Materialsammlung:

Rechnergrundlagen

WS 2004/2005

1 Einführung 9
1.1 Moore's Law 10
1.2 Turing Maschine 11
1.3 Historie 11
1.3.1 Die Entwicklung der Rechner-Hardware 11
1.4 Programmierbare Rechner 18
1.5 Schichtenmodell eines Rechnersystems 19
2 Zahlensysteme 21
2.1 Zahlen 21
2.2 Darstellung positiver Zahlen 23
2.2.1 Integer-Zahlen ohne Vorzeichen 23
2.2.2 Addition und Subtraktion 24
2.2.3 Multiplikation 25
2.2.4 Umrechnung einer Zahl der Basis 2^n in eine Zahl der Basis 2^m 25
2.3 Darstellung positiver und negativer Zahlen 26
2.3.1 Integer-Zahlen mit Vorzeichen (Vorzeichen/Betrag) 26
2.3.2 Binary Offset 27
2.3.3 Integer-Zahlen mit Vz in 1er Komplement Darstellung 28
2.3.3.1 Ermittlung des Wertes einer Einer-Komplement Darstellung 28
2.3.3.2 Wertebereich der Einer-Komplement Darstellung 29
2.3.4 Integer-Zahlen mit Vz im 2er Komplement 30
2.3.4.1 Vorbetrachtungen zum 2er Komplement 31
2.3.4.2 Ermittlung des Wertes einer Zweier-Komplement Darstellung 32
2.3.4.3 Wertebereich der Zweier-Komplement Darstellung 32
2.3.4.4 Übertrag und Überlauf 33
2.3.4.5 Bestimmung des Überlauf-Bits (Overflow) 33
2.4 Binär-kodierte Dezimalzahlen (BCD) 34
2.4.1 Wertebereich der BCD-Darstellung 35
2.4.2 BCD-Addition 35
2.5 Gebrochene Zahlen 36
2.5.1 Umwandeln einer gebrochenen Zahl in ein anderes Zahlensystem 36
2.5.1.1 Umrechnung einer Dezimalzahl in ein anderes Zahlensystem 36
2.5.1.2 Hornerschema 38
2.5.1.3 Umwandlung der Nachkommastellen 38
2.5.2 Addierer 39
2.5.3 Addition/Subtraktion positiver Festkommazahlen 40
2.5.3.1 Addition von Zweier-Komplement Zahlen 40
2.5.3.2 Subtraktion von Zweier-Komplement Festkommazahlen 40
2.5.3.3 Beschleunigung der Addition/Subtraktion durch ‚carry-look-ahead' 41
2.5.3.4 Beschleunigung der Arithmetik durch Pipelining 42
2.5.3.5 Zyklischer Einsatz eines Pipeline Addierers 44
2.5.3.6 Beschleunigung der Arithmetik durch look up Tables (LUT) 44
2.5.4 Multiplikation 45
2.5.4.1 Multiplikation einer Zahl mit einer gewichteten Ziffer 45
2.5.4.2 Multiplizierer 45
2.5.4.3 Multiplikation nach der Schulmethode 46
2.5.5 Division 48
2.5.5.1 Division nach Schulmethode 48
2.5.5.2 Berechnung einer einzelnen Quotientenziffer 49
2.6 Little Endian, Big Endian 52

2.7 Fließkommaformate, Gleitkommeformate ... 53
2.7.1 Trennung von Wertebereich und Genauigkeit ... 53
2.7.2 Fließkommazahlen (Floating Point), ... 53
2.7.3 Normalisierung ... 54
2.7.4 Darstellung nach Norm: ... 54
2.7.5 Darstellung von Fließkommazahlen ... 54
2.7.6 Darstellung des Wertes 0 ... 55
2.7.7 Verstecktes Bit ... 56
2.7.8 Darstellung des Exponenten ... 56
2.7.9 Arithmetische Operationen mit Fließkommazahlen ... 57
2.7.10 Rundung beim IEEE 754 Standard ... 57
2.7.10.1 Rundungsstrategie: ... 57
2.7.10.2 Multiplizieren und Dividieren von Gleitkommazahlen ... 58
3 Rechnerarchitekturen ... 58
3.1 Von Neumann ... 59
3.1.1 Eingangseinheit: ... 59
3.1.2 Ausgabeeinheit: ... 59
3.1.3 Speicher ... 59
3.1.4 Rechenwerk/ALU ... 60
3.1.4.1 Funktionen der ALU ... 61
3.1.4.2 Aufbau der ALU ... 62
3.1.4.3 Generierung von Konstanten ... 62
3.1.4.4 Logikfunktionen und Bit-weise logische Verknüpfungen ... 63
3.1.4.5 Schiebeoperationen, Schieben und Durchreichen ... 63
3.1.4.6 Arithmetikfunktionen ... 64
3.1.5 Leitwerk/ Kontrolleinheit/ Steuerwerk ... 65
3.1.5.1 Beispiel eines mikroprogrammierten Steuerwerks: ... 66
3.1.6 Zentrale Verarbeitungseinheit (CPU) ... 67
3.1.7 Wichtige Instruktionstypen des „von Neumann“-Rechners ... 68
3.1.8 Befehlssatz-Architekturen ... 69
3.1.8.1 Stack - Architektur ... 69
3.1.8.2 Akkumulator - Architektur ... 70
3.1.8.3 Register-Register-Architektur – RISC (auch Load-Store-Architektur) ... 71
3.1.8.4 Register-Speicher-Architektur – CISC (Mischung von Akkumulator- und Load-Store-Architektur) ... 71
3.1.9 Beispielrechner ... 72
3.1.9.1 Aufgabe ... 72
3.1.10 Bussysteme ... 74
3.1.10.1 Der Systembus ... 74
3.1.10.2 Adress- und Datenbus ... 75
3.1.10.3 Kontrollbus ... 76
3.1.10.4 Buszyklen ... 76
3.2 Havard ... 79
3.2.1 Modifizierte Havard ... 79
3.2.2 DSPs ... 80
3.2.2.1 Datenflußprinzip ... 80
3.2.2.2 Sättigungsarithmetik ... 80
3.2.2.3 Spezielle Adressierungsarten ... 80
3.2.2.4 1 Zyklus Befehle ... 80
3.2.2.5 Hartverdrahtetes Steuerwerk ... 80
3.2.2.6 Spezialbefehle ... 81

3.2.2.7 Sehr performante ALU ... 81
3.3 CISC ... 81
3.4 RISC ... 82
3.4.1 Die semantische Lücke (Semantic Gap) ... 82
3.4.2 Vorgehen beim Entwurf eines RISC-Prozessors ... 82
3.4.3 Perfektion: ... 82
3.4.4 Kennzeichen von RISC-Prozessoren ... 83
3.4.4.1 Großer Registersatz ... 83
3.4.4.2 Zwei Adressierungsarten ... 83
3.4.4.3 Feste Instruktionslänge ... 84
3.4.5 Beispiel: Instruktionsformat des DLX-Prozessors (Hennessy, Patterson). ... 84
3.4.6 Eingeschränkter Befehlssatz ... 85
3.4.7 Einsatz von Pipelining ... 86
3.4.7.1 Hazards ... 88
3.4.7.2 Hazard-Typen ... 88
3.4.7.3 Weitere Pipeline Beispiele mit Hazards ... 89
3.4.7.4 Sprungbefehle in der Pipeline ... 92
3.4.7.5 Verwendung der Register ... 93
3.4.7.6 Überlappende Registerfenster ... 94
3.4.8 Fazit ... 95
3.5 Architekturen nach Befehls- und Datenstromstruktur ... 95
3.5.1 SISD-Rechner ... 96
3.5.2 SIMD-Rechner ... 96
3.5.3 Neue moderne Architekturen ... 97
3.5.4 Superskalare Prozessoren ... 97
3.5.5 VLIW ... 99
3.5.6 MISD-Rechner ... 100
3.5.7 MIMD-Rechner ... 101
4 Rechnerleistung ... 102
5 Speicherhierarchie ... 102
5.1 Cache-Speicher, die Grundprinzipien ... 102
5.1.1 Amdahl's Gesetz ... 103
5.1.2 Zerlegung des Speichers in Speicherzeilen ... 103
5.1.3 Adressierung ... 104
5.1.4 Gespeicherte Information im Cache ... 104
5.1.5 Algorithmus zum Lesen des Cache-Speichers ... 105
5.2 Cache-Organisation ... 106
5.2.1 Voll assoziativer Cache ... 106
5.2.2 Direkt abgebildeter Cache ... 107
5.2.3 N Wege Mengenassoziativer Cache ... 107
5.3 Kenngrößen ... 108
5.3.1 Trefferrate (Hit Rate), Fehlrate (Miss Rate) ... 108
5.3.2 Mittlere Zugriffszeit ... 108
5.3.3 Überschreiben von Cache-Zeilen ... 108
5.4 Schreiben von Daten ... 108
5.4.1.1 No-Write: ... 108
5.4.1.2 Write-Through: ... 108
5.4.1.3 Write-Back: ... 109
5.5 Lesen von Daten ... 109
5.5.1 on demand, demand fetching ... 109
5.5.2 Prädiktiv, prefetch ... 109

6 Speichermanagement 109
6.1.1 Nachteile der Kopplung CPU-interner Adressen mit Speicheradressen 109
6.1.1.1 Nachteil 1: Anpassung der Speicherbelegung eines Programms an die Konfiguration eines jeweiligen Rechners. 109
6.1.1.2 Nachteil 2: Zuweisung eines speziellen Speicherbereichs an jede Task bei Multitasking. 110
6.1.1.3 Fazit 111
6.2 Virtuelle und physikalische Adressen 111
6.2.1 Umsetzung virtueller Adressen auf physikalische Adressen 112
6.2.1.1 Prinzip der Umsetzung 112
6.2.1.2 Beschleunigung der Umsetzung 113
7 Bussysteme und Schnittstellen 115
7.1 Grundsätzlicher Aufbau eines Mikrocomputers 116
7.2 Typische Anordnung der Chips auf einer Hauptplatine 117
7.3 Chipsätze für Pentium 4 und Athlon-DDR 118
7.4 interne Mikroprozessor-Bussysteme 119
7.4.1 PCI - Peripheral Component Interconnect Bus 119
7.4.2 SCSI - Small Computer Systems Interface 119
7.4.3 EIDE – Anschluss von Festplatten oder CD/DVD 120
7.5 Externe Bussysteme 121
7.5.1 Serielle und Parallele Schnittstellen 122
7.5.2 Spezifikationen und Begriffe 122
7.5.3 Asynchrone serielle Übertragung mit Start- und Stopbits 123
7.5.4 Parallele Datenübertragung 123
7.5.5 IEEE 1394 (FireWire) 125
7.5.6 IrDA - Infrarot-Datenübertragung 126
7.5.7 Bluetooth Mobilkommunikation 127
7.5.8 Ethernet 127
7.5.9 CAN 129
7.5.10 MOST 130
7.5.11 Source Port (synchronous) 131
7.5.12 Source Port (asynchronous) 132
7.5.13 Control Channel 132
8 Mikrocontroller 134
8.1 Spezielle Peripheriebausteine 134
8.1.1 A/D-Wandler 134
8.1.2 D/A-Wandlung 136
8.1.3 UART 137
8.1.3.1 RS (recommended standard) 232 137

Bild 1. Moorsches Gesetz 10
Bild 2. Turing Maschine 11
Bild 3. Komponenten eines Rechners 17
Bild 4. Daten- und Steuerfluß 18
Bild 5. Maschinenprogramm 19
Bild 6. Schichten der Rechnerarchitektur 21
Bild 7. Wertebereich ganzer Zahlen bei vorzeichenbehafteter Integer-Darstellung. 22
Bild 8. Zahlenkreis für Integer 23
Bild 9. VB-Darstellung 27
Bild 10. BO-Darstellung 28
Bild 11. Zahlenkreis für 1K 30
Bild 12. 2K Zahlendarstellung 32
Bild 13. Overflow und Carry 34
Bild 14. BCD Darstellung 35
Bild 15. Addition und Subtraktion von Festkommazahlen 41
Bild 16. Carry look ahead 42
Bild 17. Pipelining 43
Bild 18. Multiplizierer 45
Bild 19. Multiplizierer 46
Bild 20. Schulmethode 46
Bild 21. Multiplikation 47
Bild 22. Rechenwerk 48
Bild 23. Division 49
Bild 24. Division 50
Bild 25. Big/little endian 52
Bild 26. Multiplikation von Gleitkommazahlen 58
Bild 27. Von Neumann Architektur 59
Bild 28. Rechenwerk 60
Bild 29. Steuerung der ALU 61
Bild 30. Alu 62
Bild 31. Konstanten 63
Bild 32. Rechenwerk zum Rechts- und Linksschieben eines Operanden 64
Bild 33. Schieben und Durchreichen 64
Bild 34. Arithmetikfunktionen 65
Bild 35. Modell eines Steuerwerk Befehlsablaufes 66
Bild 36. Mikroprogrammiertes Steuerwerk 67
Bild 37. CPU 67
Bild 38. Stackarchitektur 70
Bild 39. Akkumulator - Architektur 70
Bild 40. Register-Register-Architektur 71
Bild 41. Register-Speicher-Architektur 72
Bild 42. Bussysteme 75
Bild 43. Schreibzyklus 77
Bild 44. Lesezyklus 78
Bild 45. Architekturvergleich 79
Bild 46. Havard Architekturen 79
Bild 47. Klassifizierung der Prozessoren 81
Bild 48. Register Indirekt 83
Bild 49. Adressierung 84

Bild 50. Pipelining ... 86
Bild 51. Pipelining ... 87
Bild 52. Pipelining ... 88
Bild 53. Pipeline hazards ... 89
Bild 54. Overlapping register ... 94
Bild 55. Architekturen ... 96
Bild 56. SISD ... 96
Bild 57. SIMD ... 97
Bild 58. superskalar ... 98
Bild 59. Mikroarchitektur ... 98
Bild 60. Superskalare Prozessorfamilien ... 99
Bild 61. VLIW ... 100
Bild 62. MISD ... 100
Bild 63. MISD ... 101
Bild 64. MIMD ... 101
Bild 65. Speicherhierarchie ... 102
Bild 66. Cache ... 103
Bild 67. Speicheradressierung ... 104
Bild 68. Speicheradressierung ... 104
Bild 69. Cache Speicher ... 104
Bild 70. Lesen des Caches ... 105
Bild 71. Voll assoziativer Cache ... 106
Bild 72. Direkt abgebildeter Cache ... 107
Bild 73. N Wege Mengenassoziativer Cache ... 107
Bild 74. Speicherverwaltung ... 110
Bild 75. Speicherverwaltung ... 111
Bild 76. Virtuelle Adressierung ... 112
Bild 77. Virtuelle Adressierung ... 113
Bild 78. TLB ... 114
Bild 79. TLB ... 115
Bild 80. Mikrocomputer ... 116
Bild 81. Hauptplatine ... 117
Bild 82. Chipsätze ... 118
Bild 83. ISO-OSI ... 121
Bild 84. Serielle und Parallele Schnittstellen ... 122
Bild 85. Serielle Übertragung ... 123
Bild 86. Parallel Übertragung ... 124
Bild 87. USB ... 124
Bild 88. USB Hierarchie ... 125
Bild 89. USB Hierarchie ... 125
Bild 90. Firewire ... 126
Bild 91. Ethernet ... 128
Bild 92. CAN ... 130
Bild 93. MOST ... 131
Bild 94. synchroner Kanal ... 131
Bild 95. asynchroner Kanal ... 132
Bild 96. Control KanalISO/OSI Schichtung ... 132
Bild 96. ISO/OSI Schichtung ... 133
Bild 97. Mikrocontroller ... 134
Bild 98. A/D ... 135
Bild 99. A/D-Wandler ... 136

Bild 100. DA_Wandler 136
Bild 101. UART 137

1 Einführung

Das Skript wurde aus mehreren Vorlagen und eigenen Texten zusammengestellt. Es ist nicht zur Weitergabe bestimmt sondern dient nur zur eigenen Vorbereitung.

Früher wurden Rechner als leistungsfähige Rechenmaschine angesehen. Mittlerweile haben sich die Rechner zu Maschinen mit gigantischen Rechenleistungen weiterentwickelt. Als Basis aller Rechnungen müssen die für die Berechnungen benötigten Zahlen in einer für den Rechner verständlichen und sinnvollen Form dargestellt werden.
Heutige Rechner können jedoch weit mehr als arithmetische Operationen ausführen. Sie dienen z.B. als Maschine zur Textverarbeitung, als Spielkonsole und als Internet-Terminal. Die dahinterliegende Information muß in Codes abgelegt werden, deren Codewörter mit einer für den Anwender sinnvollen Bedeutung verknüpft werden.

Die Informatik ist die Wissenschaft von der Technik und der Anwendung der maschinellen Informationsverarbeitung. Sie beschäftigt sich also vorwiegend mit Rechensystemen. In ihrem Zentrum stehen die Entwicklung und Bereitstellung von Methoden zur Beherrschung sehr komplexer informationsverarbeitender Vorgänge. Dazu zählen die systematische Darstellung und Verarbeitung von Information, sowie deren Übertragung und Speicherung. Aber auch der Rechner selbst ist Untersuchungsgegenstand der Informatik: wie läßt er sich aus einzelnen Komponenten aufbauen, wie wird er programmiert und wie kann er für Nichtspezialisten benutzbar gemacht werden ?
Informatik ist jedoch mehr als bloße Informationstechnik. Sie baut zwar auf den Ergebnissen und Produkten dieser Disziplin auf, weil ohne Technik eine maschinelle Informationsverarbeitung nicht möglich ist. Ihr Aufgabenbereich weist aber wesentlich über das rein Technische hinaus, schon allein deshalb, weil sie immer die ständige Wechselwirkung zwischen Maschine (Computer) und Mensch in ihre Untersuchungen mit einbeziehen muß.
Der Rechner (Computer) ist also Werkzeug und Forschungsgegenstand der Informatik zugleich. Woraus besteht er; wie funktioniert er? Darauf soll in dieser Vorlesung eine Antwort gegeben werden. Dazu müssen wir wissen, wie Information verarbeitungsgerecht dargestellt wird, und welches die geeignetsten Verfahren sind, sie maschinell zu verarbeiten

1.1 Moore's Law

Exponentialgesetz der Mikroelektronik („Mooresches Gesetz")

[Von Gordon E. Moore, Intel Chairman of the Board, 1965 geäußerte Vermutung]

- Die Anzahl der Transistoren je Prozessorchip verdoppelt sich alle 2 Jahre.
- Die Verarbeitungsleistung der Prozessoren verdoppelt sich alle 1,5 Jahre.
- Vervierfachung der Speichergröße alle 3 Jahre.
- Verdopplung der Speicherleistung alle 10 Jahre.
- Zum gleichen Preis die doppelte Leistung in weniger als zwei Jahren.

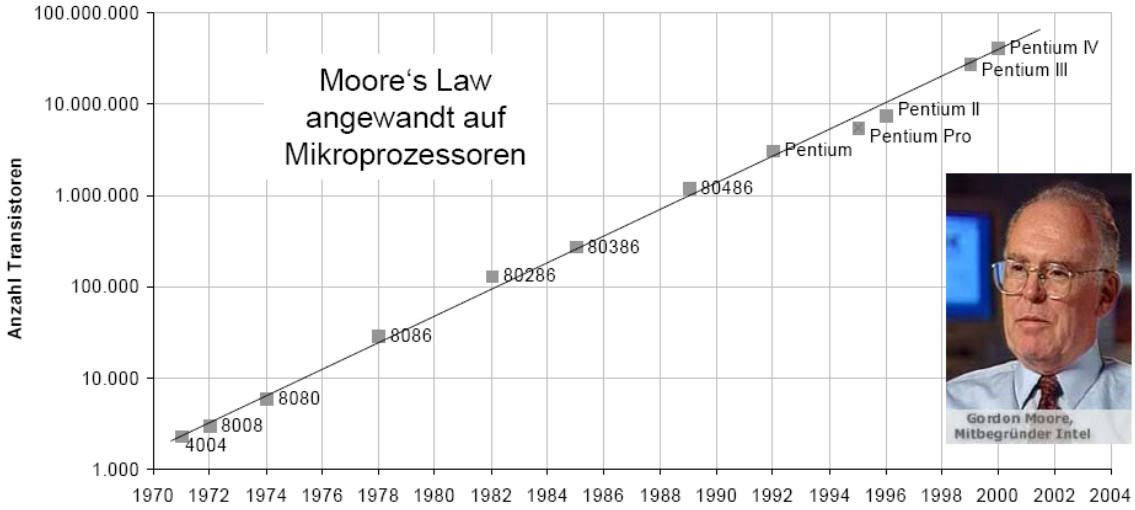
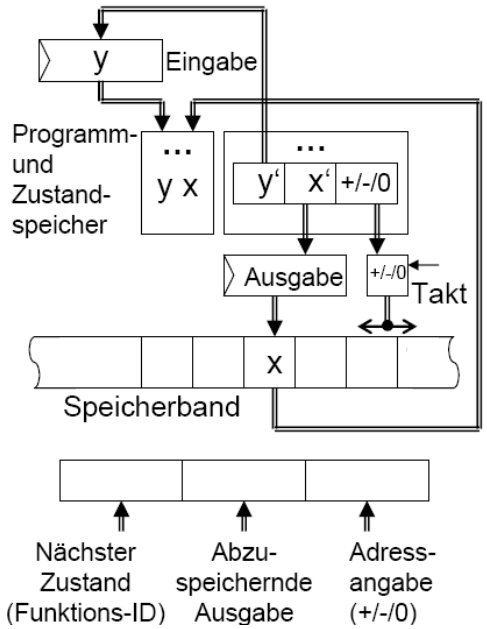


Bild 1. Moorsches Gesetz

!! -> Verhältnis Prozessortakt zu (Haupt)Speicherzugriffszeit wird immer kritischer

1.2 Turing Maschine

Alan M.Turing, 1912-1954, brit. Mathematiker
Turing-Maschine: von Turing 1936 vorgeschlagenes, universelles Automatenmodell zum Studium algorithmischer Problemstellungen (Theoretische Informatik)



- Unendlicher Automat
- Speicherkapazität ist nicht beschränkt
- Speicher besteht aus einem unendlichen Band aus aneinandergereihten Speicherzellen, das vor (+) und zurück (-) geschaltet werden kann. 0 bedeutet „keine Bewegung“.
- Ein Mechanismus realisiert durch Vor- und Zurückschalten den Zugriff auf Speicherzellen
- Das Turing-Band ist unendlich lang und stellt Speicher mit sequentiellem Zugriff dar.
- Adresse der Eingabe ist gleich der Adresse der Ausgabe.
- Den Inhalt der Tabelle (x, y) nennt man das Turing-Programm.

Alan Turing entwickelte 1942 für den britischen Geheimdienst den Rechner Colossus, mit dem die Funk-Codes der deutschen Chiffriermaschine Enigma entschlüsselt wurden.

Bild 2. Turing Maschine

1.3 Historie

1.3.1 Die Entwicklung der Rechner-Hardware

3000 v.Chr.: Erste Rechenhilfen:
Erste Rechenhilfen in Mesepotamien und Babylonien um 3000 vor Christus.

17. Jahrhundert: Maschinen für Grundrechenarten:
Prof. Wilhelm Schickard (Tübingen) entwirft um 1624 zahnradgetriebene Maschine, die sechsstellige Zahlen addiert und subtrahiert.

Blaise Pascal präsentiert 1642 eine Maschine, die achtstellige Zahlen addiert und subtrahiert.

Gottfried Wilhelm Leibniz erfindet 1673 die Staffelwalzen-Rechenmaschine, die alle vier Grundrechenarten ausführen kann.

19. Jahrhundert: Programmsteuerung:
Joseph-Marie Jacquard baut um 1801-1808 lochkartengesteuerten Webstühle

Charles Babbage entwickelt 1833 eine lochkartengesteuerte Rechenmaschine mit mechanischen Speichern.

Seine Freundin Ada Lovelace schreibt dafür die Programme.

Hollerith baut 1882 eine Lochkartenmaschine.

1859: Das Zahlensystem:
Der Mathematiker Georg Bool entwickelt die Boolschen Algebra. Zahlen werden durch Kombination der beiden Ziffern 0 und 1 dargestellt. Eine Ziffer kann durch eine einfache Ja/Nein Entscheidung oder Schaltzustand Ein/Aus repräsentiertwerden. Dies ist die mathematische Grundlage für die Arithmetik in heutigen modernen Rechnern.

1938: Der erste binäre Computer:
Bauingenieur Konrad Zuse baut Rechenmaschine ZUSE Z1 mit mechanischem Speicher, die auf binären Zahlen basiert. Nachfolgemodell ZUSE 3 arbeitet relaisgesteuert.

1944: Theorie der Programmsteuerung:
Mathematiker von Neumann schafft eine umfassende Theorie zur Programmsteuerung. Idee: Programme und Daten in gleicher Form und im gleichen Speicher ablegen.

1945: Der erste elektronische Computer:
John Eckert und John Mauchly bauen den ersten vollelektronischen Computer ENIAC für das merikanische Militär. ENIAC füllt einenRaum von 140 m2, besteht aus 18,000 Elektronenröhren und verbraucht 150kW Strom. Einsatz für Berechnungen zum Bau der ersten Atombombe.

1948: Der Transistor: William Schockley, John Bardeen und William Brattain entdecken in den Bell Laboratorien die Halbleitereigenschaften von Silizium und Germanium und bauen ersten Transistor. 1956 erhalten sie dafür den Nobelpreis für Physik.
1955: Fa. Bell: Erste Rechner mit Transistoren: Starke Erhöhung der Verarbeitungsleistung, Senken des Leistungsverbrauchs. Auftraggeber: amerikanisches Militär.

1950: Die Diskette: Der Japaner Yoshiro Nakamats erfindet die Diskette.

1959: Der integrierte Schaltkreis: Jack Kilby erfindet bei Texas Instruments den integrierten Schaltkreis (IC): Mehrere Transistoren bilden auf einer Siliziumscheibe elektronische Schaltung. Zunächst nur wenige Transistoren, heute mehrere Millionen auf ca. 1-3cm2.

60er Jahre: Großrechner und Minicomputer:
Aufbau von Großcomputern (mainframes) mit ICs: Verkleinerung der Hardware bei Abnahme des Leistungsverbrauchs und Steigerung der Rechengeschwindigkeit. Speicherung der Daten in Ringkern-, später in Plattenspeichern. Ende der 60er Jahre: Minicomputer (PDP, VAX) für wenige Benutzer oder zum Steuern industrieller Anlagen und Maschinen.

1969: Der Mikroprozessor:
Ted Hoff (Intel) entwickelt Chipsatz für japanischen Büromaschinenhersteller Busicom zur Steuerung eines programmierbaren Tischrechners. 4004 Chipsatz ist so universell, dass einfacher Aufbau unterschiedlicher Systeme (Ladenkassen, Geldautomaten, Ampeln etc.) mit gleichartiger Hardware möglich ist. 2300 Transistoren auf 1cm2. 60000 Instruktionen/sec. Intel erwirbt Nutzungsrechte von Busicom zurück. Vermarktung entwickelt sich nur zögerlich, da Anwender breite Unterstützung durch Entwicklungswerkzeuge, Programme, Entwicklungsbeispiele und Schulungen benötigen.

1970: RAM: Der Arbeitsspeicher Bob Abbott (Intel) entwirft dynamisches RAM. Der 1103 Baustein speichert 1024 Bytes.

1971: ROM - Der Programmspeicher Intel stellt EPROM (UV-löschbar) vor. Ursprünglich als Prototyp-Speicher konzipiert.

70er Jahre: Die 8-Bit Mikros.
IC-Herstellern bieten 8-Bit-Mikroprozessoren und Peripherie-ICs samt Entwicklungssoftware an. Bekannteste Typen: 8080 (Intel, 1974), 6800 (Motorola, 1974), Z80 (Zilog), 6502 (Rockwell). Verwendung zunächst für intelligente Geräte.

1975: Die ersten Mikrorechner
Januar 1975:
POPULAR ELECTRONICS stellt Altair Computer von MITS vor. Weitere Firmen (Apple Computer, Commodore und Atari, uvam.) bieten bald auch Personal Computer an. Manche Firmen sind im Elektronikbereich etabliert (z.B. Atari als Hersteller von Videospielen), andere beginnen im Wohnzimmer mit Entwicklung und Herstellung (z.B. Apple Computer, Steve Jobbs).

1979: 16-Bit Mikros
Vorstellung erster 16-Bit Prozessoren: Intel 8086, Motorola 68000 (int.,32-Bit), Zilog Z8000, ...

1981: Der IBM-PC
IBM stellt eigenen Personal Computers, den IBM-PC vor. Konzipiert als intelligentes Terminal für eigenen Großrechner, zunächst Massenmarkt nicht im Auge. Intel 16-Bit Prozessor 8088. Intel übernimmt Marktführerschaft bei Prozessoren. PC-Nachbau, insbesondere in Fernost. Die PC-Alternative 68000 basierte Systeme, Bedienung grafikorientiert. Atari Rechner häufig von Schülern und Studenten gekauft. Apple Rechner verbreitet im Desktop-Publishing Bereich. Marktanteil gegenüber IBMSystemen eher klein.
80er Jahre: Hin zu 32 Bit

Ende 1985:
Intel 80386 Prozessor vor. Es folgen 68030, 80486 (1989), 68040.

Bisher: CISC Prozessoren
Bisherige Prozessoren sind Complex InstructionSet Computer (CISC), d.h. Maschinenbefehle sind komplex und leistungsfähig. Abbildung von Hochsprachenprogramme (C, Pascal, etc.) auf Maschinenbefehle nutzt nicht vollständig deren Komplexität.

Ende der 80er Jahre: Alternative RISC
Reduced Instruction Set Computer (RISC) besitzen für Hochsprachenprogramme optimierte Maschinenbefehle. Statt komplexem CISC Befehls können in gleicher Zeit mehrere einfache RISC Befehle ausgeführt werden.

90er Jahre:
Wettbewerb um Marktführerschaft zwischen CISC (Intel: Pentium) und RISC (IBM, Motorola, Apple: PowerPC; Digital: ALPHA) wieder offen. Hohe Rechenleistungen für Personal-Bereich auf allen Seiten. Marktführerschaft entscheidet sich eher im Softwarebereich: Verfügbarkeit von modernen Betriebssystemen und Softwarepaketen, Kompatibilität zur Software-Vergangenheit.

Geschichte der Betriebssysteme
Einführung, was ist ein Betriebssystem?
Programm zur Verwaltung eines Rechners.
Aufgaben:
- interaktiver Dialog mit Benutzer,
- Verwaltung Plattenspeicher,
- Ansteuerung von Drucker, Bildschirm, etc.
- Zeitlich verschachtelte Ausführung mehrerer Programme mehrerer Benutzer, (multi-user/multi-tasking). Ermöglicht sinnvolle Bedienung von Rechnern.

Die Anfänge Anfang der 50er Jahre Betriebssysteme für die Großcomputer (IBM).
Ende der 60er Jahre Echtzeitbetriebssysteme für Minicomputer (PDP, VAX).
Abhängig von einem Computertyp, in Maschinensprache programmiert, nicht auf andere Typen übertragbar.

1969: Ein Spiel wird portiert AT&T, Bell Laboratorium. 1969 schreibt Kenneth Thompson Betriebssystem UNIX für PDP-7 Computer, um sein Spiel Space Travel, geschrieben für GE,645 Rechner, dort lauffähig zu machen. Er entwirft Sprache „B“ und portiert UNIX in dieser Sprache auf PDP-11 Computer. Dennis Ritchie stößt zu Bell und formt aus „B“ in kürzester Zeit „C“.

1973: Reimplementierung von UNIX in „C“ durch Ritchie, damit weitgehend maschinenunabhängig.

1975: UNIX etabliert sich
UNIX Version 6 wird 1975 breiter Öffentlichkeit vorgestellt. Bis 1978 ca. 600 Installationen. Vermarktung eher zögerlich, Preis 43,000,Dollar. Billige Lizenzen für Universitäten, dort erfolgt Weiterentwicklung.

1979: Version 7. Zeitgleich auf dem Markt mit 16-Bit Mikroprozessoren.
Viele Lizenznehmer (Microsoft, Onix, Zilog, ...), Namensvielfalt (Venix, Cromix, Xenix, Unidos, Idris, ...).

Mitte der 80er Jahre: Vernetzung und Grafik. Network File System, grafische Oberfläche X entsteht. UNIX wird führendes Betriebssystem für Minicomputer und Workstations.
Free Software Foundation: Freiheit für Software

Richard Stallman ruft 1985 GNU Projekt (GNU is Not Unix, mit Betonung auf dem G.) und die Free Software Foundation (FSF) ins Leben. Ziel ist die Freiheit der Software. Vorhandene Softwarepakete werden nachprogrammiert und neue Pakete entwickelt, z.B. Emacs Texteditor, GCC C-Compiler. Alle Pakete können frei weitergegeben werden und sind im Programm-Quelltext verfügbar. GNU Projekt basiert auf vielen Programmierer weltweit; Koordination durch Internet möglich. Linux: UNIX für PCs

1991 beginnt der Student Linus Torwald UNIX für PCs zu entwickeln. Er verwendet Minix (UNIX Lehrsystem für PCs) und GNU C-Compiler. Nach einem halbem Jahr ist Linux Version 0.01 entstanden, Quelltext über Internet verfügbar. Unterstützt durch freie Software vom Internet umfasstVersion 0.12 (Januar 1992) Kommando-Interpreter, Editor, C-Compiler, viele weitere Hilfsprogramme. Zahl der Programmierer, Tester und Unterstützer wächst sehr

stark. Newsgruppemboxtt alt.os.linux wird zur Koordination im Internet eingerichtet. Innerhalb kurzer Zeit entsteht Betriebssystem mit voller UNIX Funktionalität.

MS-DOS, MS-Windows: Wie konnte das passieren?
1981: David programmiert für Goliath
Die Firma Microsoft von Bill Gates und Paul Allen gewinnt den Auftrag von IBM, ein Betriebssystem PC-DOS für IBM-PCs zu liefern. Gates und Allen erwerben DOS Kernprogramme mit unkündbaren Nutzungsrechten.Mit IBM schließen sie einen nichtexklusiven Lizenzvertrag ab, können daher auch anderen PC-Herstellern das MS-DOS-Betriebssystem anbieten. Microsoft besitzt auch die Verkaufslizenz für einen BASIC-Interpreter.
DOS-Funktionalität ist sehr einfach, schränkt PC-Ausbau ein (z.B. nur 640 kBytes Hauptspeicher). Verbunden mit IBM-PC trotzdem MS-DOS Siegeszug, es macht Microsoft zum erfolgreichsten Softwarehaus unserer Zeit. Der Erfolg hält bis heute an.
MS-Windows:
Zweite Hälfte der 80er Jahre: Microsoft versucht MS-Windows als grafikorientierte Oberfläche für PCs einzuführen. Erst mit Version 3.0 gelingt 1990 der Durchbruch. Die Stabilität lässt noch Wünsche offen, trotzdem entstehen viele Programme, z.B. Textverarbeitung, Zeichenprogramme.
Mit Version 3.1 beginnt 1992 die große Verbreitung auf PCs.

Weiterentwicklung: Windows 95, Windows NT, Windows 98, Windows 2000, ...

Die Evolution der Netze
Peer to peer Netzwerke
ARPANET: Überleben beim nuklearen Angriff. Förderung von Rechnernetzen in den 60er Jahren durch amerikanisches Militär.
Motivation: Gesucht wird ein Kommunikationsnetz, in dem nach einem nuklearen Angriff verbleibende Knoten selbständig mögliche Verbindungen wieder herstellen. Vorgeschlagen werden „peer to peer“ (Gleicher zu Gleichem) Netze.
Oktober 1967: Die Entwicklung des ARPANET beginnt. Erster Knoten wird am 1. September 1969 inLos Angeles (UCLA) installiert. Es folgen Knoten in Stanford (SRI), Santa Barbara (UCSB) und Utah. In Utah ist erstmals ``remote login'' möglich.
Oktober 1972:First International Conference on Computer Communications in Washington.ARPANET wird mit 40 Rechnern vorgeführt. Gründung derInterNetwork Working Group, um Protokolle zu definieren. Vision für internationale Netzwerke: Unabhängige lokale Netze, die über standardisierte Gateways verbunden sind.
Email gewinnt im ARPANET große Beliebtheit. Der Aufwand zum Schreiben elektronischer Texte ist gering im Vergleich zum korrekt aufgesetzten Brief. Auch braucht der Empfänger weniger perfekt und förmlich angesprochen werden.
1983: ARPANET wird getrennt in ARPANET und MILNET. MILNET wird in dasDefense Data Net integriert. 1990 löst das NSFNET verbleibendes ARPANET ab.

Store-and-forward Netzwerke
Unix-to-Unix Copy: Computer unterhalten sich
Parallel zum ARPANET werden Netzwerke entwickelt, welche Nachrichten von Rechner zu Rechner kopieren. Basis dieser Systeme ist das Unix-to-Unix Copy Protokoll (UUCP). Es wird 1976 von Mike Lesk in den AT&T Bell Laboratorien entwickelt und 1977 mit UNIX Version 7 ausgeliefert.
UUCPNetzwerke sind langsamer als das ARPANET und werden auch als Netzwerke des armen Mannnes bezeichnet.

CSNET: Kommunikation über Grenzen
1979 organisiert Lawrence Landweber (Univ. Wisconsin) ein Meeting mit DARPA, NSF und Computerwissenschaftlern. Die Entwicklung eines Computer Science Research Network (CSNET) wird vorgeschlagen. Es soll auf UUCP, Modems und dem vorhandenen Telefonsystem aufsetzen. Universitäten ohne Anschluss am ARPANET erwarten damit Verbesserungen der Infrastruktur für Forschung und höhere Attraktivität als Studienstandort. CSNET ist zunächst als lokales Netzwerk geplant, doch bald wird eine Schnittstelle zum ARPANET vorgesehen. Die Kopplung zwischen ARPANET und CSNET soll für den Anwendertransparent werden, d.h. Netzdienste erscheinen auf beiden Netzwerken gleich. Dazu wird das TCP/IP Protokoll entwickelt, das Internet ist ``geboren".
Die Implementierung des CSNET erfolgt in der ersten Hälfte der 80er Jahre.

USENET: Unix User Network
Nachrichtenartikel (News) werden zwischen Rechnern kopiert und weltweit Anwendern verfügbar gemacht. Anwender können Artikel schreiben, lesen und beantworten. Artikel sind in Gruppen zusammengefasst. Gruppen werden hierarchisch einer Hauptgruppen zugeordnet: comp, misc, news, rec, sci, soc, talk und andere.

1979 erstellt Steve Bellovin(Student an derUniv. North Carolina) UNIX Kommandodateien und automatisiert die Kommunikation mit der Duke Universität. Diese Dateien werden in „C“
umgeschrieben und erweitert (news A-Version). 1981 folgt B-Version. Neben UUCP Verbindungen entstehen im USENET Datenverbindungen über ARPANET. Im TCP/IP wird einNet News Transfer Protocol (NNTP) vorgesehen und damit das USENET in das Internet integriert.
Weltweite Verteilung der News ist hierarchisch organisiert. Das hierarchische Backbone (Rückgrad) wird 1983 von Gene Spafford eingerichtet. Neue Newsgruppen werden in die hierarchischen Backbone-Struktur eingefügt. Aber: Keine autoritäre Struktur. Der Administrator jedes Rechners im USENET kann selbständig entscheiden, welche Nachrichten transportiert oder unterdrückt werden.
USENET:The backbone cabal
Mit wachsender Zahl von Anwendern entsteht der Ruf nach demokratischen Entscheidungen im USENET. Es werden Regularien zum Einrichten neuer Gruppen festgelegt.
Richard Sexton schlägt eine Gruppe rec.sex vor, eine Gruppe rec.drugs wird vorgeschlagen, und beide Gruppen werden von den Anwendern gemäß Regularien gewählt. Die Betreiber des Backbone verweigern die Einrichtung und Verteilung dieser Gruppen („backbone cabal“).
Anwender gründen daraufhin eine neue Hauptgruppe tt alt und richten alt.sex und alt.drugs ein. Die alt Verteilung wird unabhängig vom Backbone organisiert und etabliert sich im USENET. Die Revolution der Anwender untermauert die Demokratie (oder Anarchie) im Netz. Trotz Weigerungen wird Gene Spafford, der „Vater“ desBackbone, als Autorität im Netz bestätigt und bleibt für das Einrichten neuer Gruppen zuständig.
USENET:Flame wars
Flame wars (Flammenkriege) sind ein Phänomen zur Meinungsbildung im USENET. Flammenkriege entzünden sich an kontroversen Diskussionen in einer Newsgruppe und schaukeln sich langsam hoch. Lurkers und Newbies beteiligen sich plötzlich an der Diskussion. Ein Flammenkrieg kann tagelang andauern, oder er kann nach kurzer Zeit im Sande verlaufen.

Die Komponenten eines Rechners
Dennoch wollen wir uns jetzt schon eine erste - allerdings noch sehr grobe - Vorstellung über

die Funktion eines Rechners verschaffen.
Als Vorbild diene Student *Hans*, der mit Hilfe bekannter Rechenverfahren (Algorithmen), die er in den einschlägigen Lehrbüchern findet, Information (Daten) zur Lösung von Übungsaufgabe verarbeitet. Dazu benutzt er Hardware: Notizblock, Taschenrechner und Bleistift. Entsprechende Komponenten enthält auch jeder Rechner.
Die Komponenten für die Verarbeitung von Daten und für deren Steuerung bilden den sogenannten Rechnerkern (Zentraleinheit, CPU, Prozessor); den datenverarbeitenden Teil eines Rechnerkerns nennt man das Rechenwerk (Datenprozessor, Datenpfad), den steuernden Teil das Steuerwerk (Leitwerk, Steuerprozessor, Steuerpfad).

Eingabe:	Lehrbuch mit Übungsaufgaben	Peripherie
Speicherung:	Notizblock	Hauptspeicher
Verarbeitung:	Taschenrechner	Rechenwerk
Ausgabe:	Bleistift, Lösungsblatt	Peripherie
Steuerung:	Hans	Steuerwerk

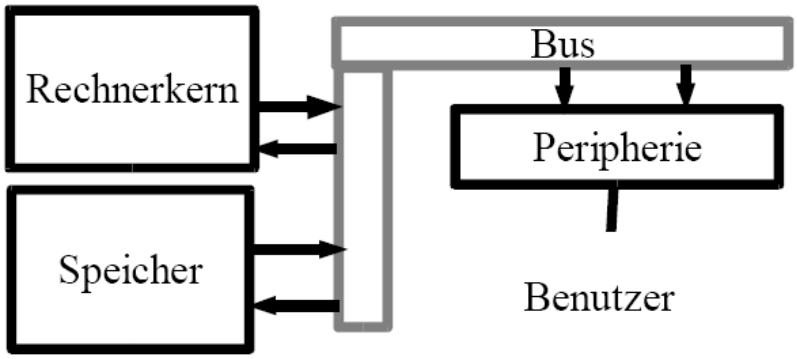


Bild 3. Komponenten eines Rechners

Daten, also Information, fließen von der Eingabe (Input) über die Verarbeitungskomponente zum Speicher oder zur Ausgabe (Output); die Steuerinformation (Signale) wird von dem Steuerteil der Verarbeitungskomponente erzeugt und bestimmt unter anderem, wann und wie Daten eingegeben (eingelesen), gespeichert und ausgegeben (ausgelesen) werden. Das untenstehende Bild zeigt den Daten- und Steuerfluß (Fluß der Steuersignale) zwischen diesen Komponenten.

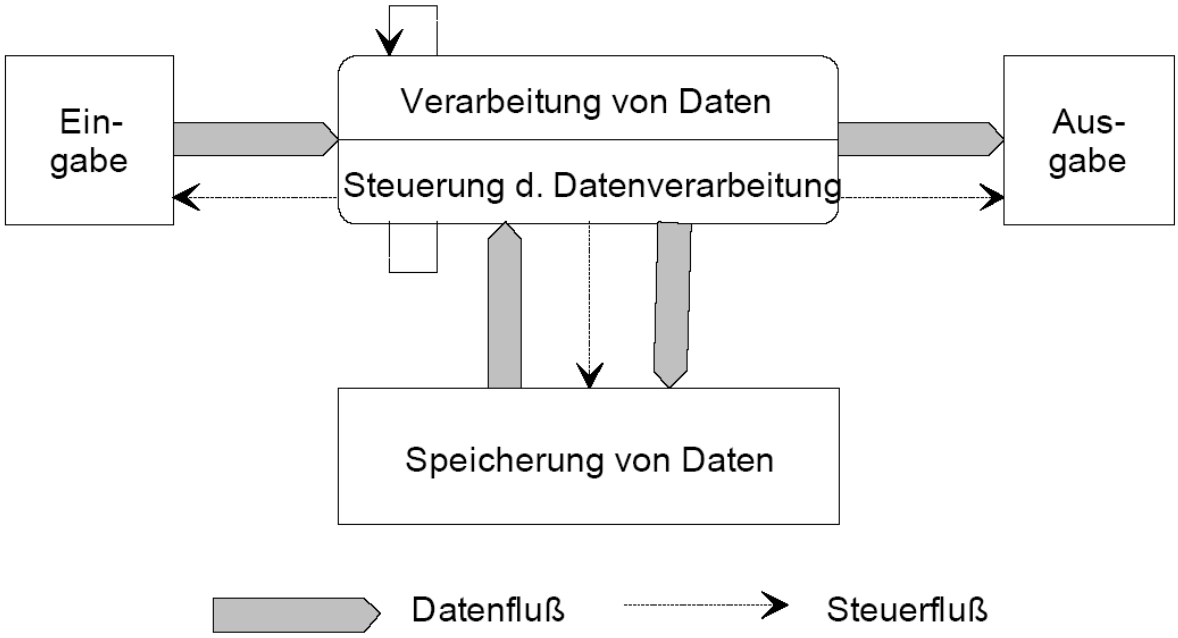


Bild 4. Daten- und Steuerfluß

Die Komponenten für die Speicherung von Information bilden das Speichersystem des Rechners. Es besteht in der Regel aus Hauptspeicher (Arbeitsspeicher) und Massenspeicher**.** Die Ein- und Ausgabekomponenten machen seine Peripherie (Ein/Ausgabe, I/O) aus. Die Peripherie z. B. des Personalcomputers (PC) in Bild 1.3 besteht aus der Tastatur, dem Bildschirm und einer Maus. Die Peripherie eines größeren Rechners enthält meist eine ganze Reihe weiterer Komponenten, sogenannte Endgeräte. Dies sind Geräte zur Erfassung und Eingabe von Informationen sowie zu deren Bereitstellung und Ausgabe - in einer Form, die von den Kommunikationspartnern, dem Mensch oder weiteren Computern, „verstanden“ werden kann.

1.4 Programmierbare Rechner

Ein weiteres wesentliches Merkmal moderner Rechner ist, daß sie programmierbar sind. D.h. für die Steuerung der Datenverarbeitung läßt sich - unabhängig von der Rechnerstruktur - ein Algorithmus angeben. Dieser enthält eine Folge von Anweisungen (Befehle, Instruktionen), die vom Rechenwerk des Rechners ausgeführt werden können.
Der Algorithmus wird zunächst in Form eines Programms in den Hauptspeicher gebracht. Von dort holt das Steuerwerk des Rechnerkerns die Befehle in den Rechnerkern, deutet (interpretiert) sie und veranlaßt das Rechenwerk**,** sie auszuführen. Untenstehende Tabelle zeigt ein solches Programm. (Der Akkumulator ist ein sogenanntes Register, eine spezielle Speicherzelle im Rechnerkern.)
Der Hauptspeicher besteht aus einzelnen Speicherzellen, die die Anweisungen (Befehle, Instruktionen) des Programms und zusätzlich die zu verarbeitenden Daten (Operanden) enthalten. Diese Speicherzellen werden durchnumeriert. Die jeweilige Nummer einer Anweisung dient als Adresse der Speicherzelle.

.

0	LDA 103	\hole Operanden aus Speicherzelle 103 des Hauptspeichers in \den Akkumulator, *load*
1	ADD 101	\addiere zum Inhalt des Akkumulators den Inhalt der \Speicherzelle 101
2	STO 102	\speichere Ergebnis in Speicherzelle 102, *store*
3	LDA 104	\hole den nächsten Operanden
4	SUB 102	\subtrahiere
5	JMZ 7	\bedingter Sprung zur Instruktion 7, wenn das Ergebnis der \vorangegangenen Instruktion gleich 0 ist, *jump if zero*
6	STO 102	\speichere
7	HLT	\beende Maschinen-Ausführung, *halt*

Bild 5. Maschinenprogramm

1.5 Schichtenmodell eines Rechnersystems

Jede Ebene ist eine Abstraktion der jeweils Tieferen. Mehrere Elemente aus der tieferen Ebene werden zusammengefaßt und bilden ein Element der höheren Ebene, unwesentliche Eigenschaften werden unterdrückt.
Zum Beispiel werden Transistoren zu logischen Gattern verschaltet.

Anwendungsprogramme: Auf Benutzerebene sind Rechner heute vielen Menschen bekannt. Auf dieser Ebene startet man Applikationsprogramme, die dem Benutzer ein der Aufgabe angepasstes Erscheinungsbild präsentieren. Vom darunterliegenden Rechner ist wenig zu sehen.
Hochsprachenebene: Auf dieser Ebene erstellen die Softwareentwickler ihre Hochsprachen-Programme z.B. in
C, Pascal, Fortran oder Java. Der Entwickler sieht die von ihm verwendete Hochsprache, deren Datenstrukturen und Objekte. Er braucht sich nicht damit beschäftigen, wie seine Funktionen, Objekte und Daten auf die jeweilige Maschine abgebildet werden. Diese Aufgabe wird ihm vom Übersetzer (Compiler) abgenommen, der die Objekte, Daten und Funktionen von der Hochsprache auf den gewählten Rechnertyp abbildet. Programme einer Hochsprache können mit einem anderen Compiler auf einen anderen Rechner abgebildet werden, und sollten dort (hoffentlich) das gleiche Verhalten zeigen. In diesem Fall spricht man von **Quellcode-Kompatiblilität**.
Assembler/Maschinencode Ebene: Die Hochsprachen haben relativ wenig mit der darunterliegenden Maschine zu tun. Der Übersetzer erzeugt aus den Hochsprachenprogrammen die zu einer Maschine gehörigen lauffähigen Instruktionen. **Instruktionen** beschreiben, wie Daten in der zugehörigen Maschine transferiert und verarbeitet werden. Die Menge aller Instruktionen einer Maschine wird als **Instruktionssatz** bezeichnet. Instruktionen lassen sich direkt auf eine Sequenz von 0 und 1 abbilden. Auf diese Weise sind die Instruktionen im Speicher kodiert. Diese Beschreibung wird als **Maschinencode** oder **Binärcode**
bezeichnet. Da ein Programmieren mit Maschinencode mühsam und fehlerträchtig ist, werden die Instruktionen zusätzlich mit **Mnemocodes** beschrieben. Mnemocodes sind Beschreibungen der Instruktionen, welche eingängiger als die binäre Instruktionssequenz und somit leichter zu merken sind als die Maschinencodes; sie entsprechen diesen jedoch 1:1. Ein mit Mnemocodes formuliertes Programm wird als

Assembler-Programm bezeichnet. Es muss vor der Ausführung mit einem als **Assembler** bezeichneten Programm übersetzt werden. Dieser Assembler übersetzt die Mnemocodes in die zugehörigen Maschinencodes und ersetzt symbolische Adressen durch reale Speicheradressen. Gemäß den Abstraktionsebenen existieren unterschiedliche Rechner mit gleichem Instruktionssatz. Diese Rechner sind somit Maschinencode-kompatibel. Sie unterscheiden sich jedoch ab der darunterliegenden Steuerungsebene.
Steuerungsebene, Mikroprogramm, Hardware: Auf der Sterungsebene werden die Instruktionen sequentiell nacheinander interpretiert und in Einzelschritte zerlegt. Diese Schritte sind Transfers zwischen Speicher und Registern, direkt zwischen Registern und notwendige Verarbeitungsschritte. Die Steuerungsebene lässt sich unterschiedlich realisieren. Früher waren in mittleren und großen Rechnern häufig Mikroprogrammsteuerungen zu finden. Heutige moderne Rechner besitzen aus Gründen der Performance und des Hardwareaufwands festverdrahtete Steuereinheiten.
Funktionseinheiten Ebene: Auf der Ebene der Funktionseinheiten werden die Datenpfade, Register und Verarbeitungseinheiten (ALU) zur Verfügung gestellt, die durch die Steuerungsebene in der zur Instruktionsausführung notwendigen Reihenfolge angesprochen werden.
Logikebene: Die Logikebene umfasst alle Gatterfunktionen, die zur Realisierung der Funktionseinheiten benötigt werden.
Transistorebene: Die Transistorebene ist die physikalische Schicht, mit der die Logikfunktionen realisiert werden.

In ‚Digitale Systeme 1+2' beschäftigen wir uns mit den unteren Schichten 1 und 2, in Rechengrundlagen mit den Ebenen 3 und 4. In den diversen Vorlesungen zu Programmiersprachen beschäftigen wir uns mit den oberen 3 Ebenen.

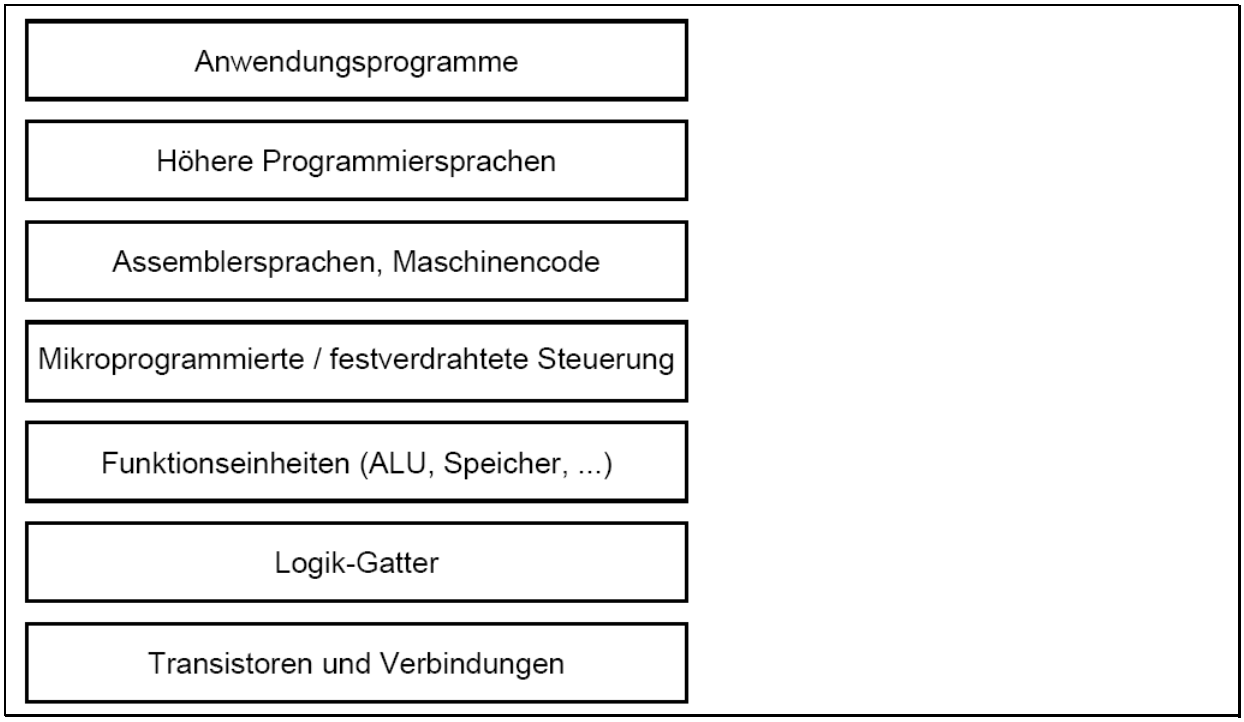


Bild 6. Schichten der Rechnerarchitektur

2 Zahlensysteme

2.1 Zahlen

Grundlage aller Funktionalität der Rechner bildet die dahinter liegende Arithmetik. Zum Verständnis der Funktionsweise von Rechnern gehört daher ein grundlegendes Verständnis, wie die Arithmetik in Rechnern funktioniert. Grundlage zum Verständnis der Arithmetik ist nun die Kenntnis, wie Zahlen in einem Rechner dargestellt werden. Diese Kenntnis wird in diesem Abschnitt vermittelt.

Beispiel: Folgendes Beispiel soll in die Thematik einführen:
136 * 14 = 2303
Der aufmerksame Betrachter wird bei dieser Rechnung sofort anmerken, daß das Ergebnis falsch ist. Er wird anführen, daß die Aufgabe wie folgt zu lösen ist:
136 * 14 = 1904
Dabei hat er jedoch übersehen, daß er implizit die Annahme getroffen hat, daß die betrachtete Aufgabe im Dezimalsystem (Basis 10) gestellt ist. Dies ist im vorliegenden Beispiel nicht der Fall.

Versierte Computerexperten könnten nun vermuten, daß die Berechnung im Hexadezimalsystem (Basis 16) durchgeführt wurde, erhalten dabei jedoch die folgende Lösung:
136 * 14 = 1838

Somit stehen nun schon 3 mögliche Rechnungen im Raum, die bei der gleichen Aufgabe unterschiedliche Ergebnisse liefern. Will man solch eine Aufgabe unabhängig von impliziten

Annahmen über die verwendete Berechnungsbasis stellen, muß die Basis der Berechnung hinzugefügt werden. Damit werden die drei Rechnungen wie folgt komplettiert:

(136)? * (14)? = (2303)?
(136)10 * (14)10 = (1904)10
(136)16 * (14)16 = (1838)16

Sie lassen sich nun eindeutig unterscheiden. Dem Leser sei es überlassen, die noch unbekannte Basis der ersten Zeile zu ermitteln.

Moderne Rechner arbeiten intern nur mit 0 und 1, daher müssen alle Ihnen gestellte Aufgaben auf Berechnungen zur Basis 2 (Dualsystem, Binärsystem) abgebildet werden. Bei arithmetischen Berechnungen werden unabhängig vom Rechnereinsatz die folgenden elementaren Zahlenarten verwendet. Dies sind:

- Natürliche Zahlen (positiv, ohne Nachkommastellen)
- Ganze Zahlen (positiv und negativ, ohne Nachkommastellen)
- Rationale Zahlen (gebrochene Zahlen)
- Reelle Zahlen (gemischt gebrochene Zahlen)

Somit werden zur Lösung arithmetischer Aufgaben im Rechner Zahlendarstellungen benötigt, welche die benötigten Zahlenarten zumindest annähern. Diese Zahlendarstellungen sind auf die Basis 2 abzubilden.

Als Zahlendarstellungen sind in Rechnern verfügbar:

Integerzahlen: Darstellung natürlicher und ganzer Zahlen.

Festkommazahlen: Darstellung rationaler und reeller Zahlen mit eingeschränktem Wertebereich. Diese Darstellung ist normalerweise nicht direkt in Rechnern implementiert, sondern Bestandteil der Fließkommadarstellung

Fließkommazahlen: Darstellung rationaler und reeller Zahlen mit erweitertem Wertebereich.

Natürliche Zahlen: 0, 1, 2, 3, ...
Ganze Zahlen: ... -3, -2, -1, 0, 1, 2, 3, ...

Bild 7. Wertebereich ganzer Zahlen bei vorzeichenbehafteter Integer-Darstellung.

Wertebereich	Bits
-128 .. 127	8
- 32768 .. 32767	16
-2147483648 .. 2147483647	32
-9223372036854775808.. 9223372036854775807	64

(Anmerkung: Wer meint, die 32 Bit-Darstellung sei ausreichend für übliche Anwendungen, sei daran erinnert, daß mit dieser Darstellung Bill Gates nicht einmal die Summe seines Vermögens darstellen kann.)

2.2 Darstellung positiver Zahlen

Allgemein lässt sich eine Zahl *a* wie folgt darstellen (Stellenwertsystem):

$$a = \sum_{i=-\infty}^{\infty} z_i \times b^i$$

$$= z_n \times b_2{}^n + z_{n-1} \times b_2{}^{n-1} + z_{n-2} \times b_2{}^{n-2} + \ldots + z_1 \times b_2{}^1 + z_0 + \frac{z_{-1}}{b_2{}^1} + .. + \frac{z_{-(m-2)}}{b_2{}^{m-2}} + \frac{z_{-(m-1)}}{b_2{}^{m-1}} + \frac{z_{-m}}{b_2{}^m}$$

Darstellung ohne Potenzen:

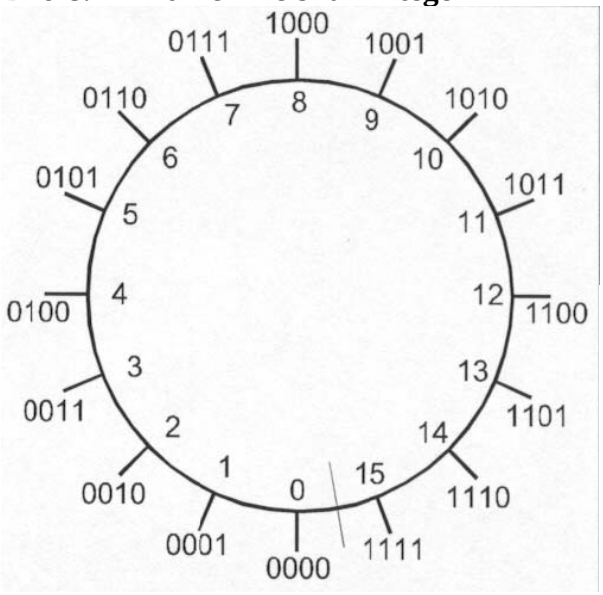
$$= b_2 \times (.. b_2 \times (b_2 \times (z_n \times b_2 + z_{n-1}) + z_{n-2}) + \ldots + z_1) + z_0 + (z_{-1} + \ldots + (z_{-(m-2)} + (z_{-(m-1)} + \frac{z_{-m}}{b_2}) \times \frac{1}{b_2}) \times \frac{1}{b_2} ..) \times \frac{1}{b_2}$$

Der Wert einer Zahl *a* lässt sich also darstellen als eine Summe von gewichteten Ziffern, wobei eine Ziffer z_i mit der mit *i* potenzierten Basis b gewichtet wird.
Es ist Konvention, die Ziffern mit absteigendem Index i als Liste darzustellen und zwischen den Ziffern z0 und z-1 ein Komma (angelsächsisch ein Punkt) einzufügen.

$$a = \ldots z_3 z_2 z_1 z_0 , z_{-1} z_{-2} z_{-3} \cdots$$

2.2.1 Integer-Zahlen ohne Vorzeichen

Bild 8. Zahlenkreis für Integer




```
  1 1 1 1
+       1
-------
1 0 0 0 0
```

Zahlensystem	Basis	Ziffernvorrat	Beispiel
dual	2	0, 1	0101_2
oktal	8	0, 1, ... 7	
dezimal	10	0, 1, ... 9	357_8
hexadezimal	16	0, 1, ... 9, A, ... F	4791

Bsp: $(3564_{16}) = 3 \times 16^3 + 5 \times 16^2 + 6 \times 16^1 + 4 \times 16^0 = 13668$

(((3*16+5)*16+6)*16+4)=13668

2.2.2 Addition und Subtraktion

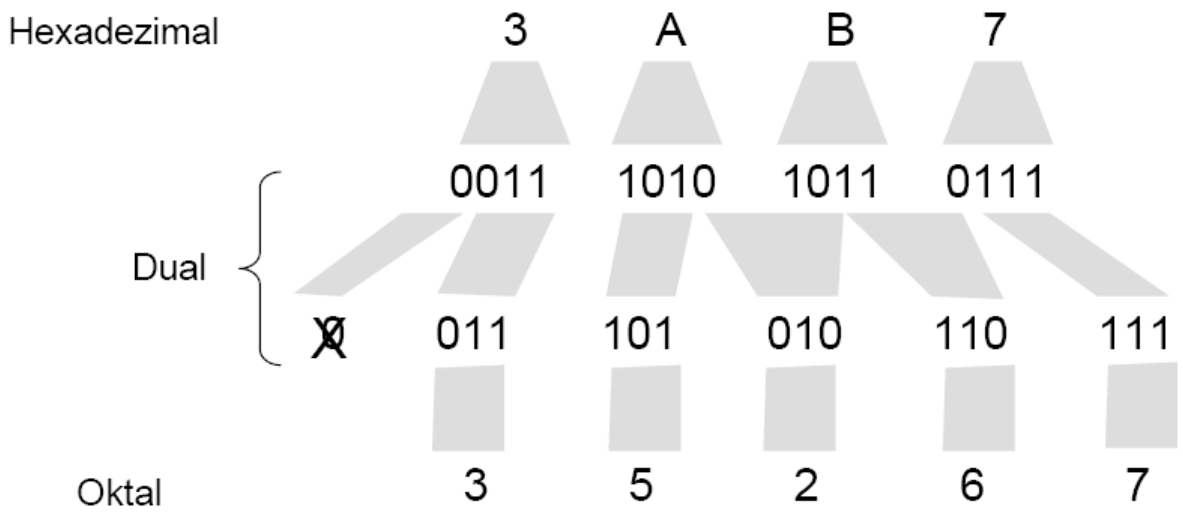
Addition:	Dual	Hexadezimal	Dezimal
	1011	B	11
+	1111	F	15
Übertrag	111		
	11010	1A	26

Subtrakt.:	Dual	Hexadezimal	Dezimal
	1100	C	12
-	1011	B	11
Borgen	11		
	0001	1	1

2.2.3 Multiplikation

	Dezimal	Dual	Hexadezimal	Oktal
	18 x 26 36 108	10010 x 11010 10010 10010 0 10010 0	12 x 1A 12 B4	22 x 36 66 44
Übertrag		1		1
	468	111010100	1D4	724

2.2.4 Umrechnung einer Zahl der Basis 2^n in eine Zahl der Basis 2^m



Multiplikation: x 2 ==> 1 x links schieben
x 4 ==> 2 x " "
x 8 ==> 3 x " "
x 16 ==> 4 x " "

Division: / 2 ==> 1 x rechts schieben
/ 4 ==> 2 x " "
/ 8 ==> 3 x " "
/ 16 ==> 4 x " "

Im Beispiel ergibt die Verschiebung jeweils um 3 Stellen nach rechts direkt den jeweiligen herausgeschobenen Rest und die neue zu dividierende Zahl.

2.3 Darstellung positiver und negativer Zahlen

Die bisherige Darstellung umfasste nur positive Zahlen. Zur Durchführung von Berechnungen wird die Erweiterung auf negative Zahlen benötigt. Es gibt unterschiedliche Vorgehen, um das Vorzeichen in die Zahlendarstellung einzubetten. Es werden die folgenden Möglichkeiten vorgestellt:

- Vorzeichen-Wert Darstellung (Signed-Magnitude)
- Binary Offset
- Einer-Komplement Darstellung
- Zweier-Komplement Darstellung

Da bei Rechnern das Dualsystem Anwendung findet, werden die folgenden Darstellungen im Detail nur noch für *b=2* ausgeführt.

In der Schulmathematik wird den Zahlen ein Vorzeichen vorangestellt, welches bestimmt, ob eine Zahl positiv oder negativ ist. Nachfolgend werden die Ziffern angehängt. In der Schulmathematik kann das positive Vorzeichen meist weggelassen werden, dies ist bei der Darstellung im Rechner nicht sinnvoll.

2.3.1 Integer-Zahlen mit Vorzeichen (Vorzeichen/Betrag)

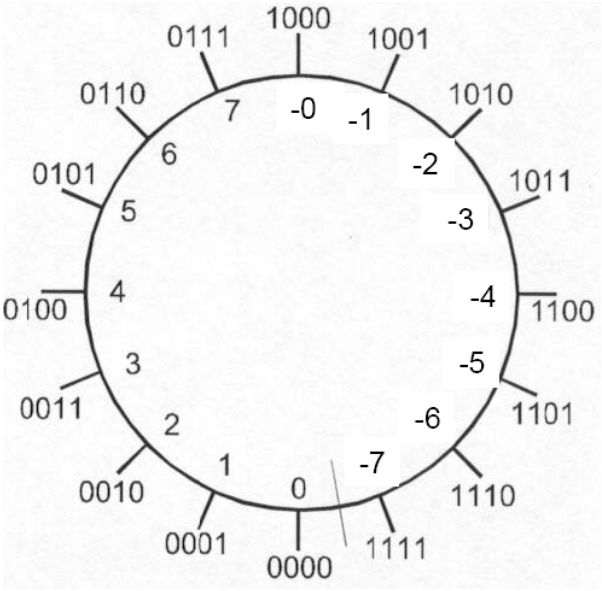
Die Vorzeichen-Wert Darstellung einer Zahl a bezüglich einer Basis b mit *n+1* Vorkomma- und *m* Nachkommastellen wird gegenüber der vorzeichenlosen Darstellung um eine Stelle *s* erweitert und hat die folgende Form:

$$(sz_n z_{n-1} \cdots z_1 z_0 , z_{-1} z_{-2} \cdots z_{-(m-1)} z_{-m})_b$$

Das Vorzeichen *s* kann dabei die Werte 0 (positive Zahl) oder 1 (negative Zahl) annehmen.

$$a = (-1)^s \times \sum_{i=-\infty}^{\infty} z_i \times b^i$$

Bild 9. VB-Darstellung



Kodierung	Vorzeichen-Wert Darstellung ($b=2$)
1111	-7
1110	-6
1101	-5
1100	-4
1011	-3
1010	-2
1001	-1
1000	-0
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0

Einführung eines Vorzeichens:
MSB = 1 ==> negative Zahl
(MSB = Most Significant Bit)
Wertebereich: $-2^{n-1}-1$ bis $+2^{n-1}-1$
Im Zahlenkreis für 4-Bit: -7 bis +7
Beispiele:

6+(-1)=5	6+(-1)=6-1
0110	0110
+1001	-0001
1111 (Fehler!)	0101

Nachteile:
• Vorzeichenbit muß gesondert ausgewertet werden
• Es existieren eine positive Null: 0000 und eine negative Null: 1000
Lösung: 2er Komplement

2.3.2 Binary Offset

Bei der binary offset Darstellung wird der kleinste mögliche Wert mit 0000, der größte mit 1111... dargestellt. Dies entspricht der Zahlendarstellung üblicher A/D-Wandler, mit denen Analogsignale digitalisiert (quantisiert) werden. Durch Invertieren des höchsten Bits verwandeln wir die Darstellung in eine 2er Komplement-Darstellung, die später noch besprochen wird.

Bild 10. BO-Darstellung

Kodierung 4 Stellen	Binary Offset
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

2.3.3 Integer-Zahlen mit Vz in 1er Komplement Darstellung

Zur Darstellung negativer Zahlen wird zunächst die Darstellung des positiven Betrags der Zahl ermittelt. Dabei muß die höchste Ziffer des Betrages gleich 0 sein. Anschließend wird jede Ziffer (einschließlich der höchsten Ziffer) in ihr Komplement $k_i=(b-1)-z_i=1-z_i$ überführt.
Beispiel: Darstellung der Zahl $a=(-27)_{10}$ in der 1er-Komplement Darstellung mit 8 Vorkommastellen (Basis $b=2$):
Zunächst wird der Betrag der Zahl gebildet und zur Basis $b=2$ dargestellt: $|a|=(0011011)_2$.
Da die höchste Ziffer gleich 0 ist, lässt sich die Zahl mit den geforderten 8 Stellen darstellen. Anschließend werden alle Ziffern komplementiert, was bei $b=2$ einer Invertierung der Ziffern entspricht. Damit erhält man direkt die 1er-Komplement Darstellung der Zahl:
a$=(11001100)_2$.

2.3.3.1 Ermittlung des Wertes einer Einer-Komplement Darstellung

Positive Zahl: Ist die höchste Ziffer z_n der Darstellung gleich 0, handelt es sich um eine positive Zahl. Dann entspricht die 1er-Komplement Darstellung der im vorhergehenden Abschnitt 8.2 behandelten Darstellung positiver Zahlen und kann direkt wie dort beschrieben ausgewertet werden.
Negative Zahl: Ist die höchste Ziffer z_n gleich 1, handelt es sich um eine negative Zahl. Dann ist zunächst die Darstellung des Betrags zu ermitteln, dazu sind alle Ziffern zu komplementieren. Der Wert des Betrags wird, da es sich um eine positive Zahl handelt, aus seiner Darstellung als positive Zahl wie bekannt ermittelt.
Die 1er-Komplement Darstellung besitzt (wie auch schon die Vorzeichen-Wert Darstellung) zwei Repräsentationen für die 0. Die Kodierung $(0000...0000)_2$ im Dualsystem ergibt die +0, die Kodierung $(1111...1111)_2$ entsprechend die -0.

Übungsbeispiel: Gesucht ist der Wert der Zahl *a*, die wie folgt in 1er-Komplement Darstellung mit *w=8* Stellen kodiert ist:
a=(1001,1100)$_2$.
Es gilt also *n=3* und *m=4*.
Da die höchste Ziffer gleich 1 ist, handelt es sich um eine negative Zahl.
Komplementieren aller Ziffern führt zu: *(0110,0011)$_2$*. Diese Darstellung entspricht dem Betrag der gesuchten Zahl, der als positive Zahl mit Basis *b=2* kodiert ist.
Umrechnung der Darstellung ins Dezimalsystem ergibt:
|a|=(0110,0011)$_2$=(6,1875)$_{10}$.
Der Wert der gesuchten Zahl *a* beträgt somit *a=(-6,1875)$_{10}$*.

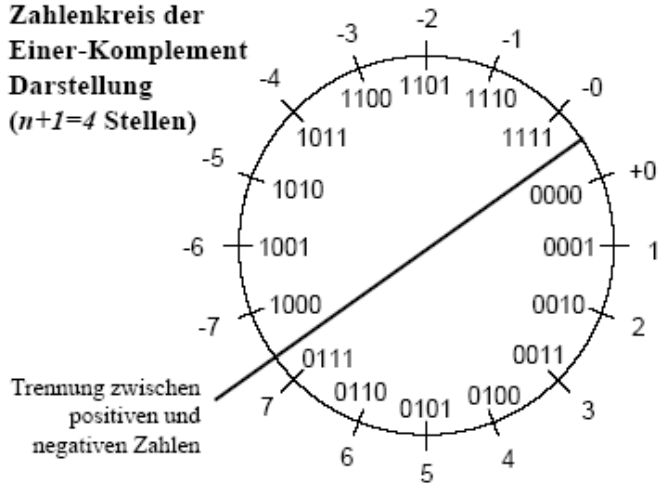
2.3.3.2 Wertebereich der Einer-Komplement Darstellung

Der Wertebereich der 1er-Komplement Darstellung ist gegenüber der Darstellung positiver Zahlen dadurch eingeschränkt, daß in der höchsten Ziffer das Vorzeichen kodiert wird. Mit der Darstellung zur Basis *b=2*, *n+1* Vorkomma- und *m* Nachkommastellen ergibt sich der Wertebereich einer Zahl *a* zu:

$$-(2^n-2^{-m}) \leq a \leq 2^n-2^{-m}.$$

Der positive und negative Wertebereich ist dabei symmetrisch

Bild 11. Zahlenkreis für 1K



Kodierung 4 Stellen	Einer-Komplement Darstellung (*b=2*)
1111	-0
1110	-1
1101	-2
1100	-3
1011	-4
1010	-5
1001	-6
1000	-7
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0

2.3.4 Integer-Zahlen mit Vz im 2er Komplement

Bei negativen Zahlen wird zunächst wie bei der 1er-Komplement Darstellung der Betrag der Zahl dargestellt und dann jede Ziffer komplementiert. Zur Vermeidung der 0 wird anschließend zum Code der Zahl eine 1 addiert.
Wie beim 1er-Komplement bestimmt bei der 2er-Komplement Darstellung die höchste Ziffer das Vorzeichen der Zahl. Bei der Darstellung zur Basis *b=2* ist die Zahl positiv, wenn die höchste Ziffer z_n (höchstes Bit) gleich 0 ist. Ist die Ziffer z_n gleich 1, ist die Zahl negativ.
Beispiel:

```
      0 0 1 1    ~    +3
      1 1 0 0
+           1
      1 1 0 1    ~    -3
```

Beispiel: Darstellung der Zahl $a=(-27)_{10}$ in der 2er-Komplement Darstellung mit 8 Stellen (Basis *b=2*):
Zunächst wird der Betrag der Zahl gebildet und zur Basis *b=2* dargestellt:

$|a|=(00011011)_2$.
Da die höchste Ziffer gleich 0 ist, lässt sich die Zahl mit den geforderten 8 Stellen darstellen. Danach werden alle Ziffern komplementiert, was bei $b=2$ einer Invertierung der Ziffern entspricht. Damit erhält man die 1er-Komplement Darstellung der Zahl:
$(11100100)_2$.
Abschließend wird zum Code eine 1 addiert und man erhält die gesuchte 2K Darstellung:
$a=(11100101)_2$.

2.3.4.1 Vorbetrachtungen zum 2er Komplement

Zu jeder Zahlenbasis (b) existieren ein
• 1er- bzw. (b-1)- Komplement und ein
• 2er- bzw. (b)- Komplement

Bildungsgesetz: Zur Bildung des 1er Komplements wird jede Ziffer einzeln von b-1 subtrahiert (Invertierung im Zahlensystem)
Beispiele:
1. Dezimalsystem: b = 10, Zahl = 3910, (b-1) = 9
9999
3910
6089

2. Dualsystem: b = 2, Zahl = 01011101, (b-1) = 1
11111111
01011101
10100010

3. Hexadezimalsystem: b= 16, Zahl = A3F9, (b-1) = 15
FFFF
-A3F9
5C06

Bildungsgesetz: 2er-Komplement = 1er-Komplement + 1
Beispiele:
1. Dezimalsystem: b = 10, Zahl = 3910, (b-1) = 9
9999
3910
6089 + 1 = **6090**

2. Dualsystem: b = 2, Zahl = 01011101, (b-1) = 1
11111111
01011101
10100010 + 1 = **10100011**

3. Hexadezimalsystem: b= 16, Zahl = 0000, (b-1) = 15
FFFF
0000
FFFF + 1 = **0000**

Merke: Die Null ist zu sich selbst das 2er-Komplement

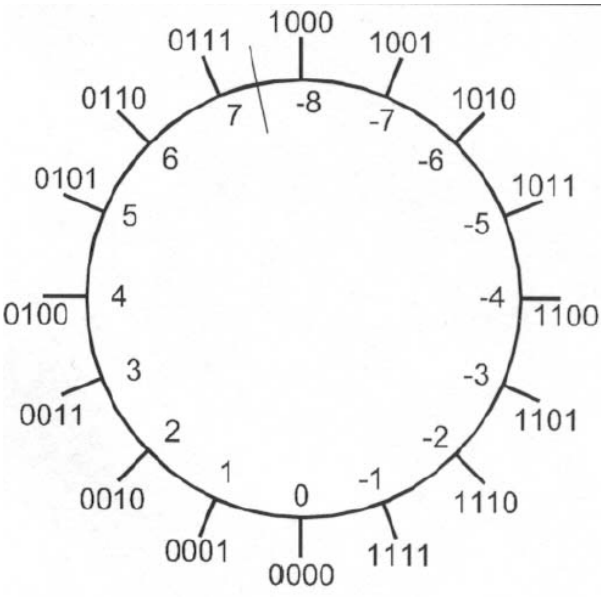
2.3.4.2 Ermittlung des Wertes einer Zweier-Komplement Darstellung

Positive Zahl: Ist die höchste Ziffer z_n der Darstellung gleich 0, handelt es sich um eine positive Zahl. Dann entspricht die 2er-Komplement Darstellung der im vorhergehenden Abschnitt behandelten Darstellung positiver Zahlen und kann direkt wie dort beschrieben ausgewertet werden.
Negative Zahl: Ist die höchste Ziffer z_n gleich 1, handelt es sich um eine negative Zahl. Dann ist zunächst die Darstellung des Betrags zu ermitteln, dazu sind alle Ziffern zu komplementieren. Der Wert des Betrags wird, da es sich um eine positive Zahl handelt, aus seiner Darstellung als positive Zahl wie bekannt ermittelt. Wegen der Verschiebung des Codes muß anschließend noch zum ermittelten Betrag der Wert 2_{-m} addiert werden.
Die 2er-Komplement Darstellung besitzt für jede Zahl genau eine Repräsentation und unterscheidet sich damit von der 1er-Komplement und Vorzeichen-Wert Darstellung.

2.3.4.3 Wertebereich der Zweier-Komplement Darstellung

Der Wertebereich der 2er-Komplement Darstellung entspricht im positiven Bereich dem der 1er-Komplement Darstellung (Abschnitt 8.3.2.3). Durch die Code-Verschiebung ist der negative Wertebereich beim 2er-Komplement leicht größer. Mit der Darstellung zur Basis $b=2$, $n+1$ Vorkomma- und m Nachkommastellen ergibt sich der Wertebereich einer Zahl a zu:
$-2^n \leq a \leq 2^n-2^{-m}$.
Durch den erweiterten negativen Bereich ist der Wertebereich bezüglich positiven und negativen Zahlen leicht asymmetrisch.

Bild 12. 2K Zahlendarstellung



Kodierung 4 Stellen	Zweier-Komplement Darstellung ($b=2$)
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0

MSB = 1 ==> negative Zahl

(MSB = Most Signifikant Bit)
Wertebereich: -2^{n-1} bis $+2^{n-1}-1$
Beispiele:
4-Bit: -8 bis +7
8-Bit: -128 bis +127
16-Bit: -32768 bis + 32767
Vorteile:
• Vorzeichenbit muß bei Addition und Subtraktion nicht gesondert ausgewertet werden
• Es existiert nur eine Null

Beispiele:

6+(-1)=5	4-6=4+(-6)=-2
0110	0100
+1111	+1010
101	1110

2.3.4.4 Übertrag und Überlauf

Übertrag (Carry, C, CY)
Ein Übertrag entsteht, wenn bei Operationen mit nur positiven Zahlen der Wertebereich überschritten wird (Vergleiche Zahlenkreis für Integer-Zahlen ohne Vorzeichen)

Überlauf (Overflow, OV)
Ein Überlauf entsteht, wenn bei Operationen von 2er-Komplement-Zahlen der Wertebereich überschritten wird (Vergleiche Zahlenkreis für Integer-Zahlen mit Vorzeichen im 2er-Komplement)

Das Carry-Bit ist das sogenannte Carry-Out der höchsten Bitposition
Beispiel:

```
   1011 0011 0110 1000
+  1101 1110 1100 1101
  11111 1111 1001 000
1000 0010 0011 0101
```

Oder in Hex

```
    B368
+   CECD
    1111
    8235
```

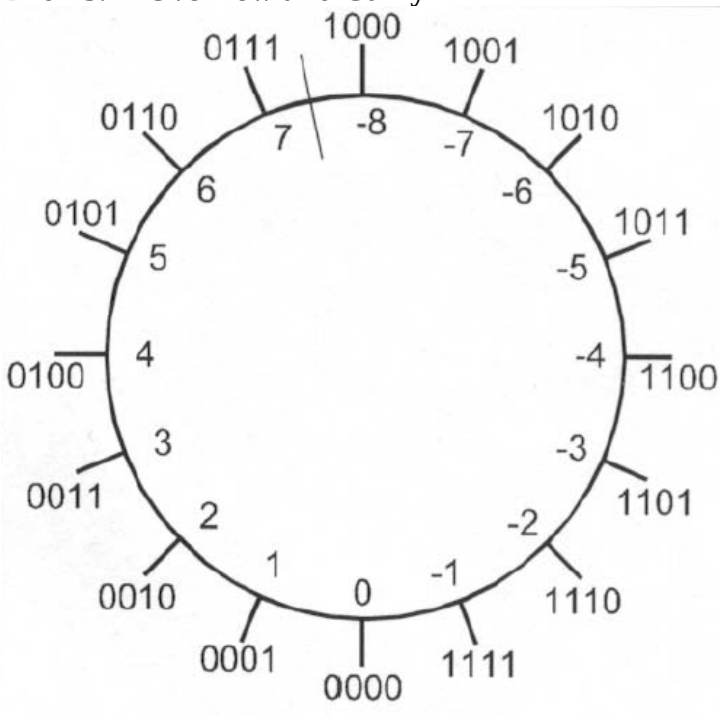
2.3.4.5 Bestimmung des Überlauf-Bits (Overflow)

Das Overflow-Bit ist nur für 2er-Komplement-Zahlen von Bedeutung. MSB gibt bei 2er-Komplement-Zahlen das Vorzeichen an.
Regel: Ein Overflow liegt vor, wenn beide Summanden ein gleiches Vorzeichen haben und das Ergebnis ein anderes
(Vergleiche Zahlenkreis!)

Summand A	Summand B	Summe	Das Ergebnis ist falsch (OV=1)	Das Ergebnis ist richtig (OV=0)	C_{in}	C_{out}
+	+	+		✓	0	0
+	+	-	✓		1	0
+	-	+		✓	1	1
+	-	-		✓	0	0
-	-	-		✓	1	1
-	-	+	✓		0	1

Reduzierung des hardwaretechnischen Aufwands bei Addierern:
Carry-In und Carry-Out werden betrachtet. **OV = Cin XOR Cout**

Bild 13. Overflow und Carry



Beispiele:

-6	1010	-6	1010
+(-7)	1001	+(-1)	1111
-13	10011	-7	11001
	OV=1	OV=0	
	Carry=1	Carry=1	

6	0110	5	0101
+5	0101	+2	0010
11	1011	7	0111
OV=1		OV=0	

2.4 Binär-kodierte Dezimalzahlen (BCD)

Bei Taschenrechnern oder früheren Tischrechnern blieb man wegen der Ungenauigkeiten, die sich durch die Umwandlung von Zahlen in eine andere Basisdarstellung ergeben, bei der

Rechnung mit dezimalen Zahlen. Da Rechner intern jedoch nur die Binärdarstellung kennen, müssen die Dezimalziffern als Binärzahlen dargestellt werden. Dies führte zur binär-kodierten Dezimaldarstellung (BCD – Binary Coded Decimal).
Vorteil: Man kann im Dezimalsystem rechnen
--> Einsparungen beim Einlesen von Daten
--> Einsparungen bei der Ausgabe von Daten
Nachteil: Erhöhter Rechenaufwand

2.4.1 Wertebereich der BCD-Darstellung

Durch die BCD-Darstellung bleiben Kodierungen ungenutzt, somit ist der Wertebereich geringer als bei der gleichen Anzahl Bits der Binärdarstellung. Der Wertebereich der BCD-Darstellung entspricht direkt dem Wertebereich der Dezimaldarstellung mit gleicher Stellenzahl. Bei *d* zu kodierenden Dezimalstellen (und damit *4´d* notwendigen Binärstellen zur Speicherung der Ziffern) ergibt sich der Wertebereich der BCD-Darstellung zu:

$$0 \le a \le 10^d - 1$$

Bild 14. BCD Darstellung

numerisches Alphabet	Tetradendarstellung			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
.	1	0	1	0
.	1	0	1	1
Pseudo- .	1	1	0	0
tetraden .	1	1	0	1
.	1	1	1	0
.	1	1	1	1

2.4.2 BCD-Addition

BCD-Addition wird wie bei der üblichen Dualaddition für jede Tetrade durchgeführt
Sonderfälle: Übertrag in die jeweils nächst höhere Dezimalpotenz
1. Natürlicher Tetradenübertrag. Übergelaufene Tetrade muß durch Addition von 6 korrigiert werden (modulo-16 Darstellung)

2. Ergebnis einer Stellen- oder auch einer Korrekturaddition liegt in der Pseudotetrade (10-15 bzw. A-F). Aufaddieren einer 6 erzwingt Überlauf und modulo-16 Darstellung

Beispiel:

mit Übertrag		mit Pseudotetrade		
8_{10}	1000	7_{10}	0111	
+ 9_{10}	1001	+ 6_{10}	0110	
= 17_{10}	0001 0001	= 13_{10}	1101	
+	0110	+	0110	<-- Korrektur (+6)
	0001 0111		0001 0011	<-- Endergebnis

Prozessoren haben of einen Decimal Adjust Befehl für diese Situationen

2.5 Gebrochene Zahlen

Übliches Format ist die Darstellung als rein gebrochene Zahl, also 0,xxxxx. Das höchste Bit gibt dann das Vorzeichen an, alle weiteren die Stellen nach dem Komma.

Beispiele:
Wert $(0{,}134_{10}) = 1 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3}$

Wert $(0{,}145_{8}) = 1 \times 8^{-1} + 4 \times 8^{-2} + 5 \times 8^{-3}$
$= 1/8 + 4/64 + 5/512$
$= 0{,}125 + 0{,}0625 + 0{,}009765625$
$= 0{,}197265625$

Übungsbeispiel: Gesucht ist der Wert der Zahl *a*, die wie folgt in 2er-Komplement Darstellung mit *w=8* Stellen kodiert ist:
$a=(1001{,}1100)_2$. Es gilt also *n=3* und *m=4*.
Da die höchste Ziffer gleich 1 ist, handelt es sich um eine negative Zahl.
Komplementieren aller Ziffern führt zu:
$(0110{,}0011)_2$. Umrechnung dieser Darstellung ins Dezimalsystem ergibt:
$(0110{,}0011)_2=(6{,}1875)_{10}$.
Zur Ermittlung des Betrags von *a* muß noch der Wert 2^{-m} addiert werden:
$|a|=(6{,}1875)_{10}+2^{-4}=(6{,}25)_{10}$
Der Wert der gesuchten Zahl *a* beträgt somit $a=(-6{,}25)_{10}$.

2.5.1 Umwandeln einer gebrochenen Zahl in ein anderes Zahlensystem

2.5.1.1 Umrechnung einer Dezimalzahl in ein anderes Zahlensystem

Dezimalzahl a in ein anderes Zahlensystem der Basis B.

Rechenvorschrift: a / B = a0, Rest y0
a0 / B = a1, Rest y1

usw. bis einschließlich ai = 0
Ergebnis unabhängig vom Zahlensystem, in dem die Rechnung ausgeführt wird

Beispiele:
1) Dezimalzahl 545 in eine Dualzahl umwandeln. Basis des Dualsystems: B = 2.
545 : 2 = 272, Rest 1, zugehörige Wertigkeit: $2^0 = 1$
272 : 2 = 136, Rest 0, " " $2^1 = 2$
136 : 2 = 68, Rest 0, " " $2^2 = 4$
68 : 2 = 34, Rest 0, " " $2^3 = 8$
34 : 2 = 17, Rest 0, " " $2^4 = 16$
17 : 2 = 8, Rest 1, " " $2^5 = 32$
8 : 2 = 4, Rest 0, " " $2^6 = 64$
4 : 2 = 2, Rest 0, " " $2^7 = 128$
2 : 2 = 1, Rest 0, " " $2^8 = 256$
1 : 2 = 0, Rest 1, " " $2^9 = 512$
Ergebnis: $1000100001_2 = 545$

2) Dezimalzahl 545 in Hexadezimalzahl umwandeln. Basis des Hexadezimalsystems: B = 16.
545 : 16 = 34, Rest 1, zugehörige Wertigkeit: $160 = 1$
34 : 16 = 2, Rest 2, " " $161 = 16$
2 : 16 = 0, Rest 2, " " $162 = 256$
Ergebnis: $221_{16} = 545$

Warum funktioniert der im Beispiel dargestellte Algorithmus? Die Untersuchung der schrittweisen Division einer positiven ganzen Zahl *a* durch eine positive ganze Zahl $b_2 \geq 2$ gibt darüber Aufschluss. Es wird vorausgesetzt, daß beide Zahlen in einer Darstellung bezüglich Basis b_1 vorliegen. Alle Rechnungen erfolgen somit bezüglich dieser Basis b_1.
Dividiert man die Zahl *a* durch *b2* so erhält man bei Ganzzahlrechnung einen Quotienten *d0* und einen Rest *r0*. Dividiert man den Quotienten *d0* ebenfalls durch *b2* erhält man einen weiteren Quotienten *d1* und den Rest *r1*. Führt man dieses Vorgehen weiter, wird schließlich ein Quotient zu 0 (*dn=0*) und man erhält einen letzen Rest *rn*:

$$
\begin{aligned}
a / b_2 &= d_0 + r_0 / b_2 \\
d_0 / b_2 &= d_1 + r_1 / b_2 \\
d_1 / b_2 &= d_2 + r_2 / b_2 \\
&\cdots \\
d_{n-2} / b_2 &= d_{n-1} + r_{n-1} / b_2 \\
d_{n-1} / b_2 &= 0 + r_n / b_2
\end{aligned}
$$

Löst man die Gleichungen von unten her auf und setzt die Ergebnisse jeweils in die darüberliegende Gleichung ein, so erhält man schließlich wieder eine Gleichung für *a*:

$$d_{n-1} = r_n$$
$$d_{n-2} = r_n \times b_2{}^1 + r_{n-1}$$
$$\dots$$
$$d_1 = r_n \times b_2{}^{n-2} + r_{n-1} \times b_2{}^{n-3} + \dots + r_2$$
$$d_0 = r_n \times b_2{}^{n-1} + r_{n-1} \times b_2{}^{n-2} + \dots + r_2 \times b_2{}^1 + r_1$$
$$a = r_n \times b_2{}^n + r_{n-1} \times b_2{}^{n-1} + \dots + r_2 \times b_2{}^2 + r_1 \times b_2{}^1 + r_0 = \sum_{i=0}^{n} r_i \times b_2{}^i$$

Die unterste Gleichung für die ganze Zahl *a* entspricht ihrer gewichteten Zerlegung bezüglich der Basis *b2*. Dabei entspricht ein Rest *ri*, wie man offensichtlich sieht, direkt der Ziffer *zi* in der Darstellung von *a* zur Basis *b2*. Damit gilt (wie das Beispiel schon zeigte) $a=(r_n r_{n-1} \dots r_2 r_1 r_0)b_2$. Die Rechnung erfolgt nach Voraussetzung in der Darstellung zur Basis *b1*, in der *a* und *b2* als Werte vorliegen.

2.5.1.2 Hornerschema

Das Horner-Schema ist eine effiziente Methode, Zahlen aus einem Stellenwertsystem zur Basis Y in das 10er System umzuwandeln. Dazu wird die sukzesive Multiplikation wie folgt verwendet:

$(x_N x_{N-1} \dots x_1 x_0) = (((x_N Y + x_{N-1})Y + x_{N-2})Y + \dots + x_1)Y + x_0$

Man wandle damit folgende Zahlen in das 10er System um: $121021_{(3)}$ und $1E3C_{(15)}$

2.5.1.3 Umwandlung der Nachkommastellen

Regel:
a x B = v0 , a0
a0 x B = v1 , a1
usw. bis einschließlich ai = 0
Beispiel: Transformation von 0,2812510 ins Dualsystem
Ausgangszahl 0,28125
0,28125 x 2 = 0,5625
0,5625 x 2 = 1,125
0,125 x 2 = 0,250
0,250 x 2 = 0,5
0,5 x 2 = 1,0
Ergebnis: $0{,}01001_2$
Problem: Bereits einfache Dezimalbrüche können zu einer erheblichen Stellenanzahl führen (Beispiel 0,3)

Beispiel: Gegeben sei die Zahl $(0{,}82421875)_{10}$ in ihrer Darstellung bezüglich Basis $b_1=10$. Gesucht ist die Darstellung der Zahl im Dualsystem (Basis $b_2=2$). Die Umrechnung soll im Dezimalsystem (Basis $b_1=10$) durchgeführt werden:

0,82421875 * 2 = 1,6484375 = 1 + 0,6484375
0,6484375 * 2 = 1,296875 = 1 + 0,296875
0,296875 * 2 = 0,59375 = 0 + 0,59375
0,59375 * 2 = 1,1875 = 1 + 0,1875
0,1875 * 2 = 0,375 = 0 + 0,375
0,375 * 2 = 0,75 = 0 + 0,75
0,75 * 2 = 1,5 = 1 + 0,5
0,5 * 2 = 1,0 = 1 + 0,0

Als Übung jetzt wieder Rückwandlung mit Hornerschema.

2.5.2 Addierer

In allen Zahlensystemen addieren wir Zahlen ziffernweise, das heißt stellenweise.
Betrachten wir deshalb zunächst nur die Addition zweier einstelliger Zahlen:

X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Die Summe S kann als X (xor) Y berechnet werden, d.h. als
$S=\neg XY + X\neg Y$
Der Übertrag C
$C= XY$
Ein solches Additionsmodul nennen wir Halbaddierer.

Übungsbeispiel:
Skizzieren Sie ein Schaltwerk, das einen HA realisiert.

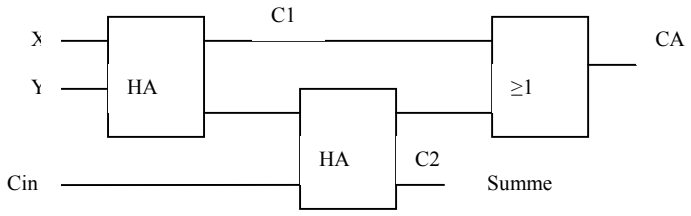
Addieren wir mehrstelligen Zahlen, können wir auch einen Übertragseingang Carry-in erwarten. Wir beschreiben nun eine beliebige Stelle i der beiden Zahlen.

$\overline{Z1_i}$	$\overline{Z2_i}$	c_{i-1}	c_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Wir nennen dies einen Volladdierer.
$S_i = x_i$ xor y_i xor c_{i-1}

$C_i = x_iy_i + yC_{i-1} + xC_{i-1}$
$C_i = G_i + P_iC_{i-1}$ P=propagation; G=generation

Er kann aus Gatterwerken direkt oder aus zwei HAs und einem Oder-Gatter generiert werden.



Übungsbeispiel:
Zeichnen Sie die Struktur eines Volladierers für N-stellige Zahlen.

2.5.3 Addition/Subtraktion positiver Festkommazahlen

Zur Addition zweier positiver Festkommazahlen müssen beide Zahlen aus der gleichen Anzahl von Vor- und Nachkommastellen bestehen. Ist dies nicht der Fall, müssen bei der Zahl mit geringerer Anzahl von Vor- oder Nachkommastellen 0-Ziffern ergänzt werden.
Die Addition zweier Zahlen *Z1* und *Z2* zur Summe *S* beginnt mit der niedrigsten Ziffer. Es werden die Ziffern der Zahlen in aufsteigender Reihenfolge unter Berücksichtigung des Übertrags addiert:
Addition der niedrigsten Stelle:
Die Addition der niedrigsten Ziffern $Z1_{-m}$ und $Z2_{-m}$ zur Bestimmung des Summenbits S_{-m} und des Übertrags c_{-m} kann mit dem vorgestellten Halbaddierer erfolgen. Ein Übertrag aus einer vorhergehenden Stelle liegt noch nicht vor:
$\text{HalbAddierer}(Z1_{-m}, Z2_{-m}, S_{-m}, c_{-m})$

Addition der weiteren Stellen:
Bei der Addition der weiteren Ziffern $Z1_i$ und $Z2_i$ ($-m+1 < i \leq n$) muß der Übertrag c_{i-1} aus der vorhergehenden Stelle bei der Bestimmung des Summenbits S_i und des Übertrags c_i berücksichtigt werden. Es müssen also die Summe und der Übertrag aus drei Bits ermittelt werden:

$$\begin{array}{rr} & Z1_i \\ + & Z2_i \\ + & c_{i-1} \\ \hline c_i & S_i \end{array}$$

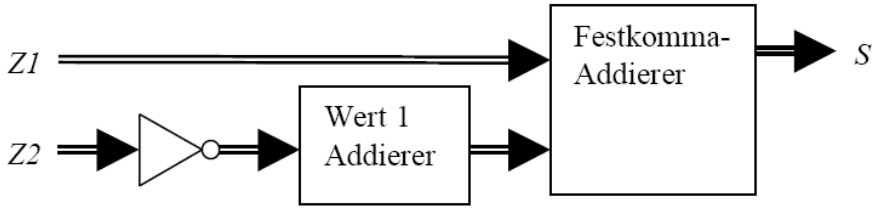
Die Verknüpfungstabelle zur Addition zweier Ziffern plus Übertrag zeigt, daß das Ergebnis mit dem Summen und Übertragbit eindeutig dargestellt werden kann:

2.5.3.1 Addition von Zweier-Komplement Zahlen

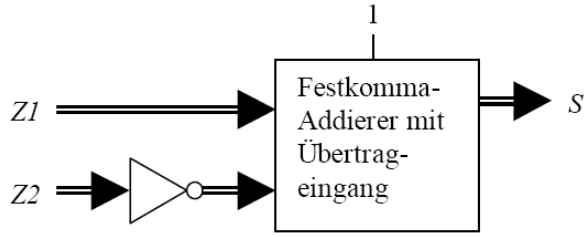
Bei der Addition von Zweierkomplement-Zahlen ist der Übergang zwischen den positiven und den negativen Zahlen nicht mehr kritisch, da die Null nur noch einmal vorhanden ist. Somit wird bei der Arithmetik kein zusätzlicher Schritt mehr benötigt.

2.5.3.2 Subtraktion von Zweier-Komplement Festkommazahlen

Die Subtraktion von Festkommazahlen kann auf die Addition zurückgeführt werden, wenn vorher der zu subtrahierende Operand negiert wird. Dazu werden, wie besprochen, z.B. alle Bits der Zahl invertiert und eine 1 aufaddiert.

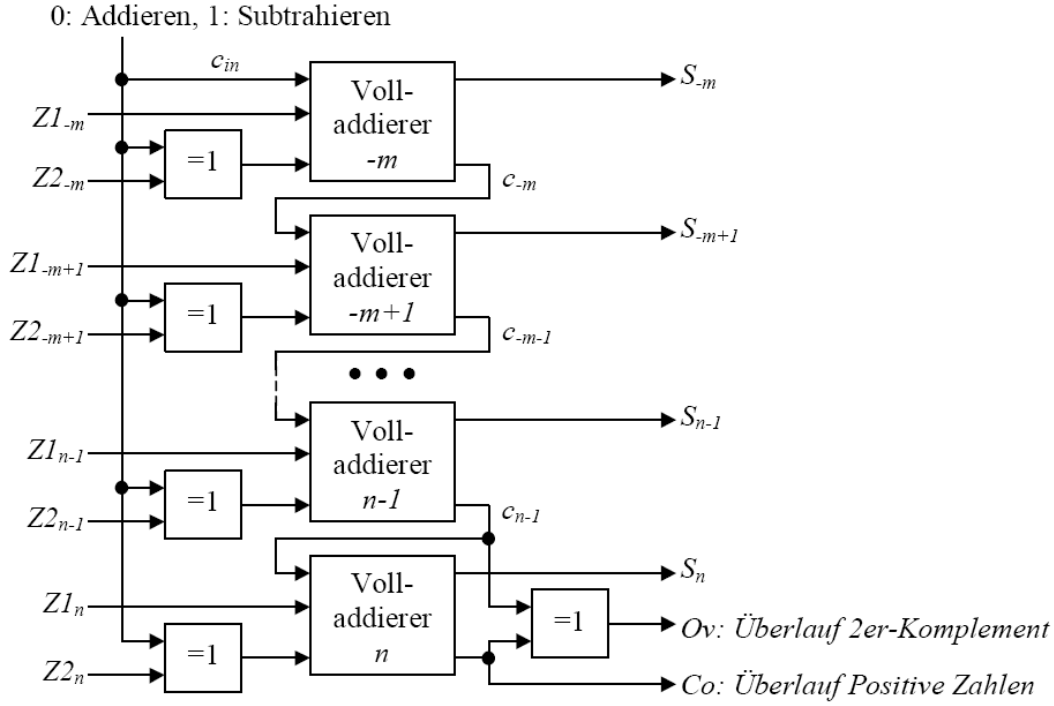


Die Schaltung des Subtrahierers lässt sich vereinfachen, wenn man das Addierwerk aus obiger Abbildung leicht modifiziert. Tauscht man in der Additionsstufe der untersten Bits (Index *-m*) den Halbaddierer gegen einen Volladdierer, so entsteht dort genau an der Stelle ein Übertrageingang, mit dem die 1 zur 2er-Komplement- Wandlung addiert werden kann.



Die Addition und Subtraktion von Festkommazahlen unterscheiden sich also, bis auf die 'Normalisierung' am Anfang, nicht von der Integer-Addition.

Bild 15. Addition und Subtraktion von Festkommazahlen



2.5.3.3 Beschleunigung der Addition/Subtraktion durch ‚carry-look-ahead'

Ein Problem der vorgestellten Addierer und Subtrahierer ist der sequentielle Pfad der Überträge vom niedrigsten zum höchsten Bit, der zu großen Verzögerungszeiten führt. Abhilfe schafft eine beschleunigte Berechnung der Überträge. Dieses Vorgehen wird im Englischen als „*carry look-ahead*" bezeichnet.
Das Prinzip ist wie folgt:

Das Netz zur Erzeugung des Übertrags kann sysstematisch von rechts substituiert werden.

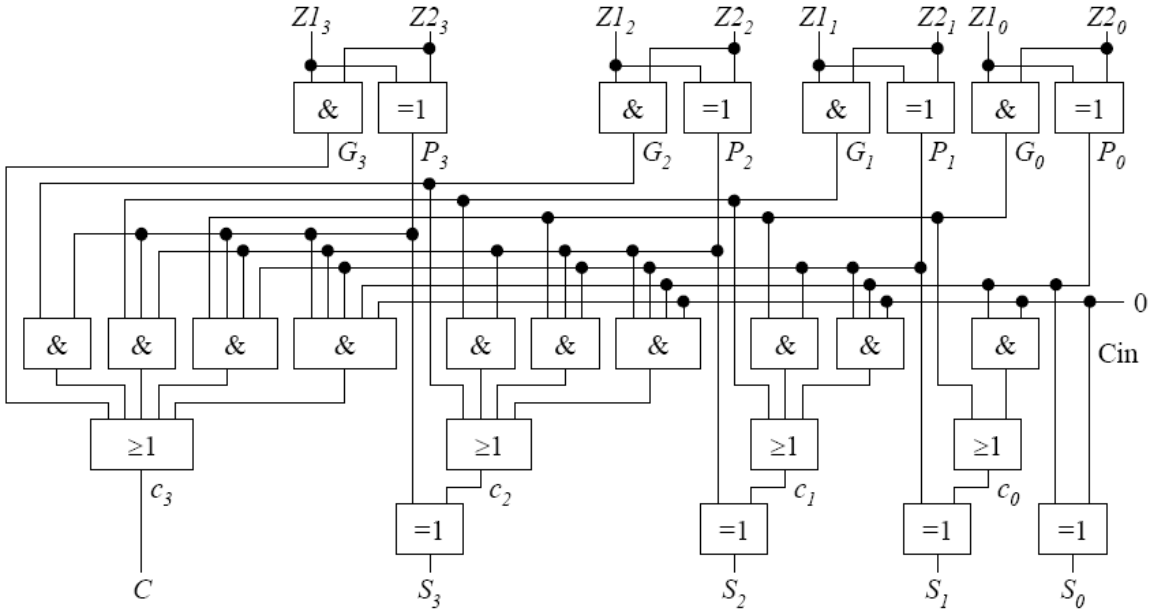
$C_1 = G_1 + P_1C_0$

$C_2 = G_2 + P_2\ G_1 + P_2\ P_1C_0$

.....

$C_n = G_n + P_n\ G_{n-1} + P_n\ P_{n-1}\ G_{n-2} + \ldots.\ P_n\ P_{n-1\ldots}\ P_2P_1C_0$ mit $G_i = x_iy_i$ und $P_i = x_{i+}y_i$

Bild 16. Carry look ahead



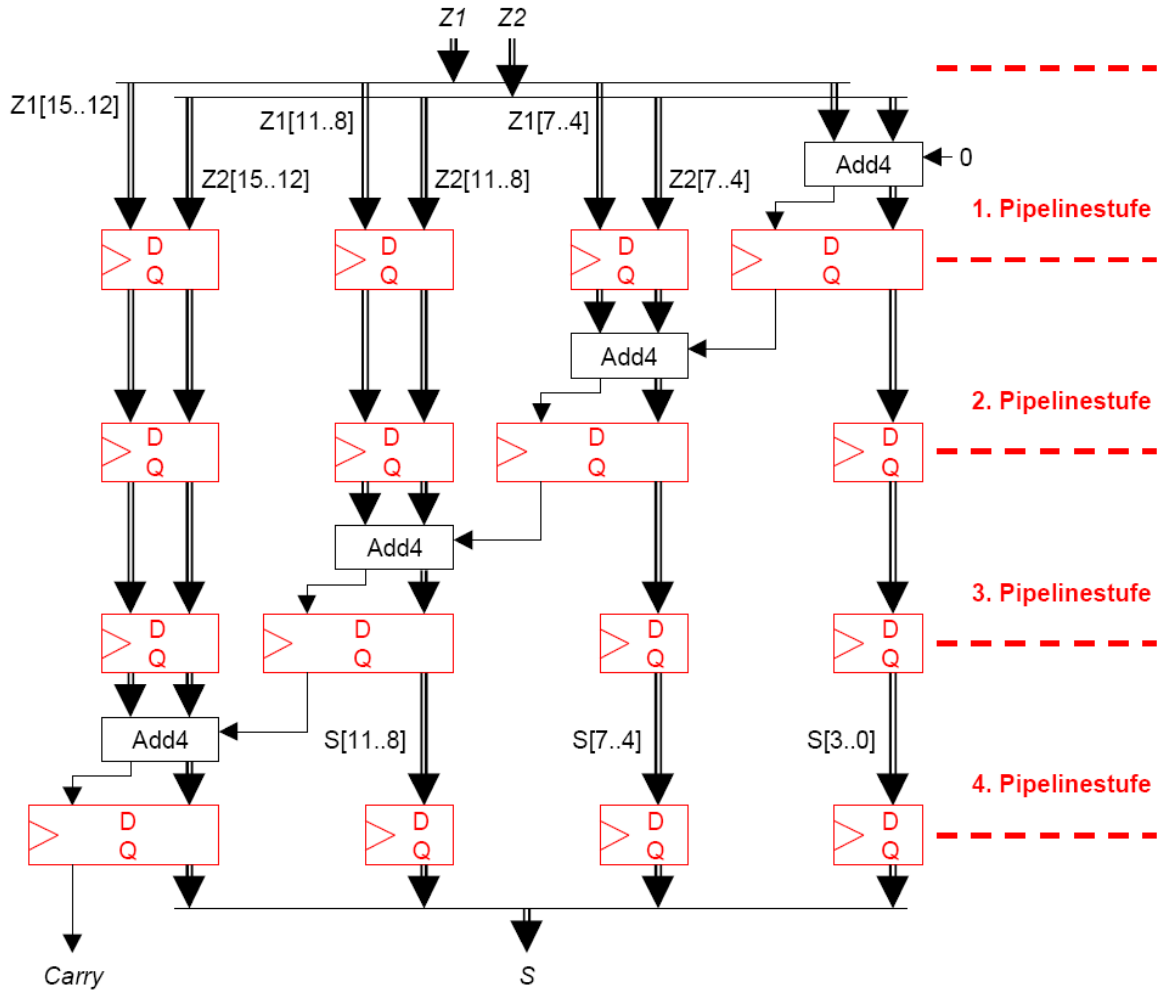
Weitere bekannte Strukturen für die Addition mehrerer Summanden sind ’wallace-tree Addierer’.

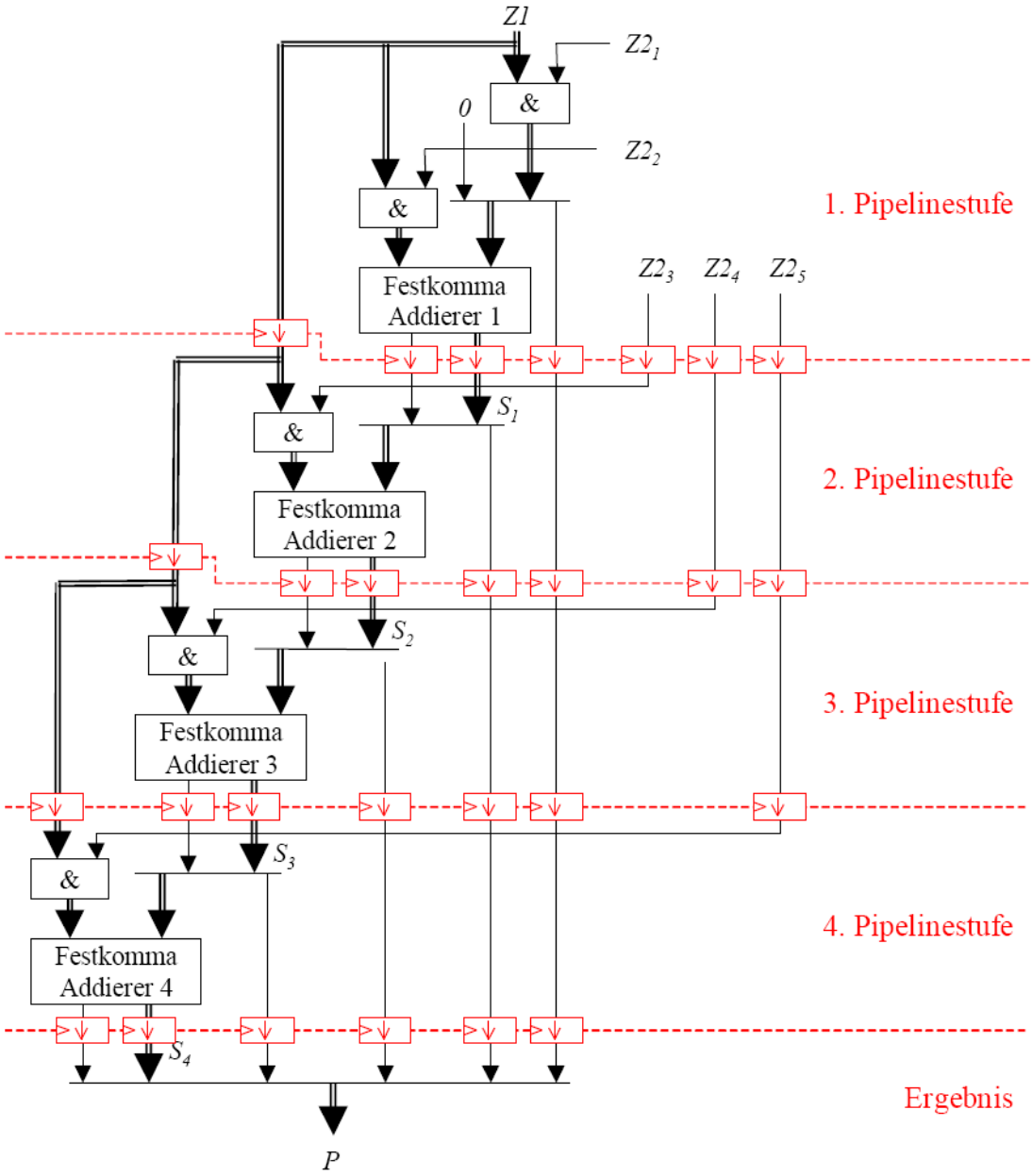
2.5.3.4 Beschleunigung der Arithmetik durch Pipelining

Arithmetische Verarbeitung kann durch Pipelining (Fließbandverarbeitung) dann beschleunigt werden, wenn mehrere Aufgaben gleicher Art im gleichen Rechenwerk direkt hintereinander berechnet werden müssen. Z.B. direkt aufeinanderfolgende Addition von Zahlenpaaren.
Beim Pipelining wird eine Operation in Teilschritte zerlegt, die in aufeinanderfolgenden Taktzyklen bearbeitet werden. Nach jedem Bearbeitungsschritt werden die Zwischenergebnisse eines Teilschritts in Register übernommen und damit stabil der nächsten Verarbeitungsstufe zugeführt. Die Bearbeitung einer einzelnen Operation wird nicht beschleunigt. Es kann zwar der Takt der Verarbeitung erhöht werden, für die Verarbeitung sind jedoch mehrere Taktzyklen nötig, bis alle Pipelinestufen durchlaufen sind. Die Anzahl dieser Verarbeitungszyklen bezeichnet man als „Latenz“ der Pipeline. Sobald die erste Teiloperation in der ersten Pipelinestufe beendet ist, können die Operanden einer weiteren Operation in die Pipeline eingespeist werden. Damit werden mehrere Operationen versetzt parallel in der Pipeline verarbeitet.
Beispiel: Addition zweier positiver 16-Bit Binärzahlen *Z1* und *Z2* Als Ergebnis wird die Summe *S* und der Überlauf *Carry* ermittelt. Zur Verfügung stehen 4-Bit Addierer. Die

Taktrate soll auf die Durchlaufzeit der Signale durch einen einzelnen 4-Bit Addierer abgestimmt werden.
In untenstehender Abbildung zeigt das Blockschaltbild eines Pipeline-Addierers. In der ersten Stufe werden die untersten 4 Bit der beiden Zahlen addiert. Da noch kein Übertrag (Carry) einer niederwertigen Stufe vorliegt, wird der Wert 0 eingegeben. Die errechneten 4 Summenbits werden über Register in den nachfolgenden Pipelinestufen verzögert, so daß sie synchron zu den anderen Summenbits am Ausgang ausgegeben werden. Der dabei anfallende Übertrag wird über ein Register zur nächsten Pipelinestufe weitergereicht.
In der zweiten Pipelinestufe werden die Bits 4 bis 7 der beiden Zahlen unter Berücksichtigung des Übertrags aus der ersten Stufe addiert. Die Operanden werden über ein Register verzögert an den Addierer gegeben, die Operanden liegen somit synchron mit dem Übertrag aus der ersten Pipelinestufe am Addierer an. Die Summenbits werden verzögert weiter zum Ausgang gereicht und der Übertrag der nächsten Pipelinestufe zugeführt.
Entsprechend wird in den Pipelinestufen 3 und 4 verfahren. Schrittweise wird somit in den 4 Pipelinestufen die gesamte Summe berechnet. Die Latenz der Pipeline beträgt also 4.

Bild 17. Pipelining





2.5.3.5 Zyklischer Einsatz eines Pipeline Addierers

Wenn man bei oben gezeigtem Bild eines Addierers mit Pipeline-Registern am Eingang und Ausgang den Ausgang auf einen der Eingänge rückkoppelt und dann den freien Eingang mit jeweils einem neuen Summanden pro Takt belegt, so kann man Summanden fortlaufend addieren.

2.5.3.6 Beschleunigung der Arithmetik durch look up Tables (LUT)

Wenn man einen Speicherbaustein (z.B. ROM oder PROM, siehe in folgenden Kapiteln) dazu verwenden will, z.B. eine Addition zu tabellieren, so muß man lediglich die Adresseingänge als Summandeneingänge interpretieren und in die adressierten Speicherplätze entsprechende Summenergebnisse hineinprogrammieren.

2.5.4 Multiplikation

2.5.4.1 Multiplikation einer Zahl mit einer gewichteten Ziffer

Eine Zahl *Z1* kann mit einer gewichteten Ziffer $Z2_i \cdot b^i$ multipliziert werden, indem man die Zahl zunächst direkt mit der Ziffer multipliziert und die Ziffern der Zahl um *i* Positionen (entsprechend dem Exponenten *i*) nach links verschiebt. Jedes Schieben der Ziffern um eine Position nach links bedeutet eine Multiplikation mit *b*.

$Z1 \cdot (Z2_i \cdot b^i)$ = *Ziffern_links_verschieben((Z1·Z2$_i$), i)*

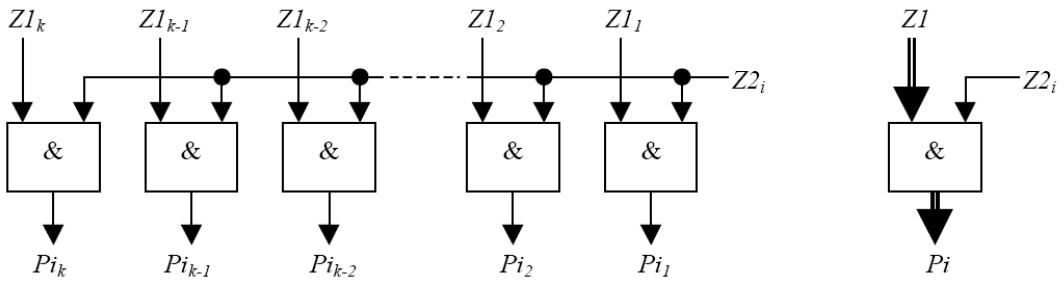
Beispiel: $1234 \cdot (2 \cdot 10^5)$ = *Ziffern_links_verschieben((2·1234), 5) = 246800000*

Besonders einfach wird diese Multiplikation im Dualsystem (*b=2*). Dort ist das Ergebnis der Multiplikation einer Zahl *Z1* mit einer Ziffer $Z2_i$ entweder 0 oder die Zahl selber. Somit wird zur Ermittlung des Produkts zunächst die Zahl ziffernweise mit der zweiten Ziffer UND-verknüpft und anschließend nach links verschoben.

Beispiel: $(10101010)_2 \cdot (1 \cdot 2^5)$ = *Ziffern_links_verschieben*
(((1^1)(0^1)(1^1)(0^1)(1^1)(0^1)(1^1)(0^1))$_2$, 5)
$=(1010101000000)_2$

Die Schaltung des zugehörigen Multiplizierers (ohne Schieben) benötigt nur UND-Gatter und ist in untenstehender Abbildung dargestellt. Auf der linken Seite der Abbildung ist die Detailschaltung gezeigt, auf der rechten Seite eine kompakte Darstellung.

Bild 18. Multiplizierer

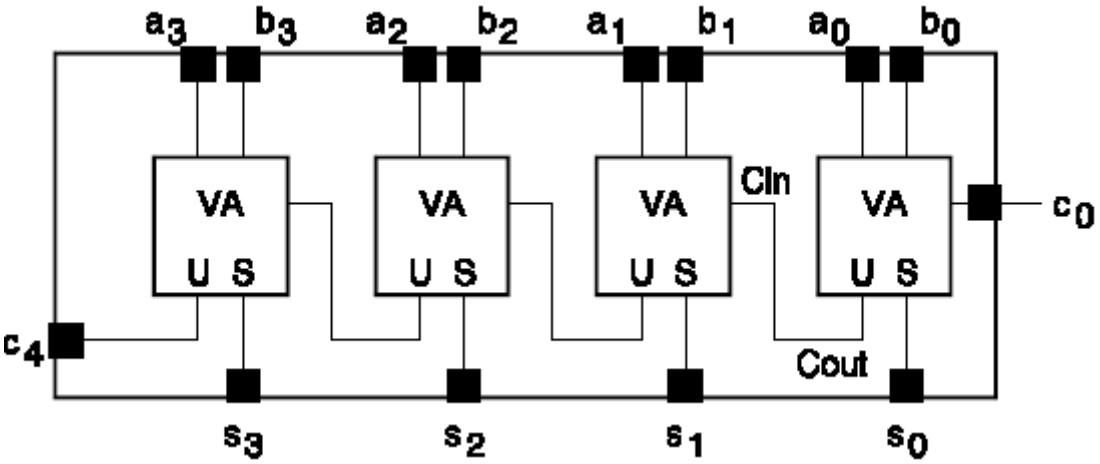


2.5.4.2 Multiplizierer

Ein Rechenwerk kann auch ein eigenes Multiplizierwerk enthalten, das z.B. Teilprodukte bildet und diese aufaddiert.
Beispiel: Multiplikand Multiplikator
1 0 1 1 (11_{10}) 0 1 0 1 (5_{10})
Teilprodukte:
1. Teilprodukt 1011 x 2^0
2. Teilprodukt 0000 x 2^1
3. Teilprodukt 1011 x 2^2
4. Teilprodukt 0000 x 2^3

Diese Teilprodukte sind zu addieren.
Im Multiplizierwerk geschieht dies, wenn wir die Teilprodukte mit Mi 2^{i-1} bezeichnen, gemäß:
$((M12^4 + 0)/2 + M2\ 2^4)/2 + M32^4)/2 + M42^4)/2$,
so daß i.W. ein Addierwerk und zwei Register ausreichen. Die Division durch 2 erfolgt implizit durch Shiften und die Multiplikation mit 2^4 erfolgt durch Positionierung des Multiplikanden.

Bild 19. Multiplizierer



Nach 4 Takten befindet sich das Ergebnis im Akkumulator-Multiplikator-Schieberegister.
Die Steuerung veranlaßt nur dann den Addierer zur Addition,wenn das niedrig-wertigste Bit dieses Schieberegisters eine 1 enthält.
Schritt 1: AKKU + (Multiplikand x 1) und Shiften
Inhalt des Schieberegisters nach Addition 01011 0101 nach Shiften 00101 1010
Schritt 2: AKKU + (Multiplikand x 0) und Shiften
Inhalt des Schieberegisters nach Addition 00101 1010 nach Shiften 00010 1101
Schritt 3: AKKU + (Multiplikand x 1) und Shiften
Inhalt des Schieberegisters nach Addition 01101 1101 nach Shiften 00110 1110
Schritt 4: AKKU + (Multiplikand x 0) und Shiften
Inhalt des Schieberegisters nach Addition 00110 1110 nach Shiften 00011 0111 (55(10))

2.5.4.3 Multiplikation nach der Schulmethode

Die Multiplikation zweier Zahlen *Z1* (Multiplikand) und *Z2* (Multiplikator) zu einem Produkt *P* kann analog zur traditionellen Schulmethode erfolgen. Dazu zerlegt man den Multiplikator *Z2* in seine einzelnen Ziffern, multipliziert den Multiplikanden *Z1* mit jeder Ziffer $Z2_i$, verschiebt die Zwischenprodukte P_i gemäß Ihrer Gewichte und summiert die Zwischenprodukte auf zum Gesamtprodukt.
Eventuelle Vorzeichen werden getrennt betrachtet und nur die Beträge in positiver Darstellung werden multipliziert.

Bild 20. Schulmethode
Beispiel: $P = Z1 \times Z2 = (11001)_2 \times (01101)_2$

											Bemerkungen
						1	1	0	0	1	$1 \times Z1 \times 2^0 = P_0$
+					0	0	0	0	0		$0 \times Z1 \times 2^1 = P_1$
+				1	1	0	0	1			$1 \times Z1 \times 2^2 = P_2$
+			1	1	0	0	1				$1 \times Z1 \times 2^3 = P_3$
+		0	0	0	0	0					$0 \times Z1 \times 2^4 = P_4$
	0	1	0	1	0	0	0	1	0	1	Produkt *P*

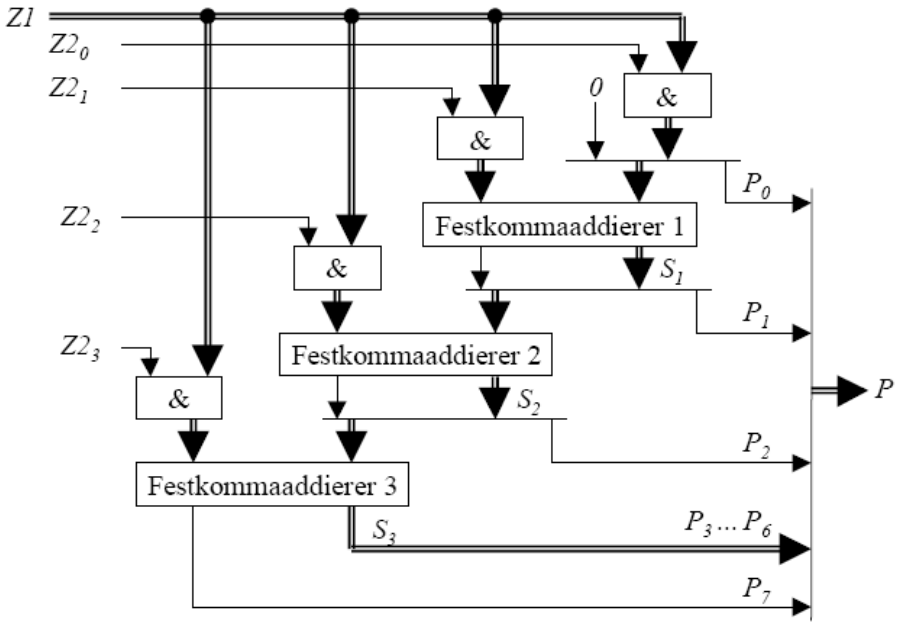
Im Rechner macht die Summierung aller Zwischenprodukte P_i in einem Schritt zum Gesamtprodukt *P*, wie wir es von der Schule gewohnt sind, hohen schaltungstechnischen Aufwand. Es bietet sich an, Zwischenergebnisse sukzessive aufzuakkumulieren.

Beispiel: $P = Z1 \times Z2 = (11001)_2 \times (01101)_2$ mit schrittweise Aufakkumulieren des Produktes

											Bemerkungen
						1	1	0	0	1	$1 \times Z1 \times 2^0 = P_0$
+					0	0	0	0	0		$0 \times Z1 \times 2^1 = P_1$
				0	0	1	1	0	0	1	$S_1 = P_0 + P_1$
+				1	1	0	0	1			$1 \times Z1 \times 2^2 = P_2$
			0	1	1	1	1	1	0	1	$S_2 = P_0 + P_1 + P_2$
+			1	1	0	0	1				$1 \times Z1 \times 2^3 = P_3$
		1	0	1	0	0	0	1	0	1	$S_3 = P_0 + P_1 + P_2 + P_3$
+		0	0	0	0	0					$0 \times Z1 \times 2^4 = P_4$
	0	1	0	1	0	0	0	1	0	1	Produkt P

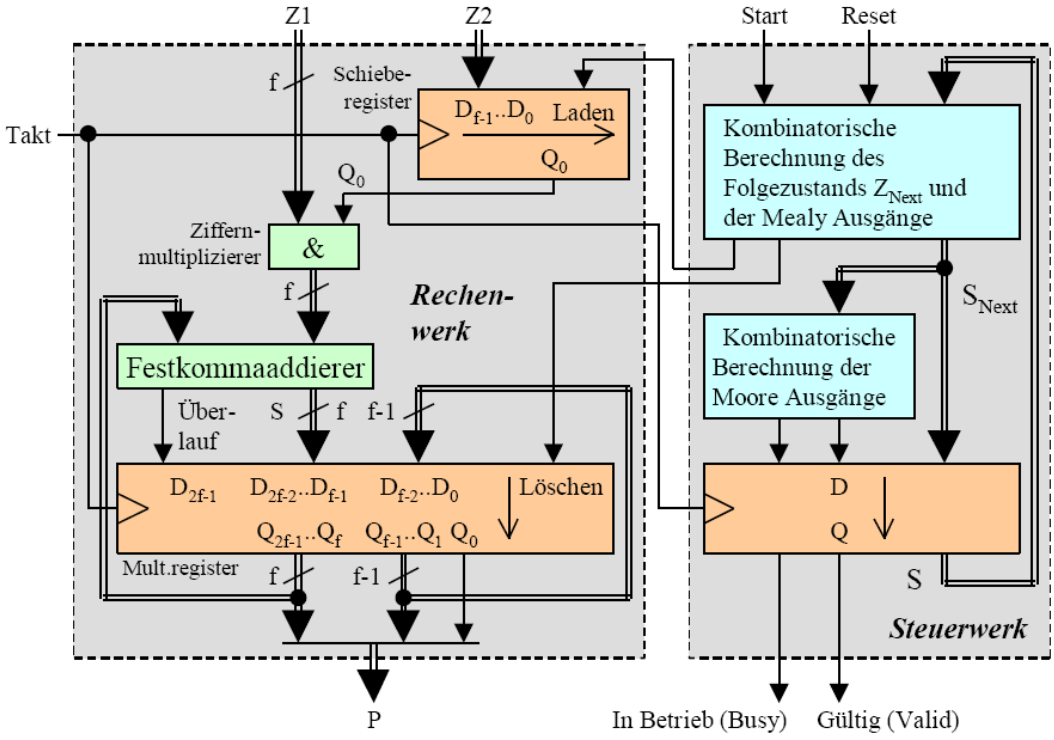
Es zeigt sich, daß in diesem Schema immer nur ein Addierer mit der Stellenzahl *f* benötigt wird, da hintere Stellen der Zwischensummen bei der Akkumulation nicht mehr verändert werden. Vom Addierer wird die erzeugte Summe und das erzeugte Überlaufbit zu einer Summe der Länge *f+1* zusammengesetzt. Vorliegendes Schema lässt sich direkt in einem parallelen Multiplizierer realisieren. Die Abbildung zeigt das Blockschaltbild für solch ein Rechenwerk.

Bild 21. Multiplikation



Mit einem eigenen Rechenwerk kann man die Multiplikation auch serialisieren.

Bild 22. Rechenwerk



Zum Start der Multiplikation sind der Multiplikand *Z1* und der Multiplikator *Z2* am Rechenwerk anzulegen. Der Multiplikand *Z1* muß konstant während des gesamten Multiplikationszyklus anliegen.
Die Multiplikation wird mit Aktivierung des Signals *Start* am Steuerwerk eingeleitet. Es bewirkt, daß der Multiplikator *Z2* ins Schieberegister geladen und das Multiplikationsregister gelöscht wird. Nach Wegnehmen des Signals *Start* beginnt die Multiplikation. Die aktive Multiplikation wird durch das Signal *Busy* des Steuerwerks angezeigt.
Beginnend mit der niedrigsten Ziffer des Multiplikators *Z2* wird der Multiplikand *Z1* Ziffernweise mit dem Multiplikator multipliziert. Das Ergebnis wird im Multiplikationsregister aufakkumuliert. Das Multiplikationsregister ist so beschaltet, daß das Ergebnis des vorhergehenden Schrittes um eine Bitposition nach rechts verschoben (Multiplikation mit 2^{-1}) in die Rechnung des neuen Schritts eingeht. Dies entspricht dem Vorgehen nach der Schulmethode. Die Multiplikation zweier Zahlen mit *f* Ziffern dauert mit dem seriellen Multiplizierer *f+1* Taktzyklen. Die notwendigen Kontrollsignale werden vom Steuerwerk generiert.

2.5.5 Division

Bei der Division eines Dividenden *Z1* durch einen Divisor *Z2* muß die Gleichung *Z1/Z2 = Q + R/Z2* so gelöst werden, daß bei gegebener Stellenzahl *f* ein Quotient *Q* und ein Rest *R* mit gleicher Stellenzahl ermittelt wird. Es werden im folgenden ganze Zahlen für *Z1, Z2, Q* und *R* angenommen.

2.5.5.1 Division nach Schulmethode

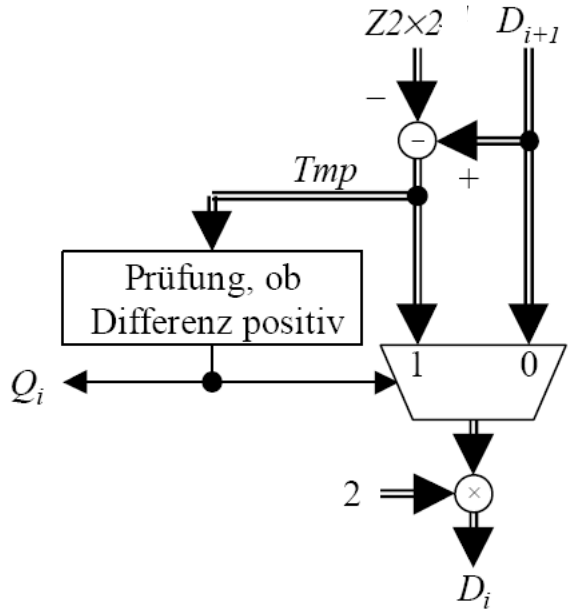
In der Schule wird die Division des Dividenden *Z1* durch den Divisor *Z2* ohne Berücksichtigung einer
Stellenzahl wie folgt gelehrt:

1. Der Divisor wird mit der potenzierten Basis b^g gewichtet, so daß er gerade noch kleiner ist als der Dividend.
2. Dann wird der gewichtete Divisor $Z2 \cdot b^g$ mit einer möglichst großen Ziffer Q_g multipliziert, so daß die Differenz
$D_g = Z1 - Q_g \cdot Z2 \cdot b^g$ aus Dividend und dem gewichteten Divisor gerade noch positiv bleibt und so, dass die Differenz
$D_g >=$ und $(Z1 - (Q_g+1) \cdot Z2 \cdot b^g) < 0$).
aus Dividend und dem doppelt gewichteten Divisor negativ wird.
Man erhält dabei die höchste Ziffer Q_g des Quotienten.
3. Mit weiteren g Schritten werden die verbleibenden Ziffern des Quotienten Q ermittelt. Die Ziffer Q_i
($g < i <= 0$) wird unter Verwendung der Differenz D_{i+1} des vorhergehenden Schrittes so gewählt, daß die Differenz
$D_i = D_{i+1} - Q_i Z2 \cdot b^i$ gerade noch positiv bleibt.
4. Nach Durchführung aller Divisionsschritte enthält D_0 den Rest der Division: $R = D_0$.
Da im Rechner immer mit festen Stellenzahlen gerechnet wird, wollen wir im folgenden annehmen, daß Dividend und Divisor aus f Ziffern bestehen und auch Quotient und Rest mit f Ziffern ermittelt werden sollen. Dann wird vor der Division der Dividend $Z1$ durch Anfügen von Nullen auf $2f-1$ Ziffern erweitert und der Divisor $Z2$ kann im ersten Schritt immer fest mit b^{f-1} multipliziert werden.

Das Problem beim oben gezeigten Vorgehen ist das Schätzen der Ziffern Q_i. Will man auf das Schätzen verzichten, muß man zur Ermittlung der Ziffer Q_i den gewichteten Divisor mit allen möglichen Ziffernwerten 0 bis $b-1$ multiplizieren und testweise von der Differenz D_{i+1} des vorhergehenden Berechnungsschritts subtrahieren.
Das Schätzen der Ziffer Q_i wird bei Basis $b=2$ einfach. Dort gibt es nur die Ziffern 0 und 1. Man kann testweise für $Q_i=1$ die Differenz $Tmp=D_{i+1}-Z2 \cdot 2^i$ bilden. Ist die Differenz positiv, stimmt die Annahme, sonst gilt $Q_i=0$.

2.5.5.2 Berechnung einer einzelnen Quotientenziffer

Bild 23. Division



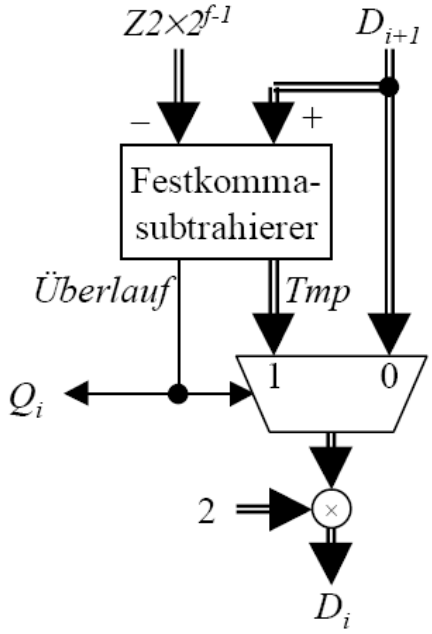
Der Vergleich, ob die Subtraktion positiv oder negativ ist, fällt beim vorgestellten Festkommaaddierer/Subtrahierer direkt ab. Betreibt man ihn als Subtrahierer mit positiven Zahlen, so erhält man am Überlauf-Ausgang für positive Zahlen (Carry) eine 1, wenn die Differenz positiv ist. Ansonsten markiert eine 0 einen Unterlauf aus dem positiven Zahlenbereich.

Beispiel: Berechnung von *Z1-Z2* für $Z1=5=(101)_2$, $Z2 = 3=(011)_2$:

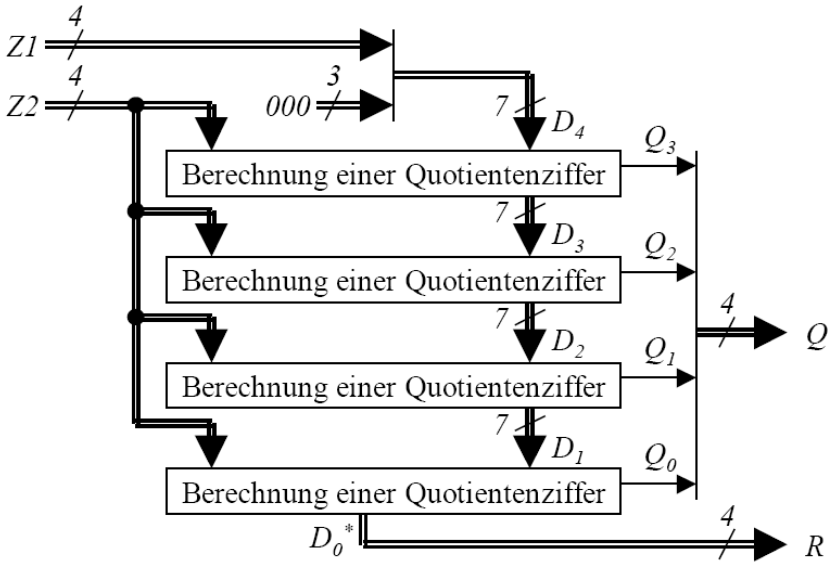
101 *Z1*
+100 *Z2* bitweise invertiert
+001 1 addieren für 2er-Komplement
1 010 Ergebnis *S=2*. Überlauf-Bit = 1, dies bedeutet bei Subtraktion: Kein Unterlauf!

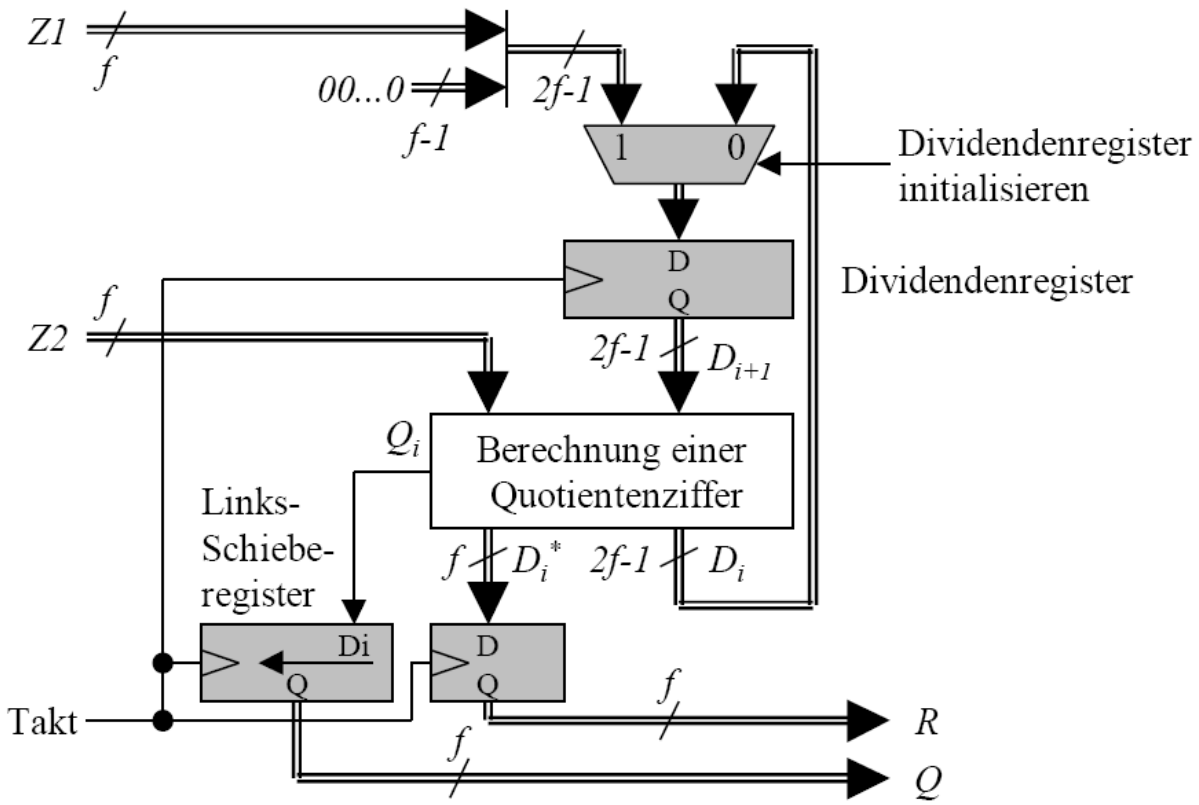
Damit kann der Festkommasubtrahierer in das oben gezeigte Blockschaltbild zur Berechnung einer Quotientenziffer eingefügt werden:

Bild 24. Division

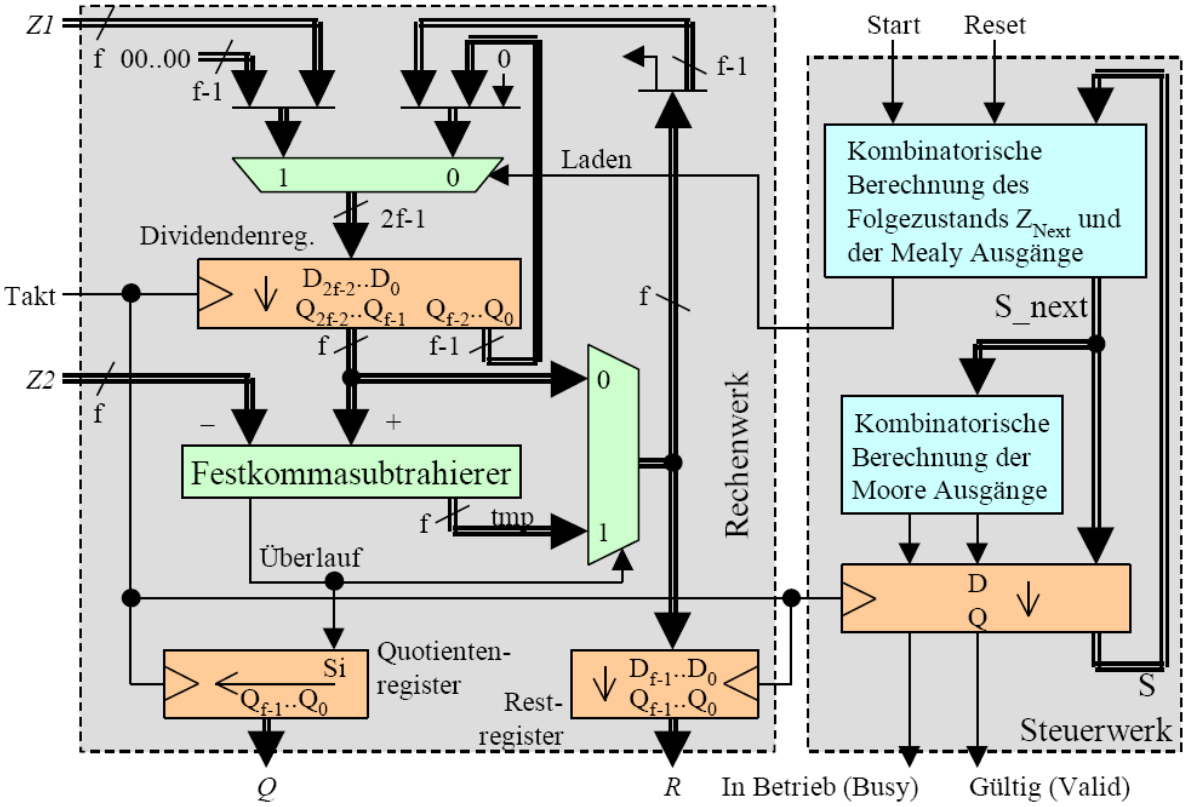


Beispiel: 4-Bit Dividierer





Rechenwerk zur seriellen Division zweier Festkommazahlen

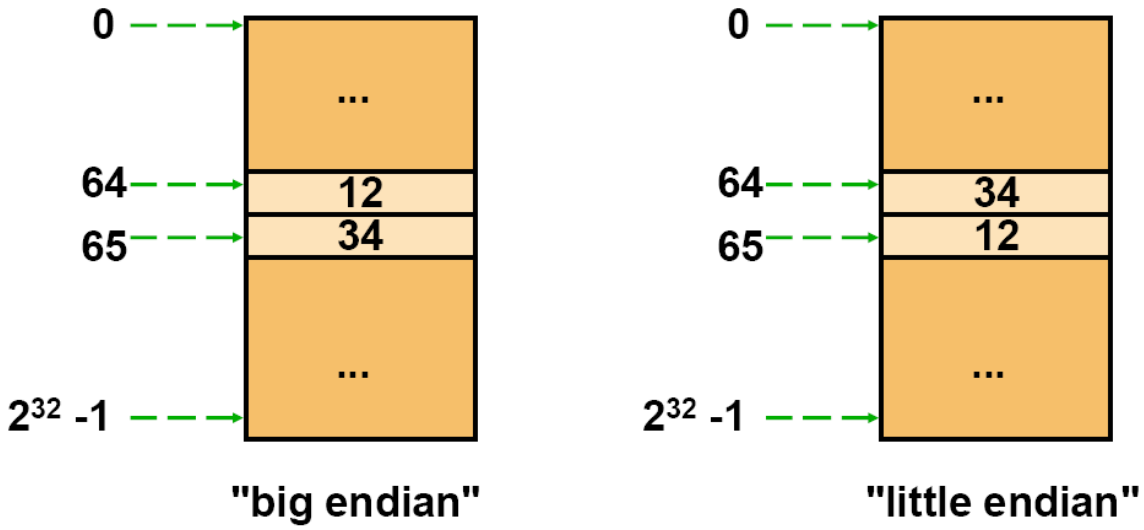


Blockschaltbild eines seriellen Dividierers

2.6 Little Endian, Big Endian

Bei der Interpretation z.B. als ganze Zahlen ist zu unterscheiden, ob das niedrigstwertige Byte an der höchsten Adresse ("*big endian*", z.B. Apple) oder der niedrigsten Adresse ("*little endian*", z.B. PC) gespeichert wird

Bild 25. Big/little endian



Beispielsweise Speicherung eines 4 Byte Langwort:
MSB → Byte3 Byte2 Byte1 Byte0 ← LSB

- **Adobe Photoshop** -- Big Endian
- **BMP (Windows and OS/2 Bitmaps)** -- Little Endian
- **DXF (AutoCad)** -- Variable
- **GIF** -- Little Endian
- **IMG (GEM Raster)** -- Big Endian
- **JPEG** -- Big Endian
- **FLI (Autodesk Animator)** -- Little Endian
- **MacPaint** -- Big Endian
- **PCX (PC Paintbrush)** -- Little Endian
- **PostScript** -- Not Applicable (text!)
- **QTM (Quicktime Movies)** -- Little Endian (on a Mac!)
- **Microsoft RIFF (.WAV & .AVI)** -- Both
- **Microsoft RTF (Rich Text Format)** -- Little Endian
- **SGI (Silicon Graphics)** -- Big Endian
- **TIFF** -- Both, Endian identifier encoded into file
- **WPG (WordPerfect Graphics Metafile)** -- Big Endian (on a PC!)

2.7 Fließkommaformate, Gleitkommeformate

Die bisher betrachteten Festpunktzahlen benötigen eine große Anzahl von Vorkomma- und Nachkommastellen, um einen brauchbaren Bereich von Zahlen darstellen zu können. Dies rührt daher, daß sowohl der Wertebereich als auch die Genauigkeit durch die Stellen bestimmt werden. Soll beispielsweise eine Milliarde durch Zahlen dargestellt werden, benötigt man zumindest 30 Vorkommastellen bei Basis b=2. Wird weiterhin auch ein Milliardstel an Genauigkeit gefordert, werden mindestens weitere 30 Bit als Nachkommastellen benötigt.

2.7.1 Trennung von Wertebereich und Genauigkeit

In realen Anwendungen werden größere Zahlen als eine Milliarde und feinere Genauigkeiten als ein Milliardstel benötigt. Dies kann aber nicht durch eine permanente Erweiterung der Stellenzahl abgedeckt werden, da dann die Rechengeschwindigkeit durch die breiten Recheneinheiten immer langsamer wird. Weiterhin wird in vielen Berechnungen nur ein kleiner Wertebereich und nur eine geringe Genauigkeit benötigt. Abhilfe schafft in diesem Zusammenhang die Beobachtung, daß in nahezu allen Anwendungen bei großen Zahlen eine geringere absolute Genauigkeit ausreicht, bei kleinen Zahlen hingegen eine hohe absolute Genauigkeit gefordert wird. Die relative Genauigkeit (Genauigkeit normiert auf den Wert) muß jedoch über den gesamten Wertebereich etwa gleich bleiben. Dies führt zur Trennung von Wertebereich und Genauigkeit in der Zahlendarstellung, welche die Grundlage für die Gleitpunktzahlen bildet. In der Gleitpunktdarstellung werden die Stellen für die Genauigkeit von den Stellen für den Wertebereich getrennt.

2.7.2 Fließkommazahlen (Floating Point),

• IEEE 754 Standard of Binary Floating Arithmetic (IEEE - Institute for Electrical and Electronic Engineers, USA)
• ANSI - American National Standards Institute

Allgemeine Darstellung:
Mit
m = Mantisse, e = Exponent., p = Basis des Exponenten

$z = \pm mp^{\pm e}$

Beispiel:
Mit Zahlenbasis = b = 10
Exponentenbasis = p = 10
kann die Zahl 16,35 geschrieben werden als
$0{,}1635 \times 10^2$ oder $1{,}635 \times 10^1$ usw.

Standards sind für einfache und doppelte Genauigkeit verfügbar (32 und 64 Bit)

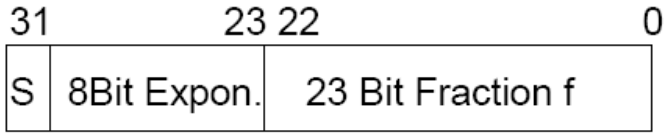
2.7.3 Normalisierung

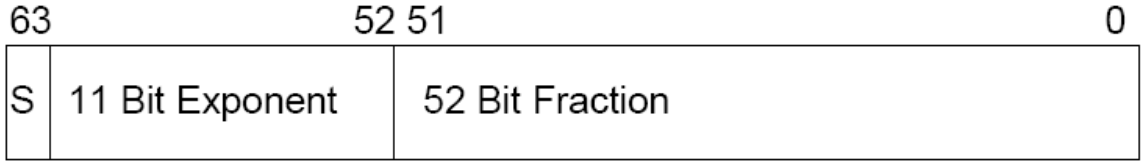
Ein Problem in der bisher beschriebenen Darstellung der Gleitpunktzahlen ist, daß die Darstellung nicht eindeutig ist. Der Wert 12,3 lässt sich z.B. darstellen als:
$0{,}0123 \times 10_3$ oder
$0{,}123 \times 10_2$ oder
$1{,}23 \times 10_1$ oder
$12{,}3 \times 10_0$ oder weitere Darstellungen
Bei mindestens zwei Vorkommastellen und 4 Nachkommastellen für die Mantisse und einer Stelle für den Exponenten können auch alle diese Möglichkeiten im Rechner dargestellt werden. Um eine einheitliche Darstellung zu erhalten, benötigt man eine normalisierte Form. Eine normalisierte Form gibt an, in welcher Stelle die höchste Ziffer der Mantisse abzuspeichern ist. Üblich ist die erste Vorkomma oder die erste Nachkommastelle zu wählen.
Normalisierungsregel 1:
Die erste Nachkommastelle enthält die höchste Ziffer ungleich 0.
$0{,}123 \times 10_2$
Normalisierungsregel 2:
Die erste Vorkommastelle enthält die höchste Ziffer ungleich 0.
$1{,}23 \times 10_1$
Entsteht während der Berechnung eine Zahl in nicht-normalisierter Form, so ist sie durch Verschieben der Ziffern in der Mantisse und entsprechender Anpassung des Exponenten vor dem Speichern zu normalisieren. Bei einer Normalisierung gemäß obiger Regel 1 braucht die 0 vor dem Komma natürlich nicht in der Mantisse abgespeichert werden.

2.7.4 Darstellung nach Norm:

Mantisse in die normalisierte Form 1.f bringen
(d.h. vorderste Dualziffer ungleich 0 in Vorkomaposition bringen, siehe TI)
Allgemeine Form:
$z = (-1)^s \times 1.f \times 2^{e+o}$
mit
s = Vorzeichen, Signum, sign (0=positiv, 1=negativ)
f = Fraktion, fraction
e = Exponent
o = Offset = halber Wertebereich des Exponenten (wg. +/-), alternativ o=0 bei 2K Darstellung des Exponenten

2.7.5 Darstellung von Fließkommazahlen





Beispiel:
$45{,}625 = 101101{,}101_2 = 1{,}01101101 \times 2^5$
(5 x „geschoben“)

s = 0 (positiv)
f = 01101101
o = 127 (für einfache Genauigkeit), nur falls keine 2K Darstellung des Exponenten

Ergebnis: 0 10000100 0110101...0
s exp. Fraction, auf 23 Bit aufgefüllt

Anmerkung: e = 0 ist für exakte 0 reserviert
e = 255 ist für Unendlich reserviert

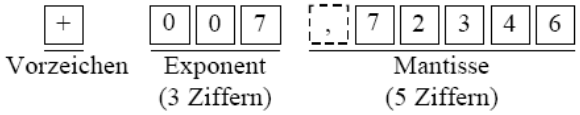
2.7.6 Darstellung des Wertes 0

Ein Problem bei der Normalisierung stellt der Wert 0 dar, da er keine Ziffer ungleich 0 enthält. Somit kann keine Normalisierungsregel Anwendung finden. Als Abhilfe kann für die 0 ein Spezialfall definiert werden:
Sind alle Ziffern im Exponent und in der Mantisse gleich 0, so handelt es sich um den Wert 0.
Dieser einfache Sonderfall ist bei verstecktem Bit (siehe unten) nicht mehr anwendbar.

Beispiel: $723{,}456\times10^4$, Darstellung nach Normalisierungsregel 1 (siehe Tabelle 1), 5 Stellen für Mantisse, 3 Stellen für Exponent, Basis *b=10*:

Normalisierung ergibt: $0{,}723456\times10^7$

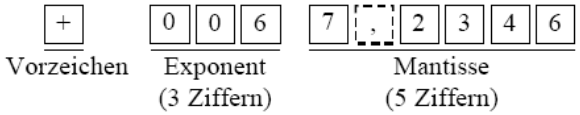
$\Rightarrow$ Exponent 7, Mantisse 0,72346 (Rundung auf 5 Stellen)



Beispiel: $723{,}456\times10^4$, Darstellung nach Normalisierungsregel 2 (siehe Tabelle 1), 5 Stellen für Mantisse, 3 Stellen für Exponent, Basis *b=10*:

Normalisierung ergibt: $7{,}23456\times10^6$

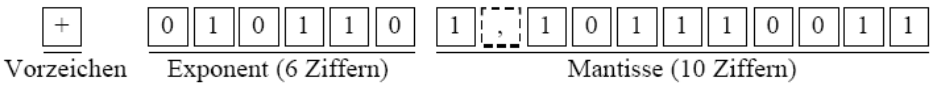
$\Rightarrow$ Exponent 6, Mantisse 7,2346 (Rundung auf 5 Stellen)



Beispiel: $723{,}456\times10^4$, Darstellung nach Normalisierungsregel 2 (siehe Tabelle 1), 10 Stellen für Mantisse, 6 Stellen für Exponent, Basis *b=2*:

Normalisierung ergibt: $(1{,}10111001100100000000)_2\times2^{22} = (1{,}10111001100100000000)_2\times2^{(10110)_2}$

$\Rightarrow$ Exponent $(010110)_2$, Mantisse $(1{,}101110011)_2$ (Rundung auf 10 Stellen)



2.7.7 Verstecktes Bit

Das obige Beispiel zeigt eine Besonderheit in der Basis *b=2* Darstellung. Nach der Normierung ist das höchste Bit immer 1. Es braucht somit nicht im Rechner gespeichert werden, sondern kann beim Laden der Zahl implizit ergänzt werden. Dadurch gewinnt man ein zusätzliches Bit in der Mantisse. Dieses nichtgespeicherte Bit der Mantisse wird als „Verstecktes Bit“ (engl. „hidden bit“) bezeichnet und findet in gängigen Gleitpunktformaten Anwendung.

Beispiel: $723{,}456\times10_4$, Darstellung nach Normalisierungsregel 2, 10 Stellen für Mantisse, 6 Stellen für Exponent, Basis *b=2*, Darstellung mit Verstecktem Bit:
Normalisierung ergibt: $(1{,}10111001100100000000)_2\times2^{22} =$
$(1{,}10111001100100000000)_2\times2^{(10110)_2}$
$\Rightarrow$ Exponent $(010110)_2$, Mantisse $(1{,}1011100110)_2$ (Rundung auf 10 Stellen)

2.7.8 Darstellung des Exponenten

In den bisherigen Beispielen wurden nur positive Exponenten verwendet. Damit konnte die allgemeine Betrachtung der Darstellung des Exponenten zunächst vermieden werden. Bei Gleitpunktzahlen ist jedoch auch die Darstellung negativer Exponenten unbedingt notwendig. Üblicherweise wird dafür die 2K Darstellung oder ein Excess-Darstellung oder BO-Darstellung gewählt.

2.7.9 Arithmetische Operationen mit Fließkommazahlen

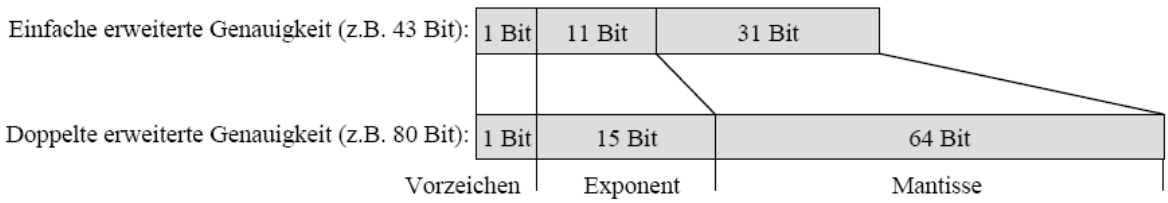
Mantissen und Exponenten werden getrennt behandelt
Addition: 1. Schritt: Exponentenangleichung
(d.h. die Zahl mit dem kleineren Exponenten wird an die mit dem größeren Exponenten angepasst)

$Z_{klein} = m_{klein} \times 2^{eklein - egroß} \times 2^{egroß}$
neue Mantisse m gemeinsamer Exponent

2. Schritt: Addition der beiden Mantissen, Exponent bleibt unverändert
3. Schritt: Ergebnis ggf. normieren

2.7.10 Rundung beim IEEE 754 Standard

Der IEEE754 Standard gibt vor, daß das Ergebnis einer Berechnung die Genauigkeit der halben letzen Mantissenstelle besitzen muß. Dies bedeutet, daß während der Berechnung die Genauigkeit über die spezifizierten Formate hinausgehen muß.
IEEE 754 spezifiziert dazu sogenannte erweiterte Formate, die **einfache erweiterte Genauigkeit** und die **doppelte erweiterte Genauigkeit**. Mit diesen Formaten werden während der Durchführung von Berechnungen die Anzahl der Bits des Exponenten und der Mantisse erhöht. Die Anzahl der Erweiterungsbits sind nicht im Standard festgeschrieben, sie variieren zwischen Implementierungen. Nachfolgende erweiterte Formate sind gebräuchlich und dienen als Beispiel:



Die zusätzlichen Bits der erweiterten Formate, welche eine erhöhte Genauigkeit während der Berechnung garantieren, werden als „guard“-Bits bezeichnet. Nach der Berechnung müssen die zusätzlichen Bits wieder gerundet werden, so daß die Zahl wieder in der Basis-Darstellung (Einfache Genauigkeit oder doppelte Genauigkeit) vorliegt. Zur Rundung sieht der IEEE 754 Standard vier Möglichkeiten vor:

2.7.10.1 Rundungsstrategie:

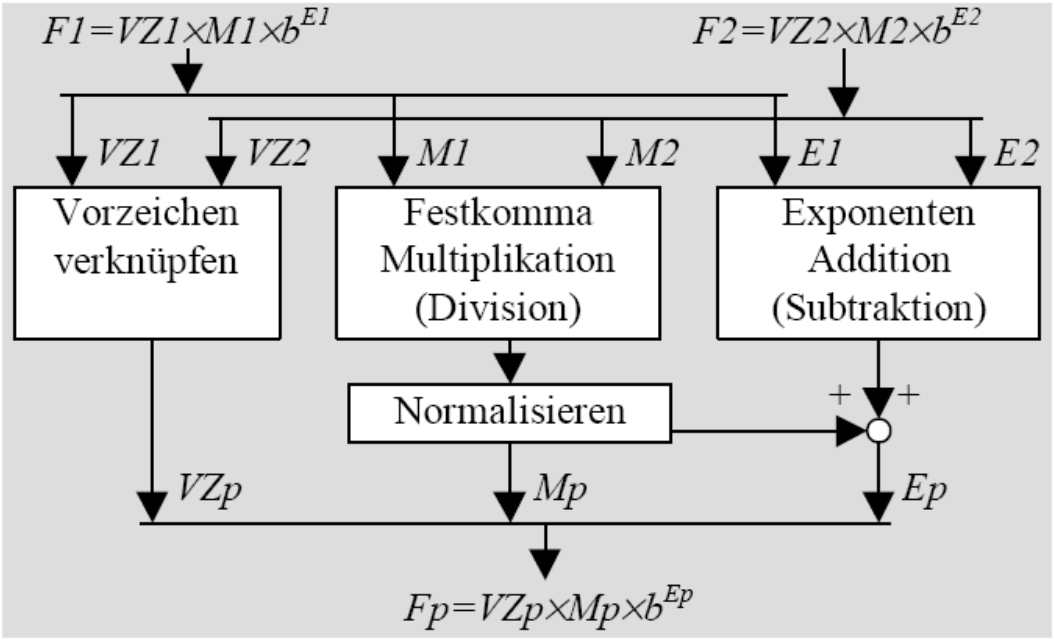
Rundung in Richtung Null
Die nächste repräsentierbare Zahl auf dem Zahlenstrahl von der erweiterten Zahl in Richtung zur Null wird verwendet.
Rundung in Richtung $+\infty$
Die nächste repräsentierbare Zahl auf dem Zahlenstrahl von der erweiterten Zahl in Richtung $+\infty$ wird verwendet.
Rundung in Richtung $-\infty$
Die nächste repräsentierbare Zahl auf dem Zahlenstrahl von der erweiterten Zahl in Richtung $-\infty$ wird verwendet.

Rundung zur nächstgelegenen repräsentierbaren Zahl mit geringstem Abstand (Defaulteinstellung)
Die repräsentierbare Zahl, die den geringsten Abstand zur erweiterten Zahl hat, wird verwendet. Liegt die erweiterte Zahl genau in der Mitte zwischen zwei repräsentierbaren Zahlen, wird die Zahl genommen, deren niedrigste Ziffer gerade ist.

2.7.10.2 Multiplizieren und Dividieren von Gleitkommazahlen

Die Multiplikation *Fp=F1´F2* und die Division *Fp=F1¸F2* von Gleitkommazahlen wird mit einer gleichartigen Struktur durchgeführt, die Operationen unterscheiden sich nur an zwei Stellen in den verwendeten Operatoren (Festkommamultiplikation versus Festkommadivision und Addition versus Subtraktion).

Bild 26. Multiplikation von Gleitkommazahlen



Bei der Multiplikation und Division von Festkommazahlen wird das Vorzeichen unabhängig von Mantisse und Exponent des Produktes bzw. des Quotienten bestimmt.
Bei der Multiplikation werden die Mantissen als Festkommazahlen multipliziert. Bei der Division muß entsprechend eine Festkommadivision durchgeführt werden. Die anschließende Normierung liefert einen Korrekturwert für den Exponenten.
Die Exponenten werden bei der Multiplikation addiert und bei der Division subtrahiert. Anschließend wird der bei der Normalisierung ermittelte Korrekturwert zur Summe bzw. Differenz der Exponenten addiert.

3 Rechnerarchitekturen

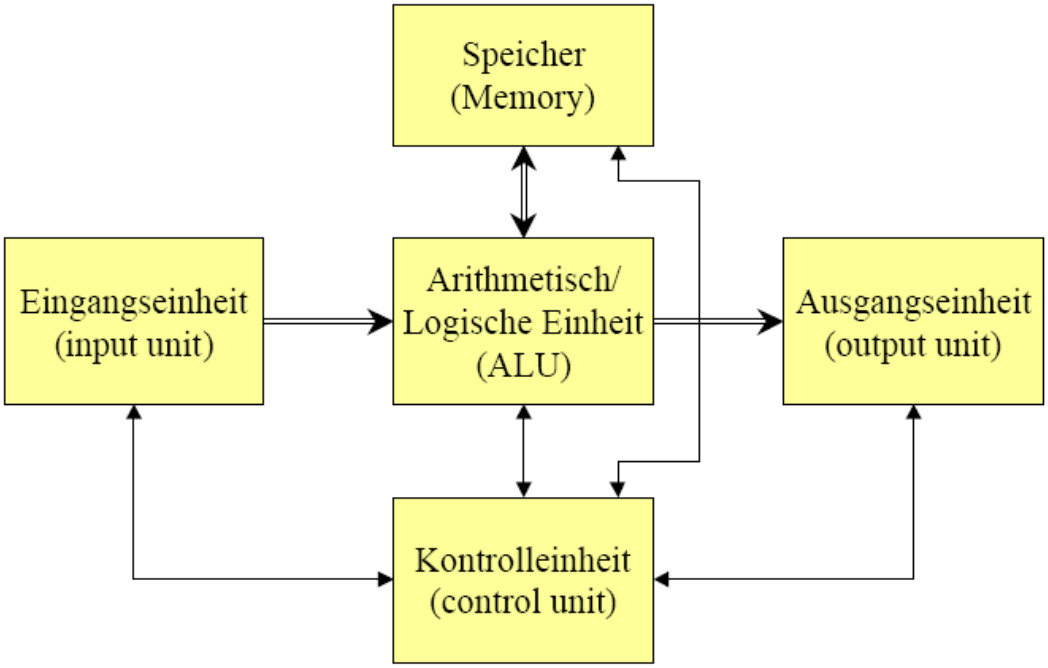
Eine Rechnerarchitektur beschreibt die Eigenschaften eines Systems, d.h. die Struktur des Entwurfs und das funktionelle Verhalten. Die Organisation des Daten- und Kontrollflusses,

der logische Entwurf und die physikalische Realisierung stellen die Umsetzung der Architektur dar.

3.1 Von Neumann

Heutige konventionelle Rechner basieren auf der „von Neumann“-Struktur. Es ist das am weitesten verbreitete Operationsprinzip von Rechnern. Ein Blockschaltbild der „von Neumann“-Struktur ist unten dargestellt. Kern des Bildes ist ein gemeinsamer Speicher für Daten und Befehle.

Bild 27. Von Neumann Architektur



3.1.1 Eingangseinheit:

Die Eingangseinheit (Eingabewerk) dient zur Aufnahme von Daten in den Rechner.

3.1.2 Ausgabeeinheit:

Die Ausgabeeinheit (Ausgabewerk) dient zur Ausgabe von Rechner ermittelter Ergebnisdaten.

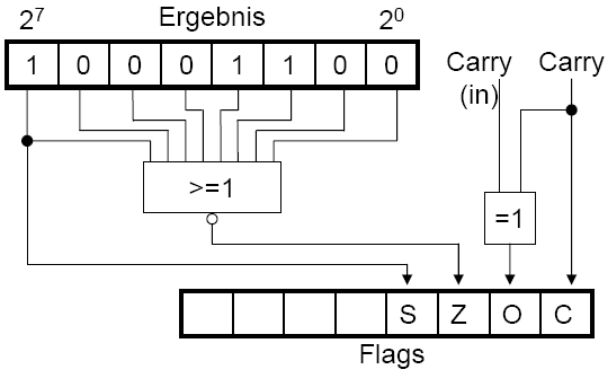
3.1.3 Speicher

Im Speicher werden Daten abgelegt. Der Speicher hat eine Informationsstruktur, die auf „Worten“ basiert. Ein Wort ist eine Binärzahl fester Länge. Die Anzahl der Bits eines Speicherworts wird als **Speicherwortbreite** bezeichnet. Typische Wortbreiten sind 8, 16, 32 oder 64 Bit. Im Speicher wird keinerlei Unterscheidung zwischen verschiedenen Datentypen vorgenommen. So werden z.B. Instruktionen und Programmdaten in der gleichen Informationsstruktur (Wort) des Speichers abgelegt.

3.1.4 Rechenwerk/ALU

Diese Einheit führt alle Berechnungen des Programms durch. Dazu beinhaltet diese Einheit arithmetische und logische Verarbeitungseinheiten zur Durchführung von arithmetischen und logischen Operationen sowie von Schiebeoperationen.
Nach Durchführung einer Operation stehen sowohl das Ergebnis als auch sogenannte Bedingungs-Bits (Flags) zur Verfügung. Die Bedingungs-Bits beschreiben das Ergebnis der Operation (z.B. Null, Negativ, Überlauf, Carry).

Bild 28. Rechenwerk



C: Carry-Flag	Übertrag, abhängig von der arithmetischen Operation
O: Overflow-Flag	Wird gesetzt, wenn ein Überlauf des Zahlenbereichs in Zweierkomplementdarstellung erfolgt
Z: Zero-Flag	Wird gesetzt, wenn der Wert im Ergebnis 0 ist.
S: Sign-Flag	(Negativ-Flag, Vorzeichen-Flag) ist eine Kopie des Vorzeichenbits des Ergebnisses

Die ALU verarbeitet ganze Datenworte in einem Verarbeitungsschritt. Sie ist ein Schaltnetz und hat nur logische Gatter, keine Speicher (außer für Ergebnisregister)
• Die ALU ist kaskadierbar, z.B. 2x4-bit-ALU –> 8-bit-ALU; eine 8-bit-ALU enthält ca. 200 logische Gatter

Die ALU-Rechenoperationen lassen sich zurückführen auf
• Addition
• Komplementbildung
• Schieben (Shift)

Die logischen Operationen sind rückführbar auf
• AND
• OR
• NOT
• Darüber hinaus fallen Informationen an, die für die nachfolgenden Operationen wichtig sein können (Flags).

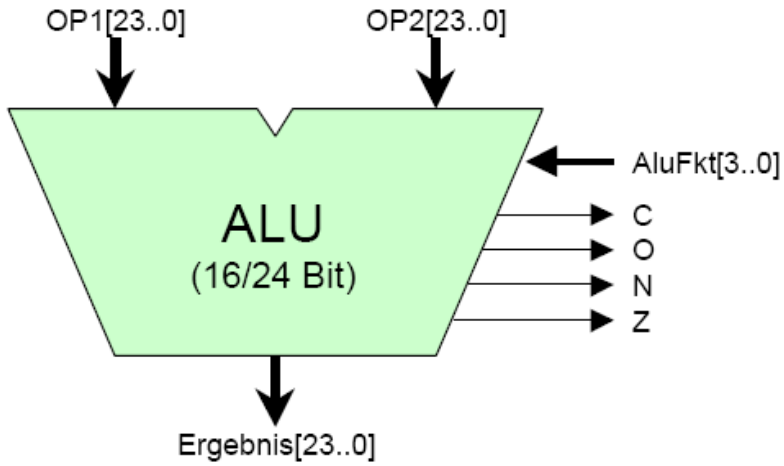
Der in nachfolgenden Abschnitten beschriebene Beispielrechner benötigt zur Realisierung von Operationen eine Arithmetisch-Logische Einheit in der elementare Rechenoperationen und notwendige Adreßberechnungen durchgeführt werden. Im Beispielrechner werden die Rechenoperationen mit einer Wortbreite von 16 Bit durchgeführt, Adressberechnungen

benötigen hingegen eine Wortbreite von 24 Bit. Somit muß die ALU beide Wortbreiten unterstützen.

3.1.4.1 Funktionen der ALU

Die ALU des Beispielrechners soll folgende Funktionsgruppen beinhalten:
1. Erzeugung von Konstanten (16/24 Bit)
2. Logikfunktionen (16 Bit)
3. Arithmetische Operationen (16/24 Bit)
4. Schiebeoperationen (16 Bit)
5. Durchreichen von Operanden durch die ALU (16/24 Bit)

Die ALU verknüpft zwei Operandenworte `OP1` und `OP2` zu einem Ergebniswort `Ergebnis`. Das Symbol der ALU ist in unten dargestellt.



Bei Ausführung der ALU-Operationen werden Bedingungsbits gesetzt, welche Informationen über die Berechnung und über das Ergebnis liefern. Diese sind:
C (Carry): Überlauf bei der Addition positiver Zahlen (nur arithmetische Operationen).
O (Overflow): Überlauf bei der 2er-Komplement-Addition (nur arithmetische Operationen).
Z (Zero): ALU-Ergebnis ist Null.
N (Negative): ALU-Ergebnis ist negativ.
Die Bedingungsbits werden aus den unteren 16 Bit der Operatoren generiert, da nur bei arithmetischen und logischen Operationen auf Datenworten diese Statusinformation von Interesse ist. Bei Adressberechnungen, welche im Beispielrechner mit 24 Bit durchgeführt werden, sind die Bedingungsbits nicht von Interesse.
In der ALU des Beispielrechners werden die in untenstehender Tabelle gelisteten Funktionen realisiert. Neben den arithmetischen, und logischen Operationen können auch Konstanten erzeugt und Operanden 1 oder 2 vom Eingang zum Ausgang weitergereicht werden. Die Auswahl der jeweiligen Funktion erfolgt durch Anlegen der gelisteten Kodierungen an die Eingänge AluFkt[3..0].

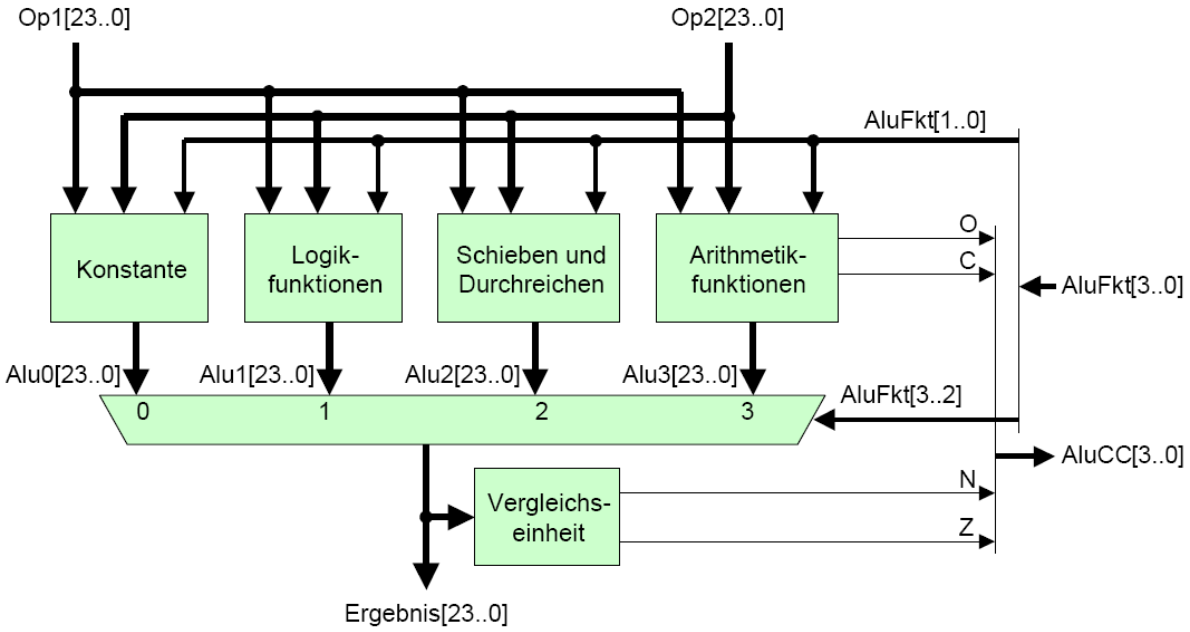
Bild 29. Steuerung der ALU

ALU Funktion	Kodierung	Einheit	Beschreibung
KONST_0	00 00	Konstante	Die Konstante 0 wird am Ausgang ausgegeben.
KONST_1	00 01		Die Konstante 1 wird am Ausgang ausgegeben.
KONST_M2	00 10		Die Konstante –2 wird am Ausgang ausgegeben.
KONST_M1	00 11		Die Konstante –1 wird am Ausgang ausgegeben.
NOT	01 00	Logik	Operand 2 wird bitweise negiert am Ausgang ausgegeben. Operand 1 wird ignoriert.
XOR	01 01		XOR-Verknüpfung von Operand 1 und Operand 2.
OR	01 10		OR-Verknüpfung von Operand 1 und Operand 2.
AND	01 11		AND-Verknüpfung von Operand 1 und Operand 2.
SHR	10 00	Schieben und Durchreichen	Operand 2 um ein Bit nach rechts schieben.
SHL	10 01		Operand 2 um ein Bit nach links schieben.
OP2	10 10		Operand 2 durchreichen, Operand 1 wird ignoriert.
OP1	10 11		Operand 1 durchreichen, Operand 2 wird ignoriert.
INC2	11 00	Arithmetik	Operand 2 wird um 2 inkrementiert und das Ergebnis am Ausgang ausgegeben. Operand 1 wird ignoriert.
DEC2	11 01		Operand 2 wird um 2 dekrementiert und das Ergebnis am Ausgang ausgegeben. Operand 1 wird ignoriert.
ADD	11 10		Operand 1 und Operand 2 werden addiert.
SUB	11 11		Operand 1 wird von Operand 2 subtrahiert.

3.1.4.2 Aufbau der ALU

Im Inneren ist die ALU des Beispielrechners aus 4 Funktionsblöcken aufgebaut, welche die 4 Funktionsgruppen aus der Tabelle realisieren. Die Blöcke arbeiten parallel, durch einen nachgeschalteten Multiplexer wird das ALUErgebnis von der durch AluFkt[3..2] selektierten Funktionseinheit ausgewählt. Die nächste Abbildung zeigt diese Struktur.

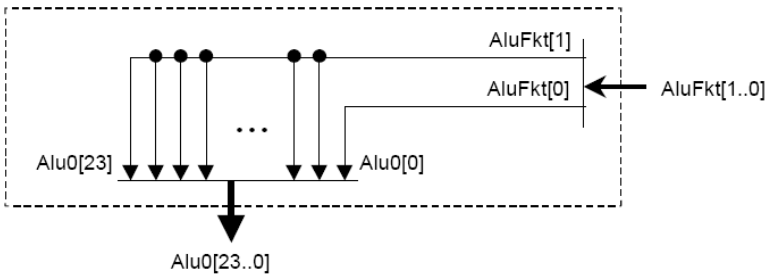
Bild 30. Alu



3.1.4.3 Generierung von Konstanten

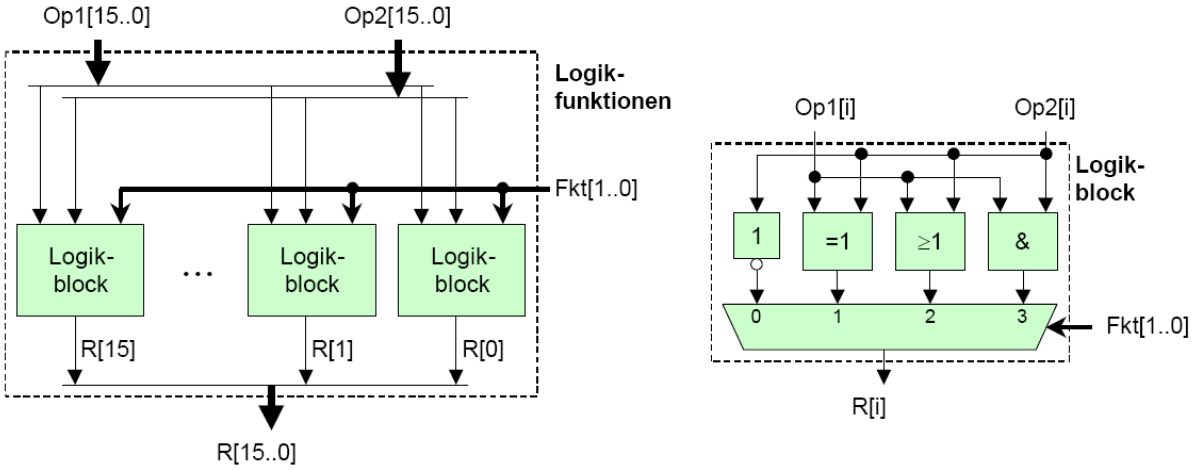
Die Generierung von Konstanten erfolgt in der ALU des Beispielrechners mit einer festen Verschaltung der Funktionseingänge auf den Ausgang der ALU.

Bild 31. Konstanten



3.1.4.4 Logikfunktionen und Bit-weise logische Verknüpfungen

Neben der Arithmetik werden in Hochsprachen logische Operationen und Schiebeoperationen als Operatoren spezifiziert. Somit müssen diese Operationen in einem Rechner durch ein Rechenwerk direkt oder indirekt zur Verfügung gestellt werden. Der Abschnitt zeigt Rechenwerke für bitweise logische Operationen und für Schiebeoperationen, wie sie in Rechnern benötigt werden, um beispielsweise die entsprechenden Operatoren von C oder C++ zu realisieren.
Die wichtigsten bitweisen logischen Operationen sind die Negation, UND-Verknüpfung, ODER-Verknüpfung und die EXKLUSIV-ODER-Verknüpfung. Die Funktionen lassen sich in einer Logikeinheit zusammenfassen, wie dies in folgender Abbildung dargestellt ist.



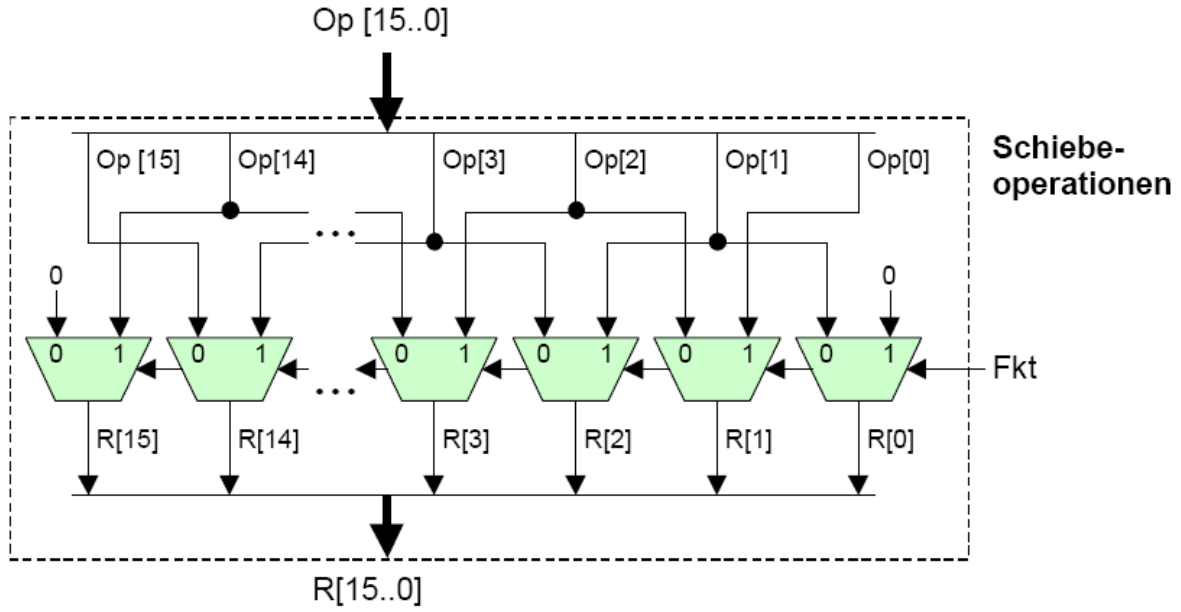
In der gezeigten Realisierung werden die zwei Operanden *OP1* und *OP2* bitweise miteinander verknüpft. Die Verknüpfung zweier Einzelbits ist in der Abbildung in einem Logikblock zusammengefasst, dessen Innenschaltung auf der rechten Seite der Abbildung im Detail gezeigt ist. Die Verknüpfung erfolgt parallel mit allen vier Logikfunktionen. Das gewünschte Ergebnis der Funktion wird mit einem nachgeschalteten Multiplexer ausgewählt.

3.1.4.5 Schiebeoperationen, Schieben und Durchreichen

Bei der Realisierung von Schiebeoperationen in einem Rechner muss entschieden werden, ob per Hardware beliebige Schiebeoperationen in einem Schritt, oder nur die Verschiebung um ein Bit pro Schritt durchführt werden kann.

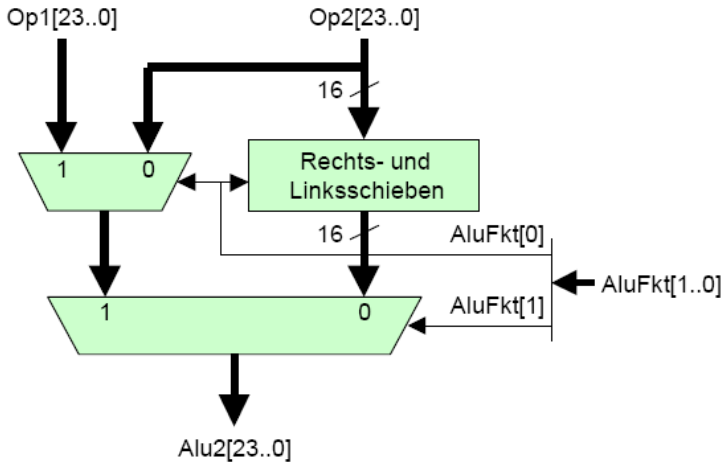
Die erste Lösung führt zu einem „Barrel-Shifter“. Der Entwurf einer zugehörigen Architektur wird dem Leser als Übung überlassen. Im zweiten Fall müssen mehrfache Schiebeoperationen durch mehrfache Anwendung einzelner Schiebeschritte per Software realisiert werden.

Bild 32. Rechenwerk zum Rechts- und Linksschieben eines Operanden



Im ALU-Block „Schieben und Durchreichen“ sind zwei Funktionalitäten kombiniert.

Bild 33. Schieben und Durchreichen



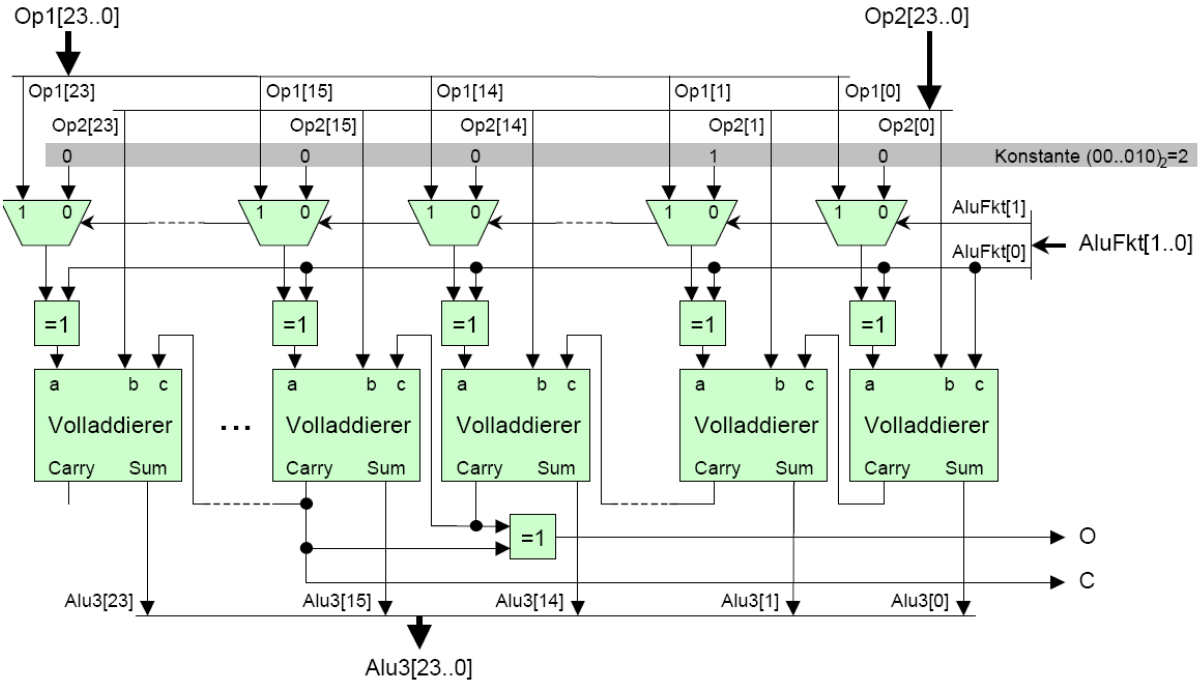
Zum einen kann in diesem Block der Operand 2 nach rechts und nach links geschoben werden. Dies belegt zwei Funktionen der vier Funktionen des Blocks. Zusätzlich ist in diesem Block ein Durchreichen der Operanden 1 und 2 durch die ALU realisiert.

3.1.4.6 Arithmetikfunktionen

Als elementare Arithmetikoperationen werden die Addition und die Subtraktion unterstützt. Carry- und Überlaufbit (C, O) werden dabei aus den unteren 16 Bit bestimmt (Datenwortbreite des Beispielrechners). Zur Adressberechnung kann die ALU die Konstante

2 zum Operanden 2 addieren oder von diesem subtrahieren, was zur sequentiellen Veränderung von Adressen im Beispielrechner benötigt wird.
Vor dem a-Eingang ist ein Multiplexer vorgeschaltet, der entweder den Operanden 1 (Summand oder Subtrahend) oder den konstanten Wert 2 selektiert. Damit kann, wie in der Funktionstabelle der ALU (Tabelle 3) angeführt, entweder Operand 1 oder der konstante Wert *2* von Operand 2 addiert bzw. subtrahiert werden.

Bild 34. Arithmetikfunktionen



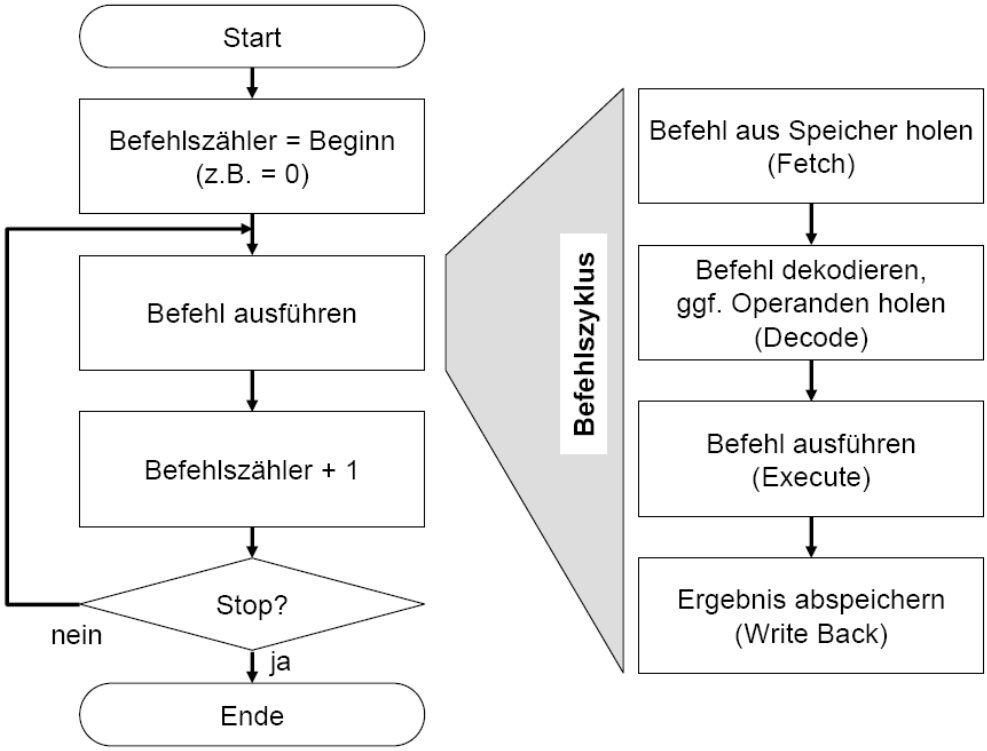
3.1.5 Leitwerk/ Kontrolleinheit/ Steuerwerk

Eines der wichtigsten allgemeinen Entwurfsprinzipien ist die Trennung von Kontrolle und der Verarbeitung von Daten, dementsprechend beim Hardware-Entwurf die Trennung von Steuerwerk und Operationswerk:

Die Kontrolleinheit (Steuerwerk, Leitwerk) interpretiert die im Speicher abgelegten Worte. Je nach Interpretation sind dies:
- Instruktionen (Befehle),
- Programmdaten oder
- Adressen.

Aufgrund der Instruktionen werden die übrigen Einheiten gesteuert, so daß die Instruktionen korrekt ausgeführt werden. In der Kontrolleinheit wird der Zustand des Rechners gehalten, der die Speicherposition der nächsten Instruktion bestimmt. Dazu ist in der Kontrolleinheit ein Register vorgesehen, welches auf die nächste auszuführende Instruktion zeigt, es wird als **Programmzähler** (program counter, **PC**) bezeichnet. Nach Ausführung einer Instruktion wird der Programmzähler üblicherweise erhöht und zeigt direkt auf das der aktuellen Instruktion nachfolgende Speicherwort. Einige als „Sprungbefehl“ bezeichnete Instruktionen verändern den Programmzähler und unterbrechen damit die sequentielle Instruktionssequenz. Die Kontrolleinheit ist so ausgelegt, daß alle Adressberechnungen mit einer endlichen Anzahl von Bits ausgeführt werden. Diese Anzahl wird als
Adresswortbreite bezeichnet. Typische Breiten sind 16, 20, 24 oder 32 Bit.

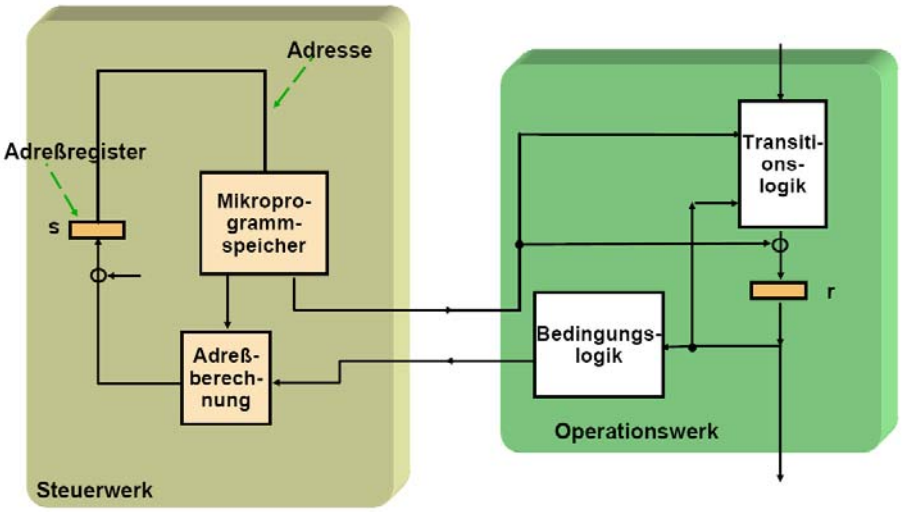
Bild 35. Modell eines Steuerwerk Befehlsablaufes



3.1.5.1 Beispiel eines mikroprogrammierten Steuerwerks:

Die Mikroprogrammierung ist eine weitverbreitete Technik der Steuerwerkrealisierung, z.B. in CISC-Rechnern. Die Speicherung der Steuersignale erfolgt in einem (i.a. read-only) Mikroprogrammspeicher (µROM). Es gab auch Systeme mit ladbarem Mikroprogrammspeicher, die Möglichkeit der Modifikation der Mikroprogramme durch den Anwender wurde jedoch fast nie genutzt. Vorteile der Mikroprogrammierung sind:
hohe Flexibilität während des Entwurfsprozesses: statt Neuentwurf der Ansteuerlogik nur Ändern des ROMInhalts
µProgrammierung: teilweise Nachbildung von Konstrukten höherer Programmiersprachen, z.B. Schleifen, Unterprogrammsprünge, etc.

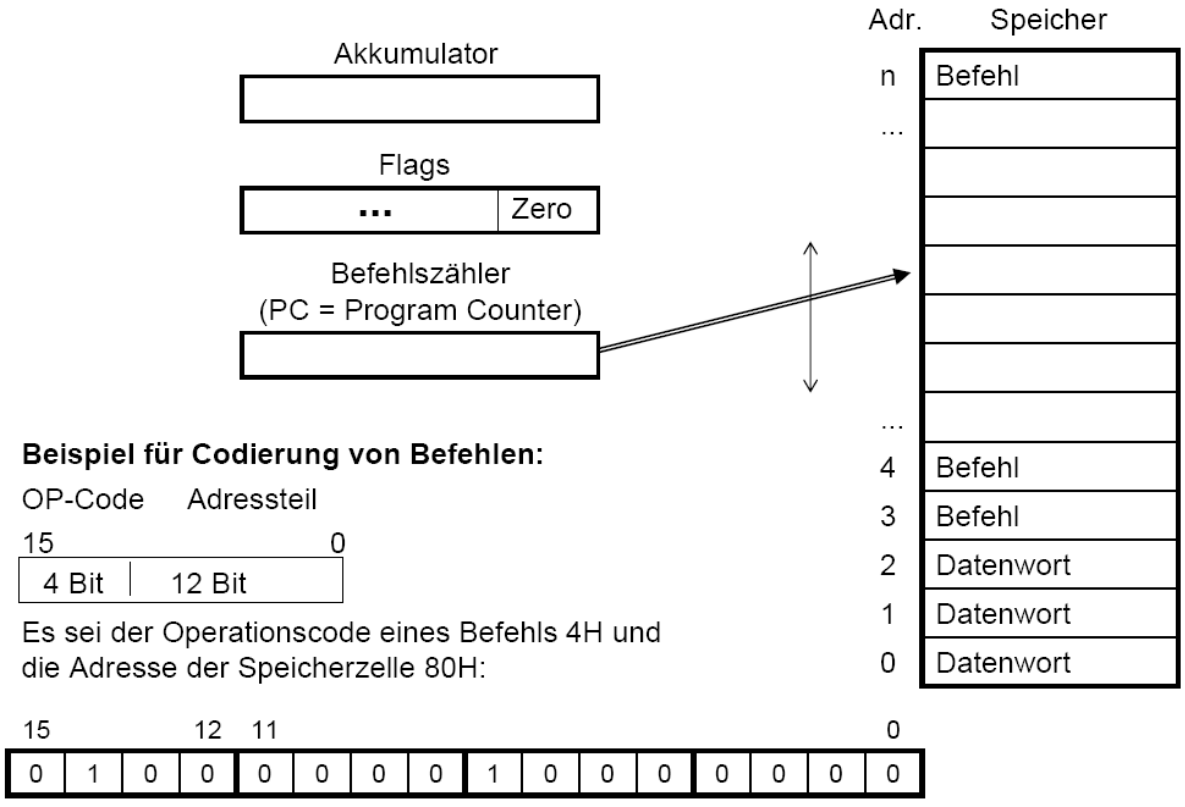
Bild 36. Mikroprogrammiertes Steuerwerk



3.1.6 Zentrale Verarbeitungseinheit (CPU)

In realen Rechnern werden die Arithmetisch/Logische Einheit und die Kontrolleinheit häufig in einer physikalischen Einheit (Chip) zusammengefasst. Diese Kombination wird als **Zentrale Verarbeitungseinheit** (central processing unit, **CPU**) bezeichnet. Auch ein Teil des Speichers rutscht in die CPU, es werden Operanden für die ALU und allgemeine Register in der CPU gespeichert und nicht jedes Mal aus dem separaten Speicher geholt.

Bild 37. CPU



Modell eines einfachen v.-Neumann Rechners

3.1.7 Wichtige Instruktionstypen des „von Neumann“-Rechners

Die wichtigsten Instruktionstypen eines „von Neumann“-Rechners sind:
- Transferbefehle (load, store, LDA, STA)
- Arithmetische Befehle (addieren, subtrahieren), Logische Befehle (AND, OR,XOR), Shift-Befehle.
- Sprungbefehle (bedingt, unbedingt, JMP, JZ, JNZ)

Die Bedeutung dieser Instruktionstypen ist die folgende:
Transfer: Die Transferbefehle dienen zum Laden von Programmdaten aus dem Speicher in die ALU (load) und zum Speichern von Ergebnissen von der ALU zurück in den Speicher (store). Nach Ausführung einer Transferinstruktion zeigt der PC auf das nachfolgende Wort im Speicher.
Arithmetik, Logik, Schieben: Die arithmetischen und logischen Befehle sowie die Schiebebefehle stellen die Operationen bereit, die für Berechnungen benötigt werden. Nach Ausführung einer Operation stehen das Ergebnis und die Bedingungs-Bits in der ALU zur Verfügung. Nach Ausführung der Instruktion zeigt der PC auf das nachfolgende Wort im Speicher.
Sprünge: Bei den Sprungbefehlen wird der PC mit einem neuen Wert geladen, damit wird die sequentielle Instruktionsreihenfolge im Speicher durchbrochen. Sprungbefehle unterteilen sich in unbedingte und in bedingte Sprünge.
Bei **unbedingten Sprüngen** wird der PC auf jeden Fall mit einem neuen Wert geladen.
Bei **bedingten Sprüngen** wird der Wert eines zugeordneten Flag-Bits ausgewertet. Ist es gesetzt, wird der PC neu geladen und damit der Sprung ausgeführt. Ansonsten wird der PC wie bei den anderen Befehlen erhöht und zeigt auf das dem Befehl nachfolgende Wort im Speicher.

Die Bedingungs- oder Flagbits eines Prozessors bilden zusammen mit dem PC und ggf. anderen Konfigurationsregistern den sogenannten **Prozessorstatus**.

Prozentualer Anteil von Anweisungen in Hochsprachenprogrammen

Anweisung	Mittlerer zeitlicher Anteil
Zuweisung	47%
if	23%
call	15%
loop	6%
goto	3%
Andere	7%

Anzahl der Operatoren	Prozentualer Anteil	Beispiel
1	80%	b
2	15%	b+c
3	3%	a×b+c
4	2%	(a+b) × (c+d)
≥5	<<1%	

Somit ergeben sich folgende aus Hochsprachen hergeleitete Anforderungen an einen Prozessor:

- Wenige, einfache Befehle.
- Schnelle Ausführung der Befehle. Dies bedeutet feste Instruktionslängen, um feste Bearbeitungszeiten zu erzielen und damit Pipelining zu ermöglichen. Weiterhin kein Mikroprogrammsteuerwerk sondern festverdrahtete Logik.
- Viele interne Register, um externe Speicherzugriffe zu minimieren.

Dies ist ein erster Hinweis auf die später besprochenen RISC Prozessoren und die DSPs.

3.1.8 Befehlssatz-Architekturen

Ein Programmierer, der ein Maschinenprogramm für einen bestimmten Rechner erstellen will, muß natürlich nicht alle Details der Organisation des Rechnerkerns beherrschen. Für ihn genügt es i.a., dessen *Instruktionssatz-Architektur* (ISA) zu kennen. Die ISA beinhaltet
a) den Maschinenbefehlssatz (Art und Format der Befehle)
b) die adressierbaren Register des Rechnerkerns
c) die Darstellungsmöglichkeiten für Daten (Maschinen-Datentypen: Byte, Integer-Zahlen, Fixpunktzahlen etc.)
d) die Adressierungsmöglichkeiten des Speichers (Adressierungsmodi)
e) die Art der Daten-Ein- und Ausgabe.

Hinsichtlich der Art des Befehlssatzes können folgende Architekturen unterschieden werden:

- Stack-Architektur
- Akkumulator-Architektur
- Allgemeinzweck-Register-Architekturen (General Purpose Register Archit.'s – GPR) in den Ausprägungen:
  - Register-Register-Maschinen – RISC (auch Load-Store-Architektur genannt)
  - Register-Speicher-Maschinen – CISC
  - Speicher-Speicher-Maschinen

Bei realen Mikroprozessoren wird oft versucht, in eine vorherrschende Befehlssatz-Architektur die Vorteile anderer Befehlssatz–Architekturen zu integrieren (Mischformen).

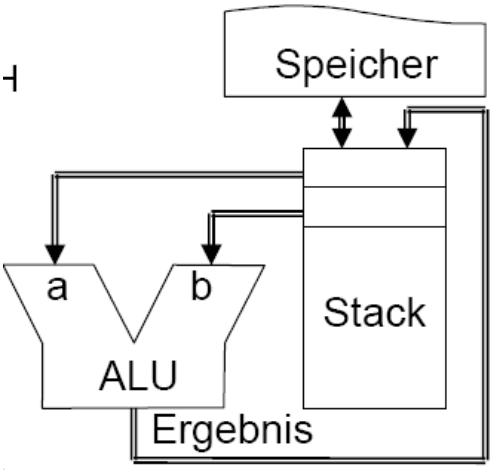
3.1.8.1 Stack - Architektur

- CPU hat LIFO-Speicher (Stack), auf den man mit PUSH und POP zugreifen kann
- Es kann meist nur auf die oberste Speicherstelle des Stacks zugegriffen werden

- Sehr einfach, effiziente Codeumsetzung bei geklammerten mathematischen Ausdrücken möglich
- implizite Adressierung von A und B bei ADD

PUSH Op_A ; Op_A von Speicher -> Stack
PUSH Op_B ; Op_B von Speicher -> Stack
ADD ; Addiere obere Werte im Stack
POP Op_C ; Op_C (= Op_A + Op_B) von Stack -> Sp.

Bild 38. Stackarchitektur



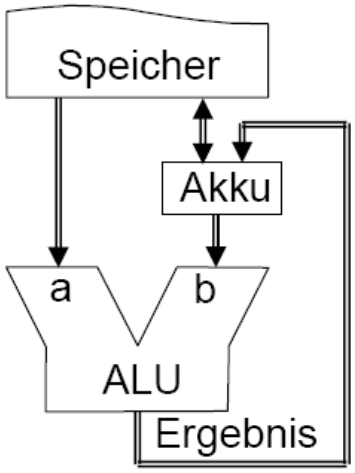
Stacks werden unabhängig von der jeweiligen Architektur bei Unterprogrammaufrufen und Parameterübergaben verwendet. Ein Call Befehl pusht implizit die Rücksprungadresse und ggf. Registerstati, ein Return popt diese Werte wieder.

3.1.8.2 Akkumulator - Architektur

- Ausgezeichnetes Register: Akku
- LOAD und STORE wirken nur auf Akku. Er ist als expliziter Operand an jeder Operation beteiligt. Jede Operation braucht nur eine Adresse
- Sehr kompaktes Befehlsformat

LDA Op_A ; Op_A von Speicher -> Akku
ADD Op_B ; Akku = Akku + Op_B
STA Op_C ; Op_C (= Op_A + Op_B) Akku -> Sp.

Bild 39. Akkumulator - Architektur

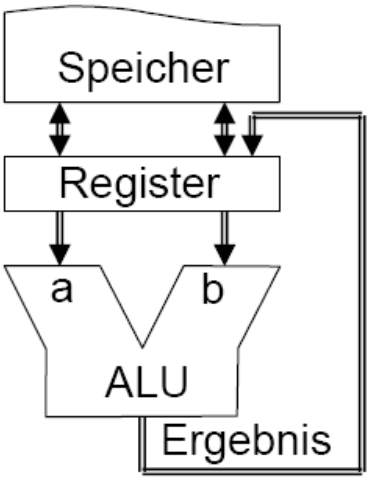


3.1.8.3 Register-Register-Architektur – RISC (auch Load-Store-Architektur)

- alle Operationen greifen nur auf Register zu,
- nur LOAD und STORE greifen auf Speicher zu
- 32 – 512 Register verfügbar
- einfaches Befehlsformat fester Länge
- alle Instruktionen brauchen in etwa gleich lange

LOAD R1, Op_A ; lade Op_A aus Speicher in R1
LOAD R2, Op_B ; lade Op_B aus Speicher in R2
ADD R3, R1, R2 ; addiere R1 und R2, Ergbn. -> R3
STORE Op_C, R3 ;speicher R3 -> Op_C (=Op_A + Op_B)

Bild 40. Register-Register-Architektur



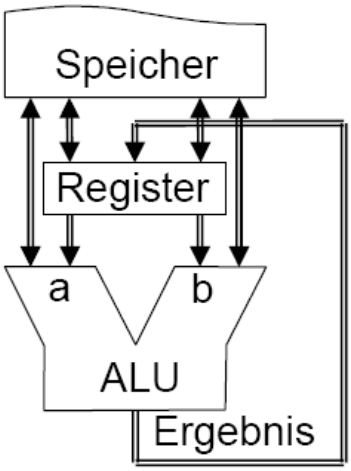
3.1.8.4 Register-Speicher-Architektur – CISC (Mischung von Akkumulator- und Load-Store-Architektur)

- Operationen greifen auf Register und/oder Speicher zu
- Befehlsformat variabler Länge
- mächtige Befehle
- stark unterschiedliche Zeiten für Instruktionsausführung

MOV AX, Op_A ; Op_A von Speicher -> Register AX
ADD AX, Op_B ; AX = AX + Op_B

MOV Op_C, AX ; Op_C = AX (= Op_A + Op_B)

Bild 41. Register-Speicher-Architektur



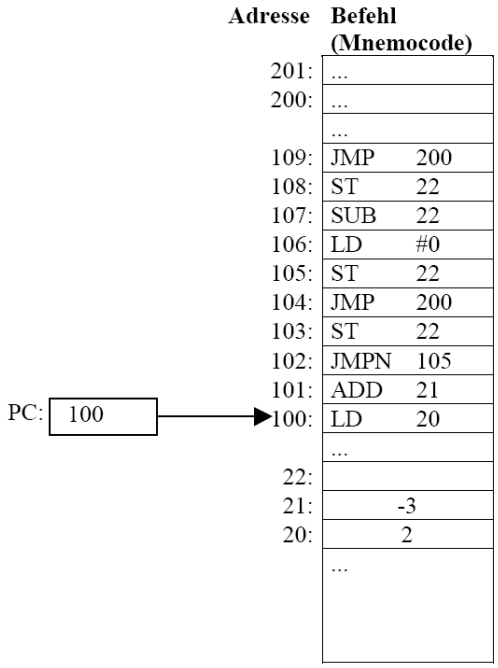
3.1.9 Beispielrechner

Das nachfolgende Beispiel zeigt die Wirkungsweise eines „von Neumann“-Rechners anhand eines einfachen Beispielrechners. Die nachfolgend verwendeten Befehle des Beispielrechners stellen keineswegs normierte Befehle dar, sie zeigen lediglich eine Möglichkeit der Befehlsimplementierung. Sie wurden so gewählt, daß sie zu Befehlen realer Rechner ähnlich sind und die wichtigsten Instruktionstypen verdeutlichen. Die Instruktionen werden als **Mnemocodes** dargestellt. Sie stellen eine verständliche Form der Befehle dar und sind direkt abbildbar auf die Bitmuster der Befehle und damit auf die Worte des Speichers.

3.1.9.1 Aufgabe

Zwei Zahlen sollen addiert und vom Ergebnis der Betrag gebildet werden:

- Die Zahlen stehen im Speicher an den Adressen 20 und 21.
- Das Programmsegment für diese Aufgabe steht ab Adresse 100 im Speicher.
- Nach Ausführung des Programmsegments soll die Ausführung an Adresse 200 fortgesetzt werden.



Erläuterung der Mnemocodes:

LD	<Adresse>	Lade Wert vom Speicher in ALU
LD	#<Wert>	Lade „Wert“ in die ALU
ST	<Adresse>	Lade Wert von ALU in Speicher.
ADD	<Adresse>	Addiere Wert vom Speicher zum Wert in der ALU.
SUB	<Adresse>	Subtrahiere Wert vom Speicher vom Wert in der ALU.
JMP	<Adresse>	Unbedingter Sprung zu Befehl im Speicher.
JMPN	<Adresse>	Bedingter Sprung bei Flag „Negativ“ zu Befehl im Speicher.

Erläuterung des Ablaufs:
Zum Start der Bearbeitung muß der Programmzähler auf die Adresse der ersten Instruktion zeigen. Dies ist im vorliegenden Beispiel die Adresse 100.
Die Kontrolleinheit liest das Datenwort mit der Adresse aus dem Speicher, auf welche der PC zeigt. Es wird implizit angenommen, daß es sich um eine Instruktion handelt.
Die Kontrolleinheit dekodiert das Datenwort als Befehl und steuert die übrigen Einheiten (z.B. ALU), um den Befehl auszuführen. Nachfolgend sind die Befehle des Beispiels in der Reihenfolge ihrer Bearbeitung erläutert.

Adresse	Befehl	Beschreibung
100:	LD 20	Dieser Befehl ist ein Transferbefehl. Er transferiert das Wort, welches im Speicher an Adresse 20 steht, in die ALU. Im Beispiel wird somit der Wert „2“ vom Speicher in die ALU transferiert. Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 101.
101:	ADD 21	Dieser Befehl ist sowohl ein Transfer- als auch ein Arithmetikbefehl. Er transferiert zunächst das Wort, welches an Adresse 21 im Speicher steht, in die ALU und addiert es zu dem in der ALU gespeicherten Wort. Das Ergebnis bleibt in der ALU gespeichert. Neben dem Ergebnis werden in den Bedingungs-Bits noch Informationen über das Ergebnis gespeichert. Bedingungs-Bits speichern z.B. ob ein Ergebnis „0“ (Z-Flag) oder negativ (N-Flag) ist. Im Beispiel wird der Wert „-3“ aus dem Speicher in die ALU transferiert und zum dort gespeicherten

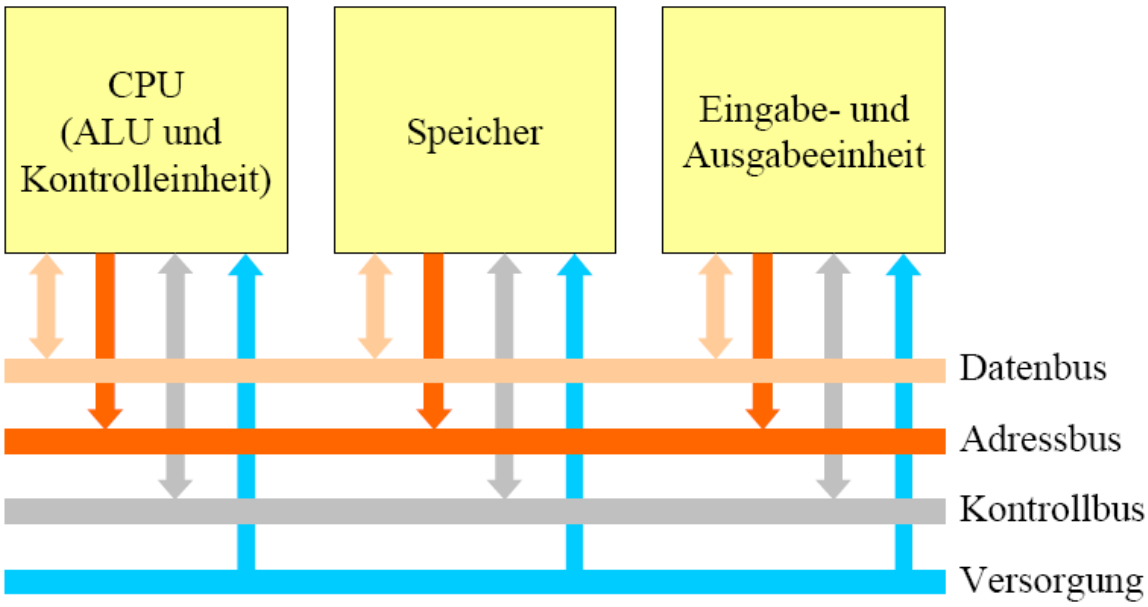
		Wert „2“ addiert. Das Ergebnis „-1“ wird in der ALU gespeichert. Von den Bedingungs-Bits wird das N-Flag gesetzt (Ergebnis ist negativ) und das Z-Flag gelöscht (Ergebnis ist nicht „0“). Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 102.
102:	JMPN 105	Dieser Befehl ist ein bedingter Sprungbefehl. Er lädt den PC mit der angegebenen Adresse, wenn das zugeordnete Bedingungs-Bit gesetzt ist. Der vorliegenden Instruktion ist das N-Flag zugeordnet. Da das N-Flag aufgrund des vorhergegangenen ADD-Befehls gesetzt ist, wird der PC neu mit dem Wert 105 geladen und somit ein Sprung ausgeführt.
105:	ST 22	Dieser Befehl ist ein Transferbefehl. Er speichert den Wert der ALU im Speicher unter der angegebenen Adresse. Im Beispiel wird das Ergebnis „-1“ von der ALU in das Speicherwort mit der Adresse 22 transferiert. Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 106.
106:	LD #0	Dieser Wert ist ein Transferbefehl. Er lädt einen Wert, der direkt („immediate“) in der Instruktion gespeichert ist, in die ALU. Im Beispiel ist der Wert „0“ in der Instruktion gespeichert und wird in die ALU transferiert. Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 107.
107:	SUB 22	Dieser Befehl ist sowohl ein Transfer- als auch ein Arithmetikbefehl. Er transferiert zunächst das Wort, welches an Adresse 21 im Speicher steht, in die ALU und subtrahiert es zu dem in der ALU gespeicherten Wort. Das Ergebnis bleibt in der ALU gespeichert. Neben dem Ergebnis werden in den Bedingungs-Bits noch Informationen über das Ergebnis gespeichert. Bedingungs-Bits speichern z.B. ob ein Ergebnis „0“ (Z-Flag) oder negativ (N-Flag) ist. Im Beispiel wird der Wert „-1“ aus dem Speicher in die ALU transferiert und dort vom gespeicherten Wert „0“ subtrahiert. Das Ergebnis „1“ wird in der ALU gespeichert. Von den Bedingungs-Bits wird das N-Flag gelöscht (Ergebnis ist nicht negativ) und das Z-Flag gelöscht (Ergebnis ist nicht „0“). Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 108.
108:	ST 22	Dieser Befehl ist ein Transferbefehl. Er speichert den Wert der ALU im Speicher unter der angegebenen Adresse. Im Beispiel wird das Ergebnis „1“ von der ALU in das Speicherwort mit der Adresse 22 transferiert. Nach Ausführung des Befehls wird der PC weitergeschaltet und zeigt auf Adresse 109.
109:	JMP 200	Dieser Befehl ist ein unbedingter Sprungbefehl. Der PC wird mit der in der Instruktion enthaltenen Adresse geladen. Im Beispiel wird der PC mit dem Wert 200 geladen und somit zur Instruktion an Adresse 200 gesprungen.

3.1.10 Bussysteme

3.1.10.1 Der Systembus

Die entscheidende Präzisierung des Systembus-Modells ist die Einführung eines **Systembus**ses, der aus

Bild 42. Bussysteme



Datenbus, **Adressbus** und **Kontrollbus** sowie einem Bus zur elektrischen **Versorgung** der Komponenten aufgebaut ist. Einige Architekturen beinhalten zusätzlich noch einen I/O-Bus. Physikalisch ist jeder Bus aus einer Anzahl von Leitungen aufgebaut. Ein 16-Bit Adressbus besteht beispielsweise aus 16 individuellen Leitungen, welche jede ein Adressbit repräsentiert. Der Systembus setzt sich aus mehreren Bussen zusammen, wobei jeder Bus eine individuelle Funktion besitzt. Über den Systembus werden die Daten zwischen den Einheiten des Rechners transferiert. Dazu werden die Signale der einzelnen Busse in einer spezifizierten Reihenfolge auf den Bussen angelegt. Für jede Busleitung darf es zu einem Zeitpunkt nur eine Einheit geben, welche die Busleitung treibt und damit einen gültigen Logikpegel auf dem Bus anlegt. Ansonsten entstehen Kurzschlüsse, die zu Konflikten bei Buszyklen oder schlimmer noch zur Zerstörung von Bustreibern führen können.

3.1.10.2 Adress- und Datenbus

Der Adress- und der Datenbus sind zwei homogene Busse, die Signale gleicher Funktion zusammenfassen. Der Datenbus ist ein bidirektionaler Bus. Im Falle des Schreibens werden Daten von der CPU (Quelle) zu Speicher oder Ausgabeeinheit (Senke) übertragen. Im Falle des Lesens werden Daten in der umgekehrten Richtung von Speicher oder Eingabeeinheit (Quelle) zur CPU (Senke) übertragen. Die Anzahl der Datenbusleitungen ergibt sich aus der **Datenbusbreite** eines Rechners, sie stimmt normalerweise mit der Speicherwortbreite überein. Diese muß jedoch nicht mit der Datenwortbreite der ALU übereinstimmen.
Der Adressbus ist (in einfachen Systemen ohne DMA) ein unidirektionaler Bus. Er transferiert Adressen von der CPU (Quelle) zu Speicher und Ein-/Ausgabeeinheiten (Senke). Er spezifiziert die Speicheradresse, unter der ein Speicherwort angesprochen wird. In den allermeisten Rechnern wird durch die Adresse ein Byte im Speicher adressiert. Besitzen diese Rechnern weiterhin eine Datenbusbreite von 16 Bit (2 Bytes) oder größer, werden beim Wortzugriff auf den Speicher die unteren Adressbits nicht benötigt.
Zur Einsparung von Signalleitungen und damit auch von Anschluss-Pins der Chips findet man Rechner mit gemultiplextem Adress- und Datenbus. Dies bedeutet, daß Adresse und Daten zeitlich versetzt auf den gleichen Leitungen angelegt werden. Die Zeitpunkte, wann Adressen und wann Daten gültig sind, wird durch Signale des Kontrollbusses festgelegt.

3.1.10.3 Kontrollbus

Der Kontrollbus ist ein inhomogener Bus, er fasst Signale unterschiedlicher Funktion zusammen. Die Signale variieren von Rechner zu Rechner. Die Hauptaufgaben der Signale sind jedoch bei allen Rechnern gleich. Diese sind:

- Markieren einer gültigen Adresse (Beginn eines Transfers).
- Auswahl eines Schreib- oder Lesetransfers.
- Abschluss des Transfers.

Manche Rechner besitzen zum Transfer zwischen der CPU und Speicher oder Ein-/Ausgabe einen
synchronen Systembus, d.h. das zeitliche Verhalten der Signale auf dem Bus wird allein durch die CPU gesteuert. Andere Rechner besitzen einen
asynchronen Systembus, dort können langsame Speicher oder Ein-/Ausgabeeinheiten das zeitliche Verhalten der Bussignale bei Bedarf verlangsamen. Zur Steuerung eines asynchronen Systembusses beinhaltet der Kontrollbus ein Quittungssignal, welches von dem adressierten Speicher oder Ein-/Ausgabegerät gesetzt werden muß, um den Transfer durchzuführen. Somit kommt bei asynchronen Systembussen noch diese Aufgabe des Kontrollbusses hinzu:

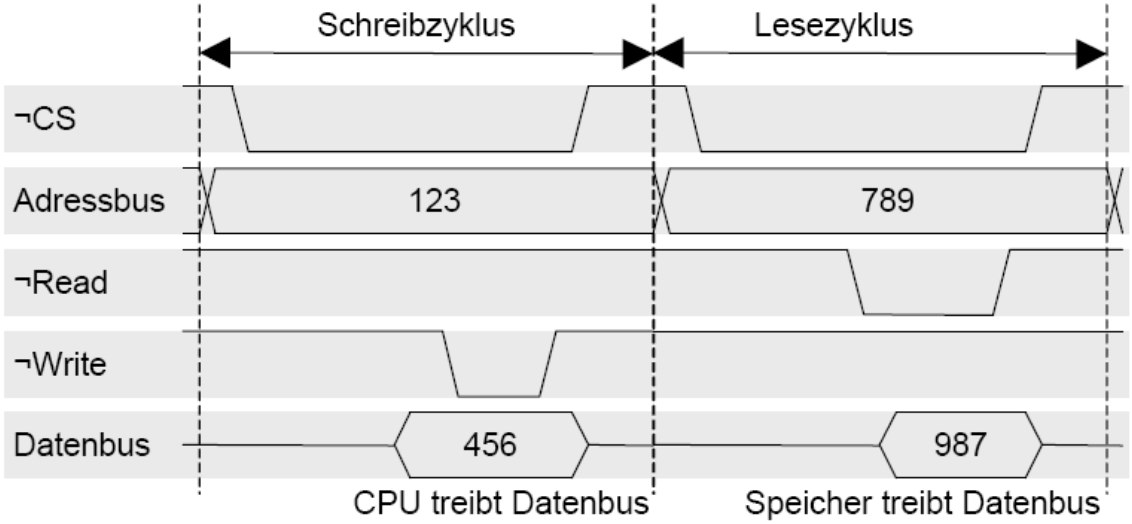
- Quittung für Datentransfer bei asynchronen Systembussen.

3.1.10.4 Buszyklen

Die zeitliche Abfolge von Signalen auf den Bussen zum Transfer eines Datenwortes zwischen den Einheiten des Rechners wird als
Buszyklus bezeichnet. Für jeden Buszyklus gibt es immer genau einen
Bus-Master, der die logische und zeitliche Abfolge der Signale beim Transfer steuert. Üblicherweise ist die CPU der Bus-Master. Beim
Lesezyklus legt die CPU (Bus-Master) die zum Lesen notwendigen Kontrollsignale und die Adresse am Bus an, um eine Speicherzelle im Speicher (oder einer Eingabeeinheit) zu adressieren. Der Speicher gibt daraufhin auf dem Datenbus den in der Speicherzelle gespeicherten Wert auf dem Datenbus aus, der von der CPU eingelesen wird. Die CPU terminiert den Zyklus durch Deaktivierung von Kontrollsignalen. Beim Lesezyklus ist der Speicher (oder eine Eingabeeinheit) die Datenquelle, Datensenke ist die CPU.
Beim
Schreibzyklus legt die CPU die zum Schreiben notwendigen Kontrollsignale und die zugehörige Adresse am Bus an und adressiert damit eine Speicherzelle im Speicher (oder eine Ausgabeeinheit). Dann gibt die CPU ein Datenwort auf dem Datenbus aus und markiert den Datenbus durch Kontrollsignale als gültig. Der Speicher übernimmt daraufhin die Daten und schreibt sie in die adressierte Speicherzelle. Die CPU terminiert den Zyklus durch Deaktivieren von Kontrollsignalen. Beim Schreibzyklus ist die CPU die Datenquelle, Speicher (oder Ausgabeeinheit) ist die Datensenke.

3.1.10.4.1 Synchroner Schreibzyklus:

Bild 43. Schreibzyklus



1. Der Schreibzyklus wird mit dem Anlegen der Adresse 123 durch die CPU eingeleitet.
2. Die CPU aktiviert das „¬CS“-Signal durch Setzen auf 0-Pegel und signalisiert damit den Beginn des Zyklus an Speicher und Ein-/Ausgabeeinheiten. Diese beginnen nun, die Adresse auszuwerten und zu überprüfen, ob sie angesprochen sind.
3. Die CPU legt den Datenwert 456 auf dem Datenbus an. Dieser wechselt damit vom hochohmigen Zustand (Keine Einheit treibt die Bussignale) in den aktiven Zustand; die Datenleitungen werden von der CPU getrieben.
4. Die CPU aktiviert das „¬Write“-Signal. Damit kann der im Beispiel adressierte Speicher das Datenwort 456 übernehmen.
5. Die CPU deaktiviert das „¬Write“-Signal. Der adressierte Speicher muß nun das Datenwort übernommen haben.
6. Die CPU deaktiviert den Datenbus, damit wechselt der Bus wieder in den hochohmigen Zustand. Weiterhin deaktiviert sie das „¬CS“-Signal und signalisiert damit das Ende des Schreibzyklus.

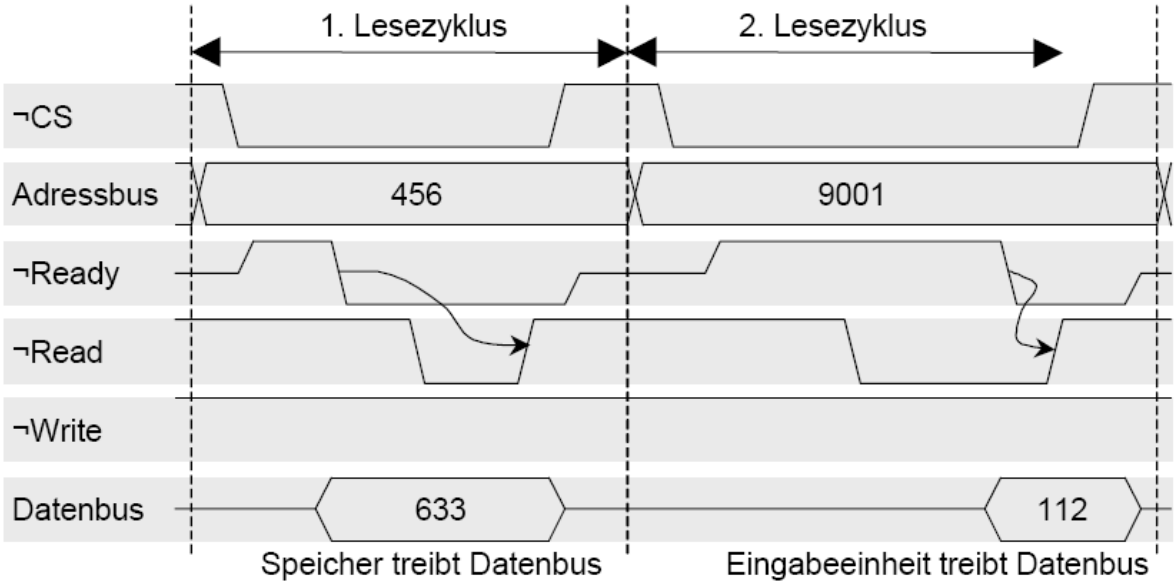
3.1.10.4.2 Synchroner Lesezyklus:

1. Der Lesezyklus wird mit dem Anlegen der Adresse 789 durch die CPU eingeleitet.
2. Die CPU aktiviert das „¬CS“-Signal und signalisiert damit den Beginn des Lesezyklus an Speicher und Ein–/Ausgabeeinheiten. Diese werten nun die Adresse aus und überprüfen, ob sie angesprochen sind.
3. Die CPU aktiviert das „¬Read“-Signal. Damit kann der im Beispiel adressierte Speicher beginnen, das Datenwort auszugeben.
4. Der Speicher legt den Datenwert 987 auf dem Datenbus an. Der Bus wechselt damit vom hochohmigen Zustand in den aktiven Zustand, wo in diesem Fall die Datenleitungen vom Speicher getrieben werden.
5. Die CPU übernimmt das Datenwort vom Bus und deaktiviert das „¬Read“-Signal. Der adressierte Speicher muß nun das Datenwort vom Bus nehmen.
6. Der Speicher deaktiviert den Datenbus, damit wechselt der Bus wieder in den hochohmigen Zustand.Weiterhin deaktiviert die CPU das „¬CS“-Signal und signalisiert damit das Ende des Lesezyklus.

Bei beiden Zyklen ist zu erkennen, daß der zeitliche Ablauf nur durch die CPU bestimmt ist. Speicher und Ein–/Ausgabeeinheiten richten sich nach den zeitlichen Vorgaben.

3.1.10.4.3 Asynchroner Lesezyklus:

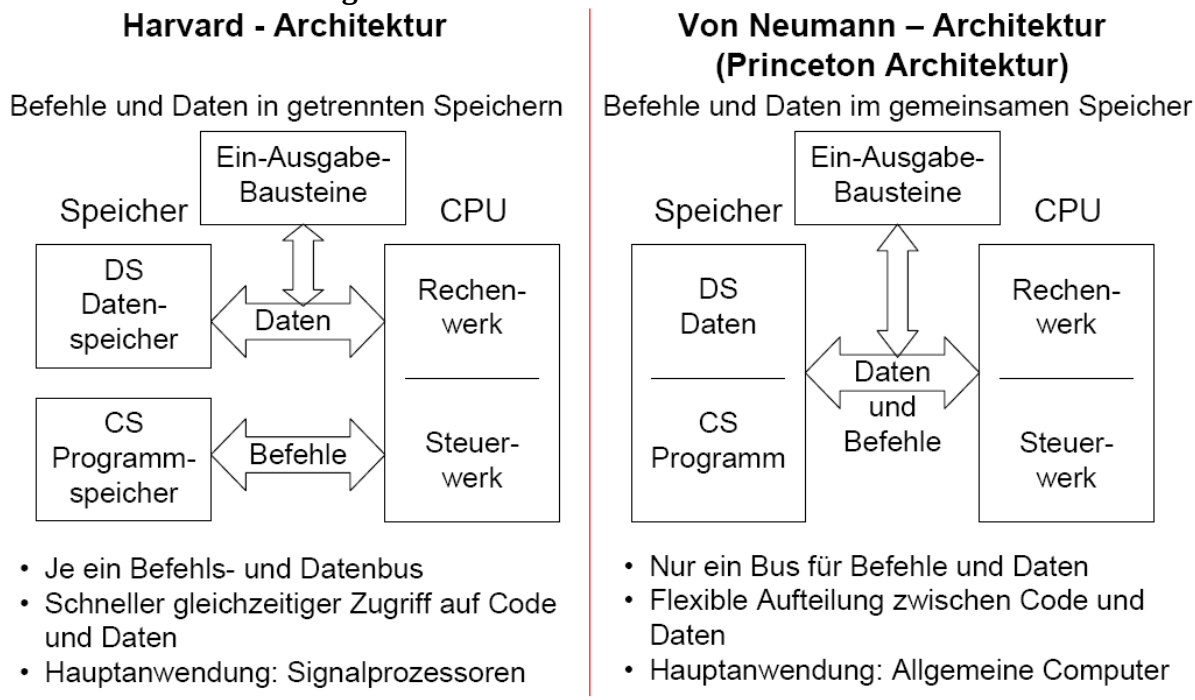
Bild 44. Lesezyklus



1. Der Lesezyklus wird mit dem Anlegen der Adresse 456 durch die CPU eingeleitet.
2. Die CPU aktiviert das „¬CS“-Signal und signalisiert damit den Beginn des Lesezyklus an Speicher und Ein–/Ausgabeeinheiten. Diese werten nun die Adresse aus und überprüfen, ob sie angesprochen sind.
3. Der Speicher stellt fest, daß er selektiert ist, und beginnt das „¬Ready“-Signal zu treiben. Da die gelesenen Daten noch nicht verfügbar sind, treibt er das Signal zunächst in den inaktiven Zustand.
4. Das Datenwort 633 wird vom Speicher auf dem Datenbus ausgegeben. Damit wechselt der Datenbus vom hochohmigen in den aktiven Zustand. Anschließend setzt der Speicher das „¬Ready“-Signal in den aktiven Zustand.
5. Die CPU aktiviert das „¬Read“-Signal. Das Datenwort vom Speicher liegt zu diesem Zeitpunkt bereits an.
6. Die CPU überprüft das „¬Ready“-Signal. Falls es noch deaktiv ist, wartet die CPU. Da es bereits aktiv ist, liest die CPU sofort das Wort vom Datenbus und deaktiviert das „¬Read“-Signal. Der adressierte Speicher muß nun das Datenwort vom Bus nehmen.
7. Der Speicher deaktiviert den Datenbus, damit wechselt der Bus wieder in den hochohmigen Zustand. Weiterhin deaktiviert die CPU das „¬CS“-Signal und signalisiert damit das Ende des Lesezyklus.
8. Der Speicher erkennt das Ende des Lesezyklus und hört auf, das „¬Ready“-Signal zu treiben. Damit wechselt das Signal wieder in den hochohmigen Zustand.

3.2 Havard

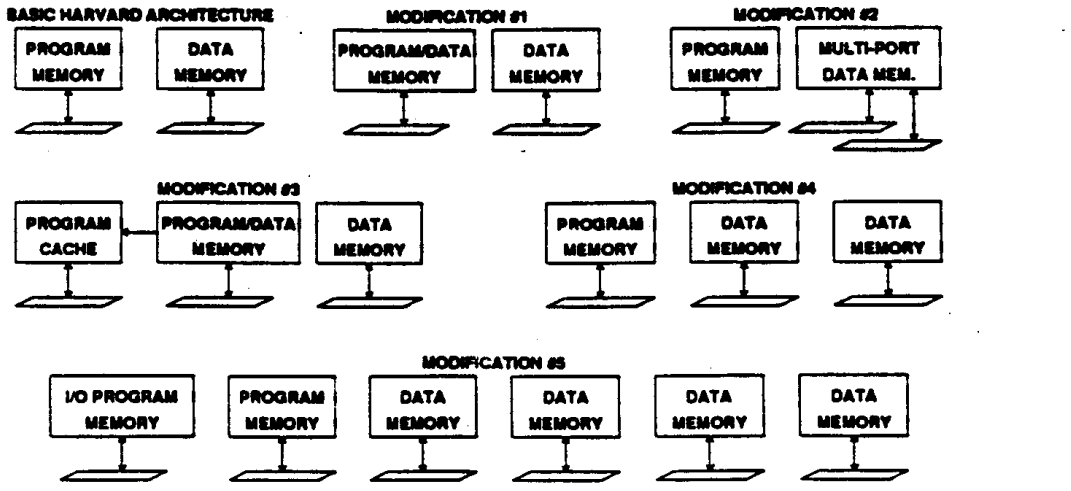
Bild 45. Architekturvergleich



3.2.1 Modifizierte Havard

In der modifizierten Havard-Struktur werden mehrere Bussysteme parallel verwendet. So gibt es bei DSPs of 2 Datenbusse zum Holen oder Speichern zweier Operanden eines Befehls.

Bild 46. Havard Architekturen



3.2.2 DSPs

Die wichtigste Vertreter der Havard-Prozessoren sind DSPs (digital signal processor).
Sie haben einige Besonderheiten, die im folgenden kurz skizziert werden.
Auch wenn DSP's noch immer Kontrollfluß-Befehle aufweisen, gibt es doch einige Spezialbefehle, z.B. für die Filterung, die nach dem Datenflußprinzip arbeiten. Auch einige graphische Programm-Entwicklungstools arbeiten nur mit der Beschreibung des Datenflusses.

3.2.2.1 Datenflußprinzip

Ein wichtiger Begriff bei der Bearbeitung von Algorithmen ist der **Datenfluss**.
Soll in einem Berechnungsgraphen ein nichtterminaler Operand ermittelt werden, beschreibt der Datenfluss alle auszuwertenden Pfade des Graphen von den relevanten Eingabewerten (terminale Operanden) über Operatoren und berechnete Operanden (nichtterminale Operanden) bis hin zum betrachteten Operand. In anderen Worten ausgedrückt beschreibt der Datenfluss den „Fluss" der relevanten Eingabewerte über Operationen und nichtterminalen Operanden bis hin zum betrachteten Operanden.

Das Operationsprinzip des Datenflusses basiert auf dem Datenfluss der Eingangswerte zu den Operanden. Eine Berechnung erfolgt aufgrund der Verfügbarkeit der Operanden, wobei das Ergebnis wieder nachfolgende Operationen triggert.
Im Datenflussprinzip existiert kein expliziter Kontrollfluss mehr, alle zur Berechnung zugehörigen Operationen stehen à-priori zur Verfügung und kommen bei Verfügbarkeit der Operanden zur Ausführung. Damit entsteht implizit ein dem Datenfluss gleichgerichteter Kontrollfluss.

3.2.2.2 Sättigungsarithmetik

Statt, wie bei normaler 2K-Arithmetik, bei Überlauf oder Unterlaufsituationen einen Vorzeichenwechsel zu bekommen, der das Signal extrem verfälschen würde, wird ein Sättigungsmode angeboten, bei dem bei Überlauf der größte darstellbare Wert eingesetzt wird, bei Unterlauf der kleinste darstellbare Wert. Speziell bei zu bearbeitenden Audiosignalen kommt es so nicht zu extremen Verzerrungen sondern nur zu evtl. tolerablem clipping.

3.2.2.3 Spezielle Adressierungsarten

DSPs arbeiten weitgehend mit indirekter Adressierung über Hilfsregister. Dies ermöglicht kurze Befehle und schnell Zugriffe

3.2.2.4 1 Zyklus Befehle

Die meisten Befehle werden in einem Zyklus ausgeführt

3.2.2.5 Hartverdrahtetes Steuerwerk

DSPs verwenden keine Microprogrammierung

3.2.2.6 Spezialbefehle

Es gibt eine Reihe von Spezialbefehlen für Filterung, FFTs, Sättigungsarithmetik, modulare Adressierung

3.2.2.7 Sehr performante ALU

Die Alu in DSPs ist sehr mächtig und unterstützt insbesondere Multiplikationen in wenigen oder in nur einem Zyklus

3.3 CISC

Klassische von-Neumann Rechner sind CISC-Rechner (complex instruction set computer). Im Laufe der Jahre waren den ursprünglich einfachen Maschinebefehlssätzen immer mehr spezialisierte Befehle hinzugefügt worden, mit dem Ergebnis, daß spezielle Aufgaben und Hochsprachenkonstrukte besser unterstützt werden konnten, der Prozessor insgesamt jedoch aufwendiger und langsamer geworden war. Die Befehlsbreite hatte z.B.zugenommen, damit brauchte die Befehls-Phase mehr Zyklen. Weiter Informationen siehe Befehlssatz-Architektur.

Bild 47. Klassifizierung der Prozessoren

Art des Prozessors / Befehlssatzes	Vorherrschende Architektur
CISC – Complex Instruction Set Computer	
• Mikroprozessoren	Princeton (Von Neumann)
• Mikrocontroller	
• Transputer	MIMD
• DSP – Digitale Signalprozessoren	Harvard
RISC – Reduced Instruction Set Computer	
• Mikroprozessoren	Princeton (Von Neumann)
• Mikrocontroller	
Spezialprozessoren	
• Numerische Co-Prozessoren	Stack-Architekturen / SIMD
• Grafikcontroller / GPU	Princeton / Harvard / SIMD
• I/O-Prozessoren	Meist wie Mikrocontroller

CISC	RISC
Komplexe Instruktionen, Ausführung in mehreren Taktzyklen.	Einfache Instruktionen, Ausführung in einem Taktzyklus.
Jede Instruktion kann auf den Speicher zugreifen.	Nur Lade- und Speicherbefehle greifen auf den Speicher zu.
Kein oder wenig Pipelining.	Intensives Pipelining.
Instruktionen werden von Mikroprogramm interpretiert.	Instruktionen werden durch festverdrahtete Hardware ausgeführt.
Instruktionsformat variabler Länge.	Instruktionen alle mit fester Länge.
Viele Instruktionen und Adressierungsarten.	Wenige Instruktionen und Adressierungsarten.
Die Komplexität liegt im Mikroprogramm.	Die Komplexität liegt im Compiler.
Einfacher Registersatz.	Mehrere Registersätze.

3.4 RISC

3.4.1 Die semantische Lücke (Semantic Gap)

Anfang der 70er Jahre untersuchten Wissenschaftler die Frage, welche Assemblerbefehle von Compilern verwendet werden, um Hochsprachenprogramme auf Maschinenebene zu übersetzen. Dabei ergab sich die Erkenntnis, daß zu einem sehr hohen Prozentsatz nur einfache Assemblerbefehle verwendet, leistungsfähige und komplexe Assemblerbefehle jedoch kaum eingesetzt werden. Diese hat mehrere Ursachen:

- Zum einen wird ein Compiler algorithmisch sehr komplex, wenn für eine Sequenz von Hochsprachenanweisungen überprüft werden muß, ob diese durch einen komplexen Assemblerbefehl ausgeführt werden können. Einfacher ist es, jede Hochsprachenanweisung einzeln in separate einfache Assemblerbefehle zu übersetzen.
- Eine zweite Ursache ergab sich dadurch, daß komplexe Assemblerbefehle teilweise nicht zu den Anweisungen der Hochsprachen passten.

3.4.2 Vorgehen beim Entwurf eines RISC-Prozessors

Die Entwickler von CISC-Prozessoren wollen dem Assembler-Programmierer möglichst mächtige Befehle zur Verfügung stellen, so daß durch einen einzigen Befehl komplexe Verarbeitungsschritte ausgeführt werden können.
Die Entwickler von RISC-Prozessoren verfolgen eine entgegengesetzte Strategie, die sich durch folgendes Zitat von Antoine de Saint-Exupery ausdrückt:

3.4.3 Perfektion:

Perfektion ist nicht erreicht, wenn nichts mehr hinzugefügt werden kann, sondern wenn nichts mehr übrig ist, was man weglassen kann.
Aus dieser Grundhaltung heraus werden folgende Schritte zur Entwicklung eines RISC-Prozessors vorgeschlagen:
1. Analyse der Applikation, um Schlüsseloperationen zu finden (Operationen, welche zu großen Anteilen auszuführen sind)
2. Entwurf eines Datenpfades (Rechenwerks), welches optimal diese Schlüsseloperationen verarbeiten kann.
3. Entwurf von Instruktionen, welche die Schlüsseloperationen effizient auf dem Datenpfad ausführen (Steuerwerk).

4. Weitere Instruktionen nur dann hinzufügen, wenn sie die Verarbeitung der Schlüsseloperationen nicht verlangsamen.
5. Die gleiche Optimierung auf andere Bereiche des Rechners (Cache, Speicher Management, Gleitkommaprozessor, etc.) anwenden.

3.4.4 Kennzeichen von RISC-Prozessoren

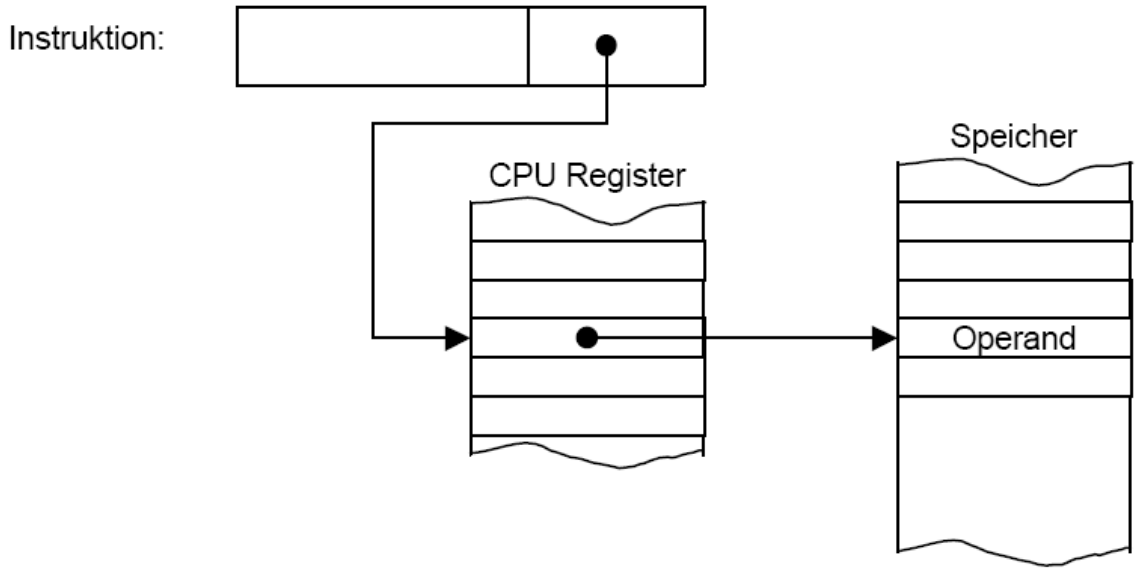
3.4.4.1 Großer Registersatz

Ein RISC-Prozessor besitzt viele Register, um möglichst viele Operanden lokal im Prozessor speichern zu können. Mindestens sind dies 16 Register, eher 32, 64 oder noch mehr Register. Damit werden Operanden lokal gehalten und langsame Zugriffe auf den externen Speicher eingeschränkt.
Bei ALU-Operationen dienen Register als Operanden und als Senke für das Ergebnis. Weiterhin wird nicht mehr zwischen Adress- und Datenregister unterschieden. Damit können komplexe Adressierungsarten per Software realisiert werden.

3.4.4.2 Zwei Adressierungsarten

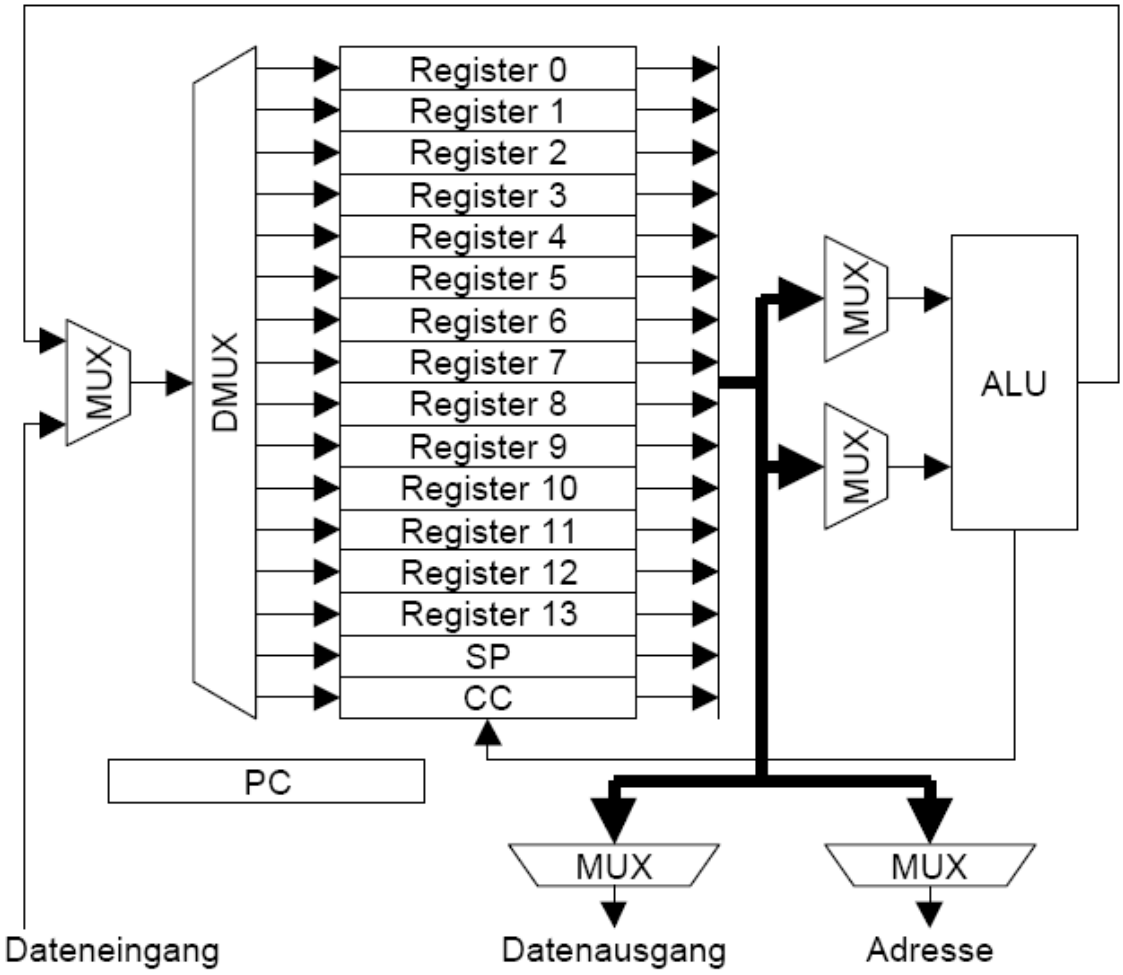
Bei den Lade- und Speicherinstruktionen gibt es nur noch zwei Adressierungsarten. Diese sind:
1. Unmittelbare Adressierung (Immediate). Dann ist der Operand, wie beim Beispielrechner vorgestellt, in der Instruktion selber gespeichert.
2. Register-indirekte Adressierung. Bei dieser Adressierung zeigt die Instruktion auf ein Register intern im RISC-Prozessor. Dieses Register enthält die Adresse des Operanden im Speicher. Untenstehende Abbildung zeigt das Schema dieser Adressierungsart.

Bild 48. Register Indirekt



Mit den genannten Eigenschaften der Register ergibt sich ein Blockschaltbild in der Art, wie es in unten dargestellt ist. Als Besonderheit sind in der Abbildung sogar die Adressregister SP und PC sowie das CC_Register mit in die Registerbank eingeschlossen. Dies variiert bei den RISC-Prozessoren. Üblicherweise gibt es kein explizites Stackregister. Wird ein Stack benötigt, wird dafür eines der Register verwendet. Der PC ist jedoch meist separat ausgeführt.

Bild 49. Adressierung



Man erkennt in der Abbildung deutlich, daß jedes Register als Operand für die ALU dienen kann. Weiterhin kann jedes Register das Ergebnis der ALU-Operation aufnehmen.
Bei Lade- und Speicheroperationen kann jedes Register die Adresse liefern und bei Schreiboperationen kann jedes Register seinen Wert auf den Datenbus ausgeben. Mit dieser Flexibilität können spezielle Adressierungsarten wie „Stack-relativ“ oder „Indirekt“, die nicht mehr im Befehlssatz enthalten sind, einfach und effektiv per Software in den Registern nachgebildet werden.

3.4.4.3 Feste Instruktionslänge

Alle Instruktionen eines RISC-Rechners weisen die gleiche Länge auf und sind in ihrer Struktur einfach aufgebaut. Durch die feste Länge können sie in fester Zeit geholt werden, was für eine Pipeline-Verarbeitung wichtig ist. Durch die einfache Struktur ist eine einfache Dekodierung durch ein Hardware-Steuerwerk möglich.

3.4.5 Beispiel: Instruktionsformat des DLX-Prozessors (Hennessy, Patterson).

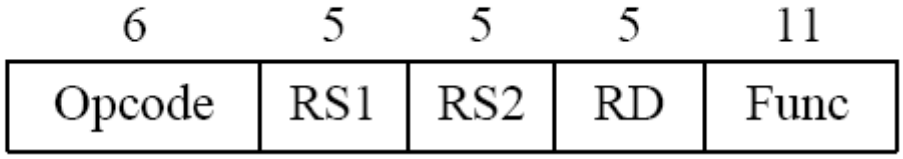
Der DLX-Prozessor ist ein RISC-Beispielprozessor aus dem Lehrbuch von Hennessy und Patterson:
„Computer Architecture A Quatitative Approach“.

Der DLX-Prozessor kennt 3 Instruktionstypen: Den I-Typ, R-Typ und J-Typ. Allen Instruktionen gemeinsam ist, daß sie aus 32 Bit bestehen und in den ersten 6 Bit des Maschinenworts der Opcode kodiert ist.

I-Typ (Immediate)

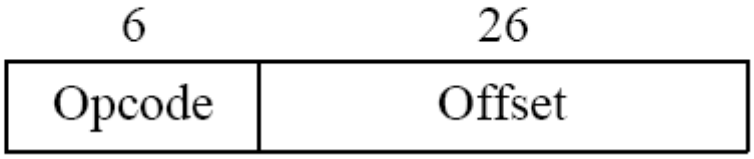
Die I-Typ Instruktionen dienen zum Laden von Registern mit unmittelbarem Wert, zum Laden von Registern aus dem Speicher, zum Speichern von Registerinhalten im Speicher, für bedingte relative Sprünge und für Sprünge mit Register-indirekter Adressierung.
RS1 bezeichnet das Quellregister (Source) z.B. beim Speichern von Registerinhalten. *RD* steht für das Zielregister (Destination).

R-Typ (Register)



Der R-Typ wird für alle Operationen verwendet, welche zwischen Registern ausgeführt werden. Typischerweise sind dies die Arithmetischen, Logischen und die Schiebe-Operationen. *RS1* und *RS2* bezeichnen die beiden Register, welche als Operanden dienen, *RD* bezeichnet das Register, wohin das Ergebnis gespeichert wird. Das Feld *Func* kodiert die ALU-Funktion (z.B. Addieren).

J-Typ (Jump)



Der J-Typ dient für relative Sprünge und für relative Unterprogrammaufrufe. Der in der Instruktion enthaltene Offset wird zum PC addiert.

3.4.6 Eingeschränkter Befehlssatz

Auf den Speicher wird nur über zwei Befehle (z.B. LD und ST) zugegriffen. Es gibt für diese beiden Befehle nur zwei Adressierungsarten:
1. Unmittelbar und
2. Register indirekt.

Alle übrigen Befehle operieren nur auf Registern. Das bedeutet, die Operanden einer Operation müssen in internen Registern verfügbar sein und das Ergebnis der Operation wird wieder in einem internen Register der CPU abgelegt.

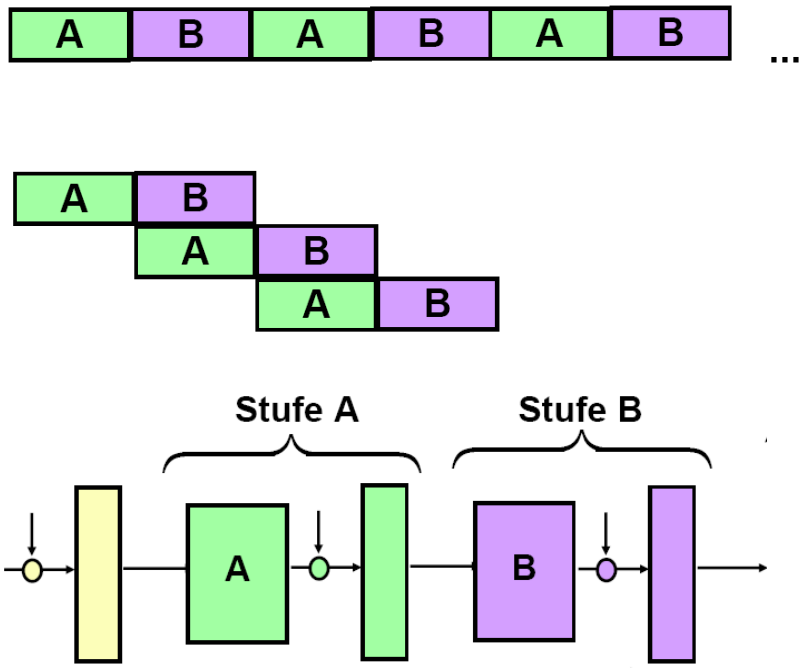
Beispiel: Addierbefehl ADD:
ADD R1, R2,R3
Operand 2
Operand 1
Ergebnis
Vorteil der Verwendung CPU-interner Registerwerte bei der Verarbeitung ist, daß sehr schnell auf diese zugegriffen werden kann. Die Ausführung einer Operation einschließlich Bereitstellung der Operanden kann in einem Taktzyklus ausgeführt werden, da das Holen von Operanden aus dem Speicher entfällt. Komplexe Operationen und Adressierungsarten werden nicht mehr als einzelner, komplexer Maschinenbefehl zur Verfügung gestellt, sondern durch mehrere einfache Maschinenbefehle realisiert.

Man kann somit festhalten, daß bei RISC-Rechnern die Komplexität von der Mikroprogrammebene, in der Mikroprogramme vom Steuerwerk ausgeführt werden, auf die Assemblerebene verschoben wird. Komplexe Befehle und Adressierungsarten, die bei CISC-Rechnern auf der Mikroprogrammebene behandelt wurden, werden nun auf Assemblerebene in Software realisiert.

3.4.7 Einsatz von Pipelining

Will man den Durchsatz und die Taktrate einer synchronen Digitalschaltung erhöhen, so wurde dazu das Prinzip des Pipelining schon bei Addierern vorgestellt. Die gesamte Rechenzeit einer Einzeloperation wird zwar nicht reduziert, es werden jedoch mehrere Operationen versetzt parallel ausgeführt.

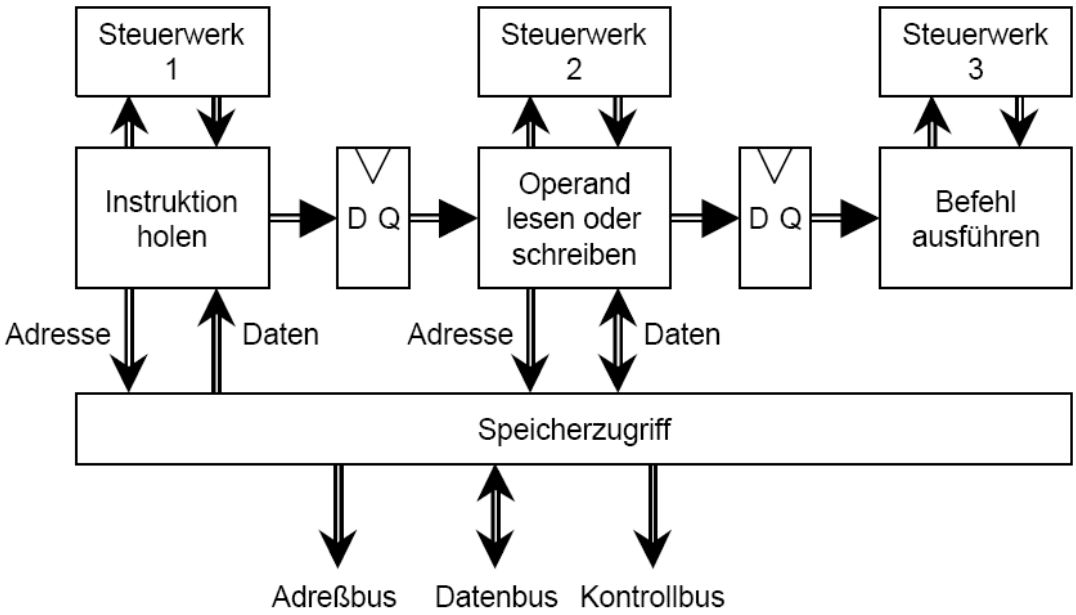
Bild 50. Pipelining



Durch Beschränkung der RISC-Assemblerbefehle auf einfache Befehle mit nur zwei Adressierungsarten kann die Bearbeitung dieser Befehle in Phasen mit fester Verarbeitungszeit (feste Anzahl von Taktzyklen) erfolgen.
Weiterhin kann die Phase zum Lesen oder Schreiben des Speichers hinter die Phase zu Ausführung der Instruktion an das Ende der Pipeline geschoben werden. Die meisten Befehle benötigen diese Phase nicht, da sie nur auf Registern operieren. Nur die Befehlen LD und ST benötigen diese Phase, um Operanden zwischen dem Speicher den Registern zu transferieren. Mit diesen Vorgaben kann die Befehlsbearbeitung beispielsweise in 3 Phasen „Instruktion holen“, „Instruktion ausführen“ und „Speicherzugriff“ zerlegt werden. Diese Phasen lassen sich dann, wie in untenstehender Abbildung gezeigt, auf eine dreistufige Pipeline abbilden. Bei modernen RISC-Prozessoren wird die Befehlsbearbeitung häufig noch feiner zerlegt und entsprechend auf mehr Pipelinestufen (4, 5, ... , 12 Stufen) abgebildet.

Es bietet sich dazu an, die drei Phasen der Verarbeitung jeweils einer Pipelinestufe zuzuordnen. Das Rechenwerk muß dafür natürlich modifiziert werden, da die ALU nicht mehr gemeinsam für Adress- und Arithmetik-Berechnungen verwendet werden kann. Eine eigene ALU muß für Adressberechnungen in Phase 2 und eine zweite ALU für Arithmetik-Berechnungen in Phase 3 vorgesehen werden. Dann kann eine Pipelinestufe für das Holen der Instruktionen, die nächste für das Lesen oder Schreiben des Operanden und die dritte für die Befehlsbearbeitung zuständig sein. Jede Stufe beinhaltet ein eigenes, lokales Steuerwerk, welches jeweils für eine Stufe optimiert ist und eine deutlich geringere Komplexität als das im vorhergehenden Abschnitt notwendige Steuerwerk für den bisherigen Beispielrechner aufweist.

Bild 51. Pipelining



Man erkennt im Blockschaltbild, daß nun ein konkurrierender Zugriff auf den Speicher vom Block „Instruktion holen“ und „Operand lesen oder schreiben“ erfolgt. Somit muß der Speicherzugriff deutlich beschleunigt werden, da von diesen beiden Einheiten versetzt auf den Speicher zugegriffen werden muß.

Bild 52. Pipelining

	Zyklus 1	Zyklus 2	Zyklus 3	Zyklus 4	Zyklus 5	Zyklus 6
Instruktion holen	Instruktion 1	Instruktion 2	Instruktion 3	Instruktion 4	Instruktion 5	
Operand lesen oder schreiben		Instruktion 1	Instruktion 2	Instruktion 3	Instruktion 4	Instruktion 5
Befehl ausführen			Instruktion 1	Instruktion 2	Instruktion 3	Instruktion 4

3.4.7.1 Hazards

Bei direkter Modifikation des Beispielrechners ergeben sich mit diesem Vorgehen jedoch mehrere Probleme:
Ein erstes Problem: Ein erstes großes Problem stellt sich dadurch ein, daß jede Stufe der Pipeline die gleiche Ausführungszeit besitzen sollte. Ansonsten bestimmt die langsamste Stufe den Takt der Pipeline. Beim modifizierten Beispielrechner benötigen die einzelnen Stufen unterschiedliche und sogar variable Rechenzeiten, was ein Pipelining zusätzlich erschwert. In der ersten Stufe wird die Ausführungszeit durch mögliches Vorhandensein eines Folgeworts beeinflusst. In der zweiten Stufe wird die Ausführungszeit in erheblichem Maß durch die gewählte Adressierungsart bestimmt. In der dritten Stufe benötigten einfache Befehle nur einen Zyklus zu ihrer Ausführung, Sprungbefehle belegen hingegen mehrere Taktzyklen.
Ein zweites Problem: Die Sprungbefehle stellen das zweite große Problem bei der Instruktionsbearbeitung mit Pipelining dar, denn bei Ausführung eines Sprungs werden die nachfolgenden Befehle in der Pipeline, mit deren Bearbeitung schon auf Verdacht begonnen wurde, ungültig.
Ein drittes Problem: Soll in aufeinanderfolgenden Instruktionen zunächst ein Operand berechnet und dann weggespeichert werden, so wird das Speichern des Akkumulatorinhalts parallel zur Berechnung des neuen Wertes durch den vorhergehenden Befehl durchgeführt werden. Somit wird nicht der neu errechnete Wert, sondern ein vorheriger alter Wert weggespeichert.
Beispiel: Instruktion 1 sei ein Addierbefehl, Instruktion 2 ein Speicherbefehl. Während Instruktion 2 schon die Stufe „Operand lesen oder speichern“ durchläuft, wird parallel für Befehl 1 das Ergebnis in der Stufe „Befehl ausführen“ ermittelt.
Durch Einsatz von Pipelining entstehen also bei der Architektur des Beispielrechners aufgrund der versetzten Bearbeitung mehrerer Befehle Konsistenzprobleme.
Es soll an dieser Stelle nicht weiter untersucht werden, ob die Architektur des Beispielrechners doch noch irgendwie mit Pipelining beschleunigt werden kann.
Fazit der Betrachtungen ist vielmehr, daß mit der gewählten Beispielrechner-Architektur, welche große Flexibilität durch ihr CISC-Steuerwerks aufweist, große Probleme bei Anwendung von Pipelining entstehen. Es müssen somit andere Architekturen gefunden werden, die einfacher eine Beschleunigung der Verarbeitung durch Pipelining ermöglichen.

3.4.7.2 Hazard-Typen

Die durch die versetzte Ausführung der Instruktionen entstehenden Konflikte werden als Hazards bezeichnet.
Die in den vorhergehenden Abschnitten vorgestellten Konflikte werden wie folgt unterschiedlichen Hazard-Typen zugeordnet:

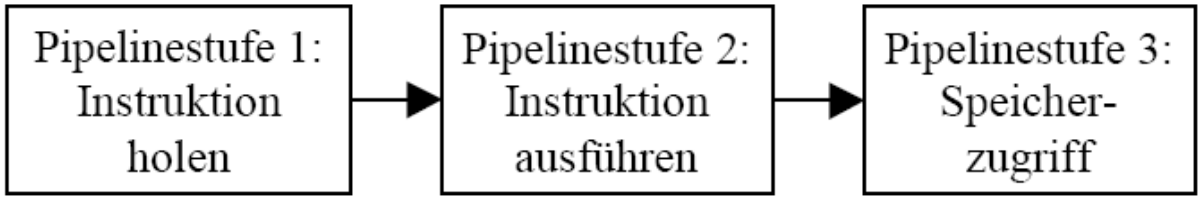
3.4.7.2.1 Struktureller Hazard:
Mehrere Teiloperationen unterschiedlicher Instruktionen müssen auf gleiche Ressourcen zugreifen.

3.4.7.2.2 Datenhazard:
Ergebnisse einer Instruktion liegen noch nicht vor, wenn sie in einer nachfolgenden Instruktion benötigt werden.

3.4.7.2.3 Kontrollhazard:
Instruktionen direkt hinter einem Sprungbefehl werden noch in die Pipeline geladen, bevor der Sprung ausgeführt wird.

3.4.7.3 Weitere Pipeline Beispiele mit Hazards

Bild 53. Pipeline hazards



Mit der Annahme, daß eine Instruktion in jedem Taktzyklus zur Verfügung steht, ergibt sich mit der dreistufigen Pipeline das in folgender Tabelle dargestellte versetzte Ausführen von Instruktionen.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
Instruktion holen	1	2	LD	4	5	6	ST	8	9	10	11	12
Instruktion ausführen		1	2	LD	4	5	6	ST	8	9	10	11
Speicherzugriff					LD	LD	LD		ST	ST	ST	

Es wird angenommen, daß die Instruktionen 1, 2, 4, 5, 6, 8, 9, 10, 11 und 12 nur auf Registern operieren und daher keinen Speicherzugriff ausführen. Die mit LD gekennzeichnete Instruktion soll einen Lesezugriff und die mit ST gekennzeichnete Instruktion einen Schreibzugriff auf den Speicher ausführen. Damit ergibt sich folgende Ausführung von Befehlen:
Instruktion 1: Instruktion 1 wird in Taktzyklus T1 geholt und in Taktzyklus T2 ausgeführt.
Instruktion 2: Parallel zur Ausführung von Instruktion 1 wird im Taktzyklus T2 die Instruktion 2 geholt. In Taktzyklus T2 wird Instruktion 2 dann ausgeführt.
Instruktion LD: In Takt T3 wird die Instruktion LD geholt, dies erfolgt parallel zur Ausführung von Instruktion 2. In Taktzyklus T4 wird LD ausgeführt, was eine Bereitstellung der zu lesenden Adresse bedeutet. Das Lesen des Operanden dauert nun länger, es werden beispielsweise 3 Zyklen in der letzten Pipelinestufe benötigt. Da die Instruktionen 4 und 5 jedoch nur auf internen Registern arbeiten und keinen externen Speicherzugriff durchführen (siehe Annahme), führt dies zu keinem Konflikt.
Zu beachten ist, daß der zu ladende Operand währen der Ausführung von nachfolgenden Operationen noch nicht zur Verfügung steht. Beim Speicherzugriff des Beispielrechners stand der Operand nach dem 2. Zugriffstakt zur Verfügung (siehe Abschnitt „Funktionsweise

der Zentraleinheit“). Nimmt man hier einen gleichartigen Zugriff an, so steht der Operand erst in Taktzyklus T7 und damit bei der Ausführung von Instruktion 6 zur Verfügung.
Instruktionen 4 bis 6: Die Instruktionen 4 bis 6 werden wieder in den Pipelinestufen 1 und 2 parallel versetzt in den Taktzyklen T4, T5, T6 und T7 ausgeführt.
Instruktion ST: Die Speicherinstruktion ST wird in Taktzyklus T7 geholt und in Taktzyklus T8 ausgeführt, was eine Bereitstellung der Speicheradresse und des zu schreibenden Operanden bedeutet. Das Schreiben des Operanden dauert entsprechend wie bei der LD-Instruktion länger, es werden wieder 3 Zyklen in der letzten Pipelinestufe angenommen.
Instruktionen 8 und 9 dürfen wiederum keinen externen Speicherzugriff durchführen, da ansonsten ein Konflikt in der dritten Pipelinestufe entstehen würde. Im Beispiel (Tabelle 6) ist diese Bedingung erfüllt.
Instruktionen 8 bis 12: Die Instruktionen 8 bis 12 werden wieder in den Pipelinestufen 1 und 2 parallel versetzt in den Taktzyklen T8 bis T13 ausgeführt.

Das Assemblerprogramm muß gewährleisten, daß ein geladener Operand erst dann verwendet wird, nachdem der Speicherzugriff erfolgt ist. Im vorliegenden Beispiel darf somit erst in der 3. Instruktion nach dem Ladebefehl auf den geladenen Operanden zugegriffen werden. Dies zu gewährleisten ist Aufgabe des Assembler-Programmierers oder des Compilers, der eine Hochsprache in die Maschinenebene übersetzt.

Beispiel: Unter Speicheradresse 200 sei eine Zahl abgespeichert. Diese Zahl soll um den konstanten Wert 5 erhöht werden. Register R1 und R2 dürfen verwendet werden.
Erster Versuch eines kleinen Assemblerprogramms:

```
1: LD R2,#5       ; Lade R2 mit dem konstanten Wert 5
2: LD R1,#200     ; Lade R1 mit dem konstanten Wert 200 (Adresse der Zahl)
3: LD R1,(R1)     ; Lade Operand aus Sp. (Reg.-indirekte Adressierung mit R1) nach R1
4: ADD R1,R1,R2   ; Zum Wert aus dem Speicher konstanten Wert 5 addieren (Konflikt !!!)
```

Betrachtet man die Ausführung obiger Instruktionssequenz in der Pipeline, so erkennt man den dabei entstehenden Konflikt:

	T1	T2	T3	T4	T5	T6	T7
Instruktion holen	1	2	3	4			
Instruktion ausführen		1	2	3	4		
Speicherzugriff					3	3	3
					Konflikt!!		Operand verfügbar

Der durch Instruktion 3 geholte Operandenwert steht erst in Taktyklus T7 zur Verfügung. Der
Addierbefehl (Instruktion 4) wird jedoch schon in Taktzyklus T5 ausgeführt.

Zweiter Versuch eines kleinen Assemblerprogramms:

```
A: LD R1,#200     ; Lade R1 mit dem konstanten Wert 200 (Adresse der Zahl)
B: LD R1,(R1)     ; Lade Operand aus Sp. (Reg.-indirekte Adressierung mit R1) nach R1
C: LD R2,#5       ; Lade R2 mit dem konstanten Wert 5
D: NOP            ; Nop-Befehl (No Operation)
E: ADD R1,R1,R2   ; Zum Wert aus dem Speicher konstanten Wert 5 addieren
```

Betrachtet man die Ausführung dieser zweiten Instruktionssequenz in der Pipeline, so erkennt man, daß nun der Operand genau zum richtigen Zeitpunkt dem Addierbefehl zur Verfügung steht:

	T1	T2	T3	T4	T5	T6	T7
Instruktion holen	A	B	C	D	E		
Instruktion ausführen		A	B	C	D	E	
Speicherzugriff				B	B	B	
						Operand verfügbar	

Der Speicherzugriff wird so früh wie möglich (Instruktion B) durchgeführt. Anschließend wird das Laden des konstanten Werts 5 (Instruktion C) durchgeführt, zu diesem Zeitpunkt reicht dies noch völlig aus. Da die folgende Instruktion D in ihrer Ausführungsphase noch keinen Zugriff auf den geladenen Operandenwert hat und keine sinnvolle Operation ohne Zugriff auf den Operanden mehr ausgeführt werden kann, wird eine `NOP`-Instruktion (No Operation) eingefügt. Als nächste Instruktion (Instruktion E) kann endlich der Addierbefehl eingefügt werden, er hat in seiner Ausführungsphase Zugriff auf den neu geladenen Operandenwert.
Das Assemblerprogramm muß gewährleisten, daß zwischen nachfolgenden Lade- oder Speicherbefehlen, die auf den externen Speicher zugreifen, so viele Instruktionen ohne Speicherzugriff liegen, daß der erste Speicherzugriff abgeschlossen ist bevor der nächste Zugriff beginnt. Auch diese Anforderung muß vom Programmierer oder vom verwendeten Compiler-Programm berücksichtigt werden.
Beispiel: Zwei Speicherzugriffe in Sequenz.
Der erste Zugriff sei ein Lesezugriff. Die Adresse steht in Register R2, der Operandenwert soll in Register R1 abgelegt werden.
Der zweite Zugriff sei ein Schreibzugriff. Die Adresse steht in Register R3, der Operandenwert in Register R4.

Erster Versuch eines kleinen Assemblerprogramms:
1: `LD R1,(R2)` ; Lade R2 mit dem Operandenwert, dessen Adresse in R2 steht
2: `ST (R3),R4` ; Speichere R4 in die Speicherzelle, deren Adresse in R3 steht
Betrachtet man die Ausführung obiger Instruktionssequenz in der Pipeline, so erkennt man den dabei entstehenden Konflikt:

	T1	T2	T3	T4	T5	T6	T7
Instruktion holen	1	2					
Instruktion ausführen		1	2				
Speicherzugriff			1	1,2	1,2	2	
				Konflikt!!	Konflikt!!		

In den Taktzyklen T4 und T5 muß sowohl Instruktion 1 als auch Instruktion 2 einen Speicherzugriff durchführen.

Zweiter Versuch eines kleinen Assemblerprogramms:
Zur Vermeidung des Konflikts müssen nach der ersten Instruktion mit Speicherzugriff so viele sinnvolle Befehle ohne Speicherzugriff eingefügt werden, daß die zweite Speicher-Instruktion bei der Bearbeitung eine freie Pipelinestufe für den Speicherzugriff vorfindet. Mit der angenommenen 3-stufigen Pipeline müssen zwei Befehle eingefügt werden. Da im Beispiel keine sinnvollen Befehle zum Einfügen bekannt sind, werden NOP-Befehle eingefügt:

```
A: LD R1,(R2)    ; Lade R2 mit dem Operandenwert, dessen Adresse in R2 steht
B: NOP           ; Verzögern des zweiten Befehls mit Speicherzugriff
C: NOP           ; Verzögern des zweiten Befehls mit Speicherzugriff
```

D: `ST (R3),R4` ; Speichere R4 in die Speicherzelle, deren Adresse in R3 steht
Diese Sequenz vermeidet den Konflikt, wie man bei Betrachtung der Pipeline leicht sieht:

	T1	T2	T3	T4	T5	T6	T7	T8
Instruktion holen	A	B	C	D				
Instruktion ausführen		A	B	C	D			
Speicherzugriff			A	A	A	D	D	D
			Speicherzugriff Instruktion A			Speicherzugriff Instruktion D		

In den Taktzyklen T3 bis T5 führt Instruktion A ihren Speicherzugriff durch, in den Zyklen T6 bis T8 schließt sich der Speicherzugriff von Instruktion B direkt an.

3.4.7.4 Sprungbefehle in der Pipeline

Bei der Ausführung von Sprungbefehlen in der Pipeline ergeben sich Probleme, da nachfolgende Instruktionen schon geholt werden und in der Pipeline stehen bevor der Sprung ausgeführt wird.
Für die betrachtete 3-stufigen Pipeline ist das Problem in untenstehender Tabelle dargestellt. Der anvisierte JMP-Sprung wird in Taktzyklus T3 bearbeitet, so daß erst in T4 der neue Wert des Programmzählers vorliegt. Die Instruktion 3 wird somit noch mit dem alten PC-Wert geholt und ist somit Folgeinstruktion des JMP-Befehls. Erst die Instruktion 4 wird von der neuen Position des Programmzählers PC geladen.

	T1	T2	T3	T4	T5	T6
Instruktion holen	1	JMP	3	4	5	
Instruktion ausführen		1	JMP	3	4	5
Speicherzugriff						
			Ermittlung des neuen PC Wertes	PC mit neuem Wert		

Es bieten sich zwei mögliche Vorgehen an, um das Problem zu lösen:

1. Löschen nachfolgender Befehle aus der Pipeline: Die einem Sprungbefehl nachfolgenden Befehle werden durch Hardware dann in der Instruktionspipeline gelöscht, wenn der Sprung durchgeführt wird. Sie werden durch NOP-Befehle ersetzt. Dabei entstehen Blasen (Bubbles) in der Pipeline. Es muß beim Löschen der Befehle sichergestellt sein, daß keinerlei Auswirkungen gelöschter Befehle in der Pipeline zurückbleiben. Nach dem Löschen muß ein Zustand hergestellt werden, der in keiner Weise mehr durch die teilweise bearbeiteten Befehle bestimmt ist.
   Beispiel: Absoluter Sprungbefehl

	T1	T2	T3	T4	T5	T6
Instruktion holen	1	JMP	3	4	5	
Instruktion ausführen		1	JMP	NOP	4	5
Speicherzugriff						

   Nach Ausführung des Sprungbefehls in Zyklus T3 wird Instruktion 3 in der Pipeline gelöscht. Somit erscheint in Taktzyklus T4 ein NOP-Befehl in der Pipeline.

2. Neudefinition des Sprungbefehls: Die dem Sprungbefehl nachfolgenden Befehle in der Pipeline werden noch ausgeführt, bevor der Sprung durchgeführt wird. **(delayed jump).** Bei der betrachteten 3-stufigen Pipeline wird dann der dem Sprungbefehl direkt nachfolgende Befehl noch ausgeführt, bevor der Sprung durchgeführt wird. Es

ist Aufgabe des Assemblerprogrammierers oder des Compilers, an diese Position einen sinnvollen Befehl einzufügen. Steht kein sinnvoller Befehl zur Verfügung, wird ein NOP-Befehl eingefügt.

Beispiel: Unterprogrammaufruf mit einem Parameter: „`hallo(1)`“.
Es wird angenommen, daß der Parameter über Register R5 an das Unterprogramm übergeben wird. Zur Ausführung des Sprungs wird R1 verwendet. Damit ergibt sich die folgende Instruktionssequenz:
1: `LD R1,#hallo` ; Adresse des Unterprogramms laden
2: `JSR R1` ; Unterprogramm anspringen (Register-indirekte Adressierung, PC:=R1)
3: `LD R5,#1` ; Übergabeparameter in R5 laden
Die Befehle werden wie folgt in der Pipeline verarbeitet:

	T1	T2	T3	T4	T5	T6
Instruktion holen	1	JSR	3	4	5	
Instruktion ausführen		1	JSR	3	4	5
Speicherzugriff						

Man erkennt, daß der Sprung zum Unterprogramm in Taktzyklus T3 ausgeführt wird. Somit werden ab Taktzyklus T4 die Instruktionen 4, 5 usw. vom Unterprogramm geholt. In T3 wird die Instruktion 3 geholt, die dem Unterprogrammaufruf JSR folgt. Sie wird in Taktzyklus 4 ausgeführt und lädt im Beispiel den Übergabeparameter.
Es ist wiederum Aufgabe des Assembler-Programmierers oder des Compilers, eine Instruktionssequenz zu erzeugen, welche eine vor dem Sprung auszuführende sinnvolle Instruktion hinter den Sprungbefehl setzt.

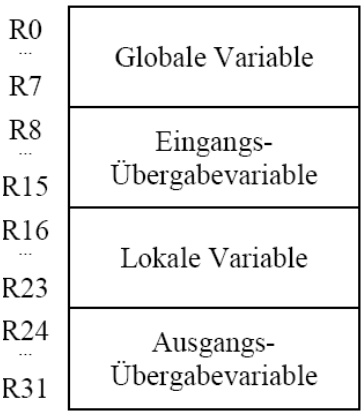
3.4.7.5 Verwendung der Register

Es wurde im vorangegangenen Abschnitt für RISC-Prozessoren eine große Anzahl von Registern gefordert. Es stellt sich nun die Frage, wie diese Register verwendet werden sollen.

Lokale Parameter		Übergabeparameter	
0	22%	0	41%
1	17%	1	19%
2	20%	2	15%
3	14%	3	9%
4	8%	4	7%
≥5	20%	≥5	8%

Eine Analyse der lokalen Variablen von Funktionen und der Übergabeparameter an Funktionen gibt dazu Anhaltspunkte. Aufgrund der Analyse macht es Sinn, möglichst viele lokale Variable statt auf einem externen Stack im Speicher in lokalen Registern der CPU zu halten. Gleiches gilt für die Übergabeparameter von Unterprogrammen.
Bei einem RISC-Prozessor mit 32 internen Registern kann man beispielsweise die unten gezeigte Zuordnung für die Register der CPU wählen.

: **Verwendung CPU internen Register**

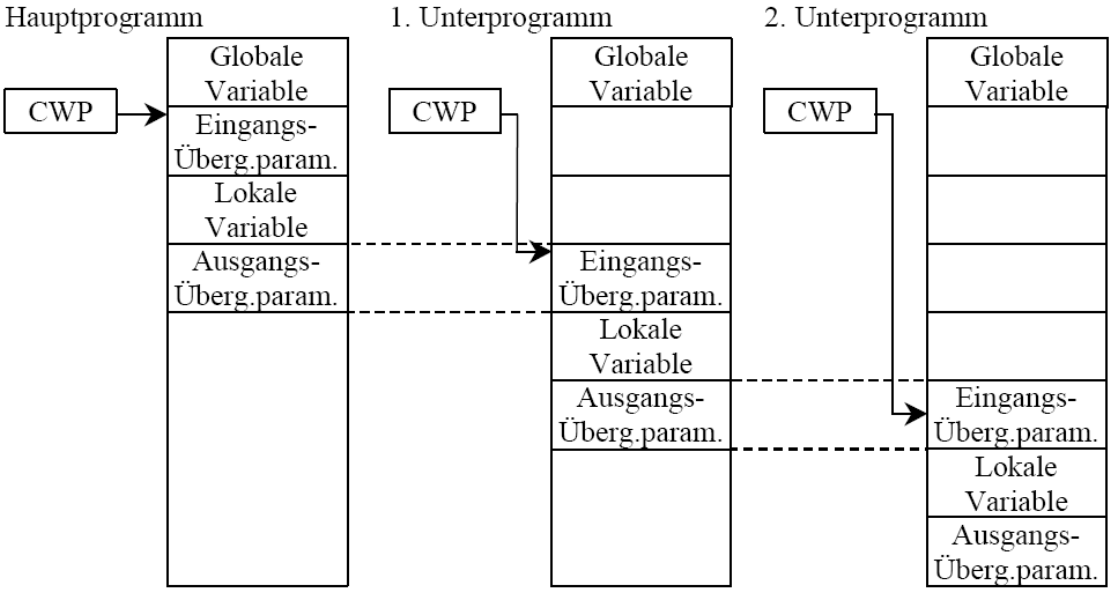


In einem ersten Registerbereich (z.B. R0 bis R7) werden die wichtigsten globalen Variablen abgelegt. In einem zweiten Registerbereich (z.B. R8 bis R15) stehen die Übergabeparameter, die von der rufenden Funktion an die aktuell ausgeführte Funktion übergeben wurden. In einem dritten Bereich (z.B. R16 bis R23) legt die aktuelle Funktion ihre wichtigsten lokalen Variablen an. Schließlich schreibt die aktuelle Funktion beim Aufruf eines Unterprogramms die Übergabeparameter in einen vierten Registerbereich (z.B. R24 bis R31).
Bei Aufruf eines Unterprogramms müssen zunächst die Eingangs-Übergabeparameter und die lokalen Variablen der rufenden Funktion gerettet und die Ausgangs-Übergabeparameter auf die Eingangs-Übergabeparameter kopiert werden.
Beim Rücksprung muß entsprechend umgekehrt der alte Registerzustand der rufenden Funktion wieder hergestellt werden.

3.4.7.6 Überlappende Registerfenster

Attraktiv wird das Vorgehen, wenn das Retten und Umkopieren der Register beim Unterprogrammaufruf entfällt. Dies ist bei RISC-CPUs möglich, welche *überlappende Registerfenster* (Overlapping Register Windows) bereitstellen.

Bild 54. Overlapping register



Prozessoren mit dieser Eigenschaft besitzen eine sehr große Anzahl von Registern (128, 256, 512, ...), von denen aber immer nur ein Teil (z.B. 32 Register) für die CPU sichtbar sind.

Über einen Zeiger CWP (CWP: current window pointer) wird die Basisadresse der Eingangs-Übergabeparameter, lokale Parameter und AusgangsÜbergabeparameter festgelegt. Der Registerbereich der globalen Variablen bleibt immer an der gleichen Position sichtbar.
Obige Abbildung zeigt im Beispiel, wie sich das Registerfenster bei zwei Unterprogrammaufrufen verschiebt. Der CWP-Zeiger wird so verstellt, daß die vom rufenden Programm als Ausgangs-Übergabeparameter sichtbaren Register für das gerufene Programm die Eingangs-Übergabeparameter werden. Damit ist keinerlei Sichern und Umkopieren von Registerwerten mehr notwendig. Die Parameter erscheinen beim gerufenen Programm an der richtigen Stelle und der notwendige Platz, um lokale Parameter und Ausgangs-Übergabeparameter anzulegen, steht zur Verfügung.
Beim Rücksprung aus dem Unterprogramm muß der CWP wieder auf den Wert des rufenden Programms zurückgestellt werden.

3.4.8 Fazit

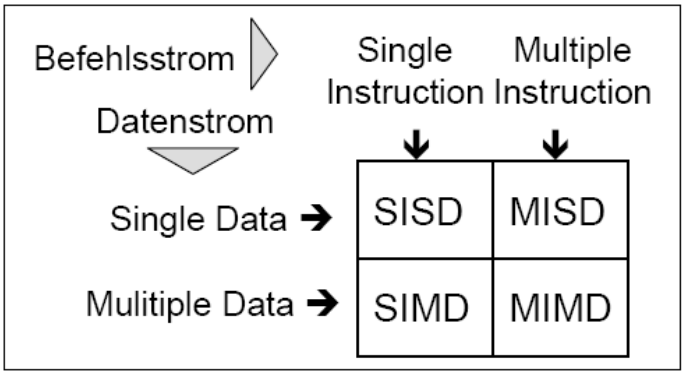
- Bei RISC-Prozessoren werden die häufig vorkommenden Anweisungen von Hochsprachenprogrammen möglichst effektiv auf einfache Maschinenbefehle abgebildet.
- Bis zu zweifache Leistung wie CISC bei gleicher Technologie („Daumenregel“, manchmal werden auch höhere Werte genannt).
- Es werden einfache, schnell zu verarbeitende Maschinenbefehle geschaffen, welche sich gut für Pipelining eignen. Kennzeichen sind: Feste Instruktionslänge und feste Bearbeitungszeit.
- Die Ansteuerung des Rechenwerks erfolgt durch festverdrahtete Logik, die ggf. hinterliegenden Zustandsautomaten sind einfach.
- Die Komplexität der Mikroprogramme bei CISC-Rechnern wird bei RISC-Prozessoren auf die Maschinenebene gehoben.
- Zur Minimierung von Speicherzugriffen wird eine große Anzahl CPU-interner Register vorgesehen. Mechanismen wie überlappende Registerfenster erlauben die Verwaltung auch großer Registeranzahlen und den schnellen Aufruf von Unterprogrammen.
- Compiler für RISC-Prozessoren müssen neben der Umsetzung von Hochsprachen auf Maschinenbefehle auch für die korrekte Beschickung und effektive Ausnutzung der Befehlspipeline sorgen.

3.5 Architekturen nach Befehls- und Datenstromstruktur

M.J. Flynn (1966) betrachtet die Anzahl der gleichzeitig zu verarbeitenden Daten und gleichzeitig ausführbaren Instruktionen und unterscheidet bei Rechnern genau vier Grundtypen:

- Single-Instruction, Single Data (SISD): Es gibt nur einen sequentiell abgearbeiteten Befehlsstrom und einen entsprechenden sequentiellen Datenstrom.
- Single-Instruction, Multiple Data (SIMD): Ein einziger sequentiell abgearbeiteter Befehlsstrom steuert einen mehrfachen parallelen Datenstrom.
- Multiple-Instruction, Single Date (MISD): Hier würden mehrere Steuerwerke einen einzigen Datenstrom steuern.
- Multiple-Instruction Multiple Data (MIMD): Hier geschieht die Abarbeitung sowohl befehls- als auch datenparallel.

Bild 55. Architekturen



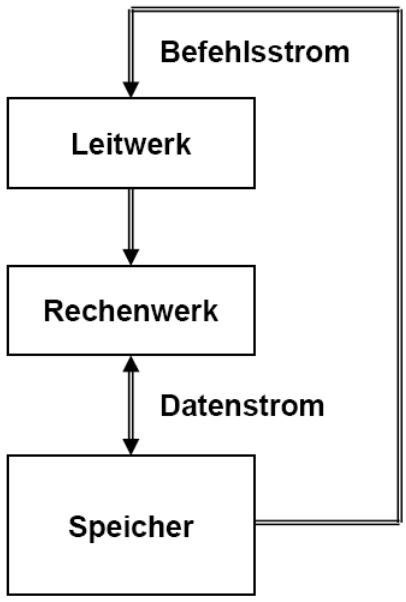
Literaturhinweise:
Flynn, M.J.: Computer Architecture. Jones and Bartlet, 1995

Flynn, M.J.: Some Computer Organizations and their Effectiveness. IEEE Transactions on Computers 21, 1972, pp 948 - 960

3.5.1 SISD-Rechner

Zu den SISD-Rechnern zählen alle Einzelrechner, die eigenständig zur Lösung ihrer Aufgaben eingesetzt werden, also
• CISC- und RISC- Rechner
• die meisten Mikroprozessoren (Intel-Serie bis mindestens 80486)
• fast alle Mainframe-Computer
• Fast alle Minicomputer der 80er Jahre.
• Pipeline-Prozessoren werden allgemein dazugezählt.
Bei einem "echten" SISD-Rechner werden auch die I/O-Vorgänge vom Prozessor selbst erledigt oder allenfalls durch intelligente Controller unterstützt.

Bild 56. SISD



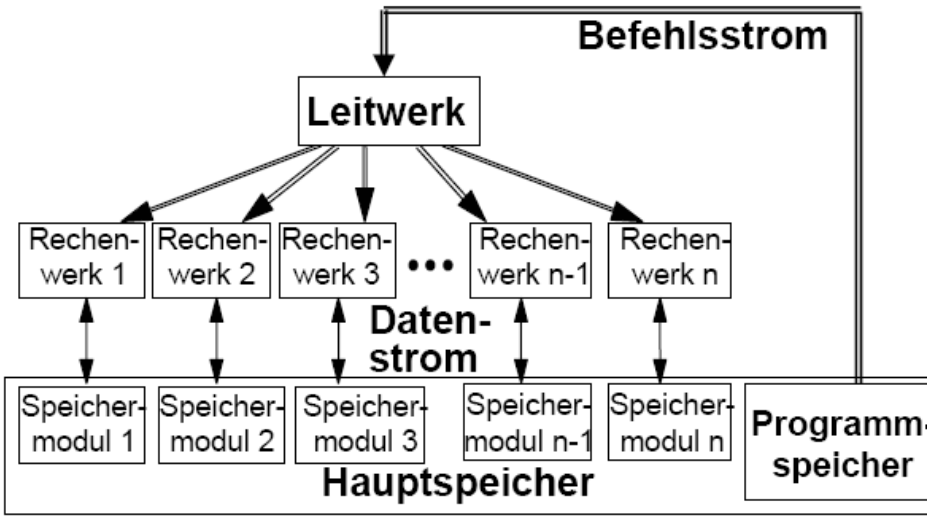
Informationsfluss beim SISD-Rechner

3.5.2 SIMD-Rechner

Es gibt nur einen Befehlsstrom, der mehrere parallele Datenströme steuert.

- Es existiert nur ein Leitwerk zur Befehlsentschlüsselung.
- Es werden parallel mehrere Rechenwerke benutzt. Das Verteilen der Befehlsinformation an die einzelnen Einheiten nennt man „Instruction Broadcasting".
- Jede Verarbeitungseinheit verfügt über einen eigenen bidirektionalen Zugriffspfad zum Hauptspeicher. Damit liegt eine Erweiterung der von Neumann Struktur vor.

Bild 57. SIMD



Beispiele für SIMD-Rechner sind:
• **Vektorrechner**, wenn der Vektorrechner parallele Rechenwerke mit parallelem Speicherzugriff besitzt. Werden die einzelnen Komponenten des Vektors dagegen nur nacheinander auf die Stufen einer einzigen Pipeline projiziert, so ist die Maschine ein SISD-Rechner.
• **VLIW-Prozessoren** (very long instruction word processor). Ein einziger Befehl kann typischerweise 5 bis 10 Operatoren haben, die über parallel laufende Funktionseinheiten miteinander verbunden werden. Anwendung: Spezialprozessoren zur Komprimierung und Dekomprimierung von Bilddaten eingesetzt.
• **MMX-Einheit** von Intel-Prozessoren

3.5.3 Neue moderne Architekturen

Neue, moderne Prozessorarchitekturen sind darauf ausgelegt, mehrere Befehle gleichzeitig zu bearbeiten. Sie werden als *Multiple-Issue*-Prozessoren bezeichnet.
Für Multiple-Issue-Prozessoren gibt es zwei Ausprägungen:

- Superskalare Architekturen
- VLIW: Architekturen mit sehr langen Befehlsworten (engl.: Very Long Instruction Word)

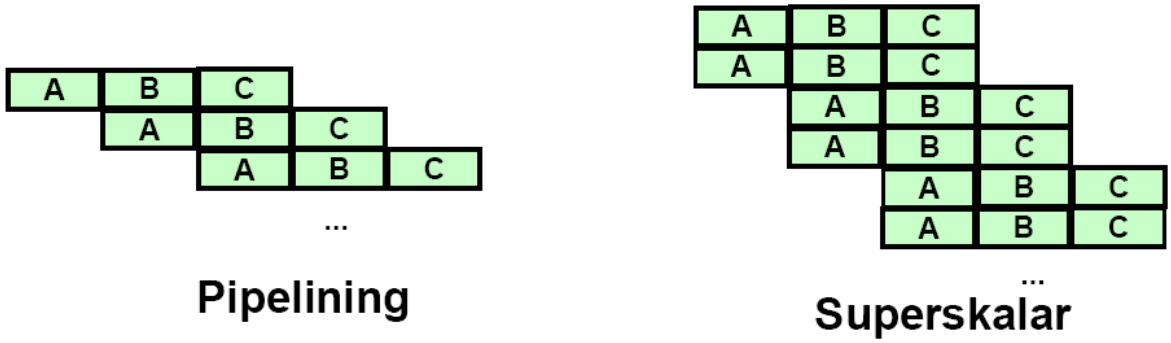
Beide Typen besitzen mehrere, parallele Verarbeitungspipelines (Arithmetik, Gleitkomma, Logik, Multimedia, etc.), in denen parallel Instruktionen ausgeführt werden. Damit lassen sich mehrere Instruktionen pro Taktzyklus ausführen.

3.5.4 Superskalare Prozessoren

Superskalare Prozessoren verarbeiten einen sequentiellen Befehlsstrom. Der nach außen sichtbare Assemblerbefehlssatz (engl.: instruction set architecture, ISA) zeigt keine

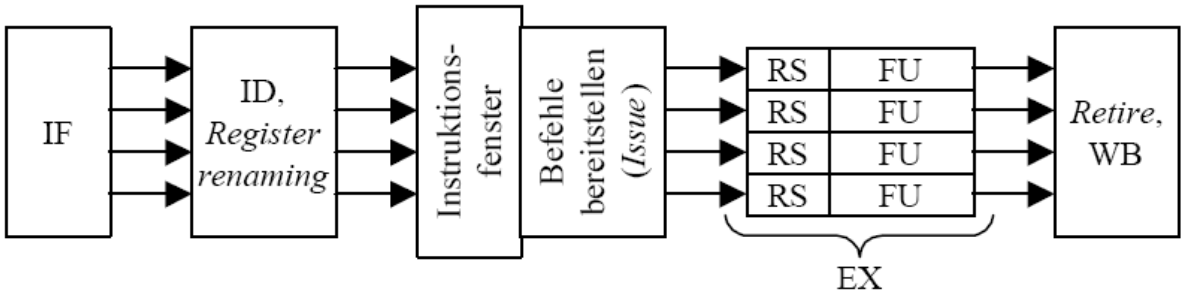
Besonderheiten gegenüber einem einfachen Prozessor auf. Die Algorithmen werden sequentiell kodiert, ohne Rücksicht auf eine parallele Bearbeitung und möglicherweise

Bild 58. superskalar



auftretende Hazards.
Der Unterschied zu einfachen Prozessoren liegt in der Architektur des Prozessors, die auch als
Mikroarchitektur
bezeichnet wird. Die Analyse des sequentiellen Befehlsstroms, die Zuteilung von Instruktionen zu den parallelen Verarbeitungseinheiten und die Auflösung von Hazards erfolgt dynamisch in der Pipeline des superskalaren Prozessors. Dabei verändern manche Prozessoren sogar die Ausführungsreihenfolge der Instruktionen (*Out-of-Order*), so daß nachfolgende, ausführbare Instruktionen schon verarbeitet werden während vorherige Instruktionen aufgrund von Hazards noch auf ihre Operanden warten. Es wird jedoch sichergestellt, daß die Konsistenz der sequentiellen Verarbeitung erhalten bleibt.

Bild 59. Mikroarchitektur



Die Abbildung zeigt die Architektur einer superskalaren Pipeline.

- Im ersten Block IF werden mehrere Instruktionen gleichzeitig geholt.
- Im zweiten Block ID werden diese Instruktionen parallel dekodiert. Zur Eliminierung von Datenabhängigkeiten werden Register umbenannt (engl.: register renaming), damit werden die im Assemblerbefehlssatz verfügbaren logischen Register auf physikalische Register der CPU abgebildet.
- In einem Instruktionsfenster werden Instruktionen gruppiert, die parallel den nachfolgenden Funktionseinheiten bereitgestellt werden (engl.: Issue). Dabei werden strukturelle Hazards eliminiert.
- Die Ausführung der Instruktionen (EX) erfolgt in parallelen Funktionseinheiten (engl.: functional unit, FU).
- Jeder Funktionseinheit sind Pufferspeicher vorgeschaltet, die als *reservation stations* (RS) bezeichnet werden. Hier warten bereitgestellte Instruktionen, daß entweder die Funktionseinheit zur Befehlsausführung bereit wird, oder daß benötigte Ergebnisse vorhergehender Instruktionen zur Verfügung stehen.

- Nach Ausführung einer Instruktion (*completion*) wird das Ergebnis der Operation übernommen (*commitment*) und das Ergebnis für nachfolgende Operationen bereitgestellt (*write back*, WB). Damit ist die Bearbeitung der Operation abgeschlossen (*retired*) und alle Einträge zur Instruktion können gelöscht werden.

Besondere Beachtung benötigen bedingte Sprünge in der Pipeline. Es ist üblich, das Sprungziel zu schätzen und sehr schnell spekulativ einen Zweig des bedingten Sprungs auszuführen. Erweist sich die Schätzung als falsch, werden die Ergebnisse der Instruktionen dieses Zweigs in der letzten Stufe der Pipeline verworfen (kein commitment), sie bleiben somit unwirksam.

Bild 60. Superskalare Prozessorfamilien

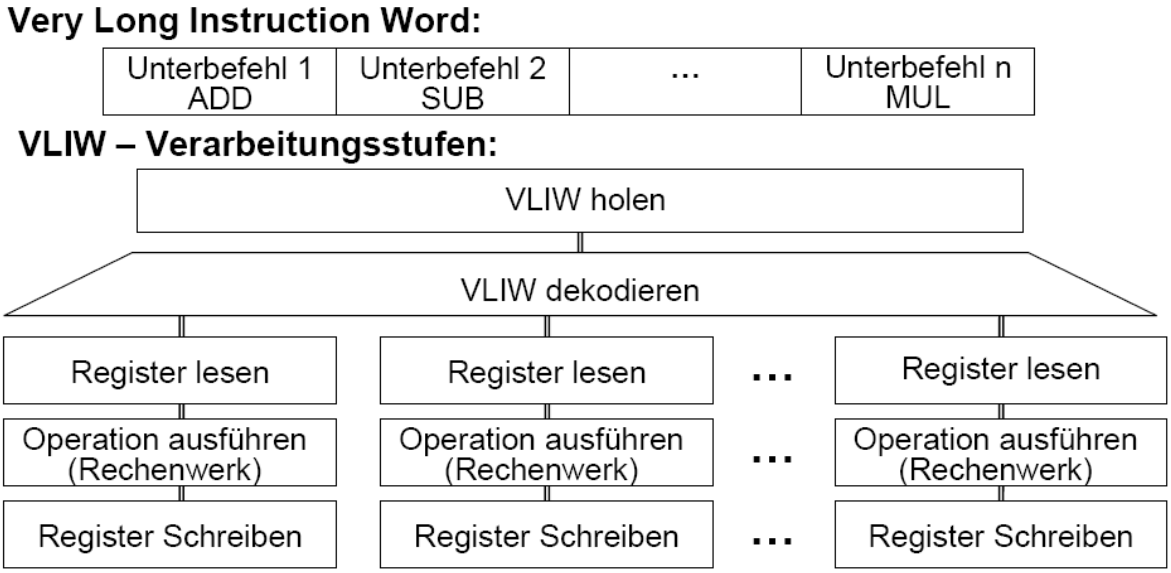
Hersteller	Prozessorfamilien
Intel	Pentium P5, Pentium P6
AMD	K5, K6, K7
Cyrix	M II, M3
Hewlett Packard	PA-7000, PA-8000
Sun	UltraSPARC
DEC	Alpha 21x64
MIPS Technologies	R1x000
IBM, Motorola, Apple	PowerPC

3.5.5 VLIW

VLIW-Prozessoren weisen eine statische Architektur auf. Weitere Eigenschaften sind:

- In einem Instruktionswort werden mehrere Instruktionen kodiert, welche in der CPU parallel ausgeführt werden. Dabei entstehen sehr breite Instruktionsworte (128 bis 1024 Bit).
- Die Befehle werden (im Gegensatz zu superskalaren Prozessoren) vom Compiler statisch zu einem breiten Instruktionswort zusammengestellt. Der Compiler muß dafür sorgen, daß bei der Bearbeitung der Befehle in der CPU keine strukturellen Hazards entstehen.
- Der Compiler analysiert die Datenabhängigkeiten zwischen Instruktionen und stellt diese so zusammen, daß bei der Ausführung keine Datenhazards entstehen.
- Da die Zusammenstellung der Instruktionen durch den Compiler unter Berücksichtigung von Datenabhängigkeiten erfolgt, benötigen VLIW-Prozessoren weder Hardware zum Zusammenstellen von Instruktionen und noch zur Synchronisation von Daten. Dies vereinfacht die Hardware, bedingt aber auch schlechtere Reaktionen auf dynamische Ereignisse (z.B. cache miss, Interrupt) im Vergleich zu superskalaren Prozessoren.
- Die Anzahl der Instruktionen in einem Instruktionswort ist fest. Stehen nicht genügend Instruktionen zur Verfügung, müssen NOP-Befehle aufgefüllt werden.
- Große Speicherbandbreite und umfangreiche Registerbänke erforderlich

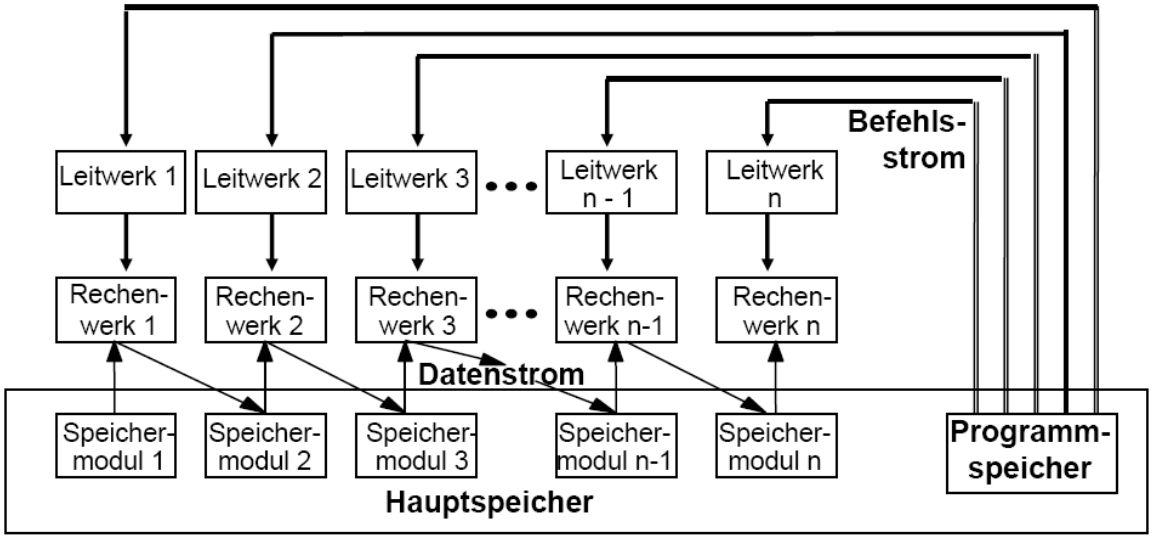
Bild 61. VLIW



Hersteller	Prozessorfamilien
Intel, HP	IA-64
Texas Instruments	TMS320C6x

3.5.6 MISD-Rechner

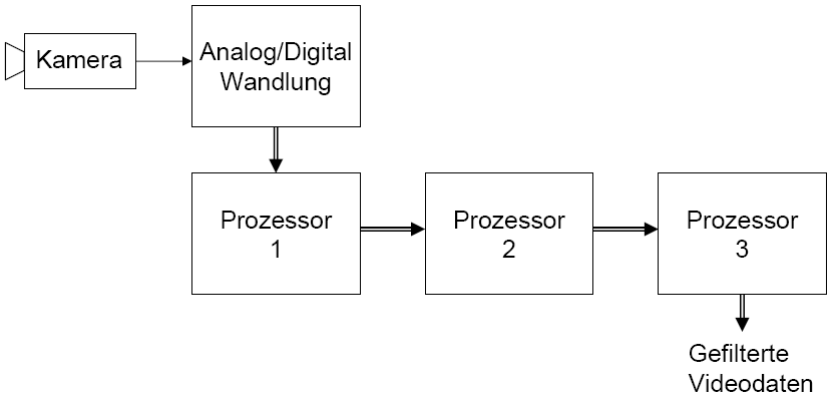
Bild 62. MISD



Mehrere Rechenwerke bearbeiten einen Datenstrom („Fließbandprinzip“)
• Struktur erinnert an Pipelining: Die Gesamtaufgabe wird in aufeinanderfolgende Teilschritte zerlegt. Daher werden die Pipelineprozessoren oft als MISD-Systeme bezeichnet (falls man sie nicht SISD zurechnet).
• Softwaretechnisch umgesetzt ist eine solche Betriebsweise auf Betriebssystemebene beispielsweise in UNIX-Systemen, wenn mittels des "pipe"- Befehls ständig ein Rechenprozess einen anderen mit Daten versorgt.
Filtern von Videodaten

• Zum Filtern von Videodaten wird der Gesamtfilter in aufeinanderfolgende Teilfilter zerlegt
• Jeder Teilfilter wird von einem Pipelineprozessor verarbeitet
• -> höchste Verarbeitungsleistungen bei akzeptablem Hardwareaufwand.

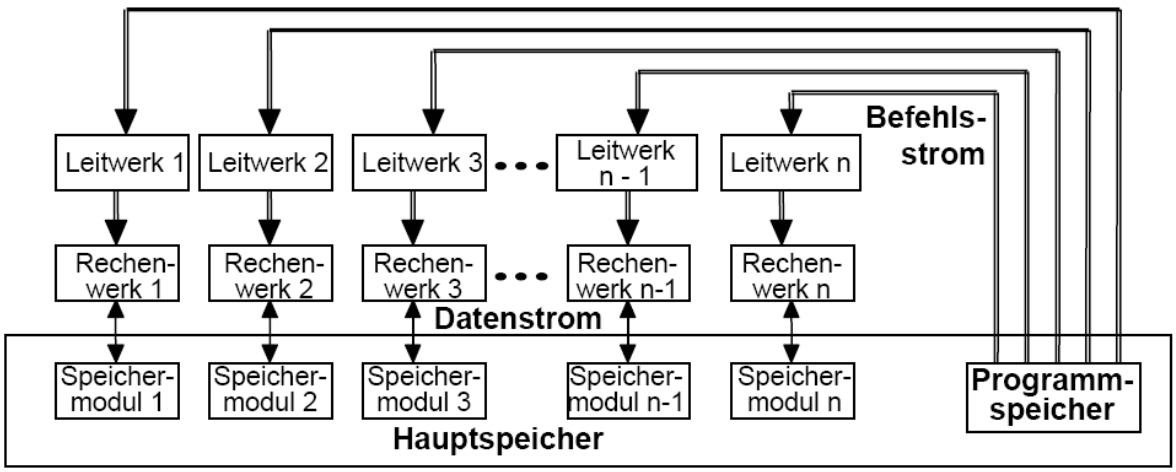
Bild 63. MISD



3.5.7 MIMD-Rechner

- Parallele Bearbeitung von Befehlen und Daten
- Ein Leitwerk und das von ihm gesteuerte Rechenwerk fasst man unter dem Begriff "Prozessor" zusammen
- Anwendung:
  - Ein moderner PC hat mit Video-Karten, Sound-Karten etc. ein erhebliches
  - Maß an Parallelarbeit implementiert.
  - Der Hauptprozessor selbst hat (ab Pentium bzw. AMD K5) ein erhebliches
  - Maß an Parallelverarbeitung.
  - PCs und Server mit mehr als einer CPU sind Stand der Technik.
  - Mehrprozessorsysteme, Parallelrechnersysteme, Supercomputer.

Bild 64. MIMD



4 Rechnerleistung

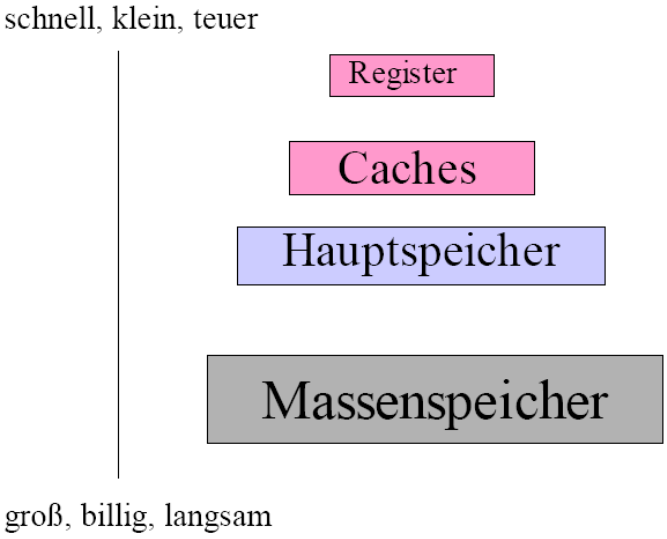
Das wohl bekannteste Leistungsmaß für Rechner ist die MIPS-Rate μ (Millionen von Instruktionen pro Sekunde). Die Definition für μ (MIPS) lautet:
$\mu = 1/ n \cdot t_c$

n: die mittlere Anzahl der Takte (Zyklus) pro Befehl; t_c: Zykluszeit in Mikrosekunden; $1/t_c$ ist die Taktrate. Dies ist jedoch eine einseitige Bewertung, da die Länge der Programme nicht berücksichtigt wird. Die relative Performanz nimmt noch die mittlere Länge eines Programmes in den Nenner.

Achtung:
MIPS hängt vom Befehlsvorrat ab,
hängt vom Compiler ab,
variiert zwischen Programmen auf derselben Maschine,
kann invers zur Leistung sein (!). Beispiel: optimierter Code ist schneller bei weniger MIPS.

5 Speicherhierarchie

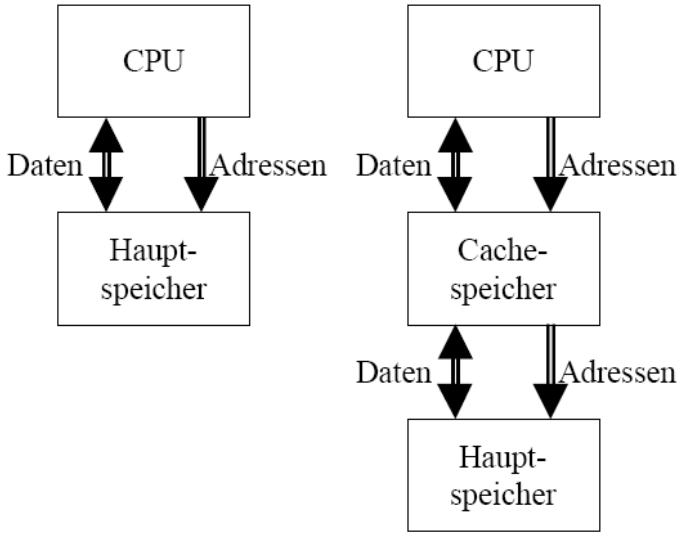
Bild 65. Speicherhierarchie



5.1 Cache-Speicher, die Grundprinzipien

Moderne Prozessoren sind so schnell, daß der Hauptspeicher in der Regel die benötigten Daten nicht schnell genug liefern kann. Um diesem Dilemma zu entkommen, sieht man schnelle Pufferspeicher, sogenannte Caches, vor. Sie befinden sich zwischen Rechnerkern und Hauptspeicher und dienen der Geschwindigkeitssteigerung wie auch der Entlastung des Speicherbusses. Prinzipiell besteht ein Cache aus einem relativ kleinen, aber schnellen Speicher und einer Schaltung (Cache-Controller), die den Zugriff des Prozessors auf den Hauptspeicher überwacht und ihn auf den Cache-Speicher "umleiten" kann

Bild 66. **Cache**



- Zugriff auf Cache ist deutlich schneller als der Zugriff auf Hauptspeicher.
- Wenn Daten im Cache stehen, kann die CPU sehr schnell darauf zugreifen
- Größe des Cache ist deutlich kleiner als Größe des Hauptspeichers

5.1.1 Amdahl's Gesetz

$$\text{Beschl.faktor}_{\text{gesamt}} = \frac{1}{(1 - \text{Anteil}_{\text{beschl.}}) + \frac{\text{Anteil}_{\text{beschl.}}}{\text{Beschl.faktor}}}$$

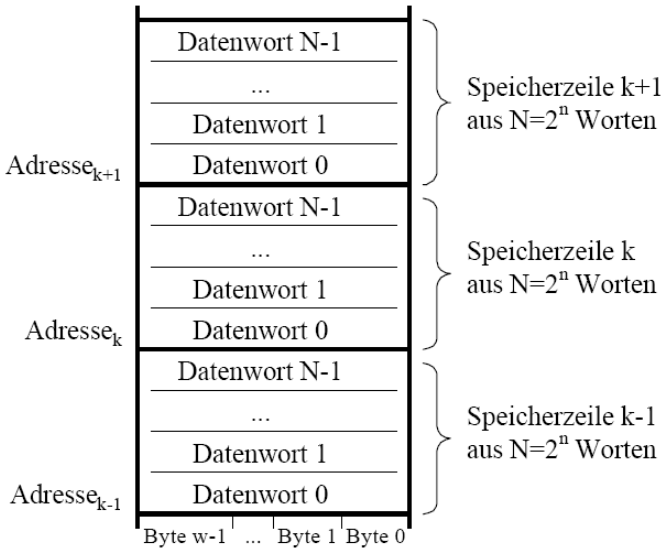
Beispiel: ein Cache ist 10* schneller als der Hauptspeicher. Er kann in 90% aller Speicherzugriffe benutzt werden.

$$\text{Beschl.faktor}_{\text{gesamt}} = \frac{1}{(1 - 0{,}9) + \frac{0{,}9}{10}} = \frac{1}{0{,}19} = 5{,}3$$

5.1.2 Zerlegung des Speichers in Speicherzeilen

Der Hauptspeicher wird in Blöcke von der Größe des Caches unterteilt, und diese Blöcke werden wiederum genauso wie der Cache in sog. Cache-Zeilen (Lines) aufgeteilt. Es werden immer nur ganze Zeilen in den Cache-Speicher übernommen. Dafür, wann eine Zeile in den Cache übernommen oder wieder entfernt wird, existieren verschiedene Strategien.

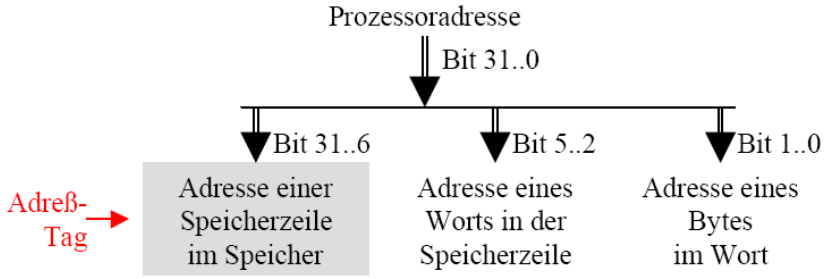
Bild 67. Speicheradressierung



5.1.3 Adressierung

Beispiel: Zerlegung einer 32 Bit Speicheradresse mit *w=2* und *n=4*
Adressbereich 32 Bit: $2^{32} = 4\ 294\ 967\ 296$ *Adressen*
Größe eines Speicherworts: $2^w = 2^2 = 4$ *Bytes*
Größe einer Speicherzeile: $2_n = 2_4 = 16$ *Worte*
Anzahl der Speicherzeilen: $2^{32-n-w} = 2^{26} = 67108864$ *Zeilen*

Bild 68. Speicheradressierung



5.1.4 Gespeicherte Information im Cache

Ein Cache speichert:

- Adress-Tag und
- Kopie der zugehörigen Speicherzeile

Bild 69. Cache Speicher

Cache-Speicher

Adreß-Tag	Datenwort N-1	...	Datenwort N-2	Datenwort 0
0x0100	0x4210	...	0x3344	0x3449
0x2000	0xFE23	...	0xAA33	0xB255
0x1C00	0x5E93	...	0xD1E4	0xCC59
...	...	...	...	...
0x4600	0x5001	...	0xE4E4	0xD453

Cache-Zeile (Line)

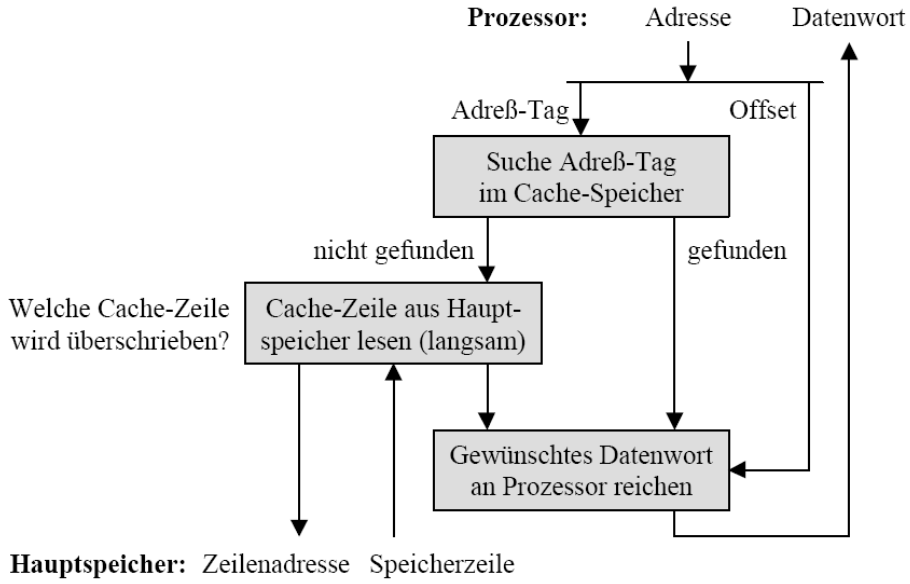
Adressen der Zeilen im Hauptspeicher

Kopien von Speicherzeilen des Hauptspeichers

Soll ein Wort aus dem Hauptspeicher gelesen werden, prüft die Cache-Logik, ob sich die Zeile, die das Wort enthält, nicht bereits schon im Cache befindet. Wenn ja, wird das Wort aus dem Cache ausgelesen und an den Prozessor geliefert ("cache hit"). Wenn nicht, wird das Wort, wie gehabt, aus dem Hauptspeicher gelesen und gleichzeitig die entsprechende Zeile im Cache zwischen gepuffert ("cache miss"), damit sie für einen weitere Zugriff dort schon zur Verfügung steht. Soll ein (verändertes) Wort zurückgeschrieben werden, muß dies sowohl im Hauptspeicher als auch in der entsprechenden Cache-Line geschehen, damit die gespeicherten Datenkopien konsistent bleiben. Beim "write-through cache" wird der Hauptspeicher und gleichzeitig der Cache aktualisiert. Bei anderen Cache-Strategien wird zunächst nur in den Cache zurückgeschrieben und der Hauptspeicher regelmäßig aktualisiert (weniger Zeitverlust).

5.1.5 Algorithmus zum Lesen des Cache-Speichers

Bild 70. Lesen des Caches

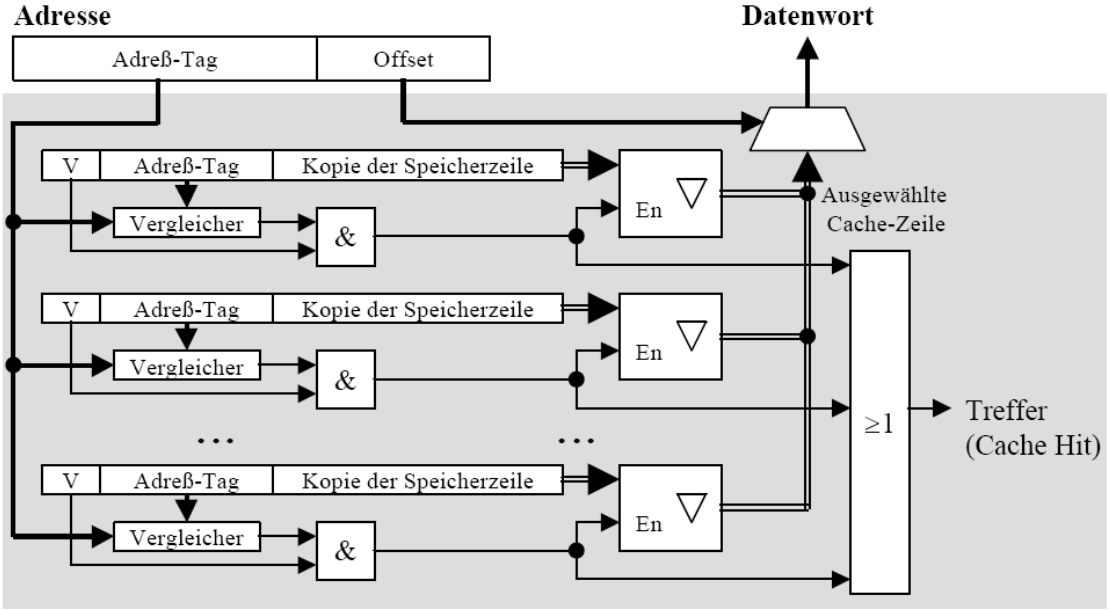


5.2 Cache-Organisation

Ein Cache muß schnell im Zugriff sein, daher ist er klein (und teuer) im Vergleich zum Hauptspeicher. Oft ist er auch prozessorintern im Gegensatz zum externen Hauptspeicher. Zum Finden eines Datenworts im Cache müssen parallele Suchvorgänge bei einer limitierten Anzahl von Einträgen durchgeführt werden. Bei der Suche nach Einträgen existieren verschiedene Strategien:
Voll assoziativ (Fully associative):
Ein Datenwort kann in einem beliebigen Cache-Eintrag abgelegt sein.
Direkt abgebildet (Direct mapped):
Ein Datenwort kann genau in einem Eintrag abgelegt sein.
Mengen-assoziativ (Set associative):
Ein Datenwort kann in wenigen Cache-Einträgen (typischerweise 2 bis 8) abgelegt sein.

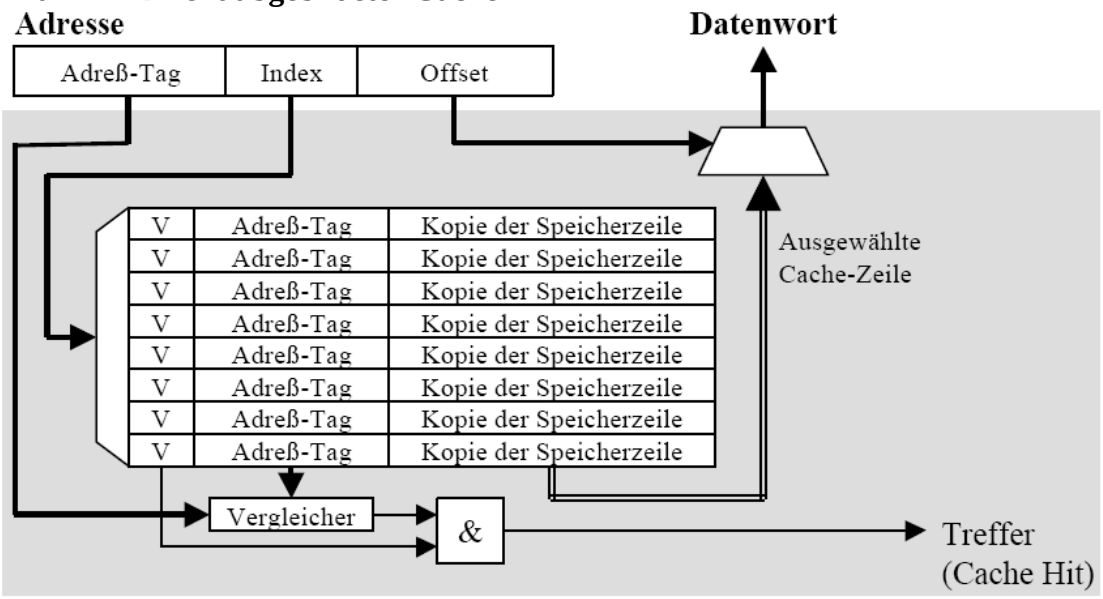
5.2.1 Voll assoziativer Cache

Bild 71. Voll assoziativer Cache



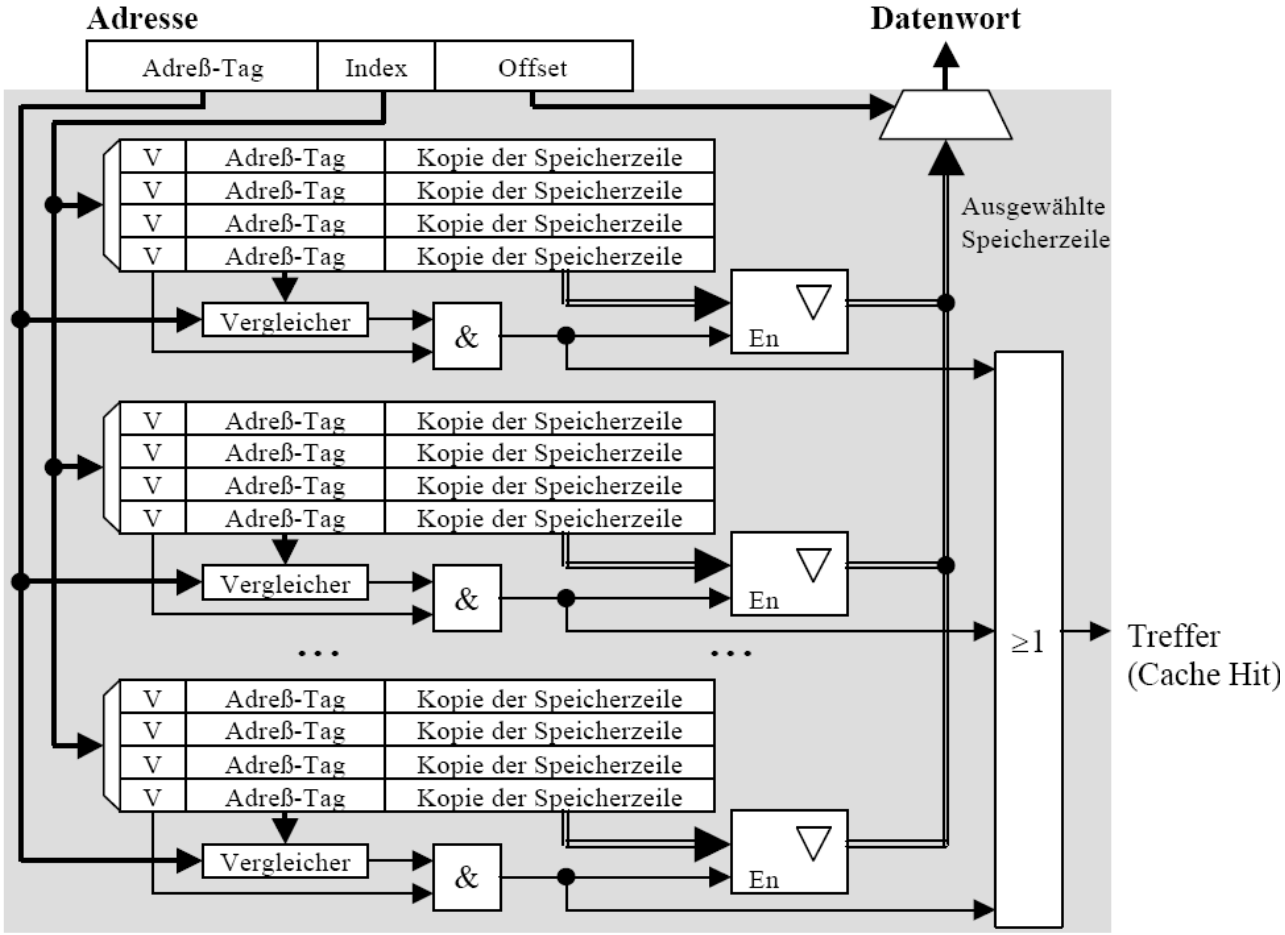
5.2.2 Direkt abgebildeter Cache

Bild 72. Direkt abgebildeter Cache



5.2.3 N Wege Mengenassoziativer Cache

Bild 73. N Wege Mengenassoziativer Cache



5.3 Kenngrößen

5.3.1 Trefferrate (Hit Rate), Fehlrate (Miss Rate)

- Man erhält einen Treffer (Hit) beim Zugriff auf einen Cache, wenn das gesuchte Wort im Cache vorliegt: a: Trefferrate in Prozent.
- Liegt das gesuchte Wort nicht im Cache, muß es aus dem Speicher gelesen werden: (1-a): Fehlrate in Prozent.

5.3.2 Mittlere Zugriffszeit

Die mittlere Zugriffszeit t_a auf ein Wort ergibt sich aus der Zugriffszeit t_c eines Wortes im Cache und der Zugriffszeit t_m eines Wortes im Hauptspeicher:
$t_a = a \cdot t_c + (1-a) \cdot t_m$

5.3.3 Überschreiben von Cache-Zeilen

Ist ein Wort nicht im Cache vorhanden, muß die zugehörige Speicherzeile vom Speicher in den Cache geladen werden.
Frage: Welche Zeile wird im Cache überschrieben?
Direkt abgebildeter Cache: Zuordnung der Adresse des Wortes zu einer Cache-Zeile ist eindeutig. Damit ist die zu überschreibende Zeile eindeutig bestimmt.
Assoziativer Cache: Mehrere Strategien möglich:
1. Wähle Zeile aus, deren Aufnahme in den Cache am längsten her ist.
2. Wähle Zeile aus, auf die am längsten nicht mehr zugegriffen wurde.
3. Wähle zufällig eine Zeile aus.
Interessant ist, daß alle drei Strategien zu weitgehend gleichen Trefferraten führen. Die 3. Strategie (Zufall) verursacht jedoch den geringsten Aufwand im Cache.

5.4 Schreiben von Daten

Beim Schreiben von Daten muß sichergestellt werden, daß die Inhalte von Cache und Hauptspeicher zueinander konsistent bleiben. Es existieren mehrere Strategien, um diese Konsistenz zu gewährleisten:

5.4.1.1 No-Write:

Cache-Speicher nicht schreiben, sondern Wort in Hauptspeicher schreiben. Eine zugehörige Zeile im Cache muß als ungültig markiert werden. Diese Strategie führt zu langsamen Schreibzugriffen. Weiterhin sind nachfolgende Lesezugriffe auf die gleiche Speicherzeile langsam, da der Zeileninhalt nicht mehr im Cache verfügbar ist.

5.4.1.2 Write-Through:

Cache- und Hauptspeicher beide schreiben. Eine zugehörige Zeile im Cache bleibt gültig. Man erhält langsame Schreibzugriffe. Übliches Verfahren.

5.4.1.3 Write-Back:

Nur Cache-Speicher schreiben, aber Cache-Zeile als modifiziert („Dirty“) markieren. Beim Überschreiben der Zeile muß der Inhalt vorher in den Hauptspeicher zurückgeschrieben werden. Zum Markieren wird ein Bit (Dirty-Bit) der Cache-Zeile angefügt. Es wird beim Laden einer Zeile aus dem Hauptspeicher gelöscht, beim Schreiben eines Wortes in der Cache-Zeile gesetzt. Diese Strategie ermöglicht ein schnelles Lesen und Schreiben von Daten. Bei Multiprozessorsystemen mit gemeinsamem Speicher entstehen jedoch Konsistenzprobleme.

5.5 Lesen von Daten

Es gibt zwei prinzipielle Lese- und Füllstrategien:

5.5.1 on demand, demand fetching

Beim ersten Lesen einer Information aus dem Hauptspeicher wird die gesamte Zeile auch in den Cache übertragen, d.h. nicht gecachte Blöcke werden nachgeladen. Diese Strategie wird von jedem Cache verwendet.

5.5.2 Prädiktiv, prefetch

Es wird versucht, vorherzusagen, welche Speicherzeilen zukünftig gebraucht werden und diese während Leerlaufphasen schon in den Cache vorgeladen. So gibt es den Begriff remote PC, mit dem gemeint ist, daß ein zweiter Programmzähler im Cache dazu verwendet wird, den künftigen Programmverlauf vorherzubestimmen und entsprechende Blöcke vorzuladen.

6 Speichermanagement

Bei den bisherigen Rechnerarchitekturen wurde davon ausgegangen, daß die intern im Prozessor verwendeten Adressen identisch mit den externen Speicheradressen sind. Moderne Prozessoren für Arbeitsplatzrechner entkoppeln die CPU-internen Adressen von den Speicheradressen.

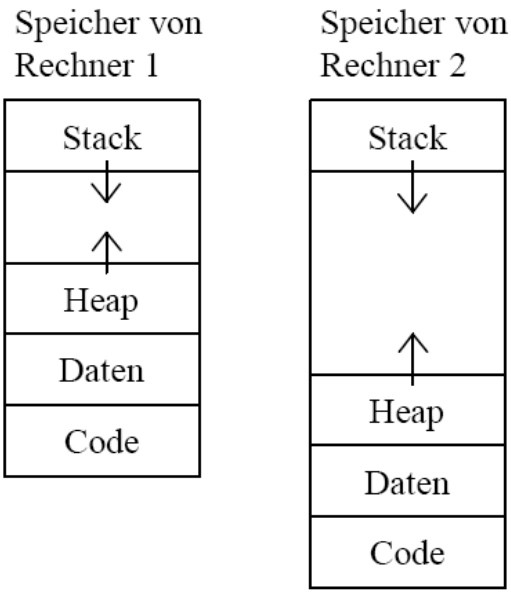
6.1.1 Nachteile der Kopplung CPU-interner Adressen mit Speicheradressen

Die Kopplung der CPU-internen Adressen mit den externen Speicheradressen hat einige Nachteile. An zwei Fällen sollen diese Nachteile verdeutlicht werden.

6.1.1.1 Nachteil 1: Anpassung der Speicherbelegung eines Programms an die Konfiguration eines jeweiligen Rechners.

Ausführbare Programme bestehen aus mehreren elementaren Segmenten. Diese elementaren Segmente werden als Code-, das Daten-, das Stack- und Heap-Segment bezeichnet. Die Größe des Code- und des Datensegments wird beim Übersetzen (Compilieren) eines Programms ermittelt. Das Stack- und das Heap-Segment besitzen jedoch dynamische Größen, die vom
jeweiligen Programmablauf abhängen. Es besteht somit der Wunsch, möglichst viel Speicherbereich für diese beiden Segmente vorzuhalten, so daß auch bei Programmläufen mit großem Speicherbedarf kein Speicherunterlauf stattfindet.

Bild 74. Speicherverwaltung



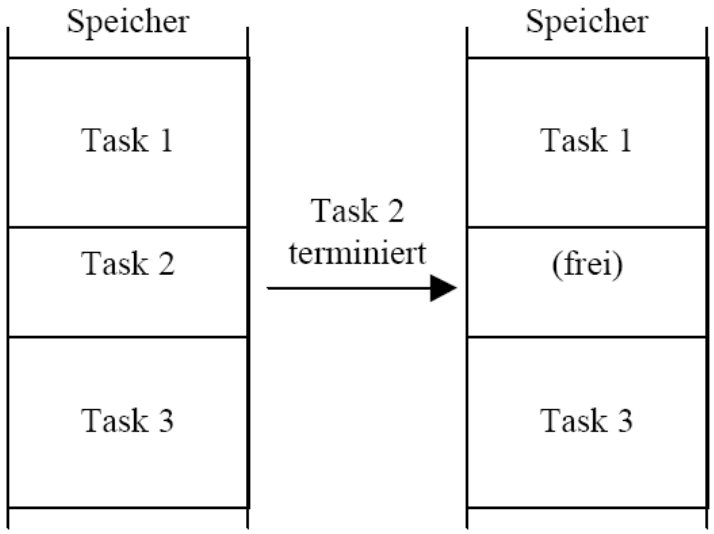
Die übliche Aufteilung der Segmente ist in obiger Abbildung dargestellt. Code- und Datenbereich haben feste Größe und sind in festen Speicherbereichen angelegt. Heap- und Stack-Segment sind einander gegenüber mit einem zwischenliegenden freien Bereich angeordnet, so daß sowohl das Stack- als auch das Heap-Segement sich in den freien Bereich hinein vergrößern können.
Zur Ausnutzung des Speichers eines Systems ist es nun wünschenswert, daß beim Start eines Programms der freie Bereich zwischen Stack und Heap möglichst groß gewählt wird. Bei fester Zuordnung zwischen internen und externen Adressen muß daher ein an der Speichergröße des aktuellen Rechners orientiertes Linken (d.h. Zuweisen der Absoluten Adressen) des Programms durchgeführt werden, um den vorhandenen Speicher auszunutzen. Damit entsteht eine speziell auf einen Rechner abgestimmte Speicheraufteilung, wie dies in der Abbildung angedeutet ist.
Bei Entkopplung der CPU-internen Adressen von den Speicheradressen kann hingegen zwischen Stack- und Heap-Segement ein großer und fester Bereich des Adressraums vorgesehen werden, der nur bei Bedarf und in den benötigten Bereichen auf vorhandene Speicheradressen abgebildet wird.

6.1.1.2 Nachteil 2: Zuweisung eines speziellen Speicherbereichs an jede Task bei Multitasking.

Bei Multitasking muß beim Start einer Task ein freier Speicherbereich gesucht und die Task an diesen Speicherbereich durch Linken angepasst werden. Jede Task benötigt einen eigenen

Ausschnitt aus dem gesamten Adressraum, der nicht von anderen Tasks verwendet werden darf.

Bild 75. Speicherverwaltung



Ein Problem stellt sich nun, wenn eine Task terminiert (siehe obige Abbildung). Es ist keinesfalls sichergestellt, daß zu einem späteren Zeitpunkt eine Task mit gleichen Speicheranforderungen gestartet wird, die genau in die Lücke der terminierten Task passt. Somit besteht immer das Problem, daß ein Speicherbereich nicht mehr in der zuerst reservierten Form weiterverwendet werden kann. Selbst wenn ein Teil des freien Bereichs durch eine andere Task ausgefüllt wird, bleiben ungewollte Speicherfragmente zurück.

6.1.1.3 Fazit

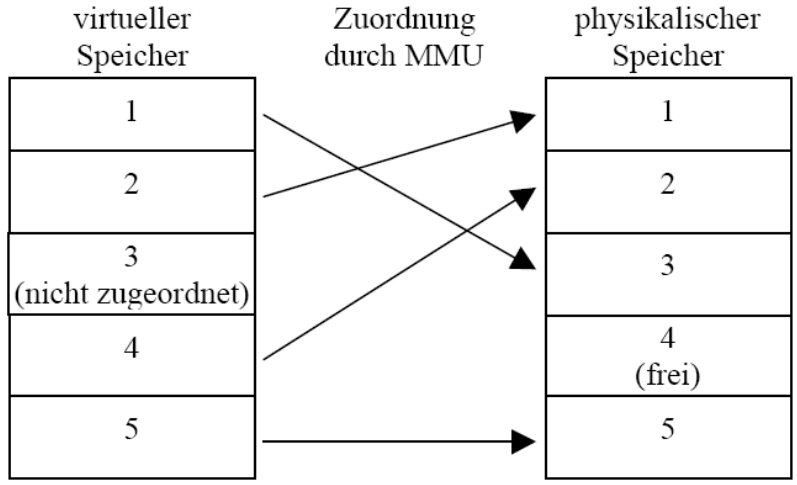
An den Beispielen erkennt man, daß durch die Kopplung von CPU-internen Adressen und Speicheradressen Abhängigkeiten entstehen, welche die flexible und dynamische Verwendung des Hauptspeichers verhindern. In Systemen, wo diese Dynamik und Flexibilität nicht gefragt ist (z.B. Embedded Systeme) ist dies akzeptabel. Im Workstation- und PC-Bereich ist jedoch eine Entkopplung interner und externer Adressen unbedingt notwendig.

6.2 Virtuelle und physikalische Adressen

Durch Entkopplung der CPU-internen Adressen von den Speicheradressen gewinnt man Flexibilität in der Vergabe der Adressen. Die CPU-internen Adressen werden nun als *virtuelle Adressen*, die Speicheradressen als *physikalische Adressen* bezeichnet.
Die Umsetzung der virtuellen auf die physikalischen Adressen erfolgt durch eine Hardwareeinheit, die als *Memory Management Unit* (MMU) oder auch als *Paging Unit* bezeichnet wird.
Zur Übersetzung der virtuellen Adressen auf physikalische Adressen wird sowohl der virtuelle als auch der physikalische Speicherbereich in Speicherseiten fester Größe aufgeteilt. Speicherseiten werden im englischen als „pages“ und im deutschen manchmal auch als „Kacheln“ bezeichnet. Seitengrößen entsprechen einer 2er Potenz.

Übliche Seitengrößen sind 4 kBytes oder 8 kBytes, aber auch deutlich größere Seiten werden verwendet. Virtuelle Speicherseiten werden durch die MMU physikalischen Seite zugeordnet. Ist eine virtuelle Seite keiner physikalischen Seite zugeordnet, erzeugt die MMU üblicherweise einen Interrupt, so daß die CPU dieses Ereignis behandeln kann. Untenstehnde Abbildung zeigt ein Beispiel für die Zuordnung virtueller Adressen zu physikalischen Adressen. Man erkennt, daß die virtuelle Seite 3 keiner physikalischen Seite zugeordnet ist. Ein Zugriff des Prozessors auf diese Seite hätte somit eine Ausnahmebehandlung (Interrupt) zur Folge.

Bild 76. Virtuelle Adressierung



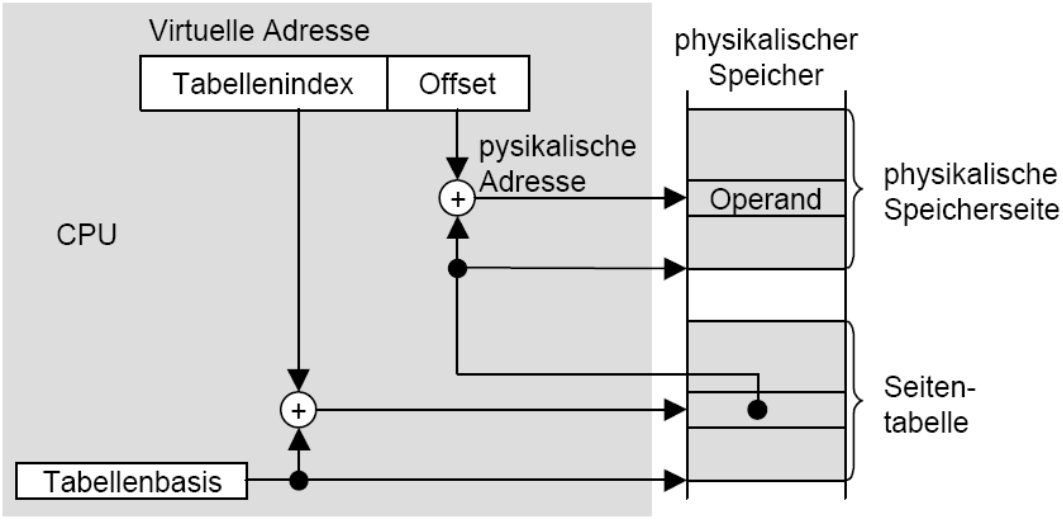
Im physikalischen Speicher ist im Beispiel die Speicherseite 4 frei. Sie kann bei Bedarf einer virtuellen Seite zugeordnet werden.
Man erkennt, daß durch die freie Zuordnung von virtuellen zu physikalischen Seiten ein physikalisch fragmentierter Speicher als kontinuierlicher virtueller Speicher dargestellt werden kann. Damit können freigegebene Speicherbereiche beim Start neuer Tasks gemäß der neuen Anforderungen zu neuen virtuellen Speicherbereichen zusammengebaut werden.

6.2.1 Umsetzung virtueller Adressen auf physikalische Adressen

6.2.1.1 Prinzip der Umsetzung

Die Umsetzung virtueller Adressen auf physikalische Adressen durch die MMU wird im englischen als
Paging bezeichnet. Die Umsetzung erfolgt durch Speichertabellen und durch Hardware.

Bild 77. Virtuelle Adressierung

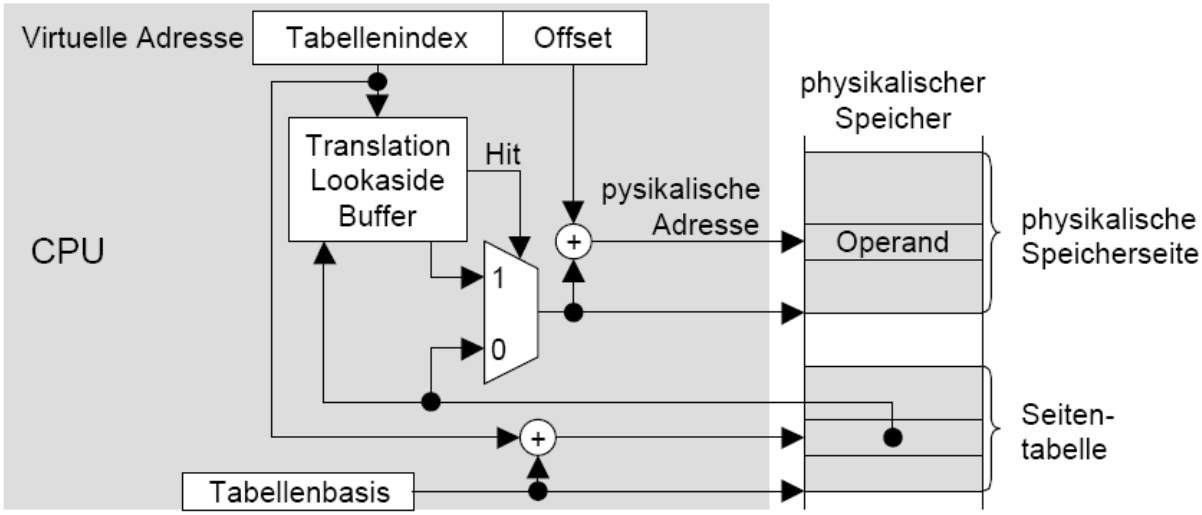


Die Abbildung zeigt das Prinzip, nach dem die MMU die virtuellen Adressen in physikalische Adressen umsetzt. Die virtuelle Adresse wird in zwei Felder zerlegt. Das untere Feld adressiert den Operanden innerhalb einer Speicherseite und ist mit *Offset* bezeichnet. Das obere Feld ist mit *Tabellenindex* bezeichnet. Es wird von der MMU in eine physikalische Speicherseite umgesetzt. Dazu greift die MMU auf eine Seitentabelle zu, die im physikalischen Speicher liegt. Die Basisadresse der Seitentabelle ist in der MMU in einem Register (Tabellenbasis) abgelegt. Auf die Seitentabelle wird indiziert mit dem Tabellenindex zugegriffen und die Basisadresse der physikalischen Speicherseite ausgelesen. Nun dient der Wert des Offset-Feldes aus der virtuellen Adresse als Index in die physikalische Speicherseite. Die Summe aus der Basisadresse der physikalischen Speicherseite und dem Offset bilden die Adresse des Operanden. In realen Prozessoren wird der Tabellenindex in mehrere Felder zerlegt und über mehrere Tabellenzugriffe die Basisadresse der physikalischen Speicherseite ermittelt. Das Prinzip ist jedoch das gleiche wie das in der Abbildung dargestellte Prinzip.

6.2.1.2 Beschleunigung der Umsetzung

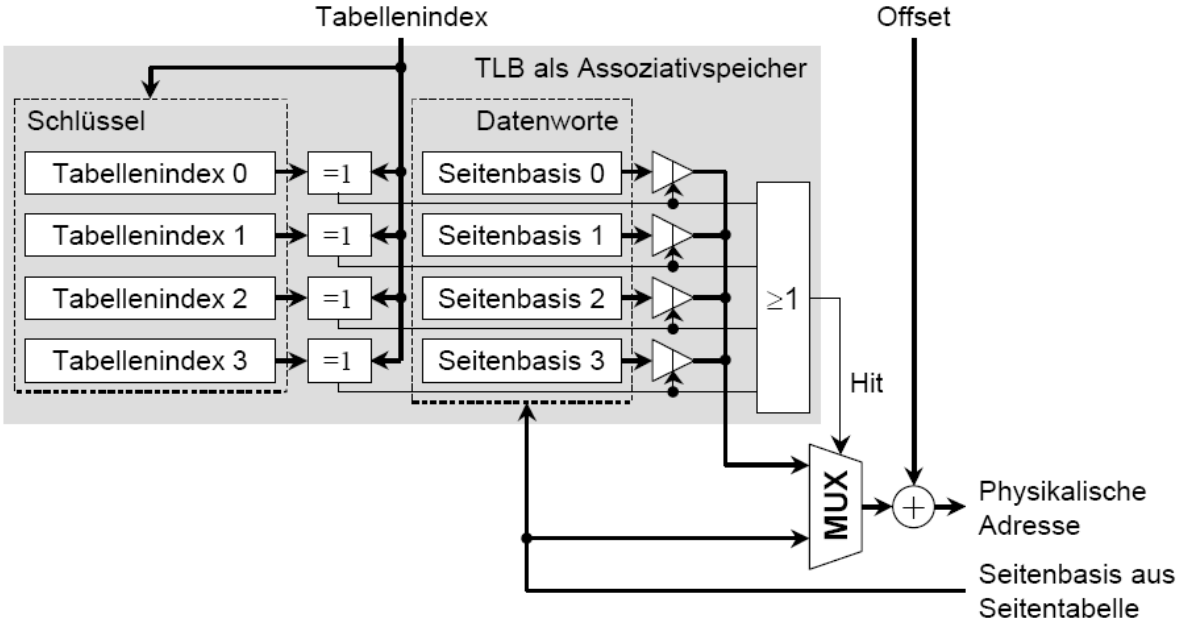
Das Problem bei der vorgestellten Umsetzung virtueller in physikalische Adressen ist, daß bei jedem Speicherzugriff ein zusätzlicher Tabellenzugriff durch die MMU und damit ein langsamer Zugriff auf den externen Speicher durchgeführt werden muß.
Zur Beschleunigung der Umsetzung wird die Hardware obiger Abbildung um einen sogenannten
Translation Lookaside Buffer (TLB) erweitert. Die resultierende Hardwarestruktur ist in untenstehender Abbildung dargestellt.

Bild 78. TLB



Beim Zugriff der CPU auf den Speicher wird der Tabellenindex der virtuellen Adresse an den TLB angelegt. Der TLB beinhaltet einen Assoziativspeicher, in dem Paare aus Tabellenindex und zugehöriger Basisadresse der physikalischen Speicherseite aus vorangegangenen Zugriffen gespeichert sind. Ist für den aktuellen Tabellenindex bereits ein Eintrag vorhanden, signalisiert der TLB dies durch ein Treffer-Signal (Hit). Dann wird die im TLB gespeicherte Basisadresse der physikalischen Speicherseite verwendet, um den Operanden zu adressieren. Der zusätzliche Speicherzugriff auf die Seitentabelle entfällt.
Ist für den aktuellen Tabellenindex kein Eintrag im TLB enthalten, so muß die Basisadresse der physikalischen Speicherseite über die Seitentabelle ermittelt werden. Diese Adresse wird dann einerseits verwendet, um den Operanden zu adressieren, zusätzlich wird die Adresse aber auch zusammen mit dem Tabellenindex im TLB gespeichert, um nachfolgende Zugriffe mit dem gleichen Tabellenindex zu beschleunigen. Da der TLB nur wenige Einträge in seinem Assoziativspeicher zur Verfügung hat (z.B. 16 Einträge), wird bei vollem Assoziativspeicher derjenige Eintrag gelöscht, dessen Zugriff am längsten zurückliegt.
Durch den TLB können bei typischen Programmen über 95% der Speicherzugriffe ohne Zugriff auf die Seitentabelle ausgeführt werden.
Die Realisierung des TLB kann mittels eines Assoziativspeichers erfolgen (siehe Abbildung unten). Enthält der Speicher einen Eintrag mit dem aktuellen Tabellenindex als Schlüssel, liefert der Assoziativspeicher die zugehörige Seitenbasis direkt und ohne Zugriffe auf den Hauptspeicher zurück.

Bild 79. TLB



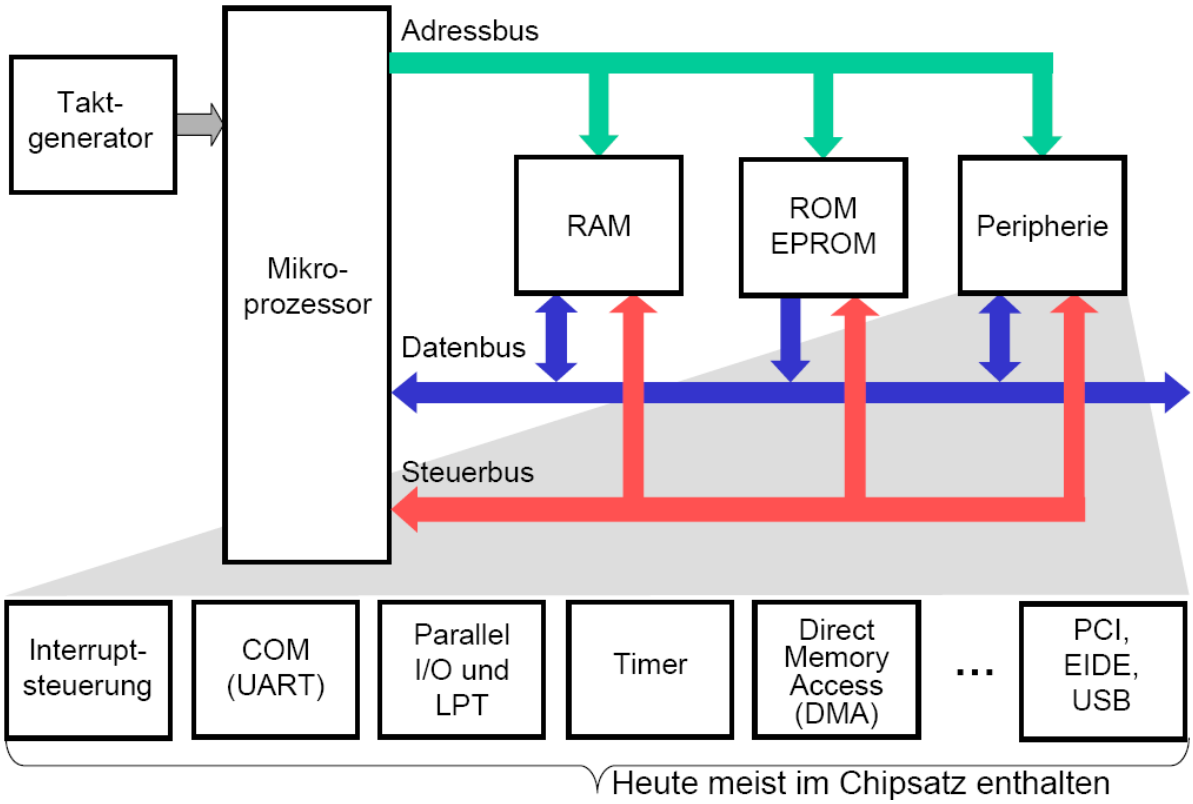
Ist der aktuelle Tabellenindex nicht im Assoziativspeicher enthalten, muß ein Speicherzugriff durchgeführt werden, der aus der Seitentabelle die gesuchte Seitenbasis ausliest. Die Seitenbasis wird dann zusammen mit dem Tabellenindex neu in den Assoziativspeicher eingetragen, so daß nachfolgende Zugriffe auf die gleiche Seite einen gültigen Eintrag vorfinden.

7 Bussysteme und Schnittstellen

Über systeminterne Bussysteme kommunizieren Prozessoren mit ihrer Peripherie. Über externe Bussystem kommunizieren Systeme miteinander in einer vernetzten Umgebung, oft auch über weite Strecken.

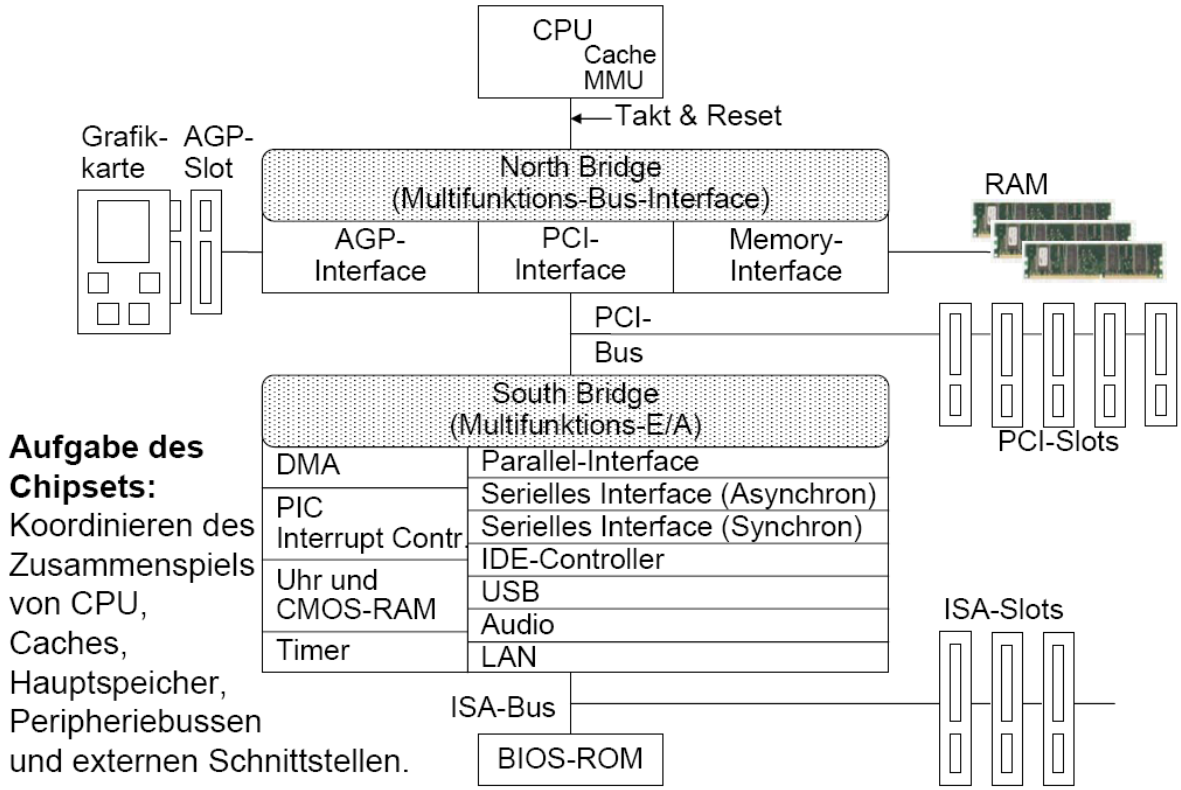
7.1 Grundsätzlicher Aufbau eines Mikrocomputers

Bild 80. Mikrocomputer



7.2 Typische Anordnung der Chips auf einer Hauptplatine

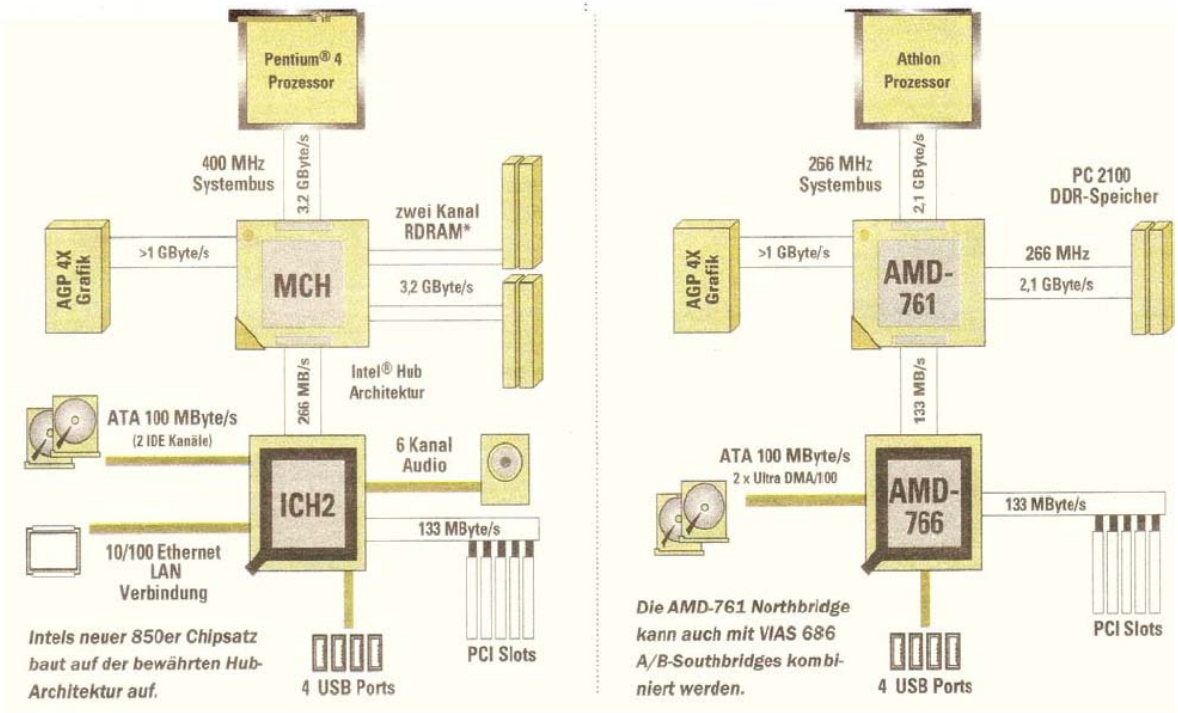
Bild 81. Hauptplatine



Aufgabe des Chipsets:
Koordinieren des Zusammenspiels von CPU, Caches, Hauptspeicher, Peripheriebussen und externen Schnittstellen.

7.3 Chipsätze für Pentium 4 und Athlon-DDR

Bild 82. Chipsätze



MCH = Memory Controlling Hub
ICH = I/O Controller Hub

Quelle: PC Professionell 2/2001

7.4 interne Mikroprozessor-Bussysteme

Veraltete Bussysteme:

ISA
(Industry Standard Architecture)
- Entwickelt für 8086/80286 PC's
- Datenwegbreite: 8/16 Bit,
- Bustakt: 8 MHz

EISA
(Extended ISA)
- Multimasterfähig
- ISA als Unterbus
- Datenwegbreite: 32 Bit
- Bustakt: 8 MHz

VLB - VESA Lokal Bus
(Video Electronics Standard Association)
- Multimasterfähig
- Datenwegbreite: 32 Bit
- Bustakt: 33 MHz

Aktuelle Bussysteme:

PCI-Bus (Peripheral Components Interface)
- Multimasterfähig
- Datenwegbreite: 32 oder 64 Bit
- Bustakt: 33 oder 66 MHz

SCSI-Bus (Small Computer Systems Interface)
- Herstellerunabhängige Schnittstelle für Peripheriegeräte bezüglich Mechanik, Elektrik, Software-Treiber
- Anschluss von max. 7 (8 mit Adapter) Geräten mittels ursprünglich 50-poligem Flachbandkabel
- Datenwegbreite: 8 oder 16 Bit
- Bustakt: 5 bis 160 MHz

AGP (Accelerated Graphics Port)
- Schnittstelle für Grafikkarten
- Datenwegbreite: 32 Bit
- Bustakt: 66 MHz

PCMCIA (PC Memory Card International Association)
- Universelle E/A-Schnittstelle für Laptops.

7.4.1 PCI - Peripheral Component Interconnect Bus

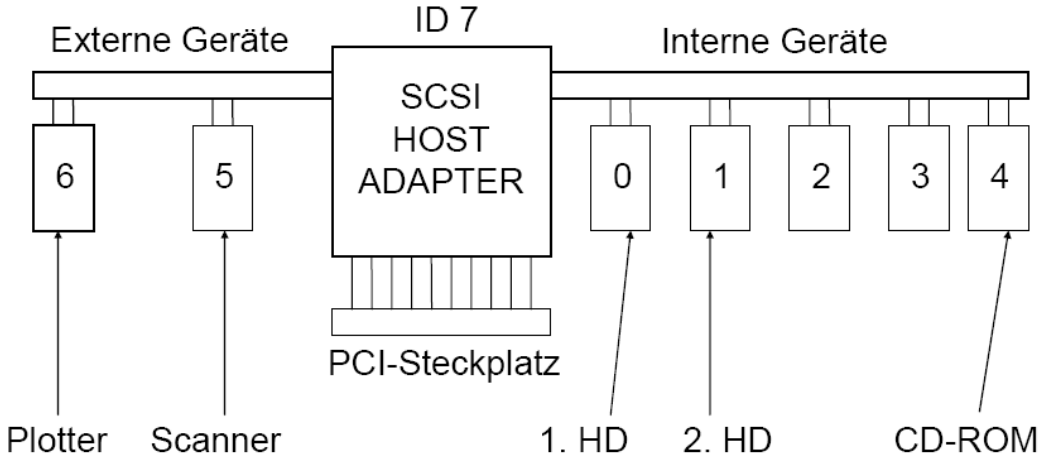
- Prozessorunabhängiger Bus zum Betreiben schneller Komponenten (< 10 Komponenten inklusive Bridge)
- Unterstützung langer Bursts, d.h. kontinuierlicher Übertragung größerer Datenblöcke als ununterbrochenes Bündel kleiner Dateneinheiten
- 32-Bit-Multiplex-Bus, auf 64 Bit erweiterbar (derzeitig für Server), davon unabhängige 32- oder 64-Bit-Adressierung
- Synchroner Bus mit 33- oder 66 MHz-Bustaktfrequenz
- Multi-Master-Fähigkeit

Taktrate (MHz)	33		66	
Busbreite (Bit)	32	64	32	64
Non-Burst-Read (MByte/s)	44	88	88	172
Non-Burst-Write (MByte/s)	66	132	132	264
Burst-Read (MByte/s)	106	211	211	423
Burst-Write (MByte/s)	117	234	234	468

7.4.2 SCSI - Small Computer Systems Interface

- International standardisiertes universelles Interface (Bus-System) für den Anschluss beliebiger Geräte

- Es gibt mehrere SCSI-Standards, die meist abwärtskompatibel sind.
- Wesentliche Unterschiede bestehen in den Steckerausführungen und Taktraten
- Beim Anschluss von Geräten an den SCSI-Bus muss das jeweils letzte einer Kette terminiert werden. Sonst entstehen elektrische Reflexionen am Ende der Leitungen, wodurch der Signaltransfer nicht korrekt funktioniert.
- Ebenso muss jedem Gerät eine eindeutige Device-ID zugeordnet werden.



	Datenweg-breite [Bit]	Maximale Datenrate [MByte/s]	Maximal anschließ-bare Geräte (incl. Adapter)	Maximale Kabellänge [m]	Kabeltyp (Anzahl Pole)
SCSI-1	8	5	8	6	50
Fast SCSI (SCSI-2)	8	10	8	3	50
Ultra SCSI	8	20	4	3	50
	8	20	8	1,5	
Fast Wide SCSI	16	20	16	3	68
Ultra Wide SCSI	16	40	4	3	68
	16	40	8	1,5	
Ultra2 Narrow SCSI	8	40	8	12	68
Ultra2 Wide SCSI	16	80	16	12	68
Ultra3 Wide (160) SCSI	16	160	16	12	68

7.4.3 EIDE – Anschluss von Festplatten oder CD/DVD

IDE = Integrated Drive Electronics,
EIDE = Enhanced IDE, oft ATA genannt (Der Plattencontroller sitzt auf dem Laufwerk).
ATA = AT-Attachment (AT = Advanced Technology, Bus ab 80286 CPU)
ATAPI= Advanced Technology Attachment Packet Interface:
Standard zur Einbindung von „Nicht-Festplatten“ in die (E)IDE- Schnittstelle.
Für ein ATAPI-Laufwerk im System braucht man einen „Treiber“.
PIO = Programmierter I/O (mittels IN- und OUT- Befehle des Prozessors)
– Es erfolgt eine direkte Übertragung der Daten ohne Sicherung
DMA = Direct Memory Access
– Der Datentransfer erfolgt weitgehend ohne Mitwirkung der CPU

Meist sind auf dem Mainboard zwei Kanäle (Primary /Secondary Channel) verfügbar. An jeden Kanal können wiederum zwei Geräte angeschlossen werden: Master und Slave.

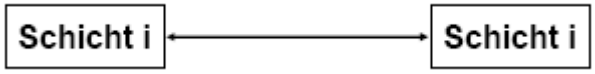
IDE-Modes							
Mode	0	1	2	3	4	5	6
PIO [MByte/s]	3,33	5,22	8,33	11,11	16,6	25	33
Ultra DMA [MByte/s]	(16,66)	(25)	33,33	44	66	100	133

7.5 Externe Bussysteme

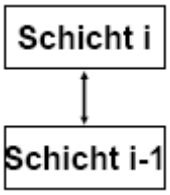
ISO-OSI (international standards organization/open systems interconnection) Schichtenmodell
Auch hier wird wieder eine Abstraktionshierarchie verwendet.

Hierarchie von Schichten, die jeweils über Protokolle miteinander kommunizieren.

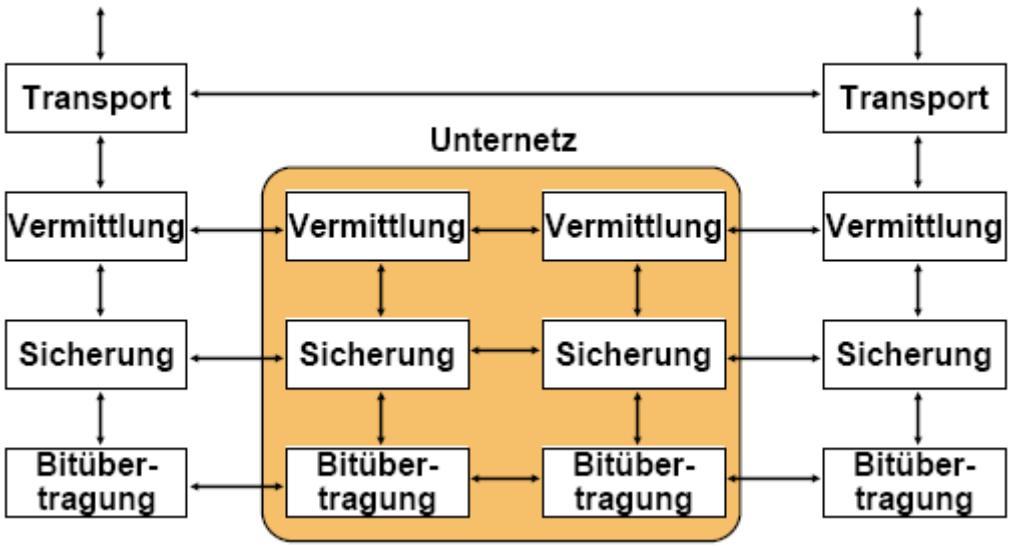
Bild 83. ISO-OSI



eine Schicht benutzt jeweils die Dienste der darunterliegenden



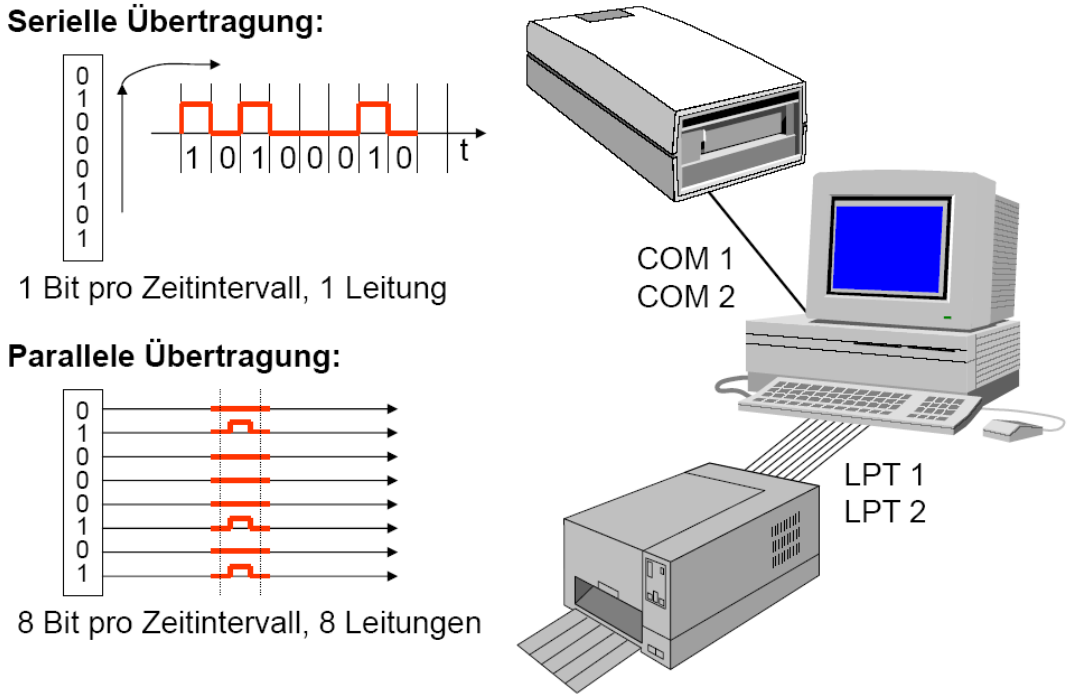
die untersten vier Schichten des OSI-Schichtenmodells



Transportschicht (transport layer):
wie werden Verbindungen für komplette Datenübertragungen aufgebaut?
Vermittlungsschicht (network layer):
auf welchen Wegen werden im Netzwerk Daten übertragen?
Sicherungsschicht (data link layer): wie werden Folgen von Einzelbits unter Fehlerkorrektur zu Rahmen zusammengefaßt?
Bitübertragungsschicht (physical layer): wie werden Einzelbits physikalisch übertragen?

7.5.1 Serielle und Parallele Schnittstellen

Bild 84. Serielle und Parallele Schnittstellen



7.5.2 Spezifikationen und Begriffe

Normen:
V.nn für analoge Übertragung, Beispiel: V.24 für die serielle Schnittstelle (COM)
X.nn für digitale Übertragung, Beispiel: X.25, meist für Telekommunikation

Betriebsarten:
• Simplex (unidirektional): Datenübertragung nur in eine Richtung
• Halbduplex (bidirektional): Nicht gleichzeitige Datenübertragung in beide Richtungen
• Vollduplex (bidirektional): Gleichzeitige Datenübertragung in beide Richtungen
-> doppelte Leitungen

Serielle/parallele Übertragung:
• Seriell: Zeichenbits sequentiell in bestimmtem Taktraster über 1 Leitung übertragen
• Parallel: Alle n Bit eines Zeichens gleichzeitig über n Leitungen übertragen

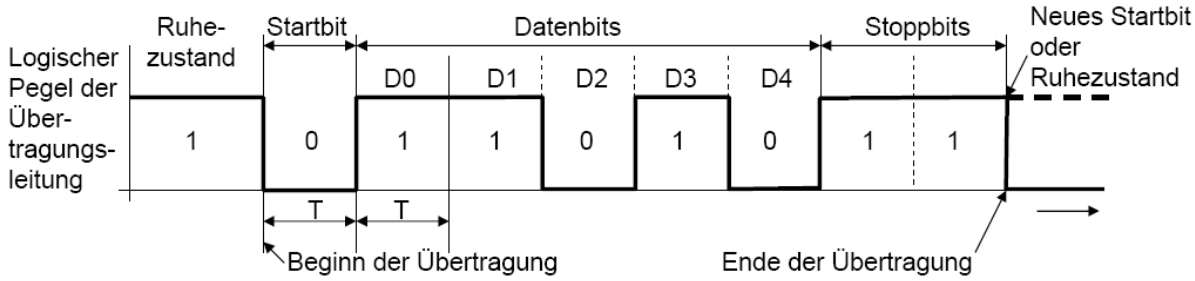
Synchrone/asynchrone Übertragung:
• Synchron: Übertragung der Zeichen im festen Zeitraster. Synchronisierung über gemeinsamen Takt oder über die Daten selbst
• Asynchron: Zeitlicher Abstand zwischen einzelnen Zeichen. Variable Synchronisierung über zusätzlich mit übertragene Steuerinformation

Geschwindigkeitsbegriffe:
• Übertragungsrate: Anzahl übertragener Bit/Byte pro Sekunde [Bit/s bzw. Byte/s]
• Schrittgeschwindigkeit: Anzahl Takte pro Sekunde [Baud]
-> seriell: Übertragungsrate =Schrittgeschw. parallel: Ü > S
• Transfergeschwindigkeit: Netto-Übertragungsrate ohne Steuerinformationen [Bit/s bzw. Byte/s]

7.5.3 Asynchrone serielle Übertragung mit Start- und Stopbits

Sender und Empfänger synchronisieren sich nach folgendem Prinzip:
• Übertragungsleitung liegt im Ruhezustand auf High-Pegel
• Empfänger tastet die Empfangsleitung üblicherweise mit 16-facher Übertragungsrate ab
• Nach Erkennen der fallenden Flanke des Startbits tastet der Empfänger die Empfangsleitung in den Bitmitten ab (durch die bekannte Übertragungsrate möglich).
• Mindestens ein Stopbit schließt die Übertragung eines Zeichens ab und ermöglicht dadurch neue Synchronisation mit der negativen Flanke des nächsten Startbit.

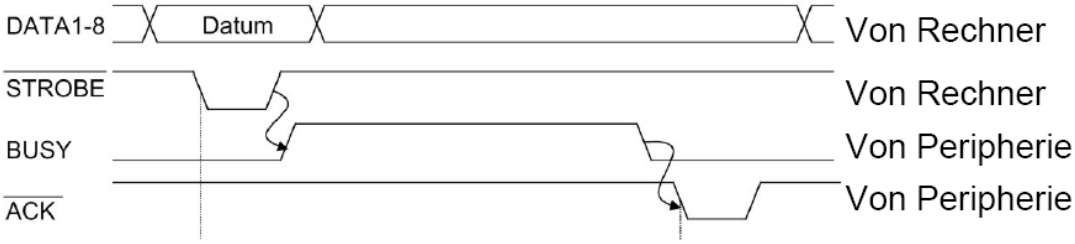
Bild 85. Serielle Übertragung



7.5.4 Parallele Datenübertragung

Centronics-Schnittstelle:
36-poliger Stecker, 18 Signalleitungen, 18 Masseleitungen
IBM-Schnittstelle:
25-poliger Stecker, 18 Signalleitungen, 7 Masseleitungen, maximale Kabellänge: 5 m
Ursprünglich nur für die Druckausgabe entwickelt (unidirektional), wird die parallele Schnittstelle heute auch zur bidirektionalen Kommunikation benutzt. (IEEE 1284)
Bsp.: Scanner, ZIP-Laufwerk, Kopplung zweier Computer

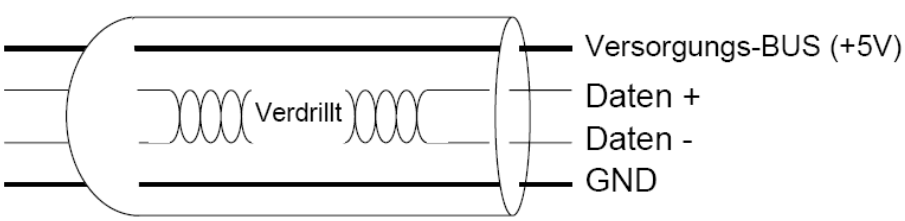
Bild 86. Parallel Übertragung



USB – Universal Serial Bus

• Bis zu 126 Geräte an einen Root Hub anschließbar
• 4 - Draht – Kabel (2 für Signale, 2 für Stromversorgung der angeschlossenen Geräte)
    • USB Low speed mode (1,5 MBit/s) für Maus, Tastatur,... ungeschirmte Kabel < 3m
    • USB 1.1 Full speed mode (15 MBit/s) geschirmte twisted pair Kabel < 5m
    • USB 2.0 High speed mode 480 MBit/s
• Host fragt angeschlossene Geräte über Polling ab, d.h.zyklische Abfrage in festgelegten Zeitintervallen (Pollingperiode je nach Gerät)
• Hot Plug & Play: Hinzufügen und Entfernen von Geräten während des Betriebs
• Verkabelung als hierarchischer Baum mit Sternkopplern (Hubs) an den Knoten
• An die Hubs können wiederum Hubs oder Peripheriegeräte angeschlossen sein
• Kabellänge zwischen Geräten maximal 5 m, maximal 7 Kabelstrecken zwischen Root Hub und letztem Gerät in einer Kette.

Bild 87. USB



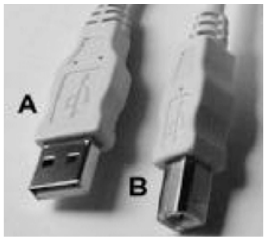
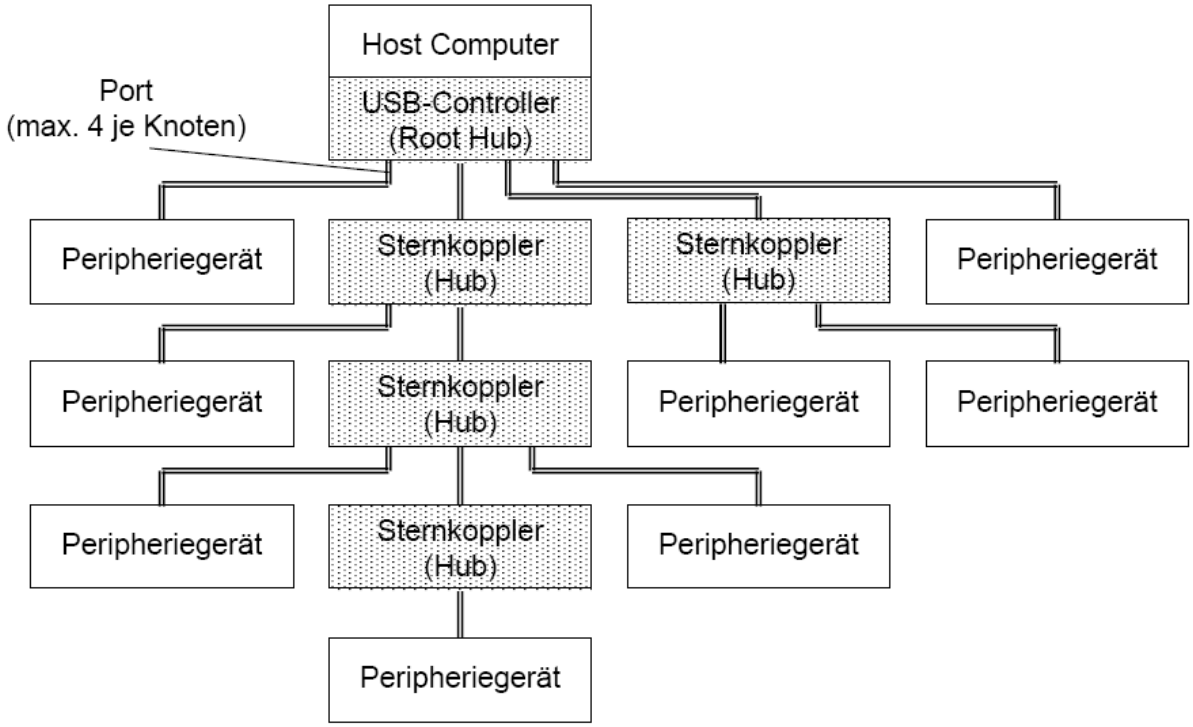
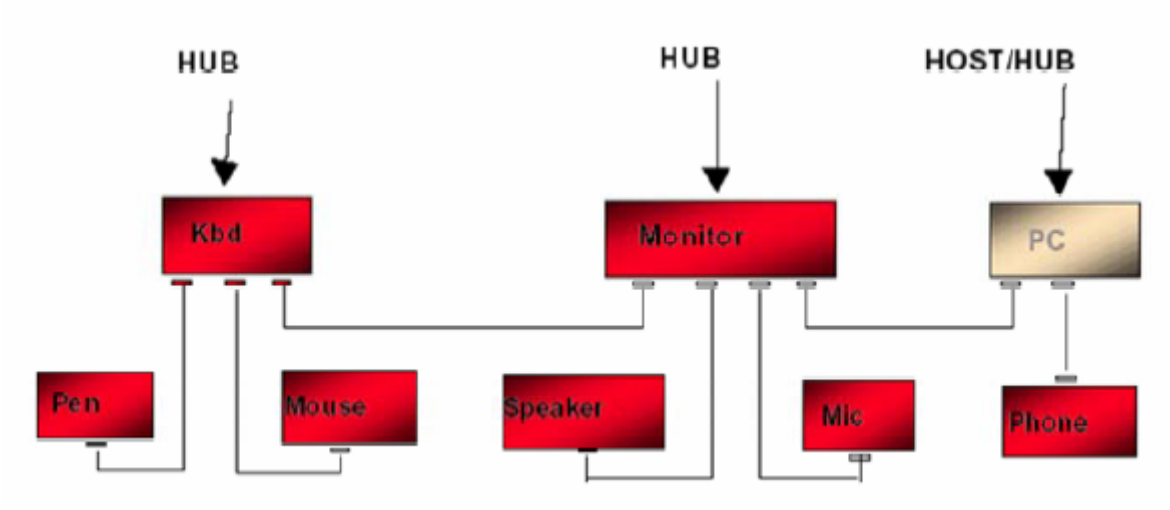


Bild 88. USB Hierarchie



Bus mit Master (Root Hub):
• Root Hub fragt die Geräte zyklisch ab (Polling), ob und wohin sie Daten übertragen wollen
• Root Hub richtet dann eine virtuelle Verbindung (Pipe) zwischen Sender und Empfänger ein
• Root Hub übergibt dann Sendeberechtigung (Token) an den Sender
• Anschließend werden die Daten übertragen
• Der Empfänger quittiert den Datenempfang mit einem Handshake-Paket
• Nach der Datenübertragung wird die Kontrolle (Token) an den Root Hub zurückgegeben

Bild 89. USB Hierarchie

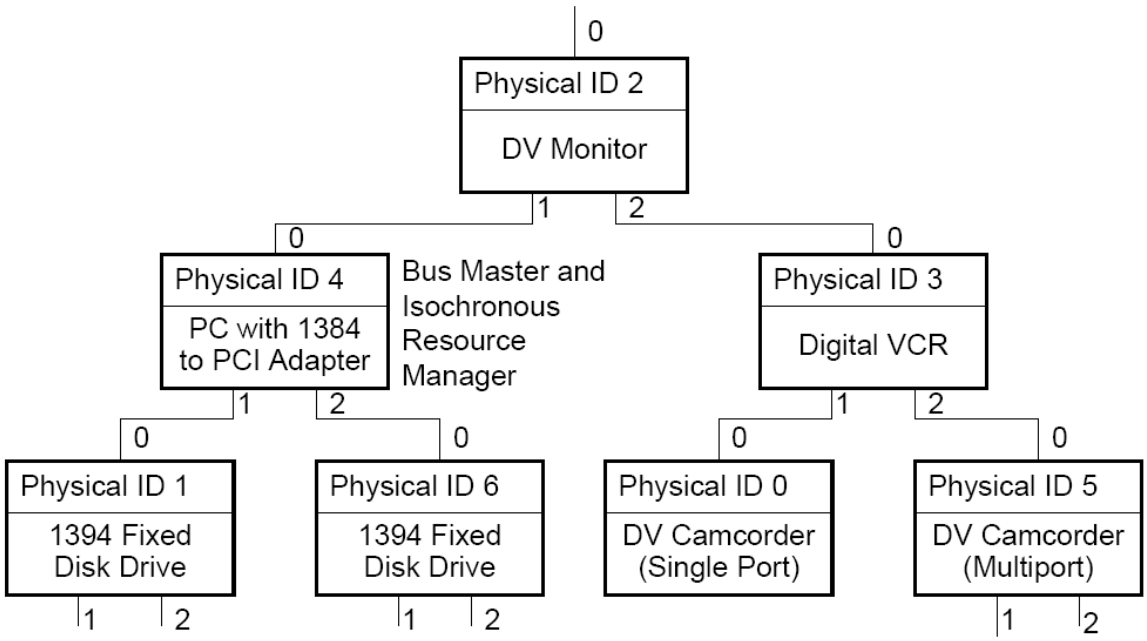


7.5.5 IEEE 1394 (FireWire)

IEEE 1394 ist auch unter dem Namen “FireWire” bekannt geworden

(Warenzeichen der Fa. Apple, Fa. Sony: iLink)
• Serielles, zentral gesteuertes Bussystem mit sternförmiger Topologie.
• Maximal 65 Knoten mit je 3 Busanschlüssen (Ports) möglich.
• Kabellänge zwischen zwei Geräten: Maximal 4,5 m; Gesamtlänge maximal 72 m.
• Speisespannungszuführung über das vieradrige Buskabel.
• Spezifizierte Übertragungsraten: 100, 200, 400, 800 und 1600 Mbit/s.
(Das langsamste Gerät in einer Kette bestimmt die maximale Bitrate)
• Systemerweiterung/Systemumbau sehr einfach (Zusammenstecken - jede zusammengesteckte Busstruktur konfiguriert sich selbst).
• Vorkehrungen zum Anstecken und Entfernen von Einrichtungen bei laufendem Betrieb (Dynamic Attach/Detach).
• Isochrone Busprotokolle mit garantierten Datenraten und Latenzzeiten.
• Multi-Master-Eigenschaften und Peer to Peer - Netzwerk.

Bild 90. Firewire



7.5.6 IrDA - Infrarot-Datenübertragung

IrDA Infrarot-Datenübertragung (IrDA = Infrared Data Association)
• IrDA DATA: Standard für Punkt-zu-Punkt Infrarot Kommunikation
• Für gerichtete Kommunikation konzipiert
• Kurze Distanz bis 1m, 30° Kegel
• Übertragung: Asynchron mit 2,4 bis 115,2 KBit/s (basiert auf COM / UART - serielle Schnittstelle)
• Räume als natürliche Grenzen; aber Abschattungsprobleme
• Anwendungen: • Kommunikation zwischen Host und Peripherie (Drucker, Maus, Tastatur,..)
• Standard in mobilen Rechnern, PDAs
• „Point-and-Shoot“-Anwendungen
– z.B. von Digitaler Kamera auf den Drucker
– z.B. von PDA zu PDA: Visitenkarten austauschen

7.5.7 Bluetooth Mobilkommunikation

Bluetooth Technologie:
• Mobilfunktechnik für ad hoc Vernetzung ohne vorbestimmten Master
• kurze Reichweiten (10m)
• Universell für Sprache und Daten ausgelegt
• Anwendung: Primär für portable, persönliche Geräte
• Ziele: Niedrige Kosten und kleine Baugröße
• Übertragung im ISM-Band, 2.4 GHz: Lizenzfrei in fast allen Ländern
• 79 Kanäle im Bereich 2,402 bis 2,480 GHz, je 1 MHz breit
  – "frequency hopping": 1600 hops / s (d.h. Frequenzwechsel alle 625 µs)
• Übertragungsraten:
  • 1 Mbit/s auf dem Medium
  • Datenrate 432 Kbit/s (full duplex) oder 723 / 57 Kbit/s (asymmetrisch)
• Simultane Übertragung von Sprache (synchron) und Daten (asynchron)
• Sicherheitskonzept: Authentisierung, Verschlüsselung auf Verbindungsebene

7.5.8 Ethernet

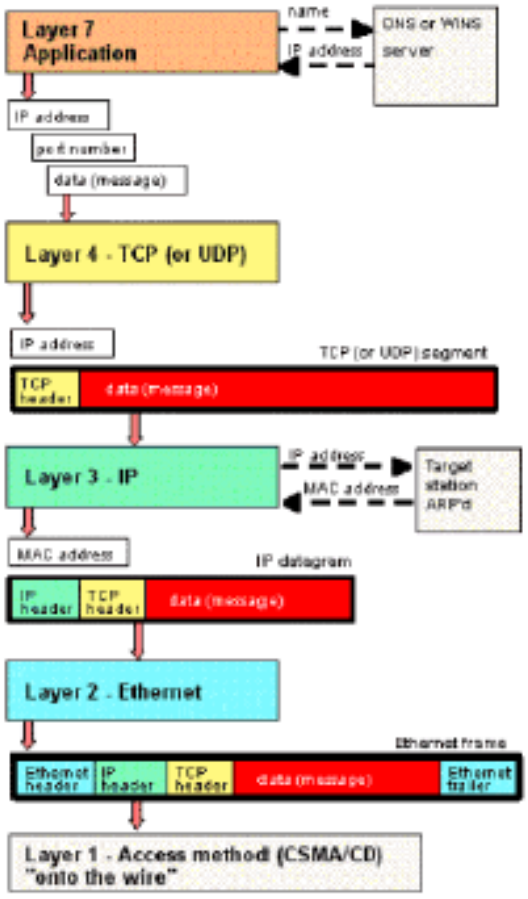
wie können unkoordinierte Benutzer eine gemeinsame Ressource (hier: Kanal) nutzen?
MAC (medium access control), ALOHA (1970)
Prinzipien:

- jeder Benutzer überträgt, wann er will (ohne Abfrage, ob Kanal belegt !)
- kollidierende Rahmen werden zerstört
- durch Abhören des Kanals kann jeder Benutzer eine Kollision feststellen
- nach Feststellen einer Kollision sendet ein Benutzer nach einer zufällig ausgewählten Zeit nochmals
- Auslastung max. 18% bei ALOHA

bei LAN´s, also lokal benachbarten Benutzern mit kurzer Signallaufzeit ist ein Test auf Kanalfreiheit möglich.
Trägererkennung mit Vielfachzugriff, CSMA (carrier sense multiple access)
CSMA/CD (collision detect): Rahmenübertragung wird nach Erkennen einer Kollision gestoppt.
Eigenschaften

- 10 Mbps (ursprüngliche Version), d.h. 100 ns pro Bit
- synchrone Übertragung
- Manchester-Kodierung, auch um Kollisionen überhaupt entdecken zu können
- -0,85 V/+0,85 V/0 V
- 50 Ω Koaxialkabel, thick + thin,
- Ausbreitungsgeschwindigkeit typisch 2 * 108 ms-1 (5ns/m),
- oder verdrilltes Doppelkabel
- Länge < 500m (dick), < 200m (dünn)
- Repeater zur Verlängerung
- zwei Transceiver dürfen nicht mehr als 2,5 km auseinanderliegen
- (4 Repeater)

Bild 91. Ethernet



7.5.9 CAN

CAN (Controller Area Network, Bosch, Ende der 70er), Einsatz z.B. Automobilelektronik
Zusammenfassung: CAN Bus

- einer von vielen Automobil-Bussen
- Echtzeitfähigkeit
- kurze Reaktions- und Latenzzeit (0,01 - 10 ms)
- kleine Datenpakete
- 10 m bis 1200 m typische Buslänge
- oft auf Mikrocontrollern integriert
- Anwendungsgebiete
  - Automobilbau inkl. Nutzfahrzeuge (Traktoren usw.)
  - öffentliche Transportmittel (Straßenbahnen, UBahnen)
  - Medizintechnik
  - Prozeßsteuerungen in der Automatisierungstechnik

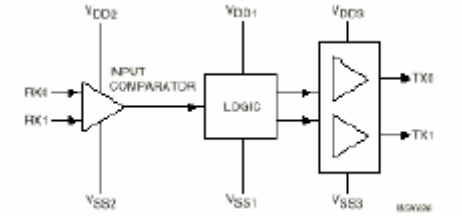
    ...

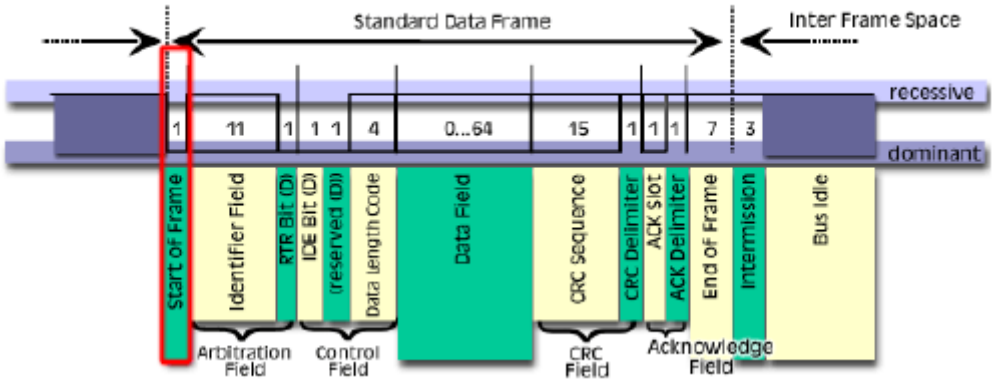
- 1996: > 107 CAN Chips, 3*106 CAN Installationen in Automobilen,
- $7*10^6$ CAN Installationen sonst
- ISO 11898 und 11519-1
- CSMA (carrier sense multiple access), aber nicht CD (collision detect), sondern CA (collision arbitration)
- Nachrichten tragen eine Marke (identifier), der zugleich die Priorität der Nachricht festlegt
- 11 Bits für Priorität (CAN 2.0B 29 Bits)
- 00...0 höchste Priorität, 11...1 niedrigste Priorität
- jede sendebereite Station sendet "gleichzeitig" das erste Bit
- wired-AND Verhalten: die 0 setzt sich durch
- Station mit sendender 0 braucht die Leitung nicht abzutasten
- Station mit sendender 1 überprüft, ob Leitung nach 0 gezogen wird; dann Zurücknahme des Sendewunsches
- verzögerungsfreie Arbitrierung für die Nachricht höchster Priorität
- praktisch ist die Anzahl der Busanschlüsse auf ca. 30-50 (max. 110) begrenzt
- RTR: remote transmission request, eine Nachricht ohne Daten, die von einer anderen Station (z.B. einem Sensor) die Übertragung von Daten verlangt, diese können direkt in den Rahmen geschrieben werden
- drei unterschiedliche Versorgungsspannung/Masse-Zuführungen für Sende-/Empfangs-/sonstige Logik

Übertragungsgeschwindigkeit abhängig von der Buslänge:

- z.B. 1 Mbit/sec bei 50 m, d.h. Übertragungszeit pro Bit $\tau bit = 1\ \mu s$,
- 500 kbit/sec bei 100 m
- bei 1 Mbit/sec dauert die Übertragung der längsten Nachricht 130 µs
-

Bild 92. CAN





Bit Timing und Synchronisation

- verschiedene angeschlossene Geräte haben unterschiedliche Oszillatoren → Synchronisationsproblematik
- "harte" Synchronisation: Beginn eines Rahmens startet Synchronisationslogik
- "weiche" Synchronisation: Abfangen von Drift während der Übertragung eines Rahmens

Basic CAN/Full CAN:

- bei Basic CAN gibt es nur ein Hardware-Filter für Nachrichten - die Software des Microcontrollers muß alle vom Filter angenommenen Nachrichten weiter analysieren
- i.a. 1 Schreibpuffer und zwei Lesepuffer im CAN Controller
- bei Full CAN übernimmt die Hardware des CANControllers die Filterung. Einrichtung mehrerer i.a. flexibel programmierbare Schreib- und Lesepuffer, die als Mailboxen funktionieren, d.h. für sie bestimmte Nachrichten aus dem Bus filtern
- Filterung jeweils über Adressregister und don´t-care Maske

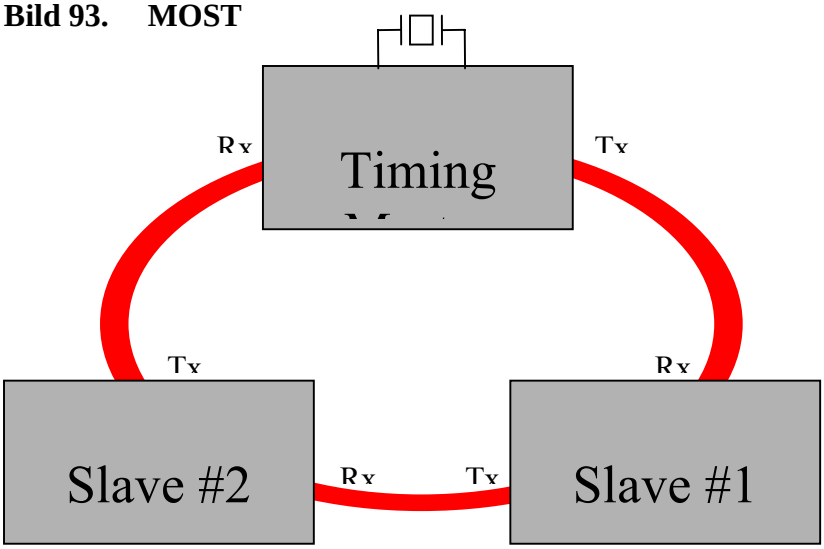
7.5.10 MOST

Der MOST (–**M**edia **O**riented **S**ystems **T**ransport) Bus ist normalerweise ein optischer Bus, der mehrere Infotainment Devices im KFZ vernetzt. MOST standardisiert die physikalische HW-Architektur, die Protokolle und definiert Teile des zu verwendenden Frameworks.
•Optical; visible non-laser light conducted through a polymer fiber
•Gross data rate of 22.5 Mbits/s
•system clock 44.1kHz (or 48 kHz)
•up to 64 nodes (in theory)

•Each device employs a FOT-unit (Fiber optical transceiver) and a network transceiver chip to connect to the network

Three main services:
–synchronous (stream)
–asynchronous (packet)
–control (small packet)
•Auxiliary Services
–resource administration
–remote control
–local routing “patchbay”
–Timing Master generates data frame
–44,100 frames / second
–slaves follow master clock with PLL

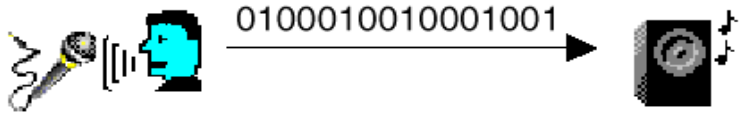
Bild 93. MOST



7.5.11 Source Port (synchronous)

•Fixed time slots in each frame allocated to application
•synchronized to timing master system clock
•time slot allocation maintained by table in timing master
•routing to/from time slots by the routing engine of each node
•High bandwith
•Administration done by the ConnectionMaster (FB) via control channel
•Easy peer-to-peer source to sink connection, directly programmed
•All synchronous data is available everywhere in the ring
•The Data can be received by more than one node

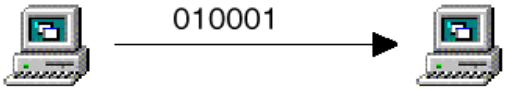
Bild 94. synchroner Kanal



7.5.12 Source Port (asynchronous)

•Bandwidth from the synchronous channels can be reserved for an asynchronous channel
•Async channel provides Ethernet-like communication
•packets can be buffered in transceiver (48 bytes / packet) or in auxiliary hardware (up to 1014 bytes / packet)
•data rate up to several MBits/sec
•Burst type data transport like navi map, internet, SDS

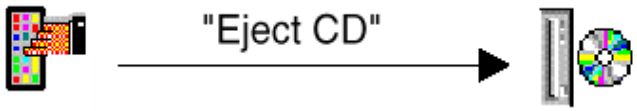
Bild 95. asynchroner Kanal



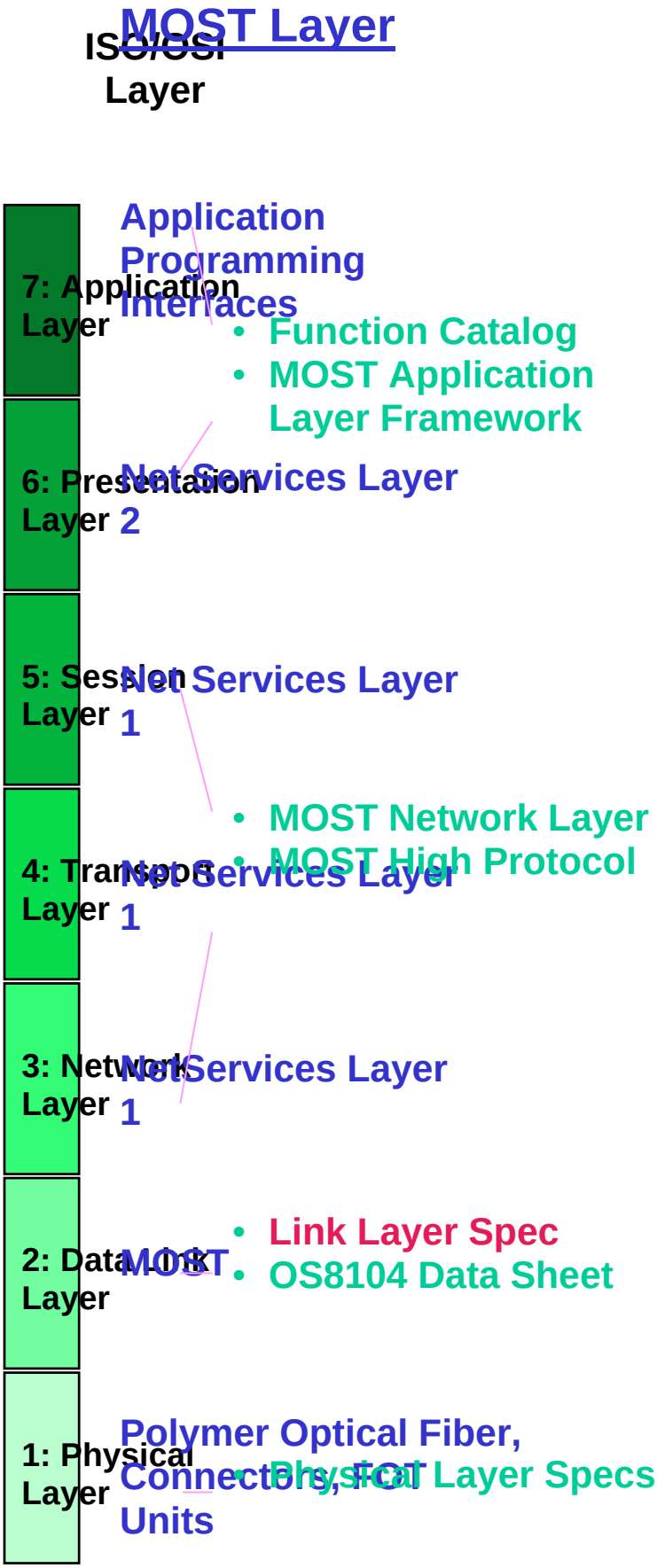
7.5.13 Control Channel

•Data packets for specified nodes, CRC checked
•ACK/NAK mechanisms and automatic retries
•Fixed bandwith reserved for control channel, approx. 10 kBit/s
•approx. 2600 msgs/sec, max 890 from the same node
•each control channel message contains 17 bytes of data
•not designed for bandwidth-extensive user data
•network admin & application control

Bild 96. Control Kanal



ISO/OSI Schichtung

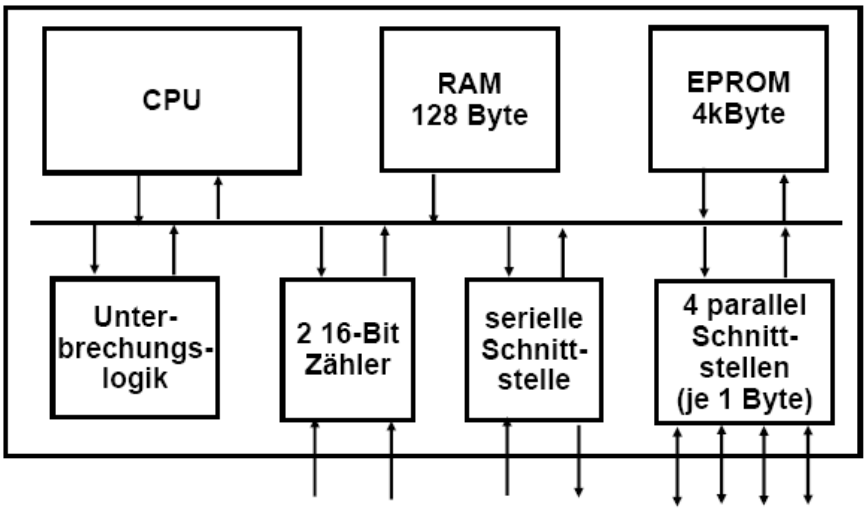


8 Mikrocontroller

- Spezialprozessoren für kontroll-dominierte Anwendungen.
- große Leistungsbereiche vom einfachen Controller bis zum komplizierten Prozessor
- Charakteristika:
  - meist nur geringer Speicherplatz auf dem Chip
  - Befehle für Einzelbit-Adressierung und -Verarbeitung
  - leider oft "krummer" Befehlssatz
  - Einstellen der Funktionsweise über Spezialregister
  - Zeitgeber für interne Zeitbasis, Zähler für externe Ereignisse, Watch-Dog-Timer
  - flexible, dem Anwender zugängliche Unterbrechungsbehandlung
  - serielle/parallele Ports auf dem Chip
  - A/D bzw. D/A-Wandler auf dem Chip

Beispiel:
87C51 (INTEL) 8-bit Ein-Chip-Mikrocontroller, CMOS, 128 Byte RAM, 12 MHz, 4kByte PROM, 2 Timer

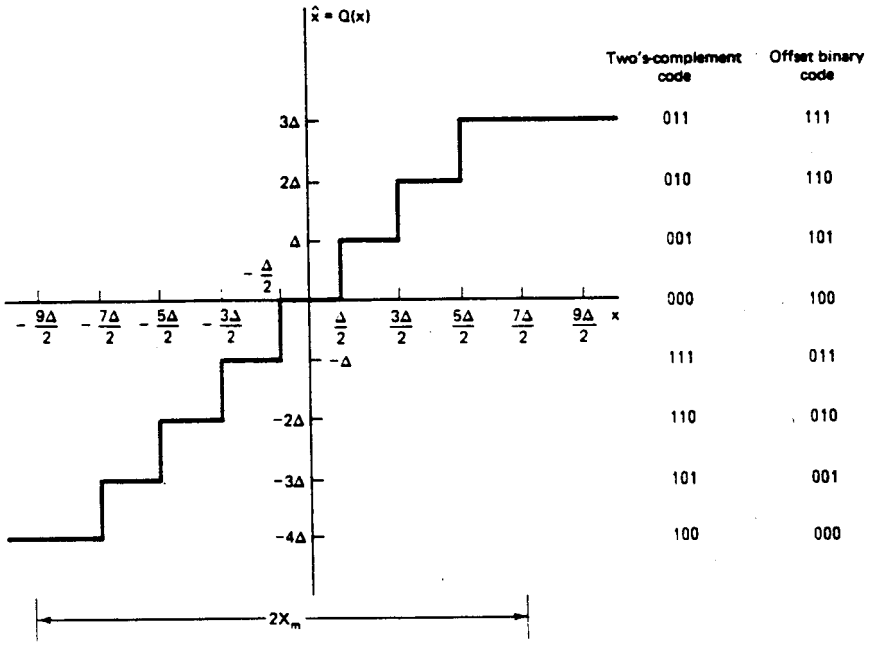
Bild 97. Mikrocontroller



8.1 Spezielle Peripheriebausteine

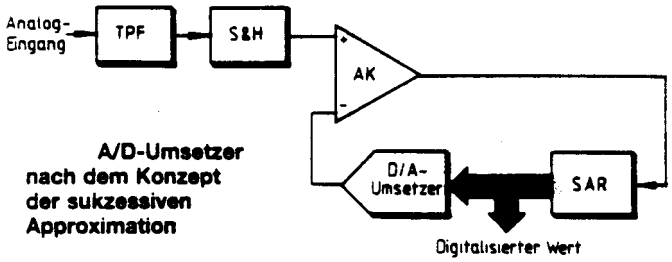
8.1.1 A/D-Wandler

Während das Abtasttheorem nur die zeitliche Abtastung beschreibt, wandelt ein realer A/D-Wandler eine Spannung oder einen Strom in einen binären Code um, das Signal wird abgetastet und quantisiert.
Übliche Zahlendarstellungen sind die 2K-Darstellung oder der offset binary code (siehe Bild).



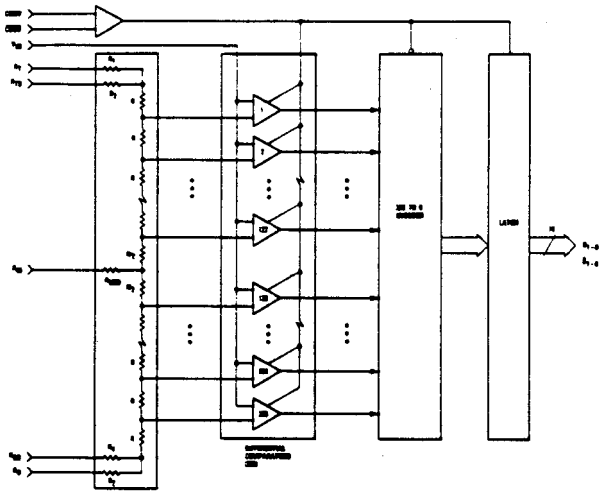
Der Wandler ist taktbetrieben und erwartet für die Zeit der Wandlung ein konstantes Eingangssignal. Deshalb wird für übliche A/D-Wandler eine Sample&Hold-Stufe wie in untenstehenden Bild vorgesehen. Die Wandler selbst arbeiten dann z.B. nach dem Prinzip der sukzessiven Approximation**.**

Bild 98. A/D



Eine andere Klasse Wandler sind die sogenannten Flash-Wandler, die durch eine parallele Anordnung von 2N Komparatoren das Eingangssignal in sehr kurzer Zeit konvertieren. Der Digitaltakt wird dann zur Übernahme des Ergebnisses in ein Ausgangsregister gebraucht. Diese Technik stellt hohe Anforderungen an die Integrationsqualität, da die zulässigen zeitlichen Toleranzen für die einzelnen Komparatorstufen sehr gering sind. Erreichbare technische Daten sind eine Abtastrate von einigen 100MHz und eine Wortbreite von z.Zt. 10 Bit. Diese Wandler benötigen keine S&H-Stufen. Für besonders hochwertige und schnelle Wandler wird diese Technik gelegentlich mit CCD-Analogschieberegistern kombiniert**.**

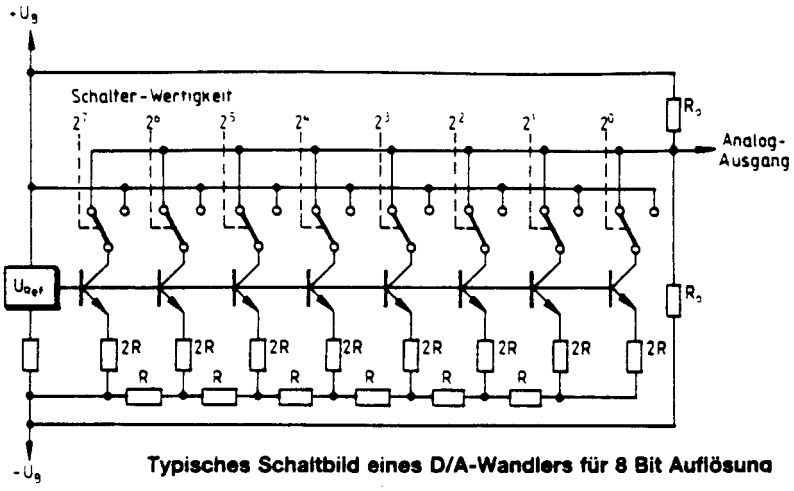
Bild 99. A/D-Wandler



8.1.2 D/A-Wandlung

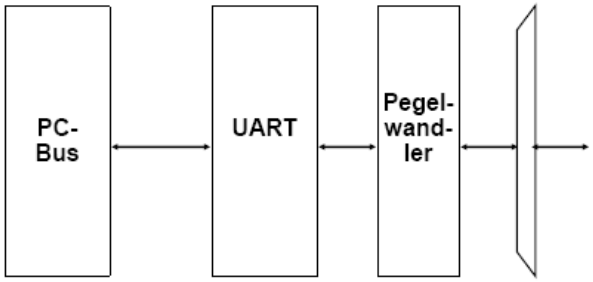
Übliche D/A-Wandler arbeiten nach dem R-2R-Prinzip, bei dem Elemente einer Widerstandsleiter umgeschaltet werden. Es gibt D/A-Wandler in großer Vielfalt und fast ohne technische Begrenzung.

Bild 100. DA_Wandler



Typisches Schaltbild eines D/A-Wandlers für 8 Bit Auflösung

8.1.3 UART



8.1.3.1 RS (recommended standard) 232

+/- 12 V bei PC´s zur Verbesserung der Störsicherheit statt z.B. TTL Pegel
spezielle Treiber zur Umsetzung UART <=> RS 232
bis 15-30 m bei 9600 bps
bei kürzeren Abständen auch hohe Datenübertragungsraten,
z.B. 115.200 bps
Anschlüsse über 9 bzw. 25 polige Anschlußbuchsen, wesentlich:

- TxD zu sendende Daten ->
- DTR (date terminal ready) ->
- RTS (request to send) ->
- RxD zu empfangende Daten <-
- DSR (data set ready) <-
- CTS (clear to send) <-

...
RTS/CTS bzw. DTR/DSR Handshaking zur Abfrage,
ob z.B. Modem bereit zu asynchroner Übertragung

Bild 101. UART