroprozessorsystem soll ein Impulsgeber aufgebaut werden, der in konstanten Zeitabständen Impulse an eine peripherie Einheit abgibt. Dazu wird ein Interface-Baustein mit einem 16-Bit-Datenregister verwendet, in dem zu Beginn der Programmausführung von der peripheren Einheit eine Zahl bereitgestellt wird, die die Periodendauer t der Impulsfolge vorgibt (Bild 1-16). Der Mikroprozessor soll, nachdem er diese Zahl übernommen hat, in

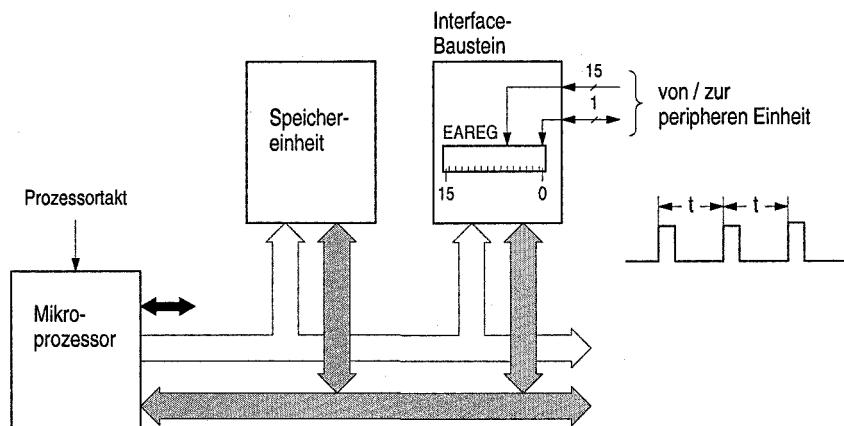


Bild 1-16. Aufbau eines Impulsgebers mit einem Mikroprozessor

dasselbe Register zunächst eine Null und dann im Abstand der Impulse für die Dauer eines Befehls eine Eins ausgeben. Der Inhalt von Bit 0 des Registers bildet das Impulssignal für die Peripherie. Das Datenregister bezeichnen wir mit der symbolischen Adresse EAREG und weisen ihm die absolute Adresse 32768 (2^{15}) zu.

Den Programmablauf zur Lösung dieser Aufgabe zeigt Bild 1-17 in der Form eines *Flußdiagramms*. Zunächst wird die Zeitkonstante t aus dem Datenregister EAREG in eine Hauptspeicherzelle ZEITK übernommen, danach wird das Datenregister mit dem Wert 0 für die anschließende Impulsausgabe initialisiert (Bit 0 = 0). Die Erzeugung eines Impulses erfolgt durch die wiederholte Übertragung der Werte 1 und 0 an das Datenregister, das den jeweils zuletzt übertragenen Wert so lange speichert, bis er durch die nächste Übertragung überschrieben wird. Beide Werte stehen in den Speicherzellen NULL und EINS als konstante Operanden.

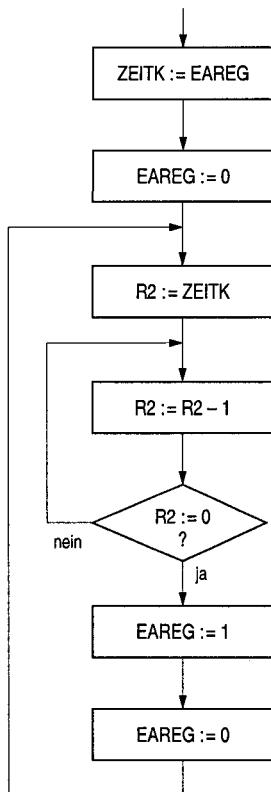


Bild 1-17. Flußdiagramm für die Impulsausgabe

den zur Verfügung. Zur wiederholten Ausgabe der Werte wird der entsprechende Programmteil wiederholt durchlaufen, man sagt, er bildet eine *Programmschleife*. Diese Schleife enthält eine weitere, innere Programmschleife, die so oft durchlaufen wird, wie es der Wert der Zeitkonstanten ZEITK vorgibt. Das heißt, der Zeitabstand t zwischen zwei Impulsen wird bestimmt durch die Verarbeitungszeiten der einzelnen Befehle und die Anzahl der Schleifendurchläufe der inneren Schleife. Die Verarbeitungszeiten der Befehle sind durch die Anzahl der Takt-schritte pro Befehl und die Taktzeit des Mikroprozessors festgelegt.

Symbolische Programmdarstellung. Die folgende Liste zeigt einen Ausschnitt des Mikroprozessor-Befehlssatzes mit den Befehlen, die zur Lösung dieser Aufgabe benötigt werden. Die Befehle sind in symbolischer Schreibweise, wie sie bereits für den Additionsbefehl verwendet wurde, und mit einer Kurzbeschreibung ihrer Funktionen angegeben.

MOVE	QADR,ZADR	Transportbefehl (move): Transportiert den Inhalt von QADR (Quelladresse) nach ZADR (Zieladresse).
SUB	QADR,ZADR	Subtraktionsbefehl (subtract): Subtrahiert den Inhalt von QADR vom Inhalt von ZADR und schreibt das Ergebnis nach ZADR.
CMP	ADR1,ADR2	Vergleichsbefehl (compare): Vergleicht die mit ADR1 und ADR2 adressierten Operanden und speichert die Aussage, ob der zweite Operand größer, größer/gleich, gleich, ungleich, kleiner/gleich oder kleiner dem ersten Operanden ist, in den CC-Bits des Prozessorstatusregisters.
BNE	SPRADR	Bedingter Sprungbefehl (branch if not equal): Lädt den Befehlszähler mit der Sprungadresse SPRADR, sofern die CC-Bits des Prozessorstatusregisters den Zustand „ungleich“ anzeigen, sonst wird der nächste Befehl im Programm ausgeführt.
JMP	SPRADR	Unbedingter Sprungbefehl (jump): Lädt den Befehlszähler mit der Sprungadresse SPRADR.

Bild 1-18 zeigt das symbolische Programm in maschinenexterner Darstellung entsprechend dem Flußdiagramm Bild 1-17. Die innere Programmschleife ist mit dem bedingten Sprungbefehl BNE und die äußere Programmschleife mit dem un-

Symbolisches Programm		Takte/Befehl	
	MOVE	EAREG , ZEITK	6
	MOVE	NULL , EAREG	6
	MOVE	NULL , R0	4
M1 :	MOVE	ZEITK , R2	4
M2 :	SUB	EINS , R2	5
	CMP	R0 , R2	4
	BNE	M2	3
	MOVE	EINS , EAREG	6
	MOVE	NULL , EAREG	6
	JMP	M1	3
ZEITK			
NULL		0	
EINS		1	

Bild 1-18. Impulsgeberprogramm in symbolischer Darstellung

bedingten Sprungbefehl JMP abgeschlossen. Die äußere Schleife kann wegen der Verwendung des JMP-Befehls nicht verlassen werden und bildet somit eine End-

losschleife. Für die Abfrage auf Null in der inneren Schleife wird, in Erweiterung der Ablaufbeschreibung durch das Flußdiagramm, das Register R0 verwendet, das zuvor mit Null geladen wird.

Die in den Sprungbefehlen verwendeten symbolischen Adressen M1 und M2 markieren in der linken Spalte diejenigen Befehle im Programm, die die Sprungziele darstellen. Auf gleiche Weise markieren die Symbole NULL, EINS und ZEITK die am Ende des Programms definierten konstanten Operanden 0 und 1 und den Speicherplatz für die Zeitkonstante. Mit diesen Marken (*labels*, Namen) können die zur Überführung des Programms in den Maschinencode notwendigen Adreßzuordnungen hergestellt werden. Beim späteren Laden des Programms in den Hauptspeicher schließt sich der Datenbereich unmittelbar an den Programmablaufbereich an (wie in der symbolischen Darstellung). – *Anmerkung:* Marken, die vor Befehlen stehen, d.h. mögliche Sprungadressen, werden der Anschaulichkeit halber um einen Doppelpunkt ergänzt.

Als Beispiel für die Ermittlung einer bestimmten Zeitkonstanten ZEITK geben wir den Impulsabstand mit $t = 0,005 \text{ s}$ und die Taktzeit mit $0,04 \mu\text{s}$ (25 MHz-Takt) vor. Für die innere Programmschleife ergibt sich mit den Angaben aus Bild 1-18 eine Durchlaufzeit von 12 Taktstufen, d.h. von $0,48 \mu\text{s}$. Diese Schleife wird so oft durchlaufen, wie der Wert von ZEITK angibt, d.h. $0,48 \cdot \text{ZEITK} \mu\text{s}$. In der äußeren Schleife kommen weitere 19 Maschinentakte, d.h. $0,76 \mu\text{s}$ dazu. Der Impulsabstand t errechnet sich damit zu $t = (0,48 \cdot \text{ZEITK} + 0,76) \mu\text{s}$. Daraus ergibt sich der Wert von ZEITK und damit die Anzahl der inneren Schleifendurchläufe zu $\text{ZEITK} = (t - 0,76) / 0,48 \approx 10415$.

Maschinencode-Darstellung. Unter Zugrundelegung unserer Befehlsformate wollen wir das symbolische Programm in seine maschineninterne Form, den *Maschinencode*, übersetzen. Der Maschinencode ist die Darstellung, in der ein Programm und seine Daten im Hauptspeicher vorliegen und die vom Mikroprozessor unmittelbar interpretiert werden kann. Dazu müssen die symbolischen Angaben durch ihre äquivalenten binären Darstellungen ersetzt werden.

Die Zuordnung der symbolischen Operationscodes zu den maschineninternen Operationscodes (Binärcodes) liegt fest und bildet die für unseren Mikroprozessor in Tabelle 1-8 dargestellte *Zuordnungstabelle*. Die Ersetzung der *symbolischen Speicheradressen* durch numerische Adressen ergibt sich aus der Lage des Programms und seiner Daten im Hauptspeicher. Die *numerischen Adressen* der allgemeinen Register und des Datenregisters der Ein-/Ausgabeeinheit liegen hingegen fest. Aus diesem Grund hatten wir bereits früher das Symbol Rn den allgemeinen Registern n zugeordnet ($n = 0, 1, \dots, 7$); desgleichen hatten wir dem Symbol EAREG die Adresse 32768 fest zugewiesen, die durch die Adreßdecodierung des Interface-Bausteins vorgegeben ist.

Für unser Beispiel wollen wir voraussetzen, daß das Programm den Hauptspeicher ab der Zelle 0 belegt, und erhalten damit die in Tabelle 1-9 angegebene

Tabelle 1-8. Zuordnungstabelle für die Operationscodes

Symbolischer Operationscode	Binärcode
MOVE	0000 0001
SUB	0000 0010
CMP	0010 0001
BNE	0000 1100
JMP	0000 0101

Adressezuordnung als *Symboltabelle*. Bei der Festlegung der Werte der symbolischen Adressen wurden die unterschiedlichen Befehslängen (Ein-, Zwei- und Dreiwortbefehle) berücksichtigt.

Als letztes bestimmen wir für jeden Befehl die beiden R/S-Bits aus der Angabe, ob es sich bei den rechts vom Operationscode stehenden Adressangaben um Registeradressen ($R/S = 1$) oder um Speicheradressen ($R/S = 0$) handelt. Hierbei werden alle Adresssymbole, die nicht den Registerspeicher betreffen, als Speichersymbole (genauer: als Symbole für den prozessorexternen Datenzugriff) aufgefaßt.

Tabelle 1-9. Symboltabelle für das Impulsgeberprogramm

Symbol	Numerische Adresse		
	Dual	Dezimal	Hex.
M1	0000 0000 0000 1000	8	0008
M2	0000 0000 0000 1010	10	000A
ZEITK	0000 0000 0001 0111	23	0017
NULL	0000 0000 0001 1000	24	0018
EINS	0000 0000 0001 1001	25	0019
EAREG	1000 0000 0000 0000	32768	8000

Tabelle 1-10 zeigt den endgültigen Maschinencode zusammen mit den Speicheradressen der einzelnen Maschinencodewörter, wobei die Dualzahldarstellung der Speicheradressen aus Gründen der Übersichtlichkeit von 16 auf 8 Stellen reduziert wurde.

Der Programmcode belegt die Zellen 0 bis 22 und der Datenbereich die Zellen 23 bis 25. Die beiden Operanden NULL und EINS sind als Dualzahlen codiert; der Inhalt des Speicherwortes ZEITK ist vor Ausführung des Programms noch unbestimmt, was durch die x-Reihe angedeutet ist. Als Orientierungshilfe ist zu jedem Befehlswort der symbolische Operationscode bzw. die symbolische Speicheradresse angegeben.

Tabelle 1-10. Impulsgeberprogramm in Maschinencode

Adresse		Maschinencode		Symbol
Dez.	Dual	Dual	Hex.	
0	0000 0001	0000 0001 0000 0000	0100	MOVE
1	0000 0001	1000 0000 0000 0000	8000	EAREG
2	0000 0010	0000 0000 0001 0111	0017	ZEITK
3	0000 0011	0000 0001 0000 0000	0100	MOVE
4	0000 0100	0000 0000 0001 1000	0018	NULL
5	0000 0101	1000 0000 0000 0000	8000	EAREG
6	0000 0110	0000 0001 0000 1000	0108	MOVE
7	0000 0111	0000 0000 0001 1000	0018	NULL
8	0000 1000	0000 0001 0000 1010	010A	MOVE
9	0000 1001	0000 0000 0001 0111	0017	ZEITK
10	0000 1010	0000 0010 0000 1010	020A	SUB
11	0000 1011	0000 0000 0001 1001	0019	EINS
12	0000 1100	0010 0001 1000 1010	218A	CMP
13	0000 1101	0000 1100 0000 0000	0C00	BNE
14	0000 1110	0000 0000 0000 1010	000A	M2
15	0000 1111	0000 0001 0000 0000	0100	MOVE
16	0001 0000	0000 0000 0001 1001	0019	EINS
17	0001 0001	1000 0000 0000 0000	8000	EAREG
18	0001 0010	0000 0001 0000 0000	0100	MOVE
19	0001 0011	0000 0000 0001 1000	0018	NULL
20	0001 0100	1000 0000 0000 0000	8000	EAREG
21	0001 0101	0000 0101 0000 0000	0500	JMP
22	0001 0110	0000 0000 0000 1000	0008	M1
23	0001 0111	xxxx xxxx xxxx xxxx		
24	0001 1000	0000 0000 0000 0000	0000	
25	0001 1001	0000 0000 0000 0001	0001	

1.3.2 Programmübersetzung (Assemblerung)

Ein Vergleich der beiden Programmdarstellungen in Bild 1-18 und Tabelle 1-10 zeigt, daß die symbolische Darstellung für den Menschen sehr viel besser lesbar ist als die Maschinencode-Darstellung, die der Mikroprozessor zur Programmausführung benötigt. Man schreibt deshalb ein Programm zunächst in symbolischer Form; anschließend oder zu einem späteren Zeitpunkt wird es in den Maschinencode übersetzt und in den Hauptspeicher geladen. Das Übersetzen kann, wie oben gezeigt wurde, von Hand geschehen. Man benötigt dazu die Zuordnungstabelle und die Adresse des ersten Maschinencodewortes im Speicher, um die Symboltabelle mit den Adreßzuordnungen aufstellen zu können. Das manuelle Übersetzen ist jedoch sehr zeitaufwendig und vor allem fehleranfällig. Außerdem können kleine Änderungen im symbolischen Programm große Ände-

rungen im Maschinencode nach sich ziehen. So ändern sich z.B. beim Entfernen oder Einschieben eines Befehls sämtliche Adreßbezüge auf die nachfolgenden Speicherzellen.

Da der Übersetzungsvorgang nach festen Regeln abläuft, kann seine Ausführung auch dem Mikroprozessor selbst übertragen werden. Dazu muß die symbolische Schreibweise eindeutig festgelegt sein, und wir benötigen ein Programm, welches das zu übersetzende symbolische Programm (als Eingabedaten) in den Maschinencode (als Ausgabedaten) umformt. Man nennt ein solches Übersetzungsprogramm *Assembler* (to assemble: montieren). Die Regeln zur symbolischen Programmierung ergeben sich aus der Definition einer Assemblersprache. Man nennt Programme, die in Assemblersprache geschrieben sind, *Assemblerprogramme*.

Assemblersprache. Die Assemblersprache legt die äußere Form einer Programmzeile fest. Eine solche Zeile ist in Zeichenfelder mit bestimmten Funktionen unterteilt. Die Feldgrenzen sind dabei üblicherweise innerhalb einer Zeile fließend und werden durch wenigstens ein Leerzeichen (SP, space) oder Tabulatorzeichen (HT, horizontal tabulation) voneinander getrennt. Letzteres deutet darauf hin, daß es sinnvoll ist, Programmzeilen in tabellierter Form untereinanderzuschreiben (wie in unseren Beispielprogrammen gezeigt), da sich dadurch eine bessere Lesbarkeit der Programme ergibt. Bild 1-19 zeigt eine solche tabellierte Zeilenunterteilung mit Benennung der einzelnen Felder. Die maximale Länge (Zeichenanzahl) einer solchen Zeile ist assemblerabhängig; im Bild ist sie zu 256 Zeichen angenommen.

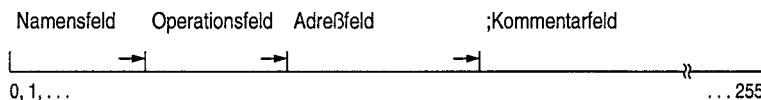


Bild 1-19. Format einer Programmzeile

Das Namensfeld dient zur symbolischen Adressierung einer Programmzeile und kann dazu ein Symbol enthalten. Das Operationsfeld nimmt den symbolischen Operationscode auf, und im Adreßfeld stehen die symbolischen Adreßangaben, z.B. durch Kommas getrennt. Das Kommentarfeld dient zur Kommentierung der entsprechenden Programmzeile. Dem Kommentar wird üblicherweise ein Sonderzeichen vorangestellt, häufig ein Semikolon. Mit einem Semikolon in der ersten Spalte können auch ganze Programmzeilen mit Kommentar gefüllt werden. Kommentare bleiben genau so wie Leerzeilen während der Programmübersetzung unberücksichtigt und haben somit keinen Einfluß auf die Erzeugung des Maschinencodes.

Durch die Assemblersprache ist auch der Zeichenvorrat vorgegeben, mit dem eine Programmzeile geschrieben werden kann (z.B. Großbuchstaben, Kleinbuchstaben, Ziffern, bestimmte Sonderzeichen). Sie schreibt außerdem vor, welche Zeichenketten zur Bildung von Adreßsymbolen erlaubt sind. So muß z.B. das erste

Zeichen eines Adresssymbols ein Buchstabe sein, und das Symbol darf nur eingeschränkt Sonderzeichen enthalten. Außerdem ist die als signifikant geltende Zeichenanzahl eines Symbols üblicherweise begrenzt, z.B. auf 32 Zeichen. Ferner legt die Assemblersprache die symbolischen Operationscodes und spezielle Adresssymbole, wie R0 bis R7 für den Registerspeicher, fest.

Im Unterschied zu höheren Programmiersprachen entspricht bei Assemblersprachen ein symbolischer Befehl genau einem Befehl im Maschinencode (1-zu-1-Übersetzung). Ein Assemblerprogramm ist somit gegenüber einem C-, PASCAL- oder Java-Programm an die Prozessorhardware angepaßt.

Assembleranweisungen. In unserem Programmbeispiel kommen neben den Befehlen auch Operanden vor, deren Werte vor der Programmausführung entweder bekannt sind oder für die lediglich eine Speicherzelle reserviert wird. Für diese und ähnliche Vorgaben sieht die Assemblersprache **Assembleranweisungen** (Assemblerdirektiven) vor, die als Programmzeilen in das symbolische Programm eingefügt werden. Assembleranweisungen dienen außerdem zur Steuerung des Übersetzungsvergangs. Im Gegensatz zu den Maschinenbefehlen erzeugen Assembleranweisungen bei der Übersetzung (Assemblierung) des Programms meist keinen Binärkode.

Die wichtigsten Assembleranweisungen sind im folgenden zusammengestellt. Wie die Maschinenbefehle unterliegen sie dem Format einer Programmzeile und bestehen dementsprechend aus einem symbolischen Code und einem Adressteil. Die Verwendung von Namensangaben ist entweder wahlweise, was in der folgenden Auflistung durch eine eckige Klammer gekennzeichnet ist, oder bindend, wie z.B. bei der EQU-Anweisung.

[Symbol] ORG c	Origin of program or data: Gibt mit der Zahl c die Anfangsadresse des nachfolgenden Programmteils oder Datenbereichs an; ein Name im Namensfeld gilt als symbolische Anfangsadresse.
END Symbol	End of program: Zeigt dem Assembler das Ende des zu assemblierenden Programms, d.h. die letzte Programmzeile, an. Das Symbol kennzeichnet den ersten auszuführenden Befehl; es muß dementsprechend auch in dessen Namensfeld stehen.
[Symbol] DS c	Define storage: Reserviert im Speicher so viele Speicherzellen, wie die Zahl c angibt; ein Name im Namensfeld gilt als symbolische Adresse der ersten reservierten Speicherzelle.
[Symbol] DC c	Define constant: Reserviert eine Speicherzelle und belegt sie mit dem Wert der Zahl c; ein Name im Namensfeld gilt als symbolische Adresse der Speicherzelle.
Symbol EQU c	Equate: Weist einem Namen im Namensfeld den Wert der Zahl c zu.

Mit diesen Assembleranweisungen lässt sich unser Programmbeispiel vollständig in Assemblersprache formulieren (Bild 1-20). Mit der ORG-Anweisung geben wir eine Speicherbelegung ab Zelle 0 vor; die END-Anweisung gibt dem Assembler die letzte zu verarbeitende Programmzeile an. Die Adresszuweisung an das Datenregister EAREG, die in der bisherigen Schreibweise nicht möglich war, kann jetzt mit der EQU-Anweisung vorgenommen werden. Die beiden DC-Anweisungen erzeugen die Dualzahlen für die Operanden NULL und EINS; mit der DS-Anweisung wird eine Speicherzelle für den zu Beginn der Ausführung des Programms noch unbekannten Wert von ZEITK reserviert. Die Übersetzung des Programms durch den Assembler führt auf den in Tabelle 1-10 dargestellten Maschinencode.

Name	Opcode	Adressangaben	Kommentar
<i>; Impulsgeberprogramm</i>			
EAREG	ORG 0		<i>; speichern ab Adr. 0</i>
EAREG	EQU 32768		<i>; E/A-Adr. festlegen</i>
START:	MOVE EAREG, ZEITK		<i>; Zeitkonst. einlesen</i>
	MOVE NULL, EAREG		
	MOVE NULL, R0		
M1:	MOVE ZEITK, R2		<i>; Zeitschleife init.</i>
M2:	SUB EINS, R2		
	CMP R0, R2		<i>; Zeitbed. abfragen</i>
	BNE M2		
	MOVE EINS, EAREG		<i>; positive Flanke</i>
	MOVE NULL, EAREG		<i>; negative Flanke</i>
	JMP M1		
ZEITK	DS 1		<i>; Datenbereichsanfang</i>
NULL	DC 0		
EINS	DC 1		
END	START		

Bild 1-20. Impulsgeberprogramm in Assemblersprache

Assemblierung. Die Übersetzung (*Assemblierung*) eines Assemblerprogramms durch den Assembler erfolgt üblicherweise in zwei Phasen, d.h. in zwei Durchgängen durch das zu assemblierende Programm. Im ersten Durchgang werden die Adresszuordnungen hergestellt und eine Fehlerliste angefertigt. Im zweiten Durchgang wird der Maschinencode erzeugt und eine Auflistung des Programms in symbolischer und binärer oder hexadezimaler Darstellung vorgenommen. In beiden Durchgängen wird das Assemblerprogramm zeilenweise gelesen und verarbeitet. Man nennt Assembler, die nach diesem Prinzip arbeiten, auch Zwei-Phasen-Assembler (zur Funktionsweise von Assemblern siehe z.B. [Barron 1970]).

Die Adresszuordnungen werden, wie bereits beschrieben, durch den Aufbau einer *Symboltabelle* ermittelt. Die Adresszählung beginnt mit der in der ORG-Anweisung angegebenen Anfangsadresse und wird durch einen sog. *Zuordnungszähler*

vom Assembler vorgenommen. Der Zuordnungszähler wird dazu mit der Anfangsadresse initialisiert und nach der Bearbeitung einer jeden Programmzeile um die Anzahl der von dieser Zeile im Maschinencode belegten Speicherzellen weitergezählt. Ein Symbol im Namensfeld der Programmzeile wird in die Symboltabelle mit dem augenblicklichen Wert des Zuordnungszählers und dem Attribut „definiert“ eingetragen. Ein Speicheradresßsymbol im Adreßfeld wird ebenfalls in die Symboltabelle übernommen, jedoch mit dem Attribut „verwendet“ versehen. Eine Ausnahme bilden Symbole im Namensfeld von EQU-Anweisungen, deren Adreßwerte sich aus den Zahlenangaben im Adreßfeld selbst ergeben.

Enthält ein Programm mehrere ORG-Anweisungen, so wird der Zuordnungszähler bei jeder ORG-Anweisung mit dem im Adreßfeld angegebenen Wert neu geladen. Damit können beispielsweise der Programm- und der Datenbereich im Speicher getrennt voneinander angelegt werden. Bei Erreichen der END-Anweisung müssen sämtliche Symbole definiert sein. Offene Adreßbezüge und Schreibweisen innerhalb der Programmzeilen, die die Assemblersprache nicht erlaubt, werden in der Fehlerliste vermerkt.

Tabelle 1-11 zeigt einen Schnappschuß beim Aufbau der Symboltabelle für unser Programmbeispiel nach Bearbeitung der BNE-Programmzeile im ersten Durchgang. Zu diesem Zeitpunkt sind die Symbole ZEITK, NULL und EINS zwar verwendet worden, es konnte ihnen aber noch kein Adreßwert zugewiesen werden, da die Programmzeilen, in deren Namensfeldern sie definiert sind, noch nicht bearbeitet wurden. Solche *Vorwärtsadresßbezüge* sind der Grund dafür, daß nicht bereits beim ersten Durchgang für jede Programmzeile unmittelbar der Maschinencode erzeugt werden kann.

Im zweiten Durchgang werden die Programmzeilen nacheinander in den *Maschinencode* übersetzt, wobei die Zuordnungs- und die Symboltabelle ausgewertet werden. Aus den fest vereinbarten Registersymbolen R0 bis R7 werden die numerischen Adressen des Registerspeichers ermittelt. Sie ermöglichen die Unterscheidung von Register- und Speicheradressen und dürfen nicht zur symbolischen Kennzeichnung von Speicheradressen herangezogen werden. Daraus ergibt sich

Tabelle 1-11. Zustand der Symboltabelle nach Bearbeitung der BNE-Zeile

Symbol	Adreßwert	verwendet	definiert
EAREG	32 768	x	x
START	0	-	x
ZEITK	-	x	-
NULL	-	x	-
M1	8	-	x
M2	10	x	x
EINS	-	x	-

für unser Beispiel die Codierung der R/S-Bits und damit die Anzahl der Maschinencodewörter pro Befehl. Mit der Codeerzeugung wird gleichzeitig eine Programmliste erstellt, die neben dem symbolischen Programm (*Quellprogramm*) den Maschinencode (üblicherweise in Hexadezimaldarstellung), die Speicheradressen des Maschinencodes, eine Zeilennumerierung und Fehlerhinweise enthält (Bild 1-21).

Nr.	Adresse	Inhalt	Name	Opcode	Adressangaben	Kommentar
1					; Impulsgeberprogramm	
2						
3				ORG	0	; speichern ab Adr. 0
4		EAREG		EQU	32768	; E/A-Adr. festlegen
5						
6	0000	0100	START:	MOVE	EAREG, ZEITK	; Zeitkonst. einlesen
		8000				
		0017				
7	0003	0100		MOVE	NULL, EAREG	
		0018				
		8000				
8	0006	0108		MOVE	NULL, R0	
		0018				
9	0008	010A	M1:	MOVE	ZEITK, R2	; Zeitschleife init.
		0017				
10	000A	020A	M2:	SUB	EINS, R2	
		0019				
11	000C	218A		CMP	R0, R2	; Zeitbed. abfragen
12	000D	0C00		BNE	M2	
		000A				
13	000F	0100		MOVE	EINS, EAREG	; positive Flanke
		0019				
		8000				
14	00012	0100		MOVE	NULL, EAREG	; negative Flanke
		0018				
		8000				
15	0015	0500		JMP	M1	
		0008				
16						
17	0017		ZEITK	DS	1	; Datenbereichsanfang
18	0018	0000	NULL	DC	0	
19	0019	0001	EINS	DC	1	
20			END		START	

Bild 1-21. Programmliste des Impulsgeberprogramms

Je nach Aufbau des Assemblers wird der erzeugte Maschinencode direkt in den durch die ORG-Anweisungen vorgegebenen Bereichen im Speicher erzeugt, oder er wird vom Assembler zunächst auf ein externes Speichermedium ausgegeben, von wo er dann ggf. zu einem späteren Zeitpunkt von einem Ladeprogramm (*Lader*) in den Hauptspeicher geladen wird. Der Lader lädt das Programm entweder an die durch die ORG-Anweisungen des Programms vorgegebenen Speicheradressen (*Absolutlader*), oder er nimmt eine Verschiebung des Programms um

eine vorgebbare Ladedistanz vor (*verschiebender Lader*, relocating loader). Hierzu müssen während des Ladevorgangs die Adressangaben in den Adreßteilen von Befehlen, die über den Zuordnungszähler ermittelt wurden, um die Ladedistanz erhöht werden. Diese Adressen werden als *relative* oder *verschiebbare Adressen* bezeichnet. Adressen, die über die EQU-Anweisung als Absolutwerte vorgegeben sind, bleiben unverändert; man bezeichnet sie auch als *absolute* oder *feste Adressen*. Um dem verschiebenden Lader diese Unterscheidung zu ermöglichen, muß der Assembler Zusatzinformation zum Maschinencode liefern.

Ein Programm kann – wie in Kapitel 3 beschrieben ist – auch aus mehreren Teilen bestehen, die unabhängig voneinander assembled werden. Um die zwischen den Programmteilen auftretenden Adressquerbezüge auflösen zu können, müssen die Programmteile vor oder während des Ladens zusammengefügt (gebunden) werden. Diese Aufgabe übernimmt ein Bindeprogramm (*Binder*, linkage editor) bzw. ein *bindender Lader* (linking loader). Die zur Herstellung der Adressquerbezüge notwendige Information liefert ebenfalls der Assembler.

Häufig wird die Entwicklung von Programmen nicht auf dem zu programmierenden Mikroprozessorsystem selbst, sondern auf sog. Entwicklungssystemen oder universellen Rechenanlagen durchgeführt. Entwicklungssysteme sind Mikroprozessorsysteme, die mit Übersetzungs- und Testprogrammen für einen bestimmten Mikroprozessortyp ausgestattet sind. Im allgemeinen arbeiten sie mit demselben Prozessortyp, so daß der erzeugte Maschinencode sowohl auf dem Entwicklungssystem als auch auf dem zu programmierenden System ausführbar ist. Arbeiten sie mit einem anderen Prozessortyp, so werden Übersetzungsprogramme benötigt, die Maschinencode für den zu programmierenden Mikroprozessor erzeugen. Das gilt auch für die Programmentwicklung auf universellen Rechenanlagen. Man bezeichnet Übersetzer, die nicht für den Prozessor, auf dem sie laufen, sondern für andere Prozessoren Maschinencode erzeugen, als *Cross-Assembler* und *Cross-Compiler* oder allgemein als Cross-Software.

1.3.3 Programmeingabe und Textausgabe

Das symbolische Programm muß bei der Eingabe als Folge von ASCII-Zeichen (ASCII string, byte string) erscheinen, um vom Mikroprozessor verarbeitet werden zu können. Das Überführen in den ASCII geschieht durch das Eingabegerät, z.B. die Tastatur eines Bildschirmterminals. Beim Betätigen einer Taste wird das entsprechende ASCII-Zeichen erzeugt und an das Mikroprozessorsystem übertragen.

Bild 1-22 zeigt diesen Vorgang an unserem Programmbeispiel. Leerzeichen sind mit SP (space) bezeichnet. Das Ende einer Zeile ist mit dem Steuerzeichen für den Wagenrücklauf CR (carriage return), der Übergang auf die nächste Zeile mit dem Steuerzeichen für den Zeilenvorschub LF (line feed) angegeben. Jeweils

zwei ASCII-Zeichen (zwei Bytes) können in einer 16-Bit-Speicherzelle untergebracht werden. Ein symbolisches Programm, das vom Assembler als ASCII-Zeichenfolge in den Speicher geladen wird, belegt somit wesentlich mehr Speicherzellen als das daraus erzeugte Maschinenprogramm.

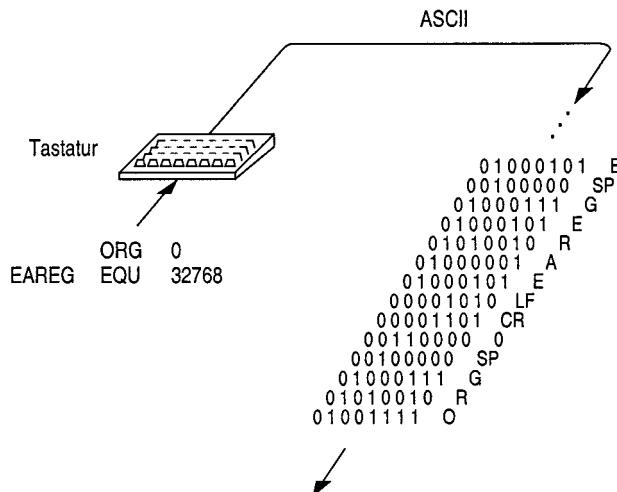


Bild 1-22. Zeicheneingabe im ASCII

Bei der Ausgabe von Text, z.B. bei der Ausgabe der Programmliste an einen Drucker oder ein Datensichtgerät, überträgt der Prozessor eine Folge von ASCII-Zeichen an das Peripheriegerät. Diese Zeichenfolge enthält neben dem eigentlichen Text ebenfalls Steuerzeichen, wie Wagenrücklauf, Zeilenvorschub und Seitenvorschub.

1.4 Der RISC-Mikroprozessor

Reduziert man die Befehlsliste eines Prozessors mit CISC-Architektur auf bestimmte elementare Befehle, die in etwa den Mikrobefehlen des Mikroprogramms eines solchen Rechners entsprechen, so entsteht eine ganz andere Rechnerarchitektur, nämlich die eines *Reduced Instruction Set Computers (RISC)*. Diese neue Architektur spiegelt sich nicht nur in einer andersartigen Prozessorstruktur wider, die gewissermaßen ohne Mikroprogrammierung auskommt, sondern hat auch erhebliche Rückwirkung auf die Maschinen-/Assemblerprogrammierung, da diese nun so diffizil und unübersichtlich wird, daß sie gewissermaßen der Mikroprogrammierung gleichzusetzen ist. Deshalb ist es auch nicht verwunderlich, daß dieser Abschnitt schwieriger ist und ggf. einer tieferen Einarbeitung bedarf. Das gilt auch für die RISC-Abschnitte 2.2 und 3.4 der beiden folgenden Kapitel.

1.4.1 Prozessorstruktur

Wie bei CISCs sind auch bei RISCs die verschiedenen Prozessoreigenschaften, wie das Format eines Befehls, die Art seiner Speicherung, der Zugriff auf seine Operanden, die Ausführung der Operation, aufeinander abgestimmt. Ihr Zusammenwirken führt auf eine Prozessorstruktur mit einer Funktionsweise wie sie für RISCs typisch ist, nämlich der Fließbandverarbeitung oder des Pipelining (oft gekoppelt mit einer Vervielfachung der Verarbeitungseinheiten, d.h. mit einem gewissen Maß an Parallelarbeit im Prozessor). Wir behandeln diese Gesichtspunkte im folgenden unter den Stichwörtern Befehlsformate, Prozessorstruktur und Fließbandverarbeitung. Dabei gehen wir von einer 32-Bit-Prozessorstruktur aus und verwenden dementsprechend die Bezeichnungen Halbwort für das 16-Bit-Datenformat und Wort für das 32-Bit-Datenformat.

Befehlsformate. RISCs sehen für ihre wichtigsten Befehle, das sind die für jeden Rechner unerlässlichen arithmetischen und logischen Befehle, das *Dreiadreßbefehlsformat* vor. Um einen Dreiadreßbefehl mit seinen zwei Quelladressen und seiner Zieladresse (vgl. Bild 1-8, S. 24) in einem einzigen 32-Bit-Speicherwort unterzubringen, muß man sich notwendigerweise auf kurze Adressen beschränken, d.h. auf die Adressierung ausschließlich des Registerspeichers (Bild 1-23a).

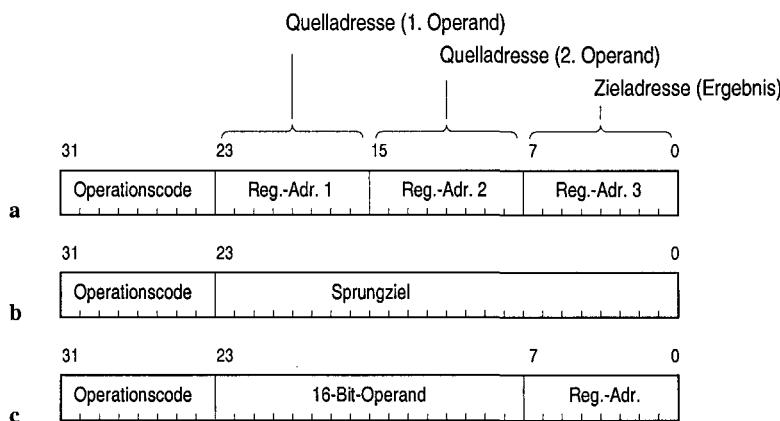


Bild 1-23. Typische RISC-Befehlsformate. **a** Register-Register-Befehle, **b** Sprungbefehle, **c** Setze-Register-Befehl

Allerdings ist der Registerspeicher bei RISCs oft erheblich größer als bei CISCs. Das kann man sich leisten, weil wegen der Einfachheit der Maschinenbefehle – sie reichen bei den ursprünglichen RISCs nur bis zur Addition/Subtraktion der ALU – das Mikroprogrammwerk eines solchen Prozessors „gegen Null geht“, so daß auf dem Prozessorchip mehr Platz für den Registerspeicher zur Verfügung steht.

Interessant in diesem Zusammenhang ist, daß Operationen mit weniger als drei Adressen, wie Lösche Register oder Transportiere von Register zu Register, sozusagen künstlich auf Dreiaußendressformat gebracht werden. Zum Beispiel wird der Transport des Inhalts des Registers r3 nach r5 ausgeführt durch die Oder-Operation (or) mit $r0 = 0$ als erstem Quellregister, r3 als zweitem Quellregister und r5 als Zielregister. (Die Registerbezeichnungen und die Befehle sind zur Abgrenzung gegenüber CISCs hier bei RISCs mit kleinen Buchstaben geschrieben. Dies ist nicht zwingend und wird von uns nur zur besseren Unterscheidung beider Prozessorarten eingesetzt.)

Neben dem Dreiaußendressbefehlsformat, abgestimmt auf den Registerspeicher, benötigen RISCs einige wenige weitere Befehlsformate, und zwar für Sprungbefehle (Bild 1-23b) sowie für den Setze-Register-Befehl (Bild 1-23c). Mit letzterem kann ein im Befehlswort angegebener Operand (*Direktoperand*) ein Register direkt (*Direktoperand-Adressierung*) auf einen bestimmten Wert „setzen“. Der Operand nimmt dabei im Befehlswort den sonst von den beiden Quelladressen belegten Platz ein. Der Setze-Register-Befehl ist u.a. deshalb nötig, um auf den prozessorexternen Hauptspeicher zugreifen zu können. Im Gegensatz zu CISCs geht das nämlich bei RISCs im Grunde nur dadurch, daß mit dem Setze-Register-Befehl (set) die Hauptspeicheradresse (als Direktoperand) in ein erstes Register gebracht wird und anschließend durch einen *Lade-Befehl* (ld) der unter dieser im Register stehenden Adresse aus dem Hauptspeicher gelesene Operand in ein zweites Register transportiert wird bzw. durch einen *Speichere-Befehl* (st) ein in einem zweiten Register stehendes Ergebnis unter dieser im ersten Register stehenden Adresse in den Hauptspeicher geschrieben wird (registerindirekte Adressierung, siehe 2.1.3 und 2.2.3). – Lade- und Speichere-Befehle benötigen keine eigenen Befehlsformate, da sie in der geschilderten Form nur Registeradressen enthalten.

Man bezeichnet Rechnerarchitekturen, bei denen neben Laden und Speichern keine weiteren Befehle mit Speicherzugriff existieren, vielfach als *Load-/Store-Architekturen*. Sollen mit einem solchen Rechner dennoch Speicher-Speicher-Operationen, wie sie z.B. beim Durchsuchen größerer Datenbestände vorkommen, effizient durchgeführt werden, so empfiehlt sich der Einbau parallel arbeitender Funktionseinheiten in Verbindung mit parallel abrufbaren Befehlen, so daß das Laden/Speichern und das Verarbeiten parallel überlappend vonstattengehen können. Das ist verglichen mit den hier ausführlich beschriebenen „skalaren“ Prozessoren der Vorteil sog. „superskalarer“ Prozessoren (siehe 2.3.2).

Bei (skalaren) RISCs sind also arithmetische und logische Registeroperationen eindeutig bevorzugt, denn Operationen mit im Speicher stehenden Operanden müssen grundsätzlich über die Register abgewickelt werden. Die Programmierung eines RISC ist somit nur dann wirklich effizient, wenn für einen größeren Programmabschnitt als erstes alle Operanden mit Lade-Befehlen in den Registerspeicher kopiert werden, dann die in diesem Programmabschnitt auszuführenden

Operationen ausschließlich im Registerspeicher abgewickelt werden und danach die Ergebnisse der Operationen mit Speichere-Befehlen zurück in den Hauptspeicher transportiert werden. Es ist einsichtig, daß dies nur effizient funktioniert, wenn die zu programmierende Aufgabenstellung sich gut in solche Abschnitte gliedern läßt; man spricht von *Programm- und Datenlokalität* und nennt einen solchen Programm-/Datenabschnitt *aktuellen Ausschnitt* (des Programms und seiner Daten).

Prozessorstruktur. Die Struktur von RISCs ist darauf ausgelegt, den jeweils aktuellen Ausschnitt sowohl des Programms als auch seiner Daten im Prozessor zur Verfügung zu haben und diesen ohne Hauptspeicherzugriffe so effizient wie möglich abzuarbeiten. Dazu bedient man sich im wesentlichen der folgenden aufeinander abgestimmten Konzepte des Rechnerbaus, woraus sich die in Bild 1-24

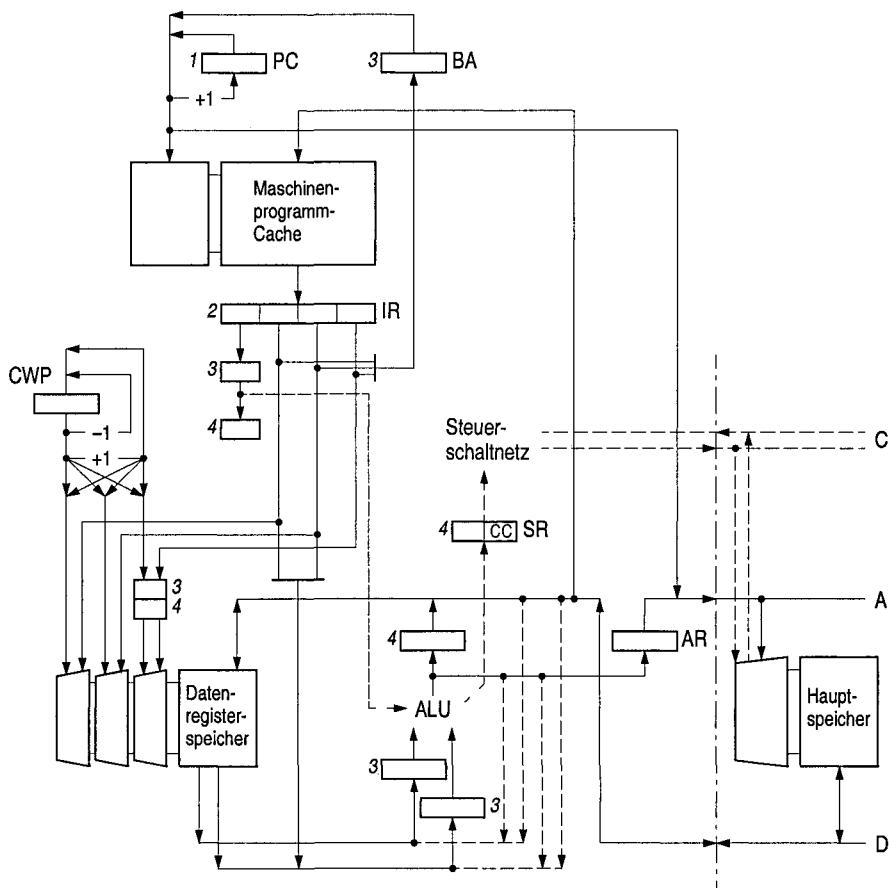


Bild 1-24. Struktur eines 32-Bit-Mikroprozessors in RISC-Architektur. PC Befehlszähler, BA Verzweigungsadreßregister, IR Befehlsregister

wiedergegebene Struktur ergibt. Diese unterscheidet sich „optisch“ nur geringfügig von Bild 1-11, S. 27 (in erster Linie anstelle des ROM ein RAM bzw. CAM); ihr ist aber eine völlig andere Funktionsweise zu eigen. Worin bestehen nun diese Unterschiede im einzelnen?

1. Programmspeicher und Datenspeicher mit Zugriff im Prozessortakt. Im Mikroprogrammspeicher in Bild 1-11 ist jetzt anstelle des Mikroprogramms das Maschinenprogramm abgelegt, so daß aus dem Mikroprogrammspeicher der *Programmspeicher*, aus dem μ PC der PC und dem μ IR das IR wird. Damit können die in Bild 1-11 mit PC bzw. IR bezeichneten Register hier in Bild 1-24 entfallen. – Im Registerspeicher in Bild 1-11 wird jetzt, wie beschrieben, für einen größeren Programmausschnitt der zugehörige Datenausschnitt untergebracht, so daß der Registerspeicher bei der Programmausführung die Funktion des *Datenspeichers* übernimmt. Auf beide Speicher, Programm- wie Datenspeicher, kann somit zeitgleich zugegriffen werden: auf den Programmspeicher mit dem Befehlszähler, auf den Datenspeicher mit dem Befehlsregister.

2. Extra-Programmspeicher für aktuellen Programmausschnitt. Der (prozessorinterne) Programmspeicher könnte nur dann wie der Mikroprogrammspeicher als ROM aufgebaut werden, wenn das Programm des RISC über seine gesamte „Lebensdauer“ unverändert bliebe. Es entstünde (als erstes) ein RISC für eine einzige, ganz bestimmte Aufgabe: ein Einzweck- oder *Spezialrechner* (*special purpose computer*), z.B. für die Bildverarbeitung. Für eine universelle Verwendung, wie sie hier von Interesse ist, muß der Programmspeicher hingegen als RAM aufgebaut sein. Auf diese Weise entsteht (als zweites) ein RISC für viele, laufend wechselnde Aufgaben, d.h. ein Allzweck- oder *Universalrechner* (*general purpose computer*). Aus technischen Gründen, weil nämlich RAMs mit sehr kurzer Zugriffszeit und gleichzeitig hinreichend großer Kapazität innerhalb des Prozessorchips nicht zur Verfügung stehen, wird der Programmspeicher jedoch in der Praxis als Cache aufgebaut, der dann naturgemäß wechselnde aktuelle Ausschnitte des Programms enthält. Es entsteht (als drittes) schließlich der typische, universell programmierbare RISC mit einem für den Programmierer gewissermaßen unsichtbaren Programmausschnitt-Speicher, der völlig automatisch mit Befehlen aus dem Hauptspeicher gefüllt wird.

Anmerkung. Mit einem Spezialrechner der ersten Art, d.h. einem RISC mit einem ROM als Programmspeicher, läßt sich auch ein Universalrechner aufbauen, und zwar als CISC, wenn im ROM des RISC das Mikroprogramm des CISC abgelegt wird. Geschieht dies ein für alle mal, dann bleibt der CISC während seiner Lebensdauer derselbe; wird das Mikroprogramm ausgetauscht, so entsteht ein neuer CISC. Benutzt man dabei für die CISC-spezifischen Register und Mikrooperationen ausschließlich den RISC-Registerspeicher und die RISC-ALU, so spricht man von *Simulation*, hier des CISC auf dem RISC. Werden hingegen zusätzlich zur RISC-Hardware CISC-spezifische Register und Extra-Schaltnetze für CISC-Mikrooperationen aufgebaut, so nennt man das *Emulation*. Emulation wird bereits dem Rechnerbau zugeordnet, hier spricht man dann also vom Bau des CISC (mit dem RISC).

3. Extra-Datenspeicher für aktuellen Datenausschnitt. Der (prozessorinterne) Datenspeicher in RISCs heutiger Bauart ist nicht als Zweiport-, sondern als *Dreiport-Registerspeicher* aufgebaut, wenngleich i.allg. deutlich größer als ein typischer Registerspeicher in einem CISC. Somit können darin – wie bereits geschildert – nur jeweils aktuelle Ausschnitte der Daten gespeichert werden. Im Gegensatz zum Cache als Programmspeicher wird der Registerspeicher nicht automatisch, sondern durch Lade-Befehle, d.h. programmiert, mit Operanden aus dem Hauptspeicher gefüllt. Korrespondierend dazu werden Ergebnisse durch Speichere-Befehle, d.h. ebenfalls programmiert, in den Hauptspeicher zurückgeschrieben. (Zur Beschleunigung des Zugriffs benutzt man allerdings auch hier einen Cache, entweder einen *Daten-Cache* zusätzlich zum *Befehls-Cache* oder einen für Daten und Befehle gemeinsamen *Befehls-/Daten-Cache*.)

4. Blockweise Adressierung bei großem Registerspeicher. Bei einem kleinen Registerspeicher umfassen die drei Registeradressen im Befehl relativ wenige Bits, so daß sie im Befehlswort problemlos untergebracht werden können. Bei einem größeren bzw. einem großen Registerspeicher benötigen diese drei Adressen jedoch entsprechend mehr Bits, so daß sie unter Umständen nicht mehr vollständig im Befehlswort Platz finden. Man geht nun bei RISCs nicht den bei CISCs üblichen Weg, einen Befehl über zwei oder mehr Worte auszudehnen, sondern bleibt beim Wortformat des Befehls von z.B. 32 Bits. Dazu gliedert man die drei Registeradressen in jeweils zwei Teile und bringt nur den „unteren“ Teil einer jeden Adresse im Befehlswort unter. Der „obere“ Teil wird in einem Spezialregister vorgegeben; dadurch wird der Registerspeicher in Blöcke, oft *Fenster (windows)* genannt, unterteilt. Dieses Spezialregister, in Bild 1-24 mit CWP (current-window-pointer) bezeichnet, enthält nun die Adresse des aktuellen Fensters, und im Befehlswort wird mit der nun kurzen Adresse ein Register innerhalb dieses Fensters adressiert. Beide Teile werden dementsprechend zur Adressierung des Registerspeichers zusammengefügt, konkateniert, wie man sagt. Der CWP kann inkrementiert und dekrementiert werden, so daß, ausgehend von einem bestimmten Fenster, das jeweils davor bzw. danach liegende Fenster angewählt werden kann.

5. Arithmetisch-logische Operationen im Prozessortakt. Charakteristisch für RISCs – was sich ja in der Bezeichnung widerspiegelt – sind arithmetische und logische Operationen von solcher Einfachheit, daß sie sich in einem einzigen TaktSchritt ausführen lassen. Das heißt für (skalare) RISCs, daß elementare Operationen lediglich bis zur Addition/Subtraktion als Maschinenbefehle vorgesehen sind. Bereits die Multiplikation muß dann (maschinen)programmiert werden. – Oder man geht den wirkungsvolleren Weg, daß man dem (skalaren) RISC-Prozessor einen Coprozessor oder sogar weitere, superskalare Verarbeitungseinheiten für Multiplikation und Division von Ganzzahlen und oft auch noch für die üblichen Rechenoperationen mit Gleitpunktzahlen zur Seite stellt, die dann oft nicht RISC-typisch, sondern eher CISC-typisch arbeiten, d.h. mikroprogrammiert sind.

6. Gleichzeitig überlappende Ausführung von Teilen mehrerer Befehle. Ebenso wie in Bild 1-11 (S. 27) gleichzeitig, d.h. in ein und demselben Schritt des Prozessortakts, mit der Ausführung eines Mikrobefehls der auf ihn folgende Mikrobefehl aus dem Mikroprogrammspeicher geholt wird (der im μ IR stehende Mikrobefehl liest z.B. Registeroperanden, während der μ PC bereits den nächsten Mikrobefehl holt), so erfolgt in Bild 1-24 das Ausführen eines Maschinenbefehls mit dem Holen des nächsten Maschinenbefehls in ein und demselben TaktSchritt. Man nennt eine solche gleichzeitig überlappende, synchron-parallele Arbeitsweise *Fließbandverarbeitung* oder *Pipelining* (pipeline: Röhrenleitung), dort für das Mikroprogramm, hier für das Maschinenprogramm. Die typische Struktur eines RISC erlaubt jedoch noch Verfeinerungen dieser Art von Parallelarbeit: in der in Bild 1-24 wiedergegebenen RISC-Struktur ist z.B. eine vierfache Überlappung von Maschinenbefehlen möglich (kenntlich an den kursiven Nummern an den Registern).

Die Fließbandverarbeitung – nachfolgend ausführlicher beschrieben – führt im Idealfall dazu, daß mit jedem Prozessortakt ein Befehl das Fließband verläßt, d.h. fertiggestellt wird (während die auf ihn folgenden Befehle noch im Fließband stecken). In das Fließband gelangt also mit jedem Prozessortakt ein Befehl hinein, und es kommt im Idealfall ein Befehl heraus. Ein und derselbe Befehl benötigt bei vierfacher Überlappung dazu natürlich vier Takte. Leider wird der Idealfall nicht immer erreicht, z.B. bei vielen RISCs, wenn Speicherzugriffe mit Lade- oder Speichere-Befehlen in mehr als einem Takt auszuführen sind; dadurch entstehen Verzögerungen im Fließband, und der Durchsatz an Befehlen geht zurück. Andere Gründe für mögliche Verzögerungen im Fließband ergeben sich aus Abhängigkeiten zwischen aufeinanderfolgenden Befehlen, bei denen das Ergebnis des ersten unmittelbar als Operand des zweiten oder dritten Befehls benutzt wird, sowie durch Sprungbefehle, bei denen es sich – bezogen auf die Kontinuität im Fließband – zu spät entscheidet, mit welchem Befehl fortgefahrene werden soll, d.h., mit welchem Befehl das Fließband als nächstes geladen werden muß. Diese Verzögerungen lassen sich durch Maßnahmen in der Hardware oder in der Software vermeiden, wie nachfolgend beschrieben (siehe vertiefend dazu auch 2.3.3, 2.3.6 und 2.3.7).

Fließbandverarbeitung. Bei der Fließbandverarbeitung nutzt man die Tatsache aus, daß jede Befehlausführung aus mehreren Teilen besteht, die nacheinander in entsprechenden Stufen des Fließbands abgearbeitet werden. Das ist z.B. bei einfachen Register-Register-Befehlen der Fall (und hier unterscheiden sich CISCs und RISCs nicht wesentlich). In Bild 1-25a sind diese Teile der Reihe nach mit

1. Befehl holen (Fetch: F),
2. Befehl decodieren und Operanden holen (Decode: D),
3. Operation ausführen (Execute: E),
4. Ergebnis schreiben (Write: W)

bezeichnet. Das parallel dazu erfolgende Inkrementieren des PC ist darin nicht mit aufgenommen.

Bei einfachen CISCs (vgl. Bild 1-12, S. 29, die Schritte 2, 3 und 4 zusammengefaßt) werden diese Teile für jeden Befehl einzeln nacheinander ausgeführt: in dem in Bild 1-25a wiedergegebenen, sicherlich unnötzen Programmstück zuerst der Befehl CLR (clear, lösche), dann der Befehl MOVE, weiter der Befehl ADD und schließlich der Befehl SUB (die letzten beiden als Dreiaußbefehle).

Bei RISCs hingegen werden diese Teile für mehrere Befehle (bei einem vierstufigen Fließband vier) gleichzeitig überlappend ausgeführt (Bild 1-25b). Für unser Programmstück ergibt sich demnach zu einem bestimmten Zeitpunkt folgende Situation: Während z.B. für clr das Ergebnis (hier 0) geschrieben wird (nach r5), wird gleichzeitig für move die Operation ausgeführt (der Inhalt von r1 durch die ALU transportiert), des weiteren werden gleichzeitig für add die Operanden geholt (aus r1 und r2), und schließlich wird gleichzeitig der Befehl sub geholt (aus dem Befehlsspeicher).

	F	D	E	W	CLR	R5
a					MOVE	R1, R6
					ADD	R1, R2, R7
					SUB	R3, R4, R8
b					clr	r5
	F	D	E	W	move	r1, r6
	F	D	E	W	add	r1, r2, r7
	F	D	E	W	sub	r3, r4, r8

Bild 1-25. Befehlausführung; a ohne, b mit Fließbandverarbeitung. Das stark umrandete Feld illustriert die Situation der einzelnen Befehlausführungen im Fließband innerhalb eines Takt-schritts. Man kann es sich als „Fenster“ vorstellen, das sich mit jedem Takt um eine Position nach rechts bewegt

Es ist evident, daß die Prozessorstruktur ein solches *Befehlspipelining* erlauben muß bzw. daß nur bei Einhaltung bestimmter Bedingungen ein reibungsloser Fluß durch das Fließband gewährleistet werden kann. Zum Beispiel müssen Befehle und Operanden in zwei verschiedenen, jedoch gleich schnellen Speichern stehen, hier im Befehls-Cache und im Registerspeicher. Auch die Verarbeitung der Operanden sowie das Rückschreiben des Ergebnisses müssen jeweils in der Fließband-Taktzeit (gleich der Prozessor-Taktzeit) abgeschlossen sein, d.h., auch die Zugriffszeiten der Speicher und die Verarbeitungszeit der ALU sollten idealerweise gleich sein.

Bild 1-26 zeigt noch einmal die Ausführung unseres Programmstücks, nun aber in einer Art, wie sich die in Bild 1-25b markierte Befehlssituation mit bestimmten Registerinhalten (welchen? siehe unten!) im Fließband darstellt. Bild 1-26 liegt die in Bild 1-24 (S. 51) wiedergegebene Struktur zugrunde. Es zeigt die mit Num-

mern versehenen Register korrespondierend zu den in Bild 1-24 numerierten Registern, in denen die in jedem Fließbandtakt zu verarbeitende Information vorliegt. Unmittelbar darunter ist die den jeweiligen Registern zugeordnete Funktion wiedergegeben.

Man muß sich nun die Arbeitsweise des Systems so vorstellen, als ob auf dem Fließband Werkstücke liegen (genauer auf den den Registern entsprechenden Arbeitsplätzen), die innerhalb eines (Fließband-)Taktschritts von verschiedenen (Fließband)arbeitern bearbeitet werden. Nach Ablauf der für einen Taktschritt

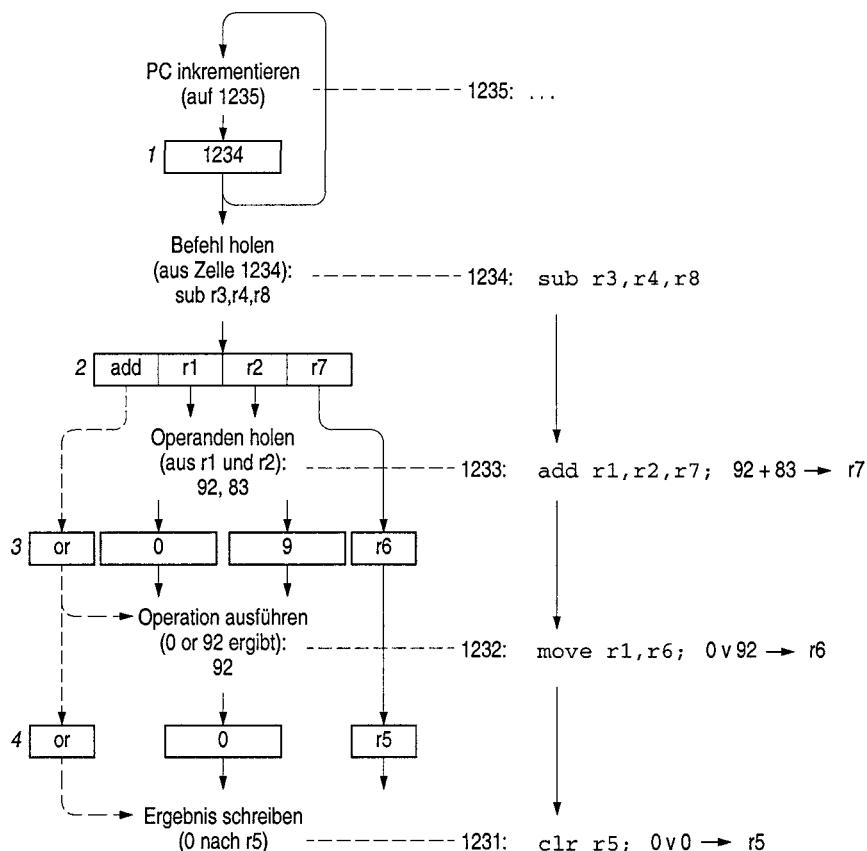


Bild 1-26. Ausführung innerhalb eines Taktschritts des in Bild 1-25 wiedergegebenen Programms. Man beachte das zur Programmaufschreibung gegenläufige Erscheinungsbild der Befehlsfolge im Fließband

notwendigen Zeit sind diese mit ihrer Arbeit fertig; und mit Erscheinen des Taktsignals werden die bearbeiteten Werkstücke gewissermaßen auf einen Schlag zu den jeweils nächsten Arbeitsplätzen weitertransportiert. Bei uns entsprechen den

Werkstücken die Befehls-Teile auf dem Fließband und den Arbeitsplätzen mit den Arbeitern die einzelnen Fließbandstationen (Fließbandstufen, Pipeline-Stufen).

In Bild 1-26 sind die Befehle `clr r5` durch `or r0,r0,r5` und `move r1,r6` durch `or r0,r1,r6` ersetzt, wie das bei RISCs üblich ist. Darin bedeutet „or“ die logische Operation „oder“, wobei vorausgesetzt wird, daß im Register `r0` – auch das ist typisch für RISCs – immer die Konstante Null steht. Somit wird bei `clr r5` Null mit Null oder-verknüpft und das Ergebnis (= 0) in `r5` abgelegt und bei `move r1,r6` Null mit `r1` oder-verknüpft und das Ergebnis (= `r1`) in `r6` abgelegt. – In Bild 1-26 ist weiterhin angenommen, daß in `r1` und `r2` die Zahlen 92 und 83 stehen. Zum besseren Verständnis spiele man das Programmstück mit diesen und weiteren Zahlenangaben (z.B. `r3 = 74, r4 = 65`) durch, und zwar unter der Annahme, daß keine weiteren Befehle in das Fließband hineingelangen, d.h., daß sich das Fließband leert.

1.4.2 Maschinen-/Assemblerprogrammierung

Im Gegensatz zu CISCs nimmt die Maschinen-/Assemblerprogrammierung bei RISCs in ihrer Detailliertheit bereits eine Stellung ein, die der Mikroprogrammierung bei CISCs entspricht. Das bedeutet, daß in der Praxis RISCs im Grunde gar nicht erst im Maschinen-/Assemblercode à la CISC programmiert werden, sondern entweder (maschinenfern) in einer höheren Programmiersprache oder (maschinennah) in einer Art Pseudomaschinen-/Assemblersprache. Das hat zur Folge, daß – wie angedeutet – der „Assembler“ eine wenn auch eher unbedeutende, so aber doch bereits „compilierende“ Rolle bei der Assemblierung/Übersetzung der Pseudobefehle eines Assemblerprogramms übernehmen muß.

Dennoch bleibt aber ein zur Optimierung des Fließbanddurchsatzes notwendiges Umstellen von Befehlen, auch von Pseudobefehlen, dem Maschinenprogrammierer überlassen. Somit wird ihm die schwierigste Aufgabe, nämlich die Einbeziehung der Zeitbedingungen in die Programmierung ähnlich der Mikroprogrammierung, vom Assembler nicht abgenommen. Erst die Programmierung in einer höheren Programmiersprache schafft hier Abhilfe, aber nur unter Verwendung optimierender Compiler. Allerdings wird die Leistungsfähigkeit der Maschine auch bei Benutzung optimierender Compiler i.allg. nicht 100%ig erreicht. Denn das compilierte Maschinenprogramm und damit die Leistungsfähigkeit des RISC – das ist schon ein geflügeltes Wort – ist nur so gut wie sein optimierender Compiler.

Wir behandeln die Einführung in die Maschinen-/Assemblerprogrammierung von RISCs im folgenden anhand der in 1.3.1 vorgestellten Programmieraufgabe zum Aufbau eines Impulsgebers, und zwar unter Zugrundelegung der in Bild 1-16 (S. 36) wiedergegebenen Systemstruktur und des in Bild 1-17 (S. 37) dargestellten Funktionsablaufs. Insbesondere gehen wir auf die Programmierung mit sog.

Pseudobefehlen, die Auflösung von Fließbandkonflikten und in diesem Zusammenhang auf die Optimierung von (skalaren) RISC-Programmen ein.

Programmierung mit Pseudobefehlen. Wie gesagt: die Maschinen-/Assembler-Programmierung von RISCs erfolgt zwar wie üblich symbolisch, aber nicht nur mit den Befehlscodes der Maschinenbefehle, sondern auch (und vor allem) mit den Befehlscodes von *Pseudobefehlen*, die der Assembler in die wirklichen Befehlscodes umsetzt. Korrespondierend zu dem links noch einmal wiedergegebenen Impulsgeberprogramm des CISC aus Bild 1-18 (S. 38) zeigt Bild 1-27 das Impulsgeberprogramm eines RISC. Charakteristisch für das RISC-Programm ist, daß die Operanden nicht wie beim CISC-Programm sowohl im Hauptspeicher als auch im Registerspeicher stehen, sondern ausschließlich im Registerspeicher vorliegen. Darüber hinaus gilt bei RISCs die Besonderheit, daß das Register r0 „fest eingebaut“ die Konstante 0 enthält. (Bei Schreibzugriffen bleibt diese natürlich unverändert gleich 0; man spricht bei einem solchen wirkungslosen Zugriff von einem Dummy-Zugriff.)

CISC-Programm Bild 1-18

```

      |
      MOVE  EAREG, ZEITK
      MOVE  NULL, EAREG
      MOVE  NULL, R0
M1:   MOVE  ZEITK, R2
M2:   SUB   EINS, R2
      CMP   R0, R2
      BNE   M2
      |
      MOVE  EINS, EAREG
      MOVE  NULL, EAREG
      JMP   M1
      |
ZEITK
NULL  0
EINS  1

```

RISC-Programm

```

      set   eareg,r3
      set   1,r4
      ld    r3,r1
      st   r0,r3
      |
m1:   move  r1,r2
m2:   dec   r2
      cmp   r0,r2
      bne   m2
      nop
      st    r4,r3
      st    r0,r3
      jmp   m1
      nop

```

Bild 1-27. Gegenüberstellung eines CISC-Programms mit Maschinenbefehlen und eines RISC-Programms mit Pseudobefehlen ohne sinnvolle Ausnutzung der Delay-Slots. Bezuglich der Adressen der beiden Programme gelten die folgenden Korrespondenzen: R0 = r0 = 0, ZEITK = r1, R2 = r2, EAREG = (r3), EINS = r4

Entsprechend den gerade gemachten Ausführungen entfallen im RISC-Programm die letzten drei Zeilen des CISC-Programms, desgleichen der Befehl MOVE NULL, R0. Andererseits müssen im RISC-Programm Konstanten, die ungleich Null sind, durch Register-Setze-Befehle unmittelbar in den Registern erzeugt werden. Im RISC-Programm wird dies durch die ersten beiden Befehle bewirkt. Dabei handelt es sich um set-Befehle, mit denen die absolute Adresse EAREG = $32768_{10} = 1000000000000000_2$ nach r3 und der konstante Operand $1_{10} =$

0000000000000001_2 nach r4 ge „setzt“ werden. Sie sind in den Befehlen jeweils im 16-Bit-Format direkt enthalten (Format siehe Bild 1-23c, S. 49). – Die restlichen Befehle des RISC-Programms sind Dreiaußbefehle mit Registeradressen (Format siehe Bild 1-23a), die lediglich im *Pseudocode* als Nicht-Dreiaußbefehle erscheinen. Eine Ausnahme bilden dabei die auf den Programmablauf wirkenden Befehle bne und jmp (Formate siehe Bild 1-23b).

Die Wirkung der Befehle (der wirklichen wie der Pseudobefehle) erklärt sich unmittelbar aus der Korrespondenz der beiden Programme, wobei jedoch ergänzend die folgenden Anmerkungen als notwendig erscheinen:

(1.) Die Befehle ld und st wirken hinsichtlich ihrer ersten bzw. zweiten Adresse registerindirekt, d.h., die in r3 stehende Größe wird als Hauptspeicheradresse bzw. in unserem Fall als Adresse des Registers EAREG interpretiert; damit wird die entsprechende Speicherzelle im Hauptspeicher bzw. hier das Register im Interface-Baustein angesprochen. In der Schreibweise werden die Register-Adressierung und die registerindirekte Adressierung nicht unterschieden, da die jeweilige Adressierungsart eindeutig dem Befehlscode zugeordnet ist. Anders ist dies bei CISC-Prozessoren. Dort kommt die registerindirekte Adressierung alternativ zur Register-Adressierung vor, weshalb erstere durch Klammenschreibweise (Ri) kenntlich gemacht wird (2.2.3).

(2.) Der Befehl dec (decrement) dekrementiert das in seinem Adreßteil angegebene Register. An seiner Stelle könnte in diesem Programm ebensogut der Befehl sub r2,r4,r2 stehen (weil hier r4 die Konstante 1 als Dekrement enthält).

(3.) Der Befehl nop (no operation) dient bei der für RISCs typischen Fließbandprogrammierung dem Zweck, die korrekte Einhaltung der Reihenfolge der einzelnen Befehlausführungen programmierungstechnisch zu gewährleisten. Auf diesen Punkt wie auch auf die Optimierung des Programms in Verbindung mit der Fließbandverarbeitung wird im folgenden genauer eingegangen.

Fließbandkonflikte. Bei RISCs mit der ihnen eigenen mehrstufigen Fließbandverarbeitung treten zwei typische Problemfälle auf, die durch Hardware (bei Datenabhängigkeit) oder durch Software (bei Sprungabhängigkeit) gelöst werden (siehe dazu vertiefend 2.3.3, 2.3.6 und 2.3.7).

Datenabhängigkeit (true data dependency). Die in den Bildern 1-25 und 1-26 benutzte (wie bereits gesagt: unnütze) Befehlsfolge durchläuft das Fließband nur deshalb fehlerfrei, weil zwischen den einzelnen Befehlen keine sog. Datenabhängigkeiten bestehen. Unter Datenabhängigkeit versteht man, daß ein Ergebnis, das durch die Ausführung eines datenverarbeitenden Befehls entstanden ist, durch den nächsten oder den übernächsten Befehl weiterverarbeitet wird. Dieser Konfliktfall tritt immer dann auf, wenn die Ziel-Registeradresse eines Befehls gleich einer der Quell-Registeradressen im nächsten Befehl oder im übernächsten Befehl ist.

Angenommen, in Bild 1-26 lauteten die Befehle in den Speicherzellen 1231 und 1232: clr r5 (unverändert) und move r5,r6 (anstelle von move r1,r6). Dann müßten zur korrekten Ausführung dieser Befehlsfolge zwei wirkungslose Befehle, d.h. Befehle ohne Operationsausführung (nop-Befehle), zwischen diese Befehle geschrieben werden. Andernfalls würde nämlich ein falscher Operand, und zwar der, der vor (und nicht etwa nach) Ausführung des clr-Befehls in r5 stand, bei der Ausführung des (ja auf clr folgenden) move-Befehls benutzt. – Wer will, spiele die drei in diesem Zusammenhang interessanten Fälle in Bild 1-26 durch: die beiden Befehle clr r5 und move r5,r6 zuerst ohne nop-Befehl, dann mit 1 nop-Befehl und schließlich mit 2 nop-Befehlen dazwischen.

Datenabhängigkeit kommt recht häufig vor, auch in dem Programm in Bild 1-27 an mehreren Stellen. Wie aber kann das dort wiedergegebene Programm funktionieren: in Bild 1-27 stehen doch keine nop-Befehle zwischen Befehlen mit Datenabhängigkeit. Es funktioniert, weil die Hardware diesen Konflikt löst. Dazu gibt es zwei Möglichkeiten, eine erste, wenig aufwendige, wenig effiziente und eine zweite, aufwendigere, aber dafür sehr effiziente Möglichkeit:

(1.) Die Verarbeitung in den ersten beiden Stufen des Fließbands wird einen oder zwei Takte gestoppt, so daß das ALU-Ergebnis im Registerspeicher im richtigen Zeitpunkt zur Verfügung steht (just in time). Nachteilig bei diesem Sperren des Fließbands ist, daß sich die Programmausführung in gleicher Weise verlangsamt, wie wenn man nop-Befehle zwischen die Befehle mit Datenabhängigkeit einfügt. Man bezeichnet dieses für die gegenseitige Rücksichtnahme erforderliche „Sperren“ des Fließbands als *Interlock* und das dadurch bedingte „Abreißen“ des Fluxus im Fließband als *Stall*.

(2.) Im Falle einer Datenabhängigkeit, wenn also die Zieladresse in der Fließbandstufe 4 gleich einer der beiden Quelladressen in der Fließbandstufe 2 oder Fließbandstufe 3 ist, wird das ALU-Ergebnis nicht nur in den Registerspeicher geschrieben, sondern – wie in Bild 1-24 (S. 51) neben der ALU gestrichelt gezeichnet – darüber hinaus auch unmittelbar dem entsprechenden ALU-Eingang bzw. -Register zugeführt, also unter Umgehung des Registerspeichers. Auf diese Weise wird das korrekte Ergebnis des vorangehenden Befehls benutzt, ohne daß ein bzw. zwei nop-Befehle zwischen Befehlen mit Datenabhängigkeit nötig sind; dadurch wird natürlich die Programmausführung beschleunigt. Wegen des „Umgehens“ des Registerspeichers bzw. des „Überspringens“ der Fließbandregister bezeichnet man diese Technik mit *Register-Bypassing* oder *Feed-Forwarding*.

Sprungabhängigkeit. Wie gezeigt, werden die aus Datenabhängigkeiten entstehenden Fließbandkonflikte bei RISCs durch die Hardware gelöst. Daneben gibt es aber noch eine andere Art von Fließbandkonflikten, die ebenfalls in Hardware gelöst werden kann, aber auch oft in Software gelöst wird. Sie ergibt sich aus der Abhängigkeit der Sprungziele vom Ausgang der Abfrage der Bedingungsbits bei Sprungbefehlen, d.h., aus der bei der Fließbandverarbeitung auftretenden verzögerten Ausführung der Sprungbefehle (*delayed branching*). Man bezeichnet diese

Abhängigkeit als Sprung- oder Programmflußabhängigkeit (*procedural dependency*). – Eine einfache, dafür aber ineffiziente Hardwarelösung ist das Sperren des Fließbands (interlock). Effizientere Hardwarelösungen sind demgegenüber sehr viel aufwendiger. Idealerweise würde man beide Befehle für die Programmflußalternativen aus dem Programmspeicher holen. Das erfordert aber mit einem Zweiport-Cache und zwei Befehlsregistern einen unverhältnismäßig hohen Aufwand. Statt dessen wird versucht, mit vorzeitiger Erkennung der Sprungbefehle im Befehlstrom die Sprungentscheidung „branch taken“ bzw. „branch not taken“ vorherzusagen, so daß der Befehlstrom ggf. rechtzeitig auf die Sprungzieladresse umgelenkt werden kann.

Die *Sprungvorhersage* (*branch prediction*) kann – weniger aufwendig – aus dem Sprungbefehl selbst abgeleitet werden, z.B. aus einem Bit im Operationscode, das vom Compiler beeinflußt wird, oder sie kann aus der im Befehl angegebenen Zieladresse abgeleitet werden, z.B. als generelle Festlegung „branch taken“ bei Rückwärtssprüngen und „branch not taken“ bei Vorwärtssprüngen. Die Vorhersage kann aber auch – mehr aufwendig – aus der Vorgeschichte des Sprungbefehls gewonnen werden, indem die bei der letzten Befehlausführung ermittelte Sprungentscheidung für die nächste Ausführung des Sprungbefehls hergenommen wird. Realisiert wird das durch einen sog. *Branch-prediction/target-Cache*. In ihm werden die Befehlsadressen der zuletzt ausgeführten Sprungbefehle (als Tag-Einträge) sowie deren Zieladressen und eine Angabe, ob zuletzt eine Verzweigung stattgefunden hat oder nicht, als Sprunginformation gespeichert. – Bei beiden Vorhersage-Mechanismen müssen die im Vorgriff gelesenen Befehle verworfen werden, falls sich die Vorhersage bei Ausführung des Sprungbefehls als falsch erweist. Das führt dann doch zu einer Verzögerung im Fließband, kommt aber statistisch erheblich seltener vor.

Die vielleicht eleganteste Konfliktlösung, da sie allein durch Software erfolgt und vom Compiler ohne großen Aufwand bewerkstelligt werden kann, ist die, die Verzögerungen, die im Fließband aufgrund von Sprungabhängigkeiten entstehen und eigentlich durch nop-Befehle abgefangen werden müßten, durch geschicktes Umordnen von Befehlen nach Möglichkeit gar nicht erst entstehen zu lassen. Dies ist im folgenden ausführlich anhand unseres Programmbeispiels erläutert, und zwar in zwei Stufen: Zunächst wird die Entstehung dieses Fließbandkonflikts und deren Primitiv-Lösung durch Einfügen von nop-Befehlen geschildert. Danach wird das angedeutete Umordnen von Befehlen zum Zweck der Eliminierung dieser nop-Befehle beschrieben, was einer Programmoptimierung gleichkommt.

Im RISC-Programm in Bild 1-27 werden die Bedingungsbits im Condition-Code-Teil des Statusregisters (in Bild 1-24, S. 51, mit CC bezeichnet) durch den cmp-Befehl gesetzt und durch den nachfolgenden bne-Befehl (branch if not equal) ausgewertet. Im Fall „not equal“ ($r0 \neq r2$ mit $r0 = 0!$) soll nach m2 gesprungen werden; im Fall „equal“ ($r0 = r2$) soll der nächste Befehl ausgeführt werden. Dabei wird im Fließband bei der Ausführung des Programms nicht entsprechend der

üblichen Programmaufschreibung verfahren. Dazu müßte nämlich die Verarbeitung im Fließband einen Schritt angehalten werden. Statt dessen wird (wie beim Bypassing) die Fließbandverarbeitung nicht unterbrochen und der auf den Sprungbefehl folgende Befehl immer, also auch beim Wegsprung, ausgeführt. Um das zu verdeutlichen, ist – wie in Bild 1-27 eingetragen – der Anfang der Sprungpfeile im RISC-Programm generell einen Befehl später als im entsprechenden CISC-Programm eingetragen. Man bezeichnet diese Stelle nach einem Sprungbefehl treffend als *Delay-Slot*. Damit erklärt sich nun im RISC-Programm das Anhängen eines nop-Befehls an jeden Sprungbefehl, hier sowohl an den bne-Befehl am Ende der (bedingten) inneren Schleife als auch an den jmp-Befehl am Schluß des Programms, genauer: am Ende der (unbedingten) äußeren Schleife. Die Lösung des Konflikts erfolgt also hier durch Programmmodifikation, d.h. durch Software.

Programmoptimierung. Statt den Delay-Slot nach Sprungbefehlen – wie geschildert – mit den eigentlich unnützen nop-Befehlen zu füllen, verwendet man, soweit möglich, an deren Stelle andere, nutzbringende Befehle, d.h. Befehle, die sowieso im Programm ausgeführt werden, wenn auch an anderer Stelle. Dieses Vorgehen entspricht der Optimierung des Programms. Zum Ausfüllen eines Delay-Slots eignet sich am besten ein Befehl, der sonst vor dem Sprungbefehl ausgeführt würde. Seine Ausführung hat ja unabhängig davon zu erfolgen, ob durch den Sprungbefehl im Programm verzweigt wird oder nicht. Da dieser Befehl durch die Umstellung in seiner Ausführung verzögert wird, bezeichnet man ihn auch als *Delay-Befehl*. – In unserem Programmbeispiel ist es aufgrund der Struktur der inneren Schleife jedoch besser, im Delay-Slot des bne-Befehls nicht einen Befehl zu verwenden, der sonst vor bne ausgeführt würde, sondern den auf den Sprungbefehl folgenden ersten Befehl der Schleife. Hier tritt dann allerdings das Problem auf, daß der Delay-Befehl beim Abbruch der Schleife (d.h., wenn die Sprungbedingung nicht erfüllt ist und im Programm fortgefahren wird) nicht noch einmal ausgeführt werden darf, sondern annulliert werden muß.

Im RISC-Programm in Bild 1-27 wird also anstelle des nop-Befehls nach dem Befehl „bne m2“ der an der Stelle m2 stehende Befehl „dec r2“ in den Delay-Slot von „bne m2“ eingefügt. Das hat natürlich zur Folge, daß die Marke m2 nun um eine Zeile nach unten versetzt werden muß. Allerdings darf der dec-Befehl nicht von dort entfernt werden, sondern muß zur jeweils erstmaligen Ausführung in der äußeren Programmschleife an der alten Stelle erhalten bleiben. Diese Änderung ist in das in Bild 1-28 links angegebene RISC-Programm eingearbeitet. Aber Achtung: Im Falle des Nicht-Rückspringens, d.h. wenn die innere Programmschleife verlassen wird (dieser Fall tritt ein, wenn r0 = r2 ist), wird durch die erfolgte Änderung der dec-Befehl einmal zu viel ausgeführt. Das kann – wie hier in unserem Programm – bedeutungslos sein, das kann aber in anderen Fällen zu einem Fehler führen. In jedem Fall verändert sich der Ablauf des modifizierten Programms gegenüber dem Originalprogramm, hier gegenüber dem RISC-Programm in Bild 1-27 (S. 58).

Um diesen Sachverhalt bei der Programmoptimierung berücksichtigen zu können, sehen RISCs vielfach eine Modifikation der bedingten Sprungbefehle vor. Und zwar kann durch Anhängen eines sog. *Annullierungsbits* im Sprungbefehl – durch .a an bne im Programm in Bild 1-28 – das beschriebene Fehlverhalten vermieden werden. Dieses Bit bewirkt nämlich, daß beim Wegsprung der Delay-Befehl ausgeführt, im anderen Fall jedoch übersprungen wird. Somit haben wir es bei der Ausführung bedingter Sprungbefehle tatsächlich mit zwei Sprüngen zu tun: bei erfüllter Bedingung zum in der Adresse stehenden Sprungziel (durchgezogener Pfeil in Bild 1-28), bei Nichterfüllung der Bedingung zum übernächsten Befehl (gestrichelter Pfeil in Bild 1-28).

RISC-Programm Bild 1-27 in Pseudocode		in Maschinencode	Takte/ Befehl
set	eareg,r3	set	32768,r3
set	1,r4	set	1,r4
ld	r3,r1	ld	r3,r0,r1
st	r0,r3	st	r0,r3,r0
m1:	move r1,r2	m1:	or r0,r1,r2
	dec r2		sub r2,r4,r2
m2:	cmp r0,r2	m2:	Subcc r0,r2,r0
	bne.a m2		bne.a m2
	dec r2		sub r2,r4,r2
	st r4,r3	st	r4,r3,r0
	jmp m1	jmp	m1
	st r0,r3	st	r0,r3,r0

Bild 1-28. Gegenüberstellung eines RISC-Programms mit Pseudobefehlen und desselben RISC-Programms mit Maschinenbefehlen in symbolischer Darstellung

Anmerkungen. (1.) Das hier Gesagte gilt sinngemäß auch für unbedingte Sprungbefehle, z.B. den jmp-Befehl. Bei den unbedingten Sprungbefehlen hat die Einbeziehung des Annullierungsbits eigentlich keinen Sinn, wird aber dennoch genutzt, und zwar in umgekehrter Weise wie bei den bedingten Sprungbefehlen: Ist es gesetzt, so wird nur der Delay-Befehl beim Wegsprung nicht ausgeführt (siehe auch 2.2.4). (2.) Die Ausführung eines Sprungbefehls erfolgt in Stufe 3 des Fließbands, vgl. die entsprechenden Ziffern in Bild 1-24 (S. 51). Das hat zur Folge, daß jeder Sprungbefehl die Stufe 4 leer und somit einen Fließbandtakt nutzlos durchläuft, anders ausgedrückt, das Fließband nach Stufe 3 verläßt. Das bedeutet aber nicht, daß dadurch eine Verzögerung in der Fließbandverarbeitung auftritt.

Das Maschinenprogramm. Das eigentliche Maschinenprogramm für unsere kleine Aufgabenstellung, die Programmierung des Impulsgebers, ist ebenfalls in Bild 1-28 angegeben. Korrespondierend zu jedem Pseudobefehl, links, ist der entsprechende Maschinenbefehl, rechts, wiedergegeben, und zwar zusammen mit der Anzahl an Takten pro Befehl zur Berechnung des Wertes der Variablen ZEITK entsprechend den Vorgaben aus 1.3.1. Die Sterne bei den Zeitangaben weisen darauf hin, daß zu Beginn der Programmausführung das Fließband erst nach vier Takten gefüllt ist (**) bzw. daß die Hauptspeicherzugriffe bei Lade-/Speichere-Befehlen in Wirklichkeit ggf. länger als 1 Takt dauern (*).

Die Bedeutung der Maschinenbefehle geht bis auf einige Ausnahmen aus der in Bild 1-28 gezeigten Gegenüberstellung hervor. Die Ausnahmen betreffen die folgenden Punkte:

(1.) Bei der registerindirekten Adressierung ist – gewissermaßen zur Ausnutzung des Dreiausdrucks – eine dritte Registeradresse mit berücksichtigt. Der Zweck dieses zusätzlichen Registers liegt darin, zur Adresse (als dem Inhalt des ersten Registers) noch eine Distanzangabe (als Inhalt des zweiten Registers) zu addieren. Erst mit der auf diese Weise entstehenden Gesamtadresse wird der Hauptspeicher adressiert. In unserem Fall ist die Distanz wegen der Verwendung von r0 gleich Null und somit diese Erweiterung wirkungslos. – Zu dieser sog. registerindirekten Adressierung mit Indizierung siehe 2.2.3.

(2.) Der Befehl sub r2,r4,r2 entspricht in seiner Wirkung nur in Verbindung mit dem Befehl set 1,r4 (und daß r4 im Programm nicht verändert wird) dem dec-Befehl. Er ist aber nicht automatisch aus dem Befehl dec r2 durch den Assembler generierbar, da diesem die Kenntnis des Gesamtzusammenhangs des Programms fehlt. Statt dessen erzeugt der Assembler standardmäßig den Befehl sub r2,1,r2 mit 1 als kurzer Konstanten an der Stelle, wo sonst die Registerangabe steht. – Zu dieser Art der Angabe von *Direktoperanden* im Befehlswort siehe ebenfalls 2.2.3.

(3.) Der Befehl subcc beeinflußt im Gegensatz zu sub (ohne cc) die Bedingungsbits im Statusregister des Prozessors und liefert dementsprechend neben seinem „arithmetischen“ auch ein „logisches“ Ergebnis. Seine Zieladresse ist aber r0, so daß in diesem Fall – wie früher ausgeführt – das arithmetische Ergebnis „ins Leere“ geschrieben wird. Daß auch andere Befehle vielfach in beiden Varianten zur Verfügung stehen, sowohl mit als auch ohne cc-Zusatz, ermöglicht eine höhere Flexibilität bei der Fließbandprogrammierung. Zu solchen Besonderheiten einzelner Befehlscodes siehe 2.2.4.

2 Der Mikroprozessor

In Kapitel 1 wurde schwerpunktmäßig eine Einführung in den Aufbau, die Arbeitsweise und die Assemblerprogrammierung von Rechnersystemen gegeben und dabei ein vereinfachter CISC-Prozessor mit 16-Bit-Verarbeitungsbreite sowie ein einfacher RISC-Prozessor mit 32-Bit-Verarbeitungsbreite vorgestellt. Heutige CISC- und RISC-Prozessoren hoher Leistungsfähigkeit sind als 32-Bit- oder 64-Bit-Prozessoren ausgelegt, jedoch sehr viel komplexer als der RISC aus Kapitel 1. Dies liegt zum einen an einem hohen Grad an Parallelität der Verarbeitung, die sich in Fließbandstrukturen und in der Anzahl und Organisation von parallel arbeitenden Funktionseinheiten ausdrückt. Damit zusammenhängend können diese Prozessoren auch sehr hoch getaktet werden. Zum andern liegt dies an prozessorexternen wie prozessorinternen breiteren Daten- und ggf. Adresswegen von meist 64 Bit (auch bei 32-Bit-Prozessoren). Erstere ermöglichen eine höhere Übertragungsleistung („Busbandbreite“) für den Transport von Befehlen und Daten unter Nutzung von prozessorinternen Caches. Letztere erlauben es, größere Adressen bereitzustellen und somit den direkt adressierbaren Adressraum über die sonst vorhandene Begrenzung von 4G Adressen hinaus ausdehnen zu können.

Prozessoren mit 64-Bit-Datenbus und mit 32-Bit-Verarbeitung bezeichnen wir als *64/32-Bit-Prozessoren*. Mikroprozessoren (und insbesondere Mikrocontroller) gibt es häufig aber auch in „Spar“versionen mit reduzierter Breite der Datenbus-schnittstelle, bei einem 32-Bit-Prozessor auf z.B. 16 oder 8 Bits. Die Übertragung eines 32-Bit-Wortes kann dann nicht mehr in einem, sondern muß in zwei bzw. vier Buszyklen durchgeführt werden. Wir kennzeichnen solche Prozessoren als *16/32-Bit-* bzw. *8/32-Bit-Prozessoren*. Eingesetzt werden sie bei „Billig“lösungen von Mikroprozessorsystemen (jeder Bausteinanschluß bedeutet eine externe Leitungsverbindung, beansprucht also Platz und verursacht Kosten).

Die folgende Zusammenstellung gibt einen ersten Überblick über einige wichtige Merkmale von Mikroprozessoren, ausgehend von der 32-Bit-Technik.

- Arithmetisch-logische Verarbeitung von 32-Bit- und ggf. 64-Bit-Ganzzahlen, unterstützt durch einen 32-Bit- bzw. 64-Bit-Registersatz,
- arithmetische 32-Bit- und 64-Bit-Verarbeitung von Gleitpunktzahlen, unterstützt durch einen 64-Bit-Registersatz (ggf. erweiterte Formate mit 80 Bit),
- interne Datenwege mit 32 Bit oder breiter,
- 32-Bit-Schnittstelle zum Systemdatenbus oder breiter,

- Adreßformat von 32 Bit oder breiter,
- mehrere Datenformate, z.B. Bit, Bitfeld, Byte, Halbwort, Wort und Doppelwort; bei RISCs üblicherweise in der Verarbeitung auf Wort und Doppelwort eingeschränkt,
- Halbwort-, Wort- und Doppelwortzugriffe im Speicher ggf. nicht an Halbwort-, Wort- bzw. Doppelwortadressen gebunden, sondern mit beliebigen Byteadressen möglich (data misalignment); üblicherweise nicht bei RISCs,
- viele Möglichkeiten der Adreßmodifizierung durch den Prozessor; bei RISCs eingeschränkt,
- umfangreicher Befehlssatz, z.B. für Gleitpunktoperationen und zur Unterstützung höherer Programmiersprachen; letzteres bei RISCs eingeschränkt,
- byte-, halbwort- oder wortorientierte Befehlsdarstellung; bei RISCs ausschließlich wortorientiert,
- Fließbandverarbeitung von Befehlen; bei RISCs sehr ausgeprägt,
- Parallelarbeit mittels parallel arbeitender, für Befehlsklassen ausgelegten Funktionseinheiten (superskalare oder VLIW-Struktur), ggf. auch mittels Coprozessoren,
- mehrere Betriebsarten mit unterschiedlichen Privilegien,
- universelles Trap- und Interruptsystem (Ausnahmeverarbeitung); bei RISCs ggf. mit eingeschränkter Hardwareunterstützung.

Hinzu kommen übergeordnete Merkmale, wie

- interne Pufferung von Befehlen (Befehls-Cache, Befehlspuffer),
- interne Pufferung von Operanden (Daten-Cache),
- interne Speicherverwaltung (memory management unit, MMU),
- getrennte Internbusse für Daten- und Befehls-Cache-Zugriffe, mit jeweils einer eigenen Speicherverwaltungseinheit.

Bei *64-Bit-Mikroprozessoren* erfolgt die Verarbeitung von Daten und Adressen grundsätzlich mit 64 Bit Breite. Für viele Anwendungen ist hier jedoch die Darstellung und Speicherung von Daten in gewissem Maße „verschwenderisch“, so z.B. wegen des häufig vorkommenden Datenwertes Null, oder dadurch, daß bei Zahlen kleiner Wertebereiche ein Großteil des 64-Bit-Datenworts durch die Bits der Vorzeichenerweiterung belegt wird. Insofern ist der Einsatz solcher Prozessoren bei Arbeitsplatzrechnern (PCs, Desktops) nicht kritikfrei. Das Arbeiten mit 64-Bit-Adressen hingegen erlaubt eine Ausdehnung des Speicheradreßraums über die bei 32-Prozessoren üblichen 4Gbyte hinaus. Dies wird insbesondere bei Servern mit ihren großen Arbeitsspeichern genutzt. Letztlich stellt man aber auch hier den möglichen Adreßraum von 16 Exabyte nicht vollständig zur Verfügung,

sondern nutzt für die Speicheranwahl nur eine verkürzte (reale) Adresse von z.B. 40 Bit (1 Terabyte). Eine Erweiterung des Adressraums über 4 Gbyte hinaus ist auch bei 32-Bit-Prozessoren gebräuchlich. Hier bleibt die Adressverarbeitung jedoch auf 32 Bit beschränkt, und lediglich die Adressbreite für die Speicheranwahl wird erweitert, z.B. auf 36 Bit. Dieses geschieht durch die Speicherverwaltungseinheit.

In Kapitel 2 wollen wir nun die Funktions- und Strukturmerkmale von CISC- und RISC-Architekturen im Hinblick auf reale Mikroprozessorsysteme ausführlicher behandeln. Dazu beschreiben wir in Abschnitt 2.1 das Programmiermodell eines CISC-Prozessors, wobei wir uns stark an die früheren Mikroprozessoren und heutigen Mikrocontroller der MC680x0-Familie anlehnen, und in Abschnitt 2.2 das eines RISC-Prozessors mit starker Anlehnung an die SPARC-Architektur. Unter Programmiermodell verstehen wir dabei jene Prozessormerkmale, die bei der Assemblerprogrammierung „sichtbar“ werden, nämlich den Registersatz, die Datenformate, die Datentypen, den Datenzugriff, die Adressierungsarten, die Befehlsformate, den Befehlssatz, das Unterbrechungssystem und die Betriebsarten. In Abschnitt 2.3 erweitern wir die Betrachtungen dann auf die mit modernen superskalaren und VLIW-Prozessoren einhergehenden komplexeren Funktions- und Strukturaspekte. Übergeordnete Merkmale, wie das Puffern von Befehlen und Operanden in Caches und die Speicherverwaltung durch MMUs, betrachten wir gesondert in Kapitel 6.

Grundsätzlich gelten die in diesem und in den folgenden Kapiteln für Mikroprozessoren gemachten Angaben auch für *Mikrocontroller*, deren Kern ja ein Mikroprozessor ist, meist eine „abgemagerte“ Version eines auf dem Markt befindlichen Prozessors. Hierbei überwiegen zwar Prozessoren mit 8-Bit-Verarbeitungsbreite, es sind aber auch Mikrocontroller hoher Leistungsfähigkeit im Handel, die als Kern einen der modernen CISC- oder RISC-Prozessoren haben.

2.1 CISC-Programmiermodell

Für die Ausführungen in diesem Abschnitt sind Vorgriffe unvermeidbar. So werden wir z.B. zur Darstellung allgemeiner Prozessormerkmale die Beschreibung einiger Befehle vorziehen. Als Beispiel für die verwendete CISC-Befehlsdarstellung sei der Datentransportbefehl

MOVE src,dst

angegeben. Die Quellangabe src (source) und die Zielangabe dst (destination) werden durch die Adressierungsarten näher bestimmt.

Bei Mikroprozessoren ist, abweichend von der vereinfachten Struktur in Kapitel 1, die kleinste unmittelbar adressierbare Einheit das Byte, d.h., eine vom Prozessor auf den Adressbus ausgegebene Speicheradresse ist immer eine *Byteadresse*.

Der Speicher selbst ist üblicherweise an den Datenbus des Prozessors angepaßt und hat dementsprechend als Zugriffsbreite die eines Worts, d.h., eine *Speicherzelle* umfaßt bei einem 32-Bit-Prozessor 32 Bits. Der Zugriff auf den Speicher ist jedoch generell byte-, halbwort- oder wortweise möglich. Das Zugriffsformat wird dazu mit jedem Zugriff vom Prozessor über den Steuerbus an die Speicher-einheit übermittelt. Die Festlegung des Datenformats einer Operation geschieht beim Programmieren durch Suffixe, wie .B (byte), .H (half word) und .W (word), z.B. wird durch MOVE.B src,dst ein Byte von src nach dst transportiert.

2.1.1 Registersatz und Prozessorstatus

Als *Registersatz* bezeichnen wir die vom Programm direkt ansprechbaren Prozes-sorregister. Sie dienen einerseits zur Speicherung von Operanden, d.h. von Rechengrößen, und von Adressen und Indizes, d.h. von Information für die Adressierung der Operanden, und andererseits zur Speicherung des Prozessorstatus. Während für den Prozessorstatus immer spezielle Register vorgesehen sind, gibt es bzgl. der Operanden, Adressen und Indizes unterschiedliche Vor-gaben. Im einen Extrem steht für diese Information der ganze Registersatz zur Verfügung, bei dem jedes einzelne Register als *Universalregister* für jede der Informationsart genutzt werden kann. Im andern Extrem gibt es vorwiegend *Spezialregister*, insbesondere für Adressen und Indizes, und die für die Operan-denspeicherung vorgesehenen Register sind dann oft noch mit Sonderfunktionen belegt, indem z.B. ein bestimmtes Register implizit als Zählregister für String-Befehle benutzt wird. – Bild 2-1 zeigt als Beispiel einen relativ homogenen Regi-stersatz, bestehend aus einem allgemeinen Registerspeicher mit universeller Registernutzung, einigen zusätzlichen speziellen Adreßregistern sowie einem Statusregister. Bei im Prozessor vorhandenen Caches und Speicherverwaltungs-einheiten wird ein solcher Registersatz durch Steuerregister zu deren Verwaltung ergänzt. Zu den gezeigten Registern nachfolgend einige Details.

Registerspeicher. Der Registerspeicher in Bild 2-1 umfaßt acht allgemeine 32-Bit-Register mit den Bezeichnungen R0 bis R7. Sie werden unmittelbar im Adreßteil der Befehle adressiert und können alle in gleicher Weise als Arbeits-register zur Speicherung von Operanden eingesetzt werden. Die Register dienen außerdem zur Speicherung von Adressen und von Indizes als Distanzangaben indizierter Adressierung (siehe 2.1.3). Operandenzugriffe im Byte- und Halbwortformat wirken auf die Bitpositionen 7 bis 0 bzw. 15 bis 0; Wortzugriffe be-ziehen sich auf den gesamten Registerinhalt; sofern Doppelwortzugriffe erlaubt sind, betreffen sie die Inhalte zweier aufeinanderfolgender Register. Adressen und Indizes haben üblicherweise Wortformat, im Halbwortformat belegen sie die Bits 15 bis 0. – *Anmerkung:* Bei den Prozessoren der MC680x0-Familie, auf die nach-folgend immer wieder Bezug genommen wird, gibt es zwei Registerspeicher mit

jeweils acht 32-Bit-Registern für Operanden (D0 bis D7) und acht 32-Bit-Registern für Adressen und Indizes (A0 bis A7).

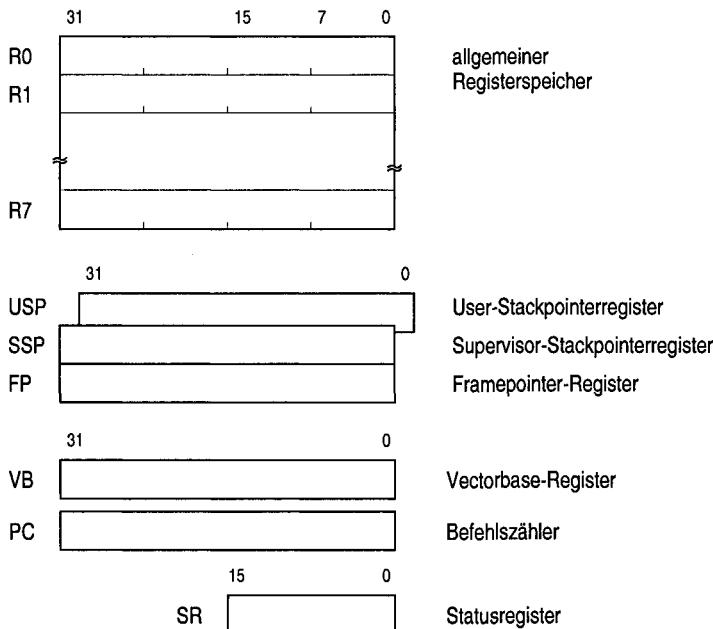


Bild 2-1. Programmierbarer Registersatz

Stackpointerregister. Das sind in Bild 2-1 USP und SSP. Jedes der Register adressiert mit seinem Inhalt, einem *Stackpointer*, einen Speicherbereich, der als *Stack* (Kellerspeicher, Stapspeicher) organisiert ist. Ein solcher Speicher kann in seiner Zugriffsorganisation mit einem Stapel verglichen werden. Stapellemente können nur oben aufgelegt bzw. oben entnommen werden. In einem Stack werden also die nacheinander eintreffenden Daten in aufeinanderfolgenden Speicherplätzen gespeichert und in umgekehrter Reihenfolge wieder gelesen. Üblicherweise wird der Stack mit absteigender Adresszählung gefüllt und mit aufsteigender Adresszählung geleert. Der Stackpointer zeigt dabei auf den zuletzt belegten Speicherplatz, d.h. auf den obersten (jüngsten) Stackeintrag.

Bei einer Schreiboperation wird zunächst der Stackpointer dekrementiert, so daß er auf die erste freie Speicherzelle des Stacks zeigt; danach wird das Datum unter dieser Adresse gespeichert. Bei einer Leseoperation wird zunächst die durch den Stackpointer adressierte Zelle gelesen und danach der Stackpointer inkrementiert, so daß er auf den davorliegenden Eintrag zeigt (*LIFO-Prinzip*: last-in first-out). Das Dekrement und das Inkrement haben abhängig vom Datenformat die Werte 1 (Byte), 2 (Halbwort) oder 4 (Wort). – Die Arbeitsweise des Stacks soll anhand der beiden *Stack-Befehle* PUSH.W src und POP.W dst gezeigt werden. PUSH.W

schreibt den Quelloperanden src als Wort auf den Stack, POP.W liest den obersten Worteintrag vom Stack und schreibt ihn an den Zielort dst. Das Stackpointerregister wird zur Bildung von Wortadressen jeweils um den Wert 4 verändert (Bild 2-2).

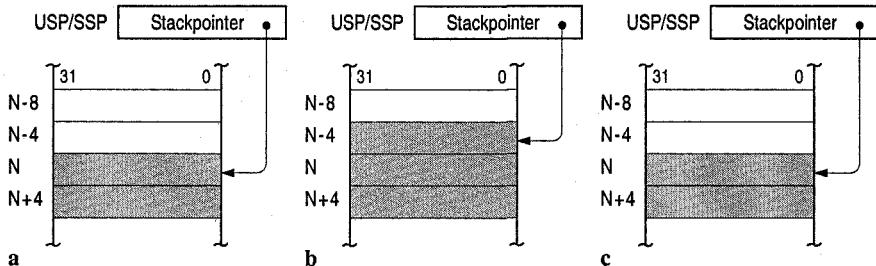


Bild 2-2. Schreiben und Lesen des Stacks mit den Befehlen PUSH.W und POP.W, a vor PUSH.W, b nach PUSH.W, c nach POP.W

Beide Stackpointerregister haben aus Sicht des Programmierers dieselbe Registeradresse, werden jedoch durch die im Statusregister vorgegebene Betriebsart User-Modus oder Supervisor-Modus unterschieden. Dementsprechend bezeichnet man die beiden Stackpointer als User-Stackpointer (USP) und als Supervisor-Stackpointer (SSP) und die zugehörigen Stacks als User-Stack und Supervisor-Stack. Beide Stacks werden außer von den PUSH- und POP-Befehlen zur Datenspeicherung auch von den Befehlen BSR/JSR (branch/jump to subroutine) und RTS (return from subroutine) zur Verwaltung von Programmadressen für den Unterprogrammanschluß benutzt. Der Supervisor-Stack wird darüber hinaus vom Prozessor zur Statusspeicherung bei der Trap- und Interruptverarbeitung verwendet. Zurückgeladen wird der Status mit dem Befehl RTE (return from exception). – Anmerkung: Bei den MC680x0-Prozessoren bilden die beiden Stackpointerregister bezüglich ihrer Adressierung das Register A7.

Framepointerregister. Stacks werden u.a. zur Speicherung der Parameter und lokalen Daten von Unterprogrammen (Prozeduren) eingesetzt, wobei man den zu einem Unterprogramm gehörenden Stackbereich als Rahmen (frame) bezeichnet. Zugriffe innerhalb eines solchen Rahmens erfolgen, da sich der Stackpointer verändern kann, meist relativ zu einer für diesen Rahmen festen Basisadresse, dem *Framepointer*. Der jeweils aktuelle Framepointer wird dazu in einem Framepointerregister FP verwaltet. Die Lage des Rahmens und damit der Framepointer können sich für ein und dasselbe Unterprogramm bei unterschiedlichen Unterprogrammaufrufen verändern (dynamische Speicherverwaltung, siehe auch 3.3.2).

Vectorbase-Register. Das Vectorbase-Register VB enthält die Basisadresse der sog. *Vektortabelle*. In dieser Tabelle sind die Startadressen für die Unterbrechungs Routinen aller möglichen Trap- und Interruptanforderungen und ggf. weitere Statusinformationen gespeichert (Trap- und Interruptvektoren). Durch bloßes

Ändern der Basisadresse ist es möglich, schnell zwischen mehreren Vektortabellen umzuschalten, z. B. beim Task-Wechsel.

Befehlszähler. Der Befehlszähler (*program counter, PC*) enthält die Adresse des jeweils nächsten Befehls. Da bei CISCs die Befehle entweder aus Vielfachen von Bytes (byteorientiertes Befehlsformat) oder von Halbworten (halbwortorientiertes Befehlsformat) bestehen, wird der Registerinhalt byteweise oder halbwortweise weitergezählt. Er kann ferner durch die in einem Sprungbefehl angegebene Sprungzieladresse überschrieben werden.

Statusregister. Das Statusregister SR (in Bild 2-1 ein 16-Bit-Register in Anlehnung an die MC680x0-Prozessorfamilie) gibt zum einen den aktuellen Mikroprozessorzustand nach jeder abgeschlossenen Befehlausführung und zum andern die momentane Betriebsart des Prozessors an. Dazu ist es in der Detaillierung nach Bild 2-3 funktionsmäßig in zwei Bytes, das Userbyte und das Supervisorbyte unterteilt.

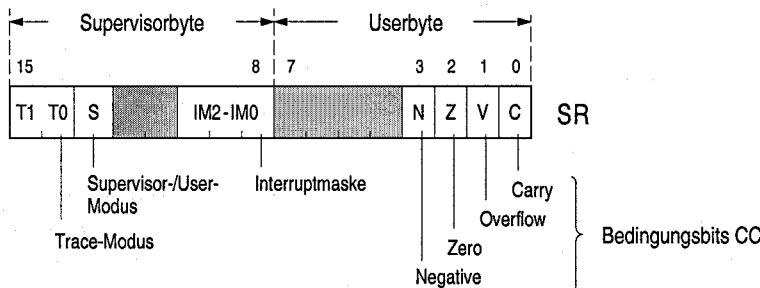


Bild 2-3. Statusregister (in Anlehnung an die MC680x0-Prozessorfamilie)

Das Userbyte enthält die Zustandsinformation des Prozessors in Form der *Bedingungsbits* N, Z, V und C (*condition code, CC*), die von der ALU bei der Ausführung fast aller Befehle beeinflußt werden und damit Aussagen über das Ergebnis arithmetischer und logischer, aber auch anderer Operationen machen. Die Bedingungsbits werden von den bedingten Sprungbefehlen zur Überprüfung ihrer Bedingungen ausgewertet. Im einzelnen haben sie folgende Funktion:

- Das *Carrybit C* (*Übertragsbit*) zeigt mit C = 1 den Übertrag bei arithmetischen Operationen an. Bei vorzeichenlosen Dualzahlen bedeutet dies Überlauf, d.h. das Überschreiten des Zahlenbereichs (siehe auch 1.1.4).
- Das *Overflowbit V* (*Überlaufbit*) zeigt mit V = 1 den Überlauf beim Überschreiten des Zahlenbereichs der 2-Komplement-Zahlen an (siehe auch 1.1.4).
- Das *Zerobit Z* (Nullbit) zeigt mit Z = 1 an, daß das Resultat einer Operation gleich Null ist.
- Das *Negativebit N* (Negativbit) zeigt mit N = 1 an, daß das Resultat einer Operation, wenn es als 2-Komplement-Zahl betrachtet wird, negativ ist. Das

N-Bit wird dementsprechend gleich dem höchstwertigen Bit des Resultats gesetzt.

Das Supervisorbyte beschreibt die *Betriebsart* des Prozessors in Form von *Modusbits* (mode bits), die i.allg. während der Ausführung eines Programms unverändert bleiben. Sie können nur im Supervisor-Modus durch bestimmte privilegierte Befehle verändert werden und werden darüber hinaus bei der Trap- und Interruptbehandlung implizit beeinflußt. Die Modusbits haben folgende Funktion:

- Das Supervisorbit S gibt mit $S = 1$ die privilegierte Betriebsart *Supervisor-Modus* und mit $S = 0$ die untergeordnete Betriebsart *User-Modus* vor. Der Supervisor-Modus gewährt gegenüber dem User-Modus erweiterte Verarbeitungsmöglichkeiten, z.B. das Ausführen privilegierter Befehle. – *Anmerkung:* Bei Prozessoren mit vier Betriebsarten unterschiedlicher Privilegien sind entsprechend zwei Modusbits vorhanden (z.B. beim Pentium).
- Die Trace-Bits T0 und T1 erlauben zwei *Trace-Modi*. Diese führen entweder nach jeder Befehlsausführung oder nur nach jeder programmverzweigenden Befehlsausführung zu einer Programmunterbrechung (*Trace-Trap*) und zur Ausführung eines Trace-Programms. Das Trace-Programm kann z.B. dazu genutzt werden, den Prozessorstatus anzuzeigen und anschließend das unterbrochene Programm fortzusetzen, das dann nach der nächsten Befehlsausführung bzw. der nächsten Programmverzweigung wieder unterbrochen wird (Programmtest, siehe Anm. in 2.2.4: Trap-Befehle).
- Die *Interruptmaske* IM0 bis IM2 gibt die *Prioritätsebene* des Prozessors und damit die Priorität des laufenden Programms in Bezug auf die Behandlung von Programmunterbrechungen an (in Anlehnung an die MC680x0-Prozessorfamilie). Die Prioritäten nehmen mit kleiner werdendem Wert ab, d.h., Ebene 7 hat die höchste und Ebene 0 die niedrigste Priorität. Externe *Unterbrechungsanforderungen* an speziellen Interruptcode-Eingängen des Mikroprozessors können das laufende Programm nur unterbrechen, wenn der Interruptcode (*Interruptebene*) der Anforderung eine höhere Priorität hat als die Ebene, in der der Prozessor gerade arbeitet. Eine Ausnahme bildet eine Anforderung in der Ebene 7, der immer stattgegeben wird (*maskierbare* und *nichtmaskierbare Interrupts*, siehe 5.5). – *Anmerkung:* Bei Prozessoren ohne diese Mehrebenenstruktur ist den maskierbaren Interruptanforderungen meist nur ein Interrupteingang zugewiesen, und dementsprechend auch nur ein Interruptmaskenbit vorhanden. Ein weiterer Interrupteingang dient den nichtmaskierbaren Interruptanforderungen.

Prozessorstatus. Der elementare Status des Prozessors wird durch den Befehlszähler PC und das Statusregister SR beschrieben. Diese beiden Register enthalten die Mindestinformation, die notwendig ist, um ein Programm, das zwischen zwei Befehlsausführungen unterbrochen wurde, fortsetzen zu können. Diese Information wird bei Programmunterbrechungen durch Traps oder Interrupts vom Prozes-

sor automatisch auf den Supervisor-Stack gespeichert. Die weiteren Spezialregister werden entweder ebenfalls automatisch oder, wie die allgemeinen Prozessorregister, nach Bedarf vom Unterbrechungsprogramm auf den Stack gerettet. Unterstützt wird dies ggf. durch spezielle Transportbefehle, die es erlauben, die Inhalte aller in einer Registerliste angegebenen Register innerhalb einer Befehlsausführung zu transportieren.

2.1.2 Datenformate, Datentypen und Datenzugriff

Datenformate und Datentypen. Als *Datentypen* bezeichnet man (aus Hardware-sicht) die unterschiedlichen Informationsarten, wie sie durch die Befehle des Mikroprozessors direkt verarbeitet werden. Sie sind charakterisiert durch eine bestimmte Anzahl von Bits (das *Datenformat*) und deren inhaltliche Bedeutung (die Interpretation). Die Interpretation wird durch die einzelnen logischen und arithmetischen Befehle des Prozessors vorgegeben, wobei die zu einem bestimmten Datentyp gehörenden Befehle die Operanden in gleicher Weise interpretieren, z.B. Gleitpunktbefehle ihre Operanden als Gleitpunktzahlen. Die gebräuchlichsten Datentypen eines 32-Bit-Mikroprozessors, basierend auf den Standardformaten Byte, Halbwort und Wort und den speziellen Datenformaten Bit, Halbbyte und Doppelwort, sind

- Zustandsgröße (1 Bit in einem der Standardformate) mit den Werten 0 und 1,
- Bitvektor (Standardformate) als Zusammenfassung von Zustandsgrößen,
- BCD-Ziffern in gepackter Form (2, 4 oder 8 Halbbytes in den Standardformaten zusammengefaßt),
- vorzeichenlose Dualzahl (Standardformate),
- 2-Komplement-Zahl (Standardformate),
- Gleitpunktzahl (Wort und Doppelwort).

Daneben werden Bitvektoren, vorzeichenlose Dualzahlen und 2-Komplement-Zahlen auch im Datenformat *Bitfeld* dargestellt und verarbeitet. Dieses Format hat eine variable Bitanzahl von bis zu 32 Bits und wird im Speicher durch eine Byteadresse, einen darauf bezugnehmenden Bitfeld-Offset und die Angabe der Bitfeldlänge angesprochen. Zur Verarbeitung wird ein Bitfeld z.B. rechtsbündig in den Registerspeicher des Prozessors geladen und um die zum gewählten Standardformat fehlenden höherwertigen Bits ergänzt. Abhängig vom Transportbefehl sind dies 0-Bits (zero extension) oder Kopien des höchstwertigen Operandenbits (sign extension). Weiterhin werden aufeinanderfolgend gespeicherte Bytes, Halbwörter oder Wörter im Datenformat *String* verarbeitet. Typische Anwendungen sind Byte-Strings, bestehend aus ASCII-Zeichen, und Byte-, Halbwort- oder Wort-Strings, bestehend aus Dualzahlen oder 2-Komplement-Zahlen. – Bild 2-4 zeigt die genannten Datentypen und Datenformate.

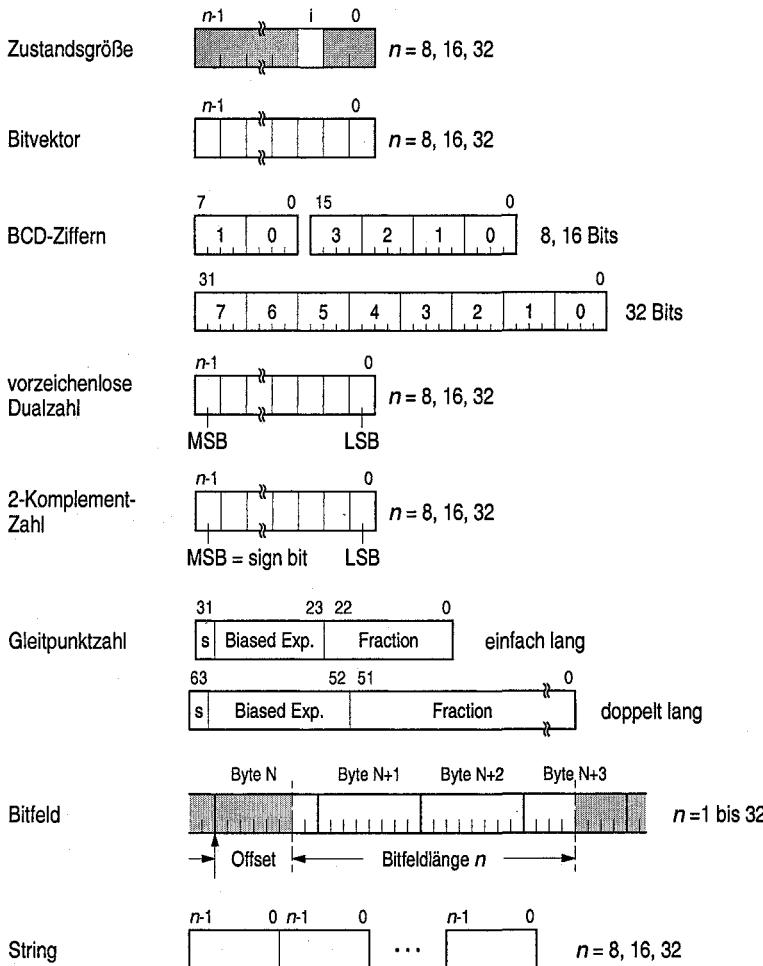


Bild 2-4. Datentypen und Datenformate (in Anlehnung an die MC680x0-Prozessorfamilie)

Datenzugriff. Direkt zugreifbar im Speicher sind das Byte, das Halbwort und das Wort, wobei sich deren Adressen, unabhängig von den Datenformaten, üblicherweise auf Bytегrenzen beziehen. Man spricht deshalb auch von *Byteadressen* und nennt einen solchen Speicher byteadressierbar. Die Speicherzugriffsbreite hingegen ist üblicherweise gleich der Datenbusbreite, d.h. bei einem 32-Bit-Prozessor gleich 32 Bit. Man sagt, der Speicher ist *wortorganisiert*. Die Zugriffsbreite kann jedoch bei einem Prozessor, der eine dynamische Festlegung der Datenbusbreite vorsieht (*dynamic bus sizing*), auf 16 Bit (halbwortorganisiert) oder 8 Bit (byteorganisiert) reduziert sein. Meist handelt es sich dabei um Speicher oder Register in Interface-Einheiten. Der Prozessor sorgt dann dafür, daß trotz der gegenüber dem Datenbus reduzierten Zugriffsbreite die Adressierung des Speichers bzw. der Register mit aufeinanderfolgenden Adressen möglich ist (siehe dazu 5.2.3). Die

Zugriffsbreite kann bei einem Prozessor mit breiterem Datenbus auch größer als das Wortformat sein, z.B. bei einem 64/32-Bit-Prozessor gleich 64 Bit.

Bei einem Prozessor ohne Dynamic-Bus-Sizing und einem Speicher oder einer Interface-Einheit mit geringerer Breite, als sie der Datenbus hat, ist die Adressierung aufeinanderfolgender Speicherzellen nur mit Sprüngen in der Adresszählung möglich. Die jeweils dazwischenliegenden Adressen würden Datentransfers (Schreiben) bzw. Transferversuche (Lesen) in jenen Bytepositionen des Datenbusses zur Folge haben, an die der Speicher nicht angeschlossen ist. So können z.B. die Speicherzellen oder Register einer 8-Bit-Einheit bei einem 32-Bit-Datenbus nur mit Adresssprüngen von vier adressiert werden.

Sind bei einem 32-Bit-Prozessor und einem Speicher mit einer Zugriffsbreite von 32 Bit Halbwort- und Wortoperanden so gespeichert, daß ihre Adressen ganzzahlige Vielfache der durch das Datenformat vorgegebenen Byteanzahl sind, d.h. durch 2 bzw. 4 teilbar sind, so läßt sich jeder Operandenzugriff mit einem einzigen Speicherzugriff durchführen (Bild 2-5a). Man bezeichnet diese Speicherung auch als „Ausrichten“ der Daten (*data alignment*); die Daten stehen bzgl. ihrer Datenformate an ihren „natürlichen“ Grenzen. Bei einem 16/32-Bit-Prozessor und einem Speicher mit einer Zugriffsbreite von 16 Bit gilt als Data-Alignment das Ausrichten der Daten innerhalb der Halbwortgrenzen (Bild 2-5b). Dabei können Wortoperanden gerade oder ungerade Halbwortadressen haben; sie erfordern immer zwei Speicherzugriffe (z.B. MC68000 [Motorola 1982]).

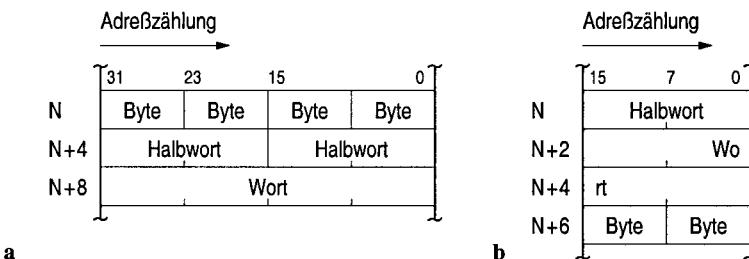


Bild 2-5. Data-Alignment im Speicher, a bei einem 32-Bit-Prozessor, b bei einem 16/32-Bit-Prozessor

Die meisten heutigen CISC-Mikroprozessoren schreiben das Ausrichten der Daten nicht vor, sondern erlauben das Speichern von Halbwort- und Wortoperanden an beliebigen Bytegrenzen (*data misalignment*). Das hat den Vorteil einer lückenlosen Nutzung des Speichers bei beliebiger Mischung der Datenformate. Gegenüber dem Data-Alignment hat es jedoch den Nachteil, daß bei Halbwort- und Wortoperanden, die nicht innerhalb der Zugriffsgrenzen des Speichers liegen, zusätzliche Speicherzugriffe erforderlich sind (Bild 2-6a). Die Anzahl der Zugriffe kann sich unter Ausnutzung des Dynamic-Bus-Sizing, d.h. beim Einsatz von Speichern und Registern mit geringeren Zugriffsbreiten, noch erhöhen (Bild

2-6b). – Zu Data-Alignment, Data-Misalignment und Dynamic-Bus-Sizing siehe auch 5.2.3.

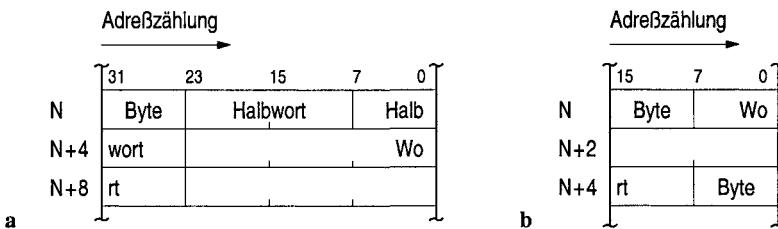


Bild 2-6. Data-Misalignment bei einem 32-Bit-Prozessor mit Dynamic-Bus-Sizing, a in einem wortorganisierten Speicher, b in einem halbwortorganisierten Speicher

In den Bildern 2-5 und 2-6 wurde von der sog. *Big-endian-Byteanordnung* (*big-endian byte ordering*) ausgegangen, d.h., der Prozessor adressiert Halbwort- und Wortoperanden jeweils an ihrem höchstwertigen Byte, und die weiteren Bytes haben aufsteigende Adressen (Adresszählung in den Bildern von links nach rechts, z.B. Motorola). Andere Prozessoren hingegen benutzen die *Little-endian-Byteanordnung* (*little-endian byte ordering*), d.h., sie adressieren jeweils das niedrigstwertige Byte mit dann ebenfalls aufsteigenden Adressen für die weiteren Bytes (Adresszählung in den Bildern von rechts nach links, z.B. Intel). Unabhängig davon werden Operanden im Registerspeicher des Prozessors so gespeichert, daß das niedrigstwertige Byte die Bitpositionen 0 bis 7 belegt und sich höherwertige Bytes in den aufsteigenden Bitpositionen anschließen. Wir werden im folgenden, wenn nicht anders erwähnt, von einer Adressierung entsprechend den Bildern 2-5 und 2-6, also von der *Big-endian-Byteanordnung* ausgehen. – Zu Big-endian- und Little-endian-Byteanordnung siehe auch 5.2.4.

2.1.3 Adressierungsarten und Befehlsformate

In der vereinfachten CISC-Mikroprozessorstruktur in Kapitel 1 wurde die Speicheradresse eines Operanden direkt im Befehlswort angegeben. Reale CISC-Mikroprozessoren sehen darüber hinaus verschiedene Möglichkeiten der Adressmodifikation vor, wobei die tatsächliche Adresse (*effektive Adresse*) erst während der Befehlausführung aus mehreren Teilen, die im Befehl oder in Registern stehen, berechnet wird (Adressrechnung zur Laufzeit, *dynamische Adressrechnung*). Hierfür sind hauptsächlich folgende Gründe maßgebend:

- Die Adresse eines Operanden ergibt sich häufig erst zur Laufzeit aus Adressberechnungen durch das Programm. Sie sind somit zur Zeit der Programmherstellung noch nicht bekannt.

- Der Operandenzugriff eines Befehls kann sich bei wiederholter Befehlausführung, z.B. in einer Programmschleife, auf aufeinanderfolgende Adressen beziehen, die erst zur Laufzeit zu bilden sind.
- Die Adresse eines Elements einer Datenstruktur setzt sich z.B. additiv aus der Basisadresse der Datenstruktur und der Distanz innerhalb der Datenstruktur zusammen. Da die Basisadresse oder die Distanz oft erst zur Laufzeit bekannt ist, kann die effektive Adresse des Elements erst dann berechnet werden.
- Das Zerlegen von Adressen in eine Basisadresse, die in einem Register steht, und in Distanzen erleichtert die Erzeugung von verschiebbaren Variablenbereichen und verschiebbarem Programmcode. Darüber hinaus haben die Distanzen häufig eine geringere Bitanzahl als die Adressen, wodurch die Befehle kürzer werden.

Im folgenden geben wir einen Überblick über die gebräuchlichsten Adressierungsarten von CISC-Mikroprozessoren, wobei wir uns wieder an die MC680x0-Prozessorfamilie anlehnen. Die effektive Adresse wird als 32-Bit-Adresse ermittelt. Adreßdistanzen, d.h. *Displacements* (Konstanten im Befehl) und *Indizes* (Variablen im Registerspeicher), werden bei der Adreßrechnung als 2-Komplement-Zahlen interpretiert, so daß Bezüge zu höheren und niedrigeren Adressen angebbar sind. Displacements, Indizes und kurze Adressen (short addresses), die weniger als 32 Bits, d.h. 8 oder 16 Bits umfassen, werden vom Prozessor zur Adreßrechnung auf das 32-Bit-Format erweitert. Dies geschieht durch *Sign-Extension*, d.h. durch Kopieren des höchstwertigen Bits in die fehlenden, höherwertigen Bitpositionen.

In den zur Illustration der Adressierungsarten verwendeten Abbildungen bezeichneten einfache Pfeile die Aktion „Transport“ und die mit einem Punkt versehenen Pfeile die Aktion „Adressierung“. Die Assemblerschreibweisen (Syntax) zur Kennzeichnung der Adressierungsarten sind in Anlehnung an auf dem Markt befindliche Assembler gewählt. Als Registerbezeichnungen verwenden wir verkürzte Schreibweisen, wie „Rn“ für die allgemeinen Prozessorregister und „Xn“ für die Benutzung dieser Register als Indexregister. – Zur Darstellung von Adressen, Operanden etc. durch Ausdrücke siehe 3.1.2.

Direktoperand-Adressierung (immediate). Der Operand steht als Konstante im Befehl (*Direktoperand*); eine eigentliche Adressierung entfällt damit (Bild 2-7a). Direktooperanden können, da sie Konstanten sind, nur als Quellgrößen angegeben werden. Als Assemblerschreibweise dient z.B. ein dem Operanden vorangestelltes # -Zeichen. – *Beispiele:* MOVE.B #125,R3 transportiert den Wert 125 im Byteformat nach R3; MOVE.W #BASE,POINTER transportiert den dem Symbol BASE zugeordneten Wert (Adresse) im Wortformat in eine Speicherzelle mit der symbolischen Adresse POINTER.

Assemblerschreibweise: #Operand

Speicher-Adressierung (direct, absolute memory). Die effektive Adresse steht als absolute Adresse im Befehl; der Operand steht im Speicher (Bild 2-7b). Die Adresse hat üblicherweise 32 Bits, kann aber auch als 16-Bit-Kurzadresse angegeben sein. Kurzadressen liegen aufgrund der Sign-Extension in den ersten und den letzten 32 Kbyte des Adreßraums. Die Adressierungsart wird ohne besondere Kennzeichnung durch eine symbolische oder numerische Adreßangabe beschrieben. – *Beispiele:* MOVE.W LOC1,LOC2 enthält zwei symbolische, und ADD.B #325,0xFFC0 eine numerische Adreßangabe.

Assemblerschreibweise: symbolische oder numerische Adreßangabe

Register-Adressierung (register). Die Adresse steht als kurze Registeradresse im Befehl (z.B. 3-Bit-Adresse); der Operand steht im Register (Bild 2-7c). Die Adressierungsart ist durch das Registersymbol gekennzeichnet. Als Register stehen die allgemeinen Register Rn, und, wenn vorhanden, auch spezielle Register, wie SP und FP, zur Verfügung. (In der formalen Darstellung beschränken wir uns im folgenden auf Rn.) – *Beispiel:* MOVE.W R0,SP transportiert den Wortinhalt von R0 in das Stackpointerregister SP.

Assemblerschreibweise: Rn

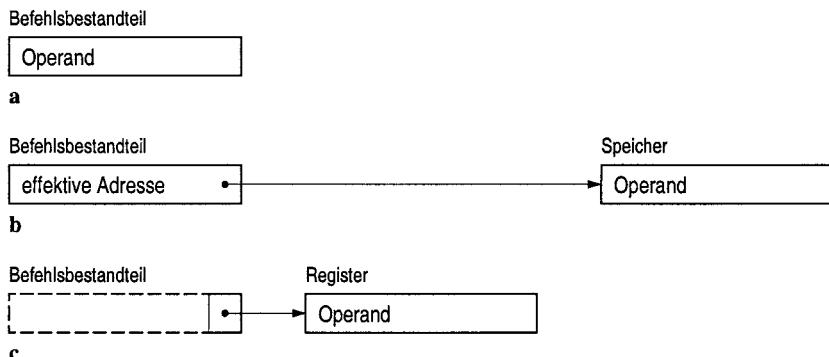


Bild 2-7. Adressierungsarten. a Direktoperand-, b Speicher-, c Register-Adressierung

Registerindirekte Adressierung (register indirect). Die effektive Adresse steht im Register; der Operand steht im Speicher. Die Adressierung erfolgt indirekt über den Registerinhalt (Bild 2-8a). Die Adressierungsart wird durch runde Klammern mit der Bedeutung „Inhalt von Register“ ausgedrückt. – *Beispiel:* MOVE.H (R0),R1 transportiert einen 16-Bit-Speicheroperanden, dessen Adresse in R0 steht, nach R1.

Assemblerschreibweise: (Rn)

Die registerindirekte Adressierung erlaubt die Datenadressierung über sog. *Zeiger* (*pointer*), die meist erst zur Programmlaufzeit ermittelt werden. Solche Zeiger

werden z.B. auch als Parameter bei Unterprogrammaufrufen übergeben (siehe 3.3.2).

Erweiterungen der registerindirekten Adressierung sind die **registerindirekte Adressierung mit Prädekrement** (Bild 2-8b) und **mit Postinkrement** (Bild 2-8c). Bei ihnen wird die im Register stehende Adresse vor ihrer Benutzung automatisch dekrementiert (autodecrement) bzw. nach ihrer Benutzung automatisch inkrementiert (autoincrement), und zwar abhängig von dem im Befehl angegebenen Datenformat um den Wert 1 (Byte), 2 (Halbwort) oder 4 (Wort). Somit wird der Registerinhalt hier verändert. Das Prädekrementieren wird für den Assembler durch ein vor der Klammer stehendes Minus-Zeichen, das Postinkrementieren durch ein nach der Klammer stehendes Plus-Zeichen gekennzeichnet.

Assemblerschreibweise (Prädekrement): -(Rn)

Assemblerschreibweise (Postinkrement): (Rn)+

Beide Adressierungsarten werden bei Prozessoren, die keine Push- und Pop-Befehle haben, zur Adressierung von Stacks eingesetzt (siehe 2.1.4). Sind Push- und Pop-Befehle vorhanden, so sind diese Adressierungsarten üblicherweise nicht explizit verfügbar. Weiterhin eignen sie sich für aufeinanderfolgende Zugriffe auf Datenfelder, z.B. in einer Programmschleife, mit wahlweise abwärts- oder aufwärtszählender Adressierung.

Beispiel 2.1. ► Registerindirekte Adressierung mit Postinkrement. Im folgenden Programmausschnitt wird ein Feld von 1000 Bytes mit dem Wert Null gefüllt. Für den Ausdruck FELD+1000 setzt der Assembler den um 1000 erhöhten Wert der Adresse FELD als Direktoperand ein.

```

FELD: DS.B    1000
      :
      MOVE.W #FELD,R1
LOOP: MOVE.B #0,(R1) +
      CMP.W #(FELD+1000),R1
      BNE     LOOP
      :

```

◀

Relative Adressierung (relative). Hierbei wird zu der in einem Register stehenden Speicheradresse eine konstante Abstandsgröße, ein *Displacement*, als 2-Komplement-Zahl addiert. Handelt es sich bei dem Register um ein allgemeines Prozessorregister Rn, so bezeichnet man die Adressierungsart als **registerindirekte Adressierung mit Displacement** (Bild 2-9a). Da hier die im Register stehende Adresse als *Basisadresse* wirkt, die durch die Adreßrechnung nicht verändert wird, spricht man auch von *basisrelativer Adressierung*. In der Assemblerschreibweise steht das Displacement als symbolische oder numerische Angabe vor der in runden Klammern angegebenen allgemeinen Registeradresse.

Assemblerschreibweise: displ(Rn)

Die Anwendungen der registerindirekten Adressierung mit Displacement liegen bei Zugriffen auf Datenbereiche mit zur Übersetzungszeit des Programms be-

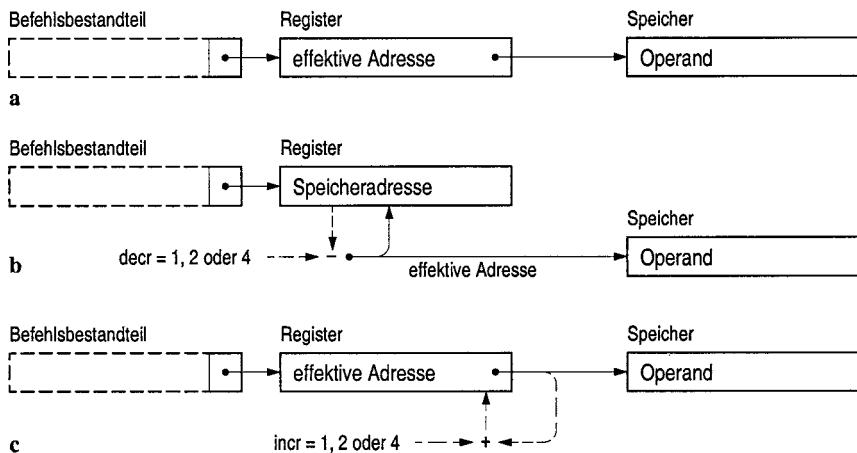


Bild 2-8. Adressierungsarten. a Registerindirekt, b mit Prädekrement, c mit Postinkrement

kannter Struktur, d.h. mit bekannten Displacements für die Datenelemente. Durch Ändern der Basisadresse sind solche Bereiche in Hauptspeicher dynamisch verschiebbar (siehe auch 3.1.3).

Beispiel 2.2. ► Registerindirekte Adressierung mit Displacement. Im folgenden Programmausschnitt wird das Statusregister eines Ein-/Ausgabebausteins gelesen, dessen Registersatz die Basisadresse IOBASE hat und dessen Register Byteformat haben. Für die Displacements wird Halbwortformat angenommen, das der Prozessor bei der Adreßrechnung durch Sign-Extension erweitert.

```

IOBASE EQU.W 0xFFFF0000
DATA    EQU.H 0
CNTRL   EQU.H 1
STATUS  EQU.H 2
:
MOVE.W #IOBASE,R0
:
MOVE.B STATUS(R0),R2
:

```

Handelt es sich bei dem Register um den Befehlszähler PC, so bezeichnet man die Adressierungsart als **befehlszählerrelative Adressierung** (PC relative, ohne Bild). Die effektive Adresse wird hierbei jeweils relativ zum aktuellen Befehlszählerstand gebildet. Die Hauptanwendung dieser Adressierungsart liegt in der Angabe von Sprungzielen bei Sprungbefehlen. Die Auswertung des Displacements als 2-Komplement-Zahl erlaubt Vorwärts- und Rückwärtsadreßbezüge, bezogen auf die Adresse des Sprungbefehls. Abhängig vom Format des Displacements (8, 16 oder 32 Bits) kann der Sprungbereich dabei eingeschränkt sein. In der Assemblerschreibweise steht das Displacement als symbolische oder numerische Angabe vor dem in runden Klammern angegebenen Befehlszähler PC; bei Sprungbefehlen, die ausschließlich die PC-relative Adressierung zulassen, kann

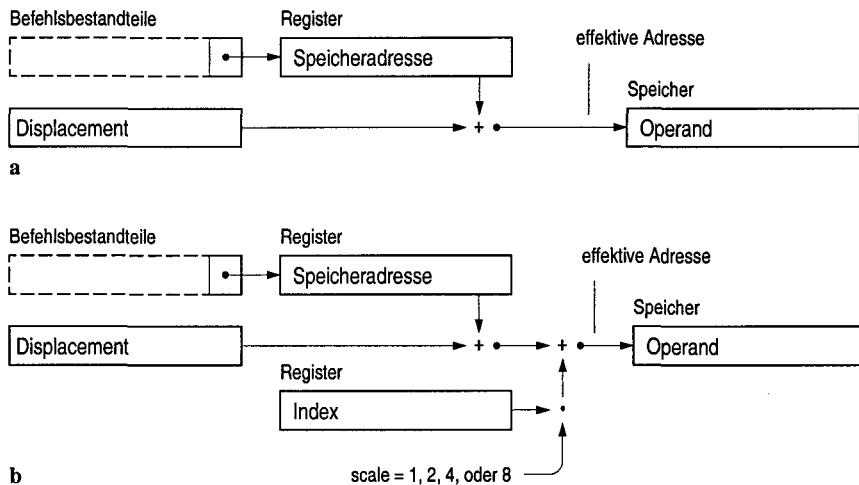


Bild 2-9. Adressierungsarten. a Relativ (registerindirekt mit Displacement), b Indizierung

auf das explizite Hinschreiben von PC als Registeradresse verzichtet werden. Angegeben wird dann die symbolische Zieladresse, hier mit label bezeichnet.

Assemblerschreibweise: displ(PC) oder label

Beispiel 2.3. ► Befehlszählerrelative Adressierung. Die folgende Programmdarstellung zeigt links einen bedingten Sprungbefehl BEQ mit der symbolischen Zieladresse LOOP. Hierbei wird vom Assembler das Displacement als Differenz der absoluten Programmadresse des auf BEQ folgenden Befehls und der absoluten Adresse der mit LOOP bezeichneten Einsprungstelle ermittelt. Die Darstellung rechts zeigt denselben Sprungbefehl, jedoch mit numerischer Vorgabe der Sprungdistanz in zwei dafür gebräuchlichen Syntaxschreibweisen.

<pre>: LOOP: CMP.B #0x80, EASTATUS BEQ LOOP :</pre>	<pre>: CMP.B #0x80, EASTATUS BEQ -10 (PC) : CMP.B #0x80, EASTATUS BEQ *-10 :</pre>
--	--

Stellt man in einem Programm sämtliche Adressbezüge innerhalb des Programm-codes durch die befehlszählerrelative Adressierung her, also auch die Zugriffe auf im Programmreich vorhandene Konstanten, so ist das Programm im Hauptspeicher verschiebbar, ohne daß dabei die Adressbezüge zuvor geändert werden müssen (*dynamisch verschiebbarer Programmcode*, siehe 3.1.3). – Anmerkung: Entsprechend der üblichen Vorgabe, daß auf den Programmcode und seine Konstanten nur lesend zugegriffen werden darf, d.h. der Code zur Programmlaufzeit nicht modifiziert werden darf, erlauben z.B. die Prozessoren der MC680x0-Familie nur PC-relative Lesezugriffe. Bei einem Schreibversuch mit PC-relativer Adressierung wird der Illegal-Instruction-Trap ausgelöst (2.1.5).

Indizierte Adressierung (indexed). Hierbei wird zu einer Speicheradresse (Feldanfangsadresse) eine variable Abstandsgröße, ein *Index*, als 2-Komplement-Zahl addiert. Der Index steht in einem der allgemeinen Prozessorregister, das damit die Funktion eines *Indexregisters* übernimmt. Ein Skalierungsfaktor erlaubt es, den Index mit einem Wert, 1, 2, 4 oder 8 zu multiplizieren. Dies erleichtert Zugriffe auf Byte-, Halbwort-, Wort- und Doppelwortstrukturen. Die Speicheradresse steht entweder als absolute Adresse im Befehl oder als variable Adresse in einem Register (registerindirekte Adressierung). Weder der Inhalt des Indexregisters noch der Inhalt des Registers, das die Speicheradresse enthält, werden durch die Adreßrechnung des Prozessors verändert.

Wir wählen zur Demonstration dieser Art der Adreßrechnung als erweiterte Form die registerindirekte Adressierung mit zusätzlichem Displacement, wobei entweder der Registerinhalt oder das 32-Bit-Displacement die Funktion einer Feldanfangsadresse haben kann (Bild 2-9b). Diese Adressierungsart eignet sich u.a. für Zugriffe auf zweidimensionale Datenfelder, indem z.B. mit der im Register stehenden Speicheradresse das Datenfeld, mit dem Displacement ein Unterfeld und mit dem Index ein Datenelement des Unterfeldes adressiert wird. (Weitere, komplexere Adressierungsarten mit Indizierung werden im Zusammenhang mit der speicherindirekten Adressierung behandelt.) In der Assemblerschreibweise sehen wir für die Indizierung wieder die runde Klammer vor, bezeichnen aber das Indexregister allgemein mit Xn.

Assemblerschreibweise: $\text{displ}(\text{Rn})(\text{Xn.scale})$

Die Indizierung ist ggf. auch in der Form **befehlszählerrelative Adressierung mit Indizierung** verfügbar ($\text{displ}(\text{PC})(\text{Xn.scale})$ oder kurz $\text{disp}(\text{Xn.scale})$). Sie erlaubt es z.B., *indirekte Sprünge* mit Anwahl eines beliebigen Sprungbefehls in einer Sprungbefehlstabelle auszuführen. Das Displacement in der Zieladresse des auslösenden Sprungbefehls zeigt dabei auf den Tabellenanfang, der variable Index auf den auszuführenden Sprungbefehl in der Tabelle (Bild 2-10a). Die Skalierung des Index trägt den möglichen (einheitlichen) Sprungbefehlsformaten in der Tabelle Rechnung. Die Indizierung kann darüber hinaus auch für Zugriffe auf Felder mit Konstanten eingesetzt werden, die dem Programmcode zugeordnet sind (Bild 2-10b).

Speicherindirekte Adressierung (memory indirect). Hier steht die eigentliche Adresse im Speicher und zeigt auf einen Operanden, der ebenfalls im Speicher steht. Für die Adressierung dieser Adresse ist die relative, d.h. die registerindirekte Adressierung mit Displacement gebräuchlich. Ergänzt wird diese Adressierungsart z.B. durch ein zweites Displacement, das nach dem Adreßzugriff in die Adreßrechnung mit einbezogen wird (Bild 2-11a). Um die speicherindirekte Adressierung von der registerindirekten zu unterscheiden, verwenden wir in der Assemblerschreibweise eckige Klammern für die Funktion „Inhalt von Speicher“.

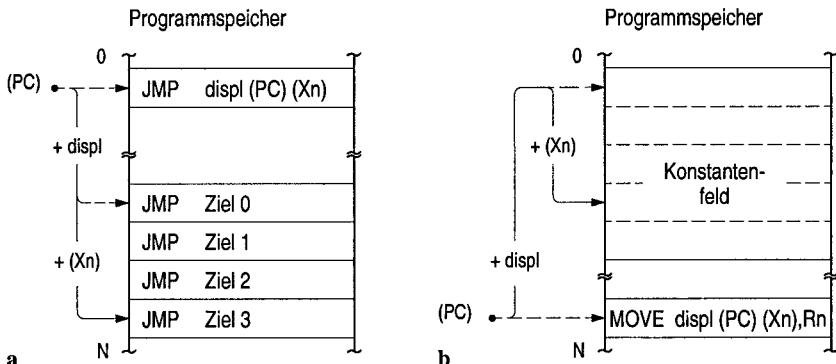


Bild 2-10. Befehlszählerrelative Adressierung mit Indizierung. **a** Anwahl eines Sprungbefehls in einer Sprungbefehlstabelle, **b** Zugriff auf ein Konstantenfeld (mit negativem Displacement)

Assemblerschreibweise: [displ1(Rn)]displ2

Die speicherindirekte Adressierung findet vorwiegend bei der Parameterübergabe bei Unterprogrammen, aber auch bei Zugriffen auf Elemente höherer Datenstrukturen, wie verkettete Listen, Verwendung. Unterstützt wird sie durch zusätzliche Indizierung, die als **Nachindizierung** nach dem Adresszugriff im Speicher, oder als **Vorindizierung** vor diesem Zugriff wirksam wird (Bilder 2-11b und c).

Assemblerschreibweise (Nachindizierung): [displ1(Rn)](Xn.scale)displ2

Assemblerschreibweise (Vorindizierung): [displ1(Rn)(Xn.scale)]displ2

Befehlsformate. 32-Bit-CISC-Prozessoren haben, wie auch ihre 16-Bit-Vorgänger, Befehlsformate unterschiedlicher Längen, die sich je nach Prozessor entweder in Vielfachen von 16-Bit-Halbwörtern (bei der MC680x0-Familie als „Wörter“ bezeichnet) oder in Vielfachen von Bytes zusammensetzen. Die Länge eines Befehls hängt dabei im wesentlichen von der Anzahl der Adressen (ein-, zweistellige Operation) und von den im Befehl verwendeten Adressierungsarten ab. Bild 2-12 zeigt dazu als Beispiel einen Register-Speicher-Befehl, in Teilbild a in Anlehnung an die MC680x0-Familie in „wort“orientierter Darstellung, in Teilbild b in Anlehnung an den Pentium in byteorientierter Darstellung.

Das erste Befehlswort in Teilbild a (operation word) umfaßt ein opcode-Feld für den Operationscode, ein opmode-Feld für das Datenformat und die Transportrichtung Register-Speicher oder Speicher-Register, ein Rn-Feld für die Registeradresse des einen Operanden und ein mode/Ri-Feld zur Adressierung des anderen Operanden. mode bezeichnet die Adressierungsart, Ri das zur Adressmodifikation benötigte Register. Wird für die gewünschte Adressierungsart kein Register benötigt, z. B. bei der Direktoperand- oder der Speicher-Adressierung, so steht das gesamte mode/Ri-Feld zu ihrer Codierung zur Verfügung. Wird zur Spezifizierung der Adressierungsart mehr Information benötigt, z. B. eine zusätzliche Indexregisteradresse und Skalierungsangaben oder Angaben zur speicherindirekten Adres-

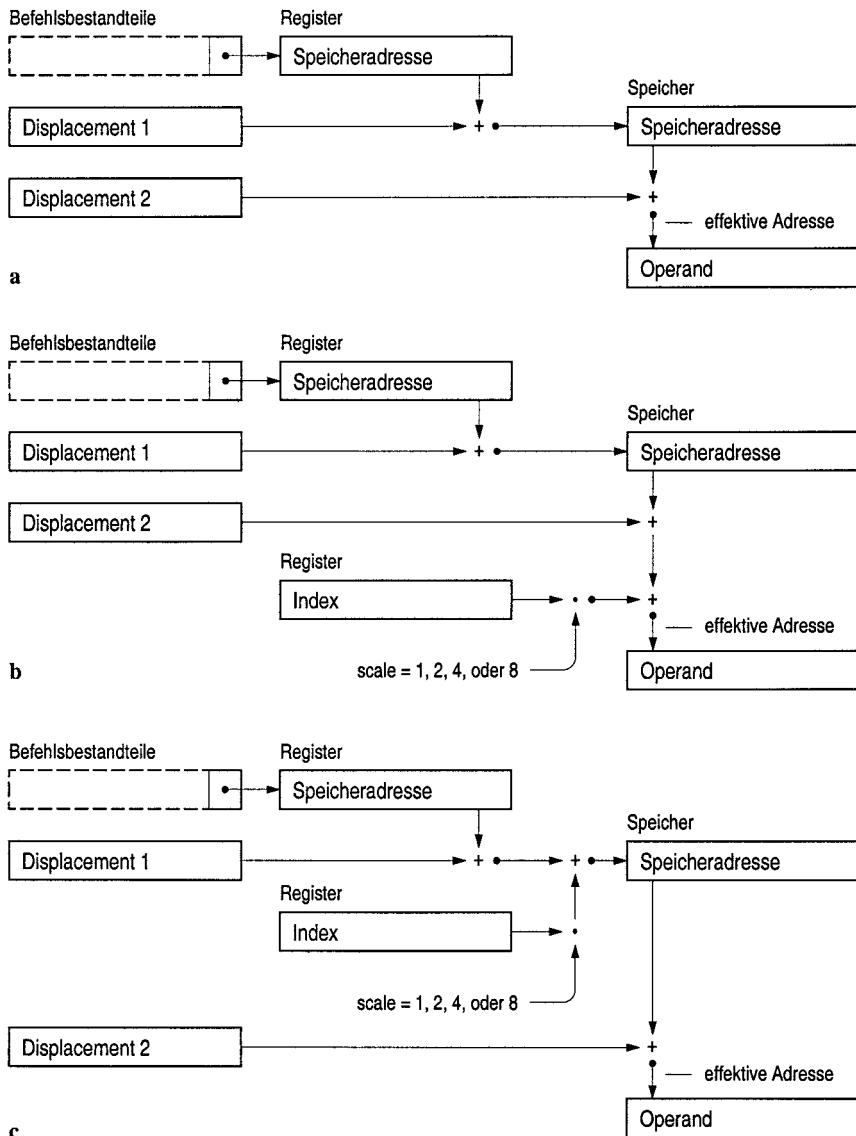


Bild 2-11. Adressierungsarten. a Speicherindirekte Adressierung, b Nachindizierung, c Vorindizierung

sierung, so wird das Befehlswort um ein oder zwei Erweiterungs„wörter“ ergänzt (extension words). Als letztes werden Direktoperanden, Speicheradressen und Displacements im 16- oder 32-Bit-Format an diese Befehlwörter angefügt.

Das erste Befehlsbyte in Teilbild b (opcode byte) enthält den Operationscode (ggf. kann dieser auch zwei Bytes umfassen). Das zweite, sog. ModR/M-Byte

spezifiziert die Adressierungsart, wobei im Rn-Feld die Registeradresse des einen Operanden und in den Feldern mod und r/m (register/memory) die Adressierungsart des anderen Operanden angegeben ist. Mit mod und r/m sind insgesamt 24 Adressierungsarten und 8 Operandenregisteradressen codierbar. Benötigt man für die Adressierungsart ein Indexregister oder ein Basisregister, so wird ein weiteres Adressierungsbyte mit den beiden Registeradressen und mit Skalierungsan-gaben angehängt. Daran werden wiederum die evtl. benötigten Direktoperanden, Speicheradressen und Displacements im 8-, 16- oder 32-Bit-Format angefügt.

ADD Rn, <ea>

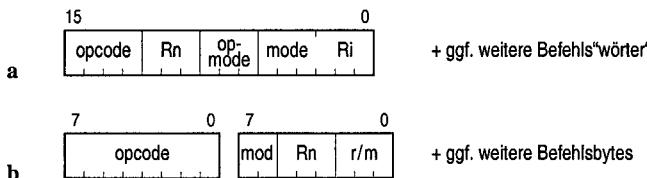


Bild 2-12. Befehlsformate am Beispiel eines Register-Speicher-Befehls; <ea> steht für eine von mehreren möglichen Adressierungsarten. **a** 16-Bit-orientiert (in Anlehnung an die MC680x0-Prozessorfamilie, **b** byteorientiert (in Anlehnung an den Pentium)

2.1.4 Befehlssatz

Befehlssätze von CISC-Prozessoren sind meist sehr umfangreich und umfassen neben elementaren Befehlen, die, wie z.B. die Ganzzahladdition, innerhalb eines Maschinentakts ausführbar sind, auch komplexe Befehle zur Unterstützung spezieller Anwendungen, deren Ausführungszeiten jedoch (meist) mehr als einen Maschinentakt erfordern. Ein wichtiges Merkmal für die Festlegung eines Befehlssatzes, ausgehend von den bei 32-Bit-CISC-Prozessoren üblichen Zweia-adréßformaten, ist idealerweise eine größtmögliche *Orthogonalität*. So sollten z.B. alle Adressierungsarten, soweit sinnvoll, sowohl für die Quell- als auch für die Zieladressierung zugelassen sein. Des weiteren sollten alle Operationen auf einen bestimmten Datentyp auch für alle hierfür definierten Datenformate ausgelegt sein. Diese Forderungen stehen teilweise im Widerspruch zu einer optimalen Befehlscodierung. Insofern stellen Befehlssätze immer einen Kompromiß zwischen den Anforderungen der Anwendungen und dem technisch vertretbaren Aufwand dar.

Befehlssätze werden in Gruppen unterteilt, in denen Befehle mit ähnlichen Funktionen zusammengefaßt sind. Typische Befehlsgruppen sind

- Transportbefehle,
- arithmetische Befehle,
- logische Befehle,

- Shiftbefehle,
- bit- und bitfeldverarbeitende Befehle,
- String- und Array-Befehle,
- Sprungbefehle,
- Systembefehle.

Im folgenden beschreiben wir exemplarisch die gebräuchlichsten Befehle dieser Befehlsguppen, wobei wir uns wieder an die MC680x0-Prozessoren anlehnen. Die im Text durch Mnemone hervorgehobenen Befehle einer jeden Gruppe sind in den Tabellen 2-1 bis 2-3 übersichtlich zusammengefaßt. Darüber hinaus gibt es im Text weitere Befehlsbeschreibungen ohne Angabe von Mnemonen. Zur Illustration ihrer Funktionsweise sind für einige Befehle textbegleitende Beispiele angegeben. Bezüglich der Anwendungen, die sich aus dem Zusammenwirken mehrerer Befehle ergeben, sei jedoch auf die Beispiele zu den Programmierungs-techniken in Kapitel 3 verwiesen.

In der Tabellendarstellung wird von größtmöglicher Orthogonalität ausgegangen und auf die sonst üblichen befehlsspezifischen Angaben der erlaubten Adressierungsarten und Datenformate verzichtet, jedoch ggf. im Text darauf hingewiesen. Eine grundsätzliche Einschränkung sei vorweggenommen: *Direktoberanden* können als im Befehl stehende Konstanten nur Quelle (und nicht Ziel) eines Zugriffs sein. In der formalen Beschreibung der Befehle verwenden wir den Zuweisungsoperator →, die Operatorzeichen der Grundrechenarten +, - , · , /, die logischen Operatoren and, or und not sowie das Operatorzeichen ⇒, das die Beeinflussung von Bedingungsbits bei Vergleichsoperationen anzeigt. Spitze Klammern bezeichnen bestimmte Bitpositionen eines Datenformats, so z. B. bei SR<7-0> die Bits 0 bis 7 des Statusregisters. – Auf die Zuordnung der im Text beschriebenen Befehle zu den Tabellen 2-1 bis 2-3 wird nicht gesondert hingewiesen.

Transportbefehle. MOVE src,dst führt den allgemeinen Datentransport zwischen einer Quelle src und einem Ziel dst durch, wobei die Quelle und das Ziel sowohl im Registerspeicher als auch im Hauptspeicher liegen können. In einer Variante als Befehl Move-Multiple-Registers kann als Quelle oder Ziel mehrerer

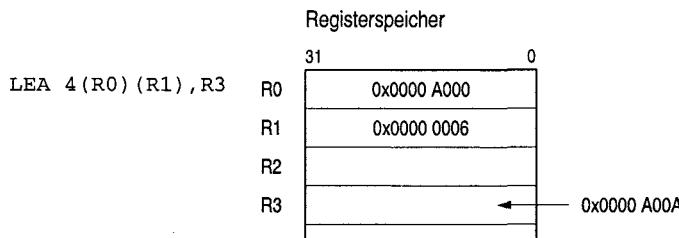


Bild 2-13. Wirkungsweise des Befehls LEA

Datentransporte eine Registerliste angegeben werden, was für das Retten und Wiederherstellen von Registerinhalten bei Unterprogrammanschlüssen nützlich ist. LEA src,dst errechnet die durch die Quellangabe src spezifizierte effektive Adresse und schreibt sie an den Zielort dst. Die so ermittelte Adresse kann z.B. zur registerindirekten Adressierung von Operanden (Register als Zielort des Adreßtransports) oder zur Parameterübergabe bei Unterprogrammanschlüssen (Parameterfeld als Zielort des Adreßtransports, z.B. im Stack) verwendet werden. Bild 2-13 zeigt ein Beispiel zu LEA, bei dem die effektive Adresse durch indizierte Adressierung vorgegeben wird.

Tabelle 2-1. Transport-, arithmetische, logische, Shift- und bitverarbeitende Befehle (CISC-Befehlssatz)

Befehl	Bezeichnung	Funktion
MOVE src,dst	move	$\text{src} \rightarrow \text{dst}$
LEA src,dst	load effective address	effektive Adresse von src \rightarrow dst
PUSH src	push on stack	$\text{SP} - n \rightarrow \text{SP}; \text{src} \rightarrow (\text{SP})$
POP dst	pop from stack	$(\text{SP}) \rightarrow \text{dst}; \text{SP} + n \rightarrow \text{SP}$
ADD src,dst	add	$\text{dst} + \text{src} \rightarrow \text{dst}$
SUB src,dst	subtract	$\text{dst} - \text{src} \rightarrow \text{dst}$
MUL src,dst	multiply	.H: $\text{dst}_H \cdot \text{src}_H \rightarrow \text{dst}_W$.W: $\text{dst}_W \cdot \text{src}_W \rightarrow \text{dst}_{WW}$
DIV src,dst	divide	.H: $\text{dst}_W / \text{src}_H \rightarrow \text{dst}_W < r_H, q_H >$.W: $\text{dst}_{WW} / \text{src}_W \rightarrow \text{dst}_{WW} < r_W, q_W >$ mit r: Rest, q: Quotient
CMP src,dst	compare	$\text{dst} - \text{src} \Rightarrow \text{CC}$
TEST src	test	$\text{src} \Rightarrow \text{CC}$
TAS dst	test operand and set sign	$\text{dst} - 0 \Rightarrow \text{CC}; 1 \rightarrow \text{dst} < \text{MSB} >$
AND src,dst	and	$\text{src} \text{ and } \text{dst} \rightarrow \text{dst}$
OR src,dst	or	$\text{src} \text{ or } \text{dst} \rightarrow \text{dst}$
NOT dst	not	$\text{not } \text{dst} \rightarrow \text{dst}$
ASL src,dst	arithmetic shift left	$\text{dst} \cdot 2^{\text{src}} \rightarrow \text{dst}$
ASR src,dst	arithmetic shift right	$\text{dst} \cdot 2^{-\text{src}} \rightarrow \text{dst}$
BTST src,dst	test bit	$\text{not } \text{dst} < \text{src} > \Rightarrow Z$
BSET src,dst	test and set bit	$\text{not } \text{dst} < \text{src} > \Rightarrow Z; 1 \rightarrow \text{dst} < \text{src} >$
BCLR src,dst	test and clear bit	$\text{not } \text{dst} < \text{src} > \Rightarrow Z; 0 \rightarrow \text{dst} < \text{src} >$

PUSH src führt einen Schreibzugriff und POP dst einen Lesezugriff auf den User- oder den Supervisor-Stack aus. Als Stackpointerregister SP wird im User-Modus USP und im Supervisor-Modus SSP benutzt. Beide Stacks werden mit abwärtszählenden Adressen gefüllt. Der *Stackpointer* SP zeigt dabei jeweils auf den letzten Eintrag (siehe Bild 2-2, S. 70). Bei Prozessoren mit Postinkrement- und Prädekrement-Adressierung wird anstelle der Befehle PUSH und POP der Befehl MOVE verwendet (z.B. bei den MC680x0-Prozessoren/Mikrocontrollern). Dies hat den Vorteil, ein beliebiges Register (SP oder Rn) als Stackpointerregister angeben zu können. Je nach Adressierung können die Stacks mit abwärts- oder aufwärtszählenden Adressen gefüllt werden (Bild 2-14). Im ersten Fall zeigt der Stackpointer auf den letzten Eintrag (dies entspricht der Wirkung von PUSH und POP), im zweiten Fall zeigt er auf die erste freie Zelle.

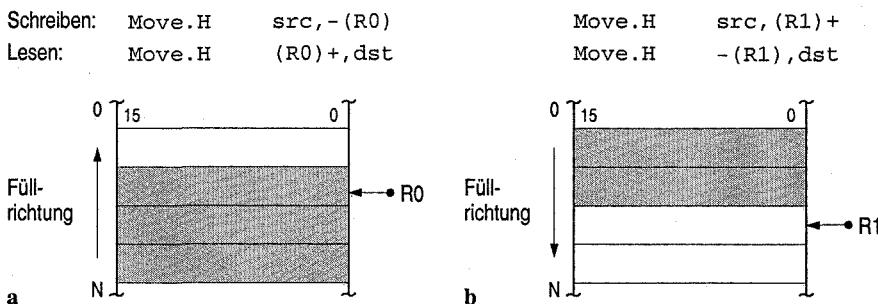


Bild 2-14. Aufbau von Stacks mit unterschiedlichen Füllrichtungen, a bei (üblicher) absteigender, b bei aufsteigender Adressierung

Arithmetische Befehle. ADD src,dst und SUB src,dst führen die Addition bzw. Subtraktion sowohl für vorzeichenlose Dualzahlen als auch für 2-Komplement-Zahlen, d.h. für ganze Zahlen aus, wobei Bereichsüberschreitungen im ersten Fall durch das *Carrybit* C und im zweiten Fall durch das *Overflowbit* V angezeigt werden. Das Carrybit dient darüber hinaus in beiden Fällen als Übertragsstelle, z.B. zur Programmierung von Operationen mit mehrfacher Wortlänge. Dafür stehen meist noch besondere ADD- und SUB-Befehle zur Verfügung, die das C-Bit in die jeweilige Operation mit einbeziehen.

MUL src,dst und DIV src,dst führen die Multiplikation bzw. Division für zwei Operanden durch. Die Multiplikation führt bei einfacher Operandenlänge der Faktoren (wahlweise Halbwort oder Wort) üblicherweise auf ein Produkt doppelter Länge (Wort bzw. Doppelwort). Bei der Division hat der Dividend üblicherweise doppelte Länge (wahlweise Wort oder Doppelwort), Divisor, Quotient und Rest haben einfache Länge (Halbwort bzw. Wort). Ein Divisor mit dem Wert Null führt unmittelbar zum Befehlsabbruch (siehe 2.1.5, Zero-Divide-Trap). Als Zielort der Operationen ist meist der Registerspeicher vorgegeben. Anders als bei ADD und SUB sind für MUL und DIV jeweils eigene Befehle für die beiden Zahlendarstellungen erforderlich (hier nicht in der Befehlstabelle angegeben), die

sich in der Mnemonik durch die Zusätze U (unsigned, MULU und DIVU) und S (signed, MULS und DIVS) unterscheiden. Bild 2-15 zeigt ein Beispiel zur Division.

DIV.W MEM, R2

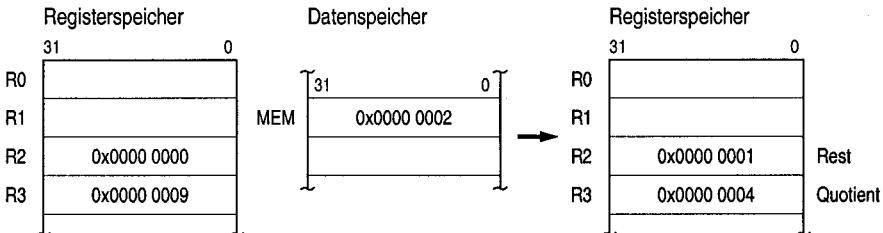


Bild 2-15. Wirkungsweise des Befehls DIV (unsigned). Der Dividend steht als Doppelwort im Registerspeicher

CMP src,dst und TEST src sind *Vergleichsbefehle*, deren Resultate sich in den Bedingungsbits niederschlagen, die wiederum durch einen nachfolgenden bedingten Sprungbefehl für Programmverzweigungen ausgewertet werden. CMP vergleicht zwei Operanden dst und src in den möglichen Relationen dst > src, dst \geq src, dst \leq src, dst < src, dst = src und dst \neq src (siehe dazu Beispiel 2.4, S. 94). TEST vergleicht einen Operanden src mit Null in den Aussagen src = 0, src \neq 0, src > 0 oder src < 0. TAS dst ist ein erweiterter Vergleichsbefehl, der für die Realisierung von *Semaphoren* ausgelegt ist (Semaphorbefehl). Er liest eine Variable dst, beeinflußt damit die Bedingungsbits N und Z (N = 0/1: Betriebsmittel ist frei/belegt), setzt anschließend das höchstwertige Variablenbit auf 1 (Betriebsmittel wird vorsorglich belegt) und schreibt die Variable an ihren Speicherplatz zurück (siehe dazu auch 8.1.4: Beispiel 8.2, S. 558).

Anmerkung. Semaphorbefehle werden bei Systemen benötigt, bei denen mehrere Prozessoren über einen globalen Bus auf *gemeinsame Betriebsmittel*, wie Speicherbereiche und Ein-/Ausgabe-einheiten zugreifen. Hierbei ist es erforderlich, die Zugriffe auf diese Betriebsmittel zu synchronisieren. Dazu muß ein Prozessor, der ein solches Betriebsmittel benötigt, überprüfen, ob dieses frei ist, und er muß, wenn er es belegt, diesen Zustand den anderen Prozessoren anzeigen. Dies geschieht über ein *Semaphor* („Flügelsignal“, Begriff aus der Eisenbahntechnik), das als binäre Variable dem Betriebsmittel zugeordnet ist und von der Software verwaltet wird [Tanenbaum 2002]. Für das Abfragen und Verändern des Semaphors benötigt man einen Befehl, der einen Doppelbuszyklus, einen sog. *Read-modify-write-Cycle* ausführt, der von der Busanforderung eines anderen Prozessors nicht unterbrochen werden kann (zur Busarbitrierung in Mehrprozessor-/Mehrmaster-systemen siehe 5.4). TAS ist ein solcher Semaphorbefehl, eine andere Realisierung eines Semaphorbefehls ist in 2.2.4 als Swap-Befehl des SPARC beschrieben.

Gleitpunktbefehle. Die Gleitpunktarithmetik wird von einer Gleitpunkteinheit (früher ein Off-chip- oder On-chip-Coprozessor, heute eine von mehreren parallel arbeitenden On-chip-Funktionseinheiten) ausgeführt, deren Befehlssatz neben den vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division Befehle für den Vergleich, für die Berechnung von Quadratwurzeln und zur Er-

mittlung trigonometrischer Funktionswerte umfaßt. Die Operandendarstellung erfolgt wahlweise mit einfacher oder doppelter Länge (32 bzw. 64 Bit, siehe 1.1.5). Hinzu kommen Befehle für das Runden von Gleitpunktzahlen sowie für das Konvertieren zwischen den verschiedenen Zahlendarstellungen (ganze Zahlen, BCD-Zahlen) unter Berücksichtigung der dabei wählbaren Datenformate. Ist keine solche Gleitpunkteinheit als Hardwareunterstützung vorhanden, so werden diese Befehle durch Software nachgebildet (siehe auch 2.1.5, OPC-Emulation-Traps).

BCD-Befehle. Spezielle Befehle erlauben die Addition bzw. Subtraktion von binärcodierten Dezimalzahlen in *gepackter Darstellung*. Die Operanden umfassen abhängig vom Datenformat 2, 4 oder 8 BCD-Ziffern. Das Einbeziehen des Carrybits in die Addition bzw. Subtraktion erleichtert das Programmieren von Operationen bei mehr als acht Dezimalstellen. Zwei weitere Befehle wandeln einen Operanden von der ungepackten in die gepackte Zifferndarstellung, bzw. kehren diesen Vorgang unter Vorgabe des Zonenteils um (siehe 1.1.6).

Logische Befehle. AND src,dst und OR src,dst bilden mit den korrespondierenden Bits ihrer Operanden die logischen Verknüpfungen and und or. NOT dst invertiert die Bits eines Operanden (logische Operation not, Bilden des 1-Komplements). Bild 2-16 zeigt dazu zwei Beispiele mit den Befehlen AND.B und OR.H.

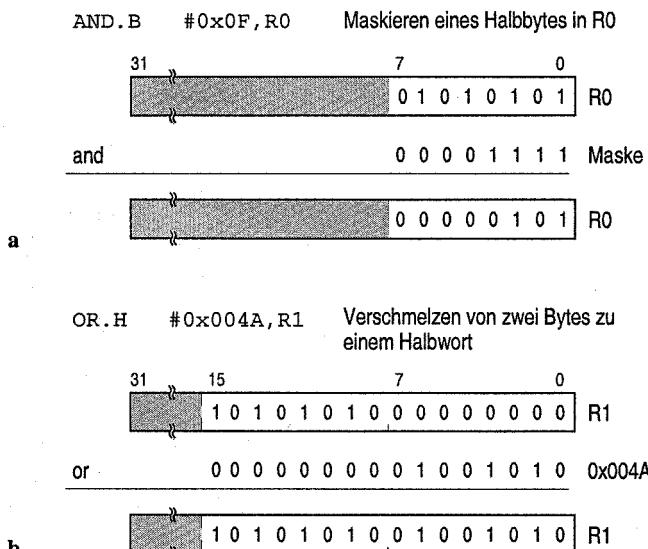


Bild 2-16. Wirkungsweise der Befehle a AND und b OR

Shiftbefehle. ASL src,dst und ASR src,dst erlauben das Verschieben der Bits eines Operanden dst um die durch src spezifizierte Anzahl n an Bitpositionen nach links (L) bzw. nach rechts (R). Das A in der Befehlsmnemonik deutet auf

arithmetische Operationen hin, und zwar mit 2-Komplement-Darstellung des zu verschiebenden Operanden. So entspricht das Linksschieben um n Stellen einer Multiplikation mit 2^n , wozu in die niedrigstwertige Bitposition n Nullen nachgezogen werden, und das Rechtsschieben einer Division durch 2^n , wozu das Vorzeichenbit des Operanden n -mal kopiert wird. Von dem Schiebevorgang sind immer nur die Bits innerhalb des angegebenen Datenformats betroffen, so auch, wenn der Operand als Halbwort oder Byte in einem der 32-Bit-Prozessorregister steht. Das jeweils zuletzt hinausgeschobene Bit wird als Bedingungsbit C im Statusregister gespeichert. Des weiteren wird auch das Bedingungsbit V vom Ergebnis der Operation beeinflußt. Bild 2-17a zeigt dazu ein Beispiel mit dem Befehl ASL.B. V wird bei dieser Befehlausführung gesetzt, da sich das höchstwertige Operandenbit während der Schiebeoperation verändert (Bereichsüberschreitung).

Weitere, in der Befehlstabelle nicht angegebene Shiftbefehle sind die logischen und die Rotationsshifts. Erstere arbeiten wie die arithmetischen Shiftbefehle, jedoch werden bei beiden Schiebeberichtigungen Nullen nachgezogen. Bei den Rotationsshifts hingegen wird bei jedem Schritt dasjenige Bit nachgezogen, das am anderen Ende hinausgeschoben wurde, so daß kein Bit verlorengeht.

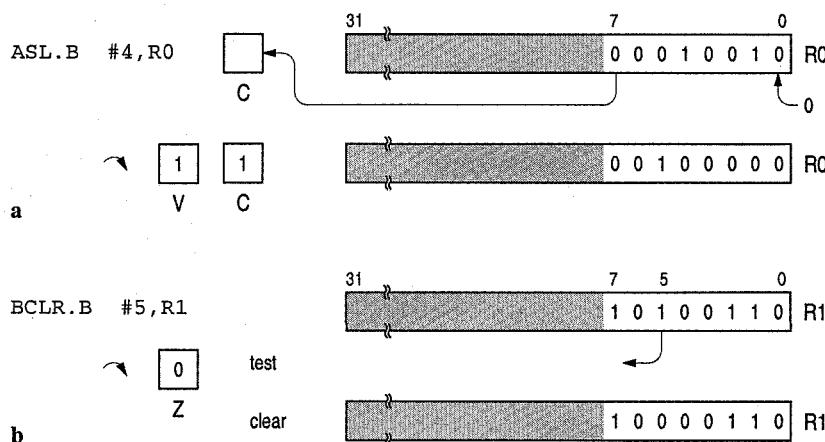


Bild 2-17. Wirkungsweise der Befehle a ASL und b BCLR

Bit- und bitfeldverarbeitende Befehle. Bei den bitverarbeitenden Befehlen wird in dem durch dst adressierten Operanden das durch src spezifizierte Bit angewählt. BTST src,dst vergleicht dieses Bit mit Null und beeinflußt entsprechend dem Vergleichsergebnis das Bedingungsbit Z. BSET src,dst und BCLR src,dst setzen darüber hinaus dieses Bit nach dem Test auf 1 bzw. 0. Bild 2-17b zeigt die Wirkung von BCLR.B an einem Beispiel.

Bitfeldverarbeitende Befehle arbeiten auf *Bitfelder* variabler Bitanzahl (z.B. von 1 bis zu 32 Bits, siehe Bild 2-4, S. 74) und beliebiger Position des Feldanfangs, also unabhängig von den Bytegrenzen im Speicher. Die wichtigsten Befehle be-

wirken das Extrahieren eines solchen Feldes im Speicher und dessen Transport in eines der Prozessorregister bzw. das Rückschreiben eines solchen Feldes in den Speicher. Beim Transport in das Register ist als Option angebar, ob das Feld, das dort rechtsbündig abgelegt wird, in den zur Registerlänge fehlenden höheren Bitpositionen mit Nullen (*zero extension*) oder mit dem Vorzeichenbit (*sign extension*) ergänzt werden soll (*extract bit field unsigned* bzw. *signed*). Auf diese Weise können Bitfelder mit den herkömmlichen arithmetischen und logischen Befehlen sowie in den Standard-Datenformaten bearbeitet werden. Spezielle Befehle erlauben das Testen, das Setzen und das Rücksetzen einzelner Bits, oder das Aufsuchen der ersten Eins in einem solchen Feld.

String- und Array-Befehle. String-Befehle verarbeiten im Speicher stehende Byte-, Halbwort- und Wortfolgen. Die Adressierung und Zählung der einzelnen String-Elemente erfolgt durch feste Zuordnung von Registern des allgemeinen Registerspeichers. Deren Inhalte werden mit jeder Elementaroperation verändert: die Adressen werden wahlweise inkrementiert oder dekrementiert, die Elementanzahl dekrementiert. Typische Operationen sind „Move-String“ für das Kopieren eines Strings 1 in einen Stringbereich 2, „Compare-String“ für den Vergleich zweier Strings bis zur Ungleichheit eines Elementpaars und „Skip-String“ für das Durchsuchen eines Strings nach einem Element, das wahlweise gleich oder ungleich einem Vergleichselement ist.

Zusätzliche Befehle führen in Erweiterung dieser Operationen vor dem Kopieren, Vergleichen bzw. Durchsuchen die Operation „Translate“ aus. Dabei wird das jeweilige Quellelement als Index einer im Speicher stehenden Umsetzungstabelle ausgewertet und der zugehörige Tabelleneintrag als Zielgröße verwendet. Die Basisadresse der Tabelle wird in einem weiteren Register vorgegeben. Die Umsetzung ist auf Byte-Strings beschränkt und dient z.B. zur Codeumsetzung. Bild 2-18 zeigt die Wirkungsweise von „Move-String-and-Translate“. – String-Operationen können üblicherweise nach jeder Elementaroperation vom Interruptsystem in ihrer Ausführung unterbrochen und später wieder aufgenommen werden.

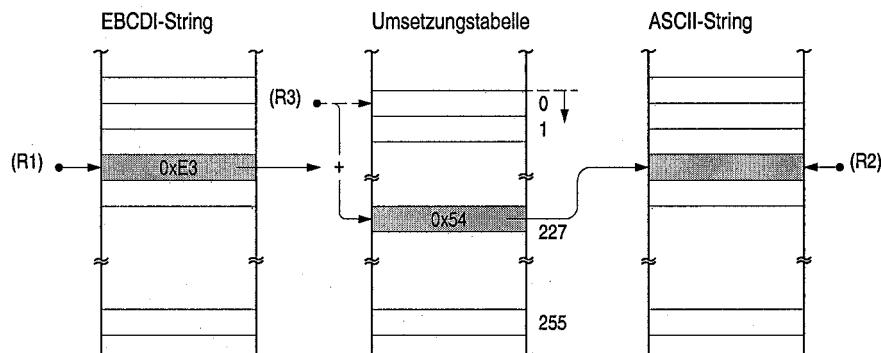


Bild 2-18. Umsetzung eines EBCDI-Strings in einen ASCII-String mittels des Befehls Move-String-and-Translate (mit Bytezählung in R0). Gezeigt ist der Zugriff auf das Zeichen T

Ein- und mehrdimensionale Felder (*arrays*) werden als höhere Datenstrukturen lediglich in den Zugriffsoperationen unterstützt. Im Gegensatz zur String-Verarbeitung mit Adressierung der Elemente durch eine variable Adresse (Zeiger, pointer) werden hier die Elemente mittels indizierter Adressierung, d.h. durch eine Basisadresse und einen variablen Index angesprochen. Array-Befehle dienen zur Bereichsüberprüfung von Indizes und zur iterativen Ermittlung des tatsächlichen Speicherindexes bei mehrdimensionaler Indizierung.

Tabelle 2-2. Sprungbefehle (CISC-Befehlssatz)

Befehl	Bezeichnung	Funktion
BRA dst	branch always	effektive Adresse von dst → PC
BSR dst	branch to subroutine	$SP - 4 \rightarrow SP; PC \rightarrow (SP);$ effektive Adresse von dst → PC
RTS	return from subroutine	$(SP) \rightarrow PC; SP + 4 \rightarrow SP$
RTE	return from exception	$(SSP) \rightarrow SR; SSP + 2 \rightarrow SSP;$ $(SSP) \rightarrow PC; SSP + 4 \rightarrow SSP$ privilegiert
Bcc dst	branch conditionally	if cc-test = true then eff. Adr. von dst → PC else nächster Befehl
Bedingungen cc: branch if		
BGT	greater than (signed) >	$Z = 0 \text{ and } N = V$
BGE	greater or equal (signed) \geq	$N = V$
BLE	less or equal (signed) \leq	$Z = 1 \text{ or } N \neq V$
BLT	less than (signed) <	$N \neq V$
BPL	plus	$N = 0$
BMI	minus	$N = 1$
BHI	higher (unsigned) >	$Z = 0 \text{ and } C = 0$
BHS = BCC	higher or same (unsigned) \geq	$C = 0$
BLS	lower or same (unsigned) \leq	$Z = 1 \text{ or } C = 1$
BLO = BCS	lower (unsigned) <	$C = 1$
BEQ	equal =	$Z = 1$
BNE	not equal \neq	$Z = 0$
BVS	overflow set	$V = 1$
BVC	overflow clear	$V = 0$
BCS	carry set	$C = 1$
BCC	carry clear	$C = 0$

Sprungbefehle. BRA dst lädt als unbedingter Sprungbefehl den Befehlszähler PC mit der im Befehl durch dst spezifizierten effektiven Adresse, wonach das Programm mit dem unter dieser Adresse gespeicherten Befehl fortgesetzt wird (*unbedingter Sprung*). Bcc dst bezeichnet eine Gruppe von bedingten Sprungbefehlen. Bei ihnen wird ein Sprung zu der im Befehl angegebenen Adresse nur dann ausgeführt, wenn die im Befehl angegebene *Sprungbedingung cc* erfüllt ist; ist sie nicht erfüllt, so wird das Programm mit dem auf Bcc folgenden Befehl fort-

gesetzt (*bedingter Sprung*). Die Sprungbedingungen beziehen sich dabei auf den Zustand der *Bedingungsbits* CC im Statusregister. *Programmverzweigungen* werden dadurch programmiert, daß ein dem bedingten Sprungbefehl vorangestellter Befehl, häufig der Vergleichsbefehl CMP, die Bedingungsbits für die Abfrage beeinflußt.

In Verbindung mit dem Vergleichsbefehl erklärt sich auch die Mnemonik für die Sprungbedingungen (Tabelle 2-2). Die Befehle BGT, BGE, BLE und BLT sind für Vergleiche mit 2-Komplement-Zahlen vorgesehen, BHI, BHS, BLS und BLO für Vergleiche mit vorzeichenlosen Dualzahlen. BPL und BMI beziehen sich auf das höchstwertige Resultatbit (Vorzeichenbit bei 2-Komplement-Darstellung), BEQ und BNE auf den Resultatwert Null bzw. ungleich Null. BVC, BVS, BCC und BCS dienen zur Abfrage des Overflow- bzw. des Carrybits.

Beispiel 2.4. ► Auswerten der Bedingungsbits durch bedingte Sprungbefehle. Im folgenden Programmausschnitt werden die beiden Operanden dst und src von zwei verschiedenen Programmverzweigungen zueinander in Relation gesetzt: im Fall a werden sie als 2-Komplement-Zahlen interpretiert, im Fall b als vorzeichenlose Dualzahlen.

dst: 00000011, src: 10000101	
CMP.B src,dst	
a BGT ZIEL_1	Sprungbedingung dst > src (signed) erfüllt
b BHI ZIEL_2	Sprungbedingung dst > src (unsigned) nicht erfüllt

◀

Für die Angabe der Zieladresse wird bevorzugt die *befehlszählerrelative Adressierung* mit Displacement verwendet (2.1.3), mit dem Vorteil, daß ein Programm dadurch im Speicher *dynamisch verschiebbar* wird (siehe 3.1.3). Deshalb sehen die Sprungbefehle mit der Bezeichnung „Branch“ (BRA, Bcc und BSR) üblicherweise ausschließlich diese Adressierungsart vor; für den Einsatz anderer Adressierungsarten stehen hingegen Befehle mit der Bezeichnung „Jump“ zur Verfügung (z.B. JMP, JSR). Das *Displacement* (*Sprungdistanz*) wird vom Assembler als 2-Komplement-Zahl entweder mit 8 Bit (Sprungbereich: -128 bis +127 Bytes), mit 16 Bits (Sprungbereich: -32768 bis +32767 Bytes) oder mit 32 Bit (Sprungbereich: -2^{31} bis $+2^{31}-1$) codiert. Dementsprechend ergeben sich Befehle

neg. Displacement

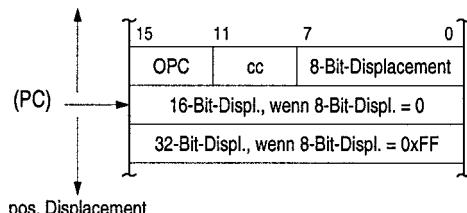


Bild 2-19. Bedingter Sprungbefehl in 16-Bit-orientierter Darstellung (in Anlehnung an den Prozessor MC68040)

unterschiedlicher Länge. Bezugspunkt für das Displacement ist z.B. bei 16-Bit-orientierter Befehlsdarstellung die um 2 erhöhte Adresse des ersten Befehls „wörtes“, wie in Bild 2-19 gezeigt.

BSR dst dient als *Unterprogrammsprung*. Wie bei BRA erfolgt der Sprung unbedingt, jedoch wird zuvor der aktuelle Befehlszählerstand (Adresse des nächsten Befehls) als *Rücksprungadresse* auf den Stack geladen. Abhängig von der Betriebsart des Prozessors, in der das Programm läuft, ist das der User- oder der Supervisor-Stack. RTS führt den Rücksprung vom Unterprogramm zu dem auf BSR folgenden Befehl durch, indem er den obersten Stackeintrag in den Befehlszähler lädt. RTS schließt dementsprechend ein Unterprogramm ab (siehe 3.1.3). Bild 2-20 zeigt den Aufruf eines Unterprogramms, dessen erster Befehl die symbolische Adresse SUBR hat. Die Speicherbelegung ist zur besseren Übersicht 16-Bit-orientiert dargestellt. – Zusätzlich zu BRA und BSR gibt es häufig noch deren Gegenstücke JMP (jump) und JSR (jump to subroutine). Sie erlauben sämtliche Adressierungsarten, mit denen eine Speicheradresse gebildet werden kann.

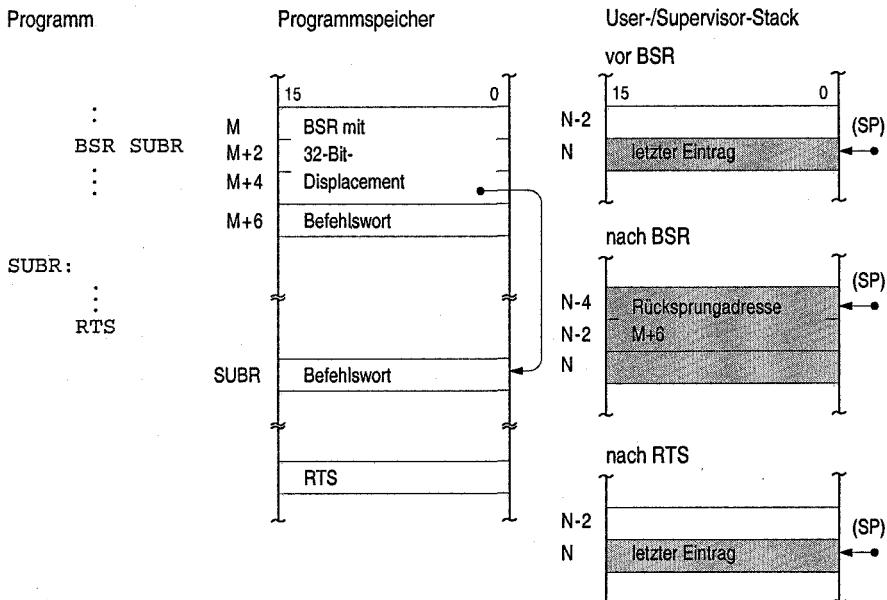


Bild 2-20. Wirkungsweise der Befehle BSR und RTS

RTE hat ähnliche Funktion wie RTS, dient jedoch zur Rückkehr aus Trap- und Interruptprogrammen. Bei diesen muß zusätzlich zum Befehlszähler PC auch das Statusregister SR geladen werden, wozu RTE zwei Stackzugriffe durchführt. Quelle dieser Zugriffe ist immer der Supervisor-Stack, da Trap- und Interruptprogramme grundsätzlich im Supervisor-Modus ausgeführt werden. Um einem im User-Modus laufenden Programm den Zugriff auf diesen Stack zu verwehren, ist RTE als *privilegierter Befehl* ausgelegt, d.h., er bewirkt bei Ausführung im User-

Modus einen Programmabbruch (privilege violation trap, 2.1.5). – *Anmerkung:* RTS und RTE führen den Rücksprung nur dann korrekt aus, wenn der Stackpointer bei der Befehlausführung den gleichen Stand wie direkt nach Eintritt in das Unterprogramm bzw. das Trap- oder Interruptprogramm aufweist.

Systembefehle. Die Systembefehle sind Befehle zur Steuerung des Systemzustands (processor control, Tabelle 2-3). Hinsichtlich ihrer Wirkungsweise gibt es *privilegierte Befehle*, die nur im Supervisor-Modus ausführbar sind, und *nichtprivilegierte Befehle*, die in beiden Betriebsarten ausführbar sind. MOVESR ermöglicht mit SR,dst den Lesezugriff auf das Statusregister, wodurch der Prozessorstatus zugänglich wird, und mit src,SR den Schreibzugriff auf das Statusregister, womit der *Processorstatus* verändert werden kann. Letzteres ist selbstverständlich nur im Supervisor-Modus zulässig, weshalb MOVESR ein privilegierter Befehl ist. Für das Lesen und Schreiben der Bedingungsbits CC des Statusregisters gibt es darüber hinaus den nichtprivilegierten Befehl MOVECC. Meist gibt es noch weitere Befehle, mit denen einzelne Bits des Statusregisters (Set- und Clear-Befehle) oder mehrere Bits gleichzeitig manipuliert werden können (And- und Or-Befehle). MOVUSP USP,dst bzw. src,USP erlaubt im Supervisor-Modus den Zugriff auf das User-Stackpointerregister USP, dessen explizite Adressierung in diesem Modus ja nicht möglich ist. Auch dieser Befehl ist privilegiert.

Tabelle 2-3. Systembefehle (CISC-Befehlssatz)

Befehl	Bezeichnung	Funktion	
MOVESR src,SR SR,dst	move status	src → SR SR → dst	privilegiert privilegiert
MOVECC src,CC CC,dst	move condition code	src → CC CC → dst	nicht privilegiert nicht privilegiert
MOVUSP src,USP USP,dst	move user stackpointer	src → USP USP → dst	privilegiert privilegiert
TRAP #n	trap	SSP – 4 → SSP; PC → (SSP); SSP – 2 → SSP; SR → (SSP); TRAP-n-Vektor → PC	
NOP	no operation		
STOP	stop	Stoppen der Programmausführung	
RESET	reset	Aktivieren des RESET-Ausgangssignals	

TRAP #n bewirkt als *Trap-Befehl* eine Programmunterbrechung und verzweigt auf ein dem Befehl zugeordnetes *Trap-Programm*, dessen Startadresse in der sog. Vektortabelle im Speicher steht. Mit dem Direktoperanden n werden dabei mehrere solcher Programme und dementsprechend Einträge in der Vektortabelle unterschieden, z.B. 16 (siehe 2.1.5). Man bezeichnet die TRAP-Unterbrechungen

auch als *Supervisor-Calls*, da sie dazu benutzt werden, vom Betriebssystem bereitgestellte Systemroutinen aufzurufen, z.B. Routinen zur Durchführung von Ein-/Ausgabevorgängen. Diese Aufrufe können sowohl vom User- als auch vom Supervisor-Modus aus erfolgen, d.h., der TRAP-Befehl ist nicht privilegiert. Vom User-Modus aus hat er die wichtige Funktion des kontrollierten Übergangs in den Supervisor-Modus. So wird z.B. einer dieser Supervisor-Calls als Abschluß für User-Programme zur *Rückkehr in das Betriebssystem* eingesetzt.

NOP führt keine Operation aus; er benötigt lediglich die Zeit für den Befehlsabruf und die Befehlsdecodierung. Mit ihm können z.B. Zeitbedingungen in Zeitschleifen vorgegeben werden, oder es können Befehle im Maschinenprogramm überschrieben und damit in ihren Wirkungen aufgehoben werden. STOP ist ein privilegierter Befehl, der den Prozessor in der Programmausführung stoppt. Sie kann nur durch eine externe Unterbrechungsanforderung wieder aufgenommen werden (Interruptsignal oder Reset-Eingangssignal, siehe 2.1.5). RESET ist ein privilegierter Befehl. Er aktiviert für einige Maschinentakte den RESET-Ausgang des Prozessors, womit Systemkomponenten, wie z.B. Interface-Einheiten, über ihre RESET-Eingänge initialisiert werden können.

2.1.5 Unterbrechungssystem und Betriebsarten

Unter *Ausnahmeverarbeitung* (*exception processing*) versteht man die Reaktion des Mikroprozessors auf Unterbrechungsanforderungen durch Traps und Interrupts. Da diese Reaktion möglichst schnell erfolgen soll, wird sie durch ein wirkungsvolles *Unterbrechungssystem* (*interrupt system*) als Teil der Prozessorhardware unterstützt. Verbunden mit der Ausnahmeverarbeitung sind die Betriebsarten des Prozessors, da Unterbrechungssituationen immer eine Umschaltung in den privilegierten Supervisor-Modus bewirken.

Traps und Interrupts. *Traps* (Fallen) sind Programmunterbrechungen, die durch Befehlsausführungen, d.h. synchron zur Prozessorverarbeitung ausgelöst werden. Sie entstehen zum einen prozessorintern, entweder bedingt durch Fehler bei der Befehlsausführung oder regulär durch Supervisor-Calls (Trap-Befehl), zum andern prozessorextern, durch die von der prozessorexternen Hardware signalisierten Fehler, so z.B. bei fehlerhaftem Buszyklus oder bei Fehlermeldungen von einer Speicher verwaltungseinheit (page fault, access violation etc.). *Interrupts* (Unterbrechungen) hingegen haben immer prozessorexterne Ursachen und erfolgen unabhängig von der Prozessorverarbeitung, d.h. asynchron dazu. Typische Ursachen sind Synchronisationsanforderungen von Ein-/Ausabeeinheiten. Traps und Interrupts gemeinsam ist die Unterbrechungsverarbeitung durch die Prozessorhardware, die wir hier betrachten. Den Signalfluß bei Interrupts beschreiben wir in Abschnitt 5.5; Beispiele zur Interruptprogrammierung folgen in Kapitel 7.

Programmunterbrechung. Eine Unterbrechungsanforderung bewirkt, sofern ihr vom Prozessor stattgegeben wird, eine Unterbrechung des laufenden Programms. Dazu führt der Prozessor eine *Ausnahmeverarbeitung* durch. Grob gesagt, rettet er dabei zunächst den gegenwärtigen *Prozessorstatus* – Befehlszähler und Statusregister – auf den Supervisor-Stack. Zusätzlich setzt er bei sog. maskierbaren Interrupts die Interruptmaske im Statusregister, um weitere maskierbare Interrupts zu blockieren. Anschließend verzweigt er zu einem der Unterbrechungsanforderung zugeordneten *Unterbrechungsprogramm*, das die eigentliche *Ausnahmebehandlung (exception handling)* durchführt. Abgeschlossen wird dieses Programm mit dem RTE-Befehl, der den ursprünglichen Prozessorstatus wieder lädt, wodurch das unterbrochene Programm an der Unterbrechungsstelle fortgesetzt wird. Bild 2-21 zeigt schematisch den Programmfluß bei einer Programmunterbrechung.

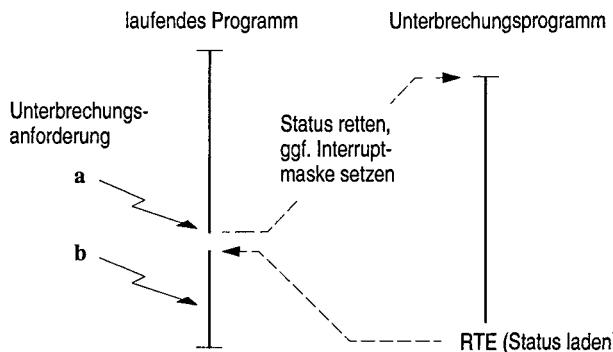


Bild 2-21. Programmfluß bei einer Programmunterbrechung. a Unterbrechungsanforderung stattgegeben, b nicht stattgegeben

Unterbrechungsvektoren. Die Adressen sämtlicher Unterbrechungsprogramme sind als sog. Unterbrechungsvektoren (*Interruptvektoren, Trap-Vektoren*) in einer *Vektortabelle* im Hauptspeicher zusammengefaßt. Bei einer Adreßlänge von 32 Bit belegt jeder Vektor 4 Bytes in der Tabelle. Die Adressen dieser Vektoren wiederum bezeichnet man als Vektoradressen. Diese werden aus *Vektornummern* gebildet, die den Unterbrechungsanforderungen fest zugeordnet sind. Die Vektornummern werden abhängig von der Art der Anforderung entweder vom Prozessor selbst erzeugt, oder sie werden dem Prozessor über den Datenbus von außen zugeführt (siehe 5.5.1). Die für die Anwahl eines Unterbrechungsprogramms erforderlichen Adressierungsvorgänge sind in Bild 2-22 dargestellt.

Tabelle 2-4 zeigt die wichtigsten *Unterbrechungsbedingungen*, geordnet nach ihren Vektornummern bzw. nach den sich daraus ergebenden Adreßdistanzen für die Einträge der Vektortabelle. Die Spalte Quelle kennzeichnet mit „intern“ und „extern“ den prozessorbezogenen Auslöseort einer Unterbrechungsanforderung. Ferner sind den Bedingungen *Prioritäten* (g.i) zugeordnet, die sich aus einer

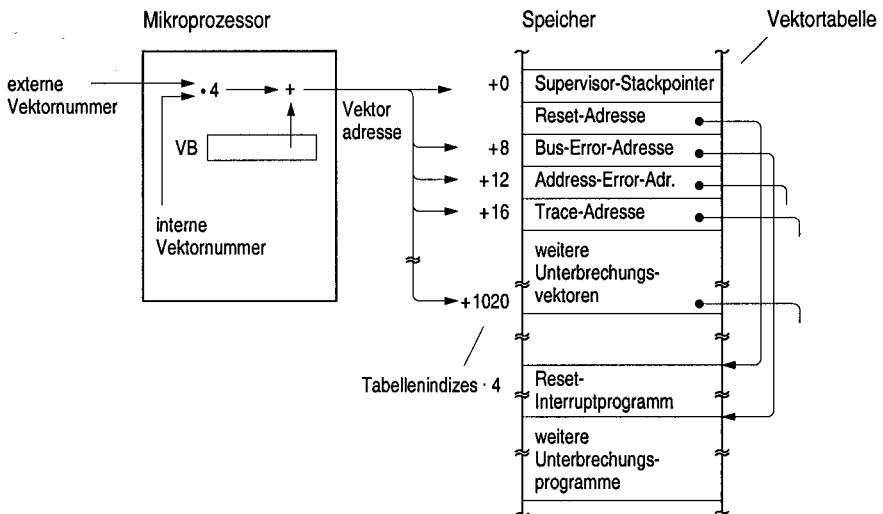


Bild 2-22. Anwahl von Unterbrechungsprogrammen. Vorgabe der Basisadresse der Vektortabelle über das Vectorbase-Register VB

Gruppenpriorität g und einer gruppeninternen Priorität i ergeben. Hierbei bedeutet 0 die höchste und 4 die niedrigste Priorität. Sie werden vom Mikroprogrammwerk des Prozessors ausgewertet, so daß ein laufendes Unterbrechungsprogramm nur durch eine Anforderung höherer Priorität unterbrochen werden kann. Bei den folgenden Erläuterungen der Tabelle unterscheiden wir zwischen Traps und Interrupts mit speziellen Auslösebedingungen und Traps und Interrupts, die allgemein verwendbar sind.

Spezielle Traps und Interrupts. Die speziellen Traps und Interrupts werden durch Bedingungen ausgelöst, die hauptsächlich der Initialisierung, dem Systemtest und der Fehlererkennung dienen. Sie haben in Tabelle 2-4 die Vektornummern 0 bis 9 und sind nach absteigenden Prioritäten geordnet.

- **Reset-Interrupt:** Er wird über den RESET-Steuereingang ausgelöst, entweder automatisch beim Einschalten der Versorgungsspannung oder bei bereits laufendem System manuell. Er führt mit seinem Unterbrechungsprogramm die Systeminitialisierung durch und hat dementsprechend die höchste Priorität. In der Unterbrechungsverarbeitung bildet er eine Ausnahme, indem er nicht nur den Befehlszähler mit der Startadresse des Unterbrechungsprogramms, sondern auch das Supervisor-Stackpointerregister mit einem Anfangswert aus der Vektortabelle lädt; zuvor setzt er das Vectorbase-Register auf Null. Außerdem löscht er die allgemeinen Prozessorregister.
- **Bus-Error-Trap:** Er wird über den BERR-Signaleingang ausgelöst, nachdem eine Einheit im System einen Busfehler festgestellt hat. Dies kann z.B. das

Ausbleiben des Quittungssignals **DTACK** bzw. **READY** an den Prozessor bei einem Lese- oder Schreibzyklus sein (siehe 5.3).

- Address-Error-Trap: Er wird bei der Speicheradressierung ausgelöst, wenn die ausgegebene Adresse der Alignment-Vorgabe nicht entspricht (z.B. ungerade Halbwortadresse) und der Prozessor ein Misalignment nicht zuläßt. Dies kann sowohl Daten- als auch Befehlszugriffe betreffen.

In allen drei Fällen erfolgt die Ausnahmeverarbeitung sofort nach dem Erkennen der Bedingung, d.h. mit dem nächsten Maschinentakt.

- Illegal-Instruction-Trap: Der Prozessor erkennt einen nicht definierten Operationscode oder eine nicht zulässige Adressierungsart. Die Unterbrechung erfolgt unmittelbar im Anschluß an die Decodierphase.

Tabelle 2-4. Unterbrechungsbedingungen (in Anlehnung an die MC680x0-Prozessoren)

Vektor-nummer	Adreß-distanz	Unterbrechungsbedingungen	Quelle	Priorität (g.i)
0	0	Reset (Init. v. SSP u. PC)	extern	0.0
2	8	Bus Error	"	1.1
3	12	Address Error	intern	1.0
4	16	Illegal Instruction	"	3.0
5	20	Zero Divide	"	2.0
:	:	:		
8	32	Privilege Violation	intern	3.0
9	36	Trace	"	4.1
10	40	OPC Emulation 1	"	3.0
11	44	OPC Emulation 2	"	3.0
:	:	:		
15	60	Uninitialized Interrupt	extern	4.2
:	:	:		
25	100	Level 1 Autovector Interrupt	extern	4.2
26	104	Level 2 "	"	4.2
27	108	Level 3 "	"	4.2
28	112	Level 4 "	"	4.2
29	116	Level 5 "	"	4.2
30	120	Level 6 "	"	4.2
31	124	Level 7 "	"	4.2
32	128	TRAP Instructions (16)	intern	2.0
:	:	"		
47	188	"		
64	256	Vector Interrupts (192)	extern	4.2
:	:	"		
255	1020	"		

- Zero-Divide-Trap: Der Prozessor findet bei der Ausführung des Divisionsbefehls einen Divisor mit dem Wert Null vor. Die Unterbrechung erfolgt während der Befehlsverarbeitung.
- Privilege-Violation-Trap: Der Prozessor befindet sich im User-Modus und erkennt den Operationscode eines privilegierten Befehls. Die Unterbrechung erfolgt unmittelbar im Anschluß an die Decodierphase.
- Trace-Trap: Das Trace-Bit T0 oder T1 im Statusregister ist gesetzt (Trace-Modus, siehe auch 2.1.1). Die Unterbrechung erfolgt nach der Befehlsverarbeitung. Das *Trace-Programm* wird zum Systemtest benutzt, indem mit ihm z.B. der Prozessorstatus ausgegeben wird.
- OPC-Emulation-Traps: Der Prozessor erkennt bestimmte Operationscodes nicht implementierter Befehle. Die Unterbrechung erfolgt unmittelbar im Anschluß an die Decodierphase; der Befehl wird durch das Trap-Programm simuliert.
- Uninitialized Interrupt: Auf das Interruptsystem des Prozessors zugeschnittene Interface-Bausteine initialisieren ihre Vektornummerregister in der Reset-Phase mit der Vektornummer 15. Wird danach durch einen solchen Baustein ein Interrupt ausgelöst, ohne daß das Vektornummerregister zuvor mit der für diese Quelle vorgesehene Vektornummer geladen wurde, so läuft dieser Interrupt hier auf. Das Interruptprogramm kann dann dazu genutzt werden, diesen Fehlerfall anzuzeigen.

Allgemeine Traps. Die allgemeinen Traps werden durch den Befehl TRAP ausgelöst, wobei durch einen Parameter einer von mehreren Unterbrechungsvektoren angewählt werden kann (in Tabelle 2-4 insgesamt 16 Vektoren). Sie werden als *Supervisor-Calls* für den kontrollierten Übergang vom User-Modus (Anwendungsprogramme) in den Supervisor-Modus (Systemprogramme) eingesetzt.

Allgemeine Interrupts. Allgemeine Interrupts werden von externen Einheiten ausgelöst. Bei der Tabelle 2-4 zugrundeliegenden Prozessorstruktur (angelehnt an die MC680x0-Prozessoren), werden sie ihm als 3-Bit-*Interruptcode* über 3 Interrupteingänge übermittelt. Der Prozessor benutzt diese Codierung zur Unterscheidung von sieben *Interruptebenen* unterschiedlicher *Prioritäten*. Der Interruptcode 7 hat hierbei die höchste, der Interruptcode 1 die niedrigste Priorität. Der Code 0 besagt, daß keine Interruptanforderung anliegt. Eine Programmunterbrechung erfolgt mit Ausnahme der Ebene 7 bei einer Interruptanforderung, die eine höhere Priorität als das laufende Programm hat. Dessen Priorität ist durch die 3-Bit-*Interruptmaske* (IM2 bis IM0) im Statusregister festgelegt (siehe Bild 2-3, S. 71); sie wird bei einer Programmunterbrechung gleich dem stattgegebenen Interruptcode gesetzt.

Interrupts der höchsten Priorität (Ebene 7) wird unabhängig von der Interruptmaske (auch wenn sie den Wert 7 hat) immer stattgegeben, weshalb man sie als *nichtmaskierbare Interrupts* (non maskable interrupts) bezeichnet. Den anderen

Interrupts (Ebenen 6 bis 1) werden in Abhängigkeit von der Interruptmaske stattgegeben oder nicht. Man bezeichnet sie deshalb als *maskierbare Interrupts* (maskable interrupts).

Bei den sog. *Autovektor-Interrupts* ist den sieben Unterbrechungsebenen je eine Vektornummer durch die Prozessorhardware fest zugeordnet (25 bis 31), d.h., die Anzahl der Unterbrechungsvektoren hängt nur vom Interruptcode ab. Bei den sog. *Vektor-Interrupts* übergibt dagegen die Interruptquelle dem Prozessor eine 8-Bit-Vektornummer (64 bis 255) auf dem Datenbus. Der Prozessor wählt damit einen von 192 möglichen Unterbrechungsvektoren aus. Die Unterscheidung zwischen Autovektor- und Vektor-Interrupts trifft der Prozessor anhand eines Eingangssignals \overline{AVEC} . Es wird von der jeweiligen Interruptquelle mit 0 (Autovektor-Interrupt) bzw. mit 1 (Vektor-Interrupt) mit der Interruptanforderung geliefert.

Anmerkung. Eine gebräuchliche Vereinfachung des Interruptsystems, wie sie viele Mikroprozessortypen aufweisen, besteht darin, anstelle des Interruptcodes einzelne Interruptleitungen vorzusehen, z.B. eine für *maskierbare* und eine für *nichtmaskierbare Interrupts*. Maskierbare Interrupts haben auch hier geringere Priorität als die nichtmaskierbaren. Die maskierbare Leitung ist dabei üblicherweise für Vektor-Interrupts ausgelegt; die ihr zugeordnete *Interruptmaske* im Statusregister reduziert sich entsprechend der Leitunganzahl auf ein Bit. Die *Priorisierung* von Vektor-Interrupts erfolgt prozessorextern (siehe 5.5.2).

Beispiel 2.5. ► *Programmunterbrechung durch ein Uhr-Interruptsignal.* Eine externe Uhr (real time clock, Bild 2-23) soll in festen Zeitabständen Programmunterbrechungen auslösen, die zum Hochzählen einer Speicherzelle COUNT benutzt werden. Die Uhr hat dazu ein 8-Bit-Statusregister TIMESR, dessen Bit 0 mit jedem Uhrimpuls auf 1 gesetzt wird und damit über einen Codierer den Interruptcode 6 an den Interrupteingängen IL2 bis IL0 des Prozessors erzeugt. Der \overline{AVEC} -Eingang des Prozessors wird gleichzeitig mit 1 vorgegeben, so daß der Prozessor eine Anforderung als Vektor-Interrupt erkennt. Hat er eine Anforderung akzeptiert, so bestätigt er dies durch ein Quittungssignal IACK (interrupt acknowledge; siehe auch 5.5.1), worauf die Uhr dem Mikroprozessor die in einem Register stehende Vektornummer 64 auf dem Datenbus übergibt. Der Pro-

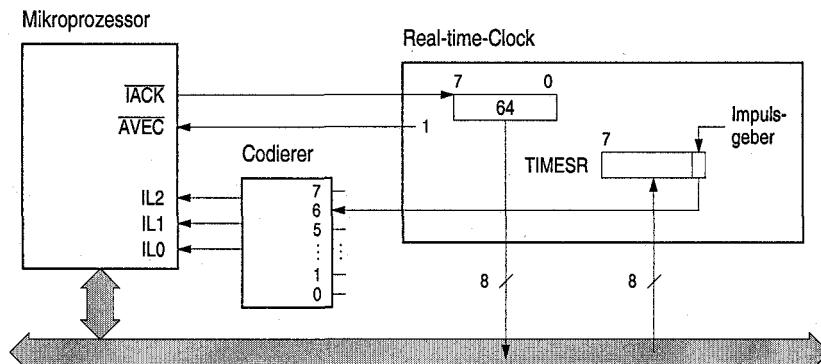


Bild 2-23. Anschluß einer Echtzeituhr (Real-time-Clock)

zessor lädt daraufhin den Befehlszähler mit dem zugehörigen Interruptvektor (Adreßdistanz 256). Dieser ist die Anfangsadresse des Interruptprogramms TIMER.

Das folgende Interruptprogramm TIMER zählt den Inhalt der Speicherzelle COUNT um Eins hoch, löscht danach im Register TIMESR das Bit 0 und setzt damit die Interruptanforderung zurück. Während der Ausführung des Interruptprogramms ist eine weitere Anforderung der Ebene 6 ebenso wie Anforderungen der Ebenen 5 bis 1 durch die Interruptmaske im Statusregister blockiert.

```
TIMER: ADD.B #1,COUNT ;Zähler erhöhen
        BCLR.B #0,TIMESR ;Anforderung löschen
        RTE
```

◀

Ausnahmeverarbeitung. Die Ausnahmeverarbeitung durch den Prozessor (*exception processing*) betrifft seine Operationen zwischen dem Akzeptieren einer Unterbrechungsanforderung und dem Starten des zugehörigen Unterbrechungsprogramms. Sie umfaßt die vier folgenden grundsätzlichen Schritte (siehe auch 5.5.1: *Interruptzyklus*).

1. Der Statusregisterinhalt wird zunächst in ein prozessorinternes Pufferregister kopiert. Danach wird der Status verändert, indem das S-Bit gesetzt (umschalten in den *Supervisor-Modus*) und die Bits T0 und T1 zurückgesetzt werden (unterdrücken der Trace-Funktion). Bei allgemeinen Interruptanforderungen wird zusätzlich die Interruptmaske gleich dem Interruptcode gesetzt; bei einem Reset-Interrupt wird sie auf 7 gesetzt (höchste Priorität). Damit werden Anforderungen gleicher und geringerer Priorität blockiert. Als Ausnahme lassen sich Interrupts der Ebene 7 durch eine Maske 7 nicht blockieren, sie sind eben nicht maskierbar.
2. Die Vektornummer wird ermittelt und daraus die Vektoradresse durch Multiplizieren mit 4 (2-Bit-Linksshift) und anschließendem Addieren zu der im Vectorbase-Register stehenden Tabellenbasisadresse gebildet.
3. Der *Prozessorstatus* wird *gerettet*. Dazu werden der Befehlszählerinhalt und der gepufferte Statusregisterinhalt auf den Supervisor-Stack geschrieben. (Dieser Schritt entfällt beim Reset-Interrupt).
4. Der Befehlszähler wird mit dem Unterbrechungsvektor geladen und der Befehlsabruft eingeleitet. Damit kommt der erste Befehl des Unterbrechungsprogramms zur Ausführung.

Betriebsarten. Ein wesentlicher Aspekt für die Betriebssicherheit eines Rechners ist der Schutz der für den Betrieb erforderlichen Systemsoftware (Betriebssystem) gegenüber unerlaubten Zugriffen durch die Benutzersoftware. Grundlage hierfür sind die in der Prozessorhardware verankerten Betriebsarten zur Vergabe von *Privilegien* für die Programmausführung. Üblich sind Mikroprozessoren mit zwei Privilegienebenen (z. B. MC680x0) und mit vier Ebenen (z. B. Pentium).

Bei einem Prozessor mit zwei Ebenen unterscheidet man als Betriebsarten den privilegierten *Supervisor-Modus* für die Systemsoftware und den ihm untergeord-

neten *User-Modus* für die Anwendersoftware. Festgelegt ist die jeweilige Betriebsart durch das S-Bit im Statusregister. Der *Schutzmechanismus* besteht zum einen in der prozessorexternen Anzeige der Betriebsart durch die Statussignale des Prozessors. Diese kann von einer prozessorexternen Speicheranwahllogik, z.B. einer *Speicherverwaltungseinheit* dazu genutzt werden, Zugriffe von im User-Modus laufenden Programmen für bestimmte Adreßbereiche einzuschränken (z.B. nur Lesezugriff erlaubt) oder sie sogar ganz zu unterbinden. Demgegenüber können der Supervisor-Ebene die vollen Zugriffsrechte eingeräumt werden (siehe 6.3.4). Der Schutzmechanismus besteht zum andern in der Aufteilung der Stack-Aktivitäten auf einen Supervisor- und einen User-Stack. Dazu sieht der Prozessor die beiden Stackpointerregister SSP und USP vor, die abhängig von der Betriebsart „sichtbar“ werden (siehe auch 2.1.1).

Einen weiteren Schutz bieten die *privilegierten Befehle*, die nur im Supervisor-Modus ausführbar sind (siehe 2.1.4). Sie erlauben es u.a., die Modusbits im Statusregister zu verändern, z.B. die Interruptmaske zu manipulieren oder den Trace-Modus einzuschalten. Mit dem privilegierten Befehl MOVUSP ist es außerdem möglich, von der Supervisor-Ebene aus auf das User-Stackpointerregister USP zuzugreifen. Der Versuch, privilegierte Befehle im User-Modus auszuführen, führt hingegen zu einer Programmunterbrechung (privilege violation trap).

Traps und Interrupts, die im User-Modus auftreten, bewirken immer eine Umschaltung in den Supervisor-Modus, d.h., alle *Trap- und Interruptprogramme* werden grundsätzlich in der privilegierten Ebene ausgeführt. Für die *Anwenderprogramme* im User-Modus sind Traps die einzige Möglichkeit, in den Supervisor-Modus zu gelangen (kontrollierte Übergänge, *supervisor calls*). Der Übergang vom Supervisor-Modus in den User-Modus hingegen wird üblicherweise durch den RTE-Befehl vollzogen, z.B. als Abschluß von Unterbrechungsprogrammen, wenn das unterbrochene Programm im User-Modus lief. RTE wird aber auch eingesetzt, um vom Betriebssystem aus ein Anwenderprogramm zu starten. Hier kommt es darauf an, die Umschaltung vom Supervisor- in den User-Modus und das Laden der Startadresse in den Befehlszähler innerhalb eines Befehls vorzunehmen. Dazu werden zunächst der neue Statusregisterinhalt (mit S = 0) und die Startadresse auf den Supervisor-Stack geschrieben, um dann durch RTE nach SR und PC geladen zu werden.

2.2 RISC-Programmiermodell

Heutige RISCs sind durch zwei Prozessorarchitekturen geprägt, die an den Universitäten Berkeley und Stanford entwickelt wurden: die *SPARC-Architektur* (Scalable Processor ARChitecture) und die *MIPS-Architektur* (Microprocessor without Interlocked Pipeline Stages). Sie unterscheiden sich von CISCs zunächst durch eine Vereinfachung der Befehlssstruktur bei Vergrößerung des Registerspei-

chers und gleichzeitig guter Abstimmung der Funktionsabläufe, um einen möglichst hohen Befehlsdurchsatz bei hoher Taktfrequenz zu erreichen (Fließbandverarbeitung). Dies führt zunächst gegenüber CISCs zu einer Vereinfachung, z.B. des Befehlssatzes und der Adreßmodifizierungen, führt ggf. aber auch zu komplexeren Erscheinungen anderer Architekturmerkmale, bei der SPARC-Architektur z.B. bei der Adressierung des allgemeinen Registerspeichers. Die Vereinfachung hat zur Folge, daß viele Funktionen, die bei CISCs durch die Hardware, d.h. automatisch ausgeführt werden, hier durch die Software nachgebildet werden müssen, z.B. die Realisierung komplexer Operationen oder komplexer Adreßmodifizierungen. Die gegenüber CISCs höhere Komplexität z.B. des Registerspeichers erleichtert andererseits die Programmierung geschachtelter Unterprogrammaufrufe. – In diesem Abschnitt werden wir die zuvor für CISCs beschriebenen Strukturmerkmale in ihren für RISC-Architekturen typischen Ausprägungen darstellen.

2.2.1 Registersatz und Prozessorstatus

Als vom Programm direkt ansprechbare Prozessorregister (Registersatz) gibt es wie bei den CISCs einen allgemeinen Registerspeicher sowie spezielle Register, wie das Statusregister und ggf. ein oder mehrere Arithmetikregister zur Unterstützung der Multiplikation und der Division durch die Software. Ergänzt wird der Registersatz durch einen oder (durch die Fließbandverarbeitung bedingt) mehrere Befehlszähler. Ein Stackpointerregister zur Adressierung eines Stackbereichs im Hauptspeicher ist nicht üblich. Der Registerspeicher enthält allerdings erheblich mehr Register als bei CISCs. Dieser Speicher ist Quelle und Ziel der Operanden aller operandenverarbeitenden Befehle, da Zugriffe auf den Hauptspeicher vermieden werden, um den Fluß im Fließband nicht zu verzögern. In diesem Zusammenhang ist man bestrebt, Registerinhalte beim Aufruf eines Unterprogramms nicht in den Hauptspeicher auszulagern, sondern jedem Unterprogramm einen Teil des Registerspeichers als lokalen Arbeitsbereich zuzuordnen. Hierzu wird dem Registerspeicher bei einigen Realisierungen eine Strukturierung in Form von „Fenstern“ aufgeprägt.

Unstrukturierter Registerspeicher. Eine erste Möglichkeit, die Zugriffshäufigkeit auf den Registerspeicher gegenüber CISC-Prozessoren zu verbessern, ist eine einfache Erweiterung dieses Speichers auf z.B. 32 Register r0 bis r31, wie sie bei RISCs, die der *MIPS-Architektur* folgen, typisch ist. Zwei der Register, z.B. r0 und r31, übernehmen dabei spezielle Funktionen:

- r0 liefert bei einem Lesezugriff den konstanten Wert Null, ein Schreibzugriff auf r0 bleibt ohne Wirkung. Letzteres ist – wie in 1.4.2 beschrieben – bei Vergleichsoperationen wichtig, da diese wegen des Fehlens eines Compare-Befehls mit dem Subtraktionsbefehl ausgeführt werden müssen und dieser einen Schreibzugriff mit einem Zieloperanden ausführt. Diese Spezialfunk-

tion von r0 gibt es auch bei den beiden nachfolgend beschriebenen Registerspeichern, und zwar jeweils für eines der dort vorhandenen globalen Register.

- r31 dient zur Speicherung der Rücksprungadresse bei Unterprogrammsprüngen. Da es kein Stackpointerregister und dementsprechend auch keinen Stackmechanismus gibt, muß das Retten und Rückschreiben der Rücksprungadressen bei der Schachtelung und Entschachtelung von Unterprogrammen von der Software vorgenommen werden.

Mit unstrukturiertem Registerspeicher sind z.B. die PowerPC-Prozessoren ausgestattet.

Nachteil eines solchen, auf 32 Register begrenzten und unstrukturierten Registerspeichers ist, daß sein Inhalt beim Unterprogrammaufruf ggf. in den Hauptspeicher ausgelagert werden muß, um dem Unterprogramm Arbeitsregister zur Verfügung stellen zu können, und daß dementsprechend bei der Rückkehr aus einem Unterprogramm der ursprüngliche Inhalt aus dem Hauptspeicher wieder geladen werden muß. Effizient läßt sich dabei lediglich die Parameterübergabe gestalten, indem die hierfür erforderlichen Register in das Aus- und Einlagern nicht mit einbezogen werden.

Registerfenster fester Größe. Ein Ansatz für eine bessere Lokalisierung der Datenzugriffe auf den Registerspeicher ist, die Kapazität des Registerspeichers wesentlich zu vergrößern und zusätzlich eine *Strukturierung des Registerspeichers* im Sinne lokaler Arbeitsbereiche für Unterprogramme einzuführen. Die SPARC-Architektur sieht einen solchen vergrößerten, strukturierten Registerspeicher vor, von dem jedem Unterprogramm ein Ausschnitt in Form eines *Fensters* (window) von 24 Registern zugänglich ist (Bild 2-24). Diese als *lokale Arbeitsregister* bezeichneten Bereiche werden durch acht *globale Register* ergänzt, auf die von allen Unterprogrammen übergeordnet zugegriffen werden kann. (g0 hat hier die Funktion des oben erwähnten Konstantenregisters r0.) Dementsprechend hat ein Unterprogramm auf insgesamt 32 Register Zugriff.

Die SPARC-Architektur sieht einen Ausbau eines solchen Registerspeichers auf bis zu 32 Fenster vor, wobei sich derzeitige Realisierungen allerdings auf acht Fenster, d.h. auf 136 Register insgesamt (einschließlich der acht globalen Register), beschränken. Wie Bild 2-24 zeigt, sind die 24 lokalen Register eines Fensters in drei Gruppen zu je acht Registern mit den Bezeichnungen In, Local und Out unterteilt. Dabei sind die In-Register des aktuellen Fensters mit den Out-Registern des vorangehenden Fensters identisch, ebenso die Out-Register des aktuellen Fensters mit den In-Registern des nachfolgenden Fensters. Zuletzt schließt sich die Anordnung, indem die Out-Register des letzten Fensters gleich den In-Registern des ersten Fensters sind. Sinn dieser Überlappung von Registern ist die Vereinfachung der *Parameterübergabe* und -rückgabe beim Unterprogrammanschluß.

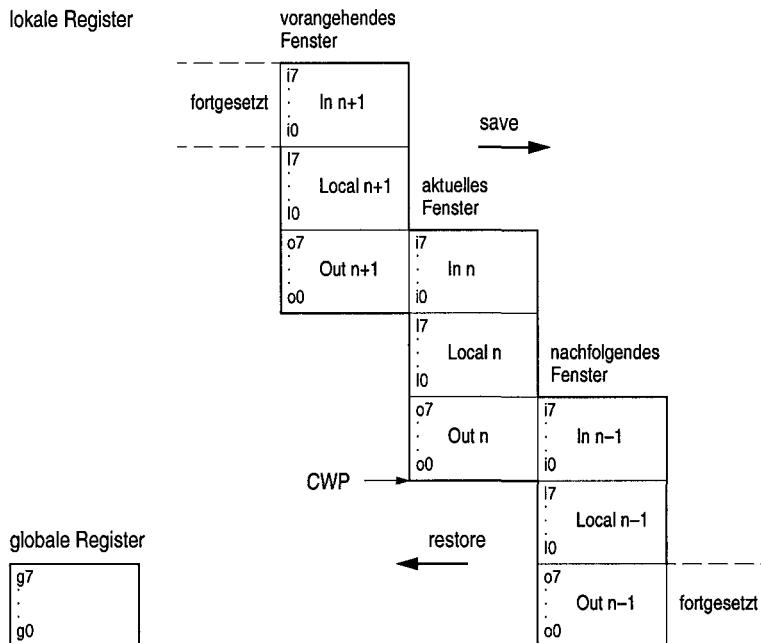


Bild 2-24. Registerspeicher mit globalen Registern und mit Registerfenstern konstanter Größe entsprechend der SPARC-Architektur

Die Nummer des aktuellen Fensters wird durch einen sog. *Current-Window-Pointer* (CWP) im Statusregister des Prozessors angezeigt. Das Umschalten auf das nächste Fenster geschieht nach dem Aufrufen eines Unterprogramms durch den Befehl call, und zwar durch den Befehl save als ersten Befehl im Unterprogramm, der den CWP dekrementiert. Die Rückkehr zum vorangehenden Fenster wird beim Rücksprung durch den Befehl restore ausgelöst, der den CWP inkrementiert. Vor Ausführen des call-Befehls schreibt das aufrufende Programm die zu übergebenden Parameter in seine Out-Register, so daß das aufgerufene Programm sie nach Ausführen von save in seinen In-Registern vorfindet. Ein weiterer Parametertransport ist somit nicht nötig. Die Rückgabe von Parametern erfolgt über dieselben Register, jedoch in umgekehrter Richtung und mit umgekehrter Fensterumschaltung. Die Bezeichnungen In und Out beziehen sich dabei auf die Eingangsparameter und nicht auf die Ausgangsparameter. – Mit dem Unterprogrammaufruf durch call wird auch der Inhalt des PC in eines der Out-Register des aktuellen Fensters geschrieben. Anders als beim unstrukturierten Registerspeicher braucht dieser vor dem nächsten Unterprogrammaufruf nicht in den Hauptspeicher gerettet zu werden, wenn, wie üblich, im Unterprogramm durch save eine Fensterumschaltung veranlaßt wird (siehe auch 3.4.2).

Wenn bei der Ausführung des save-Befehls die Anzahl der vorhandenen Registerfenster, abzüglich eines letzten Fensters für die Trap-Behandlung überschritten

wird, so wird ein Window-overflow-Trap ausgelöst. Das Trap-Fenster wird dazu in einem speziellen Prozessorregister als solches markiert. Dieser Window-overflow-Trap führt auf eine Routine des Betriebssystems, die den Inhalt des auf das Trap-Fenster folgenden und bereits belegten Fensters in den Hauptspeicher auslagert und dieses nun frei gewordene Fenster als neues Trap-Fenster markiert. Das bisherige Trap-Fenster steht dann dem unterbrochenen save-Befehl als leeres Fenster zur Verfügung. In analoger Weise löst der restore-Befehl, wenn er „von der anderen Seite kommend“ auf das Trap-Fenster läuft, einen Window-underflow-Trap aus. Dieser wird dazu genutzt, den Inhalt des zuletzt ausgelagerten Fensters wiederherzustellen und die Markierung des Trap-Fensters dementsprechend um eine Position zurückzuverschieben. Auf diese Weise kann das Fensterprinzip für eine beliebige Schachtelungstiefe, wenn auch mit wesentlich geringerer Effizienz benutzt werden. Nachteil der Fenstertechnik ist, daß die Fenster insgesamt wie auch ihre Teilbereiche In, Local und Out eine feste Größe haben und somit nicht dem tatsächlichen Speicherplatzbedarf eines Unterprogramms angepaßt werden können. – Aufgrund des mit den Unterprogrammaufrufen im Registerspeicher einhergehenden Fensterumschaltens spricht man auch von einem Fenster/Register-Stack mit dem CWP als Stackpointer.

Registerfenster variabler Größe. Eine *Strukturierung des Registerspeichers* in Fenster variabler Größe erhält man durch basisrelative Adressierung. Das heißt, die jeweils benötigten Fenster werden durch beliebig wählbare Basisadressen im Registerspeicher positioniert und sind in ihren Größen durch die Anzahl der in einem Unterprogramm jeweils benötigten Register bestimmt. Die Registeradressen eines Fensters werden dabei als Distanzen zur Basisadresse vorgegeben, d.h., in jedem Fenster beginnt die Adressierung mit r0. Die Strukturierung bezieht sich auf die „lokalen Register“ des Registerspeichers, die aktuelle Basisadresse steht in einem der globalen Register des Registerspeichers.

Bild 2-25 zeigt, angelehnt an den 32-Bit-RISC-Prozessor Am29000 [Advanced Micro Devices 1993], einen solchen Registerspeicher mit 64 globalen und 128 lokalen Registern (Teilbild a) und die Strukturierung des lokalen Teils in zwei Fenster als Schnappschuß bei der Ausführung eines Unterprogramms in der Schachtelungstiefe 1 (Teilbild b). Wie man sieht, können auch hier In- und Out-Registerbereiche für die Parameterübergabe gebildet werden, indem man die Fenster sich überlappen lässt. Der für die lokalen Variablen des Unterprogramms benötigte Local-Bereich ist zwischen den In- und Out-Bereichen eines Fensters angeordnet. Alle drei Bereiche sind in ihren Größen frei wählbar und somit an den tatsächlichen Bedarf eines Unterprogramms anpaßbar. Aufgrund des mit den Unterprogrammaufrufen und -rücksprüngen pulsierenden Auf- und Abbaus solcher Fenster im Registerspeicher spricht man auch hier wieder von einem Fenster- oder Register-Stack mit der Fenster-Basisadresse als Stackpointer. – Das gleiche Prinzip findet man bei der 64-Bit-Architektur von Intel (IA-64, [Intel 1999]), derzeit realisiert im *Itanium* 2. Hier umfaßt der allgemeine Registerspeicher 128 64-Bit-

Register, von denen die ersten 32 als globale Register statisch und die restlichen 96 als lokale Register dynamisch, d.h. als Register-Stack verwaltet werden.

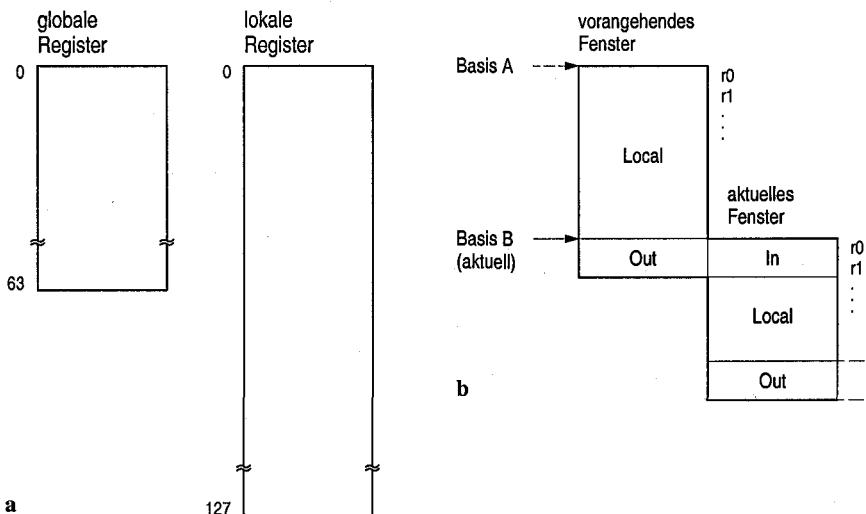


Bild 2-25. Registerspeicher mit Registerfenstern variabler Größe in Anlehnung an den RISC Am29000; a globale und lokale Register, b Beispiel für die Fensterbildung mit den lokalen Registern zum Zeitpunkt der Ausführung eines Unterprogramms (Register-Stack)

Statusregister. Das Statusregister eines RISC hat im wesentlichen die gleichen Aufgaben wie bei einem CISC, indem es den Zustand des Prozessors in Form der Bedingungsbits N, Z, V und C anzeigt und den Modus des Prozessors in Form der Betriebsebene, z.B. S (supervisor/user), und der Interruptmaske IM (z.B. 4 Bits für 16 Interruptebenen) angibt. Unterschiede gibt es aufgrund der gegenüber CISCs nur rudimentär vorhandenen Mechanismen zum Statusretten und Statusverändern bei der Ausnahmebehandlung (siehe dazu 2.2.5). So gibt es z.B. beim SPARC, wie Bild 2-26 zeigt, ein Bit PS (previous S-Bit), in das bei einer Programmunterbrechung lediglich die aktuelle Betriebsebene, d.h. das S-Bit gesichert wird, und ein Bit ET (enable trap), mit dem die maskierbaren Interrupts global blockiert werden. Das Blockieren von Interrupts gleicher und geringerer

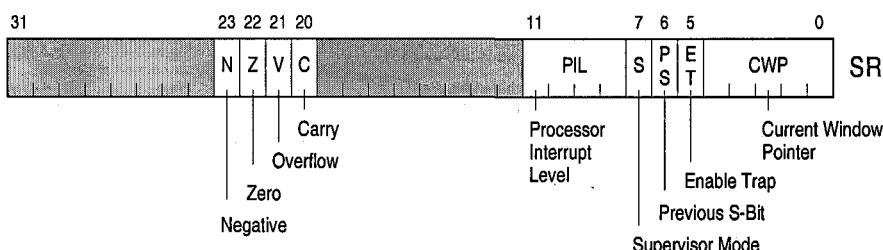


Bild 2-26. Statusregister eines RISC in Anlehnung an die SPARC-Architektur

Prioritäten mittels der Interruptmaske, hier als Processor-Interrupt-Level (PIL) bezeichnet, wird der Software überlassen. – Auch der für den SPARC typische Current-Window-Pointer CWP wird im Statusregister geführt.

Befehlszähler. Entgegen den Ausführungen in Abschnitt 1.4 gibt es bei RISCs ggf. mehr als nur einen Befehlszähler, so z.B. beim SPARC den PC und den nPC (next PC). Hierbei übernimmt der nPC die übliche in 1.4 dem PC zugewiesene Funktion der Befehlsadressierung beim „Befehl holen“. Der PC hingegen speichert den vorangegangenen Zustand des nPC und verweist dementsprechend auf den sich bereits in der Stufe „Befehl decodieren und Operanden holen“ befindlichen Befehl. Im „Normalfall“ der linearen Programmausführung erfolgt die Programmfortschaltung mit der mit jedem Befehl ausgeführten Sequenz nPC → PC, nPC+4 → nPC, deren Wirkung auch mit nur einem PC erreicht würde. Erforderlich wird der zweite PC jedoch im „Ausnahmefall“ einer Programmunterbrechung, und zwar dann, wenn die Unterbrechung während des Abrufs eines Delay-Slot-Befehls auftritt. Unter der Annahme, daß die Sprungbedingung erfüllt war, muß hier bei Wiederaufnahme des unterbrochenen Programms zunächst der Delay-Slot-Befehl erneut geholt und danach der am Sprungziel stehende Befehl abgerufen werden. Dementsprechend sind für die Wiederaufnahme sowohl die Adresse des Delay-Slot-Befehls als auch die ermittelte Sprungadresse erforderlich, die zum Zeitpunkt der Unterbrechung im PC bzw. im nPC stehen. Zum Vorgang des Wiederholens des unterbrochenen Befehls siehe 2.2.4, Trap-Befehle.

2.2.2 Datenformate, Datentypen und Datenzugriff

Die elementaren *Datenformate* bei RISCs sind das Byte (8 Bits), das Halbwort (16 Bits), das Wort (32 Bits) und das Doppelwort (64 Bits). Dargestellt werden mit ihnen im wesentlichen die Datentypen vorzeichenlose Dualzahl und 2-Komplement-Zahl (Byte, Halbwort und Wort) sowie Gleitpunktzahl (Wort für die einfache lange und Doppelwort für doppelt lange Darstellung). Bei der Verarbeitung von Daten in RISCs gibt es – anders als bei CISCs – eine grundsätzliche Festlegung auf das 32-Bit-Format. So sind zwar das Byte und das Halbwort durch entsprechende Lade- und Speichere-Befehle im Hauptspeicher als Datenformate lokalisierbar und bilden dementsprechend Transporteinheiten zwischen Hauptspeicher und Prozessor. Im Prozessor selbst findet jedoch immer eine *Verarbeitung im Wortformat* statt (siehe 2.2.4: arithmetische, logische und Shiftbefehle). Die Formattanpassung geschieht beim Laden in den Registerspeicher des Prozessors, indem ein Byte- oder Halbwortoperand grundsätzlich rechtsbündig in das Zielregister geschrieben und dann die höherwertigen Bits des Registers entweder mit Nullen oder mit dem Wert des Vorzeichenbits aufgefüllt werden. Die Art der Erweiterung, *Zero-Extension* bzw. *Sign-Extension*, kann im Lade-Befehl angegeben werden.

Die Speicherung von Operanden (und Befehlen) im Hauptspeicher muß immer mit *Data-Alignment* erfolgen, d.h. Halbwortadressen müssen durch 2, Wortadressen durch 4 und Doppelwortadressen durch 8 teilbar sein. Bytes können an jeder beliebigen Adresse stehen. Die Art der Adreßzählung innerhalb eines Speicherworts, *Big-endian-* oder *Little-endian-Byteanordnung* (siehe 2.1.2 und 5.2.4), ist abhängig von Prozessor zu Prozessor auf die eine oder andere Weise wählbar: entweder in einem Steuerregister frei programmierbar oder durch ein externes Steuersignal vorgebbar; oder der Prozessor wird in zwei Versionen angeboten.

2.2.3 Adressierungsarten und Befehlsformate

Adressierungsarten. Die Möglichkeiten zur Adreßmodifizierung sind bei RISCs stark eingeschränkt. Sie sind befehlsabhängig und lassen sich folgenden vier Befehlsgruppen zuordnen (Bild 2-27):

1. Arithmetische, logische und Shiftbefehle: Sie können nur den Registerspeicher adressieren oder einen der beiden Quelloperanden als Direktoperanden in die Operation mit einbeziehen.

- a) *Register-Adressierung.* Die Adresse steht als Registeradresse (kurze Adresse, ri) im Befehl (Quelle 1, Quelle 2, Ziel); der Operand steht im Register (Bild 2-27a).

Assemblerschreibweise: ri

- b) *Direktoperand-Adressierung.* Der Operand steht als kurze Konstante im Befehl (immediate operand, imm); er wird vor der Operation durch Sign-Extension erweitert (Bild 2-27b).

Assemblerschreibweise: imm

Anmerkung. Bei den CISC-Adressierungsarten in 2.1.3 wurde der Direktoperand im Assemblercode durch das #-Zeichen gekennzeichnet, um ihn von Speicheradressen zu unterscheiden. Bei RISCs ist dies nicht erforderlich, da die operandenverarbeitenden Befehle den Hauptspeicher nicht adressieren können und somit Speicheradressen in diesen Befehlen nicht vorkommen.

2. Lade- und Speichere-Befehle: Sie greifen sowohl auf den Registersatz als auch auf den Speicher zu. Zur Bildung einer Speicheradresse gibt es für sie zwei Adressierungsarten, die beide auf der registerindirekten Adressierung aufbauen.

- c) *Registerindirekte Adressierung.* Die effektive Adresse wird aus einer in einem Register stehenden Speicheradresse gebildet, zu der ein kurzes Displacement als 2-Komplement-Zahl vorzeichenerweitert addiert wird (Bild 2-27c); das Displacement wird ggf. mit 0 angegeben. Die Adresse ist eine Operandenadresse.

Assemblerschreibweise: ri,imm

- d) *Indizierte Adressierung.* Die effektive Adresse wird aus einer in einem Register r_i stehenden Speicheradresse gebildet, zu der ein in einem zweiten Register r_j stehender Index als 2-Komplement-Zahl addiert wird (Bild 2-27d). Die Adresse ist eine Operandenadresse.

Assemblerschreibweise: r_i,r_j

Anmerkung. Bei den CISC-Adressierungsarten in 2.1.3 wurden die Registeradressen bei registerindirekter und indizierter Adressierung zur Unterscheidung von der Registeradressierung in runde Klammern gesetzt. Bei RISCs ist dies nicht erforderlich, da sich die Unterscheidung aus der Befehlsart ergibt: Lade- und Speichere-Befehle bzw. arithmetische, logische und Shiftbefehle. So adressiert z.B. der Lade-Befehl `ld r0,0,r1` mit $r_0,0$ registerindirekt den Speicher (Displacement = 0!) und lädt den dort stehenden Operanden nach r_1 ; die Umkehrung der Operation erhält man mit dem Store-Befehl `st r1,r0,0`. Der Addiere-Befehl `add r0,2,r1` hingegen speichert den um 2 erhöhten Inhalt von r_0 nach r_1 .

3. Branch-Befehle: Bei ihnen wird die Sprungzieladresse relativ zum Befehlszähler gebildet.

- e) *Befehlszählerrelative Adressierung.* Die effektive Adresse wird aus dem Wert des nPC gebildet, zu dem ein kurzes Displacement vorzeichenverweitert addiert wird. Das Displacement wird zuvor durch Multiplikation mit 4 (Shiftoperation) um zwei niedrigstwertige Null-Bits zur Wortadresse ergänzt (Bild 2-27e). Die Adresse ist eine Programmadresse (Laden des nPC).

Assemblerschreibweise: `label`

4. Call-Befehl: Bei ihm wird die Sprungzieladresse als absolute Wortadresse angegeben.

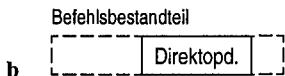
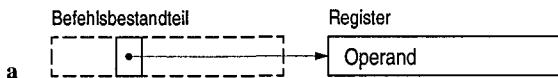
- f) *Absolute Adressierung.* Die effektive Adresse wird aus einer 30-Bit-Adresse gebildet, die durch Multiplikation mit 4 (Shiftoperation) um zwei niedrigstwertige Null-Bits zur Wortadresse ergänzt wird (Bild 2-27f). Die Adresse ist eine Programmadresse (Laden des nPC). – Die absolute Adressierung beim call-Befehl ist eine Ausnahme. Sie erlaubt es, den gesamten Adressraum von 4 G Adressen unmittelbar zu adressieren. Für den Operationscode des call-Befehls bleiben dabei nur noch 2 Bits übrig.

Assemblerschreibweise: `label`

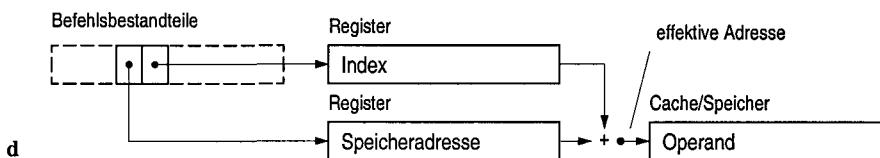
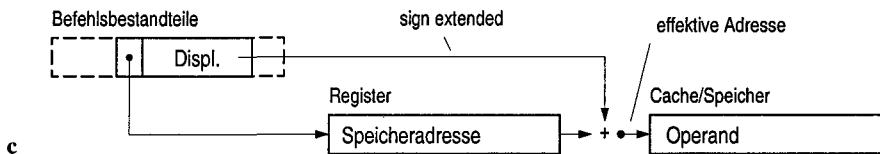
Anmerkung. Speicheradressen, die in Registern bereitgestellt werden, müssen dort zuvor mittels des Befehls `sethi` und des Pseudobefehls `setlo` zusammengesetzt werden (siehe dazu auch die Befehlsbeschreibungen in 2.2.4 und die Verwendung der Befehle in 3.4.1).

Befehlsformate. Auch bei RISCs gibt es mehrere Befehlsformate, jedoch im Unterschied zu CISCs mit einheitlichen Befehslängen von 32 Bits, d.h. wortorientiert. Drei Beispiele hierfür sind in Bild 2-28 wiedergegeben. Teilbild a zeigt ein für arithmetische und logische Befehle typisches *Dreiaußerbefehlsformat* mit dem Registerspeicher als Quelle (r_i, r_j) und Ziel (r_k). Dieses Format wird aber genauso von den Lade- und Speichere-Befehlen verwendet, wobei r_i für die regi-

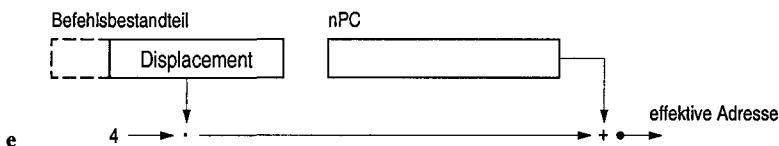
Arithmetische und logische Befehle



Lade- und Speichere-Befehle



Branch-Befehle



Call-Befehl

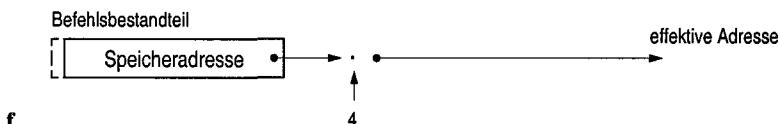


Bild 2-27. Adressierungsarten. a Register-Adressierung, b Direktoperand-Adressierung, c registerindirekte Adressierung, d indizierte Adressierung, e befehlzählerrelative Adressierung, f absolute Adressierung

sterindirekte Adressierung und rj als Indexregister benutzt wird. rk ist Ziel bzw. Quelle der Operation. Teilbild b zeigt eine Modifikation dieses Formats mit einem 13-Bit-Direktoperanden als zweiten Quelloperanden bzw. einem 13-Bit-Displacement für die registerindirekte Adressierung (imm). Unterschieden wer-

den die beiden Formate durch das Bit i. Teilbild c zeigt das Format für den sethi-Befehl. Es sieht ein größeres Konstantenfeld für die Aufnahme der oberen 22 Bits der Adresse addr vor und gibt mit rk das Register an, in dem die 32-Bit-Adresse „zusammengebaut“ wird. – Neben diesen drei Formaten gibt es zwei weitere Befehlsformate für die Branch-Befehle und den call-Befehl. Mit dem call-Format erklärt sich die Aufteilung des Operationscodes in ein 2-Bit-Feld und ein weiteres Feld, indem der call-Befehl nur einen 2-Bit-Operationscode aufweist, um für eine 30-Bit-Adresse Platz zu haben.

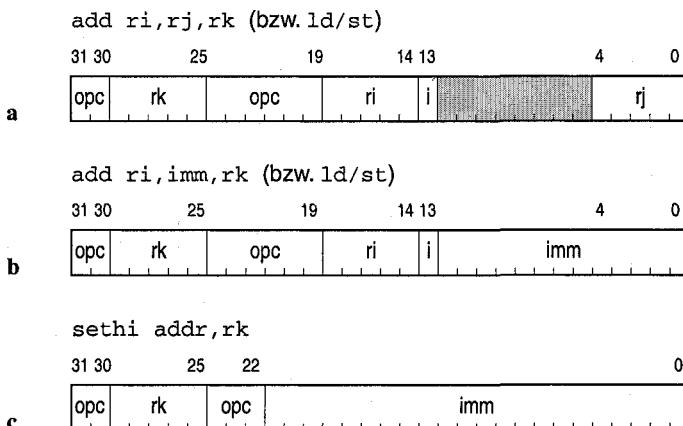


Bild 2-28. Befehlsformate in Anlehnung an die SPARC-Architektur

2.2.4 Befehlssatz

Gemessen an CISCs mit Befehlssätzen im Umfang von üblicherweise weit über 100 Befehlen haben RISCs kleinere Befehlssätze mit bis zu 60 oder 70 Befehlen (Coprozessorbefehle nicht eingerechnet). Viele dieser Befehle sind außerdem Varianten einiger elementarer Befehle, so daß die eigentliche Befehlsvielfalt als noch geringer anzusehen ist. Einen Überblick über die wichtigsten RISC-Befehle geben die beiden folgenden Tabellen 2-5 und 2-6 in Form eines Subsets des *SPARC-Befehlssatzes*. Bei ihm kommt zu den sonst RISC-typischen Befehlen die Besonderheit der Unterstützung der in 2.2.1 beschriebenen Fenstertechnik der SPARC-Architektur hinzu. Unterteilt ist der Subset in folgende Befehlsguppen:

- Lade- und Speichere-Befehle,
- arithmetische, logische und Shiftbefehle,
- Spungbefehle,
- Unterprogrammbefehle,
- Trap-Befehle.

In den formalen Befehlsbeschreibungen haben wir bei einigen Befehlen die Funktion gegenüber den SPARC-Befehlen der besseren Übersicht halber reduziert dargestellt (z. B. bei save und restore). Ebenso haben wir auf die Angaben für das Verändern der beiden Befehlszähler PC und nPC verzichtet, mit Ausnahme bei den Befehlen, bei denen von der Standardsequenz $nPC \rightarrow PC$, $nPC+4 \rightarrow nPC$ abgewichen wird. Bei den Angaben zu den Adressierungsarten verwenden wir die Symbolik „r“ für die Register des Registerspeichers, „m“ für die Speicheradressierung (registerindirekt ggf. mit Displacement oder indiziert), „addr“ für eine absolute Adresse und „label“ für eine Programmadresse (PC-relative bei Branch-Befehlen und absolut beim call-Befehl). „rj/imm“ zeigt bei den arithmetischen, logischen und Shiftbefehlen die Wahl zwischen einem Register- und einem Direktooperanden an. – Programmbeispiele mit diesem Befehlssatz sind in Abschnitt 3.4 im Zusammenhang mit der Unterprogrammtechnik und mit dem Anschluß von Unterbrechungsroutinen angegeben.

Tabelle 2-5. Lade-/Speichere-, arithmetische, logische und Shiftbefehle (RISC-Befehlssatz)

Befehl	Bezeichnung	Funktion
ld m,r	load word	$m \rightarrow r$
st r,m	store word	$r \rightarrow m$
swap m,r	swap r register with memory	$m \leftrightarrow r$ (nicht unterbrechbar)
sethi addr,r	set higher 22 bit of register	$addr<31-10> \rightarrow r<31-10>$ $0 \rightarrow r<9-0>$
rd sreg,r	read special register	$sreg \rightarrow r$
wr ri,rj/imm,sreg	write to special register	$ri \text{ xor } rj/\text{imm} \rightarrow sreg$
add ri,rj/imm,rk	add	$ri + rj/\text{imm} \rightarrow rk$
sub ri,rj/imm,rk	subtract	$ri - rj/\text{imm} \rightarrow rk$
mulscce ri,rj/imm,rk	multiply step and modify cc	(siehe Text)
and ri,rj/imm,rk	and	$ri \text{ and } rj/\text{imm} \rightarrow rk$
or ri,rj/imm,rk	inclusive-or	$ri \text{ or } rj/\text{imm} \rightarrow rk$
xor ri,rj/imm,rk	exclusive-or	$ri \text{ xor } rj/\text{imm} \rightarrow rk$
sra ri,rj/imm,rk	shift right arithmetic	$ri \cdot 2^{-rj/\text{imm}} \rightarrow rk$
sll ri,rj/imm,rk	shift left logical	$ri \cdot 2^{rj/\text{imm}} \rightarrow rk$

Lade- und Speichere-Befehle. ld m,r und st r,m stehen stellvertretend für eine Reihe von Lade- und Speichere-Befehlen, hier für den Transport von Operanden im Wortformat (32 Bits). Adressiert wird einerseits der Hauptspeicher (m: ri,rj oder ri,imm) und andererseits der Registerspeicher (rk). Die nicht gezeigten Befehle berücksichtigen die Datenformate Byte, Halbwort und Doppelwort, wobei die Byte- und Halbwort-Ladebefehle sich noch hinsichtlich der Zusatzfunk-

tion Zero- und Sign-Extension unterscheiden. swap m,r führt einen Worttausch zwischen einem Register des Registerspeichers und einer Speicherzelle des Hauptspeichers aus, ohne daß die Operation zwischen den beiden Speicherzugriffen unterbrochen werden kann. Somit kann er zur Realisierung von Synchronisationsvariablen (*Semaphoren*) in Mehrprozessorsystemen eingesetzt werden.

sethi addr,r unterstützt zusammen mit dem logischen Befehl or die registerindirekte Adressierung, indem mit beiden Befehlen eine 32-Bit-Adresse addr in einem Register r zusammengesetzt werden kann. sethi lädt dazu die 22 oberen Registerbits mit den oberen 22 Bits der Adresse als Direktoperand und setzt die unteren 10 Bits auf Null. Der or-Befehl überschreibt die unteren 10 Bits des Registers durch Oder-Bildung des Registerinhalts mit den unteren 10 Bits der Adresse als Konstante (sign-extended). Die Extraktion dieser unteren 10 Bits erfolgt durch einen Assembleroperator %lo (extract lower part). Die Wirkung des or-Befehls, zusammen mit dem Operator %lo lässt sich durch einen Pseudobefehl setlo, wie nachfolgend gezeigt, ausdrücken (siehe auch 3.4.1).

```
sethi var,10          → sethi var,10
or    10,%lo(var),10   setlo var,10
```

Die beiden speziellen Transportbefehle rd sreg,r und wr ri,rj/imm,sreg erlauben das Lesen und Schreiben spezieller Prozessorregister, z. B. des Statusregisters SR. Bei wr ist zu beachten, daß der zu schreibende Operand durch die Exklusiv-oder-Verknüpfung von ri mit rj/imm gebildet wird. Außerdem sollten auf wr drei nop-Befehle folgen, damit der wr-Befehl das gesamte 4-stufige Fließband durchlaufen kann, bevor ein nachfolgender Befehl von dessen Wirkung Gebrauch macht.

Arithmetische, logische und Shiftbefehle. add, sub, and, or und xor haben das für arithmetische und logische Befehle typische Dreiaußendreieckformat ri,rj/imm,rk, wobei der Registerspeicher als Quelle bzw. Ziel der Operanden fungiert. Wahlweise kann der zweite Quelloperand auch als Direktoperand imm vorgegeben werden. Varianten dieser Befehle sind mit dem Zusatz cc versehen, z. B. subcc, wodurch die arithmetisch-logische Einheit veranlaßt wird, die Condition-Code-Bits CC im Statusregister gemäß dem Ergebnis der Operation zu beeinflussen. Einen Vergleichsbefehl cmp gibt es als Maschinenbefehl nicht, jedoch als Pseudobefehl. Dieser wird durch subcc nachgebildet, indem als Zielregister jenes Register angegeben wird (beim SPARC das globale Register g0), das beim Lesen die Konstante Null liefert und beim Schreiben keine Wirkung zeigt. – Die Befehlsvariante ohne cc ist wichtig, um den Programmfluß durch das Umstellen von Befehlen optimieren zu können.

Der Befehl mulscC unterstützt die schrittweise Durchführung der Multiplikation zur Bildung eines 64-Bit-Produktes, vorzeichenlos oder vorzeichenbehaftet (Bildung des Teilprodukts 32-Bit-Multiplikand mal 1 Bit des Multiplikators). Hierbei wird zusätzlich zu den Quell- und Zieladresangaben im Adreßteil des Befehls ein Spezialregister des Prozessors benutzt. – Bei den Shiftbefehlen wird, wie sonst

auch, zwischen logischen und arithmetischen Links- und Rechtsshifts unterscheiden. Hier fehlt jedoch die Mnemonik für den arithmetischen Linksshift; er ist mit dem logischen Linksshift identisch, da bei beiden Shiftarten die Bedingungsbits nicht beeinflußt werden.

Tabelle 2-6. Sprung-, Unterprogramm- und Trap-Befehle (RISC-Befehlssatz)

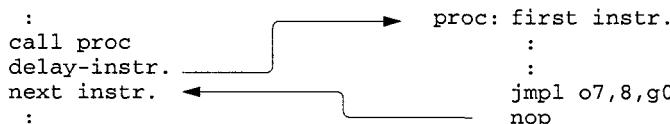
Befehl	Bezeichnung	Funktion
ba label	branch always	effektive Adresse von label \rightarrow nPC
bcc label	conditional branch	if $cc\text{-test} = \text{true}$ then eff. Addr. von label \rightarrow nPC else nächster Befehl (cc wie bei CISCs, siehe Tabelle 2-2)
jmpl m,r	jump and link	$PC \rightarrow r$ $nPC \rightarrow PC$; effektive Adresse von m \rightarrow nPC
call label	call	$PC \rightarrow o7$ effektive Adresse von label \rightarrow nPC
save	save caller's window	$CWP - 1 \rightarrow CWP$ (Fensterumschaltung) if Overflow then Window-overflow-Trap
restore	restore caller's window	$CWP + 1 \rightarrow CWP$ (Fensterumschaltung) if Underflow then Window-underflow-Trap
ta m	trap always	$0 \rightarrow ET, S \rightarrow PS, CWP - 1 \rightarrow CWP$ if Overflow then Window-overflow-Trap $PC \rightarrow l1, nPC \rightarrow l2, 1 \rightarrow S$ Unterbrechungsvektor (abh. von m) \rightarrow nPC
tcc m	trap on condition codes	if $cc\text{-test} = \text{true}$ then $0 \rightarrow ET, S \rightarrow PS, CWP - 1 \rightarrow CWP$ if Overflow then Window-overflow-Trap $PC \rightarrow l1, nPC \rightarrow l2, 1 \rightarrow S$ Unterbrechungsvektor (abh. von m) \rightarrow nPC (cc wie bei CISCs, siehe Tabelle 2-2)
rett m	return from trap	$1 \rightarrow ET$ effektive Adresse von m \rightarrow nPC $CWP + 1 \rightarrow CWP$ if Underflow then Window-underflow-Trap $PS \rightarrow S$ (privilegiert)

Sprungbefehle. ba label ist ein unbedingter Sprungbefehl, bcc label steht für eine Gruppe *bedingter Sprungbefehle* (Tabelle 2-6). Die durch cc bezeichneten Sprungbedingungen sind dieselben wie bei dem in 2.1.4 beschriebenen CISC-Befehlssatz (Tabelle 2-2, S. 93). Sie haben beim SPARC jedoch eine modifizierte

Mnemonik; außerdem ist zu beachten, daß die Bedingungsbits hier durch ri – rj gebildet werden, beim CISC-Befehlssatz durch dst – src. Aus der Sprungadresse label ermittelt der Assembler das Displacement für die befehlzählerrelative Adressierung. Beide Befehle haben einen *Delay-Slot* und können wahlweise durch ein *Annullierungsbit* a erweitert werden. Dieses legt fest, ob der auf einen Sprungbefehl folgende Befehl, d.h. der Befehl in seinem Delay-Slot (Delay-Befehl) ausgeführt werden soll oder nicht. Ist dieses Bit bei einem bedingten Sprungbefehl gesetzt (bcc.a), so wird der im Delay-Slot stehende Befehl nur dann annulliert, wenn die Sprungbedingung nicht erfüllt ist, d.h. wenn der Sprung nicht ausgeführt wird. Ist dieses Bit beim *unbedingten Sprungbefehl* gesetzt (ba.a), so wird im Gegensatz zum bedingten Sprung der im Delay-Slot stehende Befehl trotz des Sprungs annulliert. Delay-Slot und Annullierungsbit werden zur Optimierung des Programmflusses benutzt (siehe 1.4.2).

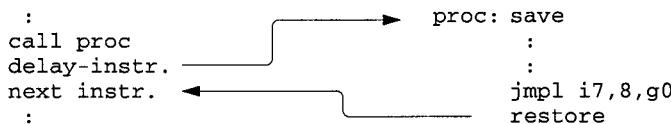
Unterprogrammbefehle. jmp m,r ist ein Sprungbefehl mit ebenfalls einem Delay-Slot, bei dem das Sprungziel als allgemeine Speicheradresse m (m: ri,rj oder ri,imm) vorgegeben werden kann, und der seine eigene Befehlsadresse als Rücksprungadresse in einem vorgebbaren Register r speichert (*link register*). Er wird in dieser Funktion sowohl für den Aufruf von Unterprogrammen als auch als für den Rücksprung aus diesen benutzt. Im letzteren Fall wird für r das Register g0 angegeben. call label, ebenfalls mit Delay-Slot, vereinfacht den Aufruf von Unterprogrammen. Er speichert seine Befehlsadresse als Rücksprungadresse in o7. Wird im Unterprogramm ein Fensterwechsel vorgenommen, so steht die Rücksprungadresse im neuen Fenster in i7 zur Verfügung. Die folgende Darstellung zeigt den Unterprogrammanschluß mit call und jmp *ohne* Fensterumschaltung (*leaf routine*). Der Pfeil zum Oberprogramm hin gibt die Rücksprungstelle an. Sie ergibt sich aus der Rücksprungadresse, die auf call zeigt, erhöht um den Wert 8 (hier als Direktoperand), um den call- und den Delay-Slot-Befehl zu überspringen.

Leaf-Routine:



save und restore führen die Fensterumschaltung auf das nächste bzw. das vorangegehende Fenster durch. save wird dazu im Unterprogramm als erster, restore als letzter Befehl (im Delay-Slot des Rücksprungbefehls jmp1) ausgeführt. Ein dabei auftretender Window-Overflow bzw. Window-Underflow löst einen Trap aus. Die folgende Darstellung zeigt den Unterprogrammanschluß *mit* Fensterumschaltung (*window routine*). Die von call in o7 gespeicherte Rücksprungadresse ist im Unterprogramm in i7 verfügbar.

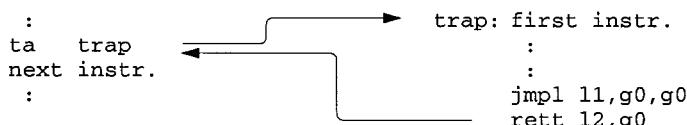
Window-Routine:



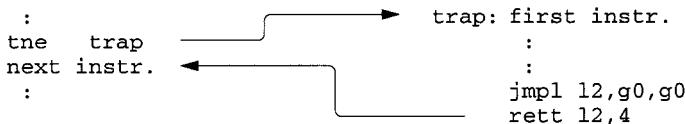
Trap-Befehle. ta m ist ein unbedingter Trap-Befehl, tcc label steht für eine Gruppe bedingter Trap-Befehle, die mit cc dieselben Bedingungen für eine Trap-Auslösung aufweisen wie die bedingten Sprungbefehle für eine Verzweigung. Beide Befehle führen als Programmunterbrechungen automatisch zu einer Fensterumschaltung. Sie retten dabei den PC und den nPC in die beiden lokalen Register l1 und l2 des neuen Fensters (Trap-Fenster) und führen dann den Sprung zur Trap-Routine durch. Dieser Sprung erfolgt über eine sog. *Trap-Tabelle*, in der für jede Art von Programmunterbrechung (Traps und Interrupts) eine Sequenz von vier Befehlen steht, mittels der zum eigentlichen Unterbrechungsprogramm verzweigt wird (siehe dazu 2.2.5). Für die Anbindung von Trap-Routinen der Befehle ta und tcc sieht die Trap-Tabelle insgesamt 128 Positionen vor. Der Einsprung in die Tabelle geschieht mittels der Angabe m im Trap-Befehl (ri,rj oder ri,imm), die als Tabellenindex ausgewertet wird. – ta und tcc haben keinen Delay-Slot.

rett m führt als letzter Befehl eines Unterbrechungsprogramms die Statusumschaltung einschließlich der Fensterumschaltung durch. Der eigentliche Rücksprung erfolgt jedoch durch einen vorangehenden jmpl-Befehl, in dessen Delay-Slot der rett-Befehl steht. Abhängig von den Adressangaben in den beiden Befehlen kann festgelegt werden, ob ein Befehl, der zum Aufruf einer Trap-Routine geführt hat, nach dem Rücksprung noch einmal ausgeführt wird oder nicht. Diese Unterscheidung ist vor allem bei Befehlen mit impliziter Trap-Auslösung wichtig, z.B. bei save und restore im Falle eines Window-Over-/Underflows, oder bei den Lade- und Speichere-Befehlen im Falle eines Page-Faults oder bei einem Page-Fault während des Befehlsabrufs. Die mögliche Wiederholung eines trap-auslösenden Befehls ist auch der Grund dafür, daß die Trap-Befehle selbst keinen Delay-Slot haben, da sonst im Wiederholungsfall auch der Delay-Slot-Befehl nochmals ausgeführt würde. Dazu nachfolgend die schematische Darstellung beider Möglichkeiten, wobei die Trap-Auslösungen hier der Deutlichkeit halber durch die Trap-Befehle ta (unbedingt) und tne (bedingt), d.h. explizit erfolgen; der auf das Aufrufprogramm zeigende Pfeil gibt dabei die jeweilige Rücksprungstelle an.

Trap-auslösender Befehl wird wiederholt:



Trap-auslösender Befehl wird nicht wiederholt:



Anmerkung. Ein Beispiel für die Wiederholung des trap-auslösenden Befehls ta (genauer: eines Befehls, der nachträglich unter der Befehlsadresse von ta gespeichert wird) ist die Realisierung des Testmodus *Single-Step* in einem sog. *Debugger-Programm*. Ein solches Programm bewirkt, daß ein zu testendes Programm nach jeder Befehlausführung unterbrochen wird, um z.B. den Prozessor-/Programmstatus auf dem Bildschirm anzuzeigen. Es ersetzt dazu den auf die Unterbrechungsstelle folgenden Originalbefehl durch den Trap-Befehl ta. In der durch ta aufgerufenen Trap-Routine wird dann der Status ermittelt und auf dem Bildschirm angezeigt. Darüber hinaus wird durch die Trap-Routine der Originalbefehl wieder an die Stelle von ta gespeichert und der Befehl ta zur Vorbereitung der nächsten Unterbrechung an die Stelle des nächsten Befehls gesetzt. Die Rückkehr aus der Trap-Routine ist, wie im ersten Programmbeispiel oben gezeigt, so organisiert, daß der zunächst entfernte Originalbefehl nun doch noch ausgeführt wird. – Bei CISCs mit hardware-unterstützter Single-step-Funktion entfällt diese etwas umständliche Programmierung (siehe 2.1.1: Trace-Bits im Statusregister). Hier wird die Programmunterbrechung bei im Statusregister gesetzter Trace-Funktion nach jedem Befehl automatisch erzeugt (trace trap).

2.2.5 Unterbrechungssystem und Betriebsarten

Wie CISCs sehen auch RISCs zwei oder mehr Betriebsarten vor, z.B. User und Supervisor. Eine Programmunterbrechung führt immer in den privilegierten Modus, d.h. in den *Supervisor-Modus*. Gemessen an CISCs gibt es weniger Unterstützung durch die Prozessorhardware, was durch Software ausgeglichen werden muß. So erfolgt z.B. beim SPARC kein automatisches Retten des Statusregisters auf einen Stack, da die Hardwareunterstützung für eine Stackverwaltung fehlt. Einer Interruptanforderung wird zwar wie bei CISCs nur dann stattgegeben, wenn die Ebene der Anforderung höher ist als die durch die Interruptmaske angezeigte aktuelle Ebene, es wird jedoch nicht automatisch die Interruptmaske im Statusregister mit dem Wert der Anforderung überschrieben (um Anforderungen gleicher und niedrigerer Priorität zu blockieren). Statt dessen wird z.B. lediglich das Statusbit für die Betriebsart innerhalb des Statusregisters gerettet, und Unterbrechungen werden pauschal blockiert. Das heißt, das Priorisieren von Programmunterbrechungen ist nur dann möglich, wenn das Retten, Verändern und Wiederherstellen des Statusregisters in der Software verankert ist.

Mit dem in 2.2.1 definierten Statusregister (Bild 2-26, S. 109) und dem in 2.2.4 vorgestellten Befehlssatz lauten die Aktionen des Prozessors bei einer Programmunterbrechung – in Anlehnung an die SPARC-Architektur – wie folgt:

- Blockieren des Unterbrechungssystems: 0 → ET,
- Retten des Status der alten Betriebsart: S → PS,

- Aktivieren des neuen Fensters: CWP-1 → CWP,
- Retten der Befehlszählerstände: PC → l1, nPC → l2,
- Umschalten in den Supervisor-Modus: 1 → S,
- Sprung in das Unterbrechungsprogramm.

Der Sprung in das Unterbrechungsprogramm erfolgt, wie in 2.2.4 für die Trap-Befehle beschrieben, durch automatisches Verzweigen in die Trap-Tabelle, in der zu jeder Unterbrechungsbedingung eine Sequenz von vier Befehlen gespeichert ist. Sie besteht aus den beiden Befehlen sethi und setlo für das Zusammensetzen der Startadresse des Unterbrechungsprogramms in einem der lokalen Register li (nicht l1 und l2), dem Sprungbefehl jmpl und einem nop-Befehl als Delay-Slot-Befehl von jmpl, wie nachfolgend für ein Unterbrechungsprogramm mit der Startadresse timer gezeigt.

```
sethi timer,10
setlo timer,10
jmpl 10,g0,g0
nop
```

Für den Rücksprung aus einem Unterbrechungsprogramm werden, wie in 2.2.4 beschrieben, die beiden Befehle jmpl und rett aufeinanderfolgend ausgeführt. rett bewirkt dabei u.a. die Freigabe von Unterbrechungen (1 → ET), die Fensterumschaltung (CWP+1 → CWP) sowie das Wiederherstellen der ursprünglichen Betriebsart (PS → S). – Für eine ausführliche Beschreibung der Programmierung von Programmunterbrechungen beim SPARC sei auf 3.4.3 verwiesen.

2.3 Moderne Prozessorarchitekturen mit parallel arbeitenden Funktionseinheiten

Die in Kapitel 1 und bisher in diesem Kapitel beschriebenen Prozessoren können als *skalare Prozessoren* bezeichnet werden, da bei ihnen mit Ausführung eines arithmetischen oder logischen Befehls genau eine arithmetische bzw. logische Operation einhergeht. Der Prozessor kann dabei mikroprogrammiert sein, wie das beispielhaft für einen einfachen CISC in Bild 1-11 (S. 27) gezeigt ist, und er kann – wie bei heutigen CISC- und RISC-Skalarprozessoren höherer Leistung üblich – in Fließbandtechnik ausgelegt sein, wie dies beispielhaft für einen RISC in Bild 1-24 (S. 51) gezeigt ist. Mit dem Übergang zur *Fließbandtechnik* erreicht man eine sog. sequentielle Parallelisierung der Befehlausführungen, mit der sich der Durchsatz an Befehlen erhöht. Der Idealfall, daß hierbei mit jedem Takt ein Befehl abgeschlossen werden kann, setzt allerdings voraus, daß die Aktionen einer jeden Fließbandstufe innerhalb eines Taktes zu erledigen sind. Dem entgegen wirken die *Datenabhängigkeit* und die *Sprungabhängigkeit*, wie in 1.4.2 beschrieben, und die sog. *Betriebsmittel-Abhängigkeit*.

Eine weitere Erhöhung des Befehlsdurchsatzes erreicht man dadurch, daß man die Verarbeitung im Prozessor durch den Einbau zusätzlicher Funktionseinheiten noch mehr parallelisiert. Hier gibt es unterschiedliche Organisationsformen, wie *Coprozessorsysteme*, *superskalare* und *VLIW-Prozessoren*. Um diese auf der Maschinenebene vorhandene Parallelisierung nutzen zu können, muß, wie schon bei skalaren Fließbandprozessoren, eine entsprechend hohe Parallelität auf der Befehlsebene vorhanden sein (instruction-level parallelism, ILP), d.h., zwei oder mehr aufeinanderfolgende Befehle des sequentiellen Programms müssen konfliktfrei den Verarbeitungseinheiten parallel zuordenbar und ausführbar sein. Dem entgegen stehen wieder dieselben Abhängigkeiten wie beim skalaren Fließbandprozessor, die sich jedoch auf die Parallelarbeit der Funktionseinheiten verstärkt auswirken.

Heutige Prozessoren begegnen diesem Problem, indem sie Befehle im Vorgriff und ggf. auch spekulativ ausführen. So wird bei Daten- oder Betriebsmittel-Abhängigkeiten die Verarbeitung mit jenen Folgebefehlen im Vorgriff fortgesetzt, die keine Abhängigkeiten aufweisen. Dies entspricht einem Umordnen der Befehlsreihenfolge, was als *Out-of-order-Issue/Execution* bezeichnet wird (wobei allerdings zwei weitere Arten der Datenabhängigkeit auftreten können). Dieses Umordnen erfolgt entweder dynamisch, d.h. zur Programmausführungszeit durch die Prozessorhardware, wie dies bei den superskalaren Prozessoren der Fall ist, oder statisch, d.h. zur Übersetzungszeit durch den Compiler, wie bei den VLIW-Prozessoren. Bei Sprungabhängigkeiten wird der wahrscheinlichere der auszuführenden Programmfade vorhergesagt, um dann dessen Befehle im Vorgriff spekulativ auszuführen. Diese *Vorhersage* wird zum Zeitpunkt der eigentlichen Sprungentscheidung, d.h. mit Ausführung des Sprungbefehls überprüft und ggf. korrigiert. Alternativ dazu gibt es die Möglichkeit der *Sprungvermeidung*. Hier verzichtet man ggf. auf bedingte Sprungbefehle und versieht stattdessen die Befehle beider Programmzweige mit Bedingungen, anhand derer entschieden wird, ob das Ergebnis einer Operation letztlich verwendet oder verworfen wird. Man spricht hier von *Prädikation*. Eine leistungsfähige Sprungvorhersage bzw. die Sprungvermeidung sind bei hochgetakteten Prozessoren unabdingbar, da deren Fließbänder bis zu 20 Stufen und mehr aufweisen und dadurch Fehlvorhersagen bei Sprüngen entsprechend viele Takte Verlust bedeuten. Man spricht bei der Befehlsverarbeitung mittels solch langer Fließbänder auch von *Superpipelining*.

Die im sequentiellen Programmfluß vorhandene Befehlsparallelität (IPL) reicht üblicherweise nicht aus, alle Funktionseinheiten (Fließbänder) eines Prozessors vollständigen auszulasten. Eine bessere Nutzung erreicht man hier, indem man die Funktionseinheiten mit Befehlen aus zwei oder mehr von einander unabhängigen Befehlssequenzen des Programms, sog. *Kontrollfäden (threads)*, versorgt. Man bezeichnet dieses Vorgehen als *vielfältige Verarbeitung (multithreading)* und spricht von Parallelität auf der Kontrollfadenebene (*thread-level parallelism, TLP*).

Die Ausführungen dieses Abschnitts werden durch Beispiele realer Prozessoren begleitet. Diese sollen nicht nur den Stoff vertiefen, sondern darüber hinaus einen Einblick in konkrete Prozessorstrukturen geben. Mit einbezogen sind dabei auch Strukturen von Vorgängern aktueller Prozessoren, um die Prozessorentwicklung anzudeuten. Bei den Strukturbeschreibungen lassen sich Vorgriffe auf Techniken und Zusammenhänge, die erst später in diesem Buch erläutert werden, nicht vermeiden. – Einen Einblick in die Thematik dieses Abschnitts geben [Bode 2002], [Ungerer 2001], und [Ungerer 1995]. Zur Vertiefung der grundlegenden Aspekte sei auf [Stallings 1996] und [Flynn 1995] verwiesen.

2.3.1 Coprozessorsysteme

Ausgehend von einem Skalarprozessor mit seiner arithmetisch-logischen Verarbeitungseinheit für die Ganz Zahlarithmetik lässt sich eine erhöhte Parallelisierung an Befehlsausführungen dadurch erreichen, daß man dem Skalarprozessor ein oder mehrere weitere Prozessoren zur Seite stellt, die *Coprozessoren* genannt werden. Dabei handelt es sich nicht um eigenständig programmierbare Prozessoren, sondern um an den Befehlstrom angehängte Prozessoren für spezielle Aufgaben. Das entspricht einer Erweiterung des Befehlssatzes, z.B. für die Ausführung von Gleitpunktbefehlen. Diese Prozessoren arbeiten mit Fließbandverarbeitung, sofern sich die Befehle gut in eine geringe, konstante Anzahl von Teilschritten zerlegen lassen (Beispiel: Gleitpunktaddition mit den Schritten Exponenten angleichen, Mantissen addieren, Ergebnis normalisieren), oder sie arbeiten mikroprogrammiert, wenn die Befehle eine größere oder variable Anzahl von Teilschritten beanspruchen (Beispiele: repetitive Ausführung der Multiplikation oder der Division für Ganzzahlen wie für Gleitpunktzahlen).

Bild 2-29 zeigt skizzenhaft ein solches Coprozessorsystem, aufbauend auf der RISC-Struktur Bild 1-24 (S. 51) in einer von Details abstrahierten Darstellung. Es besitzt eine Funktionseinheit FE1, z.B. eine Ganz Zahleinheit in der Form einer üblichen ALU, sowie eine Funktionseinheit FE2, z.B. für Multiplikation und Division von Ganzzahlen oder Gleitpunktzahlen. Ausgehend von der ursprünglichen externen Anordnung des Coprozessors ergibt sich die hierfür typische Programmabarbeitung: die Befehle werden nacheinander den Funktionseinheiten zugeordnet. Bei Funktionseinheiten, die in Fließbandtechnik arbeiten, heißt das, daß (ebenso wie bei einem Fließbandprozessor ohne Coprozessor) mit jedem Takt maximal ein Befehl der Verarbeitung zugeführt werden kann.

Von Vorteil ist hierbei die mögliche Parallelität in der „Ausführung“ von Befehlen, sofern diese mehr als einen Takt dafür benötigen. Voraussetzung ist natürlich, daß sich die Befehle nicht gegenseitig bedingen. Bild 2-30 demonstriert diesen Sachverhalt. Es zeigt zunächst in Teilbild a die Bearbeitung einer Befehlssequenz durch einen skalaren Fließbandprozessor, bei dem jede Befehlausführung nur einen Takt dauert und bei dem im dargestellten Idealfall deshalb mit jedem Takt

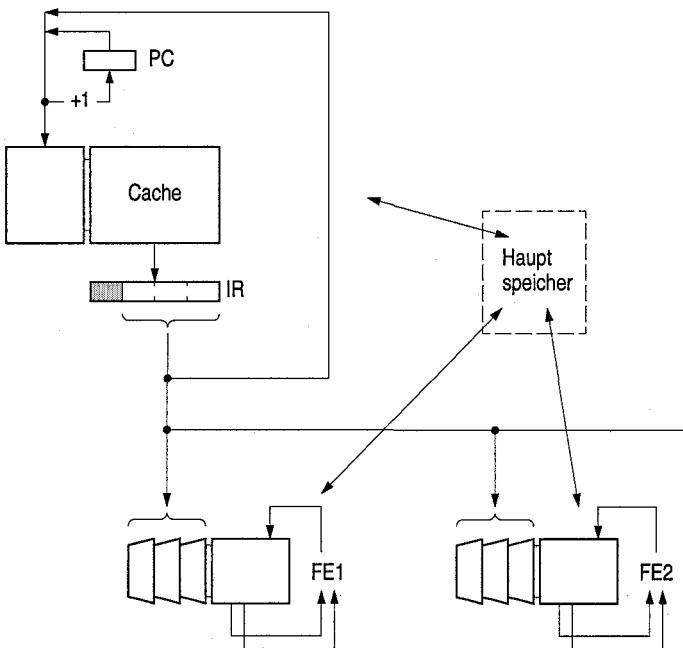


Bild 2-29. Schematische Darstellung eines Coprozessorsystems. FE1: Funktionseinheit z.B. ohne Steuerwerk, FE2: Funktionseinheit z.B. mit Steuerwerk. FE2 bildet zusammen mit ihrem Dreiport-Registerspeicher den Coprozessor. Ohne Angabe der Datenwege

auch ein Befehl ausgeführt bzw. abgeschlossen wird. Es zeigt in Teilbild b die Bearbeitung einer ähnlichen Befehlssequenz, jetzt aber für ein Coprozessorsystem entsprechend Bild 2-29. Hierbei werden die Befehle den beiden Funktionseinheiten FE1 und FE2 wiederum *nacheinander* zugeordnet, jedoch kann die vom Coprozessor ausgeführte Gleitpunktmultiplikation fmul (FE2), die hier drei Takte benötigt, von den beiden im Programm nachfolgenden Befehlen move und add (FE1) über- bzw. eingeholt werden. Das gleichzeitige Schreiben der Ergebnisse von fmul und add ist hierbei möglich, da der Coprozessor einen eigenen Registerspeicher hat. Dies hat allerdings zur Folge, daß Operanden zwischen den Registerspeichern transportiert werden müssen (der Datenweg hierfür ist im Bild der Übersichtlichkeit halber nicht gezeichnet).

Coprozessoren als eigenständige Bausteine wurden zunächst bei CISCs für Gleitpunktoperationen eingesetzt. Die CISC-Prozessoren wurden dafür eigens mit einer Coprozessorschnittstelle ausgestattet, die eine enge Kopplung beider Bausteine ermöglichte. Bei den RISCs, die gegenüber CISCs zunächst einfachere Struktur mehr Platz auf dem Prozessorbaustein hatten, wurde die Gleitpunkteinheit dann frühzeitig in den Prozessor als bausteininterne Funktioneinheit integriert. Die Coprozessorschnittstelle des Prozessors entfiel damit. Mit zunehmender Integrationsdichte wurde dieser Schritt dann auch bei den CISC-

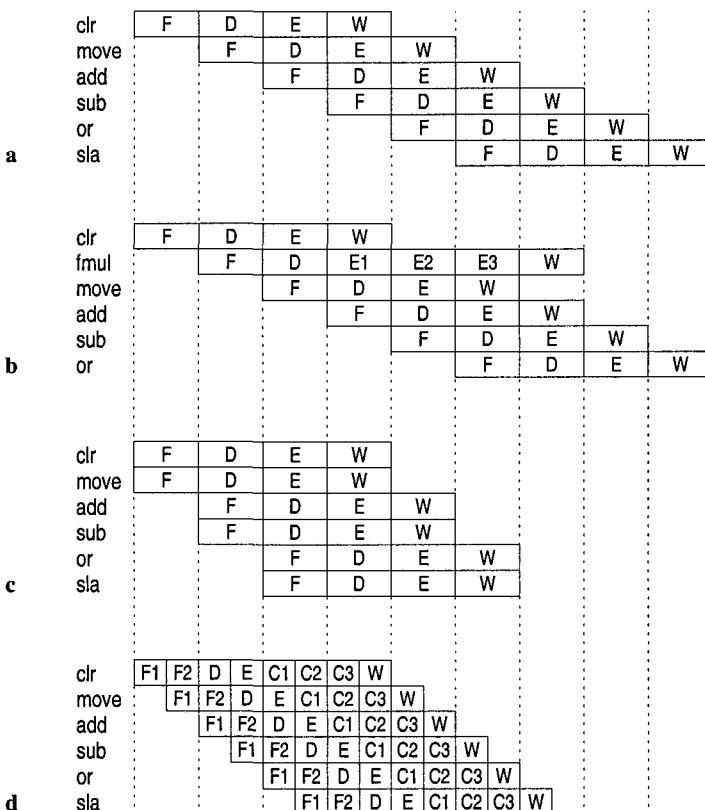


Bild 2-30. Maximal möglicher Befehlsdurchsatz bei Fließbandprozessoren. **a** Skalarer Prozessor, **b** Coprozessorsystem mit skalarem Prozessor (fmul als Coprozessorbefehl mit mehr als einem Takt Ausführungszeit), **c** superskalarer oder VLIW-Prozessor mit jeweils 2 parallel ausgeführten Befehlen, **d** Superpipeline-Prozessor mit gegenüber a doppelter Stufenzahl. Darstellung in Anlehnung an [Stallings 1996]

Prozessoren vollzogen. Heute hat das Coprozessor-Konzept eine relative geringe Bedeutung, da die Gleitpunkteinheit als der ursprünglich gebräuchlichste Coprozessor inzwischen fester Bestandteil von superskalaren und VLIW-Prozessoren ist und dort mit einem höheren Grad an Parallelität genutzt werden kann.

2.3.2 Superskalare Prozessoren

Ausgehend von unserer Betrachtung eines Coprozessorsystems mit zwei parallel arbeitenden Funktionseinheiten läßt sich eine bessere Nutzung dieser Einheiten dadurch erreichen, daß man ihnen zwei im Programm aufeinanderfolgende Befehle gleichzeitig, d.h. taktgleich zur Verarbeitung zuführt. Weisen die beiden Be-

fehle untereinander keine Abhängigkeiten auf, so können diese parallel ausgeführt werden. Im Gegensatz zu skalaren Prozessoren und zu Coprozessorsystemen können damit im konfliktfreien Fall auch zwei Befehle gleichzeitig fertiggestellt werden. Sind die Funktionseinheiten in Fließbandtechnik ausgeführt, so kann dies mit jedem Takt geschehen. Realisiert wird diese „echte“ Parallelität der Befehlsbearbeitung sowohl bei superskalaren als auch bei VLIW-Prozessoren, wenn auch in unterschiedlicher Weise: Bei den superskalaren Prozessoren wird die Entscheidung, welche Befehle taktgleich verarbeitet werden können zur Programmausführungszeit, d.h. von der Hardware getroffen, bei den VLIW-Prozessoren zur Übersetzungszeit, d.h. durch die Software. Bild 2-30c illustriert den bei beiden Prozessorvarianten verbesserten Durchsatz an Befehlen gegenüber einem Skalarprozessor (Teilbild a) oder einem Coprozessorsystem (Teilbild b).

Bild 2-31 zeigt wieder skizzenhaft die Struktur eines solchen superskalaren Prozessors, in Analogie zu Bild 2-29 ebenfalls von der RISC-Struktur Bild 1-24 (S. 51) in einer von Details abstrahierten Darstellung ausgehend. Angenommen

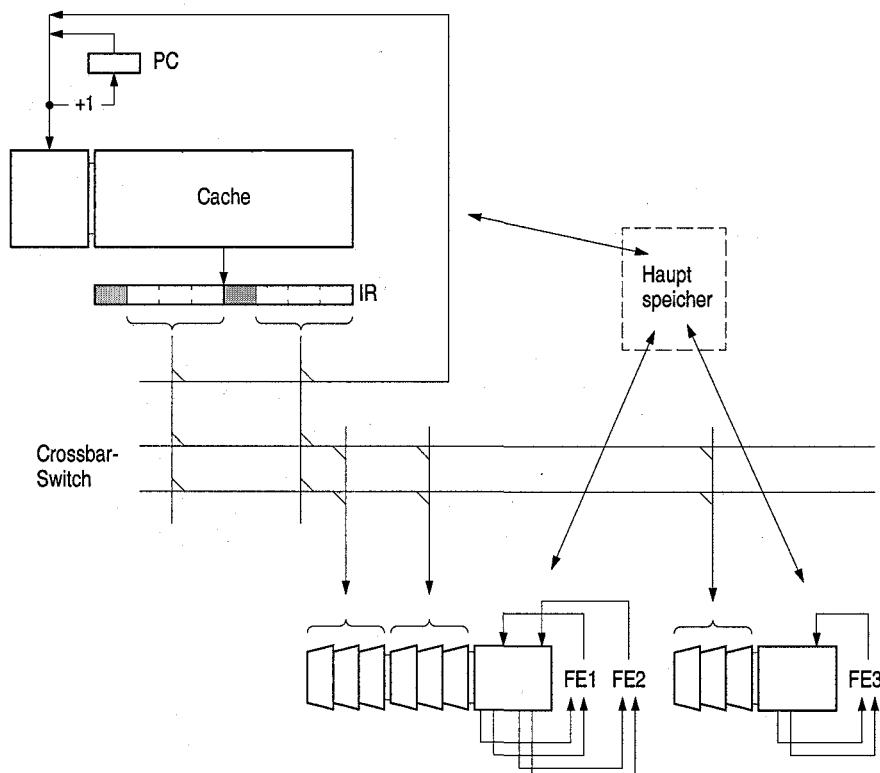


Bild 2-31. Schematische Darstellung eines superskalaren Prozessors. FE1, FE2: Funktionseinheiten z.B. ohne Steuerwerke, FE3: Funktionseinheit z.B. mit Steuerwerk. Ohne Angabe der Datenwege

sind hier drei Funktionseinheiten, z.B. zwei Ganzzahleinheiten und eine Gleitpunkteinheit. Anders als beim Coprozessorsystem müssen jetzt mit jedem Takt zwei aufeinanderfolgende Befehle gleichzeitig aus dem *Befehls-Cache* geholt werden, die natürlich dort „nebeneinander“ gespeichert vorliegen müssen. Das bedingt gegenüber Bild 2-29 eine Verdoppelung der Wortlänge des Cache wie auch eine Verdoppelung der Breite des Befehlsregisters, das z.B. die beiden Funktionseinheiten FE1 und FE2 parallel mit Befehlen versorgt. Die Funktionseinheiten haben nun entweder zwei *Dreiport-Registerspeicher* oder (wie im Bild gezeigt) einen Sechsport-Registerspeicher. Die *Crossbar*-Verbindungen jedes der beiden Befehlsregisterteile mit jeder der beiden FEs ermöglichen es, die Befehle so zu verteilen, daß beide Einheiten möglichst gleichmäßig ausgelastet sind. Die Hardware übernimmt dabei (und das ist der wesentliche Unterschied zu VLIW-Prozessoren) die Überprüfung von Abhängigkeiten und damit die Entscheidung, ob jeweils beide Befehle taktgleich den Funktionseinheiten zugeführt werden können. Man bezeichnet diese Hardwarekomponente auch als *Dispatch-Unit* (im Bild nicht extra ausgewiesen).

Wie beim Coprozessorsystem funktioniert also die Parallelarbeit der beiden Einheiten nur, solange die beiden Befehle sich nicht gegenseitig bedingen. Tritt dieser Fall auf, so muß auch hier das Holen des nächsten Befehlspaares verzögert werden. Eine Alternative dazu ist, ggf. nur den bereits abgearbeiteten Befehl im Befehlsregister mit dem nächsten Befehl aus dem Cache zu überschreiben. Das ist insbesondere auch dann sinnvoll, wenn weitere Funktionseinheiten, wie in Bild 2-31 die Funktionseinheit FE3, ins System integriert sind. FE3 ist hier in das Strukturbild des superskalaren Prozessors mit aufgenommen, um anzudeuten, daß superskalare Prozessoren (wie auch VLIW-Prozessoren) i.allg. mehr Funktionseinheiten haben als parallele Befehlsströme.

Als eigenständige Funktionseinheiten eines superskalaren Prozessors zählen auch die Einheit für Programmverzweigungen und die Einheit für das Laden und Speichern der Operanden, so sie selbstständig arbeiten. Letztere kann aber auch in die Ganzzahleinheit „integriert“ sein. Weiterhin brauchen die einzelnen Befehle nicht alle dieselbe Ausführungszeit zu haben. Zum Beispiel können Ganzzahleinheiten für ihre elementaren Befehle (RISC-typisch) einen Takt benötigen, während Gleitpunkteinheiten für ihre komplexen Befehle mehrere Takte benötigen, (RISC-typisch) fließbandbedingt oder (CISC-typisch) steuerwerksbedingt. – Beispiele für die Art und Anzahl unabhängig arbeitender Funktionseinheiten superskalarer Prozessoren finden sich in 2.3.3.

Ein Vorteil superskalarer Prozessoren ist, daß sie hinsichtlich ihrer Funktionseinheiten „skalierbar“ sind. Das heißt, Software, die auf einem Prozessor mit wenigen Funktionseinheiten läuft, im Extremfall ist das ein Skalarprozessor, ist auch auf einem Prozessor der Folgegeneration mit mehr Funktionseinheiten lauffähig. Das liegt daran, daß die mit der höheren Parallelisierung einhergehenden Probleme, z.B. das Umordnen von Befehlen und dessen Auswirkungen durch den

Prozessor, d.h. durch die Hardware gelöst werden. Dies gilt natürlich auch schon für die weniger leistungsfähigen Coprozessorsysteme.

2.3.3 Konflikte durch Abhängigkeiten

Wie in der Einleitung zu diesem Abschnitt bereits angedeutet, treten bei Prozessoren mit parallel arbeitenden Funktionseinheiten (mikroprogrammiert oder in Fließbandtechnik ausgelegt) zunächst dieselben Abhängigkeiten und damit dieselben Konflikte wie bei einem skalaren Fließbandprozessor auf. Hinzu kommen jedoch weitere Abhängigkeiten, wenn man, wie bei Coprozessoren, superskalaren und VLIW-Prozessoren üblich, die Befehlsreihenfolge verändert, um die Parallelarbeit der Funktionseinheiten möglichst gut zu nutzen. Dieses *Umordnen der Befehle* übernimmt bei Coprozessorsystemen und VLIW-Prozessoren der Übersetzer. Bei superskalaren Prozessoren erfolgt es durch den Prozessor, was allerdings eine vorangehende Optimierung durch den Übersetzer nicht ausschließt.

Bezüglich dieser Änderung der Befehlsreihenfolge gegenüber dem sequentiellen Programm, wie es vom Übersetzer für einen skalaren Prozessor erzeugt wird, sind zwei Phasen der Befehlsverarbeitung zu betrachten:

1. das Zuordnen der Befehle an die Funktionseinheiten, was im Englischen mit *Issue* bezeichnet wird, und
2. das Beenden einer Befehlausführung, d.h. das Schreiben des Resultats in das Zielregister, was im Englischen mit *Completion* bezeichnet wird.

Erfolgt die Befehlszuordnung oder die Befehlsbeendigung in der Reihenfolge des sequentiellen Programms, so spricht man von *In-order-Issue* bzw. *In-order-Completion*, erfolgt sie in veränderter Reihenfolge, so spricht man von *Out-of-order-Issue* bzw. *Out-of-order-Completion*. In der Kombination dieser vier Möglichkeiten von Issue und Completion ergeben sich vier unterschiedliche Organisationsformen von Prozessoren mit zum Teil gemeinsamen und zum Teil speziellen Problemen bezüglich der Daten-, Sprung- und Betriebsmittel-Abhängigkeiten, zu denen es wiederum eine Reihe von Lösungen gibt. In einer Verfeinerung der Betrachtung wird darüber hinaus zwischen dem Zuordnen eines Befehls an einen der Funktionseinheit vorgeschalteten Befehlspuffer (*issue*) und dem dann ggf. verzögerten Ausführen des Befehls (*execution*) unterschieden.

Da die meisten Funktionseinheiten superskalarer CISC- und RISC-Prozessoren in *Fließbandtechnik* arbeiten, gehen wir bei den folgenden Betrachtungen, wenn nichts anderes erwähnt, von der Fließbandverarbeitung aus und legen dabei eine einfache, d.h. nur 4-stufige Verarbeitung, bestehend aus

- F (Fetch / Befehl holen),
- D (Decode / Befehl decodieren und Operanden holen),

- E (Execute / Operation ausführen) und
- W (Write / Resultat schreiben)

zugrunde, so wie in 1.4.1 beschrieben und in Bild 2-32 für einen zweifach superskalaren Prozessor gezeigt.

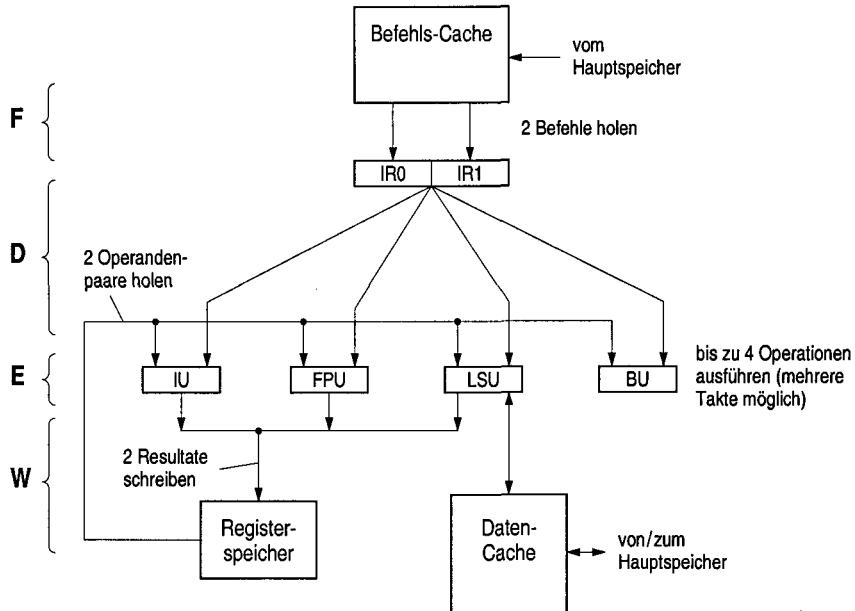


Bild 2-32. Superskalarer Prozessor in Fließbandtechnik (Stufen F, D, E, W) mit vier parallel arbeitenden Funktionseinheiten IU (Integer-Unit), FPU (Floating-Point-Unit), LSU (Load-/Store-Unit) und BU (Branch-Unit). Pro Takt können zwei Befehle aus dem Befehls-Cache in die beiden Befehlsregister IR0 und IR1 geholt sowie zwei Befehle decodiert und zusammen mit ihren Quell-operanden an die Funktionseinheiten verteilt werden (zweifach superskalar)

In-order-Issue, In-order-Completion. Bei In-order-Issue werden die Befehle des sequentiellen Programms in unveränderter Reihenfolge der Bearbeitung zugeführt. Sie werden dabei jeweils so weit decodiert und den Funktionseinheiten zugeordnet, bis eine Datenabhängigkeit, eine Sprungabhängigkeit oder eine Betriebsmittel-Abhängigkeit erkannt wird. Tritt ein solcher Konflikt auf, so wird mit dem Decodieren und Zuordnen nachfolgender Befehle so lange gewartet, d.h., die Befehlausführung wird blockiert (*interlock*), bis die Abhängigkeit und damit der Konflikt aufgelöst ist. Bei der Befehlsbearbeitung in Fließbandtechnik bedeutet das ein Abreißen des Flusses im Fließband, im Englischen als „*stall*“ bezeichnet. Dementsprechend können nachfolgende Befehle zunächst nicht bearbeitet werden, auch wenn sie selbst keine Abhängigkeiten aufweisen. – In-order-Completion legt fest, daß das Schreiben der Zieloperanden in der ursprünglichen Reihenfolge der sie erzeugenden Befehle zu erfolgen hat. Auch hier kann es, um

die korrekte Reihenfolge einzuhalten, zu einem Aufschub in der Bearbeitung kommen, z.B. wenn parallel arbeitende Funktionseinheiten ungleich lange Ausführungszeiten haben.

Datenabhängigkeit (true data dependency). Von Datenabhängigkeit spricht man, wenn ein Befehl einen Quelloperanden benötigt, der von einem im Programm vorangehenden Befehl als Zieloperand erzeugt wird, wie dies das folgende Beispiel der Operationen zweier im Programm unmittelbar aufeinanderfolgender RISC-Befehle anhand des im Register r3 erzeugten Operanden zeigt:

$$\begin{array}{ll} \text{B1: } & r1 + r2 \rightarrow \underline{r3} \\ \text{B2: } & \underline{r3} \cdot 2 \rightarrow r4 \end{array} \quad \square \quad \text{Datenabhängigkeit}$$

Im Englischen wird die Datenabhängigkeit mit dem Zusatz „true“ charakterisiert, da sie ihre Ursache in der Problemstellung hat, also unabhängig von der Programmierung und von einer Optimierungen durch den Übersetzer oder die Prozessorthardware ist. Sie wird auch, die Reihenfolge der Zugriffe auf den Operanden in den Vordergrund stellend, als *Read-after-Write-* oder als *Write-Read-Abhängigkeit* bezeichnet. (Wie später gezeigt wird, können durch die genannten Einflußnahmen weitere Datenabhängigkeiten, sog. Pseudoabhängigkeiten auftreten.)

Die bei einem skalaren Fließbandprozessor gebräuchlichen Lösungen dieses Problems sind in 1.4.2 beschrieben: (a) das Verzögern der zweiten Operation, indem der Übersetzer mit Kenntnis des Fließbandaufbaus einen *nop-Befehl* zwischen die beiden Befehle schiebt, (b) das Verzögern der zweiten Operation, indem der Prozessor den zweiten Befehl für einen Takt blockiert (*interlock*), oder (c) das direkte Weiterreichen des Zieloperanden durch einen *Bypass (feed forwarding)*, wodurch kein Zeitverlust entsteht.

Werden die beiden Befehle gleichzeitig zwei Funktionseinheiten eines superskalaren Prozessors zugeordnet, so ist eine Verzögerung für den zweiten Befehl unabdingbar, da ein Bypass erst dann wirksam werden kann, wenn der Zieloperand des ersten Befehls am Ausgang der ihn erzeugenden Funktionseinheit vorliegt. Bild 2-33 demonstriert diesen Zusammenhang. Es zeigt in Teilbild a zunächst zwei voneinander unabhängige Befehle, die unverzögert fertiggestellt werden, und in Teilbild b die durch eine Datenabhängigkeit hervorgerufene Verzögerung für den zweiten Befehl, hier von einem Takt, entsprechend der Ausführungszeit des ersten Befehls. Eine größere Verzögerung als im Bild gezeigt entsteht dann, wenn der erste Befehl eine gegenüber dem zweiten Befehl längere Ausführungszeit hat, wie z.B. ein Lade-Befehl. Hier kann ggf. durch Umordnen der Befehle durch den Übersetzer oder den Prozessor (siehe später: Out-of-order-Issue) Abhilfe geschaffen werden.

Scoreboarding. Ein einfaches Verfahren, eine Datenabhängigkeit durch Hardware zu erkennen, ist das sog. Scoreboarding. Es benutzt eine „Anzeigetafel“ (*scoreboard*), mittels der jedes der gemeinsam benutzten Register mit einer Kennung

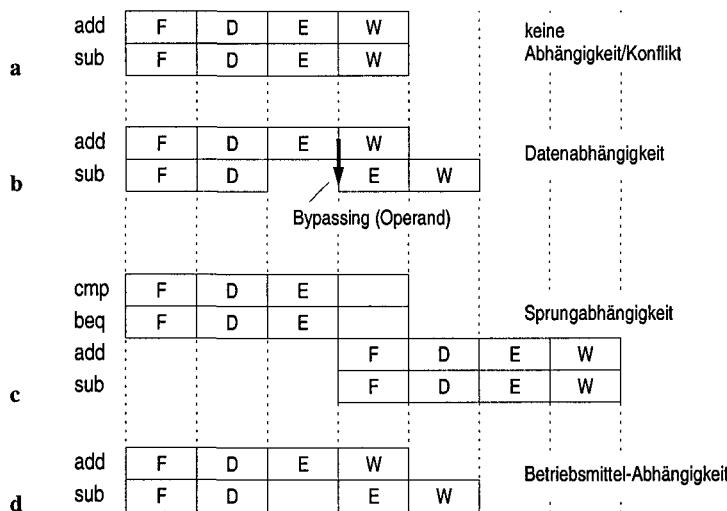


Bild 2-33. Auswirkungen der Abhängigkeiten und Konflikte auf den Befehlsdurchsatz eines zweifach superskalaren Prozessors. **a** Keine Abhängigkeit/Konflikt, **b** Datenabhängigkeit, **c** Sprungabhängigkeit, **d** Betriebsmittel-Abhängigkeit. Darstellung in Anlehnung an [Stallings 1996]

versehen werden kann. Für das obige Befehlspaar als Beispiel heißt das: Wenn der erste Befehl (Additionsbefehl) die Decode-Stufe durchläuft, wird eine Schreibkennung für das Register r3 eingetragen, und wenn dieser Befehl dann die Write-Stufe durchläuft, wird sie wieder gelöscht. Beim Operandenabruf des zweiten Befehls (Multiplikationsbefehl), also ebenfalls in der Decode-Stufe, werden die Kennungen der beiden Quellregister abgefragt und dabei die Datenabhängigkeit erkannt. Bei einem skalaren Prozessor kann diese Erkenntnis für das Verzögern der Befehlausführung oder für das Durchschalten eines Bypasses genutzt werden, bei einem superskalaren Prozessor führt es, wenn beide Befehle gleichzeitig zwei Funktionseinheiten zugeordnet wurden, zunächst zu einem Verzögern des zweiten, abhängigen Befehls, um dann ggf. einen Bypass zu benutzen.

Sprungabhängigkeit (Programmflußabhängigkeit, procedural dependency). Wie in 1.4.2 für den skalaren Fließbandprozessor gezeigt, besteht die Sprungabhängigkeit darin, daß ein bedingter Sprung erst dann ausgeführt werden kann, wenn dem Sprungbefehl die für die Sprungentscheidung erforderlichen Bedingungsbits vorliegen. Meist werden diese Bits durch den vorangehenden Befehl gesetzt bzw. rückgesetzt, z.B. durch einen Vergleichsbefehl. Wie früher beschrieben, muß dafür das Fließband um einen Takt verzögert werden. Das kann (a) durch die Hardware erfolgen (*interlock*), oder (b) durch einen *Delay-Slot*, der vom Übersetzer mit einem nop-Befehl oder einem von der Sprungentscheidung unabhängigen Befehl des Programms, also durch Befehlsumordnung gefüllt wird (siehe 1.4.2).

Betrachtet man einen zweifach superskalaren Prozessor und den Fall, daß der Sprungbefehl und der Vergleichsbefehl gleichzeitig den Funktionseinheiten zuge-

ordnet und gleichzeitig ausgeführt werden, so ergeben sich nun zwei Takte Verzögerung für den Abruf der nächsten Befehlspaare, was einem Verlust von vier Befehlsausführungen entspricht (Bild 2-33c). Der Verlust ist naturgemäß höher als bei einem skalaren Prozessor, da der superskalare Prozessor eine höhere Verarbeitungsleistung hat, die er in diesem Fall nicht nutzen kann. Grundsätzlich könnte auch hier die Technik der Sprungverzögerung eingesetzt werden; es dürfte in der Praxis jedoch schwierig sein, die hierfür benötigten vier Delay-Slots sinnvoll mit Befehlen des Programms zu füllen. Um den Durchsatzeinbruch zu mindern, nutzen superskalare Prozessoren deshalb die Möglichkeit der *Sprungvorhersage* und führen die im Programmzweig der Vorhersage liegenden Folgebefehle *spekulativ* aus. Die Ergebnisse dieser Befehlsausführungen werden später, wenn die Sprungentscheidung vorliegt, entweder bestätigt oder verworfen (siehe dazu 2.3.6). – Bei den unbedingten Sprungbefehlen bleibt es hingegen bei einer Sprungverzögerung von nur einem Takt. Auch hier kann der damit verbundene Durchsatzeinbruch durch die Technik der Vorhersage gemindert werden (siehe dazu ebenfalls 2.3.6).

Betriebsmittel-Abhängigkeit (resource dependency/conflict). Eine Betriebsmittel-Abhängigkeit tritt immer dann auf, wenn zwei Befehle gleichzeitig dasselbe Betriebsmittel benötigen. Dies gilt bereits für skalare Prozessoren mit Fließbandverarbeitung und betrifft dort z.B. den Systembus, den Hauptspeicher, einen gemeinsamen Cache für Befehle und Daten sowie die Speicherverwaltungseinheit. Als Beispiel für eine solche Abhängigkeit, wie sie in der RISC-Struktur in Bild 1-24, S. 51, auftreten kann, seien zwei Hauptspeicherzugriffe genannt, wovon einer als Datenzugriff durch einen Lade-Befehl und der andere als Befehlszugriff infolge eines Cache-Fehlzugriffs beim Befehlsabruf ausgelöst wird. Um solche Abhängigkeiten zu verhindern, müssen die Betriebsmittel vervielfacht werden. Die typische Lösung für das genannte Beispiel ist die Bereitstellung von zwei Caches für eine getrennte Daten- und Befehlsspeicherung, von zwei Speicherverwaltungseinheiten für die getrennte Adressumsetzung sowie von zwei Bussen als Befehls- bzw. Datenbus zwischen dem Prozessor und den beiden Caches. Natürlich ist eine Betriebsmittel-Abhängigkeit damit nicht gänzlich ausgeschlossen, da bei zwei gleichzeitigen Cache-Fehlzugriffen beide Zugriffe um den Hauptspeicher bzw. den Systembus konkurrieren werden.

Anmerkung. Die oben beschriebene Struktur mit getrennten Speichern für Befehle und Daten (hier Caches) und entsprechend getrennten Bussen, wie bei Prozessoren mit Fließbandverarbeitung üblich, wird in Anlehnung an den bei IBM gebauten und ab 1944 an der Harvard-Universität in Cambridge (USA) betriebenen Harvard Mark I, als *Harvard-Architektur* bezeichnet. (Eine diesbezüglich gleiche Struktur wies zuvor schon der Rechner Z3 von Konrad Zuse auf!) Ihr gegenüber steht die sog. *Von-Neumann-Architektur*, bei der die Befehle und Daten in einem gemeinsamen Speicher untergebracht sind. Diese Bezeichnung basiert auf einer Veröffentlichung von John von Neumann im Jahre 1945, den Rechner EDVAC betreffend. Alternativ dazu spricht man auch von der *Princeton-Architektur*, gemäß den Arbeiten von John von Neumann und anderen am Institute of Advanced Studies an der Universität von Princeton.

Bei superskalaren Prozessoren erhöht sich die Wahrscheinlichkeit einer Betriebsmittel-Abhängigkeit dadurch, daß mehrere Befehle gleichzeitig den Funktionseinheiten zugeordnet werden. Benötigt ein Befehlspaar dieselbe Funktionseinheit, so muß hier der zweite Befehl so lange verzögert werden, bis die Funktionseinheit wieder verfügbar ist (Bild 2-33d). Man bezeichnet dies auch als *Befehlsklassenkonflikt*. Abhilfe schafft auch hier die Vervielfachung der Betriebsmittel, d.h. der Funktionseinheiten. So wird z.B. die häufig benötigte Funktionseinheit für die Ganzzahlarithmetik (Integer-Unit) bei vielen Prozessoren doppelt oder sogar dreifach vorgesehen, ggf. mit unterschiedlichen Funktionalitäten.

Beispiel 2.6. ► Pentium I. Der CISC-Prozessor Pentium I (Intel) liefert ein historisches Beispiel für In-order-Issue und In-order-Completion. Er arbeitet zweifach superskalar mit zwei 5-stufigen Fließbändern für die Ganzzahlarithmetik (PF: Prefetch, D1: Decode1, D2: Decode2, EX: Execute, WB: Write Back), der sog. u- und der v-Pipeline, wie in Bild 2-34 gezeigt. Diesen beiden Funktionseinheiten können taktweise ein oder zwei Befehle zugeordnet werden (Issue), wobei letzteres voraussetzt, daß bestimmte Paarbildungsregeln eingehalten werden. So dürfen u.a. keine Daten-

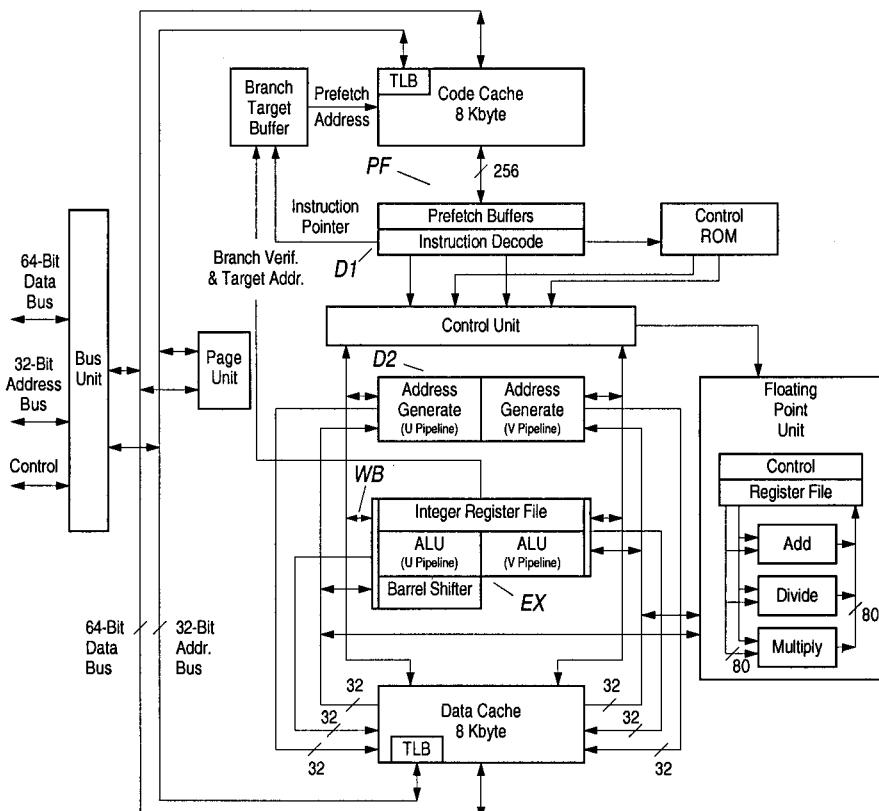


Bild 2-34. Grobstruktur des „historischen“ Pentium I (nach [Intel 1993], erweitert um die Kurzbezeichnungen der Fließbandstufen PF, D1, D2, EX und WB, um deren Wirkungsbereiche besser lokalisieren zu können)

abhängigkeiten zwischen den beiden Befehlen bestehen, die v-Pipeline darf nur mit sog. einfachen Befehlen beschickt werden, die in einem Takt ausführbar sind, und ein Sprungbefehl muß immer als zweiter Befehl im Paar stehen. Grundsätzlich kann der v-Pipeline (als zweiter Pipeline) nur ein Befehl zugeordnet werden, der in der Befehlssequenz des Programms auf den der u-Pipeline zugeordneten Befehl folgt.

Die Entscheidung, ob die Paarbildungsregeln erfüllt sind, wird in der ersten Decode-Stufe D1 von der Hardware getroffen. Sind sie nicht erfüllt, so erfolgt die Befehlsweiterleitung zeitversetzt unter Einhaltung der richtigen Befehlsreihenfolge; sind sie erfüllt, so werden beide Befehle takt synchron durch die folgenden Stufen geleitet. Wird einer der Befehle in einer der Fließbandstufen aufgehalten, so wird auch der andere Befehl aufgehalten. Eine Ausnahme bildet die Execute-Stufe. Hier wird dem Befehl der u-Pipeline erlaubt weiterzugehen, auch wenn der Befehl in der v-Pipeline festgehalten wird; umgekehrt gilt das nicht. Dies ist im Sinne von In-order-Completion, d.h., ein in der Folge zweiter Befehl (v-Pipeline) kann den in der Folge ersten Befehl (u-Pipeline) nicht überholen. (Die Execute-Stufe wird sowohl für die ALU-Operation als auch für den Cache-Zugriff benutzt. Benötigt ein Befehl beide Funktionen, so benötigt er in dieser Stufe mehr als nur 1 Takt!) – *Anmerkung:* In-order-Completion gilt beim Pentium allerdings nicht zwischen den beiden u/v-Pipelines und der zusätzlich vorhandenen Gleitpunkteinheit. Die Gleitpunkteinheit wird wie beim Vorgängerprozessor i486 in Coprozessorart mit Befehlen versorgt, d.h. taktversetzt mit den Ganzzahleinheiten, und führt wie beim i486-Prozessor zu Out-of-order-Completion. ▶

In-order-Issue, Out-of-order-Completion. Out-of-order-Completion tritt insbesondere bei skalaren Prozessoren auf, wenn diese als Coprozessorsysteme ausgelegt sind. Wie bereits in 2.3.1 beschrieben, werden hier die Befehle nacheinander einerseits der Ganzzahleinheit und andererseits der Gleitpunkteinheit zugeordnet. Da die Gleitpunkteinheit für die Befehlausführung üblicherweise mehr Zeit benötigt als die Ganzzahleinheit, kann ein in der Gleitpunkteinheit befindlicher Befehl von einem oder mehreren nachfolgenden Nicht-Gleitpunktbefehlen überholt werden, die dann vor ihm abgeschlossen werden. Out-of-order-Completion wird aber auch bei superskalaren Prozessoren angewendet, um den ggf. unterschiedlich langen Ausführungszeiten der verschiedenen Funktionseinheiten Rechnung zu tragen.

Bei dieser Organisationsform gibt es dieselben Konflikte, wie sie oben beschrieben sind. Erhöht wird bei superskalaren Prozessoren jedoch die Wahrscheinlichkeit einer Datenabhängigkeit, da diese auch durch weniger eng benachbarte Befehle hervorgerufen werden kann. Bei Coprozessorsystemen wird dieses Problem dadurch gemindert, daß die Ganzzahleinheit und die Gleitpunkteinheit jeweils einen eigenen Registerspeicher haben und daß gegenseitige Zugriffe auf die Registerspeicher selten sind.

Unpräzise und präzise Unterbrechungen. Die Befehlsbeendigung mit veränderter Reihenfolge schafft ein zusätzliches Problem, und zwar im Zusammenhang mit der Unterbrechung des Programms durch Traps oder Interrupts. Wird z.B. durch einen der Befehle ein Trap ausgelöst, und wurden zu diesem Zeitpunkt durch andere Funktionseinheiten bereits Befehle beendet, die im Programm auf den trapauslösenden Befehl folgen, so stimmt der Programmverarbeitungszustand nicht mit dem Zustand überein, den man aufgrund der Befehlsreihenfolge im Programm erwarten würde. Man spricht in diesem Fall auch von „unpräzisen“ Unter-

brechungen (*imprecise interrupts*). Präzise Unterbrechungen (*precise interrupts*) erreicht man ggf. mittels spezieller Synchronisationsbefehle, die dafür sorgen, daß so lange kein weiterer Befehl an die Verarbeitungseinheiten abgesetzt wird, bis die bereits in Verarbeitung befindlichen Befehle ohne Programmunterbrechung beendet worden sind. Eine andere Möglichkeit besteht darin, durch Setzen eines Bits im Statusregister des Prozessors generell dafür zu sorgen, daß die Befehle in korrekter Reihenfolge abgeschlossen werden. – Ein Beispiel für In-order-Issue und Out-of-order-Completion ist der Pentium I hinsichtlich seiner Gleitpunkteinheit (Coprozessor) in Bezug zu der u- und der v-Pipeline (siehe oben, Beispiel 2.6).

Out-of-order-Issue, Out-of-order-Completion. Bei In-order-Issue wird, wie bereits beschrieben, der Befehlsstrom nur so weit decodiert und zugeordnet, bis ein Konflikt erkannt wird. Dann wird mit dem Decodieren und Zuordnen gewartet, bis dieser aufgelöst ist; d.h., nachfolgende Befehle werden nicht bearbeitet, auch wenn sie selbst keinen Daten- oder Betriebsmittel-Abhängigkeiten unterliegen. Prozessoren mit Out-of-order-Issue nehmen in einem solchen Fall eine *Befehlsumordnung* vor und ziehen Befehle, die keine Konflikte verursachen, in ihrer Verarbeitung vor. Das heißt, die Konfliktbearbeitung erfolgt erst nach der Decodierstufe, wozu die konfliktbehafteten Befehle zwischengespeichert werden.

Befehlspuffer (instruction buffer). Zur Aufnahme der konfliktbehafteten Befehle werden Befehlspuffer eingesetzt, die jedoch keine eigenen Fließbandstufen darstellen, sondern lediglich zur Entkopplung der Decode-Stufe von den Execute-Stufen der Funktionseinheiten dienen. Sie sind entweder dezentral bei den einzelnen Einheiten angesiedelt (*reservation stations, entry schedulers*; ggf. auch mehrere pro Funktionseinheit) oder zentral für alle Funktionseinheiten zu einem Befehlsspeicher zusammengefaßt (*instruction queue*). Wird bei der Decodierung eines Befehls keine Daten- und keine Betriebsmittel-Abhängigkeit festgestellt, so wird er der für seine Ausführung zuständigen Funktionseinheit direkt, d.h. unter Umgehung der Befehlspuffer, zugeordnet. Wird bei der Decodierung jedoch eine solche Abhängigkeit festgestellt, so wird der Befehl im Befehlspuffer so lange gehalten, bis die Daten- oder Betriebsmittel-Abhängigkeit aufgelöst ist. Inzwischen können andere Befehle, die im Programm auf ihn folgen, vorgezogen werden, entweder direkt von der Decode-Stufe oder aus den Befehlspuffern kommend. Die Reihenfolge der Zuordnung der Befehle an die Execute-Stufen der Funktionseinheiten ergibt sich also letztlich dadurch, wann welcher Befehl konfliktfrei ausführbar ist (Out-of-order-Issue bzw. Out-of-order-Execution).

Gegenabhängigkeit (antidependency) und Ausgabeabhängigkeit (output dependency). Die Abhängigkeiten, die bei Out-of-order-Issue bzgl. der Daten auftreten können, demonstrieren die auf der nächsten Seite wiedergegebenen Operationen

eines RISC-Programms, wieder unter der Annahme eines zweifach superskalaren Prozessors.

$$\begin{array}{lll}
 \text{B1: } & r1 / r2 \rightarrow r3 & \square \quad \text{DA.} \\
 \text{B2: } & r3 + 1 \rightarrow r4 & \square \quad \text{Ausgabeabhängigkeit} \\
 \text{B3: } & r5 \cdot r6 \rightarrow r3 & \square \quad \text{Gegenabhängigkeit} \\
 \text{B4: } & r3 \cdot 2 \rightarrow r7 & \square \quad \text{DA.}
 \end{array}$$

Insgesamt gibt es drei unterschiedliche Abhängigkeiten, die sämtliche durch Scoreboarding erkannt werden können:

1. die Datenabhängigkeit, hier zwischen den Befehlen 1 und 2 wie auch zwischen den Befehlen 3 und 4, da das Zielregister des jeweils ersten Befehls, r3, als Quellregister des jeweils nachfolgenden Befehls fungiert,
2. die sog. Gegenabhängigkeit und
3. die sog. Ausgabeabhängigkeit.

Die *Gegenabhängigkeit* basiert darauf, daß ein Befehl in einen Speicherplatz schreibt, der von einem vorangehenden Befehl als Operandenquelle benutzt wird (*Write-after-Read-, Read-Write-Abhängigkeit*). Werden die beiden Befehle aufgrund von Out-of-order-Issue in der Ausführung vertauscht und findet dadurch der Schreibvorgang vor dem Lesevorgang statt, so findet der im Programm vorangehende Befehl einen falschen Quelloperanden vor. In der obigen Befehlssequenz tritt diese Abhängigkeit zwischen den Befehlen 2 und 3 auf, wiederum das Register r3 betreffend.

Die *Ausgabeabhängigkeit* entsteht dadurch, daß zwei Befehle in dasselbe Register schreiben (*Write-after-Write-, Write-Write-Abhängigkeit*). Wird die Reihenfolge des Schreibens aufgrund von Out-of-order-Completion vertauscht, so finden nachfolgende Befehle ein falsches Resultat vor. In der obigen Befehlssequenz tritt diese Abhängigkeit zwischen den Befehlen 1 und 3 auf, ebenfalls das Register r3 betreffend. Hier wird ggf. der Befehl 3 vor dem Befehl 1 fertiggestellt, und zwar einerseits, weil er aufgrund der Datenabhängigkeit zwischen den Befehlen 1 und 2 dem Befehl 2 vorgezogen wird und andererseits, weil der Befehl 1 in der Execute-Stufe mehr als 2 Takte benötigt (Division!).

Da sowohl die Gegenabhängigkeit als auch die Ausgabeabhängigkeit keine Abhängigkeiten sind, die in dem durch das Programm beschriebenen Problem begründet sind, sondern allein dadurch entstehen, daß der Übersetzer seriellen Code (für einen skalaren Prozessor) erzeugt und dabei aufgrund begrenzter Betriebsmittel verschiedenen Variablen ein und dasselbe Register als Speicherplatz zuweist, werden sie auch als *Pseudoabhängigkeiten* bezeichnet.

Registerumbenennung (register renaming). Die Ausgabe- und die Gegenabhängigkeit stellen letztlich eine Betriebsmittel-Abhängigkeit dar, die dann auftritt, wenn der Prozessor mit Out-of-Order-Techniken arbeitet. Sie können durch

Betriebsmittelvervielfachung, d.h. durch Bereitstellung zusätzlicher Register, sog. *Rename-Register* aufgehoben werden. Da diese zusätzlichen Register für den Übersetzer unsichtbar sind, ist es Aufgabe des Prozessors, eine neue Registerzuordnung ohne Doppelbenutzung an die Befehle vorzunehmen (Einmalzuweisung, Single Assignment), und zwar bevor er die Befehlsreihenfolge verändert. Dieses dynamische Zuordnen erfolgt jeweils dann, wenn ein neues Zielregister benötigt wird. In der Folge müssen dann die nachfolgenden Befehle bezüglich ihrer Quellregister einen Prozeß zur Registerumbenennung durchlaufen. Die „Entnahme“ eines Registers aus dem vorhandenen Registervorrat geschieht z.B. mit aufsteigenden Registeradressen oder in assoziativer Weise, bis alle Register „verbraucht“ sind. Dann beginnt die Entnahme von vorn, indem die inzwischen wieder freigegebenen Register erneut benutzt werden. Sind zu einem Zeitpunkt alle vorhandenen Register in Benutzung, so daß ein weiterer Registerbedarf nicht befriedigt werden kann, so kommt es zu einer Verzögerung der Ausführung der Folgebefehle.

Dieser Vorgang der Registerumbenennung sei an dem obigen Programmbeispiel durch die den Registerbezeichnungen hinzugefügten Indizes a, b, ... demonstriert. Die Registernummern 1, 2, ... entsprechen dabei den vom Übersetzer vorgenommenen Registerzuordnungen bezüglich der „sichtbaren“ Register. Die Verwendung zweier verschiedener Rename-Register r3a und r3b ermöglicht es, sowohl die Ausgabeabhängigkeit der Befehle 1 und 3 als auch die Gegenabhängigkeit der Befehle 2 und 3 zu vermeiden, wie die linke Befehlsfolge der folgenden Darstellung zeigt.

B1: r1a / r2a → r3a	DA.	Out-of-Order Issue/Execution	B1:r1a / r2a → r3a
B2: r3a + 1 → r4a	DA.		B3:r5a · r6a → r3b
B3: r5a · r6a → r3b	DA.	→	B2:r3a + 1 → r4a
B4: r3b · 2 → r7a	DA.		B4:r3b · 2 → r7a

Nach der Registerumbenennung bleiben die beiden Datenabhängigkeiten in ihrer Wirkung zunächst erhalten. Der Prozessor kann nun aber aufgrund des Ausschaltens der Pseudoabhängigkeiten den Befehl 3 mit Out-of-order-Issue vor den Befehl 2 ziehen, wie in der rechten Befehlsfolge gezeigt. Bezogen auf unsere Annahme eines zweifach superskalaren Prozessors vermeidet er damit die bei In-order-Issue durch die Befehlspaare B1/B2 und B3/B4 verursachten Verzögerungen und führt statt dessen die Befehlspaare B1/B3 und B2/B4 unverzögert aus.

Anmerkung. Die Registervervielfachung kann auch derart realisiert werden, daß zusätzlich zum allgemeinen Registersatz des Prozessors jedem der Befehlspuffer drei zusätzliche Register für zwei Quelloperanden und einen Zieloperanden zugeordnet werden.

Beispiel 2.7. ▶ PowerPC 601. Der RISC-Prozessor PowerPC 601 (Motorola) liefert ein historisches Beispiel für Out-of-order-Issue und Out-of-order-Completion. Er arbeitet dreifach superskalär und hat als Funktionseinheiten eine Sprungeinheit (Branch-Unit BU), eine Ganzzahleneinheit (Integer-Unit IU), die auch für die Ausführung der Lade- und Speichere-Befehle zuständig ist (Ld/St), und eine Gleitpunkteinheit (FPU), wobei die IU und die FPU jeweils einen eigenen Register- satz haben (Bild 2-35). Die Zuordnung der Befehle an die Funktionseinheiten erfolgt out-of-order

und unterliegt einer sog. Dispatch-Unit, die dazu auf die vorderen vier Speicherplätze eines als Warteschlange organisierten achtfachen Befehlspuffers zugreift. Sprungbefehle innerhalb dieser vier Speicherplätze werden der BPU zugeordnet und dort decodiert und ausgeführt. IU-Befehle werden hingegen in der vordersten Stelle des Befehlspuffers decodiert und können deshalb auch nur von dort aus an die IU weitergeleitet werden. Die IU hat ein zusätzliches Pufferregister, in dem ein decodierter Befehl dann zwischengespeichert wird, wenn die Execute-Stufe noch belegt ist. FPU-Befehle können wiederum aus den vorderen vier Speicherplätzen des Befehlspuffers entnommen werden; auch hier gibt es ein zusätzliches Pufferregister, und zwar vor der Decode-Stufe.

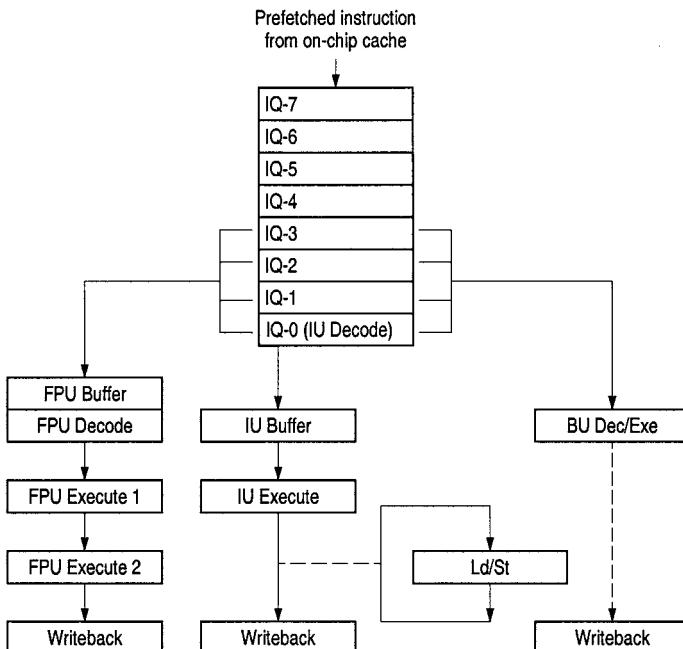


Bild 2-35. Zu Beispiel 2.7: Befehlsverteilung und Fließbandstufen des PowerPC 601 (nach [Motorola 1993])

Tritt in einer der Funktionseinheiten eine Abhängigkeit auf, so wird die Befehlausführung so lange verzögert, bis die Abhängigkeit aufgelöst ist. Im Falle nicht vorhandener Abhängigkeiten kommt es ggf. zu Out-of-order-Completion. Die Einhaltung der Befehlsreihenfolge im Konfliktfall gewährleistet, daß in den beiden Registersätzen keine Daten erzeugt werden, die später evtl. wieder verworfen werden müßten. Man erspart sich dadurch den Einsatz von Rename-Registern oder einer der vergleichbaren Techniken. Um darüber hinaus präzise Programmunterbrechungen zu erreichen, gibt es die bereits oben geschilderten Möglichkeiten der Synchronisation mittels eines sync-Befehls oder eines Synchronisationsbits im Statusregister des Prozessors. ▶

Out-of-order-Issue, In-order-Completion. Out-of-order-Issue ist, wie oben gezeigt, eine Technik, die eine gute Leistungssteigerung bei superskalaren Prozessoren (und VLIW-Prozessoren) ermöglicht, nicht zuletzt deshalb, weil sich auch die damit einhergehenden Probleme der Pseudoabhangigkeiten gut lösen lassen. Out-of-order-Completion hingegen führt zu Komplikationen, die in der Regel

nicht so einfach zu bewältigen sind, nämlich zu *unpräzisen Traps* und *Interrupts*. Heutige superskalare und VLIW-Prozessoren nutzen deshalb die Vorteile von Out-of-order-Issue, arbeiten aber weitgehend mit In-order-Completion, mit eventuellen Ausnahmen für z.B. Befehlausführungen der Gleitpunkteinheit. Sie sehen dazu Maßnahmen vor, mit denen das Beenden der Befehle in der Reihenfolge erzwungen wird, in der sie durch das Programm, d. h. durch den Übersetzer vorgegeben ist. Befehle werden somit zwar im Vorgriff ausgeführt; im Falle von Traps oder Interrupts aber ggf. wieder verworfen.

Rückordnungspuffer (reorder buffer). Das Beenden der Befehle in Programmreihenfolge erreicht man mittels eines sog. Rückordnungspuffers, der als Warteschlangenspeicher (FIFO-Speicher) ausgelegt ist. Er wird mit den die Decode-Stufe durchlaufenden Befehlen geladen, und zwar in der Programmreihenfolge. Die Befehlseinträge haben dabei die Funktion von Marken. Beendet werden kann nun immer nur der Befehl, der sich an der Spitze der Warteschlange befindet. Sein Eintrag wird dabei aus dem FIFO-Speicher entfernt. Wird eine Befehlausführung vorzeitig, d.h. außerhalb dieser Reihenfolge fertiggestellt, so wird sein Resultatwert nicht sofort in das entsprechende Zielregister eingetragen. Er wird statt dessen entweder in den Rückordnungspuffer geschrieben, nämlich an die Stelle des zugehörigen Befehls, oder, wenn Rename-Register vorhanden sind, in einem von diesen zwischengespeichert. Das Zielregister wird erst dann aktualisiert, wenn alle vor ihm im Puffer stehenden Befehle beendet und damit aus dem Puffer entfernt worden sind, und zwar mit dem Inhalt des Rückordnungspuffers bzw. des Rename-Registers.

Tritt zwischenzeitlich eine Programmunterbrechung auf, so zeigt der Registersatz des Prozessors den Zustand, der dem einer Unterbrechung bei In-order-Completion entspricht. *Traps* und *Interrupts* treten also *präzise* auf. Die Resultate vorzeitig abgeschlossener Befehle, die zum Unterbrechungszeitpunkt ja noch im Rückordnungspuffer oder in den Rename-Registern stehen, können dann schadlos verworfen werden. Die Verwaltung des Rückordnungspuffers und der Rename-Register unterliegt einer sog. *Completion-* oder *Retirement-Unit*. – Der Rückordnungspuffer begrenzt mit seiner Kapazität die maximal mögliche Anzahl gleichzeitig in Ausführung befindlicher Befehle. Ist er voll, so werden weitere Befehlszuordnungen verzögert.

Die folgenden drei Beispiele beschreiben die Strukturen moderner superskalarer Prozessoren mit Out-of-order-Issue und In-order-Completion.

Beispiel 2.8. ▶ Pentium 4. Der Pentium 4 (Intel) ist ein superskalarer 32-Bit-Prozessor, der aufgrund seines x86-Befehlssatzes als CISC-Prozessor gilt. Die Verarbeitung der x86-Befehle erfolgt jedoch in RISC-Manier, indem diese in eine oder, bei komplexen x86-Befehlen, in mehrere Mikrobefehle mit RISC-Befehlsformat, sog. μ OPs (micro-operations), umgesetzt werden. Die Umsetzung erfolgt nach dem Befehlszugriff auf den L2-Cache in einem mehrstufigen Decoder (Bild 2-36). Die so vordecodierten Mikrobefehle werden dann in einem sog. *Trace-Cache* gespeichert, der den herkömmlichen L1-Befehls-Cache ersetzt. Die für jeweils sechs Mikrobefehle ausgelegten Zeilen des Trace-Cache werden dabei nicht, wie sonst üblich, mit Befehlsblöcken belegt,

die der Speicherdarstellung entsprechen, sondern mit Folgen von Mikrobefehlen, wie sie dem Ausführungsfluß entsprechen, sog. Traces. Diese Traces werden durch Sprungbefehle angeführt, gefolgt von den Befehlen des einen bzw. anderen Programmzweiges. Dies ermöglicht es insbesondere, einen Sprungbefehl und die Befehle an seinem Sprungziel in einer Cache-Zeile zusammenzufassen. Gespeichert werden dabei nicht alle Mikrobefehle eines komplexen x86-Befehls, sondern lediglich dessen erster. Die restlichen Mikrobefehle aller komplexen x86-Befehle stehen in einem Microcode-ROM, das dann vom Trace-Cache aus aktiviert wird. Der Zugriff auf den Trace-Cache ist, wie auch schon der L2-Cache-Zugriff, mit einer dynamischen Sprungvorhersage verbunden. Beim L2-Cache-Zugriff wird außerdem ein Prefetch-Mechanismus wirksam, der Befehlsblöcke im Zusammenwirken mit der Sprungvorhersage im Vorgriff abruft. Ausgehend vom Trace-Cache beginnt dann die eigentliche Fließbandverarbeitung des Pentium 4 mit insgesamt 20 Stufen (32 Stufen beim Nachfolger Pentium 4E, Prescott).

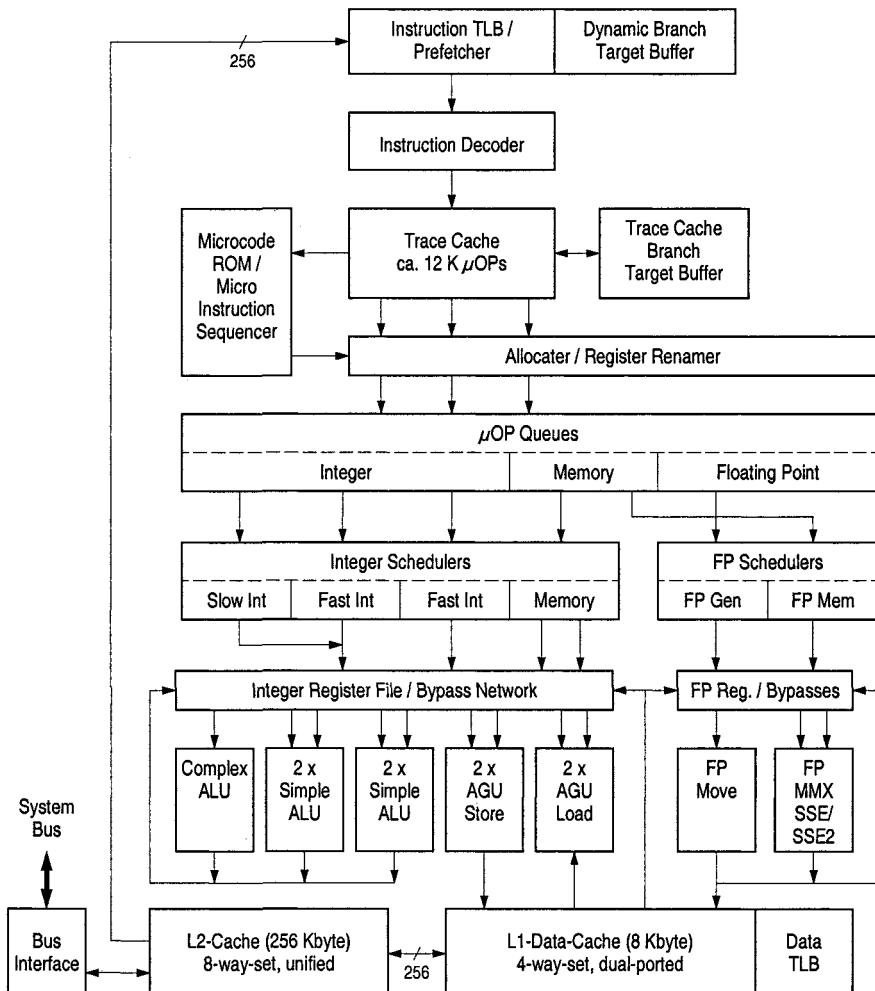


Bild 2-36. Grobstruktur des Pentium 4 (in Anlehnung an [Pabst 2000]). Nachfolger Pentium 4E mit u.a. SSE3, 8-Wege-assoziativem L1-Daten-Cache mit 16 Kbyte, L2-Cache mit 1 Mbyte

Aus dem Trace-Cache können pro Takt bis zu drei Mikrobefehle ausgelesen und der Out-of-Order-Logik zugeführt werden. Hier werden die Mikrobefehle zunächst auf Abhängigkeiten überprüft, Register zugewiesen und ggf. Registerumbenennungen vorgenommen (Allocate/Register Renamer). Außerdem erfolgt für jeden Mikrobefehl ein Eintrag in einen Rückordnungspuffer, um ihn später „in-order“ abschließen zu können, und zwar wiederum bis zu drei Mikrobefehle pro Takt. Diese Befehle werden dann (bis zu drei pro Takt) an zwei FIFO-Befehlspuffer weitergeleitet, einen für Speicherzugriffsbefehle und einen für die anderen Mikrobefehle (μ OP Queues). Die eigentliche Umordnung der Mikrobefehle erfolgt dann in den sog. Schedulers, die die Befehle an die entsprechenden Funktionseinheiten übergeben, sobald die benötigten Eingangsoperanden verfügbar sind. Bis zu sechs Mikrobefehle können hier pro Takt weitergeleitet werden, teilweise unter Nutzung von Halbtaktabständen.

Als Funktionseinheiten stehen zur Verfügung: eine Ganzzahleinheit für komplexe Befehle (Complex ALU), zwei schnelle Ganzzahleinheiten für einfache Befehle, die mit jedem halben Takt einen neuen Mikrobefehl übernehmen können (2 x Simple ALU), zwei schnelle Adreßrecheneinheiten für Speicher- bzw. Lade-Befehle, die ebenfalls halbtaktweise arbeiten (2 x AGU Store/Load) sowie eine Gleitpunkteinheit die in zwei Untereinheiten aufgeteilt ist. Die einfachere der beiden Untereinheiten führt lediglich Datentransporte durch (FP Move), die andere ist für die herkömmlichen Gleitpunktoperationen nach dem IEEE-Standard 754 (FP) sowie für Multimedia-Operationen zuständig (MMX, SSE/SSE2; SSE3 beim Nachfolger Pentium 4E). Letztere sind Vektoroperationen, sog. SIMD-Operationen (single instruction, multiple data), bei denen ggf. pro Befehl mehrere Operandenpaare gleichzeitig bearbeitet werden. MMX als die ältere Technik kennt nur Ganzzahloperationen, SSE und SSE2 erlauben auch Gleitpunktoperationen. Insgesamt können sich bis zu 126 Mikrobefehle gleichzeitig zur Bearbeitung in den Fließbandstufen befinden („in flight“). Ein hier nicht betrachtetes Merkmal des Pentium 4, auch in Bild 2-36 nicht ersichtlich, ist dessen vielfältige Verarbeitung; dazu siehe 2.3.8. Für mehr Details zum Pentium 4 siehe z.B. [Hinton, Sager et al. 2001]. ▶

Beispiel 2.9. ► Opteron und Athlon 64. Der Opteron (AMD) ist ein superskalarer 64-Bit-Server-Prozessor mit Out-of-order-Issue und In-order-Completion. In leicht modifizierter Form und höher getaktet gibt es ihn mit der Bezeichnung Athlon 64 als Desktop-Prozessor. Beide Prozessoren sind in ihrem Kern dem des 32-Bit-Vorgängers Athlon sehr ähnlich, zeichnen sich gegenüber diesem jedoch durch die 64-Bit-Verarbeitung aus. Als weitere Besonderheiten haben sie, wie Bild 2-37 zeigt, einen On-chip-Memory-Controller und drei HyperTransport-Verbindungen, die mittels eines Kreuzschienenverteilers (Crossbar-Switch) mit dem Prozessorkern verbunden sind. Das heißt, der herkömmliche Prozessorbus wird durch einen (Athlon 64) oder zwei Speicherkanäle (Opteron) sowie drei voneinander unabhängig und bidirektional übertragende Punkt-zu-Punkt-Verbindungen ersetzt. Letztere werden ggf. für den Aufbau von Mehrprozessorsystemen genutzt.

Die AMD-Prozessoren basieren wie die Pentium-Prozessoren auf dem x86-Befehlsatz, sind also CISC-Prozessoren. Auch sie lösen die x86-Befehle in einen oder mehrere Mikrobefehle mit RISC-Befehlsformat, sog. ROPs (RISC operations), auf und führen diese dann in RISC-Manier mit Fließbandverarbeitung aus. Diese Aufgabe übernehmen die in Bild 2-37 gezeigten Decoder der ersten Ebene. Sie erzeugen bis zu sechs Mikrobefehle pro Takt, die dann in der nächsten Ebene zu Makrobefehlen (MOPs), die aus maximal zwei Mikrobefehlen bestehen, zusammengefaßt werden. Bei z.B. einem arithmetischen x86-Befehl mit Speicherzugriff enthält der Makrobefehl einen Mikrobefehl für die Adreßrechnung (AGU) und einen Mikrobefehl für die arithmetische Operation (ALU). Bei einem Register-Register-Befehl reduziert sich der Makrobefehl auf nur einen ALU-Mikrobefehl. Für die Befehlausführungen stehen insgesamt neun Funktionseinheiten zur Verfügung: drei AGU/ALU-Paare für Speicherzugriffe und Ganzzahloperationen sowie eine aus drei Untereinheiten bestehende Gleitpunkt-/Multimediaeinheit (FADD, FMUL, FMISC). Letztere führt neben den Gleitpunktbefehlen der IEEE-Standards 754 und 854 auch MMX-, 3DNow!- und SSE/SSE2-Multimediatebefehle aus. Versorgt werden die neun Funktionseinheiten

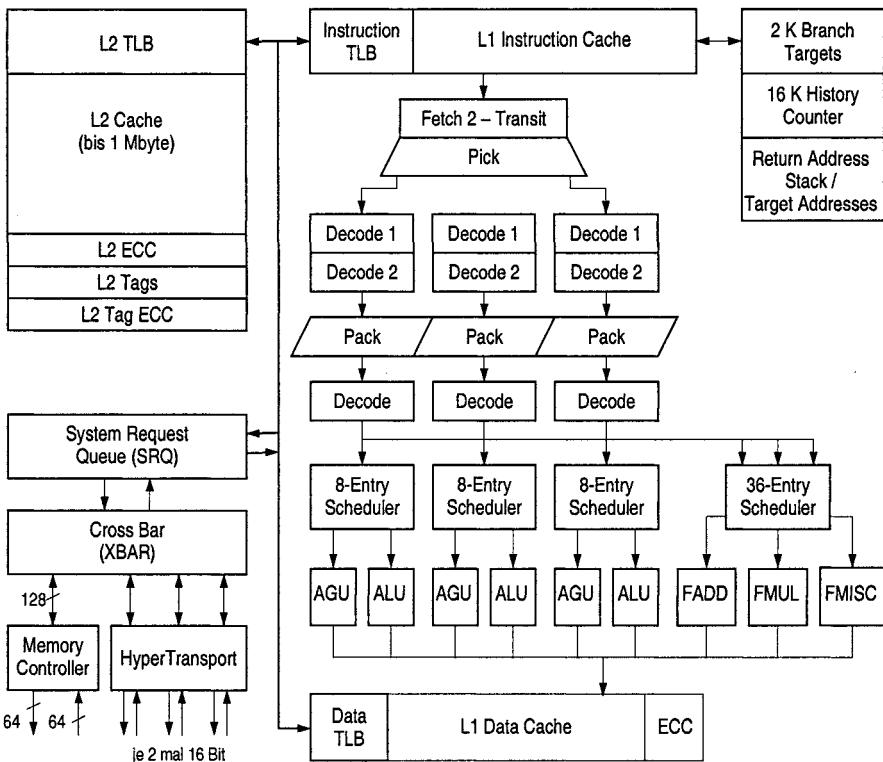


Bild 2-37. Grobstruktur des Opteron/Athlon 64 (nach [Stiller 2003])

aus je einem Befehlspuffer für jedes der drei AGU/ALU-Paare (8-Entry-Scheduler) und einem Befehlspuffer für die Gleitpunkt-/Multimediaeinheit (36-Entry Scheduler). Pro Takt können bis zu neun Befehle „out-of-order“ an die Funktionseinheiten übergeben werden. ▲

Beispiel 2.10. ► PowerPC 970 (PPC970). Der PowerPC 970 (IBM) ist ein superskalarer 64-Bit-RISC-Prozessor mit zwölf in vielstufiger Fließbandtechnik arbeitenden Funktionseinheiten (Bild 2-38). Diese sind: zwei 16-stufige Ganzzahleinheiten (ALU), zwei 1-stufige Gleitpunkteinheiten (FPU), eine 16-stufige sog. Bedingungsregister-Einheit (Condition-Register-Unit), eine 16-stufige Sprungeinheit (Branch-Unit), vier sog. SIMD-Einheiten (Vector-Engine, AltiVec) mit bis zu 25 Stufen und zwei 17-stufige Lade/Speichere-Einheiten (Load/Store-Unit). Den beiden ALUs sowie den beiden FPUs sind jeweils 32 „sichtbare“ 64-Bit-Datenregister zugeordnet, ergänzt um jeweils 48 ebensoreibre Rename-Register. Für die vier SIMD-Einheiten (*single-instruction, multiple data*) gibt es einen dritten Registersatz mit ebenfalls 32 sichtbaren und 48 Rename-Registern, jedoch 128 Bit breit. Die Datenpfade sind dazu passend 64 bzw. 128 Bit breit. Die SIMD/AltiVec-Einheiten erlauben Mehrfachoperationen mit vom Datenformat abhängiger Operationenzahl: vier mal 32-Bit-Ganzzahl, acht mal 16-Bit-Ganzzahl, 16 mal 8-Bit-Ganzzahl oder vier mal 32-Bit-Gleitpunktzahl. Eine der vier SIMD-Einheiten (PERM) ist für Permutationsoperationen zuständig. – Zur Stufung der Fließbänder der Funktionseinheiten siehe auch Beispiel 2.12 (S. 148).

Der PowerPC 970 arbeitet mit Out-of-order-Issue und In-order-Completion. Bis zu acht 32-Bit-Befehle (Dreiaadressformat) können gleichzeitig aus dem Befehls-Cache (I-Cache) geholt und dekodiert werden. Dabei werden die ersten neun Fließbandstufen durchlaufen, in denen u.a. die Ab-

hängigkeiten festgestellt werden (siehe auch Bild 2-40, S. 148). Des weiteren werden in diesen Fetch- und Decode-Stufen einige der komplexen Befehle in mehrere einfachere Mikrobefehle „aufgebrochen“ (instruction cracking), so z.B. der Multiply-Add-Befehl. In der Folge werden dann bis zu fünf Befehle gleichzeitig und ggf. umgeordnet an insgesamt sechs Befehlspuffer (instruction queues) der Funktionseinheiten verteilt (dispatch group), und zwar jeweils ein Sprungbefehl und vier andere Befehle (4-plus-1-superskalär). Bei der Ausführung der Befehle (Execute) können wiederum bis zu acht Ergebnisse pro Takt erzeugt werden und letztlich bis zu fünf Befehle pro Takt in der ursprünglichen Reihenfolge abgeschlossen werden (Completion). Insgesamt können sich mehr als 200 Operationen gleichzeitig in der Verarbeitung befinden („in flight“).

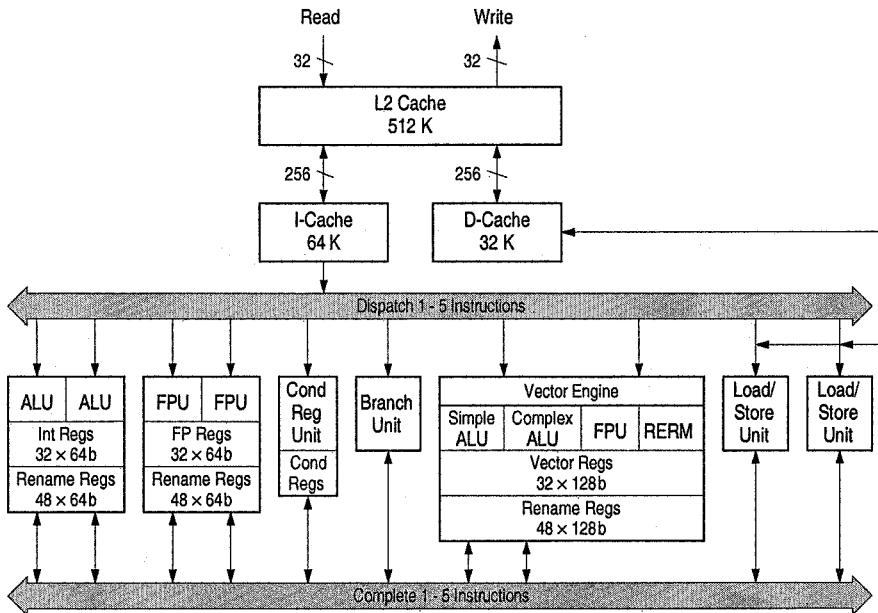


Bild 2-38. Grobstruktur des PowerPC 970 (nach [Halfhill 2002])

2.3.4 VLIW-Prozessoren

Heutige superskalare Prozessoren arbeiten vorwiegend mit Out-of-order-Issue/Execution, was aufgrund der vom Prozessor durchzuführenden (dynamischen) Befehlsumordnung mit sehr komplexen Prozessorstrukturen, d.h. hohen Entwicklungskosten und hohem Hardwareaufwand, verbunden ist. Ihr Vorteil ist die Befehlsparallelität in der Verarbeitung. Außerdem wirken sich bei ihnen Strukturänderungen, z.B. die Erhöhung der Anzahl an Funktionseinheiten oder der Superskalarität (Anzahl der pro Takt an die Funktionseinheiten absetzbaren Befehle), nicht auf die Software aus. So ist ein auf einem Prozessor lauffähiges Programm auch auf dessen Nachfolgemodell lauffähig. Man bezeichnet superskalare Prozessoren deshalb als aufwärtskompatibel oder auch als bzgl. der Funktionseinheiten und der Superskalarität *skalierbar*.

Nach dem Aufkommen der superskalaren Prozessoren hat ein zu ihnen alternatives Konzept Eingang in die Rechnerarchitektur gefunden. Will man nämlich den Hardwareaufwand verringern, jedoch Out-of-order-Issue/Execution als durchsatzsteigerndes Prinzip beibehalten, so kann man das Umordnen der Befehle an den Compiler übertragen (statische Befehlsumordnung), wie dies bei den VLIW-Prozessoren gemacht wird. Hier faßt der Compiler mehrere parallel ausführbare Befehle zu einem größeren Befehlswort zusammen, die dann vom Prozessor taktgleich an die verschiedenen Funktionseinheiten zur Bearbeitung übergeben werden. Da ein solches Befehlswort je nach Prozessor ggf. sehr viele Befehle umfaßt (z.B. acht bei den digitalen Signalprozessoren TMS320C6x) spricht man von *Very-Long-Instruction-Word* (VLIW). Beim Bilden der Befehlswörter hat der Compiler dieselben Abhängigkeiten zu berücksichtigen, wie sie in 2.3.3 für die superskalaren Prozessoren beschrieben sind, nämlich die Daten-, Sprung- und Betriebsmittel-Abhängigkeiten einschließlich der Pseudoabhängigkeiten, wie sie durch das Umordnen von Befehlen entstehen. Sprungabhängigkeiten lassen sich ggf. mit der Technik der Prädikation vermeiden (siehe dazu 2.3.7). Sie wird z.B. bei dem im untenstehenden Beispiel 2.11 beschriebenen VLIW-Prozessor Itanium 2 eingesetzt.

Bei den ersten VLIW-Prozessoren umfaßte das Befehlswort genau so viele Befehle, wie es parallel arbeitende Funktionseinheiten gab. Das heißt, die Anzahl der Befehle im VLIW und die Anzahl der Funktionseinheiten war gleich. Darüber hinaus wurde jeder Befehlsposition im Befehlswort auch eine bestimmte Funktionseinheit fest zugeordnet. Das hatte den Vorteil einer einfachen Wegeführung für die Befehlsverteilung, jedoch den Nachteil einer relativ schlechten Nutzung des Programmspeichers, da aufgrund der begrenzten Befehlsparallelität von Programmen Befehlspositionen im VLIW häufig mit nop-Befehlen besetzt werden mußten. Eine etwas bessere Nutzung des langen Befehlswortes erreichte man dadurch, daß man die Zuordnungen der Befehlspositionen im VLIW zu den Funktionseinheiten flexibel gestaltete, was den Hardwareaufwand für die Wegeführung entsprechend erhöhte. Aber auch hier ließen sich nop-Befehle nicht vermeiden. Beide Strukturvarianten hatten außerdem den Nachteil, nicht skalierbar zu sein, d.h., für ein Prozessornachfolgemodell mußten die Programme (prozessorspezifisch) neu übersetzt werden.

Bei heutigen VLIW-Prozessoren werden vom Compiler parallel ausführbare Befehle zu Bündeln zusammengefaßt, und zwar unabhängig von der festen Anzahl an Befehlen im VLIW. Dies geschieht z.B. durch ein Kennbit für jeden Befehl, das angibt, mit welchem Befehl ein neues Bündel beginnt (siehe z.B. TMS320C6xxx). Hier kann jetzt ein VLIW mehr als nur ein Bündel enthalten, und ein Bündel kann sich über mehr als nur ein VLIW erstrecken. Pro Takt werden aber immer nur Befehle eines einzigen Bündels und innerhalb desselben VLIW den Funktionseinheiten zugeführt. Damit entfallen nop-Befehle als Platzhalter.

Beispiel 2.11. ► *Itanium 2*. Der Itanium 2 ist ein RISC-Prozessor mit paralleler Befehlsverarbeitung nach dem VLIW-Prinzip. Er basiert auf der sog. *IA-64-Architektur* von Intel und ist als Server-Prozessor konzipiert. Das heißt, er ist nicht für eine extrem hohe Ausführungsgeschwindigkeit für die einzelne Anwendung, sondern für einen möglichst hohen Gesamtdurchsatz ausgelegt. Diesem trägt er u.a. dadurch Rechnung, daß er mit nur acht Fließbandstufen arbeitet und damit bei Fließbandkonflikten, z.B. bei dem für Server erhöhten Aufkommen an Speicherzugriffen, einen nicht so hohen Verlust an ungenutzten Takten hat, wie bei einem längeren Fließband, z.B. dem 20stufigen Fließband des Desktop-Prozessors Pentium 4. Damit verbunden ist allerdings auch eine entsprechend niedrigere Prozessortaktfrequenz (siehe dazu 2.3.5 Superpipelining). Server-nützlich ist auch der bei einem 64-Bit-Prozessor gegenüber einem 32-Bit-Prozessor erweiterte Adreßraum. So werden beim Itanium 2 von den 64 Bits der virtuellen Adressen 50 Bits für die Speicheranwahl, d.h. für die realen Adressen genutzt (zu Speicherverwaltung siehe 6.3).

Bild 2-39 zeigt die Struktur des Itanium 2. Als Verbindung zum Hauptspeicher (Memory) weist er drei Cache-Ebenen auf: zwei getrennte Caches für Befehle und Daten (split caches) in der Ebene 1 (L1) sowie zwei Caches für Befehle und Daten gemeinsam (unified caches) in den Ebenen 2 und 3 (L2, L3), ausgelegt als 4-, 8- bzw. 12-Wege-assoziative Caches und mit 64-, 128- bzw. 128-Byte-Blöcken pro Cache-Zeile. Die L1- und L2-Caches sind on-chip; ebenso der L3-Cache, der als Backside-Cache mit einem breiten Prozessoranschluß von 256 Bit angegeben ist. Seine hohe Speicherkapazität von bis zu 3 Mbyte ist wiederum server-typisch (zu Caches siehe 6.2).

Für die VLIW-Verarbeitung werden jeweils drei RISC-Befehle (Dreiadreßbefehle) zu einem Bündel zusammengefaßt, und jeweils zwei Bündel, d.h. sechs Befehle können taktgleich der Verarbeitung zugeführt werden. Für die Bildung der Bündel gibt es Vorgaben an den Compiler hinsichtlich der kombinierbaren Befehlsarten und deren Aufeinanderfolge im Bündel, um so Konflikte bzgl. der Betriebsmittel zu vermeiden. Die Befehlsarten sind, basierend auf den Funktionseinheiten: M (Memory, Speicherzugriff), I (Integer), F (Floating-Point), B (Branch) und LX (Long-Immediate, d.h. mit Belegung von zwei Befehlspälen im Bündel bei 64-Bit-Direktoperand-Verarbeitung).

Eine Verarbeitungskette kann dann z.B. wie folgt aussehen:

MII-MMF-BBB-MFB-...

Die Bündel werden dazu, vom L1-Befehls-Cache kommend, in einem Pufferspeicher (8 Bundles) bereitgehalten und über elf mögliche Wege (Issue Ports) den Funktionseinheiten zugeführt. Diese sind: drei Sprungbefehleinheiten (Branch Units), sechs Ganzzahleinheiten (ALU) und zwei Gleitpunkteinheiten (FP Units). Alle sechs Ganzzahleinheiten sind zusätzlich multimediafähig (MM), d.h., sie erlauben SIMD-Verarbeitung durch Aufteilung von 64-Bit-Daten in 2×32 , 4×16 oder 8×8 Bit (single instruction, multiple data). Vier von ihnen sind außerdem als Speicherzugriffseinheiten ausgelegt (MEM), je zwei für Lade- und für Speichere-Befehle. Auch die beiden Gleitpunkteinheiten haben Mehrfachfunktionen. Die Kennzeichnung der Befehlsart im Bündel und damit die Zuordnung der Befehle zu den Funktionseinheiten erfolgt durch einen 5-Bit-Code (template). Mit diesem wird ggf. auch die Begrenzung der Parallelverarbeitung im Bündel angeben. Bei 41 Bit für die Befehlscodierung und 5 Bit für das Template umfaßt ein Bündel 128 Bit. – Die in der IA-64 beschriebene und dem Itanium 2 zugrundeliegende Parallelverarbeitung wird von Intel mit dem Akronym *EPIC* (*explicitly parallel instruction computing*) benannt.

Den Funktionseinheiten stehen für die Operandenspeicherung 128 allgemeine 64-Bit-Register (GP Registers) und 128 82-Bit-Gleitpunktregister (FP Registers) zur Verfügung. Bei beiden Registerspeichern werden die 32 ersten Register statisch für globale Variablen und die restlichen 96 Register dynamisch für lokale Variablen genutzt. Die dynamischen GP-Register können dazu als Register-Stack verwaltet werden (Register Stack Engine), wodurch sich die Übergabe von Parametern bei Unterprogrammaufrufen vereinfacht (siehe 2.2.1 und Bild 2-25, S. 109). Die GP-Register können aber auch als sog. rotierende Register verwaltet werden, nämlich zur Parallelisierung

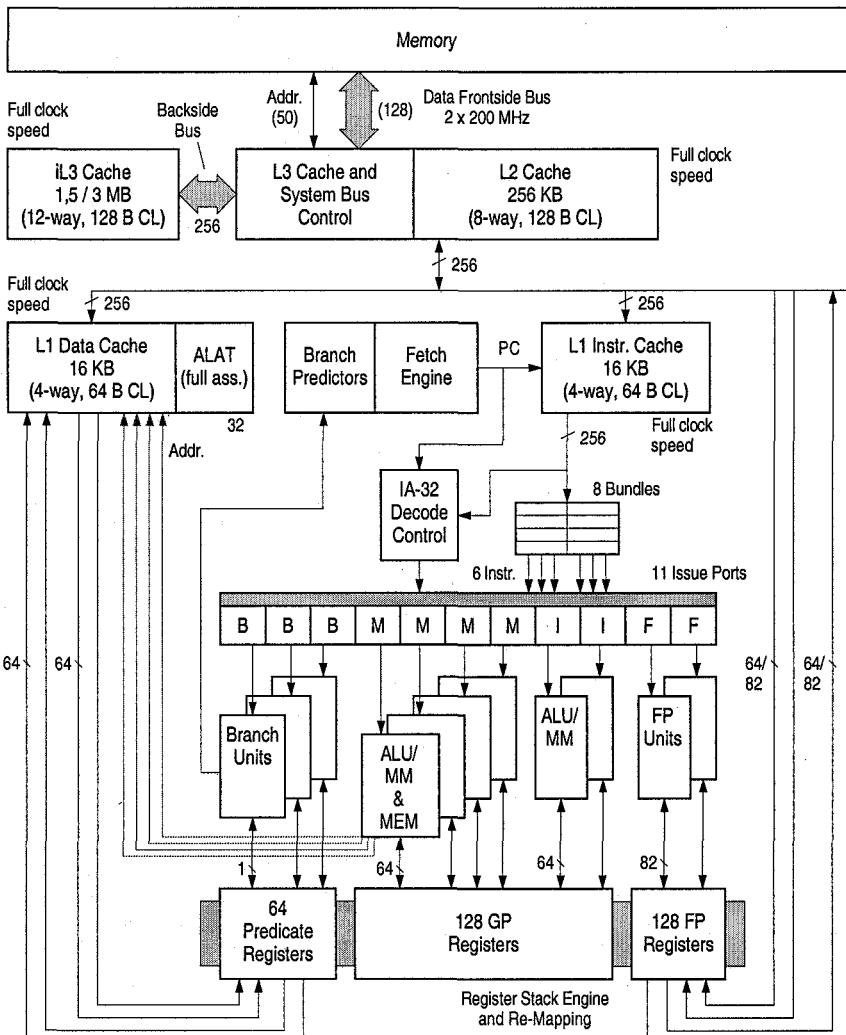


Bild 2-39. Grobstruktur des Itanium 2 (nach Intel 2002)

aufeinanderfolgender Iterationen von Programmschleifen, z.B. zur Unterstützung des Abrollens von Schleifen (loop unrolling) oder des Fließbandverarbeitens von Schleifen durch die Software (software pipelining). Hierbei werden den Befehlen der einzelnen Iteration unterschiedliche Register zugewiesen, um sie diesbezüglich zu entkoppeln (Re-Mapping). Die Rotation gibt es auch für die FP-Register, mit dem Unterschied, daß die 96 GP-Register in Vielfachen von acht (programmierbar) und die 96 FP-Register nur insgesamt (festgelegt) rotierbar sind.

Ergänzt wird der Registersatz durch 64 Flags zu Speicherung von sog. Prädikaten (Predicate Registers). Die *Prädikation*, d.h. die bedingte Ausführung von Befehlen, genauer: die bedingte Übernahme von Ergebnissen von spekulativ ausgeführten Befehlen, gehört zu den wesentlichen Merkmalen der IA-64-Architektur bzw. des Itanium 2 (siehe hierzu 2.3.7). ▲

2.3.5 Superpipelining

Der für superskalare und VLIW-Prozessoren gegenüber herkömmlichen skalaren Prozessoren typische hohe Befehlsdurchsatz basiert auf parallel arbeitenden Funktionseinheiten, die taktgleich mit Befehlen versorgt werden können, wobei durch zusätzliches Umordnen von Befehlen die einem Programm innenwohnende Befehlsparallelität möglichst gut genutzt wird. Die Funktionseinheiten selbst sind als vielstufige *Fließbänder* ausgelegt, so daß jede von ihnen tatsächlich auch im Taktabstand Befehle für die Verarbeitung übernehmen kann. Wesentlichen Einfluß auf den Befehlsdurchsatz hat hierbei die Taktfrequenz, mit der das Weiterreichen der Befehle in den Fließbändern erfolgt. Sie hängt einerseits von der Technologie ab, indem kleinere Strukturabmessungen höhere Taktfrequenzen ermöglichen. Sie hängt aber auch wesentlich vom Aufgabenumfang in der einzelnen Stufe, genauer, von den damit verbundenen Signallaufzeiten ab. Letztlich gibt hier die Stufe mit dem längsten Signalweg, d.h. mit der größten Laufzeit, die maximal mögliche Taktfrequenz vor. Man bezeichnet diesen längsten Signalweg auch als *kritischen Pfad*.

Frühe Realisierungen von Mikroprozessoren mit Fließbandverarbeitung hatten eine sehr grobe Unterteilung der Befehlsverarbeitung, d.h. wenige Fließbandstufen mit jeweils relativ umfangreichen Aufgaben. Gebräuchlich waren hier z.B. vier Stufen Fetch (F), Decode (D), Execute (E) und Write-Back (W), wie sie den bisherigen Betrachtungen in diesem Buch zugrundegelegt wurden, in erster Linie einer besseren Anschauung wegen. Diesen Aufbau wies z.B. die Realisierung des Sparc-Prozessors der Firma Cypress Ende der 1980er Jahre auf, dessen Architektur auf dem an der Universität in Berkeley Anfang der 1980er entwickelten RISC-Urahn RISC 1 basiert.

Bei späteren Mikroprozessoren wurden diese Stufen dann weiter unterteilt. Das heißt, der kritische Pfad wurde verkürzt und gleichzeitig die Stufenzahl erhöht, so daß auch die Taktfrequenz erhöht werden konnte. Die Auswirkung dieser Maßnahme zeigt Bild 2-30 (S. 125) in Teilbild d für das Fließband eines skalaren Prozessors mit gegenüber dem skalaren Prozessor in Teilbild a doppelter Stufenzahl und doppelter Taktfrequenz. Die hier gewählte Fließbandunterteilung entspricht der des MIPS-Prozessors R4000 aus den 1990er Jahren, und zwar mit einer Aufteilung des Befehls-Cache-Zugriffs auf zwei Stufen (F1, F2) und mit zusätzlichen drei Stufen für den Daten-Cache-Zugriff (C1, C2, C3). Letztere werden nur von Lade- und Speichere-Befehlen genutzt und bei arithmetischen und logischen Befehlen „leer“ durchlaufen. Solche Stufen, die nicht von jedem Befehl benötigt werden, sind üblich und sinnvoll; sie liegen in der Natur der Sache. Bei heutigen Mikroprozessoren mit mehr als nur zwei Funktionseinheiten und engerem Zuschnitt dieser Einheiten auf bestimmte Befehlsgruppen schlägt sich dieser Aspekt allerdings mehr in unterschiedlich langen Fließbändern der einzelnen Funktionseinheiten nieder (siehe unten, Beispiel 2.12).

Superskalare und VLIW-Prozessoren arbeiten inzwischen mit Fließbändern von bis zu 20 Stufen und mehr, wie Beispiel 2.12 zeigt. Man spricht bei solch langen Fließbändern auch von *Superpipelining* und *Hyperpipelining*. Sie haben zur Folge, daß der Durchlauf eines Befehls entsprechend viele Takte benötigt. Diese große *Latenzzeit* eines Befehls ist für den Befehlsdurchsatz zunächst ohne Nachteil, nämlich wenn ein solches Fließband gefüllt ist und wenn danach der Fluß im Band nicht gestört wird. Sie hat aber erhebliche Nachteile, sobald Störungen auftreten. Diese gibt es einerseits durch Programmunterbrechungen, bei denen in den Fließbandstufen bereits durchgeführte Operationen verworfen werden müssen und dann das Fließband neu gefüllt werden muß. Sie gibt es insbesondere bei Sprungabhängigkeiten, bei denen letztlich gewartet werden muß, bis eine Sprungentscheidung vorliegt. Durch die Häufigkeit von Sprungbefehlen (Größenordnung jeder siebente Befehl) hat diese Art der Störung ein hohes Gewicht, weshalb in heutigen Prozessoren aufwendige Techniken der Sprungvorhersage sowie Techniken der Sprungvermeidung angewandt werden (siehe dazu 2.3.6 und 2.3.7).

Beispiel 2.12. ► PowerPC 970 (PPC970). Bild 2-40 zeigt die Fließbandstruktur der zwölf Funktionseinheiten des PowerPC 970 (IBM), dessen Gesamtstruktur in Beispiel 2.10 (S. 142) beschrieben ist. Die Fließbandlängen dieser Einheiten liegen zwischen 7 und 16 Stufen. Hinzu kommen 9 weitere Stufen für den Befehlsabruf und die Befehlsdecodierung. Die Befehle durchlaufen also zwischen 16 Stufen (z.B. Ganzahloperationen) und 25 Stufen (Gleitpunktoperationen in der SIMD-Einheit).

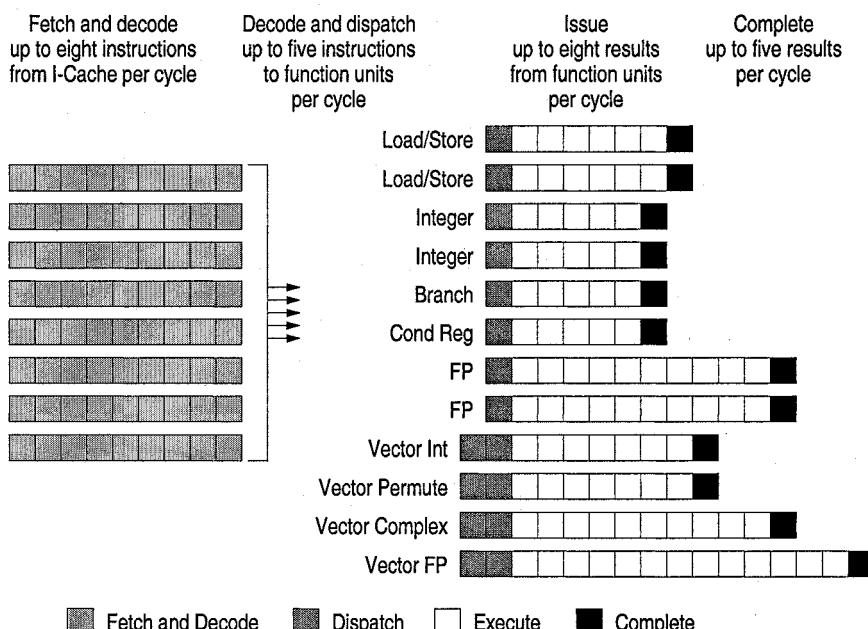


Bild 2-40. Zu Beispiel 2.12: Fließbandstruktur des PowerPC 970 (nach [Halfhill 2002]) ▶

2.3.6 Sprungvorhersage

Sprungbefehle, insbesondere bedingte Sprungbefehle, sind ein wesentliches Hemmnis in der Fließbandverarbeitung. Wie in 2.3.3 unter dem Absatz Sprungabhängigkeit beschrieben und anhand von Bild 2-33 (S. 131) in Teilbild c für ein 4-stufiges Fließband gezeigt, verursacht die Ausführung eines solchen Befehls eine Verzögerung von 2 Takten, bis weitere Befehle abgerufen werden können. (Bei den heutigen Fließbändern von 20 Stufen und mehr macht die Verzögerung natürlich mehr als 2 Takte aus!) Bei einem n -fach superskalaren Prozessor bedeutet das (bei Befehlausführungszeiten von 1 Takt) einen Verlust von $2n$ Befehlausführungen. Die Verwendung von Delay-Slots kann diesen Nachteil nur wenig mildern, da der Übersetzer diese nur im begrenzten Umfang mittels Programmumordnung füllen könnte und deshalb vielfach nop-Befehle einsetzen müßte.

Eine bessere Wirkung erreicht man hier mit *Sprungvorhersage* (*branch prediction*), die in einer sehr frühen Phase der Sprungbefehlsverarbeitung erfolgt. Mit ihr verbunden ist das *spekulative Ausführen von Befehlen* des vorhergesagten Programmzweiges, mit der Maßgabe, die Folgen dieser Ausführungen bei einer später erforderlichen Korrektur der Vorhersage beseitigen zu können. Hier kommen Techniken zum Einsatz, wie sie auch schon bei Out-of-order-Issue und In-order-Completion zur Erreichung präziser Programmunterbrechungen erforderlich sind, nämlich die der Rename-Register und des Rückordnungspuffers. Nachteil der Sprungvorhersage ist, daß eine Fehlvorhersage um so höher „bestraft“ wird, je länger das Fließband ist (misprediction penalty). Man spricht hier auch von den Kosten, die das Bereinigen einer Fehlvorhersage verursacht. Vor diesem Hintergrund wurden mit der Entwicklung immer längerer Fließbänder bei superskalaren und VLIW-Prozessoren auch die Vorhersagetechniken immer weiter verfeinert.

Die Sprungvorhersage bezieht sich auf zwei Aspekte:

1. die Vorhersage der noch ausstehenden Sprungentscheidung sowie
2. die ggf. noch zu ermittelnde effektive Sprungzieladresse.

Zur Vorhersage der Sprungentscheidung gibt es wiederum zwei grundsätzlich verschiedene Vorgehensweisen:

- a) die *statische Vorhersage*, die allein von den im Sprungbefehl gemachten Angaben ausgeht, und
- b) die *dynamische Vorhersage*, die die Vorgeschichte des Sprungbefehls, also dessen Verhalten während früherer Ausführungen des Befehls berücksichtigt (und ggf. auch noch die Vorgeschichte weiterer Sprungbefehle, die diesem Sprungbefehl vorangehen).

Statische Sprungvorhersage. Die statische Sprungvorhersage wird heute ausschließlich in Verbindung mit der dynamischen Vorhersage eingesetzt, und zwar dann, wenn es für einen Sprungbefehl noch keine Information für eine dynami-

sche Vorhersage gibt. Im einfachsten Fall geht sie davon aus, daß bei bedingten Sprüngen in der Mehrzahl der Fälle verzweigt wird. Dementsprechend lautet sie hier grundsätzlich „branch taken“, d.h., für die spekulative Befehlsausführung wird jeweils auf den durch die Sprungzieladresse vorgegebenen Programmfpad verzweigt. Eine Verbesserung dieser starren Vorhersage erreicht man, wenn man die im Sprungbefehl steckende Information mit auswertet, z.B. die Sprungrichtung. Da bedingte Sprungbefehle insbesondere zur Bildung von Programmschleifen verwendet werden und sie dabei bevorzugt am Schleifenende eingesetzt werden (*do-while*-Anweisung, 3.2.2), lautet die Vorhersageregel dann sinnvollerweise „branch taken“ bei einem Rückwärtssprung, d.h. bei negativem Displacement, und „branch not taken“ bei einem Vorwärtssprung, d.h. bei positivem Displacement. Bei n Schleifendurchläufen ist damit die Vorhersage ($n-1$)-mal erfüllt und 1-mal nicht erfüllt. Die Wahrscheinlichkeit der korrekten Vorhersage ist in diesem Idealfall entsprechend hoch.

Steht der bedingte Sprungbefehl allerdings nicht am Schleifenende, sondern am Schleifenanfang (*while*-Anweisung, 3.2.2), so kehren sich die Sprungverhältnisse um, d.h., die obige Vorhersageregel wirkt sich nachteilig aus. Abhilfe schafft hier ein Zusatzbit im Sprungbefehl, mittels dessen die von der Sprungrichtung abhängige Standard-Vorhersage negiert werden kann. Dieses Bit kann vom Übersetzer dazu genutzt werden, im Einzelfall zu entscheiden, ob von der Standard-Vorhersage abgewichen werden soll oder nicht, so auch bei Verzweigungen außerhalb von Programmschleifen.

Dynamische Sprungvorhersage. Bei der dynamischen Sprungvorhersage berücksichtigt der Prozessor das Verhalten der Sprungbefehle in der Vergangenheit, d.h. deren Sprungentscheidungen bei zurückliegenden Befehlsausführungen. Er bezieht sich dabei entweder nur auf die Vorgeschichte des aktuell vorherzusagenden Sprungbefehls, oder er bezieht zur Verfeinerung der Vorhersage die Vorgeschichte vorangegangener Sprünge mit ein, berücksichtigt also die Korrelation zwischen Sprungbefehlen. Gespeichert wird die Vorhersageinformation in einem *Branch-Prediction-Cache* (BPC), auch als *Branch-History-Table* (BHT) bezeichnet, der pro Sprungbefehl folgende Einträge vorsieht:

1. die Adresse des Sprungbefehls, oder genauer die Adresse des Blocks, in der sich der Sprungbefehl befindet und mit der der Befehlsabruf aus dem Befehls-Cache erfolgt; sie wird als Tag-Information zur assoziativen Adressierung des Cache benutzt,
2. ein Valid-Bit, das anzeigt, ob bereits ein gültiger Eintrag vorhanden ist,
3. ein oder mehrere „History-Bits“, die die zur Sprungvorhersage benötigte Vorgeschichte des Sprungbefehls repräsentieren, und
4. die zuletzt von dem Sprungbefehl benutzte effektive Zieladresse, um sie ebenfalls vorhersagen zu können (*Sprungzielvorhersage*).

Die Sprungzieladresse wird ggf. auch getrennt in einem *Branch-Target-Buffer* (BTB) oder *Branch-Target-Address-Cache* (BTAC) gespeichert. Darüber hinaus gibt es üblicherweise einen weiteren Speicher, mit dem die Rücksprungadressen von Unterprogrammen vorhergesagt werden, den sog. *Return-Address-Stack* (RAS).

Adressiert wird der Branch-Prediction-Cache üblicherweise schon in der Fetch-Stufe, und zwar mit dem PC-Inhalt, so daß eine Sprungvorhersage bereits sehr frühzeitig, nämlich bereits einen Takt nach Abruf des Sprungbefehls getroffen werden kann. Voraussetzung hierfür ist ein Cache-Hit, d.h., der Sprung muß in der Vergangenheit wenigstens einmal ausgeführt und darf inzwischen nicht wieder aus dem Branch-Prediction-Cache verdrängt worden sein. Ab hier beginnt die spekulative Befehlausführung. Sie endet mit der eigentlichen Sprungentscheidung bei Ausführung des Sprungbefehls, nach der der Prozessor ggf. seine Vorhersage fixiert (Übernahme der Ergebnisse der spekulativ ausgeführten Befehle) oder korrigiert (Verwerfen der Ergebnisse und Starten des alternativen Programmzweiges), und nach der er den Cache-Eintrag aktualisiert. Letzteres betrifft sowohl die Information zur Sprungvorhersage als auch die Zieladreßangabe. Unter welchen Bedingungen ein Cache-Eintrag verdrängt wird, hängt vom Cache-Typ (z.B. vollassoziativ oder direkt zuordnend) und von der ggf. realisierten Erstellungsstrategie ab (z. B. LRU bei vollassoziativem Cache; zur Organisation von Caches siehe 6.2).

1- und 2-Bit-Prädiktoren. Im einfachsten Fall von nur einem History-Bit (*1-Bit-Prädiktor*) läßt sich die dynamische Sprungvorhersage allein aus der letzten Ausführung des Sprungbefehls treffen, und zwar mit der Vorhersageregel „branch taken“, wenn bei der vorangegangenen Ausführung gesprungen wurde und „branch not taken“, wenn bei der vorangegangenen Ausführung nicht gesprungen wurde. Sie läßt sich durch den Zustandsgraphen in Bild 2-41a veranschaulichen. Dieses einfache Verfahren führt allerdings zu vielen Fehlvorhersagen, so nämlich bei jeder Änderung des Sprungverhaltens eines Sprungbefehls, weshalb es nicht mehr gebräuchlich ist.

Ein verbesserte Vorhersage erhält man, wenn man sie mit einer gewissen Trägheit behaftet, wenn also die Vorhersage nicht gleich nach einem einmaligen Wechsel der Sprungentscheidung mitwechselt. Gedacht ist dabei z.B. an ineinander geschachtelte Programmschleifen und den jeweiligen Neuanfang der Indizierung in der inneren Schleife. Eine solche Vorhersage umfaßt üblicherweise vier Zustände, die durch zwei History-Bits pro Sprungbefehleintrag im Branch-Prediction-Cache realisiert werden (*2-Bit-Prädiktor*). Diese bilden einen 2-Bit-Zähler, der mit jedem ausgeführten Sprung inkrementiert wird, bis zum maximalen Wert 3, und der mit jedem nicht ausgeführten Sprung dekrementiert wird, bis zum minimalen Wert 0.

Die Auswertung des Zählerwerts kann in unterschiedlicher Weise erfolgen. Gängig ist hier die Auslegung des Zählers als sog. *Sättigungszähler*, wie durch den

Zustandsgraphen in Bild 2-41b veranschaulicht. Vorhergesagt wird „taken“ bei den Werten 3 und 2 und „not taken“ bei den Werten 1 und 0. Ausgehend vom Wert 3 heißt das: Erst nach zweimaligem Nichtspringen wird „not taken“ vorhergesagt. Entsprechendes gilt für das Nichtspringen, vom Wert 0 ausgehend. Dementsprechend werden die vier Zustände auch mit „strongly taken“, „weakly taken“, „weakly not taken“ und „strongly not taken“ bezeichnet. Beeinflußt werden kann die Vorhersage durch einen günstigen Initialisierungswert. Als solcher wird 3 angesehen, da bei ca. 60 bis 70% der Ausführungen bedingter Sprungbefehle der Sprung tatsächlich durchgeführt wird.

In einer Variante dieses Verfahrens wird „taken“ bei den Werten 3, 2 und 1 und „not taken“ beim Wert 0 vorhergesagt. Ausgehend vom Wert 3 heißt das: Erst nach dreimaligem Nichtspringen wird „not taken“ vorhergesagt. Ausgehend vom Wert 0 heißt das: Bereits nach einmaligem Springen wird wieder „taken“ vorhergesagt. Hier wird also der größeren Häufigkeit von Verzweigungen Rechnung getragen. Bei einer weiteren Variante, dem sog. *Hysteresis-Schema*, ist der Zustandsgraph gegenüber Bild 2-41b derart verändert, daß ausgehend vom Zustand „weakly taken“ nach erfolgtem Nichtspringen direkt in den Zustand „strongly not taken“ und ausgehend vom Zustand „weakly not taken“ bei erfolgtem Sprung direkt in den Zustand „strongly taken“ übergegangen wird.

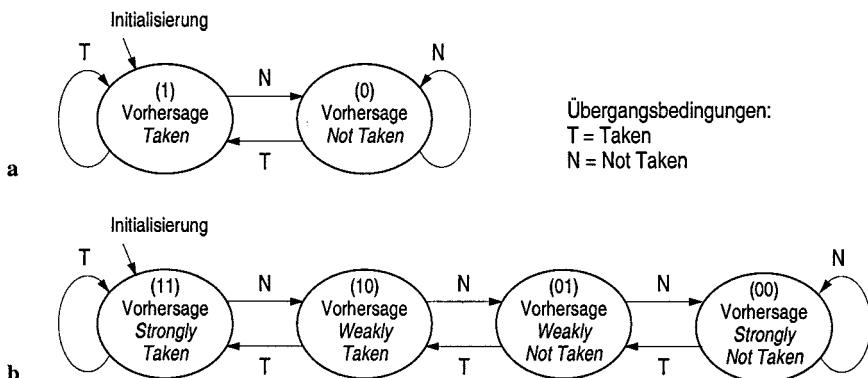


Bild 2-41. Dynamische Sprungvorhersage durch **a** 1-Bit-Prädiktor, **b** 2-Bit-Prädiktor als Sättigungszähler

Adaptiver Prädiktor. Die Sprungvorhersage mittels der beschriebenen Zähler liefert für viele der in Programmen vorkommenden Verzweigungssituationen, z.B. bei Schleifen, gute Ergebnisse. Würde man diese Zähler auf mehr als vier Zustände, also mehr als zwei Bits erweitern, so würden sich die Ergebnisse in den meisten Fällen nicht signifikant verbessern. Die Zählerverfahren eignen sich allerdings nicht so gut bei unregelmäßigem Sprungverhalten. Um hier bessere Ergebnisse zu erhalten, werden *adaptive Vorhersageverfahren* eingesetzt, d.h. Verfahren, bei denen für die Vorhersage mehrere aufeinanderfolgende Sprungentscheidungen in Form von Sprungmustern (T/N-Folgen) ausgewertet werden.

Dazu wird z.B. für jeden Sprungbefehl eine eigene Sprungmustertabelle angelegt, in der zu jedem möglichen Muster (Tabellenindex) die Vorhersage gespeichert wird. Bei Sprungmustern beispielsweise der Länge drei gibt es entsprechend den Varianten NNN, NNT, ..., TTT pro Tabelle acht solcher Vorhersageeinträge T bzw. N. Sämtliche Tabellen sowie die zu jeder Tabelle gehörenden jeweils drei letzten Sprungentscheidung werden in einem Cache gespeichert. Bei einer Fehlvorhersage wird der dem ausgewerteten Muster zugeordnete Vorhersageeintrag korrigiert.

Korrelationsprädiktor. Eine weitere Verbesserung der Vorhersage erreicht man, wenn man nicht nur das Verhalten des vorherzusagenden Sprungbefehls auswertet, sondern zusätzlich noch die Wechselbeziehungen dieses Sprungbefehls zu anderen Sprüngen berücksichtigt, d.h., wenn man die globale Vergangenheit auswertet. Das führt zu *adaptiven Korrelationsverfahren*. Beispiele hierfür sind die (m,n) -*Prädiktoren*. Sie benutzen für die Vorhersage eines Sprungbefehls das Verhalten der letzten m Sprünge und wählen damit einen von 2^m zu diesem Sprungbefehl gehörenden Prädiktoren aus, wobei jeder dieser Prädiktoren wiederum als n -Bit-Prädiktor ausgelegt ist.

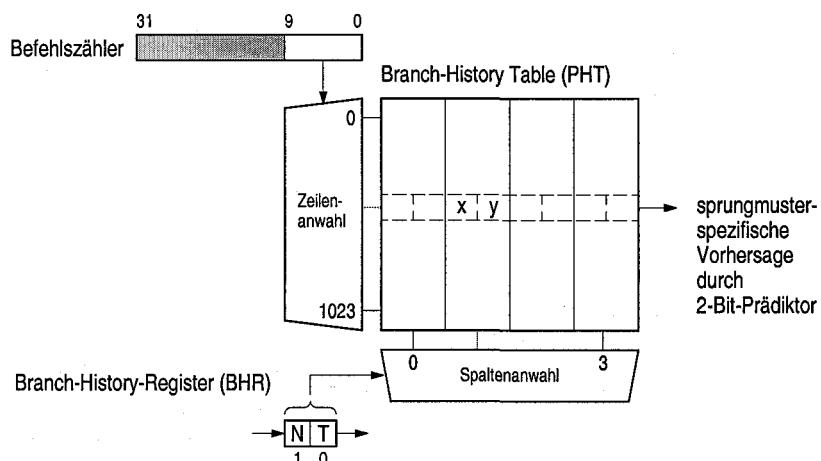


Bild 2-42. Dynamische Sprungvorhersage durch einen (2,2)-Prädiktor

Bild 2-42 veranschaulicht dieses Verfahren am Beispiel eines *(2,2)-Prädiktors*. Ausgewertet werden hier die beiden jeweils letzten Sprungentscheidungen ($m=2$), die dazu als Sprungmuster in einem 2-Bit-Schieberegister, dem *Branch-History-Register* (BHR) gespeichert werden (adaptives Verfahren). Die Vorhersageinformation selbst steht in einer Branch-History-Table mit zeilenweisen Einträgen für die einzelnen Sprungbefehle und spaltenweiser Zuordnung der Information zu den 4 möglichen Sprungmustern ($2^m=4$). Die Vorhersageinformation eines jeden Kreuzungspunktes wird mittels eines 2-Bit-Prädiktors erzeugt ($n=2$), z.B. eines Sättigungszählers (siehe Bild 2-41b). Adressiert werden die Zeilen, um

die Zeilenanzahl zu begrenzen, mit den z.B. 10 niederwertigen Bits des Befehlszählers. Dabei kann es zum gegenseitigen Verdrängen von Sprungbefehleinträgen kommen. – In der Art, wie auf die Branch-History-Table zugegriffen wird, gibt es Verfahrensvarianten mit Bezeichnungen wie *Gselect*- und *Gshare-Prädiktor* (G steht für global).

2.3.7 Sprungvermeidung

Fehlentscheidungen bei der Sprungvorhersage haben bei langen Fließbändern den Nachteil hoher Strafen an verlorenen Taktzyklen in der Fließbandverarbeitung. Um diese bestmöglich zu mindern, werden hochentwickelte Vorhersage-techniken mit entsprechend hohem Hardwareaufwand für die Speicherung von Vorhersagetabellen und für die erforderliche Steuerlogik eingesetzt. Ganz verhindern lassen sich diese Strafen allerdings nur dann, wenn man bedingte Sprungbefehle von vornherein vermeidet, indem man sie durch andere Programmkonstrukte ersetzt. Hierzu bedarf es der Unterstützung durch die Prozessorhardware in Form einer erweiterten Funktionalität von Befehlen.

Bedingte Befehle. Eine Möglichkeit der Sprungvermeidung bieten *bedingte Move-Befehle*, wie sie in den Befehlssätzen verschiedener Mikroprozessoren vorhanden sind. Ein solcher Move-Befehl führt seine Transportoperation nur dann aus, wenn eine im Befehl angegebene Bedingung erfüllt ist. Das heißt, die sonst ggf. durch einen bedingten Sprungbefehl auszuwertende Bedingung wird in die davon betroffene(n) Operation(en) verlagert. Dazu als Beispiel folgende if-else-Anweisung in C (zu Programmverzweigungen in C und in Assembler siehe 3.2.1):

```
if (a > b) x = 1; else x = 0;
```

Sie lässt sich für den in 2.2.4 beschriebenen RISC-Befehlssatz, der an die Sparc-Architektur angelehnt ist, unter Verwendung der Registerzuordnungen a: i0, b: i1 und x: i2 in herkömmliche Weise, d.h. mit Hilfe eines bedingten Sprungbefehls bcc (Sprungbedingung cc = g, greater than) wie folgt in Assembler formulieren:

```
cmp  i0,i1      ; a > b ?
bg   L1          ; springe, wenn a > b
or   g0,1,i2    ; x = 1 (Delay-Slot-Befehl!)
or   g0,0,i2    ; x = 0
L1: :
```

Für neuere Sparc-Implementierungen, z.B. den *UltraSparc IIIi* der Firma Sun, mit Erweiterung des Befehlssatzes um den bedingten Move-Befehl movcc, lässt sich die if-else-Anweisung auf der Assemblerebene wie folgt modifizieren:

```
cmp  i0,i1      ; a > b ?
or   g0,0,i2    ; x = 0
movg %icc,g0,1,i2 ; überschreibe x mit 1, wenn a > b
```

Der Vergleichsbefehl cmp beeinflußt hierbei die Integer-Bedingungsbits icc, die dann vom bedingten Move-Befehl movg ausgewertet werden. Findet dieser die Bedingung „greater than“ als erfüllt vor, führt er seine Transportoperation aus, sonst nicht. Ein bedingter Sprungbefehl ist hier also nicht erforderlich.

Bei einer Variante des bedingten Move-Befehls, movrcc, kann anstelle der Bedingungsbits ein Register angegeben werden und als Bedingung für die Befehlausführung dessen Inhalt zum Wert Null in Bezug gesetzt werden, so wie im folgenden Beispiel für das Register i2 und die Bedingung „less than or equal zero“ gezeigt:

```
movrlez i2,i4,i5      ; wenn i2 ≤ 0, dann i5 = i4
```

Im Prinzip ist es mit den bedingten Move-Befehlen möglich, die Befehle der beiden Zweige einer Programmverzweigung zu einer parallelisierbaren Befehlsfolge zusammenzufassen, um so auf den bedingten Sprungbefehl verzichten zu können. Allerdings geht das nur dann, wenn diese Zweige ausschließlich Move-Befehle enthalten. In diesem Fall werden die Move-Befehle mit komplementären Bedingung versehen, so daß jeweils nur die Befehle eines der beiden Zweige tatsächlich Transportoperationen durchführen. Hierzu als Beispiel eine if-else-Anweisung, in der (anders als im obigen Beispiel) anstelle von Variablennamen gleich die Adressen der verwendeten Local- und Input-Register angegeben sind:

```
if (i0 == 0) {
    i2 = i2;
    i3 = i3;
} else {
    i5 = i5;
    i6 = i6;
}
```

In der Sparc-Assemblerschreibweise entspricht ihr folgende sprungbefehlsfreie Befehlsfolge mit den Move-Bedingungen „equal zero“ und „not equal zero“:

```
movrz  i0,i2,12      ; wenn i0 equal zero, dann 12 = i2
movrz  i0,i3,13      ; wenn i0 equal zero, dann 13 = i3
movrnz i0,i5,15      ; wenn i0 not equal zero, dann 15 = i5
movrnz i0,i6,16      ; wenn i0 not equal zero, dann 16 = i6
```

Prädikation. Ein erweiterte Möglichkeit bedingter Befehlausführung ist die der Prädikation (predication), wie sie z.B. für die *IA-64-Architektur (EPIC)* beschrieben und im *Itanium 2* realisiert ist. Hier sind nicht nur Move-Befehle, sondern auch arithmetische und logische Befehle bedingt ausführbar, wozu sie mit einem sog. Prädikat versehen werden. Prädikate sind 1-Bit-Informationen, die durch Vergleichsbefehle gesetzt werden und die mit „1“ eine erfüllte Bedingung und mit „0“ eine nichterfüllte Bedingung signalisieren. Für den mit einem Prädikat versehenen Befehl heißt das, daß er zunächst ausgeführt wird und es sich ggf. erst später anhand des inzwischen ermittelten Prädikatwerts entscheidet, ob das von

ihm erzeugte Ergebnis als gültig in das Zielregister übernommen werden kann (Prädikat = 1), oder ob es verworfen werden muß (Prädikat = 0).

Diese Technik erlaubt es ebenfalls, die bei einem bedingten Sprungbefehl zu unterscheidenden Programmzweige beide auszuführen, um dann aber im Nachhinein die Ergebnisse des fälschlicherweise ausgeführten Zweiges zu ignorieren. Dazu wird das Programm wie folgt aufgebaut:

1. Der dem Sprungbefehl vorangehende Vergleichsbefehl wird dazu genutzt, anhand des Vergleichsergebnisses ein Prädikat sowie ein dazu komplementäres Prädikat zu setzen.
2. Auf den bedingten Sprungbefehl wird verzichtet.
3. Die Befehle beider Programmzweige werden mit dem im Vergleichsbefehl angegebenen Prädikat bzw. dem dazu komplementären Prädikat versehen.

Dazu als Beispiel folgende if-else-Anweisung

```
if (r1 != 0) r2 = r3 + r4; else r5 = r6 - r7;
```

und deren Umsetzung in den Assemblercode des Itanium 2:

```
cmp.ne p1,p2 = r1,r0;;
(p1) add      r2 = r3,r4
(p2) sub      r5 = r6,r7
```

Der Vergleichsbefehl cmp.ne überprüft hier die Bedingung „r1 not equal zero“ (in r0 steht RISC-typisch die Konstante 0) und setzt bei erfüllter Bedingung das Prädikat p1 mit dem Wert 1 und das Prädikat p2 mit dem dazu komplementären Wert 0. Bei nichterfüllter Bedingung werden die Wertzuweisungen vertauscht. Von den Befehlen add und sub der beiden Programmzweige erzeugt letztlich nur derjenige ein gültiges Ergebnis, dessen Prädikat den Wert 1 aufweist. – In Erweiterung des Beispiels kann jeder der beiden Programmzweige aus mehr als nur einem Befehl bestehen, wobei die Prädikate der Befehle auch verschieden sein können, indem mehrere Vergleichsbefehle zur Anwendung kommen. Der Itanium 2 stellt hierfür insgesamt 64 1-Bit-Prädikatregister zur Verfügung (siehe Bild 2-39, S. 146).

Fazit. Bedingte Befehle und die Prädikation bieten zwar den Vorteil, ggf. auf bedingte Sprungbefehle verzichten und damit Fehlspirationen bei einer Sprungvorhersage vermeiden zu können; sie haben aber auch einen Nachteil: Die Befehle des jeweils nichtrelevanten Programmzweiges belegen Ressourcen, wodurch ggf. relevante Befehle in ihrer Ausführung verzögert werden. Diese Ressourcen sind z.B. Befehlspuffer und Operandenregister, insbesondere aber die Funktionseinheiten, die zur Befehlausführung benötigen werden.

Weitere Techniken zur Sprungvermeidung. Neben den bedingten Move-Befehlen und der Prädikation gibt es weitere Techniken zur Sprungvermeidung, die auf die Verarbeitung von Programmschleifen abzielen, nämlich das Abrollen von Schleifen und das sog. Software-Pipelining. Bei beiden Techniken, die hier nur

angedeutet werden sollen, wird die Befehlsparallelität dadurch erhöht, daß die Schleifenrumpfbefehle aus mehreren Schleifeniterationen sprungbefehlsfrei verarbeitet werden und dabei die Unabhängigkeit zwischen den Befehlen verschiedener Iterationen für die Parallelarbeit genutzt werden kann. Beim *Abrollen von Schleifen* (*loop unrolling*) geschieht das dadurch, daß die Befehle mehrerer Iterationen zu einer einzigen größeren Iteration zusammengefaßt werden. Beim *Software-Pipelining* wird der Schleifenrumpf softwareseitig in Stufen unterteilt, von denen jede einen, mehrere oder auch keinen Befehl des Rumpfes umfassen kann. Die Rümpfe der aufeinanderfolgenden Iterationen werden im Sinne der Fließbandverarbeitung stufenweise versetzt zueinander ausgeführt, so daß auch hier wieder Befehle aus mehreren Iterationen parallel verarbeitet werden können. Bei beiden Techniken sind wegen der Parallelverarbeitung u.a. Konflikte bzgl. der Registerzuweisungen an die Befehle der Schleifenrümpfe der einzelnen Iterationen zu lösen, was durch *Register-Renaming* geschieht.

Zu diesen Techniken siehe z.B. die *IA-64-Architektur*. Sie unterstützt diese Techniken durch Hardware, so z.B. das Register-Renaming durch das Rotieren von Registern, d.h. durch die Möglichkeit der „Entnahme“ von Registern aus dem Registerspeicher in Wrap-around-Manier, wie im *Itanium 2* realisiert (siehe auch Beispiel 2.11, S. 145).

2.3.8 Vielfältige Verarbeitung (multithreading)

Die Parallelarbeit bei superskalaren und VLIW-Prozessoren ist dadurch geprägt, daß mehrere Funktionseinheiten, die als Fließbänder ausgelegt sind, taktgleich mit Befehlen versorgt werden und diese Befehle dementsprechend parallel ausgeführt werden. Hemmnisse dieser Parallelarbeit sind Daten-, Betriebsmittel- und Sprungabhängigkeiten, die – wie zuvor in diesem Abschnitt beschrieben – durch Out-of-Order-Verarbeitung, Sprungvorhersage, bedingte Befehle und Prädikation in ihren Auswirkungen zwar gemildert werden können, die jedoch immer wieder dazu führen, daß Lücken (*stalls*) in der Fließbandverarbeitung entstehen. Das heißt, die verfügbaren Funktionseinheiten können trotz vielfältiger Techniken zur Optimierung des Befehlsflusses aufgrund nicht ausreichender Befehlsparallelität im Programm nicht optimal genutzt werden. Bild 2-43a demonstriert dies für einen vierfach superskalaren Prozessor, d.h. für einen Prozessor, bei dem vier Befehle eines Programms (das im Vorgriff als Thread 1 bezeichnet wird) taktgleich der Verarbeitung zugeführt werden können. Die Taktgleichheit der Befehlsabgabe an die Funktionseinheiten wird jeweils durch die Kästchen in der Waagerechten, das zeitliche Aufeinanderfolgen der Befehlsabgaben durch die Kästchen in der Senkrechten dargestellt. Da der Prozessor mehr als nur vier Funktionseinheiten haben kann, kommen in einer Spalte ggf. unterschiedliche Funktionseinheiten zum Zuge. Das Verhältnis von grauen Kästchen (Befehlsabgabe möglich) zu wei-

ßen Kästchen (Befehlsabgabe nicht möglich) charakterisiert den Befehlsdurchsatz.

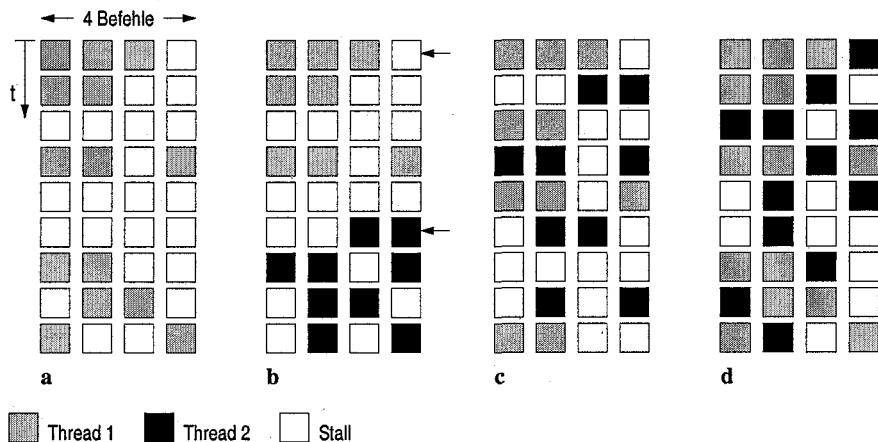


Bild 2-43. Multithreading auf der Basis eines vierfach superskalaren Prozessors: a ohne Multithreading, b blockweises Multithreading, c taktweises Multithreading, d simultanes Multithreading (SMT), auch als Hyperthreading (HT) bezeichnet

Um den Befehlsdurchsatz, d.h. die Auslastung der Funktionseinheiten zu verbessern, müssen diese mit Befehlen versorgt werden, die noch stärker voneinander unabhängig sind, als dies bei einer Befehlsfolge entlang des Programmflusses der Fall ist. Hierzu unterteilt man das Programm bereits bei seiner Erstellung in voneinander unabhängige Abschnitte, sog. *Kontrollfäden (threads)* und führt die Befehle dieser Abschnitte bzw. Threads entweder in geeigneter Weise zeitversetzt oder sogar gleichzeitig der Verarbeitung durch den Prozessor zu. Man bezeichnet diese Verarbeitungsform als *vielfädige Verarbeitung (multithreading)*; Aussagekräftiger wäre es, den englischen Begriff *thread* nicht direkt zu übersetzen, sondern von *Kontrollpfad* und *mehrpfadiger Verarbeitung* zu sprechen.)

Blockweises Multithreading. In einer „groben“ Vorgehensweise werden die Threads (ähnlich dem Multiprogrammbetrieb) zeitversetzt zueinander ausgeführt und immer dann ein Thread-Wechsel vorgenommen, wenn ein Ereignis auftritt, das die Fließbandverarbeitung für längere Zeit aufhält. Beispiel für ein solches Ereignis ist ein Cache-Miss. Die durch ihn für den erforderlichen Speicherzugriff hervorgerufene Wartezeit (bis zu 100 Takte und mehr!) kann so durch Ausführen der Befehle eines anderen Threads überbrückt werden. Man bezeichnet dies als *blockweises Multithreading* (block interleaving, switch-on-event multithreading) mit der Einordnung als *grobkörniges Multithreading (coarse-grained)*. – Bild 2-43b zeigt dazu als Beispiel die Belegung der Funktionseinheiten eines vierfach superskalaren Prozessors, hier mit der Aufeinanderfolge von Befehlen des Threads aus Teilbild a und Befehlen eines zweiten Threads. Die seitlichen Pfeile

geben die Aufsetzzeitpunkte beider Threads an. Diese Verarbeitung kann auf quasi beliebig viele Threads ausgedehnt werden.

Taktweises Multithreading. Bei einer „feineren“ Vorgehensweise erfolgen die Thread-Wechsel sehr viel enger verzahnt und ereignisunabhängig. Hier wechseln sich zwei oder mehr Threads grundsätzlich taktweise in der Befehlsabgabe an die Funktionseinheiten ab, was es dem einzelnen Thread wiederum ermöglicht, ggf. zwischenzeitlich Abhängigkeiten zwischen seinen Befehlen aufzulösen, wodurch Leertakte in den Fließbändern (stalls) entfallen. Man bezeichnet dies als *taktweises Multithreading* (*cycle-by-cycle interleaving, time-slicing multithreading*) mit der Einordnung als *feinkörniges Multithreading* (*fine-grained*). – Bild 2-43c zeigt hierzu als Beispiel die Belegung der Funktionseinheiten eines vierfach superskalaren Prozessors mit den Befehlen der beiden Threads aus Teilbild b. Wie zu erkennen ist, kann bei Thread 1 zweimal ein Leertakt vermieden werden (dritte und fünfte Zeile in Teilbild a). Um auch den zweiten der beiden aufeinanderfolgenden Leertakte von Thread 1 vermeiden zu können (sechste Zeile in Teilbild a), wäre ein taktweises Multithreading mit wenigstens drei Threads erforderlich.

Simultanes Multithreading. Ein verfeinertes feinkörniges Multithreading mit noch besserer Nutzung der Funktionseinheiten ist das sog. *simultane Multithreading* (*simultaneous multithreading, SMT*). Bei diesem Verfahren werden die Befehlausführung von zwei oder auch mehr Threads bereits innerhalb eines Taktes miteinander gemischt, so daß Funktionseinheiten, die bei den anderen Multithreading-Verfahren ungenutzt bleiben würden, mitbenutzt werden können. Hier wird also die Befehlsparallelität (*instruction-level parallelism, ILP*), wie sie den superskalaren und VLIW-Prozessoren zugrunde liegt, mit der Thread-Parallelität (*thread-level parallelism, TLP*) zusammengeführt, und somit quasi die Parallelität auf der Befehlsebene insgesamt erhöht. – Bild 2-43d zeigt dazu die Belegung der Funktionseinheiten eines vierfach superskalaren Prozessor, der jetzt die beiden Threads aus Teilbild c taktweise gemischt ausführt. Diese Art des Multithreading ist z.B. im *Pentium 4* realisiert, und zwar für zwei Threads, was von Intel als *Hyperthreading* bezeichnet wird.

Kontextwechsel. Der beim Thread-Wechsel zu wechselnde Kontext hat einen nur geringen Umfang, weshalb der im Prozessor hierfür zusätzlich erforderliche Hardwareaufwand relativ gering ist. Betroffen sind im wesentlichen die Inhalte des Befehlszählers, des Prozessorstatusregisters und die Inhalte der für den Programmierer sichtbaren allgemeinen Prozessorregister. Diese Register müssen entsprechend der verwaltbaren Thread-Anzahl mehrfach vorhanden sein. Die Renaming-Register hingegen gibt es nur einmal; sie stehen in ihrer Gesamtheit allen Threads zur Verfügung. Man spricht aufgrund der Registervervielfachung auch von *virtuellen Prozessoren*, beim Hyperthreading des Pentium 4 dementsprechend von zwei virtuellen Prozessoren. In Anlehnung an die (*schwergewichtigen*) *Prozesse* (*tasks*) des Multiprogrammbetriebs bezeichnet man Threads auch als *leichtgewichtige Prozesse*. Während schwergewichtige Prozesse als ggf. sehr um-

fangreiche Programme einen sehr viel größeren Kontext haben, so z.B. jeweils einen eigenen Adressraum, der mittels einer Speicherverwaltungseinheit (MMU) verwaltet wird, teilen sich die Threads einen Adressraum, z.B. den eines Prozesses.

Chip-Multiprocessing. Um den Befehlsdurchsatzes bei superskalaren Prozessoren zu erhöhen, wird als Alternative oder Ergänzung zum Multithreading das sog. *Chip-Multiprocessing (CMP)* eingesetzt. Hierzu wird der Prozessorchip mit zwei oder mehr Prozessorkernen ausgestattet, die parallel arbeiten. Gegenüber herkömmlichen Prozessoren haben sie eine geringere Komplexität, so z.B. eine geringere Superskalarität, mit dem Vorteil, daß die insgesamt auf dem Baustein vorhandenen Funktionseinheiten besser ausgelastet werden können. Darüber hinaus werden die Prozessorkerne, anders als die Prozessoren in symmetrischen Mehrprozessorsystemen, sehr eng miteinander gekoppelt und erlauben einen sehr schnellen Informationsabgleich. So benutzen sie z.B. den L1-Cache gemeinsam. In einfacher Organisationsform läuft auf jedem dieser Prozessorkerne ein eigener Thread (CMP als Alternative zum Multithreading), in komplexerer Organisationsform arbeitet jeder der Kerne mit z.B. simultanem Multithreading (CMP als Ergänzung zum Multithreading). – Zu den erwähnten symmetrischen Mehrprozessorsystemen (*symmetrical multiprocessing, SMP*), deren Programme die schwergewichtigen Prozesse sind, und deren Parallelverarbeitung auf einer höheren Abstraktionsebene als der des Chip-Multiprocessing erfolgt, siehe 5.1.7.

3 Assemblerprogrammierung mit C-Entsprechungen

Die Programmierung von Mikroprozessorsystemen ist, wenn diese Systeme in einem technischen Umfeld eingesetzt werden (Steuerung technischer Systeme und Geräte, embedded control), üblicherweise eine hardware-nahe Programmierung und wird entweder ganz in Assemblersprache oder aber in einer „höheren“ Programmiersprache, meist C, ggf. ergänzt um Assemblersequenzen, durchgeführt. Um dabei effiziente Programme erstellen zu können, ist es unabdingbar, grundlegende Techniken zu kennen, mit denen immer wiederkehrende Verarbeitungsabläufe optimal beschrieben werden können. Solche Programmierungstechniken sind zunächst weitgehend unabhängig vom Befehlssatz des eingesetzten Prozessors. Bei ihrer Umsetzung in Befehlsfolgen ergeben sich jedoch prozessor-abhängige Unterschiede bezüglich des Bedarfs an Programmspeicherplatz, an Programmausführungszeit und auch hinsichtlich ihrer Unterstützung durch den Befehlssatz und durch andere Programmierspezifika des Prozessors.

In diesem Kapitel stellen wir solche grundlegenden Programmierungstechniken vor, und zwar von ihrer Formulierung in Assemblersprache ausgehend. Wir legen dabei zunächst den CISC-Befehlssatz aus Abschnitt 2.1 zugrunde, ergänzen dann aber unsere Ausführungen um RISC-spezifische Programmierungstechniken anhand des Befehlssatzes aus Abschnitt 2.2. Zusätzlich wollen wir dabei zeigen, inwieweit sich die hardware-nahen Assemblersequenzen in C nachbilden lassen, wozu wir zu den meisten Assembler-Programmbeispielen auch eine C-Entsprechung hinzufügen (diese muß nicht immer die einzige mögliche sein). Dies ist auch gedacht als Vorbereitung auf Kapitel 4, in dem wir Techniken der maschinennahen Programmierung in C vorstellen.

In Abschnitt 3.1 werden zunächst einige prinzipielle Möglichkeiten zur Darstellung von Datenverarbeitungsabläufen angegeben, die insbesondere zur Erstellung von Assemblerprogrammen geeignet sind. Die in Kapitel 1 definierte und in Kapitel 2 ergänzte Assemblersprache wird dazu in ihrer Funktionalität an die gängigen Assemblersprachen von 32-Bit-CISC-Mikroprozessoren angepaßt. Abschnitt 3.2 beschreibt verschiedene Möglichkeiten der Programmflußsteuerung durch Sprungbefehle in Form von Verzweigungen und Programmschleifen. Sie werden in Abschnitt 3.3 durch Unterprogrammtechniken ergänzt, wobei verschiedene Möglichkeiten der Parameterübergabe beschrieben werden. Grundlage dieser ersten drei Abschnitte ist der in Kapitel 2 beschriebene MC680x0-ähnliche CISC-Prozessor. Abschnitt 3.4 geht dann auf die Besonderheiten der RISC-Programmierung ein, insbesondere auf die Lade-/Speichere-Problematik, den Unter-

programmanschluß und die Unterbrechungsprogrammierung. Grundlage hierfür ist der in 1.4 eingeführte und in 2.2 genauer beschriebene, an den SPARC angelehnte RISC-Prozessor.

3.1 Assemblersprache

3.1.1 Algorithmendarstellung

Algorithmus und Programm. Eine Verarbeitungsvorschrift, nach der Eingabedaten über Zwischenergebnisse in Ausgabedaten umgewandelt werden, bezeichnet man als *Algorithmus*. Die dem Mikroprozessor angepaßte Beschreibung eines Algorithmus ist das *Programm*. Liegt das Programm in *Maschinencode* vor, so kann es vom Mikroprozessor unmittelbar interpretiert werden; liegt es in symbolischer Form vor, z.B. als *Assemblerprogramm* oder in einer höheren Programmiersprache, so muß es in einem vorbereitenden Arbeitsgang erst in den Maschinencode umgeformt werden. Dazu ist ein Übersetzungsprogramm (Assembler bzw. Compiler) notwendig.

Zur Erleichterung der Programmerstellung wird ein Algorithmus zunächst in einer prozessorunabhängigen und für den Menschen besser verständlichen und überschaubaren Form beschrieben. Dies ist nicht nur bei der Programmierung in höheren Programmiersprachen, sondern auch bei der Programmierung auf relativ niedriger Sprachebene, wie der Assemblerebene, nützlich und bei umfangreichen Aufgabenstellungen unabdingbar. Die Beschreibung kann entweder sprachlich orientiert erfolgen, z.B. durch Texte der Umgangssprache (vgl. [Knuth 1961]) oder in einer höheren Programmiersprache (z.B. C, PASCAL); oder sie kann grafisch orientiert erfolgen, z.B. durch Struktogramme (vgl. [Nassi, Schneidermann 1973]) oder durch Programmablaufpläne (DIN 66001). Programmablaufpläne werden auch als Ablaufdiagramme oder Flußdiagramme bezeichnet.

Struktogramme und Flußdiagramme. Struktogramme sind an die Beschreibungselemente höherer Programmiersprachen, insbesondere an deren Kontrollstrukturen zur Programmsteuerung (Programmflußstrukturen) angelehnt. Die Beschreibungsebene ist damit höher als die der Assemblersprache und eignet sich zur komprimierten Darstellung komplexer Abläufe. Struktogramme unterstützen darüber hinaus eine strukturierte Problembeschreibung und erleichtern das Schreiben entsprechend strukturierter Programme [Schnupp, Floyd 1979]. Sie haben allerdings den Nachteil, daß in ihnen nachträgliche Änderungen schlecht durchzuführen sind. Flußdiagramme sind verglichen mit Struktogrammen mehr an die Programmdarstellung auf Assemblerebene angelehnt. Programmflußstrukturen, wie z.B. eine *do-while*-Schleife, lassen sich in aufgelöster Form darstellen. Flußdiagramme spiegeln dadurch den tatsächlichen Programmfluß mit allen sei-

nen Verzweigungen im Detail wider. Aufgrund ihres Aufbaus aus Einzelsymbolen sind sie leichter änderbar, haben dafür aber eine größere räumliche Ausdehnung, was ihre Übersichtlichkeit einschränkt. – Bild 3-1 zeigt dazu als Beispiel einen Algorithmus zur Summation der natürlichen Zahlen 1 bis N in beiden Darstellungen.

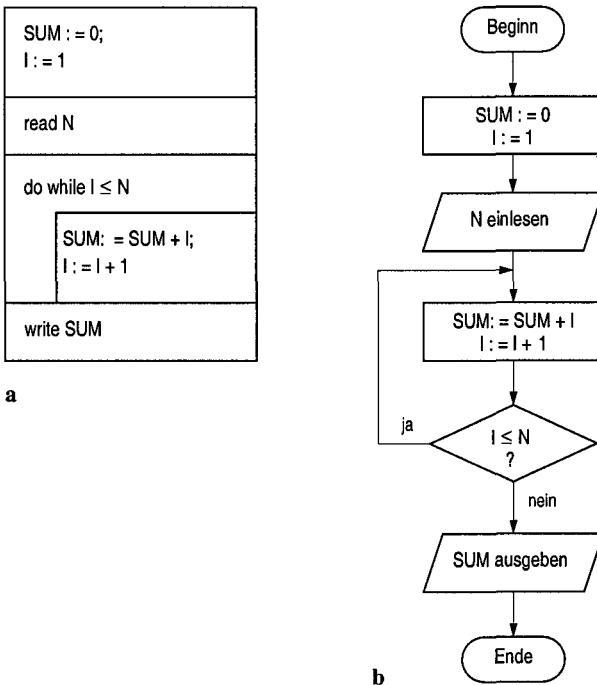


Bild 3-1. Algorithmdarstellungen am Beispiel einer *do-while*-Schleife, a Struktogramm, b Flußdiagramm

3.1.2 Assemblersyntax und Assembleranweisungen

In Kapitel 1 wurde eine einfache Assemblersprache eingeführt, die auf den dort beschriebenen einfachen CISC-Prozessor zugeschnitten ist. Wir wollen diese Sprache erweitern, indem wir zum einen die in Kapitel 2 beschriebenen CISC-Prozessorfunktionen berücksichtigen; ein erster Schritt in dieser Richtung wurde bereits in 2.1.3 mit der Einführung der Adressierungsarten des Prozessors gemacht (dynamische Adreßrechnung). Zum anderen werden wir die Funktionalität des Assemblers erweitern, nämlich durch die Adreßrechnung zur Übersetzungszeit (statische Adreßrechnung) und durch gegenüber Kapitel 1 erweiterte und weitere Assembleranweisungen.

Format einer Programmzeile. Das Format einer Programmzeile mit Namens-, Operations-, Adreß- und Kommentarfeld mit jeweils flexiblen Feldgrenzen behalten wir bei. Im Namensfeld können Symbole (Namen, *labels*) und im Operationsfeld die Befehlssymbole aus 2.1 sowie die Symbole der nachfolgend beschriebenen Assembleranweisungen verwendet werden. Namen, die vor Befehlen stehen, werden wie bereits in Kapitel 1 zur besseren Kenntlichmachung von Einsprungstellen mit einem Doppelpunkt abgeschlossen. Solche Namen können auch allein in einer Zeile stehen; sie stellen dann den Adreßbezug zum ersten danach erzeugten Maschinencodewort dar, meist zu dem in der Folgezeile stehenden Befehl. Dadurch kann eine Einsprungstelle optisch stärker hervorgehoben werden. Im Adreßfeld sind Zahlen, Zeichen, Symbole und arithmetische Ausdrücke zugelassen. Sie stellen eine Art Konstanten dar, da ihre Werte bereits zur Übersetzungszeit, d.h. durch den Assembler ermittelt werden. Kommentare müssen ein Semikolon vorangestellt werden; dieses wirkt bis zum Ende der jeweiligen Programmzeile. Leerzeilen sind zulässig.

Zahlen (numbers). Sie werden für den Assembler als ganze Zahlen in dezimaler, hexadezimaler oder dualer Form angegeben (auf die Darstellung von Gleitpunktzahlen verzichten wir hier). Da die Kennzeichnungen der Darstellungsformen von Assembler zu Assembler variieren, lehnen wir uns an die in C geläufigen Kennzeichnungen an, wobei es dort jedoch keine Dualkonstanten gibt. Dezimalzahlen haben keine besondere Kennung; den Hexadezimalzahlen ist die Zeichenfolge 0x (oder 0X) vorangestellt; den Dualzahlen stellen wir ein B voran; die Hexadezimalziffern A bis F können groß oder klein geschrieben werden, z.B.

dezimal: 1234, +527, -12

hexadezimal: 0x04, 0x0400FF03, 0xff1b

dual: B00000100, B1111000010100101.

Die hexadezimale und die duale Schreibweise werden auch für Bitvektoren benutzt.

Zeichen (characters). Sie werden als einzelne *ASCII-Zeichen*, in Hochkommas eingeschlossen, oder als ASCII-Zeichenfolge (*ASCII string*), in Apostrophe eingeschlossen, angegeben. Im ersten Fall wird genau ein Byte erzeugt, im zweiten Fall wird ein zusätzliches, abschließendes Null-Byte als String-Begrenzung erzeugt, z.B.

'T' (= 0x54) bzw.

"T" (= 0x5400), "IT'S ALL RIGHT".

Die Darstellung nicht druckbarer ASCII-Zeichen, meist *Steuerzeichen*, erfolgt in hexadezimaler Schreibweise, z.B. 0x0A für den Zeilenvorschub (siehe Tabelle 1-2, S. 7). Üblicherweise sind statt dessen auch die C-Entsprechungen angebbar, für den Zeilenvorschub \n.

Symbole (symbols). Symbole stehen stellvertretend für numerische Adreß- und Operandenangaben. Sie sind, von einigen festgelegten Symbolen abgesehen, frei wählbar; ihre Werte werden durch die Stelle ihres Auftreten im Namensfeld eines Befehls oder einer Anweisung bestimmt. Ein Symbol wird als *relatives Symbol* bezeichnet, wenn sich sein Wert auf den Anfang eines verschiebbaren Programmreichs bezieht. Es wird als *absolutes Symbol* bezeichnet, wenn sein Wert konstant ist. Ausschlaggebend für diese Unterscheidung sind die Assembleranweisungen RORG und ORG zur Kennzeichnung eines Bereichs sowie die Anweisungen EQU und SET für die unmittelbare Wertzuweisung (siehe später). Die Art eines Symbols ist bei der Bildung von Ausdrücken von Bedeutung. – In der Schreibweise sind Symbole meist darauf festgelegt, nur mit einem Buchstaben oder bestimmten Sonderzeichen beginnen zu dürfen (nicht mit einer Ziffer). Darüber hinaus ist die Anzahl der signifikanten Zeichen ggf. begrenzt.

Als festgelegtes Symbol verwenden wir das Zeichen * zur Bezeichnung des Befehlszählers bei der *befehlszählerrelativen Adressierung*. Es erhält als Wert (von 16-Bit-orientierter Befehlsdarstellung ausgehend) jeweils die um zwei erhöhte Adresse des ersten Befehls „wortes“ des Befehls, in dessen Adreßteil es verwendet wird (siehe dazu Beispiel 2.3, S. 81). Diese Adresse und damit das Symbol können absolut oder relativ sein. Weiterhin sind die Symbole R0 bis R7, USP, SSP, SP (für USP bzw. SSP stehend), FP (Framepointerregister) und VB (Vector-base-Register) als Registerbezeichnungen festgelegt. Sie dürfen wie auch das Symbol * nur im Adreßfeld (und nicht im Namensfeld) stehen.

Ausdrücke (expressions). Ausdrücke werden aus Zahlen und Symbolen gebildet, die durch die arithmetischen Operatoren +, -, * und / miteinander verknüpft werden. Wie die Symbole können sie bezüglich ihrer Werte absolut oder relativ sein. Im einfachsten Fall besteht ein Ausdruck aus nur einer Zahl oder nur einem Symbol. Der Wert eines Ausdrucks wird vom Assembler zur Übersetzungszeit ermittelt (*statische Adressrechnung*). Das Ergebnis wird als ganze Zahl dargestellt, d.h., bei der Division wird der Rest nicht berücksichtigt. Die Division durch Null gilt als nicht definiert.

In Verbindung mit den Adressierungsarten sind Ausdrücke die allgemeine Form numerischer und symbolischer Adreß- und Operandenangaben. Sie werden zur Darstellung von Adressen bei der Speicher-Adressierung, von Konstanten bei der DirektoOperand-Adressierung und von Displacements bei allen sie benutzenden Adressierungsarten verwendet. Bei Displacements gilt einschränkend, daß sie absolute Ausdrücke sein müssen. – Ausdrücke werden außerdem zur Definition von Programmkonstanten in den DC-Assembleranweisungen und zur Vorgabe der Byteanzahl bei der Speicherplatzreservierung in den DS-Anweisungen verwendet.

Einige Beispiele sollen die Möglichkeiten der Adreß- und Operandendarstellung durch Ausdrücke zeigen und die Wirkung der Adressierungsarten illustrieren.

MOVE.H	FELD+2,R0	transportiert das auf FELD folgende Speicherhalbwort (Speicher-Adressierung) nach R0.
MOVE.W	#N+5,R2	transportiert den um 5 erhöhten Wert von N als Wort (Direktoperand-Adressierung) nach R2.
LEA	BASE+4(R3),R4	addiert zum Inhalt von R3 den um 4 erhöhten Wert von BASE und lädt das Resultat (effektive Adresse) als Wort nach R4 (registerindirekte Adressierung mit Displacement).
JMP	*-4	lädt den Befehlszähler mit der Adresse des vor dem JMP-Befehl (Ein,,wort“befehl) stehenden Speicherhalbwortes (befehlszählerrelative Adressierung mit Displacement).

Üblicherweise lassen Assembler außer arithmetischen Ausdrücken auch logische Ausdrücke zu; hierzu und auf Details sei auf die Assemblerbeschreibungen der Hersteller verwiesen.

Assembleranweisungen. Im folgenden wird der in 1.3.2 benutzte einfache Satz von Assembleranweisungen an die in Kapitel 2 eingeführte Prozessorstruktur angepaßt und um einige Anweisungen erweitert. Die in eckigen Klammern stehenden Angaben sind wahlweise, d.h., sie können benutzt oder weggelassen werden.

ORG (absolute origin). ORG bezeichnet den Anfang eines festen, im Speicher *nicht verschiebbaren* Programmreichs und lädt den *Zuordnungszähler* (1.3.2) mit dem Wert eines Ausdrucks. Dem nachfolgend erzeugten Maschinencode werden *absolute Adressen*, beginnend mit diesem Wert, zugewiesen. Der Ausdruck muß definiert sein, d.h., er darf keine Vorrätsadreßbezüge (1.3.2) und keine undefinierten externen Adreßbezüge (Adreßsymbole, die in REF-Anweisungen stehen) enthalten. Ein mit ORG beginnender Programm- oder Datenbereich endet mit der nächsten ORG-, RORG-, OFFSET- oder END-Anweisung. Zur Anwendung von ORG siehe 3.1.3.

ORG	Ausdruck	beginne einen festen Programmreich
-----	----------	------------------------------------

RORG (relative origin). RORG bezeichnet den Anfang eines *verschiebbaren* Programmreichs und weist ihm ein Symbol als Namen zu. Tritt ein Name in einer RORG-Anweisung erstmals auf, so wird der Zuordnungszähler mit dem Wert Null geladen. Tritt ein Name wiederholt auf, so wird der Zuordnungszähler mit dem zuletzt unter diesem Namen erreichten Zuordnungszählerstand geladen. Dem nachfolgend erzeugten Maschinencode werden *absolute Adressen*, beginnend mit dem Wert des Zuordnungszählers, zugewiesen. Der Bereich endet mit der nächsten ORG-, RORG-, OFFSET- oder END-Anweisung. Wenn der Bereich im Speicher verschoben werden soll, müssen die im Maschinencode stehenden, auf RORG bezogenen Speicheradressen vor der Programmausführung um die Ladedistanz erhöht werden (z.B. durch einen *verschiebenden Lader*). Man bezeichnet einen solchen Programmreich als *statisch verschiebbar*. Sind die

Adreßbezüge ausschließlich basisrelativ programmiert, so entfällt die Erhöhung von Adreßwerten durch den Lader. Man bezeichnet einen solchen Programmreich dann als *dynamisch verschiebbar*. Zur Anwendung von RORG siehe 3.1.3.

RORG	Symbol	beginne einen verschiebbaren Progammreich
------	--------	---

OFFSET (*generate offset table*). OFFSET dient zur Erzeugung einer Tabelle mit Offset-Werten für Variablen-Symbole, die im Namensfeld von DS-Anweisungen stehen. Die Offset-Zählung beginnt mit dem Wert des Ausdrucks, sofern ein solcher angegeben, sonst mit Null. Der Ausdruck muß definiert sein, d.h., er darf keine Vorwärtsadreßbezüge (1.3.2) und keine undefinierten externen Adreßbezüge (Adreßsymbole, die in REF-Anweisungen stehen) enthalten. Ein mit OFFSET beginnender Vereinbarungsbereich endet mit der nächsten ORG-, RORG-, OFFSET- oder END-Anweisung. Die DS-Anweisungen hinter OFFSET haben keine Speicherplatzreservierung zur Folge. Zur Anwendung von OFFSET siehe 3.1.3.

OFFSET	Ausdruck	erzeuge eine Offset-Tabelle
--------	----------	-----------------------------

ALIGN (*alignment*). ALIGN setzt den Zuordnungszähler auf die nächste gerade Byteadresse oder gerade Halbwortadresse, sofern dieser eine solche Adresse nicht bereits aufweist, und gibt so eine natürliche Halbwort- bzw. Wortgrenze vor (*data alignment*). Übersprungene Bytes werden mit Nullen belegt.

ALIGN.H	setze Zuordnungszähler auf gerade Byteadresse (Halbwortausrichtung)
ALIGN.W	setze Zuordnungszähler auf gerade Halbwortadresse (Wortausrichtung)

EQU (*equate*). EQU weist einem Symbol den Wert eines Ausdrucks zu. Die Wertzuweisung kann durch nachfolgende EQU- oder SET-Anweisungen nicht mehr verändert werden, d.h., EQU setzt „Symbol“ gleich „Ausdruck“. Im Ausdruck auftretende Symbole müssen vor der EQU-Anweisung definiert sein.

Symbol	EQU	Ausdruck	setze gleich
--------	-----	----------	--------------

SET (*set value*). SET weist einem Symbol den Wert eines Ausdrucks zu. Im Gegensatz zu EQU kann das Symbol durch nachfolgende SET-Anweisungen neu definiert werden, wobei das bereits definierte Symbol wiederum im Ausdruck verwendet werden darf, d.h., SET ersetzt „Symbol“ durch „Ausdruck“.

Symbol	SET	Ausdruck	weise zu
--------	-----	----------	----------

DC (*define constant*). Mit DC werden im Speicher Konstanten oder initialisierte Variablen im Byte-, Halbwort- und Wortformat erzeugt. Sie werden durch einen oder mehrere durch Kommas getrennte Ausdrücke (Liste) im Adreßfeld ange-

geben. Ein Symbol im Namensfeld einer DC-Anweisung bezeichnet das erste durch die Anweisung belegte Speicherbyte.

- [Symbol] DC.B Ausdruck(liste) definiere ein bzw. mehrere Bytes
- [Symbol] DC.H Ausdruck(liste) definiere ein bzw. mehrere Halbworte
- [Symbol] DC.W Ausdruck(liste) definiere ein bzw. mehrere Worte

Die Adressvergabe für die Konstanten erfolgt fortlaufend, was nach Ausführung einer DC-Anweisung zu einem Data-Misalignment für die nachfolgende Adressvergabe führen kann. Ist ein Alignment an dieser Stelle erforderlich oder erwünscht, so kann dies durch die ALIGN-Anweisung erzwungen werden, wie in Bild 3-2 gezeigt.

<pre> ORG0x1000 NAME EQU 20 CHAR DC.B'A' WERT DC.HNAME+2 ZAHL DC.H0x0F1C </pre> <p>a</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="width: 10%;">CHAR</th> <th style="width: 10%;">WERT</th> <th style="width: 10%;">ZAHL</th> </tr> </thead> <tbody> <tr> <td>'A'</td> <td>22</td> <td>0x0F</td> </tr> <tr> <td>0x1C</td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	CHAR	WERT	ZAHL	'A'	22	0x0F	0x1C					
CHAR	WERT	ZAHL											
'A'	22	0x0F											
0x1C													
<pre> ORG0x1000 NAME EQU 20 CHAR DC.B'A' ALIGN.H WERT DC.HNAME+2 ZAHL DC.H0x0F1C </pre> <p>b</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="width: 10%;">CHAR</th> <th style="width: 10%;">WERT</th> <th style="width: 10%;">ZAHL</th> </tr> </thead> <tbody> <tr> <td>'A'</td> <td>0x00</td> <td>22</td> </tr> <tr> <td>0x0F1C</td> <td></td> <td></td> </tr> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	CHAR	WERT	ZAHL	'A'	0x00	22	0x0F1C					
CHAR	WERT	ZAHL											
'A'	0x00	22											
0x0F1C													

Bild 3-2. Erzeugen von Konstanten durch die DC-Anweisung, a mit Data-Misalignment, b mit Data-Alignment, erzwungen durch die ALIGN-Anweisung. (Bezüglich der Byteadressierung innerhalb der Speicherzellen wurde von der Big-endian-Byteanordnung ausgegangen.)

DS (define storage). Mit DS wird Speicherplatz im Byte-, Halbwort- und Wortformat reserviert, der üblicherweise mit 0 initialisiert wird. Die Anzahl der Bytes, Halbwörter bzw. Wörter wird als Ausdruck im Adressfeld vorgegeben. Steht ein Symbol im Namensfeld einer DS-Anweisung, so bezeichnet es den ersten Speicherplatz des reservierten Bereichs.

- [Symbol] DS.B Ausdruck reserviere ein bzw. mehrere Bytes
- [Symbol] DS.H Ausdruck reserviere ein bzw. mehrere Halbworte
- [Symbol] DS.W Ausdruck definiere ein bzw. mehrere Worte

Der Zuordnungszähler wird mit jeder DS-Anweisung um die Anzahl der durch sie reservierten Bytes weitergezählt. Wie bei der DC-Anweisung kann dabei ein

Data-Misalignment auftreten, das auch hier durch die ALIGN-Anweisung aufgehoben werden kann. Zur *Speicherplatzreservierung* zwei Beispiele:

STRING DS.B 11 reserviert 11 aufeinanderfolgende Bytes; das erste Byte hat die symbolische Adresse STRING.

ARRAY DS.W N*2 reserviert 2N aufeinanderfolgende Wörter; das erste Wort hat die symbolische Adresse ARRAY. Der Wert für N muß zuvor definiert worden sein.

DEF (defined). DEF gibt in einer Symboliste diejenigen im Programmteil definierten Symbole an, die in anderen, getrennt davon übersetzten Programmteilen verwendet werden. Der Assembler erzeugt daraus die Information für das spätere Zusammenfügen (*Binden*) der unabhängig voneinander übersetzten Programmteile. Die Symbole in der Symboliste werden durch Kommas getrennt.

DEF Symboliste kennzeichne Symbole als definiert

REF (referenced). REF ist das Gegenstück zu DEF und gibt in einer Symboliste diejenigen im Programmteil verwendeten Symbole an, die in anderen, getrennt davon übersetzten Programmteilen definiert sind. (REF und DEF werden z. B. auf die Startadressen von Bibliotheks Routinen angewandt, die ja getrennt von den sie aufrufenden Benutzerprogrammen übersetzt werden.)

REF Symboliste kennzeichne Symbole als verwendet

END (end of code). END zeigt dem Assembler die letzte zu übersetzende Quellcodezeile an. Nachfolgender Quellcode wird nicht übersetzt. Ein Symbol gibt die Startadresse des Programms an. (Diese Angabe ist erforderlich, um ein Hauptprogramm zusammen mit seinen Unterprogrammen schreiben und übersetzen zu können, wobei die Unterprogramme auch vor dem Hauptprogramm stehen können.)

END Symbol beende Assemblierphase

3.1.3 Feste und verschiebbare Programme

Bei der Erstellung von Programmen ist man bestrebt, Adressbezüge so zu bilden, daß Änderungen in der Aufteilung des Adressraums (memory map) nicht dazu führen, ein Programm neu übersetzen zu müssen, um es an eine neue Adresssituation anzupassen. Solche Änderungen ergeben sich z. B. beim Mehrprogrammbetrieb, bei dem die Hauptspeicherbelegung dynamisch erfolgt, indem einem Programm beim Laden ein beliebiger, gerade freier Speicherbereich zugewiesen wird. Bei größeren Systemen wird die dazu erforderliche dynamische Verschiebbarkeit von Programmen durch den Einsatz von Speicherverwaltungseinheiten (MMUs) erreicht, wie in 6.3 beschrieben. Bei kleineren Systemen ohne MMUs (wie wir sie hier betrachten) behilft man sich hingegen damit, entweder die Adressanpassung individuell beim Laden des Programms vorzunehmen oder das

Programm durch Verwendung geeigneter Adressierungsarten des Prozessors von vornherein „lageunabhängig“ zu formulieren.

Neben den variablen Adressbezügen eines Programms gibt es aber auch feste Adressbezüge, die sich bei einer Änderung der Adressraumaufteilung nicht mitverändern dürfen, z.B. solche, die auf Interface-Register, auf ROM- oder auf EPROM-Inhalte verweisen. Auch im Hauptspeicher gibt es ggf. solche Bereiche, z.B. eine nicht verschiebbare Vektortabelle.

Ausgangspunkt für eine geeignete Adressgenerierung mit festen bzw. verschiebbaren Adressbezügen ist der Übersetzungsvorgang, gesteuert durch Assembleranweisungen. Bei einfachen Assemblern gibt es hierzu die ORG-Anweisung, bei leistungsfähigeren Assemblern Anweisungen wie RORG und OFFSET. Zur Veranschaulichung der Wirkungen dieser Anweisungen werden im folgenden drei Varianten der Adressgenerierung am Beispiel von Ausschnitten aus einem Programm und seiner Datenbereiche gezeigt. Sie unterscheiden sich hinsichtlich der Verschiebemarkmale

1. fest, d.h. nicht verschiebbar,
2. statisch verschiebbar und
3. dynamisch verschiebbar.

Nicht verschiebbares Programm. Die folgende Sequenz zeigt ein Programm, bestehend aus Speicherplatzreservierungen für Variablen (DS-Anweisungen), Festlegungen von Konstanten (DC-Anweisungen), Wertzuweisungen an symbolische Adressen von z.B. Interface-Registern (EQU-Anweisungen) und dem eigentlichen Programmcode, der an der Adresse START beginnt. Diese Programmteile sind durch die Anweisungen ORG und END eingeschlossen und teilen sich, mit Ausnahme der durch EQU festgelegten Adressen, einen gemeinsamen, zusammenhängenden Adressbereich.

```

ORG      0x1000
VAR      DS.W    1          ; Variablen
FELD     DS.B    128
        :
KONST    DC.W    -73748   ; Konstanten
MAX      DC.B    128
        :
OUTREG   EQU     0xFFFF00A2 ; EQU-Zuweisungen
INREG    EQU     0xFFFF00A4
        :
START:   :
        :          ; Programmcode
MOVE.W   KONST, VAR
        :
LEA      FELD, R1
AGAIN:   MOVE.B  (R1)+, OUTREG
        :
JMP      AGAIN
END      START

```

Bei der Übersetzung des Programms werden vom Assembler sämtliche symbolischen Adressen, die sich auf ORG beziehen, als absolute, d.h. als feste Adressen erzeugt (Speicher-Adressierung), wobei die Adreßzählung mit dem in der ORG-Anweisung angegebenen Anfangsadresse, hier 0x1000, beginnt. Das betrifft die Adressen VAR, FELD, KONST, MAX, START und AGAIN und gilt auch für Adreßbezüge, die von anderen Programmen auf diese Adressen verweisen. Das heißt, um das Programm korrekt ausführen zu können, muß dieses an der durch die ORG-Anweisung vorgegebenen Anfangsadresse geladen werden; die ORG-Anweisung legt also die Ladeadresse fest. Das Programm gilt dementsprechend als fest, d.h. als *nicht verschiebbar*. Um es in einem anderen Speicherbereich ausführen zu können, muß es zuvor mit geänderter Adreßangabe in der ORG-Anweisung neu übersetzt werden. – Nicht betroffen von der durch ORG vorgegebenen Ladeadresse sind die Interface-Registeradressen OUTREG und INREG. Sie werden vom Assembler zwar ebenfalls als absolute Adressen geführt, behalten aber immer den in ihren EQU-Anweisungen angegebenen Wert.

Statisch verschiebbares Programm. Wird die ORG-Anweisung im obigen Programmbeispiel durch die RORG-Anweisung ersetzt, so erzeugt der Assembler die auf RORG bezogenen Adressen VAR, FELD, KONST, MAX, START und AGAIN zwar als absolute Adressen (Speicher-Adressierung), gibt für den Lader jedoch an, daß diese Adressen während des Ladevorgangs mit einer einheitlichen Adreßdistanz beaufschlagt werden dürfen. Wird in der RORG-Anweisung als Anfangsadresse Null angegeben, so hat der Lader die Möglichkeit, das Programm mit beliebiger Adreßdistanz, die dann gleich der Ladeadresse ist, zu laden. Ein solcher sog. *verschiebender Lader* modifiziert dementsprechend die so ausgewiesenen Adreßbezüge während des Ladevorgangs, verändert also nachträglich den vom Assembler erzeugten Programmcode. Das Programm braucht bei neuer Ladeadresse also nicht neu übersetzt zu werden; es gilt somit als *statisch verschiebbar*.

Dynamisch verschiebbares Programm. Eine noch größere Flexibilität in der Verschiebbarkeit eines Programms und seiner Daten erhält man, wenn man die Verschiebbarkeit nicht nur vom Übersetzungsvorgang, sondern auch vom Ladevorgang unabhängig macht, d.h. das Programm und seine Daten *dynamisch verschiebbar* macht. Man erreicht dies, indem man die Programm- und Datenbezüge von vornherein *basisrelativ* programmiert, vorausgesetzt, der Prozessor stellt die entsprechenden Addressierungsarten, wie sie in 2.1.3 beschrieben sind, bereit. Für die Programmbezüge, d.h. für die Sprungadressen und die Konstantenadressen bedeutet das die Verwendung der *befehlszählerrelativen Adressierung*, für die Variablenadressen die Verwendung der *registerindirekten Adressierung*, so wie im folgenden Programmbeispiel als Modifikation des obigen Programms gezeigt.

```
OFFSET 0
VAR    DS.W   1           ; Variablen
FELD   DS.B   128
```

```

        RORG
KONST  DC.W   -73748      ; Konstanten
MAX    DC.B   128
      :
OUTREG EQU    0xFFFF00A2    ; EQU-Zuweisungen
INREG  EQU    0xFFFF00A4
      :
START:  :
      MOVE.W KONST(PC), VAR(R6) ; Programmcode
      :
      LEA    FELD(R6), R1
AGAIN: MOVE.B (R1)+, OUTREG
      :
      JMP    AGAIN(PC)
END    START

```

Das Programm ist jetzt in zwei Teile aufgeteilt, nämlich in einen mit der OFFSET-Anweisung beschriebenen Variablen Teil und den mit der RORG-Anweisung beschriebenen Programmteil. Letzterer umfaßt neben dem eigentlichen Programmcode auch die Konstanten und die EQU-Anweisungen. RORG zeigt nun an, daß der nachfolgende Programmteil verschiebbar ist, ohne daß der Lader den Programmcode verändern muß. Insofern hat RORG hier eine etwas andere Bedeutung als im oben beschriebenen Fall der statischen Verschiebbarkeit. Zusätzlich könnte das Programm um ORG-Bereiche erweitert sein, um damit nicht verschiebbare Programm- oder Datenbereiche zu vereinbaren. Die Adressbezüge innerhalb des Programmteils, also die Sprung- und Konstantenadresse sind PC-relativ formuliert, d.h., die vom Assembler den Symbolen KONST, MAX, START und AGAIN zugeordneten Werte sind hier Displacements, bezogen auf den jeweiligen Inhalt des Befehlszählers. Die durch die EQU-Anweisungen festgelegten Interface-Registeradressen OUTREG und INREG werden durch RORG nicht beeinflußt.

Die Adressbezüge auf die mit OFFSET vereinbarten Variablen sind im Programmteil mittels registerindirekter Adressierung mit Displacement formuliert, d.h. relativ zu einem *Basisadreßregister*, hier dem allgemeinen Register R6. Dementsprechend muß der Assembler den symbolischen Adressen VAR und FELD Displacement-Werte zuordnen, wozu er durch die OFFSET-Anweisung angehalten wird. Mit der Angabe von 0 im Adreßteil der OFFSET-Anweisung beginnt die Offsetzählung bei Null. Die eigentliche Basisadresse des Variablenbereichs wird zur Ladezeit bestimmt, zu der das Basisadreßregister mit dieser Adresse geladen werden muß. Die Lage des Variablenbereichs im Speicher kann dementsprechend unabhängig von der des Programmberreichs gewählt werden (zur OFFSET-Anweisung siehe auch 3.1.2).

Werden die Sprungbezüge im Programmteil durch Branch-Befehle hergestellt, so kann die explizite Angabe der PC-relativen Adressierung in der Adressangabe meist entfallen, da die Prozessoren für Branch-Befehle üblicherweise sowieso nur die PC-relative Adressierung vorsehen und dem Assembler damit keine Wahl lassen. Anstatt z.B. BRA AGAIN(PC) kann dann BRA AGAIN geschrieben werden.

Bei den Zugriffen auf die Konstanten ist hingegen die explizite Schreibweise erforderlich, da es hier auch die Alternative der Speicher-Adressierung gibt. Zu beachten ist hier, daß einige Prozessoren die PC-relativen Zugriffe auf Daten nur lesend erlauben, weshalb zwar Konstanten PC-relativ adressierbar sind, Variablen jedoch nicht. PC-relative Schreibzugriffe lösen bei diesen Prozessoren einen Trap aus. Der Hintergrund dieser Maßnahme ist, daß man Programmcodebereiche aus Sicherheitsgründen vor Schreibzugriffen schützen möchte und dementsprechend Schreibzugriffe bereits durch die Hardware unterbindet.

Anmerkung. Im obigen Beispiel wurden die Variablen global vereinbart. Sie stehen während der gesamten Laufzeit des Programms für Zugriffe zur Verfügung und gelten dementsprechend bezüglich der Speicherbelegung als statisch. Häufig ist es jedoch angebracht, die Variablen Programmteilen lokal zuzuordnen (siehe hierzu 3.3.2: Beispiel 3.11, S. 195).

3.1.4 Makrobefehle und bedingte Assemblierung

Assemblersprachen heutiger Mikroprozessorsysteme erlauben neben der 1-zu-1-Umformung symbolischer Maschinenbefehle auch die 1-zu-n-Übersetzung sog. *Makrobefehle* in i.allg. mehrere Maschinenbefehle. Diese Ausdehnung einer Zeile Assemblercode in n Zeilen Maschinencode während der Übersetzungszeit wird als *Makroexpansion* bezeichnet.

Makrobefehle haben den gleichen Aufbau wie symbolische Maschinenbefehle. Dem Code des Maschinenbefehls entspricht der Name und den Adressen entsprechen die *Parameter* des Makrobefehls. Während die Maschinenbefehle jedoch in Funktion, Format und Anzahl von der Prozessorhardware bestimmt sind, können Makrobefehle vom Programmierer im Rahmen der Assemblersprache selbst definiert werden. Dementsprechend kann die Anzahl von Parametern und ihre Bedeutung vom Anwender weitgehend frei festgelegt werden. Auch die Anzahl von Makrobefehlen und ihre Wirkung ist nahezu unabhängig von der Prozessorhardware.

Makrobefehle ermöglichen es, den Befehlssatz des Prozessors auf der Assemblerebene zu erweitern und tragen damit wesentlich zur übersichtlichen Gestaltung von Assemblerprogrammen bei. Der Benutzer eines Mikroprozessorsystems kann sich eine seinem Problemkreis angepaßte problemorientierte höhere Assemblersprache schaffen, indem er sich geeignete, aufeinander abgestimmte Makrobefehle definiert. Erlaubt es die Software des Mikroprozessorsystems, eine solche durch den Anwender selbst geschaffene „Sprache“ in eine *Makrobibliothek* zu übernehmen, so kann das Mikroprozessorsystem im Extremfall ohne die Benutzung eines einzigen Maschinenbefehls programmiert werden.

Um bei einer solchen weitgehend von den Aufgaben und dem Stil des Anwenders geprägten Programmierungstechnik möglichst effiziente Maschinenprogramme zu erzeugen, bedient man sich des Mittels der *bedingten Assemblierung*. Dabei

wird in Abhängigkeit von der Art und der Anzahl der Parameter eines im Assemblerprogramm auftretenden Makrobefehls während der *Makroexpansion* unterschiedlicher Maschinencode erzeugt. Im Rahmen der von der Assemblersprache vorgegebenen Mittel hängt es vom Geschick des Programmierers bei der Definition (d.h. bei der Programmierung des Makrobefehls) ab, ob das Ziel erreicht wird, bei einem Maximum an Übersichtlichkeit des Assemblerprogramms ein Minimum an Speicherplatzbedarf und Ausführungszeit des übersetzten Maschinenprogramms zu erreichen.

Zur *Definition von Makrobefehlen* und zur Anwendung der bedingten Assemblierung wird die Assemblersprache durch die Einführung zusätzlicher Sprachelemente erweitert. Solche Makroassembleranweisungen sind z.B.

- MACRO (macro begin) in Verbindung mit ENDM (end macro) zur Definition eines Makrobefehls,
- NUMB (number) zur Feststellung der Anzahl der beim Aufruf des Makrobefehls auftretenden Parameter,
- IFEQ (if equal), IFGR (if greater) usw. in Verbindung mit ENDC (end condition) zur Abfrage von zur Assemblierzeit bekannten Bedingungen und
- REPT (repetition) in Verbindung mit ENDR (end repetition) zur wiederholten Assemblierung einer bestimmten Anzahl von Assemblerzeilen.

Die letzten beiden Anweisungsgruppen unterscheiden sich grundsätzlich von allen anderen Assembleranweisungen, da bei ihnen die sonst übliche zeilenweise fortschreitende Übersetzung des Quellprogramms durchbrochen wird. In Abhängigkeit einer Bedingung kann nämlich die Assemblierung von Programmzeilen übersprungen bzw. in Abhängigkeit eines Zählerstandes mehrfach durchlaufen werden.

Bei der Übersetzung eines Assemblerprogramms ersetzen die Parameter eines Makrobefehls (kurz *aktuelle Parameter*) als Text ihre in der Makrodefinition benutzten formalen Entsprechungen (kurz *formale Parameter*). Durch die Anwendung von Anweisungen zur bedingten Assemblierung werden je nach Art und Anzahl der Parameter verschiedene und unterschiedlich viele Programmzeilen der Makrodefinition übersetzt, so daß dementsprechend unterschiedlich viele Maschinencodewörter entstehen können.

Um einen kurzen Eindruck sowohl von den Möglichkeiten der Programmierung mit Makrobefehlen als auch den Abläufen bei der Makroexpansion zu bekommen, ist im folgenden Beispiel die Definition eines Makrobefehls zusammen mit drei *Makroaufrufen* und ihren durch den *Makroassembler* erzeugten unterschiedlichen expandierten Formen als symbolische Maschinenprogramme angegeben.

Beispiel 3.1. ► Makrobefehl mit mehreren Makroexpansionen. Es soll ein Makrobefehl SAVR geschrieben werden, der die Inhalte der ab dem zweiten Parameter angegebenen Register auf den Stack schreibt, wenn der erste Parameter STACK lautet, und der andernfalls die Registerinhalte in

```
; Retten bestimmter Registerinhalte

SAVR MACRO /
N NUMB /
I SET 2
    IFEQ STACK,/1
    REPT N-1
    PUSH.W /I
I SET I+1
    ENDR
    ENDM
    ENDC
    MOVE.W /2,/1
    IFEQ 2,N
    ENDM
    ENDC
    LEA /1,/2
    ADD.W #4,/2
I SET I+1
    REPT N-2
    MOVE.W /I,(/2) +
I SET I+1
    ENDR
    ENDM
```

; 3 Makroaufrufe von SAVR

SAVR	STACK,R0,R1,R7	PUSH.W R0 PUSH.W R1 PUSH.W R7
SAVR	FELD,R7	MOVE.W R7,FELD
SAVR	(R6),R6,R7	MOVE.W R6,(R6) LEA (R6),R6 ADD.W #4,R6 MOVE.W R7,(R6) +

Bild 3-3. Makrodefinition und drei verschiedene Makroaufrufe zu Beispiel 3.1

ein Feld rettet, dessen Anfangsadresse durch den ersten Parameter vorgegeben ist. – Bild 3-3 zeigt die Makrodefinition und drei Makroaufrufe mit ihren Expansionen. Die MACRO-Anweisung enthält im Namensfeld das Symbol des Makrobefehls und im Adressfeld ein wählbares Zeichen – hier / – zur Bezeichnung der Liste der formalen Parameter des Makrobefehls. Eine Zahl unmittelbar nach diesem Zeichen gibt das Listenelement, d.h. die Position eines beim Makroaufruf auftretenden aktuellen Parameters, an. Die NUMB-Anweisung ermittelt die Anzahl der im Makroaufruf angegebenen aktuellen Parameter und weist diese Zahl dem im Namensfeld stehenden Symbol zu. Die IFEQ-Anweisung bewirkt bei Gleichheit der beiden im Adressfeld stehenden Texte die Assemblierung der zwischen IFEQ und ENDC stehenden Programmzeilen. Die REPT-Anweisung wiederholt entsprechend der im Adressfeld erscheinenden Angabe die Assemblierung der zwischen REPT und ENDR stehenden Zeilen.

Heißt der erste aktuelle Parameter STACK, so werden bei der Expansion des Makrobefehls so viele PUSH-Befehle erzeugt, wie Registerinhalte auf den Stack gerettet werden sollen. Andernfalls wird ein MOVE-Befehl erzeugt, der den Inhalt des als zweiten aktuellen Parameter ange-

gebenen Registers unter der durch den ersten aktuellen Parameter angegebene Feldanfangsadresse speichert. Sollen nicht nur ein, sondern mehrere Registerinhalte gerettet werden, so werden zusätzlich folgende Befehle generiert: ein LEA-Befehl, der die effektive Feldanfangsadresse in das durch den vorhergehenden MOVE-Befehl freigemachte Register lädt, ein ADD-Befehl, der diese Adresse um 4 erhöht (Wortadressierung), und so viele MOVE-Befehle, wie Registerinhalte in Verbindung mit der Postinkrement-Adressierung in den Speicher transportieren werden sollen.

Die in Bild 3-3 neben den Makroaufrufen angegebenen Makroexpansionen illustrieren, wie aufgrund verschiedener Parameterangaben drei unterschiedlich lange Maschinenbefehlsfolgen mit unterschiedlichen Funktionen entstehen. ▶

3.2 Programmflußsteuerung

Bei der Verarbeitung eines Programms wird mit jedem Befehl, der aus dem Speicher zur Ausführung in den Mikroprozessor gelesen wird, der Befehlszähler erhöht, so daß er beim nächsten Befehlsabruft den nächsten Befehl im Speicher adressiert. Diese Geradeausverarbeitung von Befehlsfolgen kann durch Sprungbefehle unterbrochen werden, die es erlauben, den Befehlszähler mit einer im Befehl angegebenen Sprungadresse zu laden und damit das Programm an anderer Stelle fortzusetzen. Je nachdem, ob der Sprung von einer Bedingung abhängig ist oder nicht, spricht man von einem *bedingten* bzw. *unbedingten Sprung*. Unbedingte Sprünge setzen die Übersichtlichkeit des Programmflusses eher herab, bedingte Sprünge hingegen unterstützen die Strukturierung des Programmflusses. Mit beiden Sprungarten lassen sich vielfältige Verzweigungsstrukturen programmieren, so z.B. die Programmflußstrukturen höherer Programmiersprachen.

Im folgenden stellen wir einige typische Verzweigungsstrukturen in Beispielen vor, wobei wir die Abläufe zunächst als Flußdiagramme darstellen, um sie dann in Assemblerprogramme für den in Kapitel 2 beschriebenen CISC-Prozessor umzusetzen. (Im Prinzip könnten diese Programme mit geänderter Befehlsmnemonik genauso für den RISC-Prozessor in Kapitel 2 angegeben werden.) Im Hinblick auf die Bedeutung der Programmiersprache C für die Programmierung von Mikroprozessor- und Mikrocontrollersystemen ergänzen wir diese Assemblerprogramme durch ihre Entsprechungen in C. Hierbei wählen wir ggf. eine von mehreren Möglichkeiten unter Inkaufnahme, daß der C-Compiler nicht genau das von uns angegebene Assemblerprogramm, sondern eine andere, u.U. aufwendigere Befehlsfolge erzeugt. Insofern soll diese Gegenüberstellung von Assembler- und C-Programmen eher eine Hilfestellung geben, wie das Problem auch in C formuliert werden kann und wo aus Gründen der Speicherplatz- oder Laufzeitersparnis ggf. Assemblereinbindungen verwendet werden sollten. Zu den Techniken der Verbindung von C und Assembler sei auf Kapitel 4 verwiesen. Grundkenntnisse in der C-Programmierung setzen wir voraus. – Bei unseren C-Programmen gehen wir von einem Compiler mit den Ganzzahldatentypen *short* gleich 16 Bit und *int* gleich 32 Bit aus.

3.2.1 Programmverzweigungen

Einfachverzweigung. Die Verwendung eines bedingten Sprungbefehls ermöglicht eine sog. *Einfachverzweigung*, indem der lineare Programmfluß bei erfüllter Sprungbedingung verlassen und das Programm an anderer Stelle fortgesetzt wird. Ist die Verzweigung als Vorwärtssprung ausgeführt und überspringt somit die auf den Sprungbefehl folgende Befehlsfolge, so entspricht das im Prinzip der *if-Anweisung* höherer Programmiersprachen. Allerdings unterscheiden sich die Einfachverzweigung und die *if*-Anweisung zunächst durch ihr gegensinniges Verhalten: Während im Assemblerprogramm bei erfüllter Sprungbedingung eine Befehlsfolge übersprungen wird, wird sie in der höheren Programmiersprache bei erfüllter *if*-Bedingung ausgeführt (Bild 3-4a). Will man diesen Gegensatz aufheben, so muß die Sprungbedingung auf der Assemblerebene invers formuliert werden.

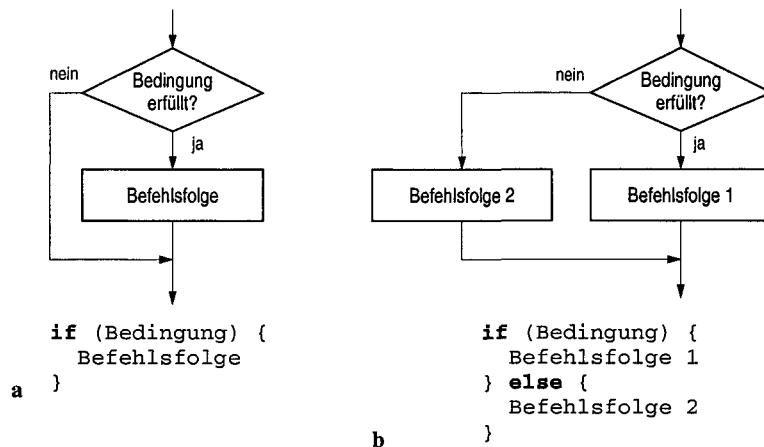


Bild 3-4. Programmverzweigungen in Flußdiagrammdarstellung mit C-Entsprechungen, a Einfachverzweigung (*if*-Anweisung), b Einfachverzweigung mit Alternative (*if-else*-Anweisung)

In Erweiterung der Einfachverzweigung, kann auch der Verzweigungspfad aus einer Befehlsfolge bestehen, die dann alternativ zu der auf den Sprungbefehl folgenden Befehlsfolge ausgeführt wird. Man bezeichnet diese Programmflußstruktur als *Einfachverzweigung mit Alternative*. Sie entspricht im Prinzip der *if-else-Anweisung* höherer Programmiersprachen und erfordert aufgrund der Hintereinanderschreibung beider Befehlsfolgen zusätzlich zum bedingten Sprung einen unbedingten Sprung, um beide Programmzweige wieder zusammenzuführen (Bild 3-4b). Auch hier ist wieder das gegensinnige Verhalten in den beiden Programmierebenen zu beachten. Beispiel 3.2 illustriert die Einfachverzweigung mit Alternative.

Beispiel 3.2. ► *Einfachverzweigung mit Alternative (if-else-Anweisung)*. Zwei 2-Komplement-Zahlen X und Y sollen mit folgender Einschränkung voneinander subtrahiert werden: Die Subtraktion ist nur dann auszuführen, wenn der Minuend Y kleiner ist als der Subtrahend X, andernfalls ist eine Fehlerbehandlung zu durchlaufen. In beiden Fällen ist das Programm gemäß der if-else-Anweisung an einer gemeinsamen Stelle fortzusetzen. – Bild 3-5 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Im Assemblerprogramm wird die Abfrage durch den für 2-Komplement-Zahlen zuständigen bedingten Sprungbefehl BLT (branch if less than) durchgeführt. Bei erfüllter Bedingung wird zur Subtraktion verzweigt (if-Zweig). Bei nicht erfüllter

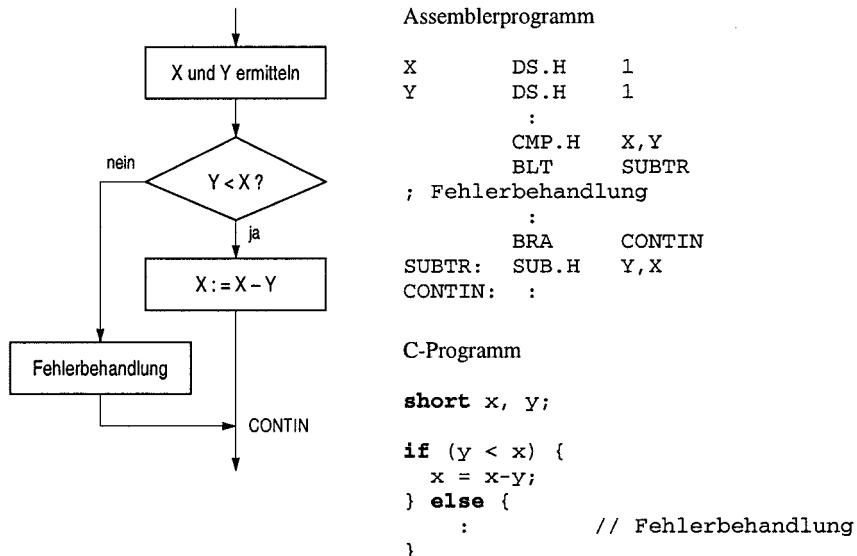


Bild 3-5. Flußdiagramm und Programme zu Beispiel 3.2: Einfachverzweigung mit Alternative (if-else-Anweisung)

Bedingung wird eine Fehlerbehandlung durchlaufen, deren Befehlsfolge unmittelbar auf den BLT-Befehl folgt (else-Zweig), und es wird dann die in der Programmaufschreibung folgende Subtraktion mittels eines unbedingten Sprungbefehls (BRA, branch always) übersprungen. Diese Reihenfolge der Aufschreibung, erst der else- und dann der if-Zweig, die nicht der Reihenfolge in höheren Programmiersprachen entspricht (siehe das C-Programm), lässt sich umkehren, indem die Abfrage mit BGE (branch if greater or equal) invers formuliert wird. Damit läge die Subtraktion als eigentliche Operation im Programmfluß und die Fehlerbehandlung als Ausnahmefall im Verzweigungspfad. ◀

Ausgangspunkt der Ausführung eines bedingten Sprungbefehls ist eine ihm vorangehende Beeinflussung der Bedingungsbits CC im Statusregister. Diese Bits werden durch fast alle Befehle beeinflußt, so daß sich fast alle Befehle zur Vorbereitung einer Programmverzweigung verwenden lassen. Insbesondere gibt es hierfür jedoch die **Vergleichsbefehle** CMP (Vergleich zweier Operanden), TEST (Vergleich eines Operanden mit Null) und BTST (Vergleich eines einzelnen

Operandenbits mit Null), bei denen das Ergebnis der Vergleichsoperation ausschließlich auf die Bedingungsbits wirkt. Wichtig dabei ist, daß die Vergleichsoperation und der Sprungbefehl aufeinander abgestimmt sind, indem der Datentyp in der Sprungbedingung (Bcc) sich nach dem Datentyp der durch den Vergleichsbefehl angesprochenen Operanden richtet. Zu unterscheiden sind hier im wesentlichen folgende Datentypen, die zu *arithmetischen* bzw. *logischen Vergleichen* führen:

- 2-Komplement-Zahl,
- vorzeichenlose Dualzahl,
- Bitvektor und
- Zustandsgröße.

Hierbei ist zu beachten, daß es für die arithmetischen Bedingungen $<$, \leq , \geq und $>$ unterschiedliche Bcc-Befehle für 2-Komplement-Zahlen und für vorzeichenlose Dualzahlen gibt (z.B. BGT und BHI für $>$, siehe dazu Beispiel 2.4, S. 94). Hingegen eignen sich die Bedingungen $=$ und \neq (BEQ und BNE) für beide Zahlenarten sowie für logische Vergleiche (siehe auch Tabelle 2-2, S. 93). Die sich auf die CC-Bits N, C und V beziehenden Bedingungen sind zunächst arithmetischer Natur (siehe Beispiel 3.2), N und C sind darüber hinaus aber auch zur Auswertung einzelner Datenbits, d.h. zur Formulierung logischer Bedingungen geeignet.

Anmerkung. Für eine übersichtliche und sichere Programmierung sollte die Beeinflussung der Bedingungsbits immer durch den Befehl unmittelbar vor dem Sprungbefehl erfolgen, da dazwischenliegende Befehle die Bedingungsbits ggf. erneut beeinflussen können. Allerdings gibt es auch Rechner mit Befehlen mit und ohne cc-Zusatz (z.B. SPARC), so daß eine Beeinflussung der CC-Bits durch dazwischenliegende Befehle ausgeschaltet werden kann.

Verknüpfung von zweiwertigen Bedingungen. Eine mittels der Bcc-Befehle, d.h. auf der Maschinenebene, formulierte Programmverzweigung basiert immer auf der Auswertung einer *zweiwertigen Bedingung*, d.h. auf einer ja/nein-Entscheidung (z.B. bzgl. der Gleichheit zweier Bits, zweier Bitmuster oder zweier Zahlen). Auf der Problemebene hingegen sind auch komplexere Bedingungen gebräuchlich, indem z.B. mehrere zweiwertige Bedingungen logisch miteinander verknüpft werden. Deren Auswertung kann entweder wieder auf zwei Programmfpade (*Einfachverzweigung*), oder aber auch auf mehr als zwei Programmfpade führen (*Mehrfachverzweigung*).

Bild 3-6 zeigt dazu zunächst ein Beispiel für eine Einfachverzweigung, bei der zwei zweiwertige Bedingungen B1 und B2 durch logisches Und miteinander verknüpft werden und mit den beiden Gesamtaussagen ja und nein auf jeweils einen von zwei Programmfpaden führen. Beispiel 3.3 zeigt dann eine Mehrfachverzweigung, bei der zwei Zahlen miteinander verglichen werden und bei der hinsichtlich der Relationen $>$, $=$, $<$ auf einen von drei möglichen Programmfpaden verzweigt wird. Auch diese Verzweigung basiert auf der Verknüpfung zweier zweiwertiger Bedingungen.

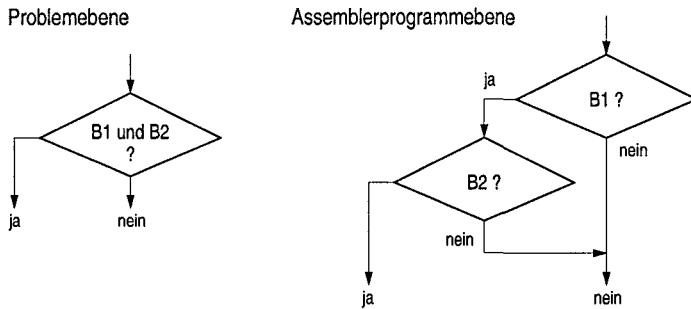


Bild 3-6. Einfachverzweigung durch Verknüpfung von zweiwertigen Bedingungen

Beispiel 3.3. ► Mehrfachverzweigung durch Verknüpfung von Bedingungen. Zwei 2-Komplement-Zahlen A und B mit je 32 Bit sollen miteinander verglichen werden. Je nach Erfüllung der Bedingungen A kleiner, gleich oder größer B soll zu einem von drei möglichen Programmfpaden verzweigt werden. – Bild 3-7 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Für die Abfragevorbereitung im Assemblerprogramm genügt ein einziger CMP-Befehl, da die von ihm beeinflußten Bedingungsbits durch den BLT-Befehl der ersten Abfrage nicht verändert werden. Das C-Programm zeigt im Prinzip denselben Aufbau wie das Assemblerprogramm. Der C-Compiler wird daraus mutmaßlich eine Befehlsfolge erzeugen, die nicht so gut optimiert ist wie das abgebildete Assemblerprogramm, indem er vor beiden Sprungbefehlen einen CMP-Befehl verwenden wird.

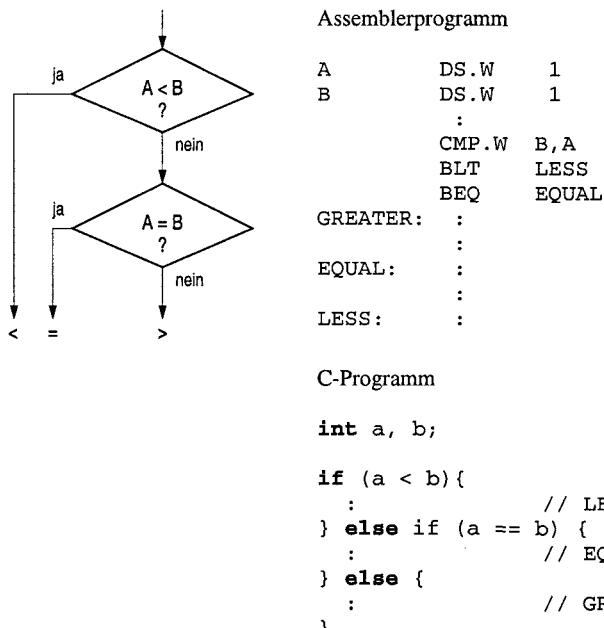


Bild 3-7. Flußdiagramm und Programme zu Beispiel 3.3: Mehrfachverzweigung durch Verknüpfung von Bedingungen

Mehrwertige Bedingung. Eine *Mehrfachverzweigung* erhält man auch dann, wenn eine einzelne Bedingungsgröße mehr als zwei Werte annehmen kann und diese auf mehr als zwei unterschiedliche Verzweigungen führen (*mehrwertige Bedingung*). Solche Werte können z.B. sämtliche Zahlen innerhalb eines Zahlenbereichs sein. Es können aber auch bestimmte Codewörter eines Codes (z.B. des ASCII) oder allgemein bestimmte Bitmuster aus einem insgesamt größeren Vorrat an Bitmustern sein.

Das Abfragen der signifikanten Bedingungswerte ist zunächst sequentiell möglich, wobei die Einzelabfragen durch CMP-Befehle vorbereitet werden (*sequentielles Verzweigen*). Dieses Vorgehen kann jedoch sehr zeitaufwendig sein, da im ungünstigsten Fall alle Abfragen durchlaufen werden müssen. Eine effizientere Programmierungstechnik zeigt Bild 3-8 für den Fall, daß sämtliche Werte eines Wertebereichs signifikant sind. Hier werden die Bits der Bedingungsgröße baumförmig ausgewertet, wodurch die Anzahl der Abfragen immer gleich der Anzahl der für die Wertedarstellung erforderlichen Bits ist (*binäres Verzweigen*). Die Abfragen können durch BTST-Befehle vorbereitet werden. Gegenüber dem sequentiellen Verzweigen wird das Programm jedoch unübersichtlicher. Bild 3-8 zeigt den Fall einer 2-Bit-Bedingungsgröße, die die Werte 0 bis 3 annehmen kann.

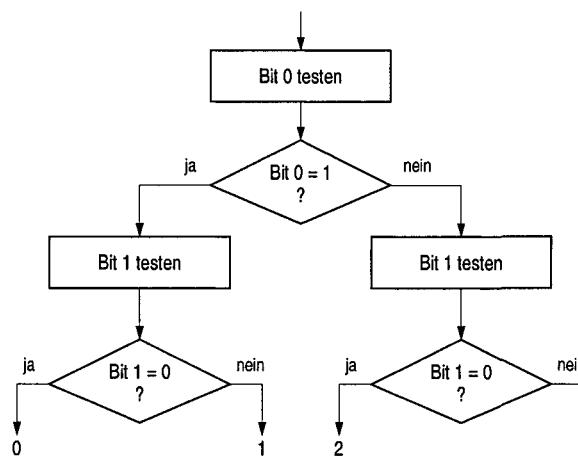


Bild 3-8. Baumförmige Zerlegung einer mehrwertigen Bedingung in mehrere zweiwertige Bedingungen

Eine weitere Möglichkeit der Programmierung von Mehrfachverzweigungen mit einer mehrwertigen Bedingungsgröße bietet die Verwendung des JMP-Befehls mit speicherindirekter Adressierung des Sprungziels. Hierbei wird ein Register zunächst mit der Anfangsadresse einer *Sprungtabelle* geladen, die ihrerseits die Adressen der einzelnen Programmfade enthält. Vor der Ausführung des Sprungbefehls wird der Registerinhalt durch Addition der Bedingungsgröße so verändert, daß er in der Sprungtabelle auf die der Verzweigung entsprechende Ziel-

adresse zeigt. Die Verzweigung erfolgt allein durch den JMP-Befehl; eine mehrschrittige Bedingungsabfrage entfällt. Hierbei wird allerdings auch wieder davon ausgegangen, daß die signifikanten Werte lückenlos sind. – *Anmerkung:* Im Fall von einzelnen signifikanten Werten aus einem größeren Wertevorrat kann ebenfalls mit einer Tabelle gearbeitet werden, die dabei nicht den Umfang des Wertebereichs insgesamt haben muß, indem nämlich die Tabellenadressierung über Hash-Funktionen durchgeführt wird.

Beispiel 3.4. ► Mehrfachverzweigung durch mehrwertige Bedingungsgröße (switch-Anweisung). In Abhängigkeit von einer 8-Bit-Bedingungsgröße COND soll für die Werte 0, 1, 2 und 3 eine Verzweigung zu einem von vier Programmfpaden mit den Adressen NULL, EINS, ZWEI und DREI durchgeführt werden; bei einem Wert größer als 3 soll das Programm an der Adresse DEFAULT fortgesetzt werden. – Bild 3-9 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Im Assemblerprogramm erfolgt zunächst die Abfrage auf den Default-Fall COND > 3. Tritt dieser nicht auf, so wird die Mehrfachverzweigung durchlaufen. Die dafür vorgesehene Sprungtabelle STAB mit den Zieladressen NULL, EINS, ZWEI und DREI wird durch vier DC.W-Anweisungen erzeugt. Die Verzweigung selbst erfolgt durch den davorstehenden JMP-Befehl mit speicher- sowie registerindirekter Adressierung der Sprünge unter Verwendung von R0. R0 wird zuvor mit der Anfangsadresse der Sprungtabelle STAB geladen und zusätzlich um den vierfachen Wert der Bedingungsgröße COND erhöht. COND wird dazu auf Wortformat erweitert (Zero-Extension durch den AND-Befehl). Die Vervierfachung des Wertes erfolgt durch einen arithmetischen 2-Bit-Linksshift; sie ist notwendig, da die Adreßbeintragungen in der Sprungtabelle jeweils ein Wort umfassen. (In einer kompakteren Variante des Programms kann auf die Befehle LEA und ADD verzichtet und die Verzweigung mit dem Befehl JMP [STAB(R1)] durchgeführt werden.) Im C-Programm stellt sich die Mehrfachverzweigung als switch-Anweisung dar. Darin enthalten sind break-Schlüsselwörter, damit jeweils nur der betreffende Programmfpad durchlaufen wird. Die Default-Abfrage erfolgt hier am Schluß. Über die daraus vom Compiler erzeugte Befehlsfolge kann nichts ausgesagt werden.

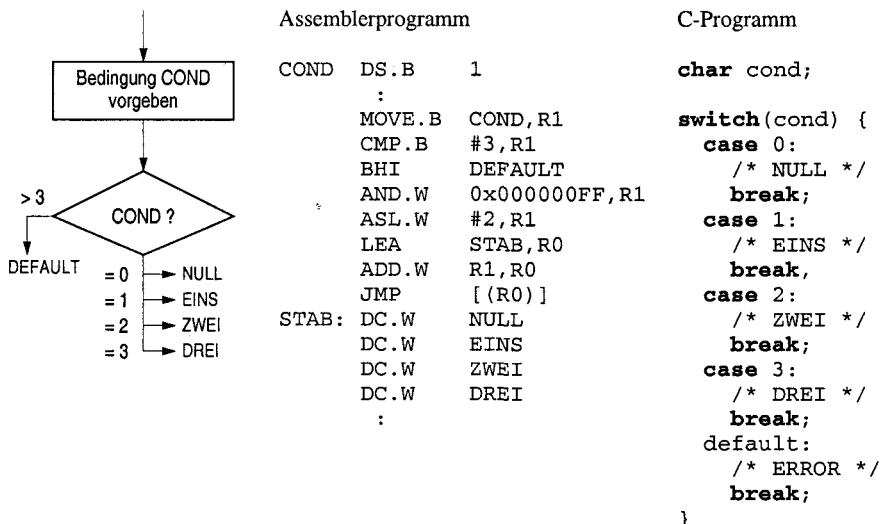


Bild 3-9. Flußdiagramm und Programme zu Beispiel 3.4: Mehrfachverzweigung durch mehrwertige Bedingungsgröße (switch-Anweisung)

3.2.2 Programmschleifen

Eine wichtige Anwendung von Programmverzweigungen ist das Bilden von Programmschleifen für das wiederholte Durchlaufen einer Befehlsfolge. Die Schleifenbildung erfolgt durch einen Sprungbefehl am Ende der Befehlsfolge, der auf ihren Anfang zurückführt. Von Endlosschleifen abgesehen, enthält die Befehlsfolge mindestens einen bedingten Sprungbefehl, um das Eintreten in die Befehlsfolge bzw. das Verlassen der Befehlsfolge von einer Bedingung abhängig zu machen. Bild 3-10 zeigt drei grundsätzliche Schleifenformen in Flußdiagrammdarstellung mit den ihnen zugrundeliegenden C-Anweisungen. In den Teilbildern a und b ist die Bedingungsabfrage (bedingter Sprung) der Befehlsfolge vorangestellt, d.h., der Schleifendurchlauf kann ggf. abgewiesen werden. Um die Schleife zu schließen, muß die Befehlsfolge mit einem unbedingten Sprung abgeschlossen werden. In Teilbild c steht die Bedingungsabfrage am Ende der Befehlsfolge,

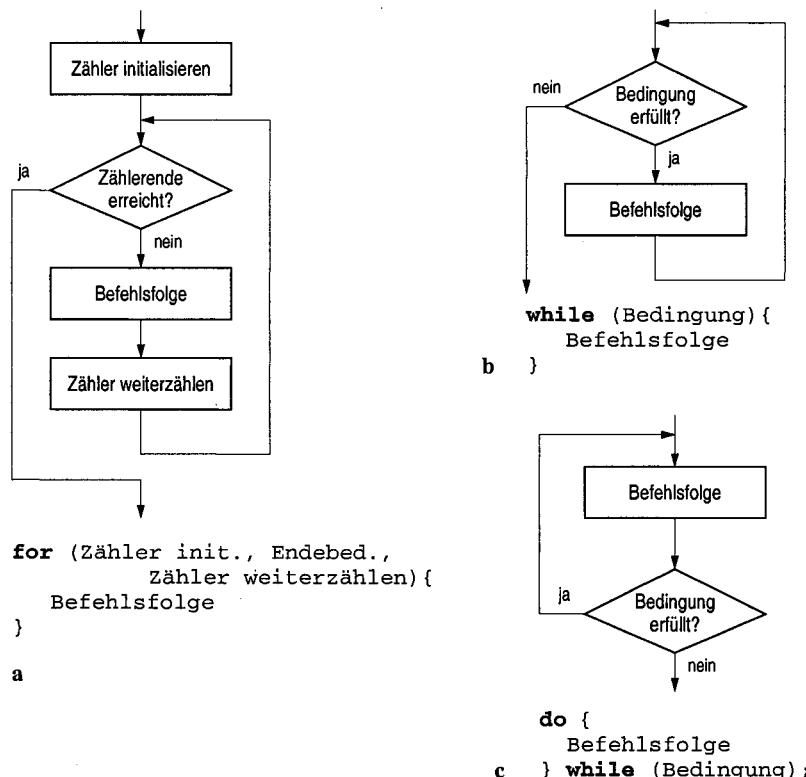


Bild 3-10. Programmschleifen in Flußdiagrammdarstellung mit C-Entsprechungen. a Vorangestellte, zählerbasierte Bedingungsabfrage (for-Anweisung), b vorangestellte, allgemeine Bedingungsabfrage (while-Anweisung), c nachgestellte, allgemeine Bedingungsabfrage (do-while-Anweisung)

weshalb die Schleife mindestens einmal durchlaufen wird, und der unbedingte Sprung entfällt.

Die Schleife nach Bild 3-10a bezeichnet man auch als *for-Schleife* (*for-Anweisung*) oder *Zählschleife* (*induktive Schleife*). Ihre Bedingung basiert auf einem Zählerwert, der vor Eintritt in die Schleife initialisiert und der am Ende eines jeden Schleifendurchlaufs inkrementiert oder dekrementiert wird. Bei der in Bild 3-10b gezeigten sog. *while-Schleife* (*while-Anweisung*) basiert die Bedingung auf der Auswertung von Rechengrößen, sie ist also allgemeiner gefaßt. Ihre typische Anwendung liegt bei iterativ zu berechnende Vorgängen (*iterative Schleife*). Bei der in Bild 3-10c dargestellten sog. *do-while-Schleife* (*do-while-Anweisung*) ist die Bedingung ebenfalls allgemein gefaßt, nur eben der Befehlsfolge nachgestellt. – Die Zuordnung einer dieser Schleifenformen zu einer bestimmten Aufgabenstellung ist oft nicht eindeutig möglich, da sich oft mehr als eine Schleifenform als Lösung eignet. Der Anwender muß sich also für eine der möglichen Formen entscheiden.

Induktive Schleife (Zählschleife). Bei einer *induktiven Programmschleife* ist die Anzahl der Schleifendurchläufe beim Eintritt in die Schleife bekannt und vom Verarbeitungsvorgang in der Schleife unabhängig. Zur Zählung der Schleifendurchläufe wird eine Laufvariable benutzt. Sie wird vor dem Eintritt in die Schleife initialisiert, mit jedem Schleifendurchlauf verändert und in der Bedingungsabfrage als Abbruchkriterium ausgewertet. Dabei ergeben sich zwei unterschiedliche Organisationsformen, die ggf. davon abhängig sind, ob bzw. wie die Laufvariable in der Schleife benutzt wird:

1. Die Laufvariable wird mit Null oder Eins initialisiert, mit jedem Schleifendurchlauf um Eins erhöht und in der Bedingungsabfrage mit der vorgegebenen Anzahl von Schleifendurchläufen verglichen (*aufwärtszählende Schleife*, siehe dazu Beispiel 3.7, S. 186).
2. Die Laufvariable wird mit der Anzahl von Schleifendurchläufen initialisiert, mit jedem Schleifendurchlauf um Eins vermindert und in der Bedingungsabfrage mit Null oder Eins verglichen (*abwärtszählende Schleife*). Hierzu folgendes Beispiel 3.5.

Beispiel 3.5. ▶ Induktive Schleife mit abwärtszählender Laufvariablen (for-Schleife). Es sollen N vorzeichenlose Dualzahlen, die in einem Datenbereich FELD stehen, summiert und das Resultat der Variablen SUM zugewiesen werden. – Bild 3-11 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Die Zählung erfolgt im Assemblerprogramm durch eine in R1 gespeicherte Laufvariable, die mit dem Wert von N initialisiert und bei jedem Schleifendurchlauf durch den SUB-Befehl um Eins vermindert wird. Die Abfrage am Schleifenanfang auf den Wert Null der Laufvariablen wird mittels des TEST-Befehls durchgeführt. Für die Adressierung der Feldelemente wird die Feldanfangsadresse mittels LEA nach R2 geladen und das Feld mit registerindirekter Adressierung mit Postinkrement adressiert. Die eigentliche Summation erfolgt aus Effizienzgründen in einem Register, hier R0; das Resultat wird nach Verlassen der Schleife der Variablen SUM zugewiesen. Im C-Programm wird die *for-Anweisung* zur Schleifenbildung benutzt. Ob die mit `feld[i]` programmierten Speicherzugriffe in die „schnelle“ registerindirekte Adressierung umgesetzt werden, ist vom Compiler abhängig.

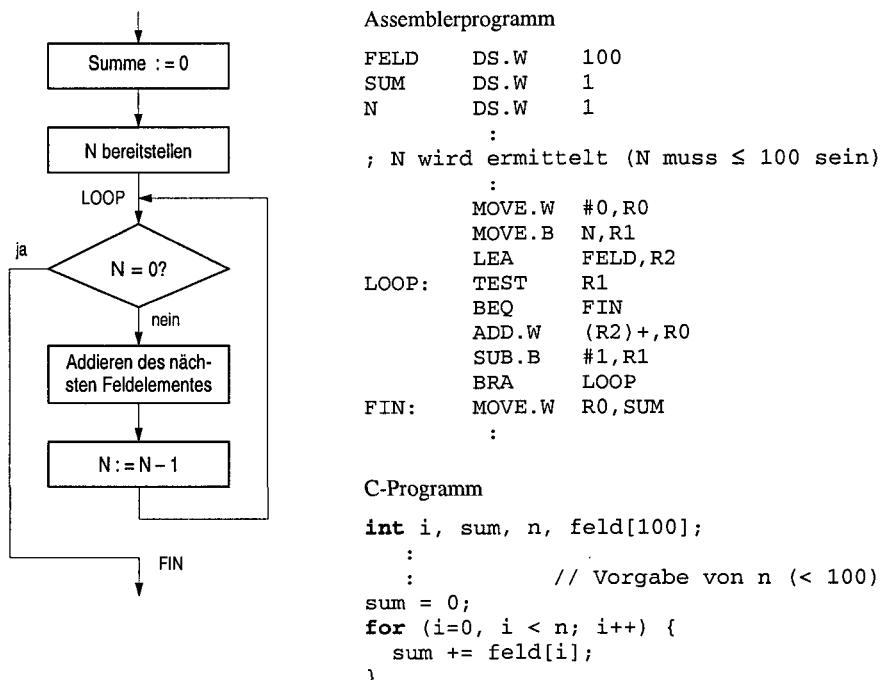


Bild 3-11. Flußdiagramm und Programme zu Beispiel 3.5: *Induktive Schleife mit abwärtszählender Laufvariablen (for-Schleife)*

Bei Schleifen zur Verarbeitung von Feldern (wie im obigen Beispiel) wird das Schleifenende häufig nicht durch eine Zählvariable, sondern durch Vergleich des für den Zugriff auf die Feldelemente verwendeten Zeigers mit der Feldendeadresse bestimmt (siehe dazu Beispiel 3.8, S. 187).

Iterative Schleife. Bei einer *iterativen Programmschleife* ist die Anzahl der Schleifendurchläufe beim Eintritt in die Schleife nicht bekannt. Sie hängt von Größen ab, die in der Befehlsfolge der Programmschleife errechnet werden oder die ggf. bereits vor dem Schleifenbeginn vorliegen. Die Bedingungsvorgabe für den Schleifenabbruch basiert also auf diesen Rechengrößen. Dazu folgendes Beispiel 3.6 für eine *while*-Schleife.

Beispiel 3.6. ▶ Iterative Schleife (while-Schleife). Für zwei vorzeichenlose Dualzahlen X und Y ist der größte gemeinsame Teiler zu ermitteln. – Bild 3-12 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Die der Schleifenbefehlsfolge vorangestellte Bedingungsabfrage basiert auf dem Vergleich der beiden Dualzahlen X und Y und wird im Assemblerprogramm mittels der Befehle CMP und BEQ durchgeführt. Bei Ungleichheit beider Variablen wird die Schleifenbefehlsfolge betreten und auf einen von zwei „Subtraktionspfaden“, X-Y bzw. Y-X, verzweigt. Realisiert ist diese Verzweigung durch den BLO-Befehl, der ebenfalls das Ergebnis des CMP-Befehls auswertet. Bei Gleichheit beider Variablen wird die Schleife verlassen (oder erst gar nicht betreten) und das Ergebnis der Berechnung der Variablen GGT zugewiesen. Im C-Pro-

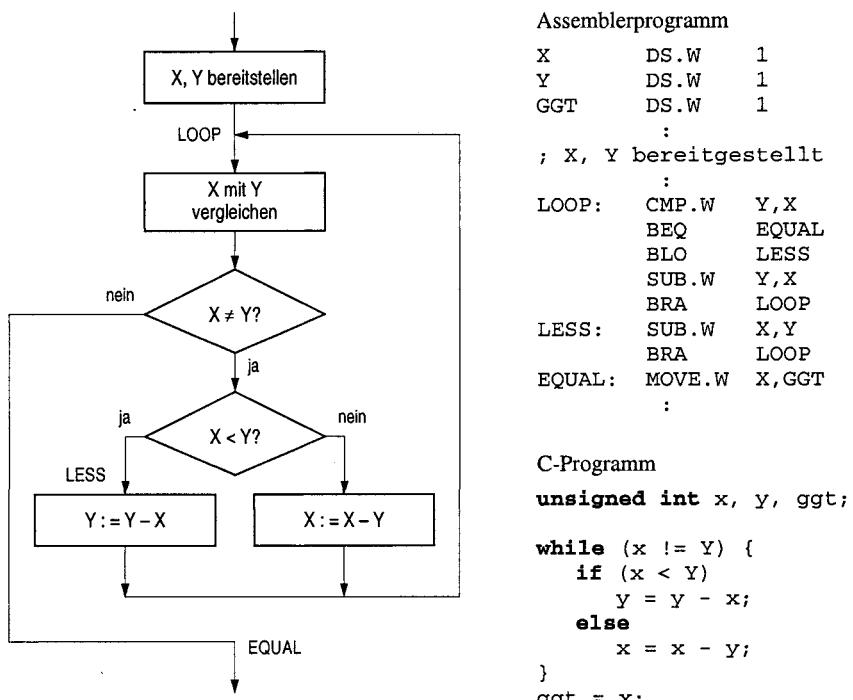


Bild 3-12. Flußdiagramm und Programme zu Beispiel 3.6: Iterative Schleife (while-Schleife)

gramm ist die Schleifenbedingung durch die *while*-Anweisung, die Verzweigung auf einen der beiden Subtraktionspfade durch die *if-else*-Anweisung beschrieben. ◀

Mischformen. Wie bereits erwähnt, lassen sich Aufgabenstellungen häufig nicht eindeutig in eine der vorgestellten Schleifenformen „pressen“. So auch nicht im folgenden Beispiel 3.7, dessen Flußdiagramm und Assemblerprogramm zwar die Struktur einer *do-while*-Schleife aufweisen, bei dem die Bedingungsvorgabe für die Schleifenbildung jedoch auf einer Zählung basiert und so eine Zählschleife, d.h. eine *for*-Schleife, nahelegt. In der C-Formulierung sind dementsprechend auch beide Formen als mögliche Varianten angegeben.

Beispiel 3.7. ▶ *Iterative/induktive Schleife (do-while-Schleife).* Es sollen die natürlichen Zahlen 1 bis N summiert und das Resultat der Variablen SUM zugewiesen werden. N sei als Variable vor-gegeben. – Bild 3-13 zeigt das Flußdiagramm sowie das Assemblerprogramm und zwei C-Pro-gramme. Im Assemblerprogramm wird zur Zählung der Schleifendurchläufe die Variable I eingeführt. Sie wird mit Eins initialisiert und mit jedem Durchlauf um Eins erhöht. Da sie in jedem Durchlauf gerade den Wert des jeweiligen Summanden aufweist, wird sie zur Summen-bildung benutzt. SUM, N und I stehen im Assemblerprogramm zur Veranschaulichung im Spei-cher; für eine effizientere Programmierung würde man sie wegen des häufigen Zugriffs im Registerspeicher halten. Als C-Programm kommt die Version A mit der *do-while*-Anweisung dem Assemblerprogramm am nächsten. Eine kompaktere Formulierung auf C-Ebene erhält man mit der *for*-Anweisung, wie sie in der Version B verwendet wurde.

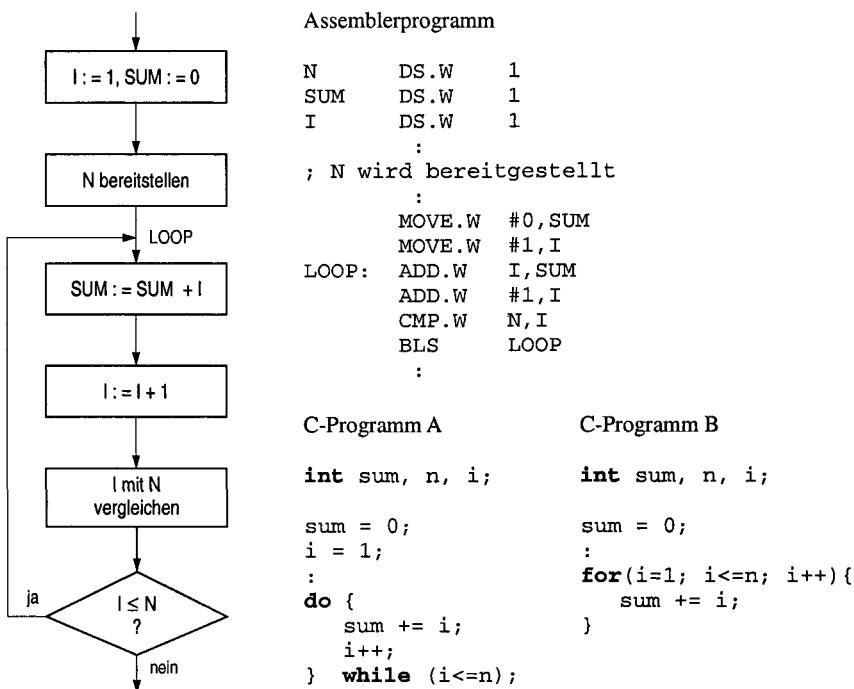
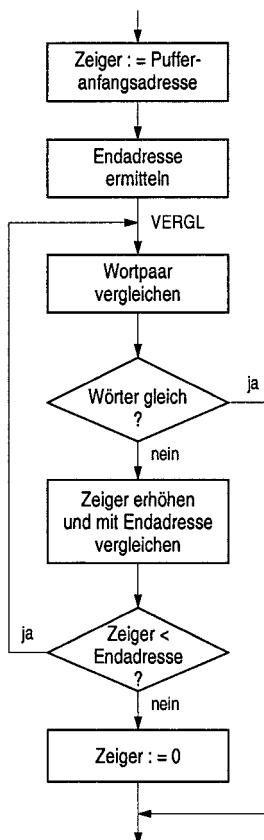


Bild 3-13. Flußdiagramm und Programme zu Beispiel 3.7: Iterative/induktive Schleife (do-while-Schleife) ▶

Da iterative Schleifen nicht immer zu einem Schleifenabbruch führen, nämlich dann nicht, wenn die Iteration nicht konvergiert, muß die Iterationsbedingung ggf. durch eine Induktionsbedingung zur Begrenzung der Anzahl von Schleifendurchläufen ergänzt werden. Damit enthält die Schleife für den Schleifenabbruch zwei Bedingungsabfragen, die in verschiedener Weise vorangestellt oder nachgestellt miteinander kombiniert werden können. Dazu folgendes Beispiel 3.8, das in der Flußdiagramm- und Assemblerdarstellung einer do-while-Schleife mit if-Anweisung in der Schleifenbefehlsfolge entspricht, das aber in der C-Version als for-Schleife mit if-Anweisung programmiert ist.

Beispiel 3.8. ► Iterative Schleife mit induktiver Begrenzung der Schleifendurchläufe (do-while-Schleife mit if-Anweisung). Ein Pufferbereich BUFFER mit $N = 512$ Wörtern soll nach einem bestimmten Schlüsselwort, das in der Speicherzelle KEY vorgegeben ist, mit aufsteigender Adresszählung durchsucht werden. Ist das gesuchte Wort im Puffer enthalten, so soll einer Zeigervariablen PTR die Adresse dieses Worts, andernfalls der Wert Null zugewiesen werden. – Bild 3-14 zeigt das Flußdiagramm sowie das Assembler- und das C-Programm. Im Assemblerprogramm erfolgt die Adressierung der Wörter im Puffer mittels des im Register R0 verwalteten Zeigers, der vor Schleifenbeginn mit der Pufferanfangsadresse initialisiert und nach jedem negativ verlaufenen Wortpaarvergleich (Befehlsfolge CMP, BEQ) per ADD-Befehl um 4 erhöht wird. Bei positivem Vergleich wird die Schleife verlassen, und der Zeiger zeigt auf das gefundene Wort. (Auf die registerindirekte Adressierung mit Postinkrement im CMP-Befehl wird hier verzichtet,

da sonst die bei positivem Vergleich an die Zeigervariable PTR übergebene Adresse bereits auf das nächste Wort im Puffer zeigen würde.) Der induktive Schleifenabbruch erfolgt mittels des Befehlspaares CMP, BLO, wonach an PTR der Wert Null übergeben wird. Das C-Programm ist als *für*-Schleife ausgelegt, mit induktivem Schleifenabbruch mittels der Zählvariablen i und mit iterativem Schleifenabbruch bei Erfüllung der Bedingung key == *r0. Die aktuelle Adresse des Pufferbereichs ist hier als Zeiger r0 angegeben, wobei es vom Compiler abhängt, ob er für die Speicherung des Adresswerts ein Register oder eine Speicherzelle einsetzt. Die Pufferlänge N ist als Konstante definiert.



Assemblerprogramm

```

N      EQU      512
BUFFER DS.W     N
KEY    DS.W     1
PTR    DS.W     1
:
VERGL: LEA      BUFFER, R0
       LEA      BUFFER+4*N, R1
       MOVE.W   KEY, R2
       CMP.W   R2, (R0)
       BEQ      FOUND
       ADD.W   #4, R0
       CMP.W   R1, R0
       BLO      VERGL
NOTFD: MOVE.W   #0, R0
FOUND:  MOVE.W   R0, PTR
:
  
```

C-Programm

```

#define N 512

int buffer[N], key, *ptr, *r0, i;

r0 = buffer;
ptr = NULL;
for (i=0; i<N; r0++, i++) {
    if (key == *r0) {
        ptr = r0;
        break;
    }
}
  
```

Bild 3-14. Flußdiagramm und Programme zu Beispiel 3.8: Iterative Schleife mit induktiver Begrenzung der Schleifendurchläufe (do-while-Schleife mit if-Anweisung)

Ein wesentlicher Unterschied zwischen dem C-Programm und dem Assemblerprogramm liegt darin, daß die Assemblerschleife bei Nichtauftreten des Schlüsselworts die in R0 stehende Adresse über den Bereich des Puffers hinaus erhöht, was beim C-Programm nicht der Fall ist. In unserem Beispiel ist das zwar ohne Belang, da unmittelbar danach der Inhalt von R0 sowieso auf Null gesetzt wird. Wollte man dies jedoch bei einer anderen Anwendung vermeiden, so müßte man das Assemblerprogramm wie in Bild 3-15 im linken Programmausschnitt gezeigt ändern: Vorgabe der Endadresse als Adresse des letzten Puffereintrags und Einführen eines zusätzlichen unbedingten Sprungs am Ende der Schleife. Bei RISCs mit *Delay-Slot* und (wie beim SPARC)

mit der Möglichkeit der *Annnullierung* des Delay-Slot-Befehls, ließe sich ein solches Assemblerprogramm allerdings sehr viel eleganter, nämlich ohne den unbedingten Sprung formulieren, wie im rechten Programmausschnitt gezeigt. (Um beide Programme miteinander besser vergleichen zu können, wurde den CISC-Registern R_i die RISC-Register li gleichgesetzt. Ferner ist zu beachten, daß im RISC-Programm das jeweilige Pufferelement zunächst in ein Register zu laden ist, hier nach I3.) Das Erhöhen der Adresse im Register R0 durch add, und darauf kommt es an, geschieht im Delay-Slot des zweiten Sprungbefehls, dessen Sprungbedingung gegenüber dem CISC-Programm invertiert ist und der beim Verlassen der Schleife diesen add-Befehl annulliert, d.h. nicht ausführt (siehe auch 1.4.2, Programmoptimierung).

N	EQU	512			
BUFFER	DS.W	N			
BEND	EQU	BUFFER+4*(N-1)			
KEY	DS.W	1			
PTR	DS.W	1			
 :					
	LEA	BUFFER, R0			
	LEA	BEND, R1			:
	MOVE.W	KEY, R2	vergl:	ld	10,0,13
VERGL:	CMP.W	R2,(R0)		cmp	12,13
	BEQ	FOUND		beq	found
	CMP.W	R1,R0		cmp	11,10
	BEQ	NOTFD		bne.a	vergl
	ADD.W	#4,R0		add	10,4,10
	BRA	VERGL	notfd:	:	
NOTFD:	MOVE.W	#0,R0	found:	:	
FOUND:	MOVE.W	R0,PTR			
 :					

Bild 3-15. Programmvarianten zu Beispiel 3.8, links CISC, rechts RISC

Das RISC-Programm hat aber auch einen Nebeneffekt, der ggf. bei der Fortsetzung des Programms zu berücksichtigen ist: Dadurch, daß der zweite cmp-Befehl im Delay-Slot des ersten Sprungbefehls steht, liegen an der Programmstelle found bereits die durch diesen zweiten cmp-Befehl erzeugten Bedingungsbits im Prozessorstatusregister vor. ▶

3.3 Unterprogrammtechniken

Unterprogramme sind in sich abgeschlossene Programmteile, die an beliebigen Stellen eines übergeordneten Programms, des *Haupt-* oder *Oberprogramms*, wiederholt aufgerufen und ausgeführt werden können. Nach Abarbeitung eines Unterprogramms wird das übergeordnete Programm hinter der Aufrufstelle fortgesetzt. Bild 3-16, Teilbild a, demonstriert diese Technik anhand des zeitlichen Ablaufs zweier Unterprogrammaufrufe, der durch aufeinanderfolgende Nummern kenntlich gemacht ist. Teilbild b zeigt im Gegensatz dazu den „linearen“ Programmablauf, wenn anstelle der Unterprogrammtechnik die Makrotechnik verwendet wird (vgl. 3.1.4). Beim Aufruf kann ein Unterprogramm analog zu einem Makro mit Rechengrößen versorgt werden, und es kann seinerseits Ergebnisse an das Oberprogramm zurückgeben. Man bezeichnet diese Größen als

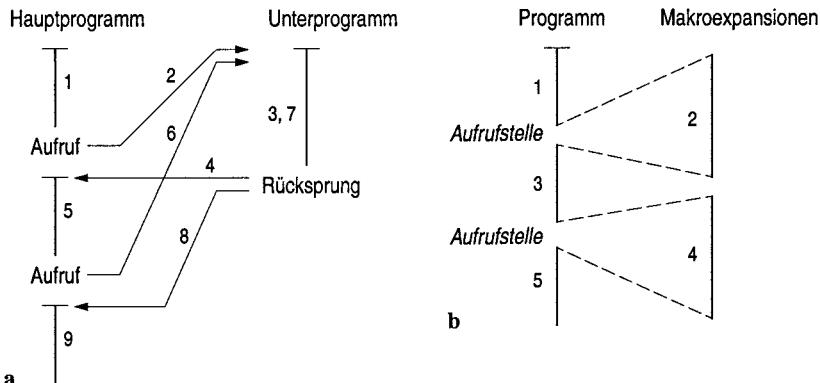


Bild 3-16. Programmfluß bei zweimaligem Aufruf **a** ein und desselben Unterprogramms, **b** ein und desselben Makros

Parameter und den Vorgang als *Parameterübergabe*. Anders als beim Makro werden diese Parameter jedoch nicht zur Übersetzungszeit, sondern zur Laufzeit übergeben.

Die Verwendung von Unterprogrammen bietet folgende Vorteile:

- Sich oft wiederholende Befehlsfolgen brauchen nur einmal programmiert und nur einmal gespeichert zu werden.
- Programme können übersichtlicher gestaltet werden, indem bestimmte Programmfunctionen in entsprechenden Unterprogrammen realisiert werden. Sie sind so auch leichter zu testen und besser zu dokumentieren.
- Ein Unterprogramm kann bei einigen Programmiersprachen unabhängig vom Oberprogramm und von anderen Unterprogrammen übersetzt werden. Bei einer Fehlerkorrektur im Unterprogramm braucht dann nur dieses neu übersetzt zu werden. (Häufig geschieht die getrennte Übersetzung auf Modulbasis; wobei ein Modul üblicherweise mehrere Unterprogramme umfaßt.)
- Standardfunktionen können dem Programmierer in Unterprogrammbibliotheken zur Verfügung gestellt werden.

3.3.1 Unterprogrammanschluß

Unterprogrammaufruf. Der Aufruf eines Unterprogramms erfolgt durch den Befehl JSR oder BSR mit Angabe der Adresse der Einsprungstelle im Unterprogramm. JSR bzw. BSR schreibt den aktuellen Befehlszählerinhalt (das ist die Adresse des auf JSR bzw. BSR nachfolgenden Befehls) zur späteren Nutzung als *Rücksprungadresse* auf den aktuellen *Stack* (User- oder Supervisor-Stack) und lädt den Befehlszähler mit der im Adreßteil von JSR oder BSR stehenden Unter-

programmadresse. Danach obliegt die Programmsteuerung dem Unterprogramm. Der Rücksprung in das Oberprogramm erfolgt mit dem Befehl RTS, der das Unterprogramm abschließt. RTS liest den letzten Stackeintrag – die Rücksprungadresse – und lädt ihn in den Befehlszähler. Wird der Stack vom Unterprogramm (z.B. mit PUSH- und POP-Befehlen) auch zur Datenspeicherung benutzt, so ist darauf zu achten, daß der RTS-Befehl genau denjenigen Stackpointer vorfindet, der auf die Rücksprungadresse zeigt; andernfalls erfolgt ein unkontrollierter Sprung. Oft werden deshalb die Rücksprungadressen und die Daten getrennt von einander in einem Programm- und einem Datenstack angelegt.

Anmerkung. Bei RISCs wird die Rücksprungadresse üblicherweise nicht auf dem Stack, sondern in einem der allgemeinen Prozessorregister (ggf. in als Stack organisierten Registerfenstern) oder in einem extra dafür ausgewiesenen Prozessorregister (*Link-Register*) abgelegt. – Zum Unterprogrammanschluß bei RISCs, speziell SPARC, siehe 3.4.2.

Retten des Prozessorstatus. Sollen das Unterprogramm und das Oberprogramm die Prozessorregister völlig unabhängig voneinander benutzen, so muß beim Unterprogrammaufruf der *Prozessorstatus* gerettet und bei der Rückkehr ins Oberprogramm wieder geladen werden. Zum Prozessorstatus gehören neben dem Befehlszählerstand der Inhalt des Statusregisters und die Inhalte der allgemeinen Prozessorregister. Während das Retten und Laden des Befehlszählers automatisch bei der Ausführung der Befehle JSR und RTS erfolgt, muß es für die übrigen Register explizit programmiert werden. Für das Retten und spätere Laden der allgemeinen Register stehen neben den herkömmlichen MOVE- und PUSH/POP-Befehlen häufig Erweiterungen dieser Befehle für Mehrfachtransporte zur Verfügung (mit Angabe einer Registerliste im Befehl). Für das Retten und Laden des Statusregisters gibt es Befehle wie MOVESR und MOVECC. Während der erste ein privilegierter Befehl ist, ist der zweite nicht privilegiert, transportiert dafür aber auch nur die Bedingungsbits CC. Dies ist jedoch keine Einschränkung, da im User-Modus die Modusbits des Statusregisters (Supervisorbyte) nicht verändert werden. Als Ort für das Speichern der Statusinformationen kann der *Stack* verwendet werden, der auch für das Retten der Rücksprungadresse benutzt wird.

Das Retten und das Zurückladen des Prozessorstatus kann sowohl im Oberprogramm als auch im Unterprogramm durchgeführt werden. Welche Möglichkeit gewählt wird, ist eine Frage der Programmorganisation und des Programmierstils.

Beispiel 3.9. ► Retten und Laden des Prozessorstatus. Beim Aufruf eines Unterprogramms im User-Modus soll ein Teil des Prozessorstatus, bestehend aus der Rücksprungadresse, den Bedingungsbits CC und den allgemeinen Registern R0 bis R3 sowie R6 auf den User-Stack gerettet werden. – Bild 3-17 zeigt das Assemblerprogramm und die Stackbelegung. Das Retten der Rücksprungadresse erfolgt durch den Befehl BSR. Die Bedingungsbits CC werden vom Unterprogramm mit dem Befehl MOVECC.W (Stack-Alignment auf Wortgrenze, sofern vom Prozessor gefordert) auf den User-Stack geschrieben. Für das Retten der allgemeinen Prozessorregister steht der Mehrfachtransportbefehl MOVEM (move multiple registers) zur Verfügung, in dem die zu transportierenden Register in einer Registerliste aufgeführt sind. Ansonsten müßten die Transporte einzeln durch MOVE erfolgen. Das Laden des Status beim Abschluß des Unterprogramms erfolgt in umgekehrter Reihenfolge; als letztes übergibt RTS über die im Stack gespeicherte

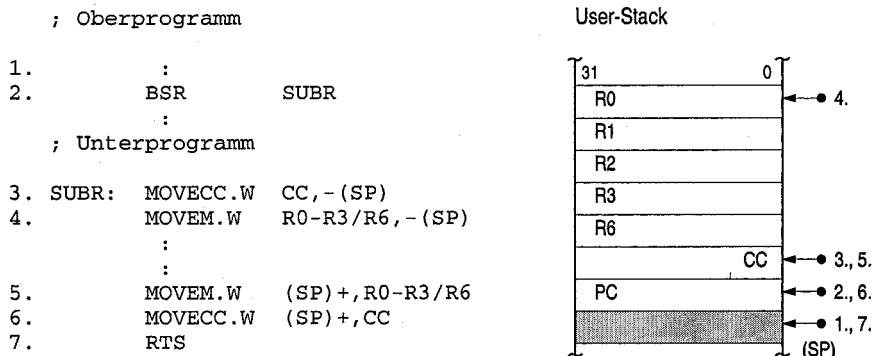


Bild 3-17. Programm und Stackbelegung zu Beispiel 3.9: *Retten und Laden des Prozessorstatus*. Die Pfeile rechts geben den jeweiligen Stand des Stackpointers an

Rücksprungadresse die Programmsteuerung wieder an das Oberprogramm. Die korrespondierenden Nummern im Programm und in der Abbildung zeigen die Wirkung der Befehle auf die Belegung des Stacks. ▲

3.3.2 Parameterübergabe

Für die Übergabe von Parametern an das Unterprogramm bzw. zurück ins Oberprogramm gibt es vier Gesichtspunkte, die die Programmierungstechnik beeinflussen:

1. Art des Parameters, z.B. Wert, Adresse,
2. Typ des Parameters, z.B. vorzeichenlose Dualzahl (unsigned) oder 2-Komplement-Zahl (signed), jeweils in bestimmten Formaten (Byte, Halbwort, Wort),
3. Ort der Parameterübergabe, z.B. allgemeine Prozessorregister oder Stack,
4. Anzahl der Parameter.

Art des Parameters. Bei einem Unterprogrammaufruf mit Wertübergabe (*Wertaufruf, call by value*) wird der Wert einer Rechengröße in den Datenbereich des Unterprogramms kopiert; dort muß ein Speicherplatz für diesen Wert reserviert sein. Das Unterprogramm hat direkten Zugriff auf den Wert und kann ihn verändern, ohne daß der ursprüngliche Wert im Oberprogramm beeinflußt wird. Bei einem Unterprogrammaufruf mit Adressübergabe (*Referenzaufruf, call by reference*) wird die Adresse einer Rechengröße in das Unterprogramm transportiert, d.h. der Operand selbst verbleibt im Oberprogramm, im Unterprogramm muß jedoch Speicherplatz für die Adresse reserviert sein. Der Zugriff auf den Operanden durch das Unterprogramm erfolgt indirekt über den Adreßparameter. – Die *Rückgabe eines Ergebnisses* an das Oberprogramm erfolgt üblicherweise mittels eines

Adreßparameters, sie kann aber auch als Wertrückgabe erfolgen (z.B. in einem der allgemeinen Register oder auf dem Stack).

Die Parameter, mit denen das Unterprogramm als fiktive Größen geschrieben wird, nennt man *formale Parameter*. Vor der Ausführung des Unterprogramms müssen diese durch die zum Zeitpunkt des Aufrufs bekannten Parameter, die *aktuellen Parameter*, ersetzt werden. Parameter, die an das Unterprogramm übergeben werden und deren Werte dort benutzt, aber nicht verändert werden, bezeichnet man als *Eingangsparameter*. Dementsprechend bezeichnet man Parameter, denen erst im Unterprogramm Werte zugewiesen werden und die auch erst danach benutzt werden können, als *Ausgangsparameter*. Parameter, deren Werte im Unterprogramm zuerst benutzt und danach verändert werden, bezeichnet man als *Übergangs-* oder *Durchgangsparameter*. – Den prinzipiellen Ablauf der Parameterübergabe zeigt Bild 3-18. In der Flußdiagrammdarstellung ist ein Unterprogrammaufruf durch doppelte Seitenlinien im Rechtecksymbol gekennzeichnet.

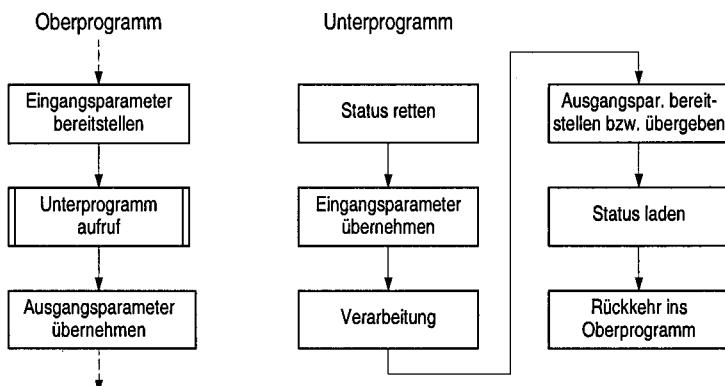


Bild 3-18. Unterprogrammanschluß mit Parameterübergabe

Ob der Wertauftrag, der Referenzauftrag oder eine Kombination beider Möglichkeiten verwendet wird, ist von Fall zu Fall verschieden und hängt unter anderem von folgenden Kriterien ab:

- Zeitaufwand für die Parameterübergabe,
- Speicherplatzbedarf für die Parameter,
- Adressierungsaufwand für den Zugriff durch das Unterprogramm,
- Schreibschutz für die Daten des Oberprogramms.

Die ersten drei Punkte legen es nahe, Einzelwerte durch Wertübergabe und Felder, Strings usw. durch Adreßübergabe dem Unterprogramm zur Verfügung zu stellen.

Typ des Parameters. Bezuglich des Parametertyps gibt es zunächst folgende Grundregel: Die Datentyp-/Formatangaben für den formalen und für den aktuellen Parameter müssen miteinander übereinstimmen. Allerdings wird in bestimmten Fällen von dieser Grundregel abgewichen und bei der Parameterübergabe eine Typenwandlung vorgenommen. Ein Grund dafür kann sein, ein Unterprogramm universell einsetzbar zu machen, indem es mit Parametern mit z.B. wechselnden Datenformaten aufgerufen werden kann. Auf der Assemblerebene gibt es hier letztlich keine Einschränkung in der Möglichkeit, solche Besonderheiten zu programmieren. Aber auch in C sind implizite Typenwandlungen vorgesehen. In anderen Programmiersprachen wie z.B. Modula 2 und Ada ist hingegen eine strenge Typenbindung vorgegeben.

Ort der Parameterübergabe. Der Ort der Parameter muß sowohl dem Ober- als auch dem Unterprogramm zugänglich sein, was bei den folgenden drei Möglichkeiten gegeben ist:

1. Man schreibt die Parameter vor dem Unterprogrammaufruf in den Registerspeicher, von wo sie im Unterprogramm unmittelbar gelesen werden können.
2. Man schreibt die Parameter vor dem Unterprogrammaufruf auf den Stack; der Parameterzugriff erfolgt über den im Stackpointerregister gespeicherten Stackpointer.
3. Man faßt die Parameter im Programmreich des Oberprogramms zu einem Parameterfeld zusammen, das unmittelbar hinter dem Unterprogrammsprungbefehl BSR steht; der Parameterzugriff erfolgt über die im User- bzw. Supervisor-Stack gespeicherte Rücksprungadresse des Unterprogramms.

Welche der Möglichkeiten der Parameterübergabe bevorzugt wird, hängt von der Problemstellung, von den Datenbereichen und vom persönlichen Stil des Programmierers ab. – Im folgenden illustrieren wir die beiden erstgenannten Möglichkeiten, da sie am gebräuchlichsten sind, und greifen dazu das in Beispiel 3.8 (S. 187) dargestellte Problem des Durchsuchens eines Pufferbereichs nach einem Schlüsselwort auf.

Parameterübergabe im Registerspeicher. Die Parameterübergabe im Registerspeicher erfordert den geringsten Organisationsaufwand und ist dementsprechend effizient. Sie setzt jedoch voraus, daß die Anzahl der Parameter die Anzahl der zur Verfügung stehenden Register nicht übersteigt. Auf diese Art der Parameterübergabe ist z.B. der SPARC mit der Besonderheit seiner Registerfenster zugeschnitten (siehe dazu 3.4.2). Wir legen dem folgenden Beispiel einen allgemeinen Registerspeicher zugrunde, wie er in den meisten CISC- und RISC-Prozessoren zu finden ist.

Beispiel 3.10. ► Parameterübergabe im Registerspeicher. Das in Beispiel 3.8 beschriebene Durchsuchen eines Pufferbereichs mit 512 Wörtern nach einem bestimmten Schlüsselwort soll von einem Unterprogramm durchgeführt werden. Beim Auffinden dieses Worts ist dem Oberprogramm dessen Adresse als Pufferzeiger zu übermitteln; steht das Wort nicht im Puffer, so ist als

; Oberprogramm			; Unterprogramm		
N	EQU	512	VERGL:	CMP.W	R2, (R0)
BUFFER	DS.W	N		BEQ	FOUND
BEND	EQU	BUFFER+4*(N-1)		CMP.W	R1, R0
KEY	DS.W	1		BEQ	NOTFD
PTR	DS.W	1		ADD.W	#4, R0
:				BRA	VERGL
LEA		BUFFER, R0	NOTFD:	MOVE.W	#0, R0
LEA		BEND, R1	FOUND:	MOVE.W	R0, (R3)
MOVE.W		KEY, R2		RTS	
LEA		PTR, R3			
BSR		VERGL			
:					

Bild 3-19. Programme zu Beispiel 3.10: Parameterübergabe im Registerspeicher

Zeiger der Wert Null zurückzugeben. – Bild 3-19 zeigt das Ober- und das Unterprogramm. Die Eingangsparameter werden als Adressen (in R0 die Pufferanfangsadresse BUFFER als Initialwert des Zeigers, in R1 die Pufferendadresse BUFFER+4*(N-1)) und als Wert (in R2 das Schlüsselwort KEY) an das Unterprogramm übergeben. Zur späteren Rückgabe des Zeigers an das Oberprogramm wird beim Unterprogrammaufruf zusätzlich die Adresse PTR in R3 als Adresse des Ausgangsparameters übergeben. ◀

Parameterübergabe auf dem Stack. Als bevorzugter Ort für die Parameterübertragung gilt der Stack. Er stellt einerseits keine Einschränkung für die Anzahl der Parameter dar, wie dies beim Registerspeicher der Fall ist, und wird andererseits meist sowieso zur Speicherung des Prozessorstatus und der lokalen Variablen eines Unterprogramms benutzt. Dementsprechend bewirkt er eine Vereinheitlichung der für die Parameter-, Status- und Datenzugriffe eingesetzten Mechanismen. Nachteilig ist allerdings, daß Stackzugriffe als Speicherzugriffe mehr Zeit als Registerzugriffe kosten. – Um die Adressierung der während der Unterprogrammausführung im Stack „verbleibenden“ Daten von einem sich möglicherweise verändernden Stackpointer unabhängig zu machen, wird häufig ein *Framepointer* eingesetzt, der während eines Unterprogrammlaufs eine feste Bezugsadresse aufweist, wie in Beispiel 3.11 gezeigt.

Beispiel 3.11. ► **Parameterübergabe auf dem Stack.** Beispiel 3.10 soll so modifiziert werden, daß die Parameter auf dem Stack übergeben werden. Der Stack soll außerdem den Status des Oberprogramms für die im Unterprogramm benutzten Register R0 bis R3 aufnehmen. Ferner soll ein Framepointer als feste Stackadresse für den Zugriff auf die Parameter verwendet und verwaltet werden. – Bild 3-20 zeigt das Ober- und das Unterprogramm sowie deren Wirkung auf die Stackbelegung, verdeutlicht durch die korrespondierenden Nummern in den Programmen und in der Abbildung. Das Oberprogramm schreibt zunächst die vier Parameter mittels LEA (Adressen) und MOVE (Wert) auf den Stack, jedoch gegenüber Beispiel 3.10 in umgekehrter Reihenfolge, um diese im Unterprogramm mittels des MOVEM-Befehls (Registerliste!) auf einfache Weise übernehmen zu können. Das Unterprogramm seinerseits rettet zunächst den aktuellen Inhalt des Framepointerregisters FP auf den Stack und lädt es dann mit dem momentanen Inhalt des Stackpointerregisters SP. Es erzeugt sich damit eine feste Basisadresse für die Zugriffe auf die im Stack stehenden Parameter. Der Stackpointer selbst wird im Programm bereits durch das dann folgende Retten der Inhalte von R0 bis R3 verändert. Die Parameter werden schließlich durch MOVEM in

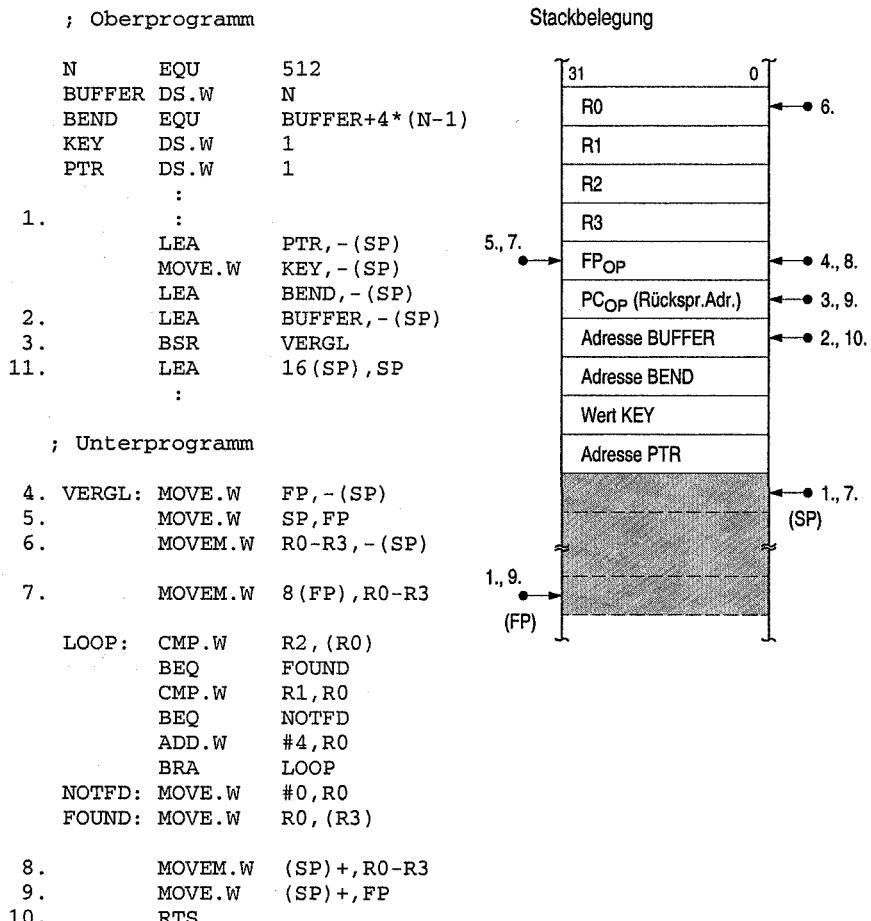


Bild 3-20. Programm und Stackbelegung zu Beispiel 3.11: *Parameterübergabe auf dem Stack*. Die Pfeile links an der Speicherdarstellung geben den jeweiligen Stand des Framepointers, die Pfeile rechts den des Stackpointers an

die Register R0 bis R3 übernommen. Am Ende des Unterprogramms wird der Registerstatus (R0 bis R3) des Oberprogramms wiederhergestellt und der ursprüngliche Framepointer wieder nach FP geladen. Nach dem Rücksprung in das Oberprogramm zeigt der Stackpointer auf das Parameterfeld. Mit dem LEA-Befehl wird dieser Stackbereich wieder freigegeben. ▶

Unterprogrammtechnik in C. Ergänzend zu den oben behandelten Unterprogrammtechniken in Assembler zeigt das nachfolgende Beispiel 3.12 für unsere Aufgabenstellung den Unterprogrammaufruf sowie die Unterprogrammdefinition als C-Funktion mit Wertrückgabe. Dieses Beispiel wurde bewußt von den beiden vorangehenden Beispielen entkoppelt, da nun die Aufgabenstellung nicht mehr auf die Demonstration bestimmter Assemblertechniken zugeschnitten ist, sondern vom Problem her neu definiert und programmiert ist. Bezüglich des Anschlusses

an das Hauptprogramm wird bei C-Compilern zwar grundsätzlich von einer Parameterübergabe auf dem Stack ausgegangen, ggf. erfolgt sie aber in Vermischung mit einer Parameterübergabe im Registerspeicher. Für Details zur Organisation des Stacks sei auf 4.3.2 verwiesen.

Beispiel 3.12. ► *Unterprogrammdarstellung in C.* Die in den Beispielen 3.10 und 3.11 behandelte Aufgabe ist nachfolgend als C-Funktion dargestellt, die hier vor ihrem Aufruf definiert ist.

```
#define N 512

int *search(int key, int *ptr, int length) {      // Definition
    int i;
    for (i=0; i<length; ptr++, i++) {
        if (key == *ptr)
            return ptr;
    }
    return NULL;
}

main () {
    int buffer[N], key, *pointer;
    :
    pointer = search(key, buffer, N);           // Aufruf
    :
}
```

Beim Aufruf der Funktion `search()` werden ihr als Parameter übergeben: das Schlüsselwort (`key`), der Name des Pufferbereichs (`buffer`, in C gleichbedeutend mit einem Zeiger namens `buffer` auf den Pufferbereichsanfang) sowie die Länge des Pufferbereichs (`N`, hier als Konstante mit dem Wert 512 definiert). Bei erfüllter Bedingung `key == *ptr` in `search()` (iterativer Schleifenabbruch) gibt die Funktion die von `ptr` repräsentierte Adresse des gefundenen Schlüsselworts zurück. Bei Nichtauffinden des Schlüsselworts (induktiver Schleifenabbruch) gibt sie `NULL` zurück. Der jeweilige Funktionswert wird der Zeigervariablen `pointer` zugewiesen. ◀

Anzahl der Parameter. In den obigen Beispielen wurde davon ausgegangen, daß die Anzahl der Parameter bei jedem Unterprogrammaufruf gleich und beim Schreiben des Unterprogramms bekannt ist. Es gibt jedoch auch Anwendungen, bei denen sich die Parameteranzahl von Aufruf zu Aufruf ändert. In diesen Fällen muß sie dem Unterprogramm als zusätzlicher Parameter mitgeteilt werden, so daß dieses die Parameterübernahme abhängig von der Anzahl vornehmen kann.

3.3.3 Geschachtelte Unterprogramme

Der Aufruf von Unterprogrammen ist nicht auf ein einziges Oberprogramm als aufrufendes Hauptprogramm beschränkt; ein Unterprogramm kann seinerseits – wie oben bereits gezeigt – wieder Unterprogramme aufrufen und erhält damit die Funktion eines Oberprogramms. Der Aufrufmechanismus kann sich somit ausgehend vom Hauptprogramm über mehrere Unterprogrammstufen erstrecken. Man spricht von einer *Schachtelung von Unterprogrammen*. Bezüglich des Auf-

ruforts und des Aufrufzeitpunkts werden drei Arten geschachtelter Unterprogramme unterschieden:

1. einfache Unterprogramme,
2. rekursive (wiederaufrufbare) Unterprogramme und
3. reentrant (wiedereintrittsfeste) Unterprogramme.

Einfache Unterprogramme. Bild 3-21 zeigt das Schema einer Schachtelung einfacher Unterprogramme. Verschiedene Programme rufen sich, ausgehend von einem Hauptprogramm, nacheinander auf, wobei die jeweilige Parameterübergabe in den beiden oben beschriebenen Weisen erfolgen kann. Der für das Retten des Status und ggf. für die Parameterübergabe benutzte *Stack* entspricht mit seinem Last-in-first-out-Mechanismus genau der Schachtelungsstruktur der Unterprogrammaufrufe: Die beim Aufruf der Unterprogramme nacheinander auf den Stack geschriebenen Informationen (im einfachsten Fall nur die durch die BSR-Befehle gespeicherten Rücksprungadressen) werden bei der Rückkehr in die Oberprogramme in umgekehrter Reihenfolge wieder gelesen.

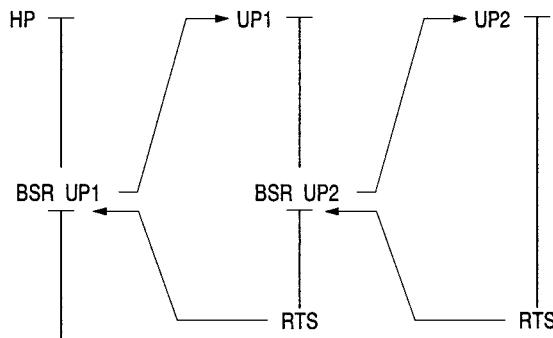


Bild 3-21. Schachtelung „einfacher“ Unterprogramme

Die jeweiligen Datenbereiche der Unterprogramme können statisch angelegt sein, d.h., jedem Unterprogramm wird entsprechend seinen DS-Anweisungen Speicherplatz für seine Variablen zugewiesen. Sie können aber auch dynamisch angelegt sein, so wie für die beiden folgenden Varianten geschachtelter Unterprogramme beschrieben. In Bild 3-21 wird von einer statischen Datenspeicherung ausgegangen, weshalb dort auch keine Hinweise zur Verwaltung von Datenbereichen stehen.

Rekursive Unterprogramme. Rekursive (wiederaufrufbare) Unterprogramme sind Unterprogramme, die wieder aufgerufen werden, bevor sie ihre gegenwärtige Verarbeitung abgeschlossen haben. Dies geschieht entweder direkt, wenn sich das Unterprogramm selbst aufruft, oder indirekt, wenn der Wiederaufruf auf dem Umweg über ein oder mehrere andere Unterprogramme erfolgt. Dementsprechend spricht man von direkt- und indirekt-rekursiven Unterprogrammen.

Gegenüber einfachen Unterprogrammen muß bei rekursiven Unterprogrammen dafür gesorgt werden, daß mit jedem Aufruf ein neuer Datenbereich für das Unterprogramm bereitgestellt wird, damit der zuletzt aktuelle Datenbereich nicht beim erneuten Aufruf überschrieben wird. Die Datenbereichszuordnung muß dementsprechend zur Laufzeit, d.h. dynamisch erfolgen. Hier bietet sich aufgrund der Schachtelungsstruktur der Unterprogrammaufrufe der *Stack* als Speicherort für die Unterprogrammdaten (und natürlich auch für die Parameterübergabe) an. Der zuletzt aktuelle und bei erneutem Aufruf des Unterprogramms verdeckt gehaltene Stack-Datenbereich gilt als Programmstatus der letzten Unterprogrammkarnation.

Bild 3-22 zeigt das Schema einer rekursiven Unterprogrammschachtelung. Ein Hauptprogramm ruft ein Unterprogramm auf, das sich wiederholt selbst aufruft, und zwar so lange, bis der Selbstaufruf durch eine Programmverzweigung übersprungen wird. Grundsätzlich muß hierbei gewährleistet sein, daß vor dem erneuten Aufruf des Unterprogramms sein momentaner Datenbereich gerettet und nach der Rückkehr in das Unterprogramm, also direkt hinter dem Aufrufbefehl, sein vorheriger Datenbereich wiederhergestellt wird. Eleganter und universeller ist es jedoch, wenn – wie im Bild und im folgenden Beispiel 3.13 gezeigt – bereits unmittelbar nach dem Eintritt in das Unterprogramm ein neuer Datenbereich eingerichtet wird, und wenn erst unmittelbar vor Verlassen des Unterprogramms sein vorheriger Datenbereich (Programmstatus) wieder zugänglich gemacht wird.

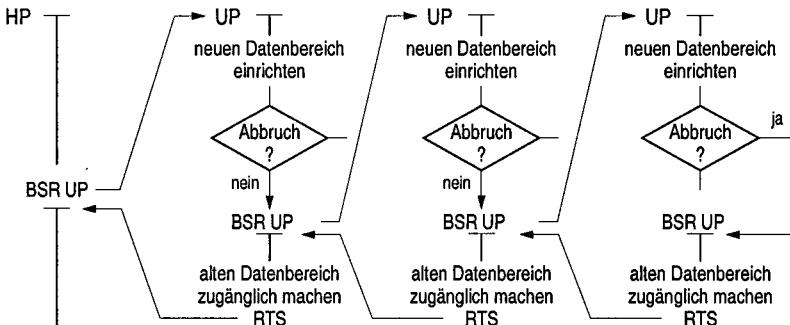


Bild 3-22. Schachtelung bei rekursiven Unterprogrammen

Beispiel 3.13. ► *Rekursive Unterprogrammaufrufe.* Die Fakultät einer natürlichen Zahl n , die im Datenbereich des Hauptprogramms unter der Adresse N steht, soll durch ein rekursives Unterprogramm FAK berechnet werden. Der mit jedem Aufruf von FAK erforderliche neue Datenbereich soll im Stack angelegt und, wie in Beispiel 3.11 (S. 195) gezeigt, durch einen Framepointer verwaltet werden. – Bild 3-23 zeigt das Hauptprogramm und das Unterprogramm in Assemblerdarstellung sowie die Speicherbelegung unter Betonung des dynamischen Verhaltens des Stackbereichs während der einzelnen Berechnungsschritte. Diese sind anhand der Numerierung einzelner Befehlausführungen und der damit einhergehenden Änderungen des Stackpointers und des Framepointers nachvollziehbar. Das Unterprogramm zeigt (durch Leerzeilen kenntlich gemacht) eine Eingangsbefehlsfolge zur Einrichtung des jeweils neuen Stackdatenbereichs, den

```

; Hauptprogramm

N      DS.H     1
      :
1.    :
2.    LEA      N,-(SP)
3.    BSR      FAK
18.   ADD.W   #4,SP
      :

; Unterprogramm FAK

4.  FAK: MOVE.W  FP,-(SP)
5.    MOVE.W  SP,FP
6.    SUB.W   #4,SP

7.    MOVE.H   [8(FP)],-2(FP)
8.    MOVE.H   -2(FP),-4(FP)
9.    SUB.H    #1,-4(FP)
      BHI     L1
12.   MOVE.H   #1,RO
      BRA     L2
10.  LL:    LEA     -4(FP),-(SP)
11.   BSR     FAK
16.   ADD.W   #4,SP
17.   MULU.H  -2(FP),RO

13.  L2:    MOVE.W  FP,SP
14.   MOVE.W  (SP)+,FP
15.   RTS

```

C-Programm

```

int fak(int n) { // Definition
    if (n > 1)
        return (fak(n - 1) * n);
    else;
        return (1);
}

main () {
    int n, y;
    :
    y = fak(n);           // Aufruf
}

```

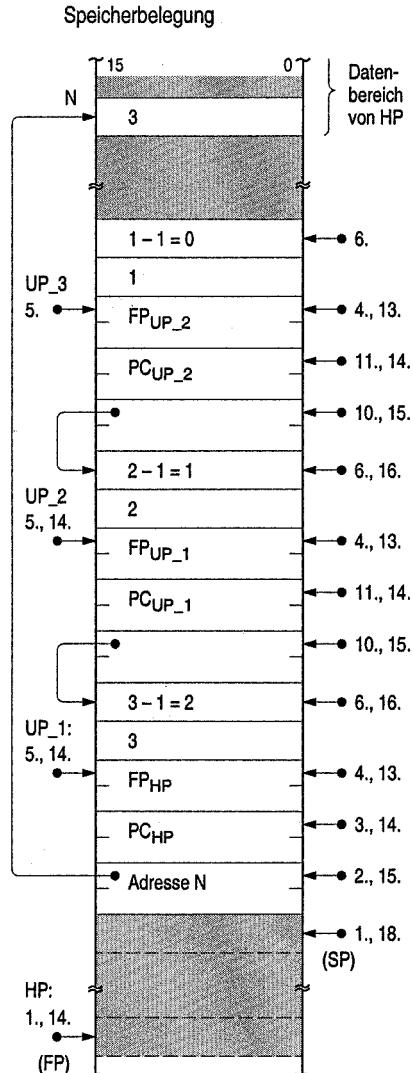


Bild 3-23. Assemblerprogramm und Speicherbelegung sowie C-Programm zu Beispiel 3.13: *Rekursive Unterprogrammaufrufe*. Der obere Teil der Speicherdarstellung umfaßt den statischen Datenbereich des Hauptprogramms (HP) mit der Variablen N, der untere Teil den für die Fakultätsberechnung verwendete Stack. Die Pfeile links am Stack geben den jeweiligen Stand des Framepointers, die Pfeile rechts den des Stackpointers an; UP_i bezeichnet den i-ten Aufruf des Unterprogramms FAK

eigentlichen Programmrumph für die Fakultätsberechnung und eine Ausgangsbefehlsfolge für die Freigabe dieses Stackdatenbereichs. Die Fakultätsberechnung basiert darauf, daß n mit jedem Aufruf von FAK um Eins vermindert im jeweils neuen Stackdatenbereich gespeichert wird.

Sobald n den Wert Null erreicht hat, werden die nacheinander begonnenen Unterprogrammläufe in umgekehrter Reihenfolge beendet, wobei jeweils ein Multiplikationsschritt ausgeführt wird. Dieser erfolgt mittels des Befehls MULU.H (multiply unsigned, Halbwortformat für Multiplikand und Multiplikator, Wortformat für das Produkt) unter Benutzung von R0 als Zielregister. In diesem Register wird schließlich der Wert $n!$ an das Hauptprogramm übergeben. Zu beachten ist, daß sich die Parameterübergabe beim ersten Aufruf von FAK auf die Variable N im statischen Datenbereich des Hauptprogramms bezieht, und daß sie sich bei den rekursiven Aufrufen auf den jeweils um Eins vermindernden Eintrag im Stack bezieht. In beiden Fällen handelt es sich um eine Adressübergabe. – Bild 3-23 zeigt zusätzlich das C-Programm, das wegen der verdeckten Stackverwaltung sehr viel übersichtlicher als das Assemblerprogramm ist. ▶

Reentrant Unterprogramme. Reentrant (wiedereintrittsfeste) Unterprogramme können von unterschiedlichen Programmen aus aufgerufen werden, ohne daß sie bei jedem Aufruf vollständig abgearbeitet sein müssen. Im Gegensatz zu rekursiven Programmen ist bei ihnen jedoch der Zeitpunkt des Wiederaufrufs nicht bekannt, so z.B. beim wiederholten Aufruf durch verschiedene Interruptprogramme. Das an beliebigen Stellen unterbrechbare Unterprogramm muß dementsprechend bei jedem Neuauftrag dafür sorgen, daß es für seine lokalen Daten einen neuen Datenbereich bereitstellt, und daß sein zuletzt aktueller Datenbereich erhalten bleibt. Dementsprechend ist auch hier die oben beschriebene Technik der dynamischen Speicherverwaltung anzuwenden, indem beim Eintritt in das Unterprogramm ein neuer Datenbereich auf dem Stack bereitgestellt, und beim Verlassen des Unterprogramms der vorherige Datenbereich reaktiviert wird.

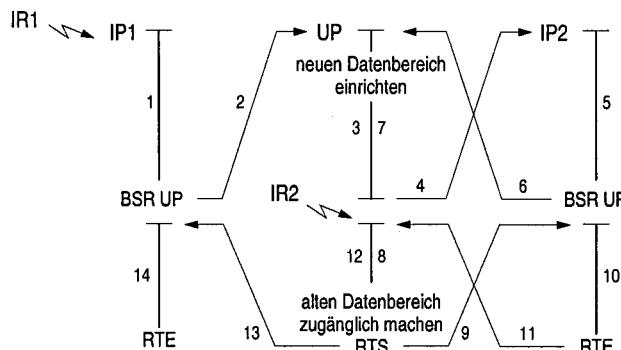


Bild 3-24. Zweimaliger Aufruf eines reentranten Unterprogramms

Bild 3-24 zeigt den zweimaligen Aufruf eines reentranten Unterprogramms. Das Auftreten eines Interrupts IR1 führt auf ein *Interruptprogramm* IP1 (1), das ein Unterprogramm UP auruft (2). Während der Ausführung des Unterprogramms (3) im Auftrag von IP1 wird einem Interrupt höherer Priorität IR2 stattgegeben (4). Dessen Interruptprogramm IP2 (5) ruft erneut das Unterprogramm UP auf (6), führt es aus (7, 8, 9) und gibt schließlich mit dem RTE-Befehl (10) die Programmsteuerung an die Unterbrechungsstelle im Unterprogramm zurück (11), wonach das Unterprogramm seine Arbeit für das erste Interruptprogramm be-

endet (12). Das Unterprogramm sorgt jeweils selbst für das Einrichten eines neuen Datenbereichs und macht unmittelbar vor dem Rücksprung (RTS) den ursprünglichen Datenbereich wieder zugänglich.

3.3.4 Modulare Programmierung

Ober- und Unterprogramme, aber auch größere Programmteile mit z. B. mehreren Unterprogrammen und den ihnen zugeordneten Daten können als eigenständige *Module* unabhängig voneinander geschrieben und übersetzt werden. Diese Technik des „Modularen Programmierens“ (modular programming), die vor allem auch von praxisrelevanten höheren Programmiersprachen unterstützt wird, gewährt eine größere Unabhängigkeit, bessere Übersichtlichkeit und geringere Fehleranfälligkeit bei der Erstellung großer Programmsysteme. Typische Erscheinungsform solcher Module ist z. B. die Zusammenfassung von höheren Datenstrukturen und ihnen zugehörigen Zugriffsoperationen (Klassenkonzept, Objektbildung).

Bei der Übersetzung eines Moduls kann der Assembler zwangsläufig nur die modulinternen Adressbezüge auflösen, d.h., er muß sich auf die Adressgenerierung für die internen Programmprünge und die Zugriffe auf die internen Daten beschränken. Die *Adressquerbezüge* zwischen den Modulen, wie z. B. das Aufrufen der einzelnen Modulfunktionen und der Zugriff auf Variablen in anderen Modulen, werden erst nach Vorliegen der einzelnen, übersetzten Module hergestellt. Hierzu muß der Assembler Information bereitstellen, die später beim Binden oder beim Laden der Module durch den *Binder* oder den *bindenden Lader* ausgewertet wird.

Um diese Adressquerbezüge herzustellen, benötigt man die in 3.1.2 beschriebenen Assembleranweisungen DEF und REF. In der DEF-Anweisung werden die in einem Modul definierten Adresssymbole aufgelistet, die von anderen Modulen als sog. externe Adressen verwendet werden dürfen. Umgekehrt werden in der REF-Anweisung die im Modul verwendeten Adresssymbole aufgelistet, die in anderen Modulen definiert sind. Der Assembler wertet diese Information aus und erstellt daraus Zusatzangaben zum Maschinencode eines jeden Moduls. Diese werden vom Binder oder vom bindenden Lader zur Auflösung der Adressquerbezüge benutzt. Um die externen Adressen als absolute Adressen ermitteln und in die Module einsetzen zu können, muß zu diesem Zeitpunkt die spätere Lage eines jeden Moduls im Speicher bekannt sein.

Beispiel 3.14. ► *Adressbezüge zwischen Modulen.* Ein Modul Counter soll, von einem Modul Main1 aufgerufen, seine lokale Zählvariable CTR um ein von Main1 vorgegebenes variables Inkrement INCR erhöhen. Weiter soll es, von einem Modul Main2 aufgerufen, den aktuellen Wert der Zählvariablen der lokalen Variablen X von Main2 zuweisen. – Bild 3-25 zeigt die Module Main1 und Main2, die über die externen Adressen COUNT1 und COUNT2 (REF-Anweisungen) das Modul Counter aufrufen und ihm ihre lokalen Variablen INCR und X (DEF-Anweisungen)

```

; Modul Main1           ; Modul Counter

    RORG   MAIN1          RORG   COUNTER
    DEF    INCR            DEF    COUNT1,COUNT2
    REF    COUNT1          REF    INCR,X
    INCR   DS.W  1          CTR    DS.W  1
    :
    JSR    COUNT1          COUNT1: ADD.W  INCR,CTR
    :
    END                COUNT2: MOVE.W  CTR,X
                           RTS
                           END

; Modul Main2           ; Modul Counter

    :
    RORG   MAIN2          RORG   COUNTER
    DEF    X               DEF    COUNT1,COUNT2
    REF    COUNT2          REF    INCR,X
    DS.W  1
    :
    JSR    COUNT2          COUNT1: ADD.W  INCR,CTR
    :
    END                COUNT2: MOVE.W  CTR,X
                           RTS
                           END

```

Bild 3-25. Programme zu Beispiel 3.14: Adreßbezüge zwischen Modulen

zur Verfügung stellen. Das Modul Counter mit den Eintrittsadressen COUNT1 und COUNT2 (DEF-Anweisung) greift für das Erhöhen bzw. das Zuweisen der Zählvariablen auf die externen Variablen INCR und X (REF-Anweisung) zu. ▶

Das obige Verfahren zur Auflösung der Adreßquerbezüge hat den Nachteil, daß beim Verändern eines Moduls oder bei dessen Verschieben im Speicher nicht nur das Binden, sondern auch das Übersetzen der auf ihn zugreifenden Module erneut erforderlich wird. Letzteres ist unabhängig davon, ob dieses Modul aufgrund befehlsszählerrelativer Adressierung dynamisch verschiebbar ist oder nicht. Änderungen in Modulen, die in Festwertspeichern (ROMs, EPROMs) stehen, sind dabei besonders aufwendig. Abhilfe schafft hier die Erweiterung der externen (absoluten) Adressierung durch indirekte und basisrelative Adressierung. Hierzu wird jedem Modul eine im Speicher stehende *Bindetabelle* (*link table*) zugeordnet, in die zum Ladezeitpunkt die von diesem Modul benutzten externen Adressen eingetragen werden. Im Programmteil des Moduls selbst sind die externen Adreßangaben durch basisrelative Bezüge auf diese Eintragungen ersetzt.

3.4 RISC-spezifische Programmierung

Für jegliche Datenverarbeitung, und das gilt gleichermaßen für CISCs wie für RISCs, wäre eigentlich ein einziger hinreichend großer und hinreichend schneller Datenspeicher die ideale Voraussetzung für eine effiziente Maschinenprogrammierung. Bei einem *CISC* wäre das der Hauptspeicher (der Registerspeicher entfiel) mit seiner unmittelbaren Adressierung durch die CISC-Befehle in jeweils mehreren Prozessortakten. Bei einem *RISC* wäre das der Registerspeicher (ohne

Hauptspeicher) mit seiner unmittelbaren Adressierung durch die RISC-Befehle mit jedem Prozessortakt. Große Speicher benötigen jedoch, und das gilt wieder gleichermaßen für CISCs wie für RISCs, breite Adressen und damit lange Befehle. Außerdem lassen sie sich nicht so gut zusammen mit der Prozessorstruktur auf dem Prozessorchip wie derzeit für sich allein, d.h. als ausschließliche Speicherstruktur auf Einzelchips integrieren. – Zur Ermöglichung einer effizienten Maschinenprogrammierung geht man deshalb bei beiden Konstruktionen im Grunde den gleichen Weg: Bei CISCs wird ein im Vergleich zum Hauptspeicher kleiner Registerspeicher, bei RISCs wird ein im Vergleich zum Registerspeicher großer Hauptspeicher hinzugefügt, so daß in beiden Fällen eine Hierarchie von Haupt- und Registerspeicher entsteht, wobei letzterer als Zwischenspeicher fungiert (abgesehen von einem weiteren Zwischenspeicher, dem Cache, siehe 6.2).

Für die Programmierung im Konzeptuellen sind die Konsequenzen die gleichen: In beiden Fällen werden in den Registern neben sehr wenigen Daten hauptsächlich Speicheradressen gehalten, über die (registerindirekt) die im Hauptspeicher vorliegenden Datenmengen adressiert werden. Hingegen gibt es für die Programmierung im Detail gravierende Unterschiede. Während bei CISCs die Daten vielfach ohne Lade-/Speichere-Befehle direkt im „Hauptspeicher“ verarbeitet werden (z.B. mit Speicher-/Speicher-Befehlen), müssen sie bei RISCs grundsätzlich mit Lade-Befehlen in die Register gelesen werden. Anschließend werden sie im „Registerspeicher“ verarbeitet (datenverarbeitende Befehle haben grundsätzlich Register-/Register-Format!), und schließlich werden sie wieder mit Speichere-Befehlen in den Hauptspeicher zurückgeschrieben.

Natürlich geschieht das beim CISC im Prinzip genauso (nämlich daß die Daten in den Prozessorregistern von der Prozessor-ALU verarbeitet werden), aber vom CISC-Konstrukteur *mikroprogrammiert* und nicht wie beim RISC vom RISC-Programmierer *maschinenprogrammiert*. Insofern merkt der CISC-Maschinenprogrammierer von den vielen „Mikroprozessen“, die der RISC-Programmierer sozusagen ausprogrammieren muß, nichts oder nur wenig. Der Unterschied zwischen CISC- und RISC-Programmierung liegt demnach hauptsächlich im Grad der Detaillierung der zu programmierenden Abläufe. – Von einer gemeinsamen Ausdrucksebene zur Programmierung einer Aufgabe aus gesehen, z.B. der Ebene der Assemblersprache als einer „niederen“ Programmiersprache, wird das Assemblerprogramm in der CISC-Umgebung *interpretiert*, und zwar ohne Zutun des CISC-Programmierers, während dasselbe Programm in der RISC-Umgebung erst in diese *compiliert* werden muß, und zwar nun vom RISC-Programmierer selbst (oder – ohne Zutun des Programmierers – von einem Compiler für den RISC).

Wie gesagt, die Unterschiede zwischen CISC- und RISC-Programmierung liegen im Detail: Der CISC unterstützt die Maschinenprogrammierung in bestimmter Weise, nämlich durch komplexe Befehle in Verbindung mit einer Vielzahl an Speicheradressierungsarten. Der RISC hingegen unterstützt die Maschinenpro-

grammierung auf andere Weise, und zwar hauptsächlich durch einen großen Registerspeicher, ggf. in Verbindung mit einem wirksamen Mechanismus zum Aufruf von Unterprogrammen (Windowing beim SPARC). Dadurch entstehen für CISCs und RISCs ganz unterschiedliche Maschinenprogramme, deren Effizienz im Grunde nur durch Vergleich ihrer realen Laufzeiten beurteilt werden kann. Auf der einen Seite blähen sich bei RISCs die Programme insbesondere wegen der Einfachheit der Adressierungsarten auf, auf der anderen Seite wird der für CISCs typische Überhang bei Unterprogrammaufrufen deutlich vermindert, insbesondere hinsichtlich des Parametertransports über den Stack. Vor allem bei geschachtelten Unterprogrammen (einfachen, rekursiven, reentranten) kann das einen bedeutenden Geschwindigkeitsvorteil bringen, jedoch nur so lange, wie die Schachtelungstiefe die Anzahl an Registerfenstern nicht überschreitet. Das dürfte bei einfacher Schachtelung immer zutreffen. Bei rekursiver Schachtelung kann es hingegen schnell zu einem „Überlauf“ kommen, der über einen Trap zum Auslagern von Registerspeicherinhalten aufgefangen werden muß.

Wir behandeln im folgenden drei wichtige, von CISCs abweichende Merkmale der RISC-Programmierung, und zwar am Beispiel des *SPARC-Prozessors*:

1. die Lade-/Speichere-Problematik, damit verbunden das Aufblähen der Maschinenprogramme,
2. den Aufruf von ungeschachtelten wie von einfachen geschachtelten Unterprogrammen sowie
3. die Programmierung des Anschlusses von Interruptprogrammen, in diesem Zusammenhang im Rahmen eines Beispiels die Wirkung einer Programmunterbrechung innerhalb eines Unterprogramms, d.h. die Technik des Aufrufs reentrant geschachtelter Unterprogramme. Dieser Aspekt ist insofern von Bedeutung, als die Interruptprogrammierung aufgrund ihrer Hardwarenahe nicht in einer höheren Programmiersprache wie z.B. C erfolgen kann, sondern in Assemblersprache durchgeführt werden muß.

Hinsichtlich der Schreibweise der Maschinenbefehle in den Programmbeispielen bedienen wir uns der vom Assembler bereitgestellten Pseudobefehle, aber nur, so weit diese genau einem Maschinenbefehl entsprechen. Sonst würde das Aufblähen der Maschinenprogramme hinter den Pseudobefehlen versteckt bleiben. Außerdem wäre besondere Vorsicht bei der Verwendung von Pseudobefehlen innerhalb der Delay-Slots aller möglicher Sprungbefehle geboten (das schließt auch Unterprogrammaufrufe und -rücksprünge ein).

3.4.1 Lade-/Speichere-Problematik

Bei CISCs ist es *deshalb* leicht, einen großen Hauptspeicher zu adressieren, weil für einen jeden Befehl so viele Befehlwörter oder Befehlsbytes (Mehrwort- bzw.

Mehrbytebefehle) bereitgestellt werden können, wie benötigt werden. Bei RISCs hingegen ist man wegen der RISC-typischen Befehlsverarbeitung im Prozessortakt darauf angewiesen, einen jeden Befehl genau in einem z.B. 32-Bit-Wort im Hauptspeicher unterzubringen (Fließbandverarbeitung!). Deshalb wird für die arithmetischen und logischen Befehle mit ihrem Dreiausdrucksbefehlsformat die unmittelbare Adressierung des Hauptspeichers mit den dann notwendigen drei langen Adressen gar nicht erst vorgesehen. Für die bedingten Sprungbefehle (auch die „unbedingten“, wie Unterprogrammaufruf) mit ihrem Einadresßformat bedient man sich hingegen der PC-relativen Adressierung, d.h. die Adreßangabe ist eine (kurze) Distanzangabe zum (langen) Befehlszählerstand.

Die *Lade- und Speichere-Befehle* schließlich benötigen im Befehlswort eine kurze Adresse für den Registerspeicher und eine lange Adresse für den Hauptspeicher. Bei den heute üblichen Speichergrößen reicht aber der neben Befehlscode und Registeradresse verbleibende „Rest“ im 32-Bit-Befehlswort nicht aus, um den Hauptspeicher in ganzer Größe direkt adressieren zu können. Man ist deshalb gezwungen, auch hier z.B. eine 32-Bit-Hauptspeicheradresse auf zwei Speicherworte zu verteilen. Dabei geht man aber einen anderen Weg als bei CISCs, bleibt gewissermaßen dem RISC-Prinzip treu: Man bringt diese 32-Bit-Adresse nicht in 1 Befehl à 2 Worte, sondern in 2 Befehlen à 1 Wort unter und lädt nun mit diesen *beiden* Befehlen die *beiden* Adreßteile in die *beiden* Teile (den oberen und den unteren Teil) eines Registers des Registerspeichers.

Die Setze-Adresse-Sequenz. Um gezielt den oberen Teil eines allgemeinen Registers laden zu können, sieht man einen Spezialbefehl vor: den Befehl sethi (set higher part). Dieser interpretiert seinen kurzen Adreßteil als Registerziel und seinen langen Adreßteil als Direktoperanden. In diesen Teil des Befehlsworts muß der Assembler den oberen Teil der Hauptspeicheradresse einfügen; wenn beispielsweise x die im Assemblerprogramm deklarierte Hauptspeicheradresse ist, so kann der obere Teil dieser Adresse durch den Assembleroperator %hi(x) separiert werden (extract higher part of x).

Um den unteren Teil des Registers ansprechen zu können, ohne seinen (komplementären) oberen Teil zu verändern, bedarf es eines zu sethi komplementären Befehls: des Befehls setlo (set lower part). setlo kann aber im Gegensatz zu sethi als Pseudobefehl verwirklicht werden, z.B. durch den Befehl or mit dem zu %hi(x) komplementären Operator %lo(x) (extract lower part of x). Das geht natürlich nur, sofern im or-Befehl ein genügend langer Direktoperand untergebracht werden kann. (Beim SPARC nimmt der sethi-Befehl aufgrund des verfügbaren Platzes im Befehlswort einen 22-Bit-Direktoperanden auf, so daß im or-Befehl nur Platz für eine 10-Bit-Konstante benötigt wird. Diese Konstante wird mit Zero-Extension verarbeitet.) Voraussetzung der Verwendung des or-Befehls ist außerdem, daß der sethi-Befehl bei seinem Ladevorgang die Bits des unteren Registerseils mit Nullen initialisiert.

Die typische Befehlssequenz zum Laden einer (32-Bit-)Adresse x in ein (32-Bit)-Register ri (Setze-Adresse-Sequenz) lautet demnach im Maschinencode ($i \neq 0$)

```
sethi %hi(x),ri
or    ri,%lo(x),ri
```

und in Kurzschreibweise für sethi bzw. im Pseudocode mit setlo (gleich or)

```
sethi x,ri
setlo x,ri
```

Die Lade- und die Speichere-Sequenz. Das Laden und Speichern wird mit den Befehlen ld und st (in vielerlei Varianten, siehe Befehlsbeschreibungen in 2.2.4, Tabelle 2-5, S. 115) durchgeführt, und zwar standardmäßig mit 32-Bit-Adressen registerindirekt mit Displacement oder registerindirekt mit Indizierung (siehe 2.2.3, insbesondere Bild 2-27, S. 113).

Die komplette Lade- und die komplette Speichere-Sequenz einschließlich Adresse-Setzen sehen dann im einfachsten Falle, d.h. Displacement = 0 (default) bzw. Indexregister = 0 (default), folgendermaßen aus ($i \neq 0, j \neq 0$):

```
sethi x,ri
setlo x,ri
ld    ri,0,rj

sethi x,ri
setlo x,ri
st    rj,ri,0
```

Beispiel 3.15. ► Iterative Schleife mit induktiver Begrenzung der Schleifendurchläufe. Zur Demonstration der Lade-/Speichere-Problematik mit ihrem Aufblähen des RISC-Programms gegenüber einem äquivalenten CISC-Programm wählen wir das in Beispiel 3.8 (S. 187) programmierte Problem des Durchsuchens eines Pufferbereichs BUFFER mit $N = 512$ Wörtern nach einem in der Speicherzelle KEY vorgegebenen Schlüsselwert. Bild 3-26 zeigt dazu noch einmal das dort angegebene CISC-Programm (Bild 3-15, S. 189) und stellt diesem das entsprechende RISC-Programm gegenüber. – Das CISC-Programm besteht aus 11 Programmzeilen (nicht identisch mit 11 Speicherworten); das in seiner Wirkung äquivalente RISC-Programm besteht demgegenüber aus 17 Programmzeilen (hier identisch mit 17 Speicherworten). – Beim Vergleich beider Programme fallen die Setze-Adresse-Sequenzen (für LEA) und die Lade- sowie die Speichere-Sequenz (für MOVE.W vor der Schleife bzw. für MOVE.W nach der Schleife) auf, während die Befehle innerhalb der Schleife – nur diese sind die „zeitrelevanten“ Befehle – sich nahezu 1:1 entsprechen.

Die folgenden Befehle aus Bild 3-26, nach ihrem ersten Auftreten im Programm geordnet, sind *Pseudobefehle*, wobei mit li die lokalen Register des aktuellen Registerfensters und mit g0 = 0 das erste Register des globalen Registerbereichs bezeichnet sind (siehe 2.2.1, insbesondere Bild 2-24, S. 107).

```
setlo (für or li,...,li)
inc   (für add li,...,li)
cmp   (für subcc li,lj,g0)
nop   (für z.B. or g0,g0,g0)
clr   (für z.B. or g0,g0,li)
```

CISC-Programm nach Bild 3-15RISC-Programm

N	EQU	512	n	equ	512
BUFFER	DS.W	N	buffer	ds.w	n
BEND	EQU	BUFFER+4*(N-1)	bend	equ	buffer+4*(n-1)
KEY	DS.W	1	key	ds.w	1
PTR	DS.W	1	ptr	ds.w	1
:					
	LEA	BUFFER, R0	sethi	buffer,10	
	LEA	BEND, R1	setlo	buffer,10	
	MOVE.W	KEY, R2	sethi	bend,11	
VERGL:	CMP.W	R2, (R0)	setlo	bend,11	
	BEQ	FOUND	sethi	key,12	
	CMP.W	R1, R0	setlo	key,12	
	BEQ	NOTFD	ld	12,0,12	
	ADD.W	#4,R0	vergl:	ld	10,0,13
	BRA	VERGL		cmp	12,13
NOTFD:	MOVE.W	#0,R0		beq	found
FOUND:	MOVE.W	R0, PTR		cmp	11,10
	:			bne.a	vergl
				inc	4,10
			notfd:	clr	10
			found:	sethi	ptr,13
				setlo	ptr,13
				st	10,13,0
				:	

Bild 3-26. Programme zu Beispiel 3.15: *Iterative Schleife mit induktiver Begrenzung der Schleifendurchläufe* ◀

3.4.2 Unterprogrammanschluß

Der Anschluß von Unterprogrammen erfolgt bei CISCs und bei RISCs im Konzeptuellen gleich. In beiden Fällen gibt es Befehle für den Aufruf (BSR bzw. call) und die Rückkehr (RTS bzw. jmpl). Diese Befehle unterscheiden sich (neben der ausgeprägten Fließbandverarbeitung bei RISCs) jedoch darin, daß bei CISCs die *Rücksprungadresse* beim Aufruf auf den Stack gebracht und von dort bei der Rückkehr wieder geholt wird, bei RISCs hingegen die Rücksprungadresse in einem der allgemeinen oder in einem speziellen Prozessorregister (*Link-Register*) verwaltet wird. Betrachtet man als RISC-Prozessor den SPARC, so gibt es einen weiteren Unterschied durch dessen Fenstertechnik. Hier werden zur Fensterumschaltung, wie in 2.2.1 beschrieben, zwei weitere Befehle benötigt, nämlich save (save registers) und restore (restore registers).

Ungeschachtelte Unterprogramme. Bei Unterprogrammen, in deren Verlauf keine weiteren Unterprogramme aufgerufen werden, genügt es, die Parameter – wie in 3.3.2 für CISCs beschrieben – z.B. in den lokalen Bereich des Register-Speichers zu bringen und mit call (call to subroutine) zum Unterprogramm zu

springen. (Achtung: Der auf call folgende Befehl wird vor dem Wegsprung mit ausgeführt – Delay-Slot von call.) Im Unterprogramm wird mit den Parametern im Registerspeicher gearbeitet und am Ende mit jmpl (jump and link) aus dem Unterprogramm zurückgesprungen. (Achtung: Nach jmpl muß noch ein Befehl stehen, der vor dem Rücksprung ausgeführt wird – Delay-Slot von jmpl.) Bei der Programmierung des Unterprogramms und seines Aufrufs ist zu beachten, daß der PC-Stand mit call im Register o7 abgelegt wird und von dort mit jmpl wieder in den nPC zurückgeladen wird, jedoch mit 8 beaufschlagt, da der call-Befehl und der Befehl in seinem Delay-Slot bereits ausgeführt sind (siehe dazu 2.2.4, insbesondere Tabelle 2-6, S. 117). Ferner ist zu beachten, daß die Link-Funktion des Befehls jmpl beim Rücksprung nicht benötigt wird, weshalb als Link-Register das Register g0 anzugeben ist.

Beispiel 3.16. ► Parameter im lokalen Bereich des Registerfensters. In Analogie zu Beispiel 3.10, Bild 3-19 (S. 195), zeigt Bild 3-27 die Formulierung der in Beispiel 3.15 beschriebenen Aufgabe als (ungeschachteltes) Unterprogramm. Die darin eingetragenen Pfeile zeigen den Ort der jeweiligen Sprungausführung, der wegen der Delay-Slots – wie bekannt – um eine Position gegenüber dem Ort des Auftretens des jeweiligen Sprungbefehls verschoben ist. – Wie zu sehen, ähnelt bei dieser Art des Aufrufs die Befehlsfolge in Bild 3-27 (mit Unterprogrammtechnik) stark der Befehlsfolge in Bild 3-26 (ohne Unterprogrammtechnik).

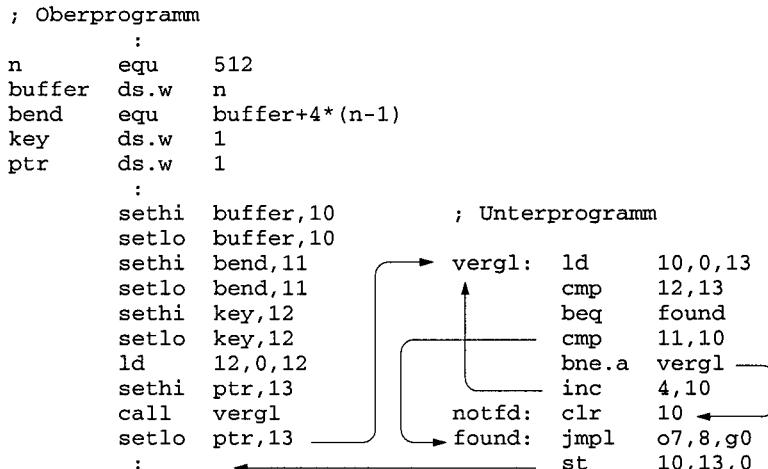


Bild 3-27. Programme zu Beispiel 3.16: Parameter im lokalen Bereich des Registerfensters

Geschachtelte Unterprogramme. Bei Unterprogrammen, in deren Verlauf weitere Unterprogramme aufgerufen werden, andere (einfach geschachtelt) oder dieselben (rekursiv oder reentrant geschachtelt), muß für jeden Aufruf ein aufrufspezifischer Datenbereich zur Verfügung gestellt werden. Dazu bedient man sich zweckmäßigerweise des *Stackprinzips*.

Bei CISCs wird als Stack ein Bereich des Hauptspeichers benutzt und dieser üblicherweise mit einem Stackpointer *halbwort-* oder *wortweise* angesprochen (Software-Realisierung!), siehe Beispiel 3.11 (S. 195). Bei einem RISC wie dem SPARC hingegen ist der Registerspeicher bereits als Stack ausgebildet (Hardware-Realisierung!). Dieser wird aber jetzt *bereichsweise* angesprochen und auf- und abgebaut, und zwar sehr wirkungsvoll mit sich gegenseitig überlappenden Stackbereichen. Wie in 2.2.1 beschrieben, wird diese Technik als *Windowing* bezeichnet. Die Rolle des Stackpointers übernimmt nun der Current-Window-Pointer, CWP, der den aktuellen Stackbereich in der Form einer Basisadresse enthält. Relativ dazu werden die einzelnen Register des Stackbereichs angegeben: mit o0 bis o7, 10 bis i7 und i0 bis i7 in aufsteigender Reihenfolge. Hierbei überlappen sich nun aufeinanderfolgende Registerfenster bei aufeinanderfolgenden Unterprogrammaufrufen in der Weise, daß die Out-Register des Oberprogramms identisch mit den In-Registern des Unterprogramms sind (vgl. Bild 2-24, S. 107). Das hat den großen Vorteil, daß – natürlich vorausgesetzt, die Kapazität eines Fensters und die Anzahl der Fenster reichen aus – im Gegensatz zu CISCs weder Registerinhalte noch Parameter in den Hauptspeicher und zurück in den Registerspeicher transportiert zu werden brauchen.

Bei einem RISC mit Windowing werden demgegenüber im Oberprogramm die *Parameter* in die Out-Register des aktuellen Fensters gebracht und anschließend mit dem Befehl save das nachfolgende Fenster aktiviert, so daß die Parameter unmittelbar über die In-Register des nun aktuellen Fensters im Unterprogramm angesprochen werden können. – Am Ende des Unterprogramms, d.h. unmittelbar vor der Ausführung des Unterprogrammrücksprungs ist mit dem Befehl restore der Stackbereich des Unterprogramms freizugeben und das vorhergehende Fenster wieder zu aktivieren. Dadurch steht dem Oberprogramm automatisch wieder der Datenbereich im nun wieder aktuellen Fenster des Oberprogramms zur Verfügung. Auch dabei können Parameter übergeben werden, nämlich Ausgangsparameter vom In-Bereich des Unterprogramms in den Out-Bereich des Oberprogramms.

Bei der Programmierung des *Unterprogrammanschlusses* ist zu beachten, daß der restore-Befehl standardmäßig im Delay-Slot des jmpl-Befehls steht. Dadurch befindet sich der jmpl-Befehl *vor* dem restore-Befehl im Fließband, d.h., er wird *vor* der Fensterumschaltung bearbeitet, obwohl erst *nach* der Ausführung des restore-Befehls zurückgesprungen wird. Deshalb darf im jmpl-Befehl für das Laden des nPC nicht wie vorher beim ungeschachtelten Unterprogrammaufruf o7 als Quellregister stehen, sondern es muß hier beim geschachtelten Unterprogrammaufruf im jmpl-Befehl i7 angegeben werden. Da diese eben geschilderte Art des Unterprogrammaufrufes, d.h. der Unterprogrammaufruf mit Windowing, die Standardtechnik im Anschluß von Unterprogrammen ist, wird die zuletzt genannte Variante des jmpl-Befehls (mit i7 zur Adressierung der geretteten Rückprungadresse) gewöhnlich vom Assembler als Pseudobefehl mit dem Code ret zur Verfügung gestellt.

Beispiel 3.17. ► *Parameter im Out-/In-Bereich benachbarter Registerfenster.* In Analogie zu Beispiel 3.11, Bild 3-20 (S. 196), zeigt Bild 3-28 die Programmierung der in Beispiel 3-15 beschriebenen Aufgabe als (schachtelbares) Unterprogramm. Neben dem Programm ist die Stackbelegung im Registerspeicher zusammen mit den im Programm korrespondierenden Positionsnummern der bis dahin jeweils ausgeführten Befehle dargestellt. Wie man sieht, erfolgt der Aufbau und der

```
; Hauptprogramm

n      equ     512
buffer ds.w   n
bend   equ     buffer+4*(N-1)
key    ds.w   1
ptr    ds.w   1
      :
      sethi  buffer,o0
1.    setlo  buffer,o0
      sethi  bend,o1
2.    setlo  bend,o1
      sethi  key,o2
3.    setlo  key,o2
      ld     o2,0,o2
      sethi  ptr,o3
4.    call   vergl
5.    setlo  ptr,o3
      :

; Unterprogramm

vergl: save
loop:  ld     i0,0,10
      cmp   i2,10
      beq   found
      cmp   i1,i0
      bne.a loop
      inc   4,i0
notfd: clr   i0
found: st    i0,i3,0
      jmp   i7,8,g0
      restore
```

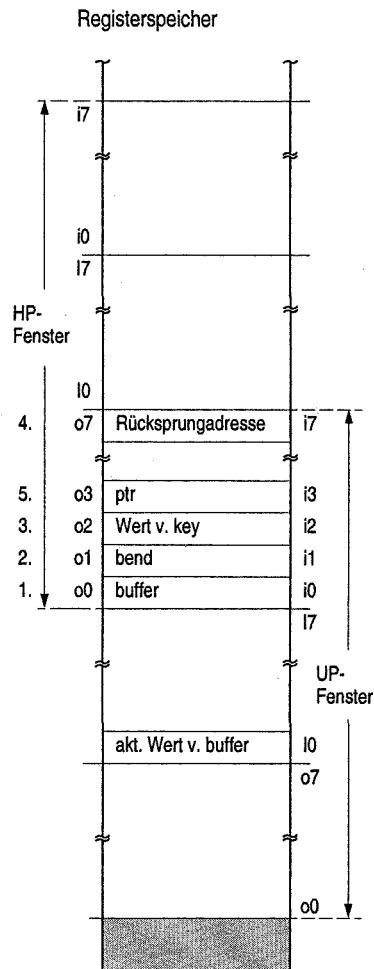


Bild 3-28. Programme und Register-Stackbelegung zu Beispiel 3.17: *Parameter im Out-/In-Bereich benachbarter Registerfenster*

Abbau des Stacks unter Beachtung der Regeln des Windowing im Grunde vollautomatisch. Das so programmierte Unterprogramm ist nun im Gegensatz zum Unterprogramm in Beispiel 3.16 reentrant, d.h., es kann z.B. von einem Interruptprogramm an beliebiger Stelle unterbrochen werden und erneut in diesem Interruptprogramm aufgerufen werden. Die Schachtelung des Unterprogramms erfolgt dann indirekt mit sich selbst. ▲

3.4.3 Programmunterbrechungen

Wie im vorangehenden Abschnitt über Unterprogramme unterscheiden sich CISCs und RISCs hinsichtlich *Programmunterbrechungen* nicht im Konzeptuellen, wohl aber im Detail. Das hat zum einen mit dem RISC-eigenen Fehlen komplexer Abläufe, wie sie ja gerade bei Programmunterbrechungen auftreten, zu tun. Hier ist dementsprechend manches zu programmieren, was bei CISCs automatisch abläuft (z. B. das Retten des Statusregisters, das Setzen der Interruptmaske im Statusregister). Zum andern ist es, wenn man den *SPARC* betrachtet, die Ausbildung des Registerspeichers als Stack zur Stapelung und damit verzögerten Weiterausführung unterbrochener Programme, die für unterschiedliche Programmierung der Unterbrechungsroutinen sorgt. Insofern ist die Unterbrechung eines (Ober)programms durch ein (Unter)programm durchaus mit dem Aufruf eines (Unter)programms von einem (Ober)programm aus vergleichbar, wobei die geklammerten Vorsilben in der Unterprogrammtechnik auf eine hierarchische Unterordnung anspielen, hier aber bei Programmunterbrechungen gerade bezüglich der Dringlichkeit der auszuführenden Arbeiten gegenteilig zu verstehen sind.

Die Interrupt-Eingangssequenz. Wie in 2.2.5 beschrieben, führt eine Programmunterbrechung, Trap wie Interrupt, immer in den Supervisor-Modus, und es sind – siehe 2.2.5, auch Bild 2-26 (S. 109) – folgende Schritte erforderlich:

1. Blockieren des Unterbrechungssystems ($0 \rightarrow ET$),
2. Retten des Status der alten Betriebsart ($S \rightarrow PS$),
3. Aktivieren des neuen Fensters ($CWP - 1 \rightarrow CWP$),
4. Retten der Befehlszählerstände ($PC \rightarrow l1, nPC \rightarrow l2$),
5. Umschalten in den Supervisor-Modus ($1 \rightarrow S$),
6. Sprung in das Unterbrechungsprogramm (Unterbrechungsvektor $\rightarrow nPC$).

Bei Interrupts tritt diese Abfolge, d.h. der *Interruptzyklus*, nur dann in Aktion, wenn dem Interrupt entsprechend seiner Ebene stattgegeben wird. Dazu wird die Ebenennummer des ankommenden Interrupts mit der Ebenennummer im Statusregister SR, d.h. der *Interruptmaske* (hier dem Processor-Interrupt-Level PIL, Bild 2-26) verglichen.

Die Interruptebenen sind gemäß steigender Priorität in steigender Reihenfolge numeriert; die Ebene Normalverarbeitung als die Ebene niedrigster Priorität hat die Nummer 0. Wenn die Ebenennummer des ankommenden Interrupts kleiner oder gleich der Interruptmaske PIL ist, so wird dem Interrupt nicht stattgegeben. Ist die Ebenennummer hingegen größer, so wird dem Interrupt in der beschriebenen Weise stattgegeben.

Bild 3-29 zeigt in seinen Teilen b und c die Wirkung einer Unterbrechung, soweit es die Unterbrechungshardware, d.h. die Schritte 1 bis 6 betrifft, wenn vorher

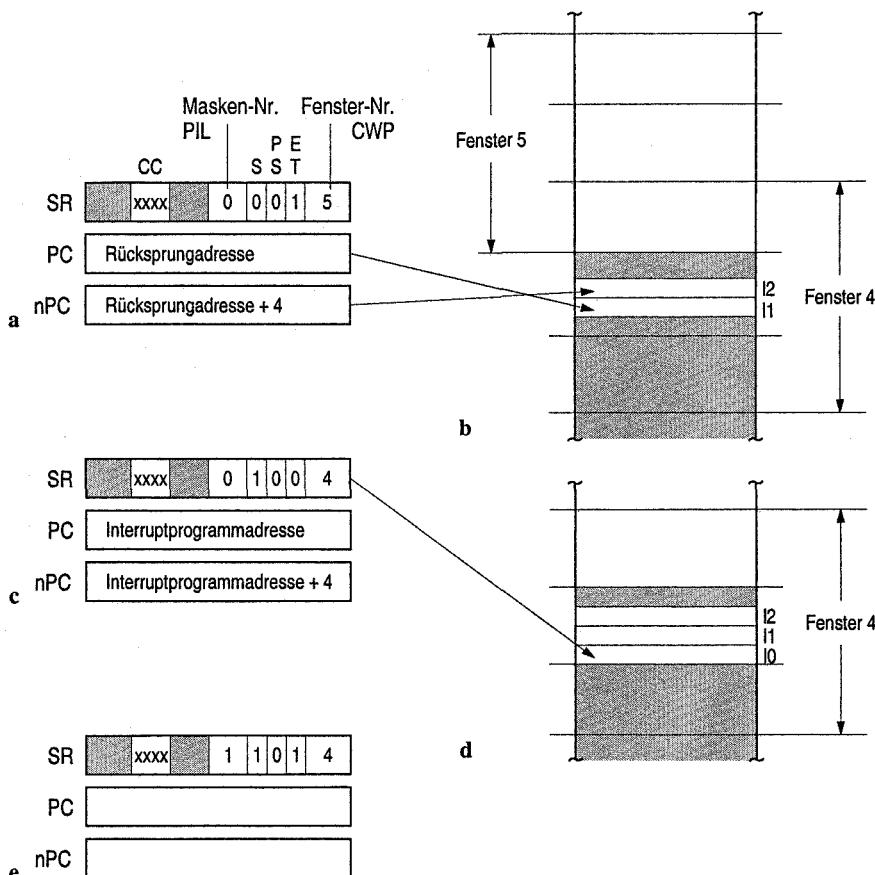


Bild 3-29. Prozessorstatus im Verlaufe eines Interrupts auf der Ebene 1, a vor dem Eintreffen des Interrupts, b und c nach dem Interruptzyklus, d und e nach der Interrupt-Eingangssequenz

(Teilbild a) Normalverarbeitung im Fenster 5 angenommen wird. Den „Rest“ muß die Unterbrechungssoftware erledigen. Dazu müssen – vorausgesetzt, die Interruptebene läßt weitere Interrupts (höherer Priorität) zu – am Anfang des Interruptprogramms die folgenden Schritte ausgeführt werden:

- i. Retten des Statusregisters auf den Stack, d.h. hier Schreiben von SR in den Registerspeicher,
- ii. Setzen der Maske PIL im Statusregister auf den Wert der Interruptebene,
- iii. Freigeben des Interruptsystems durch Setzen des Enable-Trap-Bits ET im Statusregister.

Diese Schritte sind jedoch nur dann ausreichend, wenn der Ausbau des RISC eine genügend große Zahl an Fenstern aufweist. Bei m Unterbrechungsebenen und

einer Unterprogrammschachtelungstiefe von n sowohl für das Normalprogramm als auch für die Interruptprogramme wären das $m \times n$ Fenster. Andernfalls, d.h. insbesondere dann, wenn die Schachtelungstiefe in die Nähe der Fensteranzahl kommt, muß bei jedem Interrupt zusätzlich zu den beschriebenen Schritten praktisch der gesamte Registerspeicherinhalt in den Hauptspeicher gerettet und am Ende des Interruptprogramms wieder zurückgeladen werden. Dabei muß mindestens ein Fenster, hier das Fenster 0, für das Interruptprogramm (bzw. als Trap-Fenster) frei gehalten werden, d.h., die Schachtelungstiefe darf nicht größer als die Fensteranzahl minus 1 sein.

Bild 3-29 zeigt in seinen Teilen d und e die Wirkung der Unterbrechung, soweit es die Software bezüglich der Schritte i bis iii betrifft, wenn für den auslösenden Interrupt die Ebene 1 angenommen wird. Der dazugehörige, den Schritten i bis iii entsprechende Softwareteil der Interrupt-Eingangssequenz lautet mit Verwendung von rd für das Retten des Statusregisters und mit Benutzung der Befehle rd (read) und wr (write) für den Zugriff auf die speziellen Register des Prozessors:

```

rd    sr,10          ; alten Status retten
move 10,13
and   13,B1000011011111,13 ; CC, S, PS, CWP beibehalten
or    13,B0000100100000,13 ; PIL und ET ändern
wr    13,0,sr         ; neuen Status laden
:

```

Dabei ist zu beachten, daß der mit wr in das Statusregister zurückgeschriebene Wert als Exklusiv-oder-Verknüpfung der beiden ersten Operandenangaben von wr gebildet wird, hier durch die Null als zweiten Operanden mit der Wirkung eines reinen Transportbefehls zwischen l3 und sr (siehe auch Tabelle 2-5, S. 115).

Die Interrupt-Ausgangssequenz. Nach der Ausführung des Interruptprogramms muß der Prozessorstatus, so wie er vor dem Interrupt bestand, wiederhergestellt werden. Dazu müssen die zum eigentlichen Interruptprogramm führenden Aktionen gewissermaßen rückwärts durchlaufen werden: zuerst der Softwareteil der Interrupt-Eingangssequenz, dann ihr Hardwareteil, der Interruptzyklus, dieser nun allerdings auch „programmiert“, und zwar mit dem in 2.2.4 beschriebenen Spezialbefehl rett (return from trap).

In der Veranschaulichung von Bild 3-29 ist dementsprechend von den Teilbildern d und e auszugehen. Um zu Teilbild c zu gelangen, muß aus Fenster 4 das Statusregister geladen werden. Um weiter zu den Teilbildern a und b zu gelangen, müssen das Statusregister rückgeändert werden (damit verbunden: die Fensterumschaltung) und die Befehlszähler rückgeladen werden (damit verbunden: der Rücksprung).

Im einzelnen sind also am Ende des Interruptprogramms folgende Schritte auszuführen:

- i. Zurückladen des Statusregisters vom Stack, d.h. hier Lesen aus dem Registerspeicher nach SR; damit ist das Interruptsystem wieder blockiert und die frühere Maskierung wiederhergestellt.
- ii. Aktivieren des alten Fensters ($CWP + 1 \rightarrow CWP$), zugleich
Rückladen der alten Betriebsart ($PS \rightarrow S$), zugleich
Freigabe des Unterbrechungssystems ($1 \rightarrow ET$), zugleich
Rücksprung in das unterbrochene Programm, und zwar so, daß der unterbrochene Befehl noch einmal abgerufen wird ($I1 \rightarrow nPC$, einen Takt später $nPC \rightarrow PC$, $I2 \rightarrow nPC$).

Der Schritt i der Interrupt-Ausgangssequenz entspricht dem Rückgängigmachen der Schritte i bis iii der Interrupt-Eingangssequenz. Der Schritt ii entspricht dem Rückgängigmachen der Schritte 1 bis 4 des Interruptzyklus. Die Interrupt-Ausgangssequenz – auf Bild 3-29 abgestimmt – lautet dementsprechend (zur Wirkung der Befehlskombination `jmpl/rett` siehe 2.2.4):

```
:
write 10,0,sr    ; alten Status laden
nop
nop
nop
jmp1 11,g0,g0  ; Rücksprung so, daß unterbrochener Befehl
rett   12,g0      ; erneut ausgeführt wird
```

Ein wichtiger Aspekt des Statusladens ist, daß dabei auch die bei der Unterbrechung vorhandenen Bedingungsbits CC wiederhergestellt werden. Wäre das nicht gewährleistet, so würde, wenn die Unterbrechung einen bedingten Sprungbefehl beträfe, der Sprungbefehl nach seinem erneutem Abruf auf die aus dem Unterbrechungsprogramm gelieferten „willkürlichen“ Bedingungsbits reagieren.

Zusammenfassung. Es ist geradezu RISC-typisch, wie für die Unterbrechungsbehandlung von der Prozessorhardware nur das Allernötigste bereitgestellt wird und dementsprechend von der Software wichtige Funktionen mitgestaltet werden müssen. Das mag den Vorteil haben, flexibel reagieren zu können, d.h., das Interruptsystem je nach Ausbau des Registerspeichers auszulegen: mal Interruptverarbeitung direkt im Registerspeicher, mal Interruptverarbeitung mit Auslagerung des Registerspeichers. Es hat aber sicher den Nachteil der im Zusammenhang mit der Fließbandverarbeitung sehr komplizierten Maschinenprogrammierung.

Bei der *Interrupt-Eingangssequenz* sind es die Punkte 1 bis 5, die nicht von der Software, d.h. vom Interruptprogramm übernommen werden können: Punkt 1 ($0 \rightarrow ET$) garantiert, daß der nächste nach der Unterbrechung ausgeführte Befehl auch der erste Befehl des Interruptprogramms ist; Punkt 2 ($S \rightarrow PS$) rettet ein Minimum an Statusinformation, bevor diese (Punkt 5: $1 \rightarrow S$) überschrieben wird; Punkt 3 ($CWP - 1 \rightarrow CWP$) schafft freien Speicherplatz für die Befehlszählervärstände, und Punkt 4 ($PC, nPC \rightarrow Stack$) rettet die Befehlszählervärstände,

bevor diese (Punkt 6: Unterbrechungsvektor → nPC) mit der Startadresse geladen bzw. überschrieben werden können.

Bei der *Interrupt-Ausgangssequenz* ist es der Punkt ii, der als Spezialbefehlrett zwar von der Software an die richtige Stelle plaziert, aber im Grunde als von der Hardware ausgeführt betrachtet werden muß. Die einzelnen Aktionen bedingen nämlich einander: Es muß das Aktivieren des alten Fensters ($CWP + 1 \rightarrow CWP$) „zuletzt“ geschehen, um im noch aktuellen Fenster Information nicht freizugeben; es muß das Rückladen des alten Statusbits ($PS \rightarrow S$) „zuletzt“ erfolgen, da erst nach dem Rücksprung der alte Status wieder gültig ist; es muß die Freigabe des Unterbrechungssystems ($1 \rightarrow ET$) „zuletzt“ geschehen, da die Sequenz erst nach dem Rücksprung erneut unterbrochen werden darf; und es muß schließlich der Rücksprung „zuletzt“ ausgeführt werden, denn erst damit ist das Interruptprogramm beendet, so daß im unterbrochenen Programm fortgefahrene werden kann. Mit anderen Worten: alle diese Aktionen müssen gleichzeitig geschehen.

Anmerkung. Auch beim Rücksprung ist es im Detail wieder RISC-typisch, daß für den Rücksprung aufgrund der Fließbandverarbeitung anstatt eines Befehls, wie bei CISCs, zwei Befehle erforderlich sind. Der erste dieser Befehle, `jmpl`, liest zwar die Rücksprungadresse vom Stack, der dazugehörige Befehl kann jedoch erst als übernächster Befehl ausgeführt werden, da zum Zeitpunkt des Lesens der Rücksprungadresse bereits der `rett`-Befehl abgerufen wird, der im Delay-Slot von `jmpl` steht. `rett` bewirkt die Statusumschaltung, d.h. das eigentliche Verlassen des Interruptprogramms.

Beispiel 3.18. ► Geschachtelter Aufruf eines reentranten Unterprogramms. Um das Auf und Ab der Fenstertechnik im Registerspeicher bei einem Interrupt innerhalb eines Unterprogramms zu demonstrieren, gekoppelt mit dem Aufruf desselben Unterprogramms innerhalb des Interruptprogramms, setzen wir einen Ausbau von 8 Fenstern sowie neben der Ebene 0 für die Normalverarbeitung nur noch die Ebene 1 als einzige weitere Ebene für die Ausnahmeverarbeitung voraus. Das Normalprogramm läuft als Hauptprogramm auf Ebene 0 und benutzt Fenster 7. Es ruft das Unterprogramm `vergl` in der reentranten Version Bild 3-29 (S. 213) auf, das das Fenster 6 benutzt. Mitten in der Ausführung dieses Unterprogramms wird es durch einen Interrupt unterbrochen.

Das Interruptprogramm läuft als Hauptprogramm auf der Ebene 1 und benutzt entsprechend der Interrupt-Eingangssequenz Fenster 5. Es ruft nun seinerseits in seinem Verlauf das Unterprogramm `vergl` auf, obwohl dieses auf Ebene 0 noch nicht abgeschlossen ist, und belegt Fenster 4 (indirekte Schachtelung „in sich selbst“). – In Bild 3-30 ist die Belegung des Registerspeichers in dieser Situation dargestellt. Man sieht, wie bei der Programmunterbrechung genau wie beim Unterprogrammaufruf eine Überlappung der Registerfenster entsteht. Des Weiteren ist das doppelte Zurverfügungstellen von Speicherplatz für die beiden Aufrufe des Unterprogramms gut zu erkennen.

Während beim Unterprogrammaufruf alles, ohne bei der Programmierung aufpassen zu müssen, richtig abläuft, ist bei der Programmunterbrechung zu beachten, daß der In-Bereich des neuen Fensters nicht vom Interruptprogramm genutzt werden darf (das Interruptprogramm hat schließlich keine Eingangsparameter). (Im Prinzip darf ein Unterbrechungsprogramm auch nicht den Out-Bereich seines Fensters zur Speicherung von Information benutzen; es könnte sich ja bereits im Fenster 0 befinden und würde dann evtl. in den In-Bereich des bereits benutzten Fensters 7 schreiben.) – Beim Verlassen des Unterprogramms sowie bei der Beendigung des Interruptprogramms wird der Stapel im Registerspeicher schrittweise abgebaut, so daß zuletzt wieder nur Fenster 7 „aktiv“ ist.

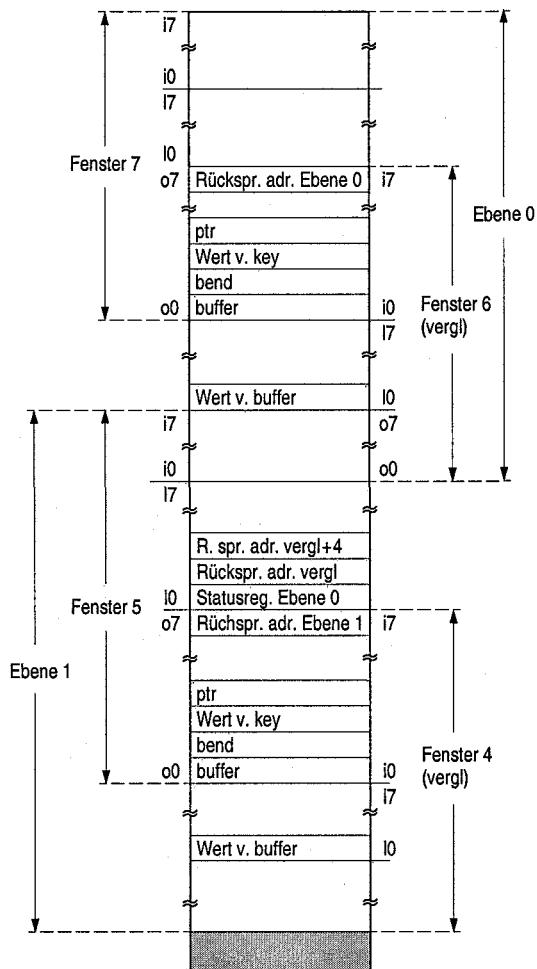


Bild 3-30. Belegung des Registerspeichers als Ergebnis folgender Aktionen: (1.) Aufruf des Unterprogramms vergl innerhalb eines Hauptprogramms (Normalprogramm) auf der Ebene 0, (2.) Unterbrechung von vergl durch ein Interruptprogramm (Ausnahmeprogramm) auf der Ebene 1, (3.) erneuter Aufruf des Unterprogramms vergl innerhalb des Interruptprogramms ◀

4 Maschinennahe Programmierung in C

(M. Menge)

Ein herkömmliches Mikroprozessorsystem, oder allgemeiner eine Rechenmaschine (kurz Maschine genannt), verarbeitet ein Programm, indem es die Befehle als Bitmuster nacheinander liest, decodiert und ausführt. Ein Programm, in dem die Befehle in dieser Weise codiert sind, wird als *Maschinenprogramm* bezeichnet. Dabei wird nicht zwischen Maschinenprogrammen unterschieden, in denen die Befehle binär, und solchen, in denen die Befehle in einer für den Menschen leichter handhabbaren Form, z.B. als hexadezimale Zahlen codiert sind. Letztere müssen jedoch einen Umsetzungsprozeß durchlaufen, in dem die Zahlen in die durch die Maschine interpretierbare binäre Form überführt werden.

Maschinenprogramme sind schlecht lesbar und besitzen eine geringe Änderungsfreundlichkeit. Das Einfügen eines Befehls kann z.B. zur Folge haben, daß die Zieladressen einiger Sprungbefehle ebenfalls geändert werden müssen. Dies ist einer der Gründe, weshalb die Maschinenprogrammierung für die Softwareentwicklung keine Bedeutung hat. Trotzdem wird sie in sehr einfachen Systemen verwendet, wobei die ggf. erforderliche Umsetzung meist Funktionalität eines sog. *Monitors* ist, also eines elementaren Betriebsprogramms, das einfache Kommandos, wie das Laden, Ausführen und Ändern von Benutzerprogrammen ermöglicht. Solche Monitore dienen z.B. zur Fehlersuche oder zur Wartung kleiner Steuerungssysteme, die mit einem Mikroprozessor oder Mikrocontroller ausgerüstet sind.

Die Nachteile der Maschinenprogrammierung, also die schlechte Lesbarkeit und die geringe Änderungsfreundlichkeit, treten bei der *Assemblerprogrammierung* nicht mehr auf. Die Befehle werden dabei nicht als Zahlen, sondern als sog. *Mnemone* dargestellt, also als für den Menschen verständliche Abkürzungen. Die ggf. erforderlichen Adressen können normalerweise sowohl numerisch als auch symbolisch angegeben werden, wobei die symbolischen Adressen vom Assembler mit den korrekten Zahlenwerten ausgefüllt werden. Dies ist z.B. bei der nachträglichen Änderung eines Assemblerprogramms und der dadurch verursachten Änderung vieler darin enthaltener Adressbezüge von Bedeutung. Zur Erzeugung eines ausführbaren Programms wird das i.allg. als Textdatei vorliegende und als *Quelltext* bezeichnete Programm durch einen Assembler übersetzt, wobei eine solche Übersetzung mit jeder Änderung des Assemblerprogramms erneut durchgeführt werden muß (siehe auch 1.3.2 und 3.1.2).

Die *Mächtigkeit* einer Programmiersprache, d.h. die Fähigkeit, komplexe Probleme einfach lösen zu können, ist auf Assemblerbasis sehr gering. Bereits einfachste Aufgaben müssen oft als Folge vieler Assemblerbefehle geschrieben werden, was eine erhöhte Fehlergefahr mit sich bringt. Hinzu kommt, daß Assemblerprogramme eine geringe sog. *Portabilität* besitzen, d.h., es ist meist nicht oder nur mit großem Aufwand möglich, ein für eine bestimmte Maschine geschriebenes Assemblerprogramm auf eine andere Maschine zu übertragen. Diese Probleme werden durch sog. Hochsprachen meist im Verbund mit Betriebssystemen gelöst.

Die Mächtigkeit dieser *problemorientiert* und *imperativ* genannten Programmiersprachen beruht im wesentlichen darauf, daß häufig wiederkehrende Aufgaben nun viel einfacher unter Verwendung eines bestimmten Regelwerks formuliert werden können. Um z.B. einen mathematischen Ausdruck durch ein Assemblerprogramm zu lösen, müssen die einzelnen Befehle explizit so angeordnet werden, daß die hierbei zu berücksichtigenden Vorrangregeln beachtet werden. Bei Verwendung einer Hochsprache sind diese Vorrangregeln meist dem Übersetzer bekannt, und der Ausdruck kann daher unmittelbar hingeschrieben werden. Dabei geht natürlich ein Teil der Flexibilität verloren, die die Assemblerprogrammierung bietet. So kann in einem Assemblerprogramm explizit auf Teilergebnisse, die sich während einer Berechnung ergeben haben, wiederholt zugegriffen werden, was in einer Hochsprache nicht möglich ist. (Zwar wird die Optimierung hinsichtlich gemeinsamer Teilausdrücke von vielen Übersetzern durchgeführt, einen Einfluß auf deren Ergebnisse hat der Benutzer jedoch nicht [Aho, Sethi, Ullman 1999].)

Thema dieses Kapitels ist die maschinennahe Programmierung in C. Dabei wird das heute sehr verbreitete ANSI C verwendet, wobei eine gewisse Vertrautheit mit dieser Hochsprache vorausgesetzt wird. Zum Verständnis der meisten Erklärungen dürfte aber ein oberflächliches Wissen in dieser Sprache ausreichen, da die einzelnen Programme detailliert kommentiert und die darin verwendeten Schlüsselwörter zur besseren Lesbarkeit in Fettschrift gesetzt sind (der Optik halber wurden Schlüsselworte im fließenden Text kursiv gesetzt). Wer sich genauer mit der Thematik der Programmierung in ANSI C auseinandersetzen möchte, dem sei die Lektüre eines der zahlreichen Bücher empfohlen. Ein Standardwerk zu diesem Thema ist das Buch von Kernighan und Ritchie „Programmieren in C“ [Kernighan, Ritchie 1990]. – Als Einstieg wird in Abschnitt 4.1 zunächst beschrieben, wie Betriebssystem und Übersetzer von einer Maschine abstrahieren. Im Anschluß daran folgen drei Abschnitte, in denen unterschiedliche Ansätze zur maschinennahen Programmierung verfolgt werden: in Abschnitt 4.2 die Programmierung mit speziell erweiterten Übersetzern, in Abschnitt 4.3 die Programmierung mit herkömmlichen Übersetzern, wobei die maschinennahen Aufgaben in Assembler programmiert werden, und in Abschnitt 4.4 schließlich die maschinennahe Programmierung unter Verwendung eines Betriebssystems.

4.1 Abstraktion von der Maschine

Die *Portabilität* der Hochsprachen ermöglicht es, ein und dasselbe Hochsprachenprogramm unverändert für unterschiedliche Maschinen zu übersetzen. Dies funktioniert natürlich nur unter der Voraussetzung, daß keine maschinenabhängigen Befehle und Betriebsmittel verwendet werden, daß also von der Maschine abstrahiert wird. Eine solche Abstraktion geschieht gewöhnlich in zwei Ebenen: (1.) durch das Betriebssystem, das von Geräten, Speichern usw. abstrahiert und (2.) darauf aufbauend durch den Übersetzer, der unmittelbar vom Prozessor und über das Betriebssystem ebenfalls von den Geräten, Speichern usw. abstrahiert. Die Unterteilung der Abstraktion in zwei Ebenen hat den Vorteil, daß der Übersetzer als ein Werkzeug zur Programmierung des Prozessors nicht die Besonderheiten unterschiedlicher Maschinen berücksichtigen muß. Allerdings ist diese Unterteilung nicht zwingend. Es ist genauso möglich, ohne ein Betriebssystem auszukommen. Soll dabei in gleichem Maße von den Geräten und Speichern abstrahiert werden, ist jedoch ein recht hoher Aufwand erforderlich.

4.1.1 Abstraktion durch das Betriebssystem

Betriebssysteme sind ein eigener Themenkomplex, der in der Literatur umfangreich behandelt wird, z.B. in [Tanenbaum 2002]. Ihre Fähigkeit der *Abstraktion von der Maschine* zeigt sich schon an einem einfachen Beispiel. Angenommen ein Datenblock soll auf einem Datenträger, wie einer Festplatte, gespeichert werden, dann muß – falls nicht abstrahiert wird – möglicherweise zuerst der DMA- und Festplatten-Controller programmiert und anschließend der Datentransfer zur Festplatte gestartet werden. Hierbei ist es erforderlich, daß die Art und Weise, wie die verwendeten Geräte gehandhabt werden, bekannt ist. Um von diesen Details zu abstrahieren, stellen Betriebssysteme daher üblicherweise sog. *Systemfunktionen* zur Verfügung (gewöhnlich unterscheiden sich solche Systemfunktionen von herkömmlichen Funktionen nur im Aufrufmechanismus). Soll ein Datenblock gespeichert werden, so wird hierzu eine Systemfunktion aufgerufen und dabei als Parameter z.B. die Startadresse des zu speichernden Blocks, die Blockgröße und eine Kennung übergeben, mit deren Hilfe das Betriebssystem herausfindet, welches Gerät angesprochen werden soll.

Beim Aufruf einer solchen Systemfunktion ist es nun egal, ob der Datenblock tatsächlich auf der Festplatte gespeichert wird, oder ob er z.B. über die serielle Schnittstelle an ein anderes Gerät oder einen anderen Rechner gesendet wird, um dort gespeichert oder verarbeitet zu werden. Ein weiterer Vorteil der Verwendung von Systemfunktionen ist, daß ein Programm, das nur über solche Funktionen auf Geräte zugreift, leicht auf einer anderen Maschine ausgeführt werden kann, falls – und das ist eine zwingende Voraussetzung – diese Funktionen von dieser Maschine verstanden werden. Die Betriebssysteme der einzelnen Maschinen

müssen hier also identische Systemfunktionen bereitstellen, können ansonsten aber unterschiedlich realisiert sein. Durch die Systemfunktionen wird somit nicht nur lokal von einem Gerät abstrahiert – es ist egal, ob die Daten gespeichert oder versendet werden – sondern auch von den Geräten einer Maschine als ganzes – ein Programm, das ein Datenblock speichern möchte, kann dies auf einer Maschine tun, die gar keine Festplatte besitzt, indem die Daten über die serielle Schnittstelle an eine andere Maschine geschickt und dort gespeichert werden.

Je nach Gerät bzw. Betriebsmittel unterscheiden sich natürlich die Anforderungen, die an das Betriebssystem gestellt werden. So kann z.B. der Hauptspeicher ähnlich einer Festplatte ebenfalls als ein Gerät zur Speicherung von Daten angesehen werden und infolgedessen auch entsprechend einer Festplatte über Systemaufrufe angesprochen werden. Ein so verwalteter Hauptspeicher wird als *RAM-Disk* bezeichnet und unterscheidet sich für den Benutzer gegenüber einer Festplatte nur in einer erheblich höheren Geschwindigkeit. Die Frage ist nun, kann eine Festplatte auch den Hauptspeicher ersetzen, kann also die begrenzte Kapazität des Hauptspeichers durch die um eine vielhundertfach höhere Kapazität der Festplatte vergrößert werden, ohne daß der Benutzer hierauf Rücksicht nehmen muß? Die Antwort ist nicht trivial. Im Gegensatz zu einer Festplatte wird der Hauptspeicher einer Maschine aus Geschwindigkeitsgründen gewöhnlich direkt angesprochen, so daß das Betriebssystem nicht die Möglichkeit erhält, durch Systemfunktionen vom Speicher zu abstrahieren. Daher wird die Unterstützung einer *Speicherverwaltungseinheit*, einer sog. *MMU*, benötigt. Mit Ihrer Hilfe wird der physikalische Speicher und der auf einer Festplatte verfügbare Speicher für den Prozessor unsichtbar als sog. *virtueller Speicher* abstrahiert (zur technischen Realisierung siehe 6.3).

Bemerkenswert ist, daß die zur Abstraktion des physikalischen Speichers erforderlichen Festplattenzugriffe ebenfalls über eine abstrakte Systemschnittstelle aufgerufen werden, so daß hier statt der Festplatte auch andere Geräte zur Speicherung verwendet werden können. Die Kapazität des gesamten zur Verfügung stehenden Speichers kann daher im konkreten Fall sehr groß sein; sie ist jedoch in jedem Fall begrenzt, weshalb auch von der zur Verfügung stehenden Kapazität des virtuellen Speichers abstrahiert werden muß. Dies geschieht gewöhnlich, indem ein Benutzerprogramm den benötigten Speicher zunächst anfordert. Wird dabei in der hierzu aufzurufenden Systemfunktion festgestellt, daß nicht genügend Speicher zur Verfügung steht, so wird eine Fehlermeldung an das aufrufende Programm zurückgegeben und so dem Programm die Gelegenheit gegeben, geeignet zu reagieren. Die Abstraktion von dem maximal zur Verfügung stehenden Speicher geschieht hier also, indem die absolut verfügbare Kapazität gehim- gehalten wird.

Moderne Betriebssysteme abstrahieren nicht nur vom Speicher und von den angeschlossenen Geräten, sondern in sog. *Multitasking*-Betriebssystemen auch teilweise vom Prozessor. Aus Geschwindigkeitsgründen ist diese Abstraktion jedoch

nicht sehr weitreichend. So wird gewöhnlich nur von der Anzahl der tatsächlich vorhandenen Prozessoren abstrahiert, damit die Anzahl der gleichzeitig ausführbaren Programme bzw. Programmteile (Tasks) davon unabhängig ist. – Ange merkt sei, daß eine vollständige Abstraktion von einem Prozessor einschließlich seiner Register und seines Befehlssatzes ebenfalls möglich ist. Jedoch ist dafür i. allg. nicht das Betriebssystem, sondern ein eigenständiges Programm, ein sog. *Interpreter* zuständig. Entsprechende Prozessoren werden als *virtuelle Prozessoren* bezeichnet. Sie haben den Nachteil, sehr langsam zu sein, da sie auf wirklichen Prozessoren simuliert werden müssen. Dementsprechend liegt ihr Einsatz gebiet in der Bearbeitung von nicht geschwindigkeitsrelevanten Programmen, bei denen die Unabhängigkeit von einer bestimmten Maschine oder Maschinenfamilie wichtiger ist als die Ausführungsgeschwindigkeit der Programme. Der momentan bekannteste virtuelle Prozessor wird als *Java Virtual Machine (JVM)* bezeichnet und wurde von Sun entwickelt [Lindholm, Yellin 1997]. Er ermöglicht es, Programme in einem großen Netzwerk mit unterschiedlichen Maschinen, also plattformunabhängig auszuführen.

4.1.2 Abstraktion durch den Übersetzer

Ähnlich wie ein Interpreter abstrahiert auch ein *Übersetzer* von einem Prozessor, nur geschieht das auf eine andere Art und Weise. Anstatt die Befehle eines virtuellen Prozessors zur Laufzeit eines Programms zu interpretieren, werden die prozessorunabhängigen Anweisungen eines Hochsprachenprogramms in ein vom wirklichen Prozessor direkt ausführbares und somit abhängiges Maschinenprogramm übersetzt. Um also dasselbe Hochsprachenprogramm auf unterschiedlichen Maschinen ausführen zu können, muß es zuvor auf jeder Maschine in das entsprechende Maschinenprogramm übersetzt werden. Hochsprachenprogramme sind somit weniger portabel als die Programme virtueller Prozessoren, die ohne Neuübersetzung direkt von einem Interpreter ausgeführt werden können. Dafür sind die durch einen Übersetzer erzeugten Maschinenprogramme jedoch laufzeit effizienter, da die Befehle nicht interpretiert, sondern direkt ausgeführt werden. Beim Entwurf eines Übersetzers gilt der Laufzeiteffizienz daher ein besonderes Augenmerk.

Anmerkung. Einen Kompromiß zwischen der Interpretation und der Übersetzung bezüglich Laufzeiteffizienz und Portabilität strebt man durch *Just-in-time-Übersetzer (JIT-Übersetzer)* an. Hierbei wird das meist sehr einfache Programm eines virtuellen Prozessors kurz vor seiner Ausführung in das entsprechende Maschinenprogramm übersetzt und sofort ausgeführt. Im Vergleich zur Interpretation ist ein Just-in-time ausgeführtes virtuelles Programm meist erheblich schneller, wobei für die Vorübersetzung normalerweise mehr Speicher benötigt wird (was in kleinen Systemen von Bedeutung sein kann). Dennoch werden die Laufzeiten herkömmlich übersetzter Programme so nicht erreicht, da (1.) die Vorübersetzung Zeit benötigt und (2.) eine maschinenabhängige Optimierung wegen des erforderlichen Zeitaufwandes nur in einem geringen Umfang möglich ist. Die Thematik wird im folgenden nicht weiter behandelt.

Im folgenden wird beschrieben, wie ein Übersetzer von einem Prozessor abstrahiert, und zwar untergliedert in die Abstraktion vom Befehlssatz, von den zur Verfügung stehenden Registern und von den direkt verarbeitbaren Datentypen. Am wenigsten kompliziert ist die Abstraktion vom Befehlssatz. Jede Anweisung eines Hochsprachenprogramms wird meist in mehrere Maschinenbefehle umgesetzt. Abhängig vom verwendeten Prozessor entstehen hierbei unterschiedliche Befehlsfolgen, die jedoch immer identische Ergebnisse erzeugen. So wird die in vielen Hochsprachen mögliche *goto*-Anweisung z.B. in einen unbedingten Sprungbefehl umgesetzt, wobei abhängig vom Prozessor entweder der Befehl Jump (z.B. JMP beim Pentium 4) oder der Befehl Branch Always (z.B. ba beim UltraSPARC III) verwendet wird. Wie kompliziertere Hochsprachenanweisungen in Maschinenbefehle umgesetzt werden, ist in Kapitel 3 beschrieben.

Schwieriger als die Abstraktion von den Befehlen ist die Abstraktion von den Registern. Zwar ist es prinzipiell möglich, alle Variablen ausschließlich im Hauptspeicher abzulegen – Register würden hierbei nur innerhalb der für einzelne Hochsprachenanweisungen erzeugten Befehlssequenzen verwendet –, dies hat jedoch zur Folge, daß die entsprechenden Programme gewöhnlich langsamer ausgeführt werden als bei Verwendung von Registern zur Speicherung häufig benötigter Variablen. Als Alternative kann die explizite Verwendung von Registern in Hochsprachen erlaubt werden, was jedoch zu einer geringeren Portabilität der Hochsprachenprogramme führt, da z.B. nicht in allen Prozessoren die gleiche Anzahl von Registern zur Verfügung steht. Besser ist es daher, dem Übersetzer nur zu empfehlen, daß er eine Variable in einem Register speichern soll. In C geschieht dies bei der Variablen Deklaration durch Verwendung des Schlüsselworts *register*. Da dies vom Übersetzer nur als eine Empfehlung verstanden wird, gibt es hierbei keine Garantie, daß die entsprechende Variable wirklich in einem Register gespeichert wird. So darf eine Variable z.B. nicht in einem Register gespeichert werden, wenn es Referenzen auf diese Variable gibt, da nur wenige Prozessoren in der Lage sind, auf Register indirekt zuzugreifen (z.B. war der heute nicht mehr auf dem Markt befindliche RISC-Prozessor AMD 29000 hierzu in der Lage [Advanced Micro Devices 1993]).

Die Verwendung solcher Schlüsselwörter hat den Nachteil, daß in den Hochsprachenprogrammen nicht nur die reinen Algorithmen formuliert, sondern zusätzlich Anweisungen verwendet werden müssen, die den Übersetzungsvorgang steuern und so die Laufzeit des ausführbaren Programms beeinflussen. Dies ist oft nicht erwünscht: das Hochsprachenprogramm soll nach Möglichkeit vollständig, d.h. auch bezüglich der Laufzeit eines Programms von einem Prozessor abstrahieren, so daß ein Programmierer sich ganz auf die Lösung seines Problems konzentrieren kann. Die bessere Lösung ist deshalb, die Register nicht vom Programmierer, sondern automatisch durch den Übersetzer zu vergeben, und zwar so, daß möglichst oft auf Register und möglichst selten auf den Hauptspeicher zugegriffen wird. Die für eine solche Optimierung verwendeten Verfahren basieren z.B. auf heuristischen Algorithmen zur sog. *Graphenfärbung*. Sie werden hier jedoch

nicht beschrieben. Genau wie bei Verwendung des Schlüsselworts *register* muß bei der automatischen Registervergabe berücksichtigt werden, daß Variablen nur in einem Register gespeichert werden können, wenn sie im Hochsprachenprogramm nicht über Zeiger adressiert werden.

Zum Abschluß sei auf die Abstraktion von den Datentypen eines Prozessors eingegangen: Die in einer Programmiersprache verwendeten Datentypen können entweder vom Prozessor direkt bearbeitet werden, oder ihre Bearbeitung wird nachgebildet, indem z.B. Funktionen aufgerufen werden, die dem ausführbaren Programm vom Übersetzer beigefügt werden. Diese Funktionen vergrößern einerseits das ausführbare Programm, andererseits wird eine solche Funktion i. allg. auch langsamer bearbeitet als ein einzelner Befehl. Daher sollten in einem Programm nur Datentypen verwendet werden, die auch vom Prozessor direkt bearbeitet werden können, was jedoch nur möglich ist, wenn dies die zu lösende Problemstellung zuläßt. So kann eine Variable mit einem benötigten Wertebereich von 0 bis 9999 nicht in einem Byte gespeichert werden. Umgekehrt sollte jedoch eine Variable mit einem Wertebereich von 0 bis 99 auch nicht in einem 16-Bit-Wort gespeichert werden. Daher sollten in einer Hochsprache möglichst viele unterschiedliche und genau definierte Datentypen zur Verfügung stehen, die dann vom Benutzer problemabhängig ausgewählt werden können. Leider gibt es in C keine eindeutige Definition der Datentypen. Lediglich der Datentyp Character (*char*) ist als 8-Bit-breit definiert.

Normalerweise muß ein Benutzer nur Kenntnisse darüber haben, was für einen Wertebereich eine Variable eines bestimmten Datentyps hat und welche Operationen darauf ausgeführt werden können, nicht jedoch wie eine Variable codiert oder gespeichert wird. Im Gegenteil: werden solche Informationen über Variablen genutzt, wird immer die Portabilität gefährdet, da oft z.B. die Art und Weise, wie eine Variable im Speicher abgelegt ist, nicht eindeutig mit der Hochsprachendefinition festgelegt wurde und daher vom Übersetzer abhängig ist. Für die maschinennahe Programmierung, die im folgenden genauer beschrieben wird, kann es jedoch sehr hilfreich sein zu wissen, wie eine Variable codiert und gespeichert wird. Diese Kenntnis nutzend kann z.B. das Laufzeitverhalten eines Programms verbessert werden. Trotzdem sollte dies nur dann geschehen, wenn die Aufgabenstellung es unbedingt erfordert.

4.2 C-Übersetzer mit Zuschnitt auf den Prozessor

Im vorangehenden Abschnitt wurde bereits beschrieben, wie ein Übersetzer unter Mithilfe des Betriebssystems von den Details einer Maschine abstrahiert. Ein zu hohes Abstraktionsniveau ist bei der maschinennahen Programmierung meist jedoch unerwünscht, da hier oft direkt auf Komponenten und Funktionen einer Maschine zugegriffen werden muß. Deshalb sind Übersetzer zur maschinennahen Programmierung gewöhnlich um spezielle Schlüsselwörter und Definitionen er-

weiter, mit deren Hilfe ein direkter Zugriff auf Komponenten und Funktionen einer Maschine möglich ist. Als Mindestanforderung muß der Zugriff auf bestimmte Speicheradressen bzw. Register von Peripheriebausteinen und die Behandlung von Interrupts programmiert werden können. Zusätzlich sollte ein Übersetzer für die maschinennahe Programmierung hinsichtlich Geschwindigkeit und Größe effiziente Programme erzeugen. Es werden nämlich meist kleine, kostengünstige Steuerungssysteme maschinennah programmiert, die wenig Speicher besitzen und in denen Prozessoren mit relativ niedrigen Taktfrequenzen verwendet werden. So sollte der Übersetzer z.B. die von einigen Prozessoren zur Verfügung gestellten Maschinenbefehle zur Manipulation von Bitfeldern zugänglich machen, da diese vorteilhaft in Steuerungssystemen eingesetzt werden können. Um solche Befehle jedoch von einem Hochsprachenprogramm aus nutzen zu können, ist normalerweise eine Erweiterung der Sprachdefinition erforderlich.

4.2.1 Speicher- und Ein-/Ausgabezugriffe

Prozessoren unterscheiden sich u.a. in der Art des Zugriffs auf bestimmte Adressen, also z.B. in der Anzahl der ansprechbaren *Adreßräume* und ob auf diese Adreßräume linear oder segmentiert zugegriffen wird. So verwenden z.B. der PowerPC von Motorola und IBM einen für Speicher- und Ein-/Ausgabezugriffe gemeinsamen linearer Adreßraum, während die Prozessoren i80x86 und deren Pentium-Nachfolger von Intel einen segmentierten Adreßraum für Befehle und Daten und davon getrennten einen linearen Adreßraum für Zugriffe auf Peripheriebausteine besitzen (siehe auch 6.3). Im folgenden werden die aus diesen Unterschieden resultierenden Spracherweiterungen der Programmiersprache C exemplarisch beschrieben. Dabei wird der Einfachheit halber nicht zwischen Speicher- und Ein-/Ausgabezugriffen unterschieden, da Ein-/Ausgabezugriffe immer als Sonderfall von Speicherzugriffen betrachtet werden können, und zwar auch dann, wenn sich die Ein-/Ausgabezugriffe auf einen separaten Adreßraum beziehen.

Zunächst wird vorausgesetzt, daß ein Prozessor verwendet wird, der einen einzigen Adreßraum für alle Befehle, Daten und Peripherieregister besitzt und der auf diesen Adreßraum über absolute Adressen zugreift und keine MMU verwendet. In C kann ein Speicher- oder Ein-/Ausgabezugriff in diesem Fall z.B. mit Hilfe einer *Zeigervariablen (pointer)* realisiert werden:

```
char *zeiger = (char *) 0x1001;      // Speicheradresse zuweisen  
*zeiger = 0x12;                      // Speicherzugriff
```

Zur Erklärung: In der ersten Zeile wird der *Zeiger* auf eine Variable vom Typ *char* definiert und mit der (hexadezimalen) Adresse 0x1001 initialisiert. Der eigentliche Zugriff erfolgt in der zweiten Zeile, wobei der Zeiger dereferenziert wird und dabei die entsprechende Speicherstelle den Wert 0x12 zugewiesen bekommt. Wird diese Anweisungsfolge für einen Prozessor übersetzt, der über ein Segmentregister auf den Speicher zugreift, so wird die Zuweisung der Adresse des Zeigers

möglicherweise als Zuweisung eines neuen Offsets interpretiert, wobei der Zeiger einem bestimmten Segment fest zugeordnet ist. Somit wird nicht die Adresse 0x1001 adressiert, sondern eine Adresse, die durch Addition der entsprechenden fest vorgegebenen Segmentadresse mit dem Offset 0x1001 entsteht. Um auf eine bestimmte physikalische Adresse zugreifen zu können, muß ein Zeiger, bestehend aus Segmentadresse und Offset, konstruiert und der Zeigervariablen zugewiesen werden. Dies ist nachfolgend dargestellt, wobei hier die Segmentadresse und der Offset in eckigen Klammern angegeben sind.

```
far char *zeiger = (far char *) [0:0x1001]; // Adr. zuweisen
*zeiger = 0x12; // Speicherzugriff
```

Die eckigen Klammern sowie das in diesem Beispiel verwendete Schlüsselwort *far* sind bereits nichtstandardisierte Erweiterungen der C-Sprachdefinition. Zur Vereinfachung könnte auf das Schlüsselwort *far* auch verzichtet werden, wenn nämlich vorgegeben wäre, daß alle Zeiger grundsätzlich aus einer Segmentadresse und einem Offset bestehen. Nachteilig hierbei wäre jedoch, daß dann keine Zeiger deklariert werden könnten, die einem Segment fest zugeordnet sind und nur einen Offset enthalten. Gerade solche „kurzen“ Zeiger, die nur aus einem Offset bestehen, beanspruchen aber weniger Speicherplatz und werden bei einem Zugriff normalerweise auch schneller ausgewertet.

Ein weiteres Beispiel soll verdeutlichen, wie aufwendig es sein kann, über Zeiger auf eine bestimmte Adresse zuzugreifen. Spezialprozessoren für die Signalverarbeitung (*Signalprozessoren*), arbeiten oft mit Algorithmen, in denen große Vektoren miteinander verknüpft werden. Um diese Berechnungen schnell durchführen zu können, besitzen sie gewöhnlich die Fähigkeit, gleichzeitig auf wenigstens zwei Operanden zugreifen zu können. Technisch wird dies realisiert, indem die beiden Vektoren in unterschiedlichen Speichern (d.h. Adreßräumen) abgelegt werden, die parallel über separate Adressbusse angesprochen werden. Hier ist es nun erforderlich, mit jeder Variablen festzulegen, in welchem Adreßraum sie gespeichert werden soll. Bei Zeigervariablen muß außerdem festgelegt werden, welchen Adreßraum sie referenzieren. Die Situation wird beliebig kompliziert, wenn Zeiger auf Zeiger verwendet werden. – Als Beispiel ist nachfolgend eine Anweisungsfolge mit einem einfachen Zeiger dargestellt.

```
char xdata *ydata zeiger = (char xdata *ydata) 0x1001;
*zeiger = 0x12; // Speicherzugriff
```

Durch die beiden neuen Schlüsselwörter *xdata* und *ydata* wird vorgegeben, daß der Zeiger im *y*-Adreßraum gespeichert werden soll und eine Referenz auf den *x*-Adreßraum enthält (die Namen dieser Adreßräume sind willkürlich gewählt). Mit der zweiten Anweisung wird also dem Byte mit der Adresse 0x1001 im *x*-Adreßraum der Wert 0x12 zugewiesen. Wird von den unterschiedlichen Adreßräumen abstrahiert, so kann auch auf die Schlüsselwörter *xdata* und *ydata* verzichtet werden. So können z.B. die beiden Adreßräume logisch aufeinander-

folgend angeordnet werden und so ein einziger, hier als virtuell bezeichneter Adreßraum geschaffen werden. Nachteil dieser Methode ist jedoch, daß zur Laufzeit bei jedem Zugriff über Zeiger mit Hilfe einer automatisch durchzuführenden Fallunterscheidung geprüft werden muß, welcher Adreßraum tatsächlich angesprochen werden soll. Ein entsprechendes Programm wäre somit größer und langsamer, als wenn nicht von den unterschiedlichen Adreßräumen abstrahiert werden würde.

4.2.2 Datentypen

Einige Aspekte, die beim Zugriff auf bestimmte Speicheradressen von Bedeutung sind, wurden bisher nicht beschrieben, nämlich die Codierung, das Alignment und das Byte-Anordnen von bestimmten Datentypen (siehe auch 2.1.2 und 5.2.4). Dies war insofern nicht notwendig, als der in den obigen Anweisungsfolgen verwendete Datentyp *char* in C als 8-Bit-breit standardisiert, die Codierung i.allg. einheitlich und das Alignment und Byte-Anordnen für 8-Bit-breite Datentypen nicht von Bedeutung sind. Probleme treten erst auf, wenn ein Datentyp verwendet wird, der breiter als 8 Bits ist, wie z.B. der Datentyp Integer. Hierzu folgendes Beispiel:

```
int *intzeiger = (int *) 0x1001;      // Adreßzuw. für Integers
char *charzeiger = (char *) 0x1001;    // Adreßzuw. für Bytes

*intzeiger = 0x1234;                  // Integerwert schreiben
printf ("0x%x", *charzeiger);       // Bytewert lesen und ausgeben
```

Das Ergebnis dieser Anweisungsfolge ist stark vom verwendeten Übersetzer und auch vom Prozessor abhängig, von dem das Programm ausgeführt wird. Wird z.B. ein Prozessor verwendet, der die *Little-endian-Adressierung* benutzt, so wird durch die printf-Anweisung der Wert 0x34 ausgegeben. Bei einem Prozessor, der die *Big-endian-Adressierung* verwendet, wird hingegen 0x12 ausgegeben, allerdings nur, wenn der Übersetzer den Datentyp Integer als 16-Bit-breit behandelt. Behandelt er ihn jedoch als 32-Bit-breit, so führt er eine Vorzeichenerweiterung durch und gibt 0x00 aus. Möglicherweise führt das Programm auch zu einem Abbruch, da nicht alle Prozessoren einen 16-Bit-Zugriff auf eine ungerade Adresse erlauben. Dies ist aber auch davon abhängig, ob der Übersetzer Zugriffe auf ungerade Adressen nicht in mehrere Bytezugriffe umwandelt, so daß ein Misalignment auch mit ungeraden Adressen nicht mehr auftritt (was jedoch selten der Fall sein dürfte, da ein solcher Übersetzer langsam produzieren würde).

Falls ein Programm für eine spezielle Maschine geschrieben und immer derselbe Übersetzer verwendet wird, sind die hier beschriebenen Unterschiede im Verhalten des Programms ohne Bedeutung. Soll es jedoch möglich sein, den benutzten Übersetzer oder Prozessor zu wechseln, ohne das Programm neu schreiben zu müssen, so sollten kritische Anweisungsfolgen dieser Art vermieden werden. Im

konkreten Fall würde das bedeuten, Hauptspeicherzugriffe nur mit Zeigern vom Datentyp *char** durchzuführen. Dies ist aus Geschwindigkeitsgründen jedoch nicht immer möglich.

4.2.3 Interrupts

Um Programme schreiben zu können, in denen Interrupts verarbeitet werden, muß ein Übersetzer Sprachmerkmale besitzen, mit denen die Initialisierung und das Freischalten eines Interrupts sowie die Behandlung des Interrupts mittels einer Interruptroutine ermöglicht wird. Die Initialisierung umfaßt zum einen das Initialisieren der Interruptquelle, zum andern das Zuordnen der Startadresse der Interruptroutine des verwendeten Interruptsignals. (Ersteres wird im folgenden nicht weiter ausgeführt, da das Programmieren von Bausteinen, die im Ein-/Ausgabeaddressraum liegen, im Prinzip bereits in 4.2.1 beschrieben ist.) Nach der Initialisierung wird der Interrupt dann freigeschaltet. Dieses Freischalten wird hier als eine von der Initialisierung losgelöste Aktion betrachtet, da es ein eigenes Sprachkonstrukt erfordert. Eine vom Prozessor nach dem Freischalten akzeptierte Interruptanforderung führt dann zum Aufruf der entsprechenden Interruptroutine. Das folgende C-Programm zeigt die einzelnen Programmschritte.

```
interrupt service_funktion () {
    // Anweisungen, um einen Interrupt zu behandeln
}

void main () {
    // Anweisungen, um die Interruptquelle zu initialisieren

    // Vektortabelle initialisieren...
void (**fktzeiger)() = 0x0064;          // Zeiger in Vektortab.
*fktzeiger = service_funktion;             // Eintrag initialisieren

    // Interrupt freigeben...
irq_on;                                // Interrupts einschalten

    // Auf interrupts warten...
for (;;) ;                            // Endlosschleife
}
```

Das Programm besteht aus einer mit dem nichtstandardisierten Schlüsselwort *interrupt* deklarierten Interruptroutine sowie aus dem Hauptprogramm *main*. In *main* wird zunächst der Interrupt initialisiert, wobei die Programmierung der Interruptquelle – meist ein interrupt-auslösender Peripheriebaustein – nur als Kommentar angedeutet ist. Die Startadresse der Interruptroutine wird dann durch die ersten beiden dargestellten Anweisungen des Hauptprogramms eingestellt. Dabei wird vorausgesetzt, daß ein Prozessor verwendet wird, der die Startadres-

sen aller Interruptroutinen in einer *Vektortabelle* speichert (siehe 2.1.5). Um die Startadresse in diese Tabelle eintragen zu können, wird ein Zeiger auf die Adresse des entsprechenden Eintrags benötigt. Dieser Zeiger wird durch die erste Anweisung deklariert und mit der Adresse 0x0064 initialisiert (Tabelle 2-4, S. 100). Mit der zweiten Anweisung wird anschließend die Startadresse der Interruptroutine in die Vektortabelle eingetragen, wobei der Name der interrupt-behandelnden Funktion in C als Platzhalter für deren Startadresse verwendet werden kann. Die komplexe Weise der Deklaration erklärt sich daraus, daß der Zeiger auf eine Speicherstelle verweisen soll, in der ein Zeiger auf eine Funktion, hier auf die Interruptroutine, zu speichern ist. Oder anders ausgedrückt: es wird ein Zeiger auf einen Zeiger auf eine Funktion benötigt. (Zu den Problemen, die beim Zugriff auf absolute Adressen unter Verwendung von Zeigern auftreten können, siehe 4.2.1.)

Nach der Initialisierung der Vektortabelle wird die *Interruptbehandlung* mit dem Schlüsselwort *irq_on* freigeschaltet. Der zum Freischalten erforderliche Code wird hierbei vom Übersetzer erzeugt. Es stellt sich die Frage, warum hierfür ein eigenes Schlüsselwort erforderlich ist? Prozessoren werden in ihrer Arbeitsweise gewöhnlich über ein Spezialregister, meist über das Statusregister gesteuert. Um einem Prozessor mitzuteilen, daß er auf ein Interruptsignal reagieren soll, muß ein Bit oder Bitfeld innerhalb des Statusregisters (die Interruptmaske) in einer bestimmten Weise programmiert werden. Im Gegensatz zu den Steuerregistern von Peripheriebausteinen kann auf das Statusregister des Prozessors jedoch meist nicht über eine Adresse zugegriffen werden, sondern nur, indem ein spezieller Befehl ausgeführt wird (z.B. MOVESR, siehe 2.1.4). Dieser Befehl wird vom Übersetzer im ausführbaren Programm eingefügt, wenn die Anweisung *irq_on* verwendet wird. Dabei wird durch *irq_on* die Bearbeitung aller Interrupts freigeschaltet. Eine Alternative wäre, die Nummer des freizuschaltenden Interrupts z.B. in Form einer Zuweisung zu spezifizieren. Der Übersetzer würde hierbei anhand des sog. l-values, also des links neben dem Gleichheitszeichen stehenden Namens erkennen, daß es sich um keine herkömmliche Zuweisung handelt, sondern um einen speziellen Befehl zum Freischalten eines speziellen Interrupts, z.B.:

```
irq_on = 4;                                // Interrupt 4 freischalten
```

Nachdem der oder die Interrupts freigeschaltet sind, können im Hauptprogramm beliebige, vom Interrupt unabhängige oder auch abhängige Aufgaben bearbeitet werden. In dem oben dargestellten Beispielprogramm gibt es jedoch außer der Interruptverarbeitung keine weiteren Aufgaben, weshalb das Hauptprogramm mit einer *Endlosschleife* abgeschlossen ist. Solche Endlosschleifen finden sich in allen maschinennahen Programmen, da das Terminieren des einzigen von einem Prozessor ausgeführten Programms zum Systemabsturz führen würde. Es gibt hier nämlich kein Betriebssystem, in das zurückgekehrt werden kann, wenn das Hauptprogramm beendet wird.

Akzeptiert der Prozessor einen Interrupt, so rettet er zunächst einen minimalen Prozessorstatus, meist die aktuellen Inhalte des Befehlszählers und des Status-

registers, liest dann den dem Interrupt zugeordneten Eintrag der Vektortabelle und setzt die Befehlsbearbeitung an der so ermittelten Speicheradresse fort. In der Interruptroutine selbst werden gewöhnlich zuerst die Inhalte aller benötigten Prozessorregister gesichert, anschließend der Interrupt bearbeitet und schließlich die Prozessorregister wieder mit den ursprünglichen Inhalten geladen und ins Hauptprogramm zurückgesprungen. Im wesentlichen entspricht dieser Ablauf dem bei Ausführung einer parameterlosen Funktion ohne Rückgabewert, nur daß der Rücksprung aus einer Interruptroutine gewöhnlich durch einen anderen Befehl ausgelöst wird als der Rücksprung aus einer Funktion. Es ist daher naheliegend, eine Interruptroutine genau wie eine parameterlose Funktion ohne Rückgabewert zu deklarieren. Damit der Übersetzer den im jeweiligen Fall korrekten Rücksprungbefehl einsetzen kann, muß es jedoch ein Sprachkonstrukt geben, welches zwischen Unterprogrammen und Interruptroutinen unterscheidet. Ein solches Sprachkonstrukt ist das im obigen Beispielprogramm verwendete Schlüsselwort *interrupt*.

Obwohl andere Arten der Deklaration einer Interruptroutine möglich sind, hat das Schlüsselwort *interrupt* nahezu den Status eines Standards. Dies gewährleistet eine hohe Portabilität – jedoch können Besonderheiten spezieller Prozessoren nicht genutzt werden. So besitzen z.B. einige Prozessoren intern mehrere sog. *Registerbänke*, die bei Bedarf umgeschaltet werden können. Anstatt also im Falle eines Interrupts die Inhalte aller benötigten Register zeitaufwendig im Speicher zu sichern, wird nur eine neue Registerbank aktiviert. Dies ist i.allg. innerhalb eines Taktes möglich. Um auch mit einer Hochsprache in den Genuss einer solchen Technik zu kommen, bedarf es wieder einer Spracherweiterung. Die nachfolgende Deklaration verwendet hierzu das Schlüsselwort *use*, um dem Übersetzer mitzuteilen, daß die Interruptroutine die Registerbank 1 verwenden soll. Der Übersetzer erzeugt daraufhin den notwendigen Code, durch den am Anfang der Interruptroutine die Registerbank 1 und am Ende der Interruptroutine erneut die zuvor aktive Registerbank aktiviert wird.

```
interrupt service_funktion () use bank1
```

4.2.4 Nutzung prozessorspezifischer Merkmale

Mit den bis hierher beschriebenen Hochsprachenerweiterungen ist es möglich, maschinennahe Programme zur Steuerung einfacher wie auch komplexer Systeme zu schreiben. Allerdings bleiben dabei noch viele prozessorspezifische Fähigkeiten, die für die maschinennahe Programmierung zwar nützlich, aber nicht unbedingt erforderlich sind, unberücksichtigt. Bei der Programmierung kleiner Systeme können solche prozessorspezifischen Details jedoch von hohem Nutzen sein, da sie die Kosten eines Systems vermindern helfen. So wird möglicherweise ein Programm durch einen einfachen Prozessor geringer Geschwindigkeit gerade noch schnell genug ausgeführt, wenn die besonderen Fähigkeiten des

benutzten Prozessors auch verwendet werden. Übersetzer für solche Prozessoren besitzen daher oft *Spracherweiterungen* – meist in Form vordefinierter Variablennamen oder zusätzlicher Datentypen – mit deren Hilfe viele der zur Verfügung stehenden Fähigkeiten eines Prozessors genutzt werden können.

Ein Beispiel für einen vordefinierten Variablenamen wurde bereits im Zusammenhang mit der Programmierung von Interrupts beschrieben. Dort wurde das Schlüsselwort *irq_on* als Variablenname verwendet, um die Interruptbearbeitung freizuschalten. Es ist leicht vorstellbar, wie ein solcher Mechanismus verwendet werden kann, um z.B. ein sog. Vectorbase-Register zu initialisieren. Ein solches in vielen Prozessoren vorhandenes Register enthält die Basisadresse der für die Interruptverarbeitung erforderlichen Vektortabelle (siehe hierzu auch 2.1.1 und 2.1.5).

```
vector_base = 0x1000;           // Startadresse der Vektortabelle
```

Als Beispiel für eine Spracherweiterung durch Datentypen wird ein Prozessor betrachtet, der einen Teil seines Speichers bitweise adressieren kann (typisch ist sonst die byteweise Adressierung). Um diese Fähigkeit des Prozessors nutzen zu können, muß vom Übersetzer der Datentyp *bit* zur Verfügung gestellt werden. Damit können z.B. acht Flags in einem einzigen Byte untergebracht werden, anstatt – wie sonst in C üblich – für jedes Flag eine eigene Variable vom Typ Integer oder zumindest Character verwenden zu müssen. Dies kann vor allem in solchen Anwendungen zu einer Speicherplatzersparnis führen, in denen viele Schalter abgefragt bzw. Akteure gesteuert werden müssen, da sich deren Zustände durch Flags oft gut repräsentieren lassen. – Normalerweise kann es dem Übersetzer überlassen werden, unter welcher Bit-Adresse er ein Flag ablegt. Falls es jedoch erforderlich sein sollte, ein Flag an einer bestimmten Bit-Adresse zu manipulieren, so kann dies auf die gleiche Weise geschehen wie mit anderen Datentypen, nämlich, indem ein Zeiger auf eine Variable vom Typ *bit* deklariert und entsprechend der Adresse initialisiert wird, z.B.:

```
bit *flag = 4;           // Bitadresse setzen
*flag = ~*flag;          // z.B. Flag invertieren
```

4.2.5 Portabilität

Der Begriff der Portabilität – in den bisherigen Ausführungen bereits einige Male erwähnt – hat eine nicht zu unterschätzende Bedeutung. So ist jede prozessor-spezifische oder übersetzerspezifische Erweiterung eines Übersetzers, die genutzt wird, als kritisch zu betrachten, da ein Wechsel des Prozessors oder Übersetzers dazu führen kann, daß bereits erstellte Programme nicht mehr übersetzt werden können. Für ein wirtschaftlich arbeitendes Unternehmen ist dies von großem Nachteil, da z.B. die gesamte Programmentwicklung eines Gerätes erneut erforderlich werden kann, wenn der bisher in dem Gerät eingesetzte Prozessor nicht

mehr verwendet werden soll oder kann. Im folgenden wird beschrieben, wie die Portabilität von Programmen, in denen prozessorspezifische Details genutzt werden, verbessert werden kann, und zwar dadurch, daß zwischen dem vom Programmierer geschriebenen und dem vom Übersetzer verarbeiteten Quellprogramm eine künstliche Schicht eingefügt wird, durch die von prozessor- bzw. übersetzerspezifischen Schreibweisen abstrahiert wird. In C kann hierzu der sog. *Präprozessor* (Vorübersetzer) verwendet werden, mit dessen Hilfe es möglich ist, beliebige Zeichenfolgen durch Namen zu repräsentieren. Wird ein solcher Name in einem Programm verwendet, so wird er durch die entsprechende Zeichenfolge textuell ersetzt. Wie damit von nicht standardisierten Schreibweisen abstrahiert werden kann, wird anhand der nachfolgend dargestellten Anweisungsfolge exemplarisch gezeigt.

Anmerkung. Präprozessoren haben in der Programmierung nichtmaschinennaher Programme heute nur noch eine geringe Bedeutung. So sollten Makros in C++ z.B. durch die typensichereren Inline- und Template-Funktionen ersetzt werden. Natürlich könnten diese programmiersprachlichen Konstrukte, sofern vorhanden, auch für die maschinennahe Programmierung verwendet werden. Allerdings erlauben Sie es i.allg. nicht, maschinen- wie auch übersetzerspezifische Funktionen, die nicht standardisiert sind, zu abstrahieren. Dies ist jedoch mit einem Präprozessor möglich.

```
// Abstraktion von Datentypen...
#define BIT           bit
#define INT          int
#define UNSIGNED     unsigned int
:
#define CHARPOINTER   far CHAR *
#define INTPOINTER    far INT *
:
// Definitionen wichtiger Konstanten...
#define MAXINT        ((INT)32767)
#define MININT        ((INT)-32768)
:
// Makros für Speicherzugriffe...
#define ADDR(addr)      [addr & ~0xffffL: addr & 0xffff]
:
// Makros und Namen zur Interruptverarbeitung...
#define INTERRUPT      interrupt
#define ATTRIBUTE(attrib) use attrib
#define INITIRQ(no,addr,fkt) {void (**fktp)() = addr; \
                           *fktp = fkt; \
                           irq_on = no; }
```

Die Definitionen sind in mehrere logisch zusammenhängende Blöcke unterteilt: Zuerst wird von den Datentypen, die ein Übersetzer zur Verfügung stellt, abstrahiert. Hierzu werden neue Bezeichnungen für die im Programm zu verwendenden Datentypen eingeführt. Um z.B. eine Variable vom Typ *bit* zu deklarieren, kann statt dessen in einem diese Definition verwendenden Programm eine Variable

vom Typ BIT (großgeschrieben) deklariert werden. Der Präprozessor ersetzt dann den Namen BIT in das Schlüsselwort *bit*. Soll nun ein Programm von einem anderen Übersetzer, dem das Schlüsselwort *bit* nicht bekannt ist, verarbeitet werden, dann muß nur die Definition des Namens BIT verändert und das Programm neu übersetzt werden (so könnte BIT z.B. als *int* definiert sein). Bei einer solchen Vorgehensweise ist es vollkommen unwichtig, wie häufig der Name BIT in dem Programm verwendet wird.

Bemerkenswert ist, daß in dem obigen Beispiel bei den Abstraktionen von den Datentypen nicht nur das wenig portable Schlüsselwort *bit* durch einen neuen Namen ersetzt wird, sondern auch die in allen C-Übersetzern bekannten Datentypen *int*, *long* usw. Da in C keine einheitliche Definition dieser Datentypen existiert, werden hier neue Typennamen eingeführt, die einen bestimmten Wertebereich garantieren. So kann der Name INT z.B. zur Deklaration von Variablen verwendet werden, die einen garantierten Wertebereich von +32767 bis -32768 (16 Bit) aufweisen. Wird ein Übersetzer verwendet, in dem der Datentyp *int* als 8-Bit breit definiert ist, so muß nur dem Namen INT der Datentyp *long* zugewiesen werden. Das die Definitionen verwendende Programm, in dem der Name INT zur Deklaration von Variablen benutzt wird, muß jedoch nicht angepaßt werden. Abgeschlossen wird die Abstraktion von den Datentypen hier durch die Definition von Namen zur Deklaration von Zeigern. Durch diese Definitionen kann die Verwendung von Schlüsselwörtern, wie *far*, *xdata*, *ydata* usw. an anderen Stellen des Programms vermieden werden.

Angemerkt sei noch, daß nach der Definition der als Datentypen zu verwendenden Namen jeweils deren Wertebereiche durch Konstanten definiert sind (MAXINT, MININT usw.). Da in einem Programm ein bestimmter Wertebereich gewöhnlich vorausgesetzt wird, sollten diesen Konstanten, sobald sie erst einmal verwendet wurden, beim Wechsel des Übersetzers nicht verändert werden. Zur Schreibweise läßt sich anmerken, daß die den jeweiligen Zahlen in Klammern vorangehenden Typennamen in C als *Cast-Operatoren* bezeichnet werden. Hierdurch werden die Zahlen entsprechend der in Klammern stehenden Angaben typisiert.

Die letzten beiden Blöcke in der oben dargestellten Anweisungsfolge definieren sog. *Makros*. Im Gegensatz zu einfachen Namen sind Makros parametrisierbar. Wird ein Makro in einem Programm verwendet, so werden die als Parameter angegebenen Namen zuerst in der dem Makro zugeordneten Zeichenfolge ersetzt und anschließend die so erzeugte Zeichenfolge statt des Makros in das Programm eingesetzt (siehe auch 3.1.4). Bild 4-1 zeigt dazu als Beispiel, wie ein Präprozessor die Deklaration eines Zeigers, dem mit Hilfe des Makros ADDR eine Adresse zugewiesen wird, umsetzt. Der Vollständigkeit halber wurden die verwendeten Definitionen ebenfalls dargestellt.

Insgesamt werden also die prozessor- bzw. übersetzerspezifischen Details zur Deklaration und Definition eines Zeigers, der auf eine bestimmte Adresse ver-

Definitionen

```
#define CHARPOINTER far CHAR *
#define ADDR (addr) [addr & ~0xffffL: addr & 0xffff]
```

Deklaration vor und nach Bearbeitung durch den Präprozessor

```
CHARPOINTER zeiger = (CHARPOINTER) ADDR (0x1001);
```

```
far CHAR * zeiger = (far CHAR *) [0x1001 & ~0xffffL: 0x1001 & 0xffff];
```

Bild 4-1. Umsetzung eines Makros bei der Deklaration eines Zeigers. Die gestrichelten Pfeile symbolisieren die Parameterersetzung, die durchgezogenen Pfeile die Namens- bzw. Makroersetzung

weist, hinter einem Namen und einem Makro verborgen. Für die Benutzung eines Zeigers ist jedoch kein Name oder Makro vorgesehen. Es wird also vorausgesetzt, daß das Dereferenzieren eines Zeigers unabhängig vom Übersetzer immer in der gleichen Art und Weise formuliert werden kann. Falls jedoch ein Übersetzer verwendet wird, der nicht Zeiger, sondern z.B. ein spezielles Sprachkonstrukt zur Verfügung stellt, um Zugriffe auf bestimmte Speicherstellen programmieren zu können, so reichen die oben dargestellten Namen und Makros nicht aus, um dahinter alle prozessor- bzw. übersetzerspezifischen Details zu verbergen. Eine gute Alternative ist es daher, Makros für den Zugriff auf bestimmte Speicherstellen zu definieren. Anstatt also einen Zeiger zu deklarieren, zu definieren und schließlich zu benutzen, werden diese Aktionen in je einem Makro für Lese- und Schreibzugriffe zusammengefaßt. Für den Lesezugriff könnte ein solches Makro z.B. folgendermaßen definiert sein:

```
#define READ(result,addr) (\n    CHARPOINTER p = (CHARPOINTER) ADDR (addr); \n    result = *p;\n}
```

Auch wenn hinter einem solchen Makro jede prozessor- oder übersetzerspezifische Besonderheit verborgen werden kann, hat es doch den Nachteil, weniger geschwindigkeits- und speichereffizient als die zuvor beschriebene Alternative zu sein. So würde z.B. das mehrfache Lesen einer bestimmten Speicherstelle zur Folge haben, daß der lokale Zeiger p auch mehrfach die jeweils gleiche Adresse zugewiesen bekommt (was von einem guten Übersetzer zugegebenermaßen optimiert würde).

Neben den Makros für Speicherzugriffe sind in der oben angegebenen Anweisungsfolge noch Makros und Namen zur Interruptverarbeitung definiert. Sie werden analog zu den soeben behandelten Makros und Namen angewandt und

werden deshalb hier nicht weiter beschrieben. Es bleibt noch anzumerken, daß Definitionen zur Abstraktion von prozessor- bzw. übersetzerspezifischen Details normalerweise in einer sog. *Definitionsdatei* zusammengefaßt werden. Um einen Namen oder ein Makro verwenden zu können, braucht diese Definitionsdatei nur über eine *include-Anweisung* eingebunden zu werden. Dabei ist es möglich, die gleiche Definitionsdatei auch in unterschiedlichen Modulen eines oder mehrerer Programme zu verwenden. Im Falle einer Änderung einer Definition ist nur die Definitionsdatei zu ändern, nicht jedoch die einzelnen Module, die die Definitionsdatei verwenden.

4.3 Standard-C mit Spezialisierung durch Assemblereinbindungen

Im letzten Abschnitt wurde gezeigt, wie herkömmliche Standard-C-Übersetzer erweitert werden müssen, um mit ihnen maschinennahe Programme schreiben zu können. Dabei wurde ein Sprachkonstrukt gänzlich außer acht gelassen, nämlich die maschinennahe Programmierung mit Hilfe von Assemblerbefehlen, die in das C-Programm eingebettet werden. Mit einer solchen Spracherweiterung können alle Möglichkeiten, die eine Maschine zur Verfügung stellt, genutzt werden, jedoch mit dem Nachteil geringer Portabilität, da Assemblerbefehle naturgemäß prozessorspezifisch sind und somit ein Programm nicht ohne Änderungen auf eine mit einem anderen Prozessor arbeitende Maschine übertragen werden kann. Die Handhabbarkeit solcher wenig portablen Funktionen kann – ähnlich wie im letzten Abschnitt beschrieben – verbessert werden, wenn die Funktionen, die entweder vollständig in Assembler oder teils in Assembler, teils in C geschrieben sind, in einer separaten Datei, dem sog. *Maschinenmodul*, zusammengefaßt werden, so daß ggf. erforderliche Änderungen nur an dieser einen Stelle vorgenommen werden müssen.

Zur Übersetzung des Maschinenmoduls wird ein C-Übersetzer benötigt, der in seiner Funktionalität erweitert ist und eingeschobene Assemblerbefehle verarbeiten kann. Es ist jedoch ebenso möglich, auf einen speziell erweiterten Übersetzer zu verzichten, wenn nämlich das Maschinenmodul vollständig in Assembler, also als Assemblermodul geschrieben ist. Ein solches *Assemblermodul* kann mit einem einfachen Assembler und alle nicht maschinennahen Module, die zu einem Projekt gehören, mit einem Standard-C-Übersetzer bearbeitet werden. – Im folgenden wird zunächst beschrieben, wie Assemblerbefehle in C-Programme eingebettet werden können. Die Details bei der Programmierung eines Assemblermoduls werden im Anschluß daran behandelt. Am Ende des Abschnitts wird schließlich der sog. Startup-Code beschrieben, der eine Art Rahmenwerk bildet, damit C-Programme auf betriebssystemlosen Maschinen ausgeführt werden können.

4.3.1 Inline-Assembler

Ein Übersetzer für C-Programme, in denen Assemblerbefehle eingebettet werden können, muß außer dem Standard-ANSI-C-Sprachumfang auch den gesamten Assemblerbefehlssatz des Prozessors verstehen, für den das Programm übersetzt werden soll. In der Praxis verarbeitet der Übersetzer die Assemblerbefehle jedoch nicht selbst, sondern überläßt dies einem sog. *Inline-Assembler*, der je nach Implementierung ein eigenständiger Assembler oder ein in den Übersetzer integriertes Funktionsmodul ist. Die Übersetzung eines Programms geschieht nun in zwei Stufen: Zuerst wird vom Übersetzer ein Assemblerprogramm generiert, in das die im C-Programm eingebetteten Assemblerbefehle (geringfügig oder gar nicht modifiziert) übertragen werden, und anschließend wird dieses Assemblerprogramm vom Inline-Assembler in ein ausführbares Programm übersetzt. Damit der Übersetzer die an den Inline-Assembler weiterzurichtenden Befehle erkennen kann, wird im folgenden Programmbeispiel das Schlüsselwort *asm* verwendet.

```
// Das Programm wartet, bis ein bestimmter Datenwert, der der
// Konstanten ACTIVATE entspricht, über einen Peripherie-
// baustein empfangen oder eine durch die Konstante TIMEOUT
// vorgegebene Zeitschranke überschritten wird. Um zu er-
// erkennen, daß ein Datenwert vom Peripheriebaustein
// empfangen wurde, wird Bit 4 des Controlregisters
// (Speicheradresse: CONTROL_REGISTER) abgefragt. Ist es
// gesetzt, wird der empfangene Wert aus dem Empfangsregister
// (Speicheradresse: RECEIVE_REGISTER) ausgelesen und in der
// C-Variablen receive gespeichert.

#define CONTROL_REGISTER 0x1001      // Adresse des Controlreg.
#define RECEIVE_REGISTER 0x1002      // Adresse des Empfangsreg.
#define ACTIVATE    0x12            // Zu suchender Adreßwert
#define TIMEOUT     1000            // Wartezeit

volatile unsigned char receive;      // Empfangspuffer

void main () {
    int time;                      // Timeout-Zähler
    receive = 0;                    // Bisher nichts empfangen
    for (time = 0; (receive != ACTIVATE) && (time < TIMEOUT);
        time++){
        asm {
            BTST.B#4,CONTROL_REGISTER    // Datum empfangen?
            BNE._11                     // Nein...
            MOVE.BRECEIVE_REGISTER,_receive // Datum lesen
            _11:
        }
    }
}
```

Die Beschreibung der Funktion dieser Anweisungsfolge ist dem Programm als Kommentar vorangestellt. Zu beachten ist, daß die Variable receive als global und als *volatile* deklariert wurde. Dies ist erforderlich, weil auf receive in dem in Assembler geschrieben Teil des Programms zugegriffen werden soll. Dabei gewährleistet das Schlüsselwort *volatile*, daß die Variable im Speicher und nicht in einem Register untergebracht wird. Es ist nämlich i.allg. nicht möglich, dem Übersetzer mitzuteilen, in welchem Register eine Variable gespeichert werden soll, weshalb Variablen, die nicht als *volatile* deklariert sind, auch nicht durch eingebettete Assemblerbefehle angesprochen werden können. Aus einem ganz ähnlichen Grund muß die Variable receive außerdem als global deklariert werden. Lokale Variablen werden nämlich normalerweise auf dem Stack untergebracht, so daß sie nur dann durch einen eingebetteten Assemblerbefehl adressiert werden können, wenn der Aufbau des Stacks bekannt ist.

Um eine als global und als *volatile* deklarierte Variable in einem eingebetteten Assemblerbefehl verwenden zu können, muß dem Variablenamen üblicherweise ein Unterstrich vorangestellt werden. Anstatt also die Variable direkt mit dem Namen receive zu adressieren, muß statt dessen der Name _receive benutzt werden. Die Begründung hierfür ist, daß *Namenskonflikte* mit Assemblerbefehlen und Assembleranweisungen auf diese Weise vermieden werden können. Würde nämlich z.B. eine Variable mit dem Namen MOVE in einem C-Programm deklariert und dieser Name vom Übersetzer unverändert im erzeugten Assemblerprogramm eingesetzt, so würde möglicherweise ein nicht lösbarer Konflikt mit dem Assemblerbefehl MOVE auftreten. Dies wird durch die automatische Änderung aller Variablennamen durch den Übersetzer verhindert. Dabei ist es ohne Bedeutung, in welcher Weise eine Änderung der Variablennamen erfolgt, solange Namenskonflikte vermieden werden. Allerdings ist die Verwendung eines dem Variablenamen vorangestellten Unterstrichs bei vielen C-Übersetzern gebräuchlich.

In dem oben dargestellten Programm wurde auch der Sprungmarke _11 ein Unterstrich vorangestellt. Da diese Sprungmarke nur in dem in Assembler geschriebenen Teil des Programms verwendet wird, ist der Unterstrich hier jedoch nicht unbedingt erforderlich. Trotzdem sollte die vom Übersetzer definierte Namenskonvention auch hier verwendet werden, da dadurch z.B. Namenskonflikte mit vom Übersetzer automatisch erzeugten Namen, wie 11, vermieden werden. Als einfache Regel sollte jedem Namen, der vom Assembler verarbeitet wird, ein Unterstrich vorangestellt werden. Diese Regel erklärt auch, warum den Konstanten CONTROL_REGISTER und RECEIVE_REGISTER kein Unterstrich vorangestellt wurde. Sie werden bereits vom Präprozessor, also noch vor der Übersetzung, textuell durch die entsprechenden Zeichenfolgen – hier also Zahlenwerte – ersetzt. Zum Abschluß ist das Assemblerprogramm angegeben, das durch Übersetzung des obigen C-Programms erzeugt wurde (es ist natürlich abhängig vom verwendeten Übersetzer).

```

        RORG    data
_receive DC.W 0
        RORG    text
_main:
; Hier fügt der Übersetzer Code ein, der zur Ausführung
; des Programms benötigt wird (siehe 4.3.3)
        MOVE.W #0,R0
11:
        BTST.B #4,0x1001
        BNE     _11
        MOVE.B 0x1002,_receive
_11:
        ADD.W  #1,R0
        CMP.B  #0x12,_receive
        BEQ    12
        CMP.W  #1000,R0
        BLO    11
12:
        EXIT
        END

```

Dieses Beispiel soll verdeutlichen, wie sich die beschriebenen Details auf das vom Übersetzer erzeugte Assemblerprogramm auswirken können. Die im ursprünglichen Programm eingebetteten Assemblerbefehle sind hier zwischen den Sprungmarken 11 und _11 eingefügt worden. Mit der Ersetzung der Konstanten CONTROL_REGISTER und RECEIVE_REGISTER, durch die Zahlen 0x1001 und 0x1002, entspricht dieser Teil des Assemblerprogramms genau den im C-Programm eingebetteten Assemblerbefehlen. Der Variablenname receive (ohne Unterstrich) tritt im erzeugten Assemblerprogramm nicht mehr auf. Statt dessen wird auch in den vom Übersetzer generierten Assemblerbefehlen auf die am Anfang des Assemblerprogramms deklarierte globale Variable _receive zugegriffen. Die zweite im C-Programm verwendete Variable mit dem Namen time ist vom Übersetzer im Register R0 untergebracht worden. Ein Zugriff auf diese Variable in dem in Assembler geschriebenen Teil des C-Programms ist somit zwar möglich, aber nicht anzuraten, da der Übersetzer genausogut ein anderes Register hätte verwenden können. Schließlich kann dem Assemblerprogramm noch entnommen werden, wie durch den Unterstrich der explizit vom Benutzer deklarierten Sprungmarke _11 ein Namenskonflikt mit der vom Übersetzer automatisch erzeugten Sprungmarke 11 vermieden werden konnte. Wäre also in dem ursprünglichen C-Programm die Sprungmarke 11 verwendet worden, hätte der Assembler eine doppelt deklarierte Sprungmarke als Fehler gemeldet.

4.3.2 Assemblermodule

Wie bereits zu Beginn dieses Abschnitts erwähnt wurde, sollten in Assembler geschriebene Befehlsfolgen möglichst in einer Datei (auch als *Maschinenmodul*

bezeichnet) zusammengefaßt werden. So können Änderungen am Programm, die durch den Wechsel des Übersetzers oder der Maschine ggf. erforderlich werden, auf diese eine Datei beschränkt werden. Ein solches Maschinenmodul kann in einem für die maschinennahe Programmierung erweiterten C geschrieben sein oder vollständig in Assembler. Letzteres, als *Assemblermodul* bezeichnet, hat den Vorteil, daß zur Übersetzung ein herkömmlicher Assembler und für alle anderen Module ein Standard-C-Übersetzer verwendet werden kann. Die auftretenden Probleme bei der Programmierung des Maschinenmoduls in C sind bereits behandelt worden, so daß im folgenden nur darauf eingegangen wird, wie eine Assembler-Funktion aufgebaut sein muß, damit sie aus einem C-Programm heraus aufgerufen werden kann, und umgekehrt, wie eine C-Funktion aus einem Assemblerprogramm heraus aufgerufen werden kann.

Der Wechsel zwischen einer in C und einer in Assembler geschriebenen Anweisungsfolge wird durch einen *Funktionsaufruf* bzw. *-rücksprung* bewirkt. Hierzu ist es erforderlich, daß die in C und in Assembler geschriebenen Funktionen den gleichen Konventionen z.B. für die *Parameterübergabe* und den Rücksprung gehorchen. Die zu berücksichtigenden Regeln werden dabei durch den jeweils verwendeten C-Übersetzer vorgegeben, da es in C nicht möglich ist, die Konventionen zur Parameterübergabe frei an eine in Assembler geschriebene Funktion anzupassen. Um eine Assembler-Funktion aus einem C-Programm heraus aufzurufen, muß also die Assembler-Funktion – bezüglich Funktionsaufruf und -rückprung – das gleiche Verhalten aufweisen, wie eine vom Übersetzer automatisch erzeugte Funktion. Ebenso müssen beim Aufruf einer C-Funktion aus einem Assemblerprogramm heraus die bei einem Funktionsaufruf erforderlichen Vor- und Nacharbeiten in Assembler programmiert werden, und zwar entsprechend den vom C-Übersetzer vorgegebenen Konventionen.

Die bei einem Funktionsaufruf bzw. -rücksprung erforderlichen Aktionen sind natürlich vom jeweiligen Übersetzer abhängig und werden deshalb hier nur exemplarisch an dem in Bild 4-2 dargestellten Beispiel beschrieben. Es wird dabei vorausgesetzt, daß die Parameterübergabe vollständig auf dem *Stack* abgewickelt wird und daß Zugriffe auf Parameter oder *lokale Variablen* mit Hilfe eines *Framepointers* (FP) erfolgen (siehe dazu 3.3.2). Vor Aufruf der Funktion f zeigt der Stackpointer auf die Speicherstelle SP_{1a} und der Framepointer auf die Speicherstelle FP₁. An den obersten drei Positionen des Stacks befinden sich zu diesem Zeitpunkt die zum Hauptprogramm lokalen Variablen a, b und c (dunkel unterlegt). Mit dem Funktionsaufruf werden zuerst die im Hauptprogramm übergebenen Argumente ausgewertet und die ermittelten Ergebnisse auf dem Stack als Parameter der Funktion f abgelegt. In C werden die Argumente einer Funktion in der umgekehrten Reihenfolge, in der sie beim Aufruf angegeben wurden, ausgewertet. In diesem Beispiel wird vom Hauptprogramm die Funktion f (c, a, b) aufgerufen, weshalb die Werte der einzelnen Variablen in der Reihenfolge b, a, c auf dem Stack abgelegt werden. Unmittelbar bevor zur Startadresse der Funktion verzweigt wird, d.h., bevor der eigentliche Funktionsaufruf ausgeführt wird, zeigt

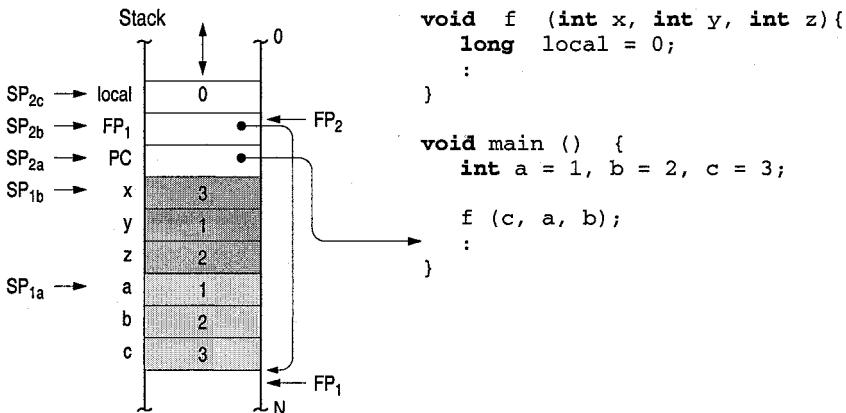


Bild 4-2. Aufbau des Stacks nach Aufruf der parametrisierten Funktion *f* (in Anlehnung an [Borland 90])

der Stackpointer auf die Speicherstelle SP_{1b} und der Framepointer unverändert auf die Speicherstelle FP_1 .

Mit dem Funktionsaufruf wird auch die Rücksprungadresse auf dem Stack gesichert, so daß der Stackpointer anschließend auf die mit SP_{2a} bezeichnete Speicherstelle weist. Als nächstes wird der alte Inhalt des Framepointers auf dem Stack gesichert und der Framepointer gleich dem aktuellen Inhalt des Stackpointers gesetzt (SP_{2b} und FP_2). Zusammen mit dem Framepointer können ggf. auch andere Register gesichert werden, deren Inhalte von der Funktion verändert werden. Die so gesicherten Inhalte können nach Ausführung der Funktion wieder in die Register zurückgeschrieben werden. In der Funktion *f* muß noch der für die lokale Variable *local* benötigte Speicher auf dem Stack reserviert werden. Hierzu wird der Stackpointer entsprechend des benötigten Speicherbedarfs vermindert. Somit ergibt sich die in Bild 4-2 dargestellte endgültige Aufteilung des Stacks (SP_{2c} und FP_2). Die Assemblerbefehle, die in der Funktion ausgeführt werden, können auf die Parameter und lokalen Variablen zugreifen, indem die basisrelative Adressierung mit dem Framepointer als Basis verwendet wird. Dabei werden Parameter mit positiven Offsets und lokale Variablen mit negativen Offsets adressiert (siehe auch 2.1.3 und 3.3.2).

Der Stack wird normalerweise durch den vom C-Übersetzer erzeugten Code automatisch in der hier beschriebenen Art und Weise aufgebaut. Sollen jedoch Assembler-Funktionen aus einem C-Programm oder C-Funktionen aus einem Assemblerprogramm heraus aufgerufen werden, so muß ein Teil der Aufgabe von dem jeweils in Assembler geschriebenen Programm bearbeitet werden. Das heißt: Wird eine C-Funktion aus einem Assemblerprogramm heraus aufgerufen, so muß im Assemblerprogramm dafür gesorgt werden, daß (1.) die Funktionsargumente in der richtigen Reihenfolge auf dem Stack abgelegt werden, (2.) die C-Funktion aufgerufen wird und (3.) nach dem Aufruf der für die Argumente benötigte Stack-

bereich wieder freigegeben wird. Nachfolgend ist wiedergegeben, wie die in Bild 4-2 dargestellte Funktion f aus einem Assemblerprogramm heraus aufgerufen werden kann, falls der Framepointer in dem Prozessorregister R6 und der Stackpointer in dem Prozessorregister R7 untergebracht sind. Dazu sei noch angemerkt, daß ein schreibender Zugriff auf eine lokale Variable keine Wirkung auf die als Argumente übergebenen Variablen des aufrufenden Programms hat, da mit dem letzten Befehl der nachfolgenden Befehlsfolge (ADD.W) die ggf. veränderten lokalen Variablen verworfen werden.

```
:
MOVE.H -4(R6),-(R7) ; Wert von b auf den Stack (gleich z)
MOVE.H -6(R6),-(R7) ; Wert von a auf den Stack (gleich y)
MOVE.H -2(R6),-(R7) ; Wert von c auf den Stack (gleich x)
BSR    f              ; f (c, a, b)
ADD.W #6,R7          ; Platz für Argumente freigeben
:
```

Soll eine Assembler-Funktion aus einem C-Programm heraus aufgerufen werden, so muß in der Assembler-Funktion nur dafür gesorgt werden, daß der Inhalt des Stacks und die Registerinhalte vor und nach Aufruf der Funktion unverändert sind. Trotzdem sollte auch hierbei der durch den Übersetzer vorgegebene Aufbau des Stacks beibehalten werden, d.h., es sollten (1.) alle in der Funktion benötigten Register gesichert, (2.) der Framepointer gesetzt und (3.) ggf. lokaler Speicher auf dem Stack reserviert werden. Entsprechend sollte am Ende der Funktion in umgekehrter Reihenfolge zuerst der lokale Speicher auf dem Stack freigegeben, anschließend alle Register einschließlich des Framepointers mit ihren alten Werten geladen und schließlich ins Hauptprogramm zurückgesprungen werden. Nachfolgend ist gezeigt, wie die in Bild 4-2 dargestellte Funktion f als Assembler-Funktion aufgebaut sein muß, wenn bezüglich der Prozessorregister dieselben Annahmen wie im vorigen Beispiel gemacht werden.

```
f:  MOVEM.W R6,-(R7)      ; Zumindest Framepointer sichern
     MOVE.W R7,R6          ; Framepointer setzen
     SUB.W #4,R7           ; Lokalen Speicher reservieren
     :
     MOVE.W #0,-4(R6)      ; local = 0
     SUB.H #1,8(R6)         ; x dekrementieren (x--)
     :
     ADD.W #2,R7           ; Lokalen Speicher freigeben
     MOVEM.W (R7)+,R6       ; Framepointer restaurieren
     RTS                   ; Zurück zum Hauptprogramm
```

Die beschriebenen Aktionen zur Anbindung einer Assembler-Funktion an ein C-Programm sind durch die Befehle am Anfang und Ende der Funktion realisiert. Zusätzlich ist in der Mitte der Funktion dargestellt, wie auf lokale Variablen und Parameter zugegriffen werden kann, indem die *basisrelative Adressierung* verwendet wird. Dabei wird die lokale Variable local mit 0 initialisiert und der Para-

meter x dekrementiert. Zu beachten ist, daß die Variable local vom Typ *long* als 32-Bit-Wert und der Parameter x vom Typ *int* als 16-Bit-Wert adressiert werden. Hier werden also für die Parameterübergabe jeweils Speicherzellen optimaler Breite verwendet (von vielen Übersetzern werden im Gegensatz hierzu Speicherzellen einheitlicher Breite verwendet).

Funktionsergebnisse werden in C-Programmen entweder indirekt mit Hilfe eines der Funktion als Parameter übergebenen Zeigers (*call by reference*) oder direkt über einen *Rückgabewert* an das aufrufende Programm übergeben. Die Übergabe eines Zeigers wird in C genauso gehandhabt wie die Übergabe eines Werts und muß in einem Assemblerprogramm daher auch nicht gesondert berücksichtigt werden (siehe auch 3.3.2). Dies gilt jedoch nicht für die Behandlung von Rückgabewerten. Im folgenden werden daher zwei alternative Techniken zur Rückgabe von Ergebnissen beschrieben, zunächst die Ergebnisrückgabe über den Speicher und dann die Ergebnisrückgabe über ein oder mehrere Prozessorregister. Bei der Ergebnisrückgabe über den Speicher wird vor dem Aufruf einer Funktion ein zusätzlicher Speicherbereich auf dem Stack reserviert, in dem die aufgerufene Funktion einen Rückgabewert ablegen kann. Der Inhalt dieses Speicherbereichs kann dann nach dem Funktionsaufruf im aufrufenden Programm verarbeitet werden. Bei diesem Mechanismus ist das aufrufende Programm dafür verantwortlich, den Speicherbereich vom Stack wieder zu entfernen. Aus Sicht der aufgerufenen Funktion erscheint der Speicherbereich für die Ergebnisrückgabe wie der eines zusätzlichen Parameters, in dem ein Wert abgelegt werden muß.

Bild 4-3 zeigt rechts eine Assembler-Befehlsfolge, in der eine in C geschriebene Funktion g(a) aufgerufen wird, die ein Ergebnis vom Typ *int* zurückliefert. Der Aufbau des Stacks bei Bearbeitung dieser Befehlsfolge ist links dargestellt. Bis auf die mit return bezeichnete Speicherstelle entspricht dieser dem Stack-Aufbau nach Bild 4-2. Mit dem ersten Befehl dieses Assemblermoduls wird der für den Rückgabewert benötigte Speicherplatz auf dem Stack reserviert (im Stack durch return bezeichnet). Dies muß vor dem Funktionsaufruf geschehen, da der Speicherplatz nach Bearbeitung der Funktion g noch benötigt wird. Die Parameterübergabe und der Funktionsaufruf werden in der gewohnten Weise durch den zweiten, dritten und vierten Befehl des Assemblermoduls realisiert. Nachdem die Funktion bearbeitet ist, befindet sich in der Speicherstelle return der Rückgabewert der Funktion (in diesem Beispiel 4712). Dieser Wert wird mit dem vorletzten Befehl des Assemblermoduls in die lokale Speicherstelle a kopiert. Schließlich wird mit dem letzten Befehl des Assemblermoduls der für den Rückgabewert belegte Speicherplatz auf dem Stack wieder freigegeben.

Ergebnisse über den Speicher zurückzugeben, hat den Vorteil, daß beliebige Datentypen verwendet werden können, jedoch den Nachteil, daß Speicherzugriffe zeitaufwendig sind. Viele Übersetzer übergeben daher einfache Datentypen, also *char*, *int*, *long* usw., über prozessorinterne Register und nur zusammengesetzte Datentypen, wie z.B. Strukturen, über den Speicher. Für die Ergebnisrückgabe

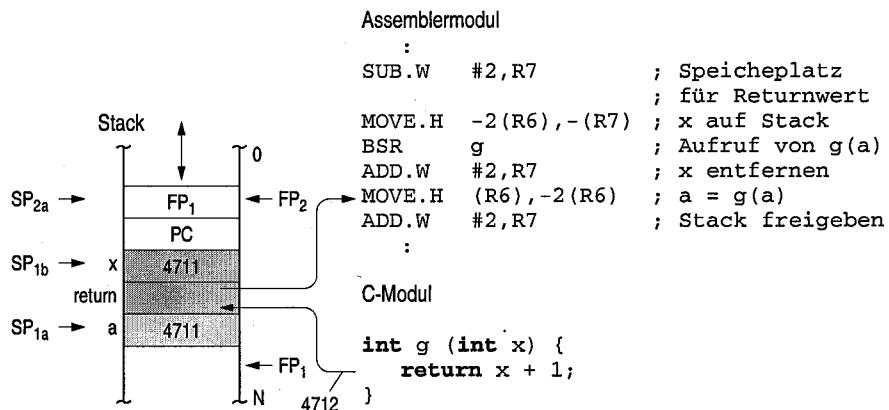


Bild 4-3. Aufruf einer C-Funktion und Rückgabe eines Ergebnisses an das aufrufende Assemblerprogramm über den Stack. Der Stackpointer SP ist gleich dem Inhalt von R7, der Framepointer FP gleich dem Inhalt von R6

einfacher Datentypen werden gewöhnlich von vornherein ein oder zwei Register reserviert. Das zweite Register wird dabei benötigt, um Datentypen zurückzugeben, die zu breit sind, um in einem einzigen Register gespeichert werden zu können (z.B. Rückgabe eines *long*-Werts über 16-Bit-Register). In der Funktion muß dafür gesorgt werden, daß Änderungen an den Inhalten dieser Register für das aufrufende Programm sichtbar sind, d.h., die für die Ergebnisrückgabe reservierten Register dürfen nicht zu Beginn der Funktion auf dem Stack gesichert und am Ende der Funktion wiederhergestellt werden. Nach dem Funktionsaufruf können die entsprechenden Registerinhalte direkt verarbeitet werden, indem auf die Register zugegriffen wird. Auf ein Programmbeispiel dieser einfachen Technik wird verzichtet.

Es sei noch einmal darauf hingewiesen, daß die hier beschriebenen Techniken stark vom verwendeten Übersetzer abhängen. Im konkreten Fall muß also jeweils geklärt werden, wie die Parameterübergabe, der Funktionsaufruf und die Ergebnisrückgabe tatsächlich realisiert sind.

4.3.3 Startup-Code

Zwischen der Aufforderung zum Start eines in C geschriebenen Programms und der Bearbeitung des ersten Befehls aus dem Hauptprogramm *main* müssen zahlreiche Aufgaben verrichtet werden. So müssen zunächst die Speicherbereiche für das Maschinenprogramm, für den benötigten Stack und ggf. für globale Variablen reserviert werden. Anschließend muß das Maschinenprogramm in den entsprechenden Speicherbereich geladen werden, wobei die darin enthaltenen Adressbezüge an die realen Verhältnisse angepaßt werden. Der Stack muß initialisiert

werden, was bedeutet, daß der Stackpointer auf die oberste Adresse des dafür reservierten Speichers gesetzt wird und ggf. die als Kommandozeile übergebenen Argumente als Parameter für das Hauptprogramm auf dem Stack abgelegt werden. Außerdem sind die globalen Variablen entweder mit Null oder mit den im C-Programm vorgegebenen Werten zu initialisieren. Möglicherweise müssen noch der Freispeicher initialisiert und andere übersetzer- und betriebssystemspezifische Aufgaben bearbeitet werden, bis schließlich das Hauptprogramm *main* als Funktion aufgerufen werden kann. Diese Initialisierungsaufgaben werden teilweise vom Betriebssystem und teilweise vom sog. *Startup-Code* bearbeitet.

In ähnlicher Weise werden am Ende eines Programmlaufs Aufräumarbeiten durchgeführt, zuerst vom Startup-Code und anschließend vom Betriebssystem. Hierzu gehört z.B., daß der Rückgabewert des Hauptprogramms *main* an das Betriebssystem weitergereicht und der gesamte reservierte Speicher freigegeben wird (also die Speicherbereiche für das Maschinenprogramm, den Stack und die globalen Daten sowie die dynamisch im Freispeicher reservierten Bereiche). Welche der hier aufgezählten Aufgaben vom Betriebssystem und welche vom Startup-Code bearbeitet werden, ist stark vom verwendeten Betriebssystem abhängig. Soll jedoch, wie hier vorausgesetzt, kein Betriebssystem verwendet werden, so müssen alle beschriebenen Aufgaben im Startup-Code bearbeitet werden. Dies ist für den Programmierer dann ohne Bedeutung, wenn ein für die maschinennahe Programmierung spezialisierter Übersetzer verwendet wird, da hier die jeweiligen Hersteller gewöhnlich den erforderlichen Startup-Code mit dem Übersetzer ausliefern. Er wird dem Programm i.allg. beim Binden (Linken) hinzugefügt.

Wird jedoch, wie in 4.3.2 beschrieben, ein herkömmlicher nicht spezialisierter Übersetzer verwendet, so muß der erforderliche Startup-Code vom Benutzer geschrieben werden. Dabei müssen die oben beschriebenen Aufgaben nicht grundsätzlich bearbeitet werden, da maschinennahe Programme oft Bedingungen vorfinden, die sich von denen unterscheiden, die in herkömmlichen aus einem Betriebssystem heraus gestarteten Programmen vorherrschen. So werden die Adressbereiche für den Maschinencode und für die globalen Daten gewöhnlich bereits zur Entwurfszeit eines Programms festgelegt. Es ist also zur Laufzeit nicht erforderlich, die entsprechenden Speicherbereiche zu reservieren und die Adressbezüge aufzulösen. Allerdings setzt dies voraus, daß mit Hilfe des Binders (Linkers) oder eines anderen Zusatzprogramms das vorzeitige Auflösen der Adressbezüge im Maschinencode vorgenommen werden kann (siehe auch Abschnitt 4.4).

Des weiteren werden maschinennahe Programme meist direkt nach dem Reset gestartet, so daß eine Bearbeitung der Kommandozeile nicht notwendig ist. Auch müssen die normalerweise notwendigen Aufräumarbeiten nicht durchgeführt werden, da Programme bei Nichtvorhandensein eines Betriebssystems nicht

explizit beendet werden dürfen, um nicht den „Absturz“ des Systems herbeizuführen (siehe 4.2.3). Als Aufgaben für den Startup-Code verbleiben also nur das Initialisieren der verschiedenen vom übersetzten Programm benötigten Register und des globalen Speicherbereichs sowie das Starten des Programms. Die Initialisierung der Register beinhaltet auch die Initialisierung des Stackpointers, der auf einen freien Speicherbereich gesetzt werden muß. Der Benutzer muß also zur Entwurfszeit nicht nur festlegen, welche Adressbereiche das Programm und die globalen Variablen verwenden sollen, sondern auch berücksichtigen, daß ein Teil des Speichers für den Stack benötigt wird.

4.4 C-Programmierung mit Betriebssystemunterstützung

In den vorangehenden Abschnitten wurden maschinennahe Programme betrachtet, die z.B. in kleinen, für die Bearbeitung einfacher Aufgaben vorgesehenen Systeme eingesetzt werden und zu ihrer Ausführung kein Betriebssystem benötigen. Die entsprechenden Programme besitzen oft jedoch eine geringe Portabilität, da Besonderheiten von Maschinen explizit genutzt werden, und zwar über spezielle Sprachkonstrukte oder über Funktionen eines speziellen Maschinenmoduls. In kleinen Systemen kann die geringe Portabilität toleriert werden, weil der Aufwand beim Wechsel des Übersetzers oder Prozessors durch den geringen Umfang des Programms beschränkt ist. Um jedoch die Portabilität eines umfangreichen Projekts zu gewährleisten, muß von den individuellen Unterschieden verschiedener Maschinen, Prozessoren oder Übersetzer abstrahiert werden, was den Einsatz eines Betriebssystems erfordert. Hinzu kommt, daß in umfangreichen Projekten oft zusätzliche Anforderungen existieren, die ohne Betriebssystem nur durch aufwendige Programmierung erfüllt werden können. So müssen z.B. in komplexen Steuerungssystemen oft zahlreiche Aufgaben gleichzeitig bearbeitet werden, was mit einem Multitasking-Betriebssystem leicht zu realisieren ist.

Dieser Abschnitt zeigt, wie maschinennahe Aufgaben unter Mithilfe eines Betriebssystems gelöst werden können. Zunächst wird der Einsatz kommerzieller bzw. frei verfügbarer Betriebssysteme behandelt. Danach und zum Abschluß dieses Abschnitts werden dann einige Details erläutert, die den Entwurf eines eigenen, minimalen Betriebssystems ermöglichen.

4.4.1 Kommerzielle und frei verfügbare Betriebssysteme

Zur Unterstützung einfacher wie komplexer Systeme existieren zahlreiche *Betriebssysteme*, die teils für Prozessoren oder Prozessorfamilien eines Herstellers, teils jedoch auch für unterschiedliche Prozessoren verschiedener Hersteller angeboten werden. Meist handelt es sich hierbei um kleine *echtzeitfähige Multitasking-Betriebssysteme* mit einem Codeumfang von 8 bis 256 Kbyte, die somit

vollständig in einem EPROM untergebracht werden können. Die verschiedenen Hersteller und Anbieter stellen normalerweise auch einen auf das jeweilige Betriebssystem abgestimmten C oder C++ Übersetzer zur Verfügung sowie Bibliotheken, in denen nichtstandardisierte Funktionen zur Steuerung und Kommunikation mit dem Betriebssystem enthalten sind. Die nichtstandardisierten Funktionen begrenzen in der Realität die Portabilität, da zwar ggf. der Prozessor gewechselt werden kann, nicht jedoch das Betriebssystem. Umgekehrt werden durch die Wahl des Betriebssystems die potentiell verwendbaren Prozessoren eingeschränkt, weil es kein Betriebssystem gibt, das alle Prozessoren unterstützt.

Ein Betriebssystem sollte, wie zu Beginn dieses Kapitels beschrieben, möglichst vollständig von den Komponenten einer Maschine abstrahieren. Die in C geschriebenen Programme greifen dabei ausschließlich über das Betriebssystem auf die Komponenten einer Maschine zu. In diesem Sinne beschränkt sich die maschinennahe Programmierung eines solchen Systems darauf, das Betriebssystem an die jeweilige Maschine anzupassen, was einerseits durch eine *parametrisierte Konfiguration* und andererseits durch sog. *Gerätetreiber (device driver)* geschieht. Bei der parametrisierten Konfiguration werden die vom Betriebssystem zu berücksichtigenden Randbedingungen durch unterschiedliche Parameter beschrieben. So kann z.B. die Start- und die Endadresse des Datenspeichers einer Maschine in zwei Konfigurationsparametern gespeichert werden. Dabei wird vorausgesetzt, daß für den Datenspeicher ein einzelner, zusammenhängender Adressbereich verwendet wird. Ist dies nicht der Fall, werden weitere Parameter benötigt. Die Flexibilität ist hier abhängig von der Anzahl der vorgesehenen Parameter.

Die Konfiguration über Parameter ist nur für die vom Betriebssystem unterstützten Komponenten, wie z.B. den Prozessor und den Speicher möglich, da nur für solche Komponenten die notwendigen Parameter unabhängig vom verwendeten System definierbar sind. Für ein nicht vom Betriebssystem unterstütztes Gerät ist dies jedoch nicht möglich, weshalb Geräte normalerweise über die bereits erwähnten Gerätetreiber angesteuert werden. Ein Gerätetreiber ist ein Modul mit einer fest vorgegebenen Schnittstelle zum Betriebssystem, über die in einheitlicher Weise auf unterschiedliche Geräte zugegriffen werden kann. Ein Gerätetreiber abstrahiert somit von einem physikalischen Gerät. Die Funktionen der Gerätetreiber werden meist in einem privilegierten Modus ausgeführt und können gewöhnlich nur von einem privilegierten Programm aus aufgerufen werden. Dies ist erforderlich, um zu gewährleisten, daß nicht von jeder Task aus jedes Gerät angesteuert werden kann. So darf es z.B. nicht möglich sein, daß die Steuerung einer Raumbeleuchtung aufgrund eines Fehlers ein Notaus in einer fatalen Fehlersituation unmöglich macht.

Ein Gerätetreiber muß sehr unterschiedliche Anforderungen erfüllen. Einerseits muß er vom konkreten Gerät abstrahieren, andererseits soll er die Möglichkeiten des realen Gerätes für den Benutzer erschließen und zwar, ohne dabei wesentlich

langsamer zu arbeiten, als würde auf die Register des Gerätes ohne Gerätetreiber zugegriffen. Dieser Konflikt wird in vielen Betriebssystemen dadurch gelöst, daß für bestimmte Gerätetypen spezialisierte Gerätetreiber verwendet werden. Sehr verbreitet ist z.B. die Unterteilung in sog. *zeichenorientierte Gerätetreiber* – für serielle Schnittstellen, Drucker usw. – und *blockorientierte Gerätetreiber* – für Festplatten, Floppy-Disk-Laufwerke usw. Je nach Betriebssystem kommen weitere Spezialisierungen hinzu. So ist z.B. die Anbindung eines Netzwerks in Linux über sog. *Netzwerk-Gerätetreiber* (*network interfaces*) realisiert [Rubini 1998].

OS9/68000. Die Art und Weise, in der mit Geräten interagiert werden kann, ist vom verwendeten Betriebssystem abhängig. So ist z.B. in Gerätetreibern für das kommerzielle Echtzeitbetriebssystem OS-9/68000 ein immer gleichbleibender Satz von Funktionen, mit festgelegten und vom Gerätetyp unabhängigen Schnittstellen, bereitzustellen, wobei nicht in jedem Gerätetreiber alle Funktionen implementiert sein müssen [Microware 1987]. Für ein reines Ausgabegerät ist es z.B. nicht erforderlich, eine Eingabefunktion zu realisieren. Die in einem Gerätetreiber enthaltenen Funktionen sind i.allg. vom Gerätetyp abhängig implementiert und können eine ebenfalls vom Gerätetyp abhängige Funktionalität besitzen. Wird eine nicht in einem Gerätetreiber realisierte Funktionalität angefordert, so kann dies z.B. mit einem Fehlercode beantwortet werden. Nachfolgend sind die Funktionen eines *Gerätetreibers* für das Betriebssystem OS9/68000 beschrieben:

- *Init:* Diese Funktion wird vom Betriebssystem aufgerufen, um ein Gerät in das System einzubinden. Sie wird zur Initialisierung des Gerätes und des Gerätetreibers benötigt. Soll z.B. von einem Gerät ein Interrupt ausgelöst werden, so wird in init die Startadresse der Interruptroutine in die Vektortabelle eingetragen und der Interrupt geräteseitig freigeschaltet.
- *Read:* Mit dieser Funktion kann ein Zeichen oder Block von einem Gerät empfangen werden. Der Zugriff auf eine Festplatte erfolgt hierin z.B., indem zunächst die notwendigen Kommandos an die Festplatte geschickt werden und anschließend auf den Empfang eines Sektors gewartet wird. Erst danach terminiert die Funktion. Die möglicherweise entstehende Wartezeit kann durch einen Prozeßwechsel überbrückt werden.
- *Write:* Write ist das Gegenstück zu read. Die Funktion wird verwendet, um ein Zeichen oder Block an ein Gerät zu übertragen.
- *Getstat:* Mit dieser Funktion kann der gerätespezifische Status eines Gerätes ermittelt werden. Der gerätespezifische Status wird bei der Programmierung eines Gerätetreibers definiert und ist auf das Gerät individuell abgestimmt. So besitzt z.B. eine serielle Schnittstelle möglicherweise den Status „Baud-Rate“, der für eine Festplatte keine Bedeutung hat.
- *Setstat:* Diese Funktion wird benötigt, um den gerätespezifischen Status eines Gerätes zu verändern. So kann hier z.B. die Baud-Rate, mit der eine serielle Schnittstelle arbeiten soll, eingestellt werden.

- *Terminate*: Terminate ist das Gegenstück zu init und wird aufgerufen, wenn ein Gerätetreiber nicht mehr benötigt wird und aus dem Betriebssystem entfernt werden soll. Mit terminate werden normalerweise die durch die Funktion init bewirkten Veränderungen am System rückgängig gemacht. So darf ein Baustein z.B. keinen Interrupt mehr auslösen, wenn der zugehörige Gerätetreiber entfernt wurde.
- *Error*: Diese Funktion wird aufgerufen, wenn es während der Bearbeitung einer Gerätetreiberfunktion zu einem Fehler gekommen ist. Sie kann z.B. verwendet werden, um ein Gerät aus einem Fehlerzustand herauszuführen.

Damit die einzelnen Funktionen lokalisiert werden können, müssen dem Betriebssystem deren Aufrufadressen mitgeteilt werden. Dies geschieht bei OS-9/68000 durch eine sog. *Entry-Point-Offset-Table*, in der die jeweils 32-Bit-breiten Aufrufadressen in genau der Reihenfolge, in der sie oben beschrieben wurden, eingetragen sind. Die Tabelle befindet sich an einer fest definierten Position innerhalb des Gerätetreibers, so daß das Betriebssystem die benötigten Aufrufadressen ermitteln kann. Für jedes Gerät kann nun ein eigener Gerätetreiber programmiert und in das System eingebunden werden. Dies ist jedoch wenig speichereffizient, da mit mehreren Geräten gleichen Typs auch mehrere Gerätetreiber geladen werden müßten und das, obwohl die einzelnen Geräte sich in der Programmierung nur in wenigen Details unterscheiden, wie z.B. der Speicheradresse des Gerätes oder der Nummer des verwendeten Interrupts. Unter OS-9/68000 wird daher einem Gerätetreiber eine sog. *Gerätebeschreibung (device descriptor)* zur Seite gestellt, in der die gerätespezifischen Details codiert sind. Um z.B. zwei Drucker gleichen Typs verwenden zu können, werden ein Gerätetreiber und zwei Gerätebeschreibungen benötigt.

Die maschinennahe Programmierung bei Verwendung eines Betriebssystems beschränkt sich im wesentlichen auf die Programmierung von Gerätetreibern für Komponenten, die vom Betriebssystem nicht direkt unterstützt werden. Sind die Gerätetreiber erst einmal implementiert, werden weitere Aufgaben auf einer abstrakteren Ebene gelöst. In den meisten höheren Betriebssystemen wird hierzu auf Geräte genau wie auf Dateien zugegriffen. So kann mit den folgenden Anweisungen unter OS-9/68000 ein Text auf dem Drucker ausgegeben werden.

```
FILE *fp = fopen ("/p", "w"); // Drucker zum Schreiben öffnen
fputs ("Hallo World\n", fp); // Zeichenfolge ausgeben
fclose (fp); // Drucker schließen
```

Der mit der Anweisung fopen angegebene Pfad /p dient zur Identifikation der zu verwendenden Gerätebeschreibung, in der, wie beschrieben, neben der Basisadresse und der zu verwendenden Interruptnummer auch der Name des zu verwendenden Gerätetreibers codiert ist.

µClinux. Seit einiger Zeit wird zur Gerätesteuerung auch das frei verfügbare Betriebssystem µClinux verwendet – eine Adaption des im Bereich der Arbeits-

platzrechner verbreiteten Betriebssystems Linux, jedoch auf die Belange einfacher Systeme zugeschnitten (genaueres ist z.B. unter www.uclinux.org zu finden). Der Betriebssystemkern von μ Clinux unterscheidet sich von dem in Linux vor allem dadurch, daß ein verändertes Speichermanagement realisiert ist, das ohne eine MMU auskommt. Trotz der Änderungen im Betriebssystemkern ist die Systemschnittstelle jedoch weitgehend mit Linux identisch. Dies hat die angenehme Konsequenz, daß viele unter Linux lauffähige Anwendungen auch auf μ Clinux übertragbar sind. Allerdings erfordert das veränderte Speichermanagement eine Neuübersetzung der Programme, da unter μ Clinux nur *verschiebbarer* (relokatibler) *Code* ausgeführt werden kann (siehe auch 3.1.3). Als Entwicklungswerkzeug kann z.B. der GNU C++-Übersetzer verwendet werden, der, gesteuert durch einen Aufrufparameter in der Lage ist, verschiebbaren Code zu erzeugen.

Die Treiberprogrammierung unter μ Clinux ist ebenfalls von Linux übernommen worden, so daß abgesehen von den ohnehin existierenden Inkompatibilitäten der unterschiedlichen Linux-Versionen, frei vorhandene Gerätetreiber, die für Linux geschrieben wurden, auch unter μ Clinux verwendet werden können. Der Aufbau eines Gerätetreibers unter μ Clinux ist mit dem von OS9 vergleichbar, wobei davon abweichend Gerätetreiber unter μ Clinux entweder direkt in den Betriebssystemkern eingebunden (was statisch mit der Übersetzung des Kerns erfolgt) oder in sog. *Module* eingebettet werden. Die Verwendung von Modulen hat den Vorteil, daß Gerätetreiber dem laufenden Betriebssystem dynamisch hinzugefügt werden können. Jedoch sind Module etwas komplizierter zu programmieren und zu benutzen. Bevor eine Gerätetreiberfunktion verwendet werden darf, muß das verantwortliche Modul installiert werden, was z.B. mit dem Kommando insmod geschehen kann. Dabei wird automatisch die Modulfunktion init_module aufgerufen, in der dann die eigentlichen Gerätetreiberfunktionen in das System eingebunden werden können. Zur Deinstallation eines Moduls kann – ganz ähnlich – das Kommando rmmod verwendet werden, wobei in der daraufhin automatisch aufgerufenen Modulfunktion cleanup_module die in init_module vorgenommenen Systemänderungen rückgängig gemacht werden müssen.

Die wichtigsten Funktionen eines *Gerätetreibers* sind:

- *Open*: Mit Aufruf dieser Funktion können das Gerät und der Gerätetreiber initialisiert werden. Das System stellt sicher, daß keine der nachfolgend beschriebenen Funktionen aufgerufen werden, bevor das Gerät mit open geöffnet wurde. Open ist vergleichbar mit der Treiberfunktion init des Betriebssystems OS9.
- *Read*: Sie ist eine Funktion für Lesezugriffe auf die durch das Gerät repräsentierten Daten. Welche Daten jeweils durch ein Gerät repräsentiert werden, ist vom Gerätetreiber abhängig. So wird diese Funktion in einem Gerätetreiber für serielle Schnittstellen normalerweise so implementiert sein, daß die emp-

fangenen Zeichen an den Aufrufer zurückgegeben werden. Die Wirkungs-gleiche Funktion in einem OS9-Gerätetreiber heißt ebenfalls `read`.

- *Write*: Die Funktion `write` bildet wie bei OS9 das Gegenstück zu `read` und wird verwendet, um ein Zeichen oder Block an ein Gerät zu übertragen.
- *Lseek*: Mit `lseek` kann die Lese- bzw. Schreibposition definiert werden. `Lseek` ist z.B. für Festplattenzugriffe definiert und gestattet es, Dateien wahlfrei zu schreiben oder zu lesen – eine Funktionalität die z.B. von Datenbank-Managementsystemen benötigt wird. Unter OS9 kann die Funktionalität mit der Gerätetreiberfunktion `setstat` implementiert werden.
- *Ioctl*: Die Funktion `ioctl` dient dazu, den Status eines Geräts oder eines Gerätetreibers abzufragen oder zu verändern. Sie faßt somit die Treiberfunktionen `getstat` und `setstat` des Betriebssystems OS9 zusammen.
- *Release*: Diese Funktion bildet das Gegenstück zu `open`. Sie wird aufgerufen, um ein Gerät zu schließen. Da in μ Clinux mehrere Benutzer mit demselben Gerätetreiber arbeiten können, muß ein Aufruf von `release` nicht dazu führen, daß das Gerät aus dem System entfernt wird. Der Programmierer eines Gerätetreibers hat dafür Sorge zu tragen, daß dies erst dann geschieht, wenn der Gerätetreiber nicht mehr benutzt wird (gewöhnlich unter Zuhilfenahme eines Referenzzählers). `Release` ist in seiner Wirkungsweise mit der Funktion `terminate` eines OS9-Gerätetreibers vergleichbar.

Die Liste der Gerätetreiberfunktionen von μ Clinux ist bei weitem nicht vollständig. Es existieren zusätzliche Funktionen, mit denen ein nichtblockierender Zugriff auf Geräte realisiert werden kann (`select`), mit denen Gerät und Gerätetreiber synchronisiert werden können (`fsync`), mit denen Medienwechsel in Wechsellaufwerken erkannt werden können (`check_media_change`) usw. Für alle Funktionen eines Gerätetreibers gilt, daß sie nicht implementiert sein müssen. Ist eine Funktion nicht im Gerätetreiber definiert, so reagiert das System bei einem Aufruf in einer vorgegebenen Art und Weise. In vielen Fällen wird z.B. eine Fehlermeldung generiert.

Gerätebeschreibungen existieren in μ Clinux nicht. Trotzdem ist es möglich, durch einen Gerätetreiber mehrere gleichgeartete Geräte zu bedienen. Um innerhalb einer Gerätetreiberfunktion zu ermitteln, auf welches Gerät sich der Aufruf bezieht, wird dem Gerätetreiber die sog. *Major- und Minor-Nummer* übergeben. Die Major-Nummer ist normalerweise eindeutig einem Gerät zugeordnet. Sie wird vom System ausgewertet, um das für das Gerät zuständige Modul zu identifizieren. Die Minor-Nummer kann anschließend in der aufgerufenen Gerätetreiberfunktion verwendet werden, um die Wirkungsweise des Gerätetreibers zu beeinflussen. So ist es z.B. möglich, zwei serielle Schnittstellen gleichen Typs über die Minor-Nummer zu unterscheiden. Im Vergleich zu den Gerätebeschreibungen von OS9 ist diese Technik, Geräte über Major- und Minor-Nummer zu unterscheiden, insoweit weniger leistungsfähig, als daß der Programmierer eines

μ Clinix-Gerätetreibers die Funktionalität einer Gerätebeschreibung, wie sie unter OS9 existiert, selbst programmieren muß.

Anmerkung. Es ist möglich, mit dem anfangs genannten Kommando insmod auch mehrere Module gleichen Typs in das System einzubinden, wobei jedes Modul bei der Installation die Basisadresse des zu bedienenden Gerätes übergeben bekommt. Eine Gerätebeschreibung ist hierbei überflüssig. Allerdings werden die Module in einem solchen Fall redundant im Speicher gehalten, was bei Arbeitsplatzrechnern tolerabel ist, was jedoch nachteilig bei den hier vorrangig betrachteten kleinen Systemen ist.

Die Verwendung von μ Clinix in kleinen Systemen hat Vorteile. Das Betriebssystem ist für zahlreiche Prozessoren portiert, viele Geräte werden von Haus aus unterstützt oder die Gerätetreiber sind frei im Netz verfügbar, es existieren viele nützliche Applikationen, es werden Entwicklungswerkzeuge angeboten und – nicht zuletzt – μ Clinix ist kostenfrei. Es gibt allerdings auch einen entscheidenden Nachteil: μ Clinix ist nicht echtzeitfähig. In vielen Anwendungen spielt das keine Rolle. Dort wo es eine Rolle spielt, muß entweder ein anderes Betriebssystem verwendet werden, oder μ Clinix muß um die benötigte Funktionalität erweitert werden. Letzteres ist z.B. mit RTLinux möglich, wobei das eigentliche Betriebssystem μ Clinix als Prozeß in einem darunterliegenden echtzeitfähigen Betriebssystemkern (RRTLinux) ausgeführt wird [Dumschat 2001]. Ein Prozeß, der Echtzeitanforderungen zu erfüllen hat, kann hierbei nicht unter μ Clinix, sondern muß unter RRTLinux ausgeführt werden.

4.4.2 Entwurf eines einfachen Betriebssystems

Der Einsatz eines Betriebssystems ist bezüglich Portabilität und Arbeitseffizienz von großem Vorteil. Nachteilig ist jedoch, daß kommerzielle Betriebssysteme oft einen großen Speicherbedarf besitzen, nicht auf allen Prozessoren oder Mikrocontrollern lauffähig sind und erhebliche Kosten verursachen können. Statt eines kommerziellen Betriebssystems kann aber auch ein auf eine gegebene Aufgabenstellung spezialisiertes Betriebssystem neu realisiert werden. Dies ist sinnvoll, wenn die Aufgabenstellung entweder so einfach ist, daß auch das Betriebssystem sehr einfach gehalten werden kann, oder wenn die Aufgabenstellung so umfangreich ist, daß die Kosten für die Entwicklung eines leistungsstarken Betriebssystems geringer als die Kosten für den Kauf eines normalerweise in Lizenz vertriebenen Betriebssystems sind. Im folgenden wird skizziert, wie ein einfaches Betriebssystem realisiert werden kann.

Ein Betriebssystem hat zum einen die Aufgabe, die sog. *Betriebsmittel* zu verwalten. Dazu gehören die Ein-/Auszabegeräte, der Hauptspeicher sowie der Prozessor, aber auch Daten und Programme. Zum andern hat es die Aufgabe, Dienste anzubieten, mit deren Hilfe Benutzer mit dem Betriebssystem interagieren können (siehe z.B. [Bengel 1990]). In kleinen Systemen ist allerdings eine Interaktion des Betriebssystems mit dem Benutzer meist nicht oder nur sehr elementar

erforderlich, so daß vom Betriebssystem nur die Betriebsmittel verwaltet werden müssen. Soll auf einem sehr kleinen System nur ein einzelnes Programm ausgeführt werden, können die Aufgaben eines Betriebssystems weiter reduziert werden: So sind z.B. Schutzmechanismen, die den Zugriff auf den Hauptspeicher oder auf Geräte regeln, nicht erforderlich. Das heißt, der Prozessor kann dem Programm fest zugeordnet werden, und die Adressen von Variablen können bereits zur Übersetzungszeit festgelegt werden usw. Ein sehr einfaches Betriebssystem, hier als *Betriebsssoftware* bezeichnet, muß somit die Verwaltung der Geräte und des freien Hauptspeichers, des sog. *Freispeichers*, übernehmen, wobei letzteres nur dann erforderlich ist, wenn das Benutzerprogramm mit dynamisch reservierbaren Speicherbereichen arbeitet.

Geräteverwaltung. Zur Verwaltung der Geräte können in einer Betriebsssoftware Gerätetreiber verwendet werden, wie sie auch bei leistungsstarken Betriebssystemen üblich sind, jedoch mit einfacherer Verwaltung der Gerätetreiber selbst. Dabei wird jedes Gerät über einen Satz von Funktionen, wie sie in 4.4.1 beschrieben wurden, angesteuert, wobei die Schnittstellen der einzelnen Funktionen vereinheitlicht und somit geräteunabhängig sind. Der Aufruf dieser Funktionen geschieht z.B. über *Trap-Befehle* (2.1.4, 2.2.4), was den Vorteil hat, daß die Startadressen der einzelnen Funktionen im Benutzerprogramm nicht benötigt werden und somit Änderungen der Betriebssoftware keine Änderungen im Benutzerprogramm nach sich ziehen. Grundsätzlich ist es möglich, jedes Gerät separat über einen oder mehrere individuelle Trap-Befehle zu steuern, sofern der Prozessor mehr als einen Trap-Befehl zur Verfügung stellt. Flexible und auch leichter portierbar ist es jedoch, für alle Geräte und Funktionen nur einen einzigen Trap-Befehl zu verwenden und die Geräte- bzw. Funktionsauswahl über Parameter zu realisieren.

Die Funktionsweise einer einfachen Geräteverwaltung soll an einem einfachen Beispiel gezeigt werden. Hierzu ist in Bild 4-4 schematisch dargestellt, wie ein einzelnes Zeichen – hier ein Ausrufungszeichen – auf einem Drucker ausgegeben wird. Zunächst werden im Benutzerprogramm die an die Betriebssoftware zu übergebenden Argumente gesetzt. Im Register R0 wird der zu verwendende Gerätetreiber und in R1 die aufzurufende Gerätetreiberfunktion codiert, wobei zur besseren Verständlichkeit die symbolischen Konstanten PRINTER und WRITE verwendet werden. Das auszugebende Zeichen wird in R2 übergeben. Durch den Trap-Befehl wird in eine Bearbeitungsroutine verzweigt, die Teil der Betriebsssoftware ist. Sie wird auch als Systemfunktion bezeichnet. In ihr werden die beiden Parameter R0 und R1 verwendet, um die Startadresse der entsprechenden Gerätetreiberfunktion aus einer Tabelle auszulesen. Diese Tabelle, hier als *Gerätetabelle* bezeichnet, ist bei Erweiterungen des Systems oder bei Veränderungen einzelner Gerätetreiber leicht änderbar.

Die so ermittelte Startadresse wird schließlich verwendet, um die eigentliche Gerätetreiberfunktion WRITE aufzurufen. Dabei werden dieser Funktion als

Parameter die gleichen Werte übergeben, die vor Ausführung des Trap-Befehls in die Register geschrieben wurden. Insbesondere befindet sich also das von der Gerätetreiberfunktion WRITE auszugebende Zeichen im Register R2. Angemerkt sei noch, daß für eine Ausgabe auf dem Bildschirm im Benutzerprogramm lediglich die Konstante PRINTER durch CONSOLE ersetzt werden muß.

Benutzerprogramm

```
:
MOVE.B #'!',R2      ; Ausrufungszeichen ausgeben
MOVE.W #PRINTER,R0 ; Auswahl des Geräts
MOVE.W #WRITE,R1    ; Write-Funktion aufrufen
TRAP               ; Systemaufruf
```

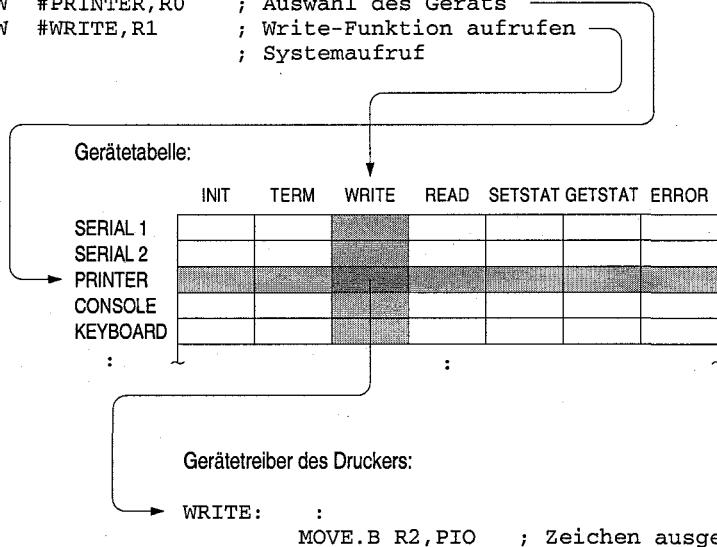


Bild 4-4. Zeichenausgabe auf einem Drucker mit Hilfe eines Trap-Befehls

Die in Bild 4-4 dargestellte Assembler-Befehlsfolge könnte zur Verwendung in C-Programmen als *Bibliotheksfunktion* zur Verfügung gestellt werden, wie dies bereits in 4.3.2 beschrieben wurde. Um die obige Ausgabe eines Zeichens auf einem Drucker aus einem C-Programm heraus durchzuführen, kann z.B. folgende Befehlsfolge verwendet werden:

```
#include <sys.h>          // Bibliothek mit Systemfunktionen
:
system (PRINTER, WRITE, '!'); // Aufruf der Gerätetreiber-
                             // funktion WRITE des Geräte-
                             // treibers PRINTER
```

Obwohl aufwendiger und dadurch weniger portabel, sollten besser jedoch unterschiedliche Gerätetreiberfunktionen auch über unterschiedliche Bibliotheksfunktionen aufgerufen werden. Dadurch werden die Programme nicht nur einfacher lesbar, sondern die Parametertypen und die Parameteranzahl können an die aufzu-

rufende Funktion angepaßt werden. Die Ausgabe eines Zeichens auf einem Drucker würde dabei folgendermaßen programmiert:

```
WRITE (PRINTER, '!');      // Aufruf der Gerätetreiberfunktion .  
                           // WRITE des Gerätetreibers PRINTER
```

Der hier beschriebene Mechanismus zum Aufruf von Systemfunktionen kann natürlich auch dazu verwendet werden, Funktionen aufzurufen, die nicht zur Steuerung von Geräten dienen, sondern andere Aufgaben lösen. So kann z.B. eine Systemfunktion zur Ausgabe einer nullterminierten Zeichenfolge (also einer Zeichenfolge, deren Ende durch eine 0 angezeigt wird) realisiert werden, die über den Trap-Mechanismus aus einem Benutzerprogramm heraus aufzurufen ist.

Freispeicherverwaltung. Die Verwaltung des freien Hauptspeichers ist eine weitere typische Aufgabe eines Betriebssystems. Sie ist insbesondere dann erforderlich, wenn Speicherbereiche zur Laufzeit eines Programms, also dynamisch belegt und auch wieder freigegeben werden sollen. Die sog. *Freispeicherverwaltung* stellt hierzu Systemfunktionen zur Verfügung, die in C-Programmen u.a. über die Bibliotheksfunktionen malloc und free aufgerufen werden. Wie eine Freispeicherverwaltung für eine einfache Betriebsssoftware realisiert werden kann, ist im folgenden angedeutet. Der freie Speicher wird hierbei über eine Liste, die *Freispeicherliste*, verwaltet, in der alle freien Speicherblöcke eingetragen sind. Wird vom Benutzerprogramm ein Speicherbereich angefordert, so wird die Freispeicherliste nach einem passenden Speicherblock durchsucht. Im einfachsten Fall wird dabei der erste passende Speicherblock aus der Freispeicherliste entnommen (*first fit*), als belegt gekennzeichnet und der dem Speicherblock angehftete Speicherbereich an das Benutzerprogramm übergeben. Falls der in der Freispeicherliste gefundene Speicherbereich größer ist als vom Benutzerprogramm angefordert, wird er in zwei Speicherblöcke zerteilt, von denen der nicht benötigte in der Freispeicherliste verbleibt und der andere dem Benutzerprogramm zugeordnet wird.

Bei der Umkehrung dieser Funktion wird ein vom Benutzerprogramm freigegebener Speicherbereich in einem entsprechend gekennzeichneten Speicherblock „verpackt“ und in die Freispeicherliste eingetragen. Der freigegebene Speicherbereich darf anschließend nicht weiter vom Benutzerprogramm verwendet werden. Um zu vermeiden, daß zwei im Hauptspeicher unmittelbar aufeinanderfolgende Speicherblöcke als separate Einträge in der Freispeicherliste erscheinen, muß ggf. geprüft werden, ob im Hauptspeicher unmittelbar vor oder nach dem freizugebenden ein bereits freigegebener Speicherblock existiert. In diesem Fall lassen sich die entsprechenden Speicherblöcke zu einer Einheit verbinden, wobei ein einzelner großer Speicherblock in die Freispeicherliste eingetragen wird. Unter Umständen kann dies jedoch eine Reorganisation der Freispeicherliste nach sich ziehen.

Eine mögliche Organisationsform zur Speicherverwaltung zeigt Bild 4-5. Der Hauptspeicher ist hierbei in Speicherblöcken organisiert, die jeweils aus einem Verwaltungsinformationen enthaltenden Blockkopf und einem frei benutzbaren Blockrumpf bestehen. Der Blockkopf gibt Auskunft über die Größe des entsprechenden Blockrumpfs, die Größe des im Hauptspeicher unmittelbar voranstehenden Blockrumpfs und die Adresse des nächsten freien Speicherblocks. Statt der Adresse kann alternativ auch einer von zwei ausgezeichneten Werten gespeichert sein, um entweder das Ende der Freispeicherliste (z.B. durch einen Zeiger auf sich selbst) oder einen belegten Speicherblock zu kennzeichnen (z.B. NULL-Zeiger). Wie sich diese Verwaltungsinformation nutzen lässt, um Speicher zu reservieren bzw. freizugeben, soll an einem Beispiel erläutert werden. Angenommen, der Speicher ist entsprechend Bild 4-5a aufgeteilt. Um einen Speicherbereich zu belegen, der größer als Blockrumpf 3 und kleiner als Blockrumpf 1 ist, wird zuerst ein Laufzeiger auf den ersten freien Speicherblock der Freispeicherliste gesetzt (hier Speicherblock 3). Über diesen Zeiger wird auf die Verwaltungsinformation von Speicherblock 3 zugegriffen und geprüft, ob der Blockrumpf ausreichend groß ist, um den angeforderten Speicherplatz zur Verfügung stellen zu können. Da dies nach Vorgabe nicht der Fall ist, wird die in Blockkopf 3 (das ist der zu Speicherblock 3 gehörende Blockkopf) gespeicherte Adresse des nächsten freien Speicherblocks in den Laufzeiger kopiert und anschließend über diesen auf Speicherblock 1 zugegriffen.

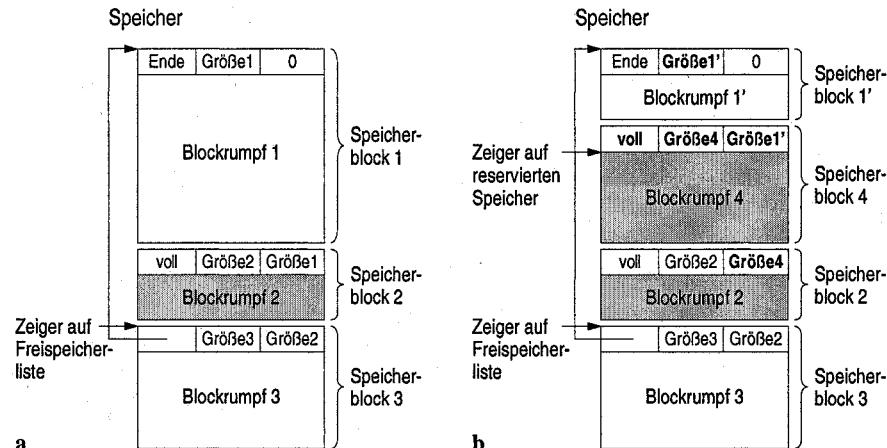


Bild 4-5. Freispeicherverwaltung. a Speicheraufteilung mit einem belegten Speicherblock (Blockrumpf grau unterlegt) und zwei freien Speicherblöcken (nicht unterlegt), b geänderte Speicheraufteilung nach Zuteilung eines Speicherblocks

Dieser Vorgang wiederholt sich, bis entweder das Ende der Freispeicherliste erreicht ist, was hier durch einen entsprechenden Wert im Laufzeiger erkannt wird, oder bis ein Speicherblock gefunden ist, der ausreichend groß ist, um den angeforderten Speicherplatz bereitzustellen zu können. Letzteres ist bei Speicherblock 1

der Fall. Da dieser sogar größer als benötigt ist, wird er, wie in Bild 4-5b dargestellt, in die zwei Teilblöcke zerlegt, und zwar in den Speicherblock 1', der in der Freispeicherliste verbleibt, und den als belegt gekennzeichneten Speicherblock 4. Zur weiteren Verwaltung müssen schließlich noch die in Bild 4-5b fett dargestellten Feldbezeichnungen aktualisiert werden, wobei sich der Blockkopf 2 über die Startadresse von Speicherblock 1 und dessen Größe zugreifen lässt. Der Vorgang wird durch Rückgabe eines Zeigers auf den nunmehr belegten Speicherbereich an das anfordernde Programm beendet. Zu beachten ist, daß der Zeiger auf den Blockrumpf 4 und nicht auf den Speicherblock 4 zurückgegeben wird (im Bild: Zeiger auf reservierten Speicher). Die Verwaltungsinformation kann dennoch adressiert werden, nämlich mit negativer Distanz zu diesem Zeiger.

Um einen Speicherblock wieder freizugeben, reicht es im einfachsten Fall aus, ihn direkt in die Freispeicherliste einzuhängen. Für das Beispiel der Freigabe von Speicherblock 4 in Bild 4-5b heißt das: Es muß nur der Inhalt des Zeigers auf die Freispeicherliste (siehe Bild) in das Adreßfeld im Blockkopf 4 eingetragen und anschließend der Zeiger auf die Freispeicherliste modifiziert werden, und zwar so, daß er auf Speicherblock 4 verweist. Der Zeiger auf die Freispeicherliste zeigt danach also auf Speicherblock 4, das Adreßfeld in Blockkopf 4 auf Speicherblock 3, das Adreßfeld in Blockkopf 3 auf Speicherblock 1', und das Adreßfeld in Blockkopf 1' enthält einen Wert, der das Ende der Freispeicherliste anzeigt. Nachteilig ist, daß der verwaltete Hauptspeicher hierbei in immer kleiner werdende Speicherblöcke zerfällt. Um diese sog. *Fragmentierung* zu vermeiden, müssen freie, im Hauptspeicher benachbarte Speicherblöcke wieder zu einem Speicherblock verschmolzen werden. Falls also in Bild 4-5b der Speicherblock 4 freigegeben wird, so kann er mit Speicherblock 1' zusammengefaßt werden. Dies ergibt dann wieder eine Speicheraufteilung, wie sie in Bild 4-5a dargestellt ist. Um Speicherblock 4 und Speicherblock 1' zusammenfassen zu können, muß jedoch über den Zeiger auf Blockkopf 4 auf den Blockkopf 1' zugegriffen werden können. Dies ist hier möglich, da als Verwaltungsinformation in Speicherblock 4 die Größe von Speicherblock 1' gespeichert ist. Anstatt Speicherblock 4 bei der Freigabe direkt in die Freispeicherliste einzuhängen, wird also die im Blockkopf 1' gespeicherte Blockgröße aktualisiert und so wieder eine Speicheraufteilung wie in Bild 4-5a erreicht.

Es sei dem Leser empfohlen, darüber nachzudenken, wie ein freizugebender Speicherblock mit einem nachfolgenden bzw. gleichzeitig mit einem vorangehenden und nachfolgenden freien Speicherblock verschmolzen werden kann. Der letzte Fall ist der komplizierteste und führt zu der bereits erwähnten Reorganisation der Freispeicherliste.

Multitasking. Wie eingangs schon angedeutet, ist der Einsatz eines Multitasking-Betriebssystems oft vorteilhaft. Dies gilt insbesondere für Steuerungsaufgaben, bei denen in der Regel mehrere, voneinander weitgehend bzw. vollständig unabhängige Teilaufgaben bearbeitet werden müssen. Dabei ist der Einsatz eines ech-

zeitfähigen und zeitscheibengesteuerten, also *präemptiven Betriebssystems* zwar wünschenswert, für kleinere Aufgabenstellungen reicht jedoch meist auch eine Betriebssoftware aus, die ein einfaches sog. *kooperatives Multitasking* ermöglicht. Das kooperative Multitasking unterscheidet sich vom präemptiven Multitasking im wesentlichen in der Art, wie zwischen den gleichzeitig bearbeiteten Prozessen hin- und hergeschaltet wird. Beim kooperativen Multitasking geschieht dies nämlich unter Mithilfe der einzelnen Prozesse, z.B., indem der laufende Prozeß eine bestimmte Systemfunktion aufruft. Beim präemptiven Multitasking wird hingegen ein Prozeßwechsel durch das Betriebssystem zeitgesteuert erzwungen.

Beim kooperativen Multitasking muß eine Systemfunktion vorhanden sein, die bei Aufruf den aktuellen Prozeß unterbricht und einen anderen Prozeß aktiviert. Dabei wird ein sog. *Kontextwechsel* ausgeführt, bei dem zuerst alle Zwischenergebnisse des unterbrochenen Prozesses gesichert und die des zu aktivierenden Prozesses geladen werden. So wird gewährleistet, daß ein unterbrochener Prozeß mit den vor der Unterbrechung ermittelten Zwischenergebnissen später korrekt weiterarbeiten kann. Normalerweise müssen bei einem Kontextwechsel die Inhalte der Prozessorregister und der Speichervariablen ausgewechselt werden. Der Wechsel der Speichervariableninhalte muß in einem einfachen System jedoch nicht explizit realisiert werden, da jeder Prozeß seine Variablen in einem separaten Speicherbereich halten kann, auf die nur dieser Prozeß zugreift. Insbesondere werden dabei die Variablen anderer Prozesse unverändert belassen, so daß das Speichern und spätere Rückladen der entsprechenden Speichervariablen entfallen kann (siehe auch 6.3, Speicherverwaltung mit Hilfe einer MMU).

Wie eine *Prozeßverwaltung* für kooperatives Multitasking arbeiten kann, wird anhand von Bild 4-6 demonstriert. Es wird dabei angenommen, daß sich drei Prozesse P1, P2 und P3 gleichzeitig in Bearbeitung befinden, von denen der Prozeß P2 gerade aktiv ist, d.h. auf dem Prozessor ausgeführt wird. Die Prozesse greifen während ihrer Ausführung nur auf die ihnen zugeordneten Speicherbereiche zu. Daher wird bei Aufruf einer Systemfunktion die Rücksprungadresse auf dem diesem Prozeß zugeordneten Stack abgelegt. Bei einem *Prozeßwechsel* durch eine Systemfunktion werden anschließend auch die Inhalte aller Prozessorregister auf dem Stack gesichert. Dies geschieht in der entsprechenden Systemfunktion, die zum Abschluß des Sicherungsvorgangs nur noch den Stackpointer in der sog. *Prozeßtabelle* speichern muß. Der dabei zu verwendende Eintrag wird über einen Prozeßzeiger adressiert. Nachdem der aktuell ausgeführte Prozeß gesichert wurde, wird von der Systemfunktion der neu zu aktivierende Prozeß geladen. Hierzu wird der Prozeßzeiger auf den nächsten Eintrag in der Prozeßtabelle gesetzt und der dort gespeicherte Stackpointer in den Prozessor geladen. Mit Hilfe des Stackpointers werden als nächstes die auf dem Stack gesicherten Registerinhalte des zu aktivierenden Prozesses in den Prozessor geladen. Der Prozeßwechsel wird schließlich abgeschlossen, indem ein Rücksprungbefehl ausgeführt wird (und damit die Rücksprungadresse PC1 in den PC geladen wird).

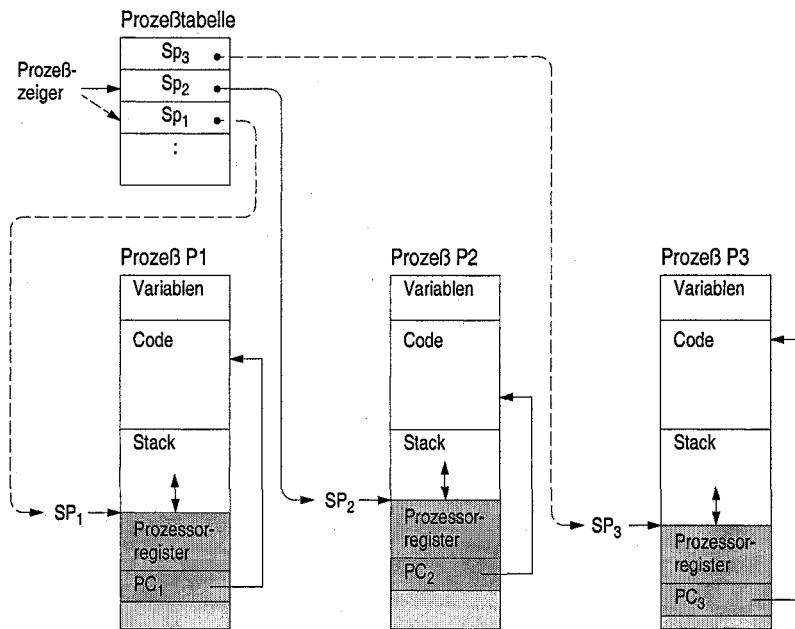


Bild 4-6. Prozeßverwaltung über eine Prozeßtabelle mittels der Stacks der einzelnen Prozesse

Für eine Prozeßverwaltung in der hier beschriebenen Weise werden zumindest zwei weitere Systemfunktionen benötigt, eine zum Starten und eine zum Beenden eines Prozesses. Wird ein Prozeß gestartet, muß zuerst der benötigte Speicher für den Programmcode und die Variablen reserviert und initialisiert werden, anschließend muß der Stack mit der Startadresse des neuen Prozesses sowie den Initialisierungswerten für die Prozessorregister gefüllt werden, und schließlich muß der Stackpointer in einem freien Eintrag der Prozeßtabelle abgelegt werden. Die Ausführung eines gestarteten Prozesses beginnt hierbei erst, wenn der Prozeß durch einen herkömmlichen Prozeßwechsel aktiviert wird. Um einen Prozeß wieder zu beenden, muß lediglich der entsprechende Eintrag aus der Prozeßtabelle entfernt und der belegte Speicher freigegeben werden. Es sei noch angemerkt, daß ein expliziter Prozeßwechsel durch eine Systemfunktion in anderen Systemfunktionen, so z.B. in einer Funktion zur Zeichenausgabe versteckt werden kann. Ein expliziter Prozeßwechsel muß z.B. dann ausgeführt werden, wenn längere Berechnungen durchzuführen sind, um so anderen Prozessen die Möglichkeit zu geben, „parallel“ dazu weiterarbeiten zu können.

Ein Prozeß kann, soweit bisher beschrieben, drei sog. *Prozeßzustände* annehmen: (1.) Er ist noch nicht gestartet bzw. bereits terminiert, d.h., er *existiert* nicht. (2.) Er ist *bereit*, ausgeführt zu werden. (3.) Er wird ausgeführt und ist damit *aktiv*. Eine so arbeitende Prozeßverwaltung hat den Nachteil, daß ein auf ein bestimmtes Ereignis wartender Prozeß von Zeit zu Zeit aktiviert wird, unabhängig davon,

ob das Ereignis bereits eingetreten ist oder nicht. Besser ist es, wenn ein wartender Prozeß in der Prozeßtabelle als *blockiert* gekennzeichnet werden kann, um bei einem Prozeßwechsel nicht berücksichtigt werden zu müssen. Dies entspricht einem vierten Prozeßzustand. Ein solcher Prozeß wird erst dann wieder aktiviert, wenn das entsprechende Ereignis eingetreten ist und dabei die Kennzeichnung „blockiert“ aufgehoben wurde. Eine derart erweiterte Prozeßverwaltung kann z.B. mit zwei zusätzlichen Systemfunktionen realisiert werden: Mit der einen Systemfunktion wird ein Prozeß als blockiert gekennzeichnet, mit der anderen werden alle blockierten Prozesse gleichzeitig „aufgeweckt“. Warten unterschiedliche Prozesse auf unterschiedliche Ereignisse, so müssen die Prozesse selbst entscheiden, ob sie sich – nachdem sie aufgeweckt wurden – ggf. selbst wieder als blockiert kennzeichnen, d.h. sich selbst „schlafen legen“.

Noch ein paar abschließende Worte zum Informationsaustausch zwischen Prozessen, der sog. *Interprozeßkommunikation*: Ein Vorteil des kooperativen Multitasking ist, daß auf Systemfunktionen zur *Prozeßsynchronisation* verzichtet werden kann, da nämlich jeder Prozeß entscheiden kann, zu welchem Zeitpunkt der Ausführung ein Prozeßwechsel durchgeführt werden soll. Zur Interprozeßkommunikation kann daher, ohne zusätzliche Maßnahmen treffen zu müssen, ein globaler Speicherbereich verwendet werden. Es ist nämlich nicht möglich, daß ein laufender Zugriff auf logisch zusammenhängende Daten ungewollt unterbrochen wird und dadurch ein logisch inkonsistenter Zustand der Daten entsteht. Beim präemptiven Multitasking kann es hingegen vorkommen, daß, da der Prozeßwechsel von „außen“ erzwungen wird, bei einem schreibenden Zugriff auf eine zusammenge setzte Datenstruktur ein Prozeßwechsel auftritt, bevor alle Elemente verändert wurden. Wird nach dem Prozeßwechsel durch einen anderen Prozeß auf diese Datenstruktur lesend zugegriffen, ist sie möglicherweise in einem inkonsistenten Zustand. Gelöst wird dieses Problem z.B. mit Hilfe von *Semaphoren* (zu Semaphoren siehe auch 2.1.4 und Beispiel 8.2, S. 558).

5 Busse und Systemstrukturen

Ein Mikroprozessorsystem umfaßt neben dem Mikroprozessor als dem zentralen Systembaustein eine Reihe weiterer Bausteine, insbesondere Speicher- und Interface-Bausteine sowie verschiedene Zusatzbausteine. In den Speicherbausteinen werden die Programme und Daten für ihre Verarbeitung bereitgestellt; über die Interface-Bausteine wird die Verbindung zur Peripherie des Mikroprozessorsystems hergestellt. Zu den Zusatzbausteinen zählen einfache Funktionseinheiten, wie Zeitgeber (timer), aber auch komplexe Funktionseinheiten, wie Speicherverwaltungseinheiten und DMA-Controller. Verbindendes Medium aller Komponenten ist in Einbussystemen der *Systembus*, dessen Signale bei einfachster Ausführung durch den Prozessor festgelegt sind, aber auch prozessorunabhängig definiert sein können. In Mehrbussystemen gibt es zusätzlich zum Systembus sog. *lokale Busse* für den Zusammenschluß von z.B. prozessornahen oder peripherienahen Funktionseinheiten. Hier hat der Systembus als *globaler Bus* eine übergeordnete Verbindungsfunktion, und zwar sowohl zwischen den lokalen Bussen als auch zu Funktionseinheiten, die unmittelbar an den Systembus angeschlossen sind.

Dieses Kapitel befaßt sich mit Strukturaspekten von Mikroprozessorsystemen, ausgehend von den verschiedenen Steuerungsabläufen des Systembusses. Abschnitt 5.1 beschreibt zunächst die grundsätzlichen Formen des Systemaufbaus (Einkarten-, Ein-chip- und modulare Mehrkartensysteme) und der Systemstruktur (Ein- und Mehrbussysteme). Er gibt ferner einen Überblick über die wichtigsten Busfunktionen, Übertragungsmerkmale, Leitungs- und Signalarten sowie über die gebräuchlichsten Busse. Ergänzt werden die Busbeschreibungen durch Darstellungen typischer Ein- und Mehrprozessorstrukturen. Abschnitt 5.2 behandelt dann verschiedene Möglichkeiten der Adressierung von Systemkomponenten, auch unter Berücksichtigung von Bussen mit dynamischer Busbreite und der beiden unterschiedlichen Möglichkeiten der Byte-Adressierung im Speicherwort. Abschnitt 5.3 zeigt darauf aufbauend die Steuerung des Datentransports auf Bussen bzw. durch den Mikroprozessor. Grundsätzliche Techniken der Buszuteilung bei mehreren Busmastern sind in Abschnitt 5.4 dargestellt, die Möglichkeiten der prozessorexternen Priorisierung von Interrupts und den Signalfluß bei der Interruptverarbeitung zeigt Abschnitt 5.5. Abschnitt 5.6 beschreibt schließlich die Merkmale und die Arbeitsweise des PCI-Busses und seines Nachfolgers PCI-X als konkretes Beispiel eines Systembusses großer Verbreitung.

5.1 Systemaufbau und Systemstrukturen

Beim Entwurf von Mikroprozessorsystemen müssen unterschiedliche Gesichtspunkte berücksichtigt werden, so z.B. wirtschaftliche Gesichtspunkte, wie Minimierung der Entwicklungs- und Herstellungskosten, und technische Gesichtspunkte, wie Festlegung der Verarbeitungsgeschwindigkeit und Speicherkapazität. Ferner müssen Anschlußmöglichkeiten für peripherie Einheiten mit zum Teil sehr verschiedenen Anforderungen vorgesehen werden. Darüber hinaus wird vielfach eine hohe Systemzuverlässigkeit gefordert. Die technischen Forderungen lassen sich i. allg. durch einen entsprechend hohen Hardwareaufwand erfüllen, was jedoch mit einer Erhöhung der Systemkosten verbunden ist. Man ist deshalb gezwungen, beim Systementwurf Kompromisse einzugehen, wobei das Spektrum von der Minimierung des Hardwareaufwands bei Einkartensystemen bis zur flexiblen Konfigurierung bei modularen Mehrkarten- und Mehrbussystemen reicht, und wobei zwischen Einprozessor-, Mehrmaster- und Mehrprozessorsystemen abzuwählen ist.

5.1.1 Einkarten- und Ein-chip-System

Mikroprozessorsysteme mit minimaler Hardware werden meist für fest umrissene Aufgaben, z.B. Steuerungsaufgaben, entworfen. Die gesamte Hardware ist dabei auf einer einzigen gedruckten Karte (Platine, board) untergebracht (Einkartensystem, Bild 5-1). Zentraler Bestandteil einer solchen Karte ist ein Bus, der die Funktionseinheiten – in Bild 5-1 den Prozessor, den Programmspeicher, den Datenspeicher und mehrere Ein-/Ausgabe-Interfaces – miteinander verbindet (Einbussystem). Dieser sog. *Systembus* wird üblicherweise aus den Signalleitungen des Prozessors, dem *Prozessorbus*, gebildet. Bei einem herkömmlichen Aufbau sind der Prozessor und die weiteren Funktionseinheiten jeweils einzelne Bausteine, mit entsprechendem Platzbedarf und Verdrahtungsaufwand auf der Karte. Einfacher, platzsparender und kostengünstiger werden Entwurf und Aufbau eines solchen Systems, wenn diese Komponenten zu einem einzigen Baustein (chip) zusammengefaßt sind und sich das System dadurch auf ein Ein-chip-System reduziert.

Einen solchen multifunktionalen Baustein bezeichnet man als *Mikrocomputer* oder, im Hinblick auf die Steuerungstechnik als Hauptanwendung, als *Mikrocontroller (embedded controller)*. Mikrocontroller sind entweder kompakte Entwicklungen mit einfachem Prozessorkern, d.h. mit einer geringen Verarbeitungsbreite von 4 oder 8 Bit, einfachem Befehlssatz und meist niedriger Taktfrequenz, oder sie basieren auf dem Prozessorkern eines komplexen 16- oder 32-Bit-Mikroprozessors mit entsprechend hoher Verarbeitungsleistung und umfangreichem Befehlssatz. Aufgrund der vielfältigen Anwendungsgebiete mit dem individuellen Bedarf an bestimmten Funktionseinheiten werden Mikrocontroller bei gleichem

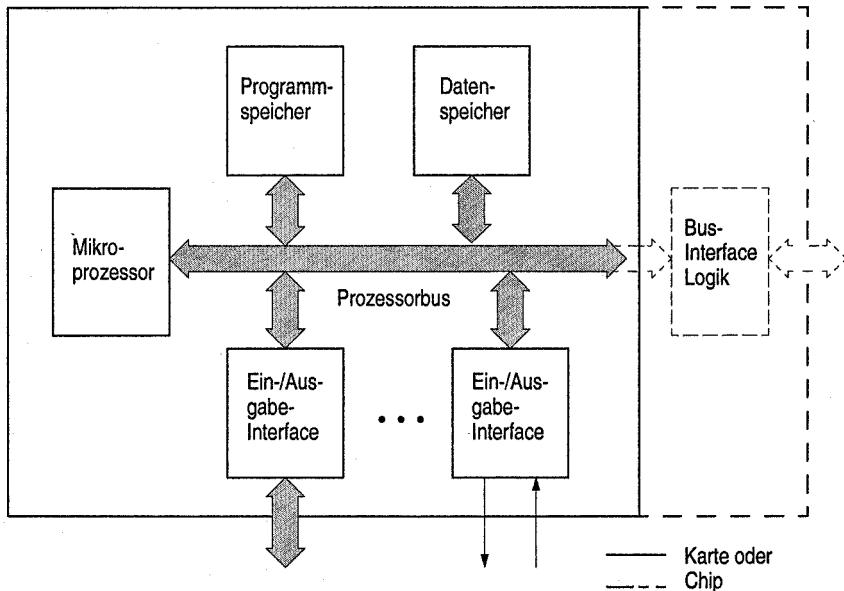


Bild 5-1. Einkarten- bzw. Ein-chip-System. Das Ein-chip-System ist mittels des nach außen geführten Busses (Bus-Interface-Logik) durch externe Komponenten erweiterbar

Prozessorkern meist in vielen Varianten angeboten, so z.B. hinsichtlich ihrer Ausstattung mit SRAM, DRAM, ROM, EEPROM, Flash-ROM, Interrupt-Controller, DMA-Controller, Zählern, Zeitgeber, parallelen und seriellen Schnittstellen, Protokoll-Schnittstellen, A/D- und D/A-Umsetzern. Zusätzliche Ausstattungsmerkmale wie Chip-select-Logik, Wait-state-Generator, Refresh-Generator und ein nach außen geführter Bus unterstützen die Möglichkeit, die durch einen Mikrocontroller vorgegebene Funktionalität zu erweitern, z.B. um zusätzlichen Speicher oder um weitere spezielle Funktionseinheiten.

5.1.2 Mehrkartensysteme

Mikroprozessorsysteme für allgemeine Anwendungen werden üblicherweise modular als Mehrkartensysteme aufgebaut, so daß man in ihrer Strukturierung flexibel ist. Dabei werden zwei Prinzipien verfolgt. Entweder wird ein festes Grundsystem durch zusätzliche Karten über Steckerverbindungen in seinen Funktionen erweitert, wobei Anzahl und Funktion der Karten in gewissem Rahmen wählbar sind, oder das System wird insgesamt durch steckbare Karten konfiguriert, so daß auch das Grundsystem als Karte wählbar bzw. austauschbar ist. Erwünscht ist dabei eine möglichst große Kartenauswahl, die nur dann gewährleistet ist, wenn viele unterschiedliche Firmen solche Karten herstellen und vertreiben. Dazu geht man den Weg der Standardisierung, indem man als *Systembus*

nicht den *Prozessorbus* des jeweils verwendeten Mikroprozessors einsetzt (firmenspezifischer, d.h. *proprietary Bus*), sondern prozessorunabhängige, *standardisierte Busse* verwendet (siehe auch 5.1.3 und 5.1.5).

Das Prinzip „des Erweiterns“ eines Grundsystems findet man bevorzugt bei Arbeitsplatzrechnern (PCs und Workstations). Hier sind die elementaren Funktionseinheiten des Rechners, wie Prozessor, Hauptspeicher, Terminal-Interface, DMA-Controller, Interrupt-Controller, Festplatten- und Floppy-Disk-Controller, üblicherweise auf einer großformatigen Grundkarte (*Hauptplatine, main board*) zusammengefaßt und über einen oder zwei hierarchisch angeordnete Busse, die als *Speicher-, System- oder Ein-/Ausgabebusse* bezeichnet werden, miteinander verbunden. Diese Busse werden über die auf der Grundkarte untergebrachten Funktionseinheiten hinaus verlängert und mit mehreren Stecksockeln pro Bus versehen. In diese Sockel, auch als Slots bezeichnet, können zusätzliche Karten zur Erweiterung des Systems gesteckt werden, z.B. eine Grafikkarte für einen bestimmten Bildschirmstandard, eine Sound-Karte für die Ein-/Ausgabe von Audiodaten oder eine Netzwerkkarte für den Anschluß des Rechners an ein lokales Rechnernetz. Man bezeichnet solche Busse deshalb auch als *Erweiterungsbusse*. Werden die Erweiterungskarten nicht senkrecht, sondern parallel zur Grundkarte gesteckt, was eine sehr flache Bauweise ermöglicht, jedoch die Kartenzahl auf meist 2 beschränkt, spricht man auch von *Huckepack- oder Mezzanine-Bussen*.

Das Prinzip des „vollständigen Konfigurierens“ eines Systems wird bei industriellen Mikroprozessorsystemen angewandt, um eine noch größere Flexibilität in der Strukturierung als bei Arbeitsplatzrechnern zu erreichen. Dazu werden die Funktionseinheiten (Baugruppen) eines Systems grundsätzlich als Steckkarten ausgelegt und diese Karten dann als Einschübe nebeneinander in einem Baugruppenträger (19-Zoll-Rahmen) mit bis zu 20 Steckplätzen untergebracht. Zur Zusammenschaltung der einzelnen Karten werden ihre Signalleitungen über Steckerverbindungen auf eine Rückwandverdrahtung geführt. Diese Verdrahtung wird durch eine gedruckte Karte (backplane) realisiert; sie ist Bestandteil des für alle Karten gemeinsamen Systembusses und für ggf. weitere Busse. Man bezeichnet solche Busse dementsprechend als *Backplane-Busse*. Die Karten haben einheitliches Format, das sog. Europaformat oder das doppelte Europaformat, mit einer bzw. zwei 64- oder 96-poligen Steckerleisten. Sie sind i.allg. wesentlich kleiner als die in ihren Formaten nicht einheitlich festgelegten Grundkarten von Arbeitsplatzrechnern.

Bild 5-2 zeigt als Beispiel die Konfigurierung eines Mikroprozessorsystems mit einem Backplane-Bus. Es umfaßt neben der Stromversorgungskarte eine Rechnerkarte und mehrere Interface-Karten. Auf der Rechnerkarte sind die elementaren Funktionseinheiten des Systems untergebracht, wie sie bei Arbeitsplatzrechnern auf der Grundkarte vorhanden sind, u.a. der Prozessor, der Hauptspeicher und z.B. ein serielles und ein paralleles Interfaces für den Terminal- und den

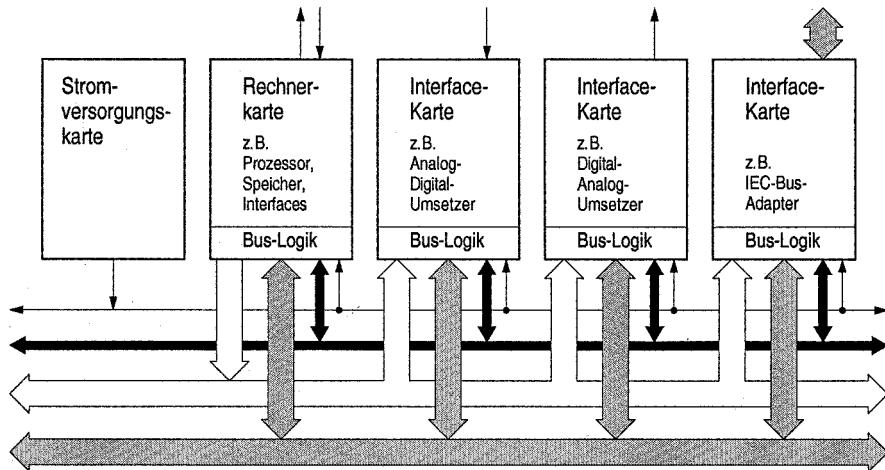


Bild 5-2. Modulares Mehrkartensystem. Die horizontal gezeichneten Busleitungen bilden die Rückwandverdrahtung

Druckeranschluß. Zwei der Interface-Karten sind in diesem System für das Erfassen bzw. das Bereitstellen von Analogwerten mit einem Analog-Digital- bzw. mit einem Digital-Analog-Umsetzer ausgestattet, die dritte Interface-Karte mit einem IEC-Bus-Adapter als Schnittstelle zu einem peripheren Bus. Zusätzliche freie Steckplätze (im Bild nicht dargestellt) können für einen Systemausbau mit Speicher, Interfaces, Controllern usw. genutzt werden. Sie können darüber hinaus aber auch mit zusätzlichen Prozessor- oder Rechnerkarten bestückt werden, um das System zum Mehrprozessor- bzw. Mehrrechnersystem auszubauen. In Bild 5-2 ist die Anpassung an den standardisierten Systembus durch die *Bus-Logik* einer jeden Karte angedeutet. Sie stellt die logische und physische Verbindung zwischen der Hardware, mit der die eigentliche Kartenfunktion realisiert ist, und dem Systembus her.

Zwischen den Stecksockeln eines Backplane- oder eines Erweiterungsbusses gibt es unterschiedliche Verbindungsarten (Bild 5-3). Die gebräuchlichste ist die *Sammelleitung*, die alle Anschlüsse gleicher Steckerposition zusammenfaßt. Daneben gibt es die *Stichleitung* als Einzelverbindung zwischen zwei Steckeranschlüssen und die *Daisy-Chain-Leitung*, die wiederum mehrere Steckerleisten miteinander verbindet, bei der jedoch das Weiterreichen des Signals von den gesteckten Karten abhängig ist. Letztere Verbindungsart wird zur Realisierung unterschiedlicher Prioritäten bei der Busarbitration (siehe 5.4) und bei Interrupts (siehe 5.5) genutzt. Dabei hängt die einer Karte zugeordnete Priorität von der Sockelposition auf dem Bus ab.

Das Konfigurieren eines Rechners mit Hintergrundspeichern und Ein-/Ausgeberäten erfolgt üblicherweise über *Peripheriebusse*. Bei einem solchen Bus werden die Einheiten durch mit Steckkontakten versehene Kabelstücke miteinander

verbunden (*Kabelbus*), entweder in Form einer Kette (auch als *Strang* bezeichnet; z.B. SCSI, 8.2.3) oder baumförmig mittels sog. *Hubs* als zentrale Verteiler und Unterverteiler (z.B. FireWire, 8.2.5, und USB, 8.2.7).

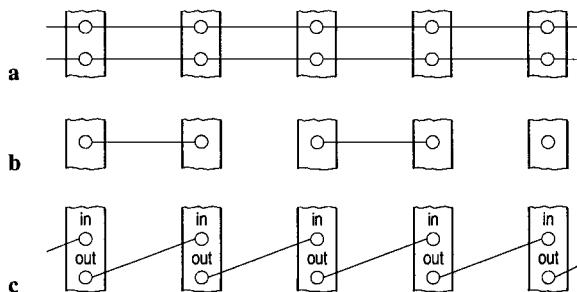


Bild 5-3. Leitungsverbindungen zwischen den Anschlüssen von Steckerleisten. **a** Sammelleitungen, **b** Stichleitungen, **c** Daisy-Chain-Leitung

Erkennt ein Rechnersystem beim Hochfahren seines Betriebssystems seine Komponenten (Steckkarten, Einheiten am Kabelbus) selbst und führt es daraufhin z.B. die Vergabe von Interruptprioritäten, das Zuordnen von DMA-Kanälen und Adressbereichen an die Komponenten selbst durch, so spricht man von *Plug and Play*; ist das Hinzufügen von Komponenten während des laufenden Betriebs möglich, von *Hot Plug and Play*.

5.1.3 Ein- und Mehrbussysteme

Einbussysteme. Bei Einbussystemen sind, wie in 5.1.1 bereits ausgeführt, sämtliche Komponenten eines Mikroprozessorsystems über einen einzigen, gemeinsamen Bus miteinander verbunden, wobei der Bus im wesentlichen aus den Signalleitungen des Prozessors gebildet wird. Solche Systeme haben den Nachteil, daß zu einer bestimmten Zeit immer nur ein einziger Datentransport stattfinden kann und somit bei mehreren Mastern im System, z.B. Prozessor und DMA-Controller, Engpässe auf dem Bus auftreten können. Hinzu kommt, daß die kapazitive Belastung eines Busses mit der Anzahl der Komponenten, die an ihn angeschlossen werden, wächst, und zwar einerseits durch die kapazitive Last einer jeden Komponente, andererseits aber auch durch die mit der Komponentenzahl zunehmende Länge der Busleitungen. Dies hat eine Verringerung der *Bustaktfrequenz* und damit der *Übertragungsgeschwindigkeit* des Busses zur Folge und wirkt sich insbesondere bei Zugriffen auf Komponenten mit kurzen Zugriffszeiten aus, so bei Hauptspeicherzugriffen. Ihre Relevanz haben Einbussysteme jedoch bei Anwendungen, die einen kompakten und kostengünstigen Aufbau bei nicht allzu hoher Leistungsfähigkeit erfordern.

Mehrbussysteme. Die geschilderten Nachteile vermindern sich wesentlich bei Mehrbussystemen. Darunter versteht man Systemstrukturen mit getrennten Bussen für schnelle und langsame Systemkomponenten, mit für jeden Bus begrenzter Komponentenanzahl und dementsprechend begrenzten Leitungslängen sowie mit relativ großer Unabhängigkeit der Datenübertragungen auf den einzelnen Bussen. Die Anordnung der Busse bildet dabei eine Art Mehrebenenstruktur mit dem Prozessorbus an oberster Position. Verbunden werden die Busebenen durch *Bussteuereinheiten* (*bus controller*), die in herkömmlichen Strukturen von PCs oder Server als sog. *Bridges* ausgelegt sind. Die Steuerung der Datentransfers auf der obersten Ebene, also auf dem *Prozessorbus* bzw. dem *Prozessor-/Speicherbus* (siehe nachfolgend), erfolgt durch den Prozessor. Betreffen diese Transfers Komponenten auf einer tieferen Busebene, so wird die Steuerung jeweils von der Bussteuereinheit übernommen, die zur nächst tieferen Ebene verbindet. Bustransfers können aber auch durch andere Master ausgelöst werden, die sich nicht auf der obersten Ebene befinden, z.B. durch einen DMA-Controller.

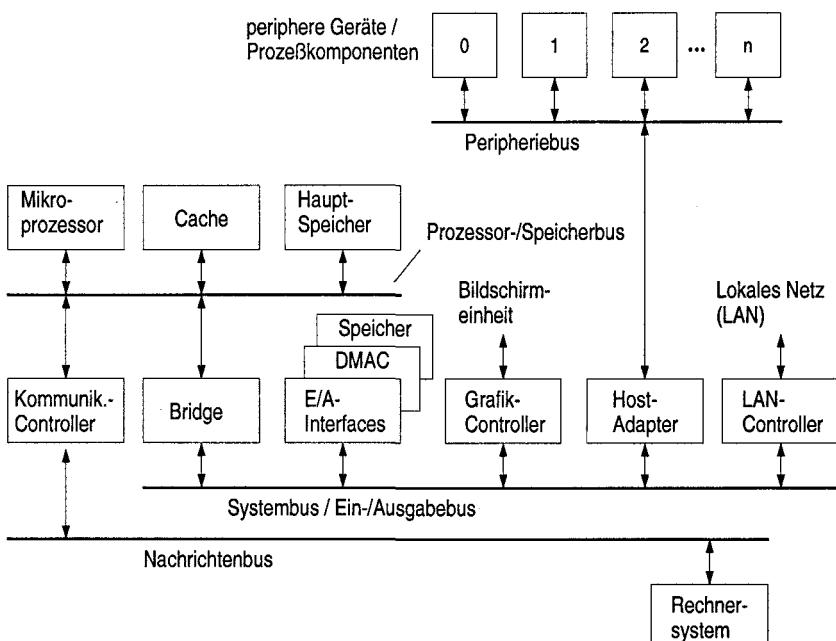


Bild 5-4. Mehrbussystem mit Prozessor-/Speicherbus (z.B. 64-Bit-parallel), Systembus (z.B. 32-Bit-parallel), Peripheriebus (z.B. 8-Bit-parallel) und Nachrichtenbus (bitseriell)

Strukturbispiel. Bild 5-4 zeigt als Beispiel eine Mehrbusstruktur mit insgesamt vier Bussen in drei Ebenen. In dieser Struktur ist der Mikroprozessor über einen schnellen *lokalen Bus*, den Prozessor-/Speicherbus, mit einem prozessorexternen Cache und dem Hauptspeicher (DRAM-Speicher) verbunden, was ihm schnelle Speicherzugriffe ermöglicht. Der Prozessor-/Speicherbus wiederum ist mittels ei-

ner Bridge, die als Systembussteuereinheit wirkt, mit dem eigentlichen *Systembus* (*Ein-/Ausgabebus*) als dem langsameren, *globalen Bus* gekoppelt. An ihn sind vor allem die E/A-Interfaces als Schnittstellen zu den Ein-/Ausgabeeinheiten und der Grafik-Controller zur Aufbereitung und Übermittlung von Text- und Bilddaten an die Bildschirmleinheit angeschlossen. Darüber hinaus sind im Bild als Beispiele für weitere Komponenten eine zusätzliche, globale Speichereinheit, ein DMA-Controller, ein sog. LAN-Controller als Zugang zu einem lokalen Rechnernetz und ein sog. Host-Adapter als Steuereinheit für einen Peripheriebus angegeben. Der Peripheriebus wiederum faßt Geräte (oder Prozessorkomponenten) mit einheitlichen Schnittstellen und den dafür erforderlichen Gerätesteuerungen zusammen. Der vierte im Bild gezeigte Bus, ein sog. *Nachrichtenbus* (*message bus*), dient als serielle Rechner-zu-Rechner-Verbindung.

Prozessor-/Speicherbus. Der Prozessor-/Speicherbus – oft auch nur als *Speicherbus* bezeichnet – erlaubt wie gesagt schnelle Datentransfers zwischen Prozessor und Speicher, und zwar unabhängig vom langsameren Systembus. Er hat die Busbreite des Prozessors und wird vom Prozessor getaktet. Aus Sicht des Prozessors können dementsprechend Buszyklen ohne Wartezyklen ausgeführt werden, vorausgesetzt, der Speicher ist ausreichend schnell, z.B. ein Cache. Die zur Koppelung mit dem langsameren *Systembus* eingesetzte *Bridge* paßt beide Busse in ihren Geschwindigkeiten und ihren Buszyklen aneinander an. Damit einher geht eine Datenpufferung sowie das Auflösen von Prozessorbuszyklen in jeweils mehrere Ein-/Ausgabezyklen und umgekehrt, z.B. eines 32-Bit-Datentransfers des Prozessors in vier aufeinanderfolgende 8-Bit-Datentransfers für eine Ein-/Ausgabeeinheit. In der Bridge können außerdem elementare Systemkomponenten integriert sein, wie DRAM-Controller (zur Ansteuerung des Hauptspeichers), Cache-Controller, Interrupt-Controller, DMA-Controller, Festplatten-Controller und Busarbiter. Sie kann ferner mit einem zusätzlichen Zugang und einer Steuerung für den Anschluß eines Grafik-Controllers ausgestattet sein (siehe im Vorg riff Bild 5-8, S. 288).

Bild 5-4 läßt erkennen, daß der an den Prozessor-/Speicherbus angeschlossene Hauptspeicher zu Zugriffskonflikten führen kann, nämlich dann, wenn zusätzlich zum Prozessor ein zweiter Master im System auf diesen Speicher zugreifen möchte, z.B. ein DMA-Controller, der vom Systembus aus agiert. Hier sind drei Fälle zu unterscheiden:

1. Kein Konflikt: Der Prozessor wickelt seine Daten- und Befehlszugriffe mittels seiner internen Caches ab (im Bild nicht dargestellt).
2. Buskonflikt: Der Prozessor greift auf seinen externen Cache zu.
3. Bus- und Speicherkonflikt: Der Prozessor (genauer: die Steuerung des externen Cache) greift auf den Hauptspeicher zu.

Abhilfe für den *Buskonflikt* schafft eine Änderung der Mehrbusstruktur, wie nach folgend in zwei Varianten gezeigt.

In der ersten Variante nach Bild 5-5a ist der Hauptspeicher unmittelbar an die zwischen Prozessor-/Speicherbus und Systembus liegende Bridge angeschlossen, so daß die vom Systembus ausgehenden Hauptspeicherzugriffe lediglich die Bridge, nicht aber den Prozessorbus belasten. Dabei ist der prozessorexterne Cache entweder mit dem Prozessor-/Speicherbus verbunden (1), wie bei früheren Einprozessorsystemen (PCs) üblich, oder er ist als *Backside-Cache* (6.2.1) direkt mit dem Prozessor verbunden (2), wie bei heutigen Ein- oder Mehrprozessorsystemen (PCs, Server) üblich (Bild 5-12, S. 295).

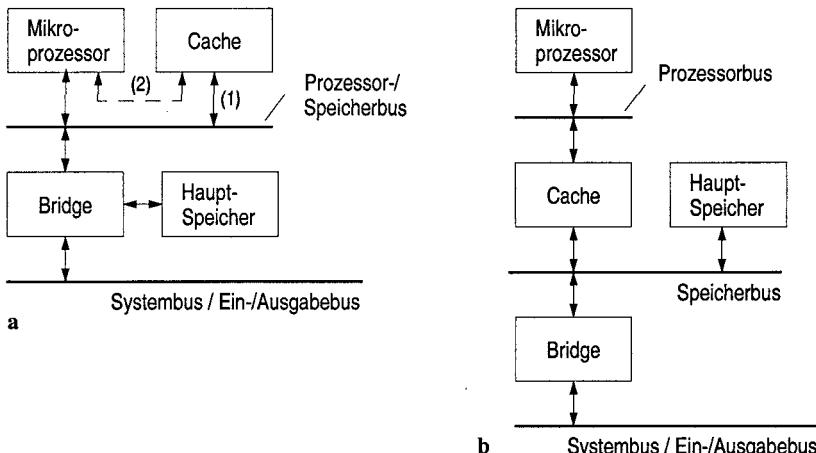


Bild 5-5. Varianten zum Mehrbussystem nach Bild 5-4. Anschluß des Hauptspeichers: a unmittelbar an die Bridge, b an einen eigenen Speicherbus. Teilbild a zeigt zwei Möglichkeiten der Anbindung des prozessorexternen Cache: (1) an den Prozessor-/Speicherbus, wie in Bild 5-4, (2) als Backside-Cache direkt an den Prozessor, wie heute üblich

In der zweiten Variante nach Bild 5-5b ist der Prozessor-/Speicherbus in den eigentlichen Prozessorbus (*CPU bus, host bus*) und einen weiteren, prozessorunabhängigen *Speicherbus (memory bus)* aufgeteilt, d.h., es gibt eine zusätzliche Busebene. Die Verbindung zwischen beiden Bussen wird hier nicht über eine weitere Bridge, sondern über die Steuereinheit des prozessorexternen Cache hergestellt, indem dieser als *Frontside-Cache* (6.2.1) eingesetzt wird. Auch hier belasten die Speicherzugriffe eines zweiten Masters den Prozessorbus nicht, so daß der Prozessor seine Cache-Zugriffe ungehindert durchführen kann. Diese Struktur kann zur Mehrprozessorstruktur erweitert werden (Bild 5-12, S. 295).

Bei beiden Strukturvarianten kann die unterste Busebene erneut aufgeteilt werden, und zwar in den eigentlichen Systembus und in den über eine Bridge mit ihm verbundenen Ein-/Ausgabebus als dann unterster Busebene (Bild 5-8, S. 288). Man bezeichnet die beiden Bridges einer solchen Struktur auch als *North-Bridge* (oberhalb des Systembusses gelegen) und als *South-Bridge* (unterhalb des Systembusses gelegen).

Speicherbus. Der Speicherbus (Bild 5-5b) wird außer für Speicherzugriffe auch für den Anschluß anderer, sonst am Systembus angeschlossenen schnellen Komponenten genutzt, z. B. für Grafik-Controller mit hohen Bildauflösungen und entsprechend hohen Datenraten, für schnelle Peripheriebusse, wie die Ultra-SCSI-Busse, und für LAN-Controller schneller Rechnernetze, z. B. Gigabit-Ethernet. Die hohen Übertragungsgeschwindigkeiten setzen allerdings, wie oben erwähnt, kurze Leitungslängen und eine möglichst geringe kapazitive Last für den Bus voraus. Das schränkt die Komponentenanzahl und vor allem auch die Erweiterungsmöglichkeiten über Stecksockel (*slots*) ein. Die Buskomponenten werden deshalb bevorzugt unmittelbar auf der Grundkarte untergebracht. Bei PCs und Workstations sind die Erweiterungsmöglichkeiten üblicherweise auf zwei oder drei Stecksockel beschränkt. Hier bietet der Speicherbus den Vorteil, daß die Erweiterungskarten, da sie vom Prozessorbus unabhängig sind, ggf. auch in Rechnern mit einem Nachfolgeprozessor eingesetzt werden können. – Bei Systemen mit Backplane-Bus wird der Speicherbus durch Stichleitungen zwischen direkt benachbarten Karten realisiert (Rückwandverdrahtung).

Systembus. Der Systembus wird, sofern er zusätzlich zu einem Ein-/Ausgabebus existiert (z. B. als Erweiterung der Struktur nach Bild 5-5a), in gleicher Weise wie ein Speicherbus genutzt, nämlich zur ausschließlichen Anbindung der schnellen Komponenten. In diesem Fall hat er als schneller Bus dieselben technischen Vorgaben wie ein Speicherbus. Übernimmt er jedoch zusätzlich die Funktion eines Ein-/Ausgabebusses mit einer Vielzahl an Komponenten, so können diese Voraussetzungen nicht eingehalten werden, wodurch der Bus langsamer wird.

Peripheriebus. Als Peripheriebus bezeichnet man einen Gerätebus oder einen Feldbus. Ein *Gerätebus* faßt rechnerspezifische Geräte, wie Festplattenspeicher, CD-ROM-Laufwerke, Magnetbandspeicher (Streamer), Laserdrucker und Scanner, zusammen und verbindet sie über eine gerätebusspezifische Steuereinheit (*Host-Adapter*, *Host-Controller*) mit z. B. dem Systembus des Rechners. Übertragungen finden je nach Bus nicht nur zwischen den Geräten und dem Hauptspeicher des Rechners statt, sondern sind ggf. auch direkt zwischen zwei Geräten möglich, ohne dabei den Systembus zu belasten. Die Datenübertragung erfolgt dabei parallel mit 8 oder 16 Bit (z. B. SCSI, 8.2.3), oder inzwischen zunehmend seriell (z. B. FireWire, 8.2.5, und Universal Serial Bus, 8.2.7).

Ein *Feldbus* (*Prozeßbus*) faßt anwendungsspezifische Funktionseinheiten der Prozeß- und Automatisierungstechnik zusammen und verbindet sie ebenfalls über einen busspezifischen Host-Adapter mit dem Systembus eines in diesem Zusammenhang Prozeßrechner genannten Mikroprozessorsystems. Die Übertragung findet je nach Bus parallel mit z. B. 8 Bit oder bitseriell statt. Parallele Busse werden z. B. für den Anschluß von Labormeßgeräten benutzt, so der IEC-Bus (siehe 8.2.8). Serielle Prozeßbusse eignen sich besonders gut zur Überbrückung größerer Entfernung, z. B., wenn die Prozeßperipherie über eine Fertigungshalle oder ein Gebäude verteilt ist.

Nachrichtenbus (message bus). Ein Nachrichtenbus ist ein bitserieller und deshalb kostengünstiger Bus, wie er als Zusatzbus zu Backplane-Bussen (5.1.5) und bei Mehrprozessorsystemen vom MPP-Typ (massive parallel processing, 5.1.7) eingesetzt wird. Er verbindet eigenständige Rechnersysteme im Sinne einer „losen“ Koppelung. Im Gegensatz zu einer „engen“ Rechnerkoppelung, wie sie bei SMP-Systemen (symmetrical multiprocessing, 5.1.7) vorgenommen wird, erfolgt hier die Kommunikation zwischen den Rechnern nicht durch Zugriffe auf einen gemeinsamen, d.h. globalen Hauptspeicher, sondern durch den Austausch von Nachrichten (messages) zwischen deren lokalen Speichern. Gesteuert und synchronisiert werden diese Übertragungen durch den jeweiligen Kommunikations-Controller, der auch die erforderliche Parallel-Serien- bzw. Serien-Parallel-Umsetzung durchführt.

Anmerkung. Wie eingangs zu Bild 5-4 (S. 266) erwähnt, ist die Verwendung von Bridges als Steuereinheiten herkömmlichen PC- und Serverstrukturen zuzuordnen. Bei neueren Strukturen sind die Bridges durch Hubs ersetzt. Zu beiden Strukturvarianten siehe 5.1.6.

5.1.4 Bus- und Übertragungsmerkmale

Busfunktionen. Wie in 5.1.2 bereits erwähnt, werden *Busse standardisiert*, um die Baugruppen verschiedener Hersteller miteinander kombinieren zu können. Dazu werden von Rechnerherstellern wie von nationalen und internationalen Standardisierungsgremien (z.B. IEEE) Spezifikationen für Bussysteme ausgearbeitet, wobei die von Standardisierungsgremien veröffentlichten Standards häufig auf zuvor gemachten Firmenvorschlägen basieren. Diese Spezifikationen beschreiben neben den mechanischen und elektrischen Eigenschaften von Bussen vor allem die Regeln für die Funktionsabläufe zwischen den Busteilnehmern. Diese Regeln, auch *Busprotokolle* genannt, legen die Bussignale, ihr Zeitverhalten (timing) und ihr Zusammenwirken fest. Um sie zu realisieren, sind bei den einzelnen Buskomponenten Steuereinheiten erforderlich, wie sie in Bild 5-6 am Beispiel des *VMEbus* (Versa Module Europe Bus, [Motorola 1985]), einem *Backplane-Bus*, gezeigt sind (data transfer layer, backplane access layer). Die wichtigsten Funktionen – im Bild durch vier Teilbusse hervorgehoben – sind:

- Die Datenübertragung (data transfer) zwischen einem Master (bussteuernde, aktive Einheit, z.B. Prozessor oder DMA-Controller) und einem Slave (passive Einheit, z.B. Speicher oder Interface). Hierzu gehören die Adressierung des Slave und die Verwaltung der für die Übertragung erforderlichen Steuersignale.
- Die Interruptverwaltung (priority interrupt). Hierzu gehören die Priorisierung der von den Interruptquellen (Interrupter) kommenden Anforderungssignale, die Erzeugung des Gewährungssignals für die akzeptierte Quelle und die Anforderung des Datenbusses für die Übernahme von Statusinformation dieser Quelle (z.B. Vektornummer). Gesteuert wird der Ablauf durch einen oder

mehrere Interrupt-Handler. Die Priorisierung erfolgt hierarchisch in zwei Stufen, zunächst im Interrupt-Handler (zentral) und dann in einer Interrupt-(IACK-)Daisy-Chain (dezentral).

- Die Busarbitration (data bus arbitration) bei mehreren Mastern. Hierzu gehören die Priorisierung der von den Mastern (Requester) kommenden Busanforderungssignale und die Verwaltung von Steuersignalen für eine eindeutige Buszuteilung. Gesteuert wird der Ablauf durch einen zentralen Busverwalter (Arbiter). Die Priorisierung erfolgt auch hier hierarchisch in zwei Stufen, zunächst im Busarbiter (zentral) und dann in einer von mehreren Daisy-Chains (dezentral).
- Hilfsfunktionen (utilities). Hierzu gehören z.B. die Stromversorgung, die Taktversorgung, die Systeminitialisierung, die Anzeige einer zu niedrigen Versorgungsspannung und die Anzeige von Hardwarefehlern im System.

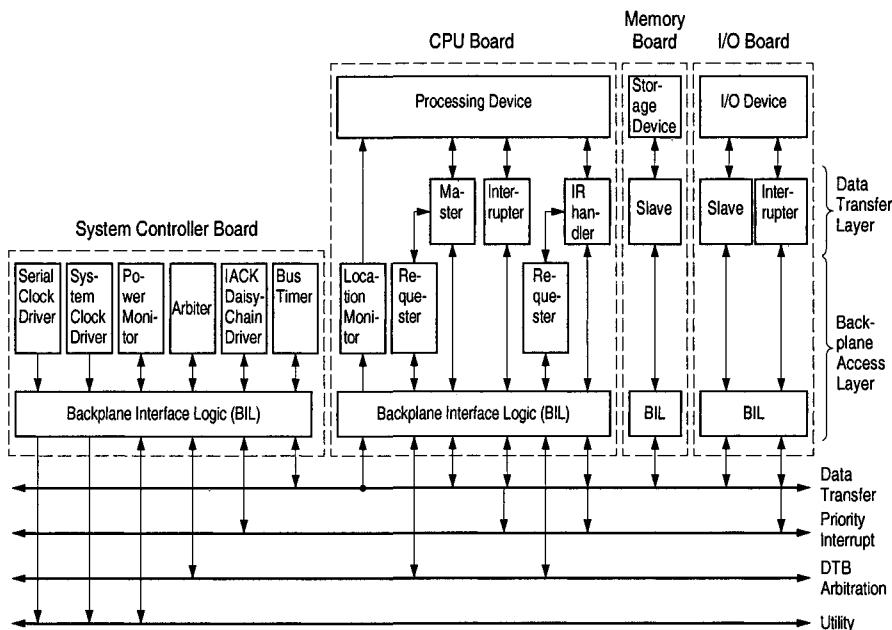


Bild 5-6. VMEbus mit Funktionseinheiten für den Datentransfer, die Interruptbehandlung, die Busarbitration und für Hilfsfunktionen (in Anlehnung an [Motorola 1985])

Hinzu kommen ggf. Funktionen wie

- die Selbstidentifizierung von an den Bus angeschlossenen Komponenten (indem z.B. Steckkarten während einer Systeminitialisierung Statusinformation liefern, *plug and play*) und
- das Testen elektronischer Bauteile von Buskomponenten zur Fehlererkennung während des Betriebs (IEEE 1149.1: JTAG Boundary Scan).

Bussignale. Etwas anders aufgeteilt unterscheidet man bei einem Prozessor- oder Systembus vier Leitungsbündel: den Datenbus, den Adreßbus, den Steuerbus und den Versorgungsbus. Bezuglich des Anschlusses der einzelnen Signalleitungen an die Buskomponenten, z. B. an den Mikroprozessor, ist der Signalfluß zu beachten. Er ist entweder *unidirektional*, d.h., eine Leitung wirkt als Eingang oder als Ausgang, oder er ist *bidirektional*, d.h., die Leitung wirkt mal als Eingang und mal als Ausgang. Zur Gruppe der unidirektionalen Signalleitungen gehören bei reinen Mastern oder Slaves die Adreßleitungen und ein großer Teil der Steuerleitungen; bei Komponenten mit sowohl Master- als auch Slave-Funktion sind diese Leitungen zum Teil auch bidirektional. Datenleitungen wirken bei Komponenten mit Lese- und Schreibfunktion bidirektional und bei Komponenten mit nur Lesezugriff (z. B. Festwertspeichern) unidirektional.

Bei Bausteinen mit Masterfunktion, wie z.B. dem Mikroprozessor, sind die meisten Signalausgänge mit *Tristate-Logik* (TsL) versehen. Sie können neben den beiden logischen Zuständen 0 und 1 einen *hochohmigen Zustand* (*high impedance state*) zur Signalabkopplung annehmen. Dies ermöglicht es dem Mikroprozessor, sich vom Bus abzukoppeln, so daß ihn ohne Probleme ein anderer Busmaster übernehmen kann. Eine weitere Eigenschaft von Signalausgängen ist das *Open-collector-Verhalten*. Open-collector-Ausgänge bilden, wenn sie miteinander verbunden (verdrahtet) werden, bei 0-aktiven Signalen eine Oder-Verknüpfung (*verdrahtetes Oder*, *wired or*). Diese Eigenschaft wird z.B. dazu genutzt, einen einzigen Interrupt-request-Eingang des Mikroprozessors mit den Anforderungssignalen mehrerer Funktionseinheiten zu verdrahten, ohne ein Oder-Verknüpfungsglied zu benötigen.

Bestimmte Steuersignale des Busses dürfen im sozusagen „abgeschalteten“ Zustand nicht hochohmig sein. Dies könnte zu Problemen führen, nämlich dann, wenn solche Signale aufgrund ihrer elektrostatischen Umgebung zufällige logisch gültige Werte annehmen. Auf diese Weise könnte z.B. ein hochohmiges Adreßgültigkeitssignal, das zufällig den Aktivpegel aufweist, zusammen mit einem hochohmigen R/W-Signal, das zufällig 0-Pegel aufweist, eine ungewollte Schreiboperation auf eine Speicherstelle bewirken. Um dies zu verhindern, werden solche Steuerleitungen über Widerstände entweder an die Spannungsversorgung (*pull-up*) oder an Masse (*pull-down*) angebunden, so daß sie im „abgeschalteten“ Zustand auf ihren Inaktivpegel bzw. den jeweils unkritischen Pegel gezogen werden. Man bezeichnet solche Signale als *sustained Tristate-Signale*. – Die beschriebene Situation kann z.B. in Mehrmastersystemen auftreten, nämlich dann, wenn sich der letzte Busmaster vom Bus getrennt und der neue Busmaster den Bus noch nicht übernommen hat. (Beim PCI-Bus wird bei einem solchen Übergang zusätzlich ein sog. Turnaround-Zyklus eingefügt; siehe dazu 5.6.2.)

Abgesehen vom hochohmigen Zustand befindet sich eine Steuerleitung entweder im aktiven Zustand (man sagt auch: das Signal ist gesetzt), d.h. ihre Funktion ist wirksam, oder im inaktiven Zustand (das Signal ist nicht gesetzt, man sagt: es ist

rückgesetzt), d.h. ihre Funktion ist ohne Wirkung. Unterschieden werden *1-aktive Signale* (Signalname nicht überstrichen) die, wenn sie im Aktivzustand sind, den logischen Wert 1 haben, und *0-aktive Signale* (Signalname überstrichen) die, wenn sie im Aktivzustand sind, den logischen Wert 0 haben. In der Mikroprozessorteknik sind fast alle Steuersignale als 0-aktive Signale ausgelegt. Sie erlauben, technologisch bedingt, ein kürzeres Umschalten in den Aktivzustand. Einige Steuersignale, wie das R/W-Signal, sind sowohl mit dem Wert 1 (read) als auch mit dem Wert 0 (write) im aktiven Zustand, weshalb ihre Bezeichnung aus zwei Teilen besteht, die durch einen Schrägstrich getrennt sind. Bei diesen Signalen stellt der Wert 1 den Aktivzustand des nichtüberstrichenen Bezeichnungsteils und der Wert 0 den Aktivzustand des überstrichenen Bezeichnungsteils dar. – Die Kurzbezeichnungen von Bussignalen und Bausteinanschlüssen sind üblicherweise aus deren Funktionen hergeleitet.

Bei parallelen *Peripheriebussen* wird üblicherweise jede der Signalleitungen mit einer Signal-Erdeleitung als Signalrückführung verdrillt, um das (elektrische) Übersprechen zwischen den Signalleitungen zu verhindern. Man spricht dabei auch von *asymmetrischer Übertragung* (massebezogenes Signal, *single-ended signal*). Bei seriellen Peripheriebussen erfolgt die Signalübertragung üblicherweise *symmetrisch*, d.h., das Signal wird auf zwei Leitungen einmal in normaler und einmal in invertierter Form übertragen (*differential signal*). Auf der Empfängerseite gibt es dazu einen Differenzverstärker; dort heben sich die auf den Einzelleitungen ggf. entstandenen Störanteile gegenseitig auf. Diese Technik ermöglicht höhere Taktfrequenzen und gestattet es darüber hinaus, größere Entfernnungen zu überbrücken. So erreicht man mit seriellen Bussen ggf. sogar höhere Übertragungsraten als mit parallelen Bussen, und das bei wesentlich geringerem Leitungsaufwand.

Daten- und Adreßbusbreite. Bei *Prozessorbussen* sind der Daten- und der Adreßbus in ihrer Leitungszahl an die Verarbeitungsbreite bzw. die Adreßlänge des Prozessors angepaßt. Bei vielen Controllern und einfacheren Prozessoren umfaßt der Datenbus 8 oder 16 Bit mit typischen Werten für den Adreßbus von 16 oder 24 Bit (Adreßräume von 64 Kbyte bzw. 16 Mbyte). Leistungsfähigere Controller bzw. Prozessoren haben Datenbusbreiten von 32 oder 64 Bit mit Adreßbusbreiten von üblicherweise 32 Bit (Adreßraum von 4 Gbyte). Systembusse gibt es mit denselben Breiten. Diese müssen in einem Mehrbussystem jedoch nicht mit denen des Prozessorbusses übereinstimmen. Die Anpassung der Busbreiten (wie auch die Umsetzung der verschiedenen Buszyklen) übernimmt die sie verbindende Bridge (5.1.3). Bei *Systembussen* mit 64-Bit-Breite werden üblicherweise 32 Daten- und 32 Adreßleitungen zu einem 64-Bit-Daten-/Adreßbus zusammengefaßt, d.h., diese Leitungen werden für die Übertragung von 64-Bit-Daten und von 32-Bit- oder 64-Bit-Adressen im Multiplexbetrieb benutzt (siehe auch weiter unten).

Bei *seriellen Bussen* wird nicht zwischen Adress- und Datenleitungen unterschieden. Vielmehr werden Adressen, Daten sowie Steuerinformation auf demselben Leitungspaar übertragen. Dementsprechend werden für die Übertragung lediglich zwei Leitungen, bei Vollduplexbetrieb ggf. vier Leitungen benötigt. – Serielle Busse haben üblicherweise eine *Strangstruktur*, indem die an sie angeschlossenen Komponenten, eine hinter der anderen, durch einzelne *Punkt-zu-Punkt-Verbindungen* (engl. *link*) miteinander verbunden sind. Oder sie haben eine *Sternstruktur*, mit einem *Hub* oder *Switch* als zentralem Verteiler, an den die Komponenten wiederum Punkt-zu-Punkt angeschlossen sind. Hierbei können Komponenten mit hohen Übertragungsraten ggf. über mehr als nur einen Link verfügen.

Übertragungsrate. Die Leistungsfähigkeit eines Busses wird durch seinen Durchsatz, d.h. durch die auf ihm erzielbare Übertragungsrate (*Übertragungsgeschwindigkeit*) bestimmt. Diese wird bei *parallelen Bussen* als Anzahl der pro Sekunde übertragbaren Bytes in den Maßeinheiten Mbyte/s oder Gbyte/s angegeben (hier $M = 10^6$, $G = 10^9$!). Als rein busbezogene Größe definiert man hier zunächst die *maximale Übertragungsrate*, die als höchstmöglicher Durchsatz allein von den physikalischen Eigenschaften des Busses einschließlich dem für ihn verwendeten Busprotokoll (Buszyklenart) abhängt:

$$\text{Maximale Übertragungsrate} = \frac{\text{Bustaktfrequenz [1/s]} \times \text{Busbreite [byte]}}{\text{Taktanzahl/Transfer}}.$$

Mit einer Taktanzahl pro Transfer von 1, wie sie lange gegeben war, lässt sich die maximale Übertragungsrate allein durch die Bustaktfrequenz und die Busbreite beschreiben. Durch die Verwendung neuerer Übertragungstechniken, wie sie sich in den Begriffen Double-Data-Rate, Quad-Data-Rate, Quad-Pumped u.a. widerspiegeln, verringert sich jedoch die Taktanzahl pro Transfer auf 1/2, 1/4 usw. Hier ist es ggf. anschaulicher, anstelle der Bustaktfrequenz und der Taktanzahl pro Transfer die Anzahl an Transfers pro Sekunde, d.h. die *Transferrate* (T/s), und die Byteanzahl pro Transfer zu betrachten:

$$\text{Maximale Übertragungsrate} = \text{Transferrate [T/s]} \times \frac{\text{Byteanzahl}}{\text{Transfer}} [\text{byte/T}].$$

Der maximalen Übertragungsrate steht die bei einer konkreten Buskommunikation tatsächlich erreichbare, *effektive Übertragungsrate* gegenüber, d.h. die reine Datenübertragungsrate. Als systembezogene Größe bezieht sie, ausgehend von der Definition der maximalen Übertragungsrate, die von den Buskomponenten ggf. verursachten Verzögerungen in die benötigte Anzahl an Bustakten mit ein, so z.B. die Wartezyklen für einen Speicher. Außerdem berücksichtigt sie die für die Adressierung benötigten Takte. Dementsprechend ist die effektive Übertragungsrate niedriger als die maximale Übertragungsrate (siehe dazu auch 7.2.7, S. 472).

Bei *seriellen Bussen* und *seriellen Punkt-zu-Punkt-Verbindungen* wird die Übertragungsrate (Übertragungsgeschwindigkeit) aufgrund der bitseriellen Übertragung in kbit/s, Mbit/s oder Gbit/s angegeben ($k = 10^3$!). Sie ergibt sich als

$$\text{Übertragungsrate} = \text{Schrittaktfrequenz [1/s]} \times \text{Bitanzahl/Schrittakt [bit]}.$$

Üblich ist die Übertragung von nur einem Bit pro Schrittakt; es gibt aber auch serielle Verbindungen, bei denen mehr als ein Bit pro Schrittakt übertragen wird (siehe 7.2.4 und 7.7.2). – Auch bei seriellen Bussen/Verbindungen muß zwischen der maximalen und der effektiven Übertragungsrate unterschieden werden. Die Differenz beider Angaben ergibt sich aus dem für die Protokollsteuerung erforderlichen Zeitaufwand, d.h. für die Übertragung von Adreß- und Steuerinformation. Sie kann erheblich sein, da hier die Datenübertragungsprotokolle ggf. Blöcke vorsehen, die nur steuernde Funktion haben, d.h. selbst keine Daten enthalten.

Anmerkung. In Anlehnung an den in der Analogtechnik gebräuchlichen Begriff Bandbreite wird in der Rechnertechnik häufig der Begriff *Busbandbreite* als Synonym für die maximale Übertragungsrate verwendet (bei Speichern: *Speicherbandbreite*). Da die Bandbreite jedoch ein Frequenzmaß mit der Einheit 1/s ist (genauer: Sinusschwingungen pro Sekunde), die Busbandbreite hingegen abweichend davon die Einheit byte/s oder bit/s hat, verwenden wir den Begriff Busbandbreite nicht.

Bustaktfrequenz. Bei *parallelen Bussen* ist die Bustaktfrequenz die maximale Frequenz, mit der die Signalleitungen eines Busses betrieben werden können. Bei *synchronen Bussen* ist diese Frequenz durch das für die Synchronisation von Master und Slave vorhandene gemeinsame *Bustaktsignal* festgelegt, nach dem die Buskomponenten ihre Handshake-Signale ausrichten; bei *asynchronen Bussen*, bei denen sich Master und Slave über voneinander unabhängig getaktete oder auch nicht getaktete Handshake-Signale synchronisieren, unterliegt deren Einhaltung den Buskomponenten, die diese Signale erzeugen (5.3.1).

Bei PC- und Server-Strukturen liegt ein besonderes Augenmerk auf einer möglichst hohen Übertragungsrate zwischen dem Prozessor (genauer: dem Cache) und dem Speicher. Maßgeblich ist hier – neben einer Busbreite von 64 Bit – die Bustaktfrequenz. Sie liegt derzeit bei Prozessorbussen bei 100, 133 und 200 MHz. (Der Prozessor selbst wird für seine interne Verarbeitung sehr viel höher getaktet.) Ausgehend von jeweils einem Datentransfer pro Takt (*single data rate, SDR*) ergeben sich hier idealisierte Übertragungsraten von 800 Mbyte/s bzw. $\approx 1,06$ und 1,6 Gbyte/s. Diese Übertragungsraten lassen sich bei gleichem Takt verdoppeln, indem man die Datenübertragungen mit beiden Flanken des Taktes steuert (*double data rate, DDR*), d.h., indem man pro Takt zwei Transfers durchführt. Prozessorbusse, wie der des Pentium 4, erlauben es darüber hinaus, die Übertragungsrate nochmals zu verdoppeln, indem sie pro Takt vier Datentransfers durchführen (*quad data rate, QDR, quad pumped bus*). Hier wird, ausgehend von einer Busaktfrequenz von 100, 133 oder 200 MHz, eine Frequenzvervierfachung durchgeführt, die technisch u.a. eine Verringerung des Spannungshubs der Signale erfordert. Man erreicht so idealisierte Übertragungsraten von 3,2, $\approx 4,2$ und 6,4 Gbyte/s. – Bezugnehmend auf die Anzahl an Transfers pro Sekunde bezeichnet man eine solchen Bus abgekürzt mit *FSB400, FSB533, FSB800* (front-side bus) oder *PSB400, PSB533, PSB800* (processor system bus).

Bei *seriellen Bussen* ist üblicherweise das Taktsignal durch geeignete Codierung des Datensignals in diesem enthalten (siehe z.B. USB, 8.2.7), d.h., es wird jeweils vom Sender erzeugt, und der Empfänger gewinnt es aus diesem zurück. Gegebenenfalls wird das Datensignal durch ein Zusatzsignal ergänzt, das die Taktrückgewinnung unterstützt (siehe z.B. FireWire-Version IEEE 1394a, 8.2.5).

Geiteilter Bus und Multiplexbus. Einen wesentlichen Einfluß auf die Leistungsfähigkeit *paralleler Busse* hat die Busbreite, d.h. die Anzahl an Bytes, die pro Datentransfer übertragen werden kann. Dies zeigt sich in der Entwicklung von Prozessor- und Systembussen mit Breiten von 8 und 16 Bit in den Anfängen der Mikroprozessortechnik hin zu Bussen mit Breiten von 32 und 64 Bit bei heutigen Mikroprozessorsystemen hoher Leistung. Dieser Entwicklung steht jedoch der mit der größeren Leitungsanzahl verbundene Aufwand an Platz und Herstellungs-kosten entgegen. Um diesen hinsichtlich der Einsatzbereiche zu optimieren, werden zwei Busarten unterschieden, die die zur Verfügung stehenden Adreß- und Datenleitungen unterschiedlich nutzen,

1. der geteilte Bus (*split bus*), bei dem die Adreß- und Datenleitungen jeweils getrennt sind und die beiden Teilbusse ausschließlich für die Übertragung von Adressen bzw. Daten benutzt werden, und
2. der Multiplexbus (*mux bus*), bei dem die Adreß- und Datenleitungen zusammengefaßt sind und die Adressen und Daten nacheinander, d.h. im Multiplexbetrieb übertragen werden.

Bei gleicher Leitungsanzahl, z.B. 64, hat dabei der Multiplexbus gegenüber dem geteilten Bus den Vorteil der doppelten Übertragungsbreite für Daten, hier 8 Bytes gegenüber 4 Bytes, und erlaubt außerdem Adressen mit mehr als 32 Bits. Als Nachteil benötigt er jedoch bei Lesezyklen einen zusätzlichen Takt zwischen der Adreßausgabe und der Dateneingabe (*turnaround cycle*), um die Richtungs-umschaltung der Multiplexleitungen vorzunehmen. Dieser Nachteil relativiert sich allerdings bei Blockbuszyklen (*burst cycles*), wie sie häufig vorherrschen. Der Vorteil des geteilten Busses liegt in der Möglichkeit, Adressen und Daten gleichzeitig zu übertragen. Dies wird einerseits für die überlappende Ausführung von Einzelbuszyklen (*address pipelining*, 6.1.4) und von Blockbuszyklen (*pipelined burst cycles*, 6.1.5) genutzt. Andererseits lässt sich die Gesamtübertragungsleistung in Mehrprozessorsystemen mit gemeinsamem Bus steigern, indem der Adreßbus und der Datenbus getrennt arbitriert werden (*split transactions*, 6.1.4).

Zur Illustrierung der Wirkungsweise von geteilten Bussen und Multiplexbussen sei auf die Darstellung der Einzelbuszyklen (*single cycles*) in Bild 5-7 verwiesen. Beziiglich der Details zu Multiplexbussen siehe Abschnitt 5.6, in dem der PCI-Bus mit seiner Multiplexfunktion ausführlich beschrieben ist.

Buszyklenarten. Die Anzahl der für eine Übertragung benötigten Bustakte ist, da es auf einem Bus mehrere Arten von Buszyklen für Datentransfers gibt, nicht einheitlich, wobei zur Bestimmung der maximalen Übertragungsrate jene Zyklusart

herangezogen wird, die die geringste Anzahl an Takten für aufeinanderfolgende Datenübertragungen benötigt. Bild 5-7 zeigt die wichtigsten dieser Buszyklen als Zeitdiagramme in Form von Strichdarstellungen mit Taktmarkierungen. Durchgezogene Linien innerhalb der Takte kennzeichnen die Belegung eines Busses, gepunktete Linien die im Prinzip freie Zeit eines Busses. Pfeile mit der Adreßkurzbezeichnung A geben den (idealisierten) Ausgabezeitpunkt der Adressen auf den Bus an, Pfeile mit der Datenkurzbezeichnung D die Datenübernahmen durch den Master (Lesezyklus) bzw. Slave (Schreibzyklus). Die Zuordnungen von Adressen zu Datenwörtern ist durch Numerierung ihrer Kurzbezeichnungen hergestellt. Gezeigt sind Buszyklen von geteilten Bussen mit 32-Bit-Adreßbus (A32) und 32-Bit-Datenbus (D32) sowie von Multiplexbussen mit 64-Bit-Adreß-/Datenbus (AD64). Um die Bus- und Zyklenarten miteinander vergleichen zu können, ist zu jedem Buszyklus seine maximal erreichbare Übertragungsrate, bezogen auf eine Bustaktfrequenz von 66 MHz, angegeben. Zu den Buszyklen im einzelnen:

- Der *Einzelbuszyklus* (single cycle) führt, wie sein Name sagt, nur eine einzige Datenübertragung durch und benötigt dafür üblicherweise 2 Takte (5.3.2). Bei einem Multiplexbus benötigt er für den Lesezyklus aufgrund des erforderlichen Turnaround-Zyklus 3 Takte. Das Datenformat kann Busbreite haben, es kann aber auch kleiner sein. Ausgeführt wird er vom Mikroprozessor.
- Der *Blockbuszyklus* (burst cycle) erweitert den Einzelbuszyklus um drei zusätzliche Datentransfers, benötigt dazu aber nur einen anfänglichen Adreßtransfer. Da die Folgetransfers aufgrund der nur einen Adressierung taktweise möglich sind, benötigt der Blockbuszyklus für 4 Datenübertragungen insgesamt nur 5 Takte (2-1-1-1-burst, siehe 5.3.3). Das Datenformat ist jeweils gleich der Busbreite. Die Anwendung liegt bei Transfers von Cache-Blöcken (*Cache-fill-* und *Write-back-Operationen*). Ausgeführt wird der Blockbuszyklus vom Cache-Controller (6.2.1).
- Der *DMA-Zyklus* (DMA cycle) besteht aus einer Folge einzelner Übertragungen mit jeweils einzelner Adressierung, wobei die Adressen entweder fortzählen (Speicherzugriff) oder sich nicht verändern (Registerzugriff, z. B. in einem Ein-/Ausgabecontroller). Das Datenformat kann Busbreite haben, es kann aber auch kleiner sein. Ausgeführt wird der DMA-Zyklus vom DMA-Controller. – In einer Sonderform des DMA-Zyklus werden zur Steuerung der Übertragungen nicht nur die jeweils gleichen Flanken des Bustakts herangezogen, sondern auch die dazwischenliegenden (DDR). Hierdurch verdoppelt sich die Übertragungsrate unter Beibehaltung der Bustaktfrequenz.
- Der *lange Blockbuszyklus* (long burst cycle) erlaubt mehr als nur vier Datentransfers mit Datenbusbreite, z.B. bis zu 256, bei wiederum nur einem anfänglichen Adreßtransfer. Hier kann für die Angabe der maximalen Übertragungsrate der zusätzliche Adressierungstakt vernachlässigt werden. Eingesetzt werden solche Zyklen unter Mitwirkung eines DMA-Controllers für Speicher-zu-Speicher-Übertragungen, z.B. zwischen dem Hauptspeicher und dem Bildspeicher einer Grafikkarte. Im Gegensatz zu einem normalen DMA-

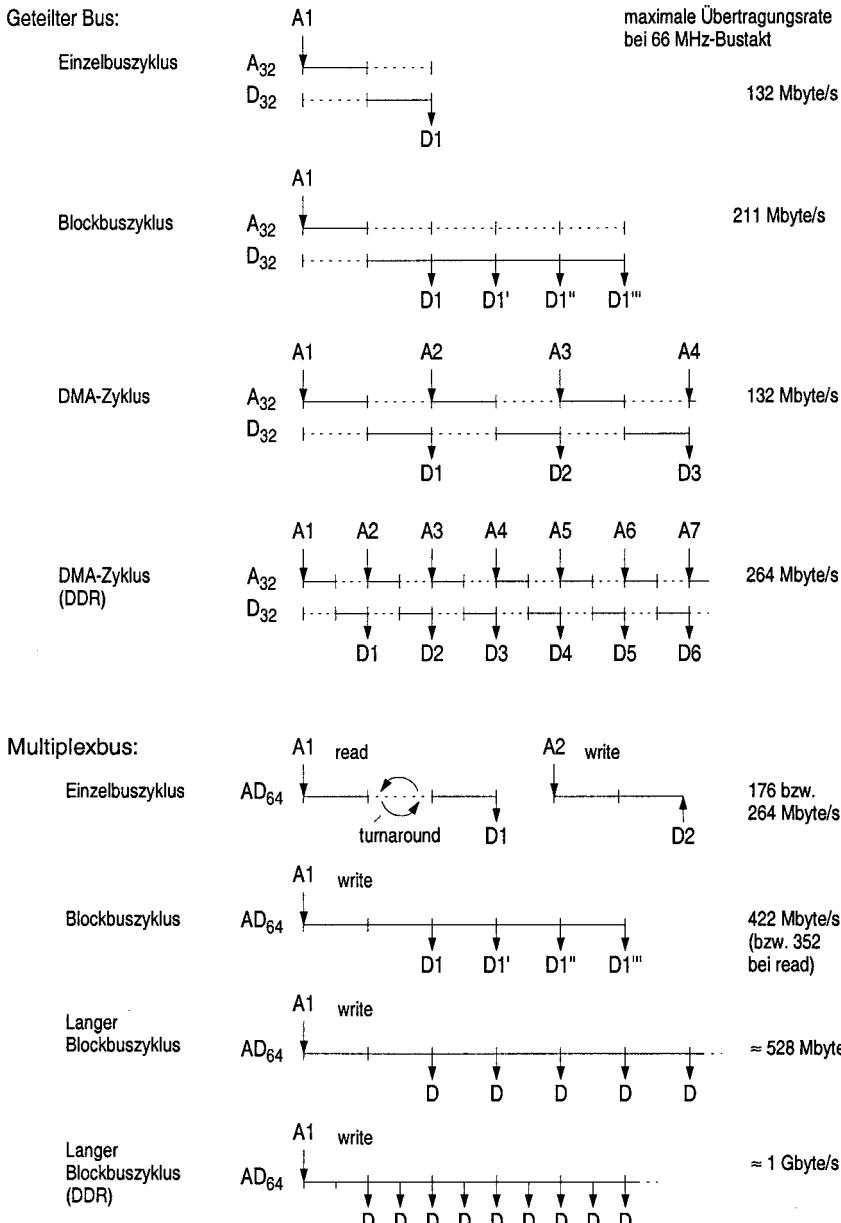


Bild 5-7. Zeitverhalten verschiedener Buszyklenarten für geteilte Busse und Multiplexbusse. Angabe der Zeitpunkte des Anlegens einer Adresse, A, und des Abschlusses des Datentransports, D, gemessen in Bustaktschritten. Angabe der jeweiligen maximalen Übertragungsrate bei einer Bustaktfrequenz von 66 MHz (z.B. PCI-Bus). Die Anzahl der gleichzeitig übertragbaren Bytes ergibt sich aus den für die Adress- und Datenübertragung insgesamt zur Verfügung stehenden 64 Signalleitungen: 4 beim geteilten Bus und 8 beim Multiplexbus

Transfer, der für eine Speicher-zu-Speicher-Übertragung zwei Takte pro Datentransfer benötigt, kommt man hier mit nur einem Takt aus. – Auch hier gibt es die Sonderform der Nutzung beider Taktflanken für die Datenübertragung (DDR), womit sich die Übertragungsrate verdoppelt. (Den langen Blockbuszyklus gibt es auch bei geteilten Bussen; in Bild 5-7 nicht gezeigt.)

In der Praxis wird die *maximale Übertragungsrate* ggf. nur für kurze Zeitdauern erreicht, nämlich dann, wenn die Transfers auf dem Bus Idealbedingungen entsprechen, z.B., wenn der Bus lückenlos mit Einzelbuszyklen genutzt wird (und wenn keine Verzögerungen durch die miteinander kommunizierenden Buskomponenten hinzukommen). Sie ist also eine idealisierende Angabe. Basiert die Angabe auf dem Blockbuszyklus (2-1-1-1-burst), so wird sie häufig verfälscht angegeben, indem die für die Adressierung benötigte Zeit nicht mit eingerechnet wird. Beispielsweise wird auf der Basis des Cache-fill-Zyklus für einen 32-Bit-Bus, der mit 66 MHz getaktet wird, die maximale Übertragungsrate mit 264 Mbyte/s angegeben (4 Takte); erreichbar sind im bestmöglichen Fall aber wegen des für die Adressierung benötigten Zusatztaktes nur 211 Mbyte/s (5 Takte, siehe Bild 5-7). Die Zeit für die Adressierung kann allerdings dann unberücksichtigt bleiben, wenn sich aufeinanderfolgende Blockbuszyklen bzgl. Adressierung und Datentransfer überlappen können (pipelined burst cycles, 6.1.5), so z.B. auf den Prozessorbussen des Pentium und des PowerPC.

Wie oben erwähnt, ist die effektive Übertragungsrate geringer als die rein busbezogene maximale Übertragungsrate, nämlich dann, wenn die Adreß- und Datentransfers aufgrund z.B. nicht genügend schneller Speicher mit *Wartezyklen* ausgeführt werden müssen. Dadurch kann sogar ein höher getakteter Bus, wenn der höhere Takt Wartezyklen bedingt, eine schlechtere Übertragungsleistung haben, als wenn er niedriger getaktet würde. Bezogen auf das obige Beispiel mit 66 MHz als Bustaktfrequenz erreicht man bei einer Taktfrequenz von z.B. 100 MHz und einem daraus ggf. resultierenden Blockbuszyklus mit je einem Wartezyklus pro Übertragung (3-2-2-2-burst) eine maximale Übertragungsrate von nur noch 178 Mbyte/s (9 Takte) gegenüber den oben genannten 211 Mbyte/s.

5.1.5 PC-Busse (ISA, EISA, PCI, PCI-X, PCI-Express)

Wie bereits gesagt, ist es sehr nützlich, Busse zu standardisieren, um eine möglichst große Vereinheitlichung an Hardwarekomponenten und zugehöriger Software zu erreichen. Damit verbunden sind eine hohe Flexibilität im Aufbau von Mikroprozessorsystemen bei insgesamt günstigen Kosten. Im Anwendungsfeld kommerzieller und industrieller Systeme haben sich im Laufe der Jahre eine Reihe von Bussen auf diese Weise etabliert, von denen wir uns im folgenden zunächst einige Speicher- bzw. System- bzw. Ein-/Ausgabebusse als systemzentrale *parallele Busse* herausgreifen und sie in ihren wichtigsten Merkmalen kurz skizzieren. Später in diesem Abschnitt gehen wir dann auch auf eine der zukunftswei-

senden seriellen Alternative ein, auf die Punkt-zu-Punkt-Verbindung PCI-Express.

Tabelle 5-1. Merkmale von PC-Bussen und (im Vorgriff auf 5.1.8) von industriellen Bussen. Varianten sind durch Schrägstriche bzw. Apostroph kenntlich gemacht. Den maximalen Übertragungsraten liegen folgende Buszyklusarten zugrunde. ISA: Einzelbuszyklus mit zwei Takten, EISA: sog. Burst-DMA-Transfer vom Typ C mit jeweils einer Adreß- und Datenübertragung pro Takt, PCI 2.2, PCI-X 1.0 und iPSB: langer Blockbuszyklus mit SDR, PCI-X 2.0: langer Blockbuszyklus mit DDR, VMEbus: langer Blockbuszyklus mit SDR (linke Spalte) bzw. DDR (rechte Spalte) mit zuletzt synchronem Protokoll (bei veränderter Backplane). Zu diesen Angaben siehe auch Bild 5-7

Merkmal	PC-Bus				Industrieller Bus		
	ISA (AT)	EISA	PCI (2.2)	PCI-X (1.0 / 2.0)	iPSB Multi- bus II	VMEbus	
Adreßbus Bit	24	32	32/64	64	32	32	64
Datenbus Bit	16	32				32	
max. Datentrans- portbreite Bit	16	32	32/64	64	32	32/64	64
Bustaktfrequenz MHz	8	8,33	33 66'	133/133	10	10	10/20
max. Übertr.- rate Mbyte/s	8	33	132/264 264'/528'	1.064/2.128	40	40/80	160/320
Multiplexbus	nein	nein	ja	ja	ja	nein/ja	ja
Synchr.-Art	syn.	syn.	syn.	syn.	syn.	asyn.	asyn/syn.
multimasterfähig	nein	ja	ja	ja	ja	ja	ja

Eine Übersicht über die parallelen Busse zeigt Tabelle 5-1, mit Angaben zu *Busmerkmalen*, die sich primär auf die Datenübertragung beziehen, so wie sie im vorangegangenen Abschnitt 5.1.4 diskutiert wurden. Multiplexbusse erkennt man (unabhängig von deren Kennzeichnung in der Tabelle) daran, daß für die Daten- und Adreßbusbreiten nicht zwei, sondern nur eine gemeinsame Angabe gemacht ist, z.B. beim PCI-Bus. Oder daß die maximale Datentransportbreite größer als die Datenbusbreite ist, wie bei einer der Versionen des VMEbus (linke Spalte). Das VMEbus-Beispiel zeigt, daß es ggf. unterschiedliche Versionen eines Busses gibt, nämlich mit 32 und mit 64 Bit Datentransportbreite, was in der Tabelle durch Schrägstriche unterschieden wird. Hier ist die schmalere Version als geteilter Bus und die breitere als Multiplexbus ausgelegt. Diese Varianten führen zu mehreren Angaben der maximalen Übertragungsrate eines Busses. Beim PCI-Bus kann darüber hinaus eine höhere Bustaktfrequenz gewählt werden, was durch Apostroph angegeben ist. Diese erfordert dann allerdings einen höheren elektrotechnischen Aufwand und schränkt die Buslänge und die Anzahl der Buskompo-

nenten ein. – Auf welcher Art von Buszyklus die jeweils angegebenen maximalen Übertragungsraten basieren, ist aus der Tabellenlegende zu entnehmen.

Ein zusätzliches Kennzeichen von Bussen ist die Art der Synchronisation zwischen Master und Slave, die entweder auf einen gemeinsamen Bustakt bezogen, d.h. synchron, oder unabhängig von einem Bustakt, d.h. asynchron, erfolgt. Beide Techniken werden in Abschnitt 5.3 ausführlich erläutert. Ein weiteres Merkmal ist die Fähigkeit von Bussen für den Multimasterbetrieb. Ein Bus gilt dann als *multimasterfähig*, wenn er spezielle Signale und Vorrichtungen für die Busarbitration vorsieht. Busse, wie der ISA-Bus, sehen solche Signale nicht vor, ermöglichen es aber dennoch, zusätzliche Master, z.B. DMA-Controller, zu betreiben. Zu den Techniken der Busarbitration sei auf Abschnitt 5.4 verwiesen, zum ISA-Bus siehe nachfolgend.

Die folgenden Busbeschreibungen haben z.T. historischen Charakter, da Busse wie ISA, EISA und die in Tabelle 5-1 nicht erwähnten Busse MCA und VESA-Local inzwischen kaum mehr anzutreffen sind. Dennoch sind diese Busse interessant, da sie die Entwicklung von den Anfängen bis hin zum heute gebräuchlichen PCI-Bus und zu dessen Nachfolgern widerspiegeln. Zur Vertiefung werden die Busbeschreibungen durch einen Abschnitt „Entwicklungsgeschichte“ ergänzt.

ISA- und EISA-Bus. Der ISA-Bus und der EISA-Bus sind Erweiterungsbusse mit bis zu ca. zehn Steckplätzen. Sie wurden über viele Jahre bei IBM-PCs und dazu kompatiblen PCs als Systembusse eingesetzt (hier als Ein-/Ausgabebusse bezeichnet) und wurden mit dem Prozessorbus über eine *Bussteuereinheit* (*bridge*), die ggf. aus mehreren Bausteinen bestand (*Chipsatz*), gekoppelt. Ausgangspunkt für den ISA-Bus (Industry Standard Architecture Bus) war der sog. *PC-Bus* (Personal Computer Bus), der von IBM für den auf dem 8/16-Bit-Prozessor 8088 von Intel basierenden PC/XT als 8-Bit-Bus entwickelt wurde. Dieser PC-Bus wurde von IBM für das Nachfolgemodell PC/AT (Advanced Technology), das auf dem 16-Bit-Prozessor 80286 von Intel basiert, zu einem 16-Bit-Bus, dem *AT-Bus* erweitert, der Jahre später von der Computerindustrie als ISA-Bus standardisiert wurde. Als patentrechtlich nicht geschützter Bus führte er zu einer Vielzahl von IBM-kompatiblen PCs (sog. clones) und erlangte so große Verbreitung, insbesondere bei den PC/AT-Nachfolgern mit den 32-Bit-Prozessoren *i386* und *i486*. Als kostengünstiger Bus mit einem großen Angebot an Steckkarten wurde der ISA-Bus dann bei den Pentium-PCs vorwiegend als Ergänzungsbuss zum schnelleren PCI-Bus eingesetzt. Inzwischen hat der ISA-Bus auch dort an Bedeutung verloren und ist bei moderneren PCs nicht mehr zu finden.

Der ISA-Bus wird unabhängig vom Prozessortakt mit 8 MHz betrieben. Die Übertragung der Daten wird entweder vom Mikroprozessor oder von einem vom Bus unterstützten DMA-Controller gesteuert. Trotz dieser beiden Master gilt der ISA-Bus jedoch als nicht multimasterfähig, da kein allgemein zugänglicher Busarbiter vorhanden ist und der ISA-Bus dementsprechend auch keine Busarbitrationssignale hat. Ein externer Master, z.B. ein DMA-Controller auf einer Steck-

karte, erreicht eine Buszuteilung nur unter Zuhilfenahme eines der acht Kanäle des vom Bus unterstützten DMA-Controllers, der wiederum in den *Chipsatz* integriert ist. – Die in Tabelle 5-1 angegebene maximale Übertragungsrate von 8 Mbyte/s basiert auf dem Einzelbuszyklus mit minimal zwei Takten Buszykluszeit (siehe auch Bild 5-7, geteilter Bus: single cycle). Die per DMA-Transfer maximal erreichbare Übertragungsrate liegt aufgrund einer ungünstigen Organisation unter diesem Wert.

Der *EISA-Bus* (Extended ISA-Bus) ist eine Erweiterung des ISA-Busses zum 32-Bit-Bus unter Gewährleistung der Kompatibilität zu diesem; dementsprechend wird auch er mit 8 MHz betrieben. Er wurde von mehreren PC-Herstellern entwickelt, um die Leistungsfähigkeit von PCs zu erhöhen. Aufgrund der mit ihm verbundenen höheren Kosten und des später hinzugekommenen, sehr viel leistungsfähigeren PCI-Busses hat er jedoch nicht die zunächst erwartete Verbreitung gefunden. Seine Entwicklung war insbesondere auch eine Reaktion auf den von IBM als Nachfolger des ISA-Busses entwickelten und patentrechtlich geschützten 32-Bit-Bus *MCA* (Micro Channel Architecture). Dieser wurde von IBM jedoch lediglich bei den mit dem i386-Prozessor ausgestatteten PS/2-Modellen mittlerer und höherer Leistungsklasse eingesetzt und ist inzwischen vom Markt verschwunden. – Die in Tabelle 5-1 angegebene maximale Übertragungsrate von 33 Mbyte/s basiert auf dem sog. Burst-DMA-Transfer vom Typ C mit minimal einem Takt pro Datenübertragung. In dieser Betriebsart werden zur Steuerung der Datenübertragung beide Taktflanken ausgenutzt, womit sich die Übertragungsrate gegenüber herkömmlichen DMA-Zyklen verdoppelt (siehe auch Bild 5-7, geteilter Bus: DMA cycle, DDR). In einer erweiterten Definition des Busses gibt es einen sog. Enhanced-Master-Burst-Cycle, bei dem die Datenübertragungen ebenfalls mit beiden Taktflanken synchronisiert werden, wodurch maximal 66 oder 132 Mbyte/s erreicht werden. Zur Erzielung der 132 Mbyte/s wird vom geteilten zum Multiplexbus übergegangen (siehe auch Bild 5-7, Multiplexbus: long burst cycle, DDR).

VESA-Local-Bus. Der VESA-Local-Bus (*VL-Bus*, in Tabelle 5-1 nicht aufgeführt) ist ein prozessornaher 32-Bit-Bus, der von Herstellern von Grafikkarten, die sich in der VESA (Video Electronics Standards Association) zusammengeschlossen hatten, für den Einsatz in i486-PCs entwickelt wurde. Als Zusatzbus zum ISA- oder EISA-Bus war er für den Anschluß von Grafikkarten mit hohen Übertragungsraten, aber auch z. B. von schnellen SCSI- und LAN-Controllern gedacht. Seine Bustaktfrequenz wurde zunächst mit 33 MHz festgelegt (der ursprünglichen Taktfrequenz des i486 entsprechend), und es wurden bis zu drei Steckplätze vorgesehen. Realisiert wurde er aber auch mit 40 und 50 MHz, ggf. jedoch mit nur einem oder auch keinem Steckplatz. Die Kopplung mit dem i486-Prozessorbus erfolgt über einen VL-Bus-Controller. – Durch die enge Bindung an den i486-Prozessorbus ist der VL-Bus mit dem Aufkommen der Pentium-PCs vom Markt verschwunden und wurde dabei durch den PCI-Bus ersetzt. Seine maximale Übertragungsrate betrug bei 33 MHz 105 Mbyte/s, basierend auf dem

4-Wort-Blockbuszyklus mit minimal fünf Takten Blockbuszykluszeit (siehe auch Bild 5-7, geteilter Bus: burst cycle).

PCI-Bus. Der PCI-Local-Bus, kurz PCI-Bus (Peripheral Component Interconnect Bus), wurde zunächst von Intel als schneller prozessornaher, jedoch prozessorunabhängiger Systembus entwickelt, um schnelle Komponenten, wie Grafik-, SCSI- und LAN-Controller, an ihm betreiben zu können. Spezifiziert wurde er letztlich 1992 von der PCI Special Interest Group (PCI-SIG); derzeit gilt Version 2.2. Die Verbindung mit dem Prozessorbus wird über die North-Bridge, auch als *Host-to-PCI-Bridge* bezeichnet, hergestellt (Bild 5-8, S. 288). Der Bus kann aber auch davon unabhängig an der South-Bridge oder an entsprechenden Hubs betrieben werden (Bild 5-9, S. 290). In der Konfigurierung mit 33 MHz Bustaktfrequenz und 32 Bit Busbreite können an ihn bis zu zehn Komponenten (einschließlich Bridge) angeschlossen werden, wenn diese auf der Platine untergebracht sind. Anstelle von jeweils zwei solcher On-board-Komponenten kann je ein Steckplatz für eine Erweiterungskarte vorgesehen werden, d.h., es sind maximal vier solcher Steckplätze möglich. Um mehr als vier Steckkarten in einem PC oder Server verwenden zu können, muß der Bus expandiert werden, entweder durch einfaches oder mehrfaches Kaskadieren mittels *PCI-to-PCI-Bridges* oder durch Verzweigen von einer entsprechend modifizierten Host-to-PCI-Bridge auf gleich mehrere PCI-Busse. In der Konfigurierung mit 66 MHz (spezifiziert 1994) halbiert sich die Anzahl der anschließbaren Komponenten. Schließlich ist der Bus bei beiden Taktfrequenzen auf 64 Bit Breite erweiterbar.

Der PCI-Bus ist für beliebig lange „Bursts“, d.h. für kontinuierliche Datenübertragungen mit beliebiger Anzahl an Datentransfers, ausgelegt, wodurch er eine hohe Übertragungsleistung hat. Als Multiplexbus zeigt er die für solche Busse typische Eigenschaft: Bei Lesezyklen ist nach Ausgabe der Adresse, die im Idealfall nur einen Takt kostet, ein Umschalten der Busrichtung für den Datentransport erforderlich, was einen zusätzlichen Takt kostet (*Turnaround -Zyklus*). Neben seiner höheren maximalen Übertragungsrate hat der PCI-Bus gegenüber den oben genannten Bussen den großen Vorteil, daß er in seinen grundsätzlichen Funktionen nicht auf einen bestimmten Prozessor oder eine bestimmte Prozessorfamilie, z.B. die Intel-Prozessoren, zugeschnitten ist, sondern universell einsetzbar ist. So ist er in heutigen PCs und Servern dominant und wird darüber hinaus in vielen anderen Rechnern eingesetzt, z.B. in den mit PowerPC-Prozessoren ausgestatteten Macintosh-Rechnern. Unterschiede in der Hardware ergeben sich jeweils in den Schnittstellen der Host-to-PCI-Bridges zu den Prozessorbussen hin. Systemänderungen, die sich durch Weiterentwicklung der Prozessoren ergeben, schlagen sich deshalb lediglich als Änderungen dieser Schnittstellen nieder, sind also lokaler Natur.

Die in Tabelle 5-1 angegebenen maximalen Übertragungsraten von 132 und 264 Mbyte/s basieren auf einem Bustakt von 33 MHz, einer Busbreite von 32 bzw. 64 Bit und dem „beliebig“ langen Burst mit minimal einem Takt pro Datenübertra-

gung unter Vernachlässigung des Adreßtransports (siehe auch Bild 5-7, Multiplexbus: long burst cycle). Bei 66 MHz Bustakt verdoppelt sich die maximale Übertragungsrate abhängig von der Busbreite auf 264 bzw. 528 Mbyte/s. – Zum PCI-Bus siehe auch 5.6.

PCI-X-Bus. Der PCI-X-Bus ist eine Weiterentwicklung von PCI 2.2 hin zu höheren Übertragungsraten, ausgelöst durch Erfordernisse schneller Ein-/Ausgabetechniken, wie Gigabit-Ethernet, Ultra-3-SCSI und Fibre-Channel. Er sieht dazu eine geänderte Übertragungstechnik vor, die Bustaktfrequenzen von 100 und 133 MHz erlaubt (derzeit gültige Revision 1.0 [PCI-X 1999]). Dennoch gibt es eine weitgehende Kompatibilität zwischen beiden Bussen. So können einerseits PCI-X-Komponenten so entworfen werden, daß sie in PCI-Systemen mit den herkömmlichen PCI-Bustaktfrequenzen und PCI-Modi zu betreiben sind. Andererseits sind herkömmliche PCI-Komponenten in PCI-X-Systemen lauffähig, indem sie mit einer ihnen verträglichen Taktfrequenz betrieben werden. Dabei wird der jeweilige Übertragungspartner ggf. auf dieselbe, reduzierte Frequenz gezwungen. Mit höherer Bustaktfrequenz schränkt sich die Anzahl an auf den Bus steckbaren Komponenten auf zwei bzw. nur eine ein. Hierzu und zu den Übertragungsraten bei PCI 2.2 und PCI-X 1.0 siehe Tabelle 5-2. Weitere Angaben zu PCI-X 1.0 finden sich in 5.6.4.

Tabelle 5-2. Technischen Merkmale der Busse PCI 2.2 und PCI-X 1.0

Busbreite Bit	Bustaktfrequenz MHz	Übertragungsrates Mbyte/s	Anzahl an PCI-Steckplätzen	Anzahl an PCI-X-Steckplätzen
32	33	133	4	–
64	66	533	2	4
64	100	800	–	2
64	133	1.064	–	1

Angekündigt ist die Version 2.0 des PCI-X-Busses mit einer Verdoppelung der maximalen Übertragungsrates auf 2,1 Gbyte/s mittels DDR; geplant ist eine Vervierfachung auf 4,3 Gbyte/s durch QDR.

PCI-Express. Grundsätzlich haben parallele Busse, wie PCI und PCI-X mit ihren 32 oder 64 Datenleitungen, den Nachteil, technisch sehr aufwendig zu sein und dennoch in den maximalen Übertragungsraten den steigenden Anforderungen an eine Systemversorgung nicht mehr nachkommen zu können. Hier bahnt ich eine Ablösung der parallelen durch die serielle Datenübertragung an, und zwar in Form von *Punkt-zu-Punkt-Verbindungen* und unter Einsatz von Switches. So ist als Nachfolger von PCI und PCI-X bereits PCI-Express als serielle Technik im Kommen.

PCI-Express ist als rechnerinterne Verbindungsart konzipiert (*chip-to-chip interconnect*). Eine PCI-Express-Verbindung (lane genannt) hat für jede Übertragungsrichtung ein differentielles Leitungspaar, also insgesamt vier Datenleitungen. Als Taktfrequenz wird derzeit von 2,5 GHz ausgegangen, womit Übertragungsraten von 2,5 Gbit/s pro Richtung (5 Gbit/s im Vollduplexbetrieb) möglich sind. Für die Zukunft nimmt man an, bei Kupferleitungen bis zu 12 GHz erreichen zu können.

Die Bytes werden, um eine Gleichverteilung von Einsen und Nullen im Byte- bzw. Bitstrom herbeizuführen, in *8B/10B-Codierung* übertragen, d. h., ihre Bitanzahl wird in geeigneter Weise von 8 auf 10 erhöht [Widmer, Franaszek 1983]. Diese Codierung wird bei vielen seriellen Verbindungsarten angewandt. Sie gewährt Gleichspannungsfreiheit sowie eine gesicherte Taktrückgewinnung auf der Empfängerseite. Dementsprechend reduziert sich die Übertragungsrate für die Nutzbits pro Richtung auf 2 Gbit/s netto, was einer Übertragungsrate von ca. 250 Mbyte/s entspricht. Die Nutzdatenrate ist allerdings um einiges geringer, da hier – wie bei allen seriellen Übertragungsverfahren – noch der Protokollaufwand in Rechnung gestellt werden muß. Bei höheren Anforderungen an die Übertragungsraten kann eine Verbindung mit mehr als nur einer Lane ausgestattet werden. Ein Beispiel einer PC-Struktur mit PCI-Express zeigt Bild 5-10, S. 292.

Tabelle 5-3. Gegenüberstellung der Übertragungsraten der parallelen Busse PCI und PCI-X sowie der seriellen Punkt-zu-Punkt-Verbindung PCI-Express. Des weiteren vergleichende Angaben der jeweils auf eine einzelne Datenleitung bezogenen Werte. Die Übertragungsraten der parallelen Busse wurden dazu umgerechnet: 1 Gbyte/s = 8 Gbit/s. DDR und QDR stehen für Double- bzw. Quad-Data-Rate; *) kennzeichnet bei PCI-Express die Nettoübertragungsrate bei Vollduplexbetrieb (8B/10B-Codierung!).

Bus bzw. Punkt-zu-Punkt-Verbindung	Idealisierte Übertragungsrate Mbit/s	Anzahl an Daten- leitungen	Übertragungs- rate/Datenlei- tung Mbit/s
parallel:			
PCI 2.2 (32 Bit, 33 MHz)	1.056	32	33
PCI 2.2 (64 Bit, 66 MHz)	4.224	64	66
PCI-X 1.0 (64 Bit, 133 MHz)	8.512	64	132
PCI-X 2.0 DDR (64 Bit, 133 MHz)	17.024	64	264
PCI-X 2.0 QDR (64 Bit, 133 MHz)	34.048	64	528
seriell:			
PCI-Express (2,5 GHz)	2.000 / 4.000*)	4	1.000*)

Übertragungsraten: Parallele versus serielle Verbindung. Einer der Gründe für den Übergang zur seriellen Übertragung ist, daß man mit ihr, verglichen mit der parallelen Übertragung, die gleichen oder höhere Übertragungsraten erreicht, und das bei sehr viel geringerem technischen Aufwand bzgl. der Leitungsanzahl. Tabelle 5-3 zeigt dazu einen Vergleich zwischen PCI-Express (ausgehend von nur

einer Lane und der derzeitigen Taktfrequenz von 2,5 GHz) und den verschiedenen PCI- und PCI-X-Bus-Versionen einschließlich des künftigen PCI-X-2.0. Gezeigt werden (1.) die maximalen Übertragungsraten der jeweiligen Verbindung und (2.) die jeweils auf eine einzelne Datenleitung bezogenen Anteile der Übertragungsraten. Bei der zweiten Betrachtung liegt PCI-Express um den Faktor 7,5 über PCI-X 1.0.

Entwicklungsgeschichte. Eine wichtige Motivation für die Entwicklung der PC-Busse waren und sind die immer wieder steigenden Leistungsanforderungen von *Grafik-Controllern* und heutigen sog. *Grafikbeschleunigern* als Controller für die 3D-Darstellung und -Animation. In der ersten Phase der PC-Entwicklung bestand die Bildschirmsdarstellung vorwiegend aus Text und ggf. einfacher Grafik. Dementsprechend war der Aufwand an Datenübertragung zwischen Hauptspeicher und Grafik-Controller (und auch an Rechenleistung des Prozessors für die Bildschirmsdarstellung) gering, so daß der Grafik-Controller bei der damals üblichen PC-Struktur (Prozessorbus, Host-to-ISA-Bridge, ISA-Bus) am *ISA-Bus* betrieben werden konnte. Mit Aufkommen der fensterorientierten Benutzeroberflächen in der Bildschirmsdarstellung und aufwendigerer Anwendungsgrafik stiegen jedoch die Leistungsanforderungen erheblich. So wurde für i486-PCs (nachdem sich der gegenüber dem ISA-Bus schnellere *EISA-Bus* nicht durchgesetzt hat) zunächst der *VL-Bus* eingeführt, der es erlaubte, den Grafik-Controller mit der Übertragungsraten des Prozessorbusses zu betreiben. Im Nebeneffekt konnte er außerdem für weitere schnelle Komponenten, wie SCSI-Host-Adapter und LAN-Controller, eingesetzt werden.

Wie oben bereits erwähnt, scheiterte dieses Vorgehen jedoch am Zuschnitt des VL-Busses auf den i486-Prozessorbus, weshalb der VL-Bus mit Einführen des Pentiums und dessen neuem Prozessorbus nicht weiterverwendet werden konnte. Hier kam nun der parallel zum VL-Bus und für hohe Übertragungsleistung entwickelte *PCI-Bus* zum Zuge mit dem Vorteil, durch sein Bridge-Konzept vom Prozessorbus unabhängig zu sein. Mit dem Wunsch nach 3D-Grafik und Videodarstellung (Bewegtbilder) stößt man allerdings auch an dessen Leistungsgrenzen. Die derzeitige Lösung dieses Problems besteht darin, den Grafik-Controller in einer Punkt-zu-Punkt-Verbindung noch „direkter“ als mit dem PCI-Bus an den Hauptspeicher anzubinden. Diese Punkt-zu-Punkt-Verbindung, die auf der Technik des PCI-Busses basiert, wird über die Host-to-PCI-Bridge hergestellt, die dazu einen eigenen Anschluß (Steckplatz) für den Grafik-Controller vorsieht. Das Stichwort heißt hier *Accelerated Graphics Port (AGP, Intel)*. Der eigentliche PCI-Bus bleibt dabei als eigenständiger Bus für die anderen schnellen Rechnerkomponenten erhalten. Bei 32 Bit Breite und 66 MHz Taktfrequenz lassen sich ohne oder mit Frequenzvervielfachung zunehmende Übertragungsraten erreichen: bei sog. AGP-1x 266 Mbyte/s, bei AGP-2x 533 Mbyte/s, bei AGP-4x 1,06 Gbyte/s und bei AGP-8x 2,1 Gbyte/s.

Anmerkung. Bei AGP-1x werden die Signalpegel der Datenleitungen mit den positiven Flanken des Taktsignals ausgewertet (SDR). AGP-2x nutzt im Prinzip beide Flanken des Taktsignals für

die Datenabgriffe (DDR). Realisiert wird das durch ein Strobe-Signalpaar mit differentieller Signaldarstellung, deren fallende Flanken im Wechsel ausgewertet werden. Als Folge davon verdoppelt sich die Häufigkeit der Pegelwechsel der Datensignale, die damit gleich der des Taktsignals ist. Bei AGP-4x (QDR) gibt es zwei solcher differentieller Signalpaare, die zeitlich zueinander versetzt sind (Frequenzvervierfachung). Die damit verbundene erneute Verdoppelung der Häufigkeit der Pegelwechsel und Verdoppelung gegenüber der des Taktsignals wird technisch durch Halbieren der Spannungspiegelhubs von 3,3 V auf 1,5 V und Verkleinern der Strukturabmessungen ermöglicht. Bei AGP-8x (ODR) wird die Frequenz erneut verdoppelt, womit das Achtfache von 66 MHz erreicht wird. Der Pegelhub wird dazu weiter gesenkt, auf 0,8 V.

Die inzwischen sehr hohen Anforderungen der Peripherie an die Übertragungsleistung, z. B. für Gigabit-Ethernet, erfordern erneut schnellere Busse, wie sie jetzt durch die *PCI-X*-Varianten vorliegen bzw. geplant sind. Da aber auch die Leistungsfähigkeiten der Mikroprozessoren und deren Prozessorbusse sowie des Speicherzugriffs enorm gestiegen sind, werden Busstrukturen zunehmend als Engpaß gesehen, da auf ihnen immer nur zwei Komponenten zu einer Zeit miteinander kommunizieren können. Der Trend geht deshalb hin zu seriellen Verbindungen wie *PCI-Express*, mit Switches als Übertragungsvermittler, die es ermöglichen, mehrere voneinander unabhängige Transfers gleichzeitig durchzuführen. Hohe Übertragungsanforderungen können durch den Einsatz mehrerer solcher Verbindungen, die parallel betrieben werden, erfüllt werden.

5.1.6 PC-Strukturen mit Bridges, Hubs und Switches

PC-Struktur mit Bridges. Bezieht man den Accelerated Graphics Port in die Darstellungen der Bilder 5-4 und 5-5a mit ein, so erhält man die inzwischen klassische PC-Struktur nach Bild 5-8. Typisch sind hier die drei Bussebenen mit Prozessorbus (host bus), PCI-Bus als Systembus und ISA-Bus als Ein-/Ausgabebus sowie die beiden sie verbindenden Bridges. Der Prozessor, z. B. ein Pentium, ist hier mit einem *Backside-Cache* ausgestattet, d.h., er weist eine eigene L2-Cache-Verbindung auf. Diese Verbindung erlaubt höhere Taktfrequenzen als der Prozessorbus und somit höhere Übertragungsraten als dieser. Wird der Prozessorbus von mehreren Prozessor-Cache-Modulen gemeinsam benutzt (typisch für Server-Strukturen), so bietet der Backside-Cache außerdem den Vorteil, daß die Prozessoren nicht laufend um den Bus miteinander konkurrieren müssen (siehe 5.1.7: SMP-Systeme).

Die *Host-to-PCI-Bridge (north bridge)* verbindet den Prozessorbus mit dem PCI-Bus (hier 32 Bit, 33 MHz) und stellt darüber hinaus zwei weitere Schnittstellen zum Hauptspeicher und zum Grafik-Controller (AGP-Port) zur Verfügung. Sie koordiniert die Datenübertragungen zwischen diesen vier Anschlüssen, davon ausgehend, daß solche nicht nur vom Prozessor (oder den Prozessoren), sondern auch vom Grafik-Controller und von PCI-Bus-Komponenten mit Masterfunktion initiiert werden können. Hierbei müssen die unterschiedlichen Busprotokolle ineinander umgesetzt werden. Um den Durchsatz zu optimieren, sieht die Bridge

Pufferregister zur Zwischenspeicherung von Daten vor. So kann z.B. ein Master von einem Schreibvorgang sofort entlastet werden, auch wenn das Ziel seiner Übertragung, z.B. der Hauptspeicher, gerade durch eine andere Übertragung blockiert ist. In die Bridge integriert ist außerdem der DRAM-Controller und der PCI-Busarbiter. Zu einer ihrer vielen Funktionen gehört z.B. auch das Abbilden des Interrupt-Acknowledge-Signals des Prozessors auf das Acknowledge-Protokoll des PCI-Busses. – Der PCI-Bus stellt primär Stecksockel für eine individuelle Systemkonfigurierung mit Steckkarten bereit.

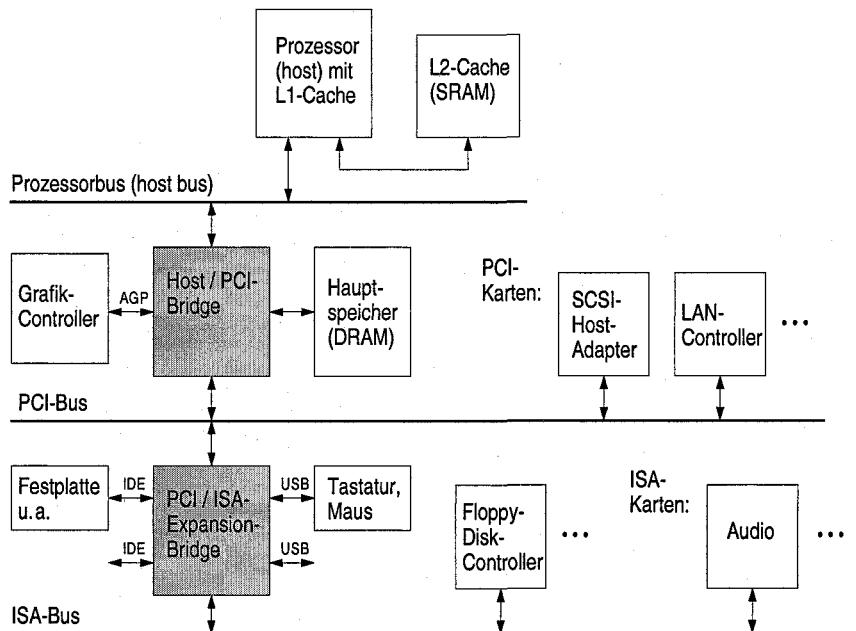


Bild 5-8. Bridge-basierte Struktur eines PC mit Prozessorbus, PCI-Systembus und ISA-Ein-/Ausgabebus, in Anlehnung an den *Chipsatz 440BX AGP* von Intel. Ankopplung der schnellen Systemkomponenten an den PCI-Bus, der langsamen Komponenten an den ISA-Bus. Zugang zur Festplatte vom PCI-Bus aus. Die beiden Bridges (grau unterlegte Kästchen) sind als Chipsatz in ihren Funktionen aufeinander abgestimmt. Anschluß des L2-Cache über eine eigene Prozessor-Cache-Verbindung, d.h. als *Backside-Cache*

Die *PCI-to-ISA-Bridge (south bridge)* verbindet den PCI-Bus mit dem ISA-Bus und koordiniert dementsprechend die Datenübertragung zwischen beiden. Als weitere Schnittstellen einschließlich der zugehörigen Controller hat sie zwei *IDE-Ports* (hier Ultra DMA/33) für den Anschluß (Punkt-zu-Punkt) von bis zu vier Festplatten und CD-ROM-Laufwerken sowie zwei *USB-Ports* (hier USB 1.0) für den Anschluß von z.B. Tastatur und Maus. Zur Optimierung von Blocktransfers zwischen PCI-Bus und IDE-Geräten stehen wiederum Pufferregister zur Verfügung. Als zusätzliche Funktionseinheiten enthält die Bridge u.a. zwei *DMA-Controller*, einen Timer und einen Interrupt-Controller, der ggf. auch für die Inter-

ruptanforderungen von PCI-Bus-Komponenten zuständig ist. – Der ISA-Bus stellt wie der PCI-Bus Stecksockel bereit, hat darüber hinaus aber auch feste Komponenten, wie den Floppy-Disk-Controller und das *BIOS-ROM* (Basic Input/Output System, im Bild nicht gezeigt), in dem die für das Laden des Betriebssystems und die als Schnittstelle zur Hardware erforderlichen Routinen gespeichert sind.

PC-Strukturen mit Hubs (und Switch, PCI-Express). Die Idee, eine Bridge nicht nur als verbindendes Element zweier Busse zu sehen, sondern sie, wie in Bild 5-8 gezeigt, als zentrale Vermittlungseinrichtung mit zusätzlichen Schnittstellen für *Punkt-zu-Punkt-Verbindungen* (AGP, IDE) und für weitere Busse (USB) auszulegen, wird bei neueren PC-Strukturen in den Vordergrund gestellt. Das heißt, auch der Systembus und der Ein-/Ausgabebus werden als zentral verwaltete Anschlüsse ausgelegt und als solche der North- bzw. der South-Bridge zugeordnet, die jetzt als zentrale Verteiler, sog. *Hubs* ausgelegt sind. Die beiden Hubs sind dementsprechend nicht mehr über den Systembus miteinander verbunden, sondern über eine eigens dafür vorgesehene, busunabhängige Schnittstelle, die wiederum eine schnelle Punkt-zu-Punkt-Verbindung ist. Derzeit ist diese Verbindung wie auch die Verbindungen zu weiteren Hubs noch parallel ausgelegt, wenn auch mit geringerer Breite als sie ein paralleler Bus hat, z.B. mit 8 oder 16 Bit Breite. Bei künftigen Systemen wird sich allerdings der Trend hin zu seriellen Punkt-zu-Punkt-Verbindungen und zu seriellen Bussen auch in den Hub-Verbindungen niederschlagen, weshalb wir im folgenden beide Szenarien betrachten.

Einsatz paralleler Verbindungen. Bild 5-9 zeigt ein solche hub-basierte Struktur. Anstelle der North-Bridge gibt es hier einen Memory-Controller-HUB (MCH), anstelle der South-Bridge einen I/O-Controller-Hub (ICH). Verbunden sind beide *Punkt-zu-Punkt* über ein 8-Bit-Hub-Interface als parallele Verbindung, das zwar lediglich mit 66 MHz getaktet ist, jedoch aufgrund einer Frequenzvervielfachung von vier eine maximale Übertragungsrate von 266 Mbyte/s aufweist. Diese ist somit doppelt so hoch wie die des am ICH angeschlossenen PCI-Busses (hier 32 Bit, 33 MHz, d.h. 133 Mbyte/s), d.h., sie birgt Reserven für die Versorgung der anderen am ICH angeschlossenen Komponenten. Hier zeigen sich die Vorteile des Hub-Konzepts gegenüber dem Bridge-Konzept (Bild 5-8):

1. Das Hub-Interface ermöglicht als Punkt-zu-Punkt-Verbindung eine höhere Übertragungsrate als ein technisch vergleichbarer Bus.
2. Der Durchsatz zwischen den beiden Hubs wird nicht wie bei North- und South-Bridge durch einen sie verbindenden Bus begrenzt. Ein solcher Bus, an dem ja die Systemkomponenten betrieben werden, erlaubt es, zu einer Zeit immer nur mit zwei Übertragungspartnern (einschließlich Bridges) Daten zu übertragen. Dabei kann die tatsächliche Übertragungsrate erheblich unter der maximal möglichen Übertragungsrate des Busses liegen, abhängig davon, wie schnell die jeweiligen Partner in der Lage sind, Daten zu übertragen.

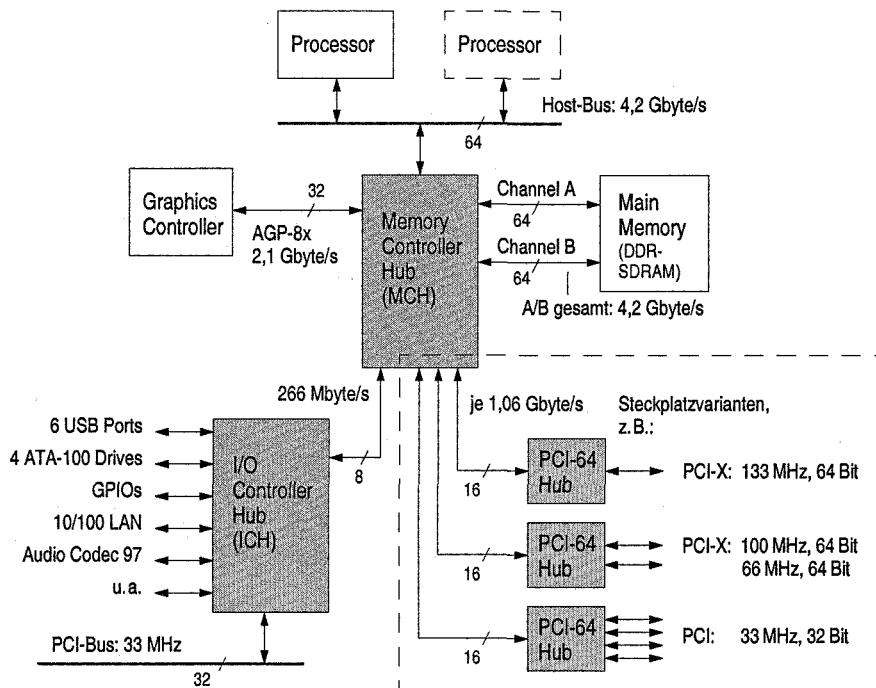


Bild 5.9. Hub-basierte PC-Struktur mit Erweiterungen zur Server-Struktur (gestrichelt umrandete Bildteile), ausgehend von den Memory-Controller-Hubs E7205 und E7501 von Intel. Insgesamt kommen drei unterschiedliche Hubs zum Einsatz (grau unterlegte Kästchen). Als Prozessoren wirken in der PC-Struktur ein Pentium 4 und in der Server-Struktur zwei Xeon-Prozessoren. Deren L1- und L2-Caches sind nicht dargestellt

Die im Bild angegebene maximale Übertragungsrate des Prozessorbusses von 4,2 Gbyte/s wird durch 64 Bit Busbreite, 133 MHz Bustakt und Quad-Pumped-Technik (QDR) erreicht. Dieselbe Übertragungsrate ist sinnvollerweise auch für den Speicheranschluß realisiert. Basierend auf der Verwendung von DDR-SDRAM-Bausteinen mit 133 MHz Taktfrequenz ist hierzu gegenüber dem Prozessorbus eine doppelt so breite Speicherverbindung erforderlich. Dazu gibt es zwei 64-Bit-Kanäle (Channel A und Channel B), die gleichzeitig betrieben werden. (Bei Verfügbarkeit von DDR-SDRAMs mit 266 MHz Taktfrequenz könnte ein solcher Chipsatz mit nur einer Speicherverbindung konzipiert sein.)

Für eine höhere Leistungsfähigkeit, wie sie z.B. bei Servern benötigt wird, kann der Prozessorbus mit einem zweiten Prozessor versehen werden. In diesem Fall ist dann der MCH (E7501) zusätzlich mit drei 16-Bit-Punkt-zu-Punkt-Verbindungen zu drei sog. PCI64-Hubs ausgestattet, von denen jeder – frequenz- und busbreitenabhängig – wahlweise vier, zwei oder eine Verbindung zu PCI- oder PCI-X-Steckplätzen vorsieht. Für die schnellste PCI-X-Variante mit 133 MHz Bustakt und 64 Bit Busbreite sind die Hubverbindungen mit 1,06 Gbyte/s optimal ausgelegt.

legt. Sie haben je 16 Bit Breite, sind mit 66 MHz getaktet und übertragen mit achtfacher Frequenz. – Zu PCI und PCI-X siehe als Überblick 5.1.5 und ausführlicher 5.6.

Übergang zu seriellen Verbindungen (PCI-Express). *Parallele Busse* mit ihrer Vielzahl an Signalleitungen haben den Nachteil, daß sie wegen ihrer relativ hohen Leitungskapazitäten nur über kurze Entfernnungen und mit eingeschränkten Taktfrequenzen betrieben werden können. Nachteilig ist außerdem, wie bereits gesagt, daß immer nur zwei der Buskomponenten gleichzeitig miteinander kommunizieren können und daß dabei ggf. die maximale Übertragungsrate des Busses nicht ausgeschöpft wird (abhängig von Art des Buszyklus, Wartezyklen).

Serielle Busse hingegen haben bei z.B. differentieller Signaldarstellung ein sehr viel besseres Übertragungsverhalten, weshalb sich mit ihnen sehr viel größere Entfernnungen überbrücken lassen. Dabei werden Übertragungsraten erreicht, die aufgrund höherer Taktfrequenzen ähnlich hoch wie bei parallelen Bussen sind. Geht man einen Schritt weiter und verwendet anstelle eines seriellen Busses mehrere serielle *Punkt-zu-Punkt-Verbindungen* (serielle *Links*), und faßt diese in einer zentralen Schalteinrichtung, einem *Switch*, zusammen, so erhält man eine Systemstruktur, wie sie bei Rechnernetzen gebräuchlich ist. Dort ist ein solcher Switch z.B. auf der einen Seite mit einem Backbone hoher Übertragungskapazität und auf der anderen Seite über Punkt-zu-Punkt-Verbindungen mit den Rechnerkomponenten eines lokalen Netzes verbunden, wobei diese ggf. geringere Übertragungskapazitäten als der Backbone haben. Der Switch erlaubt dabei das gleichzeitige Übertragen zwischen mehreren Übertragungspaaren, so daß die Übertragungsleistung des *Backbone-Netzes* insgesamt gut genutzt werden kann.

Bild 5-10 zeigt eine Weiterentwicklung der hub-basierten Struktur nach Bild 5-9 hin zu seriellen Punkt-zu-Punkt-Verbindungen. Zum Einsatz kommt hier an mehreren Stellen *PCI-Express* als der von der PCI Special Interest Group akzeptierte und inzwischen standardisierte serielle Nachfolger des PCI-Busses (basierend auf Intels Third Generation I/O, 3GIO), und zwar sowohl an zentralen Stellen, nämlich als Ersatz für AGP und als Hub-Interface, als auch zur Anbindungen von Peripherie, wofür ein *Switch* als Verteiler eingesetzt ist. Weitere serielle Verbindungen gibt es für den Anschluß von Festplatten und anderen Laufwerken in Form von SATA, dem seriellen Nachfolger der parallelen Punkt-zu-Punkt-Verbindung IDE (ATA), und in Form des USB, hier in der Version 2.0 (480 MBit/s). SATA und USB 2.0 können allerdings auch schon der Struktur in Bild 5-9 zugeordnet werden.

PCI-Express ist, wie in 5.1.5 beschrieben, eine rechnerinterne Punkt-zu-Punkt-Verbindung (*chip-to-chip interconnect*) mit je einem differentiellen Leitungspaar für jede Übertragungsrichtung, wobei zwei solcher Paare eine vollständige Verbindung (Lane) mit Vollduplexeigenschaft bilden. Übertragen wird mit 8B/10B-Codierung, d.h., bei einer Taktfrequenz von derzeit 2,5 GHz ergibt sich für jedes Leitungspaar eine Übertragungsrate von 2,5 Gbit/s brutto bzw. 2 Gbit/s netto.

Höhere Übertragungsraten, z. B. in Bild 5-10 zwischen I/O-Controller-Hub und Switch, erreicht man unter Verwendung mehrerer parallel betriebener Lanes.

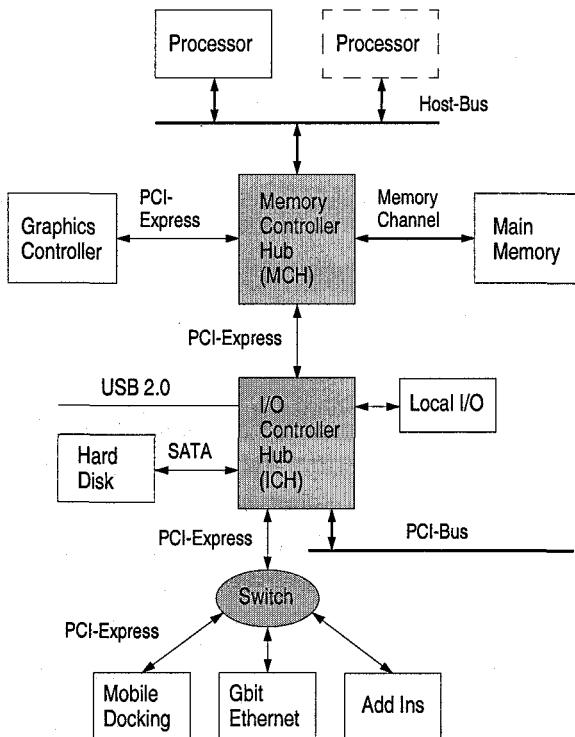


Bild 5-10. PC-Struktur auf der Basis zweier Hubs und eines Switch (graue Kästchen) mit vielfachem Einsatz serieller Punkt-zu-Punkt-Verbindungen in Form von PCI-Express und SATA (serial ATA) (nach www.pcstats.com)

Vernetzung von Rechnern und Ein-/Ausgabeeinheiten mittels serieller Verbindungen (InfiniBand) und Switches. Neben PCI-Express gibt es noch weitere serielle Verbindungsarten als Chip-zu-Chip-Verbindungen, z.B. *Rapid I/O* (IBM) und *HyperTransport* (AMD). Auch *InfiniBand* war zunächst einer dieser Konkurrenten, insbesondere als geplanter Nachfolger für den PCI-Bus. Mit der Anerkennung von PCI-Express als PCI-Nachfolger wurde dann der Einsatzbereich von InfiniBand hin zu rechnerexternen Verbindungen verlagert (*box-to-box interconnect*). Als hauptsächliche Anwendung ist hier an die Vernetzung von PCs (genauer von Servern, Rechnerknoten) mit Verbunden von externen Speichern (Ein-/Ausgabeknoten), vornehmlich aus Festplattenlaufwerken und Streamern bestehend, gedacht. Solche Verbunde sind z. B. *RAIDs* (redundant array of independent disks), sog. *JBOFs* (just a bunch of disks) und Streamer-Bibliotheken. – Bild 5-11 zeigt eine solche Struktur, bestehend aus drei Rechnerknoten und drei Ein-/Ausgabeknoten, die mittels mehrerer InfiniBand-Switches in Form einer sog.

Switched Fabric (Schaltgewebe, any-to-any network) miteinander vernetzt sind. Diese Struktur ist typisch für Speichernetzwerke, sog. SANs (storage area networks), wie sie bisher vorwiegend mit Fibre-Channel-Verbindungen gebildet wurden (siehe dazu auch 8.2.6, Fibre-Channel). Die Verwendung mehrerer Switches erhöht den Durchsatz, da für die Kommunikation zweier Übertragungspartner ggf. mehrere Verbindungswege bestehen.

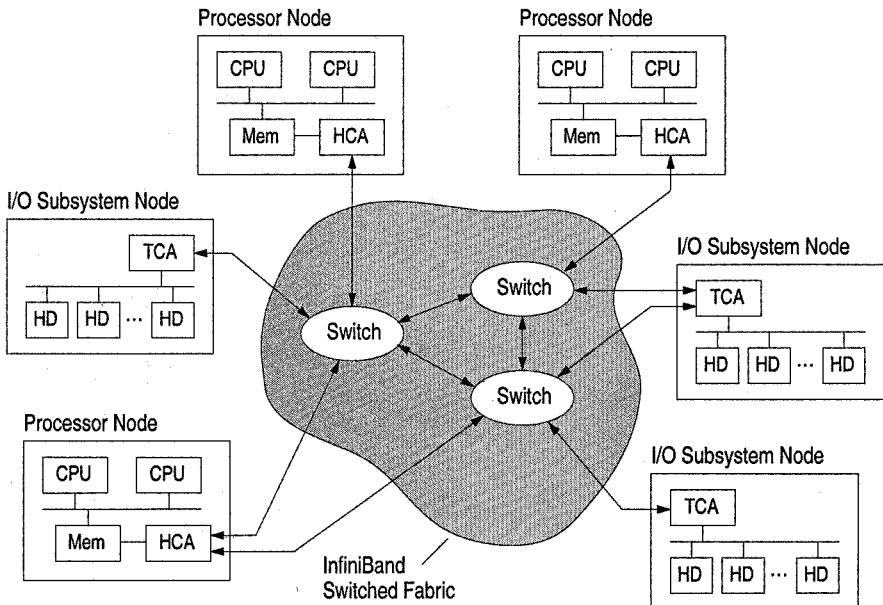


Bild 5-11. Vernetzung von Rechner- und Ein-/Ausgabeknoten mittels InfiniBand-Punkt-zu-Punkt-Verbindungen und InfiniBand-Switches zu einem Storage-Area-Network (nach www.oreillynet.com)

Wie bei PCI-Express umfaßt eine einzelne Punkt-zu-Punkt-Verbindung (link) je ein differentielles Leitungspaar für jede der beiden Richtungen, wird mit 2,5 GHz getaktet und hat dementsprechend bei Vollduplexbetrieb eine maximale Übertragungsrate von $2 \times 2,5 \text{ Gbit/s}$ brutto (bei 8B/10B-Codierung). Eine Verbindung kann aus 1, 4 oder 12 Links bestehen. Die Übertragung ist nachrichtenorientiert (*message passing*), d.h., es gibt keinen gemeinsamen Adressraum, sondern eine verbindungsbezogene Adressierung. Übertragen wird entweder zwischen zwei HCAs (Host-Channel-Adapter im Rechnerknoten) oder zwischen HCA und TCA (Target-Channel-Adapter im Ein-/Ausgabeknoten); ein Switch muß nicht dazwischenliegen. Mehrere gleichzeitige Übertragungen sind möglich, auch direkt zwischen „intelligenten“ Ein-/Ausgabeknoten (mit HCA), d.h. ohne Mitwirkung eines Rechners. Überbrückbar sind Entfernung von bis zu 17 m (Kupferkabel) oder 10 km (Lichtwellenleiter).

5.1.7 Mehrprozessorstrukturen (UMA, NUMA)

Hauptanliegen beim Entwurf und Bau neuer Mikroprozessor- oder allgemein Rechnersysteme ist es, die Leistungsfähigkeit solcher Systeme immer weiter zu steigern. Abgesehen von den technologischen Fortschritten, die zu immer „schnelleren“ Elektronikbausteinen führen, erreicht man solche Steigerungen insbesondere durch Veränderungen in den Systemstrukturen. Vom Einprozessorsystem herkommend ging man zunächst dazu über, Datenübertragungen, die blockweise durchführbar sind, wie z.B. den Datentransport zwischen dem Hauptspeicher und den Hintergrundspeichern, einem DMA-Controller als festprogrammiertem Prozessor zu übertragen (8.1). In einer Erweiterung wurden für solche Ein-/Ausgabevorgänge dann zusätzlich herkömmliche, programmierbare Prozessoren eingesetzt, um den zentralen Prozessor von Vor- und Nachbereitungsaufgaben zu entlasten (8.1). Als Engpaß dieser Mehrmastersysteme stellt sich allerdings der für alle Rechnerkomponenten gemeinsame Systembus heraus. Gelöst oder abgeschwächt wird dieses Problem durch Systeme mit Mehrbusstrukturen, wie sie in 5.1.3 und 5.1.6 beschrieben sind.

Eine weitere Steigerung der Rechnerleistung erhält man durch den Übergang zu echten *Mehrprozessorsystemen*, d.h. zu Systemen, bei denen mehr als nur ein Prozessor für die zentralen Verarbeitungsaufgaben zur Verfügung stehen. Hier gibt es im Prinzip zwei grundsätzliche Strukturkonzepte, die mit den Kürzeln SMP (symmetrical multiprocessing) und MPP (massive parallel processing) bezeichnet werden.

SMP-Systeme. SMP-Systeme bestehen aus mehreren gleichen Prozessoren, die sich den Hauptspeicher als globalen Speicher teilen und somit einen gemeinsamen, *globalen Adressraum* haben (*shared memory*). Die Kommunikation zwischen den Prozessoren erfolgt über diesen Speicher unter Benutzung z.B. eines gemeinsamen, parallelen Busses. Man spricht hier von *speichergekoppelten* oder *eng gekoppelten Mehrprozessorsystemen*. Das Betriebssystem läuft nur auf einem der Prozessoren, wobei es nicht unbedingt auf einen bestimmten Prozessor festgelegt ist (symmetrical multiprocessing). Die Parallelisierung der Anwendungen wird vom Betriebssystem vorgenommen, was den Programmierer bzw. Anwender von dieser Aufgabe entlastet. Es muß dementsprechend in der Lage sein, die bei Einprozessorsystemen im *Multiprogrammbetrieb* zeitlich nacheinander ausgeführten Prozesse auf die einzelnen Prozessoren zu verteilen, sofern eine parallele Ausführung möglich ist. Das hat den Vorteil, daß Anwendungen, die auf einem Einprozessorsystem laufen, ohne großen Aufwand auf ein Mehrprozessorsystem, also auf ein System höherer Leistungsfähigkeit übertragen werden können. Der Nachteil von SMP-Systemen ist jedoch ihre eingeschränkte *Skalierbarkeit*, da die Anzahl an Prozessoren durch den Engpaß des sie verbindenden Übertragungsmediums, z.B. eines gemeinsamen Busses, begrenzt wird. – SMP-Systeme als Systeme hoher Leistungsfähigkeit werden insbesondere als *Server* eingesetzt.

UMA-Architektur: Bild 5-12 zeigt eine erste SMP-Variante, bei der mehrere Prozessormodule, bestehend aus dem eigentlichen Prozessor mit On-chip-L1-Cache und einem L2-Cache, über einen gemeinsamen Bus Zugriff auf den globalen Hauptspeicher haben und über eine Bridge, z.B. Host-to-PCI, mit den Ein-/Ausgabeeinheiten verbunden sind. Bei SMP-fähigen Prozessoren sind die L2-Caches entweder als Frontside- oder Backside-Caches in den Prozessorbaustein integriert oder als Backside-Caches prozessorextern aufgebaut. Dementsprechend handelt es sich bei dem gemeinsamen Bus um den Prozessorbus, der jedoch für den SMP-Betrieb gewisse Funktionserweiterungen aufweist. So unterstützt er das Zusammenschalten mehrerer Prozessoren und ggf. eines weiteren Masters, z.B. eines DMA-Controllers am PCI-Bus, durch spezielle Signalleitungen für die Busarbitration. Dabei sieht er eine getrennte Arbitrierung für die Adreß- und die Datenleitungen vor, so daß sich die Adressierungs- und Datenübertragungsvorgänge der

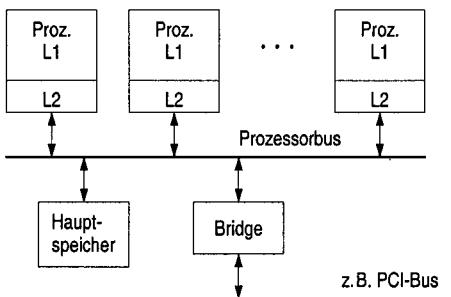


Bild 5-12. UMA-Mehrprozessorstruktur mit einem gemeinsamen Bus als Verbindungskonzept

Prozessoren als sog. *Split-Transaktionen* überlappen können (6.1.4). Dadurch wird der Engpaß, den ein (einzigster) gemeinsamer Bus bildet, entschärft. Die Datenübertragungen erfolgen dabei blockweise (Cache-fill- und Write-back-Operationen), und es können bei einer neuen Adressierung z.B. bis zu vier Datenblocktransporte ausstehen, die dann nachträglich ihren bereits früher ausgegebenen Adressen zugeordnet werden.

Die Gesamthardware eines solchen Systems ist auf nur einer Karte (SMP-Board) untergebracht und erlaubt jedem der Prozessoren aufgrund der symmetrischen Struktur einen gleich schnellen Hauptspeicherzugriff, weshalb solche Systeme als UMA-Architekturen (unified memory access) bezeichnet werden. Zur Gewährleistung der Datenkohärenz in den lokalen Caches wird z.B. das *MESI*-Kohärenzprotokoll eingesetzt (siehe dazu 6.2.4).

Die *Skalierbarkeit* eines Systems nach Bild 5-12 ist wegen des einen gemeinsamen Busses auf maximal vier bis acht Prozessoren beschränkt. Um eine größere Anzahl an Prozessoren einsetzen zu können, ist eine Verbindungsstruktur mit mehreren gleichzeitig benutzbaren Datenbussen erforderlich. Die gängige Technik ist hier der *Crossbar-Switch*, kurz Crossbar (Kreuzschienenverteiler). Diesen

kann man sich als eine $m \times n$ -Matrix vorstellen, in deren Kreuzungspunkten Datenwege zwischen m Prozessoren und n Speicher- oder Ein-/Ausgabeeinheiten durchschaltbar sind, wie in Bild 5-13 angedeutet. Auch hier haben wieder alle Prozessoren gleich schnelle Zugriffswege, womit ein solches System ebenfalls eine UMA-Architektur ist. Zur Realisierung des MESI-Kohärenzprotokolls gibt es einen zusätzlichen Bus für das Snooping (6.2.4).

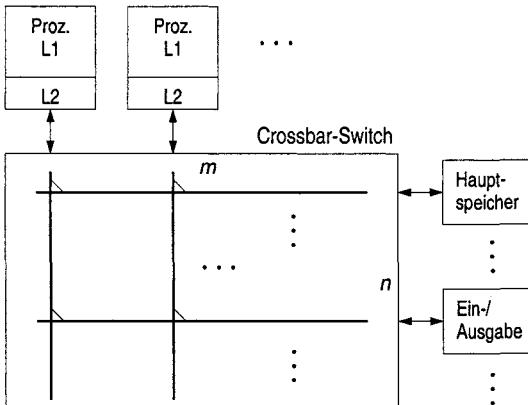


Bild 5-13. UMA-Mehrprozessorstruktur mit einem Crossbar-Switch als Verbindungsstruktur

In ihrem Aufbau bestehen Crossbar-Systeme meist aus einer zentralen Crossbar-Karte, auf die wiederum Karten mit den Master- und Slave-Komponenten gesteckt werden. Typisch ist es hierbei, die Steckkarten vollständig mit Prozessor-Cache-, Hauptspeicher- und Ein-/Ausgabeeinheiten zu bestücken, und den Crossbar als quadratische Matrix auszulegen mit je zwei Datenbuszugängen zu jeder Steckkarte (Master-Zugang und Slave-Zugang). In solchen Strukturen sind also der gemeinsame Hauptspeicher und die Ein-/Ausgabeeinheiten über alle Steckkarten verteilt. Die Skalierbarkeit ist durch den Aufwand für den Crossbar begrenzt und liegt bei z.B. 64 Prozessoren. Die Datenwege haben ggf. geringere Breite als der Prozessorbus. – Bei einer Vereinfachung der Crossbar-Struktur sind zu einem bestimmten Zeitpunkt zwar mehrere Adreßdurchschaltungen auf mehreren Adreßbussen, jedoch nur eine Datendurchschaltung als *Punkt-zu-Punkt*-Durchschaltung möglich. Das heißt, der Aufwand an Datenwegen wird so gering wie möglich gehalten. Der eine Datenweg wird allerdings bestmöglich genutzt, indem er immer nur für die Takte der eigentlichen Datenübertragung einem Master-Slave-Paar zugeschaltet wird.

NUMA-Architektur. Eine Erhöhung der *Skalierbarkeit* erreicht man auch mit einfacheren Mitteln als mit einem Crossbar, indem man mehrere SMP-Boards entsprechend Bild 5-12 über ein spezielles Interface (Link) mit hoher Übertragungsrate miteinander verbindet, wie in Bild 5-14 gezeigt. Ein solches Interface ist z.B. das aus logischer Sicht busähnliche *Scalable Coherent Interface* (SCI, ANSI/ISO/

IEEE 1596-1992), das physisch als unidirektionale Punkt-zu-Punkt-Verbindung ausgelegt ist. Es wird entweder bitseriell oder 16-Bit-parallel mit Übertragungsraten von 1 Gbit/s bzw. 1 Gbyte/s realisiert.

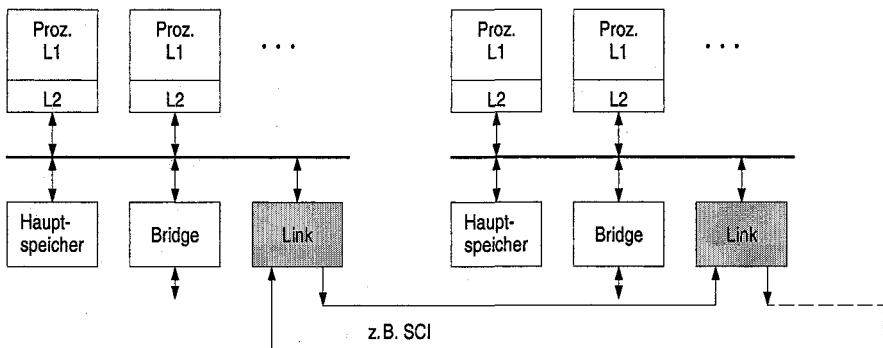


Bild 5-14. NUMA-Mehrprozessorstruktur, bestehend aus mehreren UMA-Systemen gemäß Bild 5-12 mit einer busähnlichen Verbindung zwischen den Einzelsystemen

Der Hauptspeicher, der bei einem solchen System immer noch einen *globalen Adressraum* bildet, ist jetzt auf die einzelnen SMP-Boards verteilt, d.h., es liegt eine asymmetrische Speicherstruktur vor, was zur Bezeichnung *DSM-System* führt (*distributed shared memory*). Der Nachteil dieser Asymmetrie ist, daß Speicherzugriffe der Prozessoren unterschiedlich schnell sind, je nachdem, ob der Zugriff auf die eigene Karte oder über das Interface auf eine andere Karte erfolgt. Man bezeichnet eine solche Architektur deshalb auch als NUMA-Architektur (non-uniform memory access). Als weiterer Nachteil dieser Asymmetrie kann das MESI-Protokoll nur auf die einzelnen Karten bezogen eingesetzt werden, kartenübergreifend kommen zusätzlich sog. verzeichnisorientierte Protokolle zum Tragen (directory based coherence protocols). In den SCI-Controllern (im Bild mit Link bezeichnet) werden dazu Tabellen geführt, die angeben, auf welcher Karte und in welchem Cache das jeweils aktuelle Datum gerade gespeichert ist. Dies führt zu der Bezeichnung *cc-NUMA* (cache coherent NUMA).

COMA. Um bei NUMA-Systemen nach Möglichkeit nur die kürzeren, lokalen Speicherzugriffe zu haben, muß das Betriebssystem dafür sorgen, daß sich die lokal benötigten Daten auch jeweils in den Hauptspeichereinheiten des betreffenden SMP-Boards befinden. Gelingt dies nicht, müssen also Daten verarbeitet werden, die auf anderen Karten gespeichert sind, so hängt die optimale Arbeitsweise davon ab, daß der jeweilige *L2-Cache* genügend groß ist, um den aktuell benötigten Datensatz zu puffern. Da L2-Caches jedoch in SRAM-Technik, und nicht wie der Hauptspeicher in DRAM-Technik aufgebaut sind, lassen sie sich aufgrund des damit verbundenen höheren Aufwands nicht mit „beliebig“ großer Speicherkapazität implementieren.

Man löst dieses Problem dadurch, daß man den lokal verteilten Hauptspeicher jeweils in Teilen oder insgesamt als eine dritte Cache-Ebene, d.h. als *L3-Cache*, realisiert. Als gemeinsamer Speicher fungiert dann der verbleibende gemeinsame Hauptspeicher bzw. der Hintergrundspeicher als virtueller Speicher (6.3.2), z.B. ein Verbund von Festplattenspeichern. Eine solche Architektur wird als COMA (cache only memory architecture) bezeichnet. Um den für die Adressierung der L3-Caches erforderlichen hohen Hardwareaufwand an Tag-Speichern und Vergleicherlogik einzusparen, führt man die Cache-Verwaltung üblicherweise auf Seitenbasis durch und benutzt zur Seitenverwaltung die in den Prozessoren sowieso vorhandenen Speicherverwaltungseinheiten. Man bezeichnet eine solche Architektur dann als Simple COMA.

MPP-Systeme. MPP-Systeme bestehen aus einer Vielzahl von Rechnerknoten, die über ein *Verbindungsnetz* miteinander kommunizieren. Ein solcher Knoten ist im einfachsten Fall ein Prozessor-Speicher-Paar, meist aber sind diese Knoten als SMP-Systeme, d.h. als symmetrische Mehrprozessorsysteme ausgelegt. Jeder dieser Knoten hat seinen eigenen *lokalen Adreßraum* (Hauptspeicher) und sein eigenes Betriebssystem. Es gibt keinen gemeinsamen Adreßraum über die Knoten hinweg (*distributed memory*) und damit keinen direkten Zugriff eines Knotenprozessors auf den Speicher eines anderen Knotens. Vielmehr erfolgt die Kommunikation durch den Austausch von Nachrichten (*message passing*), wozu im einfachsten Fall *Nachrichtenbusse* (siehe auch 5.1.3) oder aber ggf. sehr komplexe Verbindungsnetze eingesetzt werden. Sie erfolgt nach festgelegten Protokollen und wird, um die Prozessoren zu entlasten, durch spezielle Kommunikationshardware unterstützt. Man spricht hier von *nachrichtengekoppelten* oder *lose gekoppelten Mehrprozessorsystemen*.

Der Vorteil von MPP-Systemen liegt in deren guter *Skalierbarkeit*: im Prinzip sind beliebig viele Knoten zusammenschaltbar. Gebräuchlich sind Systeme mit bis zu mehreren tausend Prozessoren (*massive parallel processing!*), häufig standardisierte Mikroprozessoren. Einer der leistungsfähigsten „Supercomputer“ dieser Art ist derzeit (2004) der ASCI-Q (Hewlett Packard), bestehend aus zwei zusammengeschalteten MPP-Segmenten mit je 1024 SMP-Rechnerknoten (HP-Server) mit je 4 Alpha-Prozessoren, d.h. mit insgesamt 8192 Prozessoren. Übertrouffen wird seine Leistungsfähigkeit nur noch durch MPP-Systeme, die auf sog. Vektorprozessoren basieren, z.B. durch den sog. Earth Simulator (NEC) mit derzeit 640 SMP-Rechnerknoten mit je acht Vektorprozessoren, d.h. insgesamt 5120 Prozessoren. MPP-Systeme haben aber auch Nachteile, nämlich die schwierige Programmierung, indem die Parallelisierung von Anwendungsprogrammen weitgehend dem Programmierer obliegt. Eingesetzt werden MPP-Systeme vorwiegend im wissenschaftlichen Bereich zur Simulation komplexer Vorgänge. – Auf Details von MPP-Systemen, z.B. auf die Techniken der Verbindungsnetze und der Kommunikation, soll hier nicht eingegangen werden, da dies zum Thema *Supercomputing* gehört.

5.1.8 Industrielle Busse (Multibus II, VMEbus)

Multibus II. Der Multibus-II-Standard basiert auf dem von Intel entwickelten Multibus I, der dann von Intel und anderen Firmen weiterentwickelt wurde. Er umfaßt insgesamt drei verschiedene *Backplane-Busse* und ist für den Aufbau modularer Systeme mit industrialem Einsatz gedacht [Intel 1984]. Der Aufbau erfolgt in einem Baugruppenträger (19-Zoll-Rahmen), in dem bis zu 20 Steckkarten im doppelten Europaformat mittels der drei Backplane-Busse miteinander verbunden werden (siehe auch 5.1.2). Zentraler Bus ist der *iPSB-Bus* (Intel Parallel System Bus), der als 32-Bit-paralleler Multiplexbus an alle Stecksockel geführt ist. Er wird ergänzt durch den nicht gemultiplexten *iLBX-II-Bus* (Local Bus Extension), einen 32-Bit-Speicherbus. Dieser erlaubt den Zugriff auf zusätzliche Speichereinheiten außerhalb der CPU-Karte, und zwar mit ähnlich hohen Übertragungsraten, wie sie für den lokalen Speicher auf der CPU-Karte möglich sind. Die Busleitungen werden dazu kurz gehalten und als Stichleitungen nur zwischen benachbarten Stecksockeln geführt. Der dritte Bus, der *iSSB-Bus* (Serial System Bus), ist ein serieller *Nachrichtenbus*, der einerseits als Backplane-Bus ausgelegt ist, andererseits aber auch als Kabelbus mit bis zu 10 m Länge Baugruppenträger miteinander verbindet.

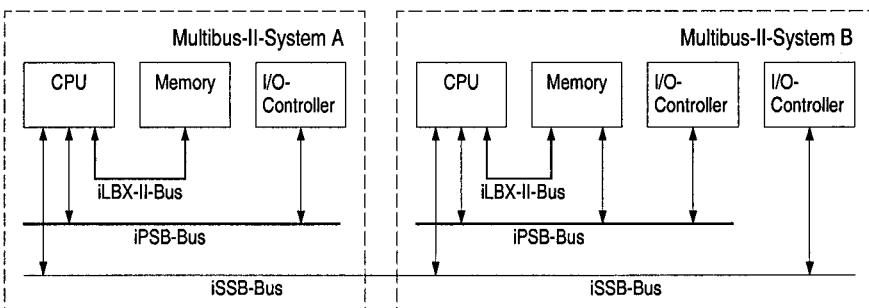


Bild 5-15. Zwei Multibus-II-Systeme mit jeweils einem eigenen 32-Bit-parallelen iPSB-Bus als Systembus, einem eigenen 32-Bit-parallelen iLBX-II-Bus als Speicherbus und einem gemeinsamen seriellen iSSB-Bus als Nachrichtenbus (in Anlehnung an [Intel 1984])

Bild 5-15 zeigt ein Beispiel für eine Mehrbusstruktur mit zwei eigenständigen Multibus-II-Systemen A und B, bestehend aus drei bzw. vier Steckkarten. Diese sind entweder gemeinsam in einem oder getrennt in zwei Baugruppenträgern untergebracht und kommunizieren mittels des iSSB-Busses miteinander. Die mit „CPU“ bezeichneten Steckkarten sind u.U. relativ komplex und können ein eigenes Rechnersystem mit Prozessor, Speicher und Ein-/Ausgabe-Interfaces bilden. Die mit „I/O-Controller“ bezeichneten Karten können ähnlich aufgebaut sein und als Controller-Komponenten ggf. einen eigenen Prozessor und einen DMA-Controller haben. Die „Memory“-Karte im System A ist über den iLBX-Bus dem Prozessor der CPU-Karte A als lokaler Speicher ausschließlich zugeordnet, die

„Memory“-Karte im System B ist für die dortigen CPU-Karte lokal über den iLBX-Bus und für den einen der beiden I/O-Controller global über den iPSB-Bus zugänglich. – Die in Tabelle 5-1 (S. 280) für den iPSB-Bus angegebene maximale Übertragungsrate von 40 Mbyte/s basiert auf dem beliebig langen Burst mit minimal einem Takt pro Datenübertragung unter Vernachlässigung des Adreßtransports (siehe auch Bild 5-7, S. 278, Mux-Bus: long burst cycle).

VMEbus. Der VMEbus (Versa Module Europe Bus) ist wie der Multibus II ein genormter *Backplane-Bus*, der auf dem von Motorola für den *MC68000-Prozessor* entwickelten 16-Bit-VERSAbus basiert und 1981 von Motorola und anderen Firmen für die in Europa gebräuchlichen Kartenmaße und Steckverbinder entwickelt wurde. Er baut dementsprechend auf dem 19-Zoll-Baugruppenträger auf und sieht 21 Steckplätze für Steckkarten im einfachen und doppelten Europaformat vor. In seiner ursprünglichen Ausführung wurde er als nicht gemultiplexter, asynchroner 32-Bit-Systembus spezifiziert (VMEbus-Standard, [Motorola 1985]). Er wurde dann 1989 von der VITA (VMEbus International Trade Association) revidiert, indem die 32 Adreßleitungen für eine 64-Bit-Datenübertragung hinzugenommen wurden, um den von ihm unterstützten Blockbuszyklus mit höherer Übertragungsrate nutzen zu können (VME64-Standard). Die bisherige 32-Bit-Adressierung wurde aber zunächst beibehalten. Bei einer erneuten Revision wurde er schließlich zum „echten“ 64-Bit-Multiplexbus mit 64-Bit-Adressierung erweitert. Um die Übertragungsraten für Blocktransfers weiter zu steigern, gab es zwei weitere Überarbeitungen der Spezifikation, zunächst hin zur Übertragung mit Double-Data-Rate (VME64x-Standard: VITA 1.1-1997) und dann hin zu einem synchronen Protokoll bei veränderter Verdrahtung in der Backplane (VME320-Standard, VITA 1.5-1999).

Die in Tabelle 5-1, S. 280, für den VMEbus angegebenen maximalen Übertragungsraten von 40, 80, 160 und 320 Mbyte/s spiegeln die vier verschiedenen Spezifikationen wider und basieren auf dem langen Blockbuszyklus mit maximal 256 Datentransfers (im ersten Fall maximal 256 Bytes) bei minimal einem Takt pro Datenübertragung und unter Vernachlässigung des Adreßtransports.

1. VMEbus: 40 Mbyte/s ergeben sich aus dem Bustakt von 10 MHz und der Busbreite von 32 Bit. Master und Slave synchronisieren sich mittels zweier Handshake-Signale, wie in 5.3.2 in Anlehnung an den MC68020 beschrieben (\overline{DS} , \overline{DTACK}). Das heißt, die insgesamt vier Flankenwechsel beider Signale erfolgen in gegenseitiger Abhängigkeit (*four-edge handshake*, Protokoll BLT: block transfer).
2. VME64: 80 Mbyte ergeben sich aus der 64-Bit-Erweiterung. Die Synchronisation erfolgt wie oben (Protokoll MBLT: multiplexed block transfer, siehe auch Bild 5-7, Mux-Bus: long burst cycle).
3. VME64x: 160 Mbyte/s ergeben sich durch eine geänderte Synchronisation, indem die jeweils zweite Flanke der beiden Handshake-Signale für die zweite

Übertragung innerhalb eines Takts genutzt wird (Protokoll 2eVME: *two-edge transfer*, siehe auch Bild 5-7, Mux-Bus: long burst cycle, DDR).

4. VME320: 320 Mbyte/s erreicht man schließlich durch ein synchrones Protokoll, indem jeweils das Handshake-Signal des Senders als Bustaktsignal genutzt wird und auf eine Quittierung durch das Handshake-Signal des Empfängers verzichtet wird (Protokoll 2eSST: *source-synchronous transfer*). Hintergrund des verwendeten Protokolls ist, daß die maximal erreichbare Übertragungsrate nicht von der Signallaufzeit (*propagation delay*) zwischen Signalquelle und Signalziel, sondern durch die Laufzeitunterschiede (*skew*) der Signale durch die Signaltreiber, die Backplane und die Signalempfänger bestimmt ist. Diese Unterschiede wurden so weit reduziert (u.a. durch eine veränderte Backplane), daß die maximale Übertragungsrate gegenüber VME64x verdoppelt werden konnte.

Der VMEbus ist ein anschauliches Beispiel dafür, die technischen Möglichkeiten aufzuzeigen, mit denen Busse, die aus der Anfangszeit der Mikroprozessortechnik stammen, an die im Laufe der Zeit gestiegenen Anforderungen an eine höhere Übertragungsleistung angepaßt werden konnten. – Zum Aufbau des VME-Busses siehe auch 5.1.4.

Wie beim iPSB-Bus gibt es auch zum VMEbus ergänzende Spezifikationen für einen *Speicherbus*, hier als *VMXbus* bezeichnet (VME Extended Bus), und für einen seriellen *Nachrichtenbus*, hier als *VMSbus* bezeichnet (VME Serial Bus). Der VMXbus überträgt die Daten 32-Bit-breit und benutzt eine 24-Bit-Adressierung. Es stehen jedoch nur 12 Adreßleitungen zur Verfügung, die für den Adreßtransport gemultiplext werden. Die Busverbindungen zwischen den Karten werden per Flachbandkabel hergestellt. Der VMSbus ist mit seinen Signalen im VMEbus definiert; er erlaubt Übertragungsraten von 200 bis 400 Mbyte/s. – Als Beispiel für eine Mehrbusstruktur können die in Bild 5-15 angegebenen Multi-bus-II-Busbezeichnungen gegen die des VME-Busses und seiner Zusatzbusse ausgetauscht werden.

5.2 Adressierung der Systemkomponenten

Voraussetzung für die Durchführung eines Datentransports zwischen dem Mikroprozessor und einer der Systemkomponenten, z.B. einer Speichereinheit oder einer Interface-Einheit, ist die Anwahl der Einheit durch den Adressierungsvorgang. Die vom Mikroprozessor auf den Adreßbus ausgegebene Adresse wird dazu von allen am Systembus angeschlossenen Einheiten ausgewertet. Eine der Einheiten identifiziert sich mit dieser Adresse und schaltet sich auf den Datenbus auf. Im folgenden werden die verschiedenen Möglichkeiten und Techniken der Adressierung sowie die Ankopplung von Systemkomponenten an den Bus behandelt.

5.2.1 Isolierte und speicherbezogene Adressierung

Für die Adressierung der für die Ein-/Ausgabe verwendeten Systemkomponenten sind zwei verschiedene Techniken üblich, die als *isiolerte* und *speicherbezogene Adressierung* oder – bezogen auf den Datentransport – auch als *isiolerte* und *speicherbezogene Ein-/Ausgabe* bezeichnet werden.

Bei der isolierten Ein-/Ausgabe (isolated input-output) sind die Adreßräume der Speicher- und Ein-/Ausgabeeinheiten voneinander getrennt (isiert). Der Datentransport mit den Interface-Einheiten wird mit speziellen *Ein-/Ausgabebefehlen*, wie INP (input) und OUT (output), durchgeführt, wobei die Adresse des Quell- bzw. Zielregisters des Interfaces im Adreßteil des Ein-/Ausgabebefehls steht. Diese Adresse wird wie eine Speicheradresse auf den Adreßbus ausgegeben. Die Unterscheidung, ob es sich um eine Speicheradresse oder eine Interface-Adresse handelt, zeigt eine Steuerleitung M/I \bar{O} (memory/input-output) an. Auf dieser wird in Abhängigkeit des Operationscodes – Speicherzugriffsbefehl oder Ein-/Ausgabebefehl – vom Prozessor eine 1 bzw. eine 0 ausgegeben und den Chip-Select-Eingängen CS der Systemkomponenten zugeführt (Bild 5-16). Während der Prozessor für die Speicheradressierung die volle Adreßlänge von z.B. 32 Bit nutzt, verwendet er für die Ein-/Ausgabeaddressierung z.B. nur 16 Bit (kürzere Adressen bedeuten kürzere Befehle). Es stehen so zwei Adreßräume mit 4 Gbyte und 64 Kbyte zur Verfügung (Bild 5-17a).

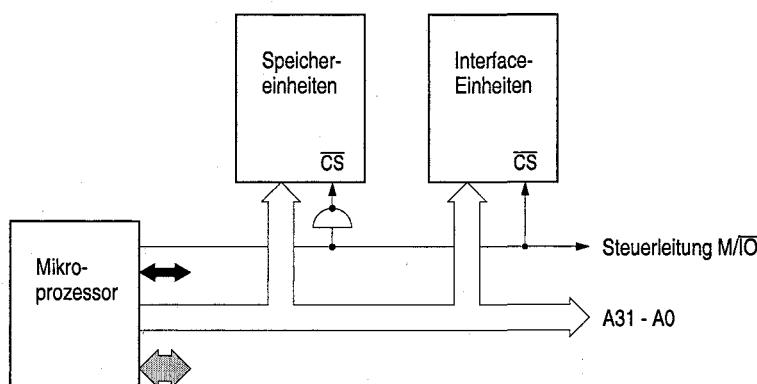


Bild 5-16. Isolierte Adressierung

Bei der speicherbezogenen Ein-/Ausgabe (memory mapped input-output) sind keine besonderen Ein-/Ausgabebefehle notwendig. Die Register der Interface-Einheiten werden genauso wie Speicherzellen adressiert; damit entfällt in Bild 5-16 die Steuerleitung M/I \bar{O} . Für Interfaces und Speicher gibt es dementsprechend auch nur einen einzigen, gemeinsamen Adreßraum, der nach Speicher- und Ein-/Ausgabeadreßbereichen aufgeteilt wird (Bild 5-17b). Der Vorteil dieser Adressierungstechnik liegt darin, daß sämtliche Befehle mit Speicherzugriff, bei

CISCs also auch die arithmetischen, die logischen und die Vergleichsbefehle, auf die Interface-Register angewendet werden können. – In beiden Fällen wird die Aufteilung der Adreßräume in *Adreßraumbelegungsplänen (memory maps)* festgehalten, in denen zusammenhängende Speicherbereiche durch ihre Anfangs- und Endadressen und die Interface-Einheiten mit ihren Registeradreßbereichen eingetragen sind.

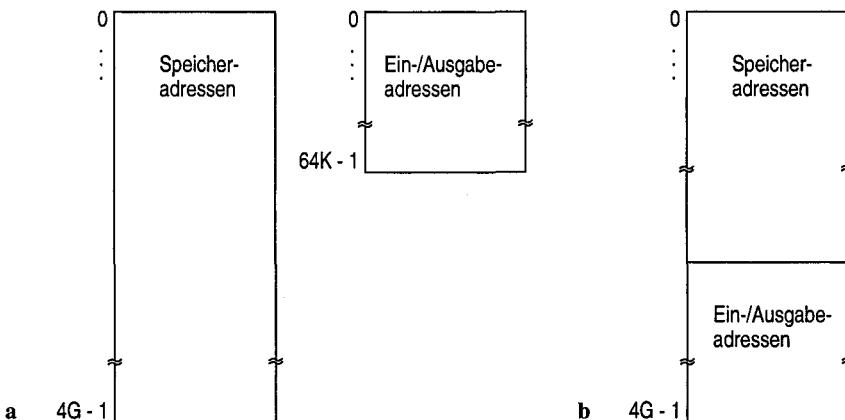


Bild 5-17. Adreßraumvergabe bei einer Adreßlänge von 32 Bits. a Isolierte Adressierung mit reduzierter Adreßlänge für die Ein-/Ausgabe, b speicherbezogene Adressierung

5.2.2 Karten-, Block- und Bausteinanwahl

Beim Entwurf eines Mikroprozessorsystems müssen den Bausteinen Adressen aus dem verfügbaren Adreßraum zugeordnet werden. Die Anzahl der benötigten Adressen kann dabei sehr unterschiedlich sein. So belegt z.B. ein Speicherbaustein mit 1 M Speicherplätzen 2^{20} Adressen, während ein Interface-Baustein mit wenigen Registern meist mit zwei bis 16 Adressen auskommt.

Bei einem Mehrkartensystem, in dem die einzelne Karte mehrere Bausteine umfassen kann und diese wiederum zu Blöcken zusammengefaßt sein können, ergibt sich eine Adressierungshierarchie, bestehend aus Kartenebene, Blockebene, Baustinebene und der Ebene der Speicherplatz- und Registeradressen. Die jeweilige Anzahl an Komponenten wird üblicherweise in Zweierpotenzen festgelegt, wodurch sich die Adressierung der einzelnen Komponenten durch eine Unterteilung der Adresse in Felder vornehmen lässt. (Der Begriff Karte steht hier stellvertretend für eine komplexere Funktionseinheit, z.B. eine Speichereinheit, die jedoch nicht unbedingt als steckbare Karte realisiert sein muß, sondern auch Bestandteil der Grundkarte eines Mikroprozessorsystems sein kann.)

Man bezeichnet diese Art der Adressierung auch als *codierte Adressierung*, da bei ihr jedes Codewort eines Feldes zur Komponentenwahl genutzt werden kann. Die codierte Adressierung erlaubt dementsprechend die volle Nutzung des Adreßraums, erfordert jedoch zur Entschlüsselung der Adreßfelder Decodier- oder Vergleicherbausteine, die üblicherweise auf der Karte selbst untergebracht sind. Bild 5-18 zeigt zwei Beispiele für die Aufteilung einer 32-Bit-Adresse: (a) für eine 16-Mbyte-Speicherkarte, bestehend aus vier Blöcken mit je 1 M Wörtern (siehe auch 6.1.2) und (b) für eine Interface-Karte, bestehend aus acht Interface-Bausteinen mit jeweils bis zu 16 8-Bit-Registern.

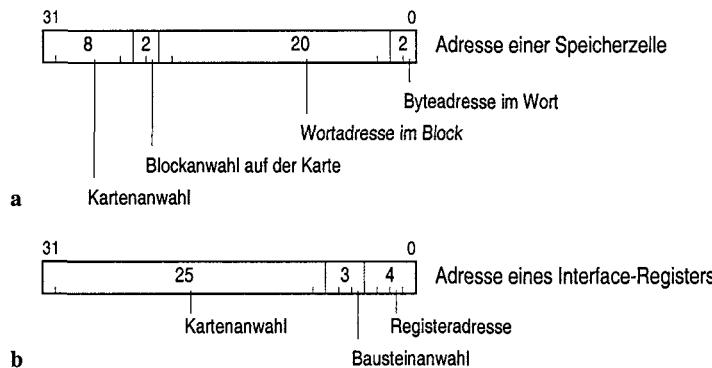


Bild 5-18. Adreßunterteilung bei codierter Karten- und Block- bzw. Bausteinanwahl. **a** 16-Mbyte-Speicherkarte, **b** Interface-Karte

Bild 5-19 zeigt ein Schaltungsbeispiel für die 16-Mbyte-Speicherkarte. Die Kartenanwahl erfolgt über eine Vergleichslogik, die die 8 Adreßbits A31 bis A24 mit einer fest vorgegebenen 8-Bit-Kartenadresse vergleicht. Stimmen beide Adressen überein, so wird das Ausgangssignal \overline{SEL} aktiviert, das seinerseits als Kartenanwahlsignal einen 1-aus-4-Decodierer über seinen Chip-Enable-Eingang \overline{CE} aktiviert. Dem Decodierer wird die 2-Bit-Blockadresse A23, A22 zugeführt, die genau einen seiner vier Ausgänge aktiviert. Dieses Ausgangssignal dient als Chip-Select-Signal \overline{CS} für einen der vier Speicherblöcke. Die Anwahl eines Wortes im Block erfolgt durch die 20 Adreßbits A21 bis A2, die zwar allen vier Speicherblöcken zugeführt, jedoch nur von dem aktivierte Block (genauer: von den bausteininternen Adreßdecodierern der Speicherbausteine dieses Blocks) ausgewertet werden. Die Auswertung der Adreßbits A1 und A0 erfolgt zusammen mit der Auswertung der Datenformatangabe Byte, Halbwort oder Wort (in Bild 5-19 nicht dargestellt; siehe 5.2.3). – Je nach Vorgabe der 8-Bit-Kartenadresse kann diese Speicherkarte einem der 28 16-Mbyte-Bereiche des Adreßraums zugeordnet werden, deren Bereichsgrenzen bei einem ganzzahligen Vielfachen von 16 Mbyte liegen.

Der codierten Adressierung steht die *uncodierte Adressierung* gegenüber. Bei ihr erfolgt die Anwahl der Komponenten durch einzelne Adreßbits, wobei der Ein-

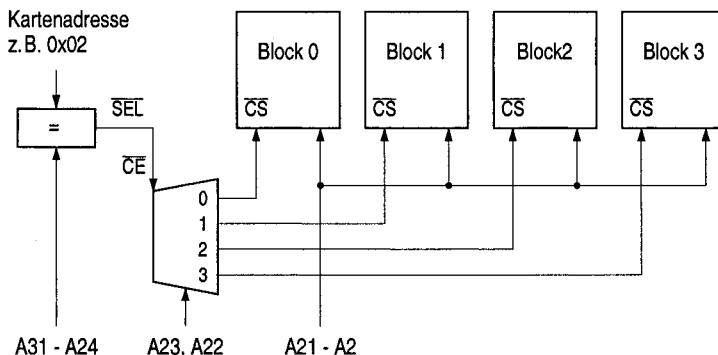


Bild 5-19. Adreßdecodierung für eine 16-Mbyte-Speicherkarte mit der Adreßunterteilung nach Bild 5-18a (Adreßbereich 32 M bis 48 M-1, vorgegeben durch die Kartenadresse 0x02)

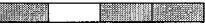
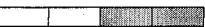
deutigkeit halber immer nur eines dieser Bits gesetzt sein darf. Man benötigt hier zwar keine Adreßdecodierlogik, dafür wird der Adreßraum schlecht genutzt. Die uncodierte Adressierung wird deshalb kostensparend bei kleineren, nicht erweiterbaren Systemen, meist Einkartensystemen, eingesetzt. Sie wird aber auch neben der codierten Adressierung innerhalb von Teilbereichen des Adreßraums eingesetzt, z.B. bei der Anwahl von Interface-Bausteinen innerhalb einer Interface-Karte.

5.2.3 Byte-, Halbwort- und Wortanwahl

Data-Alignment und Data-Misalignment. Der Datenzugriff in einem Speicher mit dem Byte als kleinstem Datenformat setzt voraus, daß jedes Byte eines Speicherwortes einzeln adressierbar ist. Hierzu sind vier Byte-Enable-Signale \overline{BE}_3 bis \overline{BE}_0 erforderlich, die je nach Hersteller entweder prozessorintern oder prozessorextern durch eine Ansteuerlogik erzeugt werden. Ausgewertet werden dazu die Adreßbits A1 und A0 sowie die Datenformatangabe im auslösenden Befehl. Bei externer Logik wird das Datenformat vom Prozessor durch zwei Steuersignale SIZ1 und SIZ0 angegeben. Tabelle 5-4 zeigt die möglichen Belegungen dieser Signale unter der Voraussetzung des Data-Alignment ($A_0 = 0$ bei Halbwörtern und $A_1, A_0 = 00$ bei Wörtern; siehe auch 2.1.2). – Um das rechtsbündige Speichern von Bytes und Halbwörtern in den Prozessorregistern zu gewährleisten, und zwar unabhängig von ihren Positionen im Speicherwort bzw. auf dem Datenbus, besitzt der Prozessor eine Verschiebelogik (barrel shifter), die die korrekte Zuordnung herstellt.

Bei Zugriffen mit Data-Misalignment – das betrifft die in Tabelle 5-4 nicht aufgeführten Kombinationen A1, A0 bei den Datenformaten Halbwort und Wort – werden zunächst nur die Byte-Enable-Signale jener Bytes aktiviert, die in einem ersten Buszyklus transportiert werden können, um dann in einem weiteren Zyklus

Tabelle 5-4. Speicheranwahl für Byte-, Halbwort- und Wortoperanden bei Data-Alignment (Byte- und Halbwortadressierung von links nach rechts, d.h. Big-endian-Byteanordnung)

Datenformat		Adresse		Byte-Enable				
	SIZ1 SIZ0	A1 A0		BE3	BE2	BE1	BE0	Speicherwort/Datenbus
Byte	0 1	0 0		0	1	1	1	
	0 1	0 1		1	0	1	1	
	0 1	1 0		1	1	0	1	
	0 1	1 1		1	1	1	0	
Halbwort	1 0	0 0		0	0	1	1	
	1 0	1 0		1	1	0	0	
Wort	0 0	0 0		0	0	0	0	

die Enable-Signale der restlichen Bytes zu aktivieren und diese Bytes zu transportieren. Dementsprechend muß die Tabelle 5-4 um diese Kombinationen von A1 und A0 erweitert werden. Hinzu kommt die SIZ1-SIZ0-Codierung 11 für die 3-Byte-Einheit als Teil eines Wortes. (SIZ1 und SIZ0 geben jeweils die Anzahl der noch zu transportierenden Bytes an.)

Dynamische Busbreite. Bei dynamischer Busbreite (*dynamic bus sizing*) können die am Datenbus angeschlossenen Einheiten wahlweise eine Torbreite (port size) von 32, 16 oder 8 Bit haben. Unabhängig von der Torbreite versucht der Prozessor bei einem Datentransport zunächst – wie oben beschrieben – alle innerhalb der Wortgrenze liegenden Bytes zu transportieren. Im Verlaufe des Buszyklus teilt ihm dann die adressierte Einheit durch zwei Steuersignale BW1 und BW0 (bus width) ihre Anschlußbreite mit. Der Prozessor weiß damit, welche Bytes er bei einem Lesezyklus vom Bus tatsächlich übernehmen darf, bzw. welche Bytes ihm bei einem Schreibzyklus tatsächlich abgenommen worden sind. In einem oder mehreren weiteren Buszyklen verfährt er mit den jeweils verbleibenden Bytes in gleicher Weise. So können sich z.B. unter Einbeziehung des Data-Misalignment bei einem Worttransport zwischen einem und vier Buszyklen ergeben. Abhängig davon, ob der Prozessor zuerst die höherwertigen oder die niederwertigen Bytes überträgt, erfolgt der Anschluß von 16- und 8-Bit-Einheiten entweder linksbündig oder rechtsbündig am Datenbus.

Da der Prozessor jeden dieser Transporte unabhängig von der Busbreitenangabe im vorangegangenen Zyklus durchführt, müssen Bytes und Halbwörter beim Schreibzyklus, abhängig von den Signalen SIZ1, SIZ0, A1 und A0, von der prozessorinternen Verschiebelogik zum Teil vorsorglich in mehrere mögliche Buspositionen kopiert werden.

Bild 5-20 zeigt dazu ein Beispiel für einen 32-Bit-Transport zwischen einem Prozessorregister R_i und einem linksbündig an den Datenbus angeschlossenen 16-Bit-Speicher bei ungerader Speicheradresse ($A_1, A_0 = 01$, Data-Misalignment). Der Prozessor, der die Speicherbreite vorab nicht kennt, belegt den Datenbus für den ersten Buszyklus derart, daß ein 32-Bit-Port die Bytes 3, 2 und 1 und ein 16- oder 8-Bit-Port das Byte 3 übernehmen können. Für den 8-Bit-Port dupliziert dazu die Verschiebelogik das Byte 3 und legt es an die höchstwertige Busposition (siehe Indizes B, H und W für die möglichen Torbreiten). Der 16-Bit-Speicher antwortet während eines ersten Buszyklus mit der Anschlußbreite Halbwort ($BW_1, BW_0 = 10$) und übernimmt dementsprechend das Byte 3. In einem zweiten Buszyklus zeigt der Prozessor mit den Signalen SIZ_1 und SIZ_0 an, daß noch 3 Bytes zu übertragen sind. Ausgehend von der durch den ersten Buszyklus veränderten Adresse ($A_1, A_0 = 10$) bietet er jedoch nur die Bytes 2 und 1 an; für einen 16-Bit-Port vorsorglich linksbündig, für einen 32-Bit-Port vorsorglich rechtsbündig. Diese werden vom Speicher übernommen. In einem dritten Buszyklus wird schließlich das Byte 0 übertragen.

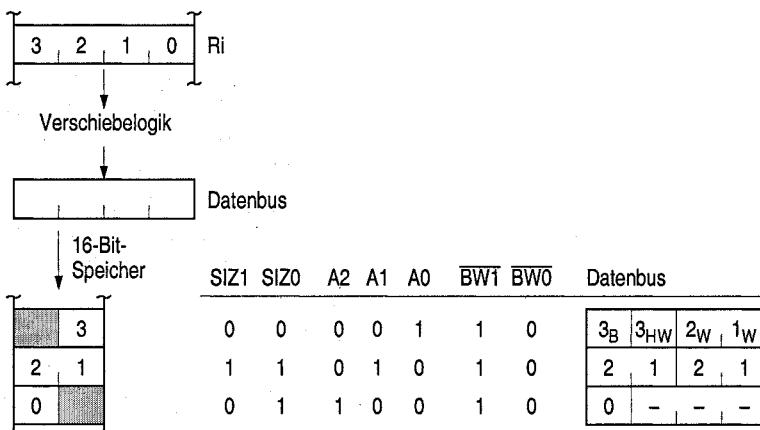


Bild 5-20. 32-Bit-Transport mit einem 16-Bit-Speicher bei Data-Misalignment; durchgeführt in drei Buszyklen (in Anlehnung an den MC68020 [Motorola 1989])

Auch Prozessoren ohne Dynamic-Bus-Sizing erlauben den Einsatz von Speicher- oder Interface-Einheiten mit reduzierter Torbreite von 8 oder 16 Bit. Bei ihnen besteht jedoch der Nachteil, daß Zugriffe auf aufeinanderfolgend gespeicherte Daten mit Sprüngen in der Adreßzählung verbunden sind. So kann eine 8-Bit-Einheit bei einem 32-Bit-Bus nur mit jeder vierten Byteadresse und eine 16-Bit-Einheit nur mit jeder zweiten Byteadresse adressiert werden. – Eine weitere Variante findet man beim *Pentium* als 64/32-Bit-Prozessor [Intel 1993]. Er richtet sich mit seinen Buszyklen nicht nach Torbreiten, sondern überläßt es der externen Hardware, diese bei schmaleren Einheiten mit 32, 16 oder 8 Bit in mehrere aufeinanderfolgende Teilzugriffe zu zerlegen. Das Dynamic-Bus-Sizing findet also

prozessorextern statt. Die externe Hardware hat dazu eine Sammel- und Verteillogik (assembly/disassembly logic) für insgesamt acht Bytes und je eine Byte-Vertauschungslogik für jede der Torbreiten (byte swap logic).

5.2.4 Big-endian- und Little-endian-Byteanordnung

Wie in 2.1.2 beschrieben, erfolgt die Adressierung von Daten, die mehr als ein Byte umfassen, je nach Prozessorrealisierung entweder mit Big-endian-Byteanordnung, d.h., die Datenadresse bezieht sich auf das höchstwertige Datenbyte, oder mit Little-endian-Byteanordnung, d.h., die Datenadresse bezieht sich auf das niedrigstwertige Datenbyte. Dies soll folgendes Beispiel eines Datensatzes veranschaulichen, der so vereinbart ist, daß die einzelnen Daten ausgerichtet gespeichert werden (*data alignment*).

	ALIGN.W		
X	DC.H	0x1112	Halbwort
	ALIGN.W		
Y	DC.W	0x21222324	Wort
T	DC.B	'A', 'B', 'C', 'D', 'E'	Byte-String
	ALIGN.H		
Z	DC.H	0x3132	Halbwort

Speicherdarstellung. Bild 5-21 zeigt den Datensatz in beiden Speicherdarstellungen, in Teilbild a für einen Prozessor mit Big-endian-Byteanordnung und in Teilbild b für einen Prozessor mit Little-endian-Byteanordnung, im folgenden mit den Bezeichnungen BE und LE abgekürzt. Beide Darstellungen sind mit Offset-Angaben versehen (Zahlen 0 bis 15), die ab der Adresse X zählen und die Adressierung aufeinanderfolgender Bytes zeigen. Die zur Adressierung der einzelnen Daten X, Y, T und Z verwendeten Offsets sind durch Kursivdruck hervorgehoben. Bei dem mit T benannten Text wurde dabei davon ausgegangen, daß auf die Zeichen (Bytes) einzeln zugegriffen wird.

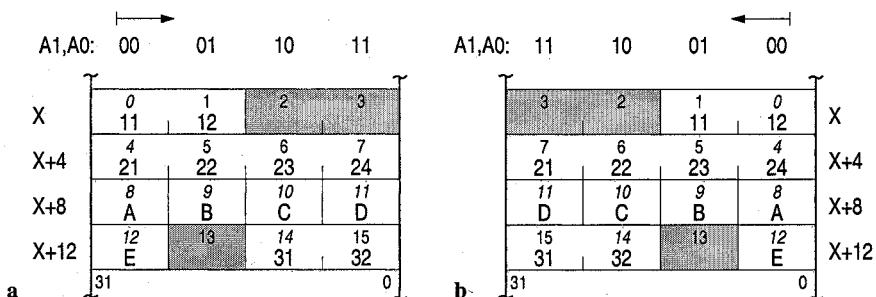


Bild 5-21. Speicherdarstellung eines ausgerichteten Datensatzes mit a Big-endian-Byteanordnung (BE), b Little-endian-Byteanordnung (LE)

Beide Darstellungen sind praktisch gleichwertig. Bei bestimmten Anwendungen gibt es allerdings kleine Vorteile, mal für die eine, mal für die andere Darstellung (siehe z.B. [Stallings 1996]). So sind als Vorteile für BE u.a. zu nennen:

- Das alphabetische Sortieren von Zeichenketten, die sich nach den Formaten der ganzen Zahlen ausrichten, geht hier schneller als bei LE, da mehrere Zeichen (entsprechend dem Ganzzahlformat) für den Vergleich zu einem „großen Zeichen“ zusammengefaßt werden können. Bei LE muß der Vergleich zeichenweise erfolgen.
- Beim Erstellen eines Speicherauszugs (memory dump), d.h. bei byteweiser Ausgabe von Speicherinhalten (Drucker, Bildschirm), z.B. in hexadezimaler Darstellung oder in ASCII-Interpretation, werden Zahlen und Texte in korrekt zusammenhängender, von links nach rechts lesbarer Form dargestellt.

Vorteile von LE sind u.a.:

- Auf eine ganze Zahl (z.B. im 32-Bit-Format codiert) kann ohne Verändern der Adresse auch mit geringerer Breite zugegriffen werden (z.B. 16 Bit). Bei BE muß die Adresse zuvor verändert werden (im Beispiel um 2 erhöht).
- Bei arithmetischen Operationen mit Zahlen höherer Genauigkeit, die in Vielfachen des vom Prozessor vorgesehenen Datenformats vorliegen, kann auf die formatgroßen Zahlenanteile mit aufsteigenden Adressen zugegriffen werden, und zwar beginnend mit dem niedrigstwertigen Anteil. Bei BE muß zunächst die Adresse des niedrigstwertigen Anteils ermittelt werden; die Adressierung der weiteren Anteile ist dann mit absteigenden Adressen durchzuführen.

Abgesehen von diesen Nuancen sind beide Arten der Adressierung gleichwertig. Da sie außerdem jede für sich problemlos zu handhaben sind, spielt es keine Rolle, ob ein Prozessor für Big-endian- oder Little-endian-Byteanordnung ausgelegt ist. Technisch vorzusehen ist natürlich eine auf die jeweilige Adressierung, BE oder LE, zugeschnittene Byteanwahl im Speicher, also eine spezifische Logik zur Erzeugung der Byte-Enable-Signale.

Aufeinandertreffen von Big-endian- und Little-endian-Byteanordnung. Probleme ergeben sich mit Big-endian- und Little-endian-Byteanordnung erst dann, wenn zwei unterschiedlich adressierende Prozessoren auf dieselben Daten zugreifen, z.B. wenn ein Programm, das für einen Rechner mit dem einen Prozessortyp geschrieben ist, auf einem Rechner mit anderem Prozessortyp ausgeführt wird. Zu betrachten sind hier zwei unterschiedliche Vorgehensweisen für die Datenübermittlung.

1. Die beiden Prozessoren gehören eigenständigen Rechnersystemen an, und die Daten gelangen auf dem Umweg eines byteseriellen Mediums vom Hauptspeicher des einen Rechners in den Hauptspeicher des anderen Rechners, z.B. mittels einer Netzverbindung, einer Festplatte oder einer Diskette.

2. Die beiden unterschiedlichen Prozessoren sind Komponenten eines „eng gekoppelten“ (inhomogenen) *Mehrprozessorsystems*, d.h., sie haben Zugriff auf einen gemeinsamen (*globalen*) *Bus*, und die Daten stehen in einem *gemeinsamen Speicher* zur Verfügung, der über diesen Bus zugänglich ist. (Jeder der beiden Prozessoren kann darüber hinaus Zugriff auf einen jeweils lokalen Speicher haben, z.B. über seinen Prozessorbus.)

Zur Lösung des Problems der inhomogenen Datenadressierung gibt es unterschiedliche Vorgehensweisen, die auf Software oder einer Erweiterung der Prozessorhardware basieren, wie im folgenden ausgeführt. Wir nehmen dazu an, daß der obige Datensatz zu einem Programm für Little-endian-Byteanordnung gehört und daß dieses Programm von einem Prozessor mit Big-endian-Byteanordnung ausgeführt werden soll. Zu beachten ist dabei, daß die vom Programm benutzten Operandenadressen unverändert beibehalten werden.

Byteserielle Datenübermittlung – Software-Lösung. Bild 5-22 zeigt in Teilbild a nochmals die LE-Speicherdarstellung des Datensatzes der Datenquelle und in Teilbild d dessen am Datenziel erforderliche BE-Speicherdarstellung. Der Transport der Daten vom LE- in den BE-Speicher geschieht byteweise, hier mittels einer Festplatte (file server), auf der die Daten zwischengespeichert werden (Übergang von Teilbild a nach b). Das byteweise Laden der Daten von der Festplatte in den Zielspeicher führt aufgrund der unterschiedlichen Byteanwahllogiken von Quell- und Zielspeicher zu fehlerhaften Byteanordnungen innerhalb der Einzeldaten im Zielspeicher (Übergang von Teilbild b nach c). Ausgehend von den geforderten korrekten Byteanordnungen (Teilbild d) lassen sich abhängig vom Datenformat folgende Maßnahmen ableiten, mit denen nachträglich die korrekte Speicherung/Addressierung der Bytes im Zielspeicher erreicht wird (Übergang von Teilbild c nach d):

- Byte: keine Maßnahme erforderlich,
- Halbwort: vertauschen der beiden Bytes,
- Wort: vertauschen der beiden Halbworte des Worts und vertauschen der beiden Bytes eines jeden Halbworts, d.h. vertauschen aller 4 Bytes über Kreuz.

Zur Durchführung der Vertauschungen sehen Prozessoren entsprechende Vertauschungsbefehle (*Byte-Swap-Befehle*) für die unterschiedlichen Formate vor, die auf die Datenregister des Prozessors wirken. Es muß also jedes einzelne Datum (das Datenformat Byte ausgenommen) aus dem Hauptspeicher in den Prozessor geladen und nach der Vertauschungsoperation wieder zurückgeschrieben werden, und zwar bevor das eigentliche Programm auf den Datensatz zugreift. Der Nachteil dieser Software-Lösung ist, daß der Aufbau des Datensatzes, d.h. die Formate der Daten für die Korrektur bekannt sein müssen, so daß sie im Grunde nur auf Dateien mit einheitlichem Datenformat der Einzeldaten anwendbar ist. Der Vorteil dieser Lösung ist, daß die im Beispiel gemachte Alignment-Vorgabe nicht

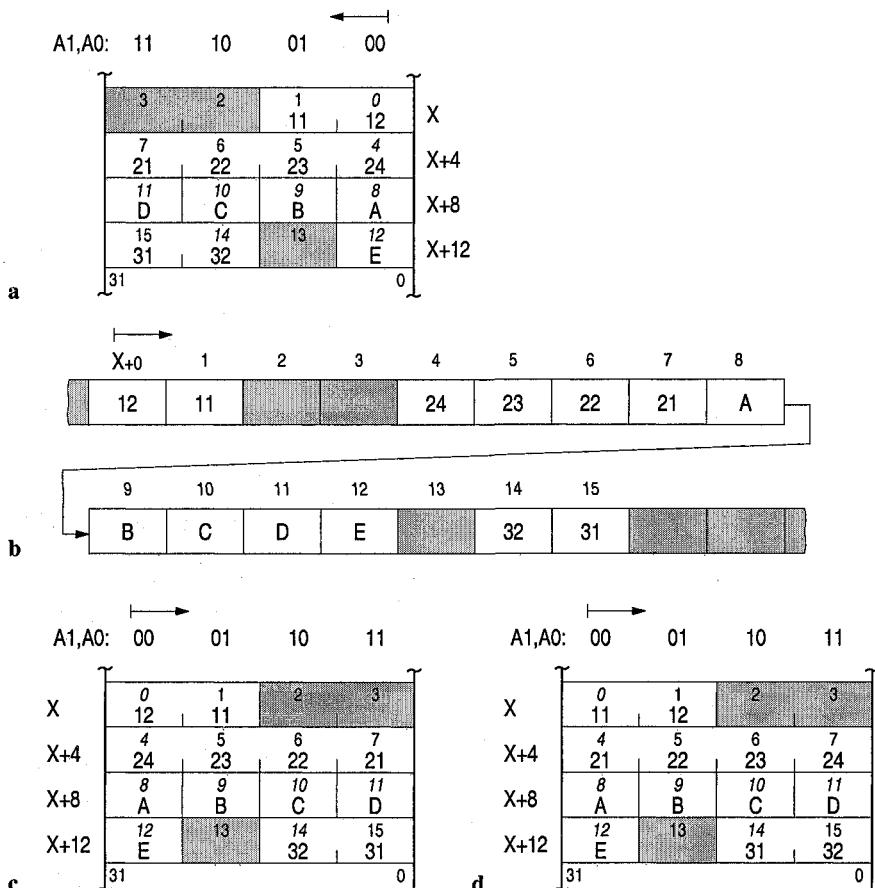


Bild 5-22. Übertragung eines Datensatzes von einem Rechnersystem mit Little-endian-Byteanordnung (LE) in ein Rechnersystem mit Big-endian-Byteanordnung (BE) mit Softwarekorrektur der Byteanordnung. Dazu folgende Speicherdarstellungen: **a** LE-Speicher als Quelle, **b** Festplatte als Zwischenspeicher nach dem byteweisen Beschreiben, **c** unkorrekte Datendarstellung im BE-Speicher nach dem byteweisen Lesen von der Festplatte, **d** korrekte Datendarstellung im BE-Speicher nach Korrektur durch Byte-Swap-Befehle. Die Pfeile über den Speicherdarstellungen geben die Zählrichtung des Byte-Offsets im Speicherwort an; die kursiven Offset-Angaben im LE- und BE-Speicher zeigen, an welchen Stellen (Adressen) die Einzeldaten adressiert werden

zwingend ist. Dies mache man sich anhand von Bild 5-22 klar, indem man dort die Adresse des Datums Y um den Wert Eins vermindert, wodurch sich die vier Datenbytes in jedem der Teilbilder zu entsprechend kleineren Offset-Werten hin verschieben.

Anmerkung. Beim Zugriff auf eine Untermenge eines Datums im BE-Speicher werden falsche Byte-Offsets generiert, so z.B. beim Zugriff auf eines der Bytes eines Halbworts, wenn – wie oben beschrieben – das Halbwort als solches dort abgelegt und mittels Bytevertauschung korrigiert wurde.

Byteserielle Datenübermittlung – Hardware-Lösung. Die im folgenden beschriebene Hardware-Lösung ist in den RISC-Prozessoren der *PowerPC*-Familie realisiert, anfänglich im 64/32-Bit-Prozessor *PowerPC 601* [Motorola 1993] und seinen Nachfolgern, heute im 64-Bit-Prozessor *PowerPC 970* [IBM 2003]. Diese arbeiten im Normalbetrieb mit Big-endian-Byteanordnung (big-endian mode), weshalb auch die Byteanwahllogik des Hauptspeichers für diese Art der Adressierung ausgelegt wird. Für die Verarbeitung von Daten von IBM-PCs, deren Adressierung aufgrund der Intel-Prozessoren (z. B. *Pentium*) auf Little-endian-Byteanordnung basiert, sehen die PowerPC-Prozessoren zusätzlich die Betriebsart „Little-Endian-Mode“ vor. In ihr werden die für den Datenzugriff benutzten Speicheradressen vom Prozessor abhängig vom Datenformat in den Adreßbits A2, A1 und A0 wie folgt manipuliert:

Byte: invertieren von A2, A1 und A0,

Halbwort: invertieren von A2 und A1,

Wort: invertieren von A2,

Doppelwort: keine Manipulation erforderlich.

Bild 5-23 zeigt die Wirkung dieser Manipulation für das obige Beispiel, wobei zur Vereinheitlichung der Darstellungen entgegen den PowerPC-Prozessoren mit ihren 64-Bit-Datenwegen von einem 32-Bit-Prozessor ausgegangen wird. Der Datensatz liegt wieder byteweise gespeichert auf einer Festplatte vor (Teilbild a), von einem LE-Speicher mit einer Speicherdarstellung gemäß Bild 5-22a kommend. Beim byteweisen Laden von der Festplatte in den Hauptspeicher des BE-Prozessors werden die beiden Adreßbits A1 und A0 invertiert (A2 bleibt aufgrund unserer 32-Bit-Betrachtung unverändert!), so daß die Datenbytes an denselben Speicherpositionen wie im LE-Speicher abgelegt werden (Teilbild b). Die korrekte Adressierung der Einzeldaten beim späteren Speicherzugriff wird dadurch erreicht, daß auch hierbei wieder die Adreßbits A1 und A0 abhängig vom jeweiligen Datenformat manipuliert werden. – Man bezeichnet diese auf Adreßmanipulation beruhende Little-Endian-Betriebsart auch als *Pseudo-Little-Endian-Mode* oder als *munged Little-Endian-Mode*.

Der Vorteil dieser Hardware-Lösung ist, daß sie für den Programmierer transparent ist und daß die Datenformatinformation (anders als bei der Software-Lösung) nicht bereits für das Laden des BE-Speichers benötigt wird, sondern erst bei den späteren Datenzugriffen. Dann ergibt sie sich automatisch aus den Suffixen der Lade- und Speichere-Befehle. Nachteilig ist, daß die im Beispiel gemachte Alignment-Vorgabe für den Little-Endian-Modus zwingend ist. Dies mache man sich wieder anhand des Datums Y mit gegenüber Bild 5-23 um Eins verminderter Adresse klar. – Es ist noch anzumerken, daß auch das zu simulierende Programm im Little-Endian-Modus zu laden ist, da nicht nur die Datenzugriffe, sondern auch die Befehlszugriffe in dieser Betriebsart erfolgen.

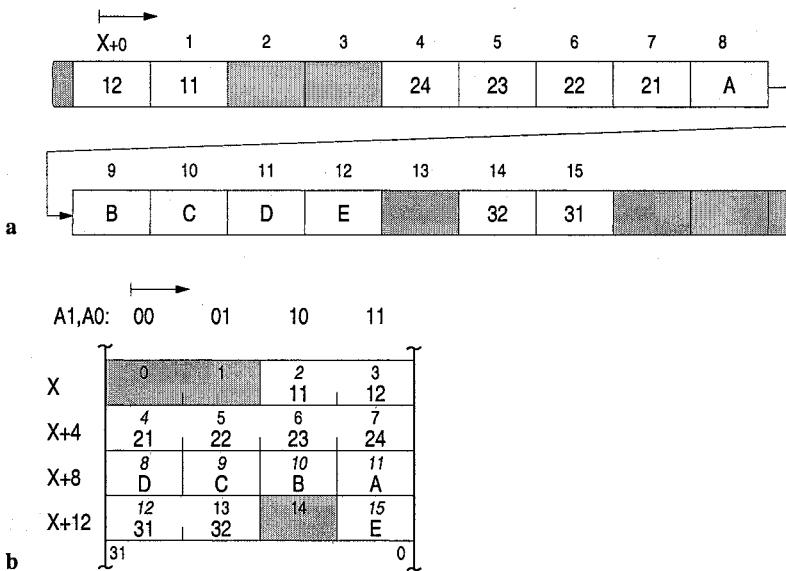


Bild 5-23. Übertragung eines Datensatzes von einem Rechnersystem mit Little-endian-Byteanordnung (LE) in ein Rechnersystem mit Big-endian-Byteanordnung (BE) mit *Hardwarekorrektur* der Byteanordnung. Dazu folgende Speicherdarstellungen: a Festplatte als Zwischenspeicher nach dem byteweisen Beschreiben aus dem LE-Speicher (Bild 5-22a), b BE-Speicher nach dem byteweisen Lesen von der Festplatte mit korrekter Datendarstellung für Zugriffe im Little-Endian-Modus. Der Pfeil über der Speicherdarstellung gibt die Zählrichtung des Byte-Offsets im Speicherwort an; die kursiven Offset-Angaben im LE- und BE-Speicher zeigen, an welchen Stellen (Adressen) die Einzeldaten adressiert werden

Gemeinsamer Speicher – Hardware-Lösungen. Bild 5-24 zeigt eine Mehrprozessorstruktur gemäß der eingangs genannten Vorgehensweise 2 der Datenübermittlung. Sie weist einen *Pentium* als LE-Prozessor und einen *PowerPC* als BE-Prozessor auf, letzterer mit der Möglichkeit, Speicherzugriffe im Little-Endian-Modus durchzuführen. Beide Prozessoren haben Zugriff auf einen gemeinsamen Speicher (*shared memory*), der für die LE-Byteanwahl ausgelegt ist. Die zusätzlichen lokalen Speicher sind mit LE- bzw. BE-Byteanwahl individuell auf die beiden Prozessoren zugeschnitten.

Der oben beschriebene Datensatz wird vom Pentium gemäß Bild 5-22a im gemeinsamen Speicher abgelegt, um vom PowerPC bearbeitet zu werden. Benutzt der PowerPC den Little-Endian-Modus, so gibt es bei den Datenzugriffen auf den gemeinsamen Speicher keine Probleme, da der PowerPC die Datenbytes ja an denselben Bytepositionen im Speicherwort bzw. auf dem Datenbus erwartet, wie sie vom LE-Prozessor im gemeinsamen LE-Speicher vorgegeben sind (vergleiche dazu die Bilder 5-22a und 5-23b). Bezüglich der Ansteuerung des gemeinsamen Speichers muß lediglich gewährleistet sein, daß bei Zugriffen des PowerPC (BE-Prozessor) die hierfür erforderlichen LE-Byte-Enable-Signale erzeugt werden, und zwar in der Bridge B des PowerPC.

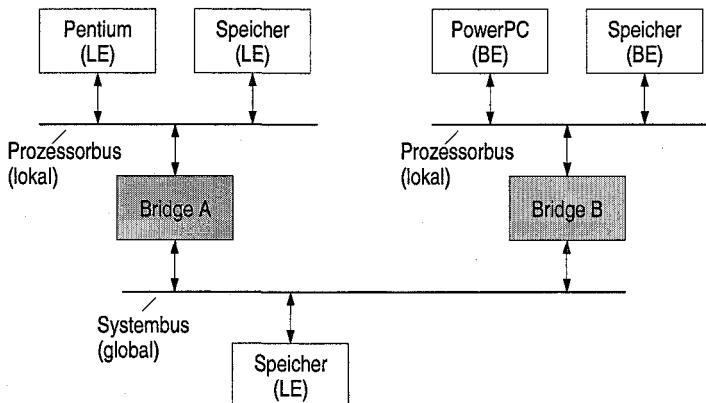


Bild 5-24. Mehrprozessorsystem mit zwei unterschiedlich adressierenden Prozessoren, einem Pentium mit Little-endian-Byteanordnung (LE) und einem PowerPC mit Big-endian-Byteanordnung (BE). Jeder der beiden Prozessoren hat Zugriff auf einen ihm angepaßten lokalen Speicher (LE bzw. BE) sowie Zugriff auf einen gemeinsamen globalen Speicher, der eine Byteanwahllogik für Little-endian-Byteanordnung aufweist

Wird anstelle des PowerPC ein BE-Prozessor verwendet, der keinen Little-Endian-Modus hat, so muß die Bridge B um einen Mechanismus zur Byteverschiebung erweitert werden, der bei jedem einzelnen Datenzugriff in Aktion tritt, und zwar abhängig vom Datenformat des jeweiligen Operanden (vergleiche dazu die Bilder 5-22a und 5-22d). Bezogen auf unsere 32-Bit-Betrachtung heißt das:

- Byte: verschieben über Kreuz, die gesamte Busbreite betreffend; auf die vier Bytepositionen des Busses bezogen heißt das $0 \rightarrow 3$ bzw. $1 \rightarrow 2$ bzw. $2 \rightarrow 1$ bzw. $3 \rightarrow 0$,
- Halbwort: verschieben auf die andere Bushälfte,
- Wort: kein Verschieben erforderlich.

Ferner müssen in der Bridge B, wie bei der Lösung mit Little-Endian-Modus, die erforderlichen LE-Byte-Enable-Signale erzeugt werden.

Anmerkung. Bei beiden Lösungsmöglichkeiten, Little-Endian-Modus und Verschieben von Daten in der Bridge, können die Daten auch im lokalen BE-Speicher abgelegt und von dort wieder korrekt gelesen werden. Wird jedoch auf eine Untermenge eines Datums im BE-Speicher zugegriffen, z.B. auf eines der Bytes eines Halbworts, so arbeitet nur das System mit Little-Endian-Modus fehlerfrei.

5.2.5 Busankopplung

Eine Speicher- oder Interface-Einheit muß, wenn sie durch den Prozessor adressiert wird, mit ihren Datenbusanschlüssen an den Systemdatenbus angekoppelt

werden. Die hierfür erforderliche Logik ist häufig in den Bausteinen selbst, nämlich in den Datenbustreibern enthalten, die über Bausteinanwahl eingänge \overline{CS} (chip select) oder \overline{CE} (chip enable) gesteuert werden. Mit $\overline{CS} = 1$ bzw. $\overline{CE} = 1$ befinden sich die Datenbusanschlüsse im *hochohmigen Zustand* (tristate) und sind damit vom Datenbus abgekoppelt; mit $\overline{CS} = 0$ bzw. $\overline{CE} = 0$ werden sie auf den Datenbus durchgeschaltet. Die Datenflußrichtung in der Treiberlogik wird durch das Lese-/Schreibsignal R/W des Prozessors bestimmt.

Häufig erfolgt die Busankopplung durch spezielle Treiberbausteine, die zwischen die Funktionseinheit und den Datenbus geschaltet werden. Bild 5-25 zeigt einen solchen *Bustreiber (bus driver)*, der aufgrund der umschaltbaren Datenflußrichtung als *bidirekionaler* Treiber bezeichnet wird. Für jeden Signalweg A_i/B_i besitzt er zwei Verstärkerstufen (Dreiecksymbol), von denen jeweils eine durch den Zustand des R/W -Signals angewählt wird. Das \overline{CE} -Signal gibt den angewählten Signalweg frei oder schaltet die Verstärkerausgänge in den *hochohmigen Zustand*.

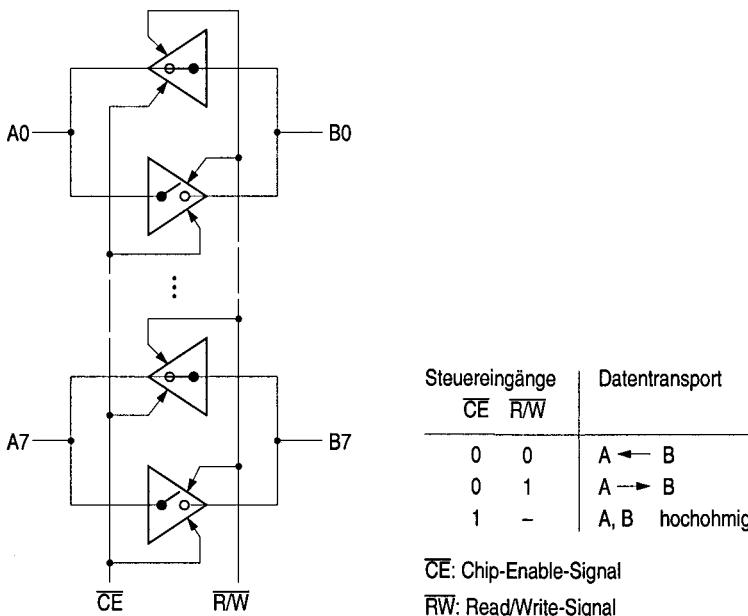


Bild 5-25. Bidirektonaler Bustreiber mit Signalverstärkern mit Tristate-Logik

Treiberbausteine haben neben ihrer Tristate-Funktion den Vorteil, wegen ihrer Verstärkereigenschaft eine größere Buslast treiben zu können, d.h., der Bus kann mit einer größeren Anzahl von Bausteinen (kapazitiv) belastet werden. Aus diesem Grunde werden üblicherweise auch Adress- und Steuerleitungen über Treiberbausteine an den Systembus geschaltet; bei *unidirektonalen* Signalen werden entsprechend unidirektionale Treiber eingesetzt. Aus Gründen der begrenzten Belastbarkeit der Prozessoranschlüsse wird auch der Mikroprozessor selbst über

Treiberbausteine an den Systembus geschaltet. Nachteilig ist die zusätzliche Signallaufzeit.

5.3 Datentransportsteuerung

Die Durchführung eines Datentransports auf dem Prozessorbus oder einem prozessorunabhängigen Bus umfaßt eine Reihe von Steuerungsabläufen, wie das Adressieren der am Transport beteiligten passiven Einheit, das Ankoppeln der adressierten Einheit an den Datenbus und die Angabe der Transportrichtung. Hinzu kommen Zeitvorgaben für die Gültigkeit der Adresse sowie die Bereitstellung und die Übernahme des Datums. Ausgelöst wird der Datentransport vom Prozessor oder einem anderen Master, dem die Steuerung obliegt. Die adressierte Einheit hat als Slave untergeordnete Funktion. Sie beschränkt sich im Steuerungsablauf meist auf das Quittieren der Datenübernahme bzw. die Signalisierung der Datenbereitstellung. Die Synchronisation der Aktivitäten des Masters und des Slaves kann asynchron, d.h. durch Steuersignale, oder synchron, d.h. durch einen Bustakt, erfolgen. – Den gesamten Ablauf eines Datentransports bezeichnet man auch als *Buszyklus*.

5.3.1 Asynchroner und synchroner Bus

Bei einem asynchronen wie bei einem synchronen Bus erfolgt die Synchronisation zwischen *Master* und *Slave* über Steuersignale, über die sie sich im Handshake-Verfahren wechselseitig informieren. Der Unterschied zwischen asynchronem und synchronem Bus liegt darin, daß bei ersterem die am Datentransport beteiligten Systemkomponenten unabhängig getaktet sind (oder z.B. nur die eine getaktet arbeitet und die andere in Asynchrontechnik aufgebaut ist) und bei letzterem beide Systemkomponenten mit einem gemeinsamen Takt, dem *Bustakt*, arbeiten.

Asynchroner Bus. Beim asynchronen Bus (Bild 5-26a) beginnt der Master den Buszyklus durch Ausgeben der Adresse, deren Gültigkeit er dem Slave etwas verzögert durch ein Adreßgültigkeitssignal anzeigt (address strobe, \overline{AS}). Bei einem *Schreibzyklus* (erster Buszyklus im Bild) gibt er danach das Datum aus und zeigt dessen Gültigkeit wieder etwas verzögert durch ein Datengültigkeitssignal (data strobe, \overline{DS}) an. Beide Signale sind mit dem Takt des Masters (CPUCLK) synchronisiert. Der adressierte Slave signalisiert zu gegebener Zeit seine Datenübernahme durch ein asynchron dazu erzeugtes Quittungssignal (data transfer acknowledge, \overline{DTACK}). Daraufhin inaktiviert der Master seine beiden Gültigkeitssignale, was den Slave wiederum veranlaßt, sein Quittungssignal zurückzunehmen. Der Schreibzyklus ist damit beendet. Bei einem *Lesezyklus* (zweiter Buszyklus im Bild) sind die Funktionen von \overline{DS} und \overline{DTACK} gegenüber dem

Schreibzyklus vertauscht: Mit dem Aktivieren von \overline{DTACK} signalisiert der Slave, daß das von ihm auf den Bus gelegte Datum gültig ist, mit dem Inaktivieren von \overline{DS} zeigt der Master seine Datenübernahme an.

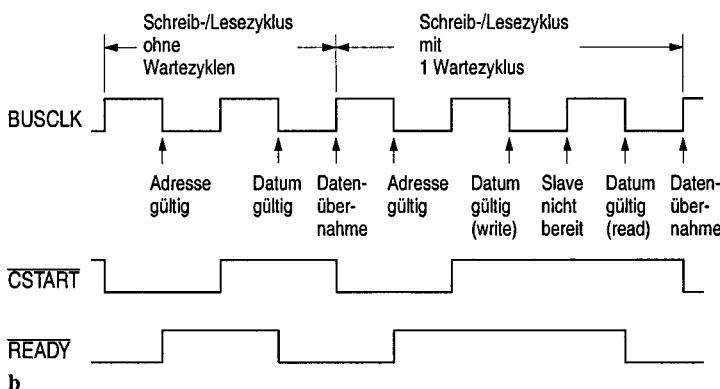
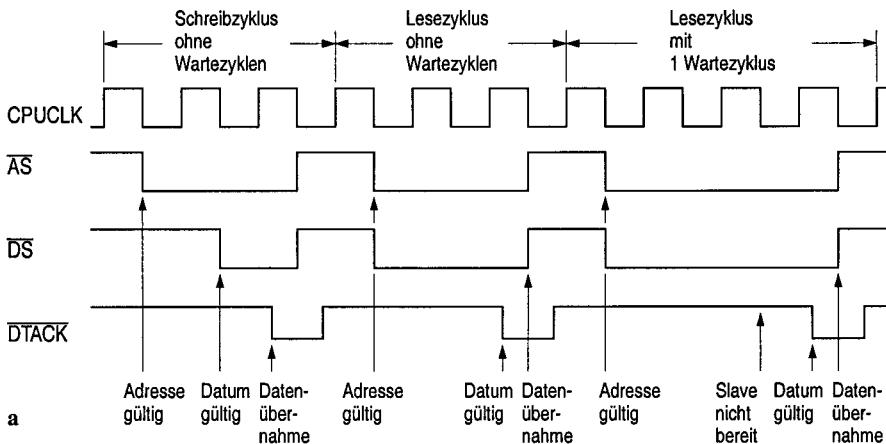


Bild 5-26. Buszyklen ohne und mit Wartezyklus. a Schreib/Lesezyklus bei asynchronem Bus, b Schreib- und Lesezyklus bei synchronem Bus

Der kürzest mögliche Ablauf eines Buszyklus umfaßt nach Bild 5-26a drei Takte des Masters, z.B. drei Prozessortakte. Wir bezeichnen diesen Ablauf als *minimalen Buszyklus*. Ein Slave, der nicht in der Lage ist, innerhalb dieser Zeit ein Datum entgegenzunehmen oder bereitzustellen, kann das Quittungssignal \overline{DTACK} verzögert aktivieren und auf diese Weise den Buszyklus um einen oder mehrere Prozessortaktschritte (*Wartezyklen, wait cycles*) verlängern (dritter Buszyklus im Bild). Das eventuelle Ausbleiben des \overline{DTACK} -Signals aufgrund eines defekten Slave, oder weil zu der ausgegebenen Adresse kein Slave existiert, wird durch einen Zeitbegrenzer (*watch-dog timer*) überwacht. Er meldet dem Master

das Überschreiten einer Grenzzeit (time out) mittels eines Unterbrechungssignals als *Busfehler*, woraufhin dieser den Buszyklus abbricht (*bus error trap*).

Synchroner Bus. Beim synchronen Bus (Bild 5-26b) sind Master und Slave durch ein gemeinsames Bustaktsignal (BUSCLK) miteinander verbunden, d.h., die Übertragung unterliegt einem festen, vom Takt geprägten Zeitraster. Ausgelöst wird der Ablauf vom Master durch ein Startsignal ($\overline{\text{CSTART}}$), das dem Slave den Beginn des Buszyklus anzeigen. Danach gibt es feste, durch den Bustakt vorgegebene Zeitpunkte für die Gültigkeit der vom Master ausgegebenen Adresse, für die Gültigkeit des vom Sender (Master oder Slave) ausgegebenen Datums sowie für die Übernahme des Datums durch den Empfänger (Slave bzw. Master), d.h. für den Abschluß des Buszyklus.

Nachteil dieser festen Zeitbindung ist zunächst, daß die Frequenz des Bustaktes an den langsamsten Slave angepaßt werden muß, um dessen Zeitbedarf für die Datenbereitstellung oder Datenübernahme gerecht zu werden. Abhilfe schafft hier eine Erweiterung des Protokolls um ein Bereitschaftssignal $\overline{\text{READY}}$, das vom adressierten Slave erzeugt wird und das ebenfalls mit dem Bustakt synchronisiert ist. Langsamere Slaves halten dieses Signal über den kürzest möglichen Buszyklus hinaus inaktiv, woraufhin der Master den Buszyklus um einen oder mehrere Bustaktschritte (*Wartezyklen*) verlängert. Der Master schließt den Buszyklus erst dann ab, wenn der Slave seine Bereitschaft durch das Aktivieren von $\overline{\text{READY}}$ signalisiert. – Der *minimale Buszyklus* umfaßt für den Master nach Bild 5-26b zwei Bustakte.

Anmerkung. Um einen Buszyklus anzuzeigen, haben Prozessoren anstelle von $\overline{\text{CSTART}}$ häufig ein Address-Strobe-Signal, das – wie für den asynchronen Buszyklus beschrieben – erst nach Beginn des Buszyklus ausgegeben wird, nämlich dann, wenn die Adresse auf dem Bus gültig ist.

Die durch die Busspezifikation vorgegebenen Signale stimmen in ihrer Funktion und ihrem Zeitverhalten meist nicht mit den Signalen des jeweils eingesetzten Prozessors überein. Hier muß – wie in 5.1.2 bereits erwähnt – zwischen Prozessor und Bus eine Signalanpassung durch zusätzliche Logik vorgenommen werden. Entsprechend muß auch jede Slave-Einheit mit einer busspezifischen Ansteuerung versehen werden. Die Vorteile synchroner Busse liegen im einfacheren Aufbau dieser Hardware (Syncronotechnik) und im einfacheren Testen der Abläufe durch die fest vorgegebenen Zeitpunkte der Signaländerungen. Der prinzipielle Vorteil asynchroner Busse liegt darin, daß die Buskomponenten ohne Bindung an eine bestimmte Taktfrequenz entwickelt werden können; es muß lediglich das Handshake-Protokoll eingehalten werden. In der Praxis wird diese Unabhängigkeit von Master und Slave häufig durchbrochen, um die Buszykluszeit zu verkürzen. So wird das DTACK-Signal vom Slave meist vorzeitig aktiviert, davon ausgehend, daß der Master eine bestimmte Zeit benötigt, sich mit diesem Signal zu synchronisieren (siehe dazu 5.3.2). Bezüglich der Übertragungsgeschwindigkeiten beider Busarten gibt es keine nennenswerten Unterschiede. – Die meisten neueren Busse sind für synchronen Datentransport ausgelegt.

5.3.2 Schreib- und Lesezyklen

Das in 5.3.1 beschriebene Prinzip asynchroner und synchroner Busse bezieht sich sowohl auf die prozessorunabhängigen Busse, wie Systembusse und Speicherbusse, als auch auf die Prozessorbusse selbst, d.h. auf die Schreib- und Lesezyklen von Prozessoren oder anderen Mastern, z.B. DMA-Controllern. Betrachtet man die 16-Bit-Mikroprozessoren und deren 32- und 64/32-Bit-Nachfolger, so galten als Hauptvertreter asynchroner Prozessorbusse lange Zeit die CISC-Prozessoren der 680x0-Familie von Motorola, angefangen mit dem MC68000 und fortgesetzt mit dem MC68020. Dessen Nachfolger, der MC68030, sah dann zusätzlich zur asynchronen Bussteuerung auch eine synchrone Bussteuerung vor, die dann bei den weiteren Nachfolgern MC68040 und MC68060 ausschließlich realisiert wurde. Die CISC-Prozessoren von Intel hingegen, beginnend mit dem i286, waren von vornherein mit synchronen Bussen ausgestattet, so wie auch die meisten Prozessorbusse anderer CISC-Hersteller. Bei den RISC-Prozessoren hat sich der synchrone Bus ebenfalls von Anfang an durchgesetzt, so z.B. bei den PowerPC-Prozessoren von IBM/Motorola. Die Präferenz heutiger Prozessorbusse liegt also, wie auch oben schon angedeutet, eindeutig beim synchronen Bus.

Im folgenden wird nun je ein Beispiel für einen asynchronen und einen synchronen Lese- und Schreibzyklus gegeben, und zwar in einer etwas ausführlicheren Darstellung des Signal-Zeitverhaltens als bei den schematisierten Abläufen im obigen Bild 5-26. So werden nicht nur die Synchronisationssignale, sondern alle am Buszyklus beteiligten Signale des Prozessorbusses berücksichtigt. Darüber hinaus wird in den Darstellungen den technischen Gegebenheiten insofern Rechnung getragen, als z.B. ein Signal seinen Zustand nicht schlagartig wechseln kann (was durch schräge Signalflanken angedeutet ist). Doppellinien für Signale symbolisieren das mögliche Auftreten unterschiedlicher Signalzustände bei mehreren Signalleitungen; Signallinien in Mittenposition stellen den hochohmigen Signalzustand von Tristate-Signalen dar. Alle Signale sind aus Gründen besserer Anschaulichkeit nach wie vor idealisiert dargestellt. BUSCLK und CPUCLK steuern mit ihren Flanken die Pegelübergänge der Prozessorausgangssignale, BUSCLK bei den synchronen Buszyklen auch die der Slave-Ausgangssignale. In Wirklichkeit sind diese Übergänge durch Gatterlaufzeiten und kapazitive Buslasten gegenüber den Taktflanken verschoben, was hier jedoch vernachlässigt wird.

Asynchroner Schreib- und Lesezyklus. Bild 5-27 zeigt schematisiert ein Schaltungsbeispiel für den Signalaustausch zwischen einem Prozessor mit asynchronem Busverhalten und der 16-Mbyte-Speicherplatine aus Bild 5-19 (S. 305). Die Speicherplatine wird durch die Adressbits A31 bis A24 angewählt (Kartenanwahllogik) und erzeugt ihr DTACK-Signal mittels einer Verzögerungseinrichtung (Delay). Die Anpassung der vom Prozessor kommenden und der auf der Karte erzeugten Signale an die Steuersignale der Speicherbausteine (6.1.1) erfolgt durch

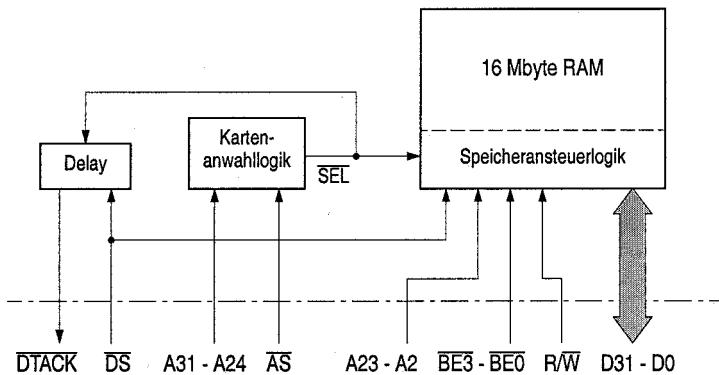


Bild 5-27. Datentransportsteuerung für die 16-Mbyte-Speicherkarte nach Bild 5-19

eine Speicheransteuerlogik, die hier nur angedeutet ist. Das Signal-Zeitverhalten für den zugehörigen Schreib- und den Lesezyklus, angelehnt an den MC68020, zeigt Bild 5-28, und zwar für den Fall, daß keine Wartezyklen erforderlich sind. Die Angaben Zi im Bild kennzeichnen dabei die Taktzustände.

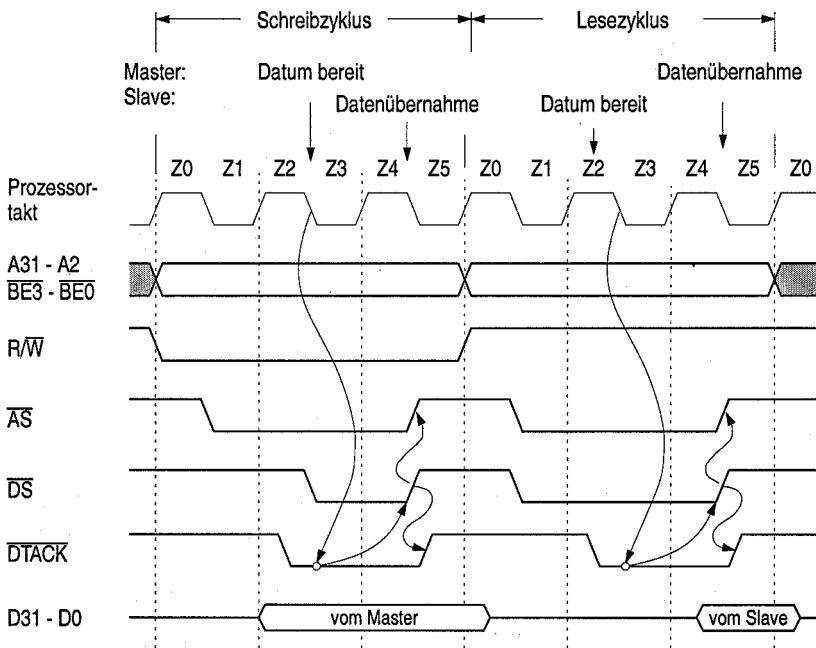


Bild 5-28. Asynchroner Schreibzyklus und Lesezyklus (in Anlehnung an den MC68020 [Motorola 1989])

Die Zyklen beginnen im Zustand Z0, in dem der Prozessor die Adresssignale A31 bis A2, die Byte-Enable-Signale BE3 bis BE0, den entsprechenden Pegel für das

Lese-/Schreibsignal R/\bar{W} sowie zusätzliche Statussignale, wie S (Supervisor-/User-Modus), an den Systembus ausgibt. Die Gültigkeit dieser Signale zeigt er im Zustand Z1 durch das Adreßgültigkeitssignal \overline{AS} an, woraufhin die adressierte Speichereinheit ihr Anwahlsignal \overline{SEL} für die Speicheranwahl erzeugt und damit ihre Verzögerungseinrichtung zur Erzeugung des Quittungssignals \overline{DTACK} startet.

Wie anhand der Pfeile im Bild zu erkennen ist, fragt der Prozessor das \overline{DTACK} -Signal bereits am Ende von Z2 ab, obwohl er z.B. beim Schreibzyklus erst zu diesem Zeitpunkt seine Datenbereitstellung durch $\overline{DS} = 0$ angezeigt. Dieser „Vorgriff“ ist damit begründet, daß der Prozessor danach noch zwei Zustände (Z3 und Z4) benötigt, um sich mit diesem Signal zu synchronisieren. Ist \overline{DTACK} am Ende von Z2 inaktiv, so fügt er einen Wartezyklus, d.h. zwei Zustände, in den Ablauf ein und fragt \overline{DTACK} mit der nächsten negativen Taktflanke erneut ab. Dementsprechend kann ein Buszyklus nur dann ohne Wartezyklen abgeschlossen werden, wenn die Speichereinheit wie in 5.3.1 beschrieben ihr Quittungssignal vorzeitig, d.h. vor der eigentlichen Datenübernahme (Schreibzyklus) bzw. Datenbereitstellung (Lesezyklus) aktiviert.

Synchroner Schreib- und Lesezyklus. In Analogie zu Bild 5-28 zeigt Bild 5-29 das Signal-Zeitverhalten für den Schreib- und den Lesezyklus eines Prozessors mit synchronem Bus, angelehnt an den i486. Hier sind die Zusammenhänge auch ohne Pfeilangaben leicht erkennbar. Nach Ablauf des *minimalen Buszyklus* von zwei Takten wird der Abschluß der Übertragung erwartet. Ist zu diesem Zeitpunkt, d.h. am Ende von Z3, das Quittungssignal \overline{READY} der Speichereinheit inaktiv, so fügt der Prozessor einen Wartezyklus (zwei Zustände) ein und fragt danach \overline{READY} erneut ab.

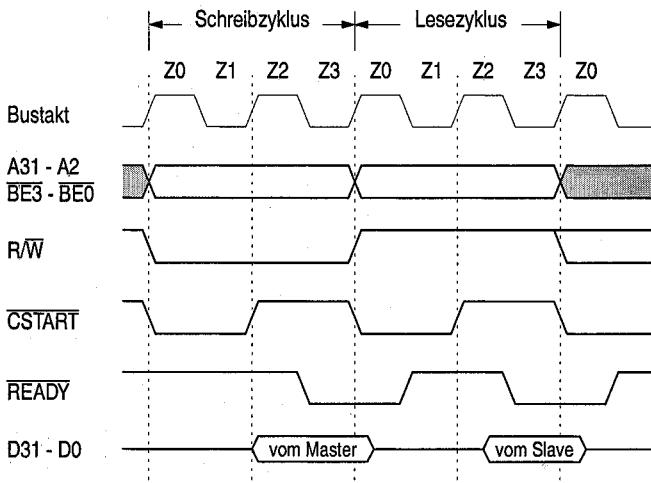


Bild 5-29. Synchroner Schreibzyklus und Lesezyklus (in Anlehnung an den i486 [Intel 1989])

Anmerkung. Unabhängig von der Synchronisationsart, asynchron oder synchron, legt der Prozessor das Lese-/Schreibsignal R/W im Busruhezustand (*bus idle*) auf Lesen fest, um der Gefahr eines fälschlichen Schreibens in eine Speichereinheit vorzubeugen.

5.3.3 Blockbuszyklus

Bei Prozessoren mit On-chip-Caches für die Zwischenspeicherung von Befehlen und Daten werden *Einzelbuszyklen* (*single cycles*), wie die oben beschriebenen Schreib- und Lesezyklen, nur dann auf dem Prozessorbus sichtbar, wenn mit ihnen auf Speicherzellen zugegriffen wird, deren Inhalte von einer Speicherverwaltungseinheit als „*non-cacheable*“ ausgewiesen sind, z.B. auf Register von Ein-/Ausgabeeinheiten. Ansonsten treten Einzelbuszyklen nur prozessorintern beim Zugriff auf die Cache-Inhalte in Erscheinung. Die *Caches* selbst werden immer blockweise geladen, wobei ein Block grundsätzlich vier oder acht Wörter (16 bzw. 32 Bytes bei einem 32-Bit-Prozessor, 32 bzw. 64 Bytes bei einem 64-Bit-Prozessor) umfaßt, die bezüglich ihrer Adressen im Hauptspeicher ausgerichtet, d.h. an den Vielfachen der Blockgröße gespeichert sind (*block alignment*).

Die vier Wortübertragungen (*Cache-fill-* oder *Write-back-Operation*) können im Prinzip zwar als vier Einzelbuszyklen ausgeführt werden, sie werden üblicherweise jedoch mittels eines speziellen Blockbuszyklus (*burst cycle*) vorgenommen. Dabei macht man sich das Aufeinanderfolgen der Wortadressen und die Blockausrichtung zunutze, indem man den Prozessor (bzw. die *Cache-Steuereinheit*) nur noch die Adresse des ersten Wortes, sozusagen die Blockadresse ausgeben läßt. Der erste Wortzugriff dauert dann zwar so lange wie bei einem Einzelbuszyklus, die Folgezugriffe benötigen jedoch weniger Zeit. Die Adreßfortschaltung geschieht üblicherweise prozessorextern, entweder durch die Speicheransteuerung (bei PCs durch den sog. *Chipsatz*) oder direkt in den Speicherbausteinen. Die Adressierung erfolgt dabei immer umlaufend (*wrap around*), d.h., die vom Prozessor ausgegebene Wortadresse kann irgendeine der vier Wortadressen sein. Verändert werden dabei nur die signifikanten Adreßbits A3 und A2 (32-Bit-Prozessor) bzw. A4 und A3 (64-Bit-Prozessor). – Auf die Cache-Problematik und auf die speicherseitige Unterstützung des Blockbuszyklus wird ausführlicher in Kapitel 6 eingegangen.

Bild 5-30 zeigt das Signal-Zeitverhalten des Blockbuszyklus für das Lesen von vier Wörtern ohne Wartezyklen, und zwar als synchronen Zyklus in Anlehnung an den *i486*. Um den Bezug zum Einzelbuszyklus (Bild 5-29) herzustellen, ist ihm ein solcher vorangestellt. Zur Unterscheidung des Blockbuszyklus vom Einzelbuszyklus und zur Angabe der „Länge“ des Blockbuszyklus gibt der Prozessor (bzw. die *Cache-Steuereinheit*) ein Signal LAST aus, das den Zeitpunkt der letzten Datenübernahme oder Datenbereitstellung anzeigen. Der Speicher, der einen Blockbuszyklus durch das Ausbleiben des Aktivpegels von LAST bei der ersten Datenübertragung als solchen erkennt, signalisiert seine jeweilige Datenbereit-

schaft mit einem speziellen Burst-Ready-Signal, BREADY, und hält dabei das Ready-Signal des Einzelbuszyklus, READY, inaktiv. Ist der Speicher nicht in der Lage, einen Blockbuszyklus durchzuführen, so reagiert er bei der ersten Datenübertragung mit READY anstatt mit BREADY, woraufhin der Prozessor (bzw. die Cache-Steuerung) den Blockbuszyklus in vier normale Einzelbuszyklen auflöst.

Den in Bild 5-30 dargestellten Blockbuszyklus bezeichnen wir als „*minimalen*“ *Blockbuszyklus*, da er ohne Wartezyklen, d.h. mit der für den Prozessor geringst möglichen Taktanzahl auskommt. Das bedeutet für die erste Wortübertragung 2 Takte (wie für den minimalen Einzelbuszyklus nach Bild 5-29) und für jede weitere Wortübertragung je 1 Takt. Man spricht deshalb auch von einem 2-1-1-1-Buszyklus oder einem 2-1-1-1-*Burst*. Kann der Speicher diese minimalen Zeitvorgaben nicht einhalten, so fordert er *Wartezyklen* an, indem er das Signal BREADY jeweils verzögert aktiviert. Bei z.B. drei Wartezyklen für das erste Wort und je einem Wartezyklus für die Folgewörter ergibt sich dann ein 5-2-2-2-*Burst*. Der Prozessor (bzw. die Cache-Steuereinheit) seinerseits aktiviert das Signal LAST immer zu dem frühest möglichen Zeitpunkt der letzten Datenübernahme oder Datenbereitstellung und hält es abhängig von BREADY ggf. mehr

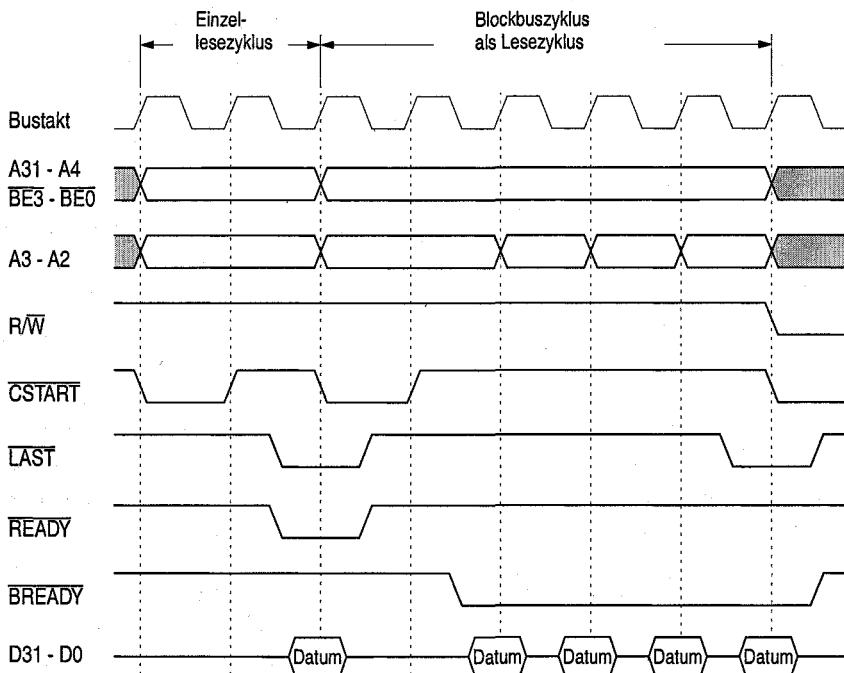


Bild 5-30. Synchroner Blockbuszyklus mit vorangehendem Einzelbuszyklus als Lesezyklen (in Anlehnung an den i486 [Intel 1989])

als einen Takt lang aktiv. – In einer Erweiterung dieser Technik kann der Prozessor (bzw. die Cache-Steuereinheit) ggf. die Adresse des nächsten Blockbuszyklus überlappenden zu den Datenübertragungen des momentanen Blockbuszyklus ausgeben. Man spricht dann von sich *überlappenden Blockbuszyklen (pipelined burst cycles*, siehe dazu 6.1.5). Bei den obigen Betrachtungen wurden Übertragungen mit Single-Data-Rate, d.h. mit Taktabstand angenommen. Bei Übertragungen mit Double-Data-Rate fallen die Folgeübertragungen im Halbtaktabstand an (siehe 6.1.6).

Anmerkung. Das in Bild 5-30 gezeigte Beispiel eines Blockbuszyklus ist eine von vielen Varianten, die für die Kommunikation zwischen Prozessor und Speichereinheit erforderlichen Signale festzulegen. Andere Prozessoren weisen z.B. ein anderes Signalverhalten für die Anzeige eines Blockbuszyklus und dessen Quittierung durch die Speichereinheit auf und haben so z.B. nur ein einziges Ready-Signal, das sowohl für Einzel- als auch für Blockbuszyklen zuständig ist. Außerdem ist der Blockbuszyklus abhängig vom Prozessor entweder auf vier Wortübertragungen festgelegt, oder aber er kann, wie im obigen Beispiel möglich, bereits nach zwei Wortübertragungen beendet werden, z.B. um eine Gleitpunktzahl doppelter Genauigkeit zu übertragen. Insbesondere führen die meisten Prozessoren die Adreßfortschaltung nicht selbst durch; statt dessen wird sie in die Speichereinheit verlagert (siehe dazu 6.1.5).

5.4 Busarbitration

Die Grundlage für den Zugriff der einzelnen Komponenten eines Mikroprozessorsystems auf den gemeinsamen Systembus bildet das *Master-Slave-Prinzip*. Aktive Komponenten, wie Prozessoren und Controller, wirken als *Master*; sie sind in der Lage, Buszyklen auszulösen. Passive Komponenten, wie Speicher und Interfaces, sind *Slaves*. Bei einfachen Systemen mit einem Mikroprozessor als einzigm Master ist der Buszugriff unproblematisch, da die Bussteuerung ausschließlich diesem Master unterliegt. Bei komplexeren Systemen mit mehreren MASTERN, z.B. bei einem Mikroprozessorsystem mit einem DMA-Controller (Mehrmastersystem), muß die zeitliche Abfolge der Buszugriffe organisiert werden.

Zu dieser Aufgabe gehören die Priorisierung der Anforderungen, die Zuteilung des Busses und ggf. die vorzeitige Abgabe des Busses an einen anfordernden Master höherer Priorität. Dabei ist zu berücksichtigen, daß ein Busmaster erst nach Abschluß seines momentanen Buszyklus vom Bus verdrängt werden kann. Man bezeichnet die Busverwaltung auch als *Busarbitration* (arbitration: schiedsrichterliche Entscheidung). – Während der Bus von einem der Master belegt wird, müssen die übrigen Master mit ihren Adreß-, Daten- und Steuerausgängen, mit Ausnahme der Leitungen für die Busverwaltung, vom Bus abgekoppelt sein. Dies erfolgt mittels *Tristate-Logik*.

Bei Systemen mit sog. *lokaler Busarbitration* ist dem Mikroprozessor der Systembus als *lokaler Bus* fest zugeordnet, und die zusätzlichen Master im System fordern von diesem den Bus nach Bedarf an (Bild 5-31a). Der Mikropro-

zessor hat hierbei niedrigste Priorität; die Priorisierung der Zusatzmaster untereinander erfolgt durch eine prozessorexterne Prioritätenlogik. Bei diesen Mastern handelt es sich i. allg. um Steuerbausteine mit eingeschränkter Prozessorfunktion, wie z.B. DMA-Controller (sie übernehmen die Datenübertragung zwischen dem Speicher und den Interfaces, siehe 8.1). – Obwohl solche Systeme mehrere Master besitzen, werden sie wegen der festen Zuordnung des Busses zum Mikroprozessor und wegen der eingeschränkten Masterfunktionen der Controller als *Einprozessorsysteme* bezeichnet.

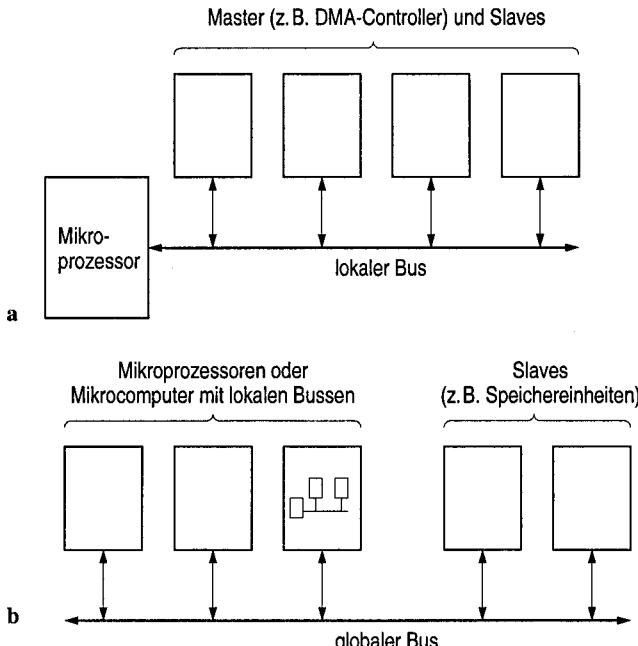


Bild 5-31. Mehrmastersysteme. **a** Einprozessorsystem (lokale Busarbitration), **b** Mehrprozessor- bzw. Mehrrechnersystem (globale Busarbitration)

Bei Systemen mit sog. *globaler Busarbitration* ist der Buszugriff für alle Master in gleicher Weise organisiert, d.h., der Bus ist nicht mehr einem der Master fest zugeordnet, sondern wird als *globaler Bus* auf Anforderung nach Prioritäten vergeben (Bild 5-31b). Die Master sind hierbei üblicherweise universelle oder spezielle Mikroprozessoren (Zentralprozessoren, Ein-/Ausgabeprozessoren) oder eigenständige Mikrorechner (Mikroprozessorsysteme mit lokalem Bus und ggf. lokaler Busarbitration, wie im Bild angedeutet). Solche Systeme werden als *Mehrprozessor-* bzw. *Mehrrechnersysteme* bezeichnet. – Anmerkung: Zugriffe eines Prozessors auf Komponenten (z.B. Speicher) seines lokalen Busses und auf solche des globalen Busses werden durch die Festlegung von lokalen und globalen Adressbereichen innerhalb des Gesamtadreßraums unterschieden.

5.4.1 Buszuteilung

Die Buszuteilung sei an einem einfachen Beispiel eines Systems mit lokaler Busarbitration, bestehend aus einem Mikroprozessor und einem *DMA-Controller* als einzigm zusätzliche Master, demonstriert. Aufgrund des nur einen Zusatzmasters entfällt hier die prozessorexterne Prioritätenlogik, so daß die Buszuteilung direkt zwischen dem Mikroprozessor und dem DMA-Controller „ausgetauscht“ werden kann. Dazu gibt es zwei Signale, über die die beiden Master miteinander verbunden sind, ein Busanforderungssignal \overline{BRQ} (bus request) vom DMA-Controller zum Mikroprozessor und ein Busgewährungssignal \overline{BGT} (bus grant) vom Mikroprozessor zum DMA-Controller. Das Zeitverhalten dieser Signale während der Busabgabe an den DMA-Controller (*Buszuteilungszyklus*) und der späteren Busrückgabe durch den DMA-Controller zeigt Bild 5-32. Der Mikroprozessor und der DMA-Controller benutzen in diesem Beispiel einen *synchronen Bus* und haben dementsprechend einen gemeinsamen *Bustakt BUSCLK*. Entsprechend dem synchronen Buszyklus nach Bild 5-26b, S. 317, beginnen und enden auch hier die Buszyklen mit steigenden Taktflanken.

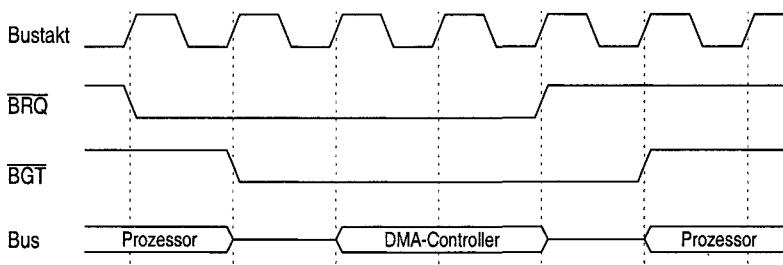


Bild 5-32. Buszuteilungszyklus für einen DMA-Controller in einem System mit lokaler Busarbitration

Um eine Datenübertragung durchzuführen, fordert der DMA-Controller den Bus vom Mikroprozessor durch Aktivieren seines Anforderungssignals \overline{BRQ} an. Der Mikroprozessor, der dieses Signal auswertet, erkennt den Aktivzustand. Er führt daraufhin zunächst seinen aktuellen Buszyklus noch zu Ende und gibt den Bus dann frei (im Bild bereits nach einem Takt). Dazu schaltet er seine Tristate-Ausgänge in den hochohmigen Zustand und signalisiert die *Busfreigabe* durch Aktivieren seines Gewährungssignals \overline{BGT} . Der DMA-Controller übernimmt daraufhin einen Takt später den Bus und führt seine Datenübertragung aus, die abhängig von der Betriebsart des Controllers aus nur einem Buszyklus (*cycle-steal mode*, wie im Bild angenommen) oder aus mehreren Buszyklen (*burst mode, block mode*) bestehen kann. Nach seiner Datenübertragung gibt der DMA-Controller seinerseits den Bus wieder frei, indem er seine Ausgänge in den hochohmigen Zustand schaltet. Er zeigt dies durch Inaktivieren seines \overline{BRQ} -Signals an. Der

Mikroprozessor nimmt daraufhin sein \overline{BGT} -Signal zurück und koppelt sich wieder an den Bus an.

Bei einem *asynchronen Bus* können der Prozessor und der Zusatzmaster unterschiedlich getaktet sein. Das bedeutet, daß sich der Prozessor mit dem \overline{BRQ} -Signal des Zusatzmasters und der Zusatzmaster mit dem \overline{BGT} -Signal des Prozessors synchronisieren müssen. Wie in 5.3.2 gezeigt, wird hierzu jeweils ein zusätzlicher Takt benötigt, so daß die Pausen während der Buszuteilung und der Busfreigabe, in denen der Bus nicht genutzt werden kann, größer sind als beim synchronen Bus.

Die meisten 8- und 16-Bit-Mikroprozessoren, wie auch die älteren 32-Bit-Mikroprozessoren (z.B. *MC68020*, *i386*) sind für die lokale Busarbitration, d.h. für Einprozessor-/Mehrmaster-systeme ausgelegt. Bei ihnen wirkt das Bus-Request-Signal \overline{BRQ} als Eingangssignal und das Bus-Grant-Signal \overline{BGT} als Ausgangssignals des Prozessors, so wie oben beschrieben. Werden sie in Systemen mit globaler Busarbitration eingesetzt, so muß die Zugriffsanforderung auf den globalen Bus, d.h. das \overline{BRQ} -Signal, aus der mit dem Zugriff verbundenen Adresse abgeleitet werden. Die neueren 32- und 64/32-Bit-Prozessoren hingegen unterstützen die globale Busarbitration, d.h. Mehrprozessorsysteme. Hier fungiert \overline{BRQ} als Ausgangssignal des Prozessors und \overline{BGT} als Eingangssignal. Die Arbitrationslogik ist dementsprechend aus den Prozessoren ausgelagert (z.B. *MC68040*, *PowerPC 601* und deren Nachfolger).

In der Kombination beider Möglichkeiten gibt es, wie bei den älteren Prozessoren, \overline{BRQ} und \overline{BGT} als Eingangs- bzw. Ausgangssignal und ein zusätzliches Busanforderungssignal als Ausgangssignal (z.B. *i486*, *Pentium*). Eine nochmals andere Konstellation weisen Prozessoren auf, die für den Aufbau von *SMP-Systemen* ausgelegt sind (5.1.7). Hier haben die Prozessoren zwei oder vier Bus-Request-Signale, mit denen zwei bzw. vier Prozessoren im Sinne einer *globalen Busarbitration* miteinander verbettet werden können. Jeweils eines der Signale wirkt als Ausgang, die anderen als Eingänge. Ein weiteres Anforderungssignal läßt einen zusätzlichen Master zu. Die Arbitrationslogik liegt hier wieder in den Prozessoren (z.B. *Pentium Pro*, ausgelegt für Vierprozessorsysteme).

5.4.2 Systemstrukturen

Erweitert man die einfache Struktur aus 5.4.1 um weitere Master, so wird über die im Mikroprozessor vorhandene Arbitrationslogik hinaus eine prozessorexterne Steuerlogik für die *Buszuteilung* (einschließlich der Priorisierung) erforderlich. Diese kann entweder dezentral realisiert sein, indem sie anteilig den Mastern zugeordnet ist (Bilder 5-33 und 5-35), oder sie kann zentral in einem für alle Master gemeinsamen Busarbiterbaustein zusammengefaßt sein (Bild 5-34). Die Signallaufzeit in der Priorisierungslogik (Priorisierungszeit, settle-up time) wird bei

einfachen Arbitrierungsschaltungen erst nach der Busfreigabe wirksam, so daß sich bis zur Busübernahme durch den neuen Master eine Bus-Totzeit ergibt, während der Bus nicht genutzt werden kann (*nichtüberlappende Busarbitration*). Bei aufwendigeren Arbitrierungsschaltungen überlappt sich die Priorisierungszeit mit dem Buszugriff des momentanen Busmasters, so daß der Bus unmittelbar nach seiner Freigabe vom neuen Master übernommen werden kann (*überlappende Busarbitration*).

Daisy-Chain. Bild 5-33 zeigt eine Struktur mit lokaler Busarbitration, bei der die Master eine Prioritätenkette (*busarbiter daisy-chain*) bilden. Jeder Master hat dazu einen eigenen Arbiter, der mit dem des Vorgängers und des Nachfolgers durch das Gewährungssignal \overline{BGT} verbunden ist. Dabei hat der Master, der dem Prozessor am nächsten ist, die höchste Priorität. Die Prioritäten der weiteren Master nehmen mit jedem Glied der Kette um eine Stufe ab. Die Priorisierungszeit ist gleich der Laufzeit des \overline{BGT} -Signals durch diese Kette. Da die Logik für die Priorisierung und Buszuteilung auf die einzelnen Glieder der Kette aufgeteilt ist, spricht man auch von *dezentraler Busarbitration*.

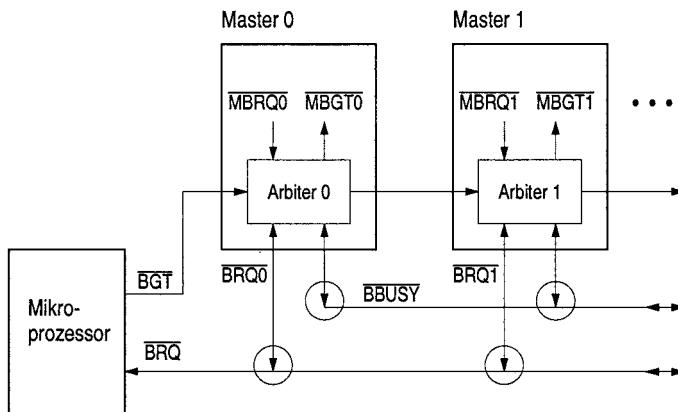


Bild 5-33. Dezentrale Busarbitration in einer Daisy-Chain bei lokaler Struktur (mit einem Mikroprozessor als ausgezeichnetem Master)

Die Master melden ihre *Busanforderungen* mit $\overline{MBRQi} = 0$ (master bus request) bei ihren Arbitern an, und erhalten von diesen nach erfolgter Buszuteilung das Gewährungssignal $\overline{MBGTi} = 0$ (master bus grant). Die Arbitren ihrerseits erzeugen Anforderungssignale $\overline{BRQi} = 0$, die durch *verdrahtetes Oder* (Kreissymbol) als \overline{BRQ} -Signal dem Prozessor zugeführt werden. Der Prozessor wiederum gibt sein Gewährungssignal $\overline{BGT} = 0$ an den ersten Arbeiter in der Kette aus, der dieses Signal nur dann an seinen Kettennachfolger weitergibt, wenn er selbst keine Anforderung hat. In gleicher Weise verfahren die anderen Kettenglieder, so daß derjenige anfordernde Master i den Bus zugeteilt erhält, dessen Arbeiter nach Ablauf

der Priorisierungszeit an seinem Verkettungseingang das Aktivsignal $\overline{BGT} = 0$ vorfindet.

Diese Art der Arbitrierung hat zunächst den Nachteil, daß ein anfordernder Master mit niedriger Priorität bei einer Häufung höherpriorisierter Anforderungen den Bus nicht zugeteilt bekommt und somit „ausgehungert“ werden kann. Man spricht hier von „*unfairer*“ *Priorisierung*. In der Schaltung nach Bild 5-33 wird dieser Nachteil jedoch dadurch vermieden, daß ein Arbiter, der eine neue Anforderung $\overline{MBRQi} = 0$ erhält, diese so lange zurückhält, d.h. $\overline{BRQi} = 1$ erzeugt, wie er an seinem bidirektionalen \overline{BRQi} -Anschluß den Aktivpegel $\overline{BRQ} = 0$ vorfindet. Dadurch können die bereits vorliegenden Anforderungen als Gruppe abgearbeitet werden, ohne daß neue hinzukommen (*gruppenweise Priorisierung*). Neue Anforderungen werden danach in einen neuen Zuteilungszyklus aufgenommen und wiederum abhängig von ihrer Position in der Kette priorisiert. Man bezeichnet dies als „*faire*“ *Priorisierung*. Ihr Nachteil ist, daß Master, die nicht beliebig lange warten können, den Bus unter Umständen zu spät zugeteilt bekommen. Dies betrifft z.B. DMA-Controller mit von außen vorgegebener Übertragungsgeschwindigkeit.

Durch den für alle Master vorhandenen bidirektionalen \overline{BBUSY} -Anschluß (bus busy) kann die *Arbitrierung überlappend* mit der Busbenutzung erfolgen. Dazu zeigt der Arbiter, der die Buszuteilung erhält, allen anderen Arbitern die Busbelegung durch $\overline{BBUSY} = 0$ an. Bereits zu diesem Zeitpunkt gibt er das an seinem Verkettungseingang anliegende Gewährungssignal an seinen Kettennachfolger weiter, so daß die Priorisierung für die nächste Buszuteilung bereits mit der Busübernahme beginnt. Der Arbiter mit nächstniedrigerer Priorität in der Gruppe wertet das daraus resultierende Gewährungssignal erst dann als Buszuteilung, wenn der busbelegende Master den Bus wieder freigibt und dies durch $BBUSY = 1$ anzeigt. – Der Vorteil von Daisy-Chain-Realisierungen liegt in der auf dem Systembus erforderlichen geringen Leitungsanzahl. Nachteilig hingegen ist, zumindest bei nichtüberlappender Arbitrierung, daß mit jedem zusätzlichen Kettenglied die Priorisierungszeit größer wird.

Zentraler Busarbiter. Bild 5-34 zeigt eine Struktur mit lokaler Busarbitration, die die gleiche Funktion wie die Daisy-Chain in Bild 5-33 aufweist. Bei ihr sind die einzelnen Arbiterschaltungen der Master in einem *Busarbiterbaustein* zusammengefaßt (*zentrale Busarbitration*); das \overline{BBUSY} -Signal ist dementsprechend nach außen hin nicht mehr sichtbar. Dies gewährt zunächst einen einfacheren Systemaufbau; darüber hinaus läßt sich der Arbiterbaustein in einfacher Weise auch für die globale Busarbitration verwenden. Dazu braucht wegen des Wegfalls des zentralen Mikroprozessors der Arbiterausgang \overline{BRQ} lediglich mit dem Arbitereingang \overline{BGT} kurzgeschlossen zu werden. – Nachteilig ist die mit der zentralen Anordnung verbundene *größere Leitungsanzahl* auf dem Bus, da für jeden Master eine \overline{MBRQi} - und eine \overline{MBGTi} -Leitung erforderlich sind.

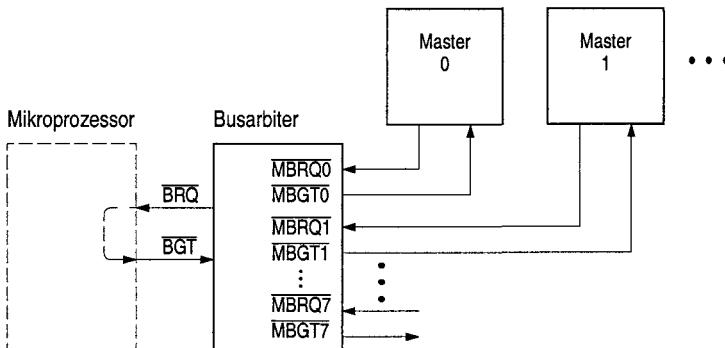


Bild 5-34. Zentrale Busarbitration durch einen für alle Master gemeinsamen Busarbiter; wahlweise lokale Struktur (mit Mikroprozessor als ausgezeichnetem Master) oder globale Struktur (ohne Mikroprozessor als ausgezeichnetem Master)

Identifikationsbus. Bild 5-35 zeigt eine Struktur mit *globaler Busarbitration* und dezentraler Anordnung der Arbitrerlogik. Die Signale \overline{BRQ} und \overline{BBUSY} haben hierbei die gleiche Funktion wie in der oben beschriebenen Daisy-Chain, d.h., es wird eine *faire Priorisierung* in Gruppen bei *überlappender Busarbitrierung* durchgeführt. Anders als bei der Daisy-Chain erfolgt die Priorisierung jedoch nicht durch Verkettung der einzelnen Arbiter, sondern über Identifikationsnummern (ID-Nummern), die von den anfordernden Arbitern auf einem Identifikationsbus (ID-Bus) ausgegeben werden. Jeder Arbitr hat dazu ein Register, in das durch das Programm eine ID-Nummer entsprechend der dem Master zugewiesenen Priorität geladen wird.

Ferner hat jeder Arbitr eine Priorisierungslogik, mit der er die auf dem ID-Bus ggf. gleichzeitig vorliegenden Anforderungen, die dort *wired-or*-verknüpft werden, auswertet. Diese Auswertung verläuft bei mehreren Anforderungen wie folgt: Alle Arbitre vergleichen Bit für Bit „von oben“, d.h. bei der höchstwertigen Bitposition beginnend, ihre ID-Nummern mit deren wired-or-Verknüpfungen auf dem ID-Bus und setzen jeweils ab der Stelle, an der eine gesendete 0 von einer 1 reflektiert wird, ihre rechts davon liegenden 1-Bits auf 0. Nach einer Einschwingzeit verbleibt die höchste ID-Nummer auf dem Bus, und der Arbitr, der keines seiner 1-Bits zurücksetzen mußte, „gewinnt“ die Priorisierung. Dies meldet er über sein MBGTi-Signal seinem Master. Bild 5-36 zeigt diesen Vorgang am Beispiel von 6-Bit-ID-Nummern für vier gleichzeitig gestellte Anforderungen mit den Prioritäten 21, 20, 19 und 8. Der Einschwingvorgang stellt sich hier in zwei Schritten dar.

In Erweiterung der gruppenweisen (fairen) Priorisierung können in diesem System Master, die nicht warten dürfen, privilegiert behandelt werden. Sie zeigen dazu ihren privilegierten Status durch eine 1 im höchstwertigen Bit ihrer ID-Nummer an. Ihre Anforderungen $\overline{MBRQi} = 0$ durchbrechen damit die aktuelle Gruppenpriorisierung und werden unmittelbar in den nächsten Priorisierungs-

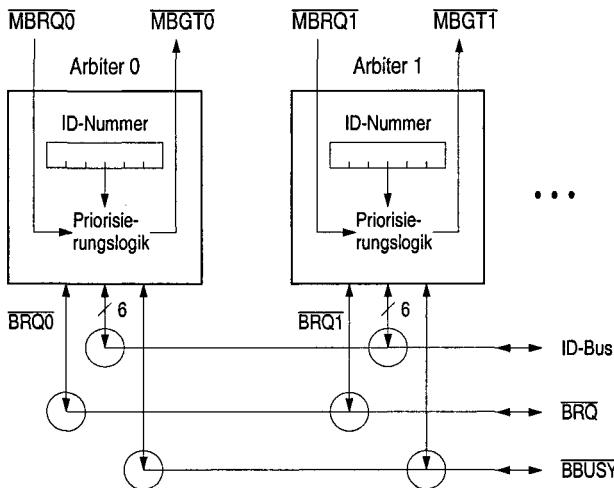


Bild 5-35. Dezentrale Busarbitration mit einem Identifikationsbus bei globaler Struktur

zyklus mit aufgenommen. Als eine Realisierung dieser Art der Busarbitration sei der *Multibus II* [Intel 1984] genannt. – Der Vorteil der Busarbitration mit Identifikationsbus liegt in der flexiblen Prioritätenvergabe, die dynamisch, d. h. programmierbar, erfolgen kann. Hinzu kommt die Möglichkeit, ggf. die faire Priorisierung zu durchbrechen. Nachteilig ist neben dem hohen Logikaufwand in den Arbitren die größere Leitungsanzahl gegenüber der Daisy-Chain; sie ist jedoch geringer als beim zentralen Busarbiter.

	Start	Schritt1	Schritt2	
Anforderer 21	010101	010101	010101	
"	010100	010100	010100	
"	010011	010 <u>0</u> 11	010 <u>0</u> 00	
"	0 <u>0</u> 1000	0 <u>0</u> 0000	0 <u>0</u> 0000	
ID-Bus	011111	010111	010101	→ Buszuteilung an Nr. 21

Bild 5-36. Priorisierung von vier Anforderungen mit abnehmenden Prioritäten 21, 20, 19 und 8 in einem System nach Bild 5-35. Darstellung der auf den Identifikationsbus ausgegebenen ID-Nummern der Anforderer und der sich auf dem ID-Bus einstellenden wired-or-Verknüpfungen. Fett-darstellung der jeweils betrachteten Bitposition, Unterstreichung der für die Änderungen von 1 nach 0 signifikanten 0-Bits. Buszuteilung an den Anforderer 21 nach Ablauf der Einschwingzeit

5.5 Interruptsystem und Systemsteuersignale

Ein Interrupt wird durch ein Anforderungssignal einer *Interruptquelle* an den Mikroprozessor verursacht. Dabei wird die Verarbeitung des laufenden Programms unterbrochen und ein der Interruptquelle zugeordnetes *Interruptprogramm* ausgeführt. Typische Interruptquellen sind z.B. Ein-/Ausgabegeräte,

wie Tastaturen und Drucker, und Hintergrundspeicher, wie Floppy-Disk- und Plattenlaufwerke. Als Interruptquellen können auch andere „prozessornähe“ Funktionseinheiten, z.B. ein Zeitgeber (timer), aber auch „prozessorferne“ Funktionseinheiten, z.B. Sensoren eines extern ablaufenden technischen Prozesses, wirken.

Während in Kapitel 2 die Interruptverarbeitung aus der Sicht des Programmierens beschrieben wurde, behandelt dieser Abschnitt den Aufbau von Interruptsystemen aus der Sicht des Systemingenieurs. Schwerpunkte sind hierbei die prozessor-externe Priorisierung von Interruptanforderungen und der Signalaustausch zwischen Interruptquelle und Mikroprozessor. Darüber hinaus betrachten wir als spezielle Unterbrechungssignale die Systemsteuersignale \overline{BERR} und \overline{RESET} sowie ein vom Unterbrechungssystem unabhängiges Steuersignal, das \overline{HALT} -Signal. – In unseren Ausführungen beziehen wir uns wegen der bei RISCs stark reduzierten Interrupt-Hardwareausstattung hauptsächlich auf CISC-Prozessoren.

5.5.1 Codierte Interruptanforderungen

Prioritätenbaustein. Bei Mikroprozessoren die, wie in Kapitel 2 beschrieben, mehrere Interruptebenen unterscheiden, müssen die Anforderungssignale der Interruptquellen prozessorextern durch einen Prioritätenbaustein codiert werden (*Prioritätencodierer*). Die Struktur und die Wirkungsweise eines solchen Bausteins zur Priorisierung von sieben pegelsensitiven *Interruptsignalen* $\overline{IRQ7}$ bis $\overline{IRQ1}$ zeigt Bild 5-37 ($\overline{IRQ0} = 0$ bedeutet „keine Anforderung“). – Alle acht Signale durchlaufen eine Prioritätenlogik, die bei mehreren aktiven Signalen $\overline{IRQ_i} = 0$ das jeweils höchstpriorisierte Signal, d.h. das Signal mit dem höchsten Index, durchschaltet. Dieser 1-aus-8-Code, der eine von sieben Anforderungen oder keine Anforderung durch eine 1 darstellt, wird von einem nachgeschalteten Codierer in einen 3-Bit-Code umgesetzt, der als *Interruptcode* dem Prozessor zugeführt wird. Die höchstpriorisierte Anforderung $\overline{IRQ7} = 0$ erzeugt (unabhängig von den restlichen sieben Interruptsignalen) den Code 7 (vom Prozessor als Prioritätsebene 7 interpretiert) und die niedrigstpriorisierte Anforderung $\overline{IRQ1} = 0$ (bei $\overline{IRQ2}$ bis $\overline{IRQ7}$ gleich 1) den Code 1 (Prioritätsebene 1). Keine Anforderung $\overline{IRQ0} = 0$ (bei $\overline{IRQ1}$ bis $\overline{IRQ7}$ gleich 1) erzeugt den Code 0 (Prioritätsebene 0).

Interruptzyklus. Bild 5-38 zeigt die Struktur eines Mikroprozessorsystems unter Benutzung des Prioritätencodierers nach Bild 5-37 und des in Kapitel 2 beschriebenen CISC-Prozessors. Von den sieben möglichen Interruptquellen sind nur zwei benutzt, davon eine mit *Vektor-Interrupt* (Ebene 6) und eine mit *Autovektor-Interrupt* (Ebene 1). Das Wechselspiel der Signale zwischen Interruptquelle und Mikroprozessor zeigt Bild 5-39 in Verbindung mit Bild 5-38. Es beschreibt den prinzipiellen Ablauf im Prozessor durch einen Zustandsgraphen, wobei die einzelnen Zustandsübergänge nicht mit jedem Prozessortakt, sondern nach mehreren Taktschritten erfolgen.

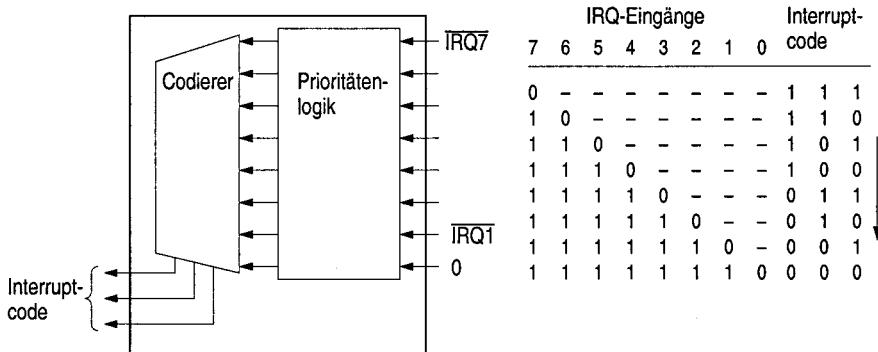


Bild 5-37. Prioritätencodierer für sieben Interruptebenen. Prioritäten von oben nach unten abnehmend

Die Unterbrechungsanforderung einer Interruptquelle wird über den Prioritätencodierer als 3-Bit-*Interruptcode* an den Prozessor übermittelt. Damit dieser die Anforderung erkennen kann, muß der Interruptcode wenigstens einen Takt an den Interrupeingängen stabil anliegen. (Der Prozessor muß sich auf die asynchrone Anforderung „aufsynchronisieren“!). Ist dies erfüllt, so entscheidet der Prozessor im Zustand Zi (also nachdem er seinen momentanen Buszyklus beendet hat), ob er der Anforderung stattgibt oder nicht.

Er übernimmt dazu den Interruptcode in ein internes Pufferregister und vergleicht ihn mit der im Prozessorstatusregister vorliegenden *Interruptmaske*. Bei positivem Ausgang des Vergleichs (Interruptcode > Maske) akzeptiert er die Anforderung und kopiert zunächst den Inhalt seines Statusregisters in ein weiteres Pufferregister (in Bild 5-38 nicht gezeigt). Anschließend setzt er die Maske gleich dem Interruptcode. Damit blockiert er Anforderungen gleicher und Anforderungen niedrigerer Prioritäten. Darüber hinaus setzt er das Statusbit S und schaltet damit in den Supervisor-Modus um. Außerdem setzt er die beiden Trace-Bits zurück, um einen eventuellen Trace-Vorgang auszusetzen.

Im Zustand Zk quittiert der Prozessor die Programmunterbrechung mit dem Signal $\overline{IACK} = 0$. Gleichzeitig gibt er auf den Adreßleitungen A2 bis A0 den Interruptcode aus und zeigt dies durch Setzen des \overline{AS} -Signals an (asynchroner Bus). Mittels dieser Signale wird über einen prozessorexternen Demultiplexer das Quittungssignal $\overline{IACK}_i = 0$ für die akzeptierte Interruptquelle erzeugt. Wurde der Interrupt dieser Ebene durch eine Vektor-Interruptquelle ($AVEC = 1$) ausgelöst, so führt der Prozessor einen Buszyklus zum Lesen der *Vektornummer* aus. In diesem Fall wird das Signal $\overline{IACK}_i = 0$ zur Anwahl des Vektornummerregisters VNR der Quelle benutzt.

Die Übergabe der Vektornummer erfolgt in unserem Beispiel auf den Datenleitungen D31 bis D24 (8-Bit-Port bei dynamischer Busbreite). Sie wird entspre-

chend dem in 5.3.2 beschriebenen asynchronen Lesezyklus mit den Signalen \overline{DS} und \overline{DTACK} synchronisiert.

Bei einem Autovektor-Interrupt ($\overline{AVEC} = 0$) entfällt das Lesen der Vektornummer; sie wird statt dessen prozessorintern abhängig vom akzeptierten Interruptcode erzeugt. In diesem Fall wird das Signal $\overline{IACK}_i = 0$ zur Bildung von $\overline{AVEC} = 0$ benutzt. (Sind mehrere Autovektor-Interruptquellen vorhanden, so werden ihre \overline{IACK}_i -Signale zur Bildung des \overline{AVEC} -Signals durch ein Oder-Gatter zusammengefaßt.)

Im Zustand Z_n speichert der Prozessor seinen Status, indem er den Inhalt des Befehlszählers und den gepufferten Statusregisterinhalt auf den Supervisor-Stack schreibt ($S = 1!$). Anschließend multipliziert er die Vektornummer mit Vier (2-Bit-Linksshift, Wortadresse), adressiert damit im Zustand Z_p die *Vektortabelle*, liest aus ihr die Startadresse des Interruptprogramms (Interruptvektor) und lädt sie in den Befehlszähler. Mit dem im Zustand Z_0 folgenden Befehlsabruf verzweigt er damit in das entsprechende Interruptprogramm.

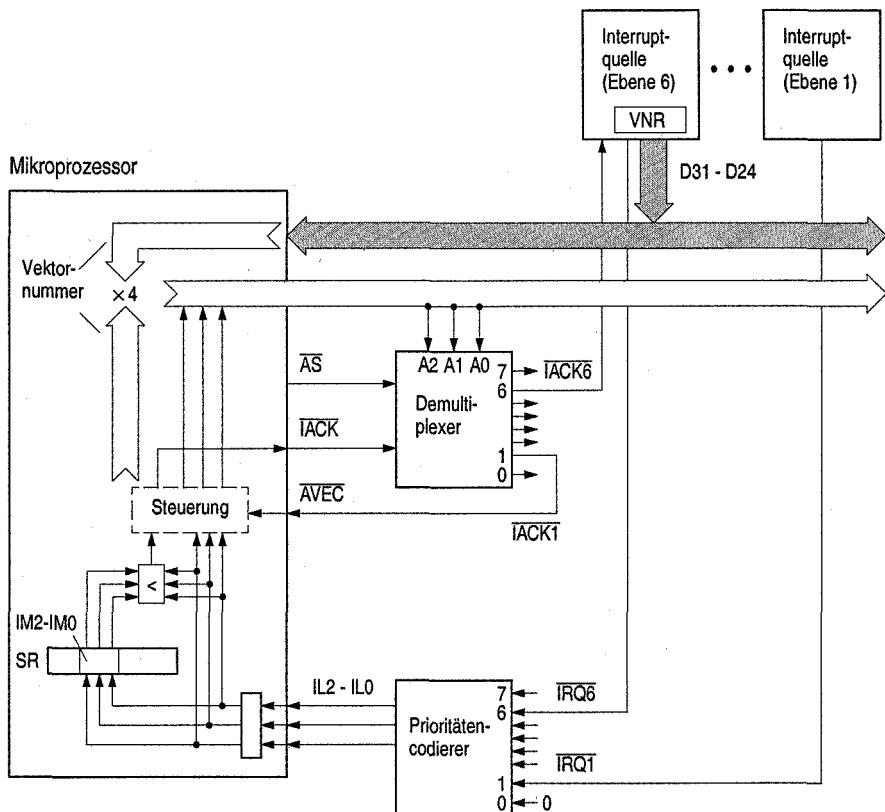


Bild 5-38. Systemstruktur für codierte Interruptanforderungen

Den gesamten Ablauf zwischen der Entgegennahme einer Interruptanforderung durch den Prozessor und dem Aufruf des zugehörigen Interruptprogramms bezeichnet man als *Interruptzyklus*, den darin enthaltenen Signalaustausch zur Ermittlung der Vektornummer als *Interrupt-Acknowledge-Zyklus*.

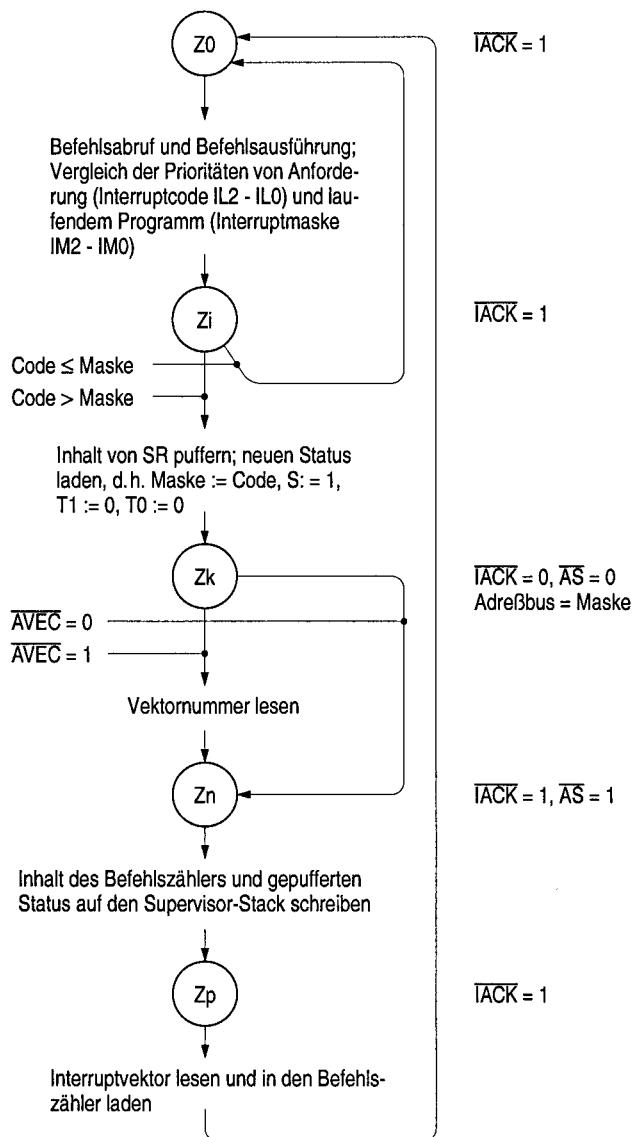


Bild 5-39. Zustandsgraph für die Behandlung codierter Interruptanforderungen durch den Mikroprozessor (Interruptzyklus)

Die Vektorisierung von Interrupts ist dann sehr nützlich, wenn innerhalb einer Interruptebene mehrere Interruptquellen verwaltet werden. Diese haben dann notwendigerweise denselben Interruptcode (siehe 5.5.2), können jedoch vom Prozessor über ihre Vektornummer identifiziert werden. Grundsätzlich muß dabei der Prozessor in der Lage sein, weitere Anforderungen dieser Ebene zu blockieren, bis die bereits akzeptierte Anforderung bearbeitet ist. Deshalb sind *vektorierte Interrupts* immer durch das Statusregister des Prozessors maskierbar. Damit nun blockierte Anforderungen nicht verlorengehen, müssen *maskierbare Interrupts* als Pegel anliegen (pegelsensitive Interrupts). Demgegenüber wird einem *nichtmaskierbaren Interrupt*, im obigen Beispiel dem Interrupt höchster Priorität (siehe 2.1.5), unabhängig von der Interruptmaske im Statusregister immer stattgegeben; er wird quasi durch seine Flanke wirksam (flankensensitiver Interrupt).

5.5.2 Uncodierte Interruptanforderungen

In 5.5.1 wurde von einem Mikroprozessor ausgegangen, der über einen 3-Bit-Interrupteingang (codierte Anforderungen) und eine 3-Bit-Interruptmaske verfügt und dementsprechend sieben Interruptebenen nach Prioritäten unterscheidet. In dieser Weise arbeiten z.B. die CISC-Prozessoren der *MC68000-Familie*. Aber auch RISCs verwenden diese Technik, so z.B. der *SPARC*, der einen 4-Bit-Interrupteingang und dementsprechend eine 4-Bit-Interruptmaske hat. Mikroprozessoren, die diese Mehrebenenstruktur nicht vorsehen, unterscheiden statt dessen einzelne Interruptarten, denen getrennte Interrupteingänge zugeordnet sind (uncodierte Interruptanforderungen), so z.B. einen nichtmaskierbaren und einen maskierbaren, vektorisierten Eingang und ggf. zusätzlich zu diesen beiden einen maskierbaren, nichtvektorisierten Eingang. Diese Eingänge sind zwar untereinander priorisiert, die Anzahl der Interruptebenen ist jedoch ohne zusätzlichen prozessorexternen Schaltungsaufwand auf die Anzahl dieser Interrupteingänge begrenzt.

Unabhängig davon, ob ein Interruptsystem mit codierten oder uncodierten Anforderungen arbeitet, kann jeder Interruptebene mehr als eine Interruptquelle zugeordnet sein. Die Priorisierung dieser Quellen erfolgt dabei entweder prozessorintern durch das Programm (z.B. in einer Polling-Routine) oder prozessorextern durch zusätzliche Hardware. Die Zusatzhardware ist entweder auf die einzelnen Interruptquellen in einer Daisy-Chain verteilt (*dezentrale Priorisierung*), oder sie ist für alle Quellen räumlich in einem Interrupt-Controller zusammengefaßt (*zentrale Priorisierung*). Diese Alternativen werden im folgenden an Interruptsystemen mit uncodierten Anforderungen demonstriert.

Priorisierung durch Polling. Bild 5-40 zeigt einen Prozessor mit einem maskierbaren, nichtvektorisierten Interrupteingang \overline{IRQ} . Diesem Eingang ist im Prozessorstatusregister SR ein *Interruptmaskenbit* IM zugeordnet, das in seiner Funktion der 3-Bit-Maske bei einem 3-Bit-Interrupteingang entspricht. Ferner ist

ihm intern eine Nummer (*Autovektornummer*) zugeordnet, die den Ort der Startadresse des Interruptprogramms angibt. – Die einzelnen Interruptquellen weisen ihre Anforderungen durch ein lokales Statusbit $IP_i = 1$ (interrupt pending) aus und bilden daraus Anforderungssignale $\overline{IRQ}_i = 0$. Diese werden durch *verdrahtetes Oder* (Kreissymbol) zusammengefaßt und so am \overline{IRQ} -Eingang des Prozessors angezeigt. Um eine akzeptierte Anforderung wieder zu löschen, muß das zugehörige IP-Bit durch die Software zurückgesetzt werden.

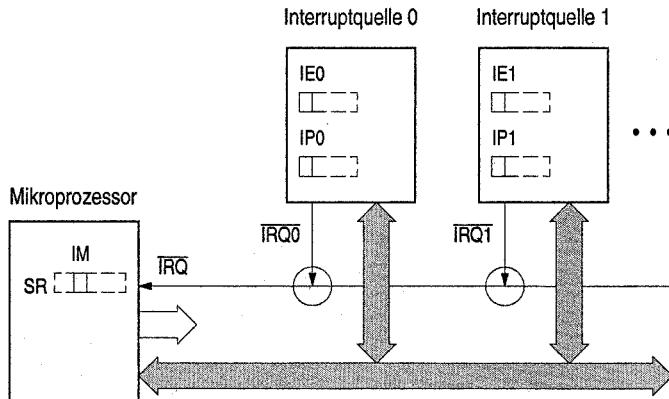


Bild 5-40. Interruptsystem mit mehreren Interruptquellen; Priorisierung und Identifizierung durch die Software

Der Prozessor, der nun nicht unterscheiden kann, von welcher Quelle eine Anforderung kommt und ob eine oder mehrere Anforderungen vorliegen, führt, sofern sein Maskenbit nicht gesetzt ist, einen Interruptzyklus durch und verzweigt auf ein für alle Interruptquellen gemeinsames Interruptprogramm. Dieses sieht für jede Quelle eine eigene Service-Routine vor, z.B. in Form eines Unterprogramms. Nach Eintritt in das Interruptprogramm bleibt die Interruptanforderung als Pegel so lange erhalten, bis das IP-Bit der Interruptquelle vom Interruptprogramm zurückgesetzt wird. Das Setzen des Interruptmaskenbits im Interruptzyklus verhindert jedoch, daß diese Anforderung oder eine andere Anforderung inzwischen erneut zu einer Unterbrechung führt.

Das Auffinden derjenigen Quelle, die die Unterbrechung verursacht hat (Identifizierung), und das Festlegen der Reihenfolge der Bearbeitung bei mehreren vorliegenden Anforderungen (Priorisierung) erfolgen im Interruptprogramm. Dieses liest dazu die Statusregister der einzelnen Interruptquellen, identifiziert die anfordernden Quellen anhand der gesetzten IP-Bits und verzweigt auf die Service-Routine der Quelle höchster Priorität. Die Prioritäten sind dabei (natürlicherweise) durch die Reihenfolge der Abfrage festgelegt: die zuerst abgefragte Quelle hat die höchste Priorität. – Den Abfragevorgang mit Zugriff auf die einzelnen Statusregister bezeichnet man als Polling, die Abfragesequenz im Interruptprogramm als Polling-Routine (siehe auch 7.1.3).

Bei Systemkonfigurationen mit mehreren Quellen ist es üblich, die Quellen zusätzlich mit einem *Interrupt-Enable-Bit* IE auszustatten (Bild 5-41) und dieses Bit als Bestandteil des Steuerregisters einer jeden Quelle programmierbar zu machen. Damit können die Anforderungen einzelner Interruptquellen gezielt freigegeben oder blockiert werden. Dies ermöglicht es unter anderem, Interrupt-Service-Routinen für Anforderungen höherer Priorität durch Rücksetzen der Interruptmaske im Prozessorstatusregister unterbrechbar zu machen. (Anforderungen niedrigerer oder gleicher Priorität müssen dabei blockiert werden.)

Nichtunterbrechbare Daisy-Chain. Die Priorisierung und Identifizierung von Interrupts durch die Software ist sehr zeitaufwendig. Sie wird deswegen bei Systemen höherer Leistungsfähigkeit von einer Zusatzhardware durchgeführt. Bei Prozessoren mit uncodierten Anforderungen bedarf es dazu einer prozessorexternen Priorisierungs- und Identifizierungslogik. Der Prozessor seinerseits muß einen Interruptzyklus durchführen können, d.h., er muß ein Quittungssignal (\overline{IACK}) bereitstellen und einen Zyklus zum Lesen der Vektornummer ausführen können. (Bei Prozessoren mit codierten Anforderungen, wie in 5.5.1 beschrieben, ist ein Teil der Priorisierungs- und Identifizierungslogik im Prozessor vorgesehen.)

Eine der gebräuchlichsten Techniken der prozessorexternen Priorisierung ist das Zusammenschalten von Interruptquellen zu einer Prioritätenkette (*interrupt daisy-chain*). Jede Interruptquelle benötigt dazu eine Priorisierungsschaltung, die mit den Priorisierungsschaltungen des Vorgängers und des Nachfolgers über Signalleitungen verbunden ist (Bilder 5-41 und 5-43). Dabei hat die Interruptquelle, die sich am Anfang der Kette befindet, die höchste Priorität. Die Prioritäten der weiteren Interruptquellen nehmen mit jedem Glied in der Kette um eine Stufe ab. Da die Logik für die Priorisierung auf die einzelnen Glieder der Kette aufgeteilt ist, spricht man von *dezentraler Priorisierung*.

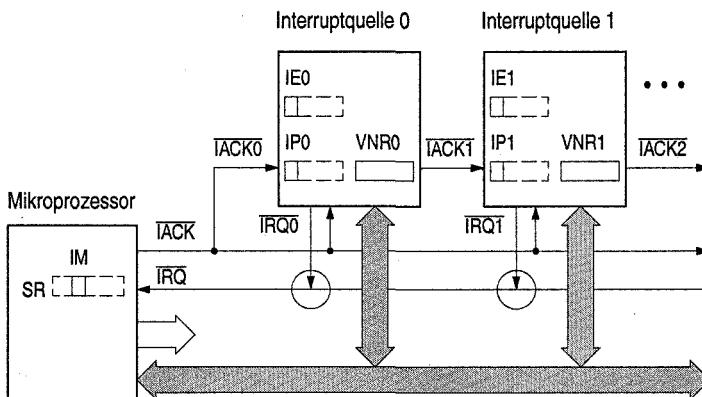


Bild 5-41. Interruptsystem mit nichtunterbrechbarer Daisy-Chain (dezentrale Priorisierung)

Bild 5-41 zeigt die Struktur einer Daisy-Chain, die eine Priorisierung mehrerer gleichzeitig vorliegender Anforderungen erlaubt, jedoch die Unterbrechung eines Interruptprogramms durch eine Anforderung höherer Priorität ausschließt. – Die Anforderungen der einzelnen Quellen werden durch *verdrahtetes Oder* zusammengefaßt und über den Interrupeingang \overline{IRQ} dem Prozessor zugeführt. Der Prozessor löst, sofern sein Maskenbit nicht gesetzt ist, einen *Interruptzyklus* und darin einen *Acknowledge-Zyklus* aus, den er durch Setzen des \overline{IACK} -Signals einleitet. \overline{IACK} wirkt auf die Priorisierungslogik und hat zwei Funktionen:

- Zum einen wirkt es auf die Interruptquellen direkt, um Anforderungen, die während des Acknowledge-Zyklus auftreten ($\overline{IACK} = 0$), zu blockieren und damit vom gerade laufenden Priorisierungsvorgang auszuschließen; dadurch können sich die Signale in der Kette bis zum Lesen der Vektornummer stabilisieren (Priorisierungszeit, siehe auch 5.4.2: Busarbiter-Daisy-Chain).
- Zum andern wirkt es auf den Verkettungseingang \overline{IACK} der ersten Quelle in der Daisy-Chain, um den Priorisierungsvorgang für die vor dem Acknowledge-Zyklus ($\overline{IACK} = 1$) vorliegenden Anforderungen durchzuführen.

Eine Quelle i , die eine Anforderung während $\overline{IACK} = 1$ anmeldet, verhindert mit ihrer Verkettungslogik die Weitergabe des Aktivpegels von \overline{IACK}_i an die nachfolgenden Kettenglieder. Damit wird gewährleistet, daß bei mehreren gleichzeitigen Anforderungen nur diejenige Interruptquelle das Quittungssignal des Prozessors erhält, die ihm in der Kette am nächsten ist und damit die höchste Priorität von allen Anforderungen hat. Der Begriff Gleichzeitigkeit bezieht sich hierbei auf den Zeitraum vom Eintreffen einer ersten Anforderung bis zum Aktivwerden des \overline{IACK} -Signals. Mit Erhalt des Aktivpegels $\overline{IACK}_i = 0$ ist die entsprechende Interruptquelle angewählt und legt ihre Vektornummer auf den Datenbus. Abgeschlossen wird der Acknowledge-Zyklus durch Rücksetzen von \overline{IACK} , womit in der Kette die Blockierung neuer Anforderungen wieder aufgehoben wird.

Während des Interruptzyklus setzt der Prozessor sein *Interruptmaskenbit*. Er blockiert damit sämtliche Anforderungen, auch diejenige, der er gerade stattgegeben hat und die erst im Interruptprogramm durch Rücksetzen des zugehörigen IP-Bits gelöscht wird. Die Unterbrechbarkeit wird erst mit Abschluß des Interruptprogramms wiederhergestellt, indem der ursprüngliche Prozessorstatus durch den RTE-Befehl (return from exception) wieder geladen und damit das System demaskiert wird.

Eine detaillierte Beschreibung des Signalverhaltens in der Prioritätenlogik einer Interruptquelle i zeigt Bild 5-42 in Form eines Zustandsgraphen. Der Graph enthält links die Bedingungen für die Zustandsänderungen und rechts die in den Zuständen erzeugten Signale und Busaktivitäten. (Das Zusammenspiel in einer ganzen Kette kann man am besten studieren, wenn man mehrere Glieder, z.B. drei, nebeneinander zeichnet und die aktiven Zustände mit Marken belegt.)

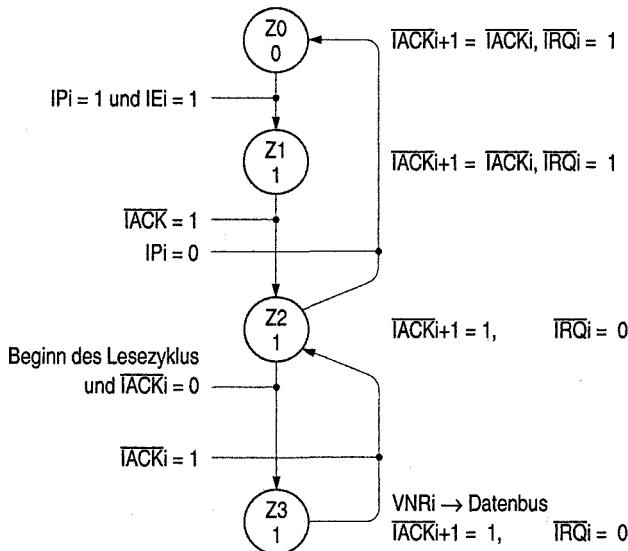


Bild 5-42. Zustandsgraph des Kettengliedes i der nichtunterbrechbaren Daisy-Chain nach Bild 5-41. Der 1-Bit-Code in den Zuständen gibt die Werte des Bits IPI an. Für das erste Kettenglied gilt $\overline{IACK}_0 = IACK$

Unterbrechbare Daisy-Chain. Bild 5-43 zeigt eine *Interrupt-Daisy-Chain*, die das Unterbrechen eines Interruptprogramms durch eine Anforderung höherer Priorität zuläßt. Dies ist jedoch nur möglich, wenn zuvor das *Interruptmaskenbit* durch den MOVESR-Befehl zurückgesetzt wird, d.h. bereits innerhalb der Interruptprogramms und nicht erst mit dessen Abschluß. – Anders als bei der Daisy-Chain nach Bild 5-41 wird das Signal am Verkettungseingang des ersten Kettengliedes nicht durch den \overline{IACK} -Ausgang des Prozessors bestimmt, sondern als prozessorunabhängiges Verkettungssignal (chain) mit $\overline{CH}_0 = 0$ vorgegeben.

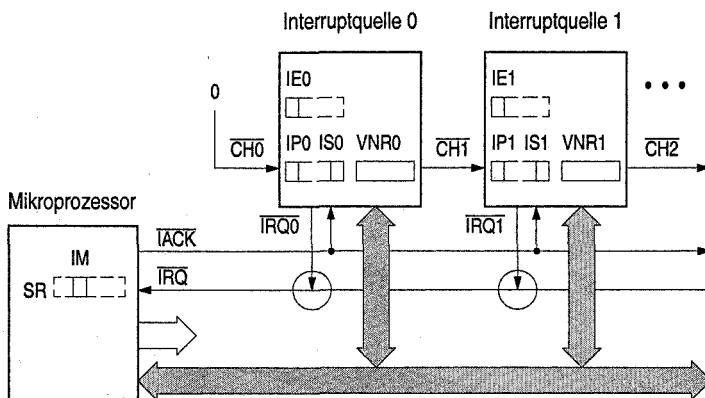


Bild 5-43. Interruptsystem mit unterbrechbarer Daisy-Chain (dezentrale Priorisierung)

Dadurch ist die Kette permanent aktiv, d.h., die Priorisierung ist nicht nur während des *Acknowledge-Zyklus*, sondern dauernd möglich. Die Funktion von \overline{IACK} beschränkt sich dabei auf das Blockieren von Anforderungen, die während eines Acknowledge-Zyklus auftreten, und auf die Ankündigung des Lesezyklus für die Vektornummer. Bereits in Bearbeitung befindliche Anforderungen (Interruptprogramm befindet sich in der Ausführung bzw. wurde darin unterbrochen) werden in den Interruptquellen durch ein Statusbit IS (interrupt serviced) angezeigt. Dieses Bit wirkt auf die Kette und blockiert mit $IS_i = 1$ die Anforderungen der in der Kette nachfolgenden Interruptquellen. Damit erhält der Prozessor an seinem \overline{IRQ} -Eingang nur dann eine Anforderung, wenn sie höhere Priorität als das laufende Interruptprogramm hat. Das IS-Bit wird im Interruptprogramm erst unmittelbar vor dem Rücksprungbefehl RTE wieder zurückgesetzt; das IP-Bit kann früher gelöscht werden. – Bild 5-44 zeigt den Zustandsgraphen für die Prioritätenlogik einer Interruptquelle i. Im Gegensatz zur nichtunterbrechbaren Daisy-Chain, bei der alle Anforderungen vom Zustand Z2 ausgehen (Bild 5-42), unter-

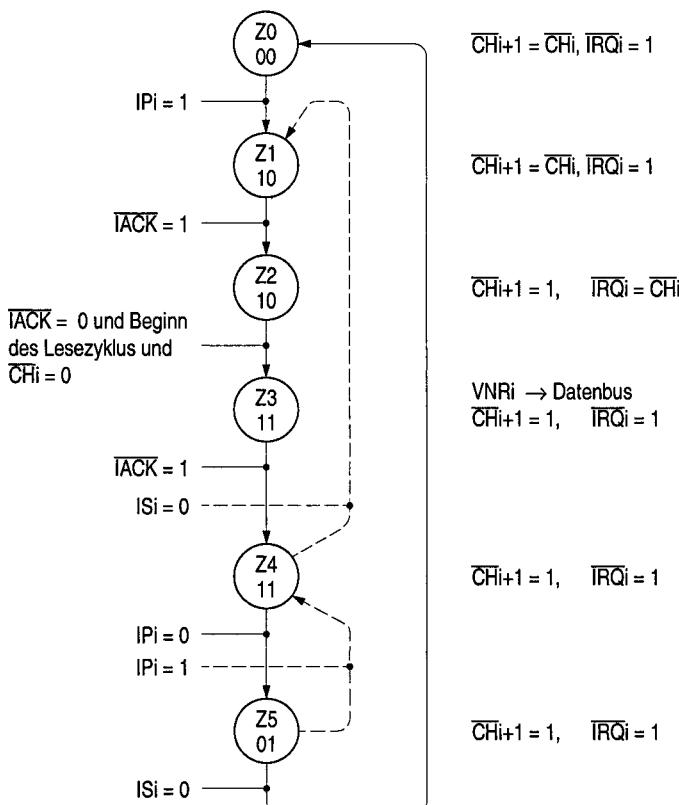


Bild 5-44. Zustandsgraph des Kettengliedes i der unterbrechbaren Daisy-Chain nach Bild 5-43. Der 2-Bit-Code in den Zuständen gibt die Werte der Bits IPi und ISi an. Für das erste Kettenglied gilt $\overline{CH}_0 = 0$

scheidet die unterbrechbare Daisy-Chain die noch nicht akzeptierten Anforderungen (Zustand Z2) und die in Bearbeitung befindlichen Anforderungen (Zustände Z4 und Z5). Nur eine noch nicht akzeptierte Anforderung erzeugt ein Interruptsignal, und das auch nur dann, wenn sie höhere Priorität als das laufende Interruptprogramm hat ($\overline{\text{IRQ}_i} = \overline{\text{CH}_i} = 0$ im Zustand Z2).

Anmerkung. Bei der nichtunterbrechbaren Daisy-Chain nach Bild 5-41 ist das vorzeitige Rücksetzen des Maskenbits zu vermeiden, da bei ihr während $\overline{\text{IACK}} = 1$ jede Anforderung, egal welcher Priorität, an den Prozessor weitergeleitet wird und damit die Unterbrechbarkeit eines Interruptprogramms allein durch Anforderungen höherer Prioritäten nicht mehr gewährleistet ist.

Interrupt-Controller. Faßt man die einzelnen Prioritätenschaltungen, die Vektornummerregister und die für die Interruptverarbeitung erforderlichen Statusbits und Steuerbits der einzelnen Quellen in einem einzigen Baustein, einem Interrupt-Controller, zusammen, so erhält man ein Interruptsystem mit zentraler Priorisierung entsprechend Bild 5-45. Bei diesem System sind bis zu acht Interruptquellen an die Interrupteingänge $\overline{\text{IRQ}0}$ bis $\overline{\text{IRQ}7}$ des Interrupt-Controllers angeschlossen. Der Baustein (angelehnt an den Programmable Interrupt Controller PIC 8259A von Intel) weist diesen acht Eingängen unterschiedliche Prioritäten (Interruptebenen) zu. Drei Register und ein Prioritätenschaltnetz steuern den Priorisierungsvorgang. Das Interrupt-Pending-Register IPR speichert die an den Eingängen liegenden Anforderungen, sofern sie nicht durch das Interruptmaskenregister IMR blockiert werden. Bereits in Bearbeitung befindliche Anforderungen werden in dem Interrupt-Service-Register ISR gespeichert. Beide Register, IPR und ISR, wirken auf das Prioritätenschaltnetz PSN. Eine Eintragung in ISR blockiert sämtliche in IPR gespeicherten und noch nicht in Bearbeitung befindlichen

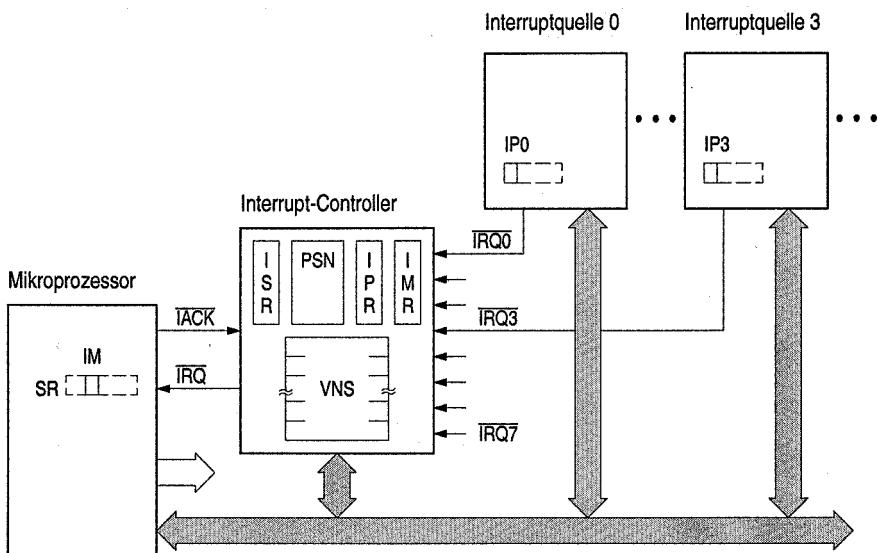


Bild 5-45. Interruptsystem mit Interrupt-Controller (zentrale Priorisierung)

Anforderungen gleicher und niedrigerer Prioritäten, wodurch die Unterbrechbarkeit durch Anforderungen höherer Prioritäten möglich wird. Die Verbindung zum Prozessor bilden die bereits bekannten Steuersignale $\overline{\text{IRQ}}$ und $\overline{\text{IACK}}$ sowie der Datenbus, über den der Prioritätenbaustein die Vektornummer aus dem Vektornummernspeicher VNS ausgibt.

Betrachtet man den Priorisierungsvorgang in einer der acht Ebenen, so ergibt sich für diese Ebene im Prinzip der gleiche Zustandsgraph wie bei der unterbrechbaren Daisy-Chain (Bild 5-44). Beim Interrupt-Controller ist gegenüber der Daisy-Chain lediglich das Verkettungssignal nach außen hin nicht sichtbar. Zusätzlich kann jedoch beim Interrupt-Controller die *Priorisierungsstrategie* durch ein in Bild 5-45 nicht gezeichnetes und in Bild 5-44 nicht berücksichtigtes Steuerregister programmiert werden. Im Standardbetrieb (geschachtelte Prioritäten) ist die Zuordnung der Prioritäten zu den Eingängen festgelegt. Bei rotierenden Prioritäten werden hingegen nach jedem akzeptierten Interrupt die Prioritäten zyklisch getauscht, wobei jeweils die zuletzt bediente Interruptquelle die niedrigste Priorität erhält (*rotating priorities*). Hierdurch entsteht, über einen längeren Zeitraum betrachtet, eine Art Gleichverteilung der Prioritäten auf die Interruptquellen. Bei programmierten rotierenden Prioritäten wird die jeweilige Ebene niedrigster Priorität durch die Programmierung festgelegt. Hiermit kann man z. B. einer bestimmten Interruptquelle für eine begrenzte Zeit eine bestimmte Priorität zuweisen. Über diese Priorisierungsmöglichkeiten hinaus kann durch ein Bit im Steuerregister des Interrupt-Controllers festgelegt werden, ob die Eingänge des Controllers auf Signalpegel oder Signalflanken reagieren.

Eine Erweiterung der Ebenenzahl ist dadurch möglich, daß den Interrupteingängen eines solchen Controllers weitere Interrupt-Controller vorgeschaltet werden, die vom zentralen Interrupt-Controller mittels eines 3-Bit-Busses über die Gewährung einer Interruptforderung informiert werden (Kaskadierung).

Die zentrale Priorisierung mit einem Interrupt-Controller hat gegenüber der dezentralen Priorisierung durch eine Daisy-Chain den Vorteil, daß die Interrupthardware in den Quellen entfällt und damit der Systemaufbau einfacher wird. Hinzu kommt die Flexibilität in der Prioritätenvergabe und die gegenüber einer Kette kürzere Priorisierungszeit. Nachteilig ist jedoch die auf dem Systembus erforderliche größere Leitungszahl für die Signale $\overline{\text{IRQ}_i}$.

5.5.3 Besondere Unterbrechungssignale

Neben den bisher beschriebenen Interrupteingängen für codierte oder uncodierte Anforderungen hat ein Mikroprozessor weitere Signaleingänge für die Programmunterbrechung, z. B. einen Bus-Error-Eingang $\overline{\text{BERR}}$ und einen Reset-Anschluß $\overline{\text{RESET}}$. Das Signal $\overline{\text{BERR}}$ wirkt als Trap, das Signal $\overline{\text{RESET}}$ als Interrupt. Beide Signale lösen eine Programmunterbrechung aus und aktivieren ein zu-

gehöriges Unterbrechungsprogramm; es findet jedoch kein Signalaustausch für eine prozessorexterne Interruptverarbeitung statt (vgl. 2.1.5). Zusammen mit einem vom Unterbrechungssystem unabhängigen Signal $\overline{\text{HALT}}$ dienen sie zur Systemsteuerung.

Bus-Error-Signal. Das Signal $\overline{\text{BERR}}$ wird dazu benutzt, Fehler der Prozessorumgebung, die von der Hardware erkannt werden, dem Prozessor mitzuteilen (*Busfehler*). Ausgelöst werden kann es z.B. durch einen Lesefehler beim Hauptspeicherzugriff (parity check), durch einen von der Speicherverwaltungseinheit gemeldeten unerlaubten Zugriffsversuch auf einen geschützten Speicherbereich oder durch das Ausbleiben des Quittungssignals bei einem Buszyklus (siehe auch 5.3.1). Eine über den $\overline{\text{BERR}}$ -Eingang veranlaßte Unterbrechungsanforderung hat aufgrund ihrer Bedeutung für die Systemsicherheit die zweithöchste Priorität des gesamten Trap- und Interruptsystems (siehe 2.1.5, Vektortabelle).

Reset-Signal. Das Signal $\overline{\text{RESET}}$ dient zur Initialisierung des gesamten Mikroprozessorsystems und hat deshalb die höchste Priorität aller Unterbrechungssignale. Es veranlaßt die Initialisierung der Prozessorhardware, wobei in unserem Prozessorbeispiel das Supervisor-Stackpointerregister aus der Vektortabelle und andere Register mit Null geladen werden. Es führt auf ein Initialisierungsprogramm, dessen Startadresse ebenfalls in der Vektortabelle steht. Dieses Programm initialisiert seinerseits sämtliche Funktionseinheiten, die der Systemsoftware unterliegen. Das $\overline{\text{RESET}}$ -Signal wird normalerweise über eine Reset-Taste manuell ausgelöst. Es kann zusätzlich mit einer Schaltung zur Überwachung der Netzspannung gekoppelt werden (*power-on reset*), so daß die Initialisierung automatisch mit dem Einschalten der Netzspannung erfolgt (*Reset-Logik*).

Der $\overline{\text{RESET}}$ -Anschluß des Mikroprozessors kann bidirektional ausgelegt sein, so daß der Prozessor auch ein RESET-Signal ausgeben kann (Befehl RESET). Damit ist es möglich, Systembausteine, die mit ihren $\overline{\text{RESET}}$ -Eingängen an die Reset-Leitung angeschlossen sind, per Software zu initialisieren.

Halt-Signal. Das Signal $\overline{\text{HALT}}$ versetzt den Mikroprozessor nach Abschluß des gegenwärtigen Buszyklus in einen Haltzustand. Hierbei sind alle Tristate-Ausgänge des Prozessors hochohmig, so daß der Prozessor vom Systembus abgekoppelt ist und dieser anderen Systemkomponenten mit Prozessorfunktion zur Verfügung steht. – Mit dem $\overline{\text{HALT}}$ -Signal lassen sich z.B. durch eine Zusatzlogik die Betriebsarten Halt, Run und Single-Step verwirklichen. Im Halt-Modus ruht die Verarbeitung im Prozessor; im Run-Modus führt er das laufende Programm aus. Beim *Single-step-Modus* wird der Prozessor zunächst in den Run-Modus gebracht. Sobald er jedoch einen Buszyklus begonnen hat, was er bei einem synchronen Bus durch $\overline{\text{CSTART}} = 0$ und bei einem asynchronen Bus durch $\overline{\text{AS}} = 0$ anzeigt, wird er in den Halt-Modus umgeschaltet. Auf diese Weise führt er genau einen Buszyklus durch. Dies erlaubt es dem Benutzer, Prozessoroperationen schrittweise auszuführen und damit das System zu testen.

5.6 Der PCI-Local-Bus und sein Nachfolger PCI-X

Im letzten Abschnitt dieses Kapitels soll nun – nachdem wir bisher auf sehr unterschiedliche Busse teils explizit teils implizit Bezug genommen haben – einer dieser Busse ausführlicher betrachtet werden, wobei die bisher beschriebenen Teilaspekte um weitere Merkmale ergänzt werden. Ausgewählt haben wir dazu den PCI-Local-Bus, kurz PCI-Bus (Peripheral Component Interconnect Bus), der bei PCs und Servern sowie allgemein bei Mikroprozessorsystemen eine weite Verbreitung gefunden hat, so auch bei industriellen Systemen. Die seit seiner Einführung im Jahre 1992 gestiegenen Leistungsanforderungen haben zu einem Nachfolger geführt, dem PCI-X-Bus, den wir in seinen wesentlichen Unterschieden mit in unsere Betrachtungen aufnehmen. Unsere Ausführungen beschränken sich auf die grundsätzliche Wirkungsweise dieser beiden Busse, d.h., viele Details bleiben unerwähnt. Diesbezüglich sei auf die Busspezifikationen verwiesen: beim PCI-Bus auf die Revision 2.2 bzw. 2.3, beim PCI-X-Bus auf die Revision 1.0. Unseren Ausführungen zum PCI-Bus liegt die Revision 2.1 zugrunde [PCI 1995]. Ihr gegenüber umfaßt die Revision 2.2 eine Reihe kleinerer Änderungen; die Revision 2.3 vollzieht den Übergang von 5 V zu 3,3 V Signalspannung. Eine sehr ausführliche, kommentierende Beschreibung zum PCI-Bus findet sich in [Shanley, Anderson 1995].

5.6.1 Motivation und technische Daten

Der Grund für die Entwicklung des PCI-Busses war die Einführung der grafikorientierten Betriebssysteme Windows und OS/2 für PCs. Mit diesen gingen höhere Anforderungen an die Übertragungsleistung zwischen Hauptspeicher und Grafikkarte einher, für die die maximal erreichbaren Übertragungsraten vieler anderer Busse einen Flaschenhals bildeten. Der PCI-Bus, der einerseits prozessorunabhängig ist, andererseits jedoch mit seiner Leistungsfähigkeit damals nahe an den Prozessorbus herankam, beseitigte diesen Flaschenhals zunächst. (Die Prozessornähe wird durch den Zusatz „Local“ im vollständigen Busnamen ausgedrückt.) Darüber hinaus sollte er die Verbindung zu anderen schnellen Komponenten herstellen, so z.B. zu Komponenten für die Bewegtbilddarstellung (motion video) und für die akustische Ein-/Ausgabe (audio), aber auch zu schnellen SCSI- und LAN-Controllern. Dazu sieht er die Möglichkeit von On-board-Komponenten und von Komponenten auf Steckkarten vor. Eine diesbezüglich typische Struktur zeigt Bild 5-46, die im wesentlichen der PC-Struktur in Bild 5-8 entspricht.

Die Anzahl der Komponenten am PCI-Bus ist durch die kapazitive Belastung des Busses begrenzt (siehe auch 5.1.5). Sollen darüber hinaus weitere Komponenten angeschlossen werden, so ist das System durch zusätzliche PCI-Busse, sog. Bussegmente, erweiterbar, wobei die Verbindung zwischen zwei PCI-Bussen durch

eine *PCI-to-PCI-Bridge* hergestellt wird. Üblich ist hier eine hierarchische Struktur an Busebenen.

Inzwischen sind die Anforderungen der Grafikkarten an Übertragungsleistung weiter gestiegen, so daß sie auch vom PCI-Bus nicht mehr erfüllt werden können. Aus diesem Grund ist bei leistungsfähigen PCs die zwischen Prozessorbus und PCI-Bus erforderliche Bridge (*Host-to-PCI-Bridge*) um einen sog. AGP-Port als Punkt-zu-Punkt-Verbindung zwischen Hauptspeicher und Grafikkarte erweitert. Gegenüber dem PCI-Bus ermöglicht er eine zwei-, vier- oder achtfache Übertragungsraten (5.1.5). Der PCI-Bus wird jedoch weiterhin für die anderen schnellen Systemkomponenten eingesetzt.

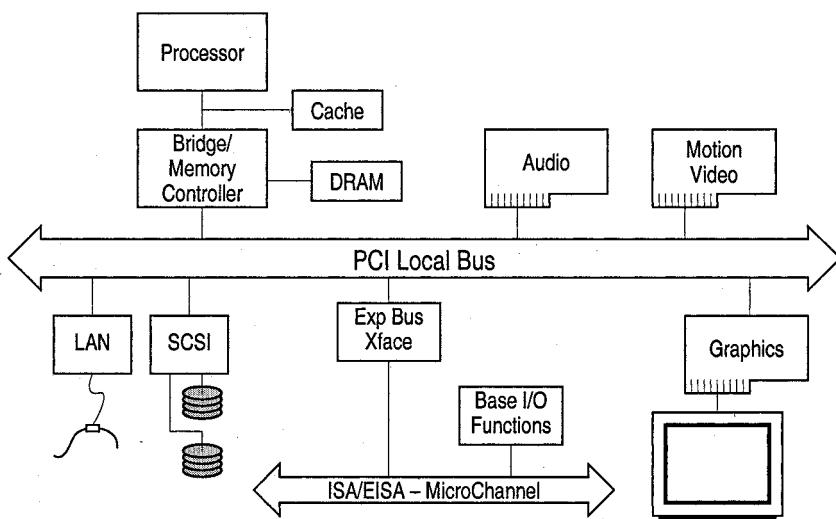


Bild 5-46. PCI-System in Blockdarstellung (nach [PCI 1995])

Die Vorteile, einen solchen Bus zu standardisieren und ihn allen Entwicklern von Mikroprozessorsystemen zugänglich zu machen, liegen vor allem darin, den Systementwurf zu vereinfachen, die Kosten zu reduzieren und ein großes Angebot an PCI-Komponenten, insbesondere in Form von Steckkarten zu haben, womit eine hohe Flexibilität bei der Systemkonfigurierung erreicht wird. Das Spektrum geht dabei von relativ einfachen Rechnerstrukturen bis hin zu hochleistungsfähigen Servern. Unterstützt werden kleine und ggf. mobile Rechnergemeinschaften durch die neben der üblichen Spannungsversorgung von 5 V möglichen 3,3 V-Versorgung der Buskomponenten. Die Flexibilität des Konfigurierens mittels Steckkarten wird durch Konfigurationsregister unterstützt, die den Komponenten zugeordnet sind. Diese können, mit teilweise fest vorgegebener Information, beim Einschalten des Systems in einer Selbstkonfigurierungsphase ausgewertet werden, und sie können während dieser Phase mit Konfigurationsinformation geladen werden. Man spricht hier von *Plug and Play*.

Die Prozessorunabhängigkeit des PCI-Busses erlaubt es, parallel zu Prozessor-Speicher-Zugriffen, die über den Prozessorbus abgewickelt werden, Transfers über den PCI-Bus zwischen den PCI-Komponenten durchzuführen. Darüber hinaus kann der Prozessor über die *Host-to-PCI-Bridge* (in Bild 5-46 als Bridge/Memory Controller bezeichnet) direkt auf die PCI-Komponenten zugreifen, deren Adressen beliebig auf seinen Speicher- oder Ein-/Ausgabeadressraum abbildbar sind. (Die Bridge setzt dabei die Prozessorbuszyklen in PCI-Buszyklen um, und umgekehrt.) Ebenso können PCI-Master direkt auf den Hauptspeicher zugreifen. Die Bridge unterstützt diese Prozessor- bzw. PCI-Master-Zugriffe ggf. durch Datenpufferregister und durch sog. *Posted-write-Buffers*. Letztere speichern Daten zwischen, die dann von der Bridge automatisch weitergereicht werden. Das entlastet den Prozessor bei seinen Schreibzugriffen, indem er nicht auf deren Abschluß warten muß. – Die Bridge kann zusätzlichen zu diesen Funktionen mit zentralen PCI-Funktionen ausgestattet sein, z.B. mit dem PCI-Busarbiter.

Die wichtigsten technischen Daten des PCI-Busses in Stichworten:

- 32-Bit-*Multiplexbus*, erweiterbar auf 64 Bit; unabhängig davon 32- oder 64-Bit-Adressierung. Vorwärts- und Rückwärtskompatibilität, d.h., es können Steckkarten mit jeweils wahlweise 32- und 64-Bit-breiten PCI-Komponenten betrieben werden.
- *Synchroner Bus* mit einer Bustaktfrequenz von üblicherweise 33 MHz, aber auch 66 MHz. Vorwärts- und Rückwärtskompatibilität, d.h., es können Steckkarten der jeweils anderen Frequenzauslegung betrieben werden.
- Transparenz in der Erhöhung der maximalen Übertragungsrate von 132 Mbyte/s (32-Bit-Datenweg bei 33 MHz) zu 264 Mbyte/s (64-Bit-Datenweg bei 33 MHz) und von 264 Mbyte/s (32-Bit-Datenweg bei 66 MHz) zu 528 Mbyte/s (64-Bit-Datenweg bei 66 MHz).
- Burst-Zyklen variabler Länge für Schreib- und Lesezugriffe.
- Kurze Latenzzeit von zwei Takten bei Schreibzugriffen (60 ns bei 33 MHz, 30 ns bei 66 MHz), wenn der Master auf dem Bus geparkt ist.
- *Multimasterfähigkeit* mit überlappender (verdeckter) Busarbitrierung.
- Bis zu vier Bussteckplätze bei 33 MHz und bis zu zwei bei 66 MHz mit der Möglichkeit, die Anzahl der Steckplätze durch weitere PCI-Busse zu erhöhen (PCI-to-PCI-Bridges).
- Möglichkeit der PCI-Anbindung anderer Busse mittels einer entsprechenden Bridge, z.B. *PCI-to-ISA*.
- PCI-Steckkarten mit 5 V oder 3,3 V Spannungsversorgung.

5.6.2 Bussignale

Zum besseren Verständnis der folgenden Ausführungen werden zunächst einige Begriffe eingeführt, wie sie in der PCI-Bus-Spezifikation verwendet werden: Die an den Bus angeschlossenen Einheiten werden als „Bus Components“, „Agents“ oder „Devices“ bezeichnet. Die Buskomponenten werden entweder direkt auf dem Motherboard montiert (planar-only devices) oder als Steckkarten mittels eines Steckverbinder auf das Motherboard gesteckt (add-in boards). Eine Buskomponente fungiert entweder als „Master“, auch „Initiator“ genannt, oder als „Target“ (Slave) oder hat beide Funktionen. Master können „Bus Transactions“ initiieren, Targets nicht. Eine Bustransaktion besteht immer aus einer „Address Phase“, gefolgt von einer oder mehreren „Data Phases“. Innerhalb einer Adress- oder Datenphase wird genau eine Informationseinheit über den Bus transportiert, wofür ein oder mehrere „Cycles“ (Bustakte) benötigt werden. Der sog. „Turnaround Cycle“ ist ein Leerzyklus, der dann eingeschoben wird, wenn ein Agent aufhört, ein Signal zu „treiben“ und anschließend ein anderer Agent das Signal treiben möchte (Vermeidung von Signalkonflikten). 0-aktive Signale werden durch das dem Signallamen nachgestellte #-Zeichen gekennzeichnet.

Bild 5-47 zeigt die Signale des PCI-Busses nach Signalgruppen geordnet. Sie bestehen aus einem Minimalsatz (links im Bild) und einem Satz optionaler Signale

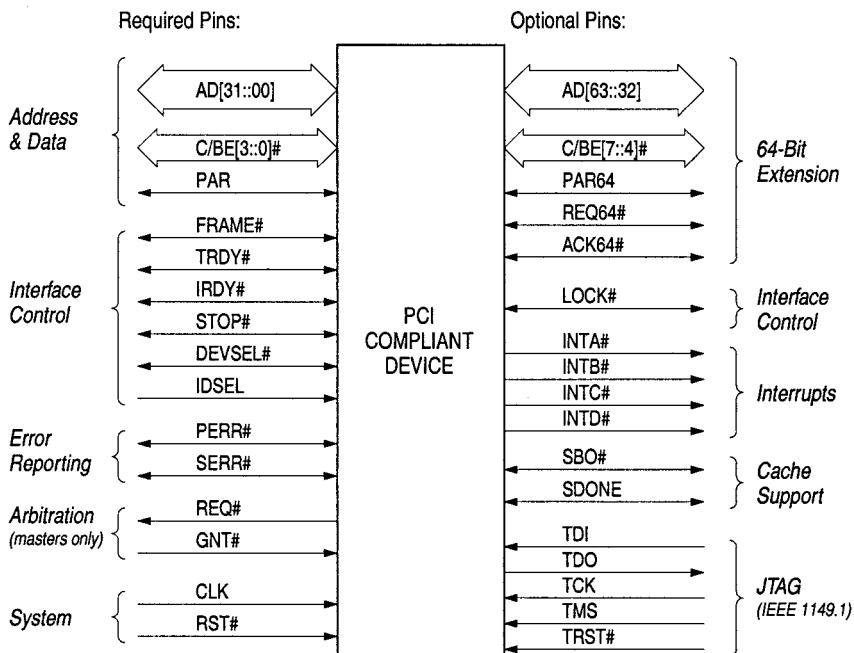


Bild 5-47. PCI-Signalanschlüsse (nach [PCI 1995])

(rechts im Bild) für den Aufbau unterschiedlich komplexer Systeme. Im Minimum benötigt ein PCI-Master als Steckkarte 49 Signale, als Planar-only-Komponente 47 Signale (die Error-Reporting-Signale sind hierbei optional). PCI-Targets benötigen jeweils zwei Signale weniger, da bei ihnen die beiden Signale für die Busarbitration entfallen. Die im Bild für viele der Signale angegebene Bidirektionalität ist darauf zurückzuführen, daß dieses Bild die Schnittstelle einer PCI-Komponente mit sowohl Master- als auch Target-Funktion zeigt. Aus elektrotechnischer Sicht werden folgende Signalarten unterschieden:

- in *Input* als Standard-Eingangssignal.
- out *Totem Pole Output* als Standard-Bustreiber-Ausgangssignal (für höhere Ströme).
- t/s *Tristate* als bidirektionales Ein- und Ausgangssignal mit Tristate-Funktion.
- s/t/s *Sustained Tristate* als 0-aktives Ein- und Ausgangssignal mit Sustained Tristate-Funktion (siehe 5.1.4), das zu einem Zeitpunkt immer nur von einem Device benutzt wird. Beendet ein Device das „Treiben“ des Signals, so schließt sich ein Turnaround-Zyklus an.
- o/d *Open Drain* als von mehreren Devices gleichzeitig nutzbares Signal mit Wired-or-Funktion (entspricht Open-Collector, siehe 5.1.4).

Im folgenden sind die Signale mit Kurzbeschreibungen ihrer Funktionen aufgelistet. Ergänzt werden diese Angaben in 5.6.3 im Zusammenhang mit verschiedenen Busoperationen.

Signale des Minimalsatzes:

- AD[31::00] *Address and Data* (t/s) sind Multiplexsignale zur Übertragung von Adressen und Daten im Rahmen von Bustransaktionen.
- C/BE[3::0]# *Bus Command and Byte Enables* (t/s) sind Multiplexsignale, die während der Adreßphase ein Buskommando vorgeben und während der Datenphase als Byte-Enable-Signale fungieren.
- PAR *Parity* (t/s) bildet die gerade Parität über die Bussignale AD[31::00] und C/BE[3::0]# für die Adreßphase und die Datenphase(n).
- FRAME# *Cycle Frame* (s/t/s) wird vom aktuellen Master aktiviert, um den Beginn und die Dauer einer Bustransaktion anzuzeigen.
- IRDY# *Initiator Ready* (s/t/s) zeigt die Bereitschaft des aktuellen Busmasters an, die aktuelle Datenphase abzuschließen.
- TRDY# *Target Ready* (s/t/s) zeigt die Bereitschaft des angewählten Target an, die aktuelle Datenphase abzuschließen.
- STOP# *Stop* (s/t/s) zeigt dem aktuellen Busmaster an, daß das Target die laufende Bustransaktion abbrechen möchte.

- DEVSEL#** *Device Select* (s/t/s) wird von dem Target aktiviert, das die aktuelle Adresse als die seine erkennt. Dem Master wird damit signalisiert, daß ein Target ausgewählt wurde.
- IDSEL** *Initialization Device Select* (in) wird als Chip-Select-Signal während der Konfigurierungszugriffe (Lesen und Schreiben) benutzt, da die Konfigurationsregister nicht „memory-mapped“ angesprochen werden können.
- REQ#** *Request* (t/s) zeigt dem Busarbiter an, daß der zugehörige Agent (Master) den Bus benutzen möchte. Das Signal ist als Punkt-zu-Punkt-Verbindung dem Master individuell zugeordnet, d.h., jeder Master hat sein eigenes Request-Signal.
- GNT#** *Grant* (t/s) zeigt dem busanfordernden Agenten an, daß ihm der Bus zugeteilt wurde. Das Signal ist als Punkt-zu-Punkt-Verbindung dem Master individuell zugeordnet, d.h., jeder Master hat sein eigenes Grant-Signal.
- PERR#** *Parity Error* (s/t/s) zeigt Daten-Paritätsfehler bei allen Bustransaktionen, (außer beim sog. Special-Cycle) an. Das Signal wird vom datenempfangenden Agenten ausgegeben.
- SERR#** *System Error* (o/d) zeigt Systemfehler mit „katastrophaler“ Auswirkung an.
- RST#** *Restart* (in) bringt PCI-spezifische Register und Signale in einen definierten Initialzustand.
- CLK** *Clock* (in) ist der für alle PCI-Komponenten gemeinsame Bustakt. Lediglich die Signale RST#, INTA#, INTB#, INTC# und INTD# werden asynchron zu diesem Takt erzeugt.

Optionale Signale:

- AD[63::32]** *Address and Data* (t/s) sind Multiplexsignale für die um 32 Bit erweiterte Adreß- und Datenübertragung. Während der Adreßphase werden über sie die 32 höherwertigen Bits einer 64-Bit-Adresse transportiert, sofern als Buskommando ein Dual-Address-Cycle, DAC, vorliegt (C/BE[3::0]#) und sofern das Signal REQ64# aktiviert ist. Während der Datenphase werden über sie zusätzlich 32 Bits an Daten übertragen, sofern die beiden Signale REQ64# und ACK64# aktiviert sind.
- C/BE[7::4]#** *Bus Command and Byte Enables* (t/s) sind Multiplexsignale. Während der Adreßphase dienen sie zur Übertragung des aktuellen Buskommandos, sofern als Buskommando ein Dual-Address-Cycle vorliegt (C/BE[3::0]#) und sofern das Signal REQ64# aktiviert ist. Während der Datenphase dienen sie als Byte-Enable-Signale für die

zusätzlichen 32 Datenbits, sofern die beiden Signale REQ64# und ACK64# aktiviert sind.

REQ64#	<i>Request 64-bit Transfer</i> (s/t/s) wird vom aktuellen Busmaster aktiviert und zeigt an, daß dieser einen 64-Bit-Transfer durchführen möchte.
ACK64#	<i>Acknowledge 64-bit Transfer</i> (s/t/s) wird vom adressierten Target aktiviert, wenn dieses bereit ist, 64-Bit-Datentransfers durchzuführen.
PAR64	<i>Parity Upper DWord</i> (t/s) bildet die gerade Parität über die Bussignale AD[63::32] und C/BE[7::4]# für die Adreßphase und die Datenphase(n).
LOCK#	<i>Lock</i> (s/t/s) zeigt eine sog. atomare Operation eines Masters an, die aus mehreren zusammenhängenden Transaktionen besteht, z.B. eine Semaphoroperation (<i>read-modify-write</i>). Dieses Signal kann dazu genutzt werden, den Busarbiter solange an einer weiteren Buszuteilung zu hindern, bis der aktuelle Master das Lock-Signal wieder deaktiviert (bus lock). Es kann aber auch dazu genutzt werden, nur eine spezielle Speichereinheit für die Zeit der Durchführung der atomaren Operation zu reservieren (resource lock). Da es nur ein einziges Lock-Signal für alle Master gibt, muß ein Master vor Benutzung dieses Signals anhand bestimmter Bedingungen prüfen, ob es frei ist.
INTA-D#	<i>Interrupt A, B, C und D</i> (o/d) sind pegelsensitive Unterbrechungssignale. Alle vier Signale sind für sog. Multi-Funktions-Agenten einsetzbar (das sind PCI-Komponenten mit mehr als einem eigenständigen PCI-Device mit jeweils eigenem Konfigurationsadreßraum); bei sog. Single-Devices ist immer das Interruptsignal A zu verwenden. Auf dieser Basis ist die Zuteilung und gemeinsame Nutzung (interrupt sharing) von Interruptleitungen frei wählbar; dementsprechend kann ein Device-Treiber keine Annahmen über das Interrupt-Sharing machen und muß in der Lage sein, Interrupts mit anderen logischen Komponenten zu teilen.
SBO#	<i>Snoop Backoff</i> (in/out) zeigt einen Cache-Hit bzgl. einer modifizierten Cache-Zeile an (siehe dazu auch 6.2.4, MESI-Protokoll).
SDONE	<i>Snoop Done</i> (in/out) zeigt im inaktivierten Zustand an, daß ein Snoop-Vorgang noch andauert. Bei aktiviertem Signal ist der Snoop-Vorgang abgeschlossen.

Die folgenden JTAG/Boundary-Scan-Signale sind ebenfalls optional. Sie folgen dem IEEE-Standard 1149.1 (Test Access Port and Boundary Scan Architecture) und werden für den Test hochintegrierter elektronischer Schaltkreise verwendet.

Das zu testende Device muß dazu einen Test-Access-Port (*TAP*) mit diesen Signalen aufweisen.

TCK	<i>Test Clock</i> (in) taktet die bitserielle Ein- und Ausgabe von Statusinformation und von Daten während einer TAP-Operation.
TDI	<i>Test Data Input</i> (in) wird zu bitseriellen Eingabe von Test-Daten und von Test-Anweisungen während einer TAP-Operation benutzt.
TDO	<i>Test Data Output</i> (out) wird zu bitseriellen Ausgabe von Test-Daten während einer TAP-Operation benutzt.
TMS	<i>Test Mode Select</i> (in) steuert die TAP-Steuereinheit im Device.
TRST#	<i>Test Reset</i> (in) erlaubt (als Option) das asynchrone Initialisieren der TAP-Steuereinheit.

5.6.3 Busoperationen

Buskommandos. Buskommandos geben dem Target die vom Master gewünschte Bustransaktion an. Sie werden mit den Signalen C/BE[3::0]# (bei der 64-Bit-Erweiterung ggf. zusätzlich mittels C/BE[7::4]#) codiert und während der Adreßphase übertragen. Insgesamt gibt es zwölf Kommandos, vier der 16 möglichen Codierungen sind für spätere Erweiterungen reserviert. Generell gilt: Wird eine in der PCI-Bus-Spezifikation ausgewiesene Reservierung in der Systemimplementierung benutzt, so gilt das entstandene Produkt als nicht-PCI-entsprechend (not PCI-compliant). Sämtliche Kommandos bewirken Lese- oder Schreibzugriffe mit expliziter oder impliziter Adressierung. Explizit adressiert werden der Memory- und der I/O-Adreßraum (*isolierte Adressierung*, siehe auch 5.2.1), implizit der Konfigurationsadreßraum und ein ggf. vorhandener Interrupt-Controller. Hinzu kommt die *implizite Adressierung* aller Targets gleichzeitig zur Übermittlung von Nachrichten.

Allgemeine Zugriffe auf den Memory-Adreßraum erfolgen durch die beiden Lese- bzw. Schreibkommandos *Memory-Read* (0110) und *Memory-Write* (0111). Sie ermöglichen Einzel- und Blocktransporte und setzen voraus, daß das adressierte Target die Datenkohärenz bzgl. eventueller von ihm verwalteten Pufferregister bei einer solchen Transaktion sicherstellt. In Erweiterung des Lesezugriffs gibt es das *Memory-Read-Line-* (1110) und das *Memory-Read-Multiple*-Kommando (1100). Ersteres bezieht sich auf das Lesen eines *Cache-Blocks*, bewirkt also eine Bustransaktion mit insgesamt vier Datenphasen. Letzteres wird für das Lesen von mehr als nur einem Cache-Block benutzt, wozu die Speichersteuerung aufgefordert wird, über den ersten Cache-Block hinaus weitere Speicherzugriffe im Vorgriff durchzuführen, und zwar so lange, wie FRAME# vom Master aktiv gehalten wird. Beide Kommandos setzen einen entsprechend großen Puffer für die im Vorgriff gelesenen Speicherinhalte voraus (für den Hauptspeicher z.B. in

der Bridge). Ein fünftes Kommando, *Memory-Write-and-Invalidate* (1111) überträgt eine oder auch mehrere Cache-Zeilen in den Speicher. Die jeweils betroffene Cache-Zeile wird dabei invalidiert (Zurückschreiben/kopieren bei Copy-Back-Caches, siehe auch 6.2.4). Jedes dieser fünf Kommandos beginnt die Bustransaktion mit einer Adreßphase, d.h. mit expliziter Adressierung.

Zugriffe auf Targets, die im I/O-Adreßraum liegen, erfolgen mittels der Kommandos *I/O-Read* (0010) und *I/O-Write* (0011) mit ebenfalls expliziter Adressierung. (In der PCI-Spezifikation wird für PCs empfohlen, die Register von I/O-Devices in den Memory-Adreßraum zu legen, da der I/O-Adreßraum begrenzt und stark fragmentiert ist.) Mit den Kommandos *Configuration-Read* (1010) und *Configuration-Write* (1011) werden Lese- bzw. Schreibzugriffe auf den Konfigurationsadreßraum mit quasi-impliziter Adressierung ausgeführt (siehe dazu später: Konfigurierungszyklus). Dieser Adreßraum besteht für jedes Device aus insgesamt 256 Bytes, die zur Systeminitialisierung und -konfigurierung benutzt werden. Der Zugriff auf einen Interrupt-Controller erfolgt mit dem *Interrupt-Acknowledge*-Kommando (0000). Es bewirkt einen Vektornummer-Lesezyklus, wobei die Byte-Enable-Signale dem Datenformat des Vektors entsprechen müssen. Die gleichzeitig mit dem Kommando vorliegenden Adreßsignale AD[31::00] haben aufgrund impliziter Adressierung durch das Kommando selbst keine Bedeutung.

Ein weiteres Kommando mit impliziter Adressierung, *Special-Cycle* (0001), erlaubt das gleichzeitige Übertragen von Nachrichten an alle am Bus befindlichen Agenten (*broadcasting*), wobei jedoch PCI-to-PCI-Bridges nicht überschritten werden. Jeder Agent muß dabei selbst entscheiden, ob er von einer Nachricht betroffen ist oder nicht. In der Adreßphase sind die Adreßsignale AD irrelevant (in den C/BE#-Signalen ist das Kommando codiert). In der Datenphase enthalten die Bits AD[31::00] den Nachrichtentyp und ggf. ein Datenfeld, das dann in AD[31::16] codiert ist. Ein letztes Kommando, das *Dual-Address-Cycle*-Kommando (DAC, 1101), erlaubt die 64-Bit-Adressierung von Targets bei Bustransaktionen mit dem Memory-Adreßraum. Ausgehend von dem Standardmodus einer 32-Bit-Verbindung werden dazu während der Adreßphase zwei aufeinanderfolgende 32-Bit-Adreßübertragungen durchgeführt, wobei zunächst die untere Adreßhälfte (lo-address), zusammen mit dem DAC-Kommando, und dann die obere Adreßhälfte (hi-address), zusammen mit dem die nachfolgende Datenphase definierenden Kommando, übermittelt werden. Zur Adreßübertragung bei auf 64 Bit erweitertem Bus siehe weiter unten. – Ein Übersicht über alle Kommandos mit Gegenüberstellung der PCI-X-Kommandos findet sich in Tabelle 5-5 (S. 368).

Adressierung. Grundsätzlich wird die Decodierung von Adressen von den einzelnen PCI-Devices selbst durchgeführt, es gibt also keinen zentralen Adreßdecoder. Der oder die Adreßbereiche, die ein Device im Memory- oder I/O-Adreßraum belegt, werden durch eines bzw. mehrere Basisadreßregister definiert, die im Konfigurationsadreßraum des Device liegen. Adressiert werden Daten in den Formaten Byte, Wort (WORD, 16 Bit) und Doppelwort (DWORD, 32 Bit). Bei

der 64-Bit-Erweiterung gibt es zusätzlich das Format Quadword (double DWORD, 64 Bit). Die Adreßzählung erfolgt mit *Little-endian-Byteanordnung*. Bei Zugriffen auf den I/O-Adreßraum wird mittels AD[31:00] eine Byteadresse übertragen, wobei jedoch die Bits AD[1::0] nur dazu benutzt werden, das DEVSEL#-Signal zu erzeugen. Welches oder welche Bytes innerhalb der verbleibenden DWORD-Adresse vom Zugriff betroffen sind, zeigen die Byte-Enable-Signale während der Datenphase an. Zugriffe auf den Memory-Adreßraum erfolgen hingegen grundsätzlich mit DWORD-Adressen. Mit den Bits AD[1::0] werden dabei zwei Fälle für die Adreßfortschaltung unterschieden, das „Linear Incrementing“ und der „Cacheline Wrap mode“. Erstes ist die Standardmethode, die Adresse bei Burst-Zugriffen weiterzuzählen, nämlich um vier Bytes (32-Bit-Transfers) oder um 8 Bytes (64-Bit-Transfers), und zwar nach jeder Datenphase. Letzteres erlaubt es, den Burst-Zugriff bei einer beliebigen Datenadresse innerhalb eines Cache-Blocks beginnen zu lassen (siehe auch 5.3.3). – Zur Adressierung des Konfigurationsadreßraums und zur Konfigurierung selbst siehe unten.

Der PCI-Bus sieht kein Vertauschen von Bytes in ihren Buspositionen und kein Dynamic-Bus-Sizing vor. Vielmehr wird davon ausgegangen, daß alle PCI-Devices mit 32 Adreß-/Datensignalen an den Bus angeschlossen sind, da sie ja 32-Bit-Adressen decodieren müssen. Eine Ausnahme macht die 64-Bit-Erweiterung. Hier teilt ein Master, der die 64-Bit-Erweiterung unterstützt, einen 64-Bit-Transfer ggf. in zwei 32-Bit-Transfers auf, nämlich dann, wenn er mit einem 32-Bit-Device kommuniziert.

Read- und Write-Transaktionen. Die eigentlichen Bustransaktionen sind die Read- und die Write-Transaktionen für das Lesen und Schreiben von Daten. Sie sind grundsätzlich als Burst-Zugriffe ausgelegt, erlauben also das Übertragen mehrerer aufeinanderfolgender Daten (burst transaction), aber auch eines einzelnen Datums (single transaction). Synchronisiert werden sie in zwei Ebenen. In der ersten Ebene wirken das FRAME#-Signal des Masters, mit dem dieser eine Transaktion einleitet und beendet, und das DEVSEL#-Signal des Target, mit dem dieses nach Decodieren der Adresse seine Anwahl bestätigt. Beide Signale kennzeichnen durch ihren Aktivzustand das Andauern einer Transaktion. In der zweiten Ebene wirken das IRDY#-Signal des Masters und das TRDY#-Signal des Target, mit denen jeder von ihnen seine Bereitschaft für jede der Datenphasen signalisiert. Ist eines der beiden Signale während einer Datenphase inaktiv, so führt dies zu einem oder mehreren Wartezyklen, je nachdem, wie lange der Inaktivzustand anhält.

Lesetransaktion. Bild 5-48 zeigt das grundsätzliche Zeitverhalten der Signale einer Lesetransaktion (basic read transaction), bestehend aus einer Adreßphase und insgesamt drei Datenphasen. Eingeleitet wird sie vom Master durch Aktivieren von FRAME#. Im ersten Takt, der Adreßphase, erfolgt die Übertragung der Adresse (AD) und des Buskommandos (C/BE#). Die Signale IRDY#, TRDY# und DEVSEL# befinden sich dabei in einer Umschaltphase, da sie zuvor ggf. von

einem anderen Master getrieben wurden. Dargestellt ist dies durch die gegenläufigen Pfeile. Eine solche Umschaltphase ist auch im zweiten Takt erforderlich, um die für die Adressausgabe benutzten Multiplexleitungen AD auf die Dateneingabe umzuschalten (*Turnaround-Zyklus*), wodurch sich die erste Datenphase gegenüber einer Schreibtransaktion um einen Takt verzögert. Zu diesem Zeitpunkt werden jedoch bereits die Byte-Enable-Signale generiert (C/BE#), die (entgegen der Darstellung im Bild, da bei Lesezugriffen unüblich) mit jeder Datenphase auch verändert werden können. Das adressierte Target meldet sich, indem es DEVSEL# aktiviert.

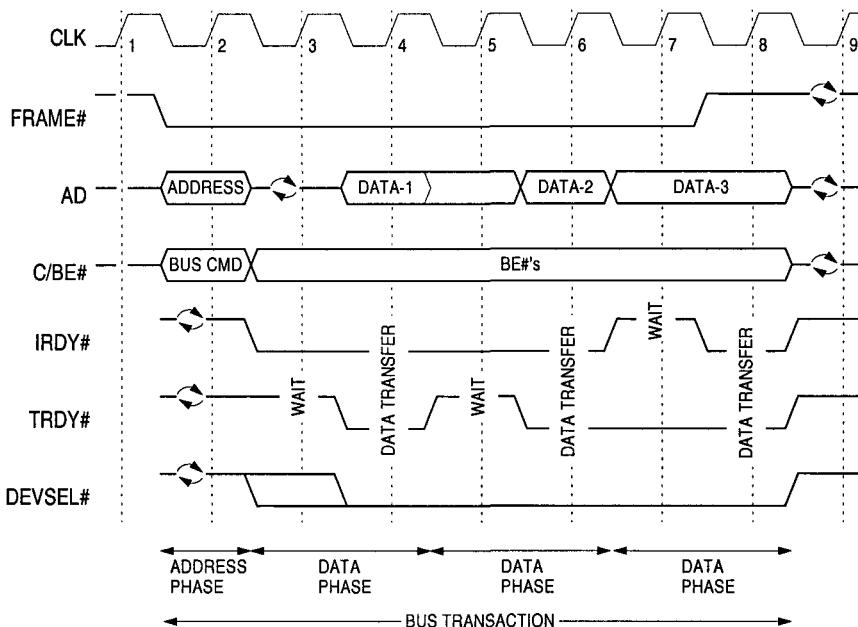


Bild 5-48. Grundsätzliches Signalverhalten bei einer Lesetransaktion (nach [PCI 1995])

Die Bereitschaft für eine Datenübertragung wird, wie bereits gesagt, vom Master mit IRDY# und vom Target mit TRDY# angezeigt. Ist eines der beiden Signale zum Datenübernahmezeitpunkt inaktiv, so wird ein Wartezyklus eingefügt, wie in Bild 5-48 für die Datenübertragungen 2 und 3 gezeigt. Das Ende der Bustransaktion zeigt der Master dadurch an, daß er sein FRAME#-Signal für die letzte Übertragung deaktiviert. Im Normalfall wäre das, auf das Bild bezogen, bereits im Takt 7 der Fall. Da er selbst aber erst im Takt 8 für die letzte Datenübertragung bereit ist (siehe IRDY#), deaktiviert er FRAME# erst für diesen Takt. Nach dieser letzten Datenübertragung deaktiviert auch das Target sein DEVSEL#-Signal. Vor einer weiteren Bustransaktion durchlaufen die Signale FRAME#, AD und C/BE# eine Umschaltphase. Mit inaktiviertem FRAME# und IRDY# befindet sich der Bus im Ruhezustand (idle state).

Schreibtransaktion. Bild 5-49 zeigt als Gegenstück zu Bild 5-48 das grundsätzliche Zeitverhalten der Signale für eine Schreibtransaktion (basic write transaction). Die Synchronisation zwischen Master und Target ist hier im Prinzip dieselbe wie bei der Lesetransaktion. Es entfällt lediglich die Umschaltpulse für die Multiplexleitungen AD, da diese in der Adress- als auch in den Datenphasen einheitlich in Ausgaberrichtung betrieben werden. Das Bild zeigt außerdem die Möglichkeit der Byte-Enable-Signale (C/BE#), sich von Datenphase zu Datenphase zu ändern.

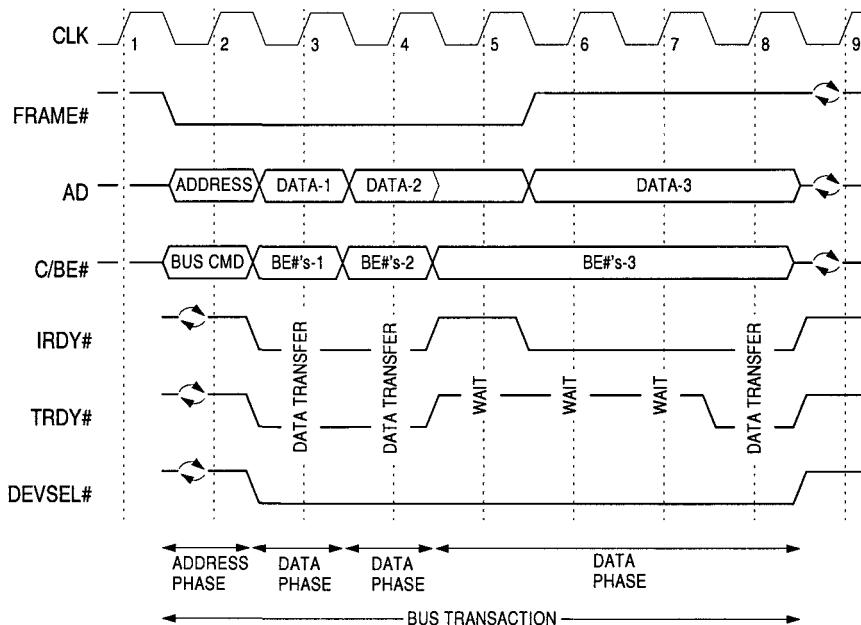


Bild 5-49. Grundsätzliches Signalverhalten bei einer Schreibtransaktion (nach [PCI 1995])

Verzögerte Transaktion (delayed transaction). Um den Bus nicht unnötig zu belegen, kann ein Target, das nicht sofort für eine vom Initiator angestoßene Transaktion bereit ist, diese abbrechen, was es dem Initiator durch eine Retry-Signalisierung mitteilt (siehe nachfolgend). Das Target übernimmt dabei jedoch die in der Adressphase übermittelte Information und bereitet die Transaktion für sich vor. Der Initiator, der nicht weiß, wann das Target letztlich dafür bereit sein wird, stößt die Transaktion immer wieder neu an, bis das Target ohne Retry reagiert und diese Transaktion mit durchführt. Für seine Wiederholungen muß der Initiator jedesmal die Busarbitration neu durchlaufen, d.h., daß der Bus in den Zwischenzeiten von anderen Mastern arbitriert und genutzt werden kann. Solche verzögerte Transaktionen sind bei allen Kommandos außer bei den beiden Memory-Write-Kommandos möglich.

Abbruchbedingungen. Der Abschluß einer Lese- oder Schreibtransaktion mittels FRAME# kann zwei Ursachen haben: (1.) Es sind alle vom Master beabsichtigten Datenübertragungen erfolgt (termination by completion). (2.) Dem Master wird vor Abschluß der Transaktion das Buszuteilungssignal GNT# entzogen, und die für die Übertragung vorgesehene Zeit ist abgelaufen (termination by timeout). Diese Zeit wird im Master durch einen sog. Latency-Timer vorgegeben. Sie kann dadurch überschritten werden, daß z.B. das Target zu viele Wartezyklen bewirkt, oder daß die beabsichtigte Transaktion sehr lang ist. Bei einer Memory-Write-and-Invalidate-Transaktion wird dieser Abbruch allerdings erst bei Erreichen der nächsten Cache-Block-Grenze vorgenommen. Zusätzlich zu diesen „normalen“ Abbrüchen gibt es auch einen „anormalen“ Abbruch einer Transaktion durch den Master, nämlich dann, wenn das DEVSEL#-Signal für eine bestimmte Taktanzahl ausbleibt (termination by master-abort).

Auch das Target kann den Abbruch einer Transaktion initiieren, wofür es drei unterschiedliche Anlässe gibt, die durch Aktivieren des STOP#-Signals signalisiert und in Verbindung mit jeweils einem zweiten Signal gekennzeichnet werden. Diese Abbrüche werden mit „Retry“, „Disconnect“ und „Target-Abort“ bezeichnet. Beim Retry erfolgt der Abbruch bereits in der ersten Datenphase, ohne daß ein Datum übertragen wird (STOP# aktiv, TRDY# inaktiv). Der Grund dafür kann z.B. sein, daß das Target nicht sofort für die Transaktion bereit ist und deshalb eine verzögerte Transaktion einleitet, oder daß der Zugriff aufgrund eines „Snoop-Hit on Modified Line“ abgebrochen werden muß (siehe später: Cache-Unterstützung). Beim Disconnect erfolgt der Abbruch in irgendeiner der Datenphasen, wobei das Datum noch übertragen wird (STOP# aktiv, TRDY# aktiv). Gründe dafür sind, daß im Target während der Transaktion ein Ressourcen-Konflikt auftritt oder daß durch Überschreiten einer Ressourcen-Grenze eine Zeitbedingung verletzt wird. Der Target-Abort schließlich ist ein „anormaler“ Abbruch, der dann ausgelöst wird, wenn das Target einen fatalen Fehler erkennt oder wenn es nicht in der Lage ist, die gewünschte Transaktion jemals korrekt zu beenden (das zunächst aktivierte DEVSEL# wird inaktiviert und gleichzeitig STOP# aktiviert).

Busarbitration. Die Zuteilung des Busses bei mehreren Mastern erfolgt durch einen zentralen Busarbiter mittels je einer Anforderungsleitung REQ# und einer Gewährungsleitung GNT# für jeden der Master. Für die Zuteilung wird in der PCI-Spezifikation kein bestimmter Algorithmus vorgegeben, es wird lediglich eine „faire“ Buszuteilung vorausgesetzt, um Blockierungen zu vermeiden. Fair heißt, daß jeder Master unabhängig von anderen Anforderern den Bus zugeteilt bekommt, es heißt aber nicht, daß alle Master dabei dieselbe Priorität haben. In der Spezifikation wird dazu eine Empfehlung für einen Algorithmus gegeben, der eine „faire“ Zuteilung für zu Gruppen zusammengefaßte Master mit unterschiedlichen Prioritäten der Gruppen beschreibt. Im Rahmen der Fairness ist auch ein Bus- oder Resource-Lock realisierbar (siehe oben: Beschreibung des LOCK#-Signals).

Bild 5-50 zeigt das grundsätzliche Signalverhalten bei der Busarbitration (basic arbitration). Gezeigt sind die Busanforderungen zweier Agenten A und B. A ist zunächst alleiniger Anforderer (REQ#-a aktiviert). Zum Zeitpunkt der Buszuteilung an ihn (GNT#-a aktiviert) stellt B eine Anforderung mit höherer Priorität (REQ#-b aktiviert). Diese bewirkt, daß B nach Abschluß der Bustransaktion von A, die nur eine einzige Datenphase umfaßt, die Buszuteilung erhält und dies, obwohl A sein Anforderungssignal weiterhin aktiv hält, da er unmittelbar an die erste Bustransaktion eine zweite anschließen möchte. Der Arbiter entzieht dazu A das Gewährungssignal (GNT#-a), und zwar einen Takt nachdem A das FRAME#-Signal aktiviert hat, und aktiviert bereits zu diesem Zeitpunkt, d.h. überlappend mit der Busbenutzung von A, das Gewährungssignal für B (GNT#-b). B übernimmt den Bus aber erst, nachdem A FRAME# und IRDY# inaktiviert hat, verzögert um den für die Umschaltphase der Signale erforderlichen Takt. B inaktiviert in diesem Beispiel sein Anforderungssignal (REQ#-b) bereits mit dem Aktivieren von FRAME#, womit er dem Arbiter anzeigen, daß er nur eine einzige Datenphase durchführen möchte.

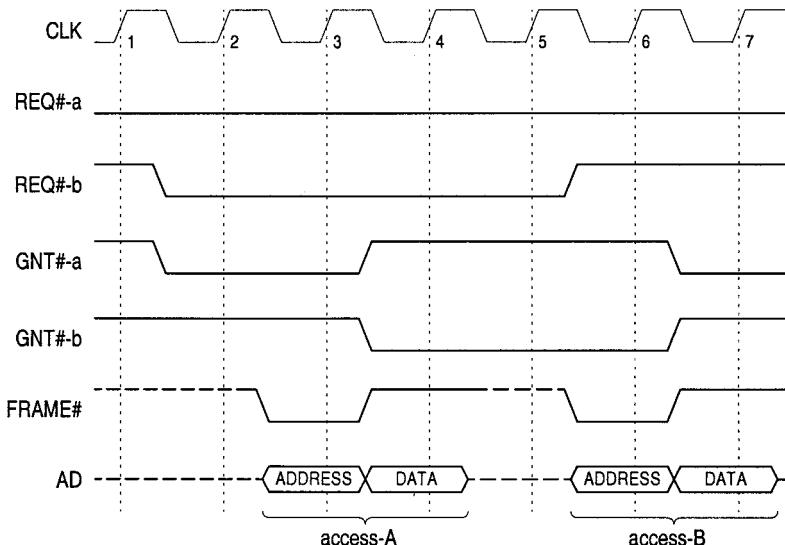


Bild 5-50. Grundsätzliches Signalverhalten bei der Busarbitration (nach [PCI 1995]). Agent A, der zwei aufeinanderfolgende Bustransaktionen durchführen möchte (wzu er sein Anforderungssignal REQ#-a auch noch nach der ersten Buszuteilung aktiv hält) muß den Bus nach der ersten Transaktion abgeben, da zwischenzeitlich eine Anforderung des höherpriorisierten Agenten B eingegangen ist

Für die Implementierung eines Arbiters wird in der PCI-Spezifikation die Möglichkeit des „Bus-Parking“ empfohlen. Hierbei soll, wenn keine Busanforderung vorliegt, das Gewährungssignal für den aktuellen Master so lange aktiv gehalten werden, bis der Bus im Ruhezustand ist (FRAME# und IRDY# inaktiv). Das er-

spart die Zeit für eine erneute Buszuteilung an denselben Master, wenn dieser eine neue Anforderung noch während seiner Busbenutzung stellt. Darüber hinaus mag der Bus, wenn er im Ruhezustand ist, einem bestimmten Master *i* vorsorglich zugeteilt werden (GNT#-*i* aktiviert), so daß dieser Master bei einer Anforderung nicht den Zuteilungszyklus durchlaufen muß. Eine weitere Optimierung der Busbenutzung erreicht man durch „fast Back-to-Back-Transactions“. Dabei kann ein Master mehrere Bustransaktionen unmittelbar aufeinanderfolgen lassen, ohne daß dabei der Bus zwischenzeitlich den Ruhezustand einnimmt. Diese Bustransaktionen können sich auf nur ein Target oder aber nacheinander auch auf mehrere Targets beziehen. Für die Durchführung dieser Bustransaktionen muß jedoch eine Reihe von Vereinbarungen und Bedingungen erfüllt sein, so muß z.B. das jeweilige Target das Aktivieren des FRAME#-Signals erkennen können, auch wenn sich der Bus zuvor nicht im Ruhezustand befand.

Latency. Der PCI-Bus erhebt den Anspruch eines „Low-Latency“-Ein-/Ausgabebusses mit hohem Durchsatz. Dementsprechend gibt es eine Reihe an möglichen Überwachungsmaßnahmen, um bestimmte Wartezeiten nicht zu überschreiten. Im Prinzip werden damit sowohl die Master als auch die Targets in der Anzahl ihrer Wartezyklen eingeschränkt, die sie während einer Bustransaktion erzeugen dürfen. Darüber hinaus haben die Master einen programmierbaren „Latency-Timer“, der die Dauer einer Bustransaktion in Zeiten hohen Verkehrsaufkommens zeitlich begrenzt. Zusammen mit der Prioritätenstruktur bei der Busarbitration und den mit der Buszuteilung einhergehenden Verzögerungszeiten läßt sich die Zeit, die ein busanfordernder Master benötigt, bis er den Bus benutzen kann, einigermaßen abschätzen.

Initialisierung, Konfigurierung. Der PCI-Bus erlaubt eine vollständig durch die Software betriebene Initialisierung bzw. Konfigurierung seiner Komponenten, sobald das Gesamtsystem eingeschaltet wird. Dementsprechend muß sich ein Benutzer, der sein Mikroprozessorsystem oder seinen Rechner um eine PCI-Karte erweitert, nicht um die Einbindung dieser Karte in das System kümmern (*plug and play!*). Zur Unterstützung dieses Vorgehens ist jedes Device mit 64 „Configuration-Register“ im DWORD-Format (insgesamt 256 Bytes) ausgestattet, wovon 16 Register als sog. Header vordefiniert (Bild 5-51) und die restlichen Register device-spezifisch festgelegt sind. Die Adressen dieser Register bilden den Konfigurationsadreßraum des Device. Über eines dieser Register kann während der Konfigurierungsphase ein ggf. vorhandener Festspeicher (Expansion ROM) angesprochen werden, der device-spezifische Software mit z.B. folgenden Funktionen enthält: Power-on-Selbsttest, Initialisierung, Interrupt-Service, BIOS. Die Software kann als Code für unterschiedliche Prozessoren vorhanden sein, so daß die Karte auf unterschiedlichen Systemen lauffähig ist.

Festlegung der Adreßbereiche. Eine wichtige Komponente bei der Konfigurierung des Gesamtsystems ist die Erstellung eines *Adreßraumbereigungsplans (memory map)*, der jedem Device einen oder mehrere Adreßbereiche entsprechend der von

31	16 15	0
Device ID	Vendor ID	0x00
Status	Command	0x04
Class Code	Revision ID	0x08
BIST	Header Type	Cache Line Size
Base Address Registers		
Cardbus CIS Pointer		
Subsystem ID	Subsystem Vendor ID	0x2C
Expansion ROM Base Address		
Reserved		
Reserved		
Max_Lat	Min_Gnt	Interrupt Pin
		Interrupt Line

Bild 5-51. Header des Konfigurierungsadreßraums, Typ 0x00 (nach [PCI 1995])

ihm geforderten Anzahl an Bereichen und deren Größen zuweist. Hierfür sind im Header eines jeden Device sechs 32-Bit-Register zur Verwendung als 32-Bit- oder 64-Bit-*Basisadreßregister* vorgesehen (Bild 5-51). Sie enthalten einerseits Konfigurierungsangaben des Device und werden andererseits von der Initialisierungssoftware mit den Basisadreßangaben der Memory-Map geladen. So gibt das Device mit Bit 0 eines Basisadreßregisters an, ob dieses für einen I/O- oder einen Memory-Adreßbereich benutzt wird. Bei einem I/O-Adreßbereich (der nur 32-Bit-Adressierung vorsieht) ist zusätzlich das Bit 1 reserviert. Dementsprechend ist der kleinstmögliche I/O-Bereich auf vier Bytes festgelegt. Bei einem Memory-Adreßbereich gibt das Device mit den Registerbits 2 und 1 an, ob der Adreßbereich irgendwo im 32-Bit-Adreßraum liegen darf, ob er innerhalb des ersten 1-Mbyte-Bereichs liegen muß, oder ob er irgendwo im 64-Bit-Adreßraum liegen darf. Letzteres besagt, daß das Device mit 64-Bit-Adressierung arbeitet und das Basisadreßregister dementsprechend zwei Header-Register umfaßt. Mit dem Registerbit 3 legt es außerdem fest, ob Zugriffe im Vorgriff (prefetch) erlaubt sind. Der kleinstmögliche Adreßbereich umfaßt hier 16 Bytes, üblicherweise werden jedoch 4 Kbyte belegt, um den Aufwand für die Adreßdecodierung im Device zu verringern.

Die Größen seiner Adreßbereiche gibt ein Device dadurch an, daß es die innerhalb der Bereiche liegenden Bits seiner Basisadreßregister (mit Ausnahme der niedrigstwertigen 2 bzw. 4 Bits) fest verdrahtet mit 0 vorgibt. Dementsprechend

können nur die für die Basisadressen signifikanten Bits, d.h. die verbleibenden höherwertigen Bits eines jeden Basisadreßregisters, von der Initialisierungssoftware verändert werden. Bei einem 1-Mbyte-Adreßraum beispielsweise sind das die höherwertigen 12 oder 44 Bits. Um eine Bereichsgröße abzufragen, schreibt die Initialisierungssoftware zunächst 1-Bits in das Basisadreßregister, liest anschließend den Registerinhalt aus und wertet dabei die Registeraufteilung in 0-Bits und 1-Bits aus. Sobald sie aus den so bei allen Komponenten erfragten Größenangaben den Adreßraumbelegungsplan erstellt hat, beschreibt sie die signifikanten Bits der Basisadreßregister mit den aus diesem Plan resultierenden Anwählcodes. – Die 0-Bit-Vorgabe impliziert, daß alle Adreßbereiche an ihren natürlichen Grenzen im Gesamtadreßraum ausgerichtet sind, d.h. an den ganzzähligen Vielfachen der jeweiligen Bereichsgröße liegen.

Konfigurierungszyklus. Der Zugriff auf den Konfigurationsadreßraum eines Device erfolgt nicht direkt, sondern auf dem Umweg über die Host-to-PCI-Bridge (Bridge/Memory Controller in Bild 5-46, S. 346). Sie allein kann Konfigurierungszyklen ausführen, ausgelöst durch die Software. Dazu besitzt sie zwei spezielle Register, ein Adreßregister (CONFIG_ADDRESS) und ein Datenregister (CONFIG_DATA), die als I/O-Register ausgelegt sind. Um ein Datum aus einem Konfigurationsregister eines Device zu lesen oder es in ein solches Register zu schreiben, wird zunächst das CONFIG_ADDRESS-Register mittels eines I/O-Write-Kommandos mit folgender Information geladen: Busnummer (es können mehrere Busse in einer hierarchischen Mehrbusstruktur vorhanden sein), Device-Nummer, Funktionsnummer (es können mehrere PCI-Komponenten mit jeweils eigenem Konfigurationsregistersatz in einem Multi-Funktions-Agenten/Device vorhanden sein) und Registernummer. Anschließend wird dann mittels eines I/O-Read- oder I/O-Write-Kommandos ein Lese- bzw. Schreibzugriff auf das CONFIG_DATA-Register ausgeführt. Dieses Kommando wird von der Bridge in den entsprechenden Konfigurationszyklus – Configuration-Read bzw. Configuration-Write – umgesetzt, wobei sie den Inhalt des CONFIG_ADDRESS-Registers zur Bildung der Adreßinformation und das CONFIG_DATA-Register als Datenpuffer benutzt. Liegt das adressierte Device am PCI-Bus dieser Bridge, so erfolgt eine Adreßumsetzung gemäß Bild 5-52 (Zyklus vom Typ 0: AD[1::0] = 00). Liegt es an einem anderen PCI-Bus (hinter einer PCI-to-PCI-Bridge), so wird der gesamte Inhalt von CONFIG_ADDRESS auf den AD-Leitungen weitergereicht (Zyklus vom Typ 1: AD[1::0] = 01).

Bild 5-53 zeigt das Signalverhalten für einen Konfigurationszyklus, d.h. für den Zugriff auf die Konfigurationsregister eines PCI-Device, hier als Lesezugriff. Entgegen der Darstellung kann der Konfigurationszyklus auch mehrere Datenphasen haben. Das jeweilige Zugriffsformat ist durch die Byte-Enable-Signale beliebig angebar. Ausgewählt wird ein Device oder ein Multi-Funktions-Agent durch Aktivieren seines IDSEL-Eingangs, der die Wirkung eines Chip-Select-Signals hat, und durch die in einer Art Adreßphase übermittelte Codierung AD[1::0] = 00. Die Adressierung der Register erfolgt in dieser Phase mittels der

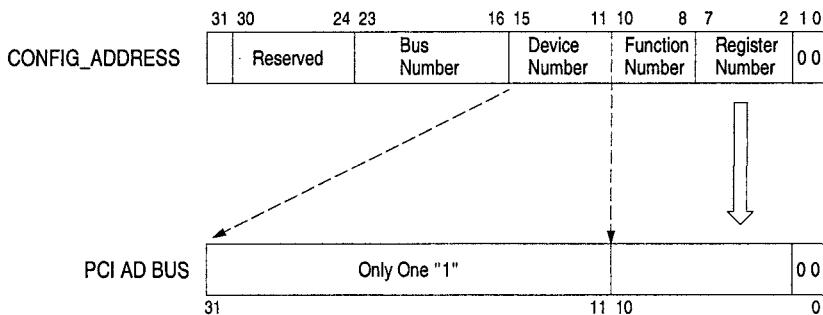


Bild 5-52. Adressumsetzung durch die Bridge für einen Konfigurationszyklus des Typs 0 (nach [PCI 1995])

Adressbits AD[7::2]. Mit den Adressbits AD[10::08] können bis zu acht Devices in einem Multi-Funktions-Agenten unterschieden werden. Wie das IDSEL-Signal für die individuelle Anwahl zu erzeugen ist, ist in der PCI-Spezifikation nicht festgelegt. Vorgeschlagen wird jedoch, bis zu 16 Devices über die Adressbits AD[31::16] im 1-aus-16-Code anzuwählen.

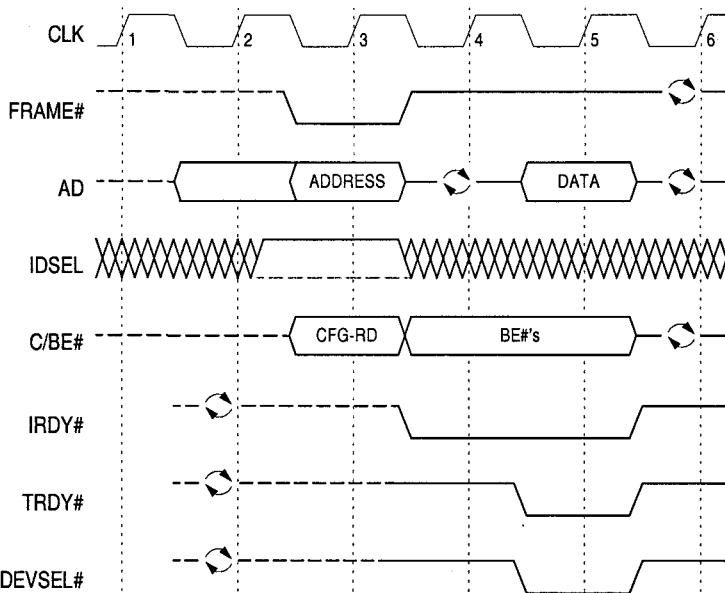


Bild 5-53. Konfigurationszyklus für das Lesen (nach [PCI 1995])

Cache-Unterstützung. In einer typischen PCI-Systemstruktur, wie sie in Bild 5-46 am Anfang dieses Abschnitts gezeigt ist (S. 346), ist der Prozessor über eine Host-to-PCI-Bridge mit dem PCI-Bus verbunden. Der Prozessor hat dabei üblicherweise On-chip-L1-Caches und meist einen Off-chip-L2-Cache, wobei der

L2-Cache-Controller in der Bridge untergebracht sein kann. Am PCI-Bus selbst gibt es weitere Agenten mit Masterfunktion, z.B. DMA-Controller, die wie der Prozessor (bzw. sein L2-Cache) Speicherzugriffe durchführen. Teilen sich nun der Prozessor und andere PCI-Master Speicherbereiche in PCI-Targets (*Shared-memory-System*) und sind diese Bereiche als „cacheable“ ausgewiesen, so muß sowohl für die Cache- und Speicherzugriffe des Prozessors wie auch für die Speicherzugriffe der PCI-Master sichergestellt werden, daß immer die aktuellen Daten vorliegen. Dieses sog. Datenkohärenzproblem und Lösungen dazu sind in 6.2.4 ausführlich beschrieben (zum besseren Verständnis der folgenden Erläuterungen siehe also zuerst dort).

Ein wichtiges Hilfsmittel zur Lösung des Kohärenzproblems auf Seiten des Cache ist das *Snooping*. Hierbei beobachtet der Cache (genauer der Cache-Controller) den Bus hinsichtlich der Speicherzugriffe anderer PCI-Master und vergleicht die von diesen Mastern auf dem Bus erzeugten Adressen mit den Adreßeinträgen seiner Cache-Zeilen (snoop operation). Bei Adreßgleichheit (snoop hit) reagiert der Cache ggf. durch Aktionen, die den jeweiligen Cache-Eintrag betreffen. Da die Snoop-Aktivitäten und -Ergebnisse des Cache aber auch Auswirkungen auf die adressierten PCI-Targets haben, müssen diese parallel dazu über die Snoop-Aktivitäten informiert werden, wofür der PCI-Bus die in 5.6.2 beschriebenen Signale SDONE (snoop done) und SBO# (snoop backoff) vorsieht. Sie signalisieren drei mögliche Snoop-Zustände, die mit STANDBY, CLEAN und HITM bezeichnet werden und die durch folgende Signalpegel codiert sind (x steht für 0 oder 1):

	SDONE	SBO#
STANDBY	0	x
CLEAN	1	1
HITM	1	0

Für das vom aktuellen Speicherzugriff betroffene PCI-Target haben sie folgende Bedeutungen. CLEAN: es handelt es sich um einen konfliktfreien Zugriff. STANDBY: der Cache ist bereit, eine Snoop-Operation durchzuführen, oder führt gerade eine solche durch; das Target muß den Speicherzugriff blockieren, bis der Zustand CLEAN signalisiert wird. HITM: die Snoop-Operation hat einen Snoop-Hit auf eine im Cache modifizierte Zeile ergeben, weshalb der Speicherzugriff abzubrechen ist. Hintergrund für HITM ist das Aktualisieren der Cache-Zeilen nach dem Copy-back-Verfahren. Dieses beinhaltet, daß nach dem Schreiben des Prozessors in eine solche Zeile der entsprechende Block im Hauptspeicher nicht mehr aktuell ist (6.2.4).

Auf der Basis des *Copy-back-Verfahrens* sind nachfolgend die drei relevanten Zugriffssituationen in groben Schritten dargestellt. Wir orientieren uns dabei an dem Buch [Shanley, Anderson 1995], in dem diese Zugriffe noch detaillierter und teilweise durch Signaldiagramme unterstützt beschrieben sind. Davon ausgehend, daß der für die Snoop-Operationen zuständige Cache-Controller Bestandteil der

Host-to-PCI-Bridge ist, ordnen wir dessen Aktionen dieser Bridge zu, geben also die Bridge als den Akteur von Snoop-Operationen an.

Clean-Snoop. Ein PCI-Master initiiert einen Lese- oder Schreibzugriff auf ein PCI-Target mit „cacheable“ Speicher. Der adressierte Block ist im Cache entweder in nichtmodifizierter Form oder nicht vorhanden. Die Schritte sind:

1. Die Bridge signalisiert dem Target den Zustand STANDBY und unterzieht die Speicheradresse einer Snoop-Operation.
2. Das Target blockiert daraufhin gegenüber dem PCI-Master den ersten Datentransport durch Nichtaktivieren von TRDY#.
3. Die Snoop-Operation in der Bridge resultiert in einem Snoop-Hit auf eine nichtmodifizierte Cache-Zeile (*clean line*) bzw. in einem Snoop-Miss (Cache-Block nicht vorhanden), woraufhin die Bridge dem Target den Zustand CLEAN signalisiert.
4. Das Target gibt der Schreibanforderung des PCI-Masters durch Aktivieren von TRDY# statt.
5. Die Bridge geht vom Zustand CLEAN wieder in den Zustand STANDBY. Der Zugriff war somit konfliktfrei. – Im Falle eines Schreibzugriffs mit Snoop-Hit auf eine nichtmodifizierte Cache-Zeile wird diese von der Bridge bei der Snoop-Operation als ungültig gekennzeichnet (*invalid line*).

Snoop-Hit on modified Line followed by Write-Back. Ein PCI-Master initiiert einen Lese- oder Schreibzugriff auf ein PCI-Target mit „cacheable“ Speicher. Der adressierte Block liegt im Cache vor und ist modifiziert. Die Schritte sind:

1. Die Bridge signalisiert dem Target den Zustand STANDBY und unterzieht die Speicheradresse einer Snoop-Operation.
2. Das Target blockiert daraufhin gegenüber dem PCI-Master den ersten Datentransport durch Nichtaktivieren von TRDY#.
3. Die Snoop-Operation in der Bridge resultiert in einem Snoop-Hit auf eine modifizierte Cache-Zeile (*dirty line*), woraufhin die Bridge den Zustand HITM signalisiert.
4. Das Target hält weiterhin TRDY# inaktiviert und leitet mit Aktivieren des STOP#-Signals den Retry-Abbruch des Zugriffs ein.
5. Der initierende PCI-Master versetzt daraufhin den Bus in den Ruhezustand.
6. Die Bridge hält HITM weiterhin aufrecht, woraufhin weitere in dieser Zeit anfallende Zugriffsanforderungen anderer PCI-Master auf irgendwelche Targets mit „cacheable“ Speicher von diesen mit Retry abgewiesen werden. Die Bridge fordert parallel dazu den Bus beim Arbiter an und leitet nach Zuteilung des Busses die Write-back-Operation mit dem Target ein. Sie zeigt dazu während der Adreßphase den Zustand CLEAN an.

7. Das Target erlaubt daraufhin das Zurückschreiben der Cache-Zeile.
8. Die Bridge zeigt STANDBY an, woraufhin der unterbrochene PCI-Master den Zugriff auf das Target erneut startet, diesmal als Clean-Snoop.

Memory-Write-and-Invalidate-Command. Ein PCI-Master initiiert eine Memory-Write-and-Invalidate-Transaktion, um eine oder mehrere Zeilen seines Cache in einen „cacheable“ Speicher eines Target zurückzuschreiben. – Hier empfiehlt sich folgendes Vorgehen: Unabhängig davon, ob die Snoop-Operation in der Bridge in einem Snoop-Miss oder einem Snoop-Hit auf eine nichtmodifizierte oder modifizierte Cache-Zeile resultiert, signalisiert sie den Zustand CLEAN, woraufhin das Target TRDY# aktiviert und die Daten vom initierenden PCI-Master zumindest immer bis zu einer vollständigen Cache-Zeile übernimmt. Bei einem Snoop-Hit auf eine modifizierte Zeile wird diese von der Bridge in ihrem Cache invalidiert, da der korrespondierende Block im Speicher ja vollständig neu geschrieben wird und die Cache-Zeile dann sowieso nicht mehr aktuell ist. (Das im vorangehenden Absatz beschriebene alternative Vorgehen mit Retry und Write-Back hätte den Nachteil, daß die Write-back-Operation zusätzlich Zeit kosten würde, die Endzustände im Target-Speicher und in der Cache-Zeile aber dieselben wären wie bei dem empfohlenen Vorgehen.)

64-Bit-Erweiterung. Für die Erweiterung des PCI-Busses von 32 auf 64 Bit werden die in 5.6.2 beschriebenen Zusatzsignale REQ64#, ACK64#, AD[63::32], C/BE[7::4]# und PAR64 benötigt. Grundlage der Erweiterung ist, daß 32-Bit-Agenten in unveränderter Form weiter am Bus betrieben werden können, also auch mit 64-Bit-Agenten zusammen. Dementsprechend müssen 64-Bit-Agenten bei Bustransaktionen grundsätzlich vom 32-Bit-Modus ausgehen und können dann ggf. eine 64-Bit-Transaktion aushandeln. Die einzige sinnvolle Anwendung dieser Erweiterung liegt bei den Memory-Kommandos mit ihren hohen Anforderungen an die maximale Übertragungsrate, die sie deshalb auch allein unterstützen. Interrupt-Acknowledge- und Special-Cycles sind grundsätzlich nur für den 32-Bit-Modus ausgelegt. Auch bei den I/O- und Konfigurierungskommandos sind die Anforderungen an die Übertragungsrate nicht so hoch, daß sie die höhere Komplexität der 64-Bit-Erweiterung rechtfertigen würden.

64-Bit-Transaktion werden jeweils zu Beginn einer Transaktion ausgehandelt. Dazu aktiviert der Master zusammen mit FRAME# das REG64#-Signal. Ist das Target für die 64-Bit-Erweiterung ausgelegt, so reagiert es durch Aktivieren von ACK64# zusammen mit DEVSEL#. Beide Signale werden bis zum Abschluß der 64-Bit-Transaktion aktiv gehalten und gleichzeitig mit FRAME# und DEVSEL# wieder inaktiviert. Ist das Target nur für 32-Bit-Übertragungen ausgelegt, so bleibt sein ACK64#-Signal inaktiv, REQ64# hingegen hat denselben Verlauf wie bei der 64-Bit-Erweiterung. Der Master nimmt dann von einer 64-Bit-Übertragung Abstand und teilt die beabsichtigten 64-Bit-Datentransfers in jeweils zwei aufeinanderfolgende Datenphasen auf. Allerdings geht er bei der ersten Datenübertragung noch von einem 64-Bit-Transfer aus und legt das erste 64-Bit-Datum

und die entsprechenden acht Byte-Enable-Signale auf den Leitungen AD[63::00] und C/BE[7::0]# an. Er überträgt jedoch dann in einer zweiten Datenphase das Hi-DWORD und dessen Byte-Enable-Signale noch einmal, und zwar auf den Leitungen AD[31::00] und C/BE[3::0]#. Die folgenden 64-Bit-Daten werden dann von vornherein nur noch auf den Leitungen AD[31::00] und C/BE[3::0]# angelegt, zuerst das Lo-DWORD, dann das Hi-DWORD.

Bei der 64-Bit-Erweiterung erfolgt die Adressierung standardmäßig im 32-Bit-Modus (single-address cycle, SAC). Soll mit 64 Bit adressiert werden, so muß dies vom Master durch das Dual-Address-Cycle-Kommando in der Adreßphase angezeigt werden (siehe oben: Buskommandos). Allerdings wird auch dann, wenn Master und Target für eine 64-Bit-Übertragung geeignet sind, die Adresse in zwei Teilen übertragen, wobei zwar bereits während der ersten Übertragung die vollständige Adresse über AD[63::00] sowie das DAC-Kommando und das Kommando für die Datenphase über C/BE[3::0]# bzw. C/BE[7::4]# übermittelt werden. Jedoch werden in der zweiten Phase nochmals der höherwertige Adreßteil (hi-address) über AD[31::00] und das Kommando für die Datenphase sowohl über C/BE[3::0]# als auch C/BE[7::4]# übertragen. Diese Aufteilung auf zwei Übertragungen hängt damit zusammen, daß der Master die 64-Bit-Fähigkeit des Target erst dann ermitteln kann, wenn er an dieses die vollständige Adresse übertragen hat, und dazu muß er für die Adreßphase zunächst einmal von einem 32-Bit-Target ausgehen.

5.6.4 PCI-X-Bus

Wie in 5.1.5 schon angedeutet, haben die Erfordernisse schneller Ein-/Ausgabetechniken, wie Gigabit-Ethernet, Ultra-3-SCSI und Fibre-Channel, zu einer Weiterentwicklung des PCI-Busses hin zum PCI-X-Bus geführt, der dann 1999 in der Revision 1.0 spezifiziert wurde [PCI-X 1999]. Dieser Bus hat, wie auch schon der PCI-Bus, eine sehr umfangreiche Spezifikation, der wir in der hier geforderten Kürze nicht gerecht werden können. Deshalb nachfolgend lediglich einige Angaben zu grundsätzlichen Unterschieden gegenüber dem PCI-Bus.

Leistungsdaten und Kompatibilität. Der PCI-X-Bus erlaubt 32-Bit- und 64-Bit-Übertragungen bei einer Busbreite von 64 Bit (Multiplexbus) und mit 64-Bit-Adressierung. Er bietet Bustaktfrequenzen von 66, 100 und 133 MHz bei (frequenzabhängig) bis zu vier, zwei bzw. einem Steckplatz und hat eine Signalspannung von 3,3 V. (Beim 66 MHz-Bustakt, den es auch beim PCI-Bus gibt, verdoppelt sich also die Anzahl der Steckplätze.) Der PCI-X-Bus ist wie der PCI-Bus kaskadierbar, d.h., es können Systeme mit bis zu 256 Bussegmenten mit ggf. unterschiedlichen Bustaktfrequenzen gebildet werden (Bild 5-54). Zum PCI-Bus gibt es eine weitgehende Kompatibilität. So können einerseits PCI-X-Komponenten so entworfen werden, daß sie in PCI-Systemen mit den herkömmlichen PCI-Bustaktfrequenzen und PCI-Modi betrieben werden können. Andererseits sind

herkömmliche PCI-Komponenten in PCI-X-Systemen lauffähig, indem sie mit ihnen verträglicher Taktfrequenz betrieben werden. – Die maximale Übertragungsrate des PCI-X-Busses liegt bei 1,06 Gbyte/s (64 Bit, 133 MHz). Geplant ist eine Verdoppelung und eine Vervierfachung der Übertragungsrate mittels Double- und Quad-Data-Rate.

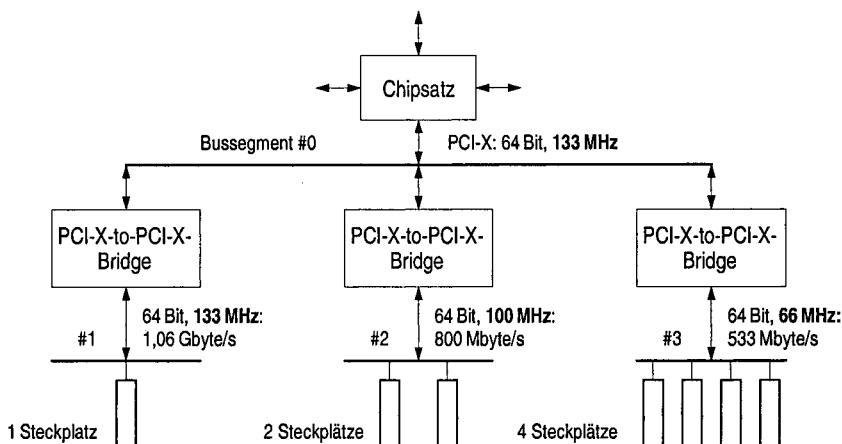


Bild 5-54. Hierarchische Struktur mit mehreren PCI-X-Bussegmenten mit unterschiedlichen Bustaktfrequenzen. Angabe der jeweils maximal möglichen Übertragungsrate und Anzahl an Steckplätzen (in Anlehnung an [Compaq 2000])

Bustechnik. Beim PCI-Bus ist die minimale Schrittweite des Bustaktes durch zwei aufeinanderfolgende Laufzeiten der Signale begrenzt, woraus sich die maximale Bustaktfrequenz von 66 MHz ergibt. Dies sind

1. die Laufzeit auf dem Bus, ausgehend vom einem Ausgangsregister des Senders bis hin zur Eingangslogik des Empfängers,
2. die Laufzeit für die Signaldecodierung in der Eingangslogik des Empfängers.

Beim PCI-X-Bus ist dieser Empfängerlogik ein Eingangsregister vorgeschaltet, mit dessen Hilfe sich die beiden Laufzeiten nach dem Prinzip der Fließbandverarbeitung auf zwei aufeinanderfolgende Takte schritte verteilen lassen. Bei gleichen Laufzeiten erreicht man dadurch die Verdoppelung der Taktfrequenz von 66 auf 133 MHz. Als Folge dieses Registers verzögert sich jedoch die Reaktion des Empfängers gegenüber der PCI-Technik um einen Takte schritt.

Kommandos. Die Kommandos des PCI-X-Busses entsprechen in ihren Bezeichnungen im wesentlichen denen des PCI-Busses und werden wie dort nach Einzelzugriffen (DWORD, 32-Bit-Doppelwort) und nach Blockzugriffen (Burst) unterschieden (Tabelle 5-5). Zu den funktionellen Unterschieden siehe nachfolgend.

Tabelle 5-5. PCI-X-Kommandos für Einzelzugriffe (DWORD) und Blockzugriffe (Burst) in Gegenüberstellung zu den PCI-Kommandos. Die mit „Alias to Memory Read Block“ und „Alias to Memory Write Block“ bezeichneten Kommandos gelten als reserviert und werden derzeit auf die genannten Kommandos abgebildet (nach [PCI-X 1999])

Codierung	PCI-Kommandos	PCI-X-Kommandos	Datenumfang
0000	Interrupt Acknowledge	Interrupt Acknowledge	DWORD
0001	Special Cycles	Special Cycles	DWORD
0010	I/O Read	I/O Read	DWORD
0011	I/O Write	I/O Write	DWORD
0110	Memory Read	Memory Read DWORD	DWORD
0111	Memory Write	Memory Write	Burst
1000	Reserved	Alias to Memory Read Block	Burst
1001	Reserved	Alias to Memory Write Block	Burst
1010	Configuration Read	Configuration Read	DWORD
1011	Configuration Write	Configuration Write	DWORD
1100	Memory Read Multiple	Split Completion	Burst
1101	Dual Address Cycle	Dual Address Cycle	–
1110	Memory Read Line	Memory Read Block	Burst
1111	Memory Write and Invalidate	Memory Write Block	Burst

Transaktionen. Die Transaktionen des PCI-X-Busses sind gegenüber denen des PCI-Busses zeitoptimiert. So darf ein Initiator generell keine Wartezeiten mehr erzeugen, und einem Target sind sie auch nur noch unmittelbar vor der ersten Datenphase erlaubt. Darüber hinaus beeinträchtigen die von einem Target benötigten Verarbeitungszeiten die zwischenzeitliche Busnutzung durch andere Busteilnehmer weniger als beim PCI-Bus. So sind auch hier sind Einzeltransfers und Blocktransfers vom jeweiligen Target unterbrechbar. Anders als beim PCI-Bus jedoch erfolgt danach ein Rollentausch, d.h., die unterbrochene Transaktion wird nicht vom ursprünglichen Initiator, dem Requester, sondern vom ursprünglichen Target, dem Completer, wiederaufgenommen, ggf. gefolgt von weiteren Unterbrechungen und Wiederaufnahmen durch den Completer. Der Completer, der die Unterbrechung verursacht hat und seine Zeiterfordernisse kennt, wird also zum Initiator und der Requester zum Target. Man bezeichnet diesen Ablauf als *geteilte Transaktion* (*split transaction*), die initiiierende Transaktion als Split-Request und die weiterführende(n) Transaktion(en) als Split-Completion(s). Zur Erinnerung: Beim PCI-Bus gibt es anstelle der geteilten Transaktion die verzögerte Transaktion (siehe S. 356). Hier bleibt die Steuerung beim unterbrochenen Initiator, und dieser fragt in Abständen immer wieder beim Target nach, bis dieses zur Fortsetzung der Transaktion bereit ist.

Bild 5-55 zeigt eine geteilte Transaktion für das Lesen von Daten in einer schematischen Darstellung mit folgenden grob wiedergegebenen Schritten:

- Der Requester A fordert als Initiator den Bus an (1.) und überträgt ein Memory-Read-Block-Kommando (2.) an den Completer B als Target.

2. Der Completer B unterbricht die Transaktion (disconnection) mittels Split-Response-Signalisierung (4.), wodurch diese zu einer Split-Request-Transaktion wird.
3. Der Completer B wird, sobald er die angeforderten Daten liefern kann, zum Initiator. Er fordert den Bus an (5.) und überträgt ein Split-Completion-Kommando (6.) und die Daten (8.) an den Requester A, der jetzt Target ist.

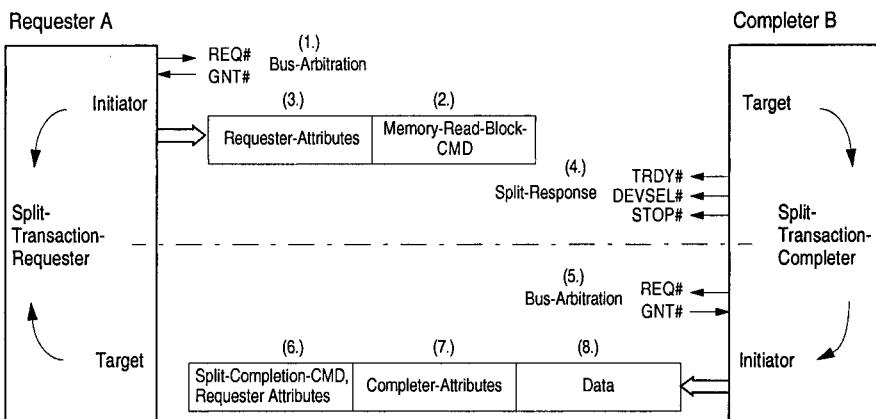


Bild 5-55. Ablauf einer geteilten Transaktion (split transaction) mit Numerierung der Reihenfolge der einzelnen Aktionen (in Anlehnung an [Compaq 2000])

Damit eine Transaktion ggf. als geteilte Transaktion ausgeführt werden kann, muß der Requester dem Completer sog. Requester-Attribute übermitteln (3.), die dieser dann für die Split-Completion-Transaktion nutzt. Dies sind u.a. die zur Requester-Identifikation erforderlichen Angaben: Busnummer, Device-Nummer und Funktionsnummer. Desgleichen übermittelt auch der Completer während der Split-Completion-Transaktion sog. Completer-Attribute (7.), u.a. die entsprechenden ihn identifizierenden Angaben. Die Transaktionsprotokolle sind dazu gegenüber dem PCI-Bus um eine Attribut-Phase erweitert, die unmittelbar auf die Adressierphase folgt.

Das Abbrechen einer Transaktion durch das Target kann bei Einzel- und Blocktransfers unmittelbar in der ersten Datenphase erfolgen, bei Blocktransfers danach nur noch an den natürlichen Grenzen von 128-Byte-Blöcken. Das heißt, bei Blocktransfers wird von Buskomponenten ausgegangen, die cache-basiert sind, was wiederum der Optimierung der Busbenutzung dient. Insgesamt kann ein Blocktransfer eine beliebige Anzahl von 1 bis 4096 Byte umfassen. Aufgrund der möglichen Aufteilung einer solchen Bytesequenz auf mehrere Split-Completion-Transaktionen wird in den Requester- und Completer-Attributen immer auch die aktuelle, d.h. noch zu übertragende Byteanzahl (byte count) mitgeführt.

Busphasen. Bild 5-56 zeigt die Busphasen einer Schreibtransaktion mit vier Datentransfers (Datenphasen). Im Unterschied zur entsprechenden PCI-Transaktion wird die Adressierphase von einer Attributphase und diese wiederum von einer Target-Response-Phase von wenigstens einem Takt Dauer gefolgt. Dieser eine Takt ergibt sich aus der beim PCI-X-Bus durch das empfängerseitige Eingangsregister verzögerten Reaktion des Target. Weitere Takte dieser Phase entstehen ggf. durch vom Target ausgelöste Wartezustände. Grundsätzlich umfaßt somit die Schreibtransaktion zwei Takte mehr als die des PCI-Busses (vergleiche dazu Bild 5-49, S. 356). Eine Lesetransaktion hat im Prinzip dieselben Busphasen, jedoch ist nach der Attributphase ein Turnaround-Zyklus für die Umkehrung der Signalrichtung des Multiplexbusses erforderlich. Dieser Zyklus überlappt sich mit dem ersten Takt der Target-Response-Phase. Dementsprechend benötigt die Lesetransaktion nur einen Takt mehr als die entsprechende PCI-Transaktion, nämlich den für die Attributphase (vergleiche dazu Bild 5-48, S. 355).

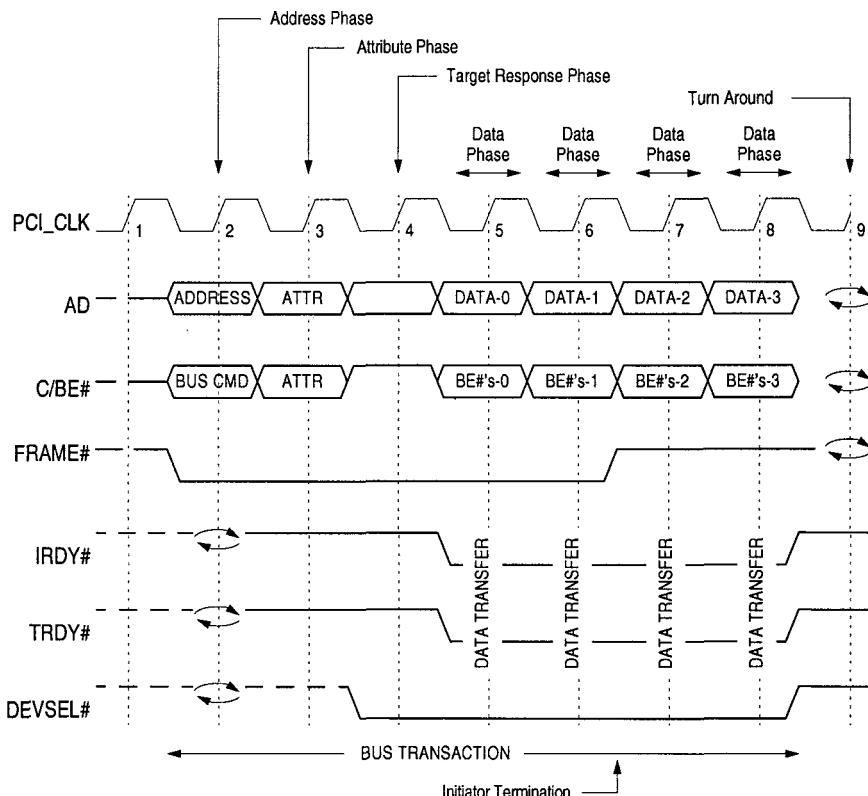


Bild 5-56. Signalverhalten bei einer Schreibtransaktion (nach [PCI-X 1999])

6 Speicherorganisation

Die Forderung nach leistungsfähigen Mikroprozessorsystemen setzt neben Prozessoren mit hohen Verarbeitungsgeschwindigkeiten auch Speicher mit großen Kapazitäten und geringen Zugriffszeiten voraus. Diese Forderungen lassen sich allein durch den Hauptspeicher nicht erfüllen. Abhilfe schafft eine hierarchische Anordnung von Speichern mit kurzen Zugriffszeiten auf der einen Seite und großen Kapazitäten auf der anderen Seite. So werden zum einen zwischen den „schnellen“ Registerspeicher des Prozessors und den „langsameren“ Hauptspeicher Pufferspeicher mit kurzen Zugriffszeiten, sog. Caches geschaltet. Zum andern wird die Speicherkapazität des Hauptspeichers durch die Einbeziehung von Hintergrundspeichern, z.B. Festplattenspeichern, um Größenordnungen erweitert. – Abschnitt 6.1 beschreibt den Aufbau von Caches und Hauptspeichern mittels statischer und dynamischer RAM-Bausteine sowie strukturelle Maßnahmen zur Erreichung möglichst hoher Datenübertragungsraten zwischen Prozessor, Cache und Hauptspeicher. In Abschnitt 6.2 wird dann auf die prinzipiellen Strukturen von Caches und auf die Verwaltung der in ihnen gespeicherten Daten eingegangen. Dabei geht es insbesondere um die Datenkohärenz, d.h. um das Aktualisieren der Daten in Cache und Hauptspeicher bei Mehrmaster- und Mehrprozessorsystemen. In Abschnitt 6.3 werden schließlich Techniken der Hauptspeicherverwaltung im Zusammenspiel mit Festplatten als Hintergrundspeicher beschrieben. Besonderes Augenmerk liegt hier auf den Strukturen von Speicher-verwaltungseinheiten (memory management units, MMUs).

6.1 Speicheraufbau und Speicherzugriff

Für den Aufbau von *Schreib-/Lesespeichern* – Hauptspeicher und Cache – stehen zwei Arten von Speicherbausteinen zur Verfügung:

- statische RAMs (SRAMs) mit kurzen Zugriffszeiten, relativ kleinen Speicherkapazitäten und hohen Kosten,
- dynamische RAMs (DRAMs) mit gegenüber SRAMs größeren Speicherkapazitäten, längeren Zugriffszeiten und geringeren Kosten.

SRAMs werden aus Gründen der höheren Kosten und des größeren Platzbedarfs (bei gleicher Kapazitätsanforderung sind mehr Bausteine erforderlich) heute vorwiegend für Caches eingesetzt, DRAMs aus den Gegengründen für den Aufbau der sehr viel größeren Hauptspeicher.

DRAMs unterscheiden sich von SRAMs zusätzlich dadurch, daß sie nach dem Zugriff eine Erholzeit benötigen, bevor der nächste Zugriff beginnen kann. Diese Zeit, die sich auf direkt aufeinanderfolgende Speicherzugriffe verzögernd auswirkt, läßt sich unter bestimmten Bedingungen durch eine verschränkte Adressierung von Speicherbänken überbrücken (6.1.3). Auf der Grundlage dieser Technik läßt sich ggf. auch die Zugriffszeit verkürzen – zumindest aus Sicht des Prozessors –, wenn dieser in der Lage ist, direkt aufeinanderfolgende Buszyklen überlappend auszuführen (6.1.4). Eine weitere Technik, Speicherzugriffe zu verkürzen, ist der blockweise Zugriff, wie er bei Datentransporten mit Caches üblich ist (6.1.5). Hier sehen statische und dynamische RAMs bausteininterne Zugriffstechniken vor, die es erlauben, Folgezugriffe innerhalb eines „Blockbuszyklus“ mit gegenüber dem ersten Zugriff verkürzten Zugriffszeiten auszuführen (6.1.6).

Anmerkung. Die Bezeichnung *RAM* (*random access memory*) resultiert daraus, daß auf die Speicherzellen eines Speicherbausteins direkt zugegriffen werden kann. Im Deutschen bezeichnet man dies als *wahlfreien Zugriff*. Allerdings stehen diese Bezeichnungen im Konflikt mit dem Begriff ROM, da auch ein ROM die genannten Merkmale aufweist.

6.1.1 Funktionsweise von SRAMs und DRAMs

In der technischen Ausführung von SRAM- und DRAM-Bausteinen unterscheidet man zwischen den älteren, *asynchronen Bausteinen* und den moderneren, *synchronen Bausteinen*. Beiden Bausteintypen gemeinsam ist die Realisierung der einzelnen Speicherzellen (Bitspeicherung) in Asynchrontechnik (wenn auch bei SRAMs und DRAMs in unterschiedlicher Weise). Diese Technik bestimmt bei den asynchronen Bausteinen auch die Signale an den Bausteinanschlüssen, was insbesondere bei DRAMs zu einer aufwendigen Speichersteuerung (DRAM-Controller) führt, da hier viele Zeitparameter zu berücksichtigen sind (≥ 50). Anders ist dies bei den synchronen Bausteinen. Hier unterliegen die Signale einem Takt, was ihre Handhabung wesentlich vereinfacht. Bei synchronen DRAMs reduzieren sich dadurch die Zeitparameter auf einige wenige. Sychrone DRAMs haben außerdem eine bausteininterne Aufteilung des Speicherfeldes in Speicherbänke, auf die überlappend zugegriffen werden kann, wodurch sich Blockzugriffe optimieren lassen. Darüber hinaus übertragen synchrone SRAMs und DRAMs ihre Daten mit Double-Data-Rate, was sehr hohe Zugriffsraten ermöglicht.

Für die nachfolgenden, grundsätzlichen Betrachtungen zur Funktionsweise von SRAMs und DRAMs können wir uns auf die asynchronen Bausteine und den Einzelzugriff beschränken. Auf die Wirkungsweise synchroner Bausteine und auf die Blockzugriffsmechanismen gehen wir in 6.1.6 ein.

Statische RAMs (SRAMs). SRAM-Bausteine haben Zugriffsbreiten von einem, vier, acht, meist aber 16 oder 32 Bit (mit ggf. zusätzlich je einem Paritätsbit pro 8 Bit) entsprechend der Breite ihrer Speicherzellen. Multipliziert mit der Anzahl an Speicherzellen ergibt sich die Gesamtkapazität eines Bausteins mit derzeit bis zu

32 Mbit. Beispiele für Bausteinausführungen (Zellenanzahl \times Zugriffsbreite) sind $256\text{ K} \times 16\text{ Bit}$ und $128\text{ K} \times 32\text{ Bit}$ (4 Mbit) oder $1\text{ M} \times 16\text{ Bit}$ und $512\text{ K} \times 32\text{ Bit}$ (16 Mbit). Bausteinintern sind die Speicherbits in einem oder mehreren zweidimensionalen Feldern angeordnet, wodurch eine hohe Packungsdichte auf dem Halbleitersubstrat erreicht wird.

Bild 6-1 zeigt dazu als Beispiel einen statischen RAM-Baustein in der Ausführung $1\text{ M} \times 1\text{ Bit}$. Sein *Speicherfeld* ist hier quadratisch mit 4 K Zeilen (*rows*) und 4 K Spalten (*columns*) ausgelegt, es könnte aber auch ein anderes Zeilen/Spalten-Verhältnis haben. Für das Schreiben und Lesen hat der Baustein je einen Datenanschluß D_{in} und D_{out} . Die Anwahl eines Speicherbits erfolgt durch Selektieren der ihm zugeordneten Zeile und Spalte. Dazu wird die eine Hälfte des Adreßworts als *Zeilenadresse* von einem Zeilenadreßdecodierer und die andere Hälfte als *Spaltenadresse* von einem Spaltenadreßdecodierer ausgewertet. Diese wählen die entsprechende Bitposition aus (Kreuzungspunkt von ausgewählter Zeile und ausgewählter Spalte). Bei einem Schreibvorgang schalten sie das an D_{in} anliegende Datensignal, verstärkt durch einen *Schreibverstärker*, auf die Bitspeicherzelle durch (Demultiplexer); bei einem Lesevorgang schalten sie den Inhalt der Bitspeicherzelle, verstärkt durch einen *Leseverstärker*, auf die Datenleitung D_{out} .

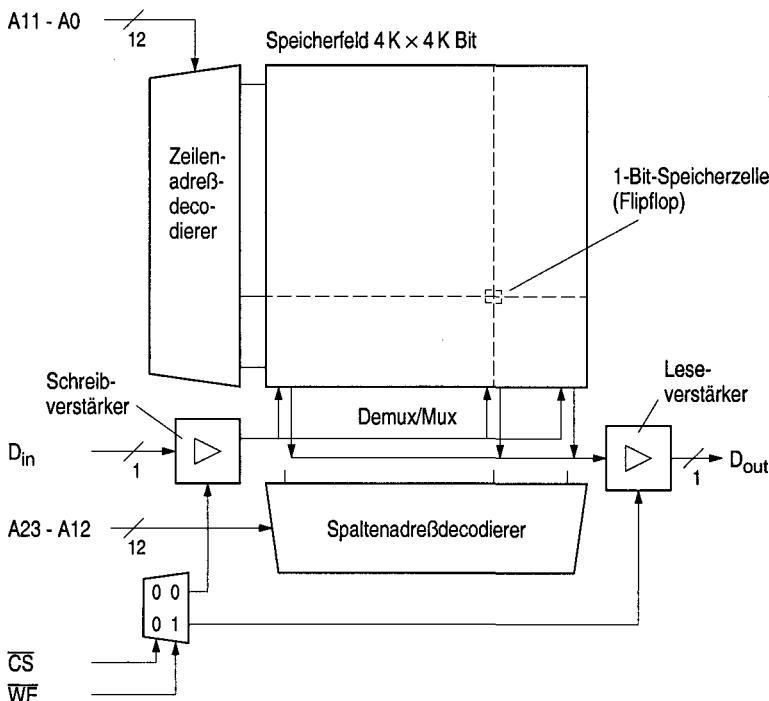


Bild 6-1. Struktur eines SRAM-Bausteins mit 16 Mbit in der Ausführung $16\text{ M} \times 1\text{ Bit}$

durch (Multiplexer). SRAMs mit Speicherzellen von mehr als einem Bit Breite haben entsprechend mehr Datenleitungen sowie Schreib- und Leseverstärker, so daß sämtliche Bits einer Speicherzelle parallel geschrieben bzw. gelesen werden können.

Zur Ansteuerung des Bausteins sind neben den Adresssignalen ein Baustein-anwahlsignal \overline{CS} sowie ein Schreibsignal \overline{WE} erforderlich. Schreib- und Lesevorgänge sind grundsätzlich nur bei aktiviertem \overline{CS} , d.h. bei $\overline{CS} = 0$ möglich. Bei $\overline{CS} = 1$ wie auch beim Schreibvorgang wird die Datenleitung D_{out} bei den meisten Bausteinen in den hochohmigen Zustand geschaltet, so daß D_{in} und D_{out} zu einem bidirektionalen Anschluß zusammengefaßt werden können. Dadurch können mehrere Speichereinheiten ohne gegenseitige Beeinflussung am Bus betrieben werden. Das Schreibsignal \overline{WE} wird von der Speicheransteuerung aus dem R/W-Signal gebildet und ist nur bei $\overline{CS} = 0$ wirksam. Es bewirkt die bausteininterne Durchschaltung für den Schreibzyklus ($\overline{WE} = 0$) bzw. den Lesezyklus ($\overline{WE} = 1$).

Bild 6-2 zeigt einen Kreuzungspunkt des Speicherfeldes des statischen RAM-Bausteins im Detail. Die eigentliche *Speicherzelle* ist ein Flipflop, bestehend aus den beiden rückgekoppelten Transistoren T_1 und T_2 sowie den beiden durch Transistoren realisierten Lastwiderständen R_L . Angewählt wird die Zelle mittels der vom Zeilenadreßdecodierer aktivierte Zeilenanwahlleitung j . Sie steuert die beiden Durchschalttransistoren T_3 und T_4 , die den nichtinvertierten und den invertierten Anschlußpunkt der Speicherzelle mit den beiden internen Datenleitungen D_i und \overline{D}_i verbinden. Beim Schreiben werden diese Leitungen mit dem Schreibpotential bzw. dem invertierten Schreibpotential belegt, beim Lesen werden die von der Zelle gelieferten Potentiale durch den Leseverstärker ausgewertet. Die Schreib- und Leseverstärker sind dementsprechend als Differenzverstär-

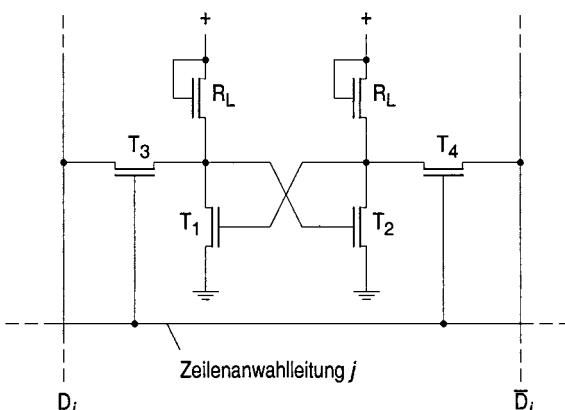


Bild 6-2. 1-Bit-Speicherzelle (Flipflop) eines statischen RAM-Bausteins (Bild 6-1)

ker ausgelegt. Die eigentliche Spaltenanwahl, d.h. das Durchschalten der beiden Datenleitungen zum jeweiligen Verstärker, geschieht durch zwei weitere, im Bild nicht gezeigte Transistoren, die vom Spaltenadreßdecodierer angesteuert werden.

Das Signal-Zeitverhalten für das Lesen und Schreiben zeigt Bild 6-3 in einer schematisierten Darstellung, bei der die in den Datenblättern üblicherweise angegebenen Toleranzen für die Zeitpunkte der Pegelübergänge nicht berücksichtigt sind. Die Zugriffe beginnen mit dem Anlegen der Adresse und, sobald diese stabil ist, mit dem Aktivieren des \overline{CS} -Signals.

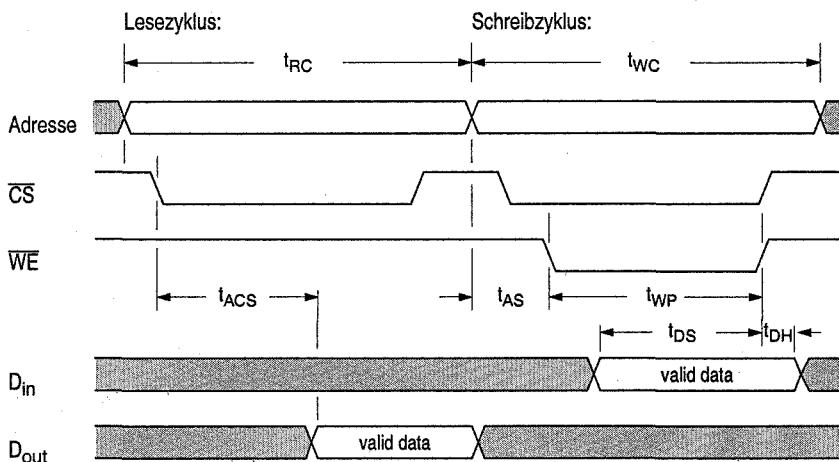


Bild 6-3. Lesezyklus und Schreibzyklus für den statischen RAM-Baustein nach Bild 6-1 mit den Zeitparametern t_{RC} Read-Cycle, t_{ACS} Access-Time-from-Chip-Select, t_{WC} Write-Cycle, t_{AS} Address-Set-Up, t_{WP} Write-Pulse-Width, t_{DS} Data-Set-Up, t_{DH} Data-Hold

Beim Lesevorgang bedarf es dann, ausgehend von $\overline{CS} = 0$, einer bestimmten Zugriffszeit (t_{ACS}), bis das Datensignal am Datenausgang D_{out} gültig ist. Das \overline{WE} -Signal wird während des gesamten Lesezyklus inaktiv gehalten. Beim Schreibvorgang wird zunächst das Stabilisieren der Adreßdecodierer abgewartet, bevor das Schreibsignal \overline{WE} aktiviert wird (t_{AS}). Dadurch wird das Schreiben in eine unter Umständen fälschlicherweise adressierte Bitposition ausgeschlossen. Das Schreibsignal muß danach für die Zeit des Schreibpulses aktiv sein (t_{WP}); es bewirkt mit seiner steigenden Flanke die Datenübernahme. Das am Dateneingang D_{in} anliegende Datensignal muß, um den Schreibvorgang korrekt durchführen zu können, eine gewisse Zeit vor der Datenübernahme bereitgestellt (t_{DS}) und eine gewisse Zeit danach noch gehalten werden (t_{DH}). Die **Zugriffszeit** ergibt sich dementsprechend aus der Summe von t_{AS} , t_{WP} und t_{DH} . Sie ist annähernd gleich der **Zykluszeit** t_{WC} (bzw. t_{RC}), d.h. der insgesamt erforderlichen Zeit bis zum nächsten möglichen Zugriff. Typische Zugriffszeiten von SRAMs liegen bei 5 bis 80 ns.

Hinsichtlich weiterer Details zum Aufbau von SRAMs siehe z.B. [Sharma 1997], [Klar 1996], [Rhein, Freitag 1992].

Dynamische RAMs (DRAMs). DRAM-Speicherbausteine haben Zugriffsbreiten von vier, acht, 16 oder 32 Bit (mit ggf. zusätzlich je einem Paritätsbit pro 8 Bit), wobei Bausteine mit Zugriffsbreiten von vier, acht und 16 Bit am gebräuchlichsten sind. Bei asynchronen DRAMs liegen die Speicherkapazitäten bei bis zu 64 Mbit, bei synchronen DRAMs sind Bausteine mit 256 Mbit am gebräuchlichsten, es gibt sie aber auch mit 512 Mbit und 1 Gbit. Typische Ausführungen von z.B. 64-Mbit-Bausteinen sind $16\text{ M} \times 4\text{ Bit}$, $8\text{ M} \times 8\text{ Bit}$ und $4\text{ M} \times 16\text{ Bit}$.

Bild 6-4 zeigt als Beispiel die Struktur eines dynamischen RAM-Bausteins in der Ausführung $16\text{ M} \times 4\text{ Bit}$, also einen Baustein mit 4-Bit-Zugriff. Sein *Speicherfeld* ist wie das des SRAM-Bausteins nach Bild 6-1 quadratisch ausgelegt (was nicht zwingend ist), hat hier jedoch die vierfache Speicherkapazität von 8 K Zeilen (*rows*) und 2 K Spalten (*columns*) zu je 4 Bit. Im Unterschied zum SRAM erfolgt die Speicherung der Bits nicht in Flipflops (Zustand, statisch), sondern in Kondensatoren (Ladung, dynamisch). Das hat zur Folge, daß bei Lese- und Schreibzugriffen zunächst die gesamte Zeile aus dem Speicherfeld ausgelesen

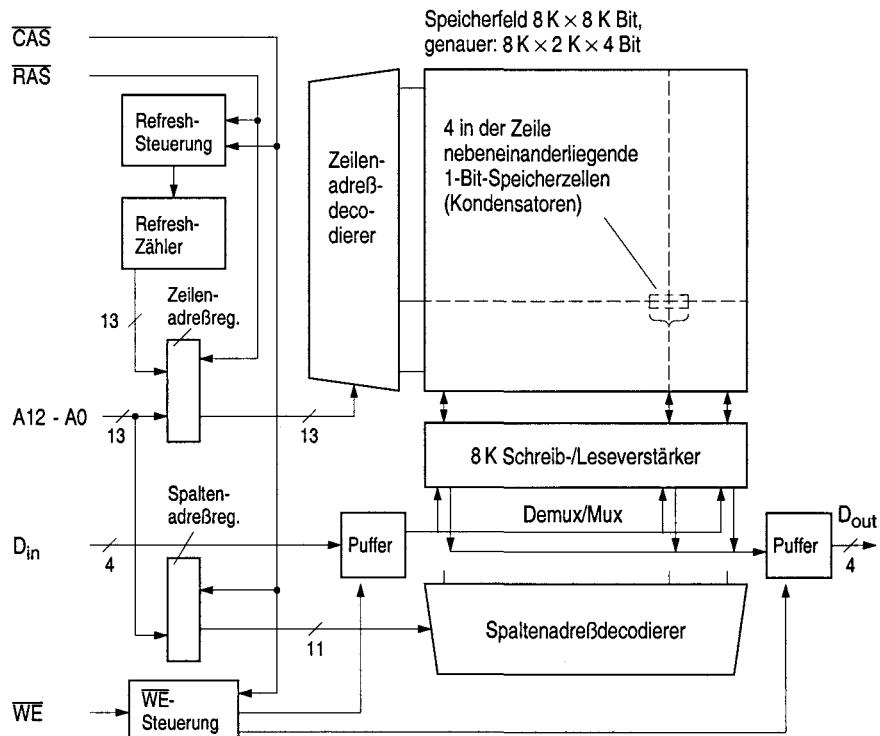


Bild 6-4. Struktur eines DRAM-Bausteins mit 64 Mbit in der Ausführung $16\text{ M} \times 4\text{ Bit}$

und bausteinintern in Flipflops, die auch als *Schreib-/Leseverstärker* dienen, zwischengespeichert werden muß, ehe dann die Spaltenanwahl wirksam werden kann. Diese Zweischrittigkeit erlaubt es, den Baustein mit nur der halben Anzahl an Adreßanschlüssen auszustatten. Sie werden im Multiplexbetrieb benutzt, indem zunächst die *Zeilenadresse* (hier 13 Bit) zum Auslesen der Zeile in die Schreib-/Leseverstärker und danach die *Spaltenadresse* (hier 11 Bit) zur Anwahl der eigentlichen Datenbits angelegt wird. Die Gültigkeit der Zeilen- bzw. Spaltenadresse wird dem Baustein mittels zweier Anwahlleitungen $\overline{\text{RAS}}$ (row address strobe) und $\overline{\text{CAS}}$ (column address strobe) signalisiert. Sie werden dazu genutzt, die beiden Adreßteile in zwei bausteininterne Adreßregister zu übernehmen.

Anmerkungen. (1.) Das Halbieren der Adreßanschlüsse hat gegenüber SRAMs den Vorteil kleinerer Bausteinabmessungen. Daß dadurch die Bausteinsteuerung komplizierter wird, ist kein großer Nachteil, da DRAMs sowieso komplexere Steuerungsvorgänge als SRAMs haben (z.B. den Refresh-Vorgang). Das macht gegenüber SRAMs, die mit einer einfachen Ansteuerlogik auskommen, eine aufwendige Steuereinheit, einen *DRAM-Controller* nötig. Dieser übernimmt dann auch das Anlegen der beiden Adreßteile. (2.) Heutige synchrone DRAM-Bausteine haben, wie erwähnt, anstelle des nur einen Speicherfeldes vier kleinere Speicherfelder, sog. Bänke, mit jeweils eigenen Zeilenadreßregistern, Zeilen- und Spaltenadreßdecodierern sowie Schreib-/Leseverstärkern (siehe Bild 6-15, S. 397). Hier werden dann zwei der Zeilenadreßbits zur *Bankanwahl* benutzt.

Bild 6-5 zeigt eine 1-Bit-Speicherzelle eines dynamischen RAM-Bausteins mit minimalem Aufwand an Transistoren, nämlich als 1-Transistor-Zelle. Die eigentliche Speicherzelle besteht aus dem Kondensator C_{Sp} (i.allg. die Kapazität eines nicht gezeichneten Transistors), der abhängig von seinem Ladezustand eine 1 oder eine 0 repräsentiert. Angewählt wird der Kondensator durch eine Wortleitung WL_j (Zeile), die ihn mittels des Durchschalttransistors T mit einer Bitleitung BL_i (Spalte) verbindet. Über die Bitleitung wird die Speicherinformation in den Kondensator geschrieben (Anlegen von 1- oder 0-Pegel, Laden des Kondensators) oder aus ihm in den der Spalte zugeordneten Schreib-/Leseverstärker gelesen (Auswerten der Kondensatorladung, Entladen des Kondensators).

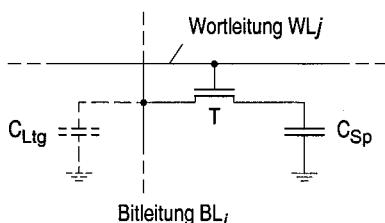


Bild 6-5. 1-Bit-Speicherzelle eines dynamischen RAM-Bausteins (Bild 6-4)

Der Vorteil einer solchen dynamischen gegenüber der statischen Zelle ist ihr geringerer Platzbedarf, wodurch sich die Speicherkapazität von DRAMs gegenüber SRAMs bei gleicher Chipfläche wenigstens vervierfachen läßt. Ihr Nachteil ist allerdings, daß sich der Kondensator beim Aufschalten auf die Bitleitung entlädt (durch Umladen auf die Leitungskapazität C_{Ltg}), so daß die in ihm gespeicherte

Information zerstört wird. Um die Information nach einem Lese- oder Schreibzugriff wiederherstellen zu können, muß, wie oben erwähnt, die jeweils adressierte Zeile des Speicherfeldes insgesamt in Flipflops ausgelesen werden. Diese Flipflops wirken dann, sobald sie sich stabilisiert haben, mit (verstärkten) Pegeln auf die Kondensatoren zurück und stellen so die ursprüngliche Information wieder her. Beim Schreibzugriff werden dabei die zu schreibenden Bits (in unserem Beispiel 4) nicht aus den Flipflops zurückgeschrieben, sondern die am Baustein anliegenden Datensignale werden mittels Spaltenadresse/Demultiplexer direkt den Bitleitungen zugeführt, so daß die korrespondierenden Schreib-/Leseverstärker nicht zur Wirkung kommen.

Ein weiterer Nachteil der DRAM-Technologie ist, daß sich die Speicher kondensatoren aufgrund von Leckströmen langsam entladen. Das heißt, daß die Inhalte sämtlicher Zellen des Speicherfeldes innerhalb einer Mindestzeit „aufgefrischt“ werden müssen, damit sie nicht verloren gehen. Hierfür sind spezielle *Refresh-Zyklen* erforderlich, die jeweils eine Zeile in die Schreib-/Leseverstärker auslesen, von wo aus sie dann in bekannter Weise sofort wieder zurückgeschrieben werden. Diese Zyklen werden üblicherweise zwischen die „normalen“ Schreib- und Lesezugriffe geschoben, sie können aber auch für alle Zeilen des Speicherfeldes unmittelbar aufeinanderfolgen. Unterstützt werden sie durch einen bausteininternen Zeilenadreßzähler (Refresh-Zähler in Bild 6-4), der mit jedem Refresh-Zyklus inkrementiert wird (wrap-around), so daß nacheinander alle Zeilen erreicht werden. Ein solcher Zyklus wird bei asynchronen DRAMs z.B. dadurch ausgelöst, daß das $\overline{\text{CAS}}$ -Signal vor dem $\overline{\text{RAS}}$ -Signal aktiviert wird ($\overline{\text{CAS}}$ -before- $\overline{\text{RAS}}$ -Refresh). Daneben gibt es andere Auslösemechanismen. Gesteuert wird der Refresh-Vorgang durch den *DRAM-Controller*. – Die Mindestzeit für das Auffrischen des gesamten Speicherfeldes, der *Refresh-Abstand*, beträgt bei heutigen DRAM-Bausteinen typischerweise 64 ms.

Bild 6-6 zeigt das Signal-Zeitverhalten eines asynchronen DRAM-Bausteins mit aufeinanderfolgender Adressierung von Zeile und Spalte, gesteuert durch die Gültigkeitssignale $\overline{\text{RAS}}$ und $\overline{\text{CAS}}$. Für das Auslesen der adressierten Zeile in die Schreib-/Leseverstärker benötigt er die Zeit t_{RCD} , für das Bereitstellen der mit der Spaltenadresse korrespondierenden Bits an den Bausteinanschlüssen (Lesen) die Zeit t_{CL} . Hieraus ergibt sich die *Zugriffszeit* des Bausteins zu $t_{\text{RAS}} = t_{\text{RCD}} + t_{\text{CL}}$. Um den Zyklus zu beenden, ist eine sog. *Erholzeit (recovery time)* erforderlich, die mit t_{RP} (*precharge time*) bezeichnet wird. Erst nach dieser Zeit kann ein erneuter Zugriff erfolgen. Die *Zykluszeit* $t_{\text{RC}} = t_{\text{RAS}} + t_{\text{RP}}$ ist bei DRAMs dementsprechend länger als die Zugriffszeit. Typische Werte bei asynchronen DRAMs sind 60 ns für die Zugriffszeit und 110 ns für die Zykluszeit. Kürzere Zugriffszeiten erreicht man für Folgezugriffe bei *Blockbuszyklen*, wie sie auch von asynchronen DRAMs unterstützt werden. Hier kommt als Zugriffszeit allein t_{CL} zum Tragen (siehe 6.1.6).

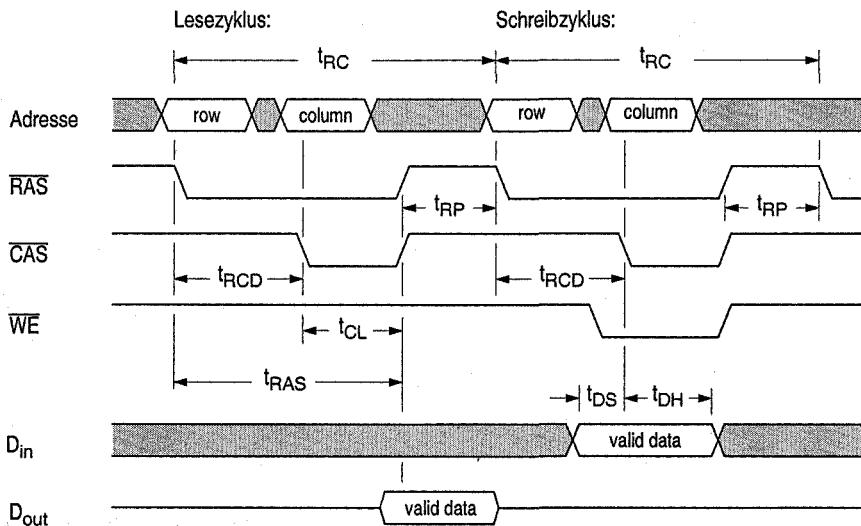


Bild 6-6. Lesezyklus und Schreibzyklus für den dynamischen RAM-Baustein nach Bild 6-4 mit den Zeitparametern t_{RC} RAS-Cycle (Zykluszeit), t_{RCD} RAS-to-CAS-Delay, t_{CL} CAS-Latency, t_{RAS} Access-Time-from-RAS (Zugriffszeit), t_{RP} RAS-Precharge (Erholzeit), t_{DS} Data-Set-Up, t_{DH} Data-Hold

Die Erholzeit t_{RP} ist dadurch bedingt, daß das Auslesen der DRAM-Zelle aufgrund der relativ kleinen Kondensatorladungen und der damit einhergehenden niedrigen Spannungspotentiale nicht, wie in Bild 6-5 etwas vereinfacht dargestellt, asymmetrisch durch nur eine Bitleitung BL, sondern symmetrisch mittels zweier Bitleitungen BL und \overline{BL} , d.h. differentiell erfolgt. Beide Leitungen müssen dazu vor dem Zugriff hinsichtlich ihrer Leitungskapazitäten „voraufgeladen“ werden. \overline{BL} wirkt dann mit dem Voraufladepotential als Referenz für die vom Kondensator an BL abgegebene Ladung. Das *Voraufladen (precharging)* erfolgt, wie Bild 6-6 zeigt, vorausschauend am Ende eines jeden Zyklus.

Hinsichtlich weiterer Details zum Aufbau von DRAMs siehe z.B. [Sharma 1997], [Klar 1996], [Rhein, Freitag 1992].

6.1.2 Aufbau einer Speichereinheit

Speichereinheiten mit ihren Zugriffsbreiten von z.B. 8, 16, 32 oder 64 Bit bestehen abhängig von der Zugriffsbreite des verwendeten Speicherbausteins meist aus mehreren dieser Bausteine, die dann parallel betrieben werden. Der Aufbau einer solchen Einheit kann individuell erfolgen, unter Nutzung der Vielfalt an SRAM- und DRAM-Bausteinen. Es kann aber auch auf vorgefertigte Steckkarten, sog. *Speichermodule*, zurückgegriffen werden. Diese gibt es ebenfalls in vielen Varianten, sowohl als DRAM-Module, wie sie z.B. als Hauptspeicher und

Video-Speicher bei Rechnern eingesetzt werden, als auch als SRAM-Module, wie sie z.B. als prozessorexterne Caches bei Rechnern und als Hauptspeicher bei kleineren Systemen, z.B. bei Steuerungen, zum Einsatz kommen. – Sind mehrere physisch eigenständige Speichereinheiten im System vorhanden, z.B. zur Erweiterung der Speicherkapazität oder um bestimmte Zugriffsmechanismen zu unterstützen, so bezeichnet man sie auch als *Speicherbänke*.

Vom Speicherbaustein zur Speichereinheit. Bild 6-7a zeigt den nach außen hin repräsentierten Adreßraum eines DRAM-Bausteins mit einer Zugriffsstruktur $16\text{ M} \times 4\text{ Bit}$ entsprechend Bild 6-4. Er stellt eine 4-Bit-Spalte in einem 16 M Adressen umfassenden Bereich dar, zu deren Adressierung die Bits A23 bis A0 des Adreßworts benötigt werden. (Die dabei getroffene Zuordnung Zeilenadresse = höherwertige Adreßbits, Spaltenadresse = niedrigerwertige Adreßbits ist zunächst willkürlich, sie ist aber unabdingbar, wenn Blockbuszyklen mit dem Speicher ausgeführt werden; siehe 6.1.5.) Um mit diesem Baustein einen Speicher im Zugriffsformat Byte aufzubauen (Teilbild b), werden zwei solcher Bausteine „nebeneinander“ angeordnet und diese mit den Adreßbits A23 bis A0 parallel adressiert. Ein solcher *Speicherblock* umfaßt jetzt 16 Mbyte. Bei einer Erweiterung des Speichers zu einer Speichereinheit mit 32-Bit-Zugriff werden wiederum vier solcher Blöcke mit Byteformat parallel betrieben, und zwar so, daß aufeinanderfolgende Adressen einen Speicherblockwechsel verursachen (Teilbild c, nicht-gestrichelter Teil). Hier werden die Adreßbits A1 und A0 zur Blockanwahl benötigt und dementsprechend die Bits A25 bis A2 zur Adressierung innerhalb eines Blocks verwendet. Die Speicherkapazität beträgt jetzt 64 Mbyte. Das ist bei dem verwendeten Baustein die kleinstmögliche Kapazität für eine Speichereinheit mit 32-Bit-Zugriff.

Bei Erweiterung der Kapazität eines Speichers um zusätzliche Speichereinheiten werden, wenn die Adreßräume aufeinanderfolgend angeordnet sein sollen, die höherwertigen Adreßbits zur Anwahl der Einheiten bzw. Bänke herangezogen. Teilbild c zeigt dies für eine Verdoppelung der Speicherkapazität von 64 auf 128 Mbyte unter Verwendung von A26 zur *Bankanwahl*. Speicherbänke können aber auch in anderer Weise angeordnet sein, wie dies z.B. beim Verschränken von Speicherbänken der Fall ist. Hier erfolgt die Bankselektion z.B. so, daß aufeinanderfolgende Wortadressen einen Bankwechsel verursachen. Für die Bankanwahl wird dann A2 herangezogen (siehe dazu 6.1.3).

DRAM-Module. Um den Aufbau oder die Konfigurierung von Rechnern zu vereinfachen, werden fertige Speichermodule mit unterschiedlichen Bausteintypen, Speicherkapazitäten und Datenanschlußbreiten angeboten, die als *SIMMs* (single in-line memory modules) und *DIMMs* (dual in-line memory modules) bezeichnet werden. Sie bestehen aus einer entsprechenden Anzahl von DRAM-Bausteinen und ggf. Zusatzbausteinen (siehe unten), die auf einer schmalen mit Steckkontakten versehenen Platine ein- oder beidseitig montiert sind, und für die es rechnerseitig entsprechende Steckfassungen auf der Grundplatine gibt. Bei SIMMs wird

die Kontaktreihe bei Datenanschlußbreiten von 8 Bit (ältere Bauart) und 32 Bit (spätere Bauart) nur einseitig genutzt, bei DIMMs (heute bei PCs gebräuchlich) wird sie bei einer Datenanschlußbreite von 64 Bit ein- oder beidseitig genutzt.

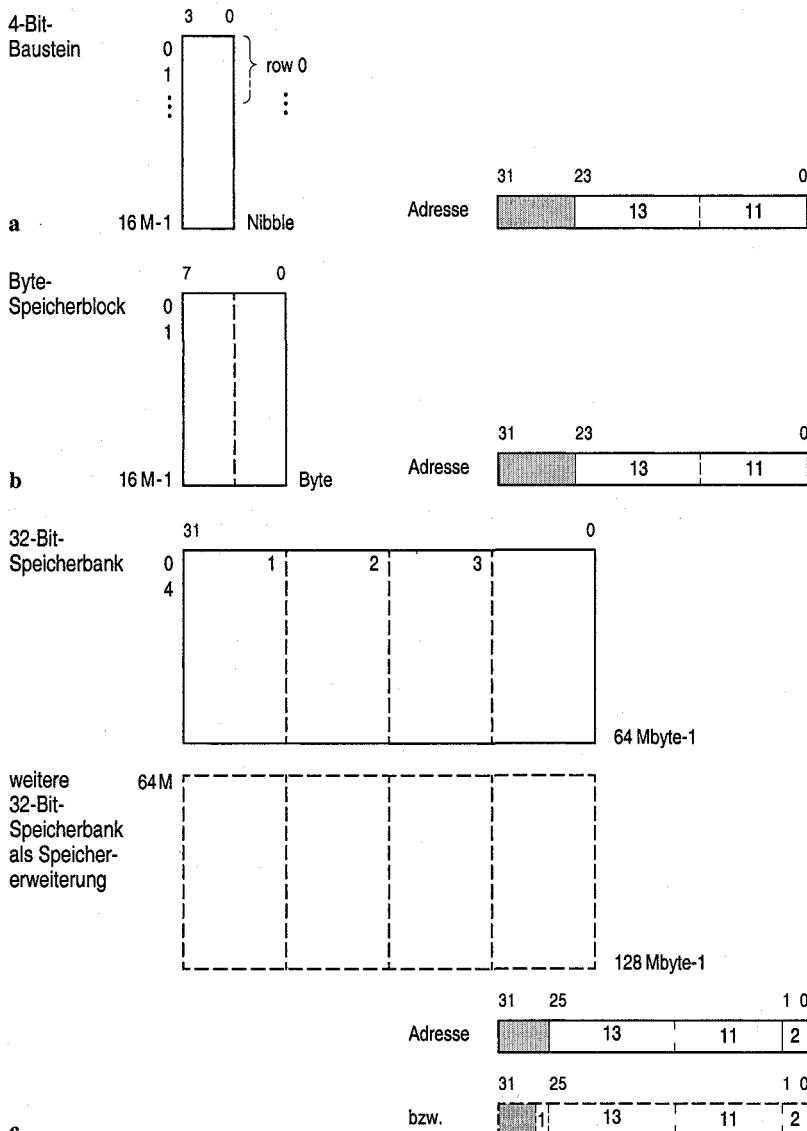


Bild 6-7. Speicheraufbau und zugehörige Adressaufteilungen. **a** DRAM-Baustein 16 M × 4 Bit (nach Bild 6-4), **b** 16-Mbyte-Speicherblock (8 Bit breit), bestehend aus 2 DRAM-Bausteinen 16 M × 4 Bit, **c** ein bzw. zwei 64-Mbyte-Speichereinheiten (32 Bit breit), bestehend aus je 4 16-Mbyte-Speicherblöcken (Adressierung: Big-endian-Byteanordnung)

Als Beispiel für ein 8-Bit-SIMM sei ein Modul mit einer Zugriffsorganisation von $4\text{ M} \times 1\text{ Byte}$ genannt, d.h. ein Modul mit 8 Datenbusanschlüssen plus ggf. einem Paritätsbit bei insgesamt 30 Kontakten, bestückt mit 8 (bzw. 9) asynchronen DRAM-Bausteinen mit je $4\text{ M} \times 1\text{ Bit}$. Bei der Speicheraufrüstung früherer PCs mit 32-Bit-Datenbus wurden vier solcher Module benötigt (pro Datenbusbyte 1 Modul), womit eine Speicherbank insgesamt 16 Mbyte umfaßte. 32-Bit-SIMMs, z.B. mit FPM- oder EDO-DRAMs bestückt, haben eine Zugriffsorganisation von $N \times 32\text{ Bit}$ plus ggf. 4 Paritätsbits bei insgesamt 72 Steckkontakten. Ihre Kapazitäten liegen bei 1, 2, 4, 8, 16, 32 und 64 Mbyte. In den Anfängen der 64-Bit-Prozessoren wurden jeweils zwei solcher Module für eine Speicherbank eingesetzt.

Den heutigen Prozessoren mit 64-Bit-Datenbus tragen DIMMS mit einer Datenanschlußbreite von 64 Bit und mit 168 (SDRAM), 184 (DDR-SDRAM) oder 240 Kontakten (DDR2-SDRAM) Rechnung. Eingesetzt werden Speicherbausteine mit Zugriffsbreiten von 4, 8, 16 oder 32 Bit, d.h., ein solches Modul kann mit 16, 8, 4 bzw. 2 dieser Bausteine bestückt sein. Es kann auch zwei solcher Bausteinreihen umfassen, bei dann doppelter Speicherkapazität. Als Beispiel sei ein einreihiges Modul mit einer Kapazität von 256 Mbyte genannt, bestückt mit 8 DDR-SDRAM-Bausteinen in der Ausführung $32\text{ M} \times 8\text{ Bit}$, d.h. mit 8 256-Mbit-Bausteinen. – Zu FPM- und EDO-DRAMs sowie zu den SDRAM-Varianten siehe 6.1.6.

DIMMs gibt es mit folgenden Zusatzmerkmalen und Bezeichnungen:

- *ECC-DIMM*: Ein ECC-DIMM ist mit einem Baustein zur Datensicherung versehen. Dieser führt eine 1-Bit-Fehlerkorrektur und eine 2-Bit-Fehlererkennung durch (*single error correction, double error detection, SECDED*, siehe auch 7.7.3), basierend auf 8 Sicherungsbits pro 64-Bit-Speicherwort. Das Speichermodul ist dazu um einen zusätzlichen Speicherbaustein erweitert; im obigen Beispiel um einen neunten, 8 Bit breiten SDRAM-Baustein.
- *Stacked* und *planar DIMM*: Ein stacked DIMM ist gegenüber dem „normalen“ planar DIMM mit doppelter Speicherkapazität ausgestattet, indem es entweder mit zwei übereinanderliegenden Bausteinreihen oder mit Bausteinen mit jeweils zwei Speichersubstraten im Gehäuse (dual-die chip) bestückt ist. Seine Verwendung liegt bei Rechnern mit großem Hauptspeicherbedarf, d.h. bei Workstations, Servern, Routern und Switches in Rechnernetzen.
- *Registered* und *unbuffered DIMM*: Das registered DIMM (gepuffertes DIMM) weist ein oder zwei Register zur Pufferung der Adresse und der Ansteuer-signale (Kommando) auf, womit Modulen mit vielen Speicherbausteinen und entsprechend hohen kapazitiven Lasten, z.B. den stacked DIMMs, Rechnung getragen wird. Die Register wirken als Fließbandregister und verzögern dementsprechend die Bausteinansteuerung um einen Takt. Zur Anpassung der Taktphase gibt es zusätzlich eine PLL-Schaltung (phase-locked loop).

Darüber hinaus sind DIMMs durch ihre Zugriffsdaten gekennzeichnet:

- Zeitparameter: z.B. DIMM 222 oder DIMM 2522, mit der Interpretation $t_{CL} = 2$ bzw. $2,5$, $t_{RCD} = 2$, $t_{RP} = 2$ (siehe 6.1.6).
- Maximal mögliche Übertragungsrate: z.B. PC3200, mit der Interpretation $3200 \text{ Mbyte/s} = 3,2 \text{ Gbyte/s}$. Dieser Wert ergibt sich aus der SDRAM-Taktfrequenz von 200 MHz, der Übertragung mit Double-Data-Rate (DDR1) und einem Datenwort von 8 Byte. Die hohe Interntaktfrequenz von 200 MHz wird durch Bausteinformen mit „Löt,kugeln“ anstelle von „Löt,beinen“ erreicht (geringere kapazitive Last). Noch höhere Übertragungsraten erlaubt die (neuere) DDR2-Technik, indem bei ihr die Übertragungsfrequenz gegenüber der Internfreqenz verdoppelt wird (siehe 6.1.6).

Eine DIMM-Variante sind die *SODIMMs* (small outline DIMM), die als Steckmodule mit geringeren Platinenabmessungen (nur vier Speicherbausteine, 144 oder 200 Kontakte) bei Laptops/Notebooks eingesetzt werden.

SRAM-Module. Sie gibt es ähnlich den DRAM-Modulen in SIMM- und DIMM-Ausführungen mit Zugriffsbreiten von 8, 16, 32 und 64 Bit (ggf. mit Paritätsbits), bestehend aus asynchronen SRAMs oder aus synchronen SRAMs in den Varianten „flow-through“ und „pipelined“ (siehe 6.1.6). Sie finden z.B. als schnelle (Haupt)Speicher mit geringen Speicherkapazitäten (und ggf. geringer Zugriffsbreite, wie bei Mikrocontroller-Anwendungen vorkommend) sowie als prozessorexterne L2-Cache-Module Verwendung. Für die Cache-Anwendung eignen sich die pipelined SRAMs besonders gut, da sie hoch taktbar sind und sich außerdem der Datenzugriff und die Folgeadressierung überlappen lassen. Als Beispiel sei ein SRAM-Modul mit 512 Kbyte und einer Zugriffsbreite von 64 Bit genannt, aufgebaut aus zwei SRAM-Bausteinen mit je $64 \text{ K} \times 32 \text{ Bit}$ für die Speicherung der Daten (16 K Cache-Zeilen zu je vier 64-Bit-Wörtern) und aus einem zusätzlichen SRAM mit $16 \text{ K} \times 8 \text{ Bit}$ für die Speicherung von 16 K Tags (siehe 6.2.2).

6.1.3 Verschränken von Speicherbänken

In Systemen ohne Caches, in denen der Mikroprozessor oder Mikrocontroller direkt und mit Einzelbuszyklen auf den DRAM-Hauptspeicher zugreift, sind unmittelbar aufeinanderfolgende Zugriffe (Folge von Einzelzugriffen) zunächst nur im Abstand der *Zykluszeit* der DRAM-Bausteine möglich. Das heißt, die Datenübertragung des ersten Zugriffs kann zwar bereits nach Ablauf der *Zugriffszeit* der Speicherbausteine abgeschlossen werden, bis zur Durchführung des nächsten Zugriffs muß aber die Erholzeit der Bausteine (das Voraufladen, siehe Bild 6-6, S. 379) abgewartet werden, was zu *Wartezyklen* des Prozessors führt. Dieser Nachteil der gegenüber der Zugriffszeit verlängerten Zykluszeit kann vermieden werden, wenn man den Speicher in eigenständige Bänke unterteilt und die Bankanwahl so auslegt, daß aufeinanderfolgende Zugriffe einen *Bankwechsel* verursa-

chen. Durch solche Bankwechsel ist es der einzelnen Speicherbank möglich, sich von einem Zugriff zu „erholen“, bevor auf sie erneut zugegriffen wird, wodurch unmittelbar aufeinanderfolgende Speicherzugriffe des Prozessors im Abstand der Zugriffszeit möglich sind. Man bezeichnet diese speicherseitige Strukturmaßnahme als Verschränken von Speicherbänken (*bank interleaving*) oder auch als *verschränkte Adressierung*.

Bild 6-8 zeigt dazu den zeitlichen Ablauf mehrerer direkt aufeinanderfolgender Lesezugriffe des Prozessors in einer schematischen Darstellung mit dem Takt des Prozessorbusses als Zeitmaßstab. Als Speicherbausteine werden DRAMs mit einer Zugriffszeit von idealisiert zwei Takten und einer *Speichererholzeit* von idealisiert einem Takt angenommen. Teilbild a demonstriert den Fall einer Speicherseinheit mit nur einer Speicherbank. Hier muß der Prozessor für jeden der Folgezugriffe jeweils einen Wartezyklus (wait) entsprechend der Erholzeit der Speicherbank (recovery) in seine Buszyklen einfügen. Teilbild b zeigt den Fall einer Speichereinheit mit zwei verschränkt adressierten Speicherbänken unter der Annahme, daß bei jedem Folgezugriff auch ein Bankwechsel stattfindet. Hierbei entfallen die Wartezyklen des Prozessors, so daß er die Daten mit der *minimalen Buszykluszeit* von zwei Takten übernehmen kann.

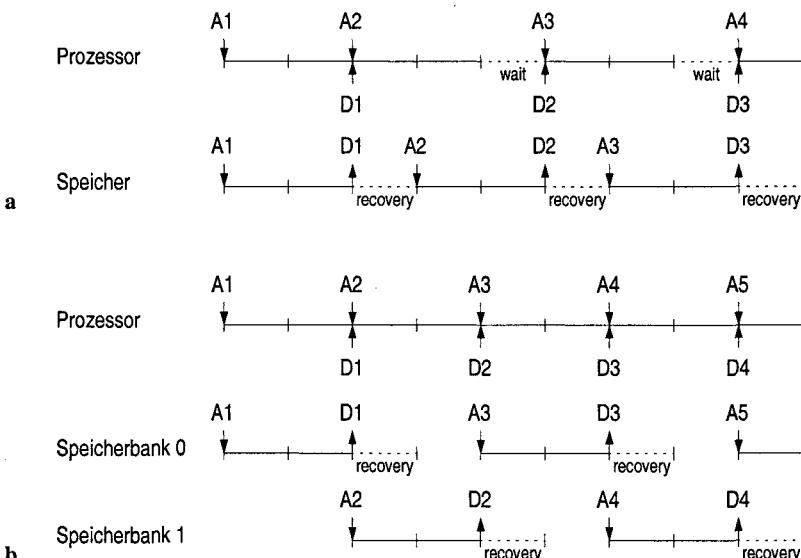


Bild 6-8. Direkt aufeinanderfolgende Lesezugriffe eines Prozessors auf einen DRAM-Speicher, **a** bei nur einer Speicherbank, **b** bei zwei verschränkt adressierten Speicherbänken. Minimale Buszykluszeit des Prozessors zwei Bustakte, Speicherzugriffszeit zwei und Speicherzykluszeit drei Bustakte. In der schematischen Darstellung ist die Zuordnung von Daten zu Adressen durch die Numerierung gegeben, z.B. D1, A1; der Takt des Busses ist durch die senkrechten Striche angedeutet

Bankwechsel mit hoher Wahrscheinlichkeit erhält man dann, wenn man die Bankumschaltung für aufeinanderfolgende Wortadressen auslegt. Bild 6-9 zeigt dies für vier 32-Bit-Speicherbänke einheitlicher Größe, deren Anwahl mittels der Adreßbits A3 und A2 erfolgt. (A1 und A0 dienen, wie üblich, zur Byteanwahl im Speicherwort.) Als Mindestanzahl reichen zwar zwei Bänke aus, da die Erholzeit der DRAM-Bausteine kürzer als deren Zugriffszeit ist. Eine über die Mindestanzahl hinausgehende Bankanzahl erhöht jedoch die Wahrscheinlichkeit eines Bankwechsels bei nichtfortzählender (zufälliger) Adressierung.

Anmerkung. Das Verschränken von Speicherbänken wurde bereits in der Frühzeit der Digitalrechner bei den als Hauptspeicher eingesetzten Magnetkernspeichern realisiert, die wie die DRAMs eine Erholzeit hatten, die dadurch bedingt war, daß das Lesen zerstörend erfolgte und die Information mit zusätzlichem Zeitaufwand wieder rückgeschrieben werden mußte.

6.1.4 Überlappen von Buszyklen

Die gegenüber der minimalen Buszykluszeit des Prozessors größeren Zugriffszeiten von Hauptspeichern und anderen Systemkomponenten können ggf. dadurch verdeckt werden, daß aufeinanderfolgende Buszyklen überlappend ausgeführt werden. Hier gibt es zwei Vorgehensweisen mit z.T. unterschiedlichen Zielrichtungen, die beide einen *geteilten Bus* (*split bus*, 5.1.4), d.h. die Trennung von Adreß- und Datenbus voraussetzen:

- Unmittelbar aufeinanderfolgende Buszyklen werden in Fließbandverarbeitung ausgeführt. Dabei bleiben die einzelnen Buszyklen in sich zusammenhängend und die direkte Aufeinanderfolge von Adressierung und zugehörigem Datentransport erhalten (address pipelining).
- Die Buszyklen werden geteilt, indem die Adressierung und der Datentransport als eigenständige Busoperationen mit eigenen Busarbitrierungen ausgeführt werden. Aufeinanderfolgende Buszyklen können sich dabei durchmischen, d.h., die herkömmliche Reihenfolge der Datentransporte kann durchbrochen werden (split transactions).

Beide Vorgehensweisen dienen der Optimierung von Speicherzugriffen, die zweite zusätzlich der Optimierung der Busnutzung bei mehreren Prozessoren.

Buszyklen in Fließbandverarbeitung (address pipelining). Bei Speicherzugriffen ergeben sich Wartezyklen – zusätzlich zu dem in 6.1.3 beschriebenen Fall – insbesondere dann, wenn die *Zugriffszeit* der verwendeten Speicherbausteine (DRAM, SRAM) größer ist als die *minimale Buszykluszeit* des Prozessors. Solche Wartezyklen können teilweise oder ganz vermieden werden, wenn der Prozessor so ausgelegt ist, daß sich Buszyklen in Fließbandtechnik überlappen können. Hierbei gibt der Prozessor die Adresse sowie die Status- und Byte-Enable-Signale für den nächsten Buszyklus bereits während des momentanen Buszyklus aus (*überlappende Adressierung*, address pipelining). Voraussetzungen auf der Spei-

cherseite sind eine Speicheraufteilung in Bänke mit verschränkter Adressierung, wie in 6.1.3 beschrieben, sowie für jede Bank zwei Pufferregister, um die Adreß- und Byte-Enable-Signale des momentanen Zyklus zu halten, wie in Bild 6-9 gezeigt. Damit kann – ein Bankwechsel vorausgesetzt – die zuerst angesprochene Bank ihren Zyklus fortsetzen und beenden, während die danach überlappend angesprochene Bank bereits die neue Adresse auswertet. Voraussetzung ist außerdem, daß die Speichersteuerung in der Lage ist, den Prozessor dazu aufzufordern, den nächsten Speicherzugriff vorzeitig einzuleiten, wozu in Bild 6-9 der DRAM-Controller mit dem Signal $\overline{\text{NARQ}}$ (next address request) ausgestattet ist.

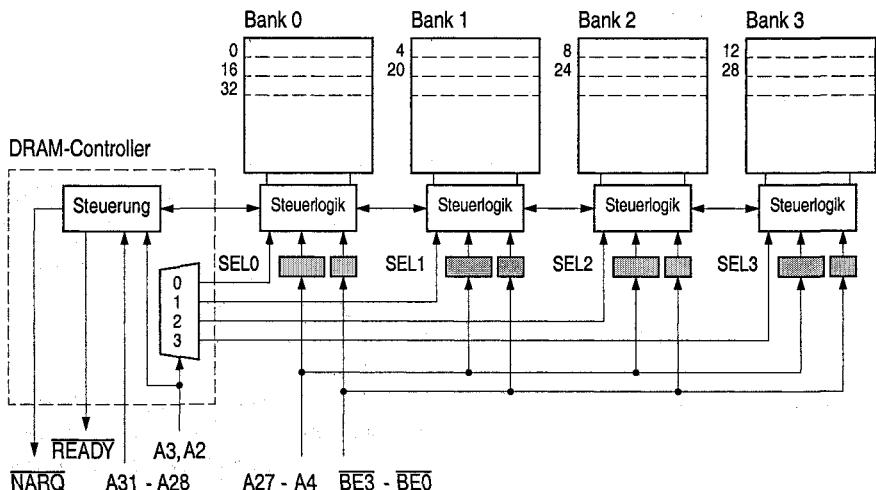


Bild 6-9. Verschränkung von vier Speicherbänken mit jeweils Wortzugriff. Erweiterung der Struktur zur Unterstützung sich überlappender Buszyklen: (1) um je zwei Pufferregister pro Bank (grau) zur Speicherung der Adreßbits A27 bis A4 und der Byte-Enable-Signale, (2) um das Signal $\overline{\text{NARQ}}$ zur Anforderung der vorzeitigen Adreßausgabe durch den Prozessor

Bild 6-10 zeigt dazu den zeitlichen Ablauf mehrerer direkt aufeinanderfolgender Lesezugriffe des Prozessors entsprechend Bild 6-8, jetzt jedoch mit einem „langsameren“ DRAM-Speicher mit einer *Zugriffszeit* von idealisiert drei Bustakten und einer *Speichererholzeit* von idealisiert zwei Bustakten. Bei dem in Teilbild a gezeigten Fall mit nur einer Speicherbank benötigt der Prozessor für jeden Folgezugriff drei Wartezyklen (wait), wovon sich einer aus der um einen Takt zu großen Zugriffszeit der DRAMs und die beiden andern aus der Speichererholzeit (recovery) ergeben. Teilbild b zeigt den Fall sich überlappender Buszyklen, in dem der Prozessor seine Adressen um einen Takt vorgezogen ausgibt. Die Speicherinheit besteht jetzt aus vier verschrankt adressierten Speicherbänken, wobei im Bild angenommen ist, daß bei jedem Folgezugriff auch ein Bankwechsel stattfindet. In diesem Fall kommt der Prozessor bei den Folgezugriffen ohne Wartezyklen aus. – Eine genauere Darstellung des Ablaufs gibt das folgende Beispiel.

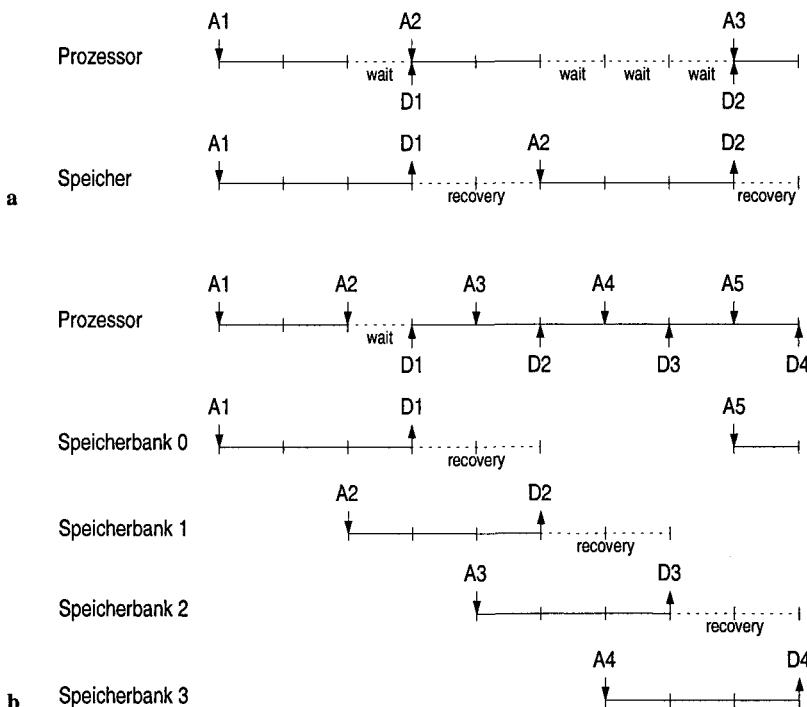


Bild 6-10. Direkt aufeinanderfolgende Lesezugriffe eines Prozessors auf einen DRAM-Speicher mit der Möglichkeit der Überlappung von Buszyklen, a bei nur einer Speicherbank, b bei vier ver-schränkt adressierten Speicherbänken. Minimale Buszykluszeit des Prozessors zwei Bustakte, Speicherzugriffszeit drei und Speicherzykluszeit fünf Bustakte

Beispiel 6.1. ► *Signalaustausch bei sich überlappenden Buszyklen.* Den für das Überlappen von Buszyklen erforderlichen Signalaustausch zwischen Speicher und Prozessor demonstriert Bild 6-11 für mehrere synchrone Lesezyklen entsprechend Bild 5-29 (S. 321). Der Prozessor hat hier eine minimale Buszykluszeit von zwei Takten, die Speicherinheit eine Zugriffszeit von drei Takten. Der erste Buszyklus benötigt, da sich der Bus zuvor im Ruhezustand befindet (bus idle), die volle Zugriffszeit von drei Takten. Dementsprechend verzögert der DRAM-Controller der Speicherinheit das READY-Signal um einen Takt und veranlaßt damit den Prozessor, für diesen ersten Zugriff einen Wartezyklus einzufügen. Er teilt jedoch dem Prozessor zuvor durch das Signal NARQ mit, daß dieser die nächste Adresse (Adresse 2) bereits während des Wartezyklus, d.h., um einen Takt überlappend ausgeben kann. Durch erneutes Aktivieren von NARQ während Buszyklus 2 und unter der Annahme, daß mit der Adressierung ein *Bankwechsel* stattfindet, wird das Überlappen mit Buszyklus 3 fortgesetzt. Die Lesedaten ab Buszyklus 2 stehen damit im Abstand von zwei Takten zur Übernahme durch den Prozessor bereit, obwohl zwischen der Ausgabe der Adresse und der Datenbereitstellung jeweils eine Speicherzugriffszeit von drei Takten liegt.

Bild 6-11 zeigt auch den Fall, daß bei der durch NARQ überlappend abgerufenen Adresse kein Bankwechsel stattfindet (Adresse 4). Hier verzögert der DRAM-Controller das READY-Signal entsprechend der zur Auflösung dieses *Bankkonflikts* erforderlichen Wartezyklen. Deren Anzahl ergibt sich aus der Erholzeit der Speicherbank für den vorangehenden Zugriff (Adresse 3), zuzüglich dem Wartezyklus, der durch das Wegfallen der Überlappung des neuen Zugriffs (Adresse 4)

anfällt. Setzt man die Speichererholzeit mit zwei Taktan um, so ergeben sich insgesamt drei Wartezyklen. Außer READY wird auch NARQ um drei Takte verzögert, so daß die nächste Adresse (Adresse 5) wieder einen Takt vor Abschluß des verlängerten Buszyklus angefordert wird. – Gibt der Prozessor eine Adresse für eine Speichereinheit aus, die nicht überlappend arbeiten kann, so wird in diesem Zyklus keine NARQ-Anforderung erzeugt.

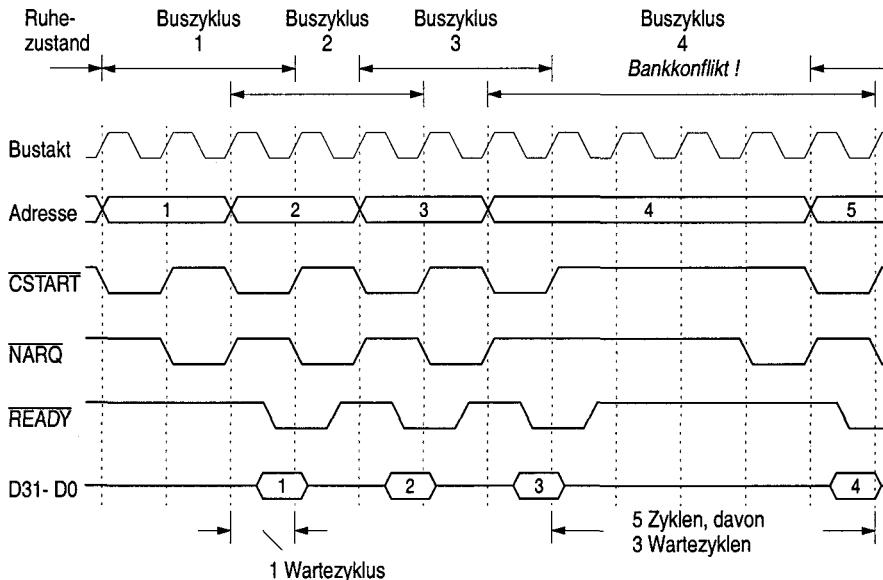


Bild 6-11. Unmittelbar aufeinanderfolgende Lesezugriffe auf einen DRAM-Speicher, der eine Zugriffszeit von drei und eine Zykluszeit von fünf Prozessortakten hat. Aufbau des Speichers mit verschränkten Bänken, die das Überlappen von Buszyklen zur Verschärfung der Zugriffe zulassen (Buszyklen 2 und 3). Bankkonflikt durch nicht stattfindenden Bankwechsel (Buszyklus 4) ◀

Das überlappende Ausführen von Buszyklen in Fließbandtechnik ist sowohl auf Einzelbuszyklen als auch auf Blockbuszyklen anwendbar, sofern der Prozessor jeweils dafür ausgestattet ist. Für *Einzelbuszyklen* typisch sind Systeme ohne Cache, bei denen also der Prozessor direkt auf den Hauptspeicher zugreift. Ein Beispiel aus der PC-Geschichte ist der Intel-Prozessor *i386* [Intel 1987] mit Überlappung zweier Buszyklen um einen Takt, wie in Beispiel 6.1 beschrieben. Andere Prozessoren aus dieser Zeit führen die Überlappung für mehr als zwei Buszyklen gleichzeitig und über mehr als nur einen Takt durch. Das Überlappen von *Blockbuszyklen* findet sich typischerweise in Systemen mit Caches. Beispiele hierfür sind der Pentium-Prozessor und seine Nachfolger. Hier gibt der Prozessor bzw. der Cache-Controller die nächste Adresse überlappend mit den Folgezugriffen des momentanen Blockbuszyklus aus, so daß sich die Datentransfers aufeinanderfolgender Blockzugriffe lückenlos aneinanderfügen lassen. Der Speicher, auf den die Blockbuszyklen wirken (Cache, Hauptspeicher) ist üblicherweise nicht wie oben beschrieben in Speicherbänken organisiert; vielmehr werden die aufeinanderfolgenden Blockzugriffe von den in den Speicherbausteinen (SRAM,

DRAM) vorhandenen Zugriffstechniken unterstützt. Man macht sich hier die Speicherung der Daten in größeren Zeilen eines Speicherfeldes und ggf. das Vorhandensein von bausteininternen Bänken zunutze (siehe 6.1.5, 6.1.6). – Man bezeichnet diese Technik des Address-Pipelining, da sie sich auf nur einen Prozessor bezieht, auch als *Intra-Prozessor-Pipelining*.

Geelter Buszyklus (split transaction). Die dem Address-Pipelining zugrundeliegende Trennung von Adreß- und Datenbus (*split bus*) kann in noch stärkerem Maße genutzt werden, indem man beide Busse jeweils einer eigenen *Busarbitrierung* unterzieht und indem man zusätzlich das Schachteln von Adreß- und Datenübertragungen aufeinanderfolgender Buszyklen ermöglicht. Dies erlaubt es, Buszyklen in anderer Reihenfolge abzuschließen, als sie begonnen wurden, wodurch sich Wartezeiten von Buszyklen durch Adreß- und Datentransporte anderer Buszyklen ausfüllen lassen. Diese Technik läßt sich sowohl bei Einprozessorsystemen als auch bei *symmetrischen Mehrprozessorsystemen* (SMP, 5.1.7) anwenden. Bild 6-12 zeigt dazu ein Beispiel für die Schachtelung dreier Buszyklen, wobei offen gelassen ist, ob diese von nur einem oder von mehreren Prozessoren initiiert wurden. Aufgrund der getrennten Adreßbus- und Datenbusverwaltung ergeben sich zwar längere Zugriffszeiten für den einzelnen Buszyklus, jedoch kann der Bus im Mittel besser ausgelastet werden. Bei Mehrprozessorsystemen besteht zusätzlich die Möglichkeit, den Prozessor, der den Adreßbus zuletzt belegt hat, auf diesem zu „*parken*“, so daß für ihn die Zeit für eine erneute Buszuweisung entfällt, sofern er nicht zuvor durch eine Busanforderung eines anderen Prozessors von diesem Parkplatz verdrängt wurde.

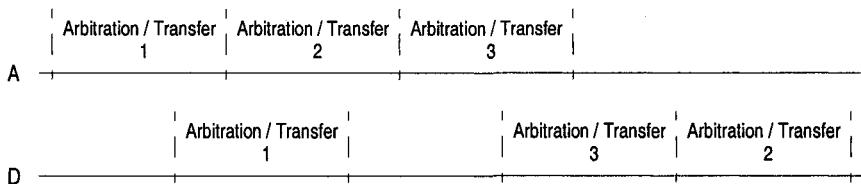


Bild 6-12. Zeitlicher Ablauf dreier Split-Transaktionen, initiiert in der Reihenfolge 1, 2, 3 durch entweder nur einen oder durch mehrere Prozessoren. Dabei getrennte Arbitrierung und Nutzung von Adreßbus A und Datenbus D

Die Möglichkeit von Split-Transaktionen erfordert zusätzlichen Hardwareaufwand, nicht nur im Prozessorbaustein, sondern auch prozessorextern. So muß der gemeinsame Speicher mit Pufferregistern versehen sein, um u.a. Adressen und Daten speichern zu können. Darüber hinaus muß die Busverwaltungslogik so ausgelegt sein, daß sie die korrekte Zuordnung von Datum und Adresse gewährleistet, auch dann, wenn die von mehreren Prozessoren durchgeföhrten Übertragungen mit unterschiedlichen Slaves stattfinden. Der dabei nötige Aufwand hängt davon ab, wie viele sich überlappende Transaktionen maximal möglich sein sollen.

Beispiele für Mikroprozessoren, mit deren Erscheinen diese Technik eingeführt wurde, sind der *PowerPC 601* [Motorola 1993] und dessen Nachfolger sowie der *Pentium Pro* [Intel 1996a] und einige seiner Nachfolger in der Pentium/Xeon-Familie. Zu nennen sind insbesondere auch die Athlon-Prozessoren von AMD. – Beim Einsatz dieser Technik in Einprozessorsystemen spricht man wie beim Address-Pipelining von *Intra-Prozessor-Pipelining*, beim Einsatz in Mehrprozessorsystemen von *Inter-Prozessor-Pipelining*.

6.1.5 Blockbuszyklen aus Prozessor- und Speichersicht

Prozessorsicht. In heutigen Mikroprozessorsystemen finden Datentransporte mit Speichern aufgrund der im Prozessor üblicherweise vorhandenen Befehls- und Daten-Caches nicht mehr als Einzelbuszyklen (single cycles), sondern fast ausschließlich als 4-Wort- oder 8-Wort-Blockübertragungen, d.h. als *Blockbuszyklen* (*burst cycles*) statt (5.3.3). Da bei einem solchen Zyklus die steuernde Einheit, d.h. der Prozessor (genauer der *Cache-Controller*) nur die Adresse des ersten Worttransports bereitstellen muß, können die drei Folgetransporte in kürzerer Zeit als der erste Transport erfolgen. Im bestmöglichen Fall des minimalen Buszyklus, d.h. ohne Wartezyklen, ergeben sich bei Single-Data-Rate Transportzeiten von 2 Takten für den ersten Transport und von jeweils nur einem Takt für die Folgetransporte (2-1-1-1-*burst* bei 4-Wortübertragung). Bei Double-Data-Rate fallen die Folgeübertragungen in halben Takschritten an (siehe 6.1.6). Die Adressierung der Folgetransporte unterliegt der Speichereinheit. Da die Blöcke bezüglich ihrer Adressen im Hauptspeicher ausgerichtet sind (*Block-Alignment*), beschränkt sich die Adreßfortzählung auf die Adreßbits A3 und A2 (32-Bit-Prozessor/Bus/Speicher) bzw. auf die Adreßbits A4 und A3 (64-Bit-Prozessor/Bus/Speicher), ist also einfach zu handhaben.

In einer Weiterentwicklung der Blockübertragungstechnik ist es heutigen Prozessoren möglich, aufeinanderfolgende Blockbuszyklen zu überlappen (*pipelined burst cycles*), wie dies in 6.1.4 ausführlich für Einzelbuszyklen beschrieben ist. Dabei legt der Prozessor die Folgeblockadresse bereits mit Abschluß der vorletzten Datenübertragung des vorangehenden Blockbuszyklus auf den Adreßbus. Aus Sicht des Prozessors entfällt hiermit der beim ersten Zugriff zusätzlich erforderliche Takt für die Adressierung, so daß zwei direkt aufeinanderfolgende Blockbuszyklen im bestmöglichen Fall zum 2-1-1-1-1-1-1-*Burst* werden. – Bild 6-13 zeigt für den Adreßbus (A) und den Datenbus (D) die Zeitverhältnisse für die drei Varianten Einzelbuszyklus, Blockbuszyklus und sich *überlappende Blockbuszyklen*, jeweils ohne Wartezyklen. Es zeigt ferner in den Belegungszuständen der Busse, daß Einzelbuszyklen und Blockbuszyklen mit gleichem Zeitverhalten auch auf Multiplexbussen stattfinden können, daß aber das Überlappen von Blockbuszyklen die hier gezeigte Trennung von Adreß- und Datenbus erfordert.

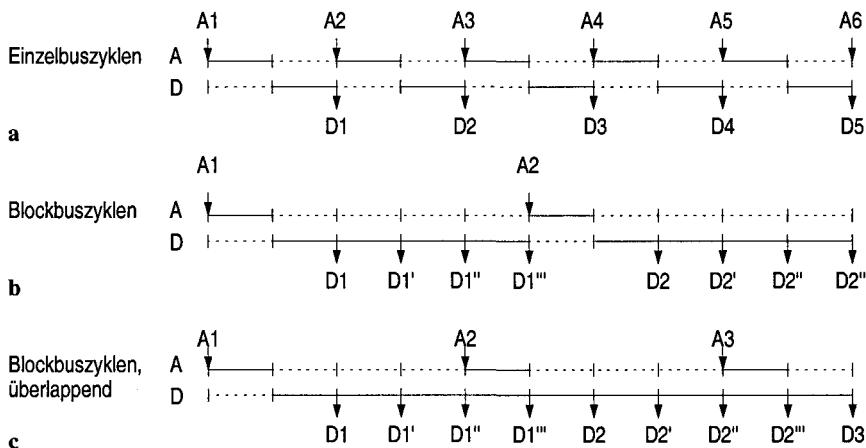


Bild 6-13. Schematische Darstellung der für den Prozessor/Cache kürzesten Folge von Adressen und Daten bei Übertragungen mit Single-Data-Rate. **a** Für vier direkt aufeinanderfolgende Einzelbuszyklen (single cycles), **b** für zwei direkt aufeinanderfolgende Blockbuszyklen (burst cycles), **c** für drei aufeinanderfolgende, sich überlappende Blockbuszyklen (pipelined burst cycles). Die Zuordnung von Daten zu Adressen ist durch die Numerierung gegeben, z.B. A1, D1; der Takt des Prozessorbusses ist durch die senkrechten Striche angedeutet. (Die durchgezogenen Linien zeigen an, in welchen Takten der Adreßbus A und der Datenbus B durch die Adreß- bzw. Datentransfers belegt sind.)

Spechersicht. Um dem Blockbuszyklus speicherseitig gerecht zu werden, müssen Folgezugriffe mit verkürzter Speicherzugriffszeit ausführbar sein. Dies betrifft neben den prozessorinternen *L1-Caches*, die dieser Anforderung durch eine schnelle Technologie entgegenkommen, insbesondere den mit DRAM-Bausteinen aufzubauenden Hauptspeicher. Es betrifft darüber hinaus aber auch einen ggf. vorhandenen und mit SRAM-Bausteinen aufzubauenden prozessorexternen *L2*- oder *L3-Cache* (siehe 6.2.1). Zur Lösung dieser Aufgabe stehen unterschiedliche Techniken zur Verfügung, die in einer Abwägung von Geschwindigkeit und Kosten eingesetzt werden. Dabei wird ggf. der bestmögliche Fall eines 2-1-1-1-Burst (Single-Data-Rate) nicht erreicht, und die Steuereinheit, die den Blockbuszyklus ausführt, muß Wartezyklen durchlaufen. Ein Blockzugriff auf den DRAM-Hauptspeicher ist dann z.B. nur mit einem 5-2-2-2-Burst oder mit noch längeren Zugriffszeiten möglich. Typisch ist jedoch immer die längere Zugriffszeit für den ersten Zugriff, den sog. *Lead-off-Cycle*, und die kürzere Zugriffszeit für die Folgezugriffe.

Zur speicherseitigen Unterstützung der Blockbuszyklen des Prozessors/Cache gibt es zwei grundsätzliche Vorgehensweisen:

- Aufbau eines strukturierten Speichers mit verschränkten Speicherbänken.
- Aufbau des Speichers mit DRAM- oder SRAM-Bausteinen, die den Blockzugriff durch bausteininterne Maßnahmen unterstützen.

Strukturierter Speicher mit verschränkten Speicherbänken. Zur Strukturierung des Speichers (Hauptspeicher oder L2-Cache) wird dieser gemäß den vier Wortübertragungen eines *Blockbuszyklus* in vier Speicherbänke mit wortbreitem Zugriff aufgeteilt und auf Wortbasis *verschränkt adressiert*. Das entspricht einer Bankanordnung wie in Bild 6-9 (S. 386), jedoch mit einer gesonderten Adresserzeugung für die Folgewörter und ohne das NARQ-Signal. Jede Speicherbank wird darüber hinaus mit einem Datenpufferregister ausgestattet, das es erlaubt, ein gelesenes Wort zwischenzuspeichern und es mit schnellem Zugriff auf dem Datenbus bereitzustellen. Ein Blocklesezyklus wirkt bei einer solchen Speicherseinheit dann auf alle vier Speicherbänke gleichzeitig, so daß nach der regulären Speicherzugriffszeit alle vier Wörter bereitstehen, wobei das erste zu übertragende Wort (am entsprechenden Datenpuffer vorbei) direkt auf den Datenbus gelegt wird. Die drei Folgezugriffe werden dann auf die Pufferregister der drei verbleibenden Speicherbänke ausgeführt. Der erste Zugriff (lead-off cycle) erfolgt also mit der Speicherzugriffszeit, die Folgezugriffe mit der entsprechend kürzeren Registerzugriffszeit.

Prinzip des Blockzugriffs bei DRAMs und SRAMs. Sämtliche bei DRAM- und SRAM-Bausteinen realisierten Blockzugriffstechniken basieren auf der feldartigen Anordnung der Speicherinformation, genauer, auf dem zeilenweisen Zugriff auf das Speicherfeld. Bei der Adressierung eines Speicherbausteins mit fortlaufenden Adressen wird das Speicherfeld (bei entsprechender Adressaufteilung in Zeilen- und Spaltenadresse) zunächst von links nach rechts, d.h. spaltenweise, und dann von oben nach unten, d.h. zeilenweise, durchlaufen. Speicherinhalte mit benachbarten Adressen liegen dementsprechend auch im *Speicherfeld* benachbart, nämlich in den Zeilen nebeneinander, abgesehen von den Wechseln zur jeweils nächsten Zeile. Für den *Blockbuszyklus* heißt das, daß dessen vier Zugriffe immer vier nebeneinanderliegende Bausteineneinträge betreffen, wobei diese Einträge aufgrund der Blockausrichtung (alignment) im Speicheradreßraum auch in der Zeile ausgerichtet sind, also keinen Zeilenwechsel verursachen.

Bild 6-14a zeigt die „Außensicht“ dieser Adressierung für eine 64-Mbyte-Speichereinheit, wie sie in Bild 6-7 (S. 381) mit Speicherbausteinen mit 4 Bit breitem Datenzugriff beschrieben ist. Bei jedem Wortzugriff des Blockbuszyklus wird also in jedem Speicherbaustein auf eine 4-Bit-Speicherstelle zugegriffen, und die in Bild 6-14a übereinander dargestellten Wortanteile liegen im Speicherfeld des einzelnen Bausteins innerhalb einer Zeile nebeneinander. Voraussetzung für diese Abbildung der Speicheradresse auf das Speicherfeld ist jedoch, daß die an den Speicherbaustein angelegte *Zeilenadresse* aus den höherwertigen Adressbits und die für das Weiterzählen im Block zuständige *Spaltenadresse* aus den niederwertigen Adressbits des Speicheradreßworts gebildet werden. – Bild 6-14b zeigt im Vorgriff die Adressaufteilung für eine 128-Mbyte-Speichereinheit, deren Speicherbausteine eine interne Bankstruktur haben, wie dies bei den in 6.1.6 vorgestellten SDRAMs der Fall ist (Bild 6-15, S. 397). Hier ist die Adressaufteilung derart, daß ein Bankwechsel immer dann erfolgt, wenn eine Zeile verlassen wird,

d.h., die Bankanwahlbits sind gedanklich den Bits der Spaltenadresse zuzuordnen. Auf diese Weise erhält man eine sog. lange Zeile (long row), indem die kürzeren Zeilen der Bänke aneinandergefügt werden.

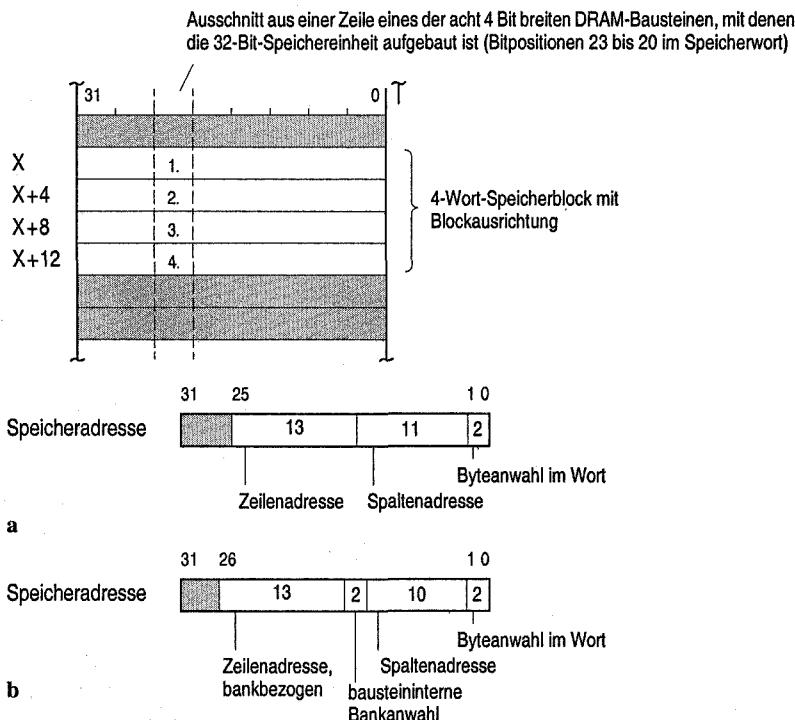


Bild 6-14. Bausteinbezogene Speicheradressierung. a 4-Wort-Blockzugriff auf eine 64-Mbyte-Speichereinheit, aufgebaut mit 8 DRAM-Bausteinen mit je $16\text{ M} \times 4\text{ Bit}$ entsprechend Bild 6-4 (S. 376). b Adressierung einer 128-Mbyte-Speichereinheit, aufgebaut mit 4 DDR-SDRAM-Bausteinen mit je $32\text{ M} \times 8\text{ Bit}$ entsprechend Bild 6-15 (S. 397), jeweils unterteilt in vier Bänke mit je 8 K Zeilen zu je 512 Spalten à 2 mal 8 Bit (d.h. 16 Bit breiter interner und 8 Bit breiter externer Datenzugriff, DDR!); Bankwechsel bei aufeinanderfolgenden Zeilenadressen. – Bei beiden Varianten bleibt die ZeilenAdresse (und bei b auch die Bankanwahl) während eines Blockbuszyklus fest, die SpaltenAdresse hingegen wird weitergezählt

Der Geschwindigkeitsvorteil bei Blockbuszyklen liegt nun darin, daß zwar für den ersten Zugriff (*lead-off cycle*) die vollständige Adressierung mit Zeilen- und Spaltenadresse durchlaufen werden muß, bei den Folgezugriffen jedoch nur noch die Spaltenadresse verändert wird und dementsprechend nur die Zugriffszeit auf die Spalte wirksam wird. Zur Erinnerung: Bei DRAMs wird mit der Zeilenadressierung die gesamte Zeile aus dem Speicherfeld in Flipflops (*Schreib-/Leseverstärker*) ausgelesen, d.h., in SRAM-Technik zwischengespeichert. Mit der Spaltenadressierung erfolgt dann nur noch die Übertragung der Datenbits zwischen den angewählten Flipflops und den Datenanschlüssen des Bausteins. Wie in Bild

6-6 (S. 379) gezeigt, beträgt die *Zugriffszeit* für den Erstzugriff $t_{RCD} + t_{CL}$, die Zugriffszeit für die Folgezugriffe jedoch nur noch t_{CL} (CAS-Latency). So lange, wie eine Zeile in den Schreib-/Leseverstärkern gespeichert ist (man sagt die Zeile ist „geöffnet“) können weitere Blockzugriffe auf diese Zeile erfolgen, mit dem Vorteil, daß auch schon der Erstzugriff nur die verkürzte Zugriffszeit t_{CL} hat.

6.1.6 Blockzugriffstechniken bei DRAMs und SRAMs

Für die Unterstützung von Blockzugriffen in Speicherbausteinen gibt es unterschiedliche Techniken, die zunächst einmal nach DRAMs und SRAMs zu trennen sind, was mit deren unterschiedlichen Arten der Datenspeicherung zusammenhängt. Unterschiede gibt es aber auch innerhalb dieser beiden Baustein-Kategorien. Wir wollen im folgenden zunächst in Kurzform die Blockzugriffstechniken asynchroner DRAMs betrachten und dann ausführlicher auf die Techniken synchroner DRAMs, insbesondere auf die der gebräuchlichen SDRAMs eingehen. Betrachtungen zu synchronen SRAMs schließen sich daran an.

Asynchrone DRAMs. Mit dem Aufkommen der Caches bei PCs wurden, ausgehend von den asynchronen DRAMs mit Einzelzugriff, wie in 6.1.1 beschrieben, Blockzugriffstechniken für diese Bausteine entwickelt, wobei die Asynchron-technik der Baustein-ansteuerung zunächst beibehalten wurde. Die Entwicklung führte hier vom sog. Nibble-Mode-DRAM bis zum EDO-DRAM. Gegenüber den heutigen synchronen DRAMs, mit den optionalen Blocklängen von 2, 4 oder 8 nebeneinanderliegenden „Wörtern“, gibt es bei den asynchronen DRAMs auch den Einzelzugriff, den wahlfreien Zugriff und den Zugriff mit quasi beliebiger Länge (innerhalb einer geöffneten Zeile), wie er z.B. bei DMA-Transfers und bei langen Blockbuszyklen vorkommt.

Nibble-Mode-DRAM. Der Nibble-Modus (Nibble: 4-Bit-Format) wurde als erste dieser Blockzugriffsarten realisiert. Er ist heute zwar nicht mehr gebräuchlich, zeigt aber die Unterstützung von 4-Wort-Blockbuszyklen in Reinform. Bei ihm werden die Zeilen durch bausteininterne Strukturierung in Einheiten von jeweils vier aufeinanderfolgenden Bits unterteilt (4-Bit-Alignment). Beim ersten Zugriff wird eines der Bits einer solchen Einheit in herkömmlicher Weise adressiert. Daran anschließend kann auf die drei restlichen Bits der Einheit in verkürzten Zyklen zugegriffen werden, indem die bereits anliegende Spaltenadresse beibehalten und lediglich das CAS-Signal für jeden Zugriff erneut aktiviert wird. Die Zugriffe erfolgen dabei in der 4-Bit-Einheit umlaufend (wrap around), was den Blockbuszyklus als Cache-Lademechanismus in folgender Weise unterstützt: Der Prozessor, der das Laden eines Cache-Blocks z.B. durch einen Lesefehlzugriff auf den Cache auslöst (cache read-miss), ist daran interessiert, das gewünschte Datum möglichst schnell zu erhalten. Dazu wird das betroffene Speicherwort unabhängig von seiner Position im Cache-Block als erstes adressiert und gelesen. Es wird dabei sowohl in den Cache geschrieben als auch unmittelbar (am Cache

vorbei) an den Prozessor weitergereicht. Die drei folgenden Wortübertragungen betreffen dann nur noch den Cache (siehe auch 6.2.2).

Page- und Fast-Page-Mode-DRAMs. Beim Page-Modus (*PM-DRAM*, Page steht synonym für eine Zeile/Row) kann auf die Bits einer Zeile wahlfrei zugegriffen werden, indem mit jedem Folgezugriff nicht nur das $\overline{\text{CAS}}$ -Signal erneut aktiviert, sondern zuvor auch eine neue Spaltenadresse angelegt wird. Das führt gegenüber dem Nibble-Modus zwar zu größeren Zugriffszeiten für die Folgezugriffe, bietet aber neben der Möglichkeit des wahlfreien Zugriffs auch den Vorteil, auf mehr als nur vier Bits aufeinanderfolgend zugreifen zu können. Die Anzahl der Zugriffe wird allerdings dadurch begrenzt, daß eine in die Schreib-/Leseverstärker ausgeselebene Zeile nach einer bestimmten Zeit wieder „geschlossen“ werden muß, um der Forderung des Auffrischens aller Zeilen innerhalb des für den Baustein vorgegebenen Refresh-Abstands nachkommen zu können. In einer Weiterentwicklung, bei der die Folgezugriffe gegenüber dem Page-Modus verkürzt sind, entstand das heute noch gebräuchliche Fast-Page-Mode-DRAM (*FPM-DRAM*) mit Zugriffszeiten von 60 ns für den ersten und von 40 ns für die Folgezugriffe (60/40 ns).

EDO- und Burst-EDO-DRAMs. Das EDO-DRAM (*extended data out*) ist im Prinzip ein FPM-DRAM, bei dem jedoch die interne Steuerung geringfügig verändert wurde, und zwar dahingehend, daß sich das Anlegen der nächsten Spaltenadresse mit dem momentanen Datenzugriff überlappt. Damit ergibt sich bei einer gleichen Zugriffszeit von 60 ns für den ersten Zugriff eine verkürzte Folgezugriffszeit von 25 ns (60/25 ns). Das Burst-EDO-DRAM (*BEDO-DRAM*) hingegen hat eine vom FPM-DRAM wesentlich abweichende Internsteuerung. Bei ihm wird nur die Spaltenadresse des ersten Zugriffs angelegt, und die Adressfortzählung erfolgt bausteinintern. Die Folgezugriffe werden mit jeweiligem Aktivieren und Deaktivieren des $\overline{\text{CAS}}$ -Signals „getaktet“. Die Zeit für den ersten Zugriff beträgt 52 ns, die für den Folgezugriff 15 ns (52/15 ns). – Da das BEDO-DRAM nur durch wenige Bridges (Chipsätze) unterstützt wurde, konnte es sich, anders als das EDO-DRAM, nicht auf dem Markt halten.

Synchrone DRAMs. Die bisher erwähnten DRAMs haben eine asynchrone Steuerung und deshalb eine Vielzahl an Zeitparametern, die das Ansteuern durch die Speichersteureinheit sehr aufwendig machen. Anders ist das bei den synchronen DRAMs. Hier werden alle zeitabhängigen Vorgänge mit dem Bustakt synchronisiert (CLK-Eingang), wodurch bausteinextern wie -intern Ansteuerzeiten verkürzt werden können. Dazu sind die vier wesentlichen Steuersignale ($\overline{\text{CS}}$, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, $\overline{\text{WE}}$) zu einem 4-Bit-Codewort zusammengefaßt, über das einige wenige Kommandos gebildet werden, mit denen die Bausteine gesteuert werden. SDRAMs sind außerdem (wie auch andere getaktete DRAMs) bausteinintern in zwei oder mehr *Bänke* mit jeweils eigenen Schreib-/Leseverstärkern unterteilt. Dadurch können Zugriffe auf verschiedene Bänke überlappend erfolgen, wodurch die Erholzeit (*precharge time*) einer einzelnen Bank verdeckt werden kann. Dies entspricht im Prinzip der in Abschnitt 6.1.3 beschriebenen Tech-

nik der Verschränkung von Speicherbänken. Anders als dort findet der Bankwechsel jedoch auf Zeilenbasis statt, was mit einer Aufteilung der Speicheradresse gemäß Bild 6-14b (S. 393) erreicht wird. – Synchrone DRAMs gibt es in folgenden Varianten:

SDRAM (synchronous DRAM). Das ursprüngliche synchrone DRAM überträgt die Daten mit *Single-Data-Rate* (SDR) und kam zunächst mit Kapazitäten von bis 64 Mbit, zwei Internbänken und mit einer Taktfrequenz von 100 MHz (10 ns) auf den Markt. Heutige SDRAMs sind mit 133 MHz (7,5 ns) bis 200 MHz (5 ns) getaktet, haben vier Internbänke und sind mit Kapazitäten von bis zu 1 Gbit erhältlich.

DDR-SDRAM. In der Weiterentwicklung der SDRAMs wurde die Übertragung der Daten mit *Double-Data-Rate* eingeführt. Die 4-Bank-Struktur wurde dabei beibehalten, und auch die anderen technischen Daten sind gleich denen des SDRAM: Taktung mit 133 MHz (DDR266) bis 200 MHz (DDR400), Speicherkapazitäten von bis zu 1 Gbit. Beiden Bausteinvarianten gemeinsam ist auch, daß die Übertragungsfrequenz gleich der Internfrequenz (core frequency) des Bausteins ist. Um dennoch beim DDR-SDRAM die gegenüber SDR erforderliche doppelte Anzahl an Daten bausteinseitig bereitzustellen/übernehmen zu können, erfolgen die Zugriffe auf die Speicherfelder (Internbänke) mit gegenüber dem Datenformat doppelter Breite (Zweifach-Prefetch). – Das DDR-SDRAM wird aufgrund seiner Weiterentwicklung zum DDR2-SDRAM inzwischen auch als DDR1-SDRAM bezeichnet.

Bild 6-15 zeigt die Struktur eines DDR-SDRAM-Bausteins (DDR1) mit vier Speicherfeldern (Internbänken) mit jeweils eigenen Schreib-/Leseverstärkern. Die Zugriffsbreite (das Datenformat) ist 8 Bit (DQ7-DQ0), die Internzugriffe erfolgen jedoch wegen der DDR-Übertragung mit 16 Bit. Übertragen wird *source-synchronous*, d.h., das Übertragungstaktsignal wird durch die jeweilige Datenquelle bereitgestellt (bidirectional data strobe, DQS). Beim Lesen ist dies der Speicherbaustein, beim Schreiben z.B. die Bridge, die dann auch den DRAM-Controller enthält. Für Schreibvorgänge gibt es ein Signal zur Datenmaskierung (data mask, DM), mit dem einzelne Daten innerhalb einer Blockübertragung vom Schreiben ausgeschlossen (maskiert) werden können.

DDR2-SDRAM. Die neueste SDRAM-Entwicklung ist das DDR2-SDRAM. Es arbeitet wie das DDR-SDRAM mit Double-Data-Rate, aber jetzt mit gegenüber der Internfrequenz doppelter Übertragungsfrequenz. Um die hier erforderliche erneute Verdoppelung der Anzahl an Daten zu erreichen, werden die Zugriffe auf die Speicherfelder mit vierfacher Breite des Datenformats ausgeführt (Vierfach-Prefetch). Üblich sind derzeit Übertragungsfrequenzen von 200 MHz (DDR2-400) und 267 MHz (DDR2-533) sowie Chipkapazitäten von bis zu 1 Gbit.

SDRAM-Kommandos. Die wichtigsten Kommandos zur Steuerung des Zugriffs auf synchrone DRAM-Bausteine sind:

- ACTIVE: Das Active-Kommando dient zum *Öffnen* einer Zeile einer Bank und wird zusammen mit der Zeilenadresse und den Bankanwahlbits angelegt. Es führt die Bankanwahl und Zeilenanwahl durch und liest die durch die Zeilenadresse adressierte Zeile in die Flipflops der Schreib-/Leseverstärker. Die Zeile bleibt dann so lange geöffnet, d.h. *aktiv*, bis ein *Precharging* für die Bank erfolgt. Soll durch das Active-Kommando eine andere Zeile in derselben Bank geöffnet werden, muß zuvor das Precharging stattfinden.
- READ, WRITE: Das Read- bzw. Write-Kommando dient zum Auslösen eines blockweisen Lese- bzw. Schreibzyklus bzgl. einer geöffneten Zeile einer Bank (burst read, burst write) und wird zusammen mit den Bankanwahlbits und der Spaltenadresse angelegt. Nach dem Buszyklus bleibt die Zeile geöffnet (aktiv), es sei denn, das Kommando hat den Zusatz AUTO PRECHARGE. Dann wird die Zeile nach Abschluß des Zugriffs „geschlossen“. Übermittelt

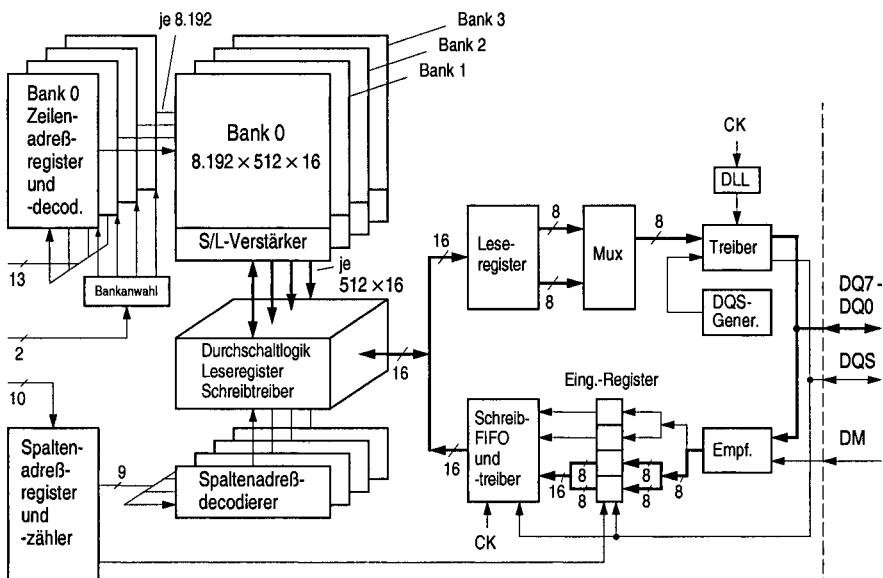


Bild 6-15. Struktur eines DDR-SDRAM-Bausteins in der Ausführung $32M \times 8\text{Bit}$ (256 Mbit; in Anlehnung an Datenblätter von Micron). Das Speicherfeld ist in vier Bänke mit je 8 K Zeilen zu je 512 Spalten à 2×8 Bit unterteilt. Für die Blockübertragungen mit Double-Data-Rate werden beim Lesezugriff im Taktabstand 16-Bit-Einheiten aus den Sense-Amplifiers gelesen (Leseregister) und diese dann als zwei 8-Bit-Einheiten an zwei aufeinanderfolgenden Taktflanken auf die Datenbusanschlüsse DQ7-DQ0 gelegt (Mux). Beim Schreibzugriff werden dementsprechend zunächst zwei 8-Bit-Einheiten gesammelt (Eingangsregister) und dann als 16-Bit-Einheit mittels eines FIFO-Puffers in das Speicherfeld übertragen. – Auf eine Vollständigkeit der Bausteinanschlüsse (Adress- und Kommandoanschlüsse) und auf einige Details der Steuerung (Adressmultiplexer, Refresh-Zähler, u.a.) wurde aus Platzgründen verzichtet. Signalwege ohne Angabe der Leitungsanzahl umfassen nur eine Leitung. Zur Aufteilung des Adressworts bzgl. der Bausteinadressierung siehe Bild 6-14b (S. 393)

wird der Zusatz über einen bei der Spaltenadressierung nicht benutzten Adressanschluß. – Aufeinanderfolgende Blockzugriffe auf dieselbe Zeile wirken bei READ auf die Flipflops der Schreib-/Leseverstärker und bei WRITE direkt auf das Speicherfeld, d.h. an den Flipflops vorbei (siehe 6.1.1).

- PRECHARGE: Das Precharge-Kommando dient zum *Schließen* der Zeile einer geöffneten Bank oder aller Bänke (Kommandozusatz ALL). Nach Absetzen dieses Kommandos ist ein erneutes Öffnen dieser *inaktiven* Bank bzw. Bänke erst nach Ablauf der Precharge-Time t_{RP} (Bild 6-6, S. 379) möglich. Die Unterscheidung, ob eine bestimmte oder alle Bänke geschlossen werden sollen, erfolgt wieder über den o.g. Adressanschluß. Ist nur eine Bank betroffen, so wird diese mittels der Bankanwahlbits angewählt. Um auf eine Bank zugreifen zu können, auf die zuletzt das Precharging wirkte (Precharge-Kommando oder Auto-Precharge-Zusatz), muß zuvor ein Active-Kommando auf diese Bank ausgeführt worden sein.
- BURST TERMINATE: Das Burst-Terminate-Kommando bricht das letzte Read-Kommando, sofern es keinen Auto-Precharge-Zusatz hat, ab. Das Abbrechen einer Zugriffsfolge (burst) kann auch dadurch erfolgen, daß ein Read-Kommando so kurz hinter dem aktuellen Kommando abgesetzt wird, daß sich die Datenzugriffstakte überlappen. Hier kommt dann das spätere Kommando zur Wirkung.
- NOP: Das Nop-Kommando hat keinen Einfluß auf die internen Abläufe. Es wird zur Überbrückung von Takten genutzt, in denen kein anderes Kommando an den Baustein angelegt wird.
- AUTO REFRESH: Das Auto-Refresh-Kommando führt das Auffrischen aller Zeilen durch, wobei die Zeilenadressierung durch einen bausteininternen Refresh-Controller erfolgt (entspricht dem CAS-before-RAS-Refresh asynchroner DRAMs, wie FPM- und EDO-DRAMs). Vor Absetzen dieses Kommandos muß ein Precharging für alle Bänke erfolgt sein.
- LOAD MODE REGISTER: Das Load-Mode-Register-Kommando erlaubt das Schreiben von Steuerinformation in ein Steuerregister. Damit läßt sich u.a. die Anzahl der aufeinanderfolgenden Datenzugriffe (burst length), 2, 4 oder 8, und die Zugriffszeit t_{CL} für die Folgezugriffe in der Anzahl an Takten, 2, 2,5 oder 3, festlegen (zu den Zeitparametern von SDRAMs siehe auch 6.1.2: DRAM-Module).

Zeitverhalten. Bild 6-16 zeigt das Signal-Zeitverhalten für zwei aufeinanderfolgende Lesekommandos, die im Speicherbaustein als 4-Word-Bursts programmiert sind und die auf dieselbe Bank und dieselbe Zeile wirken. Zum Taktzeitpunkt 0 wird die betreffende Bank durch das Precharge-Kommando für den Zugriff präpariert, indem die zuletzt geöffnete Zeile dieser Bank geschlossen wird. Dieser Vorgang benötigt hier zwei Takte (t_{RP}). Unmittelbar daran anschließend, d.h. im Taktschritt 2, wird dann mittels des Active-Kommandos und durch Anlegen der Zeilenadresse Row x und der Bankanwahlbits eine neue Zeile x dieser

Bank geöffnet, wofür in diesem Beispiel wiederum zwei Takte benötigt werden (t_{RCD}). Der eigentliche Lesezugriff wird dann im Taktschritt 4 mittels des Kommandos Read und durch Anlegen der Spaltenadresse Column a und der Bankwahlbits initiiert. Die bis zur Verfügbarkeit des ersten Datums benötigte Zeit t_{CL} beträgt hier ebenfalls zwei Takte. Sie ist im Mode-Register voreinstellbar. Übertragen werden danach vier Daten, und zwar an beiden Flanken des Taktes (DDR!). Ein zwei Takte später abgesetztes Read-Kommando auf dieselbe Zeile derselben Bank mit der Spaltenadresse b setzt den Datenfluß lückenlos fort. Wäre das zweite Read-Kommando unmittelbar nach dem ersten, also im Taktschritt 5 abgesetzt worden, hätte es die Zugriffsfolge des ersten Reads nach a1 unterbrochen, so daß die Zugriffsfolge a0, a1, b0, b1, b2, b3 entstanden wäre. – Die in Bild 6-16 angenommenen zwei Takte für jeden der drei Zeitparameter sind die derzeit günstigsten Werte. Je nach Baustein-ausführung können die Zeiten auch länger sein.

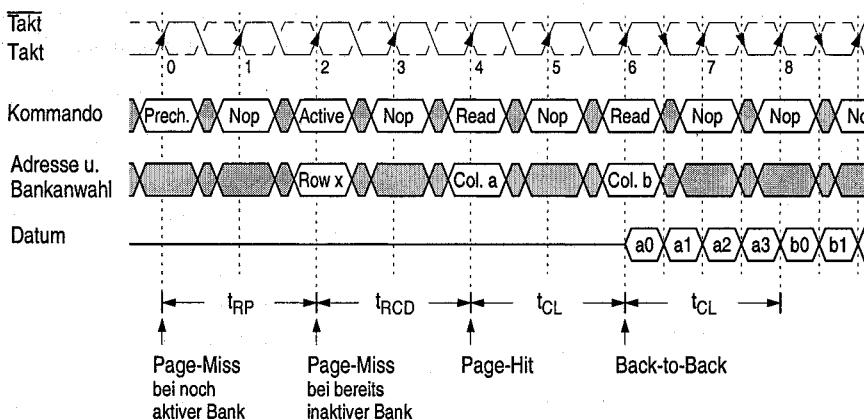


Bild 6-16. DDR-SDRAM-Lesezyklen als 4-Word-Bursts (in Anlehnung an Datenblätter von Micron), gesteuert durch die Kommandos „Precharge“ zum Voraufladen der Bitleitungen (t_{RP} : RAS-Precharge-Time), „Active“ zur Zeilen- und Bankadressierung (t_{RCD} : RAS-to-CAS-Delay) und „Read“ zur Spalten- und Bankadressierung und zum Lesen von vier Speicherwörtern (t_{CL} : CAS-Latency). Zur Datenübertragung mit beiden Taktflanken wird das Taktsignal durch das invertierte Taktsignal ergänzt. Zu den Zeitangaben siehe auch Bild 6-6 (S. 379)

Erstzugriffe (lead-off cycles). Steuereinheiten von DRAM-Speichern (DRAM-Controller) heutiger Rechnersysteme sind so ausgelegt, daß sie die Zugriffszeiten aufeinanderfolgender Blockbuszyklen optimieren. Sie führen dazu Buch über die momentan aktiven Bänke und merken sich die Adressen der geöffneten Zeilen. Auf dieser Basis setzen sie die für einen neuen Zugriff erforderlichen (situationsbedingten) Kommandos zeitoptimal ab. Das heißt, sie verdecken die für den Erstzugriff erforderliche längere Zugriffszeit bestmöglich. Diese Zeit hängt von der jeweiligen Zugriffs situation ab, wie nachfolgend beschrieben. – Bei den Situationsbezeichnungen wird der Begriff *Page* synonym zu *Row* verwendet. Die ange-

gebenen Taktzahlen beziehen sich auf die 4-Word-Read-Bursts und die Zeitangaben in Bild 6-16; dort sind auch die Zugriffssituationen markiert.

- *Page-Miss bei noch aktiver Bank:* Die (zum Blockzugriff gehörende) Adresse zeigt auf eine Speicherbank, in der eine andere Zeile geöffnet ist. Diese Zeile muß also zuerst geschlossen (Voraufladen der Bitleitungen) und dann die neue Zeile geöffnet werden. Der Lead-off-Cycle benötigt damit 6 Takte ($t_{RP} + t_{RCD} + t_{CL}$); der Zugriff insgesamt ist ein 6-1/2-1/2-1/2-Burst. Die benötigten Kommandos sind: Precharge, Active und Read.
- *Page-Miss bei bereits inaktiver Bank:* Die Adresse zeigt auf eine Speicherbank, in der die zuletzt geöffnete Zeile bereits geschlossen ist (das Voraufladen ist schon erfolgt). Der Lead-off-Cycle reduziert sich damit auf 4 Takte ($t_{RCD} + t_{CL}$); der Zugriff insgesamt ist ein 4-1/2-1/2-1/2-Burst. Die benötigten Kommandos sind Active und Read.
- *Page-Hit:* Die Adresse zeigt auf eine Speicherbank, in der die zuzugreifende Zeile bereits geöffnet ist. Der Lead-off-Cycle reduziert sich damit auf 2 Takte (t_{CL}); der Zugriff insgesamt ist ein 2-1/2-1/2-1/2-Burst. Das benötigte Kommando ist Read.
- *Back-to-Back:* Die Adresse zeigt auf eine Speicherbank, in der die zuzugreifende Zeile bereits geöffnet ist. Darüber hinaus überlappen sich die beiden Takte des Lead-off-Cycles mit den Datenübertragungen des vorangehenden Blockzugriffs, so daß die Datenübertragungen beider Zugriffe lückenlos aufeinander folgen können. Der Lead-off-Cycle reduziert sich damit auf quasi 0 Takte; die Blockzugriffsfolge ist ein 2-1/2-1/2-1/2-1/2-1/2-1/2-Burst, vorausgesetzt der erste Zugriff erfolgt mit Page-Hit. Das benötigte Kommando ist Read. – Diese Überlappung von Blockzugriffen findet sich bei den sich überlappenden Blockbuszyklen des Prozessors/Cache wieder (*pipelined burst cycles*, Bild 6-13c).

Die im Betrieb gemessenen Taktangaben für den Lead-off-Cycle sind üblicherweise höher als die obigen Angaben, da zusätzlich zu den durch die Speicherbausteine bedingten Zeitparameter Takte in der Speicheransteuerung „verbraucht“ werden.

Rambus-DRAM (RDRAM). Das RDRAM ist ebenfalls ein synchrones, d.h. getaktetes DRAM. Gegenüber den bisher betrachteten SDRAMs sieht es jedoch einen modifizierten Aufbau für die Speichereinheiten vor, indem hier nicht (wie z.B. bei SDRAM-DIMMs) mehrere Speicherbausteine nebeneinander mit z.B. 64-Bit-Speicherwort betrieben werden, sondern indem bis zu 32 dieser 16-Bit-breiten Bausteine an einem speziellen 16-Bit-Bus, dem sog. Rambus-Channel, parallel betrieben werden. Das heißt, der Speicherzugriff erfolgt lediglich in 16-Bit-Einheiten, und es bedarf einer Steuereinheit (memory controller, z.B. in der Bridge oder im Memory-Hub untergebracht), um den 16-Bit-Speicherweg an den 64-Bit-breiten Prozessordatenbus anzupassen. Neben dem Sammeln von

16-Bit-Einheiten zu 64-Bit-Wörtern bei Lesezugriffen und dem entsprechenden Verteilen bei Schreibzugriffen (Assembly- und Disassembly-Funktion) ist diese Steuereinheit auch für die Signalanpassung zwischen dem mit differentieller Signaldarstellung arbeitenden und mit bis zu 600 MHz (zzgl. DDR) hoch getakteten Rambus-Channel und dem mit bis zu 200 MHz (zzgl. quad pumped) getakteten Prozessorbus erforderlich. Sie kann darüber hinaus mehrere solcher Rambus-Channels verwalten, z.B. zwei, wobei jeder von diesen zu einer eigenständigen Speichereinheit gehört.

Die Ansteuerung der Speicherbausteine erfolgt wie bei den SDRAMs durch Kommandos. Übertragen wird die Information jedoch in Paketen, sog. Row-, Column-, und Data-Packets, die in jeweils acht aufeinanderfolgende Einzeltransfers aufgelöst werden, wobei die Transfers mit beiden Flanken des Taktsignals stattfinden (DDR). Row-Packets umfassen 24 Informationsbits und kommen aufgrund der acht Transfers mit nur drei Bausteinanschlüssen als Übertragungsweg aus. Benutzt werden sie für Kommandos wie Precharge und Active, wobei mit Active auch die Zeilenadresse übertragen wird. Column-Packets haben bis zu 40 Informationsbits und benötigen für ihre Übertragung 5 Bausteinanschlüsse. Sie werden für Kommandos wie Read und Write benutzt und beinhalten dementsprechend auch die Spaltenadresse. Data-Packets umfassen 16 Datenbytes gemäß der 16-Bit-Zugriffsbreite des Bausteins und sind die kleinste zugreifbare Dateneinheit. Als maximale Datenübertragungsrate erhält man bei lückenloser Aufeinanderfolge von Datenpaketen eine maximale Übertragungsrate von 2,4 Gbyte/s (Bustakt von 600 MHz und DDR). Unterstützt wird der lückenlose Zugriff durch eine Vielzahl interner Bänke, z.B. 32, wenn auch nur ein Teil der Bänke gleichzeitig geöffnet sein kann. Hier gibt es eine Einschränkung wegen der gemeinsamen Nutzung von Schreib-/Leseverstärker-Zeilen durch benachbarte Bänke).

Synchrone SRAMs. SRAM-Bausteine gibt es bezüglich ihres Zugriffsverhaltens in vielerlei Ausführungen. In herkömmlicher Bauweise arbeiten sie asynchron und sind für Einzelzugriffe ausgelegt, so wie in 6.1.1 beschrieben. Ihnen stehen heute getaktete, d.h. *synchrone SRAMs* gegenüber. Diese können durch die Taktung bestimmte, bei asynchroner Ansteuerung auftretende Signalverzögerungen vermeiden, was zunächst einmal die Zugriffszeit dieser SRAMs generell verkürzt. Darüber hinaus unterstützen sie Blockbuszyklen durch eine bausteininterne Adresszählung, entweder für 4-Wort- oder für 2-Wort-Übertragungen. Anders als bei DRAMs entfällt hier das „Öffnen“ und „Schließen“ von Zeilen, vielmehr wird direkt aus einer Zeile ausgelesen bzw. in sie hineingeschrieben, ohne diese insgesamt zwischenspeichern zu müssen. Die Zugriffsbreite (Speicherwort) ist üblicherweise 16 oder 32 Bit, zzgl. 2 bzw. 4 Zusatzbits, die z.B. als Paritätsbits genutzt werden können. – Von der Vielzahl an SRAM-Varianten seien folgende genannt:

SyncBurst-SRAMs. Als herkömmliche synchrone SRAMs übertragen sie ihre Daten mit *Single-Data-Rate*, wobei zwei Arten der Zugriffsoptimierung unterschieden werden: „flow-through“ und „pipelined“. Bild 6-17 zeigt deren Zugriffsver-

halten in den Teilbildern a und b für jeweils einen Einzellesezugriff mit der Adresse A1 und dem Datum D1, unmittelbar gefolgt von einem Blocklesezugriff mit der Adresse A2 und den Daten D2, D2', D2'' und D2''''. Das Pipelined-SRAM ist gegenüber dem Flow-through-SRAM mit einem Ausgangsregister ausgestattet, das aufeinanderfolgende Zugriffe durch Fließbandtechnik unterstützt. Dieses verlängert zwar, wie im Bild zu sehen, dessen Zugriffszeit um einen Takt (2-1-1-1-Burst gegenüber 1-1-1-1-Burst), es erlaubt jedoch eine höhere Taktfrequenz, wodurch sich die Blockzugriffszeiten angleichen. Pipelined-SRAMs werden derzeit mit bis zu 166 MHz, Flow-through-SRAMs mit bis zu 125 MHz getaktet.

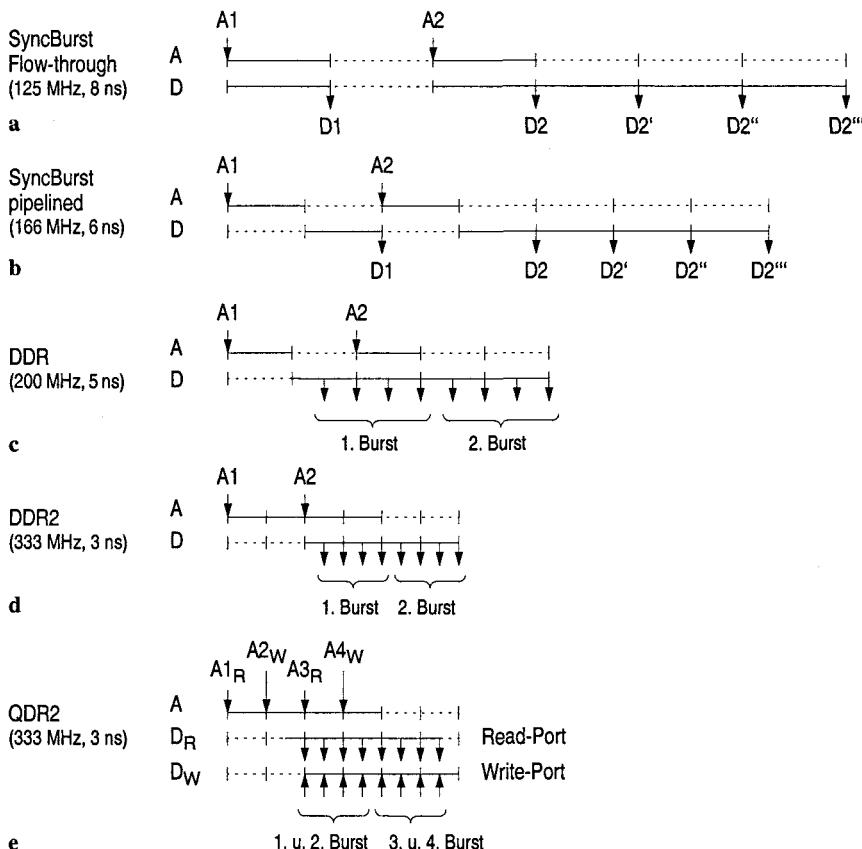


Bild 6-17. Maßstabsgerechte Darstellung der kürzest möglichen Datenfolgen für unterschiedliche synchrone SRAMs unter Nutzung der jeweils höchsten Taktfrequenz. **a** SyncBurst Flow-through-SRAM und **b** SyncBurst Pipelined-SRAM mit jeweils einem Einzelzugriff, direkt gefolgt von einem 4-Wort-Blockzugriff; **c** DDR-SRAM und **d** DDR2-SRAM, mit jeweils zwei direkt aufeinanderfolgenden 4-Wort-Blockzugriffen; **e** QDR2 mit je zwei 4-Wort-Lese- (A1, A3) und 4-Wort-Schreibzugriffen (A2, A4) (nach Datenblättern von Cypress). Die Takschritte sind durch senkrechte Striche angedeutet

DDR-SRAMs. Als die neueren synchronen SRAMs nutzen DDR-SRAMs sowohl die Fließbandtechnik als auch die Übertragung mit *Double-Data-Rate*; sie unterscheiden sich dabei in der Auslegung für 2-Wort- oder 4-Wort-Blockzugriffe. Bild 6-17c zeigt dazu ein Beispiel mit zwei direkt aufeinanderfolgenden 4-Wort-Zugriffen (back-to-back: 1 1/2-1/2-1/2-1/2-1/2-1/2-1/2-Burst). Gegenüber den Darstellungen in den Teilbildern a und b ist berücksichtigt, daß DDR-SRAMs gegenüber SyncBurst-SRAMs höher getaktet werden, nämlich mit bis zu 200 MHz. Die Bereitstellung zweier Wörter pro Takt wird wie bei den DDR-SDRAMs durch einen 2 Wort breiten Zugriff auf das Speicherfeld erreicht.

DDR2-SRAMs. Bei ihnen ist, wie beim DDR2-SDRAM, der Zugriff auf das Speicherfeld gegenüber den DDR-SRAMs nochmals in der Breite verdoppelt. Genutzt wird dies für eine bezüglich der Interaktionsfrequenz verdoppelten Übertragungstaktfrequenz mit derzeit bis zu 333 MHz (Bild 6-17d).

QDR- und QDR2-SRAMs. Sie basieren auf den Techniken der DDR- und DDR2-SRAMs, übertragen die Daten jedoch mit *Quad-Data-Rate*, indem sie getrennte Zugänge für das Lesen und das Schreiben vorsehen (read port, write port), die parallel genutzt werden können. Bei taktweise versetzten und sich abwechselnden Lese- und Schreibadressierungen können somit 4-Wort-Übertragungen sowohl auf den Datenleitungen des Read-Ports als auch auf denen des Write-Ports direkt aufeinanderfolgen (back-to-back, Bild 6-17e). Die Taktfrequenzen liegen hier bei bis zu 200 MHz (QDR) und 333 MHz (QDR2).

Zugriffsrate bei synchronen DRAMs/SRAMs. Als Maß für die Leistungsfähigkeit von DRAM- und SRAM-Speicherbausteinen und von Speichermodulen (DIMMs, Cache-Module) wird neben der Speicherkapazität die maximal mögliche Speicherzugriffsrate in Mbyte/s oder Gbyte/s angegeben. Sie ist (ebenso wie die maximale Übertragungsrate bei Bussen) eine idealisierte Größe, da ihr die bestmögliche Folge von Datenzugriffen zugrundegelegt wird, nämlich die auf den Lead-off-Cycle folgenden Zugriffe bei Blockbuszyklen unter Idealbedingungen. Sie wird entweder für die 64-Bit-Zugriffsbreite heutiger Speichereinheiten oder mehr bausteinspezifisch für eine 16-Bit-Zugriffsbreite angegeben. Tabelle 6-1 zeigt die maximalen Zugriffsichten von synchronen DRAMs und SRAMs im Hinblick auf die bei Prozessorbussen derzeit nutzbaren Taktfrequenzen von bis zu 200 MHz und deren Transferraten von bis zu 800 MT/s (quad pumped). Der durch die idealisierten Angaben teilweise entstehende Eindruck, SRAMs seien durch DRAMs ersetztbar, ist falsch, da DRAMs im Mittel erheblich mehr Takte für den Lead-off-Cycle als SRAMs benötigen. Dies hängt, wie oben beschrieben, mit Zugriffssituationen wie Page-Miss sowie mit dem zusätzlichen Bedarf an Taktzyklen in der Speichersteuereinheit zusammen. Die in Rechnersystemen für DRAMs gemessenen realen Zugriffsichten liegen dementsprechend erheblich unter den idealisierten Werten.

Tabelle 6-1. Idealierte maximale Speicherzugriffsrate synchroner DRAMs und synchroner SRAMs (nach Datenblättern von Micron und Cypress). Die für den Lead-off-Cycle benötigten Takte verringern die angegebenen Zugriffsraten, insbesondere bei den DRAMs

Speichertyp	Taktfrequenz MHz	max. 16-Bit- Zugriffsrate Mbyte/s	max. 64-Bit- Zugriffsrate Mbyte/s
DRAM:	SDRAM	133/200	266/400
	DDR-SDRAM	133/200	532/800
	DDR2-SDRAM	200/267	800/1066
	RDRAM (DDR)	600	2400 (1 Channel)
SRAM:	SyncBurst-SRAM (flow-through)	125	250
	SyncBurst-SRAM (pipelined)	166	333
	DDR-SRAM	200	800
	DDR2-SRAM	333	1328
	QDR-SRAM	200	1600
	QDR2-SRAM	333	2656

Anmerkung. Die maximale Speicherzugriffsrates wird häufig auch als *Speicherbandbreite* bezeichnet. Wir verwenden diesen Begriff nicht. Zur Begründung siehe die Anmerkung zum Begriff Busbandbreite in 5.1.4.

6.2 Caches

Caches sind Speicher mit schnellem Zugriff, die in einer ihrer wichtigsten Anwendungen als Pufferspeicher zwischen Hauptspeicher und Prozessor fungieren. Ihre Aufgabe besteht darin, die während einer Programmausführung jeweils aktuellen Hauptspeicherinhalte für Prozessorzugriffe als Kopien möglichst schnell zur Verfügung zu stellen bzw. Daten, die der Prozessor in den Speicher schreiben möchte, diesem möglichst schnell abzunehmen. Da ein solcher Speicher aus Kostengründen eine wesentlich geringere Kapazität als der Hauptspeicher hat, sind besondere Techniken für das Adressieren sowie Strategien für das Laden, Aktualisieren und Ersetzen seiner Inhalte erforderlich. Außerdem muß für *Datenkohärenz* gesorgt werden, so daß weder der Prozessor noch ein anderer Master auf veraltete Daten im Hauptspeicher oder Cache zugreifen können. Diese Vorgänge werden weitgehend von der Hardware gesteuert und müssen für die Software unsichtbar sein. Daraus erklärt sich auch die Bezeichnung Cache, die mit „Versteck“ oder „unterirdisches Depot“ zu übersetzen ist.

6.2.1 Systemstrukturen

L1-, L2- und L3-Cache. Caches zur Pufferung von Befehlen und Daten sind bei heutigen 32- und 64-Bit-Mikroprozessoren in den Prozessorbaustein als sog. *On-Chip*-Caches integriert. Sie sind in verschiedene Ebenen unterteilt:

chip-Caches integriert (Bild 6-18, Schnittstelle 1). Einfachere Mikroprozessoren hingegen haben solche On-chip-Caches nicht; bei ihnen müssen sie ggf. prozessorextern, d.h. als *Off-chip-Caches* aufgebaut werden (Schnittstelle 2). Ein prozessorinterner Cache bietet den Vorteil sehr kurzer Zugriffszeiten, wie sie die prozessorinternen Register haben (1 Prozessortakt). Voraussetzung hierfür ist eine begrenzte räumliche Ausdehnung auf dem Halbleitersubstrat, d.h. eine Begrenzung der Speicherkapazität. Sie liegt derzeit bei bis zu 64 Kbyte (größere On-chip-Caches benötigen mit ihren längeren Signalwegen ggf. mehrere Takte für den Zugriff). Ein prozessorexterner Cache wird nach Möglichkeit für den minimalen Buszyklus des Prozessors, d.h. für Zugriffe ohne Wartezyklen ausgelegt (2-1-1-1-burst). Für seinen Aufbau müssen dementsprechend schnelle SRAM-Bausteine eingesetzt werden. Bevorzugt werden hier *synchrone SRAMs* (6.1.6). Die Cache-Kapazität ist hier nur durch den vertretbaren Kostenaufwand begrenzt. Sie liegt heute bei bis zu 1 Mbyte und darüber.

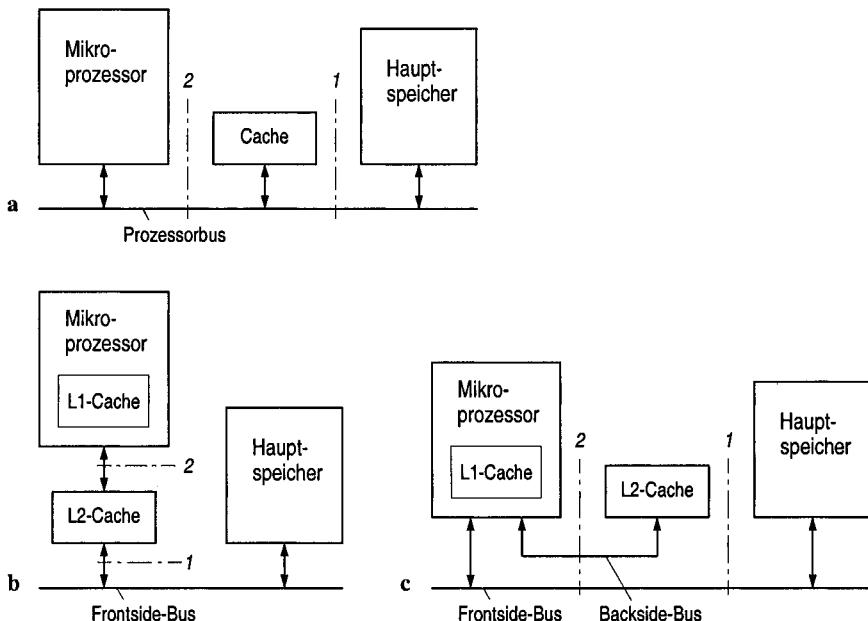


Bild 6-18. Caches als Pufferspeicher zwischen Mikroprozessor und Hauptspeicher. **a** Look-aside-Cache, **b** Look-through-Cache (hier z.B. als L2-Cache), **c** Look-through-Cache am Backside-Bus (grundsätzlich als L2-Cache). Anordnung entweder als On-chip-Cache (Schnittstelle 1) oder als Off-chip-Cache (Schnittstelle 2). Der Hauptspeicher ist ggf. mittels einer Bridge oder eines Hub mit dem Prozessor- bzw. Frontside-Bus verbunden (siehe 5.1.3: Bild 5-5 (S. 268) und 5.1.6)

Bei leistungsfähigen Mikroprozessorsystemen setzt man üblicherweise zusätzlich zu den in den Prozessorbausteinen vorhandenen On-chip-Caches auch noch Off-chip-Caches ein, um die Vorteile beider Anordnungen nutzen zu können. Das heißt, zwischen Prozessor und Hauptspeicher gibt es dann zwei Cache-Ebenen

mit unterschiedlichen Speicherkapazitäten und Zugriffszeiten. Dabei ist der prozessornahen Cache, der auch als *L1-Cache* (first-level cache, Primär-Cache) bezeichnet wird, üblicherweise für Befehle und Daten getrennt aufgebaut, d.h., es gibt sowohl einen *Befehls-Cache* als auch einen *Daten-Cache* (*split caches*). Der prozessorferne Cache, der sog. *L2-Cache* (second-level cache, Sekundär-Cache), wird hingegen meist als ein für Befehle und Daten gemeinsamer Cache, d. h. als *Befehls-/Daten-Cache* realisiert (*unified cache*). In Erweiterung dieser *Cache-Hierarchie* gibt es zusätzlich die Ebene des *L3-Cache* (third-level cache, Tertiär-Cache; als unified cache) mit gegenüber dem L2-Cache noch größeren Speicherkapazitäten von derzeit bis zu 6 Mbyte. Die Grenzziehung on-chip/off-chip in dieser Hierarchie ist nicht einheitlich. So ist bei den meisten Prozessoren der L2-Cache inzwischen on-chip, teils auch der L3-Cache (z.B. beim *Itanium 2*).

Anmerkungen. (1.) Bei On-chip-Caches der Ebenen L2 und L3 wird technisch unterschieden, ob die Caches zusammen mit dem Prozessorkern auf einem gemeinsamen Halbleitersubstrat (*on-die*) oder auf einem eigenen Halbleitersubstrat untergebracht sind (*dual-die*). Das hat Auswirkung auf die Zugriffszeiten, indem die Dual-die-Variante längere Signalwege als die On-die-Variante aufweist. (2.) Der Controller für einen prozessorexternen Cache (L2 oder L3) und ggf. die Speicherzellen für die Tag-Speicherung (tag memory) sind bei vielen Prozessoren in den Prozessorbaustein integriert, so daß prozessorextern nur die SRAM-Bausteine für die Datenspeicherung bereitgestellt werden müssen.

Die Speicherkapazität eines Cache ist ein wichtiger Parameter für seine Wirksamkeit, die sich in der *Trefferquote* (*hit ratio*), d.h. dem prozentualen Anteil der erfolgreichen Cache-Zugriffe an der Gesamtanzahl an Zugriffen ausdrückt. Eine hohe Trefferquote erhält man bei wiederholten Befehlszugriffen in Programmschleifen, die sich vollständig im Cache befinden, und bei Datenzugriffen, wenn das Programm über längere Zeitabschnitte mit einem begrenzten Umfang an Operanden arbeitet. Entscheidend ist hierbei die Verweildauer von Befehlen und Operanden im Cache, die sich mit der Cache-Kapazität erhöht, die aber auch von der Cache-Struktur und der damit zusammenhängenden Ersetzungsstrategie abhängt. Trefferquoten werden je nach Cache-Realisierung und Anwendung (Programm) im Bereich von 40 bis 95 % angegeben. Im ungünstigsten Fall, in dem sich die Cache-Inhalte laufend gegenseitig verdrängen und der Cache dabei seine Wirkung verliert, spricht man von *Trashing* (Abfall erzeugen). – Zur Dimensionierung von Caches siehe z.B. [Przybylski 1990], zu den gebräuchlichen Strukturen 6.2.3.

Cache-Hierarchien sind bzgl. ihrer *Ladestrategien* üblicherweise so ausgelegt, daß gleichzeitig mit dem Laden des kleineren Cache (L1) auch der größere Cache mit dieser Information geladen wird (L2). Wird dabei ein Eintrag aus dem kleineren Cache verdrängt, so bleibt er ggf. im größeren Cache aufgrund seiner größeren Speicherkapazität erhalten. Benötigt nun der Prozessor die im kleineren Cache verdrängte Information erneut, so kann sie vom größeren Cache geliefert werden, so daß ein Zugriff auf den langsameren Hauptspeicher entfällt. Insgesamt enthält der größere Cache also Information, die sich mit der im kleineren Cache deckt, sowie Information, die zuvor aus dem kleineren Cache verdrängt

wurde. Als davon abweichende Ladestrategie wird in den größeren Cache nur Information aufgenommen, die aus dem kleineren Cache verdrängt wird, um sie dort auch nur so lange zu halten, bis sie bei erneutem Bedarf in den kleineren Cache zurückgeladen wird (oder aus Gründen begrenzter Speicherkapazität verdrängt wird). Der Vorteil hierbei ist, daß sich beide Caches mit keinem ihrer Einträge überdecken und somit die beiden Cache-Kapazitäten besser genutzt werden. Man bezeichnet den größeren Cache hier auch als *Victim-Cache* (bei Prozessoren der Firma AMD gebräuchlich).

Look-aside- und Look-through-Cache. Bild 6-18a, S. 405, zeigt eine Systemstruktur, bei der der Cache „parallel“ zum Hauptspeicher am Prozessorbus betrieben wird. Sie ist typisch für einfache Mikrorechnersysteme, bei denen Speicherzugriffe fast ausschließlich vom Prozessor initiiert werden. Ein solcher Zugriff geht einerseits an den Cache, andererseits wird aber auch die Steuerung des Hauptspeichers aktiviert, um dessen Adressierung vorzubereiten. Kann der Cache die Anforderungen befriedigen, so wird der Hauptspeicherzugriff gestoppt; kann er sie nicht befriedigen, so ist der Speicherzugriff, da er ja vorsorglich mit dem Cache-Zugriff begonnen wurde, ohne Zeitverzögerung möglich. Nachteilig hierbei ist, daß der Hauptspeicher grundsätzlich, also auch bei erfolgreichen Cache-Zugriffen „belegt“ wird. Dadurch kommt es verstärkt zu Zugriffskonflikten mit anderen Masteern, z.B. DMA-Controllern. Man bezeichnet einen Cache in dieser parallelen Anordnung als *Look-aside-Cache*.

Bild 6-18b zeigt eine davon abweichende Struktur (hier bereits mit zwei Cache-Ebenen), bei der der Prozessor, der L2-Cache und der Hauptspeicher „in Reihe“ angeordnet sind. Sie ist typisch für symmetrische Mehrprozessorsysteme, bei denen ja mehrere Prozessor-Cache-Einheiten auf einen gemeinsamen Hauptspeicher zugreifen (5.1.7), aber auch für einfache Multimastersysteme. Hierbei werden die Zugriffsanforderungen der Prozessoren von ihren L2-Caches abgefangen und von der jeweiligen *Cache-Steuereinheit (Cache-Controller)* nur dann über den Bus an den Hauptspeicher weitergeleitet, wenn sie vom Cache nicht befriedigt werden können. Auf diese Weise werden die Speicheraktivitäten minimiert, um den Bus für die wirklich erforderlichen Hauptspeicherzugriffe des einen oder des anderen Prozessors oder auch für einen DMA-Controller freizuhalten. Nachteilig dabei ist, daß ein Hauptspeicherzugriff, wenn er dann erforderlich ist, aufgrund des zuvor versuchten Cache-Zugriffs verzögert aktiviert wird. Man bezeichnet diese Art von Cache als *Look-through-Cache* oder, auf die Reihenschaltung bezugnehmend, als *Inline-Cache*. Sie ist abweichend von der Darstellung in Teilbild b nicht auf L2-Caches beschränkt, sondern genauso bei L1-Caches üblich.

Look-aside- und Look-through-Caches unterscheiden sich nicht nur in ihrer Anordnung im Gesamtsystem und der damit einhergehenden „logischen“ Funktion, sondern auch „physikalisch“, nämlich in der Geschwindigkeit, mit der auf sie zugegriffen werden kann. So wird die Übertragungsrate für den Look-aside-Cache

durch die Taktfrequenz des Prozessorbusses begrenzt. Diese liegt derzeit bei maximal $200\text{ MHz} \times 4$ (quad data rate, quad pumped) und ist, verglichen mit der prozessorinternen Taktfrequenz von bis zu 3 GHz und darüber, relativ niedrig. Beim Look-through-Cache hingegen, bei dem der Cache (genauer der Cache-Controller) den Prozessorbus bildet, kann an der Schnittstelle Prozessor/Cache aufgrund der engen und ausschließlichen Cache-Anbindung an den Prozessor mit höherer Taktfrequenz übertragen werden.

Noch höhere Taktfrequenzen bis hin zur prozessorinternen Taktfrequenz erlaubt ein gesonderter Cache-Anschluß des Prozessors, wie in Bild 6-18c, S. 405, gezeigt. Dieser ist als „schnelle“ Punkt-zu-Punkt-Verbindung ausgelegt und ist ggf. breiter als der Prozessorbus, was die Übertragungsrate zusätzlich erhöht. Man bezeichnet ihn als *Backside-Bus* (BSB) und den Cache, der üblicherweise für den Look-through-Zugriff ausgelegt und ein L2- oder L3-Cache ist, als *Backside-Cache*. Der eigentliche Prozessorbus wird dann als *Frontside-Bus* (FSB) und ein an ihn angeschlossener Cache (z.B. der L2-Cache in Teilbild b) als *Frontside-Cache* bezeichnet.

Beispiel 6.2. ► *Itanium 2*. Der Itanium 2 (Intel) liefert ein Beispiel für eine Cache-Hierarchie mit drei Ebenen. Wie bereits in Beispiel 2.11 beschrieben und dort in Bild 2-39 (S. 146) dargestellt umfaßt er zwei für Befehle und Daten getrennte L1-Caches sowie je einen für Befehle und Daten gemeinsamen L2- und L3-Cache. Alle Caches sind on-chip und über 256 Bit breite Datenwege miteinander verbunden; die Datenverbindung zum Hauptspeicher ist 128 Bit breit. Hinsichtlich der Datenzugriffe gibt es die Besonderheit, daß für Gleitpunktzahlen der L2-Cache der prozessornächste Cache ist. Das heißt, im L1-Cache werden zwar Ganzahlen (Integers) gespeichert, aber keine Gleitpunktzahlen. Der Grund für diese Maßnahme ist eine erhöhte Parallelität beim Laden der beiden Registersätze und damit eine schnellere Versorgung der Funktionseinheiten mit Operanden. Bild 6-19 stellt diese Besonderheit dar.

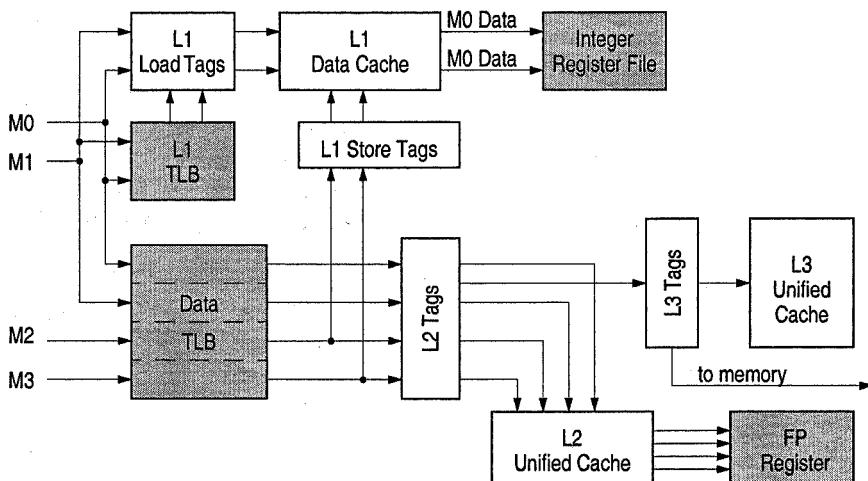


Bild 6-19. Cache-Struktur des Itanium 2 (nach [Hewlett Packard 2002])

Insgesamt sind vier Cache-Zugriffe auf Ganzzahlen oder Gleitpunktzahlen gleichzeitig möglich (M0 bis M3), wozu sowohl der L1-Cache als auch der L2-Cache als Vier-Port-Caches ausgelegt sind. Beim L1-Cache sind dadurch pro Zyklus bis zu zwei Lade- und zwei Speichere-Operationen mit den Ganzzahlregistern (Integer Register File) möglich. Beim L2-Cache sind es pro Zyklus bis zu vier Gleitpunkt-Ladeoperationen mit dem Gleitpunkt-Registersatz (FP Register File), so daß pro Zyklus bis zu zwei Gleitpunktoperationen mit Daten versorgt werden können. Diesen Cache-Zugriffen gehen Adreßumsetzungen durch die Speicherverwaltungseinheit voraus, wofür zwei Adreßumsetzungs-Caches (TLB, translation look-aside buffer) zur Beschleunigung eingesetzt werden. Die Adreßumsetzungen für die Ganzzahl-Ladeoperationen erfolgen dabei einmal gesondert über den sog. L1-TLB und dann nochmal zusammen mit den Umsetzungen für die anderen Operationen über den sog. Data-TLB. Die vom diesem erzeugten Adressen werden automatisch an den L2-Cache weitergereicht, so auch die für den L1-Cache bestimmten Adressen. Auf diese Weise kann der L2-Cache schon aktiviert werden, bevor das Zugriffsergebnis des L1-Cache vorliegt. Bei einer Fehlanzeige des L2-Cache werden die Adressen an den L3-Cache weitergeleitet. ▲

Speicherhierarchie. Bild 6-20 zeigt zusammenfassend die gesamte Speicherhierarchie bei zwei Cache-Ebenen mit den einzelnen Übertragungswegen und den für sie typischen Übertragungsmodi. (In der ersten Cache-Ebene ist nur der Daten-Cache gezeigt, nicht aber der Befehls-Cache.) Zwischen den Prozessorregistern und dem On-chip-L1-Daten-Cache finden Einzelübertragungen in den vom Prozessor durch Lade- und Speichere-Befehlen angesprochenen Datenformaten statt. Die Zugriffszeit beträgt üblicherweise einen Prozessortaktschritt, wobei die Prozessortaktfrequenz ggf. um ein Mehrfaches höher ist als die prozessorexternen Bustaktfrequenzen. Zwischen dem L1-Cache und dem aus SRAM-Bausteinen aufgebauten prozessorexternen L2-Cache, also an der Prozessorschnittstelle, finden 4-Wort- oder 8-Wort-Blockübertragungen statt, die vom L1-Cache-Controller gesteuert werden. Die bestmögliche Übertragungsrate wird hier durch den kürzest möglichen Blockbuszyklus bestimmt, z.B. durch $1\frac{1}{2}-1\frac{1}{2}-1\frac{1}{2}-1\frac{1}{2}$ -Bursts bei DDR-SRAMs (siehe 6.1.6). Voraussetzung hierfür ist, daß die Taktfrequenz der SRAM-Bausteine der Taktfrequenz an der Schnittstelle des Prozessorbausteins genügt; wenn nicht, sind die Übertragungen mit Wartezyklen behaftet.

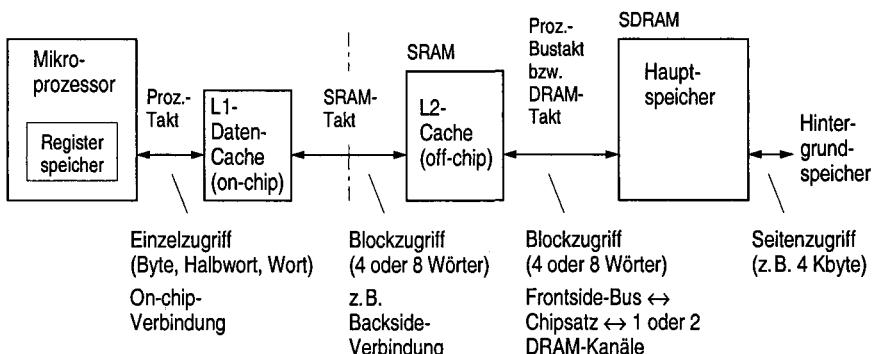


Bild 6-20. Speicherhierarchie mit den Übertragungsmodi Einzel-, Block- und Seitenzugriff für die verschiedenen Übertragungswege (der L1-Befehls-Cache ist hier nicht gezeigt)

Die Übertragungen zwischen dem L2-Cache und dem DRAM-Hauptspeicher werden vom L2-Cache-Controller gesteuert und finden ebenfalls als 4-Wort- oder 8-Wort-Übertragungen statt. Bei Verwendung von DDR-SDRAMs und unter der Annahme von Page-Hit-Zugriffen als bestmöglichem Fall sind 2-1/2-1/2-1/2-Bursts möglich, basierend auf der Taktfrequenz des Frontside-Busses. Page-Miss- und Row-Miss-Zugriffe sowie zusätzliche Wartezyklen, die der DRAM-Controller fordert, führen jedoch zu Lead-off-Cycles von mehr als zwei Takten und zu Folgezugriffen von mehr als nur einem halben Takt. – Im Bild mit angegeben ist die Verbindung zwischen Hauptspeicher und Hintergrundspeicher, auf die in 6.3 näher eingegangen wird.

Anmerkung. Innerhalb der Cache-Hierarchie, L1, L2 und L3, kann die Blockgröße variieren. Für den einzelnen Cache ergibt sie sich aus der Breite der Datenverbindung, z.B. vier, acht oder 16 Byte, und durch die Anzahl an Übertragungen pro Burst, nämlich vier oder acht. Die Anzahl an Übertragungen ist entweder durch die Cache-Organisation fest vorgegeben oder wählbar.

Cache-Anwendungen. Der Einsatz von Caches ist nicht auf die Pufferung von Befehlen und Daten des Hauptspeichers beschränkt. Caches werden insbesondere auch zur Pufferung von Deskriptoren bei Speicherverwaltungseinheiten eingesetzt, um Zugriffe auf im Hauptspeicher stehende Deskriptortabellen nach Möglichkeit zu vermeiden (*translation look-aside buffer*, TLB, siehe 6.3). Weiterhin gibt es bei Prozessoren mit spekulativer Befehlsausführung Caches zur Unterstützung der Sprungvorhersage (*branch prediction cache*) und der Sprungzielvorhersage (*branch target address cache*, siehe 2.3.6). Darüber hinaus werden Caches aber auch bei Hintergrundspeichern eingesetzt, z.B. bei Festplattenspeichern, CD-, DVD- und Streamer-Laufwerken, um die Übertragungsraten mit diesen Speichern zu erhöhen. – Die folgenden Betrachtungen nehmen von dieser Vielfalt Abstand und beziehen sich auf Caches zur Speicherung von Daten und Befehlen. Dabei ist zu beachten, daß Angaben zu Schreibzugriffen des Prozessors üblicherweise nur die Daten in einem Cache betreffen. Schreibzugriffe des Prozessors auf den Befehls-Cache sind nur zu berücksichtigen, wenn selbstmodifizierender Code erlaubt ist (eine vieldiskutierte Technik aus den Anfängen des Rechnerbaus, die inzwischen in den Hintergrund getreten ist).

6.2.2 Laden des Cache

Zu Beginn der Initialisierungsphase eines Mikroprozessorsystems sind die im System vorhanden Caches „abgeschaltet“, so daß sich die beim Hochfahren des Systems zufällig im Cache einstellenden Inhalte nicht negativ auswirken können. Das Ab- und Anschalten kann per Software über Registerbits gesteuert werden; beim Starten des Systems werden diese jedoch üblicherweise automatisch, d.h. durch die Hardware initialisiert. Bevor ein Cache nun per Software angeschaltet wird, müssen die immer noch falschen Einträge „gelöscht“ werden. Dies geschieht (wenn nicht bereits automatisch beim Starten des Systems) durch einen

Cache-Clear-Befehl, der das für jede Cache-Zeile vorhandene sog. *Valid-Bit* in allen Cache-Zeilen auf „invalid“ setzt. Danach gilt der Cache als „leer“.

Speicherzugriffe des Prozessors werden von der Cache-Steuereinheit aufgrund der „leeren“ Zeilen zunächst als Cache-Fehlzugriffe ausgewiesen (*cache-miss*). Handelt es sich dabei um einen Lesezugriff (read-miss), so wird der 4-Wort- oder 8-Wort-Block, der das Datum oder den Befehl enthält, aus dem Hauptspeicher gelesen, in den Cache geladen und dort als „gültig“ gekennzeichnet. Gleichzeitig wird die Cache-Zeile mit der erforderlichen Adressinformation geladen, um den Block bei späteren Zugriffen im Cache lokalisieren zu können. Handelt es sich um einen Schreibzugriff (write-miss), so hängt es von der für den Cache gewählten Aktualisierungsstrategie ab, ob der Cache zunächst mit dem Block geladen und dann dieser Block mit dem Datum aktualisiert wird (*write allocation*), oder ob der Hauptspeicher aktualisiert wird und der Cache unverändert bleibt (*no write allocation*, siehe 6.2.4). Wird in der Folge ein vom Prozessor adressierter Speicherblock als im Cache vorhanden erkannt (Adressinformation vorhanden und Valid-Bit gesetzt), so wird dies vom Cache-Controller als Cache-Treffer (read- bzw. write-hit, *cache-hit*) angezeigt und der Zugriff auf den Cache-Speicher ausgeführt.

Das Laden von Speicherinhalten in den Cache (*Cache-fill-Operation*) geschieht, wie beschrieben, blockweise mit einer Blockgröße von entweder vier oder acht Wörtern. Die Blockgrenzen im Speicher sind dabei an den Vielfachen der Blocklänge festgelegt, d.h. sie sind ausgerichtet (*block alignment*). Muß also ein Datum (oder Befehl) in den Cache geladen werden, so wird dieses Laden immer für den gesamten Block durchgeführt, in dem das Datum steht. Um den Prozessor dabei möglichst schnell mit dem aus dem Speicher kommenden Datum zu versorgen, auch wenn es nicht am Blockanfang steht, wird das entsprechende Wort als erstes gelesen und dann die restlichen Wörter mit umlaufender Adressierung (*wrap around*) nachgeladen. Bei einem Zugriff auf ein nichtausgerichtetes Datum (*data misalignment*) muß, wenn die Blockgrenze überschritten wird, auch der Folgeblock in den Cache geladen werden.

Das Ausrichten der Blöcke gewährt folgende Vorteile:

1. Die Blöcke schließen im Speicher überlappungsfrei aneinander an.
2. Die umlaufende Adressierung innerhalb eines Blocks lässt sich leicht realisieren. Wie eingangs zu 6.1.5 bereits gesagt, betrifft sie lediglich die Adressbits A3 und A2 (32-Bit-Datenverbindung) oder A4 und A3 (64-Bit-Datenverbindung).
3. Blockzugriffe erfolgen immer innerhalb der durch den ersten Zugriff angesprochenen Zeile des Speicherbausteins, wodurch die schnellen Folgezugriffe bei Blockbuszyklen überhaupt erst ermöglicht werden (6.1.6).

4. Blockzugriffe erfolgen immer innerhalb von Seitengrenzen, so daß während des Ladens eines Blocks kein Seitenwechsel auftreten kann (zur Seitenverwaltung siehe 6.3.2).

6.2.3 Cache-Strukturen

Charakteristisch für den Aufbau eines Cache ist, in welcher Weise die vom Prozessor an den Hauptspeicher ausgegebenen Adressen auf den relativ kleinen Specherraum des Cache abgebildet werden. Hier gibt es verschiedene Techniken, die sich zum einen im Hardwareaufwand für die Adreßauswertung und zum andern in der Flexibilität der Zuordnung von Blöcken zu *Cache-Zeilen* (cache lines) und damit auch in der Blockersetzungsstrategie unterscheiden. Um die drei wichtigsten Cache-Strukturen vergleichbar darzustellen, gehen wir von folgenden einheitlichen Vorgaben aus:

- Cache-Kapazität von 32 Kbyte, wie für heutige On-chip-L1-Caches üblich,
- Unterteilung in 2048 Zeilen zu je vier 32-Bit-Wörtern (Blockgröße von 16 Bytes),
- Adressierung mit 32-Bit-Hauptspeicheradressen.

Jeder der Cache-Strukturen ist pro Cache-Zeile ein *Valid-Bit* *V* und ein sog. *Dirty-Bit* *D* zugeordnet. Letzteres unterstützt die Aktualisierungsstrategie *Copy-Back* (siehe 6.2.4).

Vollassoziativer Cache. Beim vollassoziativen Cache (fully associative cache) ist das Zuordnen von Speicherblöcken zu Cache-Zeilen wahlfrei, d.h., jeder Block des Hauptspeichers kann in jede beliebige Zeile des Cache geladen werden. Um hierbei den Inhalt einer Cache-Zeile eindeutig zu kennzeichnen, muß zusätzlich zum Block dessen Blockadresse als sog. Etikett (*tag*) gespeichert werden. In unserem Beispiel sind das die höherwertigen 28 Bits der Hauptspeicheradresse (Bild 6-21). – Adressiert wird der Cache, indem die 28 höherwertigen Bits der vom Prozessor ausgegebene Speicheradresse mit allen im Cache gespeicherten Tags gleichzeitig verglichen werden. Zeigt einer der Vergleiche einen Treffer an (*tag hit*) und ist außerdem das Valid-Bit der so adressierten Zeile gesetzt (*valid hit*), so liegt ein Cache-Hit vor, und der Cache-Zugriff kann durchgeführt werden. Die verbleibenden Adreßbits A3 und A2 werden dabei zur Wortanwahl in der adressierten Zeile, A1 und A0 zur Byteanwahl im Wort herangezogen.

Die große Flexibilität in der Zuordnung von Blöcken zu Cache-Zeilen wird durch einen hohen Hardwareaufwand erkauft. So ist für jede Cache-Zeile ein 28-Bit-Vergleicher in Hardware vorzusehen, des weiteren muß eine *Ersetzungsstrategie* realisiert werden, die bei gefülltem Cache entscheidet, welcher der Cache-Blöcke beim Laden eines neuen Blocks ersetzt werden soll. Die bekannteste, aber nicht

in jeder Situation immer beste Ersetzungsstrategie ist die *Least-recently-used-Strategie (LRU)*, bei der jeweils jener Block im Cache überschrieben wird, dessen letzter Zugriff zeitlich am weitesten zurückliegt. Dazu wird von der Cache-Hardware für jede Zeile eine Alterungsinformation gespeichert, und die Alterungsinformation aller Zeilen wird mit jedem Cache-Zugriff aktualisiert. In einer vereinfachten Hardwarerealisierung mit nicht exaktem LRU-Verhalten wird diese Ersetzungsstrategie als *Pseudo-LRU-Strategie* bezeichnet (siehe dazu weiter unten; zu Ersetzungsstrategien umfassend siehe [Liebig 2003]).

Aufgrund dieses hohen Hardwareaufwands gibt es vollassoziative Caches nur mit wesentlich geringeren Kapazitäten, als in den obigen Vorgaben für einen L1-Cache angenommen. Sie kommen deshalb heute als Befehls- oder Daten-Caches nicht mehr vor. Es gibt sie jedoch als *Deskriptor-Caches* in Speicherverwaltungseinheiten (*TLBs*) mit z.B. 32 oder 64 Cache-Zeilen. Größere Cache-Kapazitäten werden mit teilassoziativen Cache-Strukturen realisiert, wie nachfolgend beschrieben.

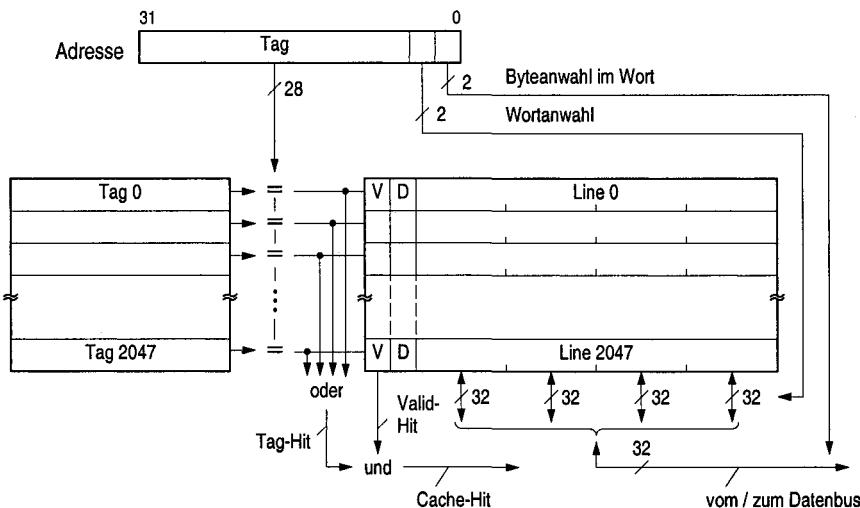


Bild 6-21. Adressierung eines vollassoziativen Cache mit 2048 Zeilen (Lines) zu je vier 32-Bit-Wörtern. Paralleler Tag-Vergleich für alle 2048 Zeilen durch 2048 Vergleicher. Cache-Hit bei Tag-Gleichheit (tag-hit) und gültigem Valid-Bit (valid-hit) für eine der 2048 Zeilen

Direkt zuordnender Cache. Beim direkt zuordnenden Cache (*direct mapped cache*) wird jeder Block des Hauptspeichers einer bestimmten Zeile im Cache fest zugeordnet, indem bei 2^n Cache-Zeilen die n niederwertigen Bits der Blockadresse als Index für die Zeilenanwahl herangezogen werden. Um einen Cache-Eintrag eindeutig zu kennzeichnen, müssen zusätzlich zum Block die verbleibenden höherwertigen Bits der Blockadresse als Tag gespeichert werden. In unserem Beispiel mit 2048 Cache-Zeilen sind das 11 Bits für den Zeilenindex und 17 ver-

bleibende Bits für das Tag (Bild 6-22). – Adressiert wird der Cache, indem der Zeilenindex mittels eines Decodierers ausgewertet und der Tag-Eintrag der dadurch ausgewählten Zeile mit den Tag-Bits der Hauptspeicheradresse verglichen wird. Zeigt der Vergleich einen Treffer an (tag hit) und ist außerdem das Valid-Bit der Zeile gesetzt (valid hit), so liegt ein Cache-Hit vor.

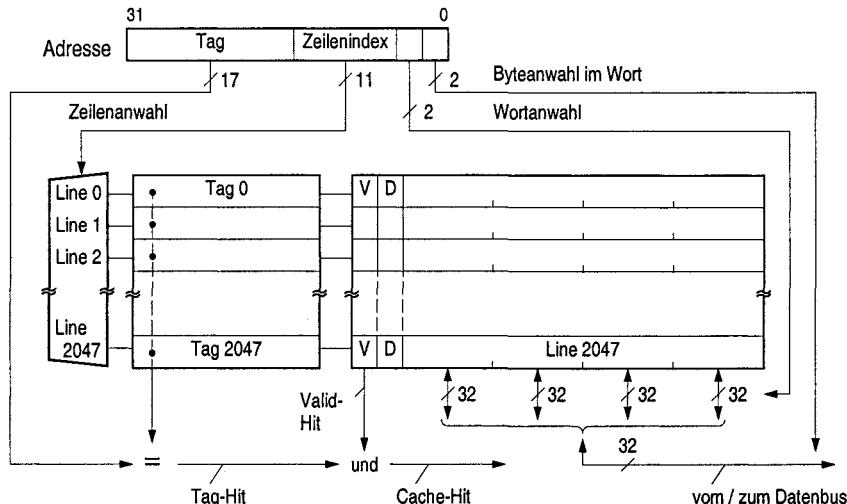


Bild 6-22. Adressierung eines direkt zuordnenden Cache mit 2048 Zeilen (Lines) zu je vier 32-Bit-Wörtern. Anwahl einer Zeile durch einen 11-Bit-Zeilenindex mittels eines Decodierers; Tag-Vergleich für diese Zeile durch einen einzigen Vergleicher. Cache-Hit bei Tag-Gleichheit (tag-hit) und gültigem Valid-Bit (valid-hit) für diese Zeile

Der Vorteil des direkt zuordnenden Cache ist, daß er unabhängig von der geforderten Speicherkapazität einfach zu realisieren ist, nämlich mit einem RAM. Zusätzlich benötigt er – unabhängig von der Zeilenanzahl – nur einen einzigen Vergleicher, weshalb man ihn auch als *einfach assoziativen Cache* bezeichneten kann. Mit der festen Zuordnung von Block zu Zeile ist jedoch auch eine starre *Ersetzungsstrategie* vorgegeben, die auf die Cache-Vergangenheit keine Rücksicht nimmt. Das wirkt sich nachteilig aus, wenn abwechselnd auf Blöcke zugegriffen wird, die derselben Zeile zugeordnet werden, z.B. wenn sich die Befehle einer Programmschleife mit den von ihr zu bearbeitenden Operanden im Cache überdecken. Hierbei geht die Wirkung des Cache durch das ständige Umladen verloren. Abhilfe schafft hier die getrennte Speicherung von Befehlen und Daten in zwei Caches (*split caches*). Dennoch ist der direkt zuordnende Cache als *L1-Cache* inzwischen nicht mehr gebräuchlich, da man mit geringem Mehraufwand einen *n*-Wege-assoziativen Cache realisieren kann und mit diesem sehr viel bessere Trefferquoten erzielt. Er wird allenfalls als Cache der zweiten oder dritten Ebene, d.h. als *L2*- oder *L3-Cache* eingesetzt.

n-Wege-assoziativer Cache. Der *n*-Wege-assoziative (*n-way set-associative cache*) oder *n*-fach assoziative Cache ist ein Kompromiß zwischen den beiden bisher beschriebenen Cache-Strukturen. Bei ihm werden jeweils zwei, vier oder mehr (allg. *n*) Zeilen zu Sätzen (sets) zusammengefaßt. Ein Block des Hauptspeichers wird einem bestimmten Satz fest zugeordnet, indem bei 2^m Sätzen die *m* niederwertigen Bits der Blockadresse als Index für die Satzanwahl herangezogen werden. (Dies entspricht beim direkt zuordnenden Cache dem Zuordnen zu einer bestimmten Zeile.) Das Zuordnen des Blocks zu einer der Zeilen des Satzes ist dann wahlfrei. Um die Cache-Einträge innerhalb eines Satzes eindeutig zu kennzeichnen, müssen zu jedem Block die verbleibenden höherwertigen Bits seiner Blockadresse als Tag gespeichert werden. (Dies entspricht beim vollassoziativen Cache der Assoziativität.) – Adressiert wird der Cache, indem der Satzindex mittels eines Decodierers ausgewertet und dann die Tag-Einträge aller Zeilen des Satzes mit den Tag-Bits der Hauptspeicheradresse verglichen werden. Zeigt einer der Vergleiche einen Treffer an (tag hit) und ist außerdem das Valid-Bit der so adressierten Zeile gesetzt (valid hit), so liegt ein Cache-Hit vor.

Bild 6-23 zeigt als Beispiel einen 2-Wege-assoziativen Cache als zwei parallel geschaltete direkt zuordnende Caches. Er hat 1024 Sätze, woraus sich ein Satzindex von 10 Bits und 18 verbleibende Bits für das Tag ergeben. Der Hardwareaufwand für die Adressierung ist wie beim direkt zuordnenden Cache relativ

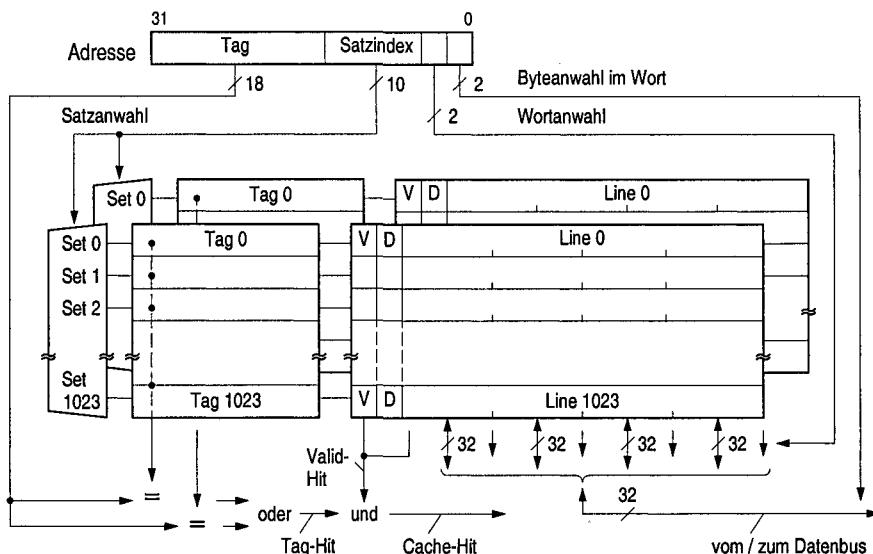


Bild 6-23. Adressierung eines 2-Wege-assoziativen Cache mit 1024 Sätzen (Sets) zu je zwei Zeilen (Lines) zu je vier 32-Bit-Wörtern. Anwahl eines Satzes durch einen 10-Bit-Satzindex mittels Decodierung; paralleler Tag-Vergleich für die beiden Zeilen des Satzes durch zwei Vergleicher. Cache-Hit bei Tag-Gleichheit (tag-hit) und gültigem Valid-Bit (valid-hit) für eine dieser beiden Zeilen

gering. Er umfaßt neben der Hardware für die Decodierung lediglich zwei Vergleicher für den gesamten Cache entsprechend der zwei Zeilen pro Satz. Hinzu kommt Hardware zur Realisierung einer *Ersetzungsstrategie* für jeden der 1024 Sätze. Sie erfordert jedoch wenig Aufwand, z.B. je ein Flipflop pro Satz (Wechselbelegung als Spezialfall der *LRU-Strategie*). Bei Caches mit mehr als zwei Zeilen pro Satz ist der Hardwareaufwand für die LRU-Strategie wesentlich höher. Alternativ dazu wird deshalb häufig die weniger aufwendige *Pseudo-LRU-Strategie* angewandt (siehe unten). Minimalen Hardwareaufwand, nämlich mit nur einer einzigen Ersetzungshardware für alle Sätze gemeinsam, erreicht man z.B. mit einem Modulozähler, der mit jedem Cache-Zugriff weitergeschaltet wird und dessen Zählwert die zu ersetzenen Zeile im betroffenen Satz bestimmt (*Zufallsstrategie*).

Der beim direkt zuordnenden Cache genannte Nachteil bei Überlagerung von zwei Blöcken ist beim 2-Wege-assoziativen Cache aufgehoben, bei Überlagerung von mehr als zwei Blöcken jedoch nicht. Hier bieten Caches mit mehr als zwei Zeilen pro Satz und entsprechend größerer Vergleicheranzahl eine höhere Trefferquote. – Heute sind n -Wege-assoziative Caches die gebräuchlichste Cache-Struktur, sowohl als L1- als auch als L2-Caches. Sie werden außerdem häufig als *Deskriptor-Caches* in Speicherverwaltungseinheiten eingesetzt. Typisch sind Realisierungen mit zwei, vier oder acht Zeilen pro Satz. Gebräuchlich sind aber auch Realisierungen mit sechs oder 12 Zeilen pro Satz.

Anmerkungen. (1.) Vor dem Hintergrund, daß ein n -Wege-assoziativer Cache durch n parallel arbeitende direkt zuordnende Caches dargestellt und aufgebaut werden kann, setzen einige Hersteller von Mikroprozessoren den Begriff *Satz* gleich der Anzahl der direkt zuordnenden Caches, d.h. der Anzahl der Zeilen pro Satz gemäß obiger Definition. Sie sprechen dann z.B. bei einem 2-Wege-assoziativen Cache von zwei Sätzen. (2.) Die Realisierung (on-chip oder off-chip) durch parallel arbeitende direkt zuordnende Caches setzt voraus, daß die über die RAM-Struktur hinaus benötigte Hardware von geringem Aufwand ist und in einfacher Weise zu dieser hinzugefügt werden kann. Dies trifft auf die Vergleicher zu, gilt aber für die Realisierung einer Ersetzungsstrategie nur dann, wenn diese pauschal für alle Sätze gemeinsam ausgelegt wird (so wie die oben erwähnte Zufallsstrategie, realisiert durch einen einzigen Modulozähler).

Pseudo-LRU-Strategie. Bei der *Ersetzungsstrategie LRU* wird jeweils jene Zeile aus dem Cache verdrängt, auf die am längsten nicht zugegriffen wurde (least recently used). Realisiert wird dieses Verfahren z.B. in Form einer Dreiecksmatrix, in der für jede Zeile eine Altersbeziehung zu allen anderen Zeile geführt wird (siehe [Liebig 2003]). Bei n Zeilen pro Satz eines n -Wege-assoziativen Cache sind das $n(n-1)/2$ Alterseinträge, also 6 Einträge bei vier Zeilen oder 28 Einträge bei acht Zeilen, die durch eine entsprechende Flipflop-Anzahl und zusätzliche Ansteuerlogik zu realisieren sind. Bei der weniger aufwendigen und deshalb gebräuchlicheren Pseudo-LRU-Strategie wird in Kauf genommen, daß der Ersetzungsmechanismus nicht immer den Eintrag mit dem am längsten zurückliegenden Zugriff auswählt. Die Anzahl benötigter Flipflops reduziert sich

dafür wesentlich auf $n-1$. Daß heißt, bei vier Zeilen pro Satz werden nur 3 (anstatt 6) und bei acht Zeilen nur 7 (anstatt 28) Flipflops benötigt.

Bild 6-24 veranschaulicht die Funktionsweise der Pseudo-LRU-Strategie am Beispiel von vier Cache-Zeilen. Die Flipflop-Stellungen zeigen zu jedem Zeitpunkt auf jene Zeile, deren Eintrag im Ersetzungsfall zu verdrängen ist. Aktualisiert werden sie dadurch, daß bei sämtlichen Cache-Zugriffen – Hit, Miss, Ersetzung, Nicht-Ersetzung – genau jene beiden Flipflops, die auf dem „Weg“ zur betroffenen Cache-Zeile liegen, so gestellt werden, daß sie von dieser Zeile bzw. diesem Eintrag wegzeigen. Folgender Ablauf, bei dem in Folge auf alle vier Zeilen zugegriffen wird, demonstriert die Pseudo-LRU-Funktionsweise:

1. Zu Beginn der Betrachtung seien alle drei Flipflops im Zustand 0 (d.h., im Ersetzungsfall würde der Eintrag der Zeile 0 verdrängt).
2. Hit-Zugriff auf Zeile 0 (danach wäre zu ersetzen der Eintrag der Zeile 2),
3. Hit-Zugriff auf Zeile 2 (danach wäre zu ersetzen der Eintrag der Zeile 1),
4. Hit-Zugriff auf Zeile 3 (danach wäre zu ersetzen der Eintrag der Zeile 1),
5. Hit-Zugriff auf Zeile 1 (danach wäre zu ersetzen der Eintrag der Zeile 2; dieser weist aber den zweitältesten Zugriff in der Zugriffsfolge auf!)

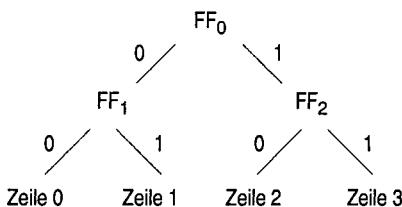


Bild 6-24. Veranschaulichung der Pseudo-LRU-Ersetzungsstrategie an vier Cache-Zeilen

6.2.4 Aktualisierungsstrategien und Datenkohärenz

Der in Hardware realisierte Cache-Controller ist nicht nur für das Laden und Adressieren des Cache zuständig, er hat auch für die *Datenkohärenz* (data coherence) zu sorgen, d.h. dafür, daß der Mikroprozessor immer auf aktuelle Speicherinhalte zugreift, sei es im Cache oder im Hauptspeicher. Bei Befehls-Caches ist das meist kein Problem, da der Prozessor üblicherweise nur Lesezugriffe ausführt. Bei Daten-Caches hingegen hängt es von der im Cache angewandten Aktualisierungsstrategie ab, ob bei Schreibzugriffen des Prozessors nur der Cache oder nur der Hauptspeicher oder ob beide Speicher aktualisiert werden. In jedem Fall muß hier der Controller gewährleisten, daß nachfolgende Lesezugriffe immer auf das zuletzt geschriebene und nicht auf ein inzwischen veraltetes Datum (*stale data*) erfolgen. – In der Literatur wird synonym zu Kohärenz häufig

der Begriff Konsistenz benutzt. Davon abweichend wird der Begriff *Konsistenz* von einigen Autoren für eine strengere Vorgabe verwendet, nämlich dafür, daß im Hauptspeicher und im Cache zu keinem Zeitpunkt verschiedene Kopien des selben Datenobjektes existieren [Gilioi 1993]. Häufig spricht man auch anstelle von Datenkohärenz von *Cache-Kohärenz*.

Tabelle 6-2 zeigt die beiden gebräuchlichsten Aktualisierungsstrategien für *Daten-Caches*, das Write-through-Verfahren (in zwei Varianten) und das Copy-back-Verfahren, mit ihren Wirkungsweisen bei den vier zu unterscheidenden Cache-Zugriffen Read-Hit, Read-Miss, Write-Hit und Write-Miss. Beide Strategien sichern die Datenkohärenz, sofern nur ein Master im System vorhanden ist. Sie geben jedoch unterschiedlich gute Gewähr, Prozessorzugriffe durch den Cache zu befriedigen, sind dafür aber auch unterschiedlich aufwendig in ihren Realisierungen.

Tabelle 6-2. Aktualisierungsstrategien für Daten-Caches mit je einem Valid- und einem Dirty-Bit pro Cache-Zeile. Das mit * gekennzeichnete Zurückkopieren eines Blocks in den Speicher ist nur dann erforderlich, wenn der Block gültig und „dirty“ ist ($V = 1, D = 1$). Die kursiven Einträge beschreiben die Aktionen in Befehls-Caches mit nur Lesezugriff

Cache-Zugriff	Write-Through with No-write-Allocation	Write-Through with Write-Allocation	Copy-Back
Read-Hit	<i>Cache-Datum → CPU</i>	<i>Cache-Datum → CPU</i>	<i>Cache-Datum → CPU</i>
Read-Miss	<i>Sp.-Block, Tag → Cache</i> <i>Sp.-Datum → CPU</i> $1 \rightarrow V$	<i>Sp.-Block, Tag → Cache</i> <i>Sp.-Datum → CPU</i> $1 \rightarrow V$	<i>Cache-Zeile → Speicher,*</i> <i>Sp.-Block, Tag → Cache</i> <i>Sp.-Datum → CPU</i> $1 \rightarrow V, 0 \rightarrow D$
Write-Hit	<i>CPU-Datum → Cache, Speicher</i>	<i>CPU-Datum → Cache, Speicher</i>	<i>CPU-Datum → Cache,</i> $1 \rightarrow D$
Write-Miss	<i>CPU-Datum → Speicher</i>	<i>Sp.-Block, Tag → Cache</i> $1 \rightarrow V;$ <i>CPU-Datum → Cache, Speicher</i>	<i>Cache-Zeile → Speicher,*</i> <i>Sp.-Block, Tag → Cache</i> $1 \rightarrow V;$ <i>CPU-Datum → Cache,</i> $1 \rightarrow D$

Beide Verfahren können in einem Cache gemischt verwendet werden, und zwar jedes für sich auf bestimmte Hauptspeicherseiten bezogen. Dazu enthalten die Seitendeskriptoren der Speicherverwaltungseinheit ein Bit, mit dem sie dem Cache-Controller signalisieren, welches der beiden Verfahren für den aktuellen Hauptspeicherzugriff zur Anwendung kommen soll (siehe 6.3.4). – Die Tabelle zeigt in kursiver Schreibweise auch die Cache-Zugriffe für einen reinen *Befehls-Cache*. Da es sich hierbei nur um Lesezugriffe handelt, ist nur eine einfache Aktualisierungsstrategie erforderlich: der Cache wird bei einem Read-Miss mit dem Block aus dem Hauptspeicher nachgeladen.

Write-through-Verfahren. Beim Write-through-Verfahren wird bei Schreibzugriffen grundsätzlich der Hauptspeicher aktualisiert, d.h., das aktuelle Datum wird unabhängig davon, ob es sich um einen Write-Hit oder einen Write-Miss handelt, in den Hauptspeicher „durchgeschrieben“. Bei einem Write-Hit wird außerdem auch der Cache aktualisiert. Dadurch stimmen zwar Cache- und Hauptspeicherinhalte jeweils überein, der Cache verliert jedoch für die Schreibzugriffe seinen Vorteil, wobei diese bis zu 30% aller Zugriffe ausmachen können. Abschwächen läßt sich dieser Nachteil dadurch, daß man für das Durchschreiben ein Pufferregister zwischen Cache und Hauptspeicher vorsieht (*buffered write-through*) und den Cache-Controller so auslegt, daß der nächste Cache-Zugriff gestartet werden kann, bevor das Durchschreiben abgeschlossen ist. Einen Zeitgewinn erhält man allerdings zunächst nur dann, wenn es sich bei dem Folgezugriff um einen Lesezugriff handelt, der vom Cache befriedigt werden kann (read-hit). Der Cache-Controller kann jedoch so ausgelegt werden, daß er das Durchschreiben ggf. gegenüber dem Folgezugriff verzögert (*posted write-through*), um einen nachfolgenden Speicherzugriff dem Durchschreiben vorzuziehen. Durch die Steuerung muß dann gewährleistet sein, daß z.B. bei einem Lesezugriff mit gleicher Adresse nicht das veraltete Datum aus dem Hauptspeicher, sondern das aktuelle Datum aus dem Pufferregister geholt wird.

Das Durchschreiben und das damit verbundene sofortige Aktualisieren des Hauptspeichers bietet aber auch Vorteile. So wird das Write-through-Verfahren z.B. auf im Hauptspeicher angelegte Bildspeicherbereiche angewandt, deren Inhalte z.B. von einem DMA-Controller kontinuierlich ausgelesen werden, um auf einem Bildschirm dargestellt zu werden. Hier würde ein verzögertes Aktualisieren des Hauptspeichers, wie dies beim Copy-Back-Verfahren der Fall ist, zu fehlerhaften Darstellungen führen. (Heute sind jedoch die Bilddaten meist in einem eigenen Speicher der Grafikeinheit abgelegt, so daß sich diese Anwendung erübrigt.)

Ein weiterer Vorteil dieses Verfahrens ist, wenn es ausschließlich implementiert wird, sein relativ geringer Hardwareaufwand, insbesondere bei der einfacheren und häufig verwendeten Variante mit *No-write-Allocation*. Bei ihr wird bei einem Write-Miss nur der Hauptspeicher und nicht der Cache aktualisiert. Aufwendiger hingegen ist die weniger gebräuchliche Variante mit *Write-Allocation*. Bei ihr wird bei einem Write-Miss zusätzlich zum Hauptspeicher auch der Cache aktualisiert, wobei zunächst der gesamte Block aus dem Hauptspeicher geladen und dann im Cache das aktuelle Datum in den Block geschrieben wird. Sie wird zur Lösung des sog. *Address-aliasing-Problems* bei virtuell adressierten Caches eingesetzt (6.2.5).

Copy-back-Verfahren. Beim Copy-back-Verfahren wird bei Schreibzugriffen grundsätzlich der Cache und nicht der Hauptspeicher aktualisiert, so daß der Cache auch bei Schreibzugriffen mit Write-Hit seine Wirksamkeit behält. Bei einem Write-Miss wird der Cache zunächst mit dem betroffenen Block aus dem

Speicher geladen (*Write-Allocation*) und in diesen dann das Datum geschrieben. Das Aktualisieren des Hauptspeichers erfolgt erst bei einer Blockersetzung, d.h. wenn ein Block im Cache durch einen neu zu ladenden Block überschrieben werden muß. Dazu wird dieser in den Hauptspeicher „zurückkopiert“. Bei einfacher Realisierung geschieht das Zurückkopieren ohne Bedingung (*simple copy-back*). Bei der gebräuchlichen, etwas aufwendigeren Realisierung wird, wie in den Bildern 6-21 bis 6-23 und in Tabelle 6-2 angenommen, das Zurückkopieren von einem für jede Cache-Zeile vorhandenen *Dirty-Bit* abhängig gemacht (*flagged copy-back*). Dieses zeigt an, ob der Inhalt der Zeile durch einen Schreibzugriff verändert wurde (*dirty line*) oder nicht (*clean line*). Darüber hinaus werden für das Zurückkopieren Blockpuffer eingesetzt, so daß der Prozessor ggf. das Zurückkopieren nicht abzuwarten braucht (*buffered/posted copy-back*). Als weitere Variante kann für jedes Wort in einer jeden Zeile ein eigenes Dirty-Bit vorgesehen sein, so daß nur noch die tatsächlich veränderten Wörter eines Blocks zurückkopiert werden müssen. – Wie oben bereits erwähnt, wird bei Daten-Caches häufig sowohl das Copy-back- als auch das Write-through-Verfahren implementiert, um die Vorteile beider Verfahren nutzen zu können.

Zusätzliche Master ohne Cache. Bild 6-25 zeigt ein Mikroprozessorsystem, das neben dem Prozessor mit Cache einen DMA-Controller als weiteren Master (ohne Cache) aufweist; es könnte auch ein zweiter Prozessor ohne Cache sein, z.B. ein Ein-/Ausgabeprozessor. Dieser zusätzliche Master hat wie der Prozessor Zugriff auf den Hauptspeicher, in dem sich beide Master z.B. einen Ein-/Ausgabe-Pufferbereich teilen (*shared data*), der eine oder mehrere Seiten umfassen kann (zur Seitenverwaltung siehe 6.3.2). Das *Kohärenzproblem* tritt dann auf, wenn bei Verwendung des Write-through-Verfahrens der DMA-Controller eine Speicherzelle beschreibt, deren zuvor vorhandener Inhalt im Cache als gültig eingetragen war, und der Prozessor danach einen Lesezugriff mit der Adresse dieser Speicherzelle durchführt. Der Prozessor wird dann den inzwischen veralteten Cache-Eintrag lesen. Das Kohärenzproblem tritt auch dann auf, wenn der Prozessor bei Verwendung des Copy-back-Verfahrens einen Schreibzugriff mit einer Adresse des Pufferbereichs ausführt und dabei lediglich seinen Cache aktualisiert, und danach der DMA-Controller einen Lesezugriff mit dieser Adresse aus-

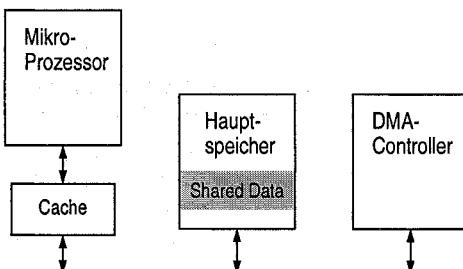


Bild 6-25. Mikroprozessorsystem mit einem Prozessor *mit* Cache und einem DMA-Controller als zusätzlichem Master *ohne* Cache. Zugriff beider Master auf gemeinsame Daten im Hauptspeicher

führt. Er wird dann den inzwischen veralteten Hauptspeichereintrag lesen. Um die Datenkohärenz für *Shared-memory*-Bereiche herzustellen, gibt es drei gebräuchliche Lösungen.

Non-cacheable-Data. Die erste Lösung wird durch die Speicherverwaltungseinheit unterstützt und geht davon aus, den vom Prozessor und dem zusätzlichen Master gemeinsam benutzten Speicherbereich von einer Speicherung im Cache generell auszuschließen. Dazu wird dieser Adressbereich in den zugehörigen Seitendeskriptoren als „non-cacheable“ gekennzeichnet (6.3.4). Die Speicherverwaltungseinheit weist dann den Cache-Controller bei allen Prozessorzugriffen, die diesen Adressraum betreffen, an, nicht aktiv zu werden. Ist keine Speicherverwaltungseinheit vorhanden, so kann diese Information auch im Cache-Controller selbst erzeugt werden, wenn dort ein entsprechendes Adressregisterpaar für die Anfangs- und Endadresse des Bereichs und eine Vergleichseinrichtung für die Überprüfung der aktuellen Adresse vorgesehen ist. – Die Kennzeichnung „non-cacheable“ wird grundsätzlich auch für die von Interfaces und Ein-/Ausbabekontrollern belegten Adressbereiche benutzt, um den direkten Zugriff auf deren Daten-, Status- und Steuerregister zu gewährleisten.

Cache-Clear, Cache-Flush. Die zweite Lösung basiert darauf, daß die Zugriffe von Prozessor und DMA-Controller auf den gemeinsamen Datenbereich durch zwei unterschiedliche Prozesse ausgeführt werden. Hier kann beim Prozeßwechsel dafür gesorgt werden, daß der Cache rechtzeitig „gelöscht“ wird. Damit wird einerseits erreicht, daß Prozessorzugriffe, die nach dem Ändern der Hauptspeicherinhalte durch den DMA-Controller erfolgen, zu einem Neuladen des Cache führen, und andererseits, daß die im Cache aktualisierten Inhalte in den Hauptspeicher zurückkopiert werden, bevor der DMA-Controller auf diesen lesend zugreift. Arbeitet der Cache nach dem Write-through-Verfahren, so geschieht das Löschen durch einen Cache-Clear- oder Cache-Invalidate-Befehl. Hierbei werden von der Cache-Steuereinheit die Valid-Bits sämtlicher Cache-Einträge auf „invalid“ gesetzt. Arbeitet der Cache nach dem Copy-back-Verfahren, so geschieht das Löschen durch einen Cache-Flush- oder Cache-Push-Befehl. Hierbei werden ebenfalls sämtliche Valid-Bits auf „invalid“ gesetzt, zuvor werden jedoch alle mit „dirty“ gekennzeichneten Einträge in den Hauptspeicher zurückkopiert. – Die Befehle Cache-Clear und Cache-Flush gibt es nicht nur auf den gesamten Cache wirkend, sondern auch mit eingeschränkter Wirkung, nämlich für einzelne Seiten (page clear, page flush) und für einzelne Blöcke (line clear, line flush).

Anmerkung. Bei Verwendung des Copy-back-Verfahrens ist das Aktualisieren des Hauptspeichers mittels des Cache-Flush-Befehls insbesondere auch dann erforderlich, wenn Hauptspeicherbereiche (Seiten), auf die schreibend zugegriffen wurde, auf den Hintergrundspeicher zurückgeschrieben werden müssen. Andererseits gibt es aber auch Situationen, in denen die Cache-Inhalte nur invalidiert werden dürfen und dementsprechend anstelle eines Cache-Flush nur ein Cache-Clear ausgeführt werden darf.

Bus-Snooping. Bei der dritten Lösung, dem Bus-Snooping (Busschnüffeln), auch Bus-Watching (Busbeobachten) genannt, ist der Cache-Controller um eine sog. *Snoop-Logik* erweitert. Mit ihr beobachtet der Cache den Bus hinsichtlich der Busaktivitäten anderer Master, z.B. des DMA-Controllers, und reagiert ggf. darauf. Bei einem „Write-through“-Cache wird bei einem Write-Hit des anderen Masters (*snoop-hit* on a write) der entsprechende Cache-Eintrag als ungültig gekennzeichnet (line clear). Bei einem „Copy-back“-Cache wird in diesem Fall der Cache-Eintrag je nach Realisierung des Cache-Controllers entweder zurückkopiert und als ungültig gekennzeichnet (line flush); der andere Master wird dann gehalten, den Hauptspeicher erst nach dem Zurückkopieren zu aktualisieren. Oder der Cache-Eintrag wird aktualisiert und als „dirty“ gekennzeichnet; dann braucht der Hauptspeicher nicht aktualisiert zu werden. Bei einem Read-Hit des anderen Masters (*snoop-hit* on a read) kann das Datum unabhängig vom Aktualisierungsverfahren vom Cache bereitgestellt werden. – Im Gegensatz zu den beiden zuvor angegebenen Lösungen des Kohärenzproblems ist beim Bus-Snooping der Cache auch für Zugriffe anderer Master von Nutzen (*snoop-hit* on a read-hit), und es wird darüber hinaus ggf. der Nutzen für den zugehörigen Prozessor erhöht (Aktualisierung des „Copy-back“-Cache bei Snoop-Hit on a Write).

Anmerkung. Wie in 6.2.1 beschrieben, werden Caches inzwischen vorwiegend als Look-through-Caches ausgelegt, um die Busaktivitäten des Prozessors zugunsten anderer Master zu minimieren. Dies gilt auch für einfache Mehrmastersysteme, bestehend aus Prozessor und DMA-Controller (Bild 6-25). Hierbei bleibt die Snoop-Logik durchgängig aktiviert, um den Bus bezüglich der Hauptspeicherzugriffe des DMA-Controllers beobachten zu können.

Zusätzliche Master mit Cache. Bild 6-26 zeigt ein *Mehrprozessorsystem* mit zwei Prozessoren mit jeweils eigenem Cache und mit einem für beide Prozessoren gemeinsamen Hauptspeicher. Davon ausgehend, daß beide Prozessoren auf gemeinsame Daten im Speicher zugreifen (*shared data*), besteht auch hier wieder das Kohärenzproblem. Unter Benutzung des Bus-Snooping und des Write-through-Verfahrens für beide Caches läßt sich dieses Problem relativ einfach lösen. Das Write-through-Verfahren zwingt jeden der Prozessoren, bei Schreibzugriffen jeweils in den Speicher durchzuschreiben, d.h., den Bus zu benutzen. Diese Busaktivität kann von der *Snoop-Logik* des jeweils anderen Cache dazu

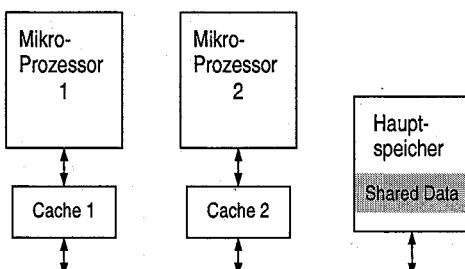


Bild 6-26. Mikroprozessorsystem mit zwei Prozessoren mit Cache. Zugriff beider Prozessoren auf gemeinsame Daten im Hauptspeicher

ausgewertet werden, bei einem Write-Hit den eigenen Eintrag als ungültig zu kennzeichnen. Die *Datenkohärenz* wird dementsprechend über den Hauptspeicher hergestellt.

Etwas aufwendiger wird die Lösung des Kohärenzproblems, wenn beide Caches nach dem Copy-back-Verfahren betrieben werden. Eine Möglichkeit besteht z.B. darin, die gemeinsam benutzten Daten in nur einem der beiden Caches (A) zu halten und sie für den anderen Cache (B) als „non-cacheable“ zu kennzeichnen. Dadurch werden die Datenzugriffe des Prozessors B auf den Bus gelenkt und können somit von der Snoop-Logik des Cache A erkannt werden. Cache A kann dann bei einem „Snoop-Hit on a Write“ seinen eigenen Eintrag entweder als ungültig kennzeichnen oder aktualisieren bzw. bei einem „Snoop-Hit on a Read“ das Datum bereitstellen. – Eine allgemeinere Lösung des Kohärenzproblems in Mehrprozessorsystemen mit für jeden Prozessor eigenem Daten-Cache (siehe dazu SMP-Systeme in 5.1.7) bietet das sehr gebräuchliche und im folgenden beschriebene MESI-Kohärenzprotokoll. Daneben gibt es andere Protokolle, über die [Archibald, Baer 1986] einen guten Überblick gibt.

MESI-Kohärenzprotokoll. Das MESI-Kohärenzprotokoll geht vom Copy-back-Verfahren aus, berücksichtigt in abgemagerter Form aber auch das Write-through-Verfahren und kommt so der durch die Speicherverwaltungseinheit gegebenen Möglichkeit entgegen, das Aktualisierungsverfahren für jede Seite durch ein Modusbit im Seitendeskriptor individuell festzulegen (siehe 6.3.4). Beispielsweise werden Seiten, die die lokalen Variablen des Prozessors enthalten nach dem *Copy-back-Verfahren*, und Seiten deren Hauptspeicherdarstellung immer aktuell sein muß, z.B. Seiten für die Grafikausgabe, nach dem *Write-through-Verfahren* aktualisiert.

Voraussetzung für das Protokoll ist auf der Hardwareseite für jeden Cache eine *Snoop-Logik*, erweitert um folgende Steuersignale, mit denen sich die Snoop-Logiken der Caches untereinander verständigen: das Invalidate-Signal zur Invalidierung von Einträgen, die sich in anderen Caches befinden, das Shared-Signal als Anzeige anderer Caches, ob der zu ladende Cache-Block bereits als Kopie vorhanden ist, sowie das Retry-Signal, mit dem ein anderer Prozessor aufgefordert werden kann, das Laden eines Block abzubrechen, um es später wieder aufzunehmen, nachdem der auffordernde Prozessor diesen Block aus seinem Cache in den Hauptspeicher zurückgeschrieben hat (line flush, siehe dazu Beispiel 6.3, S. 426). Hinzu kommen für jede Cache-Zeile zwei Statusbits, mit denen die vier Protokollzustände **Modified**, **Exclusive**, **Shared** und **Invalid**, codiert werden. Beim Write-through-Verfahren sind nur die Zustände **S** und **I** relevant.

- Zustand Invalid: Die betrachtete Cache-Zeile ist ungültig. – Ein Lesezugriff (read-miss) wie auch ein Schreibzugriff (write-miss) auf diese Zeile veranlassen den Cache, den Speicherblock in die Cache-Zeile zu laden. Die anderen Caches, die den Bus beobachten, zeigen mittels des Shared-Signals an, ob dieser Block bei ihnen bereits gespeichert ist (shared read-miss) oder nicht

(exclusive read-miss). Davon abhängig erfolgt der Übergang entweder in den Zustand S oder in den Zustand E. Beim Write-Miss erfolgt der Übergang in den Zustand M, wobei der Cache das Invalidate-Signal ausgibt, da er den geladenen Block ja verändert. Dieses wird von den anderen Caches ausgewertet, die daraufhin ihre Einträge gleicher Blockadresse invalidieren.

- Zustand Shared, genauer Shared Unmodified: Der Speicherblock existiert als Kopie in der Zeile des betrachteten Cache sowie ggf. in anderen Caches. – Bei einem Lesezugriff auf die Cache-Zeile (read-hit) bleibt der Zustand S erhalten. Bei einem Schreibzugriff (write-hit) wird die Cache-Zeile verändert und geht in den Zustand M über. Dabei wird das Invalidate-Signal aktiviert, woraufhin andere Caches, bei denen dieser Block ebenfalls den Zustand S hat, ihre Zeile als ungültig kennzeichnen (Zustand I).
- Zustand Exclusive, genauer Exclusive Unmodified: Der Speicherblock existiert als Kopie nur in der Zeile des betrachteten Cache. – Der Prozessor kann lesend und schreibend auf die Cache-Zeile zugreifen, ohne den Bus benutzen zu müssen. Beim Schreiben (write-hit) erfolgt ein Wechsel in den Zustand M. Andere Caches sind davon nicht betroffen.
- Zustand Modified, genauer Exclusive Modified: Der Speicherblock existiert als Kopie nur in der Zeile des betrachteten Cache. Er wurde nach dem Laden verändert (*dirty line*), d.h., der ursprüngliche Block im Speicher ist nicht mehr aktuell. – Der Prozessor kann lesend und schreibend auf die Cache-Zeile zugreifen, ohne den Bus benutzen zu müssen. Bei einem Lese- oder Schreibzugriff eines anderen Prozessors auf diesen Block (snoop-hit) muß dieser in den Hauptspeicher zurückkopiert werden (line flush). Die Cache-Zeile geht dabei vom Zustand M in den Zustand S (snoop-hit on a read) bzw. in den Zustand I über (snoop-hit on a write or read w. I. t. M.). Der Cache des anderen Prozessors, der diesen Block zunächst aus dem Hauptspeicher zu laden versucht, wird mittels des Retry-Signals über das erforderliche Zurückkopieren informiert, woraufhin er den Ladevorgang abbricht und ihn erst nach dem Aktualisieren des Hauptspeichers wieder aufnimmt. Dieses wird ihm durch das Deaktivieren von Retry angezeigt.

Bild 6-27 zeigt den Zustandsgraphen des MESI-Protokolls, der besseren Übersichtlichkeit halber in zwei Teilbilder aufgeteilt. Teilbild a zeigt die Zustandsübergänge, die durch die „lokalen“ Schreib- und Lesezugriffe, d.h. durch die Zugriffe des Prozessors ausgelöst werden. Hierbei ergeben sich unterschiedliche Situationen, je nachdem, ob der adressierte Speicherblock bereits im Cache vorhanden ist, ob eine bislang gültige Zeile im Cache durch den Block überschrieben wird („with Replacement“), d.h. eine andere Blockadresse wirksam wird, oder ob eine bislang als ungültig gekennzeichnete Cache-Zeile mit dem Block belegt wird. Die Bedingung Read-with-Intent-to-Modify tritt bei *Semaphorbefehlen* auf (siehe 2.1.4). Diese führen einen Lesevorgang, gefolgt von einem Schreibvorgang, in (aus Sicht der *Busarbitration*) nichtunterbrechbarer Weise aus. Man

bezeichnet diesen „Doppel“buszyklus auch als *Read-Modify-Write-Zyklus*. In Teilbild a nicht dargestellt sind die Übergänge, die sich aus Sonderfällen, wie der

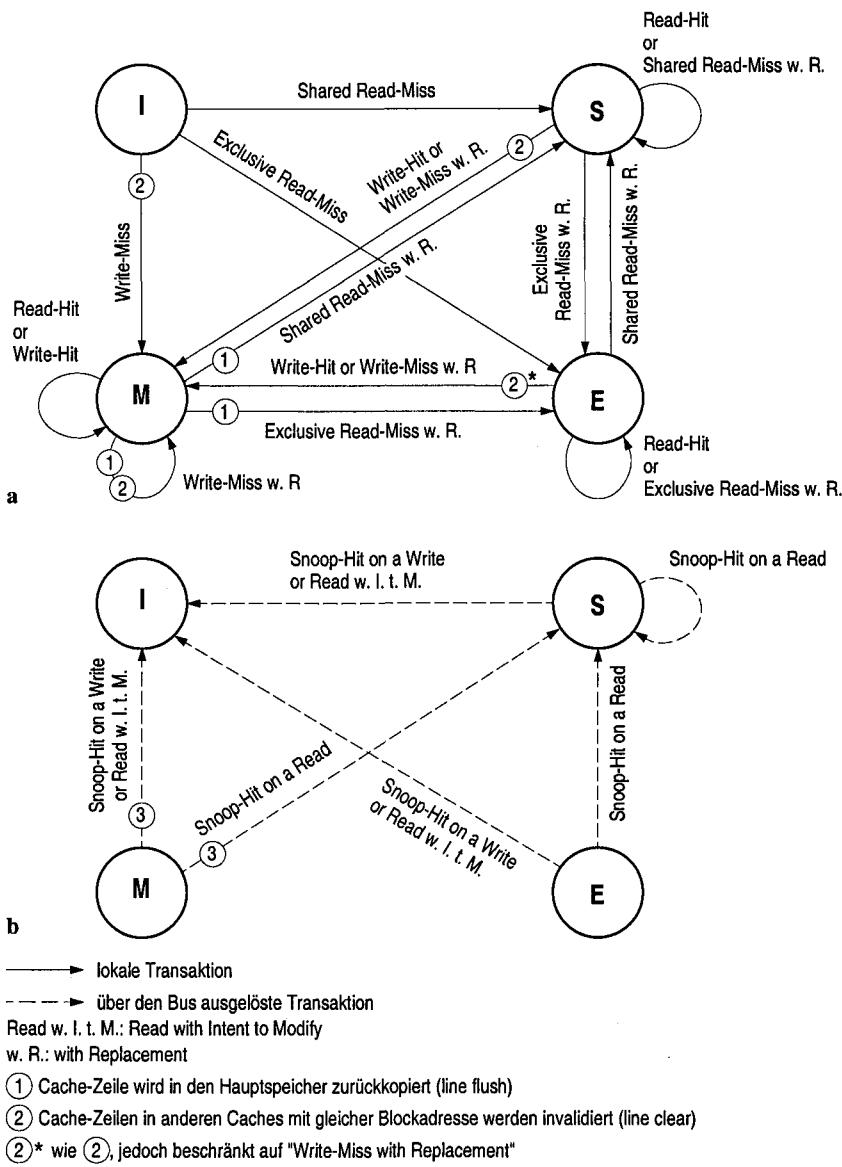


Bild 6-27. Zustandsgraph des MESI-Kohärenzprotokolls, a mit den Übergängen für die „lokalen“ Schreib- und Lesezugriffe und b mit den von „außen“, d.h. von Seiten des Busses ausgelösten Übergängen (in Anlehnung an die Darstellungen für die Prozessoren MC88110 [Motorola 1991] und PowerPC 601 [Motorola 1993]])

Cache-flush-Operation und Zugriffen auf „Non-cacheable“-Adressbereiche ergeben. Letztere werden dem Cache durch ein Steuersignal der Speicherverwaltungseinheit angezeigt. – Teilbild b zeigt die Zustandsübergänge, die sich durch Beeinflussung von „außen“, d.h. von Seiten des Busses ergeben. Sie werden durch die Snoop-Logik gesteuert.

Beispiel 6.3. ► Wirkungsweise des MESI-Protokolls. Bild 6-28 zeigt die Wirkungsweise des MESI-Protokolls für zwei Prozessor/Cache-Paare. Dargestellt sind vier Datenzugriffe, die alle denselben Speicherblock betreffen. Vor dem ersten Zugriff sei in keinem der beiden Caches eine Kopie dieses Speicherblocks vorhanden, außerdem seien in beiden Caches die für das Laden des Blocks benutzten Zeilen ungültig, d.h. im Zustand I. Dazu folgende Details:

1. Lesezugriff von Prozessor 1 – Read Miss: Der Lesezugriff von Prozessor 1 führt bei seinem Cache zu einem Read-Miss, woraufhin dieser den Block aus dem Hauptspeicher lädt. Die

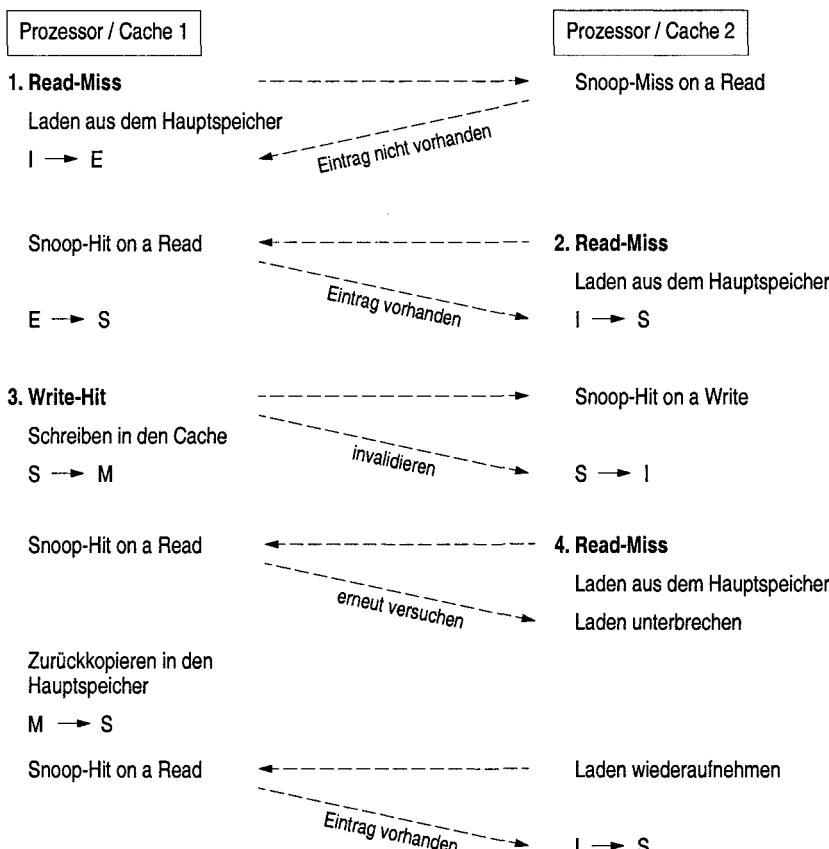


Bild 6-28. Wirkung des MESI-Kohärenzprotokolls in einem Mehrprozessorsystem mit zwei Prozessor/Cache-Paaren für vier aufeinanderfolgende Zugriffe auf ein und denselben Speicherblock. Die Interaktionen zwischen den Caches sind mit gestrichelten, die daraus resultierenden Zustandsübergänge für die Cache-Einträge mit durchgezogenen Pfeilen dargestellt

Snoop-Logik des Cache 2 beobachtet diesen Zugriff, erkennt einen Snoop-Miss und signalisiert dem Cache 1 (Shared-Signal nicht gesetzt), daß er keine Kopie des Blocks hat. Daraufhin bezeichnet der Cache 1 seinen neuen Eintrag mit E.

2. Lesezugriff von Prozessor 2 – Read-Miss: Der Lesezugriff von Prozessor 2 läuft im Prinzip wie der vorangegangene ab, jedoch hat Cache 1 einen Snoop-Hit und meldet deshalb dem Cache 2, daß bei ihm bereits eine Kopie des Blocks vorhanden ist (Shared-Signal gesetzt). Dementsprechend bezeichnen beide Caches ihren Eintrag mit S.
3. Schreibzugriff von Prozessor 1 – Write-Hit: Der Schreibzugriff von Prozessor 1 führt zu einem Write-Hit. Das heißt, der Cache-Eintrag wird verändert und dementsprechend die Cache-Zeile mit M bezeichnet. Der Cache 1 aktiviert außerdem das Invalidate-Signal, woraufhin der Cache 2 seinen durch die Snoop-Logik angewählten Eintrag ungültig macht (line clear, er wird nicht zurückkopiert!) und dementsprechend die Zeile mit I kennzeichnet.
4. Lesezugriff von Prozessor 2 – Read-Miss: Der erneute Lesezugriff von Prozessor 2 führt zu einem Read-Miss, der den Cache 2 veranlaßt, den Block aus dem Hauptspeicher zu laden. Der Cache 1, der zu diesem Zeitpunkt das einzige gültige Exemplar des Blocks enthält (Zustand M), fordert den Cache 2 auf, den Ladevorgang abzubrechen (Retry-Signal gesetzt). Er selbst kopiert nach Freigabe des Busses seinen Eintrag in den Hauptspeicher zurück (line flush) und bezeichnet ihn dann mit S. Außerdem meldet er dem Cache 2 den Abschluß des Rückschreibens (Retry-Signal rückgesetzt). Sobald dem Cache 2 der Bus wieder zur Verfügung steht, setzt er das unterbrochene Laden fort und bezeichnet danach seinen Eintrag ebenfalls mit S. ▲

Im obigen Beispiel wurde eine Zugriffsfolge gewählt, bei der die Caches ihren Nutzen dadurch verlieren, daß bei allen vier Zugriffen Snoop-Operationen mit Adreßübertragungen auf dem Bus erforderlich sind. Sie „kosten“ nicht nur Übertragungszeit, sondern auch Busarbitrationszeit. Betroffen sind davon alle Miss-Zugriffe, bei denen die Caches sowieso keinen Nutzen bringen, aber auch Write-Hit-Zugriffe, wenn zuvor der Zustand S bestand. Die Caches kommen jedoch bei allen Read-Hits (Zustand S, E oder M) und bei Write-Hits, ausgehend vom Zustand E oder M, zum Tragen. Gegenüber einem Einprozessorsystem (mit *Look-through-Cache*) gibt es also für das Snooping nur dann zusätzliche Busaktivitäten, wenn Write-Hits auf Zeilen erfolgen, die sich im Zustand S befinden.

Unabhängig davon kann bei einigen Prozessoren das Kohärenzprotokoll auf die Adressen der *Shared-Memory*-Bereiche, d.h. auf die *globalen Adressen* einschränkt werden (für die ja das Kohärenzproblem nur relevant ist). Dazu werden die globalen Speicherbereiche in den Seitendeskriptoren als solche gekennzeichnet (siehe auch 6.3.4), und die Speicherverwaltungseinheit zeigt ihrem zugehörigen Cache wie auch den anderen Caches durch ein Steuersignal an, um welche Art von Zugriff es sich handelt. Damit wird der Snoop-Mechanismus eines Cache bei den lokalen Zugriffen seines Prozessors abgeschaltet, so daß seine entkoppelnde Wirkung bei diesen Zugriffen zum Tragen kommt.

Das MESI-Protokoll ist bei Mikroprozessoren das verbreitetste Kohärenzprotokoll, sowohl für den L1- als auch den L2-Cache. Es gibt jedoch auch Systeme, bei denen der L2-Cache einem anderen Protokoll mit ggf. fünf oder auch nur drei Zuständen unterliegt. – Hinsichtlich der Unterstützung solcher Techniken zur Her-

stellung der Datenkohärenz siehe als konkretes Beispiel die diesbezüglichen Angaben zum PCI-Bus (5.6.3: Cache-Unterstützung, S. 362).

6.2.5 Virtuelle und reale Cache-Adressierung

Bei Mikroprozessorsystemen mit einer *Speicherverwaltungseinheit* gibt es drei Möglichkeiten, den Cache zu adressieren:

1. mit der *virtuellen Adresse*, d.h. mit der vom Prozessor bzw. Programm erzeugten Adresse,
2. mit der *realen Adresse*, d.h. mit der von der Speicherverwaltungseinheit erzeugten und dem Speicher zugeführten Adresse, und
3. mit je einem Teil der virtuellen und der realen Adresse.

Man spricht dann auch von *virtuellem, realem* bzw. *virtuell/realem Cache*. Bei diesen Systemen, insbesondere bei virtueller Cache-Adressierung, treten *Kohärenzprobleme* auf, die sich jedoch mit den in 6.2.4 beschriebenen Techniken lösen lassen. Ursache dieser Probleme sind zum einen zusätzliche Master im System, z.B. DMA-Controller, die ihrerseits entweder mit virtuellen oder realen Adressen arbeiten (siehe auch [Van Loo 1987]). Zum andern hängen diese Probleme mit dem *Mehrprogrammbetrieb* (*multiprogramming, multitasking*) zusammen und ergeben sich beim Programm/Prozeßwechsel sowie bei Zugriffen mehrerer Programme/*Prozesse (tasks)* auf gemeinsame Daten. Beim realen Cache gibt es darüber hinaus noch ein Zeitproblem, da dieser im Prinzip auf die Adreßumsetzung der Speicherverwaltungseinheit warten muß.

Virtueller Cache. Bild 6-29a zeigt ein System mit *virtueller Adressierung* des Cache durch den Mikroprozessor und mit realer Adressierung des Hauptspeichers durch eine Speicherverwaltungseinheit (MMU). Eine vom Prozessor ausgebogene Adresse kann in dieser Konfiguration gleichzeitig dem Cache und der Speicherverwaltungseinheit zugeführt werden, so daß bei einem Cache-Miss kein Zeitnachteil für den Hauptspeicherzugriff und bei einem Cache-Hit kein Zeitnachteil für den Cache-Zugriff entsteht.

Mehrere Master. Das Kohärenzproblem ergibt sich naturgemäß durch das Vorhandensein zusätzlicher Master wie des DMA-Controllers in Bild 6-29a. Dieser greift auf den Hauptspeicher zu und teilt sich darin Datenbereiche mit dem Prozessor. Seine Zugriffe erfolgen im Fall 1 mit realen und im Fall 2 mit virtuellen Adressen. Arbeitet er mit realen Adressen, so ergeben sich bei Schreibzugriffen auf den Hauptspeicher Inkohärenzen, da der Cache aufgrund seiner virtuellen Adressierung nicht unmittelbar an die veränderten Verhältnisse angepaßt werden kann. (*Bus-Snooping* ist aufgrund der unterschiedlichen Adreßdarstellung – virtuell, real – nicht möglich!).

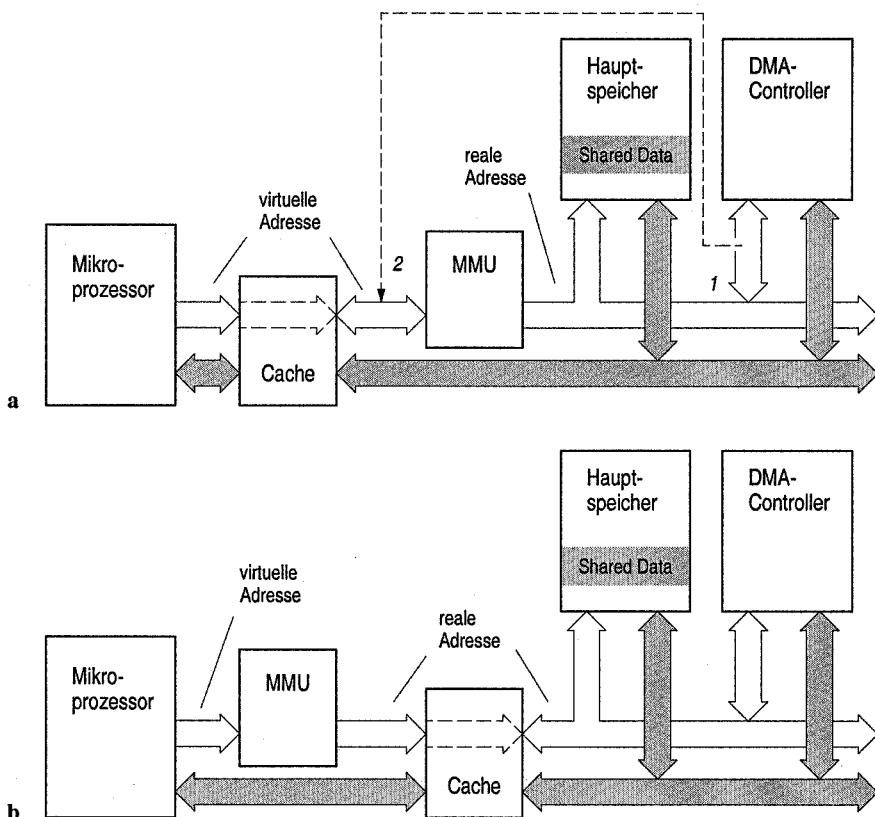


Bild 6-29. Mikroprozessorsystem mit zusätzlichem Master (DMA-Controller) und Speicherverwaltungseinheit (MMU). a Virtuelle Cache-Adressierung durch den Mikroprozessor; 1 reale, 2 virtuelle Hauptspeicheradressierung durch den zusätzlichen Master. b Reale Cache-Adressierung sowohl durch den Mikroprozessor als auch den zusätzlichen Master

Gelöst wird dieses Problem dadurch, daß entweder der gemeinsame Datenbereich von vornherein als *non-cacheable* gekennzeichnet wird, oder daß die betroffenen Cache-Inhalte vor den Zugriffen des DMA-Controllers durch *Cache-clear-* oder *Cache-flush-Operationen* (Write-through- bzw. Copy-back-Cache) als ungültig gekennzeichnet werden. Bei virtueller Adressierung durch den DMA-Controller, Fall 2, sind sowohl die eben beschriebenen Lösungen als auch das Bus-Snooping und damit auch das *MESI-Protokoll* verwendbar, da jetzt beide Master mit gleichartigen, nämlich virtuellen Adressen arbeiten. – Zu diesen Lösungen siehe auch 6.2.4.

Prozeßwechsel. Bild 6-30 zeigt schematisch mehrere voneinander unabhängige Prozesse, die sich in ihren virtuellen Adressen teilweise oder ganz überdecken (z.B. Adreßzählung jeweils ab Adresse Null) und die sich den realen Adreßraum, d.h. den Hauptspeicher teilen. Wie mit den drei gestrichelten Pfeilen beispielhaft

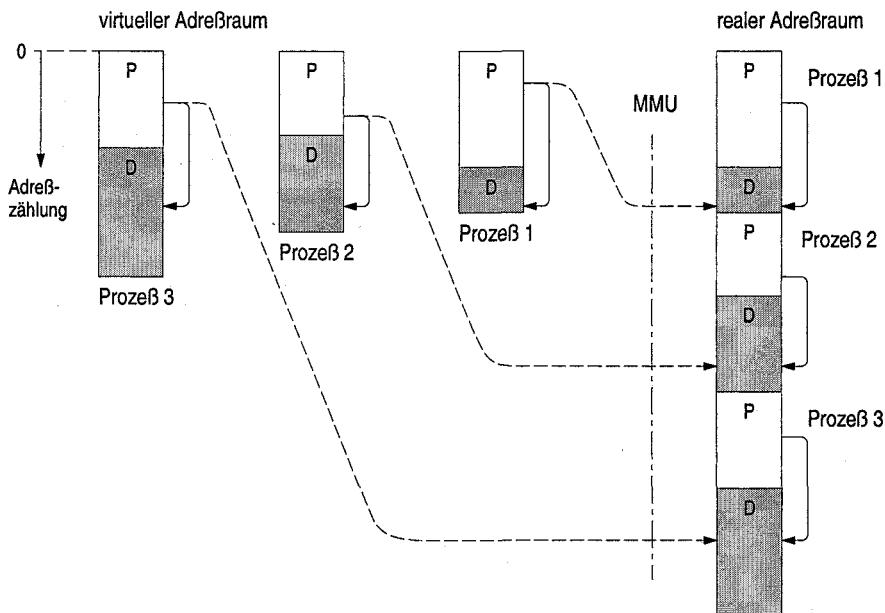


Bild 6-30. Schematische Darstellung mehrerer voneinander unabhängiger Prozesse, bestehend aus Programmteil P und Datenteil D mit teilweiser bzw. vollständiger Überdeckung ihrer virtuellen Adressen. Gleiche virtuelle Adressen werden beim Prozeßwechsel auf unterschiedliche reale Adressen abgebildet

gezeigt, hat ein und dieselbe virtuelle Adresse – auf die unterschiedlichen Prozesse bezogen – unterschiedliche reale Entsprechungen. Bei einem Prozeßwechsel führt dies zu Inkohärenzen, da der Cache virtuell adressiert wird und somit nicht berücksichtigen kann, daß die MMU prozeßabhängig unterschiedliche reale Adressen erzeugt. Eine erste Lösung des Problems ist das Löschen des Cache beim Prozeßwechsel durch eine *Cache-clear-* bzw. *Cache-flush-Operation*. Eine effizientere Lösung, bei der die Cache-Inhalte des bisherigen Prozesses nur in dem Maße gelöscht werden, wie der neue Prozeß Cache-Speicherplatz benötigt, besteht darin, die Adressierung des Cache um eine *Prozeßidentifikation* zu erweitern. Sie wird im Prozessor in einem Zusatzregister, sozusagen als Erweiterung der Adresse, prozeßspezifisch vorgegeben und als Tag-Erweiterung zur Cache-Anwahl benutzt. Dementsprechend sind auch die Tag-Felder des Cache um diese Information erweitert. – Eine Variante dieses Vorgehens ist das Erweitern der Tag-Information um das Statusbit S, womit Supervisor- und User-Prozesse im Cache gegeneinander abgegrenzt werden.

Address-Aliasing. Bild 6-31 zeigt schematisch zwei voneinander abhängige Prozesse, die sich einen gemeinsamen Datenbereich teilen und sich deswegen, anders als in Bild 6-30, im Virtuellen nicht überdecken. Wie wieder an den gestrichelten Pfeilen für eine Adresse dieses Datenbereichs gezeigt, haben hierbei

unterschiedliche virtuelle Adressen eine gemeinsame reale Entsprechung. Das *Kohärenzproblem* sei am Beispiel eines Write-through-Cache anhand von drei Zugriffen auf eine gemeinsame Variable X erklärt; es lässt sich mit derselben Zugriffsfolge auch für einen Copy-back-Cache konstruieren. Die Variable stehe zunächst nur im Hauptspeicher und werde vom Prozeß 1 mit der virtuellen Adresse V1 und vom Prozeß 2 mit der virtuellen Adresse V2 adressiert.

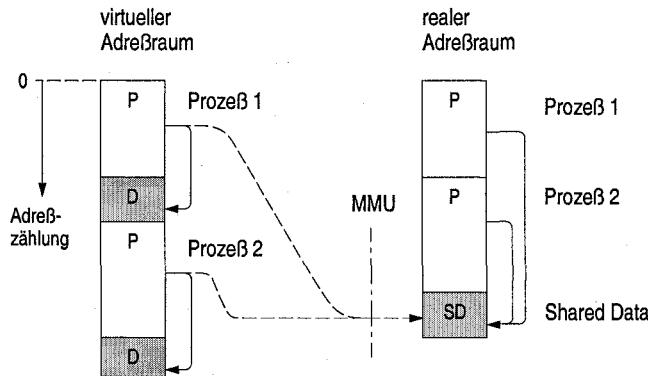


Bild 6-31. Schematische Darstellung zweier voneinander abhängiger Prozesse, bestehend aus Programmteil P und gemeinsamem Datenteil SD (shared data), ohne Überlappung im virtuellen Adressraum. Unterschiedliche virtuelle Datenadressen haben die gleiche reale Adresse (address aliasing)

1. Lesezugriff mit V1: Es wird ein Read-Miss angezeigt und die Variable X in die der Adresse V1 zugeordnete Cache-Zeile geladen.
2. Schreibzugriff mit V2: Es wird ein Write-Miss für die der Adresse V2 zugeordnete Cache-Zeile angezeigt und das Datum in den Speicher an die Stelle X geschrieben.
3. Lesezugriff mit V1: Es wird ein Read-Hit angezeigt, obwohl der Eintrag in der der Adresse V1 zugeordneten Cache-Zeile inzwischen veraltet ist.

Am einfachsten lässt sich dieses Problem dadurch lösen, daß man die gemeinsam benutzten Daten von einer Speicherung im Cache ausschließt, indem man den Datenbereich als *non-cacheable* kennzeichnet. Will man den Cache zumindest für jeweils diejenigen Zugriffe nutzen, die von einem der Prozesse aufeinanderfolgend ausgeführt werden, so sind folgende Maßnahmen zu ergreifen:

- a) Verwendung eines *direkt zuordnenden Cache*,
- b) Festlegung der virtuellen Adressen derart, daß die korrespondierenden Adressen unterschiedlicher Prozesse denselben Zeilenindex im Cache aufweisen, und
- c) Verwendung der Aktualisierungsstrategie Write-through-with-write-Allocation (siehe Tabelle 6-2, S. 418).

Der Leser möge das Funktionieren dieses Vorgehens anhand der oben aufgeführten drei Zugriffe selbst überprüfen.

Realer Cache. Bild 6-29 (S. 429) zeigt in Teilbild b ein System mit *realer Adressierung* des Cache, d.h. ein System, bei dem der Cache wie der Hauptspeicher mit den von der Speicherverwaltungseinheit erzeugten Adressen angewählt wird. Nachteil einer solchen Konfiguration ist zunächst, daß die Cache-Adressierung erst dann erfolgen kann, wenn die Speicherverwaltungseinheit ihre Adreßumsetzung abgeschlossen hat, d.h., wenn die reale Adresse zur Verfügung steht. Bei Mikroprozessoren mit On-chip-MMU kann dieser Nachteil vermieden werden, wenn die Adreßumsetzung mittels *Fließbandverarbeitung* im Vorgriff durchgeführt wird. Unabhängig davon kann bei teilassoziativen Caches (direkt zuordnend oder n -Wege-assoziativ) die durch den Zeilenindex bzw. den Satzindex erfolgende Cache-Anwahl parallel zur Adreßumsetzung der Speicherverwaltungseinheit ausgeführt werden, wenn der Index durch die von der Adreßumsetzung ausgenommenen Adreßbits gebildet wird.

Die Vorgehensweise bei der zweitgenannten Möglichkeit zeigt Bild 6-32a für eine Speicherverwaltungseinheit (MMU) mit Seitenverwaltung (siehe 6.3.2). Als Seitengröße wurden hier 4 Kbyte gewählt, so daß der von der Adreßumsetzung ausgenommene Anteil der virtuellen Adresse 12 Bits umfaßt. Da die vier niederwertigen Bits dieser sog. Bytenummer für die Byteanwahl innerhalb einer Cache-Zeile benötigt werden, verbleiben von den 12 Bits noch acht als Index für die Zeilen- oder Satzanwahl bei einem direkt zuordnenden bzw. einem n -Wege-assoziativen Cache. Die restlichen 20 Bits der virtuellen Adresse, die Seitennummer, werden für die Adreßumsetzung in der MMU herangezogen. Diese und die Zeilen- bzw. Satzanwahl im Cache erfolgen nun parallel. Ergebnis der Adreßumsetzung ist eine 20-Bit-Rahmennummer, das ist der Tag-Anteil der realen Adresse. Ergebnis der Cache-Anwahl ist die Auswahl einer Cache-Zeile bzw. eines Satzes, d.h. die Anwahl des bzw. der für den Vergleich erforderlichen Tag-Einträge. Mit dem anschließenden Tag-Vergleich wird die Cache-Adressierung abgeschlossen. – Nachteil dieses Vorgehens ist eine Begrenzung der Cache-Kapazität in der Anzahl der Zeilen beim direkt zuordnenden Cache bzw. der Anzahl der Sätze beim n -Wege-assoziativen Cache, da ja für die Index-Bildung nur Adreßbits der Bytenummer zur Verfügung stehen (siehe aber virtuell/realer Cache).

Bezüglich der *Datenkohärenz* beim realen Cache betrachten wir die beim virtuellen Cache genannten Problemfälle. DMA-Controller als zusätzliche Master verwenden üblicherweise ebenfalls reale Adressen, wodurch sich die im Problemfall mehrere Master auftretenden Kohärenzprobleme wie beim virtuellen Cache, Fall 2, lösen lassen: entweder durch Kennzeichnung der gemeinsamen Daten als *non-cacheable* oder durch Löschen des Cache mittels der Operation *Cache-Clear* bzw. *Cache-Flush* oder durch *Bus-Snooping*, z.B. mit dem *MESI-Kohärenzprotokoll*. Der Problemfall *Prozeßwechsel* tritt beim realen Cache nicht auf, da es

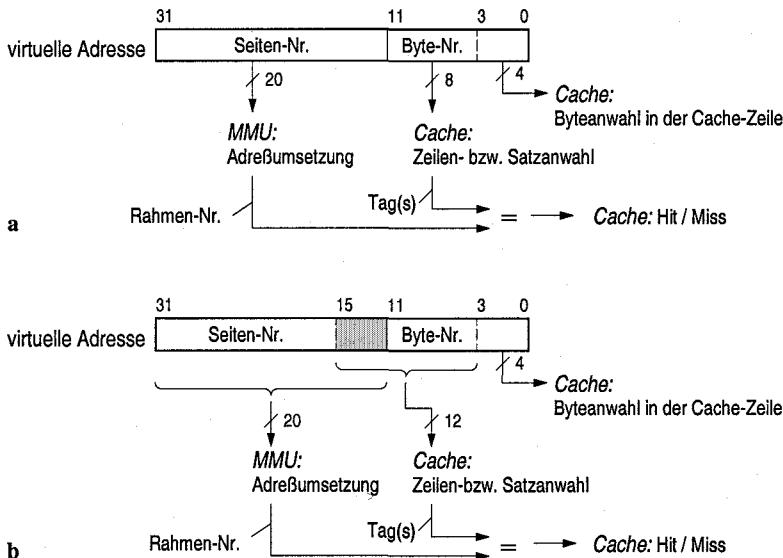


Bild 6-32. Parallel Auswertung der virtuellen Adresse durch die Speicherverwaltungseinheit und den Cache. a Realer Cache mit Zeilen- bzw. Satzanwahl durch einen Teil der von der Adreßumsetzung ausgenommenen Bytenummer (realer Zeilen-/Satzindex), b virtuell/realer Cache mit Zeilen- bzw. Satzanwahl durch Hinzunahme von Bits der Seitennummer (virtueller Zeilen-/Satzindex). In beiden Fällen Tag-Vergleich mit (realer) Rahmennummer

keine Kohärenzprobleme gibt. Die Abbildung einer bestimmten virtuellen Adresse auf die korrekte reale Entsprechung des gerade aktiven Prozesses wird von der Speicherverwaltungseinheit vorgenommen. Dazu wird ihr vom Betriebssystem beim Prozeßwechsel die für die Adreßumsetzungen jeweils gültige Tabelle bereitgestellt. Der Problemfall *Address-Aliasing* bei von mehreren Prozessen gemeinsam benutzten Daten gibt es beim realen Cache ebenfalls nicht, da in ihm (aufgrund der Auswahl mit realen Adressen) pro Datum nur ein Eintrag erzeugt wird, auf den alle Prozesse zugreifen.

Virtuell/realer Cache. In Abwandlung der Vorgehensweise beim realen Cache wird beim virtuell/realen Cache der Index für die Zeilen- bzw. Satzanwahl unabhängig von der Länge der Bytenummer gebildet, indem Bits der (virtuellen) Seitennummer zur denen der Bytenummer hinzugenommen werden, womit die Beschränkung der Cache-Kapazität aufgehoben ist. Für den Tag-Vergleich ist wie beim realen Cache dennoch die von der MMU erzeugte gesamte (reale) Rahmennummer heranzuziehen (realer Tag-Vergleich), wie in Bild 6-32b gezeigt. Man spricht bei diesem Vorgehen auch von virtueller Indizierung des Cache.

Bei dieser Variante kann die Cache-Anwahl wie beim realen Cache parallel zur Adreßumsetzung der MMU erfolgen, und es tritt das beim virtuellen Cache für den Prozeßwechsel *Kohärenzproblem* nicht auf. Allerdings gibt es

das Problem des *Address-Aliasing*, das aber gelöst werden kann, indem man die dem gemeinsam genutzten Speicherbereich entsprechenden virtuellen Teildreßräume so anlegt, daß die (virtuellen) Indizes für alle Caches einheitlich sind. Das erfordert für diese Teildreßräume ein Ausrichten (alignment) an den ganzzahligen Vielfachen der Adressraumgröße, wie sie sich aus der Adreßteillänge ergibt, beginnend bei Bit 0 bis hin zum höchstwertigen Adreßbit des virtuellen Index. Für das Snooping müssen zusätzlich zur realen Adresse auch die für die Indizierung genutzten *virtuellen* Adreßbits übertragen werden, wie dies z.B. bei Mehrprozessorsystemen mit dem aus den 1990er Jahren stammenden *hyperSPARC*-Prozessor und dem *MBus* als gemeinsamem Bus realisiert wurde (siehe hierzu auch das folgende Beispiel).

Beispiel 6.4. ► *hyperSPARC*. Bild 6-33 zeigt die Wirkungsweise des Controller-Bausteins RT625, mit dem der prozessorexterne Befehls-/Daten-Cache des *hyperSPARC* (RT620) gesteuert wird. Dieser Cache ist direkt zuordnend ausgelegt (direct mapped cache) und hat eine Speicherkapazität von 256 Kbyte, unterteilt in 4096 Cache-Zeilen zu je zwei 32-Byte-Blöcken. Die Blockgröße entspricht vier Datentransfers bei 64-Bit-breitem Prozessordatenbus. Unterschieden werden die beiden Blöcke einer Cache-Zeile durch das Adreßbit 5, d.h., im Hauptspeicher liegen sie unmittelbar hintereinander. Das Bild zeigt den Tag-Speicher und die Adressierungsgänge, zum einen für den zugehörigen Prozessor mit Angabe der virtuellen Adresse (rechts oben im Bild), zum andern für einen der anderen Prozessoren in einem Mehrprozessorsystem, wobei die für das Snooping erforderlichen Adreßangaben über den *MBus* übertragen werden (links oben im Bild). Im ersten Fall erfolgt die Zeilenanwahl durch die zwölf Bits 17 bis 6 der virtuellen Adresse (Cache Line Select). Parallel dazu werden die 20 virtuellen Adreßbits 31 bis 12 von der Speicherverwaltungseinheit in

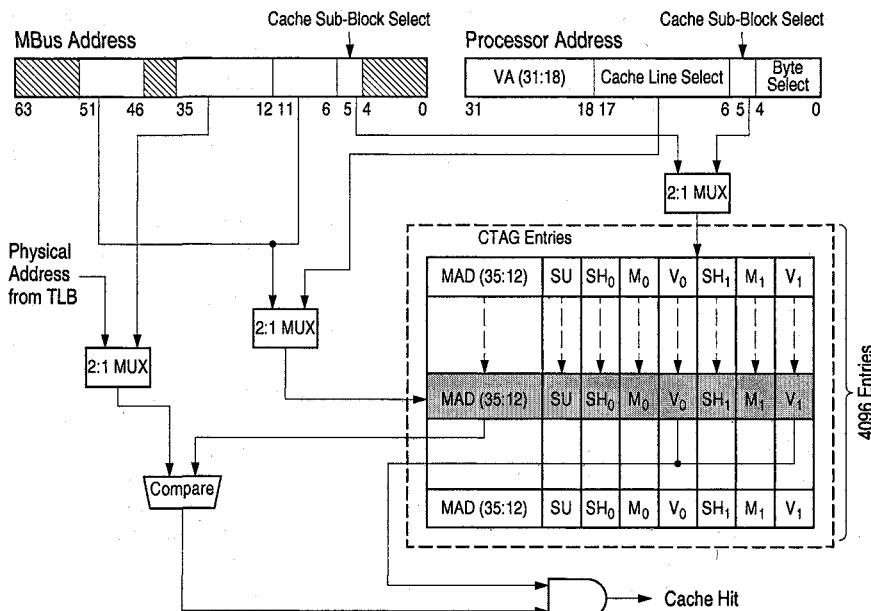


Bild 6-33. Virtuell/realer Off-chip-Cache des RISC-Prozessors *hyperSPARC* (nach [Ross 1993])

24 reale Adressbits umgesetzt (Erweiterung auf eine 36-Bit-Adresse) und dann für den Tag-Vergleich herangezogen (Physical Address from TLB). Das heißt, die sechs Adressbits 17 bis 13 werden einerseits zur „virtuellen“ Zeilenanwahl und andererseits für den „realen“ Tag-Vergleich verwendet, was den Cache als virtuell/realen Cache charakterisiert. Für das Snooping werden über den MBus, um die Zeilenanwahl „virtuell“ durchführen zu können, zusätzlich zu den realen Adressbits 35 bis 5 auch die virtuellen Adressbits 17 bis 13 übertragen. Benutzt werden hierfür die Bitpositionen 51 bis 46 des Busses. Zur Vermeidung des Address-Aliasing-Problems müssen die virtuellen Adressräume der einzelnen Prozessor/Cache-Paare an ganzzahligen Vielfachen von 256 K beginnen, was dadurch begründet ist, daß die virtuellen Adressen bis hin zu Bit 17 für die Cache-Indizierung genutzt werden. ▶

Fazit. Zusammengefaßt ist festzustellen, daß der reale Cache bzgl. der Datenkohärenz wesentlich einfacher zu handhaben ist als der virtuelle Cache. Der mit ihm zunächst verbundene Nachteil einer verzögerten Cache-Adressierung kann aufgehoben werden, indem die Adressumsetzung der Speicherverwaltungseinheit durch zusätzliche Fließbandstufen abgefangen wird oder die Cache-Indizierung parallel zur Adressumsetzung durchgeführt wird. Insofern sind Prozessoren, bei denen die Speichereinheiten integriert sind (je eine für Befehls- und für Datenbereiche), vorwiegend mit realen Caches ausgestattet. In einer Erweiterung durch Hinzunahme virtueller Adressbits für die Cache-Indizierung ist auch der virtuell/reale Cache gebräuchlich mit den für ihn typischen Nachteilen, wie Address-Aliasing. Virtuelle Caches findet man hingegen fast nur noch bei älteren Prozessoren bzw. älteren Rechnersystemen.

6.3 Speicherverwaltungseinheiten

Bei Mikroprozessorsystemen mit Mehrprogrammbetrieb können mehrere Benutzerprogramme oder mehrere z.B. durch Interrupts ausgelöste Prozesse quasiparallel, d.h. zeitlich ineinander verzahnt, ausgeführt werden (*multiprogramming, multitasking*). Um einen schnellen Programmwechsel zu gewährleisten, müssen dazu alle beteiligten Programme (einschließlich ihrer Daten) im Hauptspeicher geladen sein. Da dies jedoch bei vielen Anwendungen eine sehr große Hauptspeicherkapazität erfordern würde, werden zur Kapazitätserweiterung Hintergrundspeicher, meist Festplattspeicher, eingesetzt und diese in ihrem Zugriff so organisiert, daß sie einen Hauptspeicher entsprechend großer Kapazität vorspiegeln (Stichwort *Seitenverwaltung*). Man bezeichnet den so verfügbaren Speicherraum deshalb auch als virtuellen Speicher und spricht von *virtueller Speicherverwaltung*.

Das Laden der Programme und ihrer Daten in den realen Hauptspeicher besorgt das Betriebssystem. Es führt dazu über die freien Speicherbereiche Buch und lagert bei nicht ausreichendem *Freispeicherplatz* Speicherinhalte, sofern sie gegenüber ihrem Original auf dem Hintergrundspeicher verändert worden sind, auf diesen aus. Programme und Daten müssen, um die jeweils freien Bereiche

nutzen zu können, lageunabhängig, d.h. im Speicher *verschiebbar* (relocatable) sein. Grundsätzlich kann die Verschiebbarkeit dadurch erreicht werden, daß vor oder während des Ladevorgangs zu sämtlichen (*relativen Adressen*) im Programm die aktuelle Ladeadresse (Basisadresse) addiert wird, oder daß die eigentlichen (realen) Adressen zur Programmlaufzeit durch befehlzählerrelative und basis-relative Adressierung bezüglich ihrer Programm- bzw. ihrer Datenzugriffe gebildet werden (siehe 3.1.3). – Das Auswechseln von Seiten im Hauptspeicher bezeichnet man als *Swapping*. Im ungünstigsten Fall, in dem sich Seiten im Speicher gegenseitig verdrängen und das System vorwiegend mit Seitenwechseln beschäftigt ist, spricht man von *Trashing* (Abfall erzeugen).

Gebräuchlicher ist jedoch die Verwendung einer *Speicherverwaltungseinheit* (memory management unit, *MMU*). Auch sie führt die Adreßumsetzung zur Laufzeit eines Programms durch, indem sie die vom Prozessor erzeugten sog. *virtuellen (logischen Adressen* in *reale (physikalische Adressen*, d.h. Hauptspeicheradressen, umsetzt (siehe z.B. Bild 6-29, S.429). Die hierfür erforderliche Abbildungsinformation (*memory map*) stellt das Betriebssystem in Form von einer oder mehreren Umsetzungstabellen zur Verfügung. Dadurch, daß sich diese Tabellen immer wieder an neue Speicherbelegungen anpassen lassen, ergibt sich mit dieser Technik eine hohe Flexibilität. Sie hat außerdem den Vorteil, daß sie sich relativ leicht mit der Funktion des Zugriffsschutzes verbinden läßt, womit eine höhere Systemsicherheit erreicht wird.

Der Einsatz einer Speicherverwaltungseinheit bedeutet allerdings einen erhöhten Aufwand für ein Rechnersystem, da die Adreßumsetzung und die Zugriffsüberwachung den Adressiervorgang möglichst nicht verzögern sollen und deshalb ganz durch Hardware realisiert oder wenigstens durch Hardware unterstützt werden. Um den schnellen Zugriff auf die Umsetzungstabellen zu gewährleisten, müssen diese in der MMU gespeichert werden. Bei MMUs, die nur eine kleine Tabelle benötigen, wird diese vollständig in einem Registerspeicher der MMU gehalten, bei MMUs mit einer großen Tabelle oder mit sehr vielen kleinen Tabellen wird der jeweils aktuelle Tabellenausschnitt in einem sog. *Deskriptor-Cache* gepuffert (translation look-aside buffer, *TLB*).

Um den Speicherverwaltungsaufwand gering zu halten, bezieht man die Abbildungsinformation nicht auf einzelne Adressen, sondern auf zusammenhängende Adreßbereiche. Dazu gibt es zwei grundsätzliche Möglichkeiten, die *Segmentverwaltung* und die *Seitenverwaltung*. Bei der Segmentverwaltung werden die Bereiche so groß gewählt, daß sie logische Einheiten, wie Programmcode, Daten oder Stack, vollständig umfassen, sie haben dementsprechend variable Größen. Bei der Seitenverwaltung wird eine logische Einheit in Seiten einheitlicher Länge unterteilt, die Bereiche haben dementsprechend eine feste Größe. Für den Zugriffsschutz (*Speicherschutz*) werden die Segmente bzw. Seiten mit Zugriffsattributen versehen, die ebenfalls in den Umsetzungstabellen gespeichert und von der MMU bei der Adreßumsetzung unter Bezug auf die vom Prozessor erzeugten

Statussignale ausgewertet werden. Adreßumsetzungsinformation, Schutzattribute und zusätzliche Statusangaben eines Bereichs werden in einem oder mehreren Worten zusammengefaßt und als dessen *Deskriptor* bezeichnet.

6.3.1 Segmentverwaltung (segmentation)

Unterteilung der virtuellen Adresse. Das Bilden von Segmenten hat eine Strukturierung des *virtuellen Adreßraums* wie auch des *realen Adreßraums*, d.h. des Hauptspeichers zur Folge. Im Virtuellen erreicht man sie z.B. durch Unterteilung des virtuellen Adreßworts in eine Segmentnummer als Kennung eines Segments und in eine Bytenummer als Abstand zum Segmentanfang. Die Segmentnummer wird dabei aus den n höherwertigen Adreßbits (z.B. $n = 8$), die Bytenummer aus den verbleibenden m niederwertigen Adreßbits gebildet ($m = 24$ bei einem 32-Bit-Adreßwort). Dementsprechend ist die maximale virtuelle Segmentanzahl gleich 2^n (256) und die maximal mögliche Segmentgröße gleich 2^m (16 Mbyte). Der virtuelle Adreßraum wird demnach in lauter Bereiche der Größe 2^m aufgeteilt.

Haben die Segmente eine geringere Größe als 2^m , so gilt der ungenutzte Raum eines solchen Bereichs als Verschnitt. Dieser ist insofern hinnehmbar, als er bei Speicherung dieser kleineren logischen Einheiten auf einem Hintergrundspeicher keinen Speicherplatz benötigt. Die Adressierung dort erfolgt nämlich durch die virtuelle Adresse nicht unmittelbar, sondern auf dem Umweg über eine Adreßtabelle, die die eigentlichen, gerätespezifischen Adreßangaben enthält, z.B. Zylindernummer, Kopfnummer und Sektornummer bei einem Festplattenspeicher (siehe 8.3.2). Dabei werden von einem Segment nur so viele Sektoren des Hintergrundspeichers belegt, wie es seine tatsächliche Größe erfordert.

Eine gute Strukturierung des realen Adreßraums, d.h. eine gute Hauptspeichernutzung, enthält man, wenn man die Segmentgrenzen an jeder Byteadresse zuläßt. Dann nämlich können Segmente im Prinzip lückenlos aufeinanderfolgend gespeichert werden, so daß jedes Segment einen gerade so großen Speicherbereich belegt, wie es Bytes umfaßt. (Alternativ dazu könnten die Segmentgrenzen auf Wort- oder Cache-Block-Grenzen beschränkt werden.)

Bild 6-34 zeigt die Wirkungsweise einer solchen MMU für einen Mikroprozessor mit einem 32-Bit-Adreßwort und einer virtuellen Segmentnummer von 8 Bits. Die Adreßumsetzung erfolgt über eine *Segmenttabelle* (segment table) mit maximal 256 *Segmentdeskriptoren* als Einträge. Jeder Deskriptor beschreibt ein Segment durch eine reale 32-Bit-Segmentbasisadresse, eine 24-Bit-Segmentlängenangabe und, im Bild nicht gezeigt, durch Zugriffsattribute und Statusinformation; die Deskriptoranwahl erfolgt durch die virtuelle Segmentnummer. Die reale Adresse ergibt sich aus der Segmentbasisadresse, zu der die Bytenummer addiert wird. Gleichzeitig mit der Adreßumsetzung wird die Bytenummer mit dem Seg-

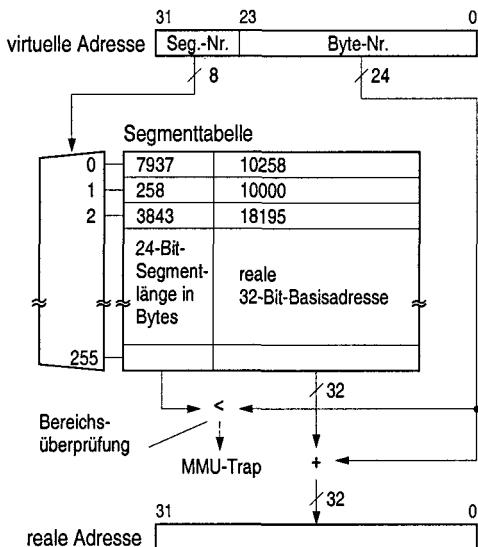


Bild 6-34. Adressumsetzung bei Segmentierung mittels Unterteilung der virtuellen Adresse; Segmentbasisadressen im Hauptspeicher an Bytegrenzen. Überprüfung auf Segmentüberschreitung durch Vergleich von Bytenummer und Segmentlängeneintrag. – Die eingetragenen Nummern beziehen sich auf das Beispiel in Bild 6-35

mentlängeneintrag der Tabelle verglichen, um segmentüberschreitende Speicherzugriffe feststellen und ggf. verhindern zu können. Im Falle einer *Segmentüberschreitung* bricht die MMU den Umsetzungsvorgang ab und zeigt dies dem Prozessor durch ein Trap-Signal an (MMU-Trap). – Ein Segment kann in diesem Beispiel bis zu $2^{24} = 16$ Mbyte umfassen. Die Segmenttabelle wird aufgrund ihres geringen Umfangs in einem Registerspeicher der MMU gehalten.

Bild 6-35 zeigt die Strukturierung des *virtuellen* und des *realen Adreßraums* am Beispiel dreier Segmente mit den in der Segmenttabelle in Bild 6-34 angegebenen Basisadressen und Segmentlängen. Die Zuordnung der virtuellen Segmentnummern zu den in der Segmenttabelle gespeicherten realen Segmentbasisadressen ist durch Pfeile mit Nummern- bzw. Adressangaben dargestellt. Das Bild zeigt außerdem die möglichen Segmentgrenzen, die sich im virtuellen Adreßraum aus den Vielfachen von 16 Mbyte und im Hauptspeicher aus den Vielfachen von Bytes ergeben. Der im Virtuellen entstehende Verschnitt ist durch die grau unterlegten Lücken zwischen den Segmenten dargestellt.

Anmerkung. Im Prinzip könnte man die Strukturierung des realen Adreßraums in gleicher Weise wie die des virtuellen Adreßraums vornehmen, indem man das reale Adreßwort in eine n -Bit-Segmentnummer und eine m -Bit-Bytenummer unterteilt. Für die MMU würde die Umsetzung einer virtuellen in eine reale Adresse dann darin bestehen, die virtuelle Segmentnummer durch die ihr durch die Hauptspeicherbelegung zugeordnete reale Segmentnummer zu ersetzen. Im Hauptspeicher könnten dann aber Segmente immer nur an den Vielfachen von 2^m gespeichert werden, und sie würden immer Bereiche der festen Größe von 2^m Bytes belegen. Das heißt, der im Virtuellen

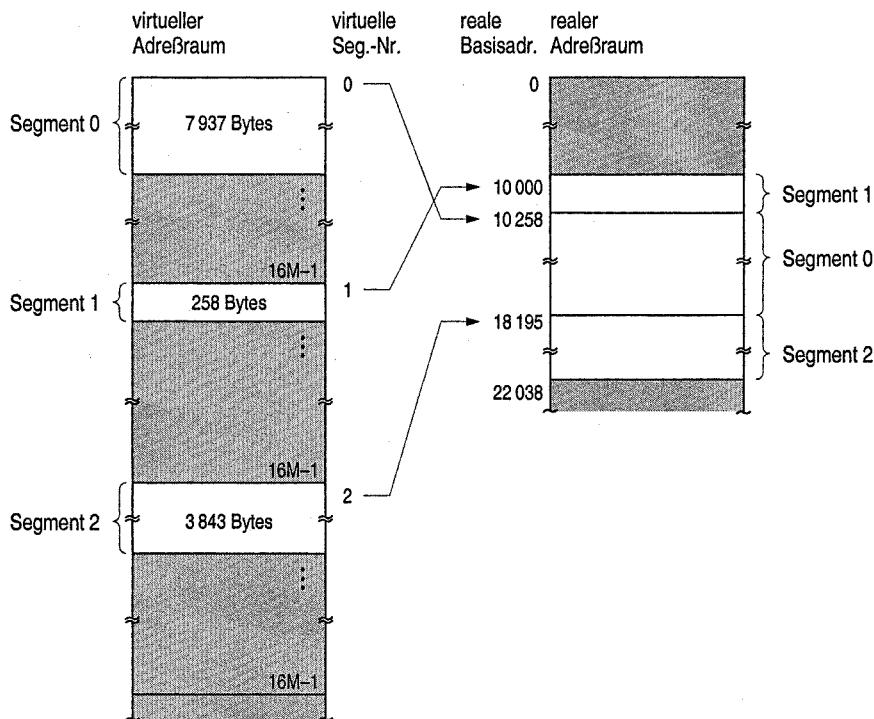


Bild 6-35. Abbildung dreier Segmente auf den realen Adressraum bei einer Segmentierung nach Bild 6-34 und mit den dort eingetragenen Basisadressen und Segmentlängen

auftretende Verschnitt wäre auch im Realen vorhanden, was insofern nicht akzeptabel ist, als man den kostenintensiven Hauptspeicherplatz möglichst lückenlos nutzen möchte.

Erweiterung der virtuellen Adresse. Die oben beschriebene Segmentverwaltung hat den Nachteil, daß die maximal mögliche Segmentgröße gegenüber dem vorhandenen Adressraum eingeschränkt ist, da ein Teil der virtuellen Adresse für die Segmentkennzeichnung benutzt wird. Dieser Nachteil läßt sich beheben, wenn man der MMU die Segmentnummer gesondert bereitstellt, so daß die gesamte virtuelle Adresse als Bytenummer zur Verfügung steht. Hierzu wird die MMU mit mehreren Segmentnummerregistern ausgestattet, die vom Betriebssystem mit den jeweils aktuellen Segmentnummern geladen werden. Die Anwahl eines solchen Registers bei der Adreßumsetzung erfolgt durch den vom Prozessor angezeigten Zugriffsstatus: Programm-, Stack- oder Datenzugriff. Das heißt, die Segmente werden nach Programmcode-, Stack- und Datenbereichen unterschieden.

Bild 6-36 zeigt die Wirkungsweise einer solchen MMU. Ausgehend von einem 32-Bit-Adreßwort enthält die *Segmenttabelle* neben den Segmentlängenangaben die 32-Bit-Segmentbasisadressen, zu denen die vollständige virtuelle Adresse als

Bytenummer addiert wird. Jedem Segment steht somit der gesamte virtuelle Adressraum von 2^{32} Adressen zur Verfügung, d.h., ein Segment kann im Prinzip den gesamten realen Adressraum nutzen. Die Segmentgrenzen im realen Adressraum liegen an Vielfachen von Bytes. Betrachtet man die Adreßerweiterung als Teil der virtuellen Adresse, so erhält man einen entsprechend vergrößerten *virtuellen Adressraum*, in dem jedes Segment auch bei maximaler Ausdehnung nur einen Bruchteil des Raumes belegt und in dem die Segmentgrenzen an Vielfachen der maximal möglichen Ausdehnung liegen. – Im Bild wurde eine etwas speziellere Darstellung als in Bild 6-34 gewählt, indem die Pfeile den Abbildungsvorgang für ein bestimmtes Segment i zeigen.

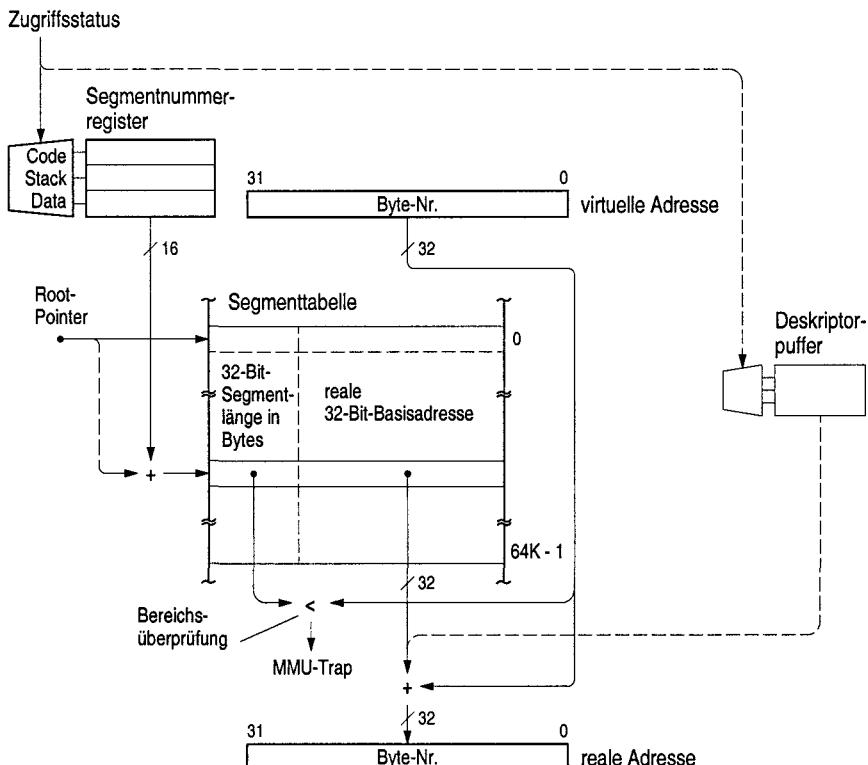


Bild 6-36. Adressumsetzung bei Segmentierung mittels Erweiterung der virtuellen Adresse; Segmentbasisadressen im Hauptspeicher an Bytegrenzen. Überprüfung auf Segmentüberschreitung durch Vergleich von Bytenummer und Segmentlängeneintrag

Aufgrund der 16-Bit-Segmentnummer ist die Segmenttabelle jetzt zu groß, um sie in einem Registerspeicher der MMU zu speichern. Sie steht statt dessen im Hauptspeicher, und ihre Basisadresse, der sog. *Root-Pointer*, wird vom Betriebssystem in einem speziellen MMU-Register vorgegeben. Um dennoch die Adressumsetzung nicht durch Hauptspeicherzugriffe zu verzögern, wird mit dem

Laden eines jeden der Segmentnummerregister gleichzeitig auch der zugehörige Deskriptor in ein dem Segmentnummerregister zugeordnetes Pufferregister der MMU eingetragen. Bei einem Segmentzugriff erfolgen dann die Adreßumsetzung und die Überprüfung der Speicherschutzbedingungen unmittelbar mit Hilfe dieses Registereintrags. (Der *Deskriptorpuffer* ist aus Platzgründen deutlich schmäler gezeichnet als zum Rest des Bildes passend.) – Realisiert ist diese Art der Segmentverwaltung bei den On-chip-MMUs der x86-Prozessoren *Pentium* (Intel) und *Athlon* (AMD). Sie haben sechs Segmentnummerregister und die Segmentnummern umfassen 13 von 16 Bits, wodurch sich Segmenttabellen mit bis zu 8 K Segmentdeskriptoren ergeben. Eines der verbleibenden drei Bits wird zur Unterscheidung sog. globaler und lokaler Tabellen herangezogen (global und local descriptor tables, GDT, LDT).

Anmerkung. Bei der Addition eines Segmentregisterinhalts zum Root-Pointer muß berücksichtigt werden, daß die so adressierten Segmenttabelleneinträge jeweils mehr als ein Byte umfassen. Dementsprechend muß der Inhalt des Segmentregisters für die Addition mit der entsprechenden Byteanzahl multipliziert, d.h. um eine entsprechende Bitanzahl linksgeshiftet werden, was in Bild 6-36 nicht gezeigt ist. Dieser Sachverhalt gilt auch für die folgenden Strukturbilder von Speicher-verwaltungseinheiten, und zwar immer dann, wenn Tabellenadressen gebildet werden.

Lineare und symbolische Segmentierung. Wie beschrieben, überprüft die MMU bei jeder Adreßumsetzung, ob die Bytenummer der virtuellen Adresse innerhalb der Segmentlängenangabe der Segmenttabelle liegt. Sie geht dabei davon aus, daß ihr der Prozessor die korrekte Segmentnummer liefert. Arbeitet der Prozessor mit sog. *linearer Segmentierung* (linear segmenting), so können sich diesbezüglich bei MMUs, bei denen die Segmentnummer durch Adreßunterteilung gebildet wird (Bild 6-34), Probleme ergeben. Bei linearer Segmentierung nämlich kennt der Prozessor keine Unterteilung der virtuellen Adresse in Segmentnummer und Bytenummer und beeinflußt dementsprechend bei Adreßrechnungen das gesamte Adreßwort, z.B. bei der indizierten Adressierung. Ändert sich dabei der von der MMU als Segmentnummer interpretierte Adreßanteil, so bedeutet dies eine Segmentüberschreitung. Diese kann von der MMU nur dann erkannt werden, wenn das dadurch fälschlicherweise angewählte Segment in der Segmenttabelle, da nicht zum aktiven Prozeß gehörig, als z.Z. nicht anwählbar bzw. nicht verfügbar gekennzeichnet ist. Dazu werden die Deskriptoren in der Segmenttabelle durch Statusattribute dieser Art ergänzt.

Das genannte Problem tritt nicht auf, wenn der Prozessor mit sog. *symbolischer Segmentierung* arbeitet (symbolic segmenting). Dann nämlich beeinflußt er bei den Adreßrechnungen nicht die gesamte virtuelle Adresse, sondern nur den als Bytenummer ausgewiesenen Adreßteil. So tritt das Problem grundsätzlich auch bei MMUs nicht auf, bei denen die Segmentnummer durch Adreßerweiterung gebildet wird (Bild 6-36), da hier ja die gesamte virtuelle Adresse als Bytenummer ausgewiesen ist.

Vor- und Nachteile. Das Festmachen der Speicherverwaltung an den logischen Einheiten, d.h. an großen zusammenhängenden Bereichen, wie Programmcode, Datenbereiche und Stackbereiche, bietet folgende Vorteile:

- Änderungen der sie beschreibenden Angaben, nämlich Basisadresse, Segmentlänge, Zugriffsattribute und Statusangaben, können mit wenig Aufwand vorgenommen werden, da sie immer nur den zugehörigen Segmentdeskriptor, also nur einen einzigen Tabelleneintrag betreffen.
- Die Umsetzungstabellen sind klein, da die Anzahl an Segmenten naturgemäß gering ist.
- Segmente können so in den Hauptspeicher geladen werden, daß sie sich teilweise oder ganz überlappen. Auf diese Weise können Speicherinhalte (Programme oder Daten) mehreren Nutzern zugänglich gemacht werden (*shared code, shared data*).

Die segmentbezogene Speicherverwaltung hat allerdings auch einige gewichtige Nachteile:

- Beim Einlagern von Programmen/Daten in den Hauptspeicher sowie bei deren Auslagerung auf den Hintergrundspeicher müssen immer Segmente als Ganzes transportiert werden, auch wenn für einen bestimmten Zeitraum nur Teile davon im Hauptspeicher benötigt werden.
- Das Ein-/Auslagern (*swapping*) führt zu einer „Zerstückelung“ des Hauptspeichers in belegte und freie Bereiche, die wegen der unterschiedlichen Segmentgrößen in ihren Größen ebenfalls variieren. Die dadurch erforderliche *Freispeicherverwaltung*, d.h. das Buchführen über die freien Bereiche und das Zuweisen eines Bereichs geeigneter Größen an ein zu ladendes Segment führt zu einem zusätzlichen Verwaltungsaufwand im Betriebssystem. Strategien hierzu sind z.B. in [Tanenbaum 2002] beschrieben.
- Die Zerstückelung kann schließlich dazu führen, daß für ein Segment kein ausreichender zusammenhängender Speicherplatz zur Verfügung steht, obwohl er in der Summe der freien Bereiche vorhanden ist. Zu lösen ist dieses Problem ggf. dadurch, daß das Betriebssystem die belegten Bereiche im Speicher zusammenschiebt und so die freien Bereiche vereinigt (*garbage collection*), was wiederum mit erheblichem Zeitaufwand verbunden ist.

6.3.2 Seitenverwaltung (paging)

Bei der Seitenverwaltung wird der bei der Segmentverwaltung vorhandene Bezug auf logische Einheiten zugunsten einer möglichst guten Speichernutzung aufgegeben. Dazu wird der *virtuelle Adressraum* in kleinere Bereiche einheitlicher Größe, sog. *Seiten* (*pages*), unterteilt, die im Hauptspeicher auf entsprechende Rahmen (*frames*) gleicher Größe abgebildet werden. Typische Seitengrößen sind

512 Bytes, 1 Kbyte, 2 Kbyte, 4 Kbyte und 8 Kbyte. Die Adreßumsetzung erfolgt dadurch, daß die höherwertigen Adreßbits als virtuelle Seitennummer auf eine (reale) Rahmennummer abgebildet und die niederwertigen Adreßbits als Byte-nummer unverändert übernommen werden (Konkatenation). Der virtuelle Adreßraum wird somit in Seiten unterteilt, die in beliebige freie, nicht notwendigerweise zusammenhängende Seitenrahmen geladen werden können.

Bild 6-37 zeigt die Wirkungsweise einer MMU mit Seitenverwaltung bei 32-Bit-Adressen und einer Seitengröße von 4 Kbyte. Die Adreßumsetzung erfolgt über eine *Seitentabelle (page table)*, die aufgrund der 20-Bit-Seitennummer bis zu 2^{20} *Seitendeskriptoren* umfassen kann. Jeder der Deskriptoren besteht aus einer Rahmennummer und, im Bild nicht gezeigt, Zugriffsattributen und Statusinformation für die betreffende Seite. Um die Funktion der MMU von Tabellenzugriffen im Speicher weitgehend unabhängig zu machen, besitzt die MMU einen *Deskriptor-Cache (TLB)*, der den aktuellen Ausschnitt der Seitentabelle, d.h. die Deskriptoren der jeweils zuletzt benutzten Seiten enthält. Zugriffe auf die Tabelle im Speicher erfolgen nur, wenn der entsprechende Eintrag im Cache nicht vorhanden

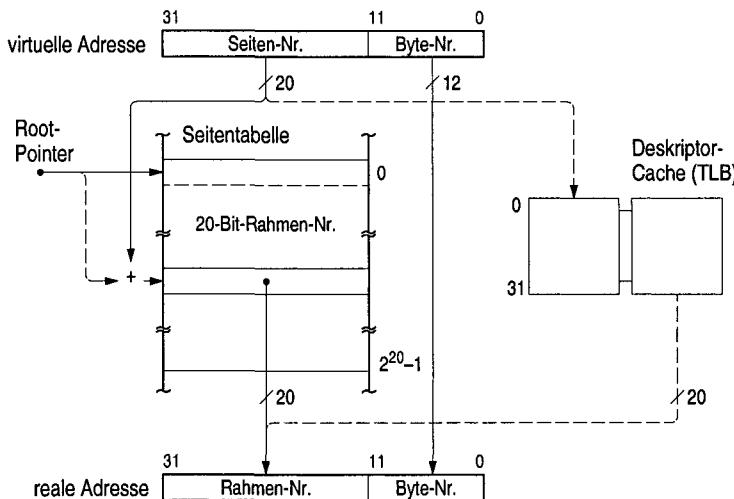


Bild 6-37. Adreßumsetzung bei Seitenverwaltung

ist; hierbei wird der Cache aktualisiert. Im Bild ist dieser Cache als *vollassoziativer Cache* für 32 Adreßabbildungen ausgelegt (TLBs heutiger Prozessoren arbeiten vollassoziativ oder n -Wege-assoziativ und haben 32, 64 oder 128 Cache-Zeilen). Die Seitennummer der virtuellen Adresse bildet die Tag-Information, die Rahmennummer sowie die Schutz- und Statusangaben den Cache-Eintrag. (Der Cache ist aus Platzgründen deutlich schmäler gezeichnet als zum Rest des Bildes passend.)

Bild 6-38 zeigt die Strukturierung des virtuellen und des realen Adreßraums am Beispiel eines Segmentes, das im virtuellen Adreßraum die zusammenhängenden Seiten 0 bis 3 belegt und entsprechend den Pfeilen auf die Rahmen 3, 6, 2 und 1 des Hauptspeichers abgebildet wird. Die Pfeile spiegeln dementsprechend die Adreßabbildungsinformation der Umsetzungstabelle, der Seitentabelle, wider.

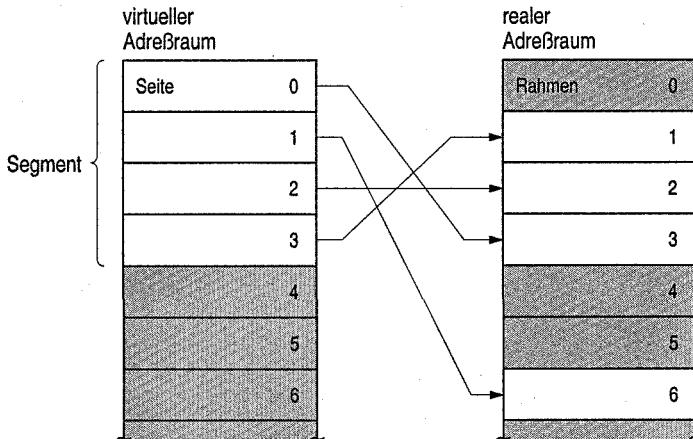


Bild 6-38. Abbildung eines vier Seiten umfassenden Segmentes auf vier Rahmen des Hauptspeichers bei einer Seitenverwaltung nach Bild 6-37

Vor- und Nachteile. Verglichen mit der Segmentverwaltung kehren sich die dortigen Nachteile in Vorteile um:

- Die in Seiten unterteilten logischen Einheiten können jetzt nichtzusammenhängend gespeichert werden, so daß der Speicherraum optimal nutzbar ist.
- Die *Freispeicherverwaltung* läßt sich aufgrund der immer gleich großen Seiten/Rahmen sehr viel einfacher handhaben.
- Diese Technik ermöglicht es, nur jene Seiten einer logischen Einheit im Hauptspeicher bereitzuhalten, die zur Programmausführung aktuell benötigt werden (*working set*), wodurch der Speicherplatzbedarf reduziert und das Swapping weniger häufig wird. Der Prozessor muß dann jedoch so ausgelegt sein, daß er eine Zugriffsoperation auf eine nicht geladene Seite (nach einer Fehlermeldung durch die MMU, *page fault trap*) unterbrechen und zu einem späteren Zeitpunkt, nachdem die Seite vom Betriebssystem in den Hauptspeicher geladen wurde, wieder aufnehmen kann (*demand paging*). In diesem Vorgehen ist letztlich der Begriff „*virtueller Speicher*“ begründet, da jetzt der Hauptspeicher eine kleinere Kapazität haben kann, als der Adreßumfang der aktuell aktivierten logischen Einheiten ausmacht.

Allerdings führt die feine Unterteilung des virtuellen und des realen Adreßraums auch zu Nachteilen, und zwar an den Stellen, an denen die Segmentverwaltung Vorteile hat:

- Änderungen der Angaben, die eine logische Einheit insgesamt betreffen (z.B. Zugriffsattribute), müssen jetzt ggf. in vielen Seitendeskriptoren durchgeführt werden, und zwar in so vielen, wie die logische Einheit Seiten hat.
- Ein Überlappen von Seiten ist aufgrund der bei der Adressumsetzung angewandten Konkatenation nicht möglich.
- Gegenüber der Segmentverwaltung wird die Umsetzungstabelle wesentlich umfangreicher. Bei einer virtuellen 32-Bit-Adresse und einer Seitengröße von 4 Kbyte kann sie, wie bereits erwähnt, bis zu 2^{20} Einträge (Seitendeskriptoren) umfassen. Bei einer Deskriptorlänge von 4 Bytes sind das 4 Mbyte! Hier benötigt man einerseits einen Cache für die aktuellen Deskriptoren; andererseits müssen Teile der Seitentabelle auf einen Hintergrundspeicher ausgelagert werden, um den Hauptspeicher nicht zu sehr zu belasten.

Invertierte Seitentabelle. Der Nachteil einer sehr großen Seitentabelle kann vermieden werden, indem man die Tabelle entgegen dem obigen Vorgehen nicht für den virtuellen Adressraum, sondern als sog. *invertierte Seitentabelle* lediglich für die Anzahl der im Hauptspeicher insgesamt ladbaren Seiten, also für den realen Adressraum auslegt. Adressiert wird sie, wie Bild 6-39 zeigt, indem die Seitennummer der virtuellen Adresse einer Hash-Funktion unterworfen und so auf den jetzt kleineren Tabellenadressraum abgebildet wird. Da es dabei zur Kollision kommen kann, nämlich wenn unterschiedliche Seitennummern auf denselben Tabellenplatz verweisen, muß in der Seitentabelle zusätzlich zum eigentlichen Seitendeskriptor auch die zugehörige Seitennummer gespeichert sein und diese

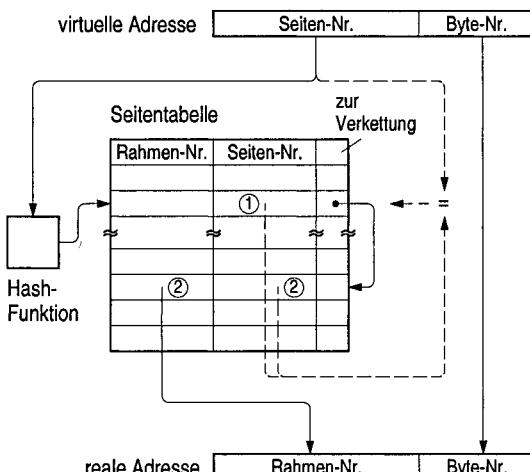


Bild 6-39. Adressumsetzung mit invertierter Seitentabelle. (1) Zugriff auf einen Eintrag mit ungleicher Seitennummer (Kollision), (2) Zugriff auf den damit verketteten Eintrag mit gleicher Seitennummer, d.h. Auslesen der Rahmennummer

beim Tabellenzugriff mit der Seitennummer der virtuellen Adresse verglichen werden (Prinzip der Cache-Adressierung). Darüber hinaus müssen Mechanismen zur Kollisionsreduktion, wie Verketten von Tabellenplätzen (siehe Bild) oder Verwenden mehrerer Hash-Funktionen, implementiert werden. – Die Technik der invertierten Seitentabelle wird z.B. beim *Itanium 2* eingesetzt.

Fazit. Im Hinblick auf die Effizienz wird die reine Seitenverwaltung im Grunde nur noch bei Prozessoren mit kürzeren Adressen von z.B. 16 Bits eingesetzt, bei denen eine handhabbare Anzahl an Seiten gegeben ist. Heutige 32- und 64-Bit-Prozessoren verwenden die Seitenverwaltung grundsätzlich als letzte Stufe einer mehrstufigen Adreßumsetzung, nämlich in Verbindung mit den oben beschriebenen Segmentverwaltungstechniken. Auf diese Weise kommen sowohl die Vorteile der Segmentverwaltung als auch die der Seitenverwaltung zum Tragen, und die Nachteile des einzelnen Verfahrens werden aufgehoben. Siehe dazu 6.3.3.

6.3.3 Mehrstufige Speicherverwaltung

Zweistufige Adreßumsetzung. Um einerseits Segmente als logische Einheiten verwalten und andererseits diese in kleineren Einheiten transportieren und „beliebig“ speichern zu können, um also Vorteile von Segmentverwaltung und Seitenverwaltung zu kombinieren, verbindet man beide Speicherverwaltungstechniken und erhält so eine zweistufige Adreßumsetzung. Den Ausgangspunkt bildet eine Segmenttabelle. Sie enthält für jedes Segment einen Deskriptor, der aus Zugriffsattributen und Statusangaben für dieses Segment und einem Zeiger auf eine Seitentabelle besteht. Die Seitentabelle enthält wiederum die Deskriptoren aller zu diesem Segment gehörenden Seiten, d.h. Zugriffsattribute und Statusangaben für jede Seite und die Rahmennummern der Seiten.

Im Gegensatz zur reinen Seitenverwaltung mit nur einer einzigen sehr großen Seitentabelle gibt es jetzt, bei der Kombination mit der Segmentverwaltung, sehr viele, jedoch kleinere *Seitentabellen*, von denen (neben der *Segmenttabelle*) nur die aktuellen im Hauptspeicher gehalten zu werden brauchen. Die anderen Seitentabellen, die ggf. wieder in Seiten organisiert sind und der Seitenverwaltung unterliegen, können auf einem Hintergrundspeicher stehen und werden dann bei Bedarf geladen. Die Information, ob eine Seitentabelle im Hauptspeicher präsent ist oder nicht, wird im zugehörigen Segmentdeskriptor geführt. Diese Möglichkeit des Speicherns von Tabellen „im Hintergrund“ ist ein weiterer großer Vorteil der mehrstufigen Adreßumsetzung. – Da die Segmentierung hier nicht mehr die ursprüngliche Organisationsform mit Basisadresse und Längenangabe aufweist, bezeichnet man die zweistufige Adreßumsetzung oft auch als *zweistufiges Paging* und die „*Segmenttabelle*“ dann als *Seitentabellenverzeichnis* (page table directory).

Bild 6-40 zeigt die Wirkungsweise einer solchen MMU, bei der die Segmentnummer als eine der beiden geschilderten Möglichkeiten durch Unterteilung der virtuellen 32-Bit-Adresse gebildet wird. Der verfügbare *virtuelle Adressraum* von 4 Gbyte wird hier unterteilt in 1024 Segmente (10-Bit-Segmentnummer) mit jeweils bis zu 1024 Seiten (10-Bit-Seitennummer) zu je 4 Kbyte (12-Bit-Bytenummer).

Im Bild ist nur eine der möglichen 1024 Seitentabellen dargestellt; der Trennstrich in den Deskriptoren soll andeuten, daß sie neben der Adressangabe auch Schutz- und Statusinformation enthalten. Um Tabellenzugriffe im Speicher möglichst zu vermeiden, wird auch hier wie bei der reinen Seitenverwaltung ein *Deskriptor-Cache (TLB)* eingesetzt. Die Segment- und die Seitennummer der

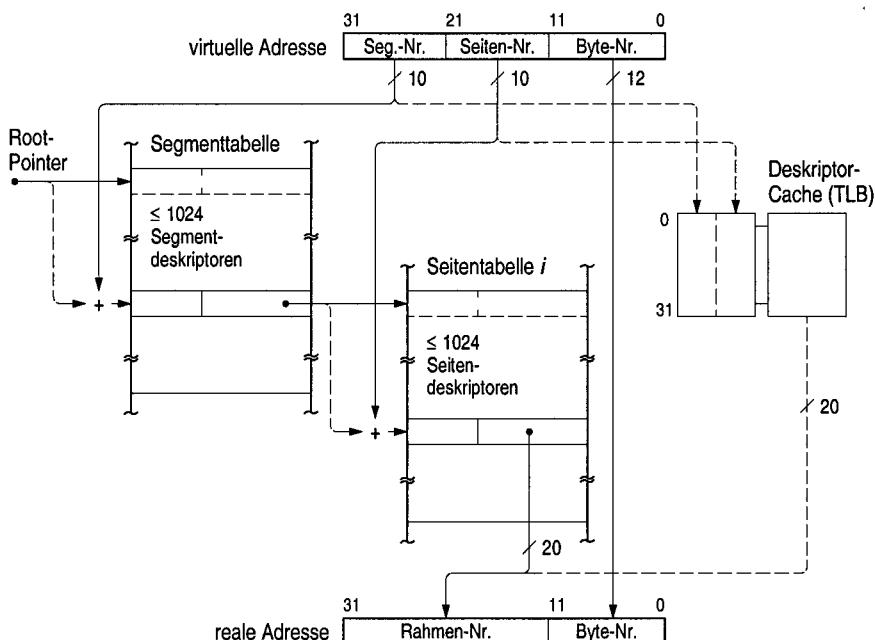


Bild 6-40. Zweistufige Adressumsetzung durch Kombination von Segment- und Seitenverwaltung, sog. zweistufiges Paging

virtuellen Adresse bilden die Tag-Information, die Rahmennummer und die Schutz- und Statusangaben den Cache-Eintrag. – Zur Bildung „großer Seiten“, wie sie bei Grafikanwendungen benötigt werden, kann die Bytenummer um die Seitennummer erweitert werden. Dies wird dann im Segmentdeskriptor vermerkt. In diesem Fall gibt es nur einen einzigen Seitendeskriptor mit z.B. der Statusinformation „Write-Through“ als Steuerinformation für den *Daten-Cache* (siehe 6.2.4 und 6.3.4).

Speicherverwaltungseinheiten heutiger Mikroprozessoren sind üblicherweise als On-chip-MMUs realisiert, d.h., sie sind Bestandteil des Prozessorchips. Zur MMU gehören neben dem Deskriptor-Cache die Steuerungen für die Adressumsetzung, für die Überprüfung der Speicherschutz- und Statusangaben sowie für das Aktualisieren dieses Cache.

Beispiel 6.5. ► MPC7451. Der MPC7451 (Motorola) ist ein 32-Bit-Prozessor mit je einer MMU für die Befehls- und für die Datenadressen. Beide MMUs arbeiten zweistufig und erzeugen aus 32-Bit-Prozessoradressen 36-Bit-Speicheradressen, erweitern also den Speicheradreßraum um den Faktor 16. Die Zweistufigkeit bezieht sich auf die Verwaltung von Seiten der Größe von 4 Kbyte, weicht aber von dem oben beschriebenen Vorgehen ab. Zusätzlich gibt es eine einstufige Adressumsetzung zur Verwaltung von Blöcken mit Größen zwischen 128 Kbyte und 256 Mbyte. Bild 6-41 zeigt die Wirkungsweise beider MMUs am Beispiel der Daten-MMU (DMMU). Beide MMUs benutzen die gezeigten 16 Segmentregister gemeinsam. Im Bild und bei den folgenden Erklärungen ist zu beachten, daß die Bitzählung bei den MPC-Darstellungen von links nach rechts erfolgt.

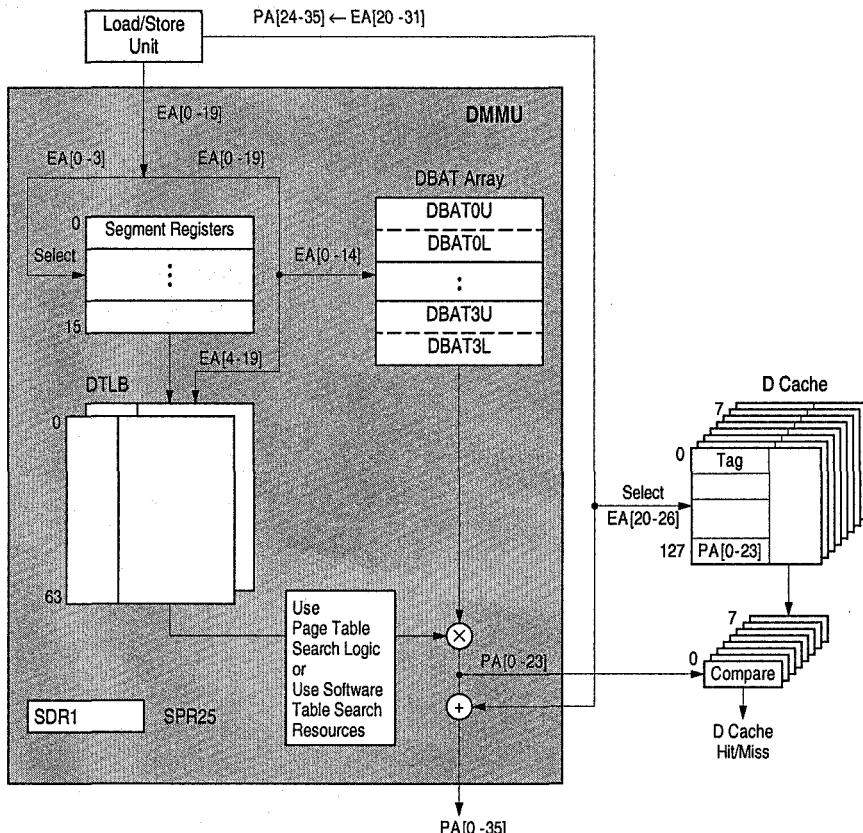


Bild 6-41. Daten-MMU des RISC-Mikroprozessors MPC7451. Umsetzung von 32-Bit-Adressen in 36-Bit-Adressen (nach [Motorola 2003]).

Ausgangspunkt für die Umsetzung einer Datenadresse ist die von der Lade-/Speichere-Einheit erzeugte effektive Adresse (EA), und zwar deren 20 höherwertige Bits EA0 bis EA19. Die restlichen Bits, EA20 bis EA31, bleiben als auf die Seite bezogener Byte-Offset (Bytenummer) unverändert.

Bei der zweistufigen Umsetzung werden die vier höchstwertigen Adreßbits EA0 bis EA3 als Segmentnummer interpretiert und damit eines der 16 Segmentregister angewählt. Diese Register enthalten Segmentdeskriptoren mit Adreßangaben, die zusammen mit den restlichen umzusetzenden Adreßbits eine sog. „virtuelle“ Adresse bilden. Mit dieser Adresse wird ein Deskriptor-Cache angewählt, der für insgesamt 128 Seitendeskriptoren ausgelegt und als 2-Wege-assoziativer Cache organisiert ist (DTLB: Data Translation Look-aside Buffer). Dieser liefert bei einem Treffer die 24 (!) höherwertigen Bits der realen 36-Bit-Adresse, PA0 bis PA23, die dann als Tag-Information zur Anwahl des Daten-Cache dienen (rechts im Bild, 128 Sätze, 8-Wege-assoziativ). Indiziert wird der Cache mit sieben der von der Adreßumsetzung nicht betroffenen Adreßbits EA20 bis EA26. Bei einem DTLB-Miss führt die MMU einen Suchvorgang in der im Speicher stehenden Seiten-tabelle durch und lädt den DTLB nach. Die Basisadresse dieser Tabelle wird dazu vom Betriebssystem im Register SDR1 bereitgestellt.

Parallel zur dieser Segment- und Seitenverwaltung erfolgt die Umsetzung der effektiven Adresse als Blockadresse mittels der vier Registerpaare des DBAT-Array (BAT: Block Address Translation). Von dieser Umsetzung sind je nach Blockgröße die 17 niederwertigen oder mehr Adreßbits ausgenommen; mit den restlichen Adreßbits wird das DBAT-Array assoziativ adressiert. Bei einem Treffer liefert das entsprechende DBAT-Registerpaar die fehlenden höherwertigen Bits der realen Adresse. In diesem Fall wird das Ergebnis der Segment- und Seitenadreßumsetzung verworfen. Das DBAT-Array wird (wie auch die Segmentregister) vom Betriebssystem geladen, ist also kein Cache mit automatischem Nachladen der Einträge. ▶

Dreistufige Adreßumsetzung. Viele On-chip-MMUs sehen Adreßumsetzungen in drei (oder mehr) Stufen vor und erlauben damit eine weitere Unterteilung der Tabellen und ggf. eine weitergehende Strukturierung von Segmenten in Subsegmente. Bei der Segmentverwaltung mit Unterteilung der virtuellen Adresse ergibt sich die dreistufige Adreßumsetzung, indem man in Erweiterung der MMU nach Bild 6-40 das virtuelle Adreßwort in vier (anstatt drei) Felder unterteilt und somit eine Segment-, eine Subsegment-, eine Seiten- und eine Bytenummer erhält. Das führt, wie Bild 6-42 zeigt, zu einer weiteren Tabellenebene für die Verwaltung von Subsegmenten, z.B. einzelner Prozeduren oder Datenbereiche, und zu kleinen Tabellen.

Eine MMU mit ebenfalls dreistufiger Adreßumsetzung, jedoch ausgehend von der Segmentverwaltung mit Erweiterung der virtuellen Adresse zeigt Bild 6-43. Sie besteht aus der MMU nach Bild 6-36, der eine MMU mit zweistufiger Adreßumsetzung gemäß Bild 6-40 nachgeschaltet ist, d.h., die bei der Segmentverwaltung entstehende Segmentadresse wird in ihrer Gesamtheit einem *zweistufigen Paging* unterworfen. Dies führt zu einer weiteren Unterteilung der Tabellen, lässt jedoch, da die Segmentadresse in ihrer vollen Länge weiterverarbeitet wird, keine Bildung von Subsegmenten im Virtuellen zu. – Im Bild aus Platzgründen nicht gezeigt sind die Pufferregister für die Segmentdeskriptoren und der MMU-Cache für das zweistufige Paging.

Realisiert ist letztere MMU bei den Intel-Prozessoren ab dem i386. Bei ihnen gibt es je ein Selektorregister für den Code (CS) und den Stack (SS) sowie vier Selektorregister für Datensegmente (DS bis GS). Die Segmentnummer (Selektor) hat 13 Bits. Die 20-Bit-Segmentlängenangabe bezieht sich entweder auf Bytes oder 4-Kbyte-Seiten, wählbar durch ein „Granularity“-Bit im Segmentdeskriptor. Beim Pentium gibt es für die Umsetzung der Segmentadresse zusätzlich die Möglichkeit, zwischen dem zweistufigen oder einstufigem Paging zu wählen (im Bild nicht gezeigt). Beim einstufigen Paging wird die Bytenummer um die Seitennummer vergrößert und die Verzeichnisnummer als Seitennummer ausgewertet. Dadurch erhält man Seiten mit der Größe von 4 Mbyte, z.B. für Grafikanwendungen. – Im Bild nicht gezeigt sind die sechs Pufferregister für die Segmentdeskriptoren und der MMU-Cache für das zweistufige Paging.

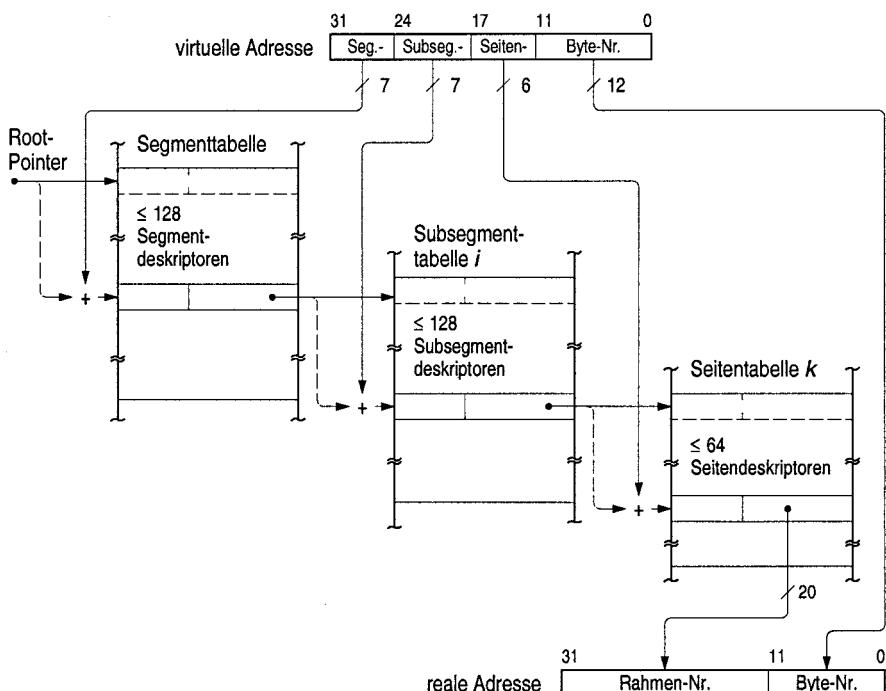


Bild 6-42. Dreistufige Adressumsetzung durch Erweiterung der zweistufigen Adressumsetzung nach Bild 6-40 um eine Subsegmentebene (in Anlehnung an die Motorola-Prozessoren MC68040 [Motorola 1989] und MC68060 [Motorola 1994])

Beispiel 6.6. ▶ x86-Prozessoren. Bild 6-44 auf der übernächsten Seite zeigt die Tabellenstruktur und die Adressumsetzung für die MMU der x86-Prozessoren (Intel). Ausgangspunkt für die Segmentbildung in der ersten Umsetzungsstufe ist die virtuelle Adresse (Offset) und eine in einem Selektorregister bereitgestellte Segmentnummer (Segment Selector). Sie führt auf ein zunächst unstrukturiertes Adresswort, weshalb die erzeugten Adressen wie auch der damit angesprochene Adreßraum als „linear“ bezeichnet werden kann (Linear Address Space). Unter Nutzung der zweistufigen Seitenverwaltung wird dann die lineare Adresse unterteilt und damit die Segmentver-

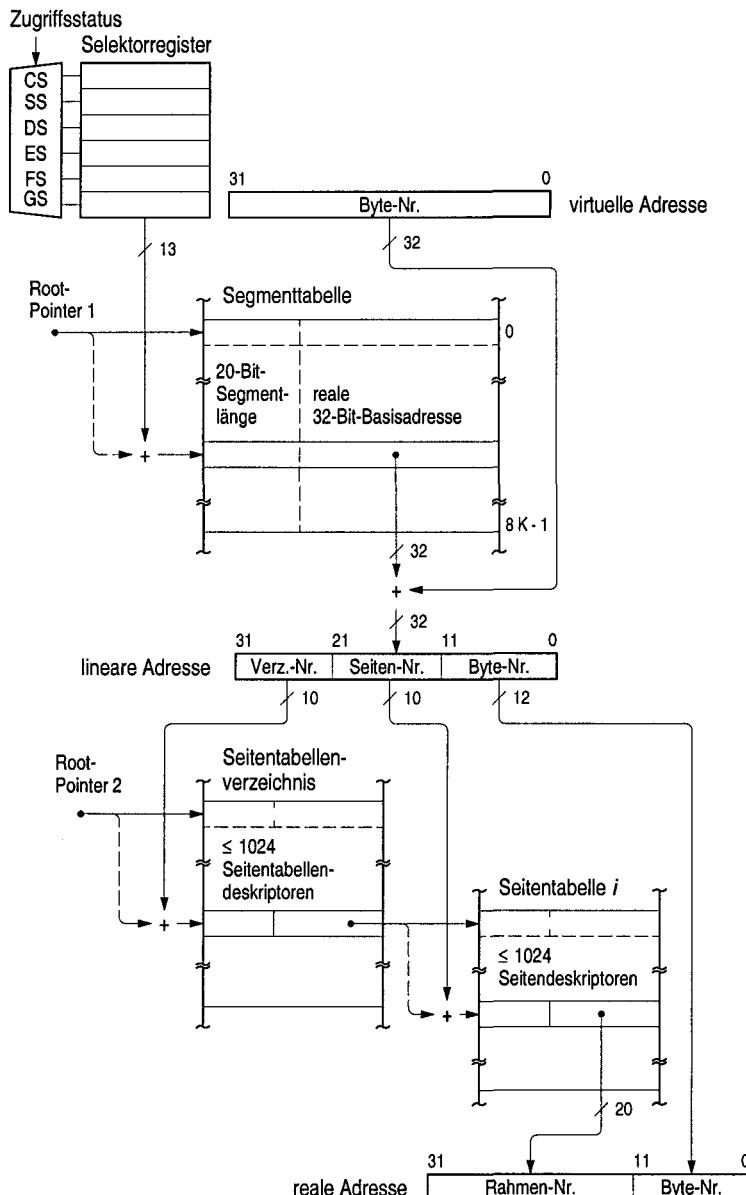


Bild 6-43. Dreistufige Adressumsetzung durch Kombination der Segmentverwaltung nach Bild 6-36 mit der zweistufigen Adressumsetzung nach Bild 6-40 (in Anlehnung an die x86-Prozessoren von Intel)

waltung mit einer Seitenverwaltung unterlegt, wie im linearen Adressraum gestrichelt angedeutet. Im eigentlichen Speicheradressraum (Physical Address Space) lassen sich die Seiten dann beliebig plazieren. ▲

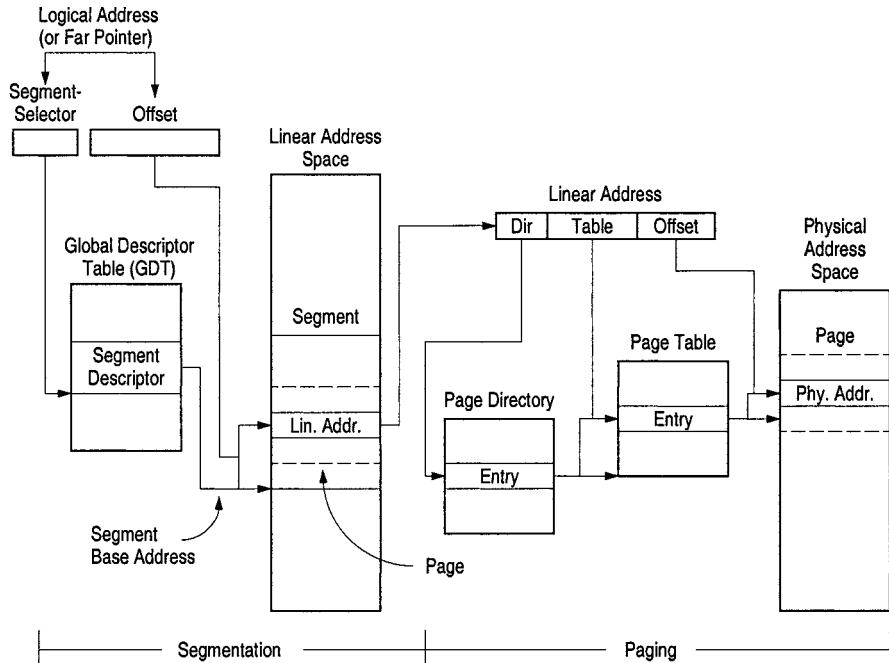


Bild 6-44. Zu Beispiel 6.6: Tabellenebenen und Adressräume der MMU der x86-Prozessoren (nach [Intel 1996b])

6.3.4 Speicherschutz und Speicherstatus

Segment- und Seitendeskriptoren. Die Segment- und Seitendeskriptoren weisen zum Teil unterschiedliche, zum Teil gleichartige Speicherschutz- und Statusangaben auf. Der *Segmentdeskriptor* gibt im wesentlichen an, ob das Segment gültig ist und – betrachtet man die zweistufigen Adressumsetzung – wie viele Seiten es umfasst; damit wird der Zugriff auf die von ihm adressierte Seitentabelle auf die entsprechende Anzahl an Einträgen begrenzt (Schutz vor Segmentüberschreitungen). Er enthält darüber hinaus Zugriffsattribute, die für das gesamte Segment gültig sind, jedoch nur für Zugriffe im User-Modus wirksam werden, z.B. „Zugriff nicht erlaubt“, „nur Lesezugriffe erlaubt“ oder „Lese- und Schreibzugriffe erlaubt“. Die Überprüfung der Speicherschutzbedingung zum Zeitpunkt der Adressumsetzung erfolgt anhand der vom Prozessor mit jedem Buszyklus erzeugten Statusinformation, wie Programm-, Daten-, Stack-, Lese-, Schreib-, User-, Supervisor-Zugriff.

Der *Seitendeskriptor* sieht die gleiche Art von Zugriffsattributen vor wie der Segmentdeskriptor, jedoch auf die jeweilige Seite bezogen. Bei der Adressumsetzung

werden sowohl die Angaben im Segment- als auch im Seitendeskriptor geprüft. Weichen sie voneinander ab, so kommt abhängig von den Attributen entweder die eine oder die andere Angabe zur Wirkung. Dieses Vorgehen wird z.B. bei sich überlappenden Segmenten genutzt (siehe unten). Der Seitendeskriptor enthält zusätzlich Statusangaben, z.B. „auf Seite wurde zugegriffen“ und „Seite wurde verändert“. Diese Angaben werden für die *Freispeicherverwaltung*, d.h. zur Implementierung von Alterungsmechanismen und für die Entscheidung zum Rückschreiben einer Seite auf den Hintergrundspeicher benötigt [Tanenbaum 2002]. Letztere Angabe hat die Bedeutung des *Dirty-Bits* bei Daten-Caches. Weitere Statusangabe lauten „cacheable/non-cacheable“, womit angegeben wird, ob der Inhalt der Seite im Cache gespeichert werden darf, „write-through/copy-back“, womit die für die Seite anzuwendende Aktualisierungsstrategie für den Daten-Cache festgelegt wird, und „global/non-global“, womit angegeben wird, ob Zugriffe auf diese Seite von der *Snoop-Logik* des Cache ausgewertet werden sollen oder nicht (siehe auch 6.2.4).

Code- und Data-Sharing. Segmente, die im virtuellen Adressraum voneinander unabhängige Adressbereiche belegen, können im Hauptspeicher so geladen sein, daß sie sich mit einer oder mehreren Seiten überlappen. Diese Möglichkeit wird z.B. für den Zugriff mehrerer Benutzer auf einen nur einmal vorhandenen Programmcode, z.B. auf einen Compiler, oder für den Zugriff unterschiedlicher Prozesse auf einen gemeinsamen Datenbereich genutzt. Man bezeichnet diese gemeinsamen Bereiche auch als Shared Code bzw. Shared Data (allg. als *Shared*

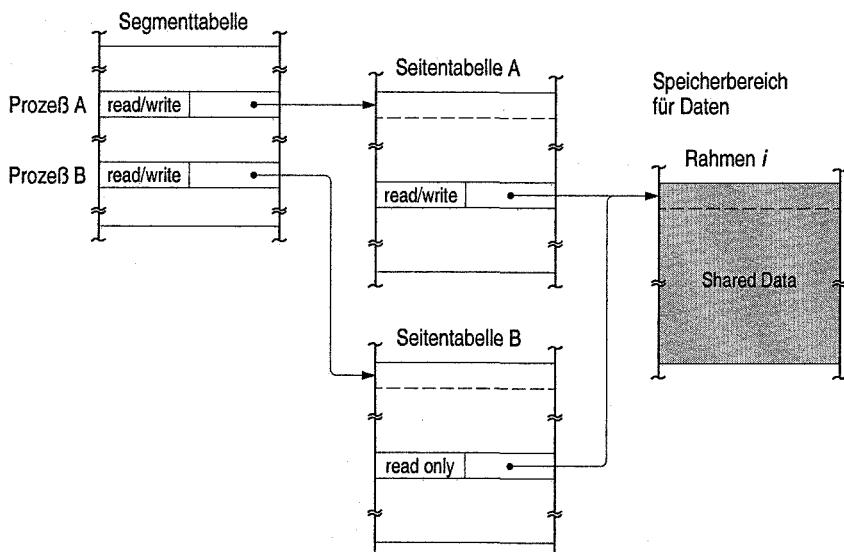


Bild 6-45. Data-Sharing zweier Prozesse A und B mit unterschiedlichen Zugriffsrechten auf eine gemeinsame Seite. Zweistufige Adressumsetzung entsprechend Bild 6-40

Memory). Mit Hilfe der Zugriffsattribute in den Segment- und Seitendeskriptoren können den verschiedenen Benutzern bzw. Prozessen unterschiedliche Zugriffsrechte auf diese Bereiche eingeräumt werden. Bild 6-45 zeigt dazu ein Beispiel für den gemeinsamen Zugriff zweier Prozesse A und B auf eine gemeinsame Seite. Prozeß A hat im Segment- wie im Seitendeskriptor das Attribut „Lese- und Schreibzugriffe erlaubt“. Prozeß B hat dieses Attribut zwar im Segmentdeskriptor, er ist jedoch in seinem Seitendeskriptor auf „nur Lesezugriffe erlaubt“ eingeschränkt. Hier kommt das einschränkende Attribut zur Wirkung.

Vereinfachter Speicherschutz. Speicherschutz ist bei Systemen mit virtueller Speicherverwaltung unabdingbar, er kann jedoch auch bei sehr einfachen Systemen, die ohne MMU arbeiten, realisiert werden. Hierbei werden die vom Prozessor kommenden Statussignale unmittelbar bei der Ansteuerung von Speicher- und Ein-/Ausgabeeinheiten ausgewertet. So können auch hier Zugriffe auf bestimmte Adressbereiche von der Betriebsart Supervisor- bzw. User-Modus (S-Signal), vom Lesen bzw. Schreiben (R/\bar{W} -Signal) und von Zugriffen auf Befehle bzw. Daten (P/\bar{D} -Signal) abhängig gemacht werden.

Bild 6-46 zeigt dazu ein Beispiel für die Anwahl verschiedener Speicherbereiche, die über die Adressdecodierung (im Bild nicht dargestellt) hinaus von den Statussignalen abhängt. Zugriffe im Supervisor-Modus ($S = 1$) sind ohne Einschränkung auf den Gesamtbereich möglich. Zugriffe im User-Modus ($S = 0$) sind auf den Bereich 1 nicht möglich, auf den Bereich 2 nur für Befehlszugriffe ($P/\bar{D} = 1$), auf den Bereich 3 für das Lesen und Schreiben von Daten ($P/\bar{D} = 0$) und auf den Bereich 4 nur für das Lesen von Daten ($R/\bar{W} = 1$ und $P/\bar{D} = 0$).

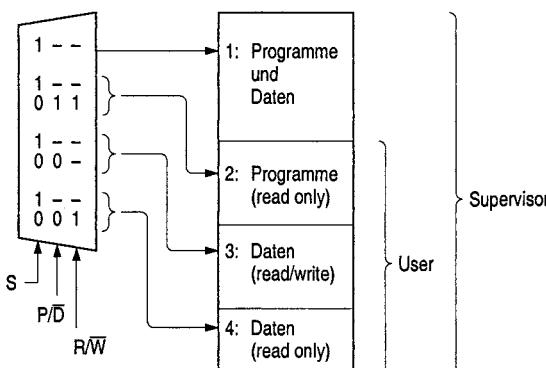


Bild 6-46. Wirkung der Statussignale des Prozessors bei Anwahl von Speicherbereichen mit unterschiedlichen Zugriffskriterien

7 Ein-/Ausgabeorganisation und Rechnerkommunikation

Mit dem Begriff Ein-/Ausgabeorganisation faßt man die Vorgänge zur Übertragung von Daten zwischen dem Hauptspeicher und der Peripherie eines Mikroprozessorsystems zusammen. Zur Peripherie zählen die Ein-/Ausgabegeräte, z.B. Bildschirmterminal, Drucker und Scanner, und die Hintergrundspeicher, z.B. Festplattenspeicher, Floppy-Disk-Speicher und Streamer. Hinzu kommen anwendungsbezogene Ein-/Ausgabeeinheiten, z.B. zur Übertragung von Steuer- und Statusinformation bei Systemen mit Steuerungs- und Regelungsaufgaben. Dehnt man den Begriff Ein-/Ausgabeorganisation weiter aus, so sind auch die Datenfernübertragung und die Datenübertragung in Rechnernetzen mit in die Betrachtungen einzubeziehen.

Aufgrund ihrer spezifischen Signale und Übertragungsabläufe können periphere Einheiten nicht, wie z.B. Speichereinheiten, unmittelbar an den Systembus angeschlossen werden. Für die jeweilige Busanpassung sind hier entsprechende Anpaßbausteine, sog. *Interface-Adapter* (kurz: *Adapter*), als Schnittstelleneinheiten erforderlich (Bild 7-1, S. 457). Man bezeichnet diese auch als *Interface-Bausteine* (kurz: *Interfaces*) oder als Ein-/Ausgabetore (i/o ports).

Bei einfachen Rechnersystemen übernimmt der Prozessor die Steuerung der Ein-/Ausgabe (*prozessorgesteuerte Ein-/Ausgabe*). Leistungsfähigere Systeme besitzen neben dem Prozessor zusätzliche Steuereinheiten mit Busmasterfunktion, die den Prozessor von der Datenübertragung entlasten (*DMA-Controller* zur schnellen Datenübertragung zwischen Speicher und Peripherie). In Erweiterung ihrer Funktion können diese Controller als programmgesteuerte Ein-/Ausgabeprozessoren ausgelegt sein. Oder es werden in einer weiteren Ausbaustufe eigenständige Rechnersysteme als *Ein-/Ausgaberechner* eingesetzt, so daß der Prozessor letztlich nur noch die Aufträge für Ein-/Ausgabevorgänge zu vergeben hat (*controllerbzw. computergesteuerte Ein-/Ausgabe*).

Eine davon abweichende Form der Ein-/Ausgabe stellt die Datenübertragung zwischen Ein-/Ausgabegeräten und einem Rechner mittels der *Datenfernübertragung* dar. Hierbei werden die Ein-/Ausgabegeräte als sog. abgesetzte Datenstationen (*remote terminals*) unter Benutzung von Leitungen und Vermittlungseinrichtungen der öffentlichen Telekommunikationssysteme an den Rechner angeschlossen. Dazu gibt es nationale und internationale Vereinbarungen als Standards für die Übertragungstechniken. Diese haben wiederum Rückwirkungen auf

die Techniken der „lokalen“ Ein-/Ausgabe, z.B. als Schnittstellenvereinbarungen für Verbindungen zwischen den Interfaces und Ein-/Ausgabegeräten.

Eine völlig universelle Form der Ein-/Ausgabe stellt das Zusammenschließen von Rechnern zu *Rechnernetzen* dar. Sie erlauben eine flexible Kommunikation zwischen den Netzteilnehmern, wie auch die Nutzung von Service-Funktionen eines solchen Netzes. Netze werden entweder als *lokale Netze* innerhalb von Gebäuden und Grundstücksgrenzen gebildet, oder sie werden unter Einbeziehung der Datenfernübertragung als *Weitverkehrsnetze* ausgelegt. Der weltweite Verbund von Netzen, wie er durch das Internet gegeben ist, ermöglicht den weltweiten Informationsaustausch.

Abschnitt 7.1 befaßt sich mit der prozessorgesteuerten Ein-/Ausgabe. Er beschreibt die wichtigsten Merkmale eines Interfaces und die für die Datenübertragung zwischen Prozessor und Peripherie gebräuchlichen Synchronisationstechniken. In Abschnitt 7.2 werden dann übergreifend Merkmale der Datenübertragung erläutert, wie sie in den folgenden Abschnitten als Begriffe vorausgesetzt werden. Abschnitt 7.3 beschreibt einige gängige parallele Schnittstellen und stellt dabei auch einen universellen Parallel-Interface-Baustein vor. Abschnitt 7.4 geht dann auf die wichtigsten seriellen Schnittstellen ein. Deren Einsatz wird in den Abschnitten 7.5 und 7.6 durch die Techniken der asynchron-seriellen und der synchron-seriellen Datenübertragung veranschaulicht. Ergänzt werden diese Betrachtungen durch je ein Beispiel eines Interface-Bausteins, mit Bausteinbeschreibungen, die die wichtigsten Funktionen der im Handel befindlichen Interface-Bausteine widerspiegeln. Abschnitt 7.7 gibt schließlich einen Einblick in die Rechnerkommunikation, d.h. in die Techniken der Rechnervernetzung und der Datenfernübertragung. Die oben erwähnte controller- und computergesteuerte Ein-/Ausgabe wird in Kapitel 8 gesondert behandelt.

7.1 Prozessorgesteuerte Ein-/Ausgabe

Abhängig von der Peripherie umfaßt ein Ein-/Ausgabevorgang mehrere Aufgaben, wie

- Starten des Ein-/Ausgabevorgangs, z.B. durch Starten des Geräts,
- Ausführen spezifischer Gerätetfunktionen, z.B. Einstellen des Zugriffsarms bei einem Plattenlaufwerk,
- Übertragen einzelner Daten mit Synchronisation der Übertragungspartner,
- Datenzählung und Adreßfortschaltung bei blockweiser Übertragung,
- Lesen und Auswerten von Statusinformation, z.B. für eine Fehlererkennung und Fehlerbehandlung,
- Stoppen des Ein-/Ausgabevorgangs, z.B. durch Stoppen des Geräts.

Bei der prozessorgesteuerten Ein-/Ausgabe obliegt die Ausführung dieser Steuerungsaufgaben dem Prozessor, genauer, dem vom Prozessor auszuführenden Ein-/Ausgabeprogramm. Unterstützt wird er dabei durch die Interface-Bausteine, die, wie oben erwähnt, die Anpassung der peripheren Datenwege und Übertragungsabläufe an den Systembus vornehmen und darüber hinaus zur *Synchronisation* der Übertragungspartner dienen. Die Notwendigkeit der Synchronisation ergibt sich durch die gerätetechnisch bedingten unterschiedlichen Verarbeitungsgeschwindigkeiten von Prozessor und Peripherie. Hinzu kommt, daß sich z.B. beim Prozessor aufgrund anderer Aktivitäten, die er verzahnt mit dem Ein-/Ausgabevorgang ausführt, die Übertragungsbereitschaft verzögern kann.

7.1.1 Ein einfacher Interface-Baustein

Ein *Interface-Baustein* besitzt mindestens ein Datenregister DR (data register) zur Pufferung der Daten zwischen dem Systembus und dem peripheren Übertragungsweg (Bild 7-1). Seine Schnittstelle zum Systembus ist bei z.B. *speicherbezogener Adressierung* (siehe 5.2.1) so ausgelegt, daß das Datenregister wie eine Speicherzelle beschrieben und gelesen werden kann. Abgesehen von der Synchronisation stellt sich damit für den Mikroprozessor der einzelne Datentransport mit der Peripherie wie ein Speicherzugriff dar. (Bei *isolierter Adressierung* unter-

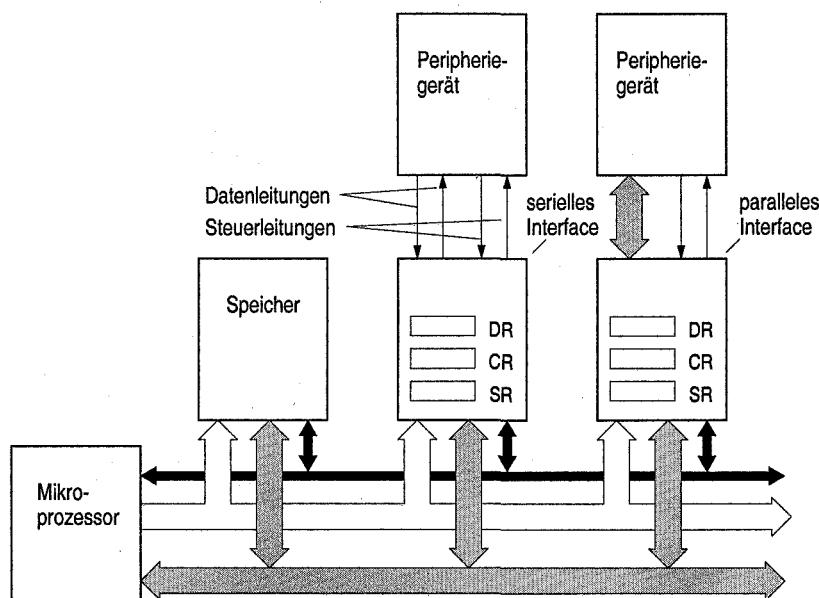


Bild 7-1. Systemstruktur mit Interface-Bausteinen für die Ein-/Ausgabe. Die Verbindungen zwischen den Peripheriegeräten und den Interface-Bausteinen bilden die peripheren Übertragungswege

scheidet sich der Zugriff nur dadurch, daß anstelle der Speicherzugriffsbefehle *Ein-/Ausgabebefehle* verwendet werden.)

Die Unterschiede der einzelnen Interface-Bausteine liegen in der Verschiedenartigkeit ihrer Schnittstellen zur Peripherie, z.B. in der parallelen oder seriellen Datenübertragung, in der Art und Anzahl der Steuerleitungen und damit in dem für die Datenübertragung zwischen Interface und Peripherie erforderlichen Steuerwerk. So erfordert z.B. die serielle Ein-/Ausgabe eine Serien-Parallel-Umsetzung bei der Eingabe und eine Parallel-Serien-Umsetzung bei der Ausgabe. Ferner muß das Steuerwerk auf die Regeln für die zeichen- oder blockweise Datenübertragung zugeschnitten sein. Man bezeichnet diese Regeln zusammengefaßt auch als *Übertragungsprotokoll*.

Durch das Laden von Steuerinformation in eines oder mehrere Steuerregister CR (control register) können unterschiedliche Betriebsarten eines Interfaces, z.B. ein bestimmtes Protokoll, durch das Programm ausgewählt (programmiert) werden. Der augenblickliche Betriebszustand des Interfaces wiederum wird in einem oder mehreren Statusregistern SR (status register) angezeigt, so z.B. Synchronisationsinformation für den Prozessor. Die Steuer- und Statusregister sind dazu ebenfalls wie Speicherzellen (bzw. durch Ein-/Ausgabebefehle) beschreib- bzw. lesbar. Bei einfachen Bausteinen mit wenigen Funktionen sind die Steuer- und Statusbits vielfach auch in einem einzigen Register zusammengefaßt.

7.1.2 Synchronisationstechniken

Die *Synchronisation* einer einzelnen Datenübertragung über ein Interface erfolgt durch den Austausch von Steuerinformation: zum einen zwischen Mikroprozessor und Interface, d.h. auf dem Systembus, zum andern zwischen Interface und Peripherie, d.h. auf dem peripheren Übertragungsweg. Hierbei signalisiert der Prozessor dem Interface seine Übertragungsbereitschaft, indem er ein entsprechendes Bit im Steuerregister des Bausteins setzt, oder indem er unmittelbar dessen Datenregister liest (Eingabe) bzw. in dessen Datenregister schreibt (Ausgabe). Das Interface seinerseits meldet dem Prozessor seine Bereitschaft entweder durch Setzen eines Statusbits in seinem Statusregister oder – durch das Setzen des Statusbits bedingt – durch Aktivieren einer Interrupt-Request-Leitung.

Der Austausch von Synchronisationsinformation auf dem peripheren Übertragungsweg erfolgt entweder durch Steuersignale, wie dies z.B. bei der parallelen Ein-/Ausgabe üblich ist, oder durch Steuerinformation, die dem zu übertragenden Datum selbst als Rahmen beigefügt ist, wie z.B. bei der asynchron-seriellen Ein-/Ausgabe. Die Übertragung aufeinanderfolgender Daten hängt dabei entweder von der jeweiligen Bereitschaft der beiden Übertragungspartner ab oder ist durch einen Taktgenerator mit fester Frequenz vorgegeben. – Im folgenden betrachten wir drei grundsätzliche Verfahren zur Synchronisierung einzelner Datenüber-

tragungen, wobei wir zu ihrer Beschreibung Steuersignale zugrundelegen, wie sie bei der parallelen Ein-/Ausgabe üblich sind (zur Synchronisierung mittels Steuerinformation im Datenrahmen siehe 7.5).

Synchronisation durch Busy-Waiting. Bei dieser Synchronisationsart wartet der Mikroprozessor so lange mit dem Aus- bzw. Eingeben eines Datums, bis das Interface seine Bereitschaft durch ein Statusbit (Ready-Bit) in seinem Statusregister anzeigt (Bild 7-2). Dieses Bit wird durch ein Steuersignal (READY) der Peripherie gesetzt, wenn diese den bisherigen Inhalt des Interface-Datenregisters übernommenen (Ausgabe) bzw. das Datenregister erneut geladen hat (Eingabe). Das Warten des Mikroprozessors erfolgt entweder in einer Warteschleife, in der er das Ready-Bit des Statusregisters laufend überprüft, oder der Prozessor führt die Abfrage in bestimmten Zeitabständen durch und nutzt die Zwischenzeit für laufende Verarbeitungsvorgänge. Ist das Ready-Bit gesetzt, so verzweigt er zu der eigentlichen Ein-/Ausgabebefehlsfolge (siehe auch Beispiel 7.1, S. 462). Dort wird zum einen die Datenübertragung durchgeführt, zum andern muß das Ready-Bit zurückgesetzt werden, um das Interface für die nächste Übertragung vorzubereiten. Dieses Rücksetzen geschieht abhängig vom Interface-Baustein entweder implizit, nämlich mit dem Zugriff des Mikroprozessors auf dessen Datenregister, oder explizit, z.B. durch einen zusätzlichen Lesezugriff oder einen Schreibzugriff (Maskieren des Bits) auf dessen Statusregister. – Das Ready-Bit wird häufig auch als *Ready-Flag* (Flagge, Signal) bezeichnet.

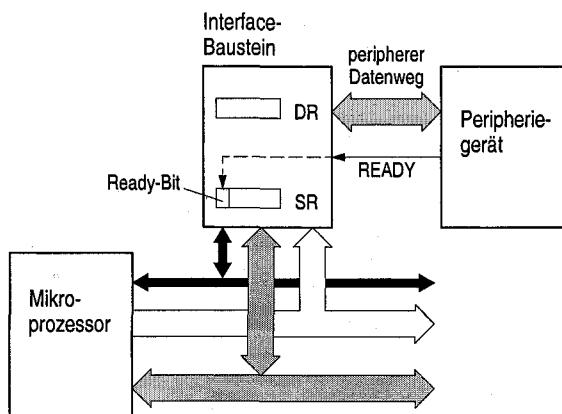


Bild 7-2. Systemstruktur bei Synchronisation durch Busy-Waiting: die Abfrage des Ready-Bits erfolgt über den Datenbus

Der Vorteil von Busy-Waiting ist die Einfachheit. Nachteilig ist jedoch, daß der Mikroprozessor während eines Ein-/Ausgabevorgangs einen Teil seiner Verarbeitungszeit für die Abfrage des Ready-Bits benötigt, unabhängig davon, ob eine Übertragungsanforderung vorliegt oder nicht. Im Falle einer Warteschleife ist er sogar überwiegend mit Warten beschäftigt und damit schlecht genutzt. Darüber

hinaus setzt Busy-Waiting voraus, daß die Verarbeitungsgeschwindigkeit des Mikroprozessors höher ist als die der Peripherie. Zeitkritische Situationen können z.B. entstehen, wenn der Mikroprozessor mehrere Ein-/Ausgabevorgänge gleichzeitig zu bearbeiten hat (siehe dazu 7.1.3).

Anmerkung. Die Bezeichnung Synchronisation durch *Busy-Waiting* ist im Grunde nicht ganz zutreffend, da der Prozessor die Wartezeiten ggf. durch andere Programmausführungen überbrücken kann. Treffender wäre die Bezeichnung Synchronisation durch *Abfrage*. Dennoch behalten wir den Begriff Busy-Waiting bei, da er alleinstehend prägnanter als der Begriff Abfrage ist. Alternativ käme auch der Begriff *Polling* in Frage. Dieser wird jedoch traditionell für das zyklische Abfragen mehrerer Ready-Bits in mehreren Interfaces benutzt (7.1.3), weshalb wir ihn für die Einzelabfrage nicht verwenden.

Synchronisation durch Programmunterbrechung. Bei der Synchronisation durch Programmunterbrechung wird der Zustand des Interface-Statusbits als Interruptanforderung an den Mikroprozessor weitergegeben, d.h., das Signal READY wirkt über das Ready-Bit als *Interruptsignal* (Bild 7-3). Der Prozessor fragt also nicht wie beim Busy-Waiting den Zustand des Statusbits laufend ab, indem er dazu das Statusregister liest und das Bit testet, sondern bekommt ihn unmittelbar als Interruptsignal übermittelt. Wird die Interruptanforderung vom Mikroprozessor akzeptiert, so reagiert er mit einer Programmunterbrechung und verzweigt zu einem Interruptprogramm, das die Ein-/Ausgabebefehlsfolge enthält und die Ein-/Ausgabe durchführt (siehe auch Beispiel 7.2, S. 463). – Das Ready-Bit wird in diesem Fall auch als *Interrupt-Bit* oder *Interrupt-Flag* bezeichnet.

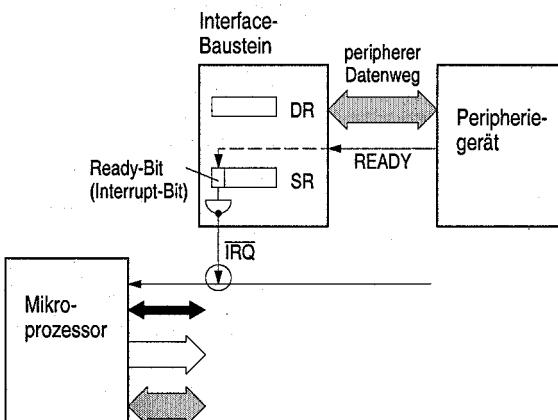


Bild 7-3. Systemstruktur bei Synchronisation durch Programmunterbrechung: Interrupt-Anforderung durch das Ready-Bit mittels einer Interrupt-Request-Leitung \overline{IRQ}

Gegenüber der Synchronisation durch Busy-Waiting gewinnt der Prozessor eine gewisse Unabhängigkeit von den Ein-/Ausgabeaktivitäten der Peripherie und kann die Zeit zwischen den Programmunterbrechungen für andere Aufgaben besser nutzen. Nachteilig ist jedoch die durch die Interruptverarbeitung (2.1.5):

Ausnahmeverarbeitung) hervorgerufene Verzögerung der Übertragung. – Voraussetzung für eine fehlerfreie Übertragung ist auch hier wieder, daß die Verarbeitungsgeschwindigkeit des Mikroprozessors höher als die der Peripherie ist.

Synchronisation durch Handshaking. Um die Datenübertragung auch für denjenigen Fall zu ermöglichen, daß die Peripherie schneller als der Prozessor arbeitet und demnach auf den Prozessor warten muß, wird die Übertragungssteuerung erweitert. Dazu zeigt der Mikroprozessor, bzw. stellvertretend für ihn das Interface, seine Bereitschaft zur Datenübertragung durch Setzen eines Quittungssignals (Eingabe) bzw. Bereitstellungssignals (Ausgabe) an. Bei der Eingabe überträgt die Peripherie dementsprechend ein Datum erst dann, wenn der Mikroprozessor die Übernahme des letzten Datums quittiert hat; der Mikroprozessor seinerseits übernimmt ein Datum aus dem Interface-Datenregister erst dann, wenn die Übertragung des Datums wie oben beschrieben von der Peripherie bzw. dem Interface signalisiert worden ist. Bei der Ausgabe übernimmt die Peripherie ein Datum aus dem Interface-Datenregister erst dann, wenn ihr die Bereitstellung vom Prozessor bzw. Interface signalisiert worden ist; der Prozessor seinerseits wartet mit der nächsten Datenbereitstellung bis ihm die Peripherie bzw. das Interface die Übernahme des letzten Datums bestätigt hat. Man nennt dieses Verfahren, bei dem sich beide Übertragungspartner sozusagen die Hände reichen, *Handshaking (Quittungsbetrieb)*.

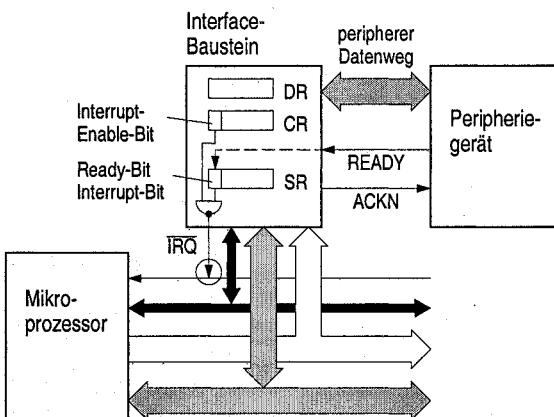


Bild 7-4. Systemstruktur bei Synchronisation durch Handshaking. Die Mnemone ACKN (Quittung) und READY (Bereitstellung) bezeichnen die Signalwirkungen für die Eingabe; bei der Ausgabe vertauschen sich die Signalwirkungen

Bild 7-4 zeigt als Erweiterung der Bilder 7-2 und 7-3 das Blockschaltbild für das Handshaking auf der Steuersignalebene. Das Lesen des Interface-Datenregisters durch den Mikroprozessor (Eingabe) bzw. das Schreiben in das Interface-Datenregister (Ausgabe) wird der Peripherie durch das Signal ACKN (acknowledge) angezeigt. Dieses Signal wird abhängig vom Interface-Baustein entweder implizit

mit dem Lese- bzw. Schreibzugriff auf das Datenregister aktiviert und nach einer vom Interface fest vorgegebenen Zeit wieder inaktiviert. Oder es muß programmiert aktiviert werden, indem der Prozessor nach dem Datenregisterzugriff ein entsprechendes ACKN-Bit im Steuerregister des Interfaces setzt (im Bild nicht gezeigt). Inaktiviert wird es dann entweder durch Rücksetzen dieses Bits oder automatisch nach einer fest vorgegebenen Zeit.

Die Peripherie ihrerseits meldet die Datenbereitstellung bzw. die Datenübernahme aus dem Datenregister wie bisher durch das Signal READY. Dieses Signal wird vom Prozessor entweder als Statusinformation (Handshaking mit Busy-Waiting) oder als Interruptsignal (Handshaking mit Programmunterbrechung) ausgewertet. Ein *Interrupt-Enable-Bit* im Steuerregister des Interfaces, mit dem man das Durchschalten des Ready-Bits auf den Interruptausgang des Interfaces steuern kann, ermöglicht die Wahl zwischen den beiden Betriebsarten. – Eine detaillierte Darstellung der Zeitverläufe der Handshake-Signale zeigt Bild 7-14 (S. 478).

Beispiel 7.1. ► Synchronisation einer Dateneingabe durch Handshaking mit Busy-Waiting. Über ein 8-Bit-Interface-Datenregister DR sollen 128 Datenbytes in einen Speicherbereich mit der Adresse BUFFER eingelesen werden. Zur Synchronisation der Übertragung gibt das Peripheriegerät mit der Übergabe eines Datums an DR ein READY-Signal aus, was in der Bitposition 7 (Ready-Bit, SR7) des 8-Bit-Interface-Statusregisters SR durch eine Eins angezeigt wird. Der Zustand dieses Bits soll durch Busy-Waiting ausgewertet werden. Dazu sei das Interrupt-Enable-Bit im Steuerregister des Interfaces bei dessen Initialisierung zurückgesetzt worden, so daß das dem Ready-Bit zugeordnete Interruptsignal blockiert ist.

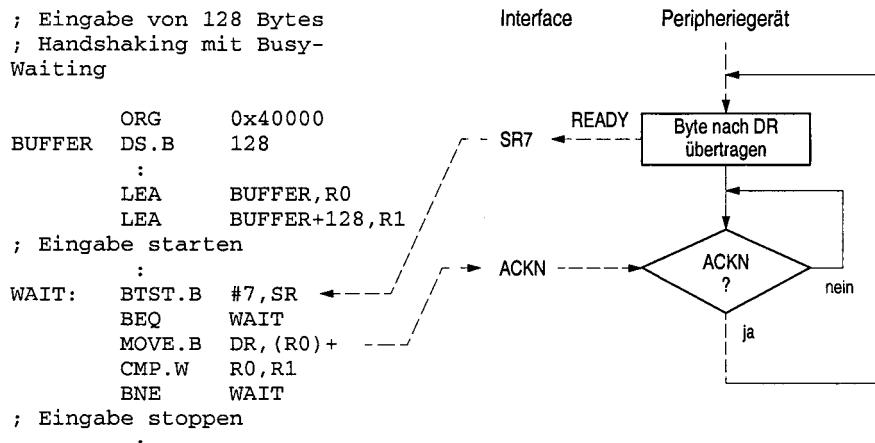


Bild 7-5. Ablauf zu Beispiel 7.1: *Synchronisation einer Dateneingabe durch Handshaking mit Busy-Waiting*

Bild 7-5 zeigt den Ablauf. Zu Beginn des Eingabeprogramms werden die Register R0 und R1 mit der Pufferanfangsadresse BUFFER als Pufferpointer und mit der um Eins erhöhten Pufferendadresse BUFFER+128 zur Ermittlung des Schleifenendes initialisiert. Die Abfrage des Ready-Bits erfolgt in einer Warteschleife durch den Bittestbefehl BTST.B dessen Direktoperand 7 die

Bitposition im Statusregister vorgibt. Ist das Ready-Bit gesetzt, so wird der Inhalt von DR in den mit R0 postinkrement-adressierten Pufferbereich gelesen. Mit diesem Lesevorgang wird vom Interface das Ready-Signal zurückgesetzt und das Quittungssignal ACKN an das Peripheriegerät übertragen. Das ACKN-Signal zeigt dem Peripheriegerät die Datenübernahme an. Der Vorgang wird so oft wiederholt, bis der Pufferpointer in R0 mit der um Eins erhöhten Pufferendadresse in R1 übereinstimmt und damit anzeigen, daß der Puffer gefüllt ist. – Bei der Programmierung der Eingabe wurde weder auf die Initialisierung des Interfaces noch auf das Starten und Stoppen des Peripheriegeräts eingegangen; hierzu siehe Beispiel 7.3 (S. 479). ▶

Beispiel 7.2. ► *Synchronisation einer Dateneingabe durch Handshaking mit Programmunterbrechung.* Die in Beispiel 7.1 beschriebene Dateneingabe von 128 Bytes soll so modifiziert werden, daß die Übertragung eines Datums an das Datenregister DR des Interfaces eine Programmunterbrechung auslöst und die Datenübernahme durch den Prozessor in einem Interruptprogramm erfolgt. Dazu sei das Interrupt-Enable-Bit im Steuerregister des Interfaces bei dessen Initialisierung gesetzt worden, so daß das dem Ready-Bit zugeordnete Interruptsignal freigegeben ist.

```
; Eingabe von 128 Bytes
; Handshaking mit Programmunterbrechung
; Hauptprogramm
```

```
ORG    0x40000
REF    BUFPTR, BUFEND
BUFFER DS.B 128
:
LEA    BUFFER, BUFPTR
LEA    BUFFER+128, BUFEND
```

```
; Eingabe starten
```

```
:
```

```
; Interruptprogramm
```

```
DEF    BUFPTR, BUFEND
BUFPTW DS.W 1
BUFEND DS.W 1
```

```
INPUT: PUSH.W R0
      MOVE.W BUFPTW, R0
      MOVE.B DR, (R0) +
      CMP.W R0, BUFEND
      BNE    RETURN
```

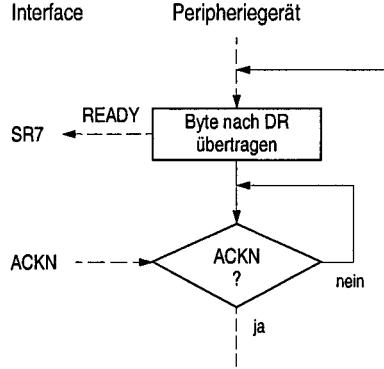
```
; Eingabe stoppen
```

```
:
```

```
RETURN: MOVE.W R0, BUFPTW
       POP.W R0
       RTE
```

Bild 7-6. Ablauf zu Beispiel 7.2: *Synchronisation einer Dateneingabe durch Handshaking mit Programmunterbrechung*

Bild 7-6 zeigt den Ablauf. Eine durch das Ready-Signal ausgelöste Programmunterbrechung führt auf das Interruptprogramm INPUT, das mit jedem Aufruf ein Datenbyte vom Datenregister DR in den Puffer übernimmt. Zur Adressierung des Puffers und zur Bytezählung werden vom Hauptprogramm, bevor es die Eingabe startet, die globalen Variablen BUFPTW und BUFEND mit der Pufferanfangsadresse BUFFER und der um Eins erhöhten Pufferendadresse BUFFER+128 initia-



lisiert. Das Interruptprogramm lädt den Pufferpointer BUFPTR nach R0 und liest den Inhalt von DR in den mit R0 postinkrement-adressierten Pufferbereich. Anschließend weist es den um Eins erhöhten Pufferpointer wieder der Variablen BUFPTR zu. (Um den ursprünglichen Inhalt von R0 nicht zu überschreiben, wird er zu Beginn der Interruptprogrammausführung auf den Supervisor-Stack gerettet und vor deren Abschluß von dort wieder geladen.) Mit der Übernahme des Datums wird das Ready-Bit im Statusregister vom Interface zurückgesetzt und damit die Interruptanforderung zurückgenommen. Gleichzeitig gibt das Interface das Quittungssignal ACKN an das Peripheriegerät aus und zeigt ihm damit die erfolgte Datenübernahme an. ▶

7.1.3 Gleichzeitige Bearbeitung mehrerer Ein-/Ausgabevorgänge

Die Ein- und Ausgabe beschränkt sich i. allg. nicht auf die isolierte Durchführung eines einzelnen Ein-/Ausgabevorgangs, sondern betrifft oftmals das gleichzeitige Durchführen mehrerer solcher Vorgänge durch den Mikroprozessor, z.B. dann, wenn mehrere Terminals gleichzeitig Übertragungsanforderungen an den Mikroprozessor stellen können (Mehrbenutzersystem). Je nachdem, ob die einzelnen Anforderungen durch *Busy-Waiting* oder durch *Programmunterbrechung* behandelt werden, benötigt man zu deren Bearbeitung unterschiedliche Programmierungstechniken. Treten Anforderungen unterschiedlicher Prioritäten auf, so kann das ein Unterbrechen eines gerade laufenden Ein-/Ausgabeprogramms zur Folge haben. Dieses Programm wird dann erst nach Bearbeitung der höherpriorisierten Anforderung, d.h. zu einem späteren Zeitpunkt fortgesetzt. Dabei dürfen jedoch keine Ein-/Ausgabezeitbedingungen verletzt werden, um einen Verlust von Daten zu vermeiden.

Polling. Bei peripheren Einheiten, deren Übertragungsanforderungen durch Busy-Waiting verarbeitet werden, wird das Abfragen der einzelnen Ready-Statusbits nacheinander in einer Abfragesequenz durchgeführt. Diese wird dazu in regelmäßigen Zeitabständen durchlaufen, z.B. durch einen Timer-Interrupt gesteuert. Zeigt eines der Interfaces eine Übertragungsanforderung an, so wird auf das zugehörige Ein-/Ausgabeprogramm verzweigt (Bild 7-7a). Man bezeichnet diesen Abfragevorgang auch als Polling und die Abfragesequenz als Polling-Routine. Bei peripheren Einheiten, deren Übertragungsanforderungen nicht durch Busy-Waiting, sondern durch Interrupts der einzelnen Einheiten erfolgen, ist ggf. ebenfalls ein Polling erforderlich, nämlich dann, wenn die Anforderungen als Interrupts derselben Ebene ein gemeinsames Interruptprogramm haben. Das Polling dient dann dazu, die jeweiligen Anforderungen zu identifizieren (siehe auch 5.5.2). Im Interruptprogramm werden dazu wiederum die einzelnen Ready-Statusbits abgefragt. Ist das Ready-Bit, das die Programmunterbrechung ausgelöst hat, gefunden, so erfolgt eine Verzweigung auf das zugehörige Ein-/Ausgabeprogramm (Bild 7-7b). – Bei beiden Varianten muß dafür gesorgt werden, daß die jeweilige Anforderung nach ihrer Bearbeitung wieder aufgehoben wird. Das geschieht durch explizites oder implizites Rücksetzen des entsprechenden Ready-Bits (siehe dazu die Interface-Bausteine in 7.3.2 und 7.5.3).

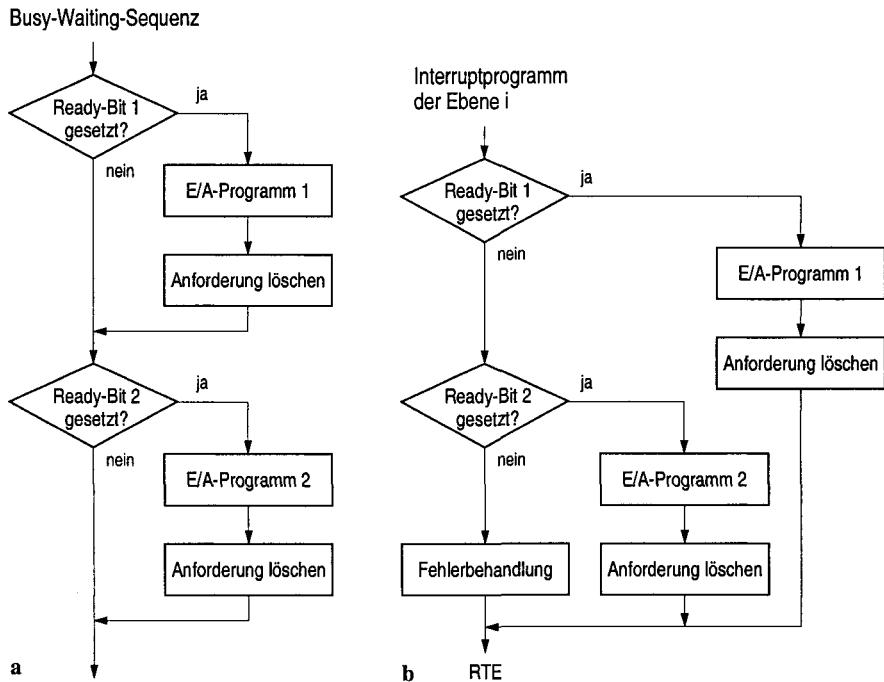


Bild 7-7. Polling. a Abfragen von Übertragungsanforderungen bei Busy-Waiting mit „fairer“ Priorisierung, b Identifizierung von Interruptanforderungen derselben Interrupebene mit „unfairer“ Priorisierung

Bezüglich der Priorisierung der einzelnen Anforderungen zeigt Bild 7-7a ein Ablaufschema, das die einzelnen Anforderungen quasi gleich behandelt. Das heißt, bei jedem Durchlauf durch die Polling-Routine werden alle Anforderungen abgearbeitet. Es gibt hier zwar eine Priorisierung bzgl. der Reihenfolge der Abarbeitung, aber es kommt jeder der Anforderer zum Zuge. Man bezeichnet dies auch als *faire Priorisierung*. Das Ablaufschema in Bild 7-7b räumt hingegen dem Anforderer 1 einen höheren Rang als dem Anforderer 2 ein, und zwar derart, daß der Anforderer 2 nicht zum Zuge kommt, wenn mehrere Anforderungen 1 schnell genug aufeinander folgen. Man bezeichnet dies als *unfair Priorisierung*. Natürlich können bei beiden Anwendungen des Polling (Busy-Waiting, Programmunterbrechung) wahlweise beide Priorisierungstechniken – fair oder unfair – angewendet werden.

Unterbrechbarkeit von Ein-/Ausgabeprogrammen. Bei der Ein-/Ausgabe synchronisation durch Busy-Waiting ist ein Ein-/Ausgabeprogramm durch eine Anforderung höherer Priorität nicht unterbrechbar, da sie vom Prozessor, während er das Ein-/Ausgabeprogramm ausführt, nicht erkannt wird. Bei der Synchronisation durch Programmunterbrechung obliegt die Priorisierung der Interrupeebenen der Hardware, womit das Unterbrechen eines laufenden Ein-/Aus-

gabeprogramms durch eine Anforderung höherer Priorität grundsätzlich möglich ist. Solche Unterbrechungen führen zu einer Schachtelung von Interruptprogrammausführungen, wobei für das *Statusretten* und Statusladen die gleichen Bedingungen wie bei reentranten Unterprogrammen gelten (3.3.3). Das heißt, unmittelbar mit Eintritt in das Interruptprogramm muß der Status gerettet und beim Verlassen des Programms wieder geladen werden. Für die Inhalte der Register PC und SR übernimmt das bei CISCs die Prozessorhardware, die dazu den Supervisor-*Stack* benutzt (2.1.5). Für die Inhalte der allgemeinen Register muß das programmiert werden, wozu z.B. der MOVEM-Befehl zur Verfügung steht (3.3.1). Als Speicherort bietet sich ebenfalls der Supervisor-Stack an, da die Unterbrechungsstruktur der Struktur der Unterprogrammschachtelung gleicht und damit dem *LIFO-Prinzip* folgt. Bei vielen RISCs hingegen, z.B. beim SPARC, muß auch das Retten des Statusregisters programmiert werden (3.4.3); das Speichern des PC erfolgt beim SPARC in den Fenstern des zu diesem Zweck stackartig organisierten Registerspeichers (2.2.1, 2.2.5).

Bei einem Mikroprozessor mit *codierten Interruptanforderungen* (5.5.1) setzt der Prozessor bei einer Anforderung der Ebene i die *Interruptmaske* in seinem Statusregister auf diesen Wert, womit er Anforderungen in einer Ebene größer als i als höherpriorisiert zuläßt. Erst mit Verlassen des Interruptprogramms durch den RTE-Befehl wird der alte Prozessorstatus, bestehend aus PC und SR, und damit auch die ursprüngliche Interruptmaske wieder geladen, so daß eine erneute Programmunterbrechung der Ebene i möglich ist. Sollen im Ausnahmefall bei Ausführung eines Interruptprogramms der Ebene i auch Unterbrechungen gleicher oder niedrigerer Priorität zugelassen werden, so muß die Interruptmaske durch das laufende Interruptprogramm auf einen Wert kleiner als i gesetzt werden. Dies ist nur durch den privilegierten Befehl MOVESR möglich, mit dem sich das Supervisor-Byte im Statusregister ändern läßt. Damit kann das durch die Interruptebenen des Prozessors vorgegebene Prioritätensystem durchbrochen werden. – Bei einem Mikroprozessor mit *uncodierten Anforderungen* (5.5.2), muß, um eine Unterbrechbarkeit zu ermöglichen, in jedem Fall das Interruptmaskenbit im Statusregister des Prozessors durch das laufende Interruptprogramm zurückgesetzt werden. Bei geeigneter prozessorexterner Priorisierungslogik werden nur höherpriorisierte Anforderungen an den Prozessor weitergeleitet (5.5.2).

7.2 Allgemeine Übertragungsmerkmale

Die digitale Übertragung von Daten zwischen dem Systembus eines Rechners und seinen peripheren Komponenten, den Ein-/Ausgabegeräten und Hintergrundspeichern, erfolgt durch Datenverbindungen, deren physische und funktionelle Eigenschaften durch einige grundlegende Merkmale geprägt sind. Diese Merkmale werden im folgenden kurz skizziert. Da einige von ihnen auch für Rechnernetze gelten, verwenden wir dabei anstelle des Begriffs *Ein-/Ausgabe*, wie er bei

der Kommunikation mit peripheren Geräten gebräuchlich ist, einheitlich den übergeordneten Begriff *Datenübertragung*, wie er für die Kommunikation in Rechnernetzen gebräuchlich ist.

7.2.1 Punkt-zu-Punkt- und Mehrpunktverbindung

Für den Anschluß peripherer Geräte (oder, allgemein, peripherer Einheiten) an ein Rechnersystem, genauer, an dessen Systembus, gibt es zwei typische Verbindungsarten: die Punkt-zu-Punkt- und die Mehrpunktverbindung.

- Die *Punkt-zu-Punkt-Verbindung* erlaubt die Kommunikation zwischen zwei Übertragungspartnern unter Benutzung eines Übertragungsweges, der nur diesen beiden Partnern zur Verfügung steht. Bild 7-8 zeigt eine solche Verbindung als Variante (1), mit der z.B. ein Drucker mittels eines Interface-Bausteins oder eine Bildschirmeinheit mittels eines Grafik-Controllers direkt mit dem Systembus verbunden ist. Werden mehrere Geräte nebeneinander am Systembus betrieben, so hat jedes seinen eigenen Systembusanschluß.
- Die *Mehrpunktverbindung* erlaubt die Kommunikation zwischen mehreren Übertragungspartnern über einen Übertragungsweg, der ihnen allen gemeinsam zur Verfügung steht, einen *Bus*. Bild 7-8 zeigt eine solche Verbindung als Variante (2), mit der z.B. ein Festplattenlaufwerk, ein CD-ROM-Laufwerk und ein Scanner über einen *Bus-Controller* an den Systembus angeschlossen sind. Der Bus-Controller für solche *Peripheriebusse* wird häufig auch als *Host-Adapter* bezeichnet.

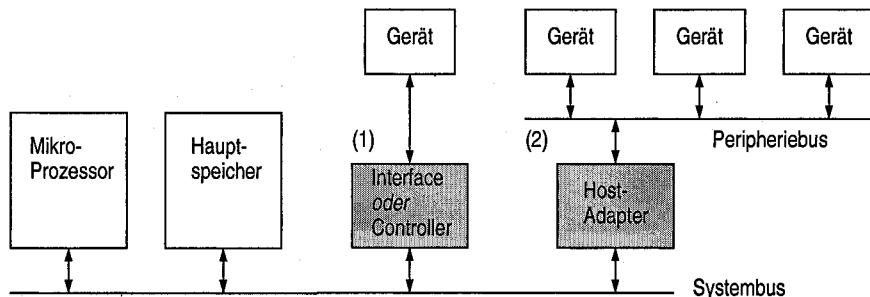


Bild 7-8. Rechnersystem mit zwei typischen Möglichkeiten der Anbindung interner oder externer Geräte an den Systembus: (1) Punkt-zu-Punkt-Verbindung für den Anschluß eines einzigen Geräts mittels eines (passiven) Interface-Bausteins oder eines (aktiven) Controllers, (2) Mehrpunktverbindung (Peripheriebus) für den Anschluß mehrerer Geräte mittels eines (aktiven) Host-Adapters

Unter der Bezeichnung Peripheriebus werden allerdings auch Verbindungen geführt, die in ihrer Struktur zwar wie ein Bus aussehen, die sich logisch jedoch wie aneinandergereihte Punkt-zu-Punkt-Verbindungen verhalten. Anders als bei einem Bus üblich, wird hier die zu übertragende Information nicht allen Übertra-

gungspartnern zugleich angeboten. Vielmehr wird sie von einem Übertragungspartner an den nächsten weitergereicht, bis der Adressat gefunden ist. In einer Erweiterung durch Wegeverzweigungen (Hubs) führt dies zu Strukturen wie Stern und Baum.

Beide Verbindungsarten, die Punkt-zu-Punkt-Verbindung wie auch die Mehrpunktverbindung, sind sowohl für den Anschluß rechnerinterner als auch rechnerexterner Geräte gebräuchlich. Peripheriebusse, wie der SCSI-Bus, werden auch in kombinierter Nutzung eingesetzt. Rechnerinterne Verbindungen waren zunächst parallele Verbindungen mit kurzen Weglängen; sie werden derzeit durch serielle Verbindungen verdrängt. Rechnerexterne Verbindungen sind als parallele Verbindungen mit ihren Weglängen auf die unmittelbare Umgebung des Rechners beschränkt, als serielle Verbindungen können sie – ähnlich wie Rechnernetzverbindungen – relativ große Entfernung überbrücken (siehe auch 7.7.2).

Bei Punkt-zu-Punkt-Verbindungen spricht man, wenn sie, wie in Bild 7-8 dargestellt, den direkten Anschluß eines Geräts an einen Rechner bilden, von *Schnittstellen*. Mehrpunktverbindungen hingegen bezeichnet man, wie oben bereits erwähnt, einheitlich als *Peripheriebusse*, auch wenn die Struktur stern- oder baumförmig ist. Aufgrund der unterschiedlichen Komplexität von Schnittstellen und Peripheriebusen trennen wir die beiden Teilbereiche und behandeln Schnittstellen samt ihren Techniken zur Datenübertragung in den folgenden Abschnitten dieses Kapitels. Peripheriebusse als die komplexeren Gebilde betrachten wir hingegen in Kapitel 8.

7.2.2 Schnittstellen

In der Hardwaretechnik beschreibt eine *Schnittstelle* die Trennlinie zwischen zwei Hardwarekomponenten, genauer, die Signalleitungen, die die beiden Komponenten miteinander verbinden. Nach DIN 44 302, in der Begriffe der Datenübertragung und Datenübermittlung im Umfeld der Datenfernübertragung und der Rechnernetze definiert sind, umfaßt diese Beschreibung die Gesamtheit der Festlegungen bezüglich

1. der physikalischen (elektrotechnischen) Eigenschaften der Signalleitungen,
2. der auf den Signalleitungen ausgetauschten (einzelnen benannten) Signale und
3. der Bedeutung (Funktion) der ausgetauschten Signale.

Dieser Schnittstellenbegriff ist elementar, indem er nur die für eine Datenverbindung benötigten Signalleitungen und die Bedeutung der Signale beschreibt. Er deckt sich mit den Festlegungen serieller Schnittstellen, wie sie für die Datenfernübertragung und in Rechnernetzen gebräuchlich sind (siehe 7.4). Will man nun Daten übertragen, z.B. zeichenweise, so reichen diese elementaren Vorgaben nicht aus. Vielmehr benötigt man eine Festlegung für die zeitliche Aufeinander-

folge der Signale für das Einleiten, Durchführen und Beenden einer Zeichenübertragung, ein sog. *Übertragungsprotokoll*, kurz *Protokoll*. Solche Protokolle werden für serielle Schnittstellen getrennt festgelegt, wodurch man in der Protokollwahl flexibel ist. Bei parallelen Schnittstellen sind sie üblicherweise in den Schnittstellenfestlegungen mit enthalten (siehe 7.3, zu Protokollen siehe auch 7.6.2).

Für eine Vereinheitlichung von Schnittstellen sorgen im wesentlichen nationale und internationale Normungsgremien. Einige der bekanntesten Gremien für Schnittstellen im Bereich der Datenübertragung in Rechnern und in Netzen sind:

- IEEE (USA): Institute of Electrical and Electronics Engineers (IEEE-Normen),
- EIA (USA): Electronic Industries Association (RS-Schnittstellen),
- ITU (international): International Telecommunication Union (ITU-T-Empfehlungen); zuvor CCITT: Comité Consultatif International Télégraphique et Téléphonique (V- und X-Empfehlungen),
- DIN: Deutsches Institut für Normung (DIN-Normen).

Zusätzlich zu den *Normen* gibt es sog. *Firmenstandards*, die sich durch Produkte einzelner Firmen oder durch Absprachen zwischen Firmen etabliert haben. Solche Festlegungen bilden häufig die Grundlage für spätere Normen.

Schnittstellen werden, wie Bild 7-8 zeigt, durch passive Interfaces, aktive Controller- oder Host-Adapter-Bausteine (ggf. erweitert um Signaltreiberbausteine) realisiert. Diese Bausteine sind naturgemäß mit zwei Schnittstellen unterschiedlicher Art ausgestattet, da sie ja das Signalverhalten zweier unterschiedlicher Hardwarekomponenten anpassen müssen. Die eine Schnittstelle ist die hier betrachtete *Datenübertragungsschnittstelle* zu einem Gerät, einem Peripheriebus oder einem Rechnernetz, die andere ist die zum Systembus. Die *Systembus-schnittstelle* ist entweder proprietär, d.h. herstellerspezifisch, z.B. als Prozessorbusschnittstelle, oder sie ist standardisiert oder genormt, z.B. als PCI-Bus-Schnittstelle.

7.2.3 Software- und hardwaregesteuerte Datenübertragung

Bei der *Datenübertragung* zwischen Interface (oder Host-Adapter) und Systembus (genauer, Hauptspeicher) unterscheidet man hinsichtlich Datentransport und Synchronisation zwischen rein softwaregesteuerter, hardwareunterstützter und rein hardwaregesteuerter Übertragung:

- Bei der rein *softwaregesteuerten* Datenübertragung unterliegen beide Aufgaben einem vom Prozessor auszuführenden Übertragungsprogramm. Der Datentransport wird mit einem Move- oder Load- bzw. Store-Befehl durch-

geführt, die Synchronisation durch Busy-Waiting, wie in 7.1.2 für die Abfrage des Signals READY gezeigt. Wie dort erwähnt, wird ggf. auch das Handshake-Signal ACKN durch das Programm gesteuert, wozu das Interface in seinem Steuerregister CR ein Bit bereitstellt, mit dem dieses Signal gesetzt und rückgesetzt werden kann.

- Bei der *hardwareunterstützten* Datenübertragung übernimmt das Übertragungsprogramm des Prozessors weiterhin den Datentransport, die Synchronisation erfolgt jedoch durch Programmunterbrechung. Das Interface muß hierzu aus dem sonst per Busy-Waiting abzufragenden Statusregisterbit Ready ein Interruptsignal erzeugen (7.1.2). Man spricht bei einem solchen Interface auch von *interrupt-fähiger Schnittstelle*.
- Bei der rein *hardwaregesteuerten* Datenübertragung wird der Datentransport von einer gesonderten Steuereinheit, einem DMA-Controller, übernommen (8.1). Das Interface muß hierzu aus dem Ready-Bit des Statusregisters ein Anforderungssignal für den DMA-Controller erzeugen. Außerdem muß es beim Handshake-Betrieb das ACKN-Signal automatisch erzeugen, so wie in 7.1.2 gezeigt. Man spricht dann von *DMA-fähiger Schnittstelle*.

7.2.4 Serielle und parallele Datenübertragung

Bei der *seriellen Datenübertragung* werden die einzelnen Bits eines Zeichens nacheinander in einem festen Schrittakt auf einer einzigen Datenleitung übertragen; man bezeichnet das auch als bit serielle Datenübertragung. Bei der *parallelen Datenübertragung* steht hingegen für jedes Bit eines Zeichens eine eigene Leitung zur Verfügung und alle Bits eines Zeichens werden parallel übertragen. Man bezeichnet das auch als bit parallele oder zeichenserrielle Datenübertragung. Die parallele Datenübertragung erfolgt häufig aber auch unabhängig von Zeichenformaten mit einer von der Anwendung bestimmten Anzahl von Datenleitungen, z.B. bei Steuerungsaufgaben. – Einen Sonderfall paralleler Übertragung stellt die Codierung mehrerer Bits innerhalb eines Schrittakts dar, wie sie durch Modulationstechniken ermöglicht wird. Hierbei werden mehrere Bits gleichzeitig über eine einzige Datenleitung übertragen, indem sie z.B. bei analoger Übertragung durch den Phasenanschnitt der als Übertragungssignal benutzten Sinusschwingung codiert werden (siehe 7.7.2).

7.2.5 Synchrone und asynchrone Datenübertragung

Bei der *synchronen Datenübertragung* unterliegt die Übertragung aufeinanderfolgender Zeichen einem festen Zeitraster, das für die Gesamtdauer der Datenübertragung aufrechterhalten wird. Dieses Zeitraster wird entweder durch ein für Sender und Empfänger gemeinsames Takt signal festgelegt (üblich bei paralleler

Übertragung). Oder der Sender und der Empfänger besitzen jeweils eigene Taktgeneratoren gleicher Frequenz, und der Empfänger synchronisiert sich fortlaufend mittels des übertragenen Datensignals mit dem Sendertakt (üblich bei serieller Übertragung). Bei der *asynchronen Datenübertragung* sind die Zeitabstände zwischen den einzelnen Zeichentransporten variabel. Sender und Empfänger synchronisieren sich entweder über Steuer- und Statusleitungen, mit denen sie sich bei jeder Zeichenübertragung ihre Bereitschaft übermitteln, z.B. in Form eines *Handshake-Protokolls* (üblich bei paralleler Übertragung). Oder sie haben wiederum jeweils eigene *Taktgeneratoren* gleicher Frequenz und synchronisieren sich mittels des Datensignals; hier jedoch jeweils nur für die Dauer einer einzelnen Zeichenübertragung (üblich bei serieller Übertragung).

7.2.6 Simplex-, Halbduplex- und Duplexbetrieb

Hinsichtlich der Richtungsvorgabe der Datenübertragung gibt es drei Betriebsarten, die sich durch unterschiedlichen Aufwand an Übertragungswegen und Sender- und Empfängereinrichtungen unterscheiden: den Simplex-, den Halbduplex- und den Duplexbetrieb (Bild 7-9). Beim *Simplexbetrieb* ist die Datenübertragung in nur einer Richtung möglich. Es gibt nur einen Übertragungsweg mit einem Sender am einen und einem Empfänger am anderen Ende des Weges. Man bezeichnet eine solche Verbindung auch als *unidirektional*. Beim *Halbduplexbetrieb* ist die Datenübertragung in beiden Richtungen möglich, jedoch nicht gleich-

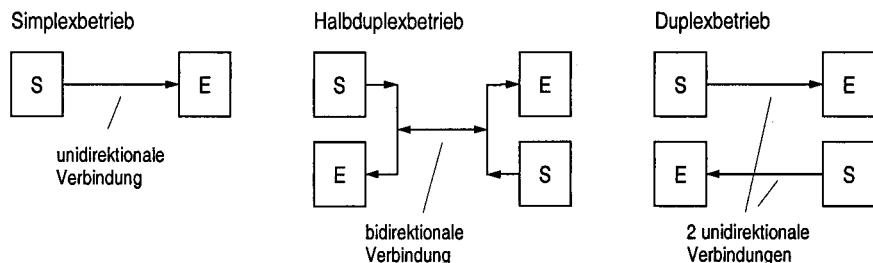


Bild 7-9. Sender-Empfänger-Anordnungen für den Simplex-, Halbduplex- und Duplexbetrieb mit unidirektionalen bzw. bidirektionalen Leitungsverbindungen

zeitig. Es gibt wiederum nur einen Übertragungsweg, jedoch an beiden Enden je einen Sender und einen Empfänger, die je nach Übertragungsrichtung wahlweise an die Signalleitung angekoppelt werden. Man bezeichnet eine solche Verbindung auch als *bidirektional*. Beim *Duplexbetrieb* (auch *Vollduplexbetrieb* genannt) ist die Übertragung von Daten in beiden Richtungen und zur gleichen Zeit möglich. Für jede Richtung existiert dazu ein eigener Übertragungsweg mit jeweils eigenem Sender und Empfänger. Gegenüber dem Halbduplexbetrieb verdoppelt sich der Leitungsaufwand; dafür entfällt die Steuerung für die Sender-

Empfänger-Umschaltung an den Endpunkten. Die beide Verbindungen wirken jeweils unidirektional (als zwei Verbindungen mit Simplexbetrieb).

7.2.7 Übertragungs-, Transfer- und Schrittgeschwindigkeit

Die *Übertragungsgeschwindigkeit*, auch als *Übertragungsrate* bezeichnet, gibt bei serieller Übertragung die Anzahl der übertragenen Bits pro Sekunde (bit/s, *bps*) und bei paralleler Übertragung die Anzahl der übertragenen Bytes pro Sekunde (byte/s) an. Wird ein Teil der übertragenen Bits zur Fehlererkennung benutzt, so ergibt sich für die eigentliche Datenübertragung eine geringere Geschwindigkeit. Man bezeichnet diese dann als *effektive Übertragungsgeschwindigkeit*. Ein Teilspekt der Übertragungsrate ist die Anzahl an Transfers pro Sekunde, die *Transferrate* (T/s; siehe dazu auch 5.1.4, S. 274). Die *Schrittgeschwindigkeit* ist ein Maß für die serielle Übertragung und gibt die Anzahl der Taktzritte (Übertragungsschritte) pro Sekunde an. Sie hat die Maßeinheit Baud, benannt nach dem französischen Techniker Baudot, und wird häufig inkorrekt auch als „*Baud-Rate*“ bezeichnet. Bei rein serieller Übertragung ist sie gleich der Übertragungsgeschwindigkeit. Hier ist die Maßzahl für Baud gleich der für bit/s. Im oben erwähnten Sonderfall der seriellen Übertragung, bei dem mehrere Bits innerhalb eines Schritts codiert übertragen werden, gilt dies jedoch nicht. Hier ist die Übertragungsgeschwindigkeit um den Faktor der Parallelität höher als die Schrittgeschwindigkeit. – Die gebräuchlichen Einheitenvorsätze für Geschwindigkeiten/Übertragungsraten sind $k = 10^3$, $M = 10^6$ und $G = 10^9$.

Die auf Leitungsverbindungen maximal mögliche Übertragungsrate hängt wesentlich von Leitungseigenschaften, wie Dämpfung, Übersprechen und Signalreflexion an den Leitungsenden, ab. Hier erweisen sich insbesondere parallele Verbindungen als sehr empfindlich, weshalb sie auf kurze Weglängen eingeschränkt, mit Masseleitungen zwischen den Signalleitungen versehen und ggf. mit Leitungsabschlüssen (Terminatoren) versehen werden. Bei seriellen Verbindungen sind diese Probleme aufgrund der geringen Leitungsanzahl relativ gut zu lösen. So kann für hohe Übertragungsraten z.B. die *differentielle Signaldarstellung* verwendet werden, wodurch Störungen auf den ggf. sehr großen Wegstrecken aufgehoben werden. Das Signal wird dazu vom Sender mit normalem und mit invertiertem Pegel auf zwei Signalleitungen übertragen und vom Empfänger durch einen Differenzverstärker übernommen. – Zu Leitungsarten siehe 7.7.1: Übertragungsmedien.

7.3 Parallele Schnittstellen

Parallele Schnittstellen und die sie realisierenden Interface-Bausteine lassen sich grob gesehen in zwei Kategorien unterteilen:

- *Universelle Schnittstellen:* Sie haben kein festes Übertragungsprotokoll, sondern erlauben eine individuelle Anpassung an unterschiedliche Peripherie. Hierbei werden einerseits Daten in den üblichen Datenformaten übertragen und synchronisiert, andererseits erlauben sie aber auch die unsynchronisierte Einzelbitübertragung, d.h. das Senden von Steuersignalen und das Empfangen von Statussignalen. Typische Anwendungen sind die Steuerung und Regelung technischer Abläufe, z.B. technischer Prozesse (Prozeßdatenverarbeitung).
- *Spezielle Schnittstellen:* Sie haben ein festes, meist standardisiertes Übertragungsprotokoll, angepaßt an eine bestimmte Art von Peripheriegerät(en). Hierbei sind das Datenformat und die Übertragungssteuerung vorgegeben, wie auch die Anzahl und Funktion weiterer Status- und Steuerleitungen für die Gerätesteuerung. Typische Anwendungen sind der Anschluß von peripheren Speichern und Ein-/Ausgabegeräten an Rechner, z.B. von Festplattenlaufwerken, CD-ROM-Laufwerken, Druckern und Scannern.

Wir werden in 7.3.1 zunächst einige allgemeine Aspekte zu universellen Parallel-Schnittstellen erörtern, um dann in 7.3.2 einen Interface-Baustein im Detail zu betrachten. In den restlichen Abschnitten 7.3.3 und 7.3.4 konzentrieren wir uns dann auf die Protokolle gebräuchlicher spezieller Schnittstellen.

7.3.1 Universelle Parallel-Schnittstellen

Universelle Parallel-Schnittstellen werden – wie gesagt – sowohl für die parallele Datenübertragung, z.B. für den Anschluß von Meßgeräten, als auch für die Übertragung einzelner Bits in Verbindung mit Steuerungs- und Regelungskomponenten eingesetzt. Dementsprechend sind sie wesentliche Bestandteile von Mikrocontrollern, sie sind aber auch als separate Interface-Bausteine gebräuchlich, wie in 7.3.2 betrachtet. Diese sog. *Ein-/Ausgabetore (i/o ports)* haben meist 8-Bit-breite Datenwege, und es sind oft mehrere Tore in einem Baustein verfügbar. Ergänzt werden die Datenleitungen durch Steuerleitungen für die Interruptverarbeitung und für die Synchronisation im Handshake-Betrieb. Die Schnittstellen signale folgen üblicherweise dem TTL-Standard, was einen einfachen technischen Aufbau ermöglicht.

Datenformate. Das Übertragen von Zeichen im 7-Bit-ASCII erfolgt im 7-Bit- oder 8-Bit-Format. Bei acht Bits ist das höchstwertige Bit entweder mit Null oder mit Eins festgelegt, oder es dient als Paritätsbit (siehe 7.7.3). Peripheriegeräte, die ausschließlich mit Dezimalziffern arbeiten, übertragen diese als 4-Bit-BCD-Zeichen. Für dieses Datenformat (Halbbyte) reichen vier Datenleitungen aus, oder es werden, um ein 8-Bit-Tor besser zu nutzen, zwei Zeichen gleichzeitig übertragen. Analoge Meßwerte, die über Analog-Digital-Umsetzer eingegeben, und Stellgrößen, die über Digital-Analog-Umsetzer ausgegeben werden, umfas-

sen entsprechend der Genauigkeit von Analogwerten bis zu 16 Bits oder auch mehr. Hier werden zwei oder drei 8-Bit-Tore parallel betrieben. – Beim Übertragen einzelner Bits können die Datenleitungen beliebig eingesetzt werden, innerhalb eines Tores für die Eingabe und die Ausgabe gemischt.

Datenpufferung und Synchronisation. Einfache Parallel-Interface-Bausteine sehen pro Ein-/Ausgabetor nur ein einziges Datenregister zur Datenpufferung vor, das zudem nur in Richtung Ausgabe wirkt (Bild 7-10a). Bei der Eingabe erfolgt der Datenzugriff des Prozessors ebenfalls durch Adressieren dieses Pufferregisters. Da das Eingabedatum jedoch dort nicht gespeichert ist, sondern nur an den peripheren Datenleitungen anliegt, muß es von der Peripherie so lange gehalten werden, bis der Prozessor seine Übernahme durch ein Synchronisationssignal bestätigt hat. Die Peripherie benutzt dazu ggf. einen gleichartigen Interface-Baustein. Aufwendigere Interface-Bausteine haben ein zusätzliches Pufferregister für die Eingabe, womit die Peripherie entlastet wird (Bild 7-10b). – Für peripherie Einheiten mit hohen Übertragungsgeschwindigkeiten gibt es Interfaces mit je zwei (oder mehr) Pufferregistern pro Übertragungsrichtung. Diese Registerpaare werden nach dem Warteschlangenprinzip verwaltet: Das zuerst gespeicherte Datum wird auch als erstes Datum wieder ausgegeben (*FIFO-Prinzip*: first-in first-out). Dies ermöglicht den Zugriff des Prozessors auf das eine Register, während die Peripherie noch mit dem Laden (Eingabe) bzw. mit dem Lesen (Ausgabe) des anderen Registers beschäftigt ist.

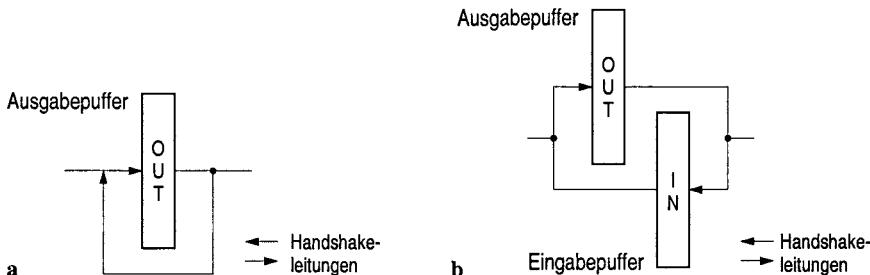


Bild 7-10. Datenpufferung bei paralleler Ein-/Auszgabe, a Pufferung nur bei Ausgabe, b Pufferung bei Ausgabe und Eingabe

7.3.2 Parallel-Interface-Baustein

Im folgenden werden der prinzipielle Aufbau und die Funktionsweise eines Parallel-Interface-Bausteins betrachtet, wie er zur Realisierung einer universellen Parallel-Schnittstelle eingesetzt werden kann. Er ist auf der Systembusseite mit acht Datenanschlüssen ausgestattet, d.h., die Datenübertragung zwischen Mikroprozessor und Interface erfolgt in Bytes entsprechend der byteorientierten Informationsstruktur vieler peripherer Komponenten. Auf der Peripherieseite gibt es

zwei voneinander unabhängige Ein-/Ausgabetore mit ebenfalls je acht Datenleitungen. Sie erlauben einerseits die handshake-synchronisierte Ein- und Ausgabe, üblicherweise im Byteformat, andererseits aber auch die Übertragung einzelner Bits mit für jedes Bit individueller Richtungsvorgabe. Beide Tore sehen darüber hinaus das Verwalten von Interruptanforderungen der Peripherie vor.

Blockstruktur. Bild 7-11 zeigt die Struktur des Parallel-Interface-Bausteins mit seinen beiden Ein-/Ausgabetoren A und B, wovon das Tor A durch Angabe des Datenflusses detaillierter dargestellt ist. Für jedes Tor besitzt der Baustein ein Datenregister DR zur Datenpufferung in Ausgaberichtung. Für die Eingabe wird eine Pufferung auf der Peripherieseite vorausgesetzt (siehe auch Bild 7-10a). Er besitzt weiterhin ein zentrales Steuerwerk und jeweils drei Register für jedes der beiden Tore: ein Datenrichtungsregister DDR (data direction register) zur Richtungsvorgabe für die einzelnen Datenleitungen, ein Steuerregister CR (control register) zur Programmierung der Wirkungsweise zweier Steuerleitungen C1 und C2 und ein Statusregister SR mit zwei mit C1 und C2 korrespondierenden Ready-/Interrupt-Bits. Jedes Tor hat außerdem eine Steuerlogik für diese beiden Leitungen (Synchronisation) sowie zur Erzeugung der Signale DMARQ (DMA request) und \overline{IRQ} (interrupt request). Hinzu kommen Einrichtungen für die Selbstidentifizierung der Tore als Interruptquellen (Vektornummerregister VNR) und zur Priorisierung.

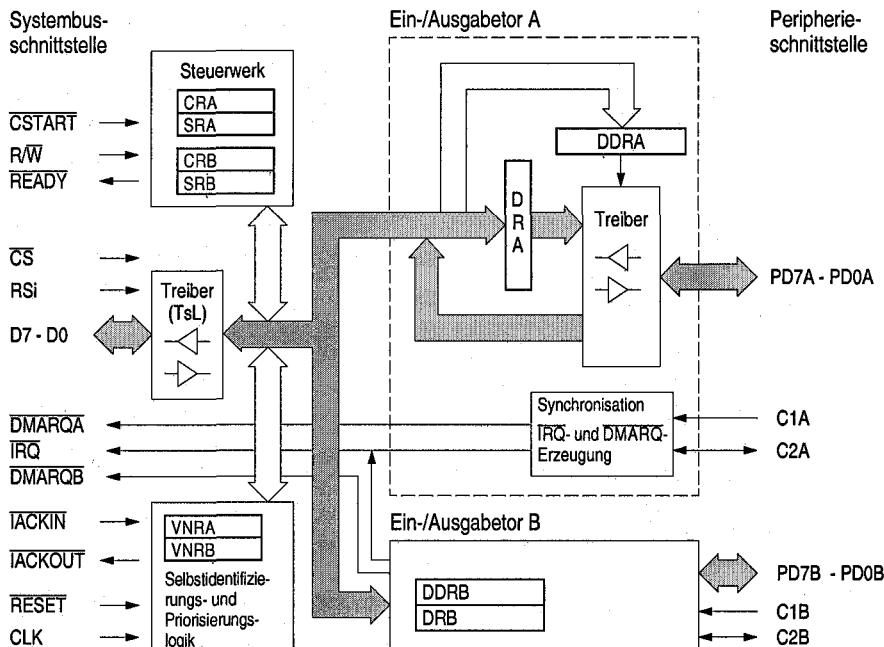


Bild 7-11. Struktur eines universellen Parallel-Interface-Bausteins mit zwei 8-Bit-Ein-/Ausgabetoren

risierung der Interruptanforderungen (IACK-Verkettung als Daisy-Chain), die bei einfacheren Baustinausführungen fehlen können.

Die Schnittstelle zum Systembus sieht folgende Anschlüsse vor: acht bidirektionale Datenleitungen D7 bis D0 mit Tristate-Logik (TsL), über die die Baustine register beschrieben bzw. gelesen werden; einen Chip-Select-Eingang CS zur Baustineanwahl; mehrere Register-Select-Eingänge RS_i zur Registeranwahl; ferner die zur Steuerung der Buszyklen notwendigen Anschlüsse CSTART, R/W und READY (bei synchronem Bus). Hinzu kommen die Ausgänge DMARQA und DMARQB für Anforderungen an einen DMA-Controller sowie der für beide Tore gemeinsame Open-collector-Ausgang IRQ als Interruptleitung und die Anschlüsse IACKIN und IACKOUT für die Bausteinverkettung in einer *Interrupt-Daisy-Chain*. Über den Eingang RESET wird der Baustein in den Initialisierungszustand gebracht, über den CLK-Eingang wird er mit dem Arbeitstakt (hier mit dem Bustakt) versorgt.

Die Schnittstelle zur Peripherie sieht für jedes der beiden Tore acht bidirektionale Datenleitungen PD7 bis PD0 (peripheral data) sowie die oben genannten Steuerleitungen C1 und C2 (control) vor. C1 und C2 werden entweder für die Handshake-Synchronisation eingesetzt, mit C1 als Eingangs- und C2 als Ausgangssignal, oder sie werden beide als Eingangssignale verwendet. In ihrer Handshake-Funktion entsprechen sie den in 7.1.2 verwendeten Signalen READY und ACKN. Ihre Bezeichnungen sind jedoch im Hinblick darauf, daß sich die ihnen im Handshake-Betrieb zugeordneten Funktionen „Bereitstellung“ und „Quittierung“ zwischen Eingabe und Ausgabe vertauschen, hier allgemeiner gewählt. Als Eingang wirkt jede der Steuerleitungen auf ein eigenes Bit im Statusregister, das als Ready-Bit abfragbar ist (Synchronisation durch Busy-Waiting) oder als Interrupt-Bit eine Interruptanforderung auslösen kann (Synchronisation durch Programmunterbrechung).

Datenrichtungsregister. Jede einzelne Datenleitung der Peripherieschnittstelle eines Tores kann individuell in ihrer Übertragungsrichtung festgelegt werden, und zwar durch die korrespondierenden Bits des torspezifischen Datenrichtungsregisters DDR. Mit dem Bit DDR_i = 0 wird die Datenleitung *i* im Datenbustreiber auf Eingabe und mit DDR_i = 1 auf Ausgabe geschaltet. Bezogen auf die Handshake-Synchronisation können einzelne Leitungen in der Synchronisationsrichtung und andere, unsynchronisiert, entgegen dieser Richtung benutzt werden. Werden die Steuerleitungen C1 und C2 nicht als Handshake-Leitungen benutzt, so ist eine unsynchronisierte Einzelbitübertragung in beiden Richtungen möglich.

Steuerregister. Durch Laden eines Steuerbytes in das Steuerregister CR können die Wirkungsweisen von C1 und C2 wie folgt vorgegeben werden (Bild 7-12). C1-TRANS legt fest, welche Flanke (transition) des Eingangssignals C1 das Statusregisterbit SR7 setzt, und C1-IR bestimmt, ob das Setzen dieses Bits eine Interruptanforderung IRQ = 0 auslöst. Mit C1-DMA kann das Setzen von SR7 zugunsten der Erzeugung des DMARQ-Signals unterbunden werden. Mit CR4

wird entschieden, ob C2 als Eingangssignal oder als Handshake-Signal (Ausgang) wirkt. Als Eingangssignal haben C2-TRANS und C2-IR dieselbe Funktion wie für C1 beschrieben, jedoch SR6 betreffend. Im Handshake-Betrieb gibt C2-HS vor, ob C2 automatisch oder programmiert gesteuert wird. Bei automatischer Steuerung unterscheiden sich die beiden Tore, indem der Automatismus bei Tor A nur für die Eingabe und bei Tor B nur für die Ausgabe ausgelegt ist. Dementsprechend reagiert C2A nur bei einem Lesezugriff auf DRA und C2B nur bei einem Schreibzugriff auf DRB (siehe dazu Bild 7-14.) Bei programmierter Steuerung ist der Signalpegel von C2 gleich dem aktuellen Wert des Bits C2-STATE.

CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0	
C1-TRANS	C1-IR	C1-DMA	C2-MODE	C2-TRANS	C2-IR	C2-HS	C2-STATE	Steuerregister CR
								C2-Zustand: 0 0-Pegel, 1 1-Pegel
								C2-Handshake: 0 automatisch, 1 programmiert (durch CR1)
								C2-Interrupt: 0 blockiert, 1 zugelassen
								C2-Signalflanke: 0 fallend, 1 steigend
								C2-Betriebsart: 0 Interrupteingang (gesteuert durch CR4 und CR3), 1 Handshake-Ausgang (gesteuert durch CR2 und CR1)
								C1-DMA-Betrieb: 0 blockiert, 1 zugelassen
								C1-Interrupt: 0 blockiert, 1 zugelassen
								C1-Signalflanke: 0 fallend, 1 steigend

Bild 7-12. Steuerregister des Parallel-Interface-Bausteins nach Bild 7-11 mit Vorgaben für die beiden Steuerleitungen C1 (Interrupteingang oder DMA-auslösend) und C2 (Interrupteingang oder Handshake-Ausgang)

Statusregister. Der Status eines Tores wird in seinem Statusregister SR angezeigt (Bild 7-13). Hierbei hat das Bit C1-IRQ die Funktion des in 7.1.2 beschriebenen Ready-Bits, d.h., es wird mit Eintreffen des C1-Signals gesetzt. Das Bit C2-IRQ hat die gleiche Funktion für das C2-Signal, wenn C2 als Eingang programmiert

SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	
C1-IRQ	C2I-RQ	0	0	0	0	0	0	Statusregister SR
								unbenutzt
								C2-Interruptstatus: 0 keine Anforderung, 1 Anforderung
								C1-Interruptstatus: 0 keine Anforderung, 1 Anforderung

Bild 7-13. Statusregister des Parallel-Interface-Bausteins nach Bild 7-11 zur Anzeige von Interruptanforderungen der beiden Steuerleitungen C1 und C2

ist ($CR5 = 0$). Abhängig von der Interruptfreigabe durch $CR6$ ($C1\text{-IR}$) bzw. $CR2$ ($C2\text{-IR}$) aktiviert der 1-Zustand des einen oder des anderen Statusbits die Interruptleitung \overline{IRQ} . Das Rücksetzen dieser Statusbits erfolgt durch einen Schreibzugriff auf das Statusregister mit einer Maske, die für die entsprechende Bitposition eine Eins aufweist. – Die Statusbits $SR5$ bis $SR0$ sind ohne Funktion und liefern beim Lesen den Wert 0.

Handshake-Synchronisation. Die Wirkungsweise der Handshake-Synchronisation zeigt Bild 7-14 an den Beispielen der Eingabe über Tor A und der Ausgabe über Tor B. – Bei der Eingabe signalisiert die Peripherie die Gültigkeit des Eingabedatums mit der positiven Flanke von $C1A$, womit es in das Datenregister DRA übernommen wird. Als Folge davon setzt das Interface das Statusbit $SR7A$ und zeigt damit dem Prozessor an, daß das Datum in DRA bereitsteht. Außerdem signalisiert es mit dem Inaktivzustand von $C2A$ der Peripherie, daß es für eine weitere Eingabe nicht bereit ist. Der Prozessor wertet das Statusbit entweder

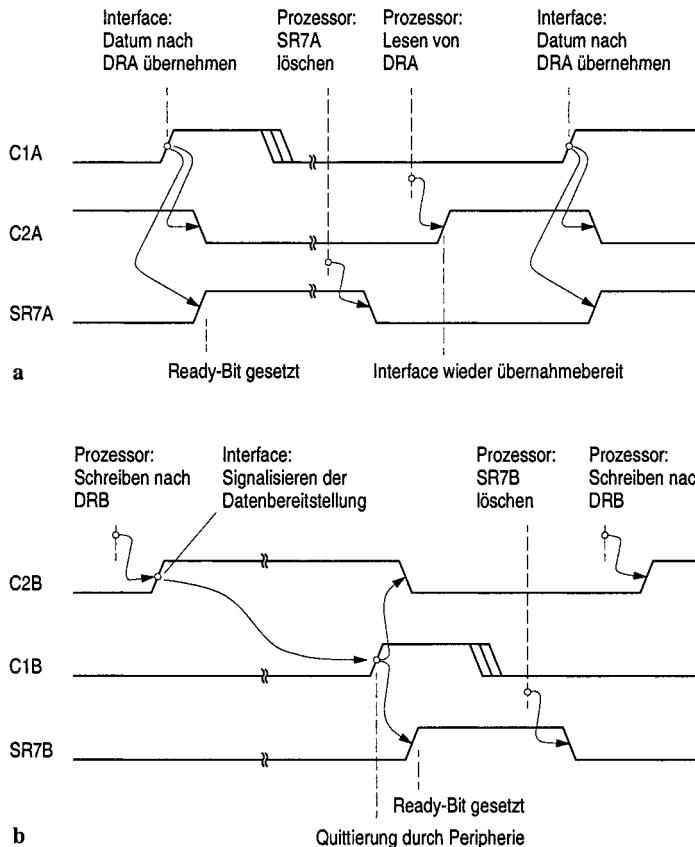


Bild 7-14. Handshake-Synchronisation. a) Eingabe über Tor A, b) Ausgabe über Tor B

durch Busy-Waiting oder als Interruptanforderung aus, setzt es zurück und liest den Inhalt von DRA. (Das Signal C1A wird nach einer bestimmten Zeit von der Peripherie wieder inaktiviert, d.h., es wirkt als Impuls.) Der Abschluß des Lesevorgangs wird der Peripherie durch Aktivieren des C2A-Signals (positive Flanke) angezeigt; sie kann danach das nächste Datum übertragen.

Bei der Ausgabe schreibt der Prozessor das Datum nach DRB, was das Interface mit positiver Flanke von C2B anzeigt. Diese Flanke wird von der Peripherie zur Datenübernahme ausgewertet. Sie quittiert ihrerseits die Übernahme mit positiver Flanke von C1B, woraufhin das Interface das Statusbit SR7B setzt und das C2B-Signal zurücksetzt. Der Prozessor wertet SR7B aus, löscht das Statusbit und schreibt das nächste Datum nach DRB.

Beispiel 7.3. ▶ Datenausgabe über den universellen Parallel-Interface-Baustein nach Bild 7-11 (S. 475). In Anlehnung an Beispiel 7.1 (S. 462) sollen 128 Datenbytes über den oben beschriebenen Parallel-Interface-Baustein an ein Peripheriegerät ausgegeben werden (Bild 7-15). Die Datenübertragung soll über die acht Datenleitungen von Tor B und durch automatisches Handshaking mit Busy-Waiting bei steigender Flanke von C1B erfolgen. Das Peripheriegerät wird zu Beginn durch Ausgeben einer Eins auf der Datenleitung PD0A (Tor A) gestartet und nach Abschluß der Übertragung mit PD0A = 0 wieder gestoppt.

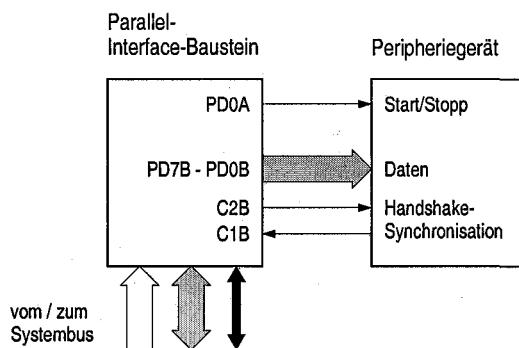


Bild 7-15. Anschluß eines Peripheriegeräts an den Parallel-Interface-Baustein nach Bild 7-11 für eine Datenausgabe mit Handshake-Synchronisation über Tor B. Starten und Stoppen des Geräts mittels einer Datenleitung von Tor A

Bild 7-16 zeigt den Ablauf. Zur Initialisierung des Interfaces wird zunächst das Steuerregister von Tor A, CRA, für die unsynchronisierte Einzelbit-Übertragung mit 0x00 geladen; ferner wird das Datenrichtungsregister DDRA mit 0x01 geladen, wodurch die Datenleitung PD0A auf Ausgabe geschaltet wird. Das Steuerregister von Tor B, CRB, wird mit 0x90 geladen, mit folgender Wirkung: Ausgabe im Handshake-Betrieb mit automatischer Steuerung von C2B, Setzen von SR7B mit steigender C1B-Flanke und Blockieren des SR7B-Interrupts. Weiterhin werden für das Tor B alle Datenleitungen mittels des Datenrichtungsregisters DDRB mit 0xFF auf Ausgabe festgelegt. (Grundsätzlich könnten einzelne Leitungen, unabhängig von der Synchronisation, auch in Gegenrichtung betrieben werden.)

Gestartet wird das Peripheriegerät durch das Laden des Datenregisters DRA mit 0x01, womit die Datenleitung PD0A auf Eins gesetzt wird. Das Gerät meldet daraufhin seine Bereitschaft durch

Setzen des Signals C1B (steigende Flanke); es setzt dieses Signal etwas später automatisch wieder zurück (Puls). Mit der steigenden Flanke von C1B wird das Ready-Bit SR7B im Interface auf Eins gesetzt und bewirkt dort grundsätzlich das Rücksetzen des Signals C2B. (Bei der ersten Byteübertragung bleibt das Rücksetzen von C2B allerdings ohne Wirkung, da C2B zuvor noch nicht gesetzt wurde.) Das Programm wartet in der Warteschleife WAIT auf SR7B = 1 (Busy-Waiting), setzt dann das Statusbit durch Beschreiben des Statusregisters mit der Maske 0x80 zurück und lädt schließlich das erste Datenbyte in das Datenregister DRB. Mit Abschluß des Ladens von DRB setzt das Interface das Signal C2B. Dieses wirkt im Peripheriegerät mit seiner steigenden Flanke als Strobe-Signal für die Datentransferübernahme. Das Peripheriegerät quittiert diese Übernahme durch erneutes Setzen des Signals C1B, was ein erneutes Setzen von SR7B bewirkt. Das Programm wartet auf dieses Setzen wieder in der Warteschleife WAIT (Busy-Waiting) und gibt dann das zweite Datenbyte aus. Nach Abschluß der Übertragung des letzten Datenbytes wird das Peripheriegerät durch Ausgeben einer Null auf der Datenleitung PD0A gestoppt. – Der Ablauf der zweiten und der folgenden Byteübertragungen entspricht der Darstellung in Bild 7-14b; die erste Übertragung unterscheidet sich davon durch die abweichende Ausgangssituation für C2B.

```
; Parallel Ausgabe von 128 Bytes  
; Handshaking mit Busy-Waiting
```

```

        ORG      0x40000
Buffer DS.B     128
        :
LEA      BUFFER,R0
LEA      BUFFER+128,R1

; Interface initialisieren
MOVE.B #0x00,CRA
MOVE.B #0x01,DDRA
MOVE.B #0x90,CRB
MOVE.B #0xFF,DDRB

; Ausgabe starten
MOVE.B #0x01,DRA - //]

; Datenblock ausgeben
WAIT: BTST.B #7,SRB    ← -
      BEQ     WAIT
      MOVE.B #0x80,SRB -- -
      MOVE.B (R0)+,DRB -- -
      CMP.W R0,R1
      BNE     WAIT

; Ausgabe stoppen
MOVE.B #0x00,DRA -- //

```

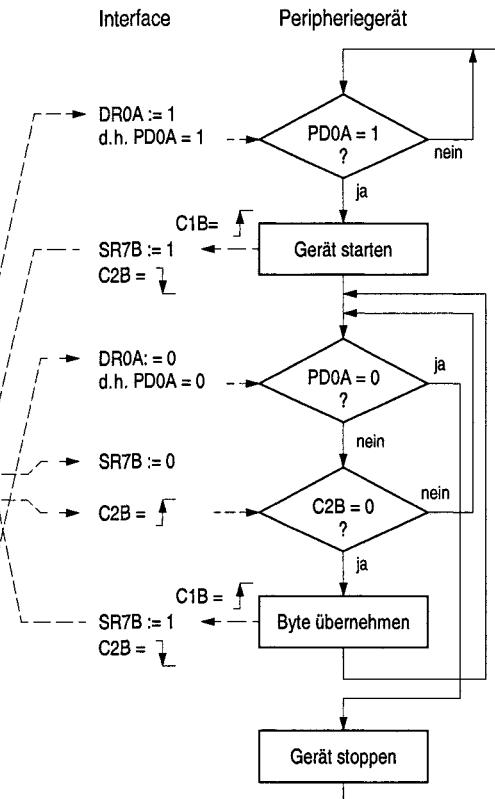


Bild 7-16. Ablauf zu Beispiel 7.3: Datenausgabe über den universellen Parallel-Interface-Baustein nach Bild 7-11 (S. 475)

Unsynchronisierte Einzelbit-Übertragung. Bild 7-17 zeigt als Anwendungsbeispiel für die Einzelbit-Übertragung die Steuerung eines industriellen Prozesses, durch die der Druck eines Kessels innerhalb eines bestimmten Druckbereichs über ein Heizaggregat und ein Überdruckventil geregelt werden soll. Zwei Alarmsignale zeigen das Verlassen des Druckbereichs, ein weiteres Alarmsignal das Überschreiten einer vorgegebenen Grenztemperatur an. Diese Signale werden als Interrupts mit fallenden Flanken über die Steuereingänge C1A, C2A und C1B der beiden Tore ausgewertet. Zusätzlich werden fünf Steuersignal- und eine Statussignalleitung eingesetzt, für die die Datenausgänge PD0A bis PD4A und der Dateneingang PD7A des Tores A genutzt werden. Die Festlegung der Datenleitungen als Ausgänge bzw. als Eingang erfolgt über das Datenrichtungsregister DDRA. Initialisiert werden die Bausteinregister für dieses Beispiel wie folgt: CRA = 0x44, CRB = 0x40, DDRA = 0x1F, DDRB = 0x00.

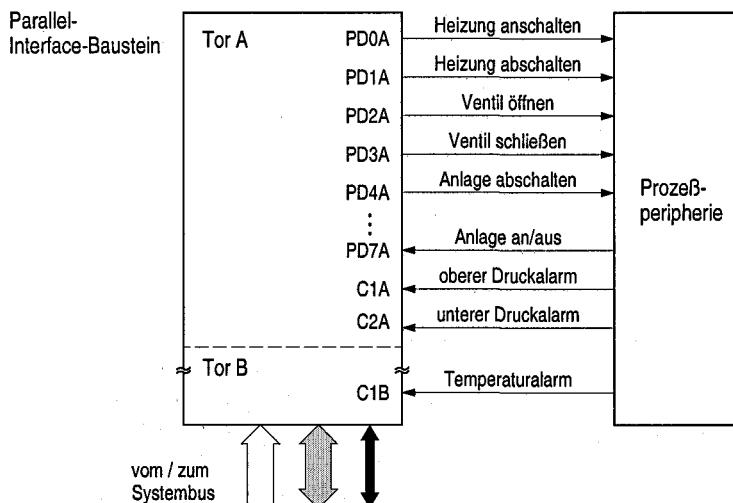


Bild 7-17. Prozeßsteuerung mit Übertragung von Steuer- und Statusinformation über den Parallel-Interface-Baustein nach Bild 7-11 (S. 475)

7.3.3 Centronics

Die *Centronics-Schnittstelle* ist eine nichtstandardisierte, „primitive“ Parallelschnittstelle für unidirektionale Verbindungen. Sie wurde von der Firma Centronics bereits Anfang der 70er Jahre für die zeichenweisen Datenausgabe an die von ihr hergestellten Drucker entwickelt und dann von fast allen Druckerherstellern wahlweise oder zusätzlich zur seriellen RS-232-C-Schnittstelle angeboten. Die Übertragung erfolgt durch Handshake-Synchronisation, d.h., es handelt sich vom Prinzip her um asynchrone Datenübertragung. Sie ist rein software-gesteuert.

ert, weshalb keine sehr hohen Übertragungsraten erreicht werden. Diese liegen bei Kabellängen von bis zu 2 m und bei Verdrillung von Signalleitungen mit Erdleitungen (Vermeidung des „Übersprechens“) bei 50 bis 150 kbyte/s. Größere Entfernungen sind bei reduzierten Übertragungsraten überbrückbar. Die Signalzustände sind durch die TTL-Pegel 0 V und 5 V festgelegt. Auf der Druckerseite wird üblicherweise ein 36-poliger, auf der Rechnerseite (PC) ein 25-poliger Steckverbinder verwendet. Auf der Druckerseite unterscheidet man dabei zwischen der Centronics-Stiftbelegung und der sog. PC-kompatiblen Stiftbelegung, was sich teilweise auch in unterschiedlichen Signalfunktionen und Signalbezeichnungen niederschlägt.

Tabelle 7-1. Centronics-Schnittstelle. Signale mit ihren Stiftbelegungen für den druckerseitig verwendeten 36-poligen Steckverbinder in zwei Varianten (Centronics bzw. PC-kompatibel) und für den rechnerseitig verwendeten 25-poligen Steckverbinder (IBM-PC) nach [IEEE 1284 1994]. Die Stiftnummern nach den Schrägstrichen geben die verdrillten Rückführungen mit Signalerde an; H = Host (Rechner) und P = Printer (Drucker) bezeichnen den jeweiligen Sender eines Signals

Centronics	PC-kompatibel		Signal	Sender	Funktion
36-polig	36-polig	25-polig			
1 /19	1 /19	1 /18	STROBE	H	Byteübernahmesignal für Drucker
2 /20	2 /20	2 /18	DATA 1	-	Datenleitung 1 (LSB)
3 /21	3 /21	3 /19	DATA 2	-	Datenleitung 2
4 /22	4 /22	4 /19	DATA 3	-	Datenleitung 3
5 /23	5 /23	5 /20	DATA 4	-	Datenleitung 4
6 /24	6 /24	6 /20	DATA 5	-	Datenleitung 5
7 /26	7 /25	7 /21	DATA 6	-	Datenleitung 6
8 /25	8 /26	8 /21	DATA 7	-	Datenleitung 7
9 /27	9 /27	9 /22	DATA 8	-	Datenleitung 8 (MSB)
10 /28	10 /28	10 /22	ACK	P	Bytequittierung u. Byteanforderung
11 /29	11 /29	11 /24	BUSY	P	Drucker ist nicht empfangsbereit, z.B. Puffer ist voll
12	12	12	PAPER EMPTY	P	Papierende
13	13	13	SELECT	P	Drucker ist selektiert
14		14	SIGNAL GND	-	Signalerde (0 V)
			AUTO FEED XT	H	automatischer Zeilenvorschub bei Wagenrücklauf
15			OSCEXT	P	externer Oszillator
16	16		SIGNAL GND	-	Signalerde (0 V)
17	17		CHASSIS GND	-	Geräteerde
18	35		+5 V	P	(für Prüfzwecke)
31 /30			INPUT PRIME	H	Druckerinitialisierung (Reset)
	31 /30	16 /25	INIT	H	Druckerinitialisierung (Reset)
32	32	15	FAULT	P	Fehleranzeige des Druckers
	36 /33	17 /23	SELECT IN	H	Druckerselektierung
33			LIGHT DETECT	P	Anzeige: Video-Lampe „aus“
34			LINE COUNT	P	Beide Seiten eines Schalters, der
35			LINE CNT. RET.	P	beim Zeilenvorschub betätigt wird
36			reserved	P	

Signale. Tabelle 7-1 zeigt die Signale mit ihren Bezeichnungen, einer Kurzangabe ihrer Funktion und mit den Stiftbelegungen für die unterschiedlichen Steckverbinder. Die wichtigsten Signale sind die acht Datenleitungen DATA 1 bis DATA 8, drei Synchronisationsleitungen STROBE, ACK und BUSY für die Bytesynchronisation und für die Steuerung des Datenflusses, eine Steuerleitung INPUT PRIME bzw. INIT zur Druckerinitialisierung, ggf. eine Steuerleitung AUTO FEED XT für den automatischen Zeilenvorschub bei Aussenden des Wagenrücklaufzeichens CR (carriage return), ggf. eine Steuerleitung SELECT IN zur Selektierung des Druckers (diese Funktion wird meist durch den Select-Schalter am Drucker ersetzt), drei Statusleitungen PAPER EMPTY, SELECT und FAULT zur Anzeige des Druckerbetriebszustands, ggf. eine 5 V-Leitung, die dem Rechner anzeigt, daß der Drucker angeschaltet und angeschlossen ist, sowie mehrere Erdleitungen als Geräteerde und für Signalrückführungen. Bei verdrillter Rückführung ist deren Zuordnung zu den signalführenden Leitungen entsprechend Tabelle 7-1 zu beachten.

Synchronisation. Die Synchronisation einer Zeichenübertragung erfolgt durch *Handshaking* mittels der Signale STROBE, BUSY und ACK. Allerdings gibt es hierfür kein einheitliches Protokoll, da die Schnittstelle nicht festgeschrieben und somit kein echter Standard ist. Die Protokolle variieren in den Signalabhängigkeiten, d.h. im Wechselspiel der drei Signale, wie auch in den Zeitvorgaben für die Signale, mit denen die Centronics-Schnittstelle zusätzlich beaufschlagt ist. Dies hat den Nachteil, daß es immer wieder zu Inkompatibilitäten zwischen Rechnern und Druckern kommt (siehe dazu auch 7.3.4: Compatibility Mode). Gebräuchlich sind folgende Abhängigkeiten von STROBE und BUSY bzw. von BUSY und ACK:

- 1a. Busy-while-Strobe: BUSY folgt der fallenden Flanke von STROBE und wird inaktiviert, nachdem STROBE inaktiviert wurde.
- 1b. Busy-after-Strobe: BUSY folgt der steigenden Flanke von STROBE.
- 2a. Ack-in-Busy: ACK wird aktiviert und inaktiviert, während BUSY aktiv ist.
- 2b. Ack-after-Busy: ACK wird aktiviert, nachdem BUSY inaktiviert wurde.
- 2c. Ack-while-Busy: ACK wird aktiviert, während BUSY aktiv ist, und inaktiviert, nachdem BUSY inaktiviert wurde.

Von der Firma Centronics selbst wurde nur die Variante in der Kombination von 1b und 2b realisiert, wie sie in Bild 7-18a gezeigt ist (zum Teilbild b siehe 7.3.4: Compatibility Mode). Der Rechner gibt ein Byte auf den Datenleitungen DATA 1 bis DATA 8 aus und setzt nach einer vorgegebenen Verzögerungszeit sein Signal STROBE in den Aktivzustand, um anzugeben, daß die Datensignale gültig sind. Der Drucker benutzt dieses Signal zur Datenübernahme und zeigt, nachdem STROBE wieder inaktiviert wurde, durch Setzen seines BUSY-Signals an, daß er vorerst für eine weitere Datenübernahme nicht bereit ist. Er hält BUSY so lange gesetzt, bis er das Byte verarbeitet hat; dies schließt z.B. auch einen nachfolgen-

den Seitenvorschub mit ein. Danach nimmt der Drucker sein BUSY-Signal zurück und quittiert dann die Übertragung durch Aktivieren seines ACK-Signals. Dieses wird vom Prozessor als Anforderungssignal für die nächste Byteübertragung ausgewertet. (Das BUSY-Signal wird insbesondere auch zur Steuerung des Füllens eines Datenpufferspeichers im Drucker benutzt, so daß einerseits kein Pufferüberlauf auftritt und andererseits der Drucker immer rechtzeitig mit Daten versorgt wird.) – Zur Centronics-Schnittstelle siehe auch [IEEE 1284 1994].

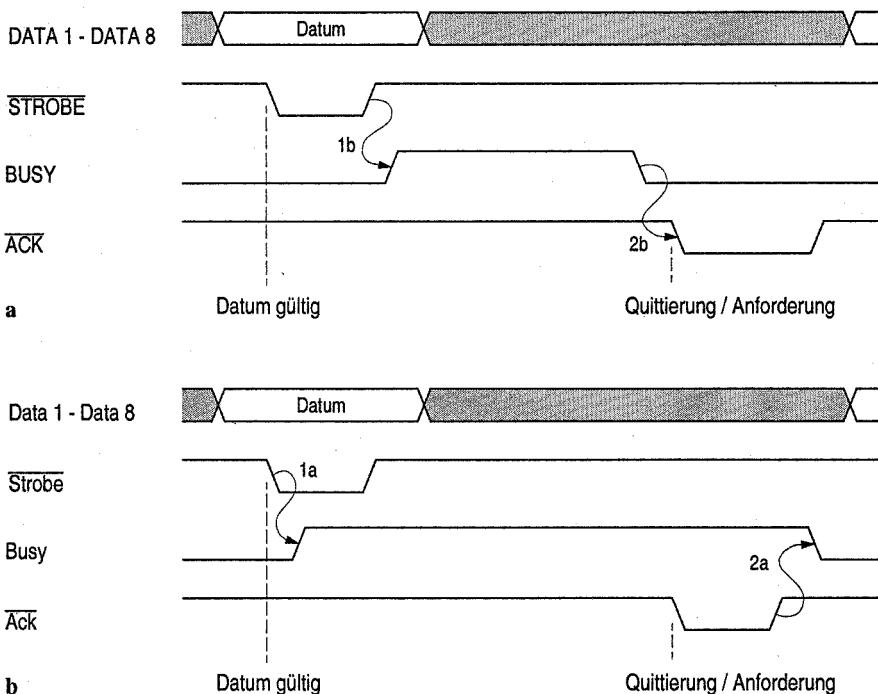


Bild 7-18. Bytesynchronisation bei 8-Bit-paralleler Datenausgabe. **a** Centronics-Schnittstelle mit den Signalabhängigkeiten Busy-after-Strobe (1b im Text) und Ack-after-Busy (2b im Text), **b** Compatibility-Mode nach IEEE 1284 mit den Signalabhängigkeiten Busy-while-Strobe (1a im Text) und Ack-in-Busy (2a im Text); in Anlehnung an [IEEE 1284 1994]

7.3.4 IEEE 1284: Compatibility-, Nibble- und Byte-Mode

Die Schnittstellennorm IEEE 1284 ist unter dem Einfluß mehrerer Firmengruppen entstanden, wodurch sich die Vielfalt ihrer Übertragungsarten (Communication Modes) erklärt. Sie beinhaltet unter der Bezeichnung Compatibility-Mode die „alte“ unidirektionale Centronics-Schnittstelle mit jedoch nun definiertem Handshake-Protokoll. Sie definiert darüber hinaus zwei weitere Übertragungsarten, den Nibble- und den Byte-Mode für die 2×4 -Bit- bzw. die 8-Bit-breite

Datenübertragung in der Gegenrichtung, also vom Peripheriegerät zum Rechner, die in Verbindung mit dem Compatibility-Mode für die bidirektionale Datenübertragung benutzt werden können. Hinzu kommen die beiden Übertragungsarten EPP-Mode (7.3.5) und ECP-Mode (7.3.6), die unabhängig vom Compatibility-Mode die bidirektionale Datenübertragung mit 8-Bit-Datenbreite ermöglichen. Diese Bidirektionalität der Schnittstelle erlaubt es, außer Drucken auch solche Peripheriegeräte zu betreiben, die nicht nur Daten vom Rechner empfangen, sondern auch Daten an den Rechner senden. – Alle fünf Übertragungsarten arbeiten mit Handshake-Synchronisation, d.h. asynchron.

Signale. Verwendet werden von der IEEE-1284-Schnittstelle die Leitungen der Centronics-Schnittstelle, diese wechseln jedoch je nach Übertragungsart ihre Funktion. Damit verbunden sind auch bezeichnungstechnische Abweichungen von den Centronics-Signalen. Tabelle 7-2 zeigt eine Auflistung der IEEE-1284-

Tabelle 7-2. IEEE 1284-Schnittstelle. Signale mit ihren Stiftbelegungen für den 25-poligen Steckverbinder IEEE 1284-A und die beiden 36-poligen Steckverbinder IEEE 1284-B und IEEE 1284-C nach [IEEE 1284 1994]. Die Stiftnummern nach den Schrägstrichen geben die verdrillten Rückführungen mit Signalerde an; H = Host (Rechner) und P = Peripheral (Gerät) bezeichnen den jeweiligen Sender eines Signals. Die mittig angeordneten Signalbezeichnungen gelten für jeweils vier bzw. fünf Übertragungsarten gemeinsam

25-pol.			36-polig			Sender		Übertragungsart		
A	B	C		Compat.	Nibble		Byte	ECP	EPP	
1/18	1/19	15/33	H	Strobe	HostClk		HostClk	HostClk	Write	
2/19	2/20	6/24	-			Data 1 (LSB)			AD1	
3/19	3/21	7/25	-			Data 2			AD2	
4/20	4/22	8/26	-			Data 3			AD3	
5/20	5/23	9/27	-			Data 4			AD4	
6/21	6/24	10/28	-			Data 5			AD5	
7/21	7/25	11/29	-			Data 6			AD6	
8/22	8/26	12/30	-			Data 7			AD7	
9/22	9/27	13/31	-			Data 2 (MSB)			AD8	
10/24	10/28	3/21	P	Ack	PtrClk	PtrClk	PeriphClk	Intr		
11/23	11/29	1/19	P	Busy	PtrBusy	PtrBusy	PeriphAck	Wait		
12/24	12/28	5/23	P	PError	AckDataReq	AckDataReq	AckReverse	User def.1		
13/24	13/28	2/20	P	Select	Xflag	Xflag	Xflag	User def.3		
14/25	14/30	17/35	H	AutoFd	HostBusy	HostBusy	HostAck	DStrb		
15/23	32/29	4/22	P	Fault	DataAvail	DataAvail	PeriphReq.	User def.2		
16/25	31/30	14/32	H	Init	Init	Init	ReverseReq.	Init		
17/25	36/30	16/34	H	SelectIn	IEEE 1284 Active	IEEE 1284 Active	IEEE 1284 Active	ASrb		
		15				Not defined				
		16				Logic Gnd				
		17				Chassis Gnd				
			18	H		Host Logic High				
			18	P		Peripheral Logic High				
		33-35				Not defined				

Signalleitungen, getrennt nach den Übertragungsarten. Sie zeigt außerdem die Stiftbelegungen für den rechnerseitig verwendeten 25-poligen Steckverbinder (IEEE 1284-A) und für die beiden druckerseitig verwendeten 36-poligen Steckverbinder (1284-B und 1284-C). Der Steckverbinder vom Typ C ist eine Miniatursausführung, er ist ggf. auch rechnerseitig vorzufinden. – Die Funktionalität dieser Signale wird nachfolgend exemplarisch anhand von Signalverläufen demonstriert.

Verhandlungsprotokoll. Vor einer Übertragung muß der Rechner (hier als Host H bezeichnet) sich zunächst einmal versichern, ob das angeschlossene Peripheriegerät (hier als Peripheral P bezeichnet) einerseits norm-entsprechend ist (IEEE-1284-compliant) und ob es andererseits die vom Rechner gewünschte Übertragungsart unterstützt. Norm-entsprechend heißt, daß das Gerät für den Compatibility-Mode sowie für den Nibble-Mode ausgelegt sein muß und daß es in der Lage sein muß, bzgl. der zu benutzenden Übertragungsart ein Verhandlungsprotokoll zu durchlaufen. Bild 7-19 zeigt dieses Protokoll, das grundsätzlich vom

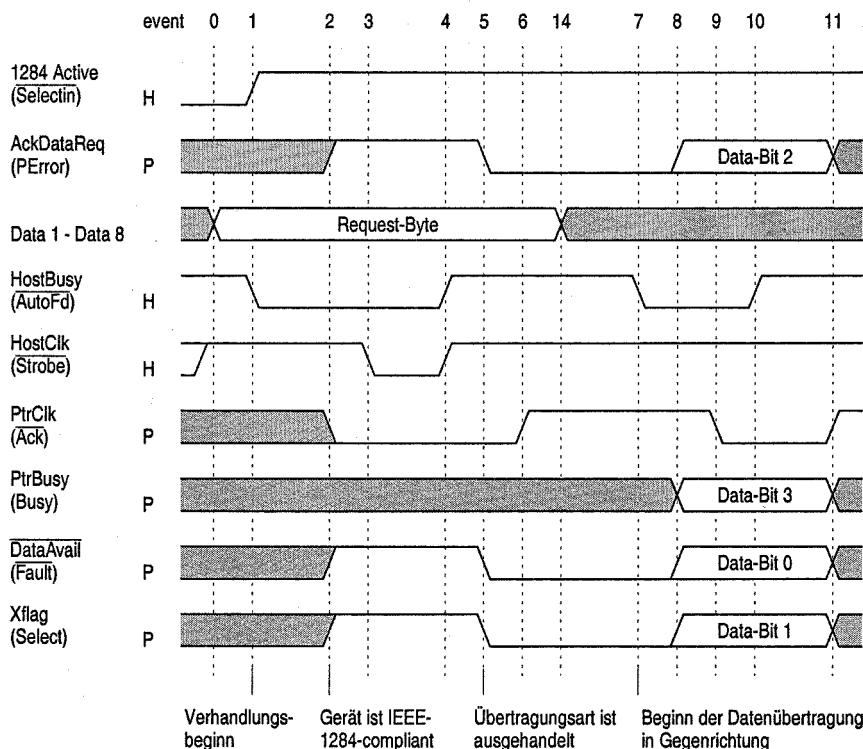


Bild 7-19. IEEE-1284-Verhandlungsprotokoll zwischen Rechner und peripherem Gerät. Abfrage auf Norm-Entsprechung des Geräts (event 2) und Aushandeln einer Übertragungsart, hier des Nibble-Mode (event 5). H = Host (Rechner), P = Peripheral (Gerät), Signalbezeichnungen für Nibble- und für Compatibility-Mode (in Klammern); in Anlehnung an [IEEE 1284 1994]

Compatibility-Mode als Default-Mode für die Vorwärtsrichtung ausgeht und das in dieser Übertragungsart zunächst einmal die Norm-Entsprechung des Geräts überprüft. An diese Überprüfung schließt sich dann das Aushandeln der vom Rechner gewünschten Übertragungsart an, in unserem Beispiel des Nibble-Mode. Die während der Verhandlung signifikanten Zeitpunkte für die Signalübergänge sind im Bild gemäß der IEEE-Norm als Ereignisse (events) durch Nummern gekennzeichnet, auf die sich die folgende Beschreibung bezieht.

Der Rechner belegt vor Verhandlungsbeginn die Datenleitungen mit einem Request-Byte mit der Codierung der gewünschten Übertragungsart, hier Nibble-Mode (event 0), und leitet dann die Verhandlung durch Übergang in die Signalzustände $1284\text{-Active} = 1$ und $\overline{\text{HostBusy}} = 0$ ein (event 1). Das Gerät gibt sich daraufhin mittels der Signalzustände $\overline{\text{PtrClk}} = 0$, $\overline{\text{DataAvail}} = 1$, $Xflag = 1$ und $\overline{\text{AckDataReq}} = 1$ als IEEE-1284-compliant zu erkennen (event 2). Der Rechner setzt dann seinerseits die Verhandlung fort, indem er mit $\overline{\text{HostClk}} = 0$ die Übernahme des Request-Bytes in ein Datenpufferregister des Geräts bewirkt (event 3) und indem er dem Gerät mit $\overline{\text{HostClk}} = 1$ und $\overline{\text{HostBusy}} = 1$ bestätigt, daß er dieses als IEEE-1284-compliant erkannt hat (event 4).

Das Gerät wiederum zeigt seine Bereitschaft für die im Request-Byte angegebene Übertragungsart Nibble-Mode durch $Xflag = 0$ an und signalisiert gleichzeitig mit $\overline{\text{AckDataReq}} = 0$ und $\overline{\text{DataAvail}} = 0$, daß es ein erstes Byte zur Übertragung an den Rechner bereithält (event 5), womit die Verhandlung erfolgreich abgeschlossen ist. Auf sie folgt eine kurze Übergangsphase (events 6 bis 14), nach der dann das Gerät (gemäß dem hier gewählten Beispiel) seine Byteeingabe durchführt. Diese ist im Bild durch die Übertragung des ersten Nibbles angedeutet (siehe dazu den übernächsten Absatz: Nibble-Mode).

Compatibility-Mode. Der *Compatibility-Mode* entspricht, wie bereits erwähnt, im wesentlichen der alten Centronics-Schnittstelle, dient also zur byte-parallelen Datenübertragung vom Rechner zum Peripheriegerät. Entgegen der *Centronics-Schnittstelle* ist die Handshake-Synchronisation mittels der Signale Strobe, Busy und $\overline{\text{Ack}}$ jedoch eindeutig festgelegt, und zwar mit den Signalabhängigkeiten Busy-while-Strobe und Ack-in-Busy (siehe 7.3.3, S. 483: 1a und 2a) und mit zusätzlicher Vorgabe von Zeitanforderungen an diese Signale. Ein Beispiel für die Signalverläufe einer Byteübertragung zeigt Bild 7-18b (S. 484). Zum Vergleich mit der Centronics-Schnittstelle sei auf Teilbild a verwiesen, das eine von mehreren Synchronisationsvarianten zeigt. Die Datenübertragung ist wie bei der Centronics-Schnittstelle rein software-gesteuert, dementsprechend liegen die maximalen Übertragungsraten ebenfalls bei 50 bis 150 kbyte/s.

Nibble-Mode. Der *Nibble-Mode* ergänzt den Compatibility-Mode hinsichtlich der Übertragung vom Peripheriegerät zum Rechner, ermöglicht also eine bidirektionale Datenübertragung. Sie erfolgt jedoch nicht über die Datenleitungen, sondern über vier der Statusleitungen (Bild 7-20). Dementsprechend sind für jede Byteübertragung zwei Übertragungsvorgänge erforderlich, bei denen jeweils

4-Bit-Einheiten (Nibbles) transportiert werden. Benutzt werden die Leitungen DataAvail (Data 0 bzw. 4), Xflag (Data 1 bzw. 5), AckDataReq (Data 2 bzw. 6) und PtrBusy (Data 3 bzw. 7). Trotz getrennter Leitungen für die Datenübertragung schließen sich der Compatibility-Mode und der Nibble-Mode aufgrund der unterschiedlichen Signalnutzungen gegenseitig aus. Das heißt, die Signalleitungen werden zu einer Zeit entweder für die eine oder für die andere Übertragungsart (Übertragungsrichtung) genutzt.

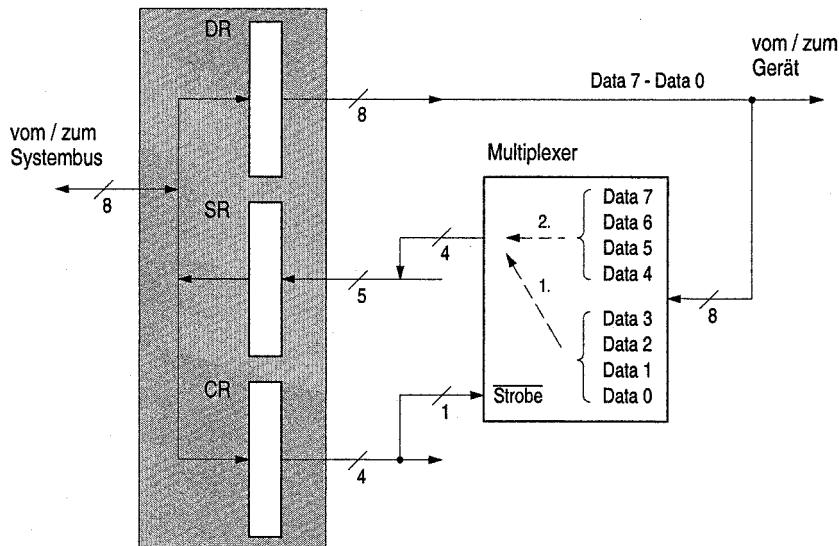


Bild 7-20. Parallel-Interface für die 8-Bit-Datenausgabe im Compatibility-Mode und die 2 × 4-Bit-Dateneingabe im Nibble-Mode (in der Reihenfolge 1., 2.), aufgebaut aus drei 8-Bit-Registern mit unidirektionalem Zugriff auf die acht Daten, fünf Steuer- und vier Statussignalen der Schnittstelle, ergänzt um einen 4-Bit-2-zu-1-Multiplexer. Umschalten des Multiplexers durch das Steuersignal Strobe

Bild 7-19 zeigt die Übertragung eines ersten Nibble unmittelbar nach Beendigung der Verhandlungsphase. Eingeleitet wird die Übertragung durch den Rechner mit HostBusy = 0 (event 7), nachdem er zuvor mittels DataAvail = 0 (event 5) den Sendewunsch des Peripheriegeräts mitgeteilt bekommen hat. Das Gerät legt danach die ersten vier Bits auf den Signalleitungen an (event 8) und gibt deren Gültigkeit mit PtrClk = 0 bekannt (event 9). Der Rechner übernimmt das Nibble und bestätigt die Übernahme mit HostBusy = 1 (event 10), woraufhin das Gerät die erste Übertragung mit PtrClk = 1 abschließt (event 11). Dieser Vorgang wird dann für das zweite Nibble wiederholt.

Byte-Mode. Der Byte-Mode ermöglicht wie der Nibble-Mode die Datenübertragung vom Peripheriegerät zum Rechner und ergänzt damit den Compatibility-Mode für die bidirektionale Übertragung. Anders als der Nibble-Mode benutzt er

jedoch für die Datenübertragung die Datenleitungen, also das Byteformat. Die Schnittstelle hat dazu anstelle des Multiplexers in Bild 7-20 einen bidirekionalen Datenpfad gemäß Bild 7-10a (S. 474). Auch hier können naturgemäß nicht beide Übertragungsarten gleichzeitig aktiviert sein, sondern der Rechner gibt jeweils vor, welche Übertragungsrichtung und damit Übertragungsart „gefahren“ wird. Vor einer Übertragung vom Peripheriegerät zum Rechner meldet dieses seinen Eingabewunsch beim Rechner an.

7.3.5 IEEE 1284: Enhanced-Parallel-Port-Mode (EPP)

Der *EPP-Mode* erlaubt die byteweise bidirektionale Übertragung zwischen Rechner und Gerät unter Benutzung der acht Datenleitungen für beide Übertragungsrichtungen, und zwar ohne daß wie beim Nibble- und beim Byte-Mode die Übertragungsart gewechselt werden muß. Die Übertragung erfolgt außerdem rein hardware-gesteuert, und zwar ähnlich wie bei einem Prozessorbus in der Ausführung als Multiplexbus, indem der Rechner (Host) sämtliche Übertragungen einleitet und dann zunächst ein Adressbyte und danach eines oder mehrere Datenbytes überträgt. Das Peripheriegerät kann seinerseits einen Übertragungswunsch anmelden, indem es das 0-aktive Intr-Signal kurzzeitig aktiviert und damit eine Unterbrechung (Interrupt) auslöst. Eine gerade laufende Byteübertragung wird dadurch nicht beeinträchtigt.

Für die Übertragung stehen dem Host die folgenden vier Operationen zur Verfügung:

- Address-Write,
- Data-Write,
- Address-Read und
- Data-Read.

Mit dem Übertragen einer Adresse an das Gerät (Address-Write) wählt der Rechner ein Gerätregister an, welches dann als Ziel bzw. Quelle der nachfolgenden Datenübertragungen benutzt wird (Data-Write, Data-Read). Die Adresse wird dazu im Gerät in einem Adressregister gehalten. Dieses Register kann vom Rechner auch gelesen werden (Address-Read).

Bild 7-21 zeigt die Aufeinanderfolge einer Adressübertragung an das Gerät und einer Datenübertragung an den Rechner. Die Synchronisation erfolgt durch Handshaking mittels der vom Rechner erzeugten Strobe-Signale \overline{ASrb} und \overline{DStrb} und des vom Gerät erzeugten Wait-Signals. Mit letzterem bestätigt das Gerät die Adressübernahme und damit die erneute Übertragungsbereitschaft (negative Flanke) bzw. die Datenbereitstellung (positive Flanke) und die erneute Übertragungsbereitschaft (negative Flanke). Die Übertragungsrichtung wird vom Rechner mit dem Write-Signal vorgegeben. – Die im EPP-Mode erreichbaren Übertra-

gungsgraten sind aufgrund der hardware-gesteuerten Übertragung sehr viel höher als beim Compatibility-Mode und liegen bei 500 kbyte/s bis 2 Mbyte/s.

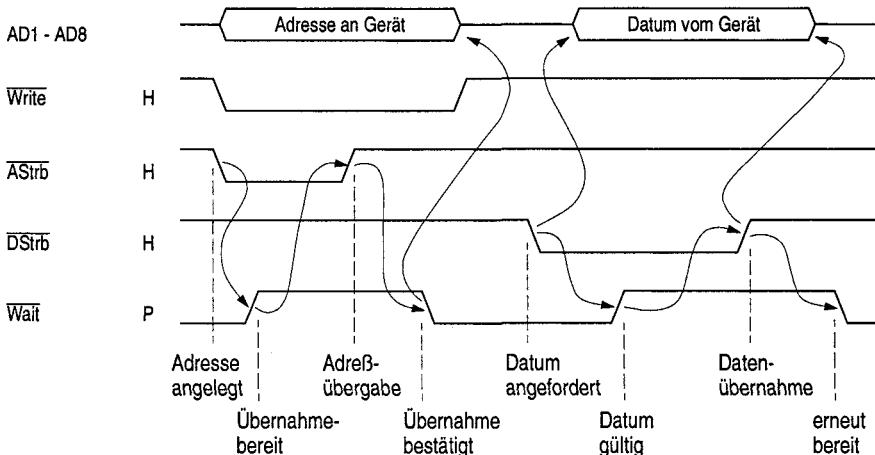


Bild 7-21. EPP-Mode nach IEEE 1284. Übertragung eines Adreßbytes vom Rechner zum Gerät und eines Datenbytes vom Gerät zum Rechner. Synchronisation der Übertragungen durch Handshaking mittels der Signale ASrb und Wait bzw. DStrb und Wait. H = Host (Rechner), P = Peripheral (Gerät); in Anlehnung an [IEEE 1284 1994]

7.3.6 IEEE 1284: Extended-Capability-Port-Mode (ECP)

Der *ECP-Mode* erlaubt wie der EPP-Mode die byteweise bidirektionale Datenübertragung zwischen Rechner und Peripheriegerät unter Benutzung der acht Datenleitungen für beide Übertragungsrichtungen. Die Übertragung erfolgt ebenfalls rein hardware-gesteuert, allerdings unterliegt nur die Steuerung der Ausgabe dem Rechner (host). Die Steuerung der Eingabe liegt beim Peripheriegerät, das entsprechend ausgelegt sein muß. Synchronisiert wird mit je einem Handshake-Signalpaar pro Übertragungsrichtung mit gleichartigen Protokollen.

Bild 7-22 demonstriert dies für eine Byteübertragung vom Rechner zum Gerät, gefolgt von einer Byteübertragung vom Gerät zum Rechner in durch die Pfeile selbsterklärender Weise. Die Umschaltung zwischen den beiden Übertragungen, also von Ausgabe auf Eingabe, erfolgt mit geringem Aufwand, indem der Rechner zunächst den Datenbus hochohmig schaltet, dabei HostAck = 0 setzt und nach einer kurzen Wartezeit (setup time) ReverseRequest = 0 setzt. Das Gerät quittiert die Umschaltung mit AckReverse = 0 und ist danach berechtigt, den Bus mit einem Byte zu belegen. Der erneute Richtungswechsel (in Bild 7-22 nicht gezeigt) erfolgt in ähnlicher Weise und wird ebenfalls vom Rechner eingeleitet.

Als Besonderheit des ECP-Mode kann anstatt eines Datenbytes ein Kommando-byte übertragen werden. In Vorförwärtsrichtung werden die beiden Informationsarten durch das für die Synchronisation in dieser Richtung nicht benötigte Signal HostAck unterschieden (HostAck = 1: Datum, HostAck = 0: Kommando), in Rückwärtsrichtung durch das Signal PeriphAck (Bild 7-22). Ein solches Kommando beinhaltet entweder eine 7-Bit-Kanaladresse oder eine 7-Bit-Längenangabe als *Lauflängencodierung* (run length encoding, RLE). Letztere besagt, wie

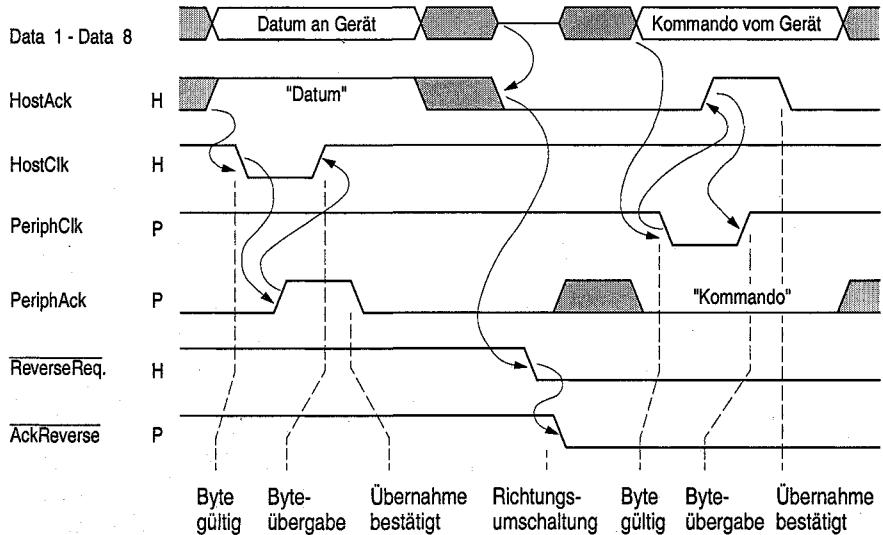


Bild 7-22. ECP-Mode nach IEEE 1284. Übertragung eines Datenbytes vom Rechner zum Gerät und eines Kommandobytes vom Gerät zum Rechner. Synchronisation der Übertragungen durch Handshaking mittels der Signale HostClk und PeriphAck bzw. PeriphClk und HostAck; Umschaltung von Vorförwärts- in Gegenrichtung mittels der Signale ReverseRequest und AckReverse. H = Host (Rechner), P = Peripheral (Gerät); in Anlehnung an [IEEE 1284 1994]

oft das nachfolgende Datum auf der Empfängerseite vervielfacht werden soll und erlaubt damit die komprimierte Übertragung aufeinanderfolgender Bytes gleichen Bitmusters, z.B. bei der Übertragung von Grafiken. – Die Übertragungsart ECP mit RLE wird während der Verhandlungsphase durch eine bestimmte Codierung des Request-Bytes ausgehandelt.

Das Peripheriegerät kann dem Host einen Übertragungswunsch mittels des Signals PeriphRequest melden, der von diesem als Interrupt ausgewertet werden kann. Unabhängig davon verbleibt dem Host jedoch die Steuerung über die Richtungsvorgabe. Die Schnittstelle ist im ECP-Mode außerdem *DMA-fähig*, d.h. Datentransfers können ohne Softwareunterstützung durchgeführt werden. Ferner sieht sie zur Erreichung hoher Übertragungsraten anstelle des Datenregisters einen FIFO-Puffer vor, für den eine Kapazität von 16 Bytes empfohlen wird. Die maximalen erreichbaren Übertragungsraten liegen bei 1 bis 2 Mbyte/s.

7.4 Serielle Schnittstellen

Wie in 7.2.1 beschrieben, gibt es für den parallelen und seriellen Anschluß peripherer Geräte (allgemeiner: peripherer Einheiten) an ein Rechnersystem (genauer: an dessen Systembus) zwei typischen Verbindungsarten, die *Punkt-zu-Punkt-Verbindung* und die *Mehrpunktverbindung*. Bei der seriellen Datenübertragung spiegeln sich diese Möglichkeiten in den herkömmlichen seriellen Schnittstellen wider, wie sie aus der Datenfernübertragung kommen, bzw. in den modernen peripheren Bussen, die diese Schnittstellen immer mehr verdrängen. Die seriellen Schnittstellen sind im wesentlichen durch ihre Signale definiert und dementsprechend passiver Natur. Die seriellen Busse hingegen sind aktiver Natur, indem bei ihnen zusätzlich zu den Bussignalen auch die Ablaufsteuerung definiert ist. – In diesem Abschnitt beschränken wir uns auf die (passiven) Schnittstellen; auf die (aktiven) Peripheriebusse gehen wir in Abschnitt 8.2 ein.

Beim Einsatz serieller Schnittstellen für die Ein-/Ausgabe von Daten werden zwei grundsätzliche Strukturen unterschieden:

- Bei der *Überbrückung kurzer Entfernungen* bis zu mehreren Metern wird der Systembus mit seinen Signalen über einen Interface-Baustein, unterstützt durch Signaltreiberbausteine, mittels eines Kabels direkt mit dem Ein-/Ausgabegerät verbunden, das seinerseits zur Anpassung ein gleichartiges Interface und Signaltreiberbausteine aufweist (Bild 7-23a). – Der Vorteil eines solchen seriellen Geräteanschlusses gegenüber einem parallelen liegt in dem geringeren technischen Aufwand, bedingt durch die geringere Anzahl an Signal- und Erdleitungen. Hinzu kommt, daß sich mit der seriellen Technik sehr viel größere Entfernungen überbrücken lassen als mit der parallelen, da das Problem der Leitungskapazitäten und des Übersprechens zwischen den Signalleitungen hier sehr viel geringer ist. Nachteilig ist u.U. die geringere Übertragungsrate; es werden ggf. aber auch höhere Übertragungsraten als mit parallelen Schnittstellen erreicht (siehe Tabelle 8-1, S. 562).
- Bei der *Überbrückung großer Entfernungen*, z.B. bei der *Datenfernübertragung*, werden Übertragungssysteme der Fernmeldeeinrichtungen eines Landes benutzt. Hier müssen die Signale der beiden Interfaces bzw. ihrer Signaltreiberbausteine an die Übertragungssysteme angepaßt werden. Bild 7-23b zeigt dies am Beispiel des öffentlichen Fernsprechnetzes, und zwar für die herkömmliche analoge Übertragung. Die Anpassung erfolgt hier durch sog. *Modems* (Modulator/Demodulator), die auf der Senderseite die Gleichspannungspiegel der logischen Größen 0 und 1 (digitales Signal) in Wechselspannungssignale (analoges Signal), z.B. mit zwei unterschiedlichen Frequenzen aus dem Fernsprechfrequenzband, umsetzen (Modulation) und dies auf der Empfängerseite wieder rückgängig machen (Demodulation). Bei digitalen Übertragungssystemen, z.B. ISDN, sind entsprechend andere Anpaßeinrichtungen erforderlich. – Die Datenfernübertragung wird aus Aufwandsgrün-

den nur über serielle Verbindungen durchgeführt; parallele Alternativen gibt es dementsprechend nicht.

Man bezeichnet die Einrichtungen zur Signalumsetzung, wie Modems, auch als *Datenübertragungseinrichtungen DÜE* (data circuit-termination equipment, *DCE*) und die als Sender und Empfänger wirkenden Mikroprozessorsysteme und Ein-/Ausgabegeräte, zusammen mit ihren Interfaces und Signaltreiberbausteinen, als *Datenendeinrichtungen DEE* (data terminal equipment, *DTE*). Um Datenendeinrichtungen möglichst herstellerunabhängig miteinander verbinden zu können, gibt es internationale Vereinbarungen über die elektrischen, physikalischen und funktionellen Eigenschaften von DEE/DÜE-Schnittstellen, die sich als Normen etabliert haben. Es sind dies die sog. V-Empfehlungen (für Telefonnetze) und X-Empfehlungen (für Datennetze) des internationalen Dachverbands der Fernmeldeinstitutionen, früher CCITT, heute ITU. Sie werden mit anderen Bezeichnungen und ggf. mit inhaltlichen Modifikationen auch als nationale Normen geführt, z.B. im DIN und in der EIA (zu den Normungsgremien siehe 7.2.2).

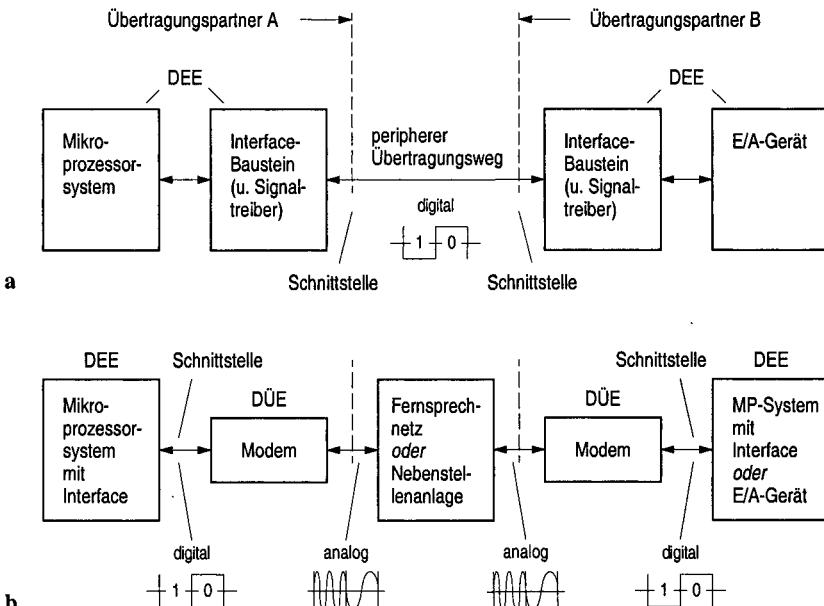


Bild 7-23. Peripherer Datenweg bei serieller Datenübertragung. **a** Direkte Verbindung, **b** Datenfernübertragung

Im folgenden betrachten wir jene Schnittstellenvereinbarungen der Datenfernübertragung, die als Standards für den direkten Anschluß von Ein-/Ausgabegeräten an den Rechner benutzt werden. Wir legen dabei den Schwerpunkt auf deren technische Gegebenheiten, so wie sie in der Schnittstellendefinition nach DIN 44 302 genannt sind (siehe 7.2.2). Die zeitlichen Zusammenhänge der

Schnittstellensignale, wie sie zur Beschreibung der asynchron- und der synchron seriellen Datenübertragungen erforderlich sind, werden dann in den beiden folgenden Abschnitten 7.5 und 7.6 dargestellt. – Betrachtet werden im einzelnen die CCITT/ITU-T-Empfehlung V.24/V.28 (ähnlich EIA RS-232-C) sowie die gegenüber V.28 leistungsfähigeren neueren Empfehlungen V.10 (ähnlich RS-223-A), V.11 (ähnlich RS-222-A) und RS-485. Ergänzende und übergreifende Angaben zu diesen Schnittstellen finden sich z.B. in [Dembowski 1997, Preuß, Musa 1993], außerdem sei auf die Normen verwiesen.

7.4.1 V.24/V.28 (RS-232-C)

Die über lange Zeit gebräuchlichste serielle Schnittstelle für den direkten Anschluß peripherer Geräte an ein Rechnersystem gemäß Bild 7-23a basiert auf den ITU-T-Empfehlungen V.24 und V.28. Sie wurden als *Punkt-zu-Punkt-Verbindung* für die bitserielle *asynchrone* und *synchrone Datenübertragung* im *analogen Fernsprechnetz* definiert. Die Empfehlung V.24 beschreibt dazu mehr als 40 Schnittstellenleitungen mit ihren Bedeutungen für die Übermittlung digitaler Daten-, Melde-, Steuer-, Takt- und Wählsignale sowie analoger Sprachsignale zwischen Rechner (DEE) und Modem (DÜE), gemäß einer Struktur nach Bild 7-23b, V.28 legt deren elektrotechnische Eigenschaften fest. Für den *direkten Geräteanschluß* wird nur ein Teil der Signale genutzt, und die dabei verwendeten Steuerleitungen erhalten eine entsprechend modifizierte Funktionalität. Unabhängig von dieser reduzierten Form spricht man von einer *V24-Schnittstelle* oder – entsprechend dem ihr ähnlichen EIA-Standard – von einer *RS-232-C-Schnittstelle*. Wegen des Punkt-zu-Punkt-Charakters der Schnittstelle (es können nur zwei Übertragungspartner miteinander verbunden werden) muß für jedes anzuschließende Gerät ein eigener Interface-Baustein vorhanden sein. – Heute wird diese Schnittstelle als Geräteschnittstelle durch serielle Peripheriebusse wie Universal Serial Bus (USB) und FireWire verdrängt (8.2).

Bild 7-24 zeigt eine V.24/RS-232-C-Verbindung in ihrer ursprünglichen Funktion, nämlich als Verbindung zwischen Rechner und Modem. Allerdings sind hier nur jene Signale dargestellt, die ggf. auch für den direkten Anschluß von Ein-/Ausbabegeräten verwendet werden. Das Senden von Daten vom Rechner (DEE) zum Modem (DÜE), d.h. in das Fernsprechnetz hinein, erfolgt hierbei über die Signalleitung TxD, und das Empfangen von Daten aus dem Fernsprechnetz über die Signalleitung RxD. Den Sendewunsch signalisiert der Rechner dem Modem mittels des RTS-Signals, womit er das Modem auffordert, seinen Sendeteil einzuschalten. Das Modem seinerseits bestätigt seine Sendebereitschaft mittels CTS. Beim Empfangen von Daten aus dem Fernsprechnetz meldet das Modem mittels DCD, ob der Empfangssignalpegel innerhalb des vorgegebenen Toleranzbereichs liegt, d.h., ob das Datensignal RxD gültig ist. DSR zeigt die grundsätzliche Betriebsbereitschaft des Modems an, DTR die des Rechners. Zwei weitere Ver-

bindungen sind die Betriebserde SGND als Bezugspotential für die Signalpegel sowie die Schutzerde PGND.

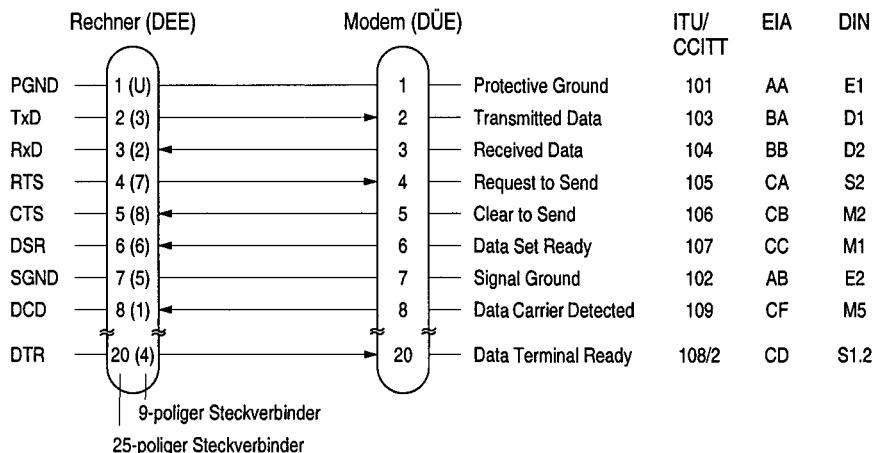


Bild 7-24. Schnittstellensignale einer V.24/RS-232-C-Verbindung zwischen Rechner (DEE) und Modem (DÜE), reduziert auf jene Signale, die auch für direkte Rechner-Geräte-Verbindungen verwendet werden. Dazu Stiftbelegungen für den 25-poligen und den 9-poligen Steckverbinder (U Ummantelung) und Signalkurzbezeichnungen verschiedener Normen

Bei *direktem Anschluß* eines Ein-/Ausgabegeräts an einen Rechner, sozusagen unter Weglassung der beiden Modems, treffen die Anschlüsse zweier DEEs unmittelbar aufeinander. Die Signalleitungen werden dementsprechend nicht mehr 1-zu-1 wie bei einer Rechner-Modem-Verbindung, sondern über Kreuz geführt (Bild 7-25a). Man spricht dann auch von einer *Nullmodem-Verbindung*. Die beiden Datenleitungen TxD und RxD erlauben auch hier wieder die Datenübertragung in beiden Richtungen. Sie reichen im Minimalfall für eine Datenverbindung aus, wie in 7.5.2 (S. 502) für die asynchron-serielle Datenübertragung beschrieben. Zusätzlich können die beiden Verbindungen RTS (Sender) nach CTS (Empfänger) zur Erhöhung der *Übertragungssicherheit* benutzt werden. Mit ihnen lässt sich nämlich die Übertragungsbereitschaft der beiden Übertragungspartner vor jeder Byteübertragung durch Handshake-Synchronisation sicherstellen. Auch die beiden Verbindungen DTR nach DSR und DCD können zur Sicherung der Übertragung verwendet werden, indem sie den beiden Übertragungspartnern signalisieren, ob überhaupt eine Kabelverbindung existiert bzw. ob der andere Partner überhaupt betriebsbereit (angeschaltet) ist (zu diesen und weiteren Sicherheitsmaßnahmen siehe 7.5.2).

Üblich ist die serielle Datenübertragung zwischen Rechner und Ein-/Ausgabegerät unter Verwendung eines 4-adrigen Telefonkabels, wodurch die Anzahl der verfügbaren Signalleitungen zwar stark eingeschränkt wird, die Kabelkosten jedoch gering gehalten werden. Bild 7-25b zeigt eine solche Verbindung, die mit nur drei Leitungen auskommt, nämlich den beiden Datenübertragungsleitungen TxD,

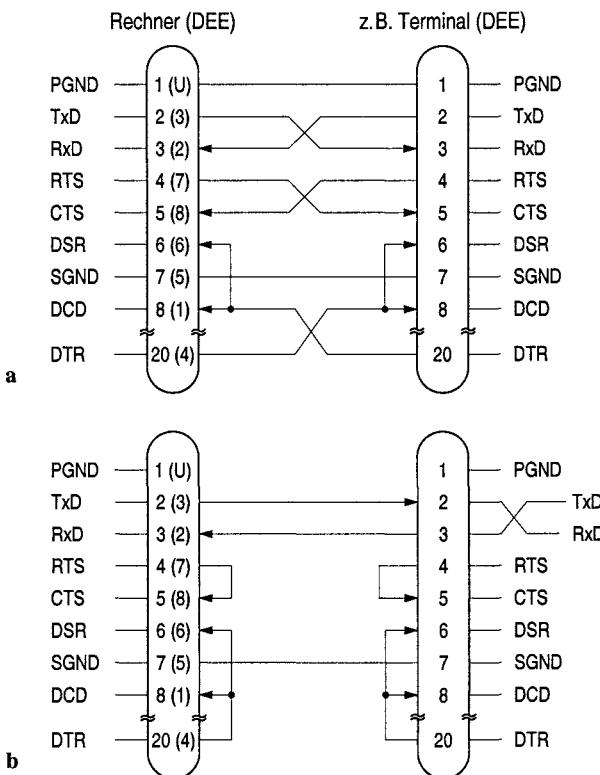


Bild 7-25. V.24/RS-232-C-Verbindung für den direkten Anschluß eines Ein-/Ausgabegeräts (DEE) an einen Rechner (DEE), a unter Nutzung sämtlicher in Bild 7-24 aufgeführten Signalleitungen (Nullmodem), b mit reduzierter Leitungsanzahl (als eine von mehreren Varianten)

RxD und der Betriebserde SGND. Die Sendebereitschaft und die Empfangsbereitschaft (Signale RTS und CTS) sowie die Anzeige einer vorhandenen Kabelverbindung (Signale DTR, DSR und DCD) werden hier von den beiden Übertragungspartnern in Form lokaler Rückkopplungen gegenüber der Software durch jeweils selbst erzeugte, ständig vorhandene Aktivpegel vorgetäuscht. Bild 7-25b zeigt gegenüber Bild 7-25a noch eine Variante der Datenverbindung, bei der die Kreuzung der Signale im Kabel entfällt, da sie im Ein-/Ausgabegerät selbst realisiert ist. Insgesamt ist zu beachten, daß es eine Vielzahl von Verbindungsvarianten gibt, so daß man sich bei Herstellung eines eigenen Verbindungskabels zunächst mit den Vorgaben des jeweiligen Geräteherstellers vertraut machen muß. – Als Stecker werden 37-, 25- und 9-polige Steckverbinder verwendet. Die Anschlußbelegungen für die bei uns gebräuchlichen 25- und 9-poligen Steckverbinder zeigt Bild 7-24.

Die elektrischen Kennwerte der Schnittstelle V.24 sind, wie bereits erwähnt, in der Empfehlung V.28 beschrieben. Darin sind die Signalpegel für die logischen

Größen 0 und 1 durch Spannungswerte zwischen +3 V und +15 V bzw. -3 V und -15 V, d.h. bipolar festgelegt; übliche Werte sind +12 V und -12 V. Die Daten-signale werden dabei in negativer Logik dargestellt (1: -12 V, 0: +12 V), die Steuer-signale in positiver Logik (aktiv: +12 V, inaktiv: -12 V). Verwendet wird eine *asymmetrische Signalübertragung*, d.h., es gibt für jedes Signal jeweils eine signalführende Leitung und als Bezugspotential die Betriebserde, das SGND-Signal (massebezogenes Signal). Da die Interface-Bausteine, mit denen die Schnittstellen realisiert werden, mit TTL-Pegeln (0 V und +5 V) arbeiten, muß eine Pegelumsetzung zwischen TTL- und V.24-Pegeln durchgeführt werden. Hierfür werden *Pegelumsetzerbausteine* (*Signaltreiber- und Signalempfängerbausteine*) eingesetzt (Bild 7-26a). Sie beeinflussen maßgeblich die maximal mögliche Übertragungsrate, die bei der V.24- bzw. RS-232-C-Schnittstelle bei 20 kbit/s liegt (genauer: 19,2 kbit/s); gebräuchlich Werte sind außerdem 110, 300, 1200, 2400, 4800 und 9600 bit/s. Bei maximaler Übertragungsrate liegt die überbrückbare Entfernung bei bis zu 15 m, bei kapazitätsarmen Leitungen bei bis zu 50 m (Tabelle 7-3). Häufig verwendet wird das 4-adige Telefonkabel.

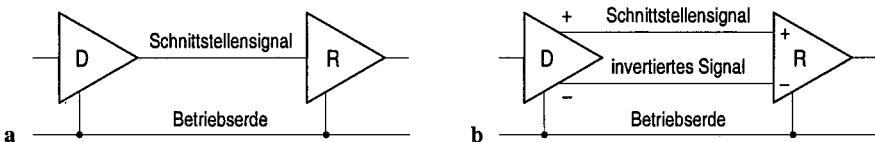


Bild 7-26. Signalübertragung einer Schnittstelle durch Signaltreiber- und Signalempfängerbausteine (D driver, R receiver), **a** asymmetrisch (massebezogen), **b** symmetrisch (differentiell). Die offengelassenen Leitungsenden sind mit den Interface-Bausteinen der beiden Übertragungspartner verbunden und führen TTL-Pegel

7.4.2 V.10 (RS-423-A) und V.11 (RS-422-A)

Eine Erhöhung der Übertragungsrate für die V.24/V.28- bzw. RS-232-C-Schnittstelle erreicht man durch eine Verbesserung der elektrotechnischen Eigenschaften der Schnittstelle, wie sie in unterschiedlichen Stufen in den beiden Normen V.10 (ähnlich RS-423-A) und V.11 (ähnlich RS-422-A) beschrieben ist. In beiden Fällen benötigt man entsprechend veränderte Signaltreiberbausteine und die dazu passenden Verbindungskabel. Beide Standards sehen außerdem eine Funktionserweiterung vor, indem sie die ursprüngliche Punkt-zu-Punkt-Verbindung zu einer *Mehrpunktverbindung* erweitern: Ein Sender kann jetzt mit bis zu zehn Empfängern verbunden sein. Dies ist nützlich, wenn mehrere Geräte mit denselben Daten versorgt werden müssen (z.B. Bildschirme für die Flugplananzeige in einem Flughafen).

Die Schnittstelle V.10 bzw. RS-423-A arbeitet wie V.24/V.28 bzw. RS-232-C mit *asymmetrischer Signalübertragung* (*single-ended signaling*, Bild 7-26a). Die maximal erreichbare Übertragungsrate beträgt bei kurzer Weglänge 100 kbit/s

und mehr, Weglängen in der Größenordnung von 1000 m und darüber sind bei geringeren Übertragungsraten erreichbar (Tabelle 7-3). Die Schnittstelle V.11 bzw. RS-422-A arbeitet hingegen mit *symmetrischer Signalübertragung*. Das heißt, die Daten- und Steuersignale werden vom Sender auf Leitungspaaren gleichzeitig in nichtinvertierter und in invertierter Form übertragen, und der Empfänger wertet die Pegeldifferenz zwischen den Leitungen eines Leitungspaares aus (*differential signaling*, Bild 7-26b). Dadurch werden Störeinflüsse aus dem Umfeld, wenn sie auf beide Leitungen gleichartig wirken, auf der Empfängerseite aufgehoben, wodurch eine sehr viel höhere Übertragungsrate als bei asymmetrischer Signalübertragung erreicht wird. Sie beträgt bei geringen Entfermungen bis zu 10 Mbit/s. Die maximal überbrückbare Entfernung liegt (bei geringeren Übertragungsraten) bei 1000 m und mehr (Tabelle 7-3). Zur Unterscheidung der beiden Leitungen eines Leitungspaares sind die Signalbezeichnungen um „+“ bzw. „-“ erweitert, z.B. TxD+ und TxD-. Die Signaltreiber- und Signalempfängerbausteine sind ggf. für RS-423-A und RS-422-A umschaltbar. Bei beiden Schnittstellen sind die Anschlußbelegungen der verwendeten 25- und 9-poligen Steckverbinder (anders als bei V.28 und RS-232-C) nicht genormt, sondern herstellerabhängig.

Tabelle 7-3. Einige technische Daten der seriellen Schnittstellen V.24/V.28 (RS-232-C), V.10 (RS-423-A), V.11 (RS-422-A) und RS-485. Der maximale Übertragungsweg ist stark von den technischen Gegebenheiten der Verbindung abhängig

Merkmale	V.24/V.28 RS-232-C	V.10 RS-423-A	V.11 RS-422-A	RS-485
max. Übertragungsrate	20 kbit/s	100 kbit/s	10 Mbit/s	10 Mbit/s
max. Übertragungsweg	15 m	\geq 1000 m	\geq 1000 m	\geq 1000 m
Signalübertragung	asymmetrisch	asymmetrisch	symmetrisch	symmetrisch
max. Senderanzahl	1	1	1	32
max. Empfängeranzahl	1	10	10	32
Signalpegel max.	± 15 V	± 7 V	± 7 V	-7 V, +12 V
Signalpegel min.	± 3 V	$\pm 0,3$ V	$\pm 0,3$ V	$\pm 0,3$ V

7.4.3 RS-485

Die Schnittstelle RS-485 hat dieselben elektrotechnischen Eigenschaften wie die Schnittstelle RS-422-A (Tabelle 7-3), d.h., sie arbeitet wie diese mit symmetrischer Signalübertragung. Im Unterschied zu dieser hat sie jedoch einen erweiterten Funktionsumfang, indem jeder der Teilnehmer *Sender* und *Empfänger* sein kann und indem sie als *Mehrpunktverbindung* für bis zu 32 Teilnehmer ausgelegt ist. Schaltungstechnisch sind die Übertragungspartner durch ein einziges gemeinsames Kabel miteinander verbunden. Angewandt wird dabei entweder die sog. Zweidrahttechnik für den Halbduplexbetrieb oder die Vierdrahttechnik für den Duplexbetrieb. Im ersten Fall besteht die Datenverbindung aus nur einem diffe-

rentiellen Signalpaar, das durch Multiplexer-/Demultiplexerschaltungen je nach Übertragungsrichtung mit dem jeweiligen Signaltreiber- bzw. Signalempfängerbaustein verbunden wird; im zweiten Fall gibt es zwei Signalpaare mit dedizierter Zuordnung der Signaltreiber- und Signalempfängerbausteine entweder zum einen oder zum anderen Leitungspaar. Die Anschlußbelegungen der verwendeten 25- und 9-poligen Steckverbinder sind wie bei RS-4-22-A nicht genormt.

Dadurch, daß es jetzt mehr als einen Sender innerhalb einer Verbindung geben kann, muß ein Verfahren zur *Kollisionsvermeidung* vorgesehen werden. Als solches wird entweder das vom Ethernet her bekannte CSMA/CD-Verfahren angewandt (siehe 7.7.1, das mit einem gewissen Hardwareaufwand verbunden ist. Oder es wird das weniger aufwendige, aber auch weniger effiziente *Master-Slave-Polling* angewandt, bei dem ein ausgewiesener Sender (Master) die übrigen Sender (Slaves) auf ihre Sendewünsche hin abfragt.

Die RS-485-Schnittstelle bildet die Grundlage für viele sog. *Feldbusse*, und zwar als unterste Schicht, d.h. Schicht 1 des *OSI-Referenzmodells* (7.7.1). Feldbusse dienen zum Anschluß von Sensoren, Aktoren und übergeordneten Einheiten, insbesondere in der Prozeß- und Automatisierungstechnik. Sie sind zumeist als serielle Busse ausgelegt und haben häufig den Charakter lokaler Netze. Gegenüber diesen zeichnen sie sich jedoch durch geringere Komplexität der Protokolle und ihre „Echtzeitfähigkeit“ aus. Sie haben außerdem eine hohe Zuverlässigkeit, hohe Fehlertoleranz und geringe Störanfälligkeit.

Anmerkung. Die genannten seriellen Schnittstellen haben ihre Entsprechungen auch in den DIN-Normen: V.24 als DIN 66022, V.28, V.10 (RS-423-A), V.11 (RS-422-A) und RS-485 insgesamt als DIN 66259, ausgewiesen als Teile 1 bis 4 dieser Norm.

7.4.4 X.21

Als Entsprechung zur Schnittstelle V.24 für analoge Netze gibt es die CCITT/ITU-T-Empfehlung X.21 für die synchron-serielle Datenübertragung in *digitalen Netzen*. Sie definiert eine 8-adrige Verbindung zwischen einem Rechner als Datenendeinrichtung DEE (DTE, auch als Host bezeichnet) und der für den Netz-zugang zuständigen Datenübertragungseinrichtung DÜE (DCE, auch als Interface-MESSAGE-Processor, IMP, bezeichnet) und erlaubt Übertragungsraten von 1200 bit/s bis 64 kbit/s. Die in Bild 7-27 gezeigten Signalleitungen C und I steuern den Auf- und Abbau einer Verbindung im Netz. T und R dienen zum Senden und Empfangen von Daten sowie zur Übertragung von Ruf- und Meldesignalen beim Verbindungsauflbau. Die Leitung S gibt den Takt für die Bitsynchronisation vor. Die Leitung B liefert wahlweise nach jeweils acht Bits einen Impuls zur Byteerkennung. Hinzu kommen die Signalerde Ga und die Schutzerde G. Die Verwendung der Signalleitungen für den Verbindungsauflbau, die Datenübertragung und den Verbindungsabbau ist z.B. in [Tanenbaum 2003] beschrieben. – Für

den direkten Anschluß von Ein-/Ausgabegeräten an einen Rechner hat diese Schnittstelle keine Bedeutung.

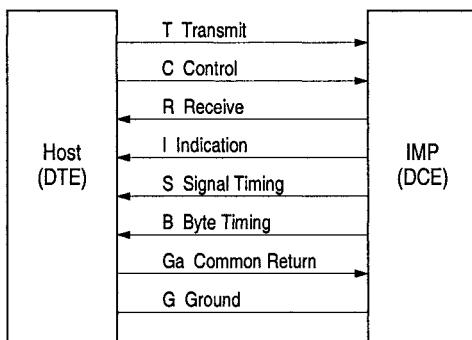


Bild 7-27. ITU-T-Empfehlung X.21.
Bezeichnungen der Schnittstellensignale

7.5 Asynchron-serielle Datenübertragung

Bei der *seriellen Datenübertragung (seriellen Ein-/Ausgabe)* werden die Bits eines Zeichens nacheinander auf ein und derselben Datenleitung übertragen. Sie werden in einem festen *Schrittakt* gesendet, auf den sich der Empfänger bei der Datenübernahme einstellt. Das gilt sowohl für die asynchron-serielle als auch für die synchron-serielle Datenübertragung. Bei der *asynchron-seriellen Datenübertragung* können die Zeitabstände zwischen den einzelnen Zeichen variieren (Bild 7-28b), d.h., der Empfänger synchronisiert sich mit dem Sender bei jedem Zeichen neu. Man bezeichnet dieses Verfahren deshalb auch als *Start-Stopp-Verfahren*. Da bei der asynchron-seriellen Übertragung die Synchronität für die Bitübertragung immer nur für die Dauer einer Zeichenübertragung aufrecht erhalten zu werden braucht (bei der synchron-seriellen Übertragung für die Gesamtdauer der Übertragung), ist der technische Aufwand für die Sende- und Empfangseinrichtungen (im Vergleich zur synchron-seriellen Übertragung) gering.

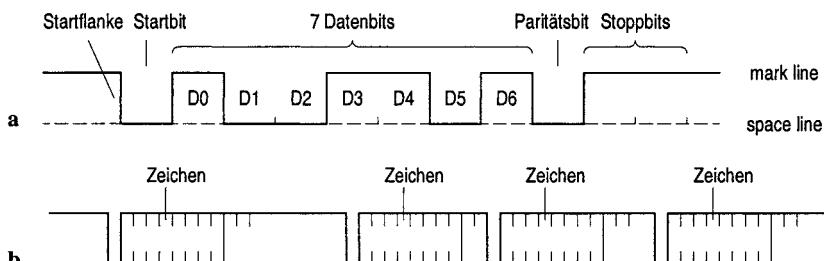


Bild 7-28. Asynchron-serielle Datenübertragung. a Darstellung des ASCII-Zeichens Y mit 7 Datenbits, einem Paritätsbit für gerade Parität und 2 Stoppbits, b Zeichenfolge mit variablem Zeichenabstand

Die Steuerung der Datenübertragung wird auf den beiden Seiten der Übertragungsstrecke von je einem Interface-Baustein durchgeführt. Ein solcher Baustein verwaltet die Schnittstellensignale gemäß der Norm V.24 (RS-232-C) auf der Basis von TTL-Pegeln: 0 V für „0“ und +5 V für „1“. Man bezeichnet diese Signaldarstellung auch als *NRZ-Codierung* (non return to zero). Der Baustein wird ergänzt durch Signaltreiber- und Signalempfängerbausteine, die die elektrotechnischen Eigenschaften der Schnittstelle, d.h. der Übertragungsstrecke, festlegen, z.B. V.28 (RS-232-C), V.10 (RS-423-A), V.11 (RS-42-A) oder RS-485. Hiervom hängt letztlich die auf der Übertragungsstrecke maximal mögliche Übertragungsrate ab, die ggf. auf 20 kbit/s begrenzt ist, aber auch sehr viel höher sein kann (siehe Tabelle 7-3).

7.5.1 Bit- und Zeichensynchronisation

Übertragungsprotokoll. Bild 7-28a zeigt das *Übertragungsprotokoll*, d.h. den zeitlichen Ablauf der Übertragung eines Zeichens, als Signaldiagramm. Der Wert 0 eines Bits wird durch die Space-Line und der Wert 1 durch die Mark-Line dargestellt. An den Signalanschlüssen des Interface-Bausteins entsprechen ihnen die TTL-Pegel 0 V und +5 V (nach V.24: Schnittstellenleitungen SGND und TxD bzw. RxD). Ein Zeichen umfaßt je nach Vereinbarung 5 bis 8 Datenbits (im Bild 7 Bits) und beginnt mit dem Bit niedrigster Wertigkeit D0. Es kann wahlweise durch ein Paritätsbit P für gerade oder ungerade Parität ergänzt werden (7.7.3). Eingerahmt wird das Zeichen einschließlich Paritätsbit durch ein *Startbit* (space) und je nach Vereinbarung ein, eineinhalb oder zwei *Stopppbits* (marks). Dieser Datenrahmen (frame) signalisiert dem Empfänger den Beginn und das Ende einer Zeichenübertragung. Zwischen zwei Zeichenübertragungen gibt der Sender als Trennsignal einen konstanten 1-Pegel beliebiger Dauer aus; ggf. kann auf das letzte Stopppbit auch sofort das nächste Startbit folgen. Die Stopppbits bzw. der 1-Pegel zwischen zwei Übertragungen sind erforderlich, um das nächste Startbit erzeugen zu können. – Aufgrund der verschiedenen Vereinbarungsmöglichkeiten ergeben sich zahlreiche Varianten an Datenformaten unterschiedlicher Bitanzahl. Die Datenformate werden ggf. in Kurzschreibweise angegeben. So bezeichnet z.B. 8-N-1 eine Übertragung mit 8 Datenbits, keinem Paritätsbit und einem Stopppbit.

Synchronisation. Zur Übernahme der einzelnen Bits benötigt der Empfänger ein Taktsignal, dessen Frequenz gleich der Senderfrequenz ist. Dieses Taktsignal wird üblicherweise durch Frequenzteilung aus einem Takt mit 16- oder 64-fach höherer Frequenz eines Empfängertaktgenerators gewonnen, wodurch eine gesonderte Taktleitung zwischen Sender und Empfänger entfällt. Die Synchronisation des Empfängertakts mit dem Sendertakt erfolgt jeweils zu Beginn einer Zeichenübertragung mit der fallenden Flanke des Startbits. Mit Hilfe der Frequenzteilung wird der Empfängertakt um die halbe Schrittweite gegenüber der

Startbitflanke verzögert, so daß die Datenübernahmeflanken des Empfängertakts gerade in die Mitte der Taktintervalle des Sendertakts fallen. Die zu diesen Zeitpunkten abgetasteten Signalpegel der einzelnen Datenbits werden nacheinander in ein Schieberegister gebracht, von wo das Zeichen in das Empfangsdatenregister übernommen wird.

Das Startbit legt somit sowohl den Bezugspunkt für die Bitübernahme (*Bitsynchronisation*), als auch den Beginn eines Zeichens fest (*Zeichensynchronisation*). Durch die Abfrage des Startbitpegels nach der halben Schrittdauer kann der Empfänger feststellen, ob die empfangene Flanke zu einem Startbit gehört oder lediglich die Flanke eines kurzen Störimpulses innerhalb einer Übertragungspause war. Durch Abfragen des bzw. der Stoppbits überprüft er zusätzlich, ob die Übertragung des Zeichens korrekt abgeschlossen wurde. Die Stoppbits gewähren darüber hinaus dem Empfänger eine „Erholzeit“ für das Speichern oder Verarbeiten des Zeichens. Außerdem gewährleistet der Stoppbit-Pegel die für die nächste Zeichenübertragung erforderliche fallende Startbitflanke.

Baud-Rate. Die für die Taktversorgung benötigten Taktgeneratoren bezeichnet man, da sie die sog. *Baud-Rate* (korrekt *Schrittgeschwindigkeit*) für die Datenübertragung vorgeben, als Baud-Raten-Generatoren. Sie können entweder in die beiden Interface-Bausteine integriert sein, oder sie werden als Zusatzbausteine zu diesen verwendet. Dadurch, daß die Synchronität zwischen Empfänger und Sender immer nur für die Dauer eines Zeichens gewährleistet sein muß, werden keine hohen Ansprüche an deren Frequenzstabilität gesetzt. Steht anders als oben beschrieben zusätzlich zur Datenleitung eine Taktleitung zur Verfügung, so genügt ein einziger Taktgenerator, z.B. auf der Senderseite. In diesem Fall wird meist der Schritttakt selbst übertragen (Frequenzteilungsverhältnis 1:1). Für den Empfänger muß dabei gewährleistet sein, daß das Taktignal für die Datenabtastung um die halbe Schrittweite gegenüber der Senderseite verzögert ist, z.B. indem er nicht die vom Sender benutzte, sondern die Gegenflanke des Taktsignals auswertet.

7.5.2 Erhöhung der Übertragungssicherheit

Gerätebereitschaft. Eine erste Sicherheitsmaßnahme für die Datenübertragung mit einem peripheren Gerät ist das Vergewissern, daß beide Übertragungspartner betriebsbereit sind (z.B. das Gerät angeschaltet und initialisiert ist), und daß es überhaupt eine Datenverbindung zwischen den beiden Übertragungspartnern gibt, d.h., daß sie durch ein Kabel miteinander verbunden sind. Dazu bieten sich die Schnittstellensignale DTR und DSR/DCD an, die bei den Interface-Bausteinen als Ausgang bzw. Eingänge wirken. Über eine jeweilige Verbindung DTR nach DSR/DCD, d.h. durch Überkreuzen dieser Anschlüsse (wie in Bild 7-25a, S. 496, gezeigt), kann die jeweilige Betriebsbereitschaft angezeigt werden. Damit wird zugleich signalisiert, daß eine Kabelverbindung besteht. Will man diese Sicherheitsmaßnahme umgehen, z.B. weil das vorhandene Kabel den Mehrauf-

wand an zwei zusätzlichen Signalleitungen nicht vorsieht, so muß bei beiden Übertragungspartnern der DTR-Ausgang mit den Eingängen DSR und DCD direkt verbunden werden, was üblicherweise durch Überbrücken im Stecker geschieht (Bild 7-25b).

Implizite und explizite Synchronisation. Im Gegensatz zur parallelen Datenübertragung, bei der die Zeichensynchronisation über eine oder zwei eigens dafür vorgesehene Steuerleitungen erfolgt (READY bzw. READY und ACKN, siehe 7.1.2), wird bei der asynchron-seriellen Datenübertragung die Synchronisation unmittelbar aus dem Datensignal abgeleitet, genauer, aus dem Datenrahmen. So ersetzt das Startbit das Bereitstellungssignal des Senders (READY), und auf die Quittierung des Empfangs (ACKN) kann aufgrund des/der Stoppbits verzichtet werden. Anders ausgedrückt: Die Taktung und die durch das Übertragungsprotokoll vorgegebene Bitanzahl einschließlich der Stoppbits sagen dem Sender implizit, wann der Empfänger ein Zeichen übernommen haben muß.

Diese Art der Synchronisation birgt die Gefahr, daß ein auf der Empfängerseite eingegangenes Zeichen ggf. nicht rechtzeitig aus dem Empfangsdatenregister des Interface-Bausteins abgeholt wird und dieses Zeichen dann verlorengeht, indem es durch das nächste empfangene Zeichen überschrieben wird. Dies wird zwar als sog. *Overrun-Error* vom Interface angezeigt (7.5.3); die dann erforderliche Fehlerbehandlung bedeutet aber einen zusätzlichen Aufwand mit Zeitverlust, z.B. das Wiederholen der Übertragung.

Eine höhere Übertragungssicherheit erreicht man, wenn man diese (implizite) Synchronisation durch eine (explizite) *Handshake-Synchronisation* ergänzt und so das obige Problem eines „Overruns“ ausschließt. Hierfür werden die bei beiden Interface-Bausteinen vorhanden Schnittstellensignale RTS und CTS verwendet und als zwei zusätzliche Schnittstellenleitungen über Kreuz, d.h. RTS nach CTS, miteinander verbunden, wie in Bild 7-25a (S. 496) gezeigt. Vor dem Senden eines Zeichens meldet nun der Sender dem Empfänger seinen Sendewunsch durch Aktivieren seines RTS-Ausgangs. Der Empfänger wertet diesen an seinem CTS-Eingang aus und meldet seinerseits dem Sender, sobald er bereit ist, seine Empfangsbereitschaft durch Aktivieren seines RTS-Ausgangs. Der Sender wiederum wertet seinen CTS-Eingang aus und gibt das Zeichen erst dann aus, wenn er die Empfangsbereitschaft als Aktivpegel angezeigt bekommt.

Im Gegensatz zur impliziten Synchronisation, die vollständig in Hardware abläuft, ist die explizite Synchronisation im Prinzip rein software-gesteuert. Eine Beeinflussung durch die Hardware gibt es ggf. dadurch, daß der Interface-Baustein im Hinblick auf eine Modem-Verbindung seinen Sendeteil bei inaktivem CTS-Eingang blockiert. Nachteil dieser Technik ist wieder, daß man im Verbindungskabel zwei Leitungen mehr benötigt. Will man diesen Mehraufwand an Leitungen nicht in Kauf nehmen und ist aber die explizite Synchronisation in der Software vorgesehen, so muß man der Software die Handshake-Synchronisation vortäuschen. Dazu wird bei beiden Interface-Bausteinen der RTS-Ausgang direkt

mit dem CTS-Eingang verbunden, indem die beiden Anschlüsse im jeweiligen Stecker überbrückt werden (Bild 7-25b, S. 496). Das heißt, der jeweilige Sender erzeugt sich das die Empfangsbereitschaft anzeigenende Signal durch Anzeigen seiner Sendebereitschaft selbst.

Die implizite und die explizite Synchronisation finden auf der Signalebene statt. Bezogen auf das *OSI-Referenzmodell* sind sie damit der untersten Schicht einer Hierarchie an Übertragungsprotokollen, d.h. der Schicht 1 zuzuordnen (7.7.1). Eine weitere Synchronisation, die in der nächst höheren Ebene stattfindet, d.h. der Schicht 2, ist nachfolgend beschrieben.

X-ON/X-OFF-Synchronisation. Bei der kontinuierlichen Übertragung von Daten an Ausgabegeräte, die mit einem Pufferspeicher ausgestattet sind (z.B. Drucker und Bildschirmterminals), besteht die Notwendigkeit, einen Datenverlust zu verhindern, der durch das Überlaufen des Puffers auftreten kann. Dazu muß sich der Empfänger mit dem Sender verständigen, indem er ihm signalisiert, die Datenübertragung bei einem bestimmten Füllungsgrad des Puffers zu stoppen (transfer disable, X-OFF) bzw. sie bei einem bestimmten Leerungsgrad wieder aufzunehmen (transfer enable, X-ON). Realisiert wird diese Synchronisation unter Nutzung des Duplexbetriebs (7.2.6), nämlich durch Übertragen von Steuerzeichen auf dem der Datenübertragung entgegengesetzten Datenweg. Die beiden benötigten Steuerzeichen haben im ASCII die Werte 0x13 (X-OFF, device control 3) und 0x11 (X-ON, device control 1). – Wird der Rückkanal nicht nur zur Steuerung, sondern auch für die reguläre Datenübertragung benutzt, so müssen die Steuerzeichen für eine *transparente Datendarstellung* besonders gekennzeichnet werden; siehe dazu 7.6.2.

Der Vorteil dieser Technik ist, daß das Verbindungskabel über die beiden Daten signale und die Betriebserde hinaus nicht erweitert werden muß. Es gibt aber auch die Variante, diese Synchronisation im Simplexbetrieb (7.2.6), d.h. über eine zusätzliche Leitungsverbindung, z.B. RTS nach CTS oder DTR nach DSR, durchzuführen, wenn diese Leitungsverbindung nicht für eine der anderen beschriebenen Sicherheitsmaßnahmen benötigt wird.

7.5.3 Asynchron-serieller Interface-Baustein

Im folgenden wird ein asynchron-serieller Interface-Baustein vorgestellt, der mit einer 8-Bit-Schnittstelle für den Systemdatenbus und zwei 1-Bit-Schnittstellen für die Peripherie ausgestattet ist. Er übernimmt die Parallel-Serien- und die Serien-Parallel-Umsetzung der zu übertragenden Zeichen, die Datensicherung und Datenüberprüfung, die Takt- und Zeichensynchronisation sowie die Steuerung der Übertragung im Duplexbetrieb. Außerdem ermöglicht er die Vereinbarung verschiedener Datenformate.

Blockstruktur. Bild 7-29 zeigt die Struktur des Bausteins mit einem Empfangsteil (receiver) und einem Sendeteil (transmitter) auf der Peripherieseite (universal asynchronous receiver transmitter, *UART*). Neben den zentralen Registern TDR (transmit data register) und RDR (receive data register) zur Datenpufferung und den beiden Schieberegistern TSR (transmit shift register) und RSR (receive shift register) zur Parallel-Seriens- bzw. Serien-Parallel-Umsetzung besitzt der Baustein ein Steuerwerk mit einem Modusregister MR, einem Steuerregister CR und einem Statusregister SR. Weiterhin sind im Steuerwerk eine Unterbrechungseinrichtung mit Selbstidentifizierung des Bausteins als Interruptquelle und eine Verkettungslogik für die Priorisierung des Bausteins nach dem Daisy-chain-Prinzip untergebracht, die bei einfacheren Baustinausführungen fehlen können. Ein zusätzliches Übertragungssteuerwerk verwaltet drei Steuerleitungen für den Signalaustausch mit der Peripherie.

Die Schnittstelle zum Systembus ist wie beim Parallel-Interface-Baustein (7.3.2) ausgeführt, d. h., der Datentransport zwischen Mikroprozessor und Interface wie auch innerhalb des Interface-Bausteins erfolgt byteweise bitparallel über die Datenanschlüsse D7 bis D0. – Die Schnittstelle zur Peripherie (Ein-/Ausgeberät oder Modem) sieht für den Sendeteil eine Sendedatenleitung TxD (transmit data) und für den Empfangsteil eine Empfangsdatenleitung RxD (receive data) zur bitseriellen Datenübertragung vor. Sende- und Empfangsteil haben außerdem

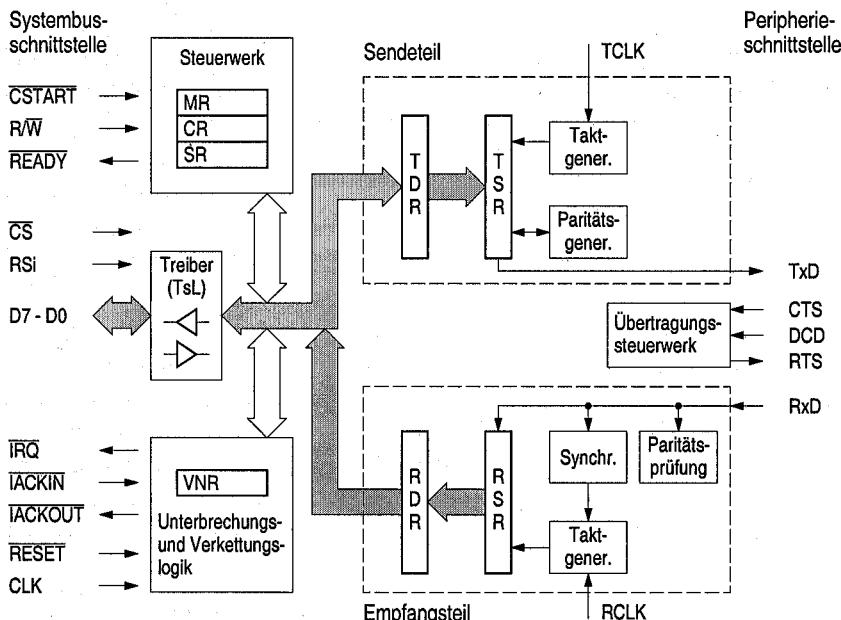


Bild 7-29. Struktur eines asynchron-seriellen Interface-Bausteins mit einem Sende- und einem Empfangsteil

je einen Takteingang **TCLK** (transmitter clock) bzw. **RCLK** (receiver clock). Aus den dort anliegenden Taktsignalen eines bausteinexternen Baud-Raten-Generators bildet der Baustein die beiden Schrittaktsignale durch Signalformung in Verbindung mit dem vorgegebenen Frequenzteilungsverhältnis 1:1 oder 1:16. Zur Steuerung der Datenübertragung mit der Peripherie gibt es einen Signalausgang **RTS** (request to send) sowie zwei Signaleingänge **CTS** (clear to send) und **DCD** (data carrier detect).

Funktionsweise. Das *Senden* eines Zeichens wird mit Abschluß des Schreibens des Zeichen in das 8-Bit-Sendedatenregister **TDR** ausgelöst. Das Zeichen wird unmittelbar in das 8-Bit-Schieberegister **TSR** übernommen und von dort bitweise im Schrittakt auf die Sendedatenleitung **TxD** ausgegeben (Parallel-Serien-Umsetzung). Dabei werden die Datenbits mit den Start-, Paritäts- und Stopppbits entsprechend dem im Modusregister **MR** vorgegebenen Datenformat versehen. Die Bereitschaft des Senders für die Übertragung des nächsten Zeichens wird im Statusregister **SR** angezeigt und kann vom Mikroprozessor entweder abgefragt werden, oder sie wird als Interruptanforderung ausgewertet (Ausgabe-Synchronisation).

Das *Empfangen* eines auf der Empfangsdatenleitung **RxD** bitweise ankommenen Zeichens wird durch dessen *Startbitflanke* ausgelöst (Bit- und Zeichensynchronisation). Ausgehend von der Startbitflanke verzögert der Empfänger seinen *Schrittakt* um eine halbe Schrittaktweite und übernimmt die Datenbits in das 8-Bit-Schieberegister **RSR**. (Bei Datenformaten mit weniger als 8 Datenbits werden die höherwertigen Bitpositionen des Registers mit 0-Bits aufgefüllt.) Von dort wird das Zeichen ohne Start-, Stoppp- und Paritätsbit in das 8-Bit-Empfangsregister **RDR** übertragen (Serien-Parallel-Umsetzung). Während des Empfangs eines Zeichens wertet der Empfänger das *Paritätsbit* und die *Stopppbits* entsprechend den im Modusregister enthaltenen Vorgaben aus. Die Verfügbarkeit des Datums in **RDR** wie auch Übertragungsfehler werden im Statusregister angezeigt. Die Statusinformation kann vom Mikroprozessor abgefragt werden, oder sie wird als Interruptanforderung ausgewertet (Eingabe-Synchronisation).

Die Datenübertragung zwischen dem Interface und der Peripherie (einem Gerät) wird durch die Steuersignale **RTS**, **CTS** sowie **DCD** unterstützt. Gegenüber der Beschreibung in 7.5.2 ist hier die Schnittstelle etwas einfacher gehalten, indem die Anschlüsse **DTR** und **DSR** fehlen. **RTS** und **CTS** können jedoch, wie in 7.5.2 gezeigt, zur expliziten Synchronisation der Zeichenübertragung genutzt werden. Dazu läßt sich das **RTS**-Signal über das Steuerregister **CR** setzen bzw. rücksetzen, und der Pegel des **CTS**-Signals kann über das Statusregister **SR** abgefragt werden. Ist **CTS** inaktiv, so wird außerdem das Statusregisterbit **TDRE** blockiert (bleibt auf 0), womit die Anzeige der Ausgabebereitschaft unterdrückt wird. Die in 7.5.2 beschriebene Sicherheitsmaßnahme, daß eine Datenübertragung nur dann eingeleitet wird, wenn beide Übertragungspartner betriebsbereit sind und eine Kabelverbindung besteht, kann durch das **DCD**-Signal allein realisiert werden. Das Signal muß dazu von der jeweiligen Gegenseite mit dem Aktivpegel belegt

werden. Ist es inaktiv, so hindert es den Empfangsteil daran, Zeichen in das Schieberegister RSR zu übernehmen. Der Pegel des DCD-Signals ist außerdem als Statusbit durch die Software auswertbar.

Betriebsarten und Übertragungssteuerung. Die Betriebsart des Interface-Bausteins wird über das 8-Bit-Modusregister MR (Bild 7-30) eingestellt, das bei der Initialisierung des Mikroprozessorsystems entsprechend den Anforderungen der Peripherie geladen wird. Die Modusbits MOD1 und MOD0 geben neben dem normalen *Duplexbetrieb* des voneinander unabhängigen Sendens und Empfangens drei weitere, spezielle Betriebsarten vor.

1. Beim „*Echobetrieb*“ wird ein empfangenes Zeichen sowohl in das Empfangsdatenregister RDR übernommen als auch auf der Sendedatenleitung TxD als „Echo“ automatisch wieder ausgegeben. Dabei ist der Sender von der Prozessorseite her nicht benutzbar.
2. Beim „*Local-loop-Betrieb*“ wird ein vom Prozessor ausgegebenes Zeichen nicht auf die Sendedatenleitung TxD gegeben, sondern unmittelbar der Empfangsdatenleitung RxD zugeführt. Diese Betriebsart erlaubt das Testen des Bausteins unabhängig vom Übertragungsweg.
3. Beim „*Remote-loop-Betrieb*“ wird ein auf der Empfangsdatenleitung RxD eintreffendes Zeichen unmittelbar wieder auf der Sendedatenleitung TxD ausgegeben, ohne im Empfangsdatenregister RDR gespeichert zu werden. Diese Betriebsart unterstützt das Testen des Übertragungsweges, ausgehend vom gegenüberliegenden Interface.

Die Modusbits DF1 und DF0 bestimmen das Datenformat, STP die Anzahl der Stoppbits und P1 und P0 die Art der Datensicherung. Das Bit FT gibt das durch den externen Baud-Raten-Generator vorgegebene Frequenzteilungsverhältnis 1:1 bzw. 1:16 an. (Bei Bausteinen mit internem Baud-Raten-Generator gibt es ein zweites Modusregister zur Vorgabe der Schrittaktfrequenz.)

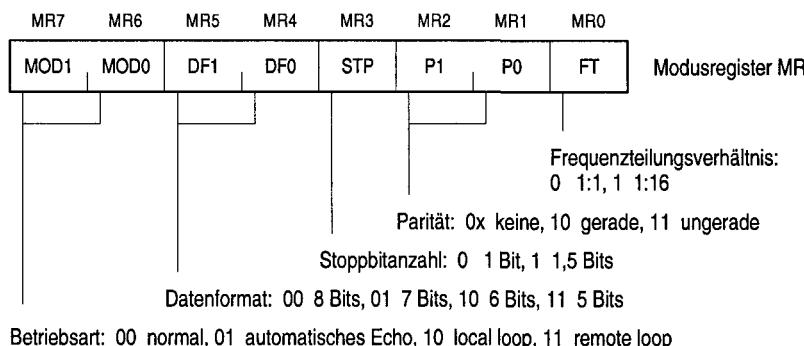


Bild 7-30. Modusregister des asynchron-seriellen Interface-Bausteins nach Bild 7-29

Die Steuerung eines Ein-/Ausgabevorgangs erfolgt durch Laden des 8-Bit-Steuerregisters CR (Bild 7-31). Mit den Bits TRE (transmitter enable) und REE (receiver enable) werden der Sender bzw. der Empfänger angeschaltet. TIRE (transmitter interrupt enable) und RIRE (receiver interrupt enable) entscheiden, ob die Statusbits TDRE (transmit data register empty) und RDRF (receive data register full) mit ihrem 1-Zustand einen Interrupt auslösen oder nicht (Synchronisation durch Programmunterbrechung bzw. Busy-Waiting). Mit RIRE wird außerdem festgelegt, ob die Statusbitanzeigen DCD = 0 (DCD-Signal inaktiv) und OVRN = 1 (*overrun error*) eine Interruptanforderung auslösen.

Mit dem RTS-Bit kann das Ausgangssignal RTS aktiviert oder inaktiviert werden. Das Setzen des BRK-Bits (break signaling) verursacht eine Unterbrechung der Übertragung, indem nach Ausgabe des zuletzt nach TDR geschriebenen Zeichens auf der Sendedatenleitung TxD für die Dauer einer Zeichenübertragung ein konstanter 0-Pegel, ein sog. *Break-Signal* erzeugt wird. Der gegenüberliegende Empfänger erkennt die Unterbrechung am Empfang eines Datenbytes 0x00 und der gleichzeitigen Anzeige eines Framing-Errors (Statusbit FE), sozusagen als Software-Interrupt. (Die Break-Funktion nützt insbesondere auch dazu, eine unterbrochene Empfangsdatenleitung RxD als Defekt zu erkennen. RxD führt in diesem Fall nicht mehr den im Ruhezustand üblichen 1-Pegel, sondern 0-Pegel.) Mit dem ERRR-Bit (error reset) können die Fehlerbits im Statusregister zurückgesetzt werden. – Die Funktionen Break und Error-Reset werden vom Prozessor durch „Schreiben“ einer „1“ in die entsprechende Bitposition im Steuerregister ausgelöst, ohne daß diese „1“ dort als Bit gespeichert wird (weil nur als Leitung ausgeführt).

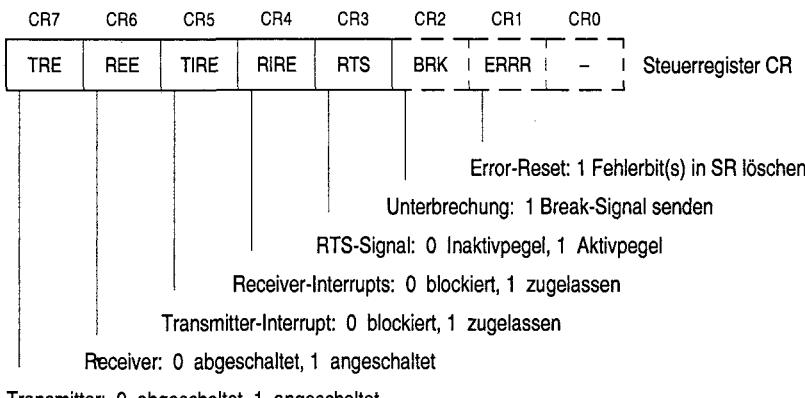


Bild 7-31. Steuerregister des asynchron-seriellen Interface-Bausteins nach Bild 7-29

Status. Der Zustand des Sende- und Empfangsteils wird im 8-Bit-Statusregister SR angezeigt (Bild 7-32). TDRE (transmit data register empty) dient zur Synchronisation der Ausgabe. Es zeigt dem Prozessor im gesetzten Zustand an, daß

das Sendedatenregister TDR leer ist und er es also mit dem nächsten auszugebenden Zeichen laden kann. Das Setzen des Bits hängt zusätzlich davon ab, ob das CTS-Eingangssignal aktiv ist. Zurückgesetzt wird das Bit jeweils mit dem nächsten Schreibvorgang. Das Bit RDRF (receive data register full) dient zur Synchronisation der Eingabe. Es zeigt im gesetzten Zustand an, daß das Empfangsdatenregister RDR voll ist und gelesen werden kann. Mit dem Lesen von RDR wird das Bit zurückgesetzt. Beim Anlegen der Versorgungsspannung an den Baustein werden diese beiden Bits initialisiert, wobei TDRE gesetzt (Ausgabebereitschaft) und RDRF gelöscht wird (power-on reset).

SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	Statusregister SR
TDRE	RDRF	CTS	DCD	FE	OVRN	PE	ERR	
								Error: 0 kein Fehlerbit gesetzt, 1 Fehlerbit(s) gesetzt
								Parity-Error: 0 kein Fehler, 1 Paritätsfehler
								Overrun-Error: 0 kein Überlauf, 1 Überlauf
								Framing-Error: 0 kein Rahmungsfehler, 1 Rahmungsfehler
								DCD-Eingang: 0 Inaktivpegel, 1 Aktivpegel
								CTS-Eingang: 0 Inaktivpegel, 1 Aktivpegel
								Receiver-Data-Register: 0 leer, 1 voll
								Transmitter-Data-Register: 0 voll, 1 leer

Bild 7-32. Statusregister des asynchron-seriellen Interface-Bausteins nach Bild 7-29

Die Statusbits CTS und DCD zeigen die Pegel der Eingangssignale CTS und DCD an. Die Bits FE, OVRN und PE dienen zur Fehleranzeig. Das FE-Bit (framing error) zeigt fehlerhafte Start- oder Stoppbits eines empfangenen Zeichens an. Das OVRN-Bit (*overrun error*) wird gesetzt, wenn ein Zeichen in das Empfangsdatenregister RDR übernommen wird, bevor das zuvor eingetroffene Zeichen gelesen worden ist. Das PE-Bit (*parity error*) wird gesetzt, wenn das empfangene Zeichen nicht die im Datenformat festgelegte Paritätsbedingung erfüllt. Zur Vereinfachung der Fehlerabfrage wirken die Bits DCD (mit dem 0-Pegel), FE, OVRN und PE als Oder-Funktion auf das Statusbit ERR. Das Rücksetzen von FE, OVRN und PE erfolgt über das Steuerbit ERR.

Sowohl der Empfänger als auch der Sender können Interrupts auslösen, sofern die entsprechenden Interrupt-Enable-Bits RIRE und TIRE im Steuerregister gesetzt sind. Unterbrechungsursachen sind im Empfangsteil die Übernahme eines Zeichens in das Empfangsdatenregister RDR (RDRF = 1), der Inaktivpegel am DCD-Eingang (DCD = 0) und der Overrun-Error (OVRN = 1). Unterbrechungsursache im Sendeteil ist ein leerer Sendedatenregisters TDR (TDRE = 1), sofern das CTS-Signal aktiv ist.

Beispiel 7.4. ► *Dateneingabe und Datenausgabe über den asynchron-seriellen Interface-Baustein nach Bild 7-29.* Ein alphanumerisches Terminal mit einer Tastatur und einem Bildschirm ist über einen asynchron-seriellen Interface-Baustein an den Mikroprozessorsystembus angeschlossen (Bild 7-33). Die von der Tastatur kommenden ASCII-Zeichen werden über die Empfangsdatenleitung und die an den Bildschirm ausgegebenen ASCII-Zeichen über die Sendedatenleitung des Interfaces übertragen. Durch die RTS-Leitung kann die Tastatur freigegeben (RTS = 1) oder gesperrt werden (RTS = 0). Die Signaleingänge CTS und DCD sind auf 1-Pegel (aktiv) festgelegt.

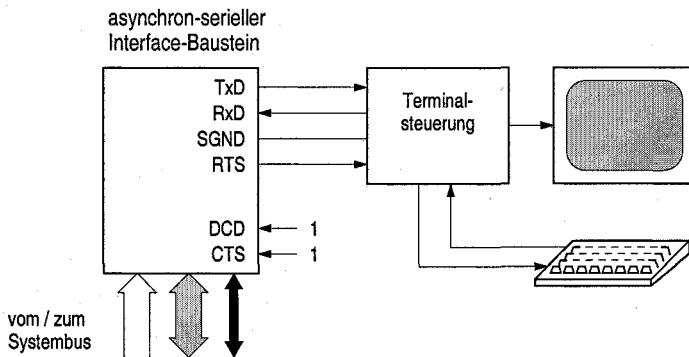


Bild 7-33. Anschluß eines Bildschirmterminals an den asynchron-seriellen Interface-Baustein nach Bild 7-29

Für den Einsatz dieses Terminals ist ein Programm zu schreiben, das eine Textzeile mit ASCII-Zeichen in einen Pufferbereich LINE des Hauptspeichers einliest und jedes dieser Zeichen als programmiertes Echo unmittelbar wieder an den Bildschirm ausgibt. Die Dateneingabe ist durch Programmunterbrechung, die Datenausgabe durch Busy-Waiting innerhalb des Interruptprogramms zu synchronisieren. Das Ende der Textzeile wird durch das Carriage-Return-Zeichen (0x0D) oder, unabhängig davon, durch maximal 80 Zeichen vorgegeben. Dies ist vom Interruptprogramm aus dem Hauptprogramm anzuzeigen, indem es einer zuvor mit 0x00 initialisierten Variablen STATUS den Wert 0xFF zuweist. Als Datenformat werden 7 Datenbits ohne Paritätsbit, als Datenrahmen 1,5 Stoppbits vorgegeben. Der externe Baud-Raten-Generator arbeitet mit dem Frequenzteilungsverhältnis 1:16. – Zur Vereinfachung der Aufgabenstellung soll auf eine Fehlerauswertung verzichtet werden.

Bild 7-34 zeigt den Ablauf. Vor Beginn der Ein-/Ausgabe wird im Hauptprogramm zunächst die Variable STATUS mit dem Wert 0 initialisiert und die Startadresse INOUT des Interruptprogramms an die erste verfügbare Adresse (256) der Vektor-Interrupts in der Vektortabelle geschrieben (siehe 2.1.5: Tabelle 2-4, S. 100); die Tabelle beginnt bei der Adresse 0. Die Initialisierung des Interfaces erfolgt (1.) durch Laden der Vektornummer 64 in das Vektornummerregister VNR, (2.) durch Laden des Modusregisters mit 0x19 zur Vorgabe des Normalmodus, des Datenformats, des Datenrahmens und des Frequenzteilungsverhältnisses und (3.) durch Laden des Steuerregisters mit der Steuerinformation 0xD8, um den Sender und den Empfänger anzuschalten, Interrupts bei der Eingabe zuzulassen, Interrupts bei der Ausgabe zu blockieren und die Tastatur über die RTS-Leitung freizugeben. – Die beiden Statusbits RDRF und TDRE befinden sich zu diesem Zeitpunkt in ihrem Initialzustand: RDRF = 0, TDRE = 1.

Mit dem Anschlagen einer Taste wird ein Zeichen bitseriell an das Interface übertragen, wodurch das Interface-Statusbit RDRF gesetzt wird. Dieses löst die Ausführung des Interruptprogramms aus, während der das Zeichen aus dem Empfangsdatenregister RDR des Interfaces in einen über R5 registerindirekt adressierten Pufferbereich LINE gelesen wird. Die anschließende Ausgabe

dieses Zeichens an das Sendedatenregister TDR zur Anzeige auf dem Bildschirm wird von der Anzeige TDRE = 1 im Statusregister abhängig gemacht (TDR ist leer). Das Programm fragt dazu das TDRE-Bit in der Warteschleife WAIT ab. Der Ein-/Ausgabevorgang wird nach Empfang eines Carriage-Return-Zeichens oder, nachdem der Puffer gefüllt ist, beendet. Dazu wird das Steuer-

```
; Asynchron-serielle Ein-/Ausgabe
; mit Programmunterbrechung und
; Busy-Waiting
```

```
; Hauptprogramm
```

```
ORG    0x40000
DEF    STATUS
REF    BUFPTR, BUFEND, INOUT
VECTAB EQU 0
LINE   DS.B 80
STATUS DS.B 1
:
LEA    LINE, BUFPTR
LEA    LINE+80, BUFEND
MOVE.B #0, STATUS
LEA    INOUT, VECTAB+64*4
MOVE.B #64, VNR
MOVE.B #0x19, MR
MOVE.B #0xD8, CR
```

```
; Interruptprogramm
```

```
DEF    BUFPTR, BUFEND, INOUT
REF    STATUS
BUFPTR DS.W 1
BUFEND DS.W 1
INOUT: PUSH.W R5
MOVE.W BUFPTR, R5
MOVE.B RDR, (R5)
WAIT: BTST.B #7, SR
BEQ    WAIT
MOVE.B (R5), TDR
CMP.B #0x0D, (R5) +
BEQ    END
CMP.W R5, BUFEND
BNE    RET
END:  MOVE.B #0xD0, CR
MOVE.B #0xFF, STATUS
RET:  MOVE.W R5, BUFPTR
POP.W R5
RTE
```

Interface

Terminalsteuerung

Terminal
angeschaltet

Initialisierung

RTS = 1 ?

ja

Tastatur freigeben

Taste
betätigt ?

ja

Zeichen mit
Start- und Stoppbits
ausgeben

Initialzust.:
RDRF = 0;
TDRE = 1;

nein

Zeichen
eingetroffen ?

ja

Zeichen in Bild-
wiederholspeicher
übernehmen

RTS = 0 ?

ja

Tastatur sperren

RDRF := 1; ->

RDRF := 0; ->

TDRE := 0;

TxD-Leitg.
aktiviert

TDRE := 1

Bild 7-34. Ablauf zu Beispiel 7.4: Dateneingabe und Datenausgabe über den asynchron-seriellen Interface-Baustein nach Bild 7-29

register mit 0xD0 geladen, so daß die Tastatur über die RTS-Leitung gesperrt wird. Zusätzlich wird dies dem Hauptprogramm durch Zuweisung von 0xFF an die Variable STATUS mitgeteilt. ▲

7.6 Synchron-serielle Datenübertragung

Bei der synchron-seriellen Datenübertragung werden im Gegensatz zur asynchron-seriellen Datenübertragung nicht einzelne Zeichen mit dazwischenliegenden Unterbrechungen übertragen (Start-Stopp-Verfahren), sondern es wird ein zusammenhängender Bitstrom gebildet, d.h. eine lückenlose Bitfolge (Bild 7-35). Die einzelnen Bits unterliegen dabei wieder einem festen *Schrittakt*. Anders als bei der asynchronen Übertragung muß jedoch der *Gleichlauf* von Sender und Empfänger für die Gesamtdauer der Übertragung aufrechterhalten werden. Bei den konventionellen, *zeichenorientierten Protokollen* (character oriented protocols, COPs) wird der Bitstrom aus einzelnen Zeichen einheitlicher Bitanzahl gebildet, bei den moderneren, *bitorientierten Protokollen* (bit oriented protocols, BOPs) wird er hingegen zeichenunabhängig gebildet.

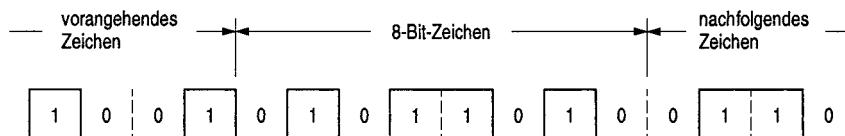


Bild 7-35. Synchron-serielle Übertragung von 8-Bit-Zeichen

Die synchron-serielle Datenübertragung erlaubt gegenüber der asynchron-seriellen Datenübertragung sehr viel höhere Übertragungsraten und wird dementsprechend zur Kommunikation zwischen Rechnern und Geräten eingesetzt, die imstande sind, einen kontinuierlichen Bitstrom mit hoher Übertragungsrate aufrechtzuerhalten. Das betrifft ggf. schnelle Ein-/Ausbabegeräte, wie Bildschirmterminals mit bildschirmorientierter Nutzung für Text und Graphik. Das betrifft insbesondere aber die Kommunikation zwischen Rechnern, z.B. in Rechnernetzen und bei der Datenfernübertragung. Bei diesen Anwendungen sind Übertragungsgeschwindigkeiten von einigen 10 kbit/s bis in den Gbit/s-Bereich hinein gebräuchlich.

7.6.1 Bit- und Zeichensynchronisation

Für die *Bitsynchronisation* zwischen Sender und Empfänger gibt es wie bei der asynchron-seriellen Datenübertragung zwei Möglichkeiten, nämlich entweder den Schrittakt für jede Übertragungsrichtung mittels einer eigenen *Taktleitung* zu übertragen oder Sender und Empfänger mit jeweils eigenen Taktgeneratoren auszustatten. Die erste Möglichkeit scheidet meist aus Kostengründen aus. So wer-

den bei der Datenfernübertragung und in Rechnernetzen keine Takteleitungen eingesetzt. Hier wird ähnlich wie bei der asynchron-seriellen Übertragung die Synchronisation des Empfängertakts mit dem Sendertakt anhand der Signalübergänge auf der Datenleitung durchgeführt. Zum Beispiel werden dazu bei zeichenorientierten Protokollen in größeren Abständen, etwa aller 100 Zeichen, spezielle *Synchronisationszeichen* in den Datenstrom eingefügt, oder es wird bei bitorientierten Protokollen die Synchronisation mit den Pegelübergängen der eigentlichen Daten durchgeführt. Im ersten Fall müssen die Frequenzen beider Taktgeneratoren aufgrund der großen Synchronisationsabstände gut stabilisiert sein. Im zweiten Fall ist es notwendig, daß im Datensignal durch eine geeignete Informationscodierung in bestimmten Abständen mit Sicherheit Pegelübergänge auftreten. Betragen die Abstände weniger als 100 Zeichen, so können die Anforderungen an die Frequenzstabilität der Taktgeneratoren entsprechend gesenkt werden.

Aufgrund des kontinuierlichen Bitstroms ist bei zeichenorientierten Protokollen zusätzlich zur Taksynchronisation eine *Zeichensynchronisation* notwendig, die dem Empfänger den Beginn des ersten gültigen Zeichens im Bitstrom anzeigt. Hierzu werden zu Beginn eines Datenübertragungsblocks ein oder zwei 8-Bit-*Synchronisationszeichen*, z.B. das ASCII-Zeichen SYN, übertragen. Der Empfänger, der sich in einer Art Suchzustand (hunt mode) befindet, beginnt mit der eigentlichen Datenübernahme erst dann, wenn er 8 bzw. 16 aufeinanderfolgende Bits als Synchronisationszeichen erkannt hat. Diese Zeichen werden auch gleichzeitig zur *Bitsynchronisation* benutzt.

7.6.2 BISYNC, HDLC- und SDLC-Protokoll

Unabhängig davon, ob der zu übertragende Bitstrom zeichen- oder bitorientiert ist, bedarf es einer Strukturierung der Information, um sie besser handhaben zu können. Hier gibt es eine Vielzahl von Festlegungen, die auf der Bildung von Datenblöcken basieren. Die Beschreibung dieser Blockbildung wie die der Übertragungssteuerung insgesamt, ist in Protokollen festgelegt. Ein *Protokoll* ist – wie früher ausgeführt – eine Sammlung von Regeln, wie sie für einen eindeutigen Ablauf der Datenübertragung erforderlich ist. Es enthält z.B. Vereinbarungen über die Art der Übertragung (asynchron oder synchron, zeichen- oder blockorientiert), über das Datenformat (bestimmter Zeichencode oder davon unabhängig), über das Format eines Datenblocks/Übertragungsblocks (Rahmen, Größe), über die Übertragungsgeschwindigkeit (Anzahl der pro Sekunde übertragenen Bits oder Zeichen) und über die Art der Datensicherung. Es bestimmt darüber hinaus die Art der Übertragung zusammenhängender Übertragungsblöcke (transmission blocks) und legt die dafür erforderliche Übertragungssteuerung (flow control) fest. Hierzu gehören z.B. das Zählen der übertragenen Blöcke, die Reklamation fehlerhaft übertragener Blöcke, die Fehlerkorrektur durch Blockwiederholung

usw. In größerem Rahmen gilt es darüber hinaus, den Informationsaustausch zwischen Sender und Empfänger zur Einleitung, zur Durchführung und zur Beendigung einer Datenübertragung festzulegen.

Diese Vielfalt an Vorgaben erreicht eine hohe Komplexität und wird deshalb in Schichten (Ebenen) unterschiedlicher Abstraktion aufgeteilt. Die bekannteste Grundlage hierfür ist das *OSI-Referenzmodell*, das die Kommunikation zwischen verschiedenen Systemen modellhaft in sieben Schichten beschreibt und insbesondere für die Kommunikation in Rechnernetzen eingesetzt wird (7.7.1). Hier gibt es eine Vielzahl von Normen und Standards. Die in 7.6.1 beschriebenen technischen Grundlagen der synchron-seriellen Datenübertragung, nämlich die Takt- und Zeichensynchronisation, bilden die Schicht 1, d.h. die unterste Schicht in dieser Hierarchie. Sie wird als *Bitübertragungsschicht* (physical layer) bezeichnet. Die darüberliegende Schicht 2 dient der Blockbildung sowie der Datensicherung auf Blockbasis und wird als *Sicherungsschicht* (data link layer) bezeichnet. Die Protokolle dieser beiden Schichten werden auch als *Übertragungsprozeduren* bezeichnet.

Als Erweiterung unserer Betrachtungen, die sich bisher nur auf die Schicht 1 bezogen, stellen wir im folgenden drei synchrone Protokolle der Schicht 2 vor, und zwar ein zeichenorientiertes, das BISYNC-Protokoll, und zwei bitorientierte, das HDLC- und das SDLC-Protokoll. Solche Protokolle der Schicht 2 gibt es auch für die asynchron-serielle Datenübertragung. Sie sind allerdings aufgrund der durch die asynchrone Übertragung eingeschränkten Übertragungsraten in Rechnernetzen wenig gebräuchlich.

BISYNC-Protokoll. Bild 7-36 zeigt als Beispiel für ein *zeichenorientiertes Protokoll* den prinzipiellen Aufbau des Übertragungsformats für das *BISYNC-Protokoll* (binary synchronous communications BSC, IBM [IBM 1970]). Die im Bild verwendeten Steuerzeichen sind als ASCII-Zeichen angegeben; die Verwendung anderer Zeichencodes ist möglich.

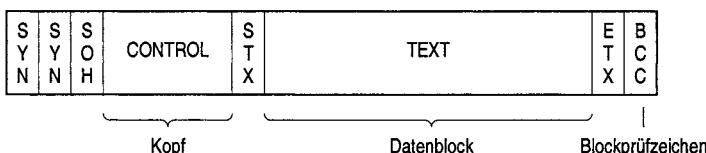


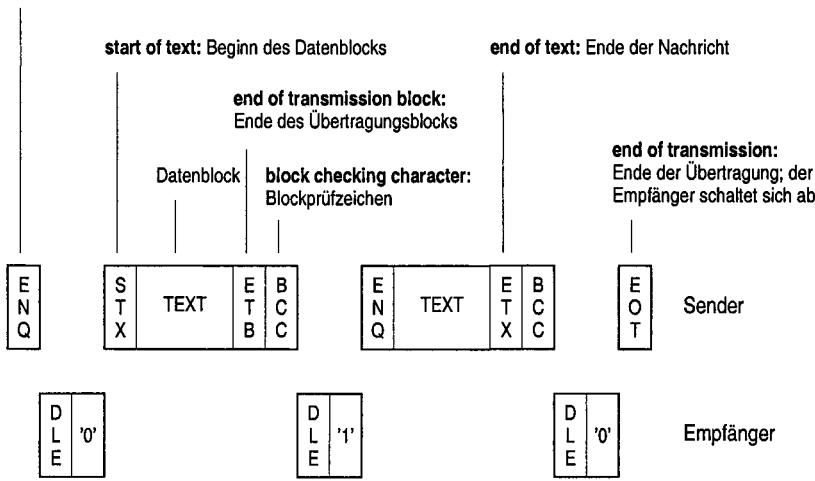
Bild 7-36. Übertragungsformat des BISYNC-Protokolls. Das Blockprüfzeichen BCC wird ab dem ersten Zeichen nach SOH bis einschließlich ETX gebildet

Ein BISYNC-Übertragungsblock beginnt mit zwei *Synchronisationszeichen* SYN (synchronous idle), auf die wahlweise ein Informationskopf (header) oder der Datenblock selbst folgt. Der Kopf ist durch das Zeichen SOH (start of heading) gekennzeichnet. Er enthält ein oder mehrere Zeichen (CONTROL), die zur Übertragungssteuerung des Blocks dienen (z.B. als Quell- oder Zieladreßangabe des

Blocks oder zur Kennzeichnung des Blockinhalts als Daten oder als Steuerinformation). Dem eigentlichen Datenblock (TEXT) ist das Zeichen STX (start of text) zur Kennzeichnung vorangestellt und das Zeichen ETX (end of text) nachgestellt. Abgeschlossen wird der Übertragungsblock durch ein 8-Bit-Blockprüfzeichen BCC (block checking character) oder eine 16-Bit-Prüfinformation BCS (block checking sequence) zur Datensicherung (siehe auch 7.7.3). Ein Sender, der nicht in der Lage ist, einen lückenlosen Zeichenstrom aufrechtzuerhalten, fügt zusätzliche SYN-Zeichen in den Zeichenstrom ein. Diese werden auf der Empfängerseite wieder aus dem Zeichenstrom entfernt und gleichzeitig zur Nachsynchronisation benutzt. Die beiden SYN-Zeichen am Blockanfang können entfallen, wenn die Synchronisation über eine gesonderte Synchronisationsleitung erfolgt. Abstrahiert betrachtet, besteht ein Übertragungsblock aus Text als der zu übertragenden Nutzinformation und aus einem diesen Text umgebenden Rahmen zur Übertragungssteuerung (frame).

Durch solche Vereinbarungen lassen sich Übertragungsformate mit unterschiedlichen Strukturen definieren, so auch Formate, die neben den Synchronisationszeichen lediglich ein oder zwei Steuerzeichen enthalten. Solche textlosen Formate werden zum Austausch von Steuerinformation zwischen Sender und Empfänger bei der Steuerung der Datenübertragung benutzt. – Bild 7-37 zeigt als Beispiel einer Übertragungssteuerung die Einleitung, Durchführung und Beendigung einer Datenübertragung. Der Sender und der Empfänger tauschen dazu

enquiry: Anmeldung des Übertragungswunsches



data link escape: Bestätigung der Empfangsbereitschaft oder eines fehlerfrei empfangenen Datenblocks.
Auf DLE folgen abwechselnd die Zeichen 0 und 1 als zusätzliche Prüfmöglichkeit.

Bild 7-37. Übertragungssteuerung beim BISYNC-Protokoll

Übertragungsblöcke im *Handshake-Verfahren* aus, die im Bild entsprechend ihrer zeitlichen Reihenfolge von links nach rechts angeordnet sind. Die Übertragung erfolgt im *Halbduplexbetrieb*, d.h., Sender und Empfänger wechseln sich in der Benutzung des Übertragungswegs ab. Die vom Sender ausgegebenen Blöcke sind in der oberen Bildhälfte, die vom Empfänger ausgegebenen Blöcke in der unteren Bildhälfte dargestellt und erläutert. Aus Platzgründen wurde auf die Angabe der SYN-Zeichen vor jedem Block verzichtet. Die zu übertragenden Daten sind zur besseren Datensicherung auf zwei Übertragungsblöcke verteilt. Im ersten Block ist dies durch das Zeichen ETB (end of transmission block) anstelle des Zeichens ETX für den Empfänger kenntlich gemacht.

Protokolle, bei denen die Felder des Übertragungsformats durch Steuerzeichen eines bestimmten Zeichencodes festgelegt sind, erfordern für ihre Implementierung wegen der notwendigen Interpretation dieser Steuerzeichen einen relativ hohen Aufwand. Außerdem müssen auch die Daten des Textfeldes in dem verwendeten Zeichencode bereitgestellt werden. Man bezeichnet diese Art der Datendarstellung, bei der bestimmte Datenbitmuster nicht vorkommen dürfen, weil sie als Steuerzeichen interpretiert werden, als *nichttransparent*. Für eine *transparente Datendarstellung* sind zusätzliche Maßnahmen erforderlich. Beim BISYNC-Protokoll werden dazu die Steuerzeichen für die Datenblockbegrenzung, STX und ETX, durch ein vorangestelltes *DLE-Zeichen* gekennzeichnet. Damit der Empfänger eine im Datenblock zufällig vorkommende Zeichenkombination DLE ETX nicht fälschlicherweise als Ende des Datenblocks interpretiert, ergänzt der Sender jedes Bitmuster im Datenblock, das mit dem DLE-Zeichen identisch ist, durch ein weiteres DLE-Zeichen (*character stuffing*). Dies wird vom Empfänger erkannt, der dieses Zeichen wieder entfernt (siehe z.B. [Tanenbaum 2003]).

Anmerkung. Mit dieser Technik kann auch das in 7.5.2 beschriebene X-On-/X-Off-Protokoll der asynchron-seriellen Datenübertragung transparent gemacht werden, was dann erforderlich ist, wenn auf dem Rückkanal außer den beiden Steuerzeichen DC1 und DC3 auch reguläre Daten übertragen werden. Die beiden Steuerzeichen müssen dann ebenfalls durch DLE-Zeichen als solche gekennzeichnet werden.

Ein weiterer Nachteil zeichenorientierter Protokolle ist die Handhabung vieler unterschiedlicher Übertragungsformate für Daten, Steuer- und Quittungsinformation, wie in Bild 7-37 zu erkennen ist. Die genannten Nachteile entfallen bei den bitorientierten Protokollen.

SDLC- und HDLC-Protokoll. Ein *bitorientiertes Protokoll* hat ein einheitliches Übertragungsformat, bei dem die einzelnen Felder durch ihre Bitpositionen im Übertragungsblock festgelegt sind. Diese Festlegung hat neben der einfacheren Interpretation den Vorteil einer *transparenten Datendarstellung*, d.h., als Daten können beliebige Bitmuster, z.B. BCD-Zeichen in gepackter Darstellung, Steuer- oder Textzeichen eines beliebigen Codes, der Maschinencode eines Programms oder eine zeichenunabhängige Bitfolge übertragen werden.

Bild 7-38 zeigt als Beispiel das Übertragungsformat für das *SDLC-Protokoll* (synchronous data-link control, IBM) [Weissberger 1979]. Der eigentliche Übertragungsblock wird durch zwei sog. Flag-Bytes (FLAG) mit dem Code 01111110 eingerahmt. Sie dienen zur Blockbegrenzung und zur *Zeichensynchronisation*. Auf das erste Flag-Byte folgt der Kopf (header) mit einem 8-Bit-Adressfeld (ADDRESS) und einem 8-Bit-Steuerfeld (CONTROL). Durch die Angabe einer Adresse kann ein Übertragungsblock gleichzeitig mehreren Empfängern angeboten werden (*broadcasting*). Jeder der angesprochenen Empfänger kann dann anhand der Adresse feststellen, ob der Block für ihn bestimmt ist oder nicht. Das Steuerfeld kennzeichnet den Inhalt des Übertragungsblocks als Daten oder

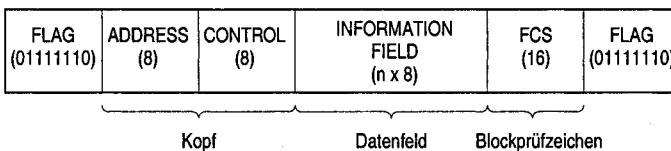


Bild 7-38. Übertragungsformat des SDLC-Protokolls

Steuerinformation, gibt Auskunft über die Anzahl der übertragenen bzw. der fehlerfrei empfangenen Datenblöcke und enthält Steuerkommandos. Auf den Kopf folgt das eigentliche Datenfeld (INFORMATION FIELD), das beim SDLC-Protokoll ein Vielfaches von 8-Bit-Informationseinheiten umfaßt. Abgeschlossen wird der Block durch zwei *Blockprüfbytes* FCS (frame checking sequence, siehe auch 7.7.3), gefolgt von einem abschließenden Flag-Byte. Eine Lücke zwischen zwei Übertragungsböcken wird durch zusätzliche Flag-Bytes gefüllt; sie erhalten den Arbeitszustand der Verbindung aufrecht.

Aus dem SDLC-Protokoll abgeleitet wurde das *HDLC-Protokoll* (high-level data-link control) der ISO. Seine Adress- und Steuerfelder können auf mehrere Bytes erweitert werden; sein Datenfeld lässt eine beliebige Anzahl von Datenbits zu und ist damit nicht mehr an die 8-Bit-Einheiten des SDLC-Protokolls gebunden [Weissberger 1979].

Bei den beiden genannten bitorientierten Protokollen gibt es lediglich drei verschiedene Bitmuster bestimmter Funktion, die vom Empfänger erkannt werden müssen: das Flag-Byte 01111110 und die Bitfolgen ABORT 01111111... (7 bis 14 Einsen) und IDLE 11111111111111... (15 oder mehr Einsen). ABORT beendet die Übertragung eines Blocks vorzeitig, IDLE zeigt den Ruhezustand einer Verbindung an. Durch die Zusammensetzung dieser drei Bitmuster aus aufeinanderfolgenden Einsen lässt sich die *Transparenz* der Informationsdarstellung in einfacher Weise herstellen. Der Sender fügt dazu innerhalb der Nutzinformation nach jeweils fünf aufeinanderfolgenden Einsen ein Null-Bit in den Bitstrom ein, so daß keine Bitfolgen auftreten können, die der Empfänger als eines der drei genannten Bitmuster interpretieren würde (*bit stuffing*). Der Empfänger wiederum

zählt die aufeinanderfolgenden Einsen und entfernt jeweils das auf fünf Einsen folgende Bit, sofern es den Wert Null hat. Hat es den Wert Eins, so handelt es sich um eines der Bitmuster FLAG, ABORT oder IDLE, und das entsprechende Bitmuster ist erkannt. Das Einfügen und Entfernen der Null-Bits wird vollständig von der Hardware durchgeführt (siehe z.B. [Tanenbaum 2003]).

Verbindet man das *Null-Einfügen* z.B. mit der *NRZI-Signalcodierung* (non return to zero with interchange), so können die Signalübergänge im Bitstrom zur Synchronisation des Empfängertaktgenerators benutzt werden. Das ist ein weiterer Vorteil bitorientierter Protokolle. Bei der NRZI-Codierung wird eine Eins durch gleichbleibende Polarität und eine Null durch einen Wechsel der Polarität des Signals dargestellt (siehe auch Bild 7-44, S. 529). Durch das Null-Einfügen erfolgen solche Wechsel spätestens nach jedem fünften Bit.

7.6.3 Synchron-serieller Interface-Baustein

Im folgenden wird ein synchron-serieller Interface-Baustein beschrieben, der die Datenübertragung für bitorientierte Protokolle, wie das SDLC- und das HDLC-Protokoll, unterstützt. Er übernimmt die Parallel-Serien- und die Serien-Parallel-Umsetzung, das Erzeugen und Erkennen der Bitmuster FLAG, IDLE und ABORT, das Null-Einfügen und -Entfernen sowie die Blocksicherung durch zwei Blockprüfbytes. Die Beschreibung ist relativ grob, da die Details dieses Bausteins über den Rahmen dieses Buches hinausgehen. Deshalb sei auf die Datenblätter der Bausteinhersteller verwiesen.

Blockstruktur. Bild 7-39 zeigt die Struktur des Bausteins. Er hat einen Sende- und einen Empfangsteil, die beide mit speziellen Einrichtungen für die oben genannten Aufgaben zur Realisierung bitorientierter Protokolle ausgestattet sind. Ein zusätzliches Übertragungssteuerwerk dient dem Austausch von Steuersignalen mit der Peripherie. Weiterhin besitzt er eine Unterbrechungseinrichtung mit Selbstidentifizierung und eine Verkettungslogik zur Priorisierung, die bei einfacheren Baustinausführungen fehlen können. – Die Schnittstellen zum Systembus und zur Peripherie sind mit denen des asynchron-seriellen Bausteins identisch (7.5.3).

Funktionsweise. Zur Aufrechterhaltung eines lückenlosen Datenbitstroms haben Sende- und Empfangsteil je einen Pufferspeicher mit drei 8-Bit-Datenregistern TDR1 bis TDR3 (transmit data registers) bzw. RDR1 bis RDR3 (receive data registers), die nach dem Warteschlangenprinzip verwaltet werden: Das zuerst in den Pufferspeicher eingegebene Byte wird als erstes wieder ausgegeben (*first-in first-out, FIFO*). Bei der Datenausgabe zum Beispiel übernimmt das 8-Bit-Schieberegister TSR (transmit shift register) des Sendeteils das im Datenregister TDR3 stehende Byte und übergibt es bitweise an die Sendedatenleitung TxD (Parallel-Serien-Umsetzung). Hierbei rücken die Inhalte der Datenregister TDR2 und

TDR1 in die jeweils nächsten Register vor. (Entsprechend rückt auch ein vom Prozessor in das Datenregister TDR1 geschriebenes Byte in das vorderste freie Datenregister vor.)

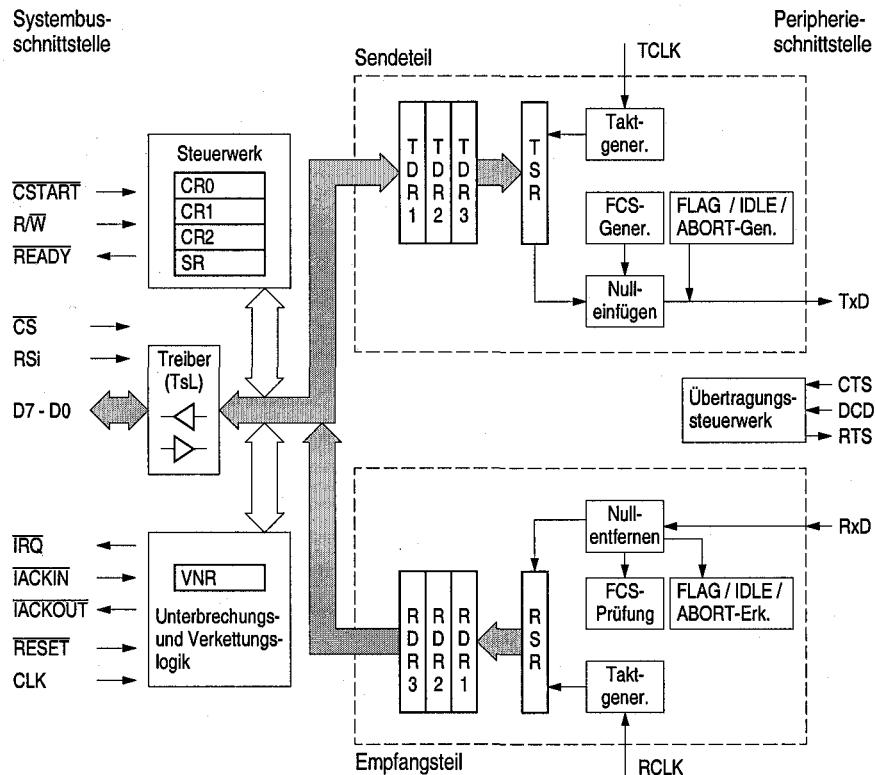


Bild 7-39. Struktur eines synchron-seriellen Interface-Bausteins für bitorientierte Protokolle mit einem Sende- und einem Empfangsteil

Vor Beginn des ersten Ausgabevorgangs wird der Sendeteil entweder durch das RESET-Signal (power-on reset) oder durch Setzen eines Reset-Steuerbits initialisiert und dann durch Laden seiner Steuerregister initialisiert. Eines der Steuerbits entscheidet dabei, ob nach Rücksetzen des Reset-Bits die Bitfolge IDLE (inactive idle) oder ob Flag-Bytes (active idle) gesendet werden. In beiden Fällen beginnt die eigentliche Datenübertragung, sobald das TDR-FIFO vom Prozessor mit Datenbytes geladen ist. Bei „inactive idle“ wird, um den Datenrahmen zu öffnen, vom Sendeteil vor Übertragen des ersten Datenbytes ein Flag-Byte in den Bitstrom eingefügt; bei „active idle“ ist dies nicht erforderlich. Um Beginn und Fortsetzung vom Ende des Datenblocks für den Sendeteil unterscheidbar zu machen, stehen für das Laden des TDR-FIFO zwei verschiedene Adressen mit den unterschiedlichen Attributen „frame continue“ und „frame terminate“ zur Verfügung. Bei „frame terminate“ wird im Unterschied zu „frame continue“ nach Senden des

Bytes ein abschließendes Flag-Byte gesendet, womit der Datenrahmen geschlossen wird.

Der Empfangsteil wird ebenfalls während des Reset-Zustands initialisiert und mit dem Rücksetzen des Reset-Steuerbits aktiviert. Im Aktivzustand untersucht er den über die Empfangsdatenleitung RxD eintreffenden Bitstrom nach den Bitmustern IDLE, FLAG und ABORT. Die auf ein Flag-Byte folgenden Bytes, die selbst keine Flag-Bytes und auch nicht Teil eines Idle- oder Abort-Bitmusters sind, überträgt er von seinem 8-Bit-Schieberegister RSR (receive shift register) in das Eingabedatenregister RDR1 (Serien-Parallel-Umsetzung). Nach dem FIFO-Prinzip rücken diese Bytes bis zum Datenregister RDR3 vor, von wo sie vom Prozessor gelesen werden können. Diese Datenübernahme wird mit dem Erkennen des nächsten Flag-Bytes beendet.

7.7 Rechnernetze und Datenfernübertragung

Datenübertragung beschränkt sich nicht allein auf die in den vorangegangenen Abschnitten beschriebenen Ein-/Ausgabevorgänge zwischen dem Prozessor und seinen Peripheriegeräten. Vielmehr sind Rechner heute in großem Maße in Rechnernetzen miteinander verbunden, mit der Möglichkeit der Kommunikation zwischen Rechnern wie auch zwischen deren Benutzern. Gegenüber dem isolierten Einsatz von Rechnern reicht das Spektrum der zusätzlichen Möglichkeiten vom „Resource-Sharing“, d.h. der gemeinsamen Benutzung von Geräten (z.B. Laserdrucker, Scanner, File-Server) und von Dateien (z.B. Compiler, Dokumente, Datenbanken), bis hin zum weltweiten Informationsaustausch z.B. durch E-Mail oder über das World Wide Web unter Nutzung des Internets.

Gegenüber einem Großrechner mit Mehrbenutzerbetrieb haben kleinere, in einem Netz verbundene Rechner den Vorteil des günstigeren Preis-Leistungs-Verhältnisses. Darüber hinaus bieten solche Netzlösungen eine hohe Systemverfügbarkeit. So kann ein Benutzer auf unterschiedlichen Rechnern im Netz arbeiten, d.h., bei Ausfall eines Rechners ist ein Ausweichen auf einen anderen Rechner möglich (sofern es nicht den Rechner betrifft, der den Netzzugang für den Benutzer herstellt). Dateien werden ggf. vervielfacht auf mehreren Hintergrundspeichern gehalten (Spiegelung von Dateien), so daß sie auch bei Ausfall einer Speichereinheit weiter verfügbar sind. Ein weiterer Vorteil ist die große Flexibilität im Konfigurieren von Netzen und die Möglichkeit, Rechnernetze wiederum untereinander zu verbinden. Ein Nachteil von Netzen ist jedoch, daß Fehler, die in der Netzsoftware oder im Übertragungssystem auftreten, sich global auf alle Netzteilnehmer auswirken können.

Der Stoff zu diesem Thema ist so umfangreich, daß er im Rahmen dieses Buches nur angeschnitten werden kann. Als Literatur, an der sich auch die nachfolgenden hardware-nahen Ausführungen orientieren, sei [Tanenbaum 2003] empfohlen.

7.7.1 Weitverkehrsnetze und lokale Netze

Ein *Rechnernetz* besteht aus sog. Anwendungsrechnern (Wirtsrechnern, hosts) als Netzteilnehmer sowie einem Übertragungssystem (subnet, communication subnet), über das sie miteinander kommunizieren. Die Leistungsfähigkeit der Rechner reicht dabei vom Personal-Computer über Workstations bis hin zu Großrechnern. Im einfachsten Fall kann der Netzzugang auch über ein Terminal erfolgen. Grundsätzlich unterscheidet man zwischen *lokalen Netzen* (inhouse nets, local area networks, LANs), *Weitverkehrsnetzen* (Telekommunikationsnetze, wide area networks, WANs) und *globalen Netzen* (global area networks, GANs). LANs sind Verbunde mit geringer räumlicher Ausdehnung meist innerhalb von Gebäuden oder Grundstücken, die von einzelnen Nutzern (z.B. Firmen, wissenschaftlichen Einrichtungen) betrieben werden. WANs haben überregionale, d.h. landesweite Ausdehnung (üblicherweise flächendeckend). Bei ihnen unterliegen die Anwendungsrechner zwar meist privaten Nutzern, jedoch werden für die Datenübertragung die Übertragungseinrichtungen der Fernmeldeinstitutionen eines Landes benutzt. GANs haben mittels Überseekabel und Satelliten globale Reichweiten und stehen üblicherweise im Verbund mit den regionalen WANs. Als Zwischenform von LAN und WAN gibt es die sog. *Metronetze* (metropolitan area networks, MANs). Sie sind Stadtnetze (Regionalnetze) und wirken innerhalb oder zwischen Ballungsräumen, oft als Hochgeschwindigkeitsnetze.

Strukturen von Weitverkehrsnetzen. Bei *Weitverkehrsnetzen* besteht das Übertragungssystem aus Übertragungsleitungen und Vermittlungseinrichtungen, sog. *Knotenrechnern* (Vermittlungsrechner, interface message processor, IMP), die für die Durchschaltung der Übertragungsleitungen zuständig sind (Bild 7-40). Die Datenübertragung zwischen zwei Anwendungsrechnern erfolgt jeweils über die

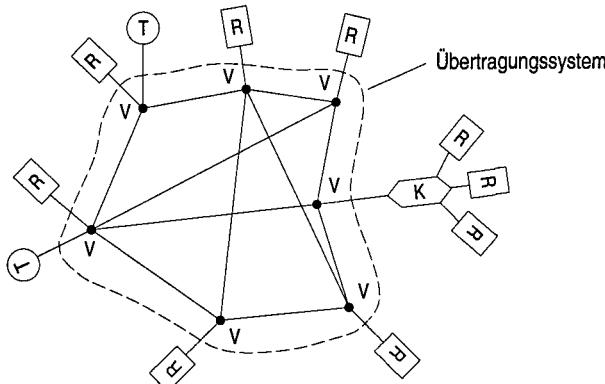


Bild 7-40. Punkt-zu-Punkt-Netzstruktur eines WAN. R Anwendungsrechner (Host), V Vermittlungsrechner (Knotenrechner), T Terminal, K Konzentrator. Durch einen Konzentrator werden die Datenpakete der Anwendungsrechner in der Reihenfolge ihres Eintreffens im Multiplex-Verfahren weitergegeben

beiden ihnen zugeordneten Knotenrechner. Sind diese Knotenrechner nicht direkt miteinander verbunden, so müssen die Daten auf dem Umweg über andere Knotenrechner transportiert werden. Meist erfolgt dieser Transport in Datenpaketen, die in den Knotenrechnern zunächst zwischengespeichert und dann weitergeleitet werden (*store-and-forward network*). Zusammengehörige Pakete können dabei unterschiedliche Wege durchlaufen, je nachdem welche Verbindungsleitungen zwischen den Knoten des Netzes jeweils verfügbar sind (*Wegewahl, routing*). Man bezeichnet diese Netze auch als *paketvermittelnde Netze* (packet switching subnet, connectionless). Wird für die Dauer der Datenübertragung ein Verbindungsweg fest durch geschaltet (so wie beim Telefonieren), so spricht man von *leitungsvermittelnden Netzen* (connection-oriented). Insgesamt bezeichnet man diese knotenorientierten Netze auch als *Punkt-zu-Punkt-Netze*. – Typische Strukturen bei Punkt-zu-Punkt-Netzen sind der Stern, der Ring und der Baum sowie die vollständige und die unregelmäßige Verbindung von Knotenrechnern. Letztere ist typisch für Weitverkehrsnetze, basierend auf den Übertragungseinrichtungen von Fernmeldeinstitutionen.

Den Punkt-zu-Punkt-Netzen stehen die sog. *Broadcast-Netze* gegenüber. Bei ihnen werden die Datenpakete vom Sender gleichzeitig an mehrere oder an alle Netzteilnehmer übertragen, und der eigentliche Empfänger erkennt an der mitgelieferten Adressinformation, daß die Nachricht für ihn bestimmt ist. Bei Weitverkehrsnetzen gibt es sie in Form der Funkübertragung (z.B. über Satellit), bei der mehrere Empfangsstationen im Sendebereich eines Senders liegen. Anstelle von Übertragungsleitungen stehen hier Funkstrecken.

Anmerkung. Die Begriffe host, subnet, communication subnet und IMP entstammen dem ersten bedeutenden Weitverkehrsnetz, dem Arpanet in den USA. Dessen Ursprung geht auf ein Testnetz zurück, das aus nur drei Knotenrechnern bestand und mit dem 1969 die erste E-Mail von Palo Alto nach Los Angeles geschickt wurde, und zwar auf dem Umweg über die University of Utah.

Strukturen von lokalen Netzen. *Lokale Netze* sind in ihrem Ursprung Broadcast-Netze mit Bus- oder Ringstruktur, die im Gegensatz zu Weitverkehrsnetzen keine Knotenrechner enthalten. Statt dessen hat jeder Anwendungsrechner eine Netzsteuereinheit (network controller) in Form einer Interface-Karte (Netzkarte, siehe *LAN-Controller* in Bild 5-4, S. 266), und das Übertragungssystem besteht im einfachsten Fall (Bus) aus nur einem Kabel. Davon abweichend haben heute Stern- und Baumstrukturen eine große Verbreitung (strukturierte Verkabelung). Hier kommen dann Vermittlungseinheiten zum Einsatz, und die Übertragung erfolgt Punkt-zu-Punkt. – Zu beachten ist, daß die logische Struktur eines LAN von der physikalischen Struktur durchaus verschieden sein kann. Dazu seien im folgenden die typischen physikalischen *Netzstrukturen* (*Netztopologien*) aufgeführt.

Ring. Bei der *Ringstruktur (Token-Ring)* sind alle Teilnehmer Punkt-zu-Punkt in einem Ring miteinander verbunden. Die Kommunikation erfolgt in fest vorgegebener Umlaufrichtung, wobei freie Übertragungskapazitäten durch eine oder mehrere umlaufende Marken (tokens) gekennzeichnet werden (Bild 7-41a). Der

sendende Teilnehmer übergibt einem freien *Token* seine Nachricht zusammen mit seiner Adresse und der des Empfängers. Der sich mit der Empfängeradresse assoziierende Netzteilnehmer übernimmt die Nachricht und übergibt dem Token eine Empfangsbestätigung für den Sender, nach deren Empfang dieser das Token wieder freigibt. Die Ringstruktur erfordert einen geringen Aufwand, hat jedoch den Nachteil, daß bei einer Unterbrechung im Ring bei einem der Ringteilnehmer der gesamte Ring funktionsunfähig wird.

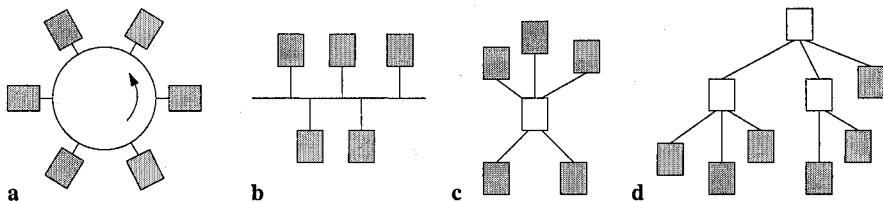


Bild 7-41. LAN-Strukturen. a Ring, b Bus, c Stern, d Baum

Ein Beispiel für die Ringstruktur ist der Token-Ring von IBM mit Übertragungsraten von 4 oder 16 Mbit/s (IEEE 802.5). Ein anderer Token-Ring, das *FDDI* (Fiber Distributed Data Interface), vermeidet den oben geschilderten Nachteil durch Verdoppelung des Rings (ANSI X3T9.5). Die maximale Länge für den „primären“ wie auch für den „sekundären“ Ring beträgt hier 100 km, die Übertragungsrate 100 Mbit/s. FDDI gilt als Hochgeschwindigkeits-LAN und wird insbesondere auch als sog. *Backbone-Netz* eingesetzt, um LANs miteinander zu verbinden (siehe auch Bild 7-42, S. 525). Es benutzt Lichtwellenleiter oder Twisted-Pair-Kabel als Übertragungsmedien.

Bus. Bei der *Busstruktur* kommunizieren die Teilnehmer miteinander über einen Bus als passiven „Knoten“ (Bild 7-41b). Ein solcher Bus wird z.B. durch ein Koaxialkabel gebildet, an das die Netzteilnehmer in Abständen angeschlossen werden. Das Anschließen kann während des Betriebs erfolgen; der Ausfall eines Teilnehmers beeinträchtigt die Funktionsfähigkeit des Netzes nicht. Die Möglichkeit des gleichzeitigen Zugriffs auf den Bus erfordert jedoch eine Strategie zur *Vermeidung von Kollisionen*. So muß – ähnlich der Busarbitration – ein Sender vor Sendebeginn anhand des Signalpegels auf dem Bus zunächst überprüfen, ob der Bus frei ist; und er muß mit Sendebeginn erneut prüfen, ob nicht gleichzeitig ein zweiter Sender mit Senden begonnen hat. Im Konfliktfall muß er seine Übertragung aufschieben bzw. abbrechen, um sie z.B. nach einer zufallsgesteuerten Wartezeit erneut zu versuchen. Eines von mehreren Verfahren mit dieser Organisationsform ist das *CSMA/CD-Verfahren* (carrier sense multiple access/collision detection), wie es bei dem weit verbreiteten *Ethernet* von Xerox mit einer Übertragungsrate von ursprünglich 10 Mbit/s angewandt wird (IEEE 802.3). Weiterentwicklungen dieses Netzes, die als IEEE-Standards vorliegen, ermöglichen Übertragungsraten von 100 Mbit/s (*Fast-Ethernet*), 1 Gbit/s (*Gigabit-Ethernet*) und 10 Gbit/s (*10-Gigabit-Ethernet*). Vorbereitet wird ein Ethernet mit 40 Gbit/s.

Als physikalische Realisierungen dieser Varianten sind jedoch Stern- und Baumstrukturen üblich, und zwar in Verbindung mit Hubs und vor allem Switches sowie den heute gebräuchlichen Kupferkabeln mit verdrillten Adernpaaren (anstelle von Koaxialkabeln) oder Glasfaserkabeln (für größere Entfernung). Die logische Busstruktur bleibt davon unbeeinflußt.

Eine andere Organisationsform als das Ethernet hat der *Token-Bus*, bei dem wie beim Token-Ring ein Token zyklisch von Teilnehmer zu Teilnehmer weitergereicht wird, womit sich eine feste Zuteilung der Sendeberechtigungen ergibt (IEEE-Standard 802.4). Ein Produktbeispiel hierfür ist das *MAP* (manufacturing automation protocol) mit Übertragungsraten von 1, 5 oder 10 Mbit/s. Auch ein Token-Bus kann physikalisch eine Sternstruktur haben, seine logische Struktur ist aber immer ein Ring.

Stern und Baum. Die *Sternstruktur* ist die klassische Struktur, wie sie bei Großrechnern mit daran angeschlossenen Terminals vorzufinden ist (Bild 7-41c). Hier beziehen die Teilnehmer Rechenleistung von einer zentralen Rechenanlage, üblicherweise nach dem Zeitscheibenverfahren (time-sharing/slicing). Das Hauptvorkommen der Sternstruktur liegt jedoch in Netzen, die mit sog. *Hubs* und *Switches* als Sternverteiler gebildet werden. Diese Technik birgt zwar eine größere Ausfallgefahr des gesamten Netzes, erleichtert aber den Netzaufbau und die Netzerweiterung wesentlich. Sie wird deshalb z.B. bei heutigen Ethernet-LANs bevorzugt eingesetzt. Ein Switch erlaubt es darüber hinaus, z.B. ein mit ihm gebildetes langsames lokales Netz mit einem schnellen Backbone-Netz zu verbinden (z.B. auf der Basis zweier unterschiedlicher Ethernet-Standards). Die *Baumstruktur* (Bild 7-41d) entsteht durch Zusammenschluß von Sternstrukturen.

Netzverbinder. Wie die oben angedeutete Switch-Anwendung zeigt, lassen sich Netze, gleiche oder unterschiedliche, miteinander verbinden, eine Technik, auf der letztlich die weltweite Vernetzung basiert. Hierfür werden Netzverbinder eingesetzt, die sich abhängig davon, in welcher Schicht des *OSI-Referenzmodells* (siehe S. 530) die Netzkopplung stattfindet, in ihrer Funktionalität unterscheiden. Herkömmliche Verbinder sind Repeater, Bridge, Router und Gateway, die verstärkt durch Hubs und Switches abgelöst werden.

- Ein *Repeater* verbindet zwei gleichartige Netze in der untersten Schicht 1, z.B. zwei räumlich auseinanderliegende Segmente eines 10-Mbit-Ethernets (Bild 7-42). Er wirkt als Signalverstärker und führt eine reine Bitübertragung durch, d.h., er nimmt keine Protokollumsetzung vor.
- Eine *Bridge* ist ein Netzverbinder in der Schicht 2. Sie verbindet zwei gleichartige Netzsegmente oder zwei unterschiedliche Netze mit jedoch gleicher Realisierung in der Schicht 1, z.B. ein Ethernet mit einem FDDI-Backbone (Bild 7-42). Dabei entscheidet sie anhand der den Daten mitgegebenen Adressinformation, was sie weiterleitet, hat also Filterfunktion.

- Ein *Router* stellt eine Netzverbindung in der Schicht 3 her und führt das Routing für die Datenpakete, d.h. die Wegewahl im Netz durch (Bild 7-42). Er wertet dazu die den Daten mitgegebene Adressinformation aus.
- Ein *Gateway* schließlich stellt eine Netzverbindung auf der jeweils obersten Netzebene her. Es ist erforderlich, wenn die Netze nicht dem Referenzmodell entsprechen und es somit keine gemeinsame Schicht gibt (Bild 7-42). Dementsprechend führt ein Gateway neben der Routing-Funktion auch Protokoll- und Codeumwandlungen durch.
- Ein *Hub* (Schaltstern) ist eine Verteilereinrichtung in der Schicht 1 mit einer sternförmigen Anordnung der Anschlüsse (ports). Datenpakete, die an einem der Anschlüsse eingehen, werden unbesehen an alle anderen Anschlüsse weitergeleitet, was aus logischer Sicht der Busstruktur entspricht. Dementsprechend besteht auch hier wieder das Problem der Kollision, das z.B. bei Ethernet-Realisierungen durch das oben erwähnte *CSMA/CD-Verfahren* gelöst wird. Außerdem müssen sich die Netzteilnehmer in bustypischer Weise die verfügbare Übertragungskapazität teilen. Hubs haben entweder Verstärkerfunktion (aktiver Hub) oder nicht (passiver Hub). Erstere wirken wie Repeater. Hubs eignen sich nicht nur zur Realisierung der logischen Busstruktur, sondern auch der logischen Ringstruktur. – Hubs finden auch außerhalb von Rechnernetzen Verwendung, so z.B. beim Universal Serial Bus für den Anschluß lokaler Peripheriegeräte an einen Rechner (siehe 8.2.7).

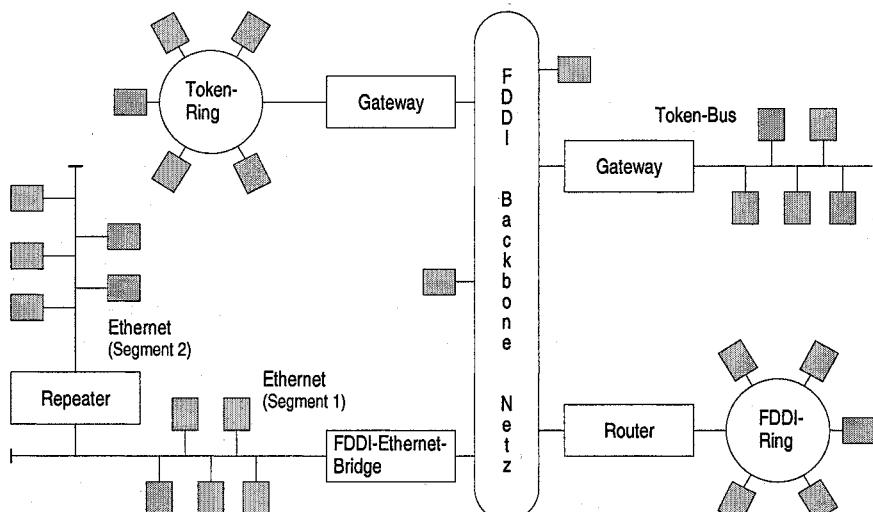


Bild 7-42. Beispiel für den Einsatz herkömmlicher Netzverbinder: Kopplung verschiedener LANs mittels eines FDDI-Backbone-Netzes durch Bridge, Router und Gateway. Kopplung zweier mit Koaxialkabeln gebildeter Ethernet-Segmente durch einen Repeater (in Anlehnung an [Dembowski 1997])

- Ein *Switch* (Schaltmatrix) ist ähnlich einem Hub eine zentrale Verteilereinrichtung. Im Unterschied zum Hub werden jedoch von ihm eingehende Datenpakete gezielt, d.h. nur an den eigentlichen Empfänger, weitergeleitet. Er wertet dazu die im Header eines Pakets stehende Zieladresse aus und stellt für die Zeit der Übertragung eine *Punkt-zu-Punkt-Verbindung* zwischen Sender und Empfänger her. Mit dieser Funktionalität ist er wie die Bridge der Schicht 2 zuzuordnen (Layer-2-Switch), hat jedoch anders als diese mehr als nur zwei Anschlüsse. Ein Switch kann um die Funktionalität des Routing erweitert sein und ist dann der Schicht 3 zuzuordnen (Layer-3-Switch, Routing-Switch).

Switches sind üblicherweise modular aufgebaut, d.h., ihre „Anschlüsse“ sind als Steckkarten nach Bedarf wählbar. So können z.B. Ethernet-Anschlüsse mit unterschiedlichen Übertragungsraten und Kabelarten miteinander kombiniert werden, aber auch unterschiedliche Netze, wie Ethernet mit ATM oder FDDI. Mittels eines switch-internen Busses mit höherer Übertragungskapazität als die des einzelnen Anschlusses oder mittels eines Kreuzschienenverteilers (*crossbar switch*) können mehrere Punkt-zu-Punkt-Verbindungen gleichzeitig (virtuell im ersten bzw. physikalisch im zweiten Fall) hergestellt werden, im Extremfall alle Anschlüsse des Switch mit einbeziehend. Gegenüber einem Hub ergibt sich dadurch der Vorteil, daß sich die Anschlüsse die verfügbare Übertragungskapazität nicht teilen müssen, sondern jedem Anschluß die volle Kapazität zur Verfügung steht. Datenpakete werden dabei ggf. zwischengespeichert. Hierdurch kann insbesondere auch die hohe Übertragungskapazität eines einzelnen Anschlusses (z.B. Ethernet mit 1 Gbit/s) auf andere Anschlüsse geringerer Kapazität (z.B. Ethernet mit 100 oder 10 Mbit/s) verteilt werden.

Switches sind Bestandteile moderner Netze im Sinne einer strukturierten Verkabelung. Sie stehen für eine hohe Flexibilität in der Netzkonfigurierung, indem sie diese dynamisch zu gestalten ermöglichen. So können z.B., ausgehend von einem Layer-3-Switch, einzelne Anschlüsse nachgeordneter Layer-2-Switches individuell bestimmten Subnetzen oder auch mehreren Subnetzen gleichzeitig zugeordnet werden. Man spricht hier von *virtuellen LANs (VLANs)*. Bild 7-43 zeigt dazu ein Beispiel einer Gebäudevernetzung, bestehend aus diversen Subnetzen.

Übertragungsmedien. Als Übertragungsmedien werden in Rechnernetzen *Kupferkabel* in Form von verdrillten Leitungspaaren (*twisted pair*) und von *Koaxialkabeln* sowie Lichtwellenleiter in Form von Glasfasern eingesetzt. *Twisted-Pair-Kabel* gibt es nicht abgeschirmt, wie sie als Telefonkabel bekannt sind, insbesondere aber abgeschirmt. Sie sind kostengünstig und leicht zu verlegen. Sie bieten jedoch keine so große Störsicherheit und haben schlechtere Hochfrequenzeigenschaften als Lichtwellenleiter, weshalb sie bei räumlich kleineren Netzen (Segmentlängen bis zu 100 m) und mit begrenzten Übertragungsraten (bis zu 1 Gbit/s) zum Einsatz kommen. Die diesen gegenüber teureren und schlechter zu verlegen-

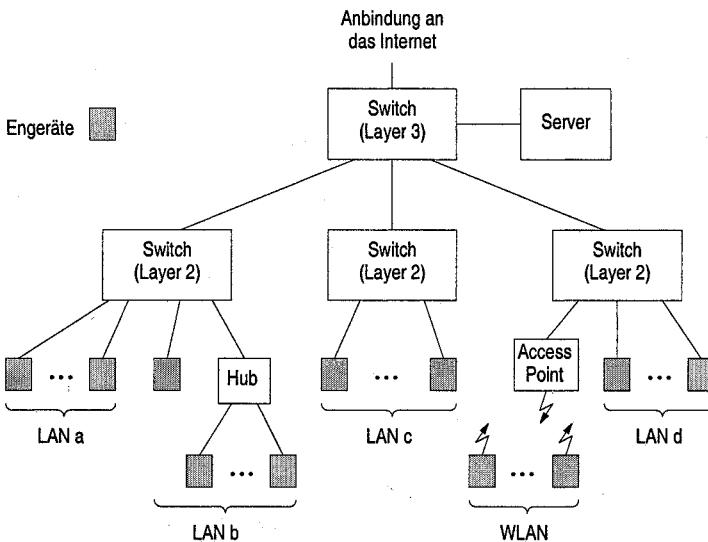


Bild 7-43. Beispiel für eine Gebäudevernetzung mit Switches und Hubs: Zentraler Layer-3-Switch (Routing-Switch) mit Internet-Anbindung und nachgeschalteten Layer-2-Switches zur Versorgung der Stockwerke mit Subnetzen (LANs) unter Einbeziehung eines Hubs zur LAN-Erweiterung und eines Funknetzes (WLAN, siehe S. 529). Hohe Übertragungsraten zum Internet, zwischen den Switches und zum Server (z.B. Ethernet mit 1 Gbit/s), geringere Übertragungsraten zwischen den Layer-2-Switches und den Endgeräten der Subnetze (z.B. Ethernet mit 100 und 10 Mbit/s)

den *Koaxialkabel* bieten aufgrund ihres Aufbaus eine sehr viel höhere Störsicherheit und auch bessere Hochfrequenzeigenschaften. Man unterscheidet hier das Schmalbandkabel (50Ω , digitale Übertragung) und das aus der Fernsehtechnik stammende Breitbandkabel (75Ω , analoge Übertragung). Das $75\text{-}\Omega$ -Kabel und später das $50\text{-}\Omega$ -Kabel wurden lange Zeit bevorzugt für lokale Netze eingesetzt, so z.B. für das 10-Mbit-Ethernet. Heute sind die Koaxialkabel im Rahmen der Hubs und Switches jedoch weitgehend durch Kupferkabel abgelöst, auch bei den „schnelleren“ Ethernet-Standards bis 1 Gbit/s.

Bei *Lichtwellenleitern* erfolgt die Übertragung durch Lichtimpulse, die sich mittels der Totalreflexion im flachen Winkel durch die Glasfaser fortbewegen. Diese Übertragung ist weitgehend ungestört, da sie z.B. durch elektromagnetische Felder nicht beeinflusst wird. Mit ihnen sind hohe Übertragungsraten im Gbit/s-Bereich möglich, und es lassen sich große Entfernungen von bis zu mehreren hundert Kilometern überbrücken. Die dann erforderliche Signalverstärkung wird z.B. mit Glasfaserzwischenstrecken mit bestimmter Dotierung des Materials erreicht. Durch Wellenlängen-Multiplexen, d.h. durch das gleichzeitige Übertragen von Lichtsignalen mit unterschiedlichen Wellenlängen (Lichtfarben), kann die Übertragung in mehrere Kanäle unterteilt werden, was einer Erhöhung der Übertragungskapazität gleichkommt. Glasfaserkabel lassen sich zwar relativ leicht

verlegen, sie erfordern jedoch besondere Techniken beim Verbinden (Schweißen, Kleben). Die Umwandlung der elektrischen Datensignale in Lichtimpulse erfolgt durch LEDs (light emitting diodes) oder Laserdioden, die Rückwandlung durch geeignete Fotodioden. Lichtwellenleiter sind bevorzugtes Medium für die Hochgeschwindigkeitsübertragung über große Entfernung (Datenautobahn, Telekommunikation), werden aber zunehmend auch im Bereich lokaler Netze eingesetzt, wenn dort hohe Übertragungsraten gefordert sind, z.B. bei Multimedia-Anwendungen.

Signaldarstellung und -codierung. Die Datenübertragung in Rechnernetzen erfolgt generell seriell (synchron oder asynchron). Bezüglich der Signaldarstellung und -codierung unterscheidet man zwei grundsätzliche Vorgehensweisen: das Schmalband- oder Basisbandverfahren (baseband) und das Breitbandverfahren (broadband). Beim *Basisbandverfahren* wird das Datensignal in digitaler Form und sozusagen in seiner ursprünglichen Frequenz, d.h. unmoduliert übertragen. Die hierbei verwendeten Übertragungsmedien sind dementsprechend schmalbandig, z.B. Twisted-Pair-Kabel und Schmalband-Koaxialkabel. Die meisten lokalen Netze basieren auf diesem Verfahren. Beim *Breitbandverfahren* wird das Datensignal einem hochfrequenten Träger aufmoduliert, d.h., die Übertragung erfolgt analog. Auf diese Weise erreicht man sehr hohe Übertragungsraten. Darüber hinaus kann eine Leitung unter Verwendung von mehreren Trägern unterschiedlicher Frequenzen in mehrere Übertragungskanäle unterteilt werden. Nachteilig ist der relativ hohe technische Aufwand, wie er aus der (analogen) Fernsehtechnik bekannt ist, z.B. bei den Analogverstärkern. Diese wirken außerdem immer nur in einer Richtung, was den Aufwand beim Duplexbetrieb verdoppelt. Als Übertragungsmedien werden Breitband-Koaxialkabel und insbesondere Lichtwellenleiter eingesetzt. Der Einsatz dieser Technik liegt bei *Hochgeschwindigkeitsnetzen*, z.B. MANs oder schnellen LANs, und generell bei der Hochgeschwindigkeits-Datenfernübertragung, z.B. bei den sog. Datenautobahnen.

Bild 7-44 zeigt einige gebräuchliche *Signalcodierungen* für das *Basisbandverfahren* am Beispiel einer Bitfolge und in Bezug auf das der Übertragung im Sender und Empfänger zugrundeliegende Takt signal. Wichtig ist hierbei die Möglichkeit, bei synchroner Übertragung das Takt signal im Empfänger aus dem Datensignal wiedergewinnen zu können. Der Empfänger besitzt dazu einen Taktgenerator, der sich mit dem Datensignal fortlaufend synchronisiert (siehe auch 7.7.1). Hierfür benutzt er die Pegelübergänge des Signals, die dazu in bestimmten Mindestzeitabständen vorhanden sein müssen. Aus technischen Gründen ist es ferner sinnvoll, eine *gleichspannungsreie Signalcodierung* zu benutzen, d.h. eine Codierung, bei der die High- und Low-Pegel symmetrisch zu 0 V festgelegt sind und die High- und Low-Pegelanteile in summa über die Zeit gleich sind.

Bei der „normalen“ Darstellung der Bitfolge, die auch als *NRZ-Codierung* bezeichnet wird (non return to zero), ist diese Bedingung nicht erfüllt, da solche Übergänge bei längeren 0- oder 1-Folgen ausbleiben. Als Abhilfe können hier in

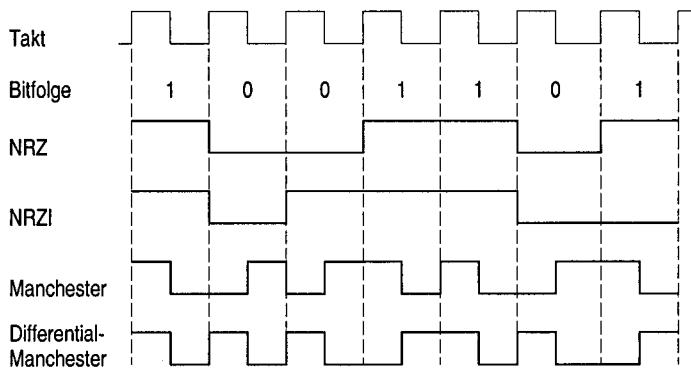


Bild 7-44. Signalcodierungen am Beispiel der Bitfolge 1001101

regelmäßigen Abständen Synchronisationszeichen in den Datenstrom eingefügt werden, die vom Empfänger wieder entfernt werden müssen (*character stuffing*, siehe 7.7.1). Bei der *NRZI-Codierung* (NRZ with interchange) wird die Null durch einen Pegelwechsel und die Eins durch gleichbleibenden Pegel dargestellt. Gesicherte Pegelwechsel (auch bei 1-Folgen) erhält man dadurch, daß man auf der Senderseite nach jeweils fünf 1-Bits ein 0-Bit in den Datenstrom einfügt, das auf der Empfängerseite wieder entfernt wird (*bit stuffing*, siehe 7.6.2). Sowohl die NRZ- als auch NRZI-Codierung sind nicht gleichstromfrei. Gleichstromfrei wie auch selbstdtaktend sind die Manchester- und die Differential-Manchester-Codierung. Bei der *Manchester-Codierung* wird eine 0 durch eine positive und eine 1 durch eine negative Flanke jeweils in Taktmitte dargestellt, d.h., die erste Takthälfte zeigt den „wahren“ Bitwert. Durchgehende 0- oder 1-Pegel während eines Takts deuten auf einen Übertragungsfehler hin. Bei der *Differential-Manchester-Codierung* wird bei einer 0 ein Pegelwechsel am Taktanfang vorgenommen, bei einer 1 wird hingegen mit dem vorhandenen Pegel fortgefahrt. In beiden Fällen findet ein Pegelwechsel in der Mitte des Takts statt.

Für das *Breitbandverfahren* gibt es zwei grundsätzliche Modulationstechniken: die *Schwingungsmodulation* und die *Pulsmodulation*. Bei der Schwingungsmodulation ist der Träger eine kontinuierliche Sinusschwingung, bei der Pulsmodulation eine Impulsfolge. Beispiele für Signalcodierungen in diesen Techniken sind in 7.7.2 im Zusammenhang mit Modems, d.h. mit der analogen Übertragung von digitalen Signalen, und im Zusammenhang mit der digitalen Übertragung von Analogsignalen (z.B. Sprache) aufgeführt.

Funknetze. Neben den drahtgebundenen lokalen Netzen kommen verstärkt drahtlose lokale Netze, d.h. Funknetze zum Einsatz. Die gebräuchlichste Technik ist hier das *WLAN* (*wireless LAN*). Es ist durch IEEE-Standards beschrieben, ausgehend von einem ersten Standard 802.11 von 1997, mit inzwischen mehreren Varianten, die durch eine zusätzliche Buchstabenkennung bezeichnet sind. Der Netzaufbau erfolgt hier über sog. *WLAN-Adapter* der einzelnen Netzteilnehmer

(Rechner) und eine vermittelnde Basestation (access point, AP), die die Verbindung zu einem drahtgebundenen lokalen Netz, z.B. einem Ethernet herstellt. In einer einfacheren Version können die Netzteilnehmer auch direkt mittels der WLAN-Adapter (und ohne eine Netzanbindung) miteinander kommunizieren (Ad-hoc-Netz, *Peer-to-Peer-Netz*). Die für die Varianten angegebenen Bruttoübertragungsraten betragen z.B. 1 und 2 Mbit/s (802.11), 5,5 und 11 Mbit/s (802.11b) und 54 Mbit/s (802.11g, 802.11a). Die tatsächlich erreichbaren Raten liegen allerdings bei nur ungefähr der Hälfte dieser Angaben, was mit den in den Protokollen vorhandenen Zeitvorgaben für das Senden von Datenpaketen (frames) zusammenhängt. Im Standardisierungsprozeß befindet sich derzeit ein neues Protokoll, mit dem eine tatsächliche Übertragungsrate von 100 Mbit/s erreicht werden soll (802.11n). – Die Funkreichweiten liegen innerhalb von Gebäuden bei 15 bis 100 m, im Freien bei 1 km.

Eine weitere drahtlose Übertragungstechnik für jedoch kürzere Funkstrecken von bis zu 12 Metern ist *Bluetooth*. Sie wurde 1998 von verschiedenen Firmen (Bluetooth Special Interest Group) als Nachfolger der Infrarot-Technik entwickelt, ursprünglich, um Peripheriegeräte wie Tastatur und Maus kabellos an den PC anzuschließen. Inzwischen wurde ihre Funktionalität erweitert, u.a. auf den Anschluß von Druckern, Digitalkameras und Camcordern, den Internet-Zugang (Verbindung zwischen PC und DSL-Modem), das Telefonieren (Verbindung zwischen Headset und Mobiltelefon) und auf die Vernetzung von PCs, Notebooks und PDAs (zur Datensynchronisation). Jedes Gerät muß dazu mit einem entsprechenden Adapter versehen sein, z.B. einem USB-Adapter. Für den Zugang zu einem LAN wird ein Access-Point als Zusatzgerät benötigt. Hinsichtlich der Übertragungsraten unterscheidet man zwei Betriebsarten: Die symmetrische Übertragung sieht in beiden Richtungen je 433 kbit/s vor; sie wird z.B. für die Gerätevernetzung eingesetzt. Die asymmetrische Übertragung, die für den Internet-Zugang typisch ist (Internet-Surfen), ermöglicht 57 kbit/s für den *Hinkanal* und 732 kbit/s für den *Rückkanal* (siehe dazu auch die xDSL-Techniken auf S. 536).

OSI-Referenzmodell. Die Kommunikation zwischen zwei Anwendungsrechnern (Wirtsrechnern, Hosts) bzw. zwischen den auf ihnen laufenden Anwenderprozessen umfaßt eine Vielzahl von Funktionen, die von den übertragungstechnischen Voraussetzungen bis zu den organisatorischen Vorgaben auf der Anwenderebene reichen. Hierzu gehören Aufbau, Aufrechterhaltung und Abbau einer Verbindung, Übertragen eines Bitstroms, Aufteilen eines Bitstroms in Übertragungsblöcke, Sichern der Datenübertragung und Fehlerbehandlung, Wegewahl im Netz, Synchronisieren der Übertragungspartner, Herstellen einer einheitlichen Datenrepräsentation, Aufteilen der zu übertragenden Information in logische und physikalische Abschnitte usw.

Aufgrund der hiermit verbundenen Komplexität der Abläufe, und um den Netzzugang für alle Teilnehmer eines Netzes zu vereinheitlichen, wurde von der ISO das sog. *OSI-Referenzmodell* (open systems interconnection) als ein für alle Netzteil-

nehmer „offenes“ System entwickelt. Dieses beschreibt modellhaft die Kommunikation in sieben hierarchischen Schichten (layers) unterschiedlicher Abstraktion (Bild 7-45). Hierbei stellt eine Schicht $i+1$ zur Verfügung; sie selbst bezieht Dienstleistungen von der unter ihr liegenden Schicht $i-1$. Die für die Kommunikation innerhalb einer jeden Schicht erforderlichen Regeln und technischen Vorgaben werden in Protokollen festgelegt. Für sie gibt es Normungsvorschläge der ISO und der ITU.

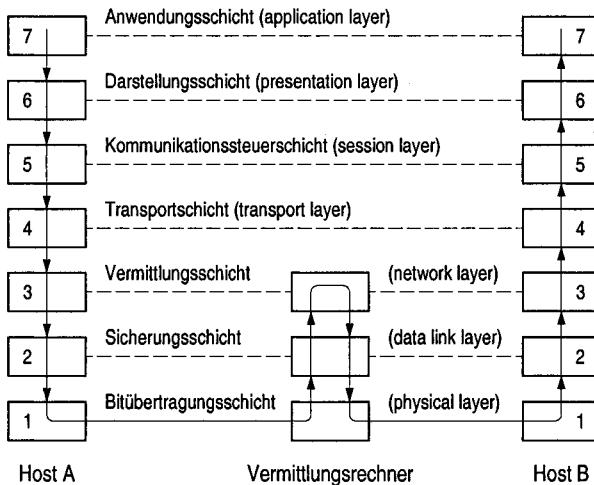


Bild 7-45. Die sieben Schichten des OSI-Referenzmodells

Das Referenzmodell wurde ursprünglich für WANs entwickelt, wird aber auch bei LANs angewandt. Es hat sich allerdings nicht als einzige Möglichkeit durchgesetzt. Viele der heute gebräuchlichen Kommunikationssysteme orientieren sich jedoch an ihm oder werden mit ihm in Beziehung gesetzt.

Bild 7-46 zeigt ein Beispiel für die unteren vier Schichten des OSI-Referenzmodells aus der Sicht der Informationsdarstellung und -übertragung. Die gesamte aus Rahmen und Text bestehende Information eines Übertragungsblocks in einer Schicht wird in der nächst tieferen Schicht als Text angesehen, der wiederum durch einen Rahmen ergänzt wird. Der Rahmen auf der untersten Schicht enthält schließlich die Steuerinformation, die zur Übertragung der aus den Rahmen der höheren Protokollebenen und dem eigentlich zu übertragenden Text bestehenden Nachricht benötigt wird. Zu den drei untersten, hardware-nahen Schichten und ihren Aufgabe nun etwas genauere Angaben:

- **Bitübertragungsschicht (physical layer).** Sie beschreibt als unterste Schicht, d.h. Schicht 1, die mechanischen und elektrotechnischen Eigenschaften der Übertragung zwischen einer Datenendeinrichtung und einer Datenübertragungseinrichtung oder zwischen zwei Datenendeinrichtungen. Hierzu gehört zunächst die Festlegung des Steckers und der Signale mit ihren elektrischen

Werten. Für WANs sind diese Vorgaben z.B. in den Schnittstellenvereinbarungen V.24/V.28 (ITU) oder RS-232-C/RS-423-A/RS-422-A (EIA) für die analoge Übertragung und X.21 (ITU) für die digitale Übertragung festgelegt (siehe 7.4). Weiterhin beschreibt diese Schicht die Art, in der Bitfolgen übertragen werden, synchron oder asynchron, sowie die Übertragungsrate.

- *Sicherungsschicht (data-link layer)*. Sie beschreibt als nächst höhere Schicht, d.h. als Schicht 2, die Übertragung von Nachrichten in der Form von Übertragungsblöcken zwischen zwei Knoten in einem Netz und dabei insbesondere die Sicherung der Datenübertragung. Hierzu gehören Mechanismen zur Fehlererkennung und -beseitigung. Ihr zuzuordnen sind z.B. die in 7.6.2 beschriebenen Protokolle BISYNC, SDLC und HDLC mit der Festlegung der Übertragungsformate, der Zeichensynchronisation, der Datensicherung, der Betriebsarten halbduplex/duplex, des Auf- und Abbaus und der Übertragungssteuerung einer Datenverbindung. Bei LANs werden die Schichten 1 und 2 z.B. durch einen der oben genannten IEEE-Standards 802.5 (Token-Ring), 802.4 (Token-Bus) oder 802.3 (Ethernet-Bus) gebildet.
- *Vermittlungsschicht (network layer)*. Sie stellt als Schicht 3 eine transparente End-zu-End-Verbindung im Datennetz her und beschreibt dazu den Verbindungsauflauf über mehrere Knoten durch Adressieren des Empfängers, die Wegewahl im Netz (routing), das Multiplexen von Verbindungen der Schicht 2 sowie die Übertragungssteuerung und Fehlerbehandlung in dieser Ebene.

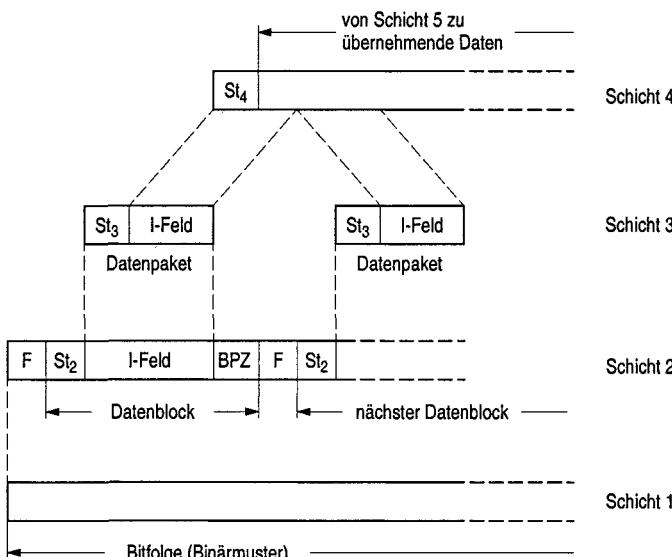


Bild 7-46. Bildung und Übertragung von Datenpaketen entsprechend der ITU/CCITT-Empfehlung X.25 (nach [Hegenbarth 1981]), I-Feld Informationsfeld, St₂ Steuerinformation der Schicht 2, F Blockbegrenzungsfeld, BPZ Blockprüfzeichenfolge

Die Datenpakete der Schicht 3 enthalten die Information der Schicht 2 als Daten und zusätzlich die für die Schicht 3 erforderliche Steuerinformation. Mit jeder weiteren Schicht wird somit eine Abstraktion erreicht, die immer mehr von den technischen Gegebenheiten wegführt und schließlich rein anwendungsorientierte Betrachtungen der Datenübertragung erlaubt. Häufig verwendetes Protokoll der Schicht 3 ist die ITU-T-Empfehlung X.25, die auf den Protokollen HDLC (Schicht 2, siehe auch 7.6.2) und X.21 (Schicht 1, siehe auch 7.4.4) aufbaut. – Zu den drei untersten Schichten des OSI-Referenzmodells siehe z.B. [Tanenbaum 2003].

7.7.2 Datenfernübertragung

Als *Datenfernübertragung (DFÜ)* bezeichnet man die Datenübertragung zwischen *Datenendgeräten* (data terminal equipment, DTE; z.B. Rechner und Terminals) unter Benutzung von Telekommunikationsnetzen, d.h. von Übertragungsleitungen und Vermittlungseinrichtungen der Fernmeldeinstitutionen eines Landes. Die Anpassung der Datenendgeräte an die Signaldarstellung und Übertragungsvorschriften dieser Institutionen erfordert an der Teilnehmerschnittstelle *Datenübertragungseinrichtungen* (data circuit-termination equipment, DCE), die bei analoger Übertragung als *Modems* und bei digitaler Übertragung als Datenanschlußgeräte oder als Adapter bezeichnet werden (Bild 7-47, siehe auch 7.4).

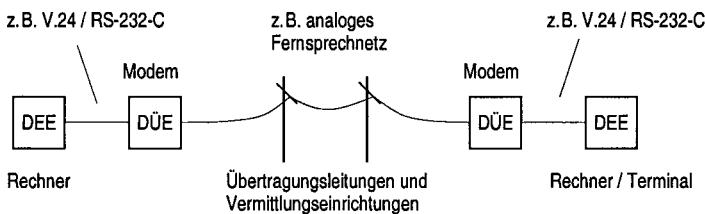


Bild 7-47. Datenfernübertragung zwischen zwei Datenendeinrichtungen (DEE) unter Verwendung zweier Datenübertragungseinrichtungen (DÜE). Als Übertragungssystem wurde das analoge Fernsprechnetz angenommen, dementsprechend sind als DÜE Modems eingesetzt

Telekommunikationsnetze. Von der Deutschen Telekom werden für die Datenfernübertragung verschiedene *Telekommunikationsnetze* unterschiedlicher Leistungsfähigkeit bereitgestellt. Ältestes Netz ist hier das *analoge Fernsprechnetz* mit einer relativ geringen Übertragungsleistung, bedingt durch den von der Sprachübertragung herrührenden geringen Frequenzbereich von 300 bis 3400 Hz. Der Netzzugang für die Datenübertragung erfolgt durch Modems mit Übertragungsraten von zunächst bis zu 9600 bit/s. Verfeinerte Techniken erlauben darüber hinaus Übertragungsraten von bis zu 33,6 und 56 kbit/s (siehe unten). Dieses analoge Netz wird durch das neuere, *digitale Fernsprechnetz ISDN* (integrated services digital network) ergänzt. Es erlaubt den direkten digitalen Netzzugang

und sieht dazu für jeden Anschluß (Kabel) zwei logische Übertragungskanäle (B-Kanäle) mit Übertragungsraten von je 64 kbit/s (ca. 8000 Zeichen pro Sekunde) und einen logischen Signalisierkanal (D-Kanal) mit 16 kbit/s vor, an die in einer Art Busstruktur bis zu acht Datenendgeräte einschließlich Telefon angeschlossen werden können. Die höhere Übertragungsleistung von ISDN ergibt sich durch eine Ausweitung des genutzten Frequenzbereichs auf 120 kHz, wobei wie beim analogen Fernsprechnetz verdrillte Kupferleitungspaare (Twisted-pair-Kabel) für den Teilnehmeranschluß an das Netz verwendet werden (siehe unten: xDSL-Techniken). – Mit diesen beiden Netzen steht dem Benutzer eine Vielzahl von Diensten für die Übertragung von Sprache, Text, Festbildern (eingeschränkt auch von Bewegtbildern) und von Daten zur Verfügung, so u.a. das Fernsprechen (Telefon, auch Bildtelefon), das Fernkopieren (Telefax) und der Anschluß an das Internet.

Neben den beiden Fernsprechnetzen mit ihrer gemischten Nutzung für Sprache und Daten gibt es reine *Datennetze*, deren Zugang entweder über analoge oder digitale Fernsprechanschlüsse (Kupferkabel) oder über spezielle Netzanschlüsse (Kupfer-, Koaxial- oder Glasfaserkabel) erfolgt. Fernsprechanschlüsse sind kostengünstig, eignen sich aber nur für relativ geringe Datenaufkommen. Sie werden deshalb vorwiegend von privaten Nutzern und kleineren Betrieben eingesetzt. Spezielle Netzanschlüsse sind kostenintensiv, ermöglichen dafür aber sehr hohe Übertragungsraten. Sie werden dementsprechend vorwiegend von Firmen und Institutionen mit hohem Datenaufkommen eingesetzt. – Im Gegensatz zu den bei den Fernsprechnetzen gebräuchlichen Leitungsvermittlung arbeiten Datennetze üblicherweise mit Paketvermittlung (S. 522).

Analoger und digitaler Netzzugang. Betrachtet man die Übertragungsstrecke zwischen Teilnehmeranschluß und Netz sowie innerhalb des Netzes und ggf. zwischen dem Netz und einem zweiten Teilnehmeranschluß, so gibt es auf diesem Weg unterschiedliche analoge und digitale Darstellungen der Daten (wie auch von Sprache). Daten fallen naturgemäß in digitaler Form an, üblicherweise in der NRZ-Codierung (Bild 7-44, S. 529) und werden in *analoger Darstellung* (Modem) oder in *digitaler Darstellung* (Datenanschlußgerät, Adapter) weitergereicht, wobei auf eine erste Umwandlung an der Teilnehmerschnittstelle ggf. weitere Umwandlungen im Netz bzw. zum zweiten Teilnehmer folgen. Vergleichbares gilt für Sprachsignale, die naturgemäß analog anfallen und ggf. bereits an der Teilnehmerschnittstelle (ISDN) oder aber an der Netzschnittstelle digitalisiert werden. – Im folgenden werden die für den *Netzzugang* gebräuchlichen analogen und digitalen Techniken kurz skizziert. Ergänzt werden diese Ausführungen durch Hinweise zur Digitalisierung analoger Signale, z.B. von Sprache, wie sie für die Übertragung der Signale in digitalen Netzen erforderlich ist. Bezuglich der digitalen Darstellung von Daten in Netzen sei auf die bereits in 7.7.1 beschriebenen Signaldarstellungen und -codierungen verwiesen.

Modem-Techniken. Bei der Datenübertragung im *analogen Fernsprechnetz* oder allgemein beim Netzzugang mittels eines analogen Teilnehmeranschlusses wird, wie bereits erwähnt, das digitale Signal auf der Senderseite von einem *Modem* (Modulator/Demodulator) in ein analoges Signal umgewandelt (Modulation) und das analoge Signal auf der Empfängerseite durch ein Modem wieder in das digitale Signal zurückgewandelt (Demodulation). Für die Erzeugung des Analogsignals wird ein sinusförmiger Träger aus dem für die Sprachübertragung vorgegebenen Frequenzbereich von 300 bis 3400 Hz verwendet, auf den das digitale Signal aufmoduliert wird (*Schwingungsmodulation*). Grundsätzlich unterscheidet man hierbei die in Bild 7-48 dargestellten drei aus der Radio- und Fernmeldetechnik bekannten Modulationsarten: die *Amplituden-*, die *Frequenz-* und die *Phasenmodulation*.

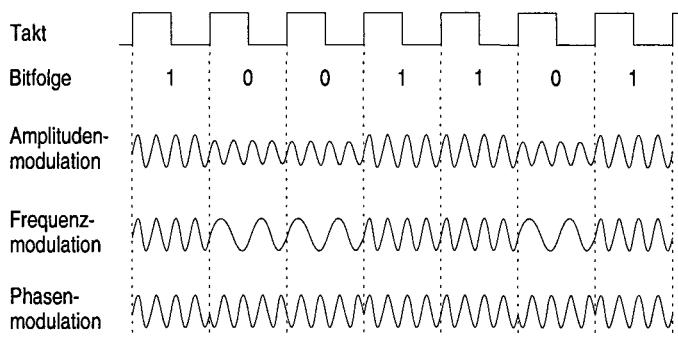


Bild 7-48. Modulationstechniken bei Modems: Amplituden-, Frequenz- und Phasenmodulation. Durch die Verwendung von zwei Signalamplituden oder zwei Signalfrequenzen oder der Phasenverschiebung um einen bestimmten Winkel (hier 180°) lässt sich ein Bit pro Takschritt darstellen

Abweichend von den Darstellungen in Bild 7-48 kann auch mehr als ein Datenbit pro Takschritt übertragen werden, nämlich dann, wenn mehr als zwei Amplituden, zwei Frequenzen oder zwei Phasenwinkel verwendet werden und wenn man ggf. unterschiedliche Modulationsarten miteinander kombiniert. Dies nutzt man z.B. bei der sog. *Quadrature Amplitude Modulation* (QAM) aus, bei der vier Bits durch acht verschiedene Phasenwinkel mit fester Signalamplitude und durch weitere vier Phasenwinkel mit je zwei möglichen Amplituden codiert werden. Man erreicht hier bei der für das analoge Fernsprechnetz typischen Schrittgeschwindigkeit von 2400 Baud eine Übertragungsrate von 9600 bit/s (V.29-Modem). Durch Erhöhung der Bitanzahl pro Takschritt erreicht man Übertragungsraten von bis zu 14,4 kbit/s (V.32bis-Modem) und bei zusätzlicher Erhöhung der Schrittgeschwindigkeit auf 3200 Baud Übertragungsraten von bis zu 28,8 kbit/s (V.34-Modem). Inzwischen wurde diese Übertragungstechnik noch verfeinert und die maximal mögliche Übertragungsrate bei einer Schrittgeschwindigkeit von 3429 Baud auf 33,6 kbit/s angehoben (V.34-Modem).

Mit einer etwas anderen Technik, bei der die Daten nicht analog durch Modulation eines Trägers, sondern digital als Leistungspegel dargestellt werden, ist eine noch höhere Übertragungsrate von bis zu 56 kbit/s möglich (*56k-Modem*, V.90). Dieser Wert gilt allerdings nur für den Datenempfang und setzt voraus, daß der Kommunikationspartner zur Gewährleistung des hierfür erforderlichen sehr guten Signal-Rausch-Verhältnisses einen rein digitalen Anschluß hat. In der Praxis werden deshalb meist nicht mehr als 48 kbit/s erreicht. Bis zu 56 kbit/s erreicht man ggf. beim „Herunterladen“ von Daten aus dem Internet. Kommunizieren zwei 56k-Modems miteinander, so liegt die maximal erreichbare Übertragungsrate von vornherein bei nur 33,6 kbit/s.

Grundsätzlich setzen sich beim Aufbau einer Modem-Datenverbindung beide Übertragungspartner mit zunächst einer geringen Übertragungsrate von z.B. 300 bit/s miteinander ins Benehmen, um vor der eigentlichen Datenübertragung die für sie höchstmögliche Übertragungsrate zu ermitteln und festzulegen. Diese hängt nicht nur von den technischen Merkmalen der beiden Modems ab, sondern auch von der Qualität der Übertragungsstrecke. Diese sog. *Connect-Rate* kann also von Verbindung zu Verbindung verschieden sein. Insbesondere kann sich die Übertragungsqualität auch während einer Verbindung noch verändern, z.B. verschlechtern. Sinkt sie dabei so weit ab, daß die Übertragung fehlerhaft wird, so führt das normalerweise zum Abbruch der Übertragung. Um das zu verhindern, werden Modems eingesetzt, die sich bei Änderung der Qualität spontan auf eine neue Übertragungsrate verstndigen knnen, um danach die Übertragung mit ggf. geringerer (oder auch hherer) Übertragungsrate fortzusetzen. – Zu den Modem-Techniken siehe auch [Tanenbaum 2003].

xDSL-Techniken. Die relative geringen Übertragungsraten bei Modems sind durch das fr die Datendarstellung verfgbare schmale Frequenzband der Sprach-übertragung bedingt. Technisch gesehen liegt dabei der Engpaß nicht bei dem verdrillten Kupferleitungspaar, mit dem der Telefonnetzteilnehmer (auf der sog. letzten Meile, local loop) mit der Ortsvermittlungsstelle verbunden ist, sondern bei den ggf. auf dieser Verbindungsstrecke vorhanden Signalverstrkern, die als Bandpaßfilter wirken. Bei Teilnehmeranschlussen ohne solche Verstrker knnen fr die Datendarstellung grere oder mehrere Frequenzbereiche genutzt werden, so daß in Verbindung mit rein digitalen Datendarstellungen sehr viel hhere Übertragungsraten als mit Modems erreicht werden. Hier gibt es verschiedene Techniken, die unter dem Begriff *xDSL* zusammengefaßt sind. x steht dabei als Platzhalter fr die Unterscheidung der einzelnen Techniken, DSL fr *Digital Subscriber Line* (subscriber: Fernsprechteilnehmer).

Die „reine“ DSL-Technik ist die Technik des ISDN-Anschlusses. Sie und ihre Varianten xDSL basieren auf der digitalen Datendarstellung unter Anwendung von sowohl Basisband- als auch Breitbandverfahren. Bei den *Breitbandverfahren* werden die Gleichspannungssignale auf Trgerfrequenzen aufmoduliert, um sie im Frequenzband verschieben zu knnen. Genutzt werden wie bei „analogen“

Modems Codierungs- und Modulationsverfahren (*Amplituden- und Phasenmodulation*), mit denen mehrere Bits pro Schritt dargestellt und übertragen werden können. Man bezeichnet deshalb die DSL-Anschlußgeräte ebenfalls als *Modems*, auch wenn sie bei Verwendung von Basisbandverfahren nur Adapter mit der Umsetzungsfunktion digital-zu-digital sind. Anders als bei der Datenübertragung über analoge Modems mit einer direkten Wählverbindung werden *DSL-Adapter* sternförmig an Server angeschlossen und die Daten dann netzartig weitervermittelt. Da die Reichweite vom Teilnehmeranschluß aus je nach xDSL-Technik und Übertragungsrate auf Entfernungen von einigen 100 Metern bis zu wenigen Kilometern eingeschränkt ist, werden die beim Teilnehmer vorhandenen verdrillten Kupferleitungspaare ggf. nur auf kurzer Strecke bis zu einem Kabelverteiler genutzt und die restliche Strecke zur Ortsvermittlungsstelle dann mittels Glasfaserkabel überbrückt.

Tabelle 7-4 zeigt die derzeit wichtigsten xDSL-Techniken und ihre Übertragungsraten. Diese sind direkt von der bis zur Ortsvermittlungsstelle zu überbrückenden Entfernung abhängig, d.h., sie werden mit zunehmender Entfernung geringer. *ADSL* (asymmetric DSL) ist die derzeit gebräuchlichste Technik. Sie ist wie ihr Name sagt asymmetrisch, d.h., die Übertragungsrate im sog. *Hinkanal (upstream, upload)*, nämlich vom Teilnehmer zum „Netz“, ist sehr viel geringer als die für den *Rückkanal (downstream, download)*, womit man den beim „Surfen“ auftretenden Verhältnissen Rechnung trägt. Mit einem zusätzlichen Filter (*splitter*) kann der Anschluß gleichzeitig auch für die Telefonie benutzt werden, indem der verfügbare Frequenzbereich in einen niederfrequenten Bereich für die Telefonie und in einen höherfrequenten Bereich für den Modem-Betrieb aufgeteilt wird. Als Techniken mit symmetrischen Übertragungsraten gibt es *HDSL* (high data rate DSL) als reine Datenverbindung und als Erweiterung davon *SDSL* (symmetric oder single-line DSL), das zusätzlich zur Datenübertragung Kanäle mit 64

Tabelle 7-4. Die wichtigsten xDSL-Techniken und ihre maximal möglichen Übertragungsraten. Die tatsächlich erreichbaren Werte hängen u.a. von den zu überbrückenden Entfernungen ab. Benötigt wird jeweils ein Adernpaar, bei HDSL ggf. zwei oder drei (zur Verringerung von Störeinflüssen). Im Vergleich dazu die maximalen Modem- und ISDN-Datenübertragungsraten

xDSL-Technik	Übertragungsrate Mbit/s	
	Hinkanal (upload)	Rückkanal (download)
ADSL (asymmetric)	0,128 bis 1	0,8 bis 8
HDSL (high data rate)	1,54 oder 2	1,54 oder 2
SDSL (symmetric/single line)	2,3	2,3
VDSL (very high data rate)	2,3	52
"	26	26
Modem	0,056	0,056
ISDN	0,128	0,128

kbit/s für Telefonie/ISDN bereitstellt. Eine weitere Technik mit sehr viel höheren Übertragungsraten ist *VDSL* (very high data rate DSL). Sie sieht sowohl die asymmetrische als auch die symmetrische Übertragung vor. – Zu den xDSL-Techniken siehe vertiefend z.B. [Tillmann 1997].

Digitalisierung von analogen Sprach- und Datensignalen. In der analogen Fernsprechtechnik sind heute neben analogen auch digitale Übertragungseinrichtungen im Einsatz, die sowohl für die Sprachübermittlung als auch für die Datenübertragung benutzt werden. Hierbei werden die vom Netzteilnehmer direkt oder über Modem kommenden Analogsignale im Fernmeldeamt digitalisiert. Zum Einsatz kommen hier sog. *Pulsmodulationsverfahren*, bei denen das analoge Signal mit 8 kHz abgetastet und quantisiert wird. Diese Frequenz ergibt sich aus dem Nyquist-Theorem, nach dem die Abtastfrequenz mindestens doppelt so hoch wie die höchste vorkommende Signalfrequenz sein muß. Bezugspunkt ist hierbei die oberhalb der Begrenzung des Fernsprechbandes liegende Frequenz von 4000 Hz. Bei der gebräuchlichen sog. *Puls-Code-Modulation (PCM)* wird jeder Abtastwert durch ein 7- oder 8-Bit-Codewort dargestellt, womit sich eine Übertragungsrate von 64 kbit/s ergibt. An den beiden Enden der Übertragungsstrecke ist dazu je ein sog. *Codec* (Codierer/Decodierer) erforderlich, der die Abtastung (Codierung) vornimmt bzw. der aus den Abtastwerten wieder ein Analogsignal (Decodierung) erzeugt. – Bei reiner Sprachübertragung kann unter Inkaufnahme von Verlusten die Datenrate verringert werden, indem mit z.B. 5 Bits jeweils nur die Differenz benachbarter Abtastwerte übertragen wird (*differentielle Puls-Code-Modulation*) oder indem mit nur einem Bit nur noch dem Signalverlauf hinsichtlich Anstieg und Abfall Rechnung getragen wird (*Deltamodulation*).

7.7.3 Sicherung der Datenübertragung

Die Übertragung von Daten ist Störungen unterworfen, die zur Änderungen von Datenbits und damit zu fehlerhaften Daten auf der Empfängerseite führen können. Um solche Fehler erkennen und ggf. sogar korrigieren zu können, muß die Nutzinformation auf der Senderseite durch Prüfinformation ergänzt werden (redundante Informationsdarstellung). Der Empfänger blendet aus der insgesamt empfangenen Information die Nutzinformation aus und bildet aus ihr seinerseits die Prüfinformation und vergleicht sie mit der übertragenen Prüfinformation. Diese *Codesicherung* erfolgt entweder für jedes einzelne Codewort oder für einen ganzen Datenblock. Die *Codewortsicherung* ist typisch für die Übertragung einzelner Zeichen, z.B. bei asynchron-serieller Übertragung; die *Datenblocksicherung* wird bei blockweiser Datenübertragung, so z.B. bei der Datenübertragung mit Festplatten und anderen Hintergrundspeichern sowie in Rechnernetzen eingesetzt (Schicht 2 des OSI-Referenzmodells). Auf eine als fehlerhaft erkannte Datenübertragung gibt es zwei grundsätzliche Möglichkeiten der Reaktion: die *Fehlerkorrektur* und die *Übertragungswiederholung*. Letztere setzt eine Halbdu-

plex- oder Duplex-Datenverbindung voraus, um die Wiederholung anfordern zu können (siehe auch 7.6.2: BISYNC-Protokoll). – Zur Codesicherung siehe [Hamming 1987] und z.B. [Tanenbaum 2003].

Einzelsicherung. Ein Maß für die Anzahl der bei einem bestimmten Code erkennbaren bzw. korrigierbaren Fehler in einem Codewort ist die *Hamming-Distanz* h des redundanten Codes. Diese gibt an, wieviele Stellen eines Codewortes mindestens geändert werden müssen, damit ein anderes gültiges Codewort entsteht. So muß die Distanz $h = n+1$ sein, um n fehlerhafte Bits in einem Codewort erkennen zu können. Um n fehlerhafte Bits in einem Codewort korrigieren zu können, muß sie $h = 2n+1$ sein.

Bei der einfachsten Codesicherung durch ein *Paritätsbit (Querparität)* ist $h = 2$, womit ein *1-Bit-Fehler* erkannt, jedoch nicht korrigiert werden kann. Der Wert des Paritätsbits wird so bestimmt, daß die Quersumme des redundanten Codewortes entweder gerade (*gerade Parität, even parity*, empfohlen für die asynchron-serielle Datenübertragung) oder ungerade wird (*ungerade Parität, odd parity*, empfohlen für die synchron-serielle Datenübertragung). Die gerade Parität entsteht durch Modulo-2-Bildung. Bild 7-49 zeigt dazu je ein Beispiel mit jeweils drei 7-Bit-ASCII-Zeichen, wobei das letzte Zeichen in der Bitposition 4 fehlerhaft übertragen wurde. Man bezeichnet diese Querparitätsbildung auch als *VRC-Sicherungsverfahren* (vertical redundancy checking). – Die Hamming-Distanz $h = 3$ erreicht man bei acht Bit Nutzinformation durch vier zusätzliche Prüfbits in geeigneter Codierung. Dies erlaubt es, entweder zwei gleichzeitige 1-Bit-Fehler zu erkennen oder einen 1-Bit-Fehler zu korrigieren.

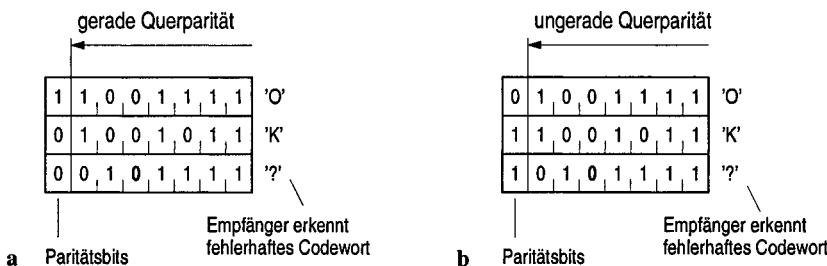


Bild 7-49. Datensicherung durch Paritätsbit, a gerade Parität, b ungerade Parität

Bild 7-50 demonstriert die Hamming-Distanzen $h = 1, 2$ und 3 anschaulich an einem 3-Bit-Code. Die gültigen und fehlerhaften Codewörter sind hier als Eckpunkte eines Würfels dargestellt, die Kanten des Würfels beschreiben die Übergänge der Codewörter zu ihren Nachbarn. Die bei $h = 3$ mögliche *1-Bit-Fehlerkorrektur* beruht darauf, daß das fehlerhafte Codewort einem „näheren“ Nachbarn eindeutig zugeordnet werden kann. Beruht eine 1-Bit-Korrektur auf einem 2-Bit-Fehler, so ist die Korrektur falsch, was nicht zu erkennen ist. Entscheidet man sich für eine *2-Bit-Fehlererkennung*, so entfällt bei $h = 3$ die 1-Bit-Fehler-

korrektur. Dazu Hamming: „Es ist selten ratsam, nur Einzelfehlerkorrektur anzuwenden, da ein Doppelfehler das System fehlleitet, wenn es eine Korrektur versucht“ [Hamming 1987]. Um nun beides zu ermöglichen, benötigt man die Hamming-Distanz $h = 4$. Sie wird z.B. zur Sicherung von Hauptspeicherzugriffen durch sog. SECDED-Prüfbausteine eingesetzt (*single error correction, double error detection*). Für ein 64-Bit-Speicherwort benötigt man dazu lediglich 7 Prüfbits, die bei den entsprechenden DRAM-Modulen (DIMMs, 6.1.2) in einem zusätzlichen neunten DRAM-Baustein untergebracht werden.

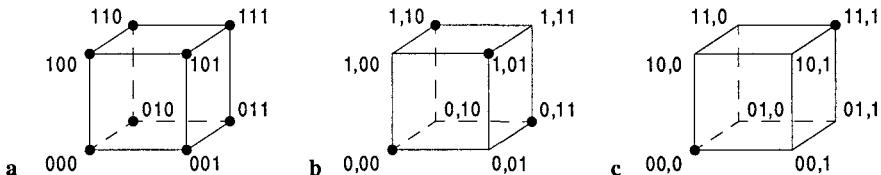


Bild 7-50. Darstellung der Hamming-Distanz h für einen 3-Bit-Code mit den höherwertigen Bits (vor dem Komma) als Prüfbits; gültige Codewörter als fette Eckpunkte hervorgehoben. **a**: $h = 1$: Mindestdistanz zwischen zwei Codewörtern für die Eindeutigkeit des Codes, **b**: $h = 2$: Mindestdistanz für eine 1-Bit-Fehlererkennung (hier als gerade Parität), **c**: $h = 3$: Mindestdistanz für eine 2-Bit-Fehlererkennung oder eine 1-Bit-Fehlerkorrektur

Blocksicherung. Bei der Übertragung von Datenblöcken verwendet man üblicherweise nur dann fehlerkorrigierende Codes, wenn aufgrund einer Simplexverbindung im Fehlerfall keine Rückmeldung und damit keine Übertragungswiederholung (retransmission) möglich ist. In allen andern Fällen werden, um die Redundanz gering zu halten, bevorzugt fehlererkennende Blocksicherungsverfahren angewandt. Im einfachsten Fall einer Blocksicherung wird dazu an den (seriellen) Bitstrom insgesamt ein einziges Paritätsbit angefügt (*Quersicherung*), womit jedoch die Erkennung von Mehrbitfehlern nur eine Wahrscheinlichkeit von 0,5 hat. Eine bessere Sicherung erhält man bei zeichenorientierter Übertragung, wenn man an den Datenblock ein *Blockprüfzeichen BCC* (block checking character) anfügt. Dieses wird so gebildet, daß die Längssumme, d.h. die Summe aller Bits derselben Bitposition der Codewörter, entweder gerade oder ungerade ist (*Längssicherung*). Auch hier gilt wieder: *gerade Parität* bei asynchron-serieller und *ungerade Parität* bei synchron-serieller Datenübertragung. Bild 7-51a zeigt dazu als Beispiel einen Datenblock mit drei 7-Bit-ASCII-Textzeichen, dessen Anfang und Ende durch die Steuerzeichen STX (start of text) und ETX (end of text) gekennzeichnet ist (siehe auch 7.6.2). Die Blockprüfung bezieht sich auf alle fünf Zeichen, die durch ein zusätzliches Null-Bit in der höchsten Bitposition zu 8-Bit-Zeichen erweitert sind. Man bezeichnet diese Längsparitätsprüfung auch als *LRC-Sicherungsverfahren* (longitudinal redundancy checking).

Eine noch größere Sicherheit bietet die sog. *Kreuz- oder Rechtecksicherung*, bei der sowohl die Querparität als auch die Längsparität gebildet wird. Bild 7-51b zeigt dazu als Beispiel die ASCII-Zeichenfolge von Bild 7-51a, bei der jetzt je-

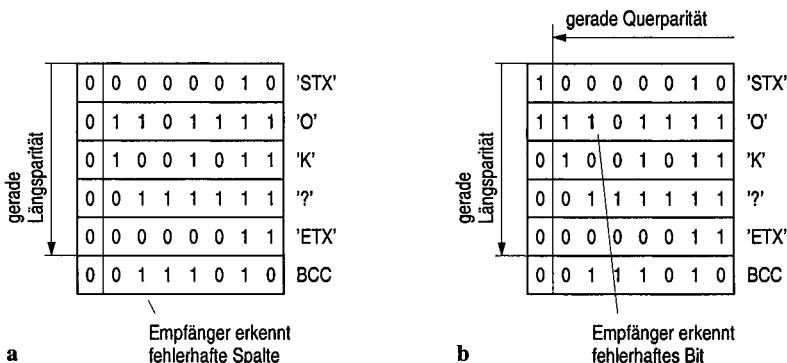


Bild 7-51. Blocksicherung. **a** Längssicherung mit Blockprüfzeichen BCC, **b** Kreuzsicherung als Kombination von Quer- und Längssicherung (Rechteckcode)

weils das achte Bit eines Codewortes für die Querparität genutzt wird, so auch beim Blockprüfzeichen BCC. Die Wahrscheinlichkeit, ein fehlerhaftes Bit nicht zu erkennen, liegt hier bei 10^{-3} . Grundsätzlich nicht lokalisierbar (wenn auch erkennbar) sind z. B. 2-Bit-Fehler innerhalb derselben Zeile oder Spalte, nicht erkennbar sind z. B. sog. Rechteckfehler, d. h. das Zusammentreffen von vier fehlerhaften Bits an den Kreuzungspunkten zweier Zeilen und zweier Spalten. Man bezeichnet diese Kombination von Quer- und Längssicherungsprüfung auch als *VRC/LRC-Sicherungsverfahren*. Es wird z. B. beim zeichenorientierten BISYNC-Protokoll angewandt (7.6.2).

Zyklische Blockprüfung. Wesentlich wirksamer als die bisher beschriebenen Verfahren und zudem mit geringerem technischen Aufwand verbunden ist die Blocksicherung mit *zyklischen Codes*, die *CRC-Sicherung* (cyclic redundancy checking). Hierbei können die n Bits des zu übertragenden seriellen Datenstroms $D(x)$ als Koeffizienten eines Polynoms $(n-1)$ ter Ordnung betrachtet werden (z. B. die Bitfolge 10100011 als Polynom $x^7 + x^5 + x^1 + 1$). Dieses wird durch ein fest vorgegebenes, sog. Generatorpolynom $G(x)$ modulo 2 dividiert, nachdem es zuvor mit der höchsten Potenz des Generatorpolynoms multipliziert wurde (hier mit x^7). Das entstehende Restpolynom $R(x)$ bildet dann die Prüfinformation, die zu dem multiplizierten Datenpolynom zu addieren ist. Auf der Empfängerseite wird das resultierende Polynom erneut durch das Generatorpolynom dividiert, wobei sich bei Fehlerfreiheit der Rest Null ergibt. (Je nach Modifikation des Verfahrens kann auch ein definierter Rest entstehen.)

In der technischen Realisierung entspricht die beschriebene Polynomdivision der *Modulo-2-Division* vorzeichenloser Dualzahlen, basierend auf der bitpaarweisen Subtraktion ohne Weiterreichung des Übertrags (siehe dazu Beispiel 7.5). Das kommt in der Wirkung der Modulo-2-Addition, d. h. der Exklusiv-Oder-Funktion gleich und lässt sich mit geringem Hardwareaufwand und mit vernachlässigbarer Zeitverzögerung durch ein über Exklusiv-Oder-Gatter rückgekoppeltes Schiebe-

register realisieren. Bild 7-52 zeigt dazu eine Schaltung für ein gebräuchliches *Generatorpolynom*, nämlich $G(x) = x^{16} + x^{15} + x^2 + 1$. Dieses Polynom führt aufgrund seiner Ordnung 16 zu 16 Prüfbits, zwei sog. CRC-Bytes. Diese werden an den zu übertragenden Datenstrom $D(x)$ angefügt. Auf der Empfängerseite wird der so ergänzte Datenstrom mit einer Schaltung gleichen Aufbaus erneut der Division durch $G(x)$ unterzogen. Von Vorteil ist also, daß ein Übertragungsteilnehmer für das Senden und Empfangen ein und dieselbe Schaltung nutzen kann.

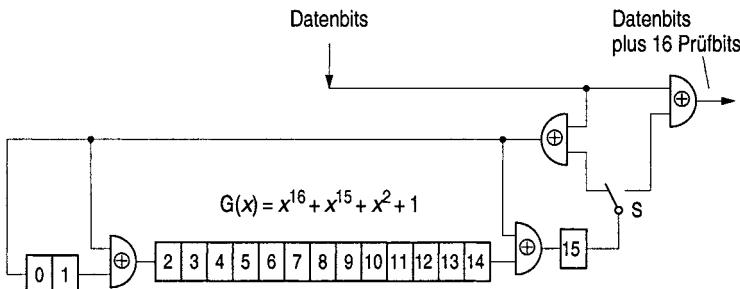


Bild 7-52. Bildung von 16 Prüfbits für zeichenorientierte Protokolle nach dem CRC-16-Verfahren mittels eines getakteten, über Exklusiv-Oder-Gatter rückgekoppelten Schieberegisters. Das Schieberegister wird vor Beginn der Datenübertragung mit 16 Nullen initialisiert. Nach Übertragung der Datenbits werden die im Schieberegister erzeugten 16 Prüfbits an den Datenbitstrom angehängt (Schalter S). Der jeweils offene Kontakt des Schalters liefert 0

Beispiel 7.5. ► Modulo-2-Division zur Bildung von CRC-Prüfbits. Für einen zu übertragenden 8-Bit-Datenstrom $D(x) = 10100011$ sollen senderseitig die sich mit dem Generatorpolynom $G(x) = x^4 + x^3 + 1$ ergebenden vier CRC-Prüfbits $R(x)$ ermittelt werden, und zwar auf der Basis von Bitmustern, die als Dualzahlen zu interpretieren sind. Der mit diesen Bits gesicherte Datenstrom soll dann empfängerseitig auf Fehlerfreiheit überprüft werden. – Für das Generatorpolynom ergibt sich entsprechend seiner Koeffizienten das Bitmuster $G(x) = 11001$. Vor Durchführung der erforderlichen Modulo-2-Division muß der Bitstrom $D(x)$ um vier Nullbits ergänzt werden, was der Multiplikation des Polynoms $D(x)$ mit der höchsten Potenz des Generatorpolynoms bzw. dem Initialisieren des Schieberegisters mit Nullbits entspricht. Die bei der Division auftretende Modulo-2-Subtraktion heißt bitpaarweises Subtrahieren ohne Bildung von Überträgen, was, wie oben erwähnt, in der Wirkung der Modulo-2-Addition, d.h. der Exklusiv-Oder-Funktion gleichkommt. Der bei der Division entstehende Quotient ist hier ohne Bedeutung, weshalb er in der in Bild 7-53 gezeigten Rechnung nicht mit angegeben wird. Auf der Empfängerseite wird der um die vier Prüfbits $R(x)$ erweiterte Datenstrom $D(x)$ erneut der Modulo-2-Division mit $G(x)$ unterzogen und liefert bei fehlerfreier Übertragung als Rest $R(x)$ das Bitmuster 0000.

Die Schaltung zur Realisierung des hier verwendeten Generatorpolynoms läßt sich in Analogie zu Bild 7-52 angeben. Initialisiert man in ihr die vier erforderlichen Schieberegisterbits mit Null und führt man ihr dann den Bitstrom $D(x)$, beginnend mit dem höchstwertigen Bit, taktweise zu, so verbleiben nach Durchlaufen aller Datenbits, d.h. nach acht Tastschritten, im Schieberegister die vier zu ermittelnden Prüfbits $R(x)$. Diese werden senderseitig mittels des Schalters S in vier weiteren Tastschritten an den Datenstrom angehängt. Empfängerseitig werden ebenfalls zwölf Takte durchlaufen, $D(x)$ gefolgt von $R(x)$, und danach die im Schieberegister stehenden Bits auf 0000 geprüft. – Der Leser möge sich diese CRC-Bit-Generierung und -Prüfung selbst veranschaulichen, indem er den Datenfluß anhand der Schaltung „durchspielt“.

Sender:

$$\begin{array}{r}
 \text{D(x)} \quad \text{Nullbits} \quad \text{G(x)} \\
 \overbrace{101000110000}^{\text{D(x)}} : \overbrace{11001}^{\text{G(x)}} \\
 \oplus \underline{11001} \\
 \underline{11010} \\
 \oplus \underline{11001} \\
 \underline{11110} \\
 \oplus \underline{11001} \\
 \underline{11100} \\
 \oplus \underline{11001} \\
 \hline
 \mathbf{a} \qquad \qquad \qquad 1010 \leftarrow \text{Prüfbits R(x)}
 \end{array}$$

Empfänger:

$$\begin{array}{r}
 \text{D(x)} \quad \text{R(x)} \quad \text{G(x)} \\
 \overbrace{101000111010}^{\text{D(x)}} : \overbrace{11001}^{\text{G(x)}} \\
 \oplus \underline{11001} \\
 \underline{11010} \\
 \oplus \underline{11001} \\
 \underline{11111} \\
 \oplus \underline{11001} \\
 \underline{11001} \\
 \oplus \underline{11001} \\
 \hline
 \mathbf{b} \qquad \qquad \qquad 0000 \rightarrow \text{fehlerfreie Übertragung!}
 \end{array}$$

Bild 7-53. CRC-Sicherung. **a** Berechnung der CRC-Prüfbits $R(x)$ für einen Datenstrom $D(x)$ durch den Sender, **b** Überprüfung des gesicherten Datenstroms $D(x) R(x)$ auf fehlerfreie Übertragung durch den Empfänger

8 Ein-/Ausgabesteuereinheiten, periphere Verbindungen und Hintergrundspeicher

In Kapitel 7 wurde davon ausgegangen, daß die Übertragung von Daten zwischen Speicher und Interface und die dazu notwendigen Organisationsaufgaben, z. B. die Adressfortschaltung und die Bytezählung, vom Mikroprozessor durchgeführt werden. Das kann für den Prozessor sehr zeitraubend sein, insbesondere bei Ein-/Auszabe mit Busy-Waiting. Aber auch wenn die Datenübertragung durch Interrupts synchronisiert wird, beanspruchen das Statusretten, das Ausführen des Interruptprogramms und das abschließende Statusladen immer noch ein Vielfaches der eigentlichen Datenübertragungszeit. Das macht sich vor allem bei hohen Übertragungsgeschwindigkeiten nachteilig bemerkbar, z. B. bei der Ein-/Auszabe mit einem Festplattenspeicher. Dieser Engpaß kann durch zusätzliche Hardwareunterstützung in Form von Ein-/Ausgabesteuereinheiten, wie DMA-Controller, Ein-/Ausgabeprozessoren oder Ein-/Ausgaberechner, behoben werden.

Auf der Seite der Peripheriegeräte gibt es ebenfalls Steuereinheiten, sog. *Device-Controller*. Sie erlauben es dem Programmierer, diese Geräte auf einer relativ hohen Ebene anzusprechen, z. B. in Form von Kommandos, und verdecken damit die Details der eigentlichen Gerätesteuerung. Von der Systemstruktur her unterscheidet man zwei grundsätzliche Anordnungen:

- Die Gerätesteuereinheit umfaßt zusätzlich die Interface-Funktionen und ist somit direkt an den Systembus ankoppelbar.
- Die Gerätesteuereinheit ist in das Peripheriegerät integriert (embedded control), und das Gerät ist zusammen mit anderen Geräten über eine periphere Punkt-zu-Punkt-Verbindung oder einen peripheren Bus mit dem Rechner verbunden.

In Abschnitt 8.1 werden zunächst drei grundsätzliche Möglichkeiten der prozessorunabhängigen Ein-/Ausgabe behandelt: die Ein-/Ausgabe mit Direktspeicherzugriff durch einen DMA-Controller, die Ein-/Ausgabe durch einen Ein-/Ausgabeprozessor und, in Erweiterung eines solchen Prozessors zu einem eigenständigen Rechnersystem, die Ein-/Ausgabe durch einen Ein-/Ausgaberechner. In Abschnitt 8.2 werden dann verschiedene periphere Punkt-zu-Punkt-Verbindungen und Peripheriebusse betrachtet, wie sie für den Anschluß von Ein-/Ausgabegeräten und Hintergrundspeichern eingesetzt werden. Hierzu zählen IDE/ATA, SATA, SCSI, SAS, FireWire, Fibre-Channel und USB. Sie verdeutlichen die Tendenz, parallele Verbindungen in Rechnersystemen vermehrt durch serielle zu ersetzen. Betrachtet wird außerdem der IEC-Bus als paralleler Bus für den An-

schluß von Anzeige- und Meßgeräten in einer Laborumgebung. In Abschnitt 8.3 werden der Aufbau, die Informationsdarstellung und die Steuerung von Hintergrundspeichern behandelt. Das sind der Floppy-Disk-Speicher, die Festplatten- und Wechselplattenspeicher, optische und magnetooptische Platterspeicher sowie die Streamer-Tape-Speicher.

8.1 DMA-Controller und Ein-/Ausgaberechner

Die zunächst einfachste Strukturmaßnahme zur Entlastung des zentralen Mikroprozessors von der Ein-/Auszgabe ist der Einsatz eines *DMA-Controllers*, d.h. einer Steuereinheit mit sog. *Direktspeicherzugriff* (direct memory access, DMA). Der DMA-Controller wird vor einem Ein-/Ausgabevorgang vom Mikroprozessor durch Beschreiben seiner Register initialisiert und ist dann in der Lage, Datentransfers zwischen Speicher und Interface eigenständig durchzuführen. Zu seinen Aufgaben gehören das Adressieren des Speichers einschließlich der Adreßfortschaltung, das Adressieren des Interface- oder Device-Controller-Datenregisters, das Durchführen der Buszyklen für die Lese- bzw. Schreibvorgänge und das Zählen der übertragenen Bytes. In einer erweiterten Betriebsart ist er in der Lage, mehrere aufeinanderfolgende Blockübertragungen durchzuführen, wobei die Speicherbereiche der Blöcke nicht zusammenhängend sein müssen.

Dem Mikroprozessor verbleiben aber neben dem „Programmieren“ des DMA-Controllers immer noch Aufgaben, wie das Initialisieren von Interfaces und Gerätesteueereinheiten, das Starten und Stoppen von Peripheriegeräten, das Ausführen spezieller Gerätelfunktionen, das Auswerten des Gerätestatus nach Abschluß einer Übertragung und ggf. eine Fehlerbehandlung. Hinzu kommen die Datenvor- und -nachbearbeitung, z.B. das Formatieren und Umcodieren von Daten. Soll er auch davon entlastet werden, muß ihm eine zusätzliche, programmierbare Steuereinheit, ein *Ein-/Ausgabeprozessor*, zur Seite gestellt werden. Das Problem hierbei ist, wie auch schon beim DMA-Controller, daß sich der Ein-/Ausgabeprozessor den Systembus mit dem zentralen Mikroprozessor teilen muß. Um auch diesbezüglich eine größere Unabhängigkeit zu erreichen, kann der Ein-/Ausgabeprozessor mittels eines lokalen Busses und eigenem, d.h. lokalem Speicher zum *Ein-/Ausgaberechner* erweitert werden.

8.1.1 Direktspeicherzugriff (DMA)

Vor Beginn einer Datenübertragung mit Direktspeicherzugriff wird der DMA-Controller, wie bereits erwähnt, vom Mikroprozessor durch Laden seiner Register (Steuer- und Adreßregister) initialisiert. Während dieser Phase wirkt der Controller als *Slave*, ist also passiver Busteilnehmer. Sobald er jedoch eine Übertragungsanforderung erhält und mit der Übertragung der Daten beginnt, führt er

eigenständig Buszyklen durch, d.h., dann arbeitet er wie der Prozessor als *Master* und teilt sich mit ihm den Bus für die Speicherzugriffe. Man spricht deshalb bei Einsatz eines DMA-Controllers auch von einem *Mehrmaster*-System. Die Übertragungen selbst finden wahlweise im Datenformat Byte, Halbwort oder Wort statt.

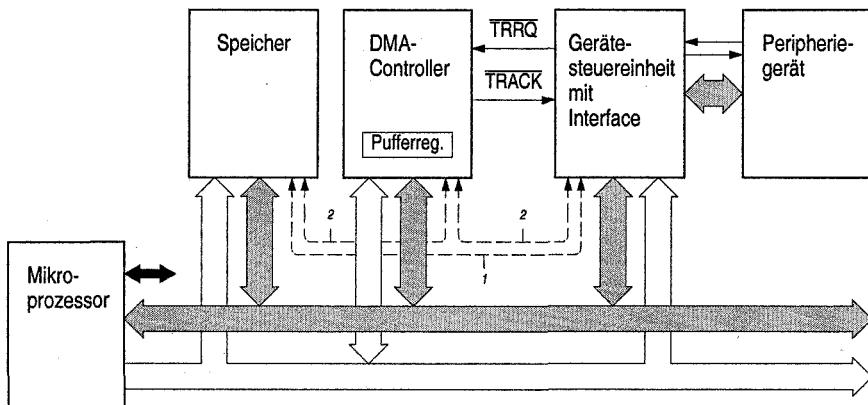


Bild 8-1. Direktspeicherzugriff mittels DMA-Controller, 1 direkte Übertragung, 2 indirekte Übertragung

Direkte und indirekte Übertragung (DMA-Zyklus). Die Übertragung der Daten erfolgt je nach Betriebsart des Controllers entweder direkt zwischen Speicher und Interface oder indirekt über ein für den Programmierer nicht sichtbares Pufferregister des DMA-Controllers (Bild 8-1). Bei der *direkten Übertragung* benötigt der DMA-Controller pro Datum nur einen Buszyklus, indem er den Speicher über den Adressbus und gleichzeitig das Interface-Datenregister über eine Steuerleitung adressiert (single address mode). Bei *indirekter Übertragung* führt der DMA-Controller zunächst einen Lesezugriff durch und speichert das Datum in seinem Pufferregister zwischen. In einem nachfolgenden Schreibzugriff transportiert er es dann zum Zielort. Speicher und Interface werden dabei über den Adressbus angewählt (dual address mode). Die indirekte Übertragung ist dadurch, daß sie zwei Buszyklen erfordert, langsamer als die direkte Übertragung, weist ihr gegenüber jedoch folgende Vorteile auf:

- Bei *Data-Misalignment* von Halbwort- und Wortoperanden im Speicher werden diese bei der Übernahme in das Pufferregister des DMA-Controllers ausgerichtet bzw. bei der Weitergabe aus dem Pufferregister mit Misalignment in den Speicher geschrieben (siehe auch 2.1.2 und 5.2.3). Das Pufferregister übernimmt hierbei das Sammeln von Bytes oder Halbwörtern (Assembly-Funktion) bzw. das Verteilen von Bytes oder Halbwörtern (Disassembly-Funktion).
- Unterschiedliche Torbreiten (port sizes) von Speicher und Interface können über die Assembly- und Disassembly-Funktion des Pufferregisters ausgeglichen werden (siehe auch 5.2.3).

- Speicher-zu-Speicher-Übertragungen sind möglich, womit z.B. Datenblöcke im Speicher verschoben, kopiert oder bei festgehaltener Quelladresse mit einem einheitlichen Wert initialisiert oder (falls der DMA-Controller diese Funktion vorsieht) durchsucht werden können.

Zum DMA-Zyklus siehe auch 5.1.4: Bild 5-7, S. 278.

Synchronisation und Übertragungssteuerung. Die Zeitpunkte der einzelnen Datentransporte werden üblicherweise von der Gerätesteuereinheit des Peripheriegeräts festgelegt und dem DMA-Controller (unmittelbar oder über einen Interface-Baustein) als Anforderungssignale TRRQ (transfer request) übermittelt (Bild 8-1). Diese Art der Anforderung ist bei am Systembus angeschlossenen Geräte- oder Peripheriebussteuereinheiten (z. B. IDE-Controller, SCSI-Host-Adapter) üblich. Für die Synchronisation von Speicher-zu-Speicher-Übertragungen und bei Geräten, die keine Anforderungssignale liefern, benutzt der DMA-Controller hingegen einen internen Taktgenerator (auto request).

Bei der direkten Übertragung zeigt der DMA-Controller den Beginn eines Buszyklus durch ein Quittungssignal TRACK (transfer acknowledge) an. Dieses wird in der Interface-Ansteuerlogik dazu genutzt, das Chip-Select- und die Register-Select-Signale für das Interface-Datenregister zu bilden. Damit ersetzt es sozusagen den zweiten benötigten Adreßbus. Bei der indirekten Übertragung ist diese besondere Art der Registeransteuerung nicht nötig. Hier werden ja, wie bereits gesagt, zwei „normale“ Buszyklen durchgeführt.

Die Synchronisation zwischen Mikroprozessor und DMA-Controller, d.h. die Synchronisation auf Blockebene, erfolgt über das Statusregister des Controllers. Dieses zeigt das Blockende durch Setzen eines Bits an, das entweder vom Prozessor abgefragt oder als Interruptanforderung ausgewertet wird. Der Prozessorzugriff auf das Statusregister und auch auf die anderen Register des Controllers ist während einer Blockübertragung immer dann möglich, wenn der Prozessor im Besitz des Busses und der DMA-Controller somit als Slave ansprechbar ist. Auf diese Weise kann der Prozessor auch den Ablauf des DMA-Controllers beeinflussen, z.B. vorzeitig abbrechen, indem er die Inhalte der Steuerregister verändert.

Zugriffsarten. Abhängig von der Zeitspanne, in der der Mikroprozessor durch den DMA-Vorgang am Systembuszugriff gehindert wird, unterscheidet man zwei Arten des Direktspeicherzugriffs.

- Beim *Vorrangmodus* (*cycle-steal mode*) belegt der DMA-Controller den Systembus jeweils für die Zeit der Übertragung eines einzelnen Datums, indem er dem Mikroprozessor sozusagen einen Buszyklus stiehlt. In Wirklichkeit ist die Zeit um einige Maschinentakte länger, um die Busanforderung und die Busfreigabe mit dem Mikroprozessor zu synchronisieren. Der Cycle-steal-Modus wird bei relativ langsamem Datenübertragungen angewendet.

- Beim *Blockmodus (burst mode)* belegt der DMA-Controller den Systembus für die Gesamtdauer der Übertragung eines Datenblocks, wodurch der Mikroprozessor für einen längeren Zeitraum an der Benutzung des Systembusses gehindert wird. Der Blockmodus wird bei schnellen Datenübertragungen eingesetzt.

DMA-Kanäle. DMA-Controller können üblicherweise mehrere DMA-Vorgänge quasi-gleichzeitig bearbeiten. Sie haben dazu bis zu acht DMA-Kanäle, d.h. bis zu acht gleichartige Registersätze und Peripherieschnittstellen bei einem gemeinsamen Steuerwerk für die Datenübertragung. Wegen des gemeinsamen Steuerwerks ist eine Abstufung der Kanäle nach Prioritäten erforderlich. Sie erfolgt durch aufeinanderfolgende Kanalnummern, wobei z.B. derjenige Kanal, der die niedrigste Priorität haben soll, über ein Steuerregister festgelegt wird. Um eine faire, d.h. gleichmäßige Zuteilung des Steuerwerks und damit des Buszugriffs an die Kanäle zu erreichen, können die Prioritäten auch automatisch verändert werden. Dabei tauscht der DMA-Controller die Prioritäten nach jeder Kanalaktivierung zyklisch aus und weist dem jeweils zuletzt aktiven Kanal die niedrigste Priorität zu. Man spricht von *rotierenden Prioritäten* (rotating priorities).

Blockverkettung. Bei der einfachen Blockübertragung muß der DMA-Controller für jede Blockübertragung durch den Prozessor neu initialisiert werden. Bei der Übertragung verketteter Blöcke hingegen werden für den DMA-Controller von vornherein die Basisadressen und Blocklängen mehrerer aufeinanderfolgend zu übertragenden Blöcke bereitgestellt. Diese Information wird entweder in einer Liste im Speicher abgelegt, deren Anfangsadresse und Länge bei der Initialisierung in zwei Register des DMA-Controllers geladen werden (array chaining), oder die Einzelblockinformation wird an den jeweiligen Vorgängerblock angehängt und der letzte Block mit einer Endeinformation versehen (linked chaining). In beiden Fällen liest der Controller nach Abschluß einer Blockübertragung die jeweilige Adress- und Längenangabe des nächsten Blocks selbst. In einer weiteren Betriebsart führt der DMA-Controller wiederholte Blockübertragungen mit einem festen Ein-/Ausgabebereich durch, z.B. mit einem festen Ein-/Ausgabepufferbereich. In diesem Fall muß nur einmal eine Basisadresse und eine Blocklänge vorgegeben werden.

8.1.2 DMA-Controller-Baustein

Im folgenden werden der prinzipielle Aufbau und die Funktionsweise eines DMA-Controller-Bausteins betrachtet, wobei wir uns, um die Darstellung übersichtlich zu halten, auf die wesentlichen Betriebsmerkmale beschränken. Für einen Einblick in weitere Details sei auf die Datenblätter der Bausteinhersteller verwiesen.

Blockstruktur. Bild 8-2 zeigt das Blockschaltbild des DMA-Controller-Bausteins mit zwei Kanälen A und B. Jeder Kanal hat zwei 8-Bit-Steuerregister CR1 und CR2 und ein 8-Bit-Statusregister SR, ferner zwei 32-Bit-Adressregister MAR (memory address register) und IAR (interface address register) zur Speicher- bzw. Interface-Adressierung, ein 32-Bit-Bytezählregister BCR (byte count register) für die Blockverwaltung und ein nicht explizit adressierbares 32-Bit-Pufferregister PR für die Zwischenspeicherung von bis zu vier Bytes, zwei Halbwörtern oder eines Wortes.

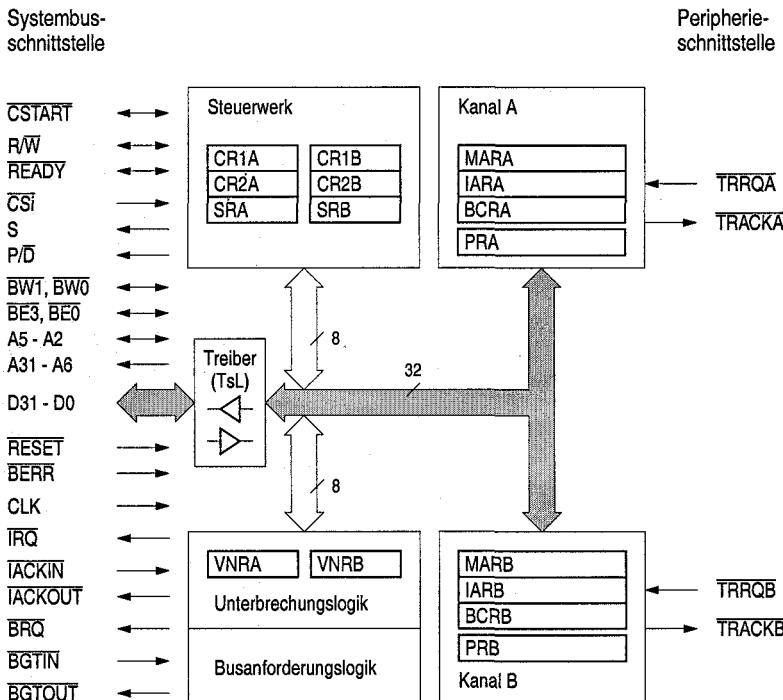


Bild 8-2. DMA-Controller-Baustein mit zwei Kanälen

Beide Kanäle haben eine gemeinsame Unterbrechungseinrichtung zur Erzeugung eines Interruptsignals **IRQ** mit Selbstidentifizierung der Kanäle (Vektornummerregister **VNR**) sowie zur Priorisierung der Interruptanforderungen (**IACK**-Verkettung als Daisy-Chain). Hinzu kommt eine Einrichtung zur Erzeugung des Busanforderungssignals **BRQ** und zur Priorisierung der Busanforderung des Bausteins im System (**BGT**-Verkettung in einer Daisy-Chain). Bezuglich der Interruptbehandlung und der Buszuteilung hat Kanal A durch interne Festlegung höhere Priorität als Kanal B.

Die Systembusschnittstelle umfaßt einen 32-Bit-Datenbusanschluß, einen 32-Bit-Adressebusanschluß (A1 und A0 sind in **BE3** bis **BE0** enthalten) und die für den

Zugriff auf seine Register (Slave-Funktion) und zur Steuerung seiner eigenen Buszyklen (Master-Funktion) erforderlichen Steuersignale (siehe auch 5.2 und 5.3). Aufgrund der Master-/Slave-Funktion sind ein Teil dieser Signalanschlüsse wie auch die Adressanschlüsse A5 bis A2 und die Byte-Enable-Anschlüsse \overline{BE}_3 bis \overline{BE}_0 bidirektional ausgelegt. Die Schnittstelle umfaßt außerdem die für die Interruptverarbeitung und für die Busarbitration erforderlichen Anschlüsse sowie einen Initialisierungseingang \overline{RESET} , einen Bus-Error-Eingang \overline{BERR} und einen Takteingang CLK für die bausteininternen Steuerungsabläufe und den Auto-Request-Taktgenerator. – Die peripheren Schnittstellen beider Kanäle haben je einen Signaleingang \overline{TRRQ} (transfer request) zur Entgegennahme von Übertragungsanforderungen und einen Signalausgang \overline{TRACK} (transfer acknowledge) zur Anwahl des Interface-Datenregisters bei direkter Übertragung.

Funktionsweise. Der DMA-Controller führt die blockweise Datenübertragung wahlweise indirekt oder direkt sowie im Cycle-steal- oder Blockmodus durch. Die einzelnen Datentransporte werden dabei entweder über den \overline{TRRQ} -Eingang oder durch den internen Taktgenerator ausgelöst. Bei indirekter Übertragung werden die Quelladresse und die Zieladresse von den beiden Adressregistern MAR und IAR sowie die Transportrichtung durch ein Steuerbit vorgegeben; bei direkter Übertragung ist IAR ohne Funktion. – Der Datentransport erfolgt abhängig von zwei weiteren Steuerbits byte-, halbwort- oder wortweise, wobei die indirekte Übertragung für Quelle und Ziel Torbreiten zuläßt, die von diesen Datenformaten unabhängig sind. Die in MAR stehende Speicheradresse wird mit jedem Datentransport in Abhängigkeit des Datenformats um Eins, Zwei oder Vier erhöht oder vermindert. Die in IAR stehende Speicheradresse kann in gleichen Schritten hochgezählt werden (Speicher-Speicher-Transfer), sie kann aber auch festgehalten werden (Speicher-Interface-Transfer).

Die Blockverwaltung und die Synchronisation des Prozessors mit dem DMA-Controller geschieht durch Bytezählung im *Bytezählregister BCR*. Dieses wird bei der Initialisierung mit der Anzahl der zu übertragenden Bytes geladen; mit jedem Datentransport wird sein Inhalt abhängig vom Datenformat um Eins, Zwei bzw. Vier vermindert. Erreicht der Bytezähler den Wert Null (Blockende), so wird ein Statusbit gesetzt. Dieses Bit wirkt entweder als Interruptsignal, oder es wird vom Mikroprozessor abgefragt.

Bild 8-3 zeigt das 8-Bit-Steuerregister CR1 mit der Funktion der einzelnen Bits. Mit dem Steuerbit START wird der Controller gestartet, mit ABORT kann die Übertragung vorzeitig abgebrochen werden. O/I gibt die Richtung der Übertragung vor. Daraus abgeleitet werden die R/W-Signale für die Speicher- und die Interface-Ansteuerung. Mit den Steuerbits D/I und C/B werden die direkte oder indirekte Übertragung bzw. die Zugriffsart Cycle-steal- oder Blockmodus ausgewählt. Das Bit AREQ legt fest, ob die Übertragungsanforderungen durch den internen Taktgenerator (auto request) oder über den \overline{TRRQ} -Eingang erzeugt wer-

den. Das Interrupt-Enable-Bit IRE erlaubt das Sperren bzw. Freigeben von Interrupts, die durch die Bits BE und ERR im Statusregister ausgelöst werden.

CR17	CR16	CR15	CR14	CR13	CR12	CR11	CR10	
IRE	0	AREQ	C/ \bar{B}	D/ \bar{I}	O/ \bar{I}	ABORT	START	Steuerregister CR1
								Starten: 0 inaktiv, 1 aktiv
								Abbruch: 1 Übertragung wird abgebrochen
								Übertragungsrichtung: 0 Eingabe, 1 Ausgabe
								Übertragungsart: 0 indirekt, 1 direkt
								Zugriffsart: 0 Blockmodus, 1 Cycle-steal-Modus
								Übertragungsanforderung: 0 Extern-Request, 1 Auto-Request
unbenutzt								
Interrupts: 0 blockiert, 1 zugelassen								

Bild 8-3. Steuerregister 1 des DMA-Controllers nach Bild 8-2

Bild 8-4 zeigt das 8-Bit-Steuerregister CR2. Die Steuerbits DF1 und DF0 bestimmen das Datenformat Byte, Halbwort oder Wort. MU/ \bar{D} legt die Speicheradressierung durch das Adressregister MAR als aufwärts- oder abwärtszählend fest; IU/ \bar{F} gibt für das Interface-Adressregister IAR eine aufwärtszählende oder feste Adressierung vor. Die restlichen vier Steuerbits bestimmen die Zugriffsattribute Supervisor/User und Program/Data für die in MAR und IAR stehenden Adressen (siehe auch 6.3.4). Sie werden vom DMA-Controller während der Speicher- und Interface-Zugriffe als Statussignale S und P/ \bar{D} an den Systembus ausgegeben.

CR27	CR26	CR25	CR24	CR23	CR22	CR21	CR20	
IS/ \bar{U}	IP/ \bar{D}	MS/ \bar{U}	MP/ \bar{D}	IU/ \bar{F}	MU/ \bar{D}	DF1	DF0	Steuerregister CR2
								Datenformat: 00 Wort, 10 Halbwort, 01 Byte
								Adresszählung (MAR): 0 abwärts, 1 aufwärts
								Adresszählung (IAR): 0 festgehalten, 1 aufwärts
								Zugriffsattribut 2 (MAR): 0 Data, 1 Program
								Zugriffsattribut 1 (MAR): 0 User, 1 Supervisor
								Zugriffsattribut 2 (IAR): 0 Data, 1 Program
								Zugriffsattribut 1 (IAR): 0 User, 1 Supervisor

Bild 8-4. Steuerregister 2 des DMA-Controllers nach Bild 8-2

Bild 8-5 zeigt das 8-Bit-Statusregister SR, in dem lediglich drei Bits mit Funktionen belegt sind; die restlichen Bits sind beim Lesen des Statusbytes auf Null festgelegt. Das Blockendebit BE wird gesetzt, wenn das Bytezählregister den Wert Null erreicht hat. Das Fehlerstatusbit ERR zeigt eine fehlerhafte Datenübertragung an, so z.B. das Ausbleiben des Signals READY, was durch den Bus-Error-Eingang BERR signalisiert wird, oder das vorzeitige Abbrechen einer Übertragung durch Setzen des Steuerbits ABORT. BE und ERR wirken, sofern das Interrupt-Enable-Bit im Steuerregister gesetzt ist, als Interruptanforderungen. Sie werden zurückgesetzt, indem das Statusregister mit einer Maske beschrieben wird, die an der entsprechenden Bitposition eine Eins aufweist. Das B/R-Bit zeigt an, ob der DMA-Kanal gerade mit einer Blockübertragung beschäftigt ist (busy), oder ob er für eine neue Blockübertragung bereit ist (ready).

SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	
BE	ERR	B/ \bar{R}	0	0	0	0	0	Statusregister SR
								konstant 0

Betriebszustand: 0 bereit, 1 beschäftigt

Fehlerstatus: 0 kein Fehler, 1 fehlerhaft

Bild 8-5 Statusregister des DMA-Controllers nach Bild 8-2

Beispiel 8.1. ▶ Datenausgabe mit Direktspeicherzugriff. In Anlehnung an Beispiel 7.3, S. 479, sollen 128 Datenbytes von einem DMA-Controller über ein Parallel-Interface aus einem Speicherbereich BUFFER (user data) an ein Peripheriegerät (supervisor data) ausgegeben werden. Das Peripheriegerät ist zuvor über die Datenleitung PDA0 des Interfaces zu starten ($PDA0 = 1$); die Übertragung ist indirekt und im Cycle-steal-Modus bei aufwärtszählender Speicheradresse durchzuführen. Mit Abschluß der Datenausgabe soll der DMA-Controller eine Programmunterbrechung auslösen. Im Interruptprogramm sind das Peripheriegerät zu stoppen ($PDA0 = 0$), das Interface und der DMA-Controller zu inaktivieren, der Fehlerstatus im DMA-Controller abzufragen und die Interruptbits im DMA-Controller zurückzusetzen.

Bild 8-6 zeigt den Schaltungsaufbau mit dem oben beschriebenen DMA-Controller und einem Parallel-Interface-Baustein entsprechend 7.3.2. Die Synchronisation für die byteweise Übertragung zwischen Interface und Peripheriegerät erfolgt wie im Beispiel 7.3 durch die Handshake-Signale C1B und C2B. Das C1B-Signal hat jetzt jedoch die Funktion, den DMARQB-Ausgang zu aktivieren. Dieser wirkt auf den TRRQ-Eingang von Kanal A des DMA-Controllers. – Der DMA-Controller teilt sich den Buszugriff mit dem Mikroprozessor, dem die *Busarbitration* obliegt. (Da keine weiteren Busmaster im System vorhanden sind, ist eine prozessorexterne Busarbeiterlogik nicht erforderlich.) Interruptanforderungen des DMA-Controllers werden dem Prozessor über einen Prioritätencodierer zugeführt (siehe 5.5.1).

Bild 8-7 zeigt die Initialisierung der Datenübertragung und das Interruptprogramm. Zur Initialisierung lädt der Mikroprozessor die Register des DMA-Kanals A: das Speicheradreßregister MARDMA mit der Bereichsadresse BUFFER, das Interface-Adreßregister IARDMA mit der Adresse des Interface-Datenregisters DRA, das Bytezählregister BCRDMA mit der Byteanzahl

128, das Vektornummerregister VNRDMA mit der Vektornummer 65 und die beiden Steuerregister CR2DMA und CR1DMA mit den Steuerbytes 0x85 und 0x95 für die indirekte Ausgabe von Bytes im Cycle-steal-Modus bei aufwärtszählender Speicheradressierung und festgehaltener Interface-Adresse. Darüber hinaus werden die Zugriffsattribute für die Speicher- und Interface-Zugriffe vorgegeben und das Auto-Request-Bit AREQ auf Null gesetzt, so daß der DMA-Controller von der Peripherie über den DMARQA-Ausgang des Interfaces aktiviert werden kann. Außerdem wird das Interrupt-Enable-Bit IRE gesetzt, um den Blockende- und den Fehler-Interrupt freizugeben, und das START-Bit gesetzt, um den Controller für die Übertragung zu starten.

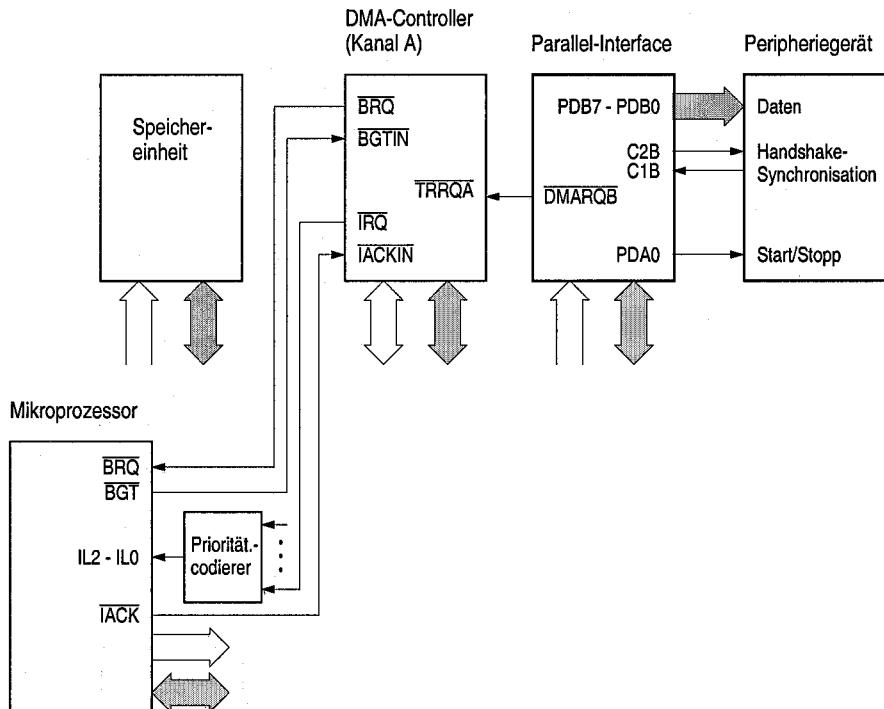


Bild 8-6. Schaltungsaufbau für eine Datenübertragung mit DMA-Controller und Parallel-Interface

Nach dem DMA-Controller wird das Interface gemäß Beispiel 7.3 initialisiert, wobei hier C1B für den DMA-Betrieb festgelegt wird. Danach wird die Datenausgabe durch Starten des Peripheriegeräts eingeleitet. Der Prozessor lädt dazu das Interface-Datenregister DRA mit 0x01 und setzt so den Datenausgang PDA0 auf Eins. Das Peripheriegeräts zeigt seine Bereitschaft durch Aktivieren des C1B-Signals an und löst damit über die DMARQB-TRRQA-Verbindung im DMA-Controller die erste Byteübertragung aus.

Der DMA-Controller aktiviert vor jeder Byteübertragung seinen \overline{BRQ} -Ausgang und fordert damit den Systembus vom Mikroprozessor an. Der Mikroprozessor beendet zunächst seinen gegenwärtigen Buszyklus, schaltet dann seine Tristate-Ausgänge in den hochohmigen Zustand und zeigt die Busfreigabe durch Setzen des \overline{BGT} -Signals am \overline{BGTIN} -Eingang des DMA-Controllers an (siehe auch 5.4.1). Der DMA-Controller übernimmt daraufhin den Bus und führt einen Lesezyklus mit dem Speicher aus. Er speichert das Datenbyte in seinem Pufferregister und überträgt es

in einem anschließenden Schreibzyklus in das Datenregister DRA des Interfaces. Speicher und Interface werden dabei mit den Inhalten der Adreßregister des Kanals A adressiert.

Mit Abschluß der Byteübertragung setzt der DMA-Controller sein BRQ-Signal zurück, worauf der Mikroprozessor das BGT-Signal inaktiviert und seinerseits den Systembus wieder übernimmt. Gleichzeitig erhöht der Controller den Inhalt seines Speicheradreßregisters um Eins und vermindert den Bytezählerstand um Eins. Wird schließlich im Bytezählregister der Wert Null erreicht, so setzt der DMA-Controller das BE-Bit in seinem Statusregister und löst damit den Blockende-Interrupt aus.

```
; Parallele Ausgabe von 128 Bytes mit einem DMA-Controller
; Programmunterbrechung bei Blockende
; Hauptprogramm

    ORG      0x40000
BUFFER  DS.B   128           ; Ausgabepuffer
VECTAB EQU     0           ; Vektortabellenbasis
LEA      IOEND, VECTAB+4*65 ; Vektortabelleneintrag

; Initialisieren des DMA-Controllers
    LEA      BUFFER, MARDMA  ; Speicheradresse
    LEA      DRA, IARDMA    ; Interface-Adresse
    MOVE.W #128, BCRDMA    ; Byteanzahl
    MOVE.B #0x85, CR2DMA   ; Steuerbyte 2
    MOVE.B #0x95, CR1DMA   ; Steuerbyte 1
    MOVE.B #65, VNRDMA     ; Vektornummer

; Initialisieren des Parallel-Interfaces und
; Starten des Peripheriegeräts
    MOVE.B #0xB0, CRB       ; Steuerbyte Port B
    MOVE.B #0xFF, DDRB      ; Richtung Port B
    MOVE.B #0x00, CRA       ; Steuerbyte Port B
    MOVE.B #0x01, DDRA      ; Richtung Port B
    MOVE.B #0x01, DRA       ; Gerät starten
    :

; Datenausgabe beendet oder abgebrochen
; Interruptprogramm

IOEND: MOVE.B #0x00, DRA      ; Gerät stoppen
       MOVE.B #0x00, CRB      ; Parallel-Interf. inaktivieren
       MOVE.B #0x00, CR1DMA   ; DMA-Controller inaktivieren
       BTST.B #6, SRDMA      ; Fehlerstatus im DMAC abfragen
       BEQ     RET
; Fehlerbehandlung
       :
RET:   MOVE.B #0xC0, SRDMA   ; Interruptbits im DMAC löschen
       RTE
```

Bild 8-7. Haupt- und Interruptprogramm zu Beispiel 8.1: *Datenausgabe mit Direktspeicherzugriff*

Im zugehörigen Interruptprogramm wird die Datenübertragung abgeschlossen. Dazu stoppt der Prozessor zunächst das Peripheriegerät, indem er 0x00 in das Interface-Datenregister DRA schreibt (PDA0 = 0), und lädt dann die Steuerregister des Interfaces und des DMA-Controllers ebenfalls mit 0x00, womit er beide Bausteine inaktiviert. Da auch das Fehlerstatusbit ERR des DMA-Controllers zu einer Programmunterbrechung geführt haben kann, wird es abgefragt und

ggf. eine Fehlerbehandlung durchgeführt. Vor Verlassen des Interruptprogramms werden die beiden Statusbits BE und ERR in einem Schreibzugriff durch die Maske 0xC0 angesprochen und, sofern sie gesetzt sind, zurückgesetzt. ▲

8.1.3 Ein-/Ausgabeprozessor

Ein *Ein-/Ausgabeprozessor*, früher auch als *Ein-/Ausgabekanal* bezeichnet, ist in der Lage, ein im Speicher bereitgestelltes Ein-/Ausgabeprogramm (*Kanalprogramm*) abzurufen und auszuführen. Bei einfacheren Ein-/Ausgabeprozessoren besteht ein solches Programm aus speziellen Kommandowörtern, mit denen Interfaces und Gerätesteuereinheiten initialisiert, gerätespezifische Steueroperationen ausgeführt, der Status dieser Einheiten abgefragt und Datenübertragungen durch den Prozessor gesteuert werden. Unter Verwendung von Verzweigungsbefehlen können dabei Statusbedingungen ausgewertet und so z.B. Abläufe wiederholt werden. Heutige, universelle Ein-/Ausgabeprozessoren entsprechen hingegen in ihrem Befehlsvorrat und den verfügbaren Adressierungsarten den universellen Mikroprozessoren, so daß sie wie diese universell programmiert werden können. Im Unterschied zu Universalprozessoren verfügen sie jedoch über zusätzliche Funktionseinheiten zur Unterstützung der Ein-/Ausgabe, insbesondere über meist mehrere DMA-Controller bzw. DMA-Kanäle. Die Struktur eines solchen Ein-/Ausgabeprozessors hoher Leistungsfähigkeit ist im folgenden Abschnitt in Bild 8-9 (S. 557) gezeigt. – Anstatt eines speziellen Ein-/Ausgabeprozessors kann natürlich auch ein herkömmlicher universeller Mikroprozessor eingesetzt werden und dieser durch DMA-Controller-Bausteine in seiner Funktion ergänzt werden.

8.1.4 Ein-/Ausgaberechner

Bild 8-8a zeigt die Grobstruktur eines *Mehrmastersystems* mit einem Mikroprozessor für die zentralen Verarbeitungsaufgaben (CPU) und einem Ein-/Ausgabeprozessor mit weitgehend universellen Fähigkeiten (IOP). In diesem System teilt sich der Ein-/Ausgabeprozessor den Systembus mit dem Mikroprozessor und hat damit wie dieser Zugriff auf die an den Bus angeschlossenen Funktionseinheiten, d.h. auf den Speicher und die Ein-/Ausgabeeinheiten. Die Benutzung dieser gemeinsamen Betriebsmittel (Ressourcen) erfordert eine Synchronisation der Zugriffe beider Prozessoren. Für den Buszugriff erfolgt sie über die *Busarbitration*, d.h. durch die Hardware (siehe 5.4), für den Zugriff auf gemeinsam benutzte Speicherbereiche (Ein-/Ausgabedatenpuffer, Kanalprogrammbereiche) erfolgt sie über Synchronisationsvariablen, sog. *Semaphore*, d.h. durch die Software. Die Verwaltung von Semaphoren wird durch spezielle Mikroprozessorbefehle, wie TAS oder swap, unterstützt (siehe 2.1.4 bzw. 2.2.4 und Beispiel 8.2, S. 558).

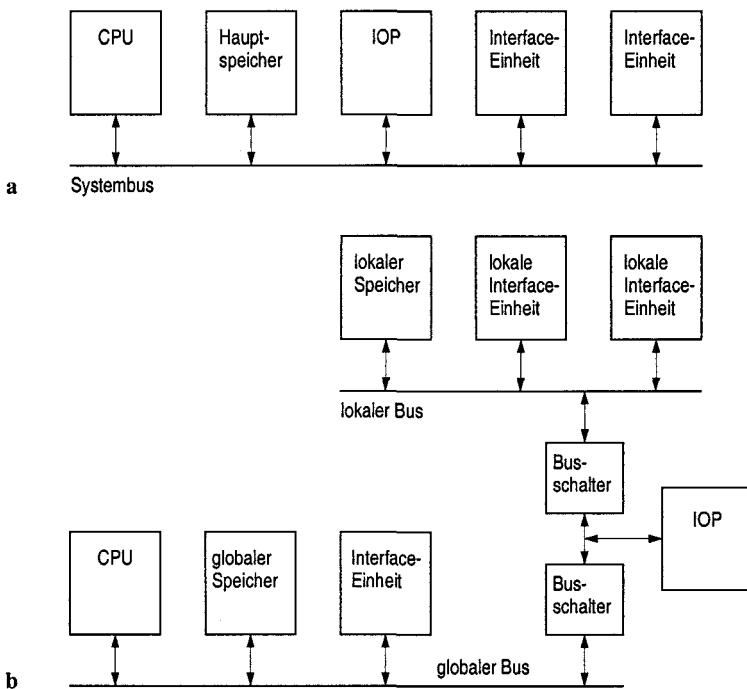


Bild 8-8. Mehrmastersystem mit einem Zentralprozessor (CPU) und einem Ein-/Ausgabeprozessor (IOP), a mit gemeinsamem Systembus für die CPU und den IOP, b erweitert um einen lokalen Bus und einen lokalen Programm- und Datenspeicher für den IOP (Ein-/Ausgaberechner)

Das *Einbussystem* nach Bild 8-8a hat den bekannten Nachteil, daß es zu Buskonflikten kommen kann, indem sich die Master in ihren Buszugriffen gegenseitig behindern. Abhilfe schafft hier, wie bereits in 5.1.3 ausgeführt, ein *Mehrbus-system*, wie z.B. in Bild 8-8b gezeigt. Hier hat der Ein-/Ausgabeprozessor einen eigenen, *lokalen Bus*, an den einerseits die von ihm verwalteten Interface-Einheiten angeschlossen sind und über den er andererseits auf einen eigenen, lokalen Programm- und Datenspeicher zugreifen kann. Das Ein-/Ausgabesystem wird auf diese Weise zum selbständigen *Ein-/Ausgaberechner*. Der Ein-/Ausgabeprozessor führt jetzt seine Programmspeicher- und Interface-Zugriffe lokal durch und belastet mit diesen den Systembus nicht. Buskonflikte sind jedoch nach wie vor beim Zugriff auf den am *globalen Bus* angeschlossenen, gemeinsam benutzten Speicher möglich, der neben den Programmen und Daten des Mikroprozessors auch die für beide Prozessoren gemeinsamen Ein-/Ausgabedatenbereiche (*shared memory*) enthält.

Bild 8-9 zeigt die Internstruktur eines *Ein-/Ausgabeprozessors*, des i960 von Intel. Um nicht auf die Vielfalt der im Bild angedeuteten Funktionen einzugehen, seien nur einige davon aufgezählt. So hat der Prozessor einen chip-internen, lokalen Bus, der auch nach außen verfügbar ist und für den Anschluß des lokalen

Speichers genutzt wird. Unterstützt wird der Speicheranschluß durch einen On-chip-memory-Controller. Der Zugang zum globalen Bus erfolgt über eine PCI-Bus-Schnittstelle, d.h., der Ein-/Ausgabeprozessor ist für PCI-Bus-basierte Systeme ausgelegt. Dieser Bus wird als Primary PCI-Bus bezeichnet. Eine interne *PCI-to-PCI-Bridge* stellt darüber hinaus eine zweite PCI-Bus-Schnittstelle zur Verfügung, die auf einen lokalen, sog. Secondary PCI-Bus führt. Dieser wird für den Anschluß der lokalen Interfaces genutzt. Die in Bild 8-8b gezeigten Bus-schalter sind dementsprechend durch die PCI-to-PCI-Bridge on-chip realisiert. Signifikant für einen Ein-/Ausgabeprozessor sind außerdem die beiden DMA-Controller, je einer für jeden der beiden PCI-Busse, wobei der DMA-Controller des Primary PCI-Busses zwei Kanäle aufweist.

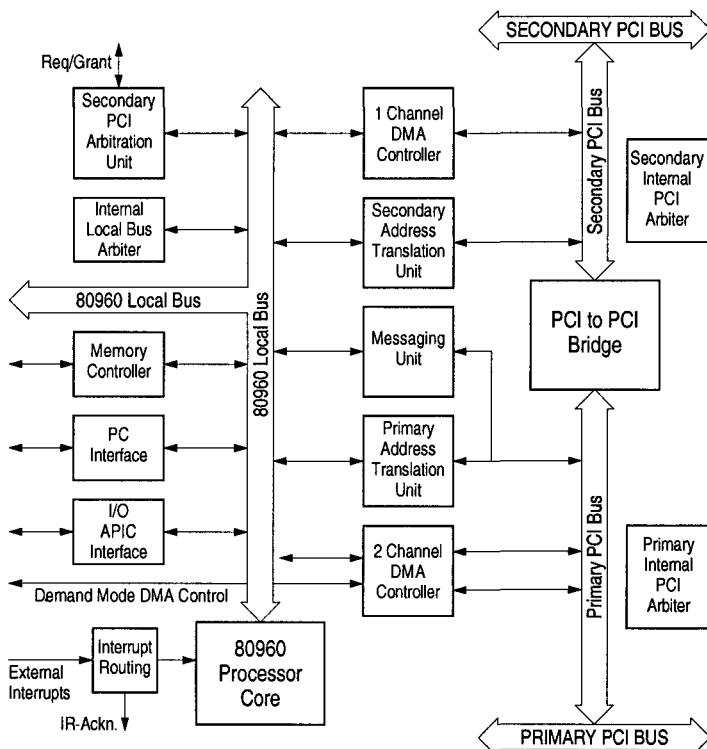


Bild 8-9. Struktur des Ein-/Ausgabeprozessors i960 von Intel (in Anlehnung an [Intel 1997])

Bild 8-10 zeigt die Einbindung des i960-Prozessors in ein PCI-Bus-System, das auf der Einprozessorstruktur nach Bild 5-8 (S. 288) basiert. Der oben erwähnte Zugriffskonflikt auf den Hauptspeicher als dem globalen Speicher wird dadurch entschärft, daß der Mikroprozessor einen Großteil seiner Zugriffe über seine L1- und L2-Caches abwickelt und so der primäre PCI-Bus für die sonstigen Busaktivitäten gut verfügbar ist.

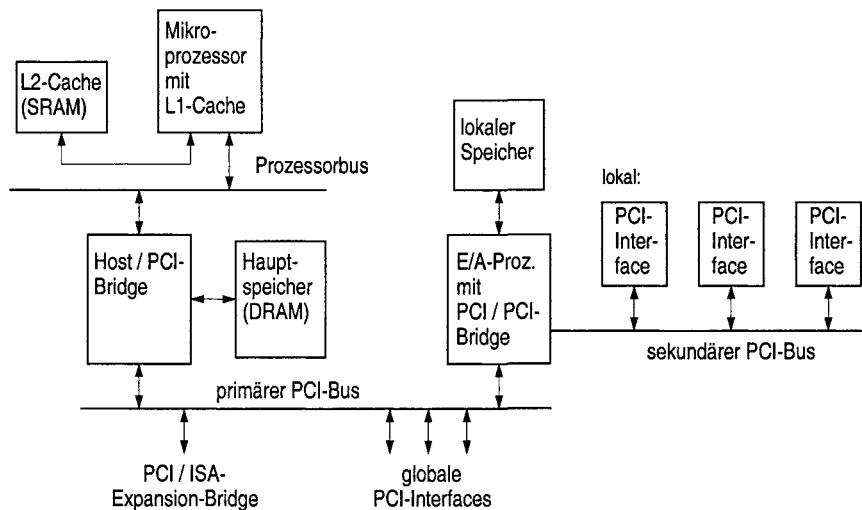


Bild 8-10. Mehrmastersystem mit dem i960 auf der Basis von PCI-Bussen, ausgehend von der Systemstruktur Bild 5-8 (S. 288)

Beispiel 8.2. ► Synchronisation von Prozessorzugriffen. In einem Mehrmaster- oder einem Mehrrechnersystem entsprechend den Bildern 8-8 und 8-10, erweitert um einen zweiten Mikroprozessor, sind im Auftrag von einem der beiden Mikroprozessoren vom DMA-Controller des Ein-/Ausgabeprozessors mehrere Datenblöcke in den gemeinsamen Speicher einzulesen. Der verfügbare Eingabepuffer im Speicher hat dabei gerade die Größe eines Blocks, d.h., er wird wiederholt gefüllt. Um dabei keine Daten zu überschreiben, ist er nach jedem Füllvorgang vom auftraggebenden Mikroprozessor zu leeren. Der zweite Mikroprozessor ist während des gesamten Eingabevorgangs vom Zugriff auf den Eingabepuffer auszuschließen. Generell soll jedoch auch er die Möglichkeit haben, diesen Puffer für Eingaben benutzen zu können; er muß dazu in der Lage sein, seinerseits den anderen Prozessor vom Pufferzugriff auszuschließen.

Der *gegenseitige Ausschluß* (*mutual exclusion*) und die Synchronisation der wechselnden Zugriffe auf den Eingabepuffer werden durch Semaphore realisiert. Als *Semaphor* bezeichnet man eine Variable mit Signalfunktion, auf die die Operationen „Sperre Semaphor“ und „Entsperre Semaphor“ anwendbar sind. Dargestellt wird ein Semaphor z.B. durch das höchstwertige Bit einer Bytevariablen, deren symbolische Adresse die Semaphorvariable bezeichnet. Die Operation „Entsperre S“ setzt das Bit 7 der Semaphorvariablen S auf Null und gibt damit das dem Semaphor zugeordnete Betriebsmittel frei (hier den Eingabepuffer). Die Operation „Sperre S“ fragt das Bit 7 der Semaphorvariablen ab: hat es den Wert Eins (Zugriff blockiert), so wird die Abfrage wiederholt; hat es den Wert Null (Zugriff erlaubt), so wird der Zugriff für andere Anfrager blockiert, indem das Semaphor auf Eins gesetzt wird. Voraussetzung für das Funktionieren der Sperre-Operation ist, daß das Lesen, das Verändern und das Rückschreiben des Semaphors in einer nicht unterbrechbaren Folge ausgeführt wird. Auf der Maschinenebene wird das durch den in 2.1.4 beschriebenen TAS-Befehl gewährleistet.

Entsperre S: MOVE.B #0,S S<7> := 0

Sperre S:	WAIT: TAS.B	S	WAIT: if S<7> = 1 then N := 1
	BMI	WAIT	else N := 0, S<7> := 1;
			if N = 1 then goto WAIT

Bild 8-11 zeigt den Eingabevorgang als Flußdiagramm. Der gegenseitige Ausschluß der beiden Mikroprozessoren auf den Eingabepuffer erfolgt über die Semaphorvariable EXCL1. Sie wird vom auftraggebenden Mikroprozessor vor Beginn des Eingabevorgangs gesetzt (sofern oder sobald sie entsperrt ist) und wird mit dessen Abschluß wieder zurückgesetzt. Für die Synchronisation der Pufferzugriffe zwischen dem Mikroprozessor und dem Ein-/Ausgabeprozessor werden die beiden Variablen EMPTY und FULL verwendet. Mit „Entsperrre EMPTY“ signalisiert der Mikroprozessor, daß der Puffer leer ist und er den Zugriff darauf freigibt; entsprechend meldet der Ein-/Ausgabeprozessor mit „Entsperrre FULL“, daß der Puffer voll ist und geleert werden kann.

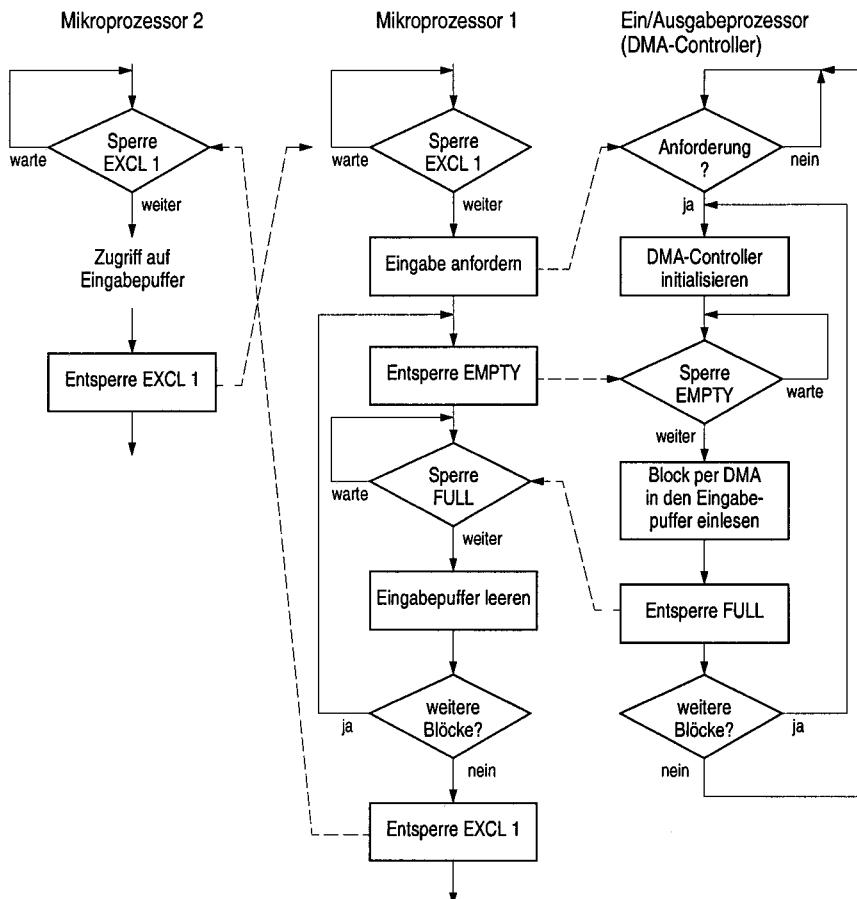


Bild 8-11. Synchronisation zwischen zwei Mikroprozessoren und einem Ein-/Ausgabeprozessor bei einem Eingabevorgang mit mehrfachem Füllen und Leeren eines Eingabepuffers

Die zugehörigen Sperre-Operationen werden, anders als bei EXCL, vom jeweiligen Synchronisationspartner ausgeführt. Dadurch entsteht eine gegenseitige Abhängigkeit, die die Reihenfolge der beiden Vorgänge festlegt. – Zur Darstellung der Sperre-Operation wird in Bild 8-11 das Abfragesymbol verwendet. Dabei ist zu beachten, daß die Operation zusätzlich zum Abfragen auch das Setzen des Semaphorbits enthält. Im Bild wird außerdem vorausgesetzt, daß die beiden Variablen EMPTY und FULL zuvor durch die Operation „Sperre“ initialisiert wurden. ◀

8.2 Periphere Punkt-zu-Punkt-Verbindungen und Peripheriebusse

Für Peripheriegeräte, wie Hintergrundspeicher (z.B. Festplatte, DVD/CD-Laufwerk, Streamer) und Ein-/Ausgabegeräte (z.B. Drucker, Scanner), sind derzeit zwei verschiedene Arten des Anschlusses an den Systembus gebräuchlich:

- Das Gerät hat eine eigene *Punkt-zu-Punkt-Verbindung* zum Systembus und benötigt dafür am Systembus ein eigenes Interface oder eine eigene Steuereinheit. Man spricht hier auch von einer Geräteschnittstelle. Beispiele hierfür sind die parallele IDE-Schnittstelle (zunächst mit einem einfachen passiven Interface versehen, später dann mit einem IDE/ATA-Controller) und ihr serieller Nachfolger, Serial ATA. – Die überbrückbaren Entfernung sind bei dieser Verbindungsart gering, weshalb sie im Grunde nur für in das Rechnergehäuse eingebaute Geräte zum Einsatz kommt.
- Mehrere Geräte werden an einem *Peripheriebus* zusammengefaßt und dieser mittels einer Bussteuereinheit, einem sog. Bus- oder *Host-Adapter* an den Systembus angebunden. Die Struktur des Busses wird dabei üblicherweise durch Ketten der Geräte mittels einzelner steckbarer Kabel realisiert. Typisches Beispiel ist hier der parallele SCSI-Bus. In Abwandlung des Busprinzips kann die Verkettung aus Punkt-zu-Punkt-Verbindungen zwischen den Geräten bestehen, d.h., die zu übertragende Information wird von Gerät zu Gerät in der Kette weitergereicht. Beispiele hierfür sind die moderneren, seriellen Peripheriebusse USB und FireWire. Bei beiden Busarten spricht man von einem *Kabelbus* als *Strang*. – Diese Art der Verbindung lässt größere Entfernung überbrücken, so daß auch rechnerexterne Geräte angeschlossen werden können.

Künftig wird es eine weitere Art des Geräteanschlusses geben, die im Zusammenhang mit der zunehmenden Serialisierung der Übertragung sowohl zwischen den prozessornahen Komponenten als auch zwischen Prozessor und Peripherie steht und die sich an der Verbindungstechnik moderner lokaler Netze orientiert:

- Mehrere Geräte werden jeweils mittels einer Punkt-zu-Punkt-Verbindung an einem *Switch* zusammengefaßt und dieser an eine zentrale Kommunikationseinheit des Rechners, z.B. an dessen Chipsatz angebunden. Beispiele hierfür sind PCI-Express (als Nachfolger des PCI- bzw. PCI-X-Busses) und InfiniBand. Auf sie wurde bereits in 5.1.5 und 5.1.6 eingegangen, weshalb sie hier nicht weiter betrachtet werden sollen.

Die Ziele bei der Entwicklung bzw. Weiterentwicklung peripherer Übertragungswege sind höhere *Übertragungsraten*, das Überbrücken größerer Entfernung sowie eine flexible Handhabung, und das bei begrenzten Kosten. Höhere Übertragungsraten erreicht man einerseits mit breiteren Datenwegen und andererseits mit höheren Übertragungsfrequenzen. Breitere Datenwege sind insofern proble-

matisch, als sie einen hohen Aufwand an Leitungen und Steckern bedeuten. Höhere Übertragungsfrequenzen wiederum führen zu Störungen auf der Übertragungsstrecke, wie elektrisches Übersprechen und Signalreflexionen. Um sie zu vermeiden, müssen Leitungen abgeschirmt und ggf. mit aktiven Abschlüssen versehen werden. Bei einer parallelen Verbindung hat dies erhebliche Mehrkosten zur Folge, weshalb man ähnlich wie in Rechnernetzen inzwischen serielle Übertragungstechniken mit den dort gebräuchlichen Übertragungsmedien bevorzugt, d.h. abgeschirmte *Twisted-Pair-Leitungen*, *Koaxialkabel* und *Lichtwellenleiter*. Mit diesen Techniken lassen sich die Übertragungsfrequenzen derart erhöhen, daß die zunächst vorhandene Einbuße an Übertragungskapazität (Übergang von 8 oder 16 Datenleitungen auf 1 Datenleitung) wieder ausgeglichen wird. Zum Teil erreicht man mit seriellen Verbindungen sogar höhere Übertragungsraten als mit ihren parallelen Vorgängern, und dies über erheblich größere Entfernung. Als Nebeneffekt erhält man biegssamere Leitungen und Stecker mit sehr viel weniger Kontakten (z.B. 7 bei Serial Attached SCSI gegenüber 50 bei „Narrow“-SCSI oder 68 bei Wide-SCSI), so daß sich auch Geräte sehr kleiner Bauweise anschließen lassen. Darüber hinaus erlauben die seriellen Verbindungen flexiblere Verbindungsstrukturen.

Im Blickpunkt dieses Abschnitts stehen zunächst die IDE/ATA-Schnittstelle (16 Bit) und der SCSI-Bus (8 oder 16 Bit) als die herkömmlichen, parallelen Verbindungen sowie deren aktuelle serielle Nachfolger Serial ATA (SATA) und Serial Attached SCSI (SAS). Hinzu kommen der serielle Bus FireWire und die serielle Verbindung Fibre-Channel, die beide auf der Kommandoebene des SCSI-Busses basieren und so schon seit längerer Zeit als dessen serielle Ausprägungen im Einsatz sind. Ergänzt werden sie um den zu FireWire konkurrierenden Universal Serial Bus (USB).

Tabelle 8-1 zeigt für diese Verbindungen einige ihrer technischen Daten, insbesondere die maximal möglichen Übertragungsraten. Für die seriellen Verbindungen sind diese außer in Mbit/s auch in Mbyte/s angegeben, um sie mit denen der parallelen Verbindungen vergleichen zu können. Zu beachten ist dabei, daß bei einigen seriellen Verbindungen die sog. *8B/10B-Codierung* verwendet wird. Sie sorgt für ein *gleichspannungsfreies* Datensignal und dafür, daß aus diesem auf der Empfängerseite das Taktsignal sicher abgeleitet werden kann. Dazu werden vom Sender jeweils 8 aufeinanderfolgende Datenbits in nichttrivialer Weise auf 10 Bits erweitert, und zwar so, daß Nullen und Einsen im Bitstrom gleichmäßig verteilt sind [Widmer, Franaszek 1983]. Als Nachteil beträgt die Nettoübertragungsrate allerdings nur 80% der Gesamtübertragungsrate.

Eine in diesem Buch vernachlässigte Kategorie ist die der Feldbusse, d.h. der Peripheriebusse, wie sie als Kommunikationssysteme in der Prozeß-, Automatisierungs- und Steuerungstechnik eingesetzt werden. Sie sind vorwiegend serieller Natur, und viele von ihnen basieren in ihrer physikalischen Repräsentation auf der Schnittstellennorm RS-485 (siehe 7.4.3). Einer der wenigen parallelen Ver-

Tabelle 8-1. Parallele und serielle Peripheriebusse und Punkt-zu-Punkt-Verbindungen mit ihren maximalen Übertragungsraten (abhängig von der Leitungsart). Angabe des maximalen Geräteabstands zur Verdeutlichung der Einsatzmöglichkeiten (rechnernah, rechnerfern) sowie der Anzahl adressierbarer Geräte. Bei Serial ATA, Serial Attached SCSI und Fibre Channel sind mit Mbit/s die Brutto- und mit Mbyte/s die Nettoübertragungsraten angegeben (8B/10B-Codierung!), sonst in beiden Fällen die Nettoraten (auch bei FireWire 1394b mit ebenfalls 8B/10B-Codierung); bei Serial Attached SCSI verdoppelt sich die angegebene Rate beim Vollduplexbetrieb

Peripheriebus/ Punkt-zu-Punkt- Verbindung (*)	Anzahl an Daten- leitungen	max. Übertragungsrate		Leitungsart	max. Geräte- abstand m	Anzahl adressier- barer Geräte
		Mbit/s	Mbyte/s			
parallel:						
IDE/ATA*	16	–	133	Flachband	0,45	2
SCSI (wide)	16	–	320	Twisted-Pair	12 Buslänge	15
seriell:						
Serial ATA*	4	1.500	150	unverdrillt	1	2
Serial Attached SCSI*	4	3.000	300	unverdrillt	8	4032
FireWire (1394a)	4	400	50	Twisted-Pair	4,5	63
" (1394b)	4	800/1.600	100/200	"	"	"
" "		"	"	Glasfaser	100	"
Fibre-Channel (Arbitrated Loop)	4	1.062	100	Koax	30	126
		1.062	100	Glasfaser	10.000	"
Fibre-Channel (Switched Fabric)	4	1.062	100	Koax	30	16 Mio.
		1.062	100	Glasfaser	10.000	"
USB low speed	2	1,5	0,18	unverdrillt	3	127
" full speed	2	12	0,18	Twisted-Pair	4,5	"
" high speed	2	480	60	"	"	"

treter, den wir in die Betrachtung mit aufnehmen, ist der IEC-Bus, ein sog. *Instrumentierungsbus*, d.h. ein Bus, der den Datenaustausch zwischen dem Rechner und Meß- und Anzeigegeräten erlaubt.

8.2.1 IDE/ATA, ATAPI

IDE oder *IDE/ATA* (Integrated Drive Electronics, AT-Attachment) bezeichnet die standardisierte parallele Schnittstelle, wie sie ursprünglich für den Anschluß von Festplattenspeichern an den ISA-Bus für PCs entwickelt wurde (ANSI X3.221-1994). Sie basiert auf der Integration des zuvor am PC/AT-Bus als separater Baustein vorhandenen *Hard-Disk-Controllers* in das Festplattenlaufwerk. Als Konsequenz dieser Integration vereinfacht sich der Anschluß des Laufwerks an den ISA-Bus auf einen auf 40 Signale reduzierten ISA-Steckplatz mit einem *IDE*-

Adapter als einfacher Anpaßlogik, der lediglich Bausteine zur Decodierung von Adressen und zur Stromverstärkung (Treiber) umfaßt, sowie auf ein 40-adriges Flachbandkabel als Verbindung zwischen IDE-Adapter und Laufwerk. Die Datenübertragung erfolgt dementsprechend mit der Breite des ISA-Busses, d.h. mit 16 Bit. Die Länge des Kabels ist auf 46 cm (18 Zoll) begrenzt, weshalb IDE-Verbindungen nur *innerhalb* von Rechnern einsetzbar sind. – Seit es eine serielle Nachfolgerin dieser Schnittstelle gibt, Serial ATA (SATA, 8.2.2), wird sie auch als *Parallel ATA (PATA)* bezeichnet.

Laufwerksteuerung. In dieser ersten Form erlaubt die IDE/ATA-Schnittstelle, obwohl sie quasi als *Punkt-zu-Punkt-Verbindung* ausgelegt ist, den Anschluß von zwei Geräten, genauer, von zwei Festplattenlaufwerken. Beide Geräte sind an dasselbe Flachbandkabel angeschlossen, wobei eines als Master und das andere als Slave bezeichnet wird. Gesteuert (programmiert) werden die beiden Laufwerke über je zwei Registersätze (Kommando- und Steuerregisterblock) mit je acht Registern mit der Wirkung als Kommando-, Sektornummer-, Zylindernummer-, Daten-, Steuer- und Statusregister. Adressiert werden die Registersätze und die Einzelregister eines Laufwerks durch zwei bzw. drei Steuersignale, die durch die o.g. Adreßdecodierung im IDE-Adapter aus den ISA-Bus-Adressen gewonnen werden. Kommandos werden an beide Laufwerke übertragen; die Auswahl von Master- oder Slave-Laufwerk erfolgt durch ein Bit, das in einem zuvor an beide Laufwerke übertragenen Steuerwort steht.

Die Integration des ursprünglich separaten Hard-Disk-Controllers in das Festplattenlaufwerk sei anhand von Bild 8-12 verdeutlicht. Es zeigt zugleich die Schritte der Entwicklung des Festplattenanschlusses, wie sie sich in unterschiedlichen Schnittstellen innerhalb der insgesamt erforderlichen Elektronik niederschlugen. Die ursprünglich am PC/AT-Bus vorhandene Geräteschnittstelle lag als sog. *ST506/412-Schnittstelle* (Seagate Technology) unmittelbar zwischen dem eigentlichen Laufwerk und dem sog. Datenseparator. Sie wurde aus der Shugart-Schnittstelle für Floppy-Disk-Laufwerke entwickelt (siehe 8.3.1) und übertrug als serielle Schnittstelle die Daten in der zur Speicherung benutzten asynchronen Darstellung. Dabei lagen die erreichbaren Übertragungsraten, abhängig von der Datendarstellung, MFM- oder RLL-Codierung (Bild 8-35, S. 605), bei 5 bzw. 7,5 Mbit/s. In einem nächsten Schritt wurde dann der Datenseparator, der die Takt- und Dateninformation trennt bzw. zusammenfaßt, in das Laufwerk integriert, und es entstand die serielle *ESDI-Schnittstelle* (Enhanced Small Device Interface). Sie erlaubte höhere Aufzeichnungsdichten und mit der nun synchronen Datendarstellung Übertragungsraten von bis zu 24 Mbit/s (NRZ-Codierung, Bild 7-44, S. 529).

Durch weitere Integration, nämlich des eigentlichen Laufwerk-Controllers, entstand dann eine parallele Geräteschnittstelle, die in ihrer Abstraktion der *SCSI-Schnittstelle* entspricht. (Die SCSI-Schnittstelle ist entgegen der Darstellung in Bild 8-12 allerdings als Bus ausgelegt; siehe 8.2.3.) Zur Bildung der *IDE-Schnitt-*

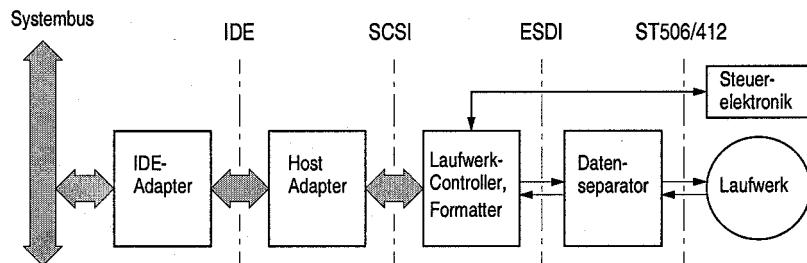


Bild 8-12. Schnittstellen bei Festplattenlaufwerken in historischer Folge (von rechts nach links); in Anlehnung an [Schmidt 1998]

stelle wurde schließlich der bei der SCSI-Schnittstelle zur Anpassung an den Systembus erforderliche Host-Adapter ebenfalls in das Laufwerk integriert. – Die Integration des Laufwerk-Controllers, der auch für die Formatierung der Daten auf den Magnetplatten zuständig ist (siehe 8.3.2), ergab für die Gerätehersteller eine Unabhängigkeit vom Rechnersystem, was der Entwicklung der Laufwerke, hin zu großen Speicherkapazitäten und großen Übertragungsraten, sehr förderlich war.

IDE-Erweiterung. Um auch andere Geräte als nur Festplattenlaufwerke über die IDE-Schnittstelle betreiben zu können, z.B. CD-ROM-Laufwerke, CD-Brenner und andere Laufwerke mit Wechselmedien, wurde die Funktionalität der Schnittstelle erweitert. Dazu wurde der für die Kommunikation mit Festplattenlaufwerken vorhandene Kommandosatz ergänzt, in einem ersten Schritt um den im SCSI-3-Standard festgelegten Kommandosatz für CD-ROM-Laufwerke und später dann um weitere SCSI-3-Kommandosätze für die anderen Geräte. Da diese neuen Kommandos in dem „alten“ ATA-Kommandoformat „verpackt“ werden, bezeichnet man die so erweiterte IDE-Schnittstelle auch als *ATAPI-Schnittstelle* (ATA Packet Interface). Um die neue Funktionalität der Schnittstelle umzusetzen, wurde der passive IDE-Adapter durch zwei aktive *IDE-Kanäle* (zusammengefaßt zu einem *IDE/ATAPI-Controller*) ersetzt, von denen jeder wiederum einen Master und einen Slave verwalten kann, insgesamt also vier Geräte.

Mit dieser Erweiterung ging auch der Übergang vom ISA- zum PCI-Bus (und anderen Bussen) einher. Dabei wurden die IDE-Kanäle mit der Funktionalität von PCI-Busmastern ausgestattet, so daß sie eigenständig Daten mit dem Hauptspeicher übertragen können. Die aufgrund der Kanäle jetzt aufwendigere Hardware ist bei PCs üblicherweise in den Chipsatz integriert, d.h., die beim ISA-Bus als Busanschluß ausgeführte Schnittstelle ist bei PCI-Bus-basierten Systemen als direkte Verbindung mit der South-Bridge oder dem I/O-Controller-Hub realisiert (siehe dazu die Bilder 5-8 und 5-9, S. 288 und S. 290). Sie kann unabhängig davon aber auch mittels einer PCI-Bus-Steckkarte realisiert werden, so daß sich ein Rechnersystem unabhängig vom Chipsatz um zusätzliche vier IDE-Geräteanschlüsse erweitern läßt.

Übertragungsraten. Die Datenübertragung mit IDE-Geräten am ISA-Bus fand aufgrund von dessen eingeschränkter DMA-Leistungsfähigkeit vorwiegend programmiert statt (*PIO, programmed I/O*). Die Übertragungsraten lagen hier bei 3,33, 5,2 und 8,33 Mbyte/s (PIO-Modes 0, 1 und 2), begrenzt durch die Taktfrequenz des ISA-Busses von 8,33 MHz. Durch erste Modifikationen der Schnittstelle, die unter dem Begriff *EIDE (Enhanced IDE)* und in Verbindung mit dem PCI-Bus entstanden, konnten die Übertragungsraten dann auf 11,11, 16,66 und 20 Mbyte/s (PIO-Modes 3, 4 und 5) erhöht werden, z.T. auch unter Einsatz von DMA, um den Prozessor zu entlasten. Mit der heute gebräuchlichen sog. Ultra-DMA-Technik werden Übertragungsraten von anfänglich 33 Mbyte/s (Ultra ATA/33) und heute 66, 100 und 133 Mbyte/s (Ultra ATA/66 bzw. /100 bzw. /133) erreicht, wobei man mit Ultra ATA/133 die max. Übertragungsrate des mit 33 MHz getakteten PCI-Busses ausschöpft (siehe auch Tabelle 8-1, S. 562). Erreicht werden diese hohen Übertragungsraten, indem die Daten auf der IDE-Verbindung mit beiden Flanken des Taktsignals übertragen werden (double data rate). Zur Unterstützung der Übertragung werden die Daten blockweise (sektorweise) im IDE-Kanal in einem als FIFO-Speicher organisierten Puffer und im Laufwerk in einem sog. Sektorpuffer zwischengespeichert. Darüber hinaus ist für die Übertragung ab 66 Mbyte/s ein 80-adriges Kabel erforderlich, das durch zusätzliche Masseleitungen die Störanfälligkeit der Übertragung vermindert. Der eigentliche Anschluß ist aber nach wie vor 40-adrig. – Zu mehr Details zur IDE/ATA-Schnittstelle siehe z.B. [Schmidt 1998].

8.2.2 Serial ATA (SATA)

Serial ATA (SATA) wurde von einer von mehreren Firmen gebildeten Arbeitsgruppe standardisiert ([Serial ATA 2001]), und zwar als serielle Nachfolgerin der parallelen IDE/ATA-Schnittstelle. Zwei für den Benutzer offensichtliche Merkmale sind: das aufgrund von nur sieben Leitungen dünneren und flexiblere Verbindungskabel zwischen dem *SATA-Controller* und dem angeschlossenen Gerät sowie die für die geringe Leitungsanzahl hohe Nettoübertragungsrate von umgerechnet 150 Mbyte/s. Übertragen wird mit differentieller Signaldarstellung (± 250 mV) mit je einem unverdrillten Leitungspaar pro Übertragungsrichtung. Zusätzlich umfaßt das Kabel drei Masseleitungen. Die Übertragung ist vollduplex für Handshake-Information, jedoch nur halbduplex für die eigentlichen Daten. Die Signalcodierung erfolgt mit der eingangs Abschnitt 8.2 bereits erwähnten *8B/10B-Codierung*, bei der jeweils 8 aufeinanderfolgende Bits in geeigneter Weise auf 10 Bits erweitert werden. Dementsprechend spricht man auf der 10-Bit-Seite von Bruttoübertragungsrate und auf der 8-Bit-Seite von der um 20% niedrigeren Nettoübertragungsrate. In der derzeitigen, ersten SATA-Generation (SATA I) betragen diese Werte 1,5 Gbit/s und 1,2 Gbit/s (150 Mbyte/s, siehe auch Tabelle 8-1, S. 562). In der zweiten Generation (SATA II) sollen sie auf 3 bzw. 2,4 Gbit/s (300 Mbyte/s) verdoppelt werden; in der dritten Generation (SATA III) sollen sie er-

neut verdoppelt werden, also letztlich auf 600 Mbyte/s netto. SATA sieht Leitungslängen von bis zu 1 m vor und wird faktisch nur für rechnerinterne Geräte eingesetzt. Die Einbeziehung rechnerexterner Geräte wird es mit SATA II geben.

Emulation von Parallel ATA. Serial ATA ist zu *Parallel ATA* softwarekompatibel, was bedeutet, daß sämtliche auf der Parallelschnittstelle stattfindenden Zugriffe emuliert werden müssen. Dazu wird die zwischen SATA-Controller und SATA-Gerät seriell auszutauschende Information in Rahmen (frames) verpackt und auf der Empfängerseite wieder ausgepackt. Der Inhalt eines solchen Rahmens besteht im wesentlichen aus einem oder mehreren sog. Frame-Information-Structures (FISs), die die eigentliche Information enthalten und die je nach Aktion unterschiedliche Formate mit unterschiedlicher Anzahl an 32-Bit-Wörtern aufweisen. Ergänzt werden sie durch einen 32-Bit-CRC-Code und zwei sog. Primitives, die den Rahmen begrenzen (SOF, EOF). Die Übertragung der Rahmen erfolgt handshake-gesteuert. Das SATA-Protokoll insgesamt ist in Anlehnung an das OSI-Referenzmodell (S. 530) in vier Schichten unterteilt. – Als weitere Konsequenz aus der seriellen Übertragung müssen im SATA-Controller Duplikate der beiden Registersätze des Geräts, sog. Schattenregister vorhanden sein und diese dort regelmäßig durch Auslesen der Geräteregister aktualisiert werden.

SATA-Controller. Bild 8-13 zeigt die wichtigsten Komponenten eines SATA-Controllers, und zwar reduziert auf nur einen von üblicherweise vier SATA-Anschlüssen (ports). Hier befinden sich die Schattenregister (Command Block, Control Block), ergänzt durch einen SATA-spezifischen Registersatz (Superset Registers). Weiterhin gibt es je eine Steuerung für die programmierte Ein-/Ausgabe

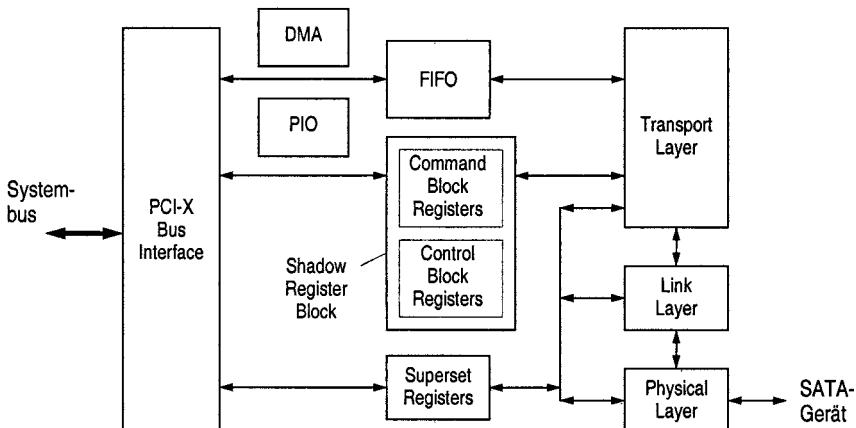


Bild 8-13. SATA-Controller mit den Hardwarekomponenten eines einzelnen SATA-Anschlusses. Bei üblicherweise vier solcher Anschlüsse sind diese Komponenten vervierfacht und an der Bus-schnittstelle durch einen Multiplexer/Demultiplexer und einen Arbiter für die DMA-Anforderungen ergänzt (in Anlehnung an [Intel 2003])

(PIO) und für die DMA-gesteuerte Ein-/Ausgabe (DMA). Letztere arbeitet mit einem FIFO-Pufferspeicher. Des weiteren zeigt es die in Hardware realisierten drei unteren Schichten des Protokolls, nämlich die Bitübertragungsschicht (Physical Layer), die Verbindungsschicht (Link Layer) und die Transportschicht (Transport Layer). Die darüberliegende, vierte Schicht ist als Anwendungsschicht (Application Layer) in der Software verankert. Die Verbindung zum SATA-Gerät ist nun eine echte *Punkt-zu-Punkt-Verbindung*, indem anders als bei Parallel ATA pro Anschluß nur noch ein Gerät versorgt wird. Systembus ist der PCI-X-Bus, der mit seiner Übertragungsrate von 1.064 Mbyte/s für vier SATA-Anschlüsse mit jeweils 150 Mbyte/s ausreichend dimensioniert ist (Tabelle 5-2, S. 284).

Zu den drei unteren Schichten des *SATA-Protokolls* einige erläuternde Angaben:

- In der *Transportschicht* werden senderseitig die Frame-Information-Structures (FISs) zusammengesetzt und dann an die Verbindungsschicht weitergegeben. Ausgelöst wird ein solcher Vorgang z.B. durch ein aus der Anwendungsschicht stammendes Kommando, das in das Kommandoregister der Schattenregister geladen wird. Dieses Kommando und weitere Einträge der Schattenregister bilden dann die für die FIS erforderliche Information. – Auf der Empfängerseite werden die ankommenden FISs in die Registerinhalte zerlegt.
- In der *Verbindungsschicht* werden senderseitig die Rahmen zusammengestellt, was die CRC-Code-Generierung mit einschließt. Hier werden außerdem die Datenbytes miteinander „verwürfelt“ (*data scrambling*), so daß aufeinanderfolgende gleichcodierte Bytes unterschiedlich codiert werden. Dies dient der Verringerung der elektromagnetischen Interferenz (EMI, Abstrahlung) auf der Übertragungsstrecke. Danach werden die Bytes der 8B/10B-Codierung unterzogen. Die so präparierten Rahmen werden dann an die Bitübertragungsschicht übergeben. – Auf der Empfängerseite werden diese Codierungen rückgängig gemacht, die CRC-Fehlerprüfung durchgeführt und die Rahmen in ihre FISs aufgelöst.
- In der *Bitübertragungsschicht* werden senderseitig die 32-Bit-Daten serialisiert und in differentieller Darstellung mit Pegeln von ± 250 mV auf die Übertragungsstrecke gegeben. – Auf der Empfängerseite werden die Signalpegel in Bits gewandelt und diese zu 32-Bit-Datenwörtern parallelisiert.

In allen drei Schichten werden darüber hinaus „Signale“ verwaltet, wie sie zur Steuerung und für die Handshake-Synchronisation der Übertragung benötigt werden. Sie werden als Primitives bezeichnet und sind als 32-Bit-Wörter codiert.

Als umfangreichere Übersicht zu Serial ATA siehe z.B. [Stiller 2002], für Details sei [Serial ATA 2001] empfohlen.

8.2.3 SCSI-Bus

Der *SCSI-Bus* (Small Computer System Interface) wurde aus dem *SASI-Bus* (Shugart Associates System Interface), einem Firmenstandard, entwickelt und 1982 durch das ANSI genormt (siehe z.B. [Seagate 1988]). Grundsätzlich unterscheidet er sich gegenüber dem für den Single-Master-Betrieb ausgelegten SASI-Bus durch die Erweiterung für den Multimaster-Betrieb. Bild 8-14 zeigt dazu als Beispiel eine Konfiguration, bei der zwei Rechnersysteme (hosts) A und B über jeweils einen eigenen Host-Adapter, des weiteren ein Festplattenlaufwerk und ein Streamer an den SCSI-Bus angeschlossen sind. Der SCSI-Bus erlaubt hierbei sowohl den Zugriff beider Rechnersysteme auf die Geräte als auch den Datentransfer zwischen den beiden Systemen (Rechnerkopplung). Darüber hinaus kann ggf. ein Backup-Vorgang zwischen dem Plattspeicher und dem Streamer über diesen Bus autonom durchgeführt werden, nachdem der Vorgang von einem der Rechner gestartet worden ist. Üblicherweise sind SCSI-Konfigurationen jedoch auf einen einzigen Host eingeschränkt, und auch das für den autonomen Backup-Vorgang erforderliche Copy-Kommando ist meist nicht implementiert.

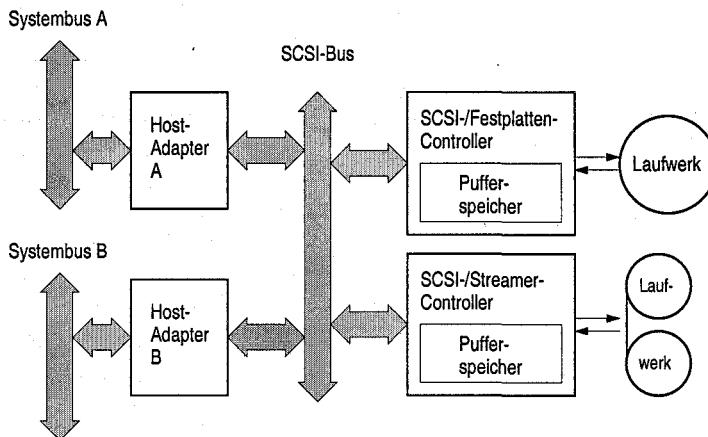


Bild 8-14. SCSI-Bus mit zwei Mastern (Host A und Host B) und zwei Hintergrundspeichern mit logischer Darstellung der Busstruktur; physikalisch werden sämtliche Buskomponenten durch einzelne Kabel jeweils Gerät-zu-Gerät miteinander verbunden

Überblick über die Entwicklung. Tabelle 8-2 zeigt einige technische Daten verschiedener SCSI-Entwicklungen. Der ursprüngliche 8-Bit-SCSI-Bus (heute auch als Narrow-SCSI bezeichnet) arbeitet mit Datenübertragungsraten von 2,5 Mbyte/s bei asynchroner und 5 Mbyte/s bei synchroner Übertragung. Im Laufe der Zeit wurde dieser Bus weiterentwickelt, um ihn an die steigenden Übertragungsraten insbesondere von Festplattenspeichern anzupassen. Dazu wurde einerseits die Busbreite von 8 auf 16 Bit erweitert und andererseits die Taktfrequenz der Übertragung erhöht. Die Frequenzerhöhung beruht im wesent-

lichen auf verbesserten Übertragungstechniken, u.a. in einer weniger störanfälligen *Signaldarstellung*. Die ursprüngliche asymmetrische Darstellung als TTL-Signal (+5 V, massebezogen, *single-ended*, SE), wurde durch die differentielle, d.h. symmetrische Darstellung gemäß RS-485 (± 5 V, *high-voltage differential*, HVD) und durch die differentielle Darstellung mit geringen Pegeln ($\pm 1,215$ V, *low-voltage differential*, LVD) ersetzt (siehe auch 7.4). Damit verbunden können auch größere Entfernung als zuvor überbrückt werden. Dennoch gilt auch hier generell: Mit zunehmender Entfernung verringert sich (bei gleicher Technik) die maximal erreichbare Übertragungsrate. Bei *Ultra-3-SCSI* (Ultra160) und *Ultra-4-SCSI* (Ultra320) werden als weitere Maßnahme zur Erhöhung der Übertragungsraten beide Taktflanken für die Datenübertragung genutzt (*double data rate*). Der Nachfolger, Ultra640, soll bei einer Taktfrequenz von 160 MHz 640 Mbyte/s übertragen können. – Die mögliche Anzahl an Busteilnehmern (Peripheriegeräte einschließlich Host-Adapter) ist jeweils gleich der Anzahl an Datenleitungen, d.h. 8 oder 16 (1-aus-n-Code als Adresse).

Tabelle 8-2. SCSI-Bus in unterschiedlichen technischen Ausführungen hinsichtlich Busbreite, Taktfrequenz, maximal möglicher Übertragungsrate und größtmöglicher Buslänge (abhängig von der Signaldarstellung: SE single-ended, LVD low-voltage differential, HVD high-voltage differential)

SCSI-Bezeichnung	Busbreite Bit	Taktfrequenz MHz	Übertragungsrate Mbyte/s	Buslänge m		
				SE	LVD	HVD
Narrow	8	5	5	6	–	25
Fast	8	10	10	3	–	25
Fast Wide	16	10	20	3	–	25
Ultra	8	20	20	1,5	–	25
Ultra Wide	16	20	40	–	–	25
Ultra 2	8	40	40	–	12	–
Ultra 2 Wide	16	40	80	–	12	–
Ultra 3 Wide / Ultra160	16	40 (DDR)	160	–	12	–
Ultra 4 Wide / Ultra320	16	80 (DDR)	320	–	12	–

Wie eingangs 8.2 erwähnt, versteht man unter SCSI seit einigen Jahren auch *serielle Busse*, nämlich *FireWire* und *Fibre-Channel* (sowie die hier nicht weiter ausgeführte *Serial-Storage-Architecture*, SSA). Hierfür wurde unter der Bezeichnung SCSI-3 ein Architekturmodell in vier Ebenen definiert, das für alle Varianten die SCSI-Kommandos einheitlich vorgibt. Diese Ebenen sind:

1. eine Kommandoebene, in der gerätespezifische SCSI-Kommandos nach Gruppen (command sets) zusammengefaßt sind (z.B. Block-, Stream-, Graphic-Commands),
2. eine Kommandoebene mit den für alle Gruppen gemeinsamen Kommandos (primary commands),

3. eine Ebene mit den unterschiedlichen Transportprotokollen für die parallelen und die seriellen Übertragungstechniken (transport protocols) sowie
4. eine Ebene, in der die physikalischen Vorgaben für diese Transportprotokolle festgelegt sind (physical interconnects).

SCSI-Buskommunikation. Die eigentliche Busverbindung besteht beim 8-Bit-SCSI-Bus aus einem 50-poligen Kabel (68-polig bei Wide-SCSI). Sie sieht für die Datenübertragung acht bidirektionale Datenleitungen und eine Paritätsleitung vor; ferner insgesamt neun zum Teil bidirektionale Signalleitungen für die Steuerung der Buskommunikation. Diese Steuerung obliegt der Steuereinheit des *SCSI-Host-Adapters* sowie den SCSI-Controllern der angeschlossenen Geräte. Jeder von ihnen, d.h. der Host-Adapter als auch die Geräte, können im Prinzip Auslöser (*initiator*) oder Ziel (*target*) einer Kommunikation sein. Initiator ist zwar meist der Host-Adapter; Initiator kann aber auch z.B. ein Streamer bei einem Backup-Vorgang sein. Sobald ein Target vom Initiator angewählt ist, geht die Steuerung an dieses über. Der Grund für die Steuerungsübergabe ist, daß sich das Übertragungszeitverhalten aus der Funktion des Target, d.h. aus der entsprechenden Gerätefunktion ergibt und sich damit die Steuerungsabläufe vom Target aus einfacher gestalten lassen.

Normaler Ablauf. Der durch ein SCSI-Kommando ausgelöste Ablauf einer Kommunikation zwischen Initiator und Target umfaßt mehrere sog. *SCSI-Busphasen*, die mittels der Steuersignale verwaltet werden. Bild 8-15 zeigt dazu als Beispiel eine Datenausgabe an z.B. einen Streamer, bei der sieben solcher Busphasen durchlaufen werden. Initiator ist hierbei der Host-Adapter, der Streamer fungiert als Target.

Ausgangspunkt für die Kommunikation ist der unbelegte Bus, d.h. die Bus-Free-Phase. Sie ist dadurch charakterisiert, daß die Steuersignale BSY (busy) und SEL (select) für eine bestimmte Zeit inaktiv sind. Von diesem Zustand ausgehend, fordert der Host-Adapter den Bus an. Sind mehrere Initiatoren am Bus vorhanden oder sieht das System vor, daß ein Target die physikalische Verbindung zwischenzeitlich unterbrechen kann (disconnect/reconnect, siehe unten), so muß der Initiator eine (sonst nicht zu implementierende) *Busarbitrationsphase* durchlaufen. Hierzu legt er ein Identifikationsbyte auf die Datenleitungen und aktiviert das Signal BSY. Dieses Byte gibt im 1-aus-8-Code die SCSI-Adresse des Anforderers und damit auch dessen Buspriorität an. Es wird mit ggf. gleichzeitig auf dem Bus liegenden Identifikationsbytes anderer Anforderer oder-verknüpft und vom Anforderer selbst ausgewertet. Die Belegung der Datenleitung D7 mit einer Eins entspricht der Adresse 7; sie hat die höchste Priorität. Die Prioritäten nehmen zur Adresse 0 (D0) hin ab.

Anmerkung. Dieses Priorisierungsverfahren ist eine besonders einfache Variante der in 5.4.2 ausführlich behandelten Priorisierung mit Identifikationsbus. Dort sind die Identifikationsnummern als Dualzahlen, hier als 1-aus-n-Code dargestellt. Dadurch ergibt sich hier der Vorteil eines beson-

ders einfachen Schaltungsaufbaus (mit dem Nachteil der viel geringeren Zahl anschließbarer Geräte).

Zeigt der Zustand des Datenbusses dem Host-Adapter nach einer vorgegebenen Zeit an, daß er von allen konkurrierenden Anforderern die höchste Priorität hat, so leitet er eine Selektionsphase ein, in der er das Target anwählt. (Die anderen Anforderer nehmen ihre Identifikationsbytes wieder vom Bus.) Dazu aktiviert er das Signal \overline{SEL} , legt verzögert das Identifikationsbyte des Target oder-verknüpft mit seinem eigenen Identifikationsbyte auf die Datenleitungen (2-aus-8-Code)

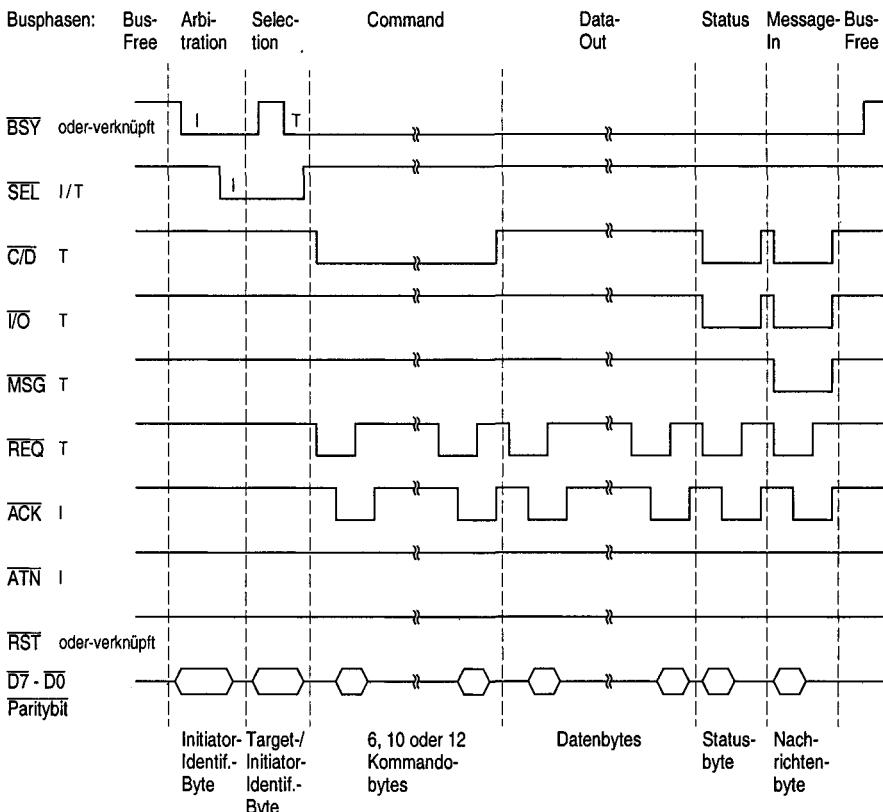


Bild 8-15. Signaldarstellung der Busphasen am Beispiel der Ausführung eines SCSI-Kommandos für eine Datenausgabe mit asynchroner Steuerung. Die Bezeichnungen I (initiator) und T (target) geben an, welcher der beiden Übertragungspartner für das Aktivieren der einzelnen Steuersignale zuständig ist

und inaktiviert wiederum verzögert \overline{BSY} . Der SCSI-Controller des adressierten Target quittiert diese Anwahl, indem er seinerseits \overline{BSY} aktiviert, woraufhin der Host-Adapter die Selektionsphase durch Inaktivieren von \overline{SEL} abschließt. Damit ist die Verbindung zwischen Initiator und Target hergestellt.

Für die folgenden Busphasen übernimmt das Target die Steuerung. Es leitet zunächst eine Kommandophase ein, indem es $\overline{C/D}$ (Control/Data) aktiviert und $\overline{I/O}$ (Input/Output) und \overline{MSG} (Message) inaktiv hält. Es fordert vom Host-Adapter mehrere Kommandobytes (command descriptor block) zur Spezifikation der folgenden Operation, hier einer Datenausgabe ein, wobei die einzelnen Byteübertragungen durch die Handshake-Signale \overline{REQ} (Request, Target) und \overline{ACK} (Acknowledge, Initiator) synchronisiert werden. Die Anzahl der Kommandobytes, 6, 10 oder 12, hängt von der im ersten Byte angegebenen Kommandogruppe ab (siehe unten). In der dann folgenden Data-Out-Phase inaktiviert das Target das $\overline{C/D}$ -Signal, hält \overline{MSG} weiterhin inaktiv und gibt mit inaktivem $\overline{I/O}$ -Signal die Datenübertragungsrichtung Ausgabe vor. Die Übertragung der einzelnen Datenbytes selbst wird wieder mittels der Handshake-Signale \overline{REQ} und \overline{ACK} synchronisiert.

Nach der Datenübertragung leitet das Target eine Statusphase ein, wozu es die Signale $\overline{C/D}$ und $\overline{I/O}$ aktiviert und \overline{MSG} inaktiv hält. In dieser Phase sendet es ein Statusbyte, das dem Initiator (Host-Adapter) das Ergebnis der Datenübertragung, erfolgreich/nicht erfolgreich abgeschlossen, übermittelt. (Bei nicht erfolgreichem Abschluß kann weitere Statusinformation abgerufen werden.) Im Anschluß daran leitet das Target in unserem Beispiel eine Message-In-Phase ein, wozu es zusätzlich zu $\overline{C/D}$ und $\overline{I/O}$ noch \overline{MSG} aktiviert. In dieser Phase hat es die Möglichkeit, ein oder mehrere Nachrichtenbytes an den Initiator zu übertragen, z.B. um ihm mitzuteilen, ob die Verbindung für eine weitere Übertragung aufrechterhalten bleiben soll oder nicht. Auch hier erfolgt die Synchronisation wieder mittels der Signale \overline{REQ} und \overline{ACK} . Mit dem Inaktivieren von \overline{BSY} gibt das Target schließlich den Bus wieder frei.

Das in diesem Ablauf nicht verwendete Signal \overline{ATN} ermöglicht es dem Initiator, eine Message an das Target auszugeben, indem er damit das Target auffordert, eine Message-Out-Phase einzuleiten. Das ebenfalls nicht verwendete Reset-Signal RST , das von jedem SCSI-Teilnehmer aktiviert werden kann, dient dazu, sämtliche SCSI-Teilnehmer und damit auch den Bus unmittelbar in den Inaktivzustand zu versetzen.

Im obigen Beispiel ist die Übertragung der Kommando-, Daten-, Status- und Nachrichtenbytes als *asynchrone* Übertragung beschrieben, bei der die Signale \overline{REQ} und \overline{ACK} als Handshake-Signale benutzt werden. Eine schnellere Übertragung erreicht man mit *synchroner* Steuerung, bei der das vom Target ausgegebene \overline{REQ} -Signal als Takt fungiert. Sie ist jedoch auf die eigentliche Datenübertragung beschränkt (d.h.: Kommando-, Status- und Nachrichtenbytes werden immer asynchron übertragen). Um synchron übertragen zu können, müssen sich Initiator und Target zuvor durch einen Nachrichtenaustausch darüber verständigen, daß sie beide dazu in der Lage sind.

Anmerkungen. (1.) Mit *Ultra160* wurde eine *CRC-Datensicherung* eingeführt, die der mit der erhöhten Taktfrequenz einhergehenden höheren Störanfälligkeit der Übertragung Rechnung trägt: Bei erkanntem Übertragungsfehler kann die Übertragung wiederholt werden. (2.) *Ultra320* sieht

als neuere Technik ein paketorientiertes Protokoll vor, mit dem nicht nur die Daten, sondern auch die Kommandos und die Statusinformation synchron übertragen werden können, was zu einer höheren Übertragungsleistung führt.

Ablauf mit Disconnect/Reconnect. Ein Target kann sich während einer Kommandoausführung vom Bus abkoppeln (*disconnect*) und sich später mittels einer Arbitrations- und einer Reselektionsphase (SEL-Signal) mit dem Initiator wieder in Verbindung setzen (*reconnect*). Dabei wird die physikalische Verbindung zwischen Initiator und Target unterbrochen, so daß der SCSI-Bus für andere Übertragungen frei ist; die logische Verbindung zwischen ihnen wird jedoch aufrechterhalten. Nützlich ist dies z.B. während der Wartezeit für das Positionieren eines Plattenarms oder eines Streamer-Tapes.

Bild 8-16 zeigt dazu das Beispiel einer Dateneingabe. Während der Selektionsphase aktiviert der Host-Adapter das Attention-Signal ($\overline{\text{ATN}}$) und zeigt dem Target damit an, daß er eine Nachricht ausgeben möchte, deren Steuerung ja dem Target unterliegt. Mit dieser „Identify-Message“ teilt er dem Target mit, daß ein „Disconnect“ erlaubt ist. Das Target übernimmt zunächst das Kommando und führt dann nach Bedarf die Abkopplung mittels einer „Disconnect-Message“ durch. Nach der späteren Reselektion, der eine durch das Target ausgelöste Busarbitration vorausgeht, fordert das Target den Initiator (hier den Host-Adapter) mittels einer „Identify-Message“ auf, die alte Übertragungssituation auf der Rechnerseite wiederherzustellen, damit die Übertragung fortgesetzt werden kann. Zu

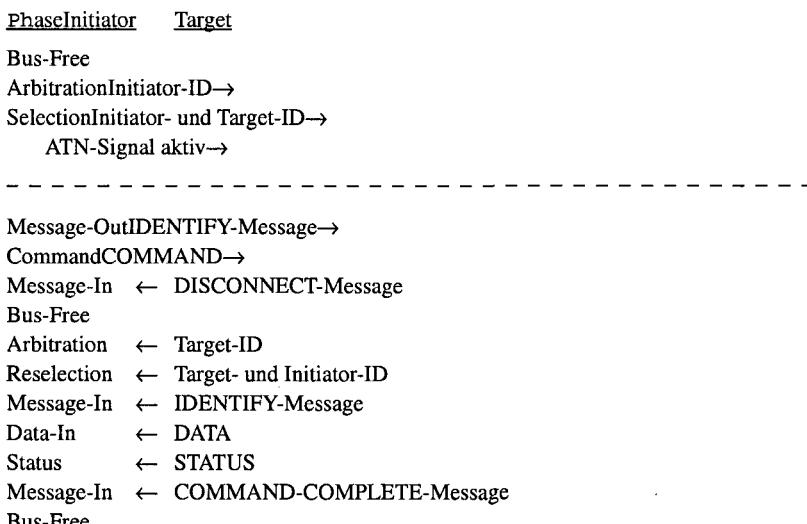


Bild 8-16. Ausführung eines SCSI-Kommandos mit zwischenzeitlicher Unterbrechung der physikalischen Verbindung (disconnect/reconnect) bei Aufrechterhaltung der logischen Verbindung. Die Pfeile geben die jeweilige Richtung der Datenbusaktivitäten zwischen Initiator und Target an; die Steuerung der Übertragungen obliegt nach Abschluß der Phase Selection (gestrichelte Linie) dem Target

diesem Wiederherstellen gehört das z.B. Reinitialisieren von Registern im Host-Adapter, insbesondere das Bereitstellen von Zeigern auf die für die Übertragung benutzten Speicherbereiche (z.B. durch Umschalten zwischen mehreren Registersätzen).

Aufbau der SCSI-Kommandos. SCSI-Kommandos werden in Gruppen zusammengefaßt. Maximal gibt es acht solcher Gruppen mit jeweils bis zu 32 Kommandos. Ein Kommando besteht aus dem eigentlichen Operationscode und aus weiteren, für die Ausführung benötigten Angaben. Es wird dem Initiator in Form eines *Command-Descriptor-Blocks* bereitgestellt, der abhängig von der Kommandogruppe aus 6, 10 oder 12 Bytes besteht.

Bild 8-17 zeigt als Beispiel das 6-Byte-Format der Gruppe 0, und zwar für ein Lese- oder Schreibkommando mit Angaben für die blockweise Datenübertragung. Es enthält im ersten Byte den 3-Bit-Gruppencode und den 5-Bit-Operationscode des Kommandos. Die folgenden vier Bytes spezifizieren eine 3-Bit-Adresse (logical unit number, *LUN*) mittels der bei einem Target (SCSI-Gerät) bis zu 8 Untereinheiten unterschieden werden können, eine logische 21-Bit-Blockadresse (Blocknummer), ab der im SCSI-Gerät geschrieben bzw. gelesen werden soll, und eine 8-Bit-Blockanzahl für die zu übertragenden Blöcke. Vorausgesetzt wird dabei eine für Initiator und Target einheitliche Blockgröße, beispielsweise von 256 Bytes. Das letzte Byte enthält Steuerinformation, so z.B. ein Link-Bit, das anzeigt, ob nach Beendigung des momentanen Kommandos ein weiteres Kommando ausgeführt soll, ohne daß die Bus-Free-Phase durchlaufen wird (Verkettung von Kommandos).

	7	0
Byte 0	Gruppencode	Operationscode
1	LUN	
2		Blockadresse
3		
4		Blockanzahl
5		Steuerinformation

Bild 8-17. SCSI-Kommandoformat für Lese- und Schreibkommados der Gruppe 0

Kommandos mit 10 und 12 Bytes haben eine erweiterte logische Blockadresse von 32 Bits und sehen für die Angabe der Blockanzahl 16 bzw. 32 Bits vor. Bei einer Blockgröße von 256 Bytes und 16 Bits für die Blockanzahl können damit bereits Übertragungen von bis zu 16 Mbyte in einem durch die 32 Bits der logischen Blockadresse und die Blockgröße vorgegebenen „Geräte“adreßraum von 1024 Gbyte durchgeführt werden, was z.B. bei Backup-Vorgängen genutzt wird. Mit 32 Bits für die Blockanzahl wird die Anzahl der durch ein einziges Kommando übertragbaren Datenbytes nur noch durch den Geräteadreßraum begrenzt.

SCSI-Host-Adapter und übergeordnete Übertragungssteuerung. In den ersten Jahren nach Einführung des SCSI-Busses bestand die Hardware der Host-Adapter-Schnittstelle aus einem einfachen passiven *SCSI-Interface-Baustein*, der lediglich den Zugang zu den Steuer- und Datensignalen des SCSI-Busses über Register herstellte. Das Erzeugen und Auswerten dieser Signale und somit ihr Zeitverhalten während der einzelnen Busphasen wurde durch Software gebildet, wobei eine Vielzahl an Zeitparametern einzuhalten war. Eingesetzt wurde hierfür häufig der zentrale Prozessor, der dadurch entsprechend belastet wurde, oder aber auch ein zusätzlicher, dem Host-Adapter zugeordneter lokaler Prozessor. Zur Unterstützung der eigentlichen Datenübertragung wurde ferner der DMA-Controller des Rechners oder ein zusätzlicher lokaler DMA-Controller eingesetzt.

Heutige Host-Adapter-Busschnittstellen werden durch aktive *SCSI-Controller-Bausteine* gebildet. Ein solcher Baustein arbeitet auf der Ebene der einzelnen Busphasen selbstständig, indem er das Erzeugen und Auswerten der Signale in Hardware durchführt. Das Aktivieren dieser Phasen erfolgt durch einzelne Kommandobytes, die in ein Kommandoregister des Bausteins geschrieben werden. (Sie sind nicht identisch mit den SCSI-Kommandobytes, die ein Target vom Initiator anfordert!)

Diese Aufgabe, d. h. die Ablaufsteuerung auf der Ebene von Busphasen, übernimmt wiederum ein Prozessor. Da sie nun sehr viel weniger Aufwand verursacht, kann hierfür der zentrale Prozessor, aber auch ein Ein-/Ausgabeprozessor eingesetzt werden. Unterstützt wird er ggf. durch einen in den Controller integrierten Spezialprozessor, der die Abfolge der Busphasen als Kommandofolge in einem ebenfalls im Controller vorhandenen RAM bereitgestellt bekommt.

Weitere Unterstützung bietet ein solcher Controller-Baustein durch einen integrierten DMA-Controller und einen FIFO-Speicher, der die zwischen Systembus und SCSI-Bus zu übertragenden Daten zwischenspeichert und so für einen Geschwindigkeitsausgleich sorgt.

Zur Vorbereitung für die Übertragung mit einem SCSI-Gerät wird im Hauptspeicher ein sog. *Command-/Parameter-Block* als Informationsblock angelegt. Er umfaßt sämtliche Steuerinformationen für den Host-Adapter und sieht darüber hinaus Speicherplatz für Statusinformation des Target und des Host-Adapters vor. Er enthält im einzelnen den Command-Descriptor-Block, also das eigentliche SCSI-Kommando, einen Zeiger auf den im Hauptspeicher bereitgestellten Datenbereich, die Datenbereichsgröße, die Target-Adresse und die Logical-Unit-Number, ggf. einen Verkettungszeiger auf den nächsten Command-/Parameter-Block sowie Initialisierungsinformation für den Host-Adapter und den DMA-Controller.

Angaben zur Geschwindigkeit. Für die Geschwindigkeit der Datenübertragung zwischen einem SCSI-Gerät und dem Rechner werden üblicherweise zwei Werte angegeben, die *Burst-Data-Transfer-Rate* und die *sustained Data-Transfer-Rate*.

Die Burst-Data-Transfer-Rate ist ein Spitzenwert und gilt für kurzeitige Übertragungen, bei denen der Datentransfer allein zwischen Host-Adapter und einem im Gerät vorhanden Pufferspeicher (Cache) stattfindet. Dadurch, daß der Pufferspeicher ein Halbleiterspeicher mit schnellem Zugriff ist, liegt diese Übertragungsrate im Bereich der maximal möglichen Transferleistung des SCSI-Busses.

Die sustained Data-Transfer-Rate hingegen bezieht sich auf die kontinuierliche Datenübertragung, sie ist gleich der Übertragungsrate bei direktem Zugriff auf das Speichermedium, und diese liegt oft wesentlich unter der des SCSI-Busses. Darüber hinaus wird die Datenübertragung durch Wartezeiten beeinträchtigt, z.B. beim Positionieren des Arms eines Plattspeichers (Seek-Operation) oder bei der Übertragung von Kommando-, Status- und Nachrichtenbytes, die, wie bereits erwähnt, ggf. sehr viel langsamer als der eigentliche Datentransfer ist.

Würdigung. Die relativ aufwendige Steuerung des SCSI-Busses ist kostenintensiver als die bei PCs für den Anschluß von Festplatten (und inzwischen auch anderen Peripheriegeräten) gebräuchliche *IDE*- bzw. *ATAPI-Schnittstelle* (8.2.1). Dafür bietet das Buskonzept eine größere Flexibilität, überbrückt außerdem größere Entfernung und erlaubt höhere Übertragungsraten. SCSI wird deshalb insbesondere bei Rechnern mit hohem Datenaufkommen mit Hintergrundspeichern eingesetzt, so z.B. bei *Servern*. Inzwischen werden aber beide parallelen Übertragungstechniken verstärkt durch die neueren seriellen Techniken verdrängt, bei PCs durch SATA, USB und FireWire, bei Servern durch Fibre-Channel und künftig vermutlich durch den angekündigten seriellen Nachfolger von SCSI, SAS. Da diese seriellen Übertragungstechniken unterschiedlich leistungsfähig sind, muß ihr Einsatz anwendungsbezogen entschieden werden.

Zur Vertiefung des SCSI-Busses siehe z.B. [Schmidt 1998].

8.2.4 Serial Attached SCSI (SAS)

Serial Attached SCSI (SAS) ist der designierte Nachfolger des SCSI-Busses, wird also als serielle Übertragungstechnik das parallele SCSI ablösen. Wie bei SATA wird auch hier der technische Aufwand für die „Verkabelung“ von Ein-/Ausgabe-geräten, insbesondere Festplattenlaufwerken, vereinfacht, ohne dabei eine Einbuße in der Übertragungsrate hinnehmen zu müssen. Vielmehr werden sich die angestrebten Steigerungen der Übertragungsrate durch die serielle Technik einfacher realisieren lassen als mit paralleler Technik. Die Übertragung erfolgt wie beim parallelen SCSI jeweils zwischen einem *Initiator* und einem *Target*, und zwar mit differentieller Signaldarstellung auf zwei Leitungspaaren für die beiden Übertragungsrichtungen. Mit zusätzlichen drei Masseleitungen umfaßt das Kabel für einen SAS-Anschluß (SAS-Port) insgesamt sieben Leitungen, so wie bei SATA. Die derzeitige Übertragungsrate eines Leitungspaares beträgt bei *8B/10B-Codierung* der Daten 3 Gbit/s brutto, d.h. 300 Mbyte/s netto (siehe auch Tabelle

8-1, S. 562). Anders als bei SATA können für die Übertragung von Daten beide Leitungspaare gleichzeitig genutzt werden (Vollduplexbetrieb), womit ein SAS-Port eine Übertragungsrate von insgesamt 2×300 Mbyte/s aufweist.

Schichtenmodell. SAS basiert auf einem Modell mit insgesamt fünf Schichten mit den Bezeichnungen (1.) Physical, (2.) Phy, (3.) Link, (4.) Transport und (5.) Application. Die ersten beiden Schichten beschreiben das technische Umfeld eines SAS-Ports, so z.B. den Stecker, das Kabel bzw. die beiden Übertrager (transmitter, receiver) und die 8B/10B-Codierung. Die dritte Schicht umfaßt u.a. die CRC-Datensicherung, das „Verwürfeln“ der Bytes (*data scrambling*, siehe SATA S. 567), die Steuerung der Übertragung durch Steuerwörter (primitives), die Adressierung (address frames), das Verwalten logischer Verbindungen und das Zusammenfassen mehrerer physikalischer SAS-Verbindungen zu einem sog. Wide-Port. Darauf aufbauend ermöglicht diese Schicht das Verzweigen auf drei verschiedene Anwendungen, nämlich auf das Serial SCSI Protocol (SSP), das Serial ATA Tunneled Protocol (STP) und das Serial Management Protocol (SMP). An diesen Protokollen orientieren sich dann die Schichten vier (frame definitions) und fünf (SCSI bzw. ATA specific features) mit jeweils individuellen Vorgaben. Dieses Verzweigen eröffnet die Möglichkeit, in ein SAS-Ein-/Ausgabesystem auch SATA-Geräte einzubinden. So können dort nicht nur die zuverlässigeren und leistungsfähigeren SCSI-Festplatten, sondern auch die kostengünstigeren IDE/ATA-Festplatten eingesetzt werden (diese jeweils mit SAS- bzw. SATA-Port/Controller ausgestattet).

Systemstrukturen. Die Verbindung von Initiator und Target erfolgt anders als beim parallelen SCSI nicht über einen Bus, sondern als *Punkt-zu-Punkt-Verbindung*. Diese weist im einfachsten Fall an den beiden Endpunkten (Geräten) je einen Port auf, so daß als Übertragungsrate die oben angegebenen 2×300 Mbyte/s nutzbar sind. Für Übertragungen mit höheren Geschwindigkeiten können die Geräte an den Endpunkten mit jeweils mehreren Ports ausgestattet (wide port) und dann mehrere solcher Verbindungen parallel betrieben werden (wide links), so daß sich eine entsprechend vervielfachte Übertragungsrate ergibt. Abweichend von diesem einfachen, direkten Verbindungsaufbau läßt sich ein SAS-Ein-/Ausgabesystem auch netzartig strukturieren, indem, wie in Bild 8-18 gezeigt, sog. Expander als Verzweigungselemente eingesetzt werden. Unterschieden werden dabei sog. Edge-Expander mit einfacher Datenweiterleitung und sog. Fan-Out-Expander mit komplexerer Datenweiterleitung, jeder von ihnen mit bis zu 128 Ports ausgestattet. Auch hier sind die Verbindungen wieder Punkt-zu-Punkt, d.h. zwischen jeweils zwei Ports ausgeführt. Das Bild zeigt außerdem die Möglichkeit des gemischten Einsatzes von SAS- und SATA-Geräten (hier Festplattenlaufwerke, HD) sowie die Möglichkeit der redundanten Wegebildung. Letzteres wird unterstützt durch SAS-Festplatten, die dazu mit zwei Ports ausgestattet sind (dual-ported drives).

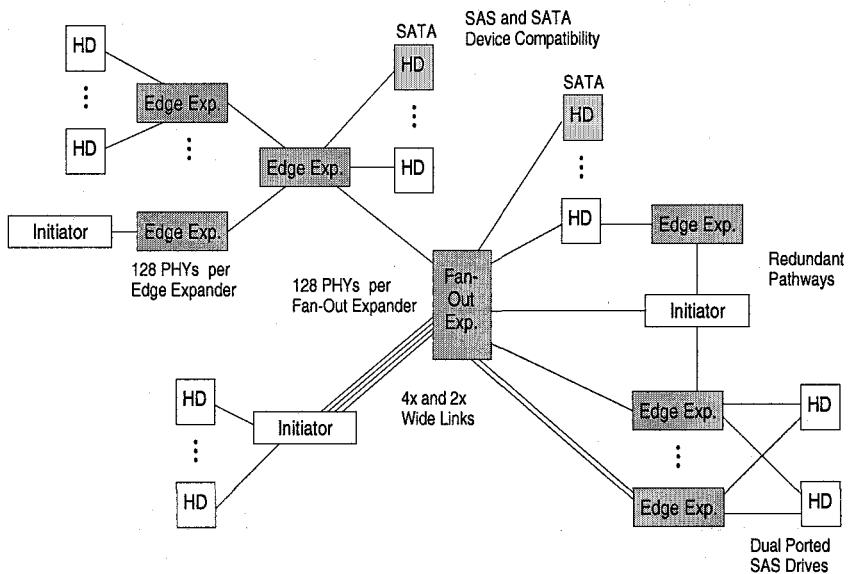


Bild 8-18. SAS-Struktur mit mehreren Initiatoren (z.B. Server) und Targets (SAS- und SATA-Festplatten), die durch mehrere Edge-Expander und einen Fan-Out-Expander miteinander verbunden sind (in Anlehnung an [Griffith 2003])

8.2.5 FireWire

Die Entwicklung *serieller SCSI-Techniken* hat bereits Jahre vor dem erst jüngst spezifizierten eigentlichen Nachfolger des SCSI-Busses, Serial Attached SCSI (SAS), begonnen, und zwar mit drei unterschiedlichen Aktivitäten, die sich in folgenden genormten Varianten niedergeschlagen haben:

- dem von Apple initiierten *FireWire* (IEEE 1394),
- dem von einem Firmenkonsortium getragenen *Fibre-Channel* (ANSI-Standard) und
- der von IBM entwickelten *Serial Storage Architecture* (ANSI-Standard).

Allen gemeinsam ist die Kommandostruktur des herkömmlichen SCSI, wie sie in den beiden oberen Ebenen des SCSI-3-Architekturmodells festgeschrieben ist (8.2.3, S. 569). Unterschiede im Logischen ergeben sich jedoch in der Abbildung der Kommandos auf die jeweilige serielle Schnittstelle und darin, wie die SCSI-Teilnehmer miteinander kommunizieren, d.h. in der Ebene der Transportprotokolle. Damit einher gehen unterschiedliche physikalische Realisierungen, wie sie in der untersten Ebene des Architekturmodells (physical interconnects) beschrieben sind. – In diesem Abschnitt stellen wir den FireWire-Bus exemplarisch etwas ausführlicher dar und verweisen zur Vertiefung auf [Anderson 1999].

Technische Merkmale. Der *FireWire-Bus* (Namensgebung von Apple) wurde unter der Bezeichnung *IEEE 1394* im Jahr 1995 standardisiert und ist heute in der überarbeiteten Version *IEEE 1394a-2000* weit verbreitet. Er ist ein Kabelbus, bei dem die Geräte, die üblicherweise über zwei oder drei Anschlußbuchsen (ports) verfügen, mittels Kabel zu einem Strang oder Baum miteinander verbunden werden. Das Kabel besteht aus zwei jeweils für sich abgeschirmten verdrillten Leitungspaaren (shielded twisted pairs) sowie zwei Leitungen für eine Spannungsversorgung und hat einen 6-poligen Stecker. Bei einem verkleinerten, 4-poligen Stecker, wie er für Geräte mit kleinen Abmessungen benötigt wird, z. B. für Digitalkameras, wird auf die Spannungsversorgung über das Kabel verzichtet. Insgesamt lassen sich mit FireWire bis zu 63 Geräte miteinander verbinden. Der Abstand zwischen zwei Geräten, die Kabellänge, beträgt maximal 4,5 m. Die Verbindung zwischen zwei beliebigen Geräten darf über bis zu 16 Kabel führen. Rechnerseitig gibt es einen Host-Adapter, von dem aus mehrere solcher Kabelbusse abgehen können. Übertragungen auf dem Bus sind jedoch vom Host-Adapter unabhängig; sie erfolgen *Peer-to-Peer* und belasten den Rechner ggf. nicht. Die Übertragungsraten von *IEEE 1394a* liegen bei 100, 200 und 400 Mbit/s (FireWire 400, 50 Mbyte/s), wobei der jeweils langsamere Teilnehmer die Übertragungsrate bestimmt.

Ein neuerer Standard, *IEEE 1394b-2002*, erlaubt mit einer veränderten Signalcodierung, dem sog. Beta-Mode, Übertragungsraten von 800 Mbit/s (FireWire 800) und 1 600 Mbit/s (FireWire 1600). Die Kabellängen liegen hier bei Verwendung von Twisted-Pair-Kabeln ebenfalls bei 4,5 m. Es sind jedoch auch andere Übertragungsmedien verwendbar, z. B. sog. Multimode-Glasfaserkabel, mit denen sich der Gerätetypstand auf bis zu 100 m vergrößern lässt. Der Stecker des Twisted-Pair-Kabels ist hier 9-polig, mit gegenüber dem alten FireWire zwei zusätzlichen Masseleitungen und einer weiteren Leitung für eine künftige Nutzung. Um Geräte mit altem FireWire mit Geräten des neueren Standards verbinden zu können, gibt es neben den neuen Beta-Ports auch sog. bilinguale Ports. – Zu den technischen Daten siehe auch Tabelle 8-1, S. 562.

Ein besonderes Merkmal von FireWire ist die Möglichkeit der *isochronen Datenübertragung*. Sie garantiert eine quasi-kontinuierliche Übertragungsraten, wie sie für Multimedia-Anwendungen, z. B. für die unterbrechungsfreie Übertragung von Musik und Videofilmen, erforderlich ist. Für solche Anwendungen des Konsumerbereichs wurde der Bus ursprünglich konzipiert. Inzwischen wird er aber auch verstärkt für den mobilen Anschluß rechnerspezifischer Peripheriegeräte eingesetzt, so z. B. für rechnerexterne Festplatten. – Die nachfolgenden Ausführungen zu FireWire beschreiben den Standard *IEEE 1394a* in einigen Details. Davon abweichende Merkmale von *IEEE 1394b* werden daran anschließend kurz zusammengefaßt.

Struktur und Konfigurierung. Bild 8-19 zeigt die für den FireWire typische *Baumstruktur*, ausgehend von einem *Host-Adapter* (Host-to-IEEE-1394a-Adap-

ter), der die Schnittstelle zum Systembus (z.B. einem PCI-Bus) bildet und der ggf. mehrere Busanschlüsse (ports) bereitstellt. Vor der Initialisierung des Busses ist seine Struktur noch unbekannt, und es existieren sozusagen nur *Punkt-zu-Punkt-Verbindungen* zwischen benachbarten Busteilnehmern, den sog. Knoten des Busses. Diese Knoten haben entweder Verzweigungsfunktion, d.h. mehr als nur einen Port (*branch node*), oder bilden das Ende eines Zweigs (*leaf node*). Während der Initialisierung, die durch eine Reset-Signalisierung ausgelöst wird (bus reset), ermitteln die Knoten eigenständig die Struktur des Busses (tree identification). Sie kommunizieren dazu mit ihren Nachbarn und vermitteln diesen unter Auswertung der Eigenschaft Branch- oder Leaf-Node, ob sie zu diesen in einer Kind- oder einer Elternbeziehung stehen (child bzw. parent notify). Dabei wird letztlich dann auch jener Knoten ermittelt, der nur Child-Anschlüsse aufweist und damit die Wurzel des Baumes bildet (*root hub*, root node). Er muß nicht zugleich auch der Host-Adapter sein. Gibt es mehr als einen solchen Knoten (root contention), so wird eine Zufallsauswahl auf der Basis von Zufallszeiten für die dann erforderlichen Wiederholungen der Anfragen getroffen.

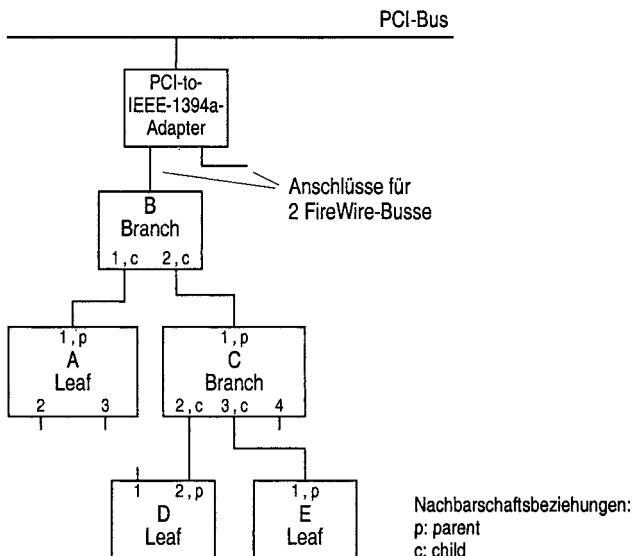


Bild 8-19. Typische FireWire-Busstruktur, wie sie sich nach der Selbstkonfiguration darstellt (nach [Anderson 1999]). Anschluß an einen PCI-Bus mittels eines PCI-to-IEEE-1394a-Adapters

In einem nächsten Schritt weisen sich die Knoten ihre Identifikationsnummern (IDs) in eindeutiger Weise selbst zu (self-identification). Hierzu schickt der Root-Node in Wiederholung ein Signal aus (self-ID grant), das von den Branch-Nodes gezielt weitergeleitet wird, beginnend bei ihren Ports kleinster Nummer. Der beim ersten Durchgang erreichte Leaf-Node gibt sich selbst die ID 0 und schickt diese Nummer dann zurück (self-ID packet), was von den anderen Knoten registriert wird. Der im nächsten Durchgang erreichte Knoten, entweder ein Leaf-

Node oder ein Branch-Node, dessen Nachfolger schon identifiziert ist, gibt sich dann die nächst höhere ID. Dieser Vorgang wird so lange wiederholt, bis sich auch der Root-Node eine ID gegeben hat.

Diese vom Bus eigenständig ausgeführten Vorgänge des Sich-Konfigurierens und Sich-Identifizierens werden auch dann durchlaufen, wenn ein Gerät während des Busbetriebs in die Struktur eingebunden oder aus ihr entfernt wird (*plug and play*). – Als Konsequenz dieses Konfigurierungsverfahrens sind Ringstrukturen nicht erlaubt.

Datenübertragung. Daten werden in Form von *Paketen* (packets) übertragen. Diese bestehen aus einem Datenblock variabler Länge und einem ihm vorangestellten Kopf (header), der u.a. die erforderlichen Adressangaben und die Datenblocklänge enthält. Kopf und Datenblock sind beide durch CRC-Bytes gesichert (7.7.3). Die Übertragung selbst erfolgt Punkt-zu-Punkt zwischen jeweils zwei benachbarten Knoten, und zwar in eigenständiger Weise, d.h. ohne Unterstützung durch den Host-Adapter. In Anlehnung an Rechnernetze, bei denen die Netzteilnehmer direkt, d.h. ohne den Umweg über einen Server, miteinander kommunizieren, spricht man deshalb auch von *Peer-to-Peer*-Transfers. Knoten, die mehr als einen Port haben, wirken dabei als *Repeater*, d.h., sie reichen ein empfangenes Paket über all ihre anderen Ports weiter. Der im Kopf adressierte Knoten übernimmt schließlich das Paket, schickt es aber zusätzlich ebenfalls über seine ggf. anderen Ports weiter (*broadcasting*).

Hinsichtlich des Übertragungsprotokolls und der im Kopf stehenden Information werden zwei Übertragungsarten unterschieden: die asynchrone und die isochrone Übertragung. Die *asynchrone Datenübertragung* basiert auf einem Informationsaustausch zwischen den beiden Übertragungspartnern. Das heißt, der Initiator einer Übertragung verschi ckte zunächst ein Anforderungspaket (request packet), dessen Kopf neben der Zieladresse auch die Quelladresse enthält, und er erhält dann vom Adressaten ein Antwortpaket (response packet) als Quittierung seiner Anforderung. Dabei bilden die zu schreibenden Daten den Datenblock des Anforderungspaket und die zu lesenden Daten den Datenblock des Antwortpaket. Das Antwortpaket enthält darüber hinaus Statusinformation, die es erlaubt, eine fehlerhafte Übertragung zu beanstanden, um sie dann vom Initiator wiederholen zu lassen. Fehler werden anhand der *CRC-Sicherungsbytes* erkannt. (Es gibt keine Fehlererkennung auf der physikalischen Ebene.) Die Asynchronität der Übertragung ergibt sich aus dem momentanen Zeitverhalten des Busses, indem das Verschicken von Paketen von einer zuvor zu durchlaufenden Busarbitration abhängt.

Die *isochrone Datenübertragung* trägt dem Einsatz des FireWire-Busses für das Übertragen von Audio- und Videodaten Rechnung. Hier muß den Übertragungspartnern eine vorher zu beantragende Übertragungskapazität garantiert werden, damit sie einen quasi-kontinuierlichen Datenstrom für die Dauer der Übertragung aufrecht erhalten können. Diese wird jedoch nur in gewissen Grenzen gewährle-

stet, indem die Zeitskala der Busaktivitäten in Zeitschlüsse (cycles) von $125 \mu\text{s}$ Dauer unterteilt wird, in denen je nach Bedarf bis zu $100 \mu\text{s}$, d.h. bis zu 80%, für isochrone Übertragungen reserviert werden. Die daneben stattfindenden asynchronen Übertragungen werden dementsprechend nachrangig behandelt (Bild 8-20). Das Übertragungsprotokoll wird außerdem vereinfacht, indem es nur einen Anforderungsblock und keine Quittierung gibt. Es gibt zwar die CRC-Fehlerprüfung, man geht aber davon aus, daß Audio- und Videodaten kurzlebig sind (z.B. beim Abspielen eines Videofilms auf einem Monitor) und somit Fehler nicht unbedingt korrigiert werden müssen. Dementsprechend ist auch der Kopf des Pakets vereinfacht: Es gibt keine explizite Zieladresse, sondern nur eine relativ „grobe“ Kanalnummer als Zielangabe, mit dem Hintergrund einer *broadcast-orientierten* Übertragung.

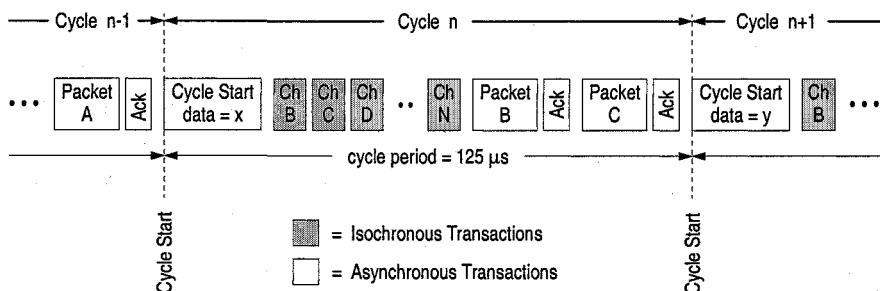


Bild 8-20. Zuordnung von isochronen und asynchronen Übertragungspaketen zu Zeitschlitten (cycles) von $125 \mu\text{s}$ Dauer (nach [Anderson 1999])

Übertragungsgeschwindigkeit. IEEE 1394a sieht Übertragungsraten von 100, 200 und 400 Mbit/s vor, wobei vorausgesetzt wird, daß jeder der Knoten zumindest 100 Mbit/s unterstützt. Während der Konfigurationsphase teilen sich die benachbarten Knoten ihre jeweils maximal mögliche Übertragungsrate mit, so daß danach für jede der Punkt-zu-Punkt-Verbindungen die nutzbare Übertragungsr率 bekannt ist. Beim Verschicken eines Pakets wird diesem vom Sender eine bestimmte Übertragungsgeschwindigkeit zugeordnet. Ein Knoten, der es ggf. weiterreichen muß, tut dies nur, wenn die Verbindung zu seinem Nachbarn der Geschwindigkeitsanforderung entspricht. Wenn nicht, bricht er die Übertragung für diesen Verbindungsweg ab, erfüllt ggf. aber seine Repeater-Funktion für andere Nachbarn, die die geforderte Übertragungsgeschwindigkeit aufweisen. Damit ein Knoten die Übertragungsgeschwindigkeit überhaupt erkennen kann, wird dem Paket eine Signalisierungsphase vorangestellt (speed signaling).

Busarbitration. Wie bereits erwähnt, geht einer Datenübertragung jeweils eine Busarbitration voraus. Hierbei hat der Root-Node die höchste Priorität in der Buszuteilung, gefolgt von seinen unmittelbaren Nachbarn usw. Das heißt, die Busstruktur, genauer der Abstand der Knoten zum Root-Node, legt sozusagen „natürliche“ Buszuteilungsprioritäten fest (ähnlich einer Daisy-Chain). Dadurch, daß in den verschiedenen Zweigen Anforderungen gleichzeitig gestellt werden

können, kann diese natürliche Priorisierung auch durchbrochen werden. Eine Busanforderung kann allerdings nur gestellt werden, nachdem auf dem Bus eine Ruhephase bestimmter Dauer aufgetreten ist. Dies setzt die Leistungsfähigkeit des Busses herab.

Bild 8-21 zeigt den Arbitrierungsvorgang an einem Beispiel. Die beiden Leaf-Nodes A und E stellen gleichzeitig je eine Anforderung (TX-Request), die sie in Richtung Root-Node D absetzen. Beim diesem trifft z.B. die Anforderung von E als erste ein, woraufhin er über seinen Port 2 dem Leaf-Node E die Busgewährung (TX-Grant) signalisiert. Zusätzlich signalisiert er über seinen Port 1 (und, sofern vorhanden, über weitere Ports), daß er den Bus bereits vergeben hat. Der Branch-Node C reicht diese Meldung über all seine Ports weiter, wodurch sie schließlich über B auch den Leaf-Node A erreicht. Dieser nimmt dann seine Anforderung vom Bus, um sie später zu wiederholen. Auch der Node B, der inzwischen ebenfalls eine Anforderung generiert hat, nimmt diese wieder vom Bus. – Wäre die Anforderung von A vor der von E beim Root-Node eingetroffen, so hätte dieser das Busgewährungssignal an seinem Port 1 ausgegeben. Dieses wäre dann von Node B „abgefangen“ worden, da dieser aufgrund seiner Lage im Bus höher priorisiert ist als A.

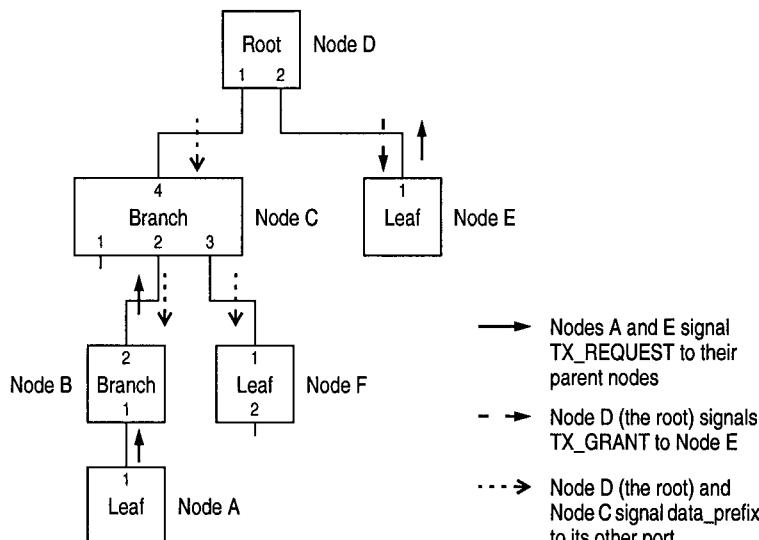


Bild 8-21. Beispiel für den Ablauf einer Buszuteilung (nach [Anderson 1999])

FireWire-Schnittstelle. Für die Vielfalt der hier erwähnten (auch der nicht erwähnten) Signalisierungen sieht die Verkabelung zwischen zwei Knoten lediglich zwei verdrillte Adernpaare vor, die beim Standard IEEE 1394a einerseits zur *differentiellen Signaldarstellung* und andererseits für die Übertragung von Gleichspannungen und Konstantströmen genutzt werden. Ergänzt werden sie durch eine

Stromversorgungsleitung (zur Versorgung von Geräten mit begrenztem Stromverbrauch) und durch eine Masseleitung. Die Kabelverbindung ist also 6-adrig mit relativ kleinem, 6-poligem Steckverbinder.

Um eine Vorstellung über die verschiedenen eingesetzten Signalisierungstechniken zu vermitteln, zeigt Bild 8-22 die Busschnittstelle in ihrer elektrotechnischen Darstellung. Gezeigt ist aus Platzgründen allerdings nur eines der beiden Adernpaare mit den auf seinen beiden Seiten (Knoten Y und Knoten Z) erforderlichen Sende- und Empfangseinrichtungen. Die beiden Anschlußpunkte sind mit Twisted Pair A (TPA) und Twisted Pair B (TPB) bezeichnet. Beim zweiten Adernpaar sind diese Einrichtungen und somit auch die Bezeichnungen der Anschlußpunkte genau spiegelbildlich angeordnet, d.h., die beiden Adernpaare wirken „über Kreuz“. Im folgenden dazu die wichtigsten Signalisierungen als Kurzbeschreibungen mit Nennung der daran beteiligten Einrichtungen (zu den Details siehe [Anderson 1999]).

- *Device Attachment/Detachment:* Zeigt dem Port eines Knotens an, ob er mit einem aktiven Knoten (device) verbunden ist. Ein aktiver Knoten legt dazu eine Gleichspannung an seinen TPA-Anschluß (Twisted Pair Biased Source),

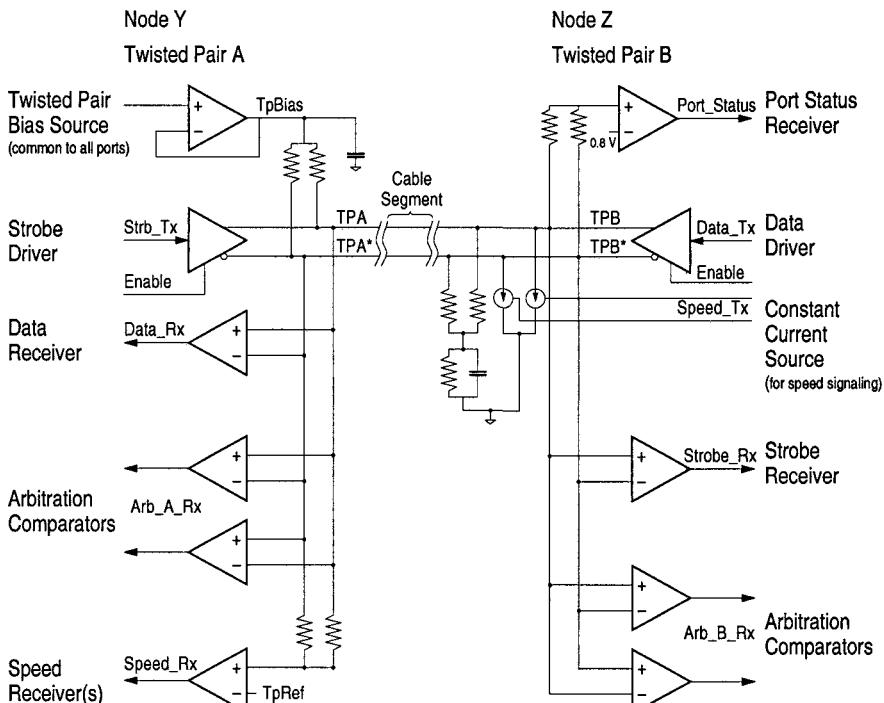


Bild 8-22. Elektrische Schnittstelle des FireWire-Busses nach IEEE 1394a, gezeigt für eines der beiden Adernpaare. Das andere Adernpaar ist dazu spiegelbildlich (nach [Anderson 1999])

und der abfragende Port wertet diese an seinem TPB-Anschluß (Port Status Receiver) aus.

- *Bus Idle State*: Beschreibt den Ruhezustand der Busverbindung. Er ist dann gegeben, wenn von beiden Seiten Device Attachment angezeigt wird und gleichzeitig keine der beiden Seiten eine Datenübertragung signalisiert (beide Data Drivers und beide Strobe Drivers sind inaktiv).
- *Arbitration Signaling*: Wird für das Signalisieren von u.a. Bus-Reset, Tree-Identification, Self-Identification und Normal Arbitration verwendet. Daran beteiligt sind beide Adernpaare, jeweils mit den Data Drivers und Strobe Drivers als Sender und mit den Arbitration Comparators als Empfänger. Die Data und Strobe Drivers senden entweder eine 0 oder eine 1 als Konstantpegel oder sind im hochohmigen Zustand (Z). Bei den auf beiden Seiten empfangenen Arbitrationsignalen wird ebenfalls zwischen den Zuständen 0, 1 und Z unterschieden. Die Interpretation des empfangenen Signals, z.B. bei der (normalen) Busarbitration als TX-Request oder TX-Grant, hängt davon ab, was der Empfänger selbst gesendet hat.
- *Speed Signaling*: Zeigt an, ob ein Paket anstatt mit 100 Mbit/s mit 200 oder 400 Mbit/s zu übertragen ist. Dazu entzieht der signalisierende Knoten mittels einer Konstantstromquelle (Constant Current Source an TPB) dem gegenüberliegenden Knoten einen Konstantstrom bestimmter Stärke (aus Twisted Pair Biased Source), was dieser an den veränderten Spannungsverhältnissen am Adernpaar TPA erkennt (Speed Receivers an TPA, pro Geschwindigkeit einer).
- *Data/Strobe Signaling*: Die eigentliche Datenübertragung erfolgt unter Nutzung beider Adernpaare, womit sich eine bidirektionale Übertragung trotz zweier Adernpaare auf den Halbduplexbetrieb beschränkt. Das Datensignal wird vom Sender mit *NRZ-Codierung* im Übertragungstakt erzeugt (Data Driver an TPB, siehe Bild 8-23). Der Sender generiert außerdem parallel dazu ein Strobe-Signal (Strobe Driver an TPA), und zwar so, daß dieses immer

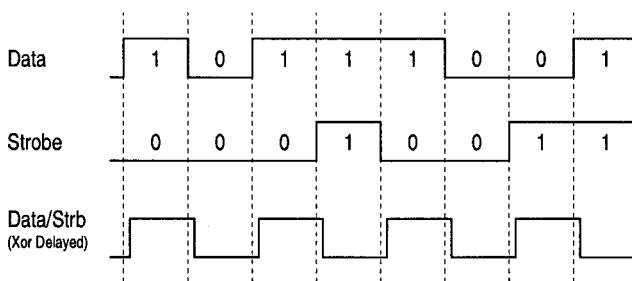


Bild 8-23. Datendarstellung des FireWire-Busses nach IEEE 1394a. Taktrückgewinnung auf der Empfängerseite durch Exklusiv-Oder-Verknüpfung der Signale Data und Strobe (nach [Anderson 1999])

dann den Pegel wechselt, wenn es das Datensignal nicht tut. Der Empfänger verknüpft beide Signale exklusiv-oder und gewinnt so das Taktsignal für die *Bitsynchronisation*.

IEEE 1394b. Der Vorteil des neueren Standards IEEE 1394b gegenüber IEEE 1394a liegt in der Erhöhung der Übertragungsrate auf zunächst 800 Mbit/s und künftig 1600 Mbit/s (geplant sind 3200 Mbit/s). Möglich wird diese Erhöhung durch technische Änderungen beim Übertragungsverfahren. So wird für die Datenübertragung als Neuerung die *8B/10B-Codierung* eingesetzt, mit dem Vorteil, daß das für die empfängerseitige Taktgewinnung bisher erforderliche Strobe-Signal entfällt. Die Gleichspannungsfreiheit der 8B/10B-Codierung läßt ferner eine galvanische Trennung beider Adernpaare zu, so daß das zweite Adernpaar jetzt als eigenständiges Leitungspaar für den Vollduplexbetrieb zur Verfügung steht. Damit einhergehend wird auch auf die bisher eingesetzte Signalisierung mittels Gleichspannungen und Konstantströmen verzichtet und stattdessen eine Signalisierung mit 10-Bit-Steuercodes in 8B/10B-Codierung durchgeführt. Bezeichnet wird diese modifizierte Übertragungstechnik als Beta-Mode. – Die Gleichspannungsfreiheit läßt als Neuerung Übertragungsmedien zu, wie sie bei Rechnernetzen gebräuchlich sind, z.B. Glasfaserkabel, wodurch jetzt Kabellängen von bis zu 100 m möglich sind.

Auch im Beta-Mode erfolgt die Datenübertragung zwischen zwei Knoten (Geräten) immer nur in einer Richtung (also halbduplex!), nämlich zwischen dem aktuellen Sender von Daten und dem Empfänger dieser Daten, d.h. über jeweils nur eines der beiden Leitungspaare. Das andere Leitungspaar wird parallel dazu für eine schnelle Busarbitrierung genutzt, indem auf ihm Busanforderungen übertragen werden, und zwar in Gegenrichtung zum momentanen Datenfluß. Empfänger der Busanforderungen ist der jeweilige Datensender, der als der gerade aktive Knoten die Aufgabe des Busarbiters vorübergehend übernimmt (Bild 8-24). Seine

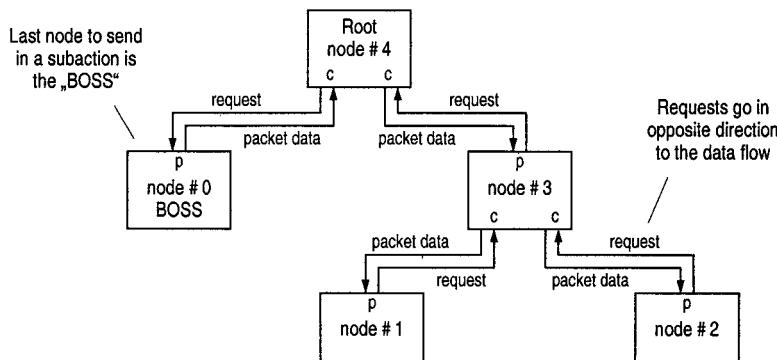


Bild 8-24. IEEE-1394b: Datenübertragung (data packets) mit dazu parallelen Busanforderungen (requests). Busvergabe durch den jeweiligen BOSS (bus owner/supervisor/selector), d.h. durch den aktuellen Datensender (nach [IEEE 1394b 2002])

Funktion wird mit BOSS (bus owner/supervisor/selector) bezeichnet. Anders als beim alten FireWire ist hier also nicht mehr der bei der Konfigurierung festgelegte Root-Node für die Busarbitrierung zuständig. Mit dieser Technik entfallen auch die beim alten FireWire mit der Busarbitration einhergehenden Wartezeiten (Ruhephasen). – Die oben genannten Übertragungsraten sind, bezogen auf die 8B/10B-Codierung, Nettoübertragungsraten, d.h., der Datenübertragung mit 800 Mbit/s liegt eine Taktfrequenz von 1 GHz zugrunde.

8.2.6 Fibre-Channel

Der *Fibre-Channel* (FC, ANSI-Standard) ist eine serielle Verbindungstechnik mit *Punkt-zu-Punkt*-Datenübertragung im Vollduplexbetrieb. Die Übertragung wird derzeit bevorzugt mit 1 GHz getaktet und weist bei *8B/10B-Codierung* der Daten eine Nettoübertragungsrate von 800 Mbit/s (100 Mbyte/s) pro Übertragungsrichtung auf (siehe auch Tabelle 8-1, S. 562). Eine Spezifikation für 2 GHz liegt vor, 10 GHz sind in Vorbereitung.

Als Übertragungsmedien dienen entweder abgeschirmte Twisted-Pair-Leitungen, Koaxial-Kabel oder (wie der Name dieser Technik sagt) Glasfasern, d.h. Lichtwellenleiter. Abhängig von diesen Medien sind Entfernung von 30 m bis zu 10 km überbrückbar. Die Anzahl der Fibre-Channel-Knoten (Rechner, Geräte) ist mit bis zu 16 Millionen im Grunde unbegrenzt. Das Hinzufügen und Entfernen von Knoten ist im laufenden Betrieb möglich (*hot plug and play*).

Die Datenübertragung erfolgt nach einem 5-Schichten-Protokoll in Datenblöcken (frames) von bis zu 2 Kbyte Größe, ergänzt durch einen Header mit Adreß- und Inhaltsinformation sowie durch vier CRC-Bytes zur Datensicherung. Gesteuert wird die Übertragung durch 4-Byte-Steuerwörter in 8B/10B-Codierung (primitive signals). Als besondere Übertragungsform gibt es die *isochrone Datenübertragung*, d.h. eine garantierte Übertragungskapazität zur Aufrechterhaltung eines kontinuierlichen Datenstroms für z.B. Audio- und Videodaten.

Entwickelt (und heute durch einen großen Marktanteil bestätigt) wurde Fibre-Channel für den Aufbau von Speichersystemen, insbesondere solchen mit vernetztem Zugriff von mehreren Rechnern aus, sog. *Speichernetzen (storage area networks, SANs)*. Unterschieden werden dabei drei verschieden aufwendige Verbindungsformen.

- Die *Point-to-Point-Topology* verbindet zwei Fibre-Channel-Knoten (Geräte) direkt miteinander und stellt so die einfachste Verbindungsform dar. Einer der beiden Knoten muß dabei Initiatorfunktion haben, wie dies z.B. bei der Verbindung von Server und Festplattenspeicher gegeben ist.
- Bei der *Arbitrated Loop Topology* werden in der ursprünglichen Ausführung bis zu 126 Rechner und Geräte Punkt-zu-Punkt in einer Ringstruktur mitein-

ander verbunden (oft als Doppelring ausgelegt). Heute wird anstelle des Rings eine Sternstruktur bevorzugt, bei der ein Hub als zentraler Vermittler fungiert.

In beiden Fällen ist der Verbindungsaufwand zwar relativ gering, jedoch sinkt der Datendurchsatz mit steigender Knotenzahl erheblich. Arbitrated Loop, die Fibre-Channel-Version der seriellen SCSI-Varianten, gilt deshalb als Kompromißlösung für kleinere SANs.

- Die *Fabric Topology* ist die allgemeinste und leistungsfähigste der drei Verbindungsformen. Mit ihr können sehr viele Rechner und Geräte (bis zu 16 Millionen adressierbar) in großen, netzförmigen Systemen miteinander verbunden werden (Bild 8-25). Dies wird insbesondere dazu genutzt, Server lokaler Netze mit ihren Festplatten-Speichereinheiten, sog. *RAIDs* (redundant arrays of independent/inexpensive disks), und Backup-Geräten (Streamer) zu einem eigenen Netz zusammenzufassen, um unabhängig vom lokalen Netz auf Dateien schnell und wahlfrei zugreifen zu können, und um netz- und serverunabhängig Backup-Vorgänge durchführen zu können.

Die Netzstruktur ist hier üblicherweise eine sog. *Switched-Fabric* (Schaltgewebe), bei der mittels mehrerer *Switches* Mehrwegeverbindungen zwischen allen Netzteilnehmern herstellbar sind (*any-to-any network*).

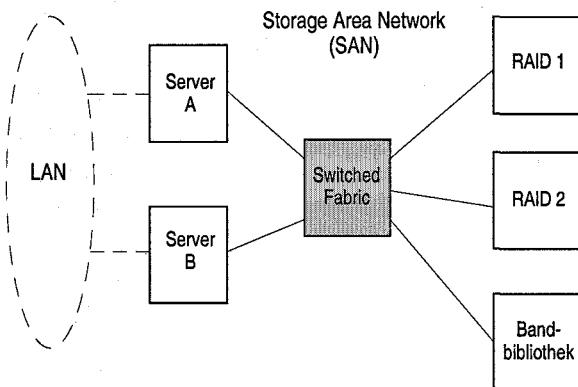


Bild 8-25. Symbolisiertes lokales Netz (LAN) mit einem über einen Switch gebildeten Speicher-Netz (SAN), z.B. als Fabric Topology des Fibre-Channel

Anmerkung. Festplatteneinheiten bei Servern sind häufig als *RAIDs* ausgelegt. Sie gewähren einerseits eine gewisse Datensicherheit durch mehrfache, d.h. redundante Speicherung der Dateien auf unterschiedlichen Festplattenlaufwerken. Sie erlauben andererseits ggf. auch schnellere Dateizugriffe, indem die Information so auf den Festplatten verteilt wird, daß Parallelzugriffe auf unterschiedliche Festplattenlaufwerke möglich sind. Backup-Geräte werden bei hohem Speicherbedarf ggf. als Bandbibliotheken ausgeführt, d.h. als Automaten, die Streamerlaufwerke nach Bedarf mit Kassetten laden können.

8.2.7 Universal Serial Bus

Der *Universal Serial Bus (USB)* ist eine von Intel und anderen Firmen eingeleitete Entwicklung mit dem Ziel, die bei PCs vorhandene Vielfalt an Geräteschnittstellen durch eine für alle Peripheriegeräte einheitliche Steckverbindung zu ersetzen und die Geräte an einem einzigen PC-Anschluß zusammenzufassen. Damit verbunden ist eine Senkung der Kosten und der Vorteil, Geräte unterschiedlicher Hersteller in gleicher Weise verwenden zu können. Der USB erlaubt es darüber hinaus, Geräte während des laufenden Betriebs an den Bus anzuschließen und sie von ihm zu trennen (*hot plug and play*), womit sich das Konfigurieren eines PC-Systems stark vereinfacht. Die ersten Standardisierungen, USB 1.0 und 1.1, sahen lediglich zwei relativ geringe Übertragungsgeschwindigkeiten vor, nämlich 1,5 Mbit/s (*low speed*) für langsame Geräte, wie Tastatur und Maus, und 12 Mbit/s (*medium* oder *full speed*, 1,5 Mbyte/s) für schnellere Geräte, wie Scanner, Drucker und Geräte mit digitalen Audiodaten. Der neuere Standard, USB 2.0, erlaubt zusätzlich die Datenübertragung mit 480 Mbit/s (*high speed*, 60 Mbyte/s), so daß auch Hintergrundspeicher am Bus betrieben und Videodaten unkomprimiert übertragen werden können. Übertragen werden die Daten in Paketen, entweder asynchron, z.B. bei nichtzeitkritischen Anwendungen, oder isochron, z.B. bei Audio- und Videoübertragungen, was die Aufrechterhaltung eines kontinuierlichen Datenstroms erfordert. Insgesamt können bis zu 127 Geräte am Bus betrieben werden. – Zur Vertiefung der folgenden Ausführungen sei auf [Anderson, Dzatko 2001] verwiesen.

Struktur. Die Struktur des USB basiert auf zentraler Verteilung. Wird mehr als nur ein Sternverteiler (*USB hub*) eingesetzt, so ergibt sich eine sternförmige Ausdehnung in mehreren Stufen, wie in Bild 8-26 an einem zweistufigen Beispiel gezeigt. Die Anbindung an den Rechner, hier an den PCI-Bus, geschieht durch einen sog. *USB-Host-Controller*, der die Wurzel der Struktur bildet und deshalb auch als *Root-Hub* bezeichnet wird. Er koordiniert sämtliche Busaktivitäten. Seine Anschlüsse sind entweder direkt mit Geräten bestückt, oder sie sind wiederum mit Hubs als Sternverteiler der nächsten Stufe verbunden. Ein solcher Hub ist beispielsweise Bestandteil einer Tastatur und versorgt mit seinen Anschläüssen die Tastatur selbst und eine Maus. Die Datenübertragung in einer solchen Struktur erfolgt *Punkt-zu-Punkt*, d.h. jeweils zwischen zwei benachbarten Geräten/Hubs. Diese wirken dabei als Repeater und reichen die Datenpakete entsprechend der Zieladreßangabe in den Paketen weiter. Der Root-Hub selbst führt seine Repeater-Funktion mit dem die Ein-/Auszug steuernden Prozessor (host, PC) aus, und erhält von diesem über im Speicher bereitgestellte Information die Auftragserteilungen. – Für den Anwender unterscheidet sich der USB in zwei wesentlichen Merkmalen vom Konkurrenten *FireWire*. Das Erweitern einer USB-Struktur geschieht hier durch den Einsatz von Hubs, bei FireWire durch einfache Geräteterverbindungen. Die USB-Übertragungen sind als sog. *Master-Slave*-Übertragungen auf den steuernden Prozessor (PC) angewiesen, bei FireWire erfolgen sie *Peer-to-Peer* zwischen den autonomen Übertragungspartnern.

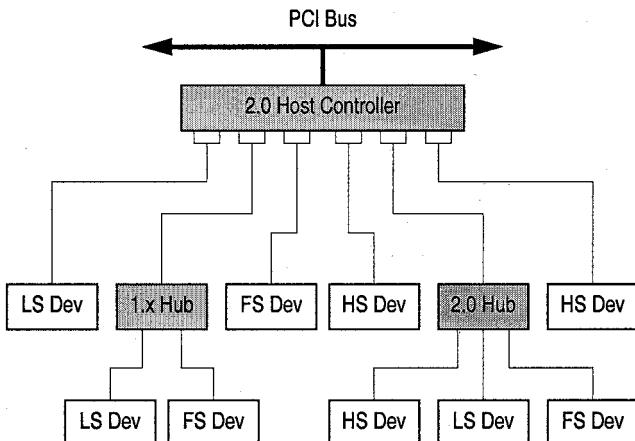


Bild 8-26. USB-Struktur mit Low-Speed-, Full-Speed- und High-Speed-Komponenten (nach [Anderson, Dzatko 2001])

Datenübertragung. Die drei möglichen Geschwindigkeiten, in Bild 8-26 durch die Abkürzungen LS (low speed), FS (full speed) und HS (high speed) kenntlich gemacht, lassen sich innerhalb einer solchen Struktur im Sinne einer Abwärtskompatibilität kombinieren. Eine High-Speed-Übertragung mit einem High-Speed-Gerät setzt voraus, daß in jedem Fall der Host-Controller, aber auch die auf dem Übertragungsweg liegenden Hubs dem Standard 2.0 genügen. Wird an einem USB 2.0 ein Gerät nach dem 1.x-Standard betrieben, so erfolgt die Übertragung vom Host-Controller aus so weit mit der hohen Geschwindigkeit, wie die dazwischenliegenden Hubs dem Standard 2.0 genügen. Danach wird die Übertragung auf die geringere Geschwindigkeit herabgesetzt. Zwei miteinander direkt verbundene Busteilnehmer führen ihre Übertragungen also immer mit der Geschwindigkeit des langsameren Partners durch.

Die Aufteilung der Übertragungskapazität des Busses erfolgt mittels Zeitrahmen, innerhalb derer die Geräte „Raum“ für ihre Übertragungspakete zugeteilt bekommen. Unterschieden werden dabei sog. Frames von 1 ms Dauer für Full-Speed-Übertragungen (12-MHz-Frames) mit einer Übertragungskapazität von 1,5 kbyte und sog. Microframes von 125 µs Dauer für High-Speed-Übertragungen (480-MHz-Frames) mit einer Übertragungskapazität von 7,5 kbyte (Bild 8-27). In welchem Umfang ein Gerät Raum innerhalb dieser Rahmen zugeteilt bekommt, hängt von der Art der Übertragung ab. Unterschieden werden hier vier Übertragungstypen (transfer types):

1. *Interrupt Transfer:* Er dient dazu, ein interrupt-gesteuertes Gerät auf seine Interruptanforderungen hin abzufragen, ersetzt also die Interrupt-Synchronisation durch Busy-Waiting. Im Sinne des damit verbundenen *Polling*-Verfahrens muß der Interrupt-Transfer periodisch durchgeführt werden, weshalb ihm eine feste Übertragungskapazität zu garantieren ist. Die Polling-Intervalle

werden vom Host-Controller festgelegt. Sie sind ggf. größer als der Rahmen, d.h., der Interrupt-Transfer findet nicht unbedingt in jedem Rahmen statt (Paketgröße $\leq 64/1024$ Bytes bei FS/HS).

2. *Bulk Transfer*: Er wird zur Übertragung größerer Datenmengen, die ohne zeitkritische Vorgaben auskommen, eingesetzt, z.B. zur Ausgabe von Dateien an einen Drucker. Dementsprechend erhält er, bezogen auf einen einzelnen Rahmen, keine Übertragungsgarantie (Paketgröße $\leq 64/512$ Bytes).
3. *Isochronous Transfer*: Er wird eingesetzt, wenn eine feste Übertragungsrate aufrechterhalten werden muß, z.B. bei Video- oder Audiodaten. Er findet dementsprechend periodisch statt und benötigt eine garantierte Übertragungskapazität. Deshalb werden ihm, zusammen mit dem Interrupt-Transfer, bis zu 90/80% der Kapazität eines jeden Rahmens reserviert. Anders als der Interrupt-Transfer erhält der isochrone Transfer seine Kapazitätszuweisung in jedem Rahmen (Paketgröße $\leq 1023/1024$ Bytes).
4. *Control Transfer*: Er dient zur Übertragung spezieller Anforderungen an die Geräte und tritt insbesondere während der Konfigurationsphase auf. Für ihn werden bis zu 10/20% der Kapazität eines jeden Rahmens reserviert (Paketgröße $\leq 64/64$ Bytes).

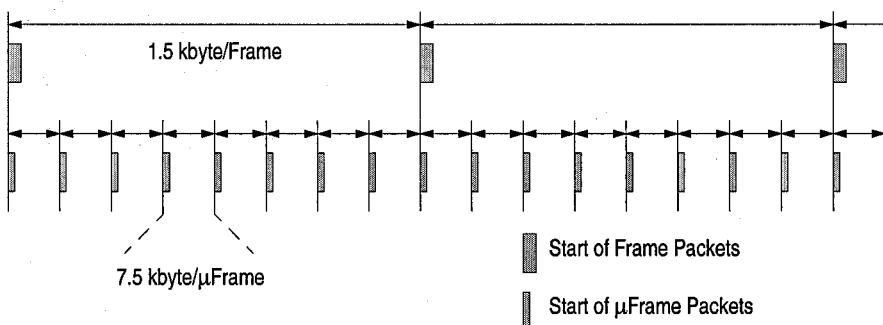


Bild 8-27. Aufteilung der Übertragungskapazität des USB mittels Frames und Microframe (nach [Anderson, Dzatko 2001])

Ein Gerät unterstützt ggf. nicht nur einen, sondern mehrere dieser vier Übertragungstypen und ermöglicht deren Auswahl über sog. Endpunkte (endpoints). Diese werden von der Geräte-Treibersoftware während der Konfigurationsphase anhand der Geräte-Deskriptoren für jedes einzelne Gerät ermittelt und sind somit vor der Übertragung bekannt. Ein Beispiel für ein Gerät mit mehreren Endpunkten ist der CD-ROM-Speicher: Für den Zugriff einer Dateiverwaltungssoftware bietet er als Endpunkt den Bulk-Transfer, für die Übertragung von Audiodaten den Endpunkt Isochronous-Transfer.

Sämtliche Übertragungen (*transactions*) werden vom Host-Controller ausgelöst, wodurch Buskonflikte vermieden und Übertragungskapazitäten garantiert werden

können. Sie bestehen üblicherweise aus drei aufeinanderfolgenden Paketen, wovon das erste ein sog. *Token-Packet* ist. Dieses sendet der Host-Controller entweder an alle Geräte, um ihnen den Beginn eines neuen Rahmens anzuzeigen, oder er sendet es an ein bestimmtes Gerät, um ihm die Details einer Übertragung, die er mit ihm durchführen möchte, mitzuteilen. Im diesem Fall enthält das Token-Paket neben der Geräteadresse die Vorgabe der Übertragungsrichtung (Lese- oder Schreibzugriff auf das Gerät) sowie die Angabe des Übertragungstyps (endpoint: Interrupt, Bulk, Isochronous oder Control).

Das zweite Paket ist dann das eigentliche *Data Packet* vom entsprechenden Übertragungstyp. Es wird, abhängig von der im Token-Paket angegebenen Übertragungsrichtung, entweder vom adressierten Gerät oder vom Host-Controller abgeschickt. (Im High-Speed-Modus können bei isochronen und bei Interrupt-Transfers bis zu drei Datenpakete aufeinanderfolgen, im Full-Speed-Modus ist nur ein Paket pro Rahmen erlaubt.) Auf das Datenpaket folgt, mit Ausnahme bei isochronen Transfers, als drittes Paket ein sog. *Handshake Packet*, mit dem der Empfänger der Daten dem Sender die Übertragung quittiert, sofern sie fehlerfrei war. Dies erkennt er u.a. an der zur Übertragungssicherung verwendeten CRC-Prüfinformation. Der Sender wartet eine bestimmte Zeit auf den Eingang dieses Pakets und meldet danach ggf. einen Übertragungsfehler. Der Host-Controller wird dann die Übertragung bis zu zweimal erneut versuchen. – In einer weiteren Variante des Token-Pakets können bestimmte Aufträge an ein Gerät übermittelt werden, z.B. Konfigurierungsvorgaben (setup).

USB-Schnittstelle. Das USB-Verbindungskabel besteht aus lediglich vier Leitungen, von denen zwei für die Signaltübertragung mit *differentieller* Signaldarstellung (D+, D-) benutzt werden. Mittels der beiden anderen Leitungen können Hubs und Geräte begrenzt mit Strom versorgt werden (VBus, GND). Für Full- und High-speed-Übertragungen werden Twisted-Pair-Kabel benötigt, für Low-speed-Übertragungen reichen unverdrillte und nichtabgeschirmte Leitungen aus (siehe auch Tabelle 8-1, S. 562). Zur Signalisierung der unterschiedlichen Buszustände weist die Schnittstelle diverse Sende- und Empfangseinrichtungen auf, die die beiden Signalleitungen entweder in differentieller oder in asymmetrischer Weise beeinflussen bzw. auswerten. Bild 8-28 soll davon einen Eindruck vermitteln, ohne daß die einzelnen Funktionen hier erläutert werden (siehe dazu [Anderson, Dzatko 2001]). Als grober Überblick über die Art der Buszustände seien jedoch die für die High-speed-Übertragung relevanten Zustände aufgezählt: Device Detection, Device Disconnect, Device Reset, Bus Idle, Differential Signaling, Start of Packet and Synchronization Sequence, End of Packet.

Die eigentliche Datensignalisierung geschieht in *NRZI-Codierung*. Das heißt, 0-Bits werden durch einen Pegelwechsel und 1-Bits durch gleichbleibenden Pegel dargestellt (Bild 8-29). Da sich der Empfänger anhand der Pegelwechsel mit dem Sender synchronisiert, muß dafür gesorgt werden, daß Pegelwechsel auch dann stattfinden, wenn längere 1-Bit-Folgen zu übertragen sind. Dazu fügt

der Sender nach jeweils sechs aufeinanderfolgenden 1-Bits ein 0-Bit in den Datenstrom ein, das vom Empfänger dann wieder entfernt wird (*bit stuffing*).

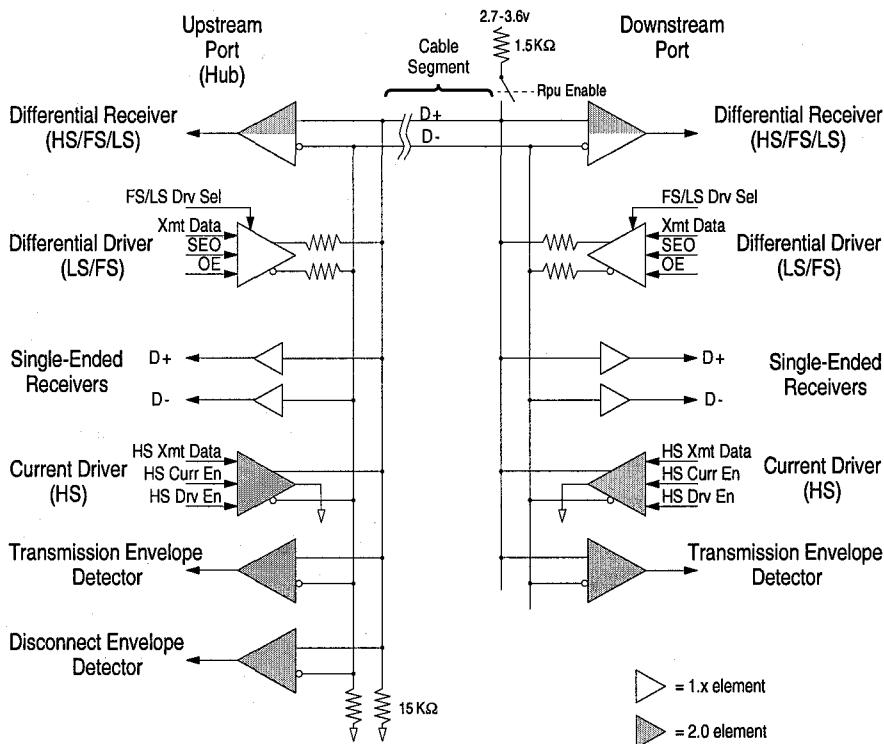


Bild 8-28. USB-Schnittstelle (nach [Anderson, Dzatko 2001])

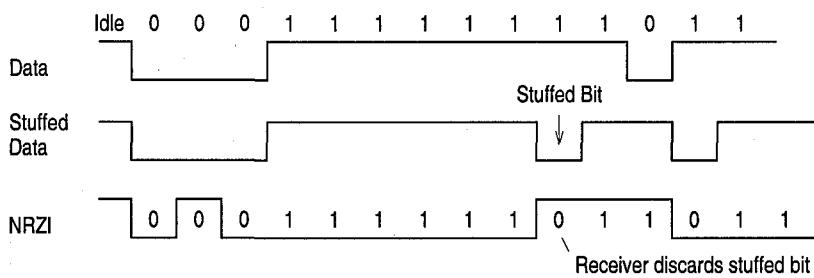


Bild 8-29. Signaldarstellung des USB (nach [Anderson, Dzatko 2001])

8.2.8 IEC-Bus

Der *IEC-Bus* (International Electrotechnical Commission) ist ein peripherer Bus, der den Datenaustausch zwischen Meß- und Anzeigegeräten erlaubt, wie dies

z.B. bei Laborinstrumentierungen gebräuchlich ist. Er basiert auf einem Vorschlag der Firma Hewlett Packard, der dann genormt wurde: zum einen als IEEE-Standard 488-1978, zum andern als IEC-Norm 625. Beide Normen unterscheiden sich lediglich in den Steckverbindungen. Der Bus wird auch als *General-Purpose-Interface-Bus (GPIB)* bezeichnet. – Die maximale Buslänge ist 20 m, die Übertragungsrate beträgt bei geringeren Entfernungen bis zu 1 Mbyte/s. Die Busteilnehmer haben entweder Sprecherfunktion (*talker*, z.B. Meßgeräte), Hörerfunktion (*listener*, z.B. Signalgeneratoren und Drucker), oder sie haben beide Funktionen (z.B. Meßgeräte mit einstellbaren Meßbereichen).

Busstruktur. Die IEC-Bus-Schnittstelle umfaßt acht bidirektionale Datenleitungen, die im Multiplexbetrieb auch als Adreßleitungen dienen, drei Handshake-Leitungen zur Steuerung der Datenübertragung und fünf allgemeine Steuerleitungen zur Bus- und Gerätesteuerung (Bild 8-30). Der Bus und die Gerätefunktionen werden von einer Steuereinheit (controller) verwaltet, die gleichzeitig als Talker und Listener arbeiten kann. Sie führt Initialisierungsaufgaben durch und legt jeweils fest, welche Busteilnehmer miteinander kommunizieren. Die Steuerfunktion wird üblicherweise von einem Rechner wahrgenommen, z.B. einem Mikroprozessorsystem; dadurch kann die Gerätebenutzung durch Programmierung flexibel gestaltet werden. Der Busanschluß an diesen Rechner erfolgt über einen besonderen Interface-Baustein, einen sog. General-Purpose-Interface-Adapter (*GPIA*).

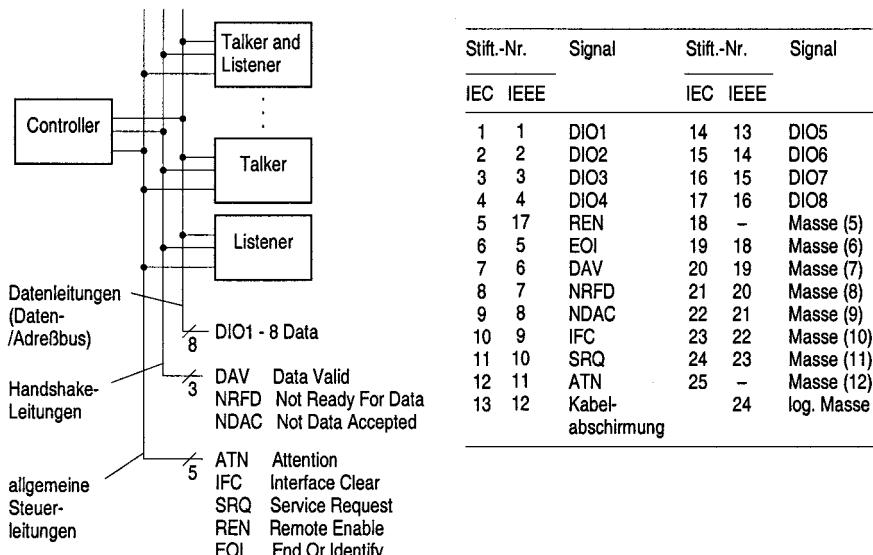


Bild 8-30. IEC-Bus. Bezeichnungen der Schnittstellensignale und Stiftbelegungen für den 25-poligen IEC-Steckverbinder (Cannon) und den 24-poligen IEEE-Steckverbinder (Amphenol). „Masse (i)“ steht für „Masseleitung, verdrillt mit Leitung i“

Buskommunikation. Während einer Datenübertragung, die aus einem oder mehreren aufeinanderfolgenden Bytes besteht, gibt es jeweils einen Talker und einen oder mehrere Listener. Welche Funktion ein Gerät hat, wird durch die Steuereinheit festgelegt, die als Talker die teilnehmenden Geräte zuvor über den Datenbus adressiert und damit aktiviert. Die Benutzung des Datenbusses als Adreßbus wird dabei durch das Steuersignal ATN angezeigt. (Nicht angewählte Geräte nehmen an der dann folgenden Datenübertragung nicht teil.) Jeweils 31 Talker- und 31 Listener-Adressen sind durch bestimmte ASCII-Codewörter definiert; bei einem Gerät mit beiden Funktionen sind die beiden Adressen in den fünf niederwertigen Bits identisch. Weitere Codewörter haben die Wirkung von Befehlen, die entweder einzelne Geräte betreffen (z.B. „Parallel-Poll-Configure“ für das Festlegen von Geräten für eine gleichzeitige Statusabfrage) oder für alle Geräte bestimmt sind (z.B. „Device-Clear“ zum Einstellen des Geräteausgangszustands).

Das Übertragen von Adressen, Befehlen und Daten verläuft nach einem einheitlichen asynchronen Busprotokoll, bei dem der Sprecher sich nach dem langsamsten Busteilnehmer richtet. Die Synchronisation erfolgt durch einen *Dreileitungs-Handshake* (Bild 8-31). Im Bild sind die Signalnamen entsprechend der IEC-Bus-Norm nichtnegiert angegeben, trotz 0-aktiver Logik.

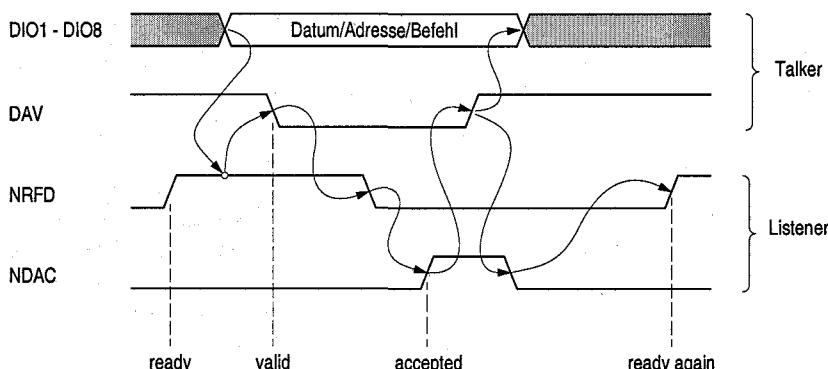


Bild 8-31. IEC-Bus. Dreileitungs-Handshake für die Übertragung eines Datenbytes, einer Adresse oder eines Befehls für einen Talker und einen oder mehrere Listener

Der Talker gibt das zu übertragende Byte auf den Datenleitungen DIO1 bis DIO8 aus und setzt, sobald die aktivierte Listener für eine Datenübertragung bereit sind, sein Data-Valid-Signal ($DAV = 0$). Die Listener-Bereitschaft wird durch das aus den Einzelsignalen durch verdrahtetes Oder gebildete inaktive Not-Ready-For-Data-Signal angezeigt ($NRFD = 1$). Mit dem Setzen von DAV schalten die Listener dann ihre NRFD-Signale in den Aktivzustand ($NRFD = 0$), womit sie anzeigen, daß sie vorerst für eine weitere Datenübertragung nicht bereit sind. Sie übernehmen danach das aktuelle Byte und signalisieren dies dem Talker durch das ebenfalls durch verdrahtetes Oder gebildete inaktive Not-Data-Accepted-Signal ($NDAC = 1$). Mit Erhalt dieses Signals setzt der Talker zunächst sein DAV -

Signal und dann seine Datensignale zurück. Erst nachdem alle Listener das übernommene Byte verarbeitet haben, zeigt das NRFD-Signal durch seinen Inaktivzustand ($\text{NRFD} = 1$) deren erneute Bereitschaft für eine Datenübertragung an. – Der Talker überträgt auf diese Weise ein oder mehrere Bytes und signalisiert das Ende der Gesamtübertragung durch ein bestimmtes ASCII-Zeichen oder mittels der Steuerleitung EOI.

Für eine ausführlichere Beschreibung der IEC-Bus-Funktionen und des Einsatzes eines GPIA siehe [Tietze, Schenk 1993].

8.3 Hintergrundspeicher

Hintergrundspeicher dienen zur Speicherung von Dateien (files), d.h. von Programmen und Daten, die von der Betriebssoftware eines Rechners mittels symbolischer Dateinamen und Dateiverzeichnisse (directories) verwaltet werden. Sie halten zum einen Dateien bereit, die für die aktuelle Bearbeitung in den Hauptspeicher geladen werden müssen, und dienen zum andern zur Sicherung (backup) und zur Archivierung von Dateien, aber auch für deren Austausch/Transport. In der Mikroprozessortechnik wie in der Rechnertechnik allgemein werden als Hintergrundspeicher sowohl *magnetische Speicher*, wie Festplatten-, Wechselplatten- und Streamer-Tape-Speicher, als auch *optische* und *magnetooptische Speicher* eingesetzt. Sie alle haben gegenüber dem in Halbleitertechnik aufgebauten Hauptspeicher den Vorteil, daß bei ihnen die gespeicherte Information beim Abschalten der Stromversorgung nicht verlorengeht. Man bezeichnet sie deshalb auch als *nichtflüchtige Speicher* (nonvolatile memories). Gegenüber Halbleiter-Speichern haben sie außerdem den Vorteil großer Speicherkapazitäten bei geringen Kosten pro Bit, jedoch den Nachteil größerer Zugriffszeiten (Tabelle 8-3). Diese Parameter entscheiden letztlich, welcher Speicher für welchen Zweck (Datenbereithaltung, Datensicherung, Datenarchivierung, Datenaustausch) eingesetzt wird. Für die Speicher mit auswechselbaren Speichermedien gilt grundsätzlich, daß die Kosten für Wechselplatten (bzw. Disketten) sehr viel höher sind als die für optische Medien (z.B. CD-R) und für Streamerbänder.

Alle genannten Hintergrundspeicher haben mechanisch bewegte Speichermedien, auf denen die Speicherung von Dateien grundsätzlich *blockweise* in festen Aufzeichnungsformaten erfolgt. Die Steuerung der Formatierungs-, Schreib- und Lesevorgänge obliegt dabei gerätespezifischen Steuereinheiten, sog. *Device-Controllern*, die als Steuerbausteine heutzutage meist in die Speichergeräte integriert sind. Die Anbindung eines Device-Controllers/Speichergeräts an den Systembus und damit an den Hauptspeicher erfolgt üblicherweise über eine weitere Steuereinheit, einen Host-Adapter, entweder Punkt-zu-Punkt oder mittels eines Peripheriebusses im Verbund mit anderen Geräten (siehe auch eingangs 8.2). Unabhängig von diesen beiden Möglichkeiten wird der Datentransport zwischen Hauptspeicher und Device-Controller/Speichergerät wahlweise programmgesteuert oder

DMA-gesteuert durchgeführt, wobei DMA-Übertragungen ggf. durch Zwischen speichern der Datenblöcke in einem FIFO-Speicher des Host-Adapters unterstützt werden.

Tabelle 8-3. Exemplarische Gegenüberstellung von Speicherkapazitäten, Kosten und Zugriffszeiten unterschiedlicher Speichermedien, geordnet nach den Kosten pro Gbyte. Bei den Bandspeichern liegen die mittleren Zugriffszeiten (*) um Größenordnungen über denen der anderen Speicherarten. Im Vergleich dazu ein SDRAM-Modul (DDR400)

Speicherart	Speicher- kapazität	Gesamt kosten (ca.) Euro	Kosten pro Gbyte (ca.) Euro	mittlere Zugriffszeit
SDRAM-Modul	256 Mbyte	100	400	40/2,5 ns
Floppy-Medium	1,44 Mbyte	0,26	180	80 ms
Zip-Medium	250 Mbyte	10	40	29 ms
SuperDisk-Medium	240 Mbyte	8	33	65 ms
MO-Medium	640 Mbyte	5	8	23 ms
CD-R-Medium	700 Mbyte	0,5	0,72	80 ms
VXA-2-Band	80/160 Gbyte	110	0,70	*
Festplatte (IDE intern)	160 Gbyte	100	0,63	8 ms
DVD+/-RW-Medium	4,7 Gbyte	2	0,43	95 ms
DVD+/-R-Medium	4,7 Gbyte	1	0,22	95 ms
LTO-2-Band	200/400 Gbyte	60	0,15	*

Anmerkung. Die in Tabelle 8-3 aufgeführten Disketten-Speichermedien (Floppy, Zip, SuperDisk) mit ihren vergleichsweise kleinen Speicherkapazitäten werden als Wechselmedien verstärkt durch nichtflüchtige Halbleiterspeicher, sog. Flash-Speicher ersetzt. Diese haben als Steckkarten, wie sie aus dem Konsumerbereich kommen (z.B. Digitalkameras), Abmessungen bis hinab zur Briefmarkengröße (mit Kapazitäten von derzeit bis zu 1 Gbyte). In anderer Bauform ähneln sie einem USB-Stecker und sind wie ein solcher am Universal Serial Bus anschließbar.

Zur Vertiefung der folgenden Ausführungen zu Hintergrundspeichern mit mechanisch bewegten Speichermedien sei auf [Völz 1996] verwiesen.

8.3.1 Floppy-Disk-Speicher

Der Floppy-Disk-Speicher war im PC-Bereich für viele Jahre der gebräuchlichste Wechselspeicher und wurde zur Sicherung und für das Verschicken kleiner Dateien eingesetzt, wie sie in der Textverarbeitung anfielen. Heute ist er fast gänzlich durch Wechselspeicher mit höheren Kapazitäten abgelöst. Dennoch soll er hier ausführlich beschrieben werden, da seine Technik im Prinzip auch die von Festplattenspeichern ist und er somit in vereinfachter Weise auch eine Vorstellung von deren Funktionsweise vermittelt.

Der Datenträger des Floppy-Disk-Speichers besteht aus einer biegsamen, kreisförmigen Kunststoffscheibe mit einer dünnen magnetisierbaren Beschichtung beider Oberflächen und ist in einer festen Plastikumhüllung untergebracht (*Diskette*, Floppy-Disk). Die Aufzeichnung der Information erfolgt bitseriell in konzentrischen Spuren (tracks), wobei beide Oberflächen genutzt werden. Der Zugriff auf die Spuren der beiden Oberflächen erfolgt über zwei Schreib-/Leseköpfe, die auf einem radial bewegbaren Arm montiert sind. Für den Zugriff werden die Schreib-/Leseköpfe wegen der Flexibilität der Diskette bis auf die Oberflächen abgesenkt; der Diskettenzugriff ist somit nicht verschleißfrei. Die Diskettenhülle weist für den Zugriff zwei Öffnungen auf. Heute üblich sind Floppy-Disk-Speicher mit einem sog. *Formfaktor* von 3,5 Zoll (früher 5,25 und auch 8 Zoll), was dem Maß der Breite der Diskettenhülle entspricht; der Durchmesser des eigentlichen Speichermediums ist etwas geringer.

Die wesentlichen Bestandteile eines Floppy-Disk-Speichers sind das mechanische Laufwerk (Floppy-Disk-Drive) mit der Schreib-/Leseeinrichtung sowie eine Steuerelektronik zur Umsetzung von Steuersignalen in Bewegungsabläufe, von Datensignalen in Magnetisierungen (und umgekehrt) und zur Erzeugung von Statusinformation. Der Anschluß an den Bus des Rechnersystems erfolgt über einen *Floppy-Disk-Controller (FDC)*, dessen wesentliche Aufgaben die Parallel-Serien-Umsetzung der Datenbytes bei der Ausgabe und die Serien-Parallel-Umsetzung der Datenbytes bei der Eingabe, die Auswertung von Adressierungsangaben für den Diskettenzugriff sowie die Erzeugung und Auswertung von Steuer- bzw. Statussignalen des Laufwerks sind (siehe auch Bild 8-12, S. 564).

Formatieren einer Diskette. Das Speichern der Daten erfolgt für jede Diskettenoberfläche in *Spuren*, die in *Sektoren* gleicher Größe unterteilt sind. Bild 8-32 zeigt den Aufbau eines solchen Sektors, bestehend aus einem Identifikationsfeld zur Adressierung des Sektors und dem eigentlichen Datenfeld. Das Identifikationsfeld beginnt mit einem Identifikationsbyte als Anfangsmarkierung (identification address mark, ID-AM), das zur *Zeichensynchronisation* dient. Es umfaßt weiterhin je ein Byte zur Angabe der Spurnummer, der Diskettenseite (oben/unten), der Sektornummer und der Datenblockgröße in codierter Form. Hinzu kommen zwei *CRC-Bytes* zur Blocksicherung (7.7.3). Das Datenfeld beginnt ebenfalls mit einem Identifikationsbyte (data address mark, Data-AM), gefolgt von dem eigentlichen Datenblock und von ebenfalls zwei Blocksicherungsbytes. Zwischen beiden Feldern wie auch zwischen den einzelnen Sektoren einer Spur existieren Lücken von mehreren Bytes bestimmter Codierung (gaps). Durch sie können beim Beschreiben von Datenfeldern kleinere Verschiebungen der Feldgrenzen, bedingt durch Laufzeitschwankungen des Motorantriebs, aufgefangen werden. Jede Spur hat darüber hinaus ein Gap am Spuranfang, das ein Markierungsbyte (index mark) mit einschließt, und ein Gap am Spurende. Der Spurbeginn selbst wird durch ein Loch (index hole) auf der Diskette festgelegt, das vom Laufwerk optoelektrisch abgefragt wird.

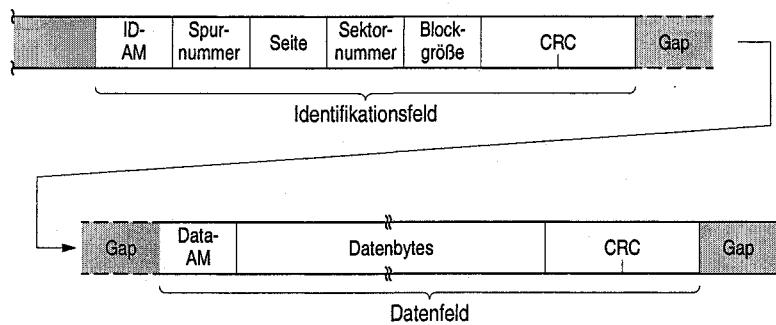


Bild 8-32. Sektorformat einer Diskette

Vor der ersten Nutzung einer Diskette müssen sämtliche Spuren mit ihren Identifikationsfeldern, Datenfeldern (mit beliebigen Datenbytes) und den Gaps bitweise beschrieben werden. Dabei werden die logisch aufeinanderfolgenden Sektoren einer Spur (aufsteigende Sektornummern) unmittelbar aufeinanderfolgend gespeichert. Man bezeichnet diesen Vorgang als *Formatieren*. Das Betriebssystem stellt dazu ein Formatierungsprogramm bereit, das den Floppy-Disk-Controller mit dem Beschreiben der Spuren beauftragt (Write-Track-Kommando) und ihn mit den Formatierungsdaten versorgt.

Bezüglich der Schreibdichte und der damit verbundenen Kapazität einer Diskette unterscheidet man drei Laufwerktypen, deren wichtigste technische Daten in Tabelle 8-4 aufgelistet sind: das *DD-Laufwerk* (double density), das *HD-Laufwerk* (high density) und das *ED-Laufwerk* (extra-high density). Sie arbeiten einheitlich mit 80 Spuren pro Diskettenseite, mit einer Sektorgroße von 512 Byte und mit einer Umdrehungszahl von 300 U/min (bei HD-Lauffwerken ggf. auch 360 U/min). Die Kapazitätsunterschiede ergeben sich durch eine unterschiedliche Sektanzahl pro Spur, nämlich 9, 18 bzw. 36 Sektoren. Für die zugehörigen

Tabelle 8-4. Technische Daten verschiedener Floppy-Disk-Laufwerke (3,5 Zoll)

Merkmal	DD-Laufwerk (Double Density)	HD-Laufwerk (High Density)	ED-Laufwerk (Extra-high Density)
Seitenanzahl (Diskette)	2	2	2
Spuranzahl pro Seite	80	80	80
Sektanzahl pro Spur	9	18	36
Sektorgroße Byte	512	512	512
Gesamtkapazität Mbyte	1	2	4
Nutzkapazität Mbyte	0,72	1,44	2,88
Umdrehungszahl U/min	300	300	300
Übertragungsrate kbit/s	250	500	1000

Floppy-Disk-Controller ergeben sich daraus unterschiedliche Übertragungsraten von 250, 500 bzw. 1000 kbit/s. Die Angaben zur Gesamtkapazität beziehen sich auf die unformatierte Diskette, die Angaben zur Nutzkapazität auf die Datenkapazitäten der formatierten Diskette, abzüglich einiger Sektoren für Information zur Diskettenverwaltung. – Zu den für die Datenaufzeichnung benutzten *Signalcodierungen* siehe Bild 8-35, S. 605.

Datenzugriff. Der Zugriff auf ein Datenfeld geschieht durch Positionieren des Zugriffsarms über der gewünschten Spur und durch Vergleich der vorgegebenen Spurnummer, Seitenangabe und Sektornummer mit den Inhalten aufeinanderfolgender Identifikationsfelder dieser Spur, bis vom Floppy-Disk-Controller eine Übereinstimmung aller Angaben festgestellt wird. Der Vergleich der Spurnummern dient dabei zur Überprüfung der korrekten Spurposition des Zugriffsarms. Die eigentliche Datenübertragung erfolgt für alle Bytes eines Blocks unmittelbar aufeinanderfolgend, d.h. sektorweise. Die Adressierung der gespeicherten Daten ist dementsprechend für die Spur- und Sektoranwahl direkt (wahlfrei) und für die Bytes innerhalb eines Sektors sequentiell. – Die *Zugriffszeit* auf ein Datenfeld hängt im wesentlichen von der Einstellzeit des Zugriffsarms ab; hinzu kommt die Zeit zum Auffinden des Sektors, die maximal einer Diskettenumdrehung entspricht. Aufgrund der unterschiedlichen Ausgangspositionen wird die Zugriffszeit eines Floppy-Disk-Speichers als Mittelwert angegeben.

Floppy-Disk-Controller-Baustein. Bild 8-33 zeigt den prinzipiellen Aufbau eines Floppy-Disk-Controller-Bausteins und dessen Datenwege. Die zwischen Hauptspeicher und Laufwerk zu übertragenden Bytes werden in einem Datenregister DR zwischengespeichert. Dieses kann vom Mikroprozessor oder DMA-Controller über die Systembusschnittstelle beschrieben und gelesen werden. Aufgrund der bitseriellen Datenspeicherung führt der Controller bei der Ausgabe eine Parallel-Serien-Umsetzung und bei der Eingabe eine Serien-Parallel-Umsetzung durch. Er benutzt dazu das Schieberegister SHIFT. Ein Spurnummerregister TNR zeigt die Spurnummer an, über der sich der Schreib-/Lesekopf befindet, ein Sektornummerregister SNR dient zur Aufnahme der zur Adressierung eines Sektors erforderlichen Sektornummer. Die Auswahl einer Controller-Operation erfolgt durch Laden eines Steuerbytes in das Kommandoregister CR; das Statusregister SR zeigt den Zustand des Controllers an.

Kommandos. Die wichtigsten Kommandos sind:

- **WRITE TRACK.** Die unter dem Schreib-/Lesekopf befindliche Spur wird mit all ihren Bytes (auch den Gap-Bytes) beschrieben, d.h. formatiert, wobei das Format vorgegeben werden kann.
- **READ TRACK AND VERIFY.** Alle Bytes der unter dem Schreib-/Lesekopf befindlichen Spur werden gelesen und dabei die CRC-Überprüfung für die Identifikations- und die Datenfelder durchgeführt (verify).

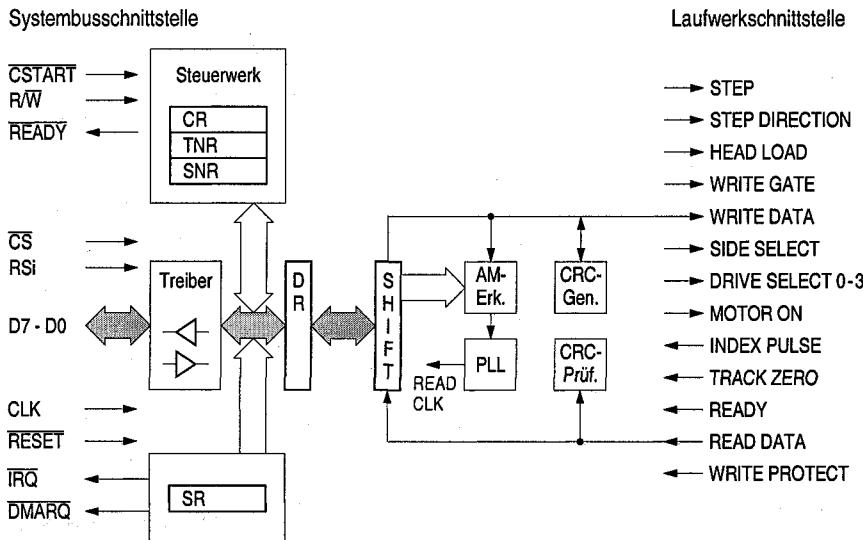


Bild 8-33. Struktur eines Floppy-Disk-Controllers

- SEEK TRACK ZERO. Der Schreib-/Lesekopf wird über der äußersten Spur positioniert (Spur 0) und das Spurnummerregister mit Null geladen.
- STEP IN, STEP OUT. Der Schreib-/Lesekopf wird um eine Spur nach innen in Richtung Spur 79 bzw. um eine Spur nach außen in Richtung Spur 0 bewegt. Der Inhalt des Spurnummerregisters wird entsprechend um Eins erhöht oder vermindert.
- SEEK TRACK. Der Schreib-/Lesekopf wird über derjenigen Spur positioniert, deren Nummer im Datenregister vorgegeben ist. Der Controller bewegt dazu den Schreib-/Lesekopf, bis die dabei im Spurnummerregister aktualisierte Spurnummer mit dem Datenregisterinhalt übereinstimmt.
- SINGLE SECTOR READ. Die Datenbytes eines durch die Spurnummer und die Sektornummer adressierten Sektors der aktuellen Spur werden gelesen. Dabei wird wahlweise die Blocksicherungsprüfung durchgeführt (verify). Stimmen die im Spurnummer- und im Sektornummerregister vorgegebenen Adressierungssangaben nicht mit den Angaben eines der Identifikationsblöcke der Spur überein, so wird die Kommandoausführung als fehlerhaft abgebrochen.
- SINGLE SECTOR WRITE. Die Bytes eines Datenblocks werden in den mit der Sektornummer adressierten Sektor der aktuellen Spur geschrieben. Dabei werden die Blockprüfbytes generiert und an den Datenblock angefügt. Wie beim Lesekommando wird eine Überprüfung der Spurnummer durchgeführt.
- READ SECTOR MULTIPLE, WRITE SECTOR MULTIPLE. Diese beiden Kommandos erlauben das Lesen bzw. Schreiben mehrerer aufeinanderfolgen-

der Sektoren, wobei die Adresse des ersten Sektors im Sektornummerregister und die Sektoranzahl in einem in Bild 8-33 nicht angegebenen Zählregister vorgegeben werden.

Bei den Kommandos für das Lesen und das Schreiben von Sektoren bestimmt eines der Kommandobits die Diskettenoberfläche (Side 0 oder 1).

Synchronisation und Status. Die Synchronisation der einzelnen Byteübertragungen erfolgt mittels der Anforderung „Data Request“. Sie wird im Statusregister des Controllers angezeigt und kann wahlweise dort abgefragt werden, oder sie wird als *Interruptanforderung* oder als *DMA-Anforderung* ausgewertet. Der Floppy-Disk-Controller erzeugt diese Anforderungen in festen Zeitabständen, abhängig von der Übertragungsrate. Wird nach einer solchen Anforderung das Datenregister bei der Ausgabe nicht rechtzeitig geladen bzw. bei der Eingabe nicht rechtzeitig gelesen, so wird das in der Ausführung befindliche Kommando abgebrochen und im Statusregister der Fehlerzustand „Lost Data“ angezeigt.

Die Synchronisation auf der Kommandoebene geschieht durch das Controller-Statusbit „Busy“, das während einer Kommandoausführung gesetzt ist und mit Abschluß des Kommandos zurückgesetzt wird. Das Rücksetzen kann wahlweise abgefragt werden, oder es wird als Interruptanforderung benutzt. – Weitere Statusangaben sind „Not Ready“ (Laufwerk nicht bereit), „Seek Error“ (Spur nicht gefunden), „Record Not Found“ (Sektor nicht gefunden), „CRC-Error“ (Fehleranzeige beim Lesen) und „Write Protect“ (Diskette ist schreibgeschützt).

Schnittstelle. Der Floppy-Disk-Controller stellt die Verbindung zwischen dem Systembus und einem Floppy-Disk-Laufwerk her und hat dementsprechend Signalanschlüsse für beide Schnittstellen (Bild 8-33). Auf der Systembusseite sind das die bekannten Anschlüsse für die Anwahl eines Bausteins und den Zugriff auf dessen Register, hier ergänzt um eine Signalleitung DMARQ für DMA-Anforderungen. Auf der Laufwerkseite sind das (in Anlehnung an den aus der Anfangszeit stammenden Shugart-Firmenstandard) die Ausgänge

- WRITE DATA für die bitserielle Datenausgabe,
- STEP zum Weitersetzen des Schreib-/Lesekopfes um eine Spur,
- STEP DIRECTION zur Vorgabe der Bewegungsrichtung von STEP,
- HEAD LOAD zur Absenkung des Schreib-/Lesekopfes auf die Diskettenoberfläche für den Schreib- oder Lesezugriff,
- WRITE GATE zur Angabe der Datenflußrichtung Lesen bzw. Schreiben,
- SIDE SELECT zur Anwahl der Diskettenseite,
- DRIVE SELECT_i zur Anwahl eines von mehreren angeschlossenen Laufwerken und
- MOTOR ON zum Anschalten des Motors

sowie die Eingänge

- READ DATA für die bitserielle Dateneingabe,
- INDEX PULSE zur Anzeige des Index-Lochs,
- TRACK ZERO zur Anzeige, ob der Schreib-/Lesekopf über der Spur 0 steht,
- READY zur Anzeige der Bereitschaft des Laufwerks für eine Übertragung (Diskette ist eingelegt, Motor hat Nenndrehzahl erreicht, siehe auch Statusbit Ready) und
- WRITE PROTECT zur Anzeige, ob die Diskette mit einer Schreibschutzmarke versehen ist (siehe auch Statusbit Write-Protect).

Eine ausführliche Beschreibung von Floppy-Disk-Speichern befindet sich in [Mueller 1995].

8.3.2 Festplattenspeicher

Technische Merkmale. Beim Festplattenspeicher (*hard disk, HD, Festplatte*) bildet das Speichermedium mit dem Laufwerk eine feste Einheit, ist also von diesem nicht zu trennen. Im Unterschied zum Floppy-Disk-Speicher als einem der Wechselplattenspeicher besteht es aus einem oder mehreren übereinander angeordneten *starren* Aluminium- oder Glasträgern mit beidseitiger magnetischer Beschichtung, was es erlaubt, die Schreib-/Leseköpfe beim Datenzugriff auf einem dünnen Luftkissen, d.h. verschleißfrei, über deren Oberflächen „schweben“ (besser: „fliegen“) zu lassen. Dadurch können diese Plattenlaufwerke mit sehr viel höheren Umdrehungszahlen als Floppy-Disk-Laufwerke arbeiten (typisch 5 400 und 7 200 U/min, aber auch 10 000 und 15 000 U/min), wodurch sich höhere Schreibdichten, d.h. größere Speicherkapazitäten und größere Übertragungsraten realisieren lassen. Diese Technik erfordert dafür aber eine größere Präzision in der Mechanik, die entsprechend höhere Gerätekosten zur Folge hat. Außerdem wächst mit der Umdrehungszahl auch der Geräuschpegel, weshalb derzeit Speicher mit 7 200 U/min gegenüber solchen mit höheren Umdrehungszahlen empfohlen werden.

Festplatten haben gegenüber Wechselplatten außerdem kurze mittlere Zugriffszeiten von typisch 7 bis 15 ms sowie hohe Übertragungsraten von im Mittel bis zu 30 Mbyte/s und mit Spitzenwerten bis zu 70 Mbyte/s. Bei IDE-Festplatten mit einem Formfaktor von 3,5 Zoll (3,5-Zoll-IDE-Platten), wie sie bei PCs eingesetzt werden, beträgt die Speicherkapazität einer Scheibe derzeit 80 Gbyte, bei 2,5-Zoll-IDE-Platten, wie sie für Notebooks typisch sind, 40 Gbyte. Üblich sind in beiden Fällen Laufwerke mit 2 Scheiben. Darauf hinaus gibt es aber auch Festplatten mit größeren Formfaktoren von 5,25, 8 und 14 Zoll, z.B. für den Einsatz bei Großrechnern, sowie solche mit kleineren Formfaktoren von 1,8 und 1 Zoll, z.B. bei Geräten im Konsumerbereich.

Festplattenspeicher dienen zur ständigen Programm- und Datenbereithaltung für den Hauptspeicher und sind deshalb feste Komponenten von Rechnern mit IDE/ATA-, SATA- oder SCSI-Anschluß. Sie können aber auch als transportable Speichermedien eingesetzt werden, z.B. als Rechnereinschübe oder als prozessorexterne Geräte mit z.B. SCSI-, FireWire- oder USB-Kabelanschluß. Die Einführung der SAS-Technik (Serial Attached SCSI) steht noch aus.

Datenspeicherung. Der Datenträger eines Festplattenspeichers besteht, wie bereits erwähnt, entweder aus nur einer Magnetplatte, oder aus einem Stapel übereinander angeordneter Magnetplatten, wobei jeweils beide Oberflächen nutzbar sind. Für jede Plattenoberfläche gibt es dazu einen eigenen Schreib-/Lesekopf; die zugehörigen Arme sind zu einem Schreib-/Lesekamm miteinander verbunden und werden gemeinsam bewegt (Bild 8-34). Der Plattenstapel bietet neben der insgesamt größeren Speicherkapazität den Vorteil, daß nach Positionieren des Kamms ein ganzer *Zylinder*, d.h. mehrere übereinanderliegende Spuren, ohne weitere Armbewegung erreichbar ist. Entsprechend werden die Datenblöcke einer Datei zylinderweise gespeichert, um die Zugriffszeiten zu minimieren.

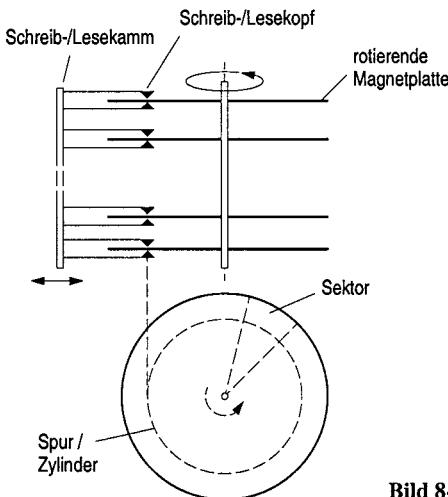


Bild 8-34. Aufbau eines Festplattenspeichers

Formatierung. Die Formatierung des Datenträgers erfolgt nach dem gleichen Prinzip wie bei einer Floppy-Disk durch Aufteilen einer *Spur* in *Sektoren*, d.h. in Identifikations- und Datenfelder mit dazwischenliegenden Gaps. Die Adressierungsinformation eines Sektors besteht jetzt aus einer Zylindernummer (bisher Spurnummer), einer Kopfnummer (bisher Seitenangabe) und der Sektornummer. Zusätzlich können Datenfelder, die aufgrund eines Defektes der Magnetisierungsschicht nicht mehr benutzbar sind, in ihrem Identifikationsfeld als sog. Bad Blocks gekennzeichnet werden.

Die Datenfeldgröße eines Sektors beträgt üblicherweise 512 Byte; gegenüber einer Floppy-Disk finden jedoch aufgrund der höheren Schreibdichte wesentlich mehr Sektoren in einer Spur Platz. Im einfachsten Fall der Spurunterteilung werden alle Spuren mit der gleichen Anzahl an Sektoren belegt. Dabei nimmt man allerdings in Kauf, daß die äußeren Spuren, die ja länger als die inneren sind, schlecht genutzt werden. Der Vorteil dabei ist jedoch, daß die Übertragungsrate bei der benutzten konstanten Umdrehungszahl spurunabhängig ist. – Bei Festplattenspeichern hoher Leistungsfähigkeit nutzt man hingegen die Längenunterschiede der Spuren. Dementsprechend sieht man bei der Formatierung in den äußeren Spuren mehr Sektoren als innen vor und erreicht so größere Speicherkapazitäten. Um das dabei auftretende Problem spurabhängiger Übertragungsraten zu minimieren, hält man die Sektoranzahl für eine bestimmte Anzahl benachbarter Spuren konstant, bildet also Zonen. Man bezeichnet dieses Vorgehen deshalb auch als *Multiple-Zone-Recording* oder *Zone-Bit-Recording (ZBR)*.

Datenaufzeichnung. Die Aufzeichnung der Daten bei Floppy-Disk- und Festplattenspeichern erfolgt, wie bereits gesagt, bitseriell. In Analogie zur Datenübertragung in Netzen (7.7.1) benötigt man auch hier Datendarstellungen, die es ermöglichen, sich beim Lesen der Daten mit dem Datenstrom zu synchronisieren. Beim einfachsten Aufzeichnungsverfahren, der *Frequenzmodulation* (frequency modulation, FM), wird dazu in jeder sog. Bitzelle zunächst ein Taktimpuls und dann je nach Wertigkeit des Bits ein Datenimpuls (1-Bit) oder kein Datenimpuls (0-Bit) gespeichert (Bild 8-35a). Man spricht bei dieser Art der Aufzeichnung, die heute zwar nicht mehr gebräuchlich ist, aber sehr gut das Prinzip der Daten- und Taktepeicherung verdeutlicht, von einfacher Schreibdichte (*single density, SD*). Ein verbessertes Verfahren, die modifizierte Frequenzmodulation (modified frequency modulation, MFM), reduziert die Anzahl der Impulse gegenüber dem FM-Verfahren auf die Hälfte und verdoppelt damit die Schreibdichte (*double density, DD*). Bei ihm wird ein 1-Bit durch einen Datenimpuls und ein 0-Bit durch keinen Impuls dargestellt. Um bei 0-Bit-Folgen dennoch die Synchroni-

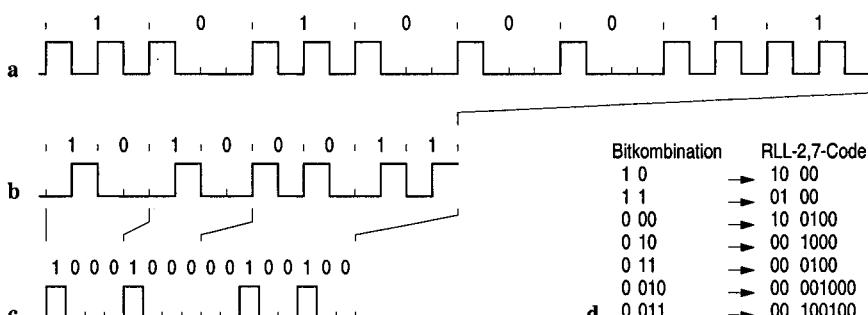


Bild 8-35. Datenaufzeichnung bei Festplattenspeichern am Beispiel der Bitfolge 10100011 (nach [Bähring 1994]). a Frequenzmodulation (FM), b Modifizierte Frequenzmodulation (MFM), c Lauflängenbegrenzung (RLL), d Umcodierungstabelle, wie sie bei c verwendet wurde

sation aufrecht erhalten zu können, wird ab dem zweiten 0-Bit je ein Taktimpuls aufgezeichnet. Dieser unterscheidet sich von einem Datenimpuls durch eine andere Position in der Bitzelle (Bild 8-35b). Das MFM-Verfahren wird insbesondere bei Floppy-Disk-Speichern (DD-Laufwerk) angewandt, bei Festplatten hat es inzwischen an Bedeutung verloren.

Das für Festplattenspeicher gebräuchlichste Aufzeichnungsverfahren ist die sog. *Lauflängenbegrenzung* (*run length limited, RLL*), für die es eine Vielzahl von Varianten gibt. Bei ihr wird ein aufzuzeichnendes Datenbit im Zusammenhang mit seinen Folgebits betrachtet, indem Bitfolgen (unterschiedlicher) Längen extrahiert und diese in Bitfolgen doppelter Bitanzahl umcodiert werden. Ziel ist es dabei, 0-Bit-Folgen auf eine Mindest- und eine Maximallänge zu begrenzen. Bei dem im Bild 8-35c gezeigten Beispiel des RLL-2,7-Verfahrens (Mindestlänge 2, Maximallänge 7) läßt sich die Schreibdichte gegenüber dem MFM-Verfahren im Verhältnis 3:2 erhöhen. Die hierbei verwendete Codierungstabelle (Bild 8-35d) ist eine von vielen möglichen. – RLL-Verfahren variieren nicht nur in den Codetabellen, sondern auch in den Längenvorgaben. So können z. B. auch das FM- und das MFM-Verfahren als RLL-Verfahren betrachtet werden, nämlich als RLL-0,1- und RLL-1,3-Verfahren.

Zu den Signalverläufen in Bild 8-35, Teilbilder a bis c, ist zu bemerken, daß die eigentliche Darstellung auf dem Speichermedium durch Wechsel des magnetischen Flusses erfolgt. Da solche Wechsel jeweils nur an den steigenden Flanken der im Bild gezeigten Daten- und Taktimpulse auftreten, ist die tatsächliche Aufzeichnungsfrequenz halb so groß wie die der Impulsdarstellung. Zu dieser „korrekten“ Darstellung siehe z. B. [Mueller 1995]; die Impulsdarstellung wurde hier aus Gründen der besseren Verständlichkeit gewählt (siehe dazu auch [Bähring 1994]).

Sektor-Interleaving. Voraussetzung für einen schnellen Plattenzugriff ist, daß die Sektoren einer Spur entsprechend ihren fortlaufenden Nummern direkt hintereinander angeordnet sind (also in ihrer logischen Reihenfolge) und daß die Laufwerksteuerung in der Lage ist, alle Sektoren während einer einzigen Umdrehung der Platte nacheinander zu lesen bzw. zu beschreiben. Ist sie dazu nicht in der Lage, so kann auf einen Folgesektor jeweils erst nach einem neuen Umlauf der Platte zugegriffen werden, also mit großer Verzögerung. Abhilfe schafft hier das sog. *Sektor-Interleaving*, bei dem der jeweilige logische Folgesektor um ein oder mehrere physische Sektoren gegenüber seinem Vorgänger versetzt gespeichert wird. – Die Anzahl n an Sektoren, um die der Folgesektor gegenüber seinem Vorgänger versetzt ist, drückt man durch den *Interleave-Faktor* $n:1$ aus. Bei einem Interleave-Faktor von z. B. 3:1 liegen zwischen zwei logisch aufeinanderfolgenden Sektoren 2 physische Sektoren.

Festplatten-Cache. Die zwischen Festplattenanschluß und Systembus mögliche Datenübertragungsrate, z. B. 100 Mbyte/s bei Ultra ATA/100 (8.2.1), ist üblicherweise sehr viel höher als die vom Speichermedium her erreichbare. Letztere liegt,

wie eingangs dieses Abschnittes angegeben, im Mittel bei derzeit bis zu 30 Mbyte/s und ist durch die Taktfrequenz begrenzt, mit der die Bits auf die Magnetplatte geschrieben bzw. von ihr gelesen werden. Das heißt, sie ist einerseits von der Umdrehungszahl der Platte und andererseits von der Schreibdichte der Aufzeichnung abhängig. Die Schreibdichte ist beim Multiple-Zone-Recording außerdem variabel und hängt davon ab, ob der Zugriff auf eine äußere Spur oder weiter innen erfolgt. Um dieses Mißverhältnis der Übertragungsraten zu verbessern, ergänzt man Festplattenspeicher durch Caches mit Speicherkapazitäten von bis zu mehreren Mbyte. Ein solcher Festplatten-Cache erlaubt dann als Halbleiterspeicher einen schnelleren Zugriff, als er auf die Platte selbst möglich ist.

Wie bei den zwischen Hauptspeicher und Prozessor eingesetzten Caches gibt es auch bei Platten-Caches Optimierungsstrategien. Gebräuchlich ist z.B. die *Read-Ahead-Strategie*, bei der bei einem Lesezugriff jeweils die gesamte Spur oder der gesamte Zylinder in den Cache geladen wird, in der Annahme, daß ein Teil der zunächst nicht angesprochenen Sektoren bei Folgezugriffen benötigt wird. Beim Einsatz des Cache für Schreibvorgänge ist besondere Vorsicht geboten. Hier kann es ggf. zum Datenverlust kommen, z.B. dann, wenn der Rechnerbenutzer den Rechner abschaltet, bevor das Schreiben auf die Platte abgeschlossen ist. Dieses Problem kann auch beim „Absturz“ des Rechners auftreten. – Die ohne Cache mögliche und über längere Zeit aufrechterhaltbare Übertragungsrate bezeichnet man auch als *sustained Data-Transfer-Rate* und die durch den Cache unterstützte und nur für kurze Zeit mögliche Übertragungsrate als *Burst-Data-Transfer-Rate*.

8.3.3 Solid-State-Disk und RAM-Disk

Festplattenspeicher sind zwar die vom Preis-Leistungs-Verhältnis her günstigsten Hintergrundspeicher zur Datenbereithaltung für den Hauptspeicher. Bei Anwendungen mit „regem“ Plattenverkehr bilden sie jedoch ein Hemmnis, das die Rechnerleistung stark beeinträchtigen kann. Abhilfe schaffen hier *Solid-State-Disks (SSDs)*. Sie bestehen aus Halbleiterspeichern, verhalten sich aber gegenüber einem Ein-/Ausgabecontroller (z.B. IDE, SCSI oder Fibre-Channel) wie Festplattenspeicher. Der Vorteil liegt dabei in der für Halbleiterspeicher kürzeren Zugriffszeit und im Wegfall der für die Positionierung des Plattenarms und für die Rotation der Speichermediums benötigten Zeiten. Als Halbleiterspeicher werden entweder Flash-Speicher, z.B. bei PCs im Industriebereich, oder die sehr viel schnellere SDRAM-Speicher eingesetzt, z.B. bei Servern der höheren Leistungsklasse. Die Speicherkapazitäten liegen für zwei konkrete Beispiele bei 512 Mbyte bis 34,8 Gbyte für eine 3,5-Zoll-Flash-Einheit und bei 16 bis 128 Gbyte für eine SDRAM-Einheit größerer Bauart. Nachteilig sind natürlich die gegenüber Festplatten höheren Kosten. Gegen Stromausfall werden die Inhalte von SDRAM-Einheiten durch Pufferbatterien geschützt.

Rechnersysteme ohne Solid-State-Disk machen sich den Vorteil von „Plattenzugriffen“ ohne mechanische Verzögerungszeiten ggf. durch eine sog. *RAM-Disk* zunutze. Hierbei wird ein Teil des Hauptspeichers zur Nachbildung eines Speichers mit der Zugriffsverwaltung eines Festplattenspeichers herangezogen. Die auf diese Weise realisierbare Speicherkapazität ist jedoch zwangsläufig geringer als bei einer Solid-State-Disk. Sie wird entweder vom Benutzer voreingestellt, oder das Betriebssystem legt sie anhand des Bedarfs und der Verfügbarkeit von Speicherplatz dynamisch fest.

8.3.4 Wechselplattenspeicher

Bei den *Wechselplattenspeichern* ist das Speichermedium, das aus nur einer einzigen magnetisierbaren, meist flexiblen Scheibe besteht, vom Laufwerk trennbar, weshalb sich diese Speicher insbesondere für die Datensicherung und für den Datentransport eignen. Zu ihnen zählt auch der in 8.3.1 ausführlich beschriebene *Floppy-Disk-Speicher*, dessen Einsatz aufgrund der geringen Speicherkapazitäten der Disketten von 1,44 Mbyte und 720 Kbyte stark zurückgegangen ist. Neuere Wechselplattenspeicher haben demgegenüber sehr viel höhere Speicherkapazitäten und Übertragungsraten und außerdem wesentlich kürzere mittlere Zugriffszeiten. Der direkte Nachfolger des Floppy-Disk-Speichers ist der sog. *SuperDisk-Speicher* mit Speicherkapazitäten von 120 und 240 Mbyte (LS-120/240). Er ist abwärtskompatibel, d.h., er kann neben den LS-120- und LS-240-Disketten auch die 1,44-Mbyte- und 720-Kbyte-Disketten des Floppy-Disk-Speichers lesen und beschreiben. Ein weiterer Wechselplattenspeicher mit großer Verbreitung ist der *Zip-Speicher*, der hinsichtlich der Kapazitäten der Disketten mehrere Entwicklungsstufen durchlaufen hat, nämlich zunächst 100 Mbyte, dann 250 Mbyte und schließlich 750 Mbyte. Mit der Ausnahme, daß das 750-Mbyte-Laufwerk 100-Mbyte-Disketten nicht beschreiben kann, besteht auch hier Abwärtskompatibilität. – Wechselplattenspeicher gibt es mit dem Formfaktor 3,5 Zoll als rechnerinterne Geräte mit z.B. IDE/ATA-Anschluß und als rechnerexterne Geräte mit z.B. SCSI-, USB- oder FireWire-Anschluß.

8.3.5 Optische Plattenspeicher

Optische Plattenspeicher (optical disks) sind Hintergrundspeicher mit einem den Festplattenspeichern ähnlichen Informationsträger, nämlich einer starren, rotierende Scheibe, die jedoch in Durchmesser (12 cm) und Handhabung (Wechselmedium ohne Hülle) der Audio-Compact-Disc (Audio-CD) entspricht. Auch der Zugriff erfolgt wie bei der Compact-Disc, nämlich berührungslos mit Laserlicht. Zu unterscheiden sind Speicher mit herkömmlicher CD-Technik und mit der leistungsfähigeren DVD-Technik, beide Techniken wiederum mit Varianten hinsichtlich

- Nurlesbarkeit (ROM, Read-Only Memory),
- Einmalbeschreibbarkeit (R, Recordable) und
- Wiederbeschreibbarkeit (RW, ReWritable).

Compact Disc – CD. Die *CD-ROM (Read-Only Memory)* ist ein nur lesbares Medium mit Kapazitäten von 650 Mbyte und darüber. Die Datenaufzeichnung erfolgt anders als bei Festplattenspeichern nicht in konzentrischen Spuren, sondern in nur einer einzigen, spiralförmigen Spur, innen beginnend (Bild 8-36). Dennoch werden unterschiedliche (logische) *Spuren* (tracks) in Form von Bereichen unterschieden, die wiederum in *Sektoren* unterteilt sind. Dargestellt werden die Daten durch längliche Vertiefungen in einer ebenen, reflektierenden Schicht der Scheibe, die bei der Herstellung durch Pressung (also mechanisch) mittels eines Masters erzeugt werden. Die unveränderten Bereiche der Schicht werden als *Land*, die Vertiefungen als *Pit* bezeichnet. Die Übergänge von Land nach Pit und von Pit nach Land werden jeweils als Eins gewertet. Das Lesen der Information basiert nun darauf, daß der Lichtstrahl eines im Lesekopf des Laufwerks untergebrachten Lasers von der Platte unterschiedlich reflektiert wird, je nachdem, ob es die Oberfläche oder eine Vertiefung trifft.

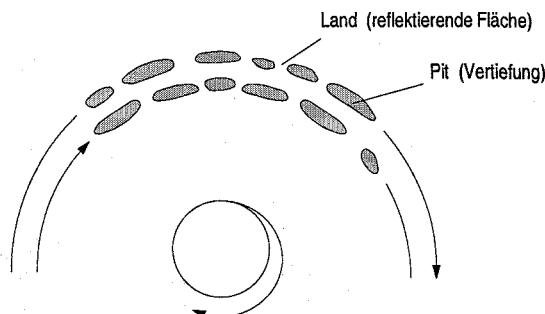


Bild 8-36. Datenaufzeichnung bei der CD-ROM (nach [Strass 1994]): spiralförmige Spurführung mit Datendarstellung durch die Übergänge zwischen Land-Bereichen (reflektierende Schicht der Platte) und Pit-Bereichen (Vertiefungen in der reflektierenden Schicht)

Bezugspunkt für die Geschwindigkeitsangaben bei CD-ROM-Laufwerken ist die sog. einfache *Geschwindigkeit* von Audio-CD-Laufwerken mit 150 kbyte/s. Gebräuchlich ist ein Vielfaches dieser Geschwindigkeit, womit sich auch die Übertragungsrate um diesen Faktor erhöht, allerdings ggf. nicht über die gesamte CD konstant bleibt. Bei einem Antrieb mit konstanter Drehzahl (constant angular velocity, CAV) erhöht sich die Übertragungsrate von innen nach außen, da ja die äußeren „Spuren“ länger sind und damit mehr Inhalt haben als die inneren. Das führt z.B. zu einer 16fachen Geschwindigkeit im Innenbereich (2,4 Mbyte/s) und einer 40fachen Geschwindigkeit im Außenbereich (6 Mbyte/s). Je voller also die CD ist, desto besser ist die Zunahme der Übertragungsleistung nutzbar. Andere CD-ROM-Laufwerke arbeiten, um die Transferleistung einigermaßen konstant zu

halten, im Innenbereich mit konstanter Drehzahl und im Außenbereich mit konstanter Spurgeschwindigkeit (constant linear velocity, *CLV*), d.h. mit gleichbleibender Übertragungsgeschwindigkeit. Schnelle CD-ROM-Laufwerke führen ihre Lesezugriffe derzeit mit bis zu 52facher Geschwindigkeit, d.h. mit bis zu 7,8 Mbyte/s durch. – Eingesetzt werden CD-ROMs zur Bereitstellung großer Datenmengen, z.B. von Softwarepaketen, Lexika, Wörterbüchern, Gesetzbüchern, Grafiksammlungen, Zeitschriftenjahrgängen, Versandkatalogen, Katalogen von Reiseanbietern usw. Da die CD-ROM Audio- und Bewegtbilddaten enthalten kann, eignet sie sich auch für Multimedia-Präsentationen.

Die *CD-R (Recordable)* entspricht im Aufbau der CD-ROM, ist jedoch einmal (ggf. in mehreren Sitzungen) beschreibbar. Sie hat die gleiche Datendarstellung wie die CD-ROM, d.h. eine Speicherung durch Pits und Lands. Basis hierfür ist eine auf einem reflektierenden Untergrund aufgebrachte organische Beschichtung (*dye*), die an den Stellen der Vertiefungen (*pits*) mittels des Laserstrahls lichtdurchlässig gemacht wird und somit ihr Reflexionsverhalten verändert. Die Datenaufzeichnung muß nicht innerhalb eines einzigen Durchgangs erfolgen, sondern kann auf mehrere Sitzungen ausgedehnt werden. Allerdings sind bereits beschriebene Bereiche nicht mehr löschar oder überschreibbar. Bei ebenfalls 120 mm Durchmesser des Mediums beträgt die Speicherkapazität 700, 800 oder 870 Mbyte. Die Übertragungsrate eines CD-R-Laufwerks hängt wie beim CD-ROM-Laufwerk davon ab, mit welcher wievelfachen Geschwindigkeit das Gerät arbeitet. Üblich ist derzeit eine bis zu 52fache Geschwindigkeit für das Schreiben und Lesen (7,8 Mbyte/s). – Einmalbeschreibbare CDs eignen sich für die Archivierung großer Datenmengen, z.B. großer Textdateien. Eingesetzt werden sie insbesondere aber auch für das Speichern von Musik und von Fotos.

Die *CD-RW (ReWritable)* erfordert als wiederbeschreibbare CD eine andere Aufzeichnungstechnik als die der CD-R. Verwendet wird hier das sog. *Phase-Change-Verfahren (PC)*, ein ebenfalls rein optisches Verfahren. Es basiert auf einer Beschichtung der Scheibe, die sich beim Schreibvorgang durch unterschiedlich hohe Erhitzung mittels des Laserstrahls in einen unstrukturierten (amorphen) oder einen kristallinen Zustand bringen lässt. Beim Lesevorgang reflektieren diese beiden Materialzustände dann den Laserstrahl unterschiedlich gut. Die CD-RW hat ebenfalls eine Speicherkapazität von 700 Mbyte. Aufgrund des gegenüber der CD-ROM und der CD-R anderen Reflexionsverhaltens des Datenträgers sind CD-RW-Scheiben auf älteren CD-ROM-Geräten nicht unbedingt lesbar. Üblich sind derzeit bis zu 24fache Geschwindigkeiten für das Schreiben und bis zu 52fache Geschwindigkeiten für das Lesen (3,6 bzw. 7,8 Mbyte/s). Mit CD-RW-Laufwerken können auch CD-R-Medien beschrieben und gelesen werden. – Der Einsatz von CD-RWs entspricht dem der CD-R. Insbesondere werden sie aber auch als Backup-Medien eingesetzt, d.h. für die Sicherung von Dateien, die in gewissen Zeitabständen aktualisiert werden. Gegenüber eines Backups mittels Streamer haben sie den Vorteil des wahlfreien Zugriffs auf die Dateien (wie bei Magnetplattenspeichern). Da sich die beim Phase-Change-Verfahren verwendeten Sub-

stanzen durch die Brennvorgänge verändern, wird die Wiederbeschreibbarkeit der RW-Medien mit ca. 1000mal angegeben.

Digital Versatile Disk – DVD. Ziel der Entwicklung der DVD war es, ein digitales Speichermedium mit genügend großer Kapazität zu schaffen, das es erlaubt, z.B. einen Spielfilm von 90 oder 120 Minuten Dauer zu speichern. Damit sollte einerseits die analog speichernde Videokassette (VHS) abgelöst und andererseits die Einschränkung, wie sie durch die zu geringe Speicherkapazität von 650 Mbyte der CD-ROM gegeben war, aufgehoben werden. Ansätze hierzu gab es schon im CD-ROM-Bereich, u.a. mit der sog. Super Density Disk (SD), einer CD, bei der durch Erhöhung der Schreidichte eine Speicherkapazität von 5 Gbyte erreicht wurde. Die DVD arbeitet nach demselben Prinzip. Bei ihr wurde gegenüber der CD-ROM die Mindestlänge der Pit-Bereiche wie auch der Abstand zwischen den Spuren verringert, womit bei gleichem Durchmesser wie bei der CD eine Kapazität von 4,7 Gbyte, d.h. ungefähr ihre 7fache Kapazität erreicht wird. Ermöglicht wird das dadurch, daß die Trägerschicht näher an der Plattenoberfläche liegt, so daß der Laserstrahl stärker fokussiert werden kann. Die Verringerung des Abstands erlaubt es auch, entweder beide Seiten der Platte zu nutzen (unter Beibehaltung der Plattenhöhe von CDs) oder die Information in zwei unterschiedlich tiefen Schichten unterzubringen. Durch Kombination beider Möglichkeiten werden heute Speicherkapazitäten von bis zu 17 Gbyte erreicht.

Video- und Audiodaten werden komprimiert gespeichert, wobei für die Videodaten das *MPEG2-Verfahren*, der heutige digitale Standard zur Codierung von Fernsehbildern, verwendet wird. Unterschiede gibt es hier in der Aufzeichnungsqualität, indem unterschiedliche Auflösungen und Komprimierungsgrade wählbar sind. Gegenüber einer VHS-Aufzeichnung zeichnet sich die DVD-Technik neben dem schnellen (quasi wahlfreien) Zugriff durch eine sehr viel höhere Qualität hinsichtlich der Bildschärfe, der Farbstabilität und der Tonqualität (mehrspuriger Kinoton) aus. Genutzt wird das Medium allerdings nicht nur für Filmaufzeichnungen, sondern auch für die Datenspeicherung. Die Übertragungsraten für DVD-Zugriffe werden wie bei den CD-Laufwerken in Vielfachen der einfachen *Übertragungsgeschwindigkeit* angegeben. Sie beträgt hier jedoch nicht CD-typisch 150 kbyte/s, sondern DVD-typisch 1,385 Mbyte/s. Das Abspielen von Filmen erfolgt immer mit einfacher Geschwindigkeit, das von Daten üblicherweise mit höherer Geschwindigkeit.

Wie eingangs 8.3.5 erwähnt, werden DVDs wie CDs in nur lesbare, einmal beschreibbare und wiederholt beschreibbare Medien unterschieden. Für die nur lesbare, d.h. vom Hersteller beschriebene DVD ist das Format durch einen einheitlichen, vom sog. DVD-Forum verfaßten Standard festgelegt. Für die einmalbeschreibbaren und wiederbeschreibbare Medien gibt es einen solchen einheitlichen Standard nicht, weshalb verschiedene Firmen und Firmenkonsortien unterschiedliche Techniken auf den Markt gebracht haben, die entweder nicht oder nur eingeschränkt zueinander und zu den CD-Techniken kompatibel sind.

Die nur lesbare DVD (*DVD-Video, DVD-ROM*) gibt es mit vier unterschiedlichen Kapazitäten, basierend auf einer oder zwei Datenschichten und auf ein- oder zweiseitiger Nutzung, nämlich mit 4,7 Gbyte (DVD-5: einschichtig, einseitig), 8,5 Gbyte (DVD-9: zweischichtig, einseitig), 9,4 Gbyte (DVD-10: einschichtig, zweiseitig) und 17 Gbyte (DVD-18: zweischichtig, zweiseitig). Gespeichert werden können auf ihnen Video-Filme mit Spieldauern von 133, 241, 266 und 482 Minuten Dauer (MPEG2-Format) oder entsprechende Datenmengen. Da die zur Video-Wiedergabe bzw. zum Lesen der Daten benutzten DVD-Laufwerke auch für das Lesen von CDs geeignet sind, werden für diese Geräte zwei unterschiedliche Geschwindigkeiten angegeben: Derzeit schnelle Geräte weisen eine bis zu 16fache Lesegeschwindigkeit für DVDs (22,16 Mbyte/s) und eine bis zu 48fache Lesegeschwindigkeit für CDs auf (7,2 Mbyte/s).

Bei den einmalbeschreibbaren und den wiederbeschreibbaren DVDs unterscheidet man die sog. Minus-Medien, *DVD-R* und *DVD-RW*, die sog. Plus-Medien, *DVD+R* und *DVD+RW* sowie die *DVD-RAM*. Die einmalbeschreibbaren Medien (-R, +R) nutzen zur Datenspeicherung wie die CD-R eine organische Beschichtung mit darunterliegender Reflexionsschicht, die wiederbeschreibbaren Medien (-RW, +RW, RAM) das von der CD-RW her bekannte Phase-Change-Verfahren mit der damit verbundenen Einschränkung, daß die Wiederbeschreibbarkeit auf ca. 1000 Brennvorgänge begrenzt ist.

Für die gemischte Nutzung von Minus- und Plus-Medien gibt es sog. Kombilaufwerke, mit denen alle vier Medien-Varianten sowie CD-R- und CD-RW-Medien beschrieben und gelesen werden können. Hier liegen die Geschwindigkeitsbegrenzungen derzeit bei 8facher Schreibgeschwindigkeit für -R und +R (11,08 Mbyte/s), 4facher Schreibgeschwindigkeit für -RW und +RW (5,54 Mbyte/s) und 12facher Lesegeschwindigkeit (16,62 Mbyte/s) für alle DVD-Varianten, einschließlich der nur lesbaren DVD. Als Speicherkapazitäten dieser Medien sind 4,7 Gbyte gebräuchlich. – DVD-RAM-Laufwerke ermöglichen derzeit 1fache Schreibgeschwindigkeit (1,385 Mbyte/s) und 2fache Lesegeschwindigkeit (2,77 Mbyte/s) bei Medien mit Speicherkapazitäten von 2,6 oder 4,7 Gbyte (einseitig) und 5,2 oder 9,4 Gbyte (zweiseitig). Schreibkompatibel sind diese Geräte mit DVD-R, lesekompatibel mit DVD-ROM/-R und CD-ROM/-R/-RW.

DVD-Nachfolger. Wie oben erwähnt, werden die CD und insbesondere die DVD auch für die Datenarchivierung genutzt. Hier stellt sich das Problem, daß die Festplatten schnell wachsende Speicherkapazitäten aufweisen und man dementsprechend auch die Kapazitäten optischer Plattenspeicher erhöhen möchte. Ein weiterer Bedarf für höhere Speicherkapazitäten liegt darin, Video-Filme im hochauflösenden HDTV-Format zu speichern. Gegenüber Video-Filmen mit herkömmlicher Auflösung, die üblicherweise auf DVD-9-Medien gespeichert werden, ist hierfür eine ca. dreifache Speicherkapazität erforderlich. Vor diesem Hintergrund sind als DVD-Nachfolger derzeit zwei konkurrierende Systeme in der Entwicklung: die *Advanced-Optical-Disc (AOD)*, die einseitig beschreibbar und

einschichtig 20 Gbyte und zweischichtig 40 Gbyte speichern können soll, und die *Blu-ray-Disc* mit einseitig und einschichtig 27 Gbyte und zweischichtig 54 Gbyte. Bei beiden werden als Durchmesser die 12 cm der CD und DVD beibehalten. – Bereits verfügbar als DVD-ablösende Systeme sind die sog. *Professional-Disc-for-Data (PDD)* und die *Ultra-Density-Optical (UDO)*, beide auf dem Phase-Change-Verfahren basierend, mit Nutzdatenmengen pro Seite von 20,5 bzw. 14 Gbyte und mit Durchmessern von 13 bzw. 12 cm.

8.3.6 Magnetooptische Plattenspeicher

Magnetooptische Speicher (magneto-optical Disk, *MO* oder *MOD*) sind wiederbeschreibbare Speicher in Form einer in einer festen Hülle untergebrachten starren Platte, die wie eine Wechselplatte ausgewechselt werden kann. Bei ihnen wird zum Schreiben der sog. *thermo-magnetische Effekt* genutzt, wonach eine magnetisierbare Schicht bei Überschreiten einer bestimmten Temperatur ihre Ausrichtung verliert. Die hierfür erforderliche punktuelle Erhitzung erfolgt durch einen Laserstrahl, der bei MO-Disks dazu genutzt wird, die Magnetschicht durch einen auf der Gegenseite der Platte aktivierten Magneten auszurichten. Beim Lesen der MO wird der *Kerr-Effekt* genutzt, wonach polarisiertes Licht unter Einfluß eines Magnetfeldes in seiner Polarisationsebene gedreht wird.

Diese Technik verleiht dem MO-Speichermedium eine höhere Datensicherheit als allen anderen Speichermedien. So läßt sich eine MO-Disk bis zu 100 Millionen mal wiederbeschreiben, und es wird ihr eine Lebensdauer von mehr als 60 Jahren zugesprochen. Nachteilig ist jedoch, daß das Schreiben gegenüber z.B. DVD-Medien relativ langsam ist, da eine bereits beschriebene MO-Disk nicht direkt überschrieben werden kann, sondern der zu überschreibende Bereich zuerst gelöscht werden muß. Das heißt, es ist zunächst ein Löschdurchgang erforderlich, bei dem die magnetische Beschichtung einheitlich ausgerichtet wird. Darauf folgt dann der eigentliche Schreibvorgang, bei dem einzelne Bitpositionen anders ausgerichtet werden. Gegebenenfalls wird noch ein Prüfdurchgang durchgeführt. Das Lesen hingegen ist in nur einem Durchgang möglich und damit wesentlich schneller als das Schreiben. Jedoch erlaubt eine andere Technik, das sog. *LIMDOW-Verfahren* (light intensity modulation - direct overwrite), auch das direkte Überschreiben der Daten, also das Schreiben in nur einem Durchgang. Möglich wird das durch ein mehrlagiges Speichermedium sowie durch ein Laufwerk, das die Daten mit zwei unterschiedlichen Laserstärken in eben nur einem Durchgang schreibt. Je nach Laufwerk werden beide Techniken unterstützt, ggf. also auch LIMDOW-Medien (kurz *DOW-Medien*). – Als MO-Variante gibt es die *WORM*-Laufwerke (write once read many), bei denen sich das MO-Medium nur einmal beschreiben, aber beliebig oft lesen läßt. Sie werden dort eingesetzt, wo man eine nachträgliche Datenmanipulation ausschließen möchte.

MO-Medien müssen vor ihrer Benutzung formatiert werden. Dies geschieht ähnlich den Magnetplattenspeichern in konzentrischen Spuren mit Unterteilung der Spuren in Sektoren. Die Speicherkapazitäten von MO-Medien betragen bei einem Formfaktor von 3,5 Zoll 128, 230, 540 und 640 Mbyte sowie 1,3 und 2,6 Gbyte; die Laufwerke sind üblicherweise abwärtskompatibel. Bei Geräten mit 5,25 Zoll Formfaktor gibt es Speicherkapazitäten von bis zu 5,2 und 9,1 Gbyte.

8.3.7 Magnetbandspeicher

Magnetbandspeicher sind Hintergrundspeicher mit großen Speicherkapazitäten bei geringen Kosten. Sie haben als Speichermedium ein flexibles Kunststoffband mit magnetisierbarer Beschichtung, das zum Beschreiben und Lesen an einem Schreib-/Lesekopf mit (oder bei manchen Geräten ohne) Kopfberührungsleitung vorbeigezogen wird. Das Band befindet sich dabei auf einer Abwickelspule und wird von einer Aufwickelspule aufgenommen. Grundsätzlich unterscheidet man hier die früher eingesetzten *Langbandgeräte* mit offenen Spulen und die seit langem gebräuchlichen *Streamer* mit Magnetbandkassette (cartridge).

Langbandgeräte. Langbandgeräte sind die ursprünglichen Magnetbandgeräte, wie sie in früherer Zeit bei Rechnern als Hintergrundspeicher zum Einsatz kamen. Ihre Technik ist inzwischen veraltet und ihre Leistungsfähigkeit heutigen Anforderungen nicht mehr gewachsen (Speicherkapazität von z.B. 200 Mbyte), sie werden aber ggf. noch zum Lesen alter Bänder benötigt. Das Band hat bei ihnen eine Breite von 0,5 Zoll und wird parallel in z.B. 9 übereinanderliegenden Längsspuren (8 Datenbits, 1 Paritätsbit, Bild 8-37a), d.h. byteweise beschrieben. Die Datenspeicherung erfolgt in Blöcken mit variabler oder fester Anzahl an

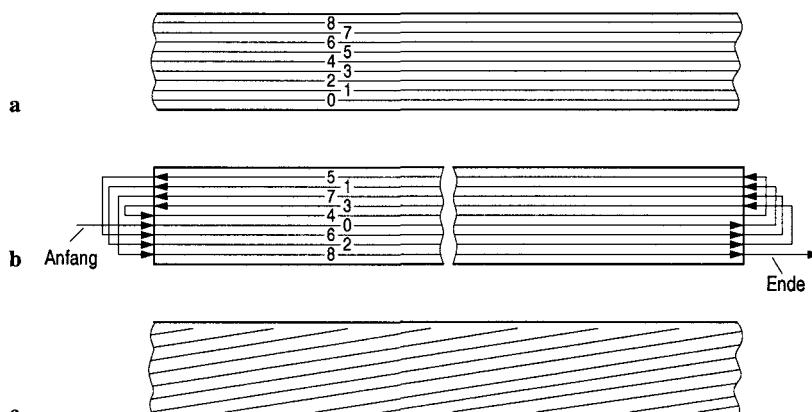


Bild 8-37. Datenspeicherung auf Magnetband. a Langbandgerät mit z.B. 9 parallelen Spuren, b QIC-Streamer mit z.B. 9 serpentinartigen Spuren (*linear recording*), c Video- oder DAT-Streamer mit Schrägspuren (*helical scan recording*)

Bytes (2 Kbyte bis 32 Kbyte) und mit dazwischenliegenden Lücken, sog. Gaps. Typisch dabei ist, daß beim Schreiben und Lesen aufeinanderfolgender Blöcke das Band an jedem Blockende gestoppt und anschließend neu gestartet wird (*Start-Stopp-Verfahren*). Da das Band vom Stillstand aus und wegen möglicher Vorwärts- und Rückwärtsbewegung von der Gap-Mitte aus gestartet werden muß, sind dazu große Gaps erforderlich, wodurch vor allem bei kleinen Blöcken ein verhältnismäßig großer Anteil der möglichen Bandkapazität verlorengeht. Das Start-Stopp-Verfahren erfordert außerdem aufgrund seiner hohen Bandbeschleunigungen und -verzögerungen eine aufwendige Technik zur Entlastung des Bandzuges (Bildung von Pufferschläufen mittels Pneumatik oder Pendelarmen), die sich nachteilig auf die Gerätekosten und die Gerätebaugröße auswirkt. Darüber hinaus wird durch das Stoppen und Starten die mittlere Übertragungsgeschwindigkeit gegenüber der eigentlichen Übertragungsrate eines Blocks erheblich reduziert.

Streamer (Kassettengeräte). Streamer als die heute gängigen Magnetbandspeicher werden insbesondere für die kurzfristige Datensicherung, d.h. für Backups, aber auch für die langfristige Datenarchivierung (z.B. in Bandbibliotheken) und für den Datenträgeraustausch eingesetzt. Ihr Vorteil liegt in den geringen Speicherkosten und in der einfachen Handhabung der Kassetten. Nachteilig ist zunächst die durch die Berührung des Schreib-/Lesekopfs mit dem Band bedingte Bandabnutzung, die bei häufiger Bandbenutzung ggf. zum Datenverlust führen kann. Nachteilig sind außerdem die mit dem sequentiellen Zugriff verbundenen großen mittleren Zugriffszeiten. Dennoch eignen sie sich aufgrund ihrer Speicherkapazitäten von bis zu 500 Gbyte und mehr insbesondere dazu, das Sichern des gesamten Inhalts einer Festplatte durchzuführen. Dieser ggf. mehrstündige Vorgang kann ohne Bandwechsel und damit automatisiert, z.B. nachts erfolgen. Bei einem teilweisen Sichern besteht aufgrund des sequentiellen Speicherns der Nachteil, daß eine bereits gespeicherte Datei, wenn sie nicht als zuletzt gespeicherte auf dem Band steht, nicht einfach durch ihre aktualisierte Version überschrieben werden kann, sondern die aktualisierte Version hinten angefügt werden muß. Die Übersicht über den Bandinhalt erhält man dabei durch ein Dateienverzeichnis, das mit auf dem Band gespeichert ist.

Speichermedium. Die Kassette als Bandspeichermedium hat die Vorteile, daß das Band vor unbeabsichtigter Berührung geschützt ist und daß die Handhabung des Bandes, z.B. das Einlegen in das Laufwerk, sehr einfach ist. Außerdem hat die *Streamerkassette* gegenüber der herkömmlichen offenen Magnetbandspule wesentlich kleinere Abmessungen. Bezuglich der Mechanik des Ab- und Aufwickelns des Bandes lassen sich im Prinzip zwei Bauformen unterscheiden:

1. mit Aufwickel- und Abwickelpule innerhalb des Kassettengehäuses (Bild 8-38), wie von Audio- und Videokassetten bekannt, und
2. mit nur der Abwickelpule innerhalb des Gehäuses und dementsprechend einer Verlagerung der Aufwickelpule in das Laufwerk (Bild 8-39). Letztere

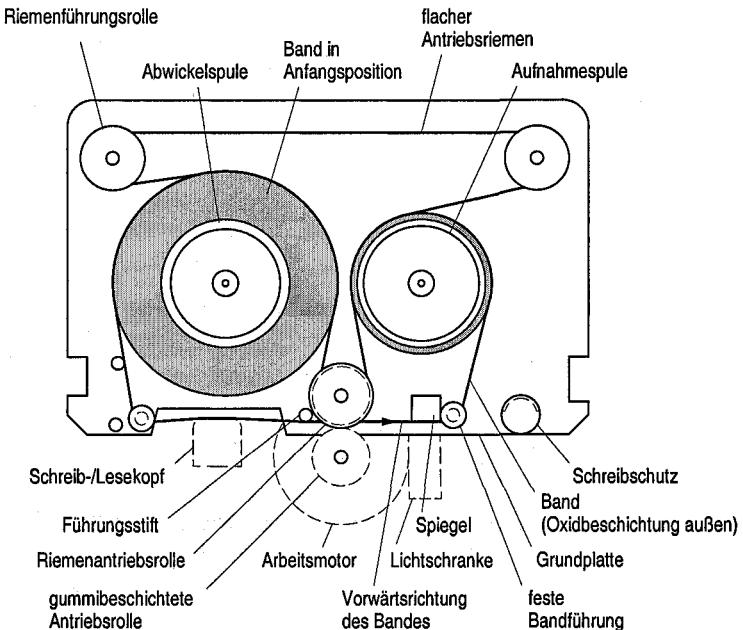


Bild 8-38. Aufbau einer QIC- oder Travan-Kassette (nach [Glass 1990])

Technik bietet mehr Platz im Kassettengehäuse und ermöglicht dadurch größere Bandlängen, d.h. Speicherkapazitäten. Die Spulen bei Kassetten sind auf den Spulenkerne (Rolle) reduziert.

Datenaufzeichnung. Die Datenaufzeichnung erfolgt wie bei den Langbandgeräten in Blöcken variabler oder fester Länge (von z.B. 512 Bytes). Im Gegensatz zu diesen erfolgt das Schreiben und Lesen von Dateien jedoch bitseriell und mit einem kontinuierlichen Datenstrom (stream), wobei das Band zwischen den Blöcken nicht angehalten wird, was zu der Gerätebezeichnung *Streamer* führte. Die Kontinuität des Datenstroms erlaubt eine wesentlich einfache und kostengünstigere Bandführungsmechanik als bei den Langbandgeräten. Um sie aufrecht erhalten zu können, sind die Laufwerke mit Pufferspeichern von ggf. mehreren Mbyte ausgestattet, in die eine Vielzahl an Blöcken aufgenommen werden kann. Reißt der Datenstrom dennoch zwischenzeitlich ab, so muß das Band gestoppt und für das erneute Schreiben zurückgesetzt (repositioniert) werden. Daß heißt, neben dem für Kassettengeräte typischen *Stream-Modus* gibt es auch einen *Start-Stopp-Modus*. (Einige Geräte können sich an einen „verlangsamten“ Datenstrom durch Verringern ihrer Aufzeichnungsgeschwindigkeit anpassen, womit das Repositionieren entfällt.)

Bild 8-40 zeigt den Vorgang des *Repositionierens* (backhitching) am Beispiel des Schreibens, unter der Annahme, daß der Datenblock N+5 vom Prozessor oder DMA-Controller nicht rechtzeitig im Pufferspeicher des Streamers bereitgestellt

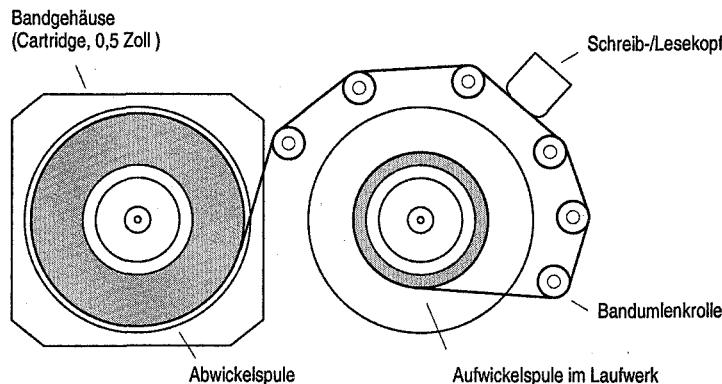


Bild 8-39. Mechanik eines DLT-Streamers

wurde. Der Streamer erzeugt in diesem Fall eine größere Lücke als sonst und bremst zum Zeitpunkt A das Band bis zum Stillstand in Punkt B ab. Sobald der Puffer dann mit dem Block N+5 gefüllt ist, wird das Band in Rückwärtsrichtung bewegt. Dabei läuft es mit den zuletzt geschriebenen Blöcken am Schreib-/Lesekopf vorbei und wird am Punkt E wieder gestoppt. Anschließend wird es in Vorfahrtsrichtung beschleunigt. Nach Erreichen der regulären Bandgeschwindigkeit liest das Laufwerk dabei ab Punkt F die bereits geschriebenen Blöcke, um den erweiterten Gap-Bereich zu lokalisieren und an diesen unmittelbar anschließend die Blöcke N+5 und folgende zu schreiben.

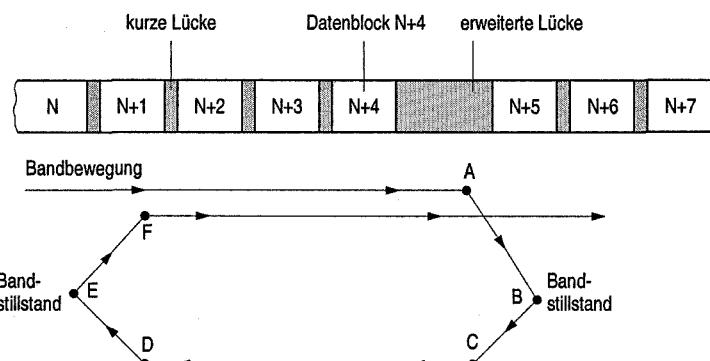


Bild 8-40. Repositionieren des Bandes (nach [Grundy 1984])

Das Repositionieren des Bandes ist zwar sehr viel zeitaufwendiger als das Stoppen und Starten bei Langbandgeräten, weshalb der Datenstrom nach Möglichkeit nicht unterbrochen werden sollte. Es macht jedoch die bei Langbandgeräten notwendige Mechanik zur Entlastung des Bandzuges überflüssig. Daraus ergibt sich die kleine Baugröße von Streamern und eine wesentlich bessere Nutzung des Bandes aufgrund der sehr viel kürzeren Gaps.

Magnetbandbreite und Aufzeichnungsverfahren. Streamer gibt es in unterschiedlichen Ausführungen, u.a. mit unterschiedlicher Magnetbandbreite. Sie hat Einfluß auf die maximal mögliche Speicherkapazität einer Kassette und auf die Kosten für Geräte und Kassetten. Damit verbunden ist aber auch eine Inkompatibilität zwischen den verschiedenen Streamer-Ausführungen. Im Desktop-Bereich sind schmale Bänder mit 0,25 Zoll und 4 mm Breite vorherrschend. Für größere Datenaufkommen, z.B. bei Servern in Rechnernetzen, werden Bänder mit 0,5 Zoll und 8 mm Breite eingesetzt.

Ein weiteres Unterscheidungsmerkmal für Streamer ist die verwendete Aufzeichnungstechnik: Längsspur- oder Schrägspurverfahren. Beim *Längsspurverfahren* (*linear recording*) werden die Bits in horizontal zum Band verlaufenden Einzelspuren aufgezeichnet. Je nach Technik ist das Band dazu in bis zu 200 und mehr Spuren unterteilt, die bei der Datenaufzeichnung serpentinenartig durchlaufen werden. Das heißt, es wird zunächst eine Spur in Vorwärtsrichtung, dann eine Spur in Rückwärtsrichtung, dann wieder eine Spur in Vorwärtsrichtung usw. durchlaufen, wobei der Schreib-/Lesekopf jeweils auf die betreffende Spur positioniert wird (Bild 8-37b, S. 614). Unterstützt wird die Positionierung ggf. durch zusätzliche Servospuren.

Beim *Schrägspurverfahren* (*helical scan recording*) erfolgt die bitserielle Aufzeichnung wie bei Audio-DAT-Geräten und analog aufzeichnenden kommerziellen Videorecordern in schräg zum Band verlaufenden, relativ kurzen Spuren (Bild 8-37c, S. 614). Das Magnetband wird dazu, wie in Bild 8-41 am Beispiel eines *Video-Streamers* gezeigt, um eine rotierende, schräg zum Band gestellte Trommel

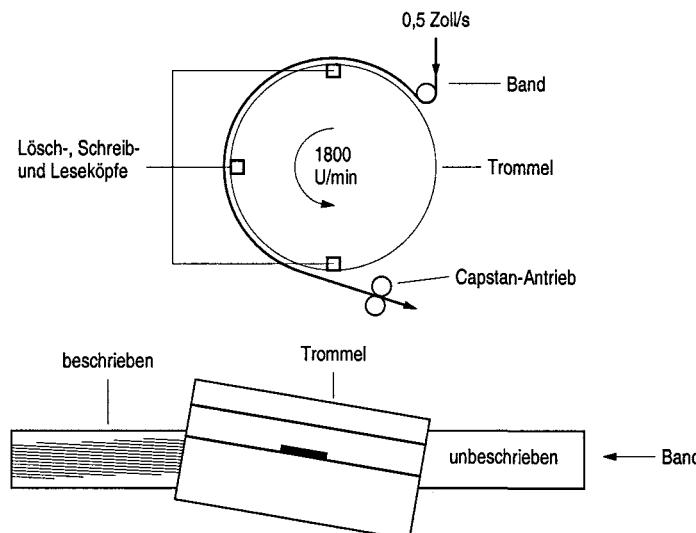


Bild 8-41. Schrägspralaufzeichnung bei Video-Streamern mit einer Relativgeschwindigkeit von 5,6 m/s zwischen Band und Trommel (schematisierte Darstellung nach [Exabyte 1987])

geführt, in der die Schreib- und Leseköpfe in zueinander versetzter Position untergebracht sind. Ein zusätzlicher Servokopf erzeugt Information zur Spurfindung (automatic track finding, ATF). Die durch die Vorwärtsbewegung des Bandes und das Rotieren der Trommel erzeugten parallel zueinander liegenden Spuren verlaufen hier unter einem Winkel von ca. 5 Grad von einem Bandrand zum andern, wobei die Spurlänge annähernd dem Zehnfachen der Breite des Bandes ist. Jede Spur enthält mehrere Blöcke an Nutzdaten, Blockadressen, Sicherungsbytes sowie Servoinformation.

Gerätetypen. Magnetbandbreite, Aufzeichnungstechnik und Kassettenaufbau (eine oder zwei Wickelspulen) haben in ihrer Kombination und in zusätzlichen technischen Varianten (z.B. gleichzeitiger Zugriff auf mehrere Spuren) zu einer Vielzahl an Gerätetypen geführt. Tabelle 8-5 zeigt dazu eine Auswahl, geordnet nach den maximal möglichen Speicherkapazitäten bei unkomprimierter Speicherung (*native*). Bei vielen dieser Geräte lassen sich die Speicherkapazität und die Aufzeichnungsrate durch Datenkompression verdoppeln oder sogar verdreifachen (*compressed*). Neben den Kassetten mit maximaler Speicherkapazität stehen üblicherweise auch Kassetten mit geringeren Kapazitäten zur Verfügung. – Die als Gerätbezeichnungen benutzten Akronyme bedeuten: QIC quarter inch cartridge, DAT/DDS digital audio tape/digital data storage, DLT digital linear tape, SLR scalable linear recording, AIT advanced intelligent tape, ADR advanced digital recording, LTO linear tape open.

Tabelle 8-5. Speicherkapazitäten und Übertragungsraten von Streamern bei unkomprimierter Speicherung. Linear steht für Längsspur- und helical für Schrägspurverfahren

Streamertyp	Breite des Bandes	Aufzeichnung	Speicherkapazität Gbyte	Übertragungsraten Mbyte/s
QIC-5010-DC	0,25 Zoll	linear	13	1,5
Travan 40	8 mm	linear	20	2
DAT DDS-5	4 mm	helical	36	3
DLT8000	0,5 Zoll	linear	40	6
SLR 100	0,25 Zoll	linear	50	5
Mammoth-2	8 mm	helical	60	12
VXA-2	8 mm	helical	80	6
AIT-3	8 mm	helical	100	12
ADR2.240	8 mm	linear	120	8
Super DLT 320	0,5 Zoll	linear	160	16
LTO 2 200	0,5 Zoll	linear	200	35
S-AIT	0,5 Zoll	helical	500	30

Bei Desktop-Streamern folgen die Gerätegrößen meist dem Formfaktor 3,5 Zoll, bei Server-Streamern dem Formfaktor 5,25 Zoll.

Literatur

Allgemeine Literatur

- Bähring, H.: Mikrorechner-Technik; Band I, Band II. 3. Aufl. Berlin: Springer 2002
- Flynn, M.J.: Computer Architecture – pipelined and parallel processor design. Boston: Jones and Bartlett 1995
- Liebig, H.: Rechnerorganisation. 3. Aufl. Berlin: Springer 2003
- Patterson, D.A.; Hennessy, J.L.: Computer organization and design – the hardware/software interface. San Mateo: Morgan Kaufmann 1994
- Rechenberg, P.; Pomberger, G. (Hrsg.): Informatik-Handbuch – Technische Informatik. 3. Aufl. München: Hanser 2002
- Stallings, W.: Computer organization and architecture – designing for performance. 4th ed. Upper Saddle River: Prentice-Hall 1996
- Ungerer, T.: Mikroprozessortechnik. Bonn: Thomson 1995
- Völz, H.: Informationsspeicher. Renningen-Malmsheim: expert 1996

Spezielle Literatur

Kapitel 1

- Barron, D.W. (1970): Assembler und Lader. München: Hanser
- Bohn, W.F.; Flik, Th. (2002): Daten – Zeichen- und Zahlendarstellungen. In: Rechenberg, P.; Pomberger, G. (Hrsg.): Informatik-Handbuch. 3. Aufl. München: Hanser
- Goldberg, D. (1990): Computer arithmetic. In: Hennessy, J.L.; Patterson, D.A.: Computer architecture – a quantitative approach. San Mateo: Morgan Kaufmann
- Goldberg, D. (1991): What every computer scientist should know about floating point arithmetic. ACM Comp. Surv. 23, 1, 5-48
- Hoffmann, R. (1993): Rechnerentwurf: Rechenwerke, Mikroprogrammierung, RISC. 3. Aufl. München: Oldenbourg
- Klar, H. (1996): Integrierte Digitale Schaltungen MOS/BICMOS. 2. Aufl. Berlin: Springer
- Liebig, H.: Rechnerorganisation. 3. Aufl. Berlin: Springer 2003
- Mackenzie, C.E. (1980): Coded character sets, history and development. Reading: Addison-Wesley
- Mead, C.; Conway, L. (1980): Introduction to VLSI systems. 2nd printing. Reading: Addison-Wesley

Kapitel 2

- Advanced Micro Devices (1993): Am29000™ and Am29005™ user's manual and data sheet
- Bode, A. (2002): Technische Informatik – Rechnerarchitektur und Prozessoren. In: Rechenberg, P.; Pomberger, G. (Hrsg.): Informatik-Handbuch. 3. Aufl. München: Hanser
- Flynn, M.J. (1995): Computer Architecture – pipelined and parallel processor design. Boston: Jones and Bartlett
- Halfhill, T.R. (2002): IBM trims Power4, adds AltiVec. Microprocessor Report (www)
- Hinton, G., Sager, D. et al. (2001): The microarchitecture of the Pentium® 4 processor. Intel Technology Journal Q1 (www)
- Intel (1993): Pentium™ processor user's manual, vol. 1: Pentium processor data book
- Intel (1999): IA-64 application developer's architecture guide
- Motorola (1982): MC68000 16-bit microprocessor user's manual. 3rd ed.
- Motorola (1993): PowerPC™ 601 – RISC microprocessor user's manual
- Pabst, T. (2000): Intel's new Pentium 4. tom's hardware guide (www)
- Stallings, W. (1996): Computer organization and architecture – designing for performance. 4th ed. Upper Saddle River: Prentice-Hall
- Stiller, A. (2003): Die Architekturen der 64-Bit-Prozessoren. c't, H.20, 112-117
- Tanenbaum, A.S. (2002): Moderne Betriebssysteme. 2. Aufl. München: Pearson/Prentice Hall
- Ungerer, T. (2001): Mikroprozessoren – Stand der Technik und Forschungstrends. Informatik Spektrum 24, H.1, 3-15
- Ungerer, T. (1995): Mikroprozessortechnik. Bonn: Thomson

Kapitel 3

- Knuth, D.E. (1961): The art of computer programming. Vol. 1. Reading: Addison-Wesley
- Nassi, I.; Schneidermann, B. (1973): Flowchart techniques for structured programming. SIGPLAN Notices 8, H.8, 12-26
- Schnupp, P.; Floyd, C. (1979): Software, Programmierung und Projektorganisation. 2. Aufl. Berlin: de Gruyter

Kapitel 4

- Advanced Micro Devices (1993): Am29000™ and Am29005™ user's manual and data sheet
- Aho, A.V.; Sethi, R.; Ullmann, J.G. (1999): Compilerbau. (Teil 1 und Teil 2). jew.eils 2. Aufl. München: Oldenbourg
- Bengel, G. (1990): Betriebssysteme. Heidelberg: Hüthig
- Borland (1990): Turbo C++ – Programmierhandbuch. München: Borland
- Dumschat, U. (2001): Pinguin mit Turbolader. Electronic. Februarheft, 56-61
- Kernighan, B.W; Ritchie, D.M. (1990): Programmieren in C. 2. Aufl. München: Hanser
- Microware [1987]: OS-9 / 68000 – Operating system technical manual, Rev. H. Microware Systems

- Lindholm, T.; Yellin, F. (1997): Java™ – Die Spezifikation der virtuellen Maschine. Bonn: Addison-Wesley
- Rubini, A. (1998): Linux device drivers. Cambridge: O'Reilly
- Tanenbaum, A.S. (2002): Moderne Betriebssysteme. 2. Aufl. München: Pearson/Prentice Hall

Kapitel 5

- Compaq (2000): PCI-X, architecture overview (www)
- IBM (2003): PowerPC microprocessor family: programming environments manual for 64 and 32-bit microprocessors
- Intel (1984): Multibus II, Bus architecture specification handbook
- Intel (1989): i486™ Microprocessor
- Intel (1993): Pentium™ processor user's manual, vol. 1: Pentium processor data book
- Motorola (1985): VMEbus, Specification manual. revision C
- Motorola (1989): MC68020 32-bit microprocessor user's manual. 3rd ed.
- Motorola (1993): PowerPC™ 601 – RISC microprocessor user's manual
- PCI (1995): PCI local bus specification, rev. 2.1. Portland: PCI Special Interest Group
- PCI-X (1999): PCI-X addendum to the PCI local bus specification, rev. 1.0. Hillsboro: PCI Special Interest Group
- Shanley, T.; Anderson, D. (1995): PCI system architecture, 3rd ed. Reading: Addison-Wesley
- Stallings, W. (1996): Computer organization and architecture – designing for performance. 4th ed. Upper Saddle River: Prentice-Hall
- Widmer, A.X.; Franaszek, P.A. (1983): A DC-balanced, partitioned-block, 8b/10b transmission code. IBM Journal of Research an Development 27, H.5, 440

Kapitel 6

- Archibald, J; Baer J.-L. (1986): Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM Transactions on Computer Systems 4, H.4, 273-298
- Gilo, W. K. (1993): Rechnerarchitektur. 2. Aufl. Berlin: Springer
- Hewlett Packard (2002): Inside the Intel® Itanium® 2 Processor. Technical White Paper (www)
- Intel (1987): 80386 Hardware reference manual
- Intel (1996a): Pentium® Pro family developer's manual, vol. 1: Specifications
- Intel (1996b): Pentium® Pro family developer's manual, vol. 3: Operating system writer's manual
- Klar, H. (1996): Integrierte Schaltungen MOS/BICMOS. 2. Aufl. Berlin: Springer
- Liebig, H.: Rechnerorganisation. 3. Aufl. Berlin: Springer 2003
- Motorola (1989): MC68040 32-Bit microprocessor user's manual
- Motorola (1991): MC88110 Second generation RISC microprocessor user's manual
- Motorola (1993): PowerPC™ 601 – RISC microprocessor user's manual
- Motorola (1994): MC68060 user's manual
- Motorola (2003): MPC7450 RISC microprocessor family user's manual. Rev. 3
- Przybylski, S. A. (1990): Cache and memory hierarchy design. San Mateo: Morgan Kaufmann

- Rhein, D.; Freitag, H. (1992): Mikroelektronische Speicher. Wien: Springer
- Ross (1993): SPARC RISC user's guide – hyperSPARC edition, 3rd ed. Austin: Ross Technology
- Sharma, A.K. (1997): Semiconductor memories – technology, testing and reliability. Piscataway: IEEE Press
- Tanenbaum, A.S. (2002): Moderne Betriebssysteme. 2. Aufl. München: Pearson/Prentice Hall
- Van Loo, W. (1987): Maximize performance by choosing best memory. Computer Design 26, H.14, 89-94

Kapitel 7

- Dembowski, K. (1997): Computerschnittstellen und Bussysteme. 2. Aufl. Heidelberg: Hüthig
- Hamming, R.W. (1987): Information und Codierung. Englewood Cliffs: Prentice-Hall
- Hegenbarth, M. (1981): Stand der Normung im CCITT, Ebenen 2-6. GI-Fachtagung Kommunikation in verteilten Systemen. Berlin: Springer
- IBM (1970): Binary synchronous communications. 3rd ed. IBM systems reference library
- IEEE 1284 (1994): IEEE Standard signaling method for a bidirectional parallel peripheral interface for personal computers
- Preuß, L.; Musa, H. (1993): Computer-Schnittstellen. 2. Aufl. München: Hanser
- Tanenbaum, A.S. (2003): Computernetzwerke. 4. Aufl. Pearson
- Tillmann, W. (1997): Internet-Schnellbahn für jedermann. c't, H.11, 324-330
- Weissberger, A.J. (1979): Orient your data-link protocols toward bits, though characters still count. Electronic Design 27, H.15, 86-92

Kapitel 8

- Anderson, D. (1999): FireWire system architecture. 2nd. ed. Reading: Addison-Wesley
- Anderson, D.; Dzatko, D. (2001): Universal Serial Bus system architecture. 2nd. ed. Boston: Addison-Wesley
- Bähring, H.: Mikrorechner-Technik; Band I, Band II. 3. Aufl. Berlin: Springer 2002
- Dembowski, K. (1997): Feuerdraht – Firewire und andere serielle Bussysteme. c't, H.2, 284-260
- Exabyte (1987): EXB-8200 8mm Cartridge tape subsystem – product specification. Boulder: Exabyte Corporation
- Glass, L.B. (1990): Daten einwickeln. Bandbackup-Systeme – jetzt für alle interessant. c't, H.8, 148-154
- Griffith, P. (2003): Maximizing server storage performance with PCI express™ and serial attached SCSI. InfoStor (www.infostor.com)
- Grundy, K.P. (1984): Streaming tape controller adopts SCSI interface. Electronic Design 32, 179-186
- IEEE 1394b™ (2002): IEEE standard for a high-performance serial bus – amendment 2
- Intel (1997): Enhanced server i/o performance with i960® RP/RD i/o processors
- Intel (2003): 31244 PCI-X to Serial ATA controller. Developer's manual
- Mueller, S. (1995): Hardware-Praxis. 4. Aufl. Bonn: Addison-Wesley
- Schmidt, F. (1998): SCSI-Bus und IDE-Schnittstelle. 3. Aufl. Bonn: Addison-Wesley

- Seagate (1988): SCSI Interface manual
- Serial ATA (2001): Serial ATA – High speed serialized AT Attachment. Rev. 1.0. Serial ATA Workgroup (www)
- Stiller, A. (2002): Schlanker, schöner, schneller – Serial ATA beim Aufwärmen. c't, H.16, 186-189
- Strass, H. (1994): Massenspeicher. Poing: Franzis
- Tietze, U.; Schenk, Ch. (1993): Halbleiterschaltungstechnik. 10. Aufl. Berlin: Springer
- Völz, H. (1996): Informationsspeicher. Renningen-Malmsheim: expert
- Widmer, X.A.; Franaszek, P.A. (1983): A DC-balanced, partitioned-blocked, 8b/10b transmission code. IBM Journal of Research and Development 27, H.5, 440

Akronyme

ADR Advanced Digital Recording	C Carry
ADSL Asymmetric DSL	CAM Content Addressable Memory
AGP Accelerated Graphics Port	CAV Constant Angular Velocity
AIT Advanced Intelligent Tape	CC Condition Code
ALU Arithmetic and Logical Unit	CCITT Comité Consultatif International Télégraphique et Téléphonique
AMD Advanced Micro Devices	cc-NUMA Cache Coherent Non-Uniform Memory Access
ANSI American National Standards Institute	CD Compact Disk
AOD Advanced Optical Disc	CD-R CD Recordable
ASCI Advanced Simulation and Computing Program (der US-Regierung)	CD-RW CD ReWriteable
ASCII American Standard Code for Infor- mation Interchange	CISC Complex Instruction Set Computer
AT Advanced Technology	CLV Constant Linear Velocity
ATA AT Attachment	CMP Chip Multiprocessing
ATAPI ATA Packet Interface	Codec Codierer/Decodierer
ATF Automatic Track Finding	COMA Cache Only Memory Architecture
ATM Asynchronous Transfer Mode	COP Character Oriented Protocol
BCC Block Checking Character	CPU Central Processing Unit
BCD Binary Coded Decimals	CSMA/CD Carrier Sense Multiple Access/ Collision Detection
BCS Block Checking Sequence	CRC Cyclic Redundancy Checking
BE Big-Endian	CWP Current-Window-Pointer
BEDO Burst Extended Data-Out	
BHR Branch History Register	DAT Digital Audio Tape
BHT Branch History Table	Datex Data Exchange
BIOS Basic Input-Output System	DCE Data Circuit-termination Equipment
BISYNC Binary Synchronous Communica- tions	DD Double Density
Bit Binary Digit	DDR Double Data Rate
BOP Bit Oriented Protocol	DDS Digital Data Storage
BPC Branch Prediction Cache	DEE Datenendeinrichtung
bps bit per second	DFÜ Datenfernübertragung
BSB Backside Bus	DIMM Dual In-line Memory Module
BSC Binary Synchronous Communications	DIN Deutsches Institut für Normung
BTAC Branch Target Address Cache	DLT Digital Linear Tape
BTB Branch Target Buffer	DMA Direct Memory Access
BTAC Branch Target Address Cache	DMAC DMA-Controller
	DOW Direct Overwrite

DRAM Dynamic RAM	HDLC High-level Data-Link Control
DRV Deutsche Referenzversion	HDSL High data rate DSL
DSL Digital Subscriber Line	HDTV High-Definition Television
DSM Distributed Shared Memory	HT Hyperthreading
DTE Data Terminal Equipment	HVD High Voltage Differential
DÜE Datenübertragungseinrichtung	
DVD Digital Versatile Disk	
EAROM Electrically Alterable ROM	IA-64 Intel Architecture 64-bit
EBCDI Extended Binary Coded Decimal Interchange Code	IBM International Business Machines
ECC Error Correction Code	IDE Integrated Drive Electronics
ECP Extended Capability	IDT Integrated Device Technology
ED Extra-high Density	IEC International Electrotechnical Commission
EDO Extended Data-Out	IEEE Institute of Electrical and Electronics Engineers
EDVAC Electronic Discrete Variable Automatic Computer	iLBX Intel Local Bus Extension
EEPROM Electrically Erasable ROM	ILP Instruction-Level Parallelism
EIA Electronic Industries Association	IMP Interface Message Processor
EIDE Enhanced IDE	Intel Intelligent/Integrated Electronics
EISA Extended ISA	iPSB Intel Parallel System Bus
EPIC Explicitly Parallel Instruction Computing	IRV Internationale Referenzversion
EPP Enhanced Parallel Port	ISA Industrie Standard Architecture
EPROM Erasable Programmable ROM	ISDN Integrated Services Digital Network
ESDI Enhanced Small Device Interface	ISO International Standard Organization
 	iSSB Intel Serial System Bus
FC Fibre Channel	ITU International Telecommunication Union
FCS Frame Checking Sequence	IU Integer Unit
FDC Floppy Disk Controller	
FDDI Fiber Distributed Data Interface	JBOD Just a Bunch of Disks
FIFO First-In First-Out	JIT Just-in-time
FM Frequency Modulation	JTAG Joint Test Action Group
FP Framepointer (-Register)	JVM Java Virtual Machine
FPM Fast Page Mode	
FPU Floating-Point Unit	LAN Local Area Network
FSB Frontside Bus	LE Little-Endian
 	LIFO Last-In First-Out
GAN Global Area Network	LIMDOW Light Intensity Modulation – Direct Overwrite
GPIA General Purpose Interface Adapter	LRC Longitudinal Redundancy Checking
GPIB General Purpose Interface Bus	LRU Least Recently Used
GPR General Purpose Registers	LSB Least Significant Bit
 	LTO Linear Tape-Open
HD Hard Disk	LUN Logical Unit Number
HD High Density	LVD Low Voltage Differential
	L1, L2, L3 Level 1, Level 2, Level 3

MAN Metropolitan Area Network	QAM Quadrature Amplitude Modulation
MAP Manufacturing Automation Protocol	QDR Quad Data Rate
MBus Module Interconnect Bus	QIC Quarter Inch Cartridge
MCA Micro Channel Architecture	
MESI Modified, Exclusive, Shared, Invalid	RAID Redundant Array of Independent/ Inexpensive Disks
MFM Modified Frequency Modulation	RAM Random Access Memory
MIPS Microprocessor without Interlocking Pipeline Stages	RAS Return Address Stack
MMU Memory Management Unit	RDRAM Rambus DRAM
MMX Multimedia Extension	RIMM Rambus In-line Memory Module
MO/MOD Magneto-Optical Disk	RISC Reduced Instruction Set Computer
Modem Modulator/Demodulator	RLE Run Length Encoding
MPEG Moving Pictures Experts Group	RLL Run Length Limited
MPP Massive Parallel Processing	ROM Read-Only Memory
MSB Most Significant Bit	
MT Multithreading	SAN Storage Area Network
NaN Not-a-Number	SAS Serial Attached SCSI
NGIO New Generation I/O	SASI Shugart Associates System Interface
NRZ Non Return to Zero	SATA Serial ATA
NRZI NRZ with interchange	SBus System Expansion Bus
NUMA Non-Uniform Memory Access	SCI Scalable Coherent Interface
ODR Octal Data Rate	SCSI Small Computer System Interface
OSI Open Systems Interconnection	SD Single Density
OS Operating System	SDLC Synchronous Data-Link Control
PATA Parallel ATA	SDR Single Data Rate
PBX Private Branch Exchange	SDRAM Synchronous DRAM
PC Personal Computer	SDSL Symmetric/Single-line DSL
PC Phase Change	SE Single-Ended
PC Program Counter	SECDED Single Error Correction, Double Error Detection
PCI-Bus Peripheral Component Interconnect Bus	SIMD Single Instruction, Multiple Data
PCI-SIG PCI Special Interest Group	SIMM Single In-line Memory Module
PCM Puls-Code-Modulation	SLR Scalable Linear Recording
PDA Personal Digital Assistant	SMP Symmetrical Multiprocessing
PDD Professional Disc for Data	SMT Simultaneous Multithreading
PIO Programmed I/O	SODIMM Small Outline DIMM
PLL Phase-Locked Loop	SP Stackpointer (-Register)
PM Page Mode	SPARC Scalable Processor Architecture
PowerPC Performance Optimization with Enhanced Risc Processor Chip	SR Statusregister
PROM Programmable ROM	SRAM Static RAM
PSB Processor System Bus	SSA Serial Storage Architecture
	SSD Solid State Disk
	SSE Streaming SIMD Extension
	SSE2 Extension to the SSE
	SSP Supervisor Stackpointer (-Register)

SST Source Synchronous Transfer

TAP Test Access Port

TLB Translation Look-aside Buffer

TLP Thread-Level Parallelism

TR Travan

TTL Transistor-Transistor-Logik

UART Universal Asynchronous Receiver
Transmitter

UDMA Ultra-DMA (bei IDE)

UDO Ultra Density Optical

UMA Uniform Memory Access

UMTS Universal Mobile Telecommunications System

USB Universal Serial Bus

USP User Stackpointer (-Register)

V Overflow

VB Vectorbase (-Register)

VDSL Very high bit rate DSL

VESA Video Electronics Standards Association

VITA VMEbus International Trade Association

VL-Bus VESA Local Bus

VLAN virtuelles LAN

VLIW Very Long Instruction Word

VMEbus Versa Module Europe Bus

VMSbus VME Serial Bus

VMXbus VME Extended Bus

VRC Vertical Redundancy Checking

WAN Wide Area Network

WB Write Back

WLAN Wireless LAN

WORM Write Once Read Many

ZBR Zone-Bit Recording

2eVME Two-Edge transfer VME

2eSST Two-Edge Source Synchronous
Transfer

Sachverzeichnis

Seitenzahlen in Fettschrift verweisen auf Erklärungen der Begriffe, Seitenzahlen in Normalschrift auf deren Verwendung.

A

- Absolutlader 46
- Abstraktion v.d. Maschine 220
 - durch das Betriebssystem 220
 - durch den Übersetzer 222
- Accelerated Graphics Port (AGP) 286, 287
- Adapter 455
 - für Geräte 455, 467
 - für Netze 533
 - für Peripheriebusse 467
- Address-Aliasing 419, 430, 433, 434
- Address-Pipelining 385
- Adresse 19
 - absolute (feste) 47, 166
 - effektive 76
 - globale 427
 - logische 436
 - numerische 39
 - physikalische 436
 - reale 47, 428, 436
 - relative 79, 436
 - symbolische 39
 - verschiebbare 47
 - virtuelle 428, 436, 439
- Adressierung, basisrelative 79, 241
 - befehlzählerrrelative 80, 94, 112, 165, 171
 - codierte 304
 - direkte 78
 - Direktoperand- 50, 77, 111
 - immediate 77
 - implizite 24, 352
 - indizierte 82, 112
 - isolierte 302, 352, 457
- , Postinkrement- 79
- , Prädekkrement- 79
- , reale 432
- , Register- 78, 111
- , registerindirekte 78, 111, 171
- , relative 79
- , Speicher- 78
- , speicherbezogene 302, 457
- , speicherindirekte 82
- , überdeckte 24
- , überlappende 385
- , uncodierte 304
- , verschränkte 384, 392
- , virtuelle 428
- Adreßbus 22
- Adreßerweiterung (durch MMU) 439
- Adreßlänge 19
- Adreßraum 19, 225
 - globaler 294, 297
 - lokaler 298
 - realer 437, 438
 - virtueller 437, 438, 440, 442, 447
- Adreßquerbezug 202
- Adreßraumbelegungsplan (memory map)
303, 359, 436
- Adreßrechnung, dynamische 76
 - statische 165
- ADSL (asymmetric DSL) 537
- Advanced-Optical-Disc (AOD) 612
- AGP 286, 287
- Akkumulator 24
- Aktualisierungsstrategie 417-428
- Algorithmus 162
- Alignment 75, 111, 167, 305, 308, 322
- ALU 26, 71
- Amplitudenmodulation 535, 537
- Am29000 108, 223

- analoge Signaldarstellung 534, **535**
- Annullierungsbit **63**, 118, 189
- anti dependency → Gegenabhängigkeit
- AOD (Advanced-Optical-Disc) 612
- Arbeitsspeicher **18**
- ASCII **6**, 17, 164
- ASCI-2-Rechner (HP) 298
- asm (Schlüsselwort in C) **236**
- Assembler **36**, **42**, 218
- Assembleranweisung **43**, **166**
- Assemblercode **35**
- Assemblereinbindung (in C) **235**
- Assemblermodul **235**, **238**
- Assemblerprogramm **35**, **42**, 162
- Assemblersprache **35**, **42**
- Assemblierung **41**, **44**
 - , bedingte 173
- Assoziativspeicher **21**, **33**
- asymmetrische Signaldarstellung (single-ended) **273**, **497**, 569
- asynchron-serieller Interface-Baustein **504**-**512**
- AT-Bus **281**
- ATA (AT attachment) **562**
- ATAPI-Schnittstelle **564**, 576
- ATF (automatic track finding) **619**
- Athlon-64 **141**, 441
- Abrollen von Schleifen (loop unrolling) 146, **157**
- Ausdruck (expression) **165**
- Ausgabeabhängigkeit **135**
- Ausgangsparameter **193**
- Ausnahmebehandlung **98**
- Ausnahmeverarbeitung **97**, **103**
- Ausschnitt, aktueller **51**
- Autovektor-Interrupt **102**, 332

- B**
- Backbone-Netz **291**, 523
- Backplane-Bus **263**, 270, 299
- backside bus (BSB) **408**
- Backside-Cache 268, 287, **408**
- Back-to-Back-Cycle **400**, **403**
- Bankanwahl, off-chip **380**
 - , on-chip **393**, **395**
- bank interleaving **384**
- Bankkonflikt **387**
- Bankwechsel **383**
- Basisadresse **79**
- Basisadreßregister 172, 360
- Basisbandverfahren **528**
- Baud-Rate **472**, **502**
- Baud-Raten-Generator **477**
- Baumstruktur **524**, **579**
- BCC (block checking character) **515**, **540**, **541**
- BCD-Zahl **17**, 73, 90
- BCS (block checking sequence) **515**,
- Bedingung, mehrwertige **181**
 - , zweiwertige **179**
- Bedingungsbits **28**, **71**, 94, 116
- Befehl **24**
 - , bedingter **154**
 - , nichtprivilegierter 96
 - , privilegierter 96, 104
 - , spekulativ ausgeführter **149**
- Befehls-Cache **53**, 127, 406, 409, **418**
- Befehls-/Daten-Cache **53**, **406**
- Befehlsformat **24**, **49**, **83**, **112**
 - , byteorientiertes **83**
 - , wortorientiertes **83**
- Befehlsklassenkonflikt **133**
- Befehlspipelining **54**
- Befehlspuffer **135**
- Befehlsregister **27**
- Befehlsumordnung **135**
- Befehlszähler **28**, **71**, **110**
- Befehlszyklus **28**
- Bereich, dynamisch verschiebbarer **81**, **94**, **167**, **171**, **436**
 - , nicht verschiebbarer 166, **170**
 - , statisch verschiebbarer 166, **171**
- Betriebsart des Prozessors **72**, **103**, **120**, 212
- Betriebsmittel **251**
 - , gemeinsames 528
- Betriebsmittel-Abhängigkeit **121**, **132**
- Betriebssystem **245**
 - , echtzeitfähiges 245
 - , kooperatives **257**
 - , Multitasking- 221, **256**, 294
 - , präemptives **257**

- Betriebsssoftware 252
BHT (branch history table) **150**
biased exponent **12**
Bibliotheksfunktion 253
bidirektionale Verbindung **23**, **30**, **272**, **471**
Big-endian-Byteanordnung **76**, **111**, **227**, **308**
binary coded decimal (BCD) **17**, **73**, **90**
binary number **9**
-, signed **10**
-, unsigned **9**
Binder (linker) **47**, **202**
Bindetabelle 203
BIOS-ROM 289
BISYNC-Protokoll **514**
Bit **4**, **91**
bit (Schlüsselwort in C) **231**, **232**
Bitfeld **5**, **73**, **91**
bit stuffing **517**, **529**
Bitsynchronisation **502**, **512**
Bitübertragungsschicht (OSI) **514**, **531**
Bitvektor **73**, **179**
Block-Alignment **322**, **390**, **411**
Blockbuszyklus (burst cycle) **322**, **378**, **390**-
394
-, langer **277**
-, minimaler **323**
-, überlappender (pipelined burst) **324**, **388**,
390
Blockmodus (DMA) **548**
Blockprüfzeichen **515**, **517**, **540**
Blocksicherung **540**, **541**
Blu-ray-Disc **613**
Bluetooth **530**
BOP (bit oriented protocol) **512**
box-to-box interconnect **292**
bps (bit per second) **472**
branch prediction **61**, **149**
Branch-History-Register (BHR) **153**
Branch-History-Table (BHT) **150**
Branch-Prediction-Cache (BPC) **61**, **150**,
410
Branch-Target-Address-Cache (BTAC) **151**,
410
Branch-Target-Buffer (BTB) **151**
break (Schlüsselwort in C) **182**
Break-Signal **508**
Breitbandverfahren **528**, **529**, **536**
Bridge bei Bussen **266**, **267**, **281**, **283**, **287**
-, bei Netzen **524**
broadcasting **353**, **517**, **581**, **582**
Broadcast-Netz **522**
BSB (backside bus) **408**
BSC-Protokoll → BISYNC-Protokoll
burst cycle **277**, **322**, **378**, **390-394**
Burst-Data-Transfer-Rate **575**, **607**
Burst-Mode bei DMA **326**, **548**
-, bei DRAMs **394**
-, bei synchronen SRAMs **401**
Bus **22**
-, Adreß- **22**
-, asynchroner **275**, **316**, **327**
-, Backplane- **263**, **270**, **299**
-, Backside- **408**
-, CPU- **268**
-, Daten- **22**
-, Ein-/Ausgabe- **263**, **267**
-, Erweiterungs- **263**
-, Feld- **269**, **499**
-, Frontside- (FSB) **275**, **408**
-, Geräte- **269**
-, geteilter **276**, **385**
-, globaler **260**, **267**, **310**, **325**
-, Host- (CPU-) **268**
-, Huckepack- **263**
-, industrieller **299**
-, Kabel- **265**
-, lokaler **260**, **266**, **324**, **556**
-, Mezzanine- **263**
-, multimasterfähiger **281**
-, Multiplex- (Mux-) **276**, **347**
-, Nachrichten- **267**, **270**, **298**, **299**, **301**
-, paralleler **274**, **279**
-, PC- **281**
-, PCI- **283**, **345-366**
-, Peripherie- **264**, **269**, **273**, **467**, **468**, **560**-
596
-, proprietärer **263**
-, Prozeß- **269**
-, Prozessor- (CPU/Host-) **261**, **266**, **273**
-, Prozessor-/Speicher- **266**, **267**

- , serieller 274, 276, 291
 - , Speicher- 263, 268
 - , split- 276, 385
 - , standardisierter 263
 - , Steuer- 22
 - , synchroner 275, 318, 326, 347
 - , System- 22, 260-266, 269, 273
 - Busanforderung** 326, 327, 328
 - Busankopplung** 314
 - Busarbiter-Daisy-Chain** 328
 - Busarbiterbaustein** 329
 - Busarbitration** 324-331, 357, 389, 424, 552, 555, 570, 582
 - , dezentrale 328
 - , globale 325, 327, 330
 - , lokale 324
 - , nichtüberlappende 328
 - , überlappende 328, 330
 - , zentrale 329
 - Busbandbreite** 275
 - Busbreite** 273, 274
 - , dynamische 74, 306
 - Busfehler** (bus error) 99, 318, 344
 - Busfreigabe** 326
 - Buskommando** (PCI, PCI-X) 352, 367
 - Buskonflikt** 267
 - Bus-Logik** 264
 - Busmerkmale** 280
 - Bus-Parking** 358, 389
 - Busprotokoll** 270
 - Bus-Snooping** 363, 422, 428, 432
 - Bussteuereinheit** 266, 281
 - Busstruktur** (bei Netzen) 523
 - Bustakt(frequenz)** 265, 274, 275, 316, 326
 - Bustaktsignal** 275
 - Bustransaktion** (PCI) 354
 - , geteilte (PCI-X: split) 368
 - , verzögerte (PCI: delayed) 356
 - Bustreiber** (bus driver) 315
 - Busy-Waiting** 459, 462, 464, 470
 - Buszuteilung** 327
 - Buszuteilungszyklus** 326
 - Buszyklus** 277, 316
 - , Block- 277, 322, 390-394
 - , DMA- 277
 - , Einzel- 277, 322, 388
 - , geteilter 282, 295, 368, 389
 - , minimaler 317, 318, 320, 323, 384, 385
 - , überlappender 390
 - Bypassing** 60, 130
 - Byte** 5
 - Byteadresse** 67, 74
 - Byte-Mode** (IEEE 1284) 488
 - Byte-Swap-Befehl** 310
 - Bytezählregister** (DMA) 550
- C**
- C** (carry bit) 10, 71, 88
 - C** (Sprache) 164, 176, 218
 - Cache** 21, 33, 52, 404-417
 - , Backside- 268, 287, 408
 - , Befehls- 53, 127, 406, 409, 418
 - , Befehls-/Daten- 53, 406
 - , Daten- 53, 406, 418, 447
 - , Deskriptor (TLB) 410, 413, 416, 436, 441, 443, 447
 - , direkt zuordnender (direct mapped) 413, 431
 - , einfach assoziativer 414
 - , first-level 406
 - , Frontside- 268, 408
 - , Inline- 407
 - , Look-aside- 407
 - , Look-through- 407, 427
 - , L1-/L2-Cache 297, 362, 391, 404, 409, 414, 416
 - , L3-Cache 298, 391, 404, 406, 414
 - , *n*-Wege-assoziativer (*n*-way set-associative) 415
 - , On-chip- 404
 - , Off-chip- 405
 - , Primär- 406
 - , realer 428, 432
 - , second-level 406
 - , Sekundär- 406
 - , split 406, 414
 - , Tertiär- 406
 - , third-level 406
 - , Trace- 139
 - , unified 404
 - , Victim- 407

- , virtueller **428**
 - , virtuell/realer **428, 433**
 - , vollassoziativer **412, 443**
 - bei Festplattenspeichern **606**
 - Cache-Adressierung, reale **428, 432**
 - , virtuelle **428**
 - , virtuell/reale **428, 433**
 - Cache-Block **352, 410**
 - Cache-Clear-Operation **411, 421, 429, 430, 432**
 - Cache-Controller **390, 407**
 - Cache-fill-Operation **277, 322, 411**
 - Cache-Flush-Operation **421, 429, 430, 432**
 - Cache-Hierarchie **406, 409**
 - cache-hit (Treffer) **411**
 - Cache-Kapazität **406**
 - Cache-Kohärenz → Datenkohärenz
 - Cache-Ladestrategie **406**
 - cache-miss (Fehlzugriff) **411**
 - Cache-Steuereinheit **322, 407**
 - Cache-Zeile (line) **364, 412**
 - , clean **364, 420**
 - , dirty **364, 420, 424**
 - , invalid **364**
 - call by reference **192, 242**
 - call by value **192**
 - CAM → Assoziativspeicher
 - carry bit (C) **10, 71, 88**
 - Cast-Operator **233**
 - CAV (constant angular velocity) **609**
 - CC (condition code) **28, 71, 116**
 - CCITT **469**
 - cc-NUMA-Architektur **297**
 - CD (compact disk) **609**
 - R **610**
 - ROM **609**
 - RW **610**
 - Centronics-Schnittstelle **481-484, 487**
 - char (Schlüsselwort in C) **224**
 - character **6, 164**
 - character stuffing **516, 529**
 - Chip-Multiprocessing (CMP) **160**
 - Chipsatz **281, 322**
 - chip-to-chip interconnect **285, 291, 292**
 - CISC **26, 67, 121**
 - clean line (Cache) **364, 420**
 - CLV (constant linear velocity) **610**
 - Codec **538**
 - Codesicherung **538**
 - Codewortsicherung **538**
 - Codewort **5**
 - Codierer **20**
 - column (RAM) **373, 376**
 - COMA **297**
 - Compatibility-Mode (IEEE 1284) **487**
 - condition code (CC) **28, 71, 94, 116**
 - Connect-Rate (Modem) **536**
 - Content-Addressable-Memory (CAM) →
Assoziativspeicher
 - COP (character oriented protocol) **512**
 - Coprozessor **89, 122, 128**
 - Coprozessorsystem **123, 134**
 - Copy-back-Verfahren **363, 412, 419, 423, 453**
 - , buffered/posted **420**
 - , flagged **420**
 - , simple **420**
 - CPU-Bus **268**
 - CRC-Sicherung **541, 542, 572, 581, 598**
 - Crossbar **127, 141, 295, 526**
 - Cross-Software **47**
 - CSMA/CD-Verfahren **523, 525**
 - CWP (current window pointer) **107**
 - cycle-steal mode **326, 547**
- ## D
- Daisy-Chain, nichtunterbrechbare **338**
 - , unterbrechbare **340**
 - Daisy-Chain-Leitung **264**
 - data alignment **75, 111, 167, 305, 308, 546**
 - data link layer **514, 532**
 - data misalignment **75, 305, 411, 546**
 - data scrambling **567, 577**
 - Daten **4, 22**
 - Datenabhängigkeit **59, 121, 130**
 - Datenanschlusgerät **533, 534**
 - Datenblocksicherung **538**
 - Datenbus **22**
 - Datenbusbreite **273, 274**
 - , dynamische **74, 306**

- Daten-Cache 53, 406, **418**, 447
 Datendarstellung, nichttransparente **516**
 -, transparente 504, **516**, **517**
 Datenendeinrichtung (DEE) **493**, **533**
 Datenfernübertragung (DFÜ) 455, 492, **533**-
538
 Datenformat **73**, 110
 Datenkohärenz **404**, **417**, **420**, **423**, **428**, **431**,
432, **433**
 Datenkonsistenz **418**
 Datenlokalität **51**
 Datennetz **534**
 Datenpufferung **474**
 Datenpaket **581**
 Datenspeicher **52**
 Datentyp **73**, **227**
 Datenübertragung → Übertragung
 Datenübertragungseinrichtung (DÜE) **493**,
533
 DCE → Datenübertragungseinrichtung
 DD (double density) **605**
 DD-Laufwerk **599**
 DDR (double data rate) 274, **275**, 279, **287**,
 300, 390, 396, 401, 403, 569
 DDR/DDR2-SDRAM **396**
 DDR/DDR2-SRAM **403**
 Debugger-Programm 120
 Decodierer 20
 DEE → Datenendeinrichtung
 Definitionsdatei 235
 Delay-Befehl **62**
 delayed branching **60**
 Delay-Slot **62**, 118, 131, 188
 Deltamodulation **538**
 demand paging 444
 Deskriptor **437**
 Deskriptor-Cache (TLB) 410, 413, 416, 436,
 441, 443, 447
 Device-Controller 544, **596**
 Device-Descriptor 248
 Device-Driver 246, **247**, **249**
 DFÜ → Datenfernübertragung
 differential signaling **273**, 472, **498**, 569,
 583, 592
 digitale Signaldarstellung **534**
 digital subscriber line (DSL) **536**
- Digital Versatile Disk → DVD
 DIMM (dual in-line memory module) **380**
 -, ECC- **382**
 -, Registered-/Unbuffered- **382**
 -, Stacked-/Planar- **382**
 -, Zeitparameter **383**
 direct mapped cache **413**, 431
 Direktoperand 50, 64, **77**, 86
 Direktspeicherzugriff 35, **545**
 dirty bit **412**, **420**
 dirty line (Cache) **364**, **420**, 424
 Diskette **598**
 -, Formatieren einer **598**
 Displacement **77**, **79**, 94
 distributed memory **298**
 distributed shared memory (DSM) **297**
 DMA (direct memory access) 35, **545**
 DMA-Anforderung **602**
 DMA-Controller (DMAC) 35, 288, 326, 420,
 455, 545
 DMA-Controller-Baustein **548**-**555**
 DMA-Kanal **548**
 DMA-Zyklus **277**, **546**
 Doppelwort **5**
 double data rate (DDR) 274, **275**, 279, **287**,
 300, 390, 396, 401, 403, 569
 double density (DD) **605**
 double precision **12**
 do-while-Schleife **184**, 186
 DOW-Medium 613
 downstream (download) **537**
 DRAM **32**, **376**
 -, asynchrones **376**-**379**, 394
 -, Blockzugriffe bei **392**, **394**
 -, extended data-out (EDO-) **382**, **395**
 -, Fast Page-Mode- (FPM-) **382**, **395**
 -, Nibble-Mode- **394**
 -, Page-Mode- (PM) **395**
 -, Rambus- (RDRAM) **400**
 -, synchrones (SDRAM) **395**-**400**
 DRAM-Controller **377**, **378**
 DRAM-Modul **380**
 Dreiaußbefehlsformat **24**, **49**, **112**
 Dreipart-Registerspeicher 53, 127
 DRV (Deutsche Referenzversion 7-Bit-

- Code **6**
- DSL (digital subscriber line) **536**
- DSL-Adapter **537**
- DSL-Techniken **536**
- DSM-System **297**
- DTE → Datenendeinrichtung
- Dualcode **9**
- dual-die **406**
- dual-port RAM **21**
- Dualzahl, vorzeichenbehaftete **10, 73, 94, 179**
- , vorzeichenlose **9, 73, 94, 179**
- DÜE → Datenübertragungseinrichtung
- Duplexbetrieb **471, 507**
- Durchgangsparameter **193**
- DVD (digital versatile disk) **611**
- Nachfolger **612**
 - RAM **612**
 - ROM/Video **612**
 - -R **612**
 - -RW **612**
 - +R **612**
 - +RW **612**
- dynamic bus sizing **74, 306**
- E**
- EAROM **33**
- Earth Simulator (NEC) **298**
- EBCDI **6, 17**
- Echobetrieb **507**
- ECP-Mode (IEEE 1284) **490**
- ED-Laufwerk **599**
- EDO-DRAM **382, 395**
- EDVAC **132**
- EEPROM **33**
- EIDE **565**
- Einadreßbefehlsformat **24**
- Ein-/Ausgabe **456-466**
- , asynchron-serielle **500-512, 539**
 - , hardwaregesteuerte **470**
 - , hardwareunterstützte **470**
 - , isolierte **302**
 - , parallele **472-491**
 - , prozessorgesteuerte **455, 456**
 - , softwaregesteuerte **469**
- , speicherbezogene **302**
- , synchron-serielle **512-520, 539**
- mit Direktspeicherzugriff **545**
- Ein-/Ausgabebefehl **302, 467**
- Ein-/Ausgabebus **263, 267**
- Ein-/Ausgabeeinheit **19, 34**
- Ein-/Ausgabekanal **555**
- Ein-/Ausgabeprozessor **35, 455, 545, 555, 556**
- Ein-/Ausgaberechner **35, 455, 545, 555**
- Ein-/Ausgabetor **473**
- Ein-/Ausgabezugriff (in C) **225**
- Einbussystem **265, 556**
- Einfachverzweigung **177, 179**
- mit Alternative **177**
- Eingangsparameter **193**
- Einprozessorsystem **325**
- Einzelbuszyklus **277, 322, 388**
- Einzelsicherung **539**
- EISA-Bus **282**
- embedded controller **261**
- Emulation **52**
- Endlosschleife **229**
- eng gekoppeltes Mehrprozessorsystem **294-298**
- Enhanced IDE (EIDE) **565**
- Enhanced-Parallel-Mode (EPP) **489**
- EPIC (explicitly parallel instruction computing) **145, 155**
- EPP-Mode (IEEE 1284) **489**
- EPROM **32**
- Ergebnisrückgabe **192**
- Ersetzungsstrategie **412, 414, 416**
- Erweiterungsbus **263**
- ESDI-Schnittstelle **563**
- Ethernet **523**
- even parity **539**
- exception handling **98**
- exception processing **97, 103**
- Extended-Capability-Mode (ECP) **490**
- extended data-out DRAM **382, 395**
- F**
- faire Priorisierung **329, 465**
- Falle (trap) **97, 99, 134**
- far (Schlüsselwort in C) **226**

- Fast-Ethernet 523
 Fast-Page-Mode-DRAM 382, **395**
FDDI 523
 feed forwarding 60, 130
 Feldbus 269, 499
 Fenster (window) 53, **106**
 Fernsprechnetz, analoges 494, 533, 535
 -, digitales **533**
 Festplattenspeicher **603**
 Festwertspeicher **21**, **32**
 Fibre-Channel (FC) 569, 578, **587**
 FIFO-Prinzip **474**, 518
 FireWire (IEEE 1394a) **569**, **578-586**
 - Host-Adapter 579
 - Schnittstelle **583**
 FireWire (IEEE 1394b) **586**
 First-Fit-Verfahren 254
 Fließbandkonflikt **59**
 Fließbandverarbeitung **54**, 121, 128, **147**,
148, **385**, 432
 floating-point number **12**
 Floppy-Disk-Controller (FDC) **598**, **600-603**
 Floppy-Disk-Kommandos **600**
 Floppy-Disk-Speicher **597**, 608
 Flußdiagramm **162**
 FM (frequency modulation) **535**, **605**
 for-Schleife **184**
 Formfaktor **598**
 four-edge handshake 300
 FPM-DRAM 382, **395**
 FPU (floating-point unit) 12, 16
 Fragmentierung **256**
 Framepointer (FP) **70**, **195**, **239**
 Freispeicher 435, 444, 453
 Freispeicherliste **254**
 Freispeicherverwaltung **254**, 442, 444
 Frequenzmodulation (FM) **535**, **605**
 -, modifizierte (MFM) **605**
 frontside bus (FSB) **275**, **408**
 Frontside-Cache 268, 408
 FSB (frontside bus) **275**, **408**
 Funknetze **529**
 Funktionsaufruf (in C) **239**
 Funktionsrücksprung (in C) **239**
- G**
 garbage collection 442
 GAN (global area network) 521
 Gateway **525**
 Gegenabhängigkeit **135**
 gegenseitiger Ausschuß **558**
 General-Purpose-Interface-Bus (GPIB) **594**
 Generatorpolynom **542**
 Gerätebeschreibung 248
 Gerätetabelle **252**
 Gerätetreiber **246**, **247**, **249**
 -, blockorientierter 247
 -, Netzwerk- 247
 -, zeichenorientierter 247
 Geräteverwaltung **252**
 Gigabit-Ethernet **523**
 Gleitpunktrecheneinheit (FPU) 12, 16
 Gleitpunktzahl **12**, **73**
 globales Netz (GAN) 521
 Grafikbeschleuniger 286
 Grafik-Controller 286
 Graphenfärbung 223
 Gselect-Prädiktor 154
 Gshare-Prädiktor 154
- H**
 Halbbyte **5**
 Halfduplexbetrieb **471**, **516**
 Halbleiterspeicher 30
 Halbwort **5**
 Hammingdistanz **539**
 Handshake-Synchronisation **461**, **471**, **478**,
483, **503**, **516**, **595**
 hard disk (HD) **603**
 Hard-Disk-Controller 562
 Harvard-Architektur **132**
 Hauptplatine 263
 Hauptprogramm **189**
 Hauptspeicher **18**
 HD (hard disk) **603**
 HD-Laufwerk (high density) **599**
 HDLC-Protokoll **517**
 HDSL (high data rate DSL) **537**
 helical scan recording **614**, **618**
 Hexadezimalcode **7**

- high impedance state 272, 315
Hinkanal 530, 537
Hintergrundspeicher 596-619
History-Bit 150
hit ratio (Cache) 406
hochohmiger Zustand 272, 315
Host-Adapter 269, 467, 560, 579
Host-Bus 268
Host-Controller 269
Host-to-PCI-Bridge 283, 287, 346
Hot Plug and Play 265, 587, 589
Hub bei Bussen 265, 274, 580, 589
- bei Netzen 524, 525
- bei Rechnern 289
Huckepackbus 263
HVD (differentielle Signale) 569
Hyperpipelining 148
hyperSPARC (RT600) 434
Hyperthreading (HT) 159
HyperTransport (AMD) 292
- I**
- IA-64 (Intel Architecture 64-bit) 108, 145, 154, 155
IDE, IDE/ATA 562
- Adapter 562
- Kanal 564
- Port 288
- Schnittstelle 562-565, 576
IDE/ATAPI-Controller 564
IEC-Bus 593-596
IEEE 1284 484-491
IEEE 1394 578-587
if-Anweisung 177
if-else-Anweisung 154, 177
iLBX-II-Bus (Multibus II) 299
ILP (instruction-level parallelism) 122, 159
IMP (interface message processor) 499, 521
imperative Programmiersprache 219
include (Schlüsselwort in C) 235
Index 77, 82
Indexregister 82
InfiniBand 292
Informationseinheit 4
Inline-Assembler 236
- Inline-Cache 407
In-order-Issue 129, 134
In-order-Completion 129, 138
instruction-level parallelism (ILP) 122, 159
Instruction-Queue 135
int (Schlüsselwort in C) 176
integer 10
Intel-Prozessoren
- i286 281
- i386 281, 327, 388
- i486 281, 286, 320, 322, 327
- i8088 281
- i960 556
- Itanium-2 108, 145, 154, 155, 406, 408, 446
- Pentium 281, 307, 312, 327, 441
- Pentium-I 133
- Pentium Pro 327, 390
- Pentium-4 139, 159, 223
- x86-Prozessoren 441, 450
Interface 455
Interface-Adapter 455
Interface(-Baustein) 34, 455, 457
-, asynchron-serieller 504-512
-, synchron-serieller 518-520
Interleave-Faktor (Festplatten) 606
interleaving (Hauptspeicher) 384, 392
interlock 60, 61, 129, 130, 131
Interpreter 222
Interprozeßkommunikation 259
Inter-Prozessor-Pipelining 390
Interrupt 97
-, allgemeiner 101
-, maskierbarer 72, 102, 336
-, nichtmaskierbarer 72, 101, 336
-, präziser 134
-, spezieller 99
-, unpräziser 134, 139
-, vektorisierter 336
Interrupt (in C) 228, 230
interrupt (Schlüsselwort in C) 228
Interrupt-Acknowledge-Zyklus 335, 339, 341
Interruptanforderung, codierte 332, 466
-, uncodierte 336, 466, 602
Interruptbehandlung 98, 229

- Interrupt-Bit/Flag **460**
 Interruptcode **101, 333**
 Interrupt-Controller **342**
 Interrupt-Daisy-Chain **338, 340**
 Interruptebene **72, 101**
 Interrupt-Enable-Bit **338, 462**
 Interruptmaske **72, 101, 212, 333, 466**
 Interruptmaskenbit **236, 339, 340**
 Interruptprogramm **95, 104, 201, 331**
 Interruptquelle **331**
 Interruptsignal **332, 460**
 Interruptvektor **98**
 Interruptzyklus **103, 212, 332, 335, 339**
 Intra-Prozessor-Pipelining **389, 390**
 invalid line (cache) **420**
 i/o port **473**
 iPSB-Bus (Multibus II) **299**
 irq-on (Schlüsselwort in C) **229**
 IRV (Internationale Referenzversion 7-Bit-Code) **6**
 ISA-Bus **281**
 ISDN **533**
 isochrone Übertragung **579, 581, 587**
 iSSB-Bus (Multibus II) **299**
 Itanium-2 **108, 145, 154, 155, 406, 408, 446**
 ITU 469
- J**
- Java Virtual Machine (JVM) **222**
 JBOD (just a bunch of disks) **292**
 JTAG **351**
 Just-in-time-Übersetzer (JIT) **222**
- K**
- Kabelbus **560**
 Kanalprogramm **555**
 Kassettengeräte (streamer) **615-619**
 Kellerspeicher → Stack
 Knotenrechner **521**
 Koaxialkabel **526, 527, 561**
 Kohärenzproblem → Datenkohärenz
 Kohärenzprotokoll → MESI-Protokoll
 Kollisionsvermeidung **499, 523**
 Komplementierung **10**
 Konfigurierung (PCI-Bus) **359**
 Konsistenz → Datenkonsistenz
- Kontextwechsel **159, 257**
 Kontrollfaden (thread) **158**
 kooperatives Multitasking **257**
 kritischer Pfad (beim Fließband) **147**
 Kreuzsicherung **540**
 Kupferkabel **526**
- L**
- label **39, 164**
 Lade-Befehl **50, 115, 206**
 Lader **46**
 -, bindender **47, 202**
 -, verschiebender **47, 166, 171**
 Ladestrategie bei Caches **406**
 Längssicherung **539, 540**
 Längsspurverfahren **614, 618**
 LAN (local area network) **456, 521, 522-526**
 LAN-Controller **522**
 land (CD-ROM) **609**
 Langbandgerät **614**
 last-in first-out (LIFO) **69, 466**
 Latenzeit (eines Befehls) **148**
 Lauflängenbegrenzung (RLL) **606**
 Lauflängencodierung (RLE) **491**
 Lead-off-Cycle **391, 393, 399, 410**
 leaf routine (SPARC) **118**
 least recently used (LRU) **151, 413, 416**
 least significant bit (LSB) **5**
 Leitwerk **27**
 Leseverstärker (SRAM) **373**
 Lesezyklus **316, 319, 354, 375, 379, 399**
 Lichtwellenleiter **527, 561**
 LIFO-Prinzip **69, 466**
 LIMDOW-Verfahren **613**
 linear recording **614, 618**
 Link **291**
 linking loader **47, 202**
 link register **118, 191, 208**
 link table **203**
 Linux (μ Clinux) **248**
 Little-endian-Byteanordnung **76, 111, 227, 308, 354**
 Load-/Store-Architektur **50**
 Local-loop-Betrieb **507**
 lokales Netz (LAN) **456, 521, 522-526**

- Look-aside-Cache **407**
Look-through-Cache **407**, 427
loop unrolling 146, **157**
lose gekoppeltes Mehrprozessorsystem **298**
LRC-Sicherungsverfahren **540**
LRU-Strategie 151, **413**, 416
LSB (least significant bit) **5**
LUN (SCSI) **574**
LVD (differentielle Signale) **569**
L1-, L2-Cache 297, 362, 391, **404**, 409, 414, 416
L3-Cache 298, 391, 404, 406, 414
- M**
- Mächtigkeit einer Programmiersprache 219
Magnetbandspeicher **614-619**
magnetooptische Speicher (MO, MOD) **613**
main (Schlüsselwort in C) **228**
main board 263
major number 250
Makro **174**
Makro (in C) **233**
Makroassembler 174
Makroaufruf **174**
Makrobefehl **173**
Makrobibliothek **173**
Makrodefinition **174**
Makroexpansion **174**
MAN (metropolitan area network) 521
Manchester-Signalcodierung **529**
MAP (manufacturing automation protocol) 524
Maschinenbefehl **28**
Maschinenbefehlszyklus **28**
Maschinencode 35, **39**, **45**, 162
Maschinenmodul **235**, **238**
Maschinenprogramm **35**, **218**
massive parallel processing (MPP) **298**
Master **22**, **35**, 316, 324, 546
Master-Slave -Polling **499**
Master-Slave-Prinzip **324**, 589
MBus 434
MCA-Bus 282
Mehrbusssystem **266-270**, **556**
Mehrachverzweigung 179, **181**
Mehrmastersystem **546**, **555**
Mehrprogrammbetrieb 221, 245, **256**, 294, 428, 435
Mehrprozessorsystem **294**, 310, 325, 389, **422**
-, massiv paralleles **298**
-, nachrichtengekoppeltes (lose gek.) **298**
-, speichergekoppeltes (eng gek.) **294-298**
-, symmetrisches (SMP) **294-298**
Mehrpunktverbindung **467**, 492, 497, 498
Mehrrechnersystem 325
memory bus 263, **268**
Memory-Management-Unit (MMU) → Speicher verwaltung
memory map **303**, **359**, 436
message bus 267, **270**, 298, 299, 301
message passing 293, 298
MESI-Kohärenzprotokoll 295, 297, **423**, 425, 429, 432
Metronetz (MAN) 521
Mezzanine-Bus **263**
MFM (modified frequency modulation) **605**
Mikrobefehl 28
Mikrocomputer **261**
Mikrocontroller 4, **261**
Mikrooperation 28
Mikroprogramm **28**
Mikroprozessor **23**
minor number 250
MIPS-Architektur 104
MIPS R4000 147
Misalignment **75**, **305**, 411
MMU → Speicher verwaltung
mnemonic 28, 218
MMU-Cache (TLB) 410, 413, 416, 436, 441, 443, 447
MO/MOD **613**
Modem 492, 494, **533**, **535**, 537
Modul **202**, 249
modulare Programmierung **202**
Modulo-2-Division **541**
Modusbit **72**
Monitor (Betriebsssoftware) 218
most significant bit (MSB) **5**
Motorola (und ggf. IBM)
- MC680x0-Prozessoren 336

- MC68000 300, 336
- MC68020 307, 320, 327
- MC68040 327, **450**
- MC68060 **450**
- MC88110 **425**
- MPC7451 **448**
- PowerPC 601 **137, 312**, 327, 390, **425**
- PowerPC 970 (IBM) **142, 148, 312**
- MPEG2-Verfahren 611
- MPP-System 270, **298**
- MSB (most significant bit) **5**
- Multibus II **299**, 331
- Multiple-Zone-Recording **605**
- Multiplexbus **276, 347**
- multimasterfähiger Bus **281**, 347
- Multiport-RAM **21**
- Multiprogramming/Multitasking 221, 245, **256**, 294, 428, 435
 - , echtzeitfähiges **245**
 - , kooperatives **257**
 - , präemptives **257**
- Multithreading 122, **157-160**
 - , blockweises **158**
 - , feinkörniges (fine-grained) **159**
 - , grobkörniges (coarse-grained) **158**
 - , taktweises **159**
 - , simultanes (SMT) **159**
- munged Little-Endian-Mode 312
- mutual exclusion **558**

- N**
- N (negative bit) **71**
- Nachindizierung **83**
- Nachrichtenbus 267, **270**, 298, 299, 301
- nachrichtengekoppeltes Mehrprozessor-system **298**
- Namenskonflikt 237
- NaN (floating-point) **13**
- Negativbit (N) **71**
- network layer **532**
- Netz → Rechnernetz
- next PC (nPC beim SPARC) **110, 112, 115**, 119, 121
- nibble **5**
- Nibble-Mode (IEEE 1284) **487**
- Nibble-Mode-DRAM **394**

- non-cacheable 322, **421**, 429, 431, 453
- normalisierte Zahl **12**
- North-Bridge 268, 283, **287**
- Not-a-Number **13**
- no write allocation (Cache) **411, 419**
- NRZ-Signalcodierung 501, **528, 585**
- NRZI-Signalcodierung **518, 529, 592**
- Nullbit (Z) **71**
- Null-Einfügen **518**
- Nullmodem-Verbindung **495**
- NUMA-Architektur **296**

- O**
- Oberprogramm **189**
- odd parity **539**
- ODR (octal data rate) 287
- Oktalcode 7
- on-die **406**
- Open-collector-Verhalten 272
- Operand 4
- Operationscode 24
- Operationswerk 18, **26**
- Opteron (AMD) **141**
- optische Plattspeicher **608-613**
- OSI-Referenzmodell 499, 504, 514, 524, **530**
- OS-9/68000 **247**
- Out-of-order-Issue **135, 138**
- Out-of-order-Completion **134, 135**
- output dependency **135**
- overflow bit (V) **12, 71, 88**
- overrun error **503, 508, 509**

- P**
- packed BCD 17, 73, 90
- page (DRAM) **399**
 - hit **400**
 - miss **400**
- page (MMU) **442**
- page fault trap (MMU) **444**
- Page-Mode-DRAM (PM-DRAM) **395**
- page table (MMU) **443, 446**
- paging (MMU) **435, 436, 442-446**
 - , zweistufiges **446, 449**
- Paketvermittlung **522**
- Parallel ATA (PATA) **563, 566**

- Parallel-Interface-Baustein **474-481**
Parameter **173, 190**
-, aktueller **174, 193**
-, formaler **174, 193**
Parameterübergabe **106, 190, 192, 210, 239**
parametrisierte Konfiguration **246**
Parität, gerade (even) **539, 540**
-, ungerade (odd) **539, 540**
Paritätsbit **6, 506, 539**
parity error **509**
PC (phase change) **610**
PC-Busse **279-287**
PCI-Bus **283, 345-366**
PCI-Express **284, 287, 291**
PCI-to-ISA-Bridge **288, 347**
PCI-to-PCI-Bridge **283, 346, 557**
PCI-X-Bus **284, 366-370**
PCM (pulse code modulation) **538**
PC-Struktur mit Bridges **287**
- mit Hubs **289**
PDD (Professional-Disc-for-Data) **613**
Peer-to-Peer-Netz **530**
Peer-to-Peer-Transfer **579, 581, 589**
Pegelumsetzer-Baustein **497**
Pentium → Intel-Prozessoren
Peripheriebus **264, 269, 273, 467, 468, 560-596**
Phase-Change-Verfahren **610**
Phasenmodulation **535, 537**
physical layer **514, 531**
PIO (programmed i/o) **565**
pipelined burst cycle **324, 390, 400**
pipelining → Fließbandverarbeitung
pit (CD-ROM) **609**
Plattenspeicher, magnetische **603-607**
-, magnetooptische **613**
-, optische **608-613**
plug and play **265, 271, 346, 359, 581**
PM-DRAM **395**
pointer (Zeiger) **78, 225**
Polling **460, 464, 590**
Portabilität **219, 220, 231**
Posted-write-Buffer **347, 419**
Postinkrement-Adressierung **79**
power-on reset **344**
PowerPC 970 (IBM) **142, 148, 312**
PowerPC → s. a. Motorola
Prädikament-Adressierung **79**
Prädikation (predication) **122, 144, 146, 155**
präemptives Multitasking **257**
Präprozessor **232**
precharge time **378, 395**
precharging **379, 395, 398**
Princeton-Architektur **132**
Priorisierung, dezentrale **336, 338**
-, faire **329, 465**
-, gruppenweise **329**
-, prozessorexterne **102**
-, rotierende **548**
-, unfaire **329, 465**
-, zentrale **336**
- durch Polling **336**
Priorisierungsstrategie **343**
Prioritäten (Interrupts) **98, 101**
Prioritätencodierer **332**
Privilegien **103**
privilegierter Befehl **96, 104**
procedural dependency → Sprungabhängigkeit
processor system bus (PSB) **275**
Professional-Disc-for-Data (PDD) **613**
program counter (PC) **28, 71, 120**
Programm **162**
-, dynamisch verschiebbare **77, 81, 94, 167, 171, 436**
-, nicht verschiebbare **166, 170**
-, statisch verschiebbare **166, 171**
Programmflußabhängigkeit → Sprungabhängigkeit
Programmiersprache, imperative **219**
Programmierung in C **218**
Programmlokalität **51**
Programmoptimierung (RISC) **62**
Programmschleife **183**
-, induktive **184**
-, iterative **184, 185**
Programmspeicher **52**
Programmunterbrechung **98, 212**
Programmverzweigung **94**
-, if- **177**

- , if-else- 177
 - PROM 32**
 - , löschbares 32
 - propagation delay (Laufzeit) 301
 - Protokoll → Übertragungsprotokoll
 - Prozessorbus 261, 266, 273
 - Prozessor-/Speicherbus 266, **267**
 - Prozessorstatus 72, 96, 98, 103
 - Prozessorstatusregister 27, **71**, 109
 - Prozessortakt 28
 - Prozeß (task) 257, 428
 - , leichtgewichtiger 159
 - , schwergewichtiger 159
 - Prozeßbus **269**
 - Prozeßidentifikation 430
 - Prozeßsynchronisation **259**
 - Prozeßtabelle 257
 - Prozeßverwaltung **257**
 - Prozeßwechsel 257, **429**, 432
 - Prozeßzustände **258**
 - PSB (processor system bus) 275
 - Pseudoabhängigkeit 136
 - Pseudobefehl **58**
 - Pseudocode 59
 - Pseudo-Little-Endian-Mode **312**
 - Pseudo-LRU-Strategie 413, **416**
 - Pseudomaschinensprache 57
 - Pufferspeicher → Cache
 - Pull-down/up-Widerstand 272
 - Puls-Code-Modulation (PCM) **538**
 - Pulsmodulation 529, **538**
 - , differentielle **538**
 - Punkt-zu-Punkt-Netz **522**
 - Punkt-zu-Punkt-Verbindung 274, 284, **289**, 291, 296, **467**, 492, 494, 526, **560**, 563, 567, 577, 580, 587, 589
- Q**
- QDR (quad data rate), QDR2-SRAM **403**
 - quad data rate (QDR, quad pumped bus) 274, **275**, **287**, 403, 408
 - Quadrature-Amplitude-Modulation (QAM) **535**
 - Quelladresse **24**
 - Quellprogramm **46**
 - Quelltext 218
- Querparität **539**, **540**
 - Quittungsbetrieb → Handshake-Synchronisation
- R**
- RAID 292, **588**
 - RAM **21**, **30**, **372-379**
 - , dynamisches (DRAM) **32**, **376**
 - , statisches (SRAM) **32**, **372**
 - rambus DRAM (RDRAM) **400**
 - RAM-Disk 221, **608**
 - Rapid I/O (IBM) 291
 - RDRAM **400**
 - Read-after-Write-Abhängigkeit **130**
 - Read-modify-write-Cycle 89, 351, 425
 - read only memory (ROM) **21**, **32**
 - Ready-Bit/Flag **459**
 - realer Cache 428, **432**
 - Rechenwerk **26**
 - Rechnernetz 456, **520-533**
 - , Backbone- **291**, 523
 - , Broadcast- **522**
 - , globales (GAN) 521
 - , leitungsvermittelndes **522**
 - , lokales (LAN) 456, 521, **522-526**
 - , paketvermittelndes **522**
 - , Punkt-zu-Punkt- **522**
 - , Weitverkehrs- (WAN) 456, **521**
 - , virtuelles LAN (VLAN) 526
 - Rechnernetzkoppelung **524**
 - Rechnernetzstrukturen **522**
 - Rechnernetzzugang **534**
 - recovery time (DRAM) → Speichererholzeit
 - Referenzaufruf **192**
 - Refresh-Abstand **378**
 - Refresh-Controller 398
 - Refresh-Zyklus **378**
 - Register **19**
 - , globales **106**
 - , lokales **106**
 - register (Schlüsselwort in C) 223
 - Registeradresse **25**
 - Register-Adressierung **78**, **111**
 - Registerbank 230
 - Register-Bypassing **60**

- Registerfenster fester Größe **106**
- variabler Größe **108**
register renaming **136**, **157**
Registersatz, allgemeiner **25**, **68**
Registerspeicher **19**
-, strukturierter **106**, **108**
-, unstrukturierter **105**
Registerumbenennung **136**
relative Adressierung **79**
relocating loader **47**, **166**, **171**
Remote-loop-Betrieb **507**
remote terminal **455**
Rename-Register **137**
reorder buffer **139**
Repeater **524**, **581**
Repositionieren eines Bandes **616**
Reservation-Station **135**
Reset-Logik **344**
resource dependency → Betriebsmittel-
Abhängigkeit
Return-Address-Stack (RAS) **151**
Ringstruktur **522**
RISC **48**, **104**, **121**, **203**
RLE (run length encoding) **491**
RLL (run length limited) **606**
ROM **21**, **32**
-, programmierbares **32**
root hub **580**, **589**
root pointer **440**
rotating priorities **343**
Router **525**
row (RAM) **373**, **376**, **399**
RS-232-C **494**-**497**
RS-422-A **497**
RS-423-A **497**
RS-485 **498**
Rückgabewert **242**
Rückkanal **530**, **537**
Rückordnungspuffer **139**
Rücksprungadresse **95**, **190**
Runden bei Gleitpunktzahlen **16**
run length encoding (RLE) **491**
run length limited (RLL) **606**
R4000 **147**
- S**
Sättigungszähler **151**
Sammelleitung **264**
SAN (storage area network) **587**
SAS (Serial Attached SCSI) **576**
SASI-Bus **568**
SATA (Serial ATA) **565**-**567**
- Anschluß (port) **566**
- Controller **565**, **566**
- Protokoll **567**
SATA II, SATA **565**
Scalable Coherent Interface (SCI) **296**
Schnittstelle **468**
-, Centronics- **481**-**484**, **487**
-, DMA-fähige **470**, **491**
-, ECP-**490**
-, EPP- **489**
-, ESDI- **563**
-, IDE- **562**-**565**, **576**
-, IEEE 1284- **484**-**491**
-, interrupt-fähige **470**
-, parallele **472**-**474**, **481**-**491**
-, RS-232-C- **494**-**497**
-, RS-422-A- **497**
-, RS-423-A- **497**
-, RS-485- **498**
-, SCSI- **563**
-, serielle **492**-**500**
-, Shugart- **602**
-, ST506/412- **563**
-, universelle (parallele) **473**
-, V.10- **497**
-, V.11- **497**
-, V.24- **494**-**497**
-, V.28- **496**
-, X.21- **499**, **532**, **533**
-, X.25- **532**, **533**
Schrägspurverfahren **614**, **618**
Schreib-/Lesespeicher **371**
Schreibverstärker (SRAM) **373**
Schreib-/Leseverstärker (DRAM) **377**, **393**
Schreibzyklus **316**, **319**, **356**, **370**, **375**, **379**
Schrittakt **274**, **500**, **506**
Schrittgeschwindigkeit **472**, **502**
Schwingungsmodulation **529**, **535**

- SCI (bei NUMA-Systemen) 296
- scoreboarding 130
- SCSI-Bus, paralleler **568-576**
 - , serieller **569, 576, 578**
- SCSI-Busphasen **570**
- SCSI-Host-Adapter **570, 575**
- SCSI-Schnittstelle **563**
- SD (single density) **605**
- SDLC-Protokoll **517**
- SDR (single data rate) **275, 286, 401**
- SDRAM 396**
 - , DDR- **382, 393, 396**
 - , DDR2- **382, 396**
- SDRAM-Kommandos **396**
- SDSL (symmetric/single-line DSL) **537**
- SECDED **382, 540**
- Sedezimalcode **8**
- Segmentdeskriptor **437, 452**
- Segmenttabelle **437, 439, 446**
- Segmentüberschreitung **438**
- Segmentverwaltung **436, 437-442**
 - , lineare **441**
 - , symbolische **441**
 - mit Seitenverwaltung **446**
- Seite (page) **442**
- Seitendeskriptor **443, 452**
- Seitentabelle **443, 446**
 - , invertierte **445**
- Seitentabellenverzeichnis **446**
- Seitenverwaltung (paging) **435, 436, 442-446**
 - , zweistufige **446**
- Sektor **518, 604, 609**
- Sektor-Interleaving **606**
- Semaphor **89, 116, 259, 424, 555, 558**
- Serial ATA → SATA
- Serial Attached SCSI (SAS) **576**
- Serial-Storage-Architecture (SSA) **569, 578**
- Server **294, 576**
- shared code **442**
 - data **422, 442**
 - memory **294, 310, 313, 363, 421, 427, 453, 556**
- short (Schlüsselwort in C) **176**
- Shugart-Schnittstelle **602**
- Sicherungsschicht (OSI) **514, 532**
- Sicherung (der Datenübertragung) **538-543**
 - , CRC- **541, 542, 572, 581, 598**
 - , Kreuz-/Rechteck- **540**
 - , Längs- **539, 540**
 - , LRC- **540**
 - , Quer- **539, 540**
 - , VRC- **539**
 - , VRC/LRC- **541**
- Signal, bidirektionales **23, 30, 272, 315**
 - , unidirektionales **23, 32, 272, 315**
 - , 0-aktives **273**
 - , 1-aktives **273**
- Signalcodierung **528**
 - , Differential Manchester- **529**
 - , gleichspannungsfreie **285, 528, 561**
 - , Manchester- **529**
 - , NRZ- **509, 528, 585**
 - , NRZI- **518, 529, 592**
- Signaldarstellung, analoge **534, 535**
 - , asymmetrische (single-ended) **273, 497, 569**
 - , digitale **534**
 - , symmetrische (differential) **273, 472, 498, 569, 583, 592**
- Signalprozessor **226**
- Signaltreiber/-empfänger-Baustein **497**
- signed binary number **10**
- sign extension **12, 77, 92, 110**
- SIMD (single instruction, multiple data) **142**
- SIMM 380**
- Simplexbetrieb **471**
- Simulation **52**
- Simultanes Multithreading (SMT) **159**
- single data rate (SDR) **275, 286, 401**
- single density (SD) **605**
- single-ended signaling (SE) **273, 497, 569**
- single error correction, double error detection (SECDED) **382, 540**
- single precision **12**
- Single-step-Modus **120, 344**
- skalarer Prozessor **121**
- skalierbar **127, 143, 294, 295, 296**
- skew (Laufzeitunterschied) **301**
- Slave **22, 35, 316, 324, 545**
- slot **269**

- SMP-System 160, 270, **294-298**, 327, 389
snoop-hit **422**
snooping **363**, **422**, 428, 432
Snoop-Logik 422, **423**, 453
SODIMM 383
Software-Pipelining 146, **157**
Solid-State-Disk **607**
source-synchronous transfer (SST) **301**, 396
South-Bridge 268, **288**
Spaltenadresse (RAM) **373**, **377**, 392
SPARC-Architektur 104, 106, 110, 114, 147, **154**, 179, 188, 191, 205, 210, 222, 336
Speicher 19
-, globaler (gemeinsamer) **294**, 310, 313, **363**, 421, 427, **453**, 556
-, inhaltsadressierbarer (CAM) **21**, **33**
-, nichtflüchtiger **596**
-, strukturierter **392**
-, verschränkter **383**, **392**
-, virtueller **221**, **435**, **444**
- mit wahlfreiem Zugriff (RAM) **21**, **30**, **372**
Speicher-Adressierung **78**
Speicherbandbreite **275**, **404**
Speicherbank, Off-chip- **380**
-, On-chip- **377**, 395
Speicherbankkonflikt **387**
Speicherbankverschränkung **383**, **392**
Speicherbankwechsel **383**, 387
Speicherblock **380**
Speicherbus 263, **268**
Speichere-Befehl **50**, **115**, **206**
Speichereinheit **379**
Speichererholzeit (DRAM) **378**, **384**, **386**
Speicherfeld (SRAM, DRAM) **373**, **376**, **392**
speichergekoppeltes Mehrprozessorsystem
 294-298
Speicherhierarchie **409**
Speichermodul **379**
Speichernetz **288**, **587**
Speicherplatzreservierung **168**
Speicherschutz **436**, **452-454**
Speicherverschränkung **383**, **392**
Speicherverwaltung **435-454**
-, mehrstufige **446-452**
-, virtuelle **435**
Speicherverwaltungseinheit (MMU) 104, **160**, 221, 428, **435-452**
Speicherwort **5**
Speicherzelle **19**, **68**, **374**, **377**
Speicherzugriff (in C) **225**
Speicherzugriffsrate **403**
spekulative Befehlsausführung **132**, **149**
Spezialregister **68**
split bus **276**, 385
Split-Transaktion **295**, **356**, **368**, **389**
splitter **537**
Spracherweiterungen **231**
Sprung, bedingter **94**, 117, 176
-, indirekter **82**
-, unbedingter **93**, 118, 176
Sprungabhängigkeit **121**, **131**
Sprungbedingung **93**
Sprungdistanz **94**
Sprungtabelle **181**, **182**
Sprungvermeidung **122**, **154**
Sprungvorhersage **61**, 122, 132, **149**, **150**
-, adaptive **152**
-, dynamische **150**
-, statische **149**
- durch Hysterese-Schema **152**
- durch Korrelation **153**
- durch Sättigungszähler **151**
Sprungzielvorhersage **150**
Spur (track) **598**, **604**, 609
SRAM **32**, **372**
-, asynchrones **372**
-, DDR, DDR2 **403**
-, flow-through- **401**
-, pipelined- **401**
-, QDR, QDR2 **403**
-, syncburst **401**
-, synchrones **401**, **405**
SRAM-Modul **383**
SSA (serial storage architecture) 569, 578
SSD (solid state disk) **604**
SST (source synchronous transfer) **301**, 396
ST506/412-Schnittstelle **563**
Stack **69**, 103, 190, 191, **195**, 198, 199, **239**, **466**
Stackpointer **69**, 88

- stale data **417**
 - stall **60**, 129, 157
 - Stapelspeicher→ Stack
 - Startbit **501**, **506**
 - Start-Stopp-Verfahren **500**, 615, 616
 - Startup-Code **243**
 - Statusregister (Interface-) **458**, 459, 470, **477**, **508**
 - Statusregister (Prozessor-) **27**, **71**, **109**
 - Statusretten **103**, 120, **191**, 213, **466**
 - Sternstruktur **274**, **524**
 - Steuerbus **22**
 - Steuerzeichen **7**, 164
 - Steuerwerk **18**, **28**
 - Stichleitung **264**
 - Stoppbit **501**, 506
 - storage area network (SAN) **587**
 - Strang (Bus) **274**, **560**
 - Streamer **615**-**619**
 - Gerätetypen **619**
 - Streamerkassette **615**
 - Stream-Modus **616**
 - string **73**, 92
 - Struktogramm **162**
 - Supercomputing **298**
 - SuperDisk-Speicher **608**
 - Superpipelining **122**, **147**
 - superskalarer Prozessor **122**, **125**, **128**
 - supervisor call **97**, 101, 104
 - Supervisor-Modus **72**, **103**, 120, 212
 - sustained Data-Transfer-Rate **575**, 607
 - sustained Tristate-Signal **272**, 349
 - Swapping **436**, 442
 - Switch bei Bussen **274**, 291, **560**, 588
 - bei Netzen **524**, **526**
 - bei Rechnern **292**
 - switch-Anweisung **182**
 - switched fabric **293**, 588
 - Symbol **165**
 - absolutes **165**
 - relatives **165**
 - Symboltabelle **40**, 44
 - symmetrical multiprocessing (SMP) **160**, **270**, **284**-**298**, 327, 389
 - symmetrische Signaldarstellung (differential signaling) **273**, 472, **498**, 569, 583, 592
 - synchrones SRAM **401**, 405
 - synchrones DRAM (SDRAM) **395**-**400**
 - Synchronisation **457**, **458**-**464**, 474, **501**, **547**, 558, 602
 - , X-ON-/X-OFF- **504**, 516
 - durch Abfrage **460**
 - durch Busy-Waiting **459**
 - durch Programmunterbrechung **460**
 - durch Handshaking **461**, **503**, 516, **595**
 - Synchronisationszeichen **513**, 514
 - synchron-serieller Interface-Baustein **518**-**520**
 - Systembus **22**, **260**-**266**, 269, 273
 - Systembusschnittstelle **469**
 - Systemfunktionen **220**
 - Systeminitialisierung **99**
- T**
- tag (Etikett) **412**
 - Taktgenerator **471**
 - TAP (test access port) **352**
 - Task → Prozeß
 - Telekommunikationsnetz **533**
 - Test-Access-Port (TAP) **352**
 - Thread (Kontrollfaden) **158**
 - thread-level parallelism (TLP) **122**, **159**
 - Thread-Wechsel **159**
 - TLB (Deskriptor-Cache) **410**, 413, 416, 436, **441**, 443, 447
 - TLP → thread-level parallelism
 - TMS320C6x (Signalprozessor) **144**
 - Token-Bus **524**
 - Token-Ring **522**
 - Topologien (Rechnernetzstrukturen) **522**
 - Trace-Cache **139**
 - Trace-Modus **72**
 - Trace-Trap **72**, 101
 - Transfersgeschwindigkeit **472**
 - Transferrate **274**, 472
 - translation look-aside buffer → TLB
 - transparente Datendarstellung **504**, **516**, **517**
 - Trap **97**
 - , allgemeiner **101**
 - , präziser **134**
 - , spezieller **99**

- , unpräziser 134, 139
- Trap-Befehl 96, 119, 252
- Trap-Fenster (SPARC) 53, 108
- Trap-Programm 94, 96, 104
- Trap-Tabelle (SPARC) 119
- Trap-Vektor 98
- Trashing 406
- Trefferquote (Cache) 406
- Treiber 246, 247, 249, 252
- Tristate 272, 324, 349
 - , sustained 272, 349
- true data dependency 59, 130
- turnaround cycle 276, 283, 355
- Twisted-Pair-Kabel 526, 561
- two-edge transfer 301
- U**
- UART 505
- Übergangsparameter 193
- Überlaufbit (V) 12, 71
- Überlappung von Buszyklen 324, 385-390
- Übertragsbit (C) 10, 71
- Übertragung 467, 470
 - , asymmetrische 273
 - , asynchrone 470, 494, 500, 539, 581
 - , broadcast-orientierte 582
 - , hardwaregesteuerte 469
 - , hardwareunterstützte 469
 - , isochrone 579, 581, 587
 - , parallele 289, 470
 - , serielle 291, 470, 500, 512
 - , Sicherung der 538
 - , softwaregesteuerte 469
 - , symmetrische 273
 - , synchrone 470, 494, 512, 539
- Übertragungsgeschwindigkeit → Übertragungsrate
- Übertragungsmedien 526
- Übertragungsprotokoll 458, 469, 501, 513
 - , bitorientiertes (BOP) 512, 516
 - , BISYNC- 514
 - , HDLC- 517
 - , SDLC- 517
 - , zeichenorientiertes (COP) 512, 514
- Übertragungsprozedur 514
- Übertragungsrate 265, 274, 279, 285, 472, 560
 - , effektive 274, 472
 - , Erhöhung der 560
 - , maximale 274
 - bei CDs 609
 - bei DVDs 611
- Übertragungssicherheit 495, 502
- Übertragungswiederholung 538
- Ultra-ATA 565
- Ultra-Density-Optical (UDO) 613
- Ultra DMA 565
- Ultra-SPARC III 223
- Ultra-SPARC IIIi 154
- Ultra-3-SCSI (Ultra160) 569, 572
- Ultra-4-SCSI (Ultra320) 569, 572
- Ultra640 569
- UMA-Architektur 295
- unfaire Priorisierung 329, 465
- unidirektionale Verbindung 23, 32, 272, 471
- Universalregister 68
- Universal Serial Bus → USB
- unpacked BCD 17, 90
- unsigned binary number 9
- Unterbrechung, unpräzise 134, 139
 - , präzise 134
- Unterbrechungsanforderung 72
- Unterbrechungsbedingung 98
- Unterbrechungsprogramm 98
- Unterbrechungssystem 97, 120, 212
- Unterbrechungsvektor 98
- Unterprogramm 189, 208
 - , einfaches 198
 - , geschachteltes 197, 209
 - , reentrant (wiedereintrittsfestes) 201, 216
 - , rekursives 198
- Unterprogrammsprung 95
- upstream (upload) 537
- USB (universal serial bus) 589-593
 - Host-Controller 589
 - Hub 589
 - Port 288
 - Schnittstelle 592
- use (Schlüsselwort in C) 230
- User-Modus 72, 104

V

- V (overflow bit) **12**
- valid bit (Cache) **411, 412**
- Variable, globale **173, 237**
- , lokale **195, 201, 239**
- VDSL (very high data rate DSL) **538**
- Vektor-Interrupt **102, 332**
- Vektornummer **98, 333, 337**
- Vektortabelle **70, 98, 229, 334**
- verdrahtetes Oder **272, 328, 337, 339, 349**
- Vergleichsbefehl **89, 178**
- Vergleich, arithmetischer **179**
- , logischer **179**
- Vermittlungsrechner **521**
- Vermittlungsschicht (OSI) **532**
- verschiebbarer Programmreich **77, 81, 94, 166, 169, 171, 249**
- Variablenbereich **77, 79**
- Verschränken von Speicherbänken **383, 392**
- Verzweigung **177**
 - , binäre **181**
 - , einfache **177, 179**
 - , mehrfache **179, 181**
 - , sequentielle **181**
- VESA local bus (VL-Bus) **282, 286**
- Victim-Cache **407**
- Video-Streamer **618**
- vielfädige Verarbeitung → Multithreading
- Vierfachwort **5**
- virtueller Cache **428**
- virtueller Prozessor **159, 222**
- virtueller Speicher **221, 435, 444**
- virtuell/realer Cache **428, 433**
- VITA 300
- VL-Bus **282, 286**
- VLAN (virtuelles LAN) **526**
- VLIW-Prozessor **122, 128, 143**
- VMEbus **270, 300**
- VMSbus **301**
- VMXbus **301**
- volatile (Schlüsselwort in C) **237**
- Vollduplexbetrieb **471**
- Von-Neumann-Architektur **132**
- Voraufladen (precharging) **379, 395, 398**
- Vorindizierung **83**

Vorrangmodus (cycle-steal mode) **326, 547**

Vorwärtsadreßbezug **45**

VRC-Sicherungsverfahren **539**

VRC/LRC-Sicherungsverfahren **541**

V.10 **497**

V.11 **497**

V.24 **494-497**

V.28 **496**

W

WAN (wide area network) **456, 521-522**

Wartezyklus (wait cycle) **279, 317, 318, 323, 383, 385**

watch-dog timer **317**

Wechselplattenspeicher **608**

Weitverkehrsnetz (WAN) **456, 521-522**

Wertaufruf **192**

while-Schleife **184, 185**

window (SPARC) **53, 106**

windowing (SPARC) **210**

window routine (SPARC) **118**

wired or → verdrahtetes Oder

WLAN (wireless LAN) **529**

WLAN-Adapter **529**

working set **444**

Wort **5**

Write-after-Read-Abhängigkeit **136**

Write-after-Write-Abhängigkeit **136**

write allocation (Cache) **411, 419**

Write-back-Operation **277, 322**

Write-back-Verfahren → Copy-back-Verfahren

Write-through-Verfahren **419, 423, 447, 453**

-, buffered **419**

-, posted **419**

X

xDSL-Techniken **536**

- ADSL **537**

- HDSL **537**

- SDSL **537**

- VDSL **538**

X-ON-/X-OFF-Synchronisation **504, 516**

X.21 **499, 532, 533**

X.25 **532, 533**

Z

- Z** (zero bit) **71**
Zahl, BCD- **17**, **73**, **90**
-, Gleitpunkt- **12**, **73**
-, normalisierte **12**
-, unnormalisierte **13**
-, 2-Komplement- **10**, **73**, **94**, **179**
Zählschleife **184**
Zeichen (character) **6**, **164**
Zeichensynchronisation **502**, **513**, **517**, **598**
Zeiger (pointer) **78**, **225**
Zeigervariable **225**
Zeilenadresse (RAM) **373**, **377**, **392**
zero bit (*Z*) **71**
zero extension **10**, **92**, **110**
Zieladresse **24**
Zip-Speicher **608**
Zone-Bit-Recording (ZBR) **605**
Zufallsstrategie **416**
Zugriffsschutz (Speicher) **436**
Zugriffszeit (RAM) **32**, **375**, **378**, **383**, **385**,
 394, **600**
Zuordnungstabelle **39**
Zuordnungszähler **44**, **166**
Zuse Z3 **132**
Zustandsgröße **73**, **179**
Zweidreßbefehlsformat **25**
Zwei-Phasen-Assembler **44**
Zweiport-RAM **21**, **27**
zyklischer Code (CRC) **541**
Zykluszeit (RAM) **32**, **375**, **378**, **383**
Zylinder (bei Festplattenspeichern) **604**

Zahlen

- 1-Bit-Fehler **539**
1-Bit-Fehlerkorrektur **539**
1-Bit-Prädiktor 151
2-Bit-Prädiktor 151
2-Bit-Fehlererkennung **539**
2-Komplementierung **10**
2-Komplement-Zahl **10**, **73**, **94**, **179**
2-1-1-1-burst **277**, **279**, **323**, **390**
2-1-1-1-1-1-1-burst **390**
2-1/2-1/2-1/2-1/2-1/2-1/2-burst **400**, **403**
3-2-2-2-burst **279**

Druck: Strauss GmbH, Mörlenbach
Verarbeitung: Schäffer, Grünstadt