Wolfram Schiffmann

Technische Informatik 2

Grundlagen der Computertechnik

5., neu bearbeitete und ergänzte Auflage

Springer

Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Vorwort zur 5. Auflage

Der vorliegende Band 2 des Buches **Technische Informatik** entstand aus Skripten zu Vorlesungen, die wir an der Universität Koblenz für Informatikstudenten gehalten haben. Es ist unser Anliegen zu zeigen, wie man elektronische Bauelemente nutzt, um Computersysteme zu realisieren. Mit dem dargebotenen Stoff soll der Leser in die Lage versetzt werden, die technischen Möglichkeiten und Grenzen solcher Systeme zu erkennen. Dieses Wissen hilft ihm einerseits, die Leistungsmerkmale heutiger Computersysteme besser zu beurteilen und andererseits künftige Entwicklungen richtig einzuordnen. Der Stoff ist vom Konzept her auf das Informatikstudium ausgerichtet — aber auch für alle diejenigen geeignet, die sich intensiver mit der Computerhardware befassen möchten. Somit können z.B. auch Elektrotechniker oder Maschinenbauer von dem vorliegenden Text profitieren.

Für die Lektüre genügen Grundkenntnisse in Physik und Mathematik. Die Darstellung des Stoffes erfolgt „bottom-up“, d.h. wir beginnen mit den grundlegenden physikalischen Gesetzen und beschreiben schließlich alle wesentlichen Funktionseinheiten, die man in einem Computersystem vorfindet.

Der Stoff wurde auf insgesamt drei Bände aufgeteilt: Der Band 1 **Technische Informatik – Grundlagen der digitalen Elektronik** führt zunächst in die für die Elektronik wesentlichen Gesetze der Physik und Elektrotechnik ein. Danach werden Halbleiterbauelemente und digitale Schaltungen behandelt. Der Band 1 schließt mit dem Kapitel über einfache Schaltwerke, wo dann der vorliegende zweite Band **Technische Informatik — Grundlagen der Computertechnik** anknüpft. Als Ergänzung zu den beiden Lehrbücher gibt es einen weiteren Band **Technische Informatik — Übungsbuch zur Technischen Informatik 1 und 2**. Um den Lesern das Auffinden geeigneter Aufgaben im Übungsband zu erleichtern, haben wir in der Neuauflage Verweise auf themenspezifische Aufgaben eingebaut. Außerdem wurde auch eine Webseite zu den Büchern eingerichtet, die Links auf weitere nützliche Materialen zum Thema enthält. Die Adresse dieser Webseite lautet:

Technische-Informatik-Online.de

Der Text beginnt mit einer kurzen Zusammenfassung über Aufbau, Funktion und Entwurf von Schaltwerken. Manche Aufgabenstellungen für Schaltwerke sind so komplex, dass die Schaltwerke nicht mehr mit Hilfe von Zu-

standsgraphen und –tabellen entwickelt werden können. Im Kapitel *Komplexe Schaltwerke* wird zunächst das Zeitverhalten von Schaltwerken untersucht und in das Konzept kooperierender Schaltwerke eingeführt. Dann wird der Entwurfsprozess bei so genannten *Algorithmic State Maschines* mit Hilfe von ASM–Diagrammen erläutert und es werden am Beispiel eines „Einsen–Zählers" verschiedene Varianten komplexer Schaltwerke vorgestellt.

Die überwiegende Zahl heutiger Computer arbeitet nach dem Operationsprinzip, das im Kapitel *von NEUMANN–Rechner* vorgestellt wird. Die grundlegenden Funktionseinheiten eines solchen Rechners werden beschrieben und ihr Einfluss auf eine Prozessorarchitektur wird diskutiert.

Alle modernen Prozessoren nutzen Hardware–Parallelität zur Leistungssteigerung, wie z.B. Funktionseinheiten für Gleitkomma–Arithmetik oder direkter Speicherzugriff zur Ein–/Ausgabe. Im Kapitel *Hardware–Parallelität* werden neben solchen Coprozessoren auch das Pipeline–Prinzip und Array–Rechner vorgestellt. Diese beiden Architekturmerkmale findet man sowohl bei neueren Mikroprozessoren als auch bei sogenannten Supercomputern.

Im Kapitel *Prozessorarchitektur* wird auf die drei Ebenen der Rechnerarchitektur eingegangen. Die Befehls(satz)architektur legt die Schnittstelle zwischen Software und Hardware fest. Eine einmal definierte Befehlsarchitektur kann auf verschiedene Arten implementiert werden. Zunächst muss sich der Rechnerarchitekt die logische Organisation zur Umsetzung der Befehlsarchitektur überlegen. Wie bei den komplexen Schaltwerken teilt man den Entwurf in eine Datenpfad– und Steuerungsstruktur auf. Damit der Prozessorentwurf nicht nur auf dem Papier existiert, muss er in Halbleitertechnik realisiert werden. Die resultierende Prozessorleistung ergibt sich schließlich aus dem Zusammenwirken der Entwurfsschritte auf den drei Ebenen *Befehlsarchitektur* sowie *logischer* und *technologischer* Implementierung.

Im Kapitel *CISC–Prozessoren (Complex Instruction Set Computer)* werden zunächst die Merkmale eines Prozessortyps erläutert, dessen Befehlsarchitektur ein besonders komfortables Programmieren auf Maschinenbefehlsebene zum Ziel hat. Als typischer Vertreter dieser Klasse wird der Motorola 68000 beschrieben und die Entwicklungsgeschichte zum Modell 68060 wird zusammengefasst.

Ende der siebziger Jahre wurde die Verwendung komplexer Befehlssätze neu überdacht. Man untersuchte die von Compilern erzeugten Maschinenbefehle und stellte fest, dass bei CISCs nur ein Bruchteil der verfügbaren Befehle verwendet wird. Diese Situation war der Ausgangspunkt für die Entwicklung neuartiger Prozessorarchitekturen, die man wegen ihres einfachen Befehlssatzes als *RISC–Prozessoren (Reduced Instruction Set Computers)* bezeichnet. Nach einer kurzen historischen Einführung werden das Befehlspipelining und die dabei auftretenden Pipelinekonflikte behandelt. Dann wird gezeigt, wie diese Pipelinekonflikte software– oder hardwaremäßig beseitigt werden können. Optimierende Compiler sorgen durch eine entsprechende Befehlsumordnung dafür, dass die zur Behebung von Konflikten notwendigen Leerbefehle durch

nützliche Befehle ersetzt werden können. Die Beseitigung von Konflikten, die zur Laufzeit des Programms auftreten ist jedoch nur durch zusätzliche Hardware möglich. Durch dynamische Befehlsplanung mit Hilfe sogenannter Reservierungsstationen gelingt es, Prozessoren mit mehreren Funktionseinheiten zu realisieren, die gleichzeitig mehrere Maschinenbefehle ausführen können. Alle heutigen Hochleistungsprozessoren nutzen *Superskalarität*, um ihre Verarbeitungsleistung zu maximieren. Als Beispiel für einen superskalaren RISC–Prozessor wird der PowerPC 620 vorgestellt.

Das Kapitel *Kommunikation* behandelt Techniken zur Datenübertragung innerhalb eines Computers und zwischen verschiedenen Computersystemen. Die Art der Datenübertragung hängt stark von der Entfernung der zu verbindenden Komponenten ab. Wir unterscheiden die parallelen und seriellen Verbindungen. Zu jeder Klasse werden zunächst die Prinzipien und dann typische Vertreter beschrieben. Wir beginnen bei prozessornaher Kommunikation und kommen schließlich über die lokalen Netze (LANs) zu den Weitverkehrsnetzen (WANs). In der neuen Auflage wurden Abschnitte über verschiedene Ethernet-Varianten sowie drahtlose Netzwerke hinzugefügt.

Heutige Computer verwenden eine Mischung verschiedener Speichertechnologien. Diese Speicher unterscheiden sich bezüglich Speicherkapazität, Zugriffszeit und Kosten. Im Kapitel *Speicher* werden nach einer Übersicht über die verschiedenen Speicherarten zunächst Halbleiterspeicher behandelt. Die Beschreibung magnetomotorischer Speicher wurde grundlegend überarbeitet und enthält nun auch Einzelheiten über die Dateisysteme der heute üblichen Betriebssysteme LINUX und Windows. Schließlich werden auch optische Wechselspeichermedien wie CD-ROM und DVD ausführlicher vorgestellt.

Bei der *Ein–/Ausgabe* kann man digitale und analoge Schnittstellen unterscheiden. Für die wichtigsten Vertreter aus diesen beiden Klassen wird die prinzipielle Funktionsweise erklärt. Anschließend werden ausgewählte Peripheriegeräte beschrieben. Gegenüber der letzten Auflage wurden hier einige neue Komponenten hinzugenommen. Neben Tastatur, Maus, Scanner und Digitalkamera werden nun auch LCD-Monitore und die verschiedenen Druckertypen ausführlich behandelt. Ihre Funktionsweise wird anhand von zahlreichen Abbildungen illustriert.

Das in der letzten Auflage neu hinzugekommene Kapitel *Aktuelle Computersysteme* wurde stark überarbeitet und findet sich nun am Ende des Buches. Dies hat den Vorteil, dass der Leser bereits alle Grundbegriffe über Kommunikation, Speicher und Ein-/Ausgabe aus den vorangegangenen Kapiteln kennt und dass daher vorwärts gerichtete Verweise entfallen. In dem Kapitel wird ein kurzer Überblick über aktuelle Computersysteme gegeben. Zunächst stellen wir die verschiedenen Arten von Computern vor. Dann betrachten wir am Beispiel von Desktop-Systemen deren internen Aufbau, die aktuellsten Desktop-Prozessoren, Funktionsprinzipien der aktueller Speichermodule und die verschiedenen Standards für Ein- und Ausgabeschnittstellen. Schließlich

gehen wir auch auf die Bedeutung von Grafikadaptern ein und geben einen Ausblick auf die künftige Entwicklung.

Wir haben uns bemüht, zu den einzelnen Themen nur die grundlegenden Prinzipien auszuwählen und durch einige Beispiele zu belegen. Wir hoffen, dass es uns gelungen ist, den Stoff klar und verständlich darzustellen. Trotzdem möchten wir die Leser auffordern, uns ihre Ergänzungs– und Verbesserungsvorschläge oder Anmerkungen mitzuteilen. Im Text werden immer dann englischsprachige Begriffe benutzt, wenn uns eine Übersetzung ins Deutsche nicht sinnvoll erschien. Wir denken, dass diese Lösung für den Leser hilfreich ist, da die Literatur über Computertechnik überwiegend in Englisch abgefasst ist.

Bei der mühevollen Arbeit, das Manuskript der ersten Auflage mit dem LaTeX–Formatiersystem zu setzen bzw. Bilder zu zeichnen, wurden wir von Sabine Döring, Christa Paul, Inge Pichmann, Jürgen Weiland und Edmund Palmowski unterstützt. Unsere Kollegen Prof. Dr. Alois Schütte und Prof. Dr. Dieter Zöbel ermunterten uns zum Schreiben dieses Textes und gaben uns wertvolle Hinweise und Anregungen. Prof. Dr. Herbert Druxes, Leiter des Instituts für Physik der Universität in Koblenz, förderte unser Vorhaben.

Für ihre Mitarbeit und Unterstützung möchten wir allen herzlich danken. Besonders sei an dieser Stelle auch unseren Familien für ihre Geduld und ihr Verständnis für unsere Arbeit gedankt.

Robert Schmitz, der Mitautor der bisherigen Auflagen, ist bereits seit fünf Jahren im wohlverdienten Ruhestand. Er wollte nach der vierten Auflage nicht mehr länger am Band 2 mitarbeiten. Daher werde ich, Wolfram Schiffmann, künftig das Buch als alleiniger Autor weiter betreuen. Ich danke Robert Schmitz als zuverlässigem Mitautor und als gutem Freund für die stets fruchtbare und erfreuliche Zusammenarbeit bei den ersten vier Auflagen.

Die freundliche Aufnahme der letzten Auflagen gibt mir nun wieder die Möglichkeit, das Buch erneut in überarbeiteter Form herauszugegeben. Neben Korrekturen und den o.g. Erweiterungen habe ich mich bemüht, die Anregungen aus dem Leserkreis nun auch in die fünfte Auflage von Band 2 aufzunehmen.

Hagen, im Dezember 2004 Wolfram Schiffmann

Inhaltsverzeichnis

Auszug des Inhalts von Band 1

Man wird dadurch fast gänzlich unabhängig von den Flipflop–spezifischen Schaltzeiten. Die Taktphase T_1 bestimmt im Wesentlichen die Zeit T_{WK} und die Taktphase T_0 bestimmt T_{KW} (Abb. 1.3). Die maximal mögliche Taktfrequenz eines Schaltwerks wird durch die Verzögerungszeit T_g des Rückkopplungs–Schaltnetzes begrenzt. Diese Verzögerungszeit kann vor allem bei Schaltnetzen zur Berechnung arithmetischer Funktionen sehr groß werden. Daher ist es wichtig, dass die Laufzeiten bei arithmetischen Schaltnetzen durch geeignete Maßnahmen minimiert werden.

1.2 Entwurf von Schaltwerken

Die wesentlichen Schritte beim Entwurf eines Schaltwerks können wie folgt zusammengefasst werden (vgl. auch Band 1):

Ausgehend von einer verbalen (informalen) Beschreibung, kann man das gewünschte Schaltverhalten mit einem Zustandsgraphen formal spezifizieren. Die Zahl der Zustände wird minimiert, indem man äquivalente Zustände zusammenfasst. Dadurch werden weniger Speicherflipflops benötigt. Durch eine geeignete Zustandscodierung kann der Aufwand zur Bestimmung der Folgezustände minimiert werden. Um dann die Schaltfunktionen für die Eingänge der D–Flipflops zu finden, überträgt man den Zustandsgraphen in eine *Übergangstabelle*. In dieser Tabelle stehen auf der linken Seite die Zustandsvariablen z^t und der Eingabevektor x^t. Auf der rechten Seite stehen die gewünschten Folgezustände z^{t+1} und der Ausgabevektor y^t. (Wir gehen von einem MEALY-Schaltwerk aus).

Die Entwurfsaufgabe besteht daher in der Erstellung der Übergangstabelle und der Minimierung der damit spezifizierten Schaltfunktionen. Diese Entwurfsmethode ist aber nur anwendbar, solange die Zahl der möglichen inneren Zustände „klein“ bleibt.

Beispiel: *8–Bit Dualzähler*
Mit k Speichergliedern sind maximal 2^k Zustände codierbar. Um z.B. einen 8–Bit Dualzähler nach der beschriebenen Methode zu entwerfen, müsste eine Übergangstabelle mit 256 Zeilen erstellt werden. Hiermit müssten 8 Schaltfunktionen für die jeweiligen D–Flipflops ermittelt und (z.B. mit dem Verfahren von Quine McCluskey) minimiert werden. Obwohl es sich bei diesem Beispiel um eine relativ einfache Aufgabenstellung handelt und außer der Taktvariablen keine weiteren Eingangsvariablen vorliegen, ist ein Entwurf mit einer Übergangstabelle nicht durchführbar. ◇

Schaltwerke für komplizierte Aufgabenstellungen, d.h. Schaltwerke mit einer großen Zahl von Zuständen, wollen wir als *komplexe Schaltwerke* bezeichnen. Um bei komplexen Schaltwerken einen effektiven und übersichtlichen Entwurf zu erreichen, nimmt man eine Aufteilung in zwei kooperierende Teilschaltwerke vor: ein *Operationswerk* (data path) und ein *Steuerwerk* (control path).

Die funktionale Aufspaltung in eine *verarbeitende* und eine *steuernde* Komponente vereinfacht erheblich den Entwurf, da beide Teilschaltwerke getrennt voneinander entwickelt und optimiert werden können.

Beispiele für komplexe Schaltwerke sind:

- Gerätesteuerungen (embedded control)
- Ampelsteuerung mit flexiblen Phasen
- arithmetische Operationen in Rechenwerken

1.3 Kooperierende Schaltwerke

Abb. 1.5 zeigt den Aufbau eines komplexen Schaltwerks. Wie bei einem normalen Schaltwerk sind funktional zusammengehörende Schaltvariablen zu Vektoren zusammengefasst. Das Operationswerk bildet den verarbeitenden Teil, d.h. hier werden die Eingabevektoren X schrittweise zu Ausgabevektoren Y umgeformt.

Ausgangspunkt für den Entwurf eines Operationswerks ist ein *Hardware–Algorithmus*, der die einzelnen Teilschritte festlegt. Der Eingabevektor, Zwischenergebnisse und der Ausgabevektor werden in Registern gespeichert und werden über Schaltnetze (arithmetisch oder logisch) miteinander verknüpft. Die Datenpfade zwischen den Registern und solchen *Operations*–Schaltnetzen können mit Hilfe von Multiplexern und/oder Bussen geschaltet werden.

Die in einem Taktzyklus auszuführenden Operationen und zu schaltenden Datenpfade werden durch den *Steuervektor* bestimmt. Mit jedem Steuerwort wird ein Teilschritt des Hardware–Algorithmus abgearbeitet. Das Steuerwerk hat die Aufgabe, die richtige Abfolge von Steuerworten zu erzeugen. Dabei berücksichtigt es die jeweilige Belegung des *Statusvektors*, der besondere Ergebnisse aus dem vorhergehenden Taktzyklus anzeigt. Über eine Schaltvariable des Statusvektors kann z.B. gemeldet werden, dass ein Registerinhalt den Wert Null angenommen hat. Bei Abfrage dieser Schaltvariable kann das Steuerwerk auf die momentane Belegung des betreffenden Registers reagieren. So ist es z.B. möglich, in einem Steueralgorithmus Schleifen einzubauen, die verlassen werden, wenn das Register mit dem Schleifenzähler den Wert Null erreicht hat.

Wir können *universelle* und *anwendungsspezifische* Operationswerke unterscheiden. Mit *einem* universellen Operationswerk können alle berechenbaren Problemstellungen gelöst werden. Lediglich der Steueralgorithmus muss an die Problemstellung angepasst werden. Bei einem von NEUMANN–Rechner stellt z.B. das *Rechenwerk* ein universell einsetzbares Operationswerk dar. Es kann eine begrenzte Zahl von Programmvariablen aufnehmen und diese arithmetisch oder logisch miteinander verknüpfen. Alle Aufgabenstellungen müssen dann letztendlich auf die im Rechenwerk verfügbaren Operationen zurückgeführt werden.

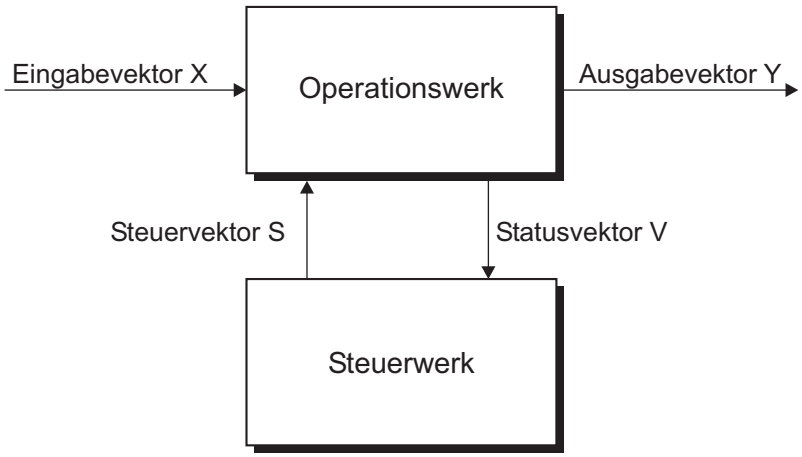


Abb. 1.5. Aufbau eines komplexen Schaltwerks

Mit Hilfe des Leitwerks bildet man aus diesen elementaren (Mikro–)Operationen höherwertige *Makro–Operationen* oder *Maschinenbefehle*. Zu jedem dieser Maschinenbefehle gehört ein Steueralgorithmus, der durch einen *Operationscode* ausgewählt wird. Leitwerke sind also *schaltbare* Steuerwerke, die durch die Befehle in Maschinenprogrammen umgeschaltet werden.

Wir werden später sehen, dass die Komplexität der im Operationswerk realisierten Operationen direkten Einfluss auf die Länge der Steueralgorithmen hat. Bei einem von NEUMANN–Rechner muss demnach ein Abgleich zwischen dem Hardware–Aufwand für Rechen– und Leitwerk erfolgen.

Dies erklärt auch die Unterschiede zwischen den sogenannten CISC– und RISC–Architekturen[3]. Bei CISC–Architekturen findet man umfangreiche Befehlssätze und Adressierungsmöglichkeiten, um die Operanden eines Befehls auszuführen. Daher muss auch ein hoher Aufwand zur Realisierung und Mikroprogrammierung des sogenannten Leitwerks[4] betrieben werden.

Bei RISC–Architekturen beschränkt man sich auf einen minimalen Befehlssatz und verzichtet ganz auf komplizierte Adressierungsarten. Man spricht daher auch von LOAD/STORE–Architekturen. Die Ablaufsteuerung vereinfacht sich dadurch erheblich, d.h. man kann auf die Mikroprogrammierung verzichten und festverdrahtete Leitwerke verwenden. Die Einsparungen zur Realisierung des Leitwerks nutzt man dann, um komplexere Operationen direkt als Schaltnetze zu realisieren. Die Implementierung von Befehlen erfordert dadurch weniger Mikrooperationen (Taktschritte) und das vereinfacht wiederum den Aufbau des Leitwerks.

[3] Complex bzw. Reduced Instruction Set Computer.
[4] Entspricht dem Steuerwerk eines komplexen Schaltwerks.

1.4 Konstruktionsregeln für Operationswerke

Da eine gegebene Aufgabenstellung durch beliebig viele verschiedene Algorithmen lösbar ist, gibt es für den Entwurf eines komplexen Schaltwerks keine eindeutige Lösung. Die *Kunst* des Entwurfs besteht darin, einen guten Kompromiss zwischen Hardwareaufwand und Zeitbedarf zu finden.

Ist die Architektur des Operationswerks vorgegeben, so muss der Steueralgorithmus daran angepasst werden. Dies ist z.B. bei einem Rechenwerk eines von NEUMANN–Rechners der Fall. Das Rechenwerk ist ein universelles Operationswerk, das zur Lösung beliebiger (berechenbarer) Problemstellungen benutzt werden kann.

Wenn dagegen die Architektur des Operationswerks frei gewählt werden darf, hat der Entwickler den größten Spielraum — aber auch den größten Entwurfsaufwand. Für diesen Fall sucht man ein an die jeweilige Problemstellung angepasstes Operationswerk. Wir werden im folgenden Konstruktionsregeln für den Entwurf solcher anwendungsspezifischen Operationswerke angeben.

Ausgangspunkt ist stets ein Algorithmus, der angibt, wie der Eingabevektor zum Ausgabevektor verarbeitet werden soll. Diese Berechnungsvorschrift enthält Anweisungszeilen mit Variablen, Konstanten, Operatoren, Zuweisungen oder bedingten Verzweigungen. Daraus folgt für die Architektur des Operationswerks:

1. Für jede Variable, die auf der linken Seite einer Zuweisung steht, ist ein Register erforderlich. Um die Zahl der Register zu minimieren, können sich zwei oder mehrere Variablen auch ein Register teilen. Voraussetzung für eine solche Mehrfachnutzung ist aber, dass diese Variablen nur eine begrenzte „Lebensdauer“ haben und dass sie nicht gleichzeitig benutzt werden.
2. Wenn einer Variablen mehr als ein Ausdruck zugewiesen wird, muss vor die Eingänge des Registers ein Multiplexer (Auswahlnetz) geschaltet werden. Um die Zahl der Verbindungsleitungen (Chipfläche) zu reduzieren, können die Register auch in einem Registerblock zusammengefasst werden. Der Zugriff erfolgt in diesem Fall *zeitversetzt* über einen Bus. Soll auf zwei oder mehrere Register gleichzeitig zugegriffen werden, so muss der Registerblock über eine entsprechende Anzahl von Busschnittstellen verfügen, die auch als *Ports* bezeichnet werden. Man unterscheidet Schreib– und Leseports. Ein bestimmtes Register kann zu einem bestimmtem Zeitpunkt natürlich nur von einem Schreibport beschrieben werden, es kann aber gleichzeitig an zwei oder mehreren Leseports ausgelesen werden.
3. Konstanten können fest verdrahtet sein, oder sie werden durch einen Teil des Steuervektors definiert.
4. Die Berechnung der Ausdrücke auf den rechten Seiten von Wertzuweisungen erfolgt mit Schaltnetzen, welche die erforderlichen Operationen mit Hilfe Boolescher Funktionen realisieren.

5. Wertzuweisungen an unterschiedliche Variablen können parallel (gleichzeitig) ausgeführt werden, wenn sie im Hardware Algorithmus unmittelbar aufeinander folgen und wenn keine Datenabhängigkeiten zwischen den Anweisungen bestehen. Eine (echte) Datenabhängigkeit liegt dann vor, wenn die vorangehende Anweisung ein Register verändert, das in der nachfolgenden Anweisung als Operand benötigt wird.
6. Zur Abfrage der Bedingungen für Verzweigungen müssen entsprechende Statusvariablen gebildet werden.

Die gleichzeitige Ausführung mehrerer Anweisungen verringert die Zahl der Taktzyklen und verkürzt damit die Verarbeitungszeit. Andererseits erhöht dies aber den Hardwareaufwand, wenn in den gleichzeitig abzuarbeitenden Wertzuweisungen dieselben Operationen auftreten. Die Operations–Schaltnetze für die parallel ausgeführten Operationen müssen dann mehrfach vorhanden sein.

In den nachfolgenden Beispiel–Operationswerken wird die Anwendung der oben angegebenen Konstruktionsregeln demonstriert. Dabei wird deutlich, dass komplexe Operationen wie z.B. eine Multiplikation auch durch einfachere Operations–Schaltnetze in Verbindung mit einem geeigneten Steueralgorithmus realisiert werden können. Diese Vorgehensweise ist kennzeichnend für CISC–Architekturen. Wenn komplexe Operationen, die oft gebraucht werden, bereits durch Schaltnetze bereitgestellt werden, vereinfacht sich der Steueralgorithmus. Wenn es für jede Maschinenoperation ein besonderes Operations–Schaltnetz gibt, erfolgt ihre Ausführung in einem einzigen Taktzyklus. Der Operationscode wird lediglich in ein einziges Steuerwort umcodiert. Diese Strategie liegt den RISC–Architekturen zugrunde. Der Idealzustand, dass pro Taktzyklus ein Befehl ausgeführt wird, lässt sich jedoch nur näherungsweise erreichen.

1.5 Entwurf des Steuerwerks

Die Struktur eines Steuerwerks ist unabhängig von einem bestimmten Steueralgorithmus. Wir haben bereits zwei Möglichkeiten für den Entwurf eines Steuerwerks kennengelernt.

Für den systematischen Entwurf kommen drei Möglichkeiten in Frage:

1. Entwurf mit Zustandgraph/Übergangstabelle und optimaler Zustandscodierung[5],
2. wie 1., allerdings mit einem Flipflop pro Zustand (one–hot coding),
3. Entwurf als Mikroprogrammsteuerwerk.

Die beiden erstgenannten Methoden kommen nur in Frage, wenn die Zahl der Zustände nicht zu groß ist (z.B. bis 32 Zustände). Für umfangreichere

[5] Wird auch als Finite State Machine (FSM) Design bezeichnet.

2. von NEUMANN–Rechner

Der von NEUMANN–Rechner ergibt sich als Verallgemeinerung eines komplexen Schaltwerkes. Im letzten Kapitel haben wir gesehen, dass beim komplexen Schaltwerk (nach der Eingabe der Operanden) immer nur ein einziger Steueralgorithmus aktiviert wird. Angenommen wir hätten Steueralgorithmen für die vier Grundrechenarten auf einem universellen Operationswerk entwickelt, das lediglich über einen Dualaddierer verfügt. Durch eine Verkettung der einzelnen Steueralgorithmen könnten wir nun einen übergeordneten Algorithmus realisieren, der die Grundrechenarten als Operatoren benötigt. Die über Steueralgorithmen realisierten Rechenoperationen können durch *Opcodes*, z.B. die Ziffern 1-4, ausgewählt werden und stellen dem Programmierer Maschinenbefehle bereit, die er nacheinander aktiviert. Die zentrale Idee von NEUMANN's bestand nun darin, die Opcodes in einem *Speicher* abzulegen und sie vor der Ausführung durch das Steuerwerk selbst holen zu lassen. Ein solches Steuerwerk benötigt ein Befehlsregister für den Opcode und einen Befehlszeiger für die Adressierung der Befehle im Speicher. Man bezeichnet ein solches Steuerwerk als *Leitwerk*. Ein universelles Operationswerk, das neben einer ALU auch mehrere Register enthält bezeichnet man als *Rechenwerk.* Leitwerk und Rechenwerk bilden den *Prozessor* oder die *CPU* (für *Central Processing Unit*). Damit der Prozessor nicht nur interne Berechungen ausführen kann, sondern dass auch von außen (Benutzer) Daten ein- bzw. ausgegeben werden können, benötigt der von NEUMANN–Rechner eine Schnittstelle zur Umgebung, die als *Ein-/Ausgabewerk* bezeichnet wird. Im folgenden wird zunächst das Grundkonzept der vier Funktionseinheiten beschrieben und dann werden Maßnahmen vorgestellt, die sowohl die Implementierung als auch die Programmierung vereinfachen. Dann wird ausführlich auf den Aufbau von Rechen- und Leitwerk eingegangen. Am Ende dieses Kapitels wird ein Simulationsprogramm für ein einfaches Rechenwerk vorgestellt, mit dem die Mikroprogrammierung von Maschinenbefehlen erarbeitet werden kann.

2.1 Grundkonzept

Ein von NEUMANN–Rechner besteht aus einem verallgemeinerten komplexen Schaltwerk, das um einen Speicher und eine Ein–/Ausgabe erweitert wird. Wir

unterscheiden insgesamt vier Funktionseinheiten: Rechenwerk, Leitwerk, Speicher und Ein–/Ausgabe. Rechenwerk und Leitwerk bilden den *Prozessor* bzw. die *CPU* (Central Processing Unit). Das Blockschaltbild in Abb. 2.1 zeigt, wie die einzelnen Komponenten miteinander verbunden sind. Die Verbindungen wurden entsprechend ihrer Funktion gekennzeichnet: Datenleitungen sind mit D, Adressleitungen mit A und Steuerleitungen mit dem Buchstaben S gekennzeichnet. Die Zahl der Datenleitungen bestimmt die Maschinenwortbreite eines Prozessors.

Rechenwerk. Wie bereits im Kapitel 1 angedeutet wurde, kann man ein *universelles Operationswerk* entwerfen, das elementare arithmetische und logische Operationen beherrscht und über mehrere Register verfügt. Diese Register werden durch Adressen ausgewählt und nehmen die Operanden (Variablen oder Konstanten) auf, die miteinander verknüpft werden sollen. Ein steuerbares Schaltnetz, das als ALU (Arithmetic Logic Unit) bezeichnet wird, kann jeweils zwei Registerinhalte arithmetisch oder logisch miteinander verknüpfen und das Ergebnis auf den Registerblock zurückschreiben. Meist ist auch eine Schiebeeinrichtung (Shifter) vorhanden, mit der die Datenbits um eine Stelle nach links oder rechts verschoben werden können. Der Shifter ist besonders nützlich, wenn zwei binäre Zahlen multipliziert oder dividiert werden sollen, und die ALU nur addieren kann. Das Status–Register dient zur Anzeige besonderer Ergebnisse, die das Leitwerk auswertet, um bedingte Verzweigungen in Steueralgorithmen für Maschinenbefehle auszuführen. Die einzelnen Bits des Status–Registers bezeichnet man als *Flags*. Das Status–Register kann wie ein normales (Daten–)Register gelesen und beschrieben werden.

Leitwerk.

Das Leitwerk stellt ein *umschaltbares Steuerwerk* dar, das –gesteuert durch den *Operationscode* (Opcode) der Maschinenbefehle– die zugehörigen Steueralgorithmen auswählt. Eine Folge von Maschinenbefehlen mit den dazugehörenden Daten nennt man ein Maschinenprogramm. Das Leitwerk steuert den Ablauf eines Programms, indem es Maschinenbefehle aus dem Speicher holt, im *Befehlsregister* IR (Instruction Register) speichert und die einzelnen Operationscodes in eine Folge von Steuerworten (Steueralgorithmus) umsetzt. Es arbeitet zyklisch, d.h. die Holephase (fetch) und die Ausführungsphase (execute) wiederholen sich ständig (Abb. 2.2). Der *Befehlszähler* PC (Program Counter) wird benötigt, um die Befehle im Speicher zu adressieren. Während der Holephase zeigt sein Inhalt auf den nächsten Maschinenbefehl. Nachdem ein neuer Maschinenbefehl geholt ist, wird der Befehlszähler erhöht.

Wie man sieht wird ein Maschinenbefehl in mehreren Teilschritten abgearbeitet. Die Holephase ist für alle Befehle gleich. Damit ein neuer Maschinenbefehl in einem Taktzyklus geholt werden kann, setzt man eine Speicherhierarchie mit einem schnellen Cache-Speicher ein (vgl. Kapitel 8). Abhängig von der Befehlssatzarchitektur des Prozessors (vgl. Kapitel 4) kann die Ausführungsphase eine variable oder feste Anzahl von Taktschritten umfassen. Aus Abb. 2.2 entnimmt man, dass ein Prozessor mindestens drei Befehlsklassen unterstützt:

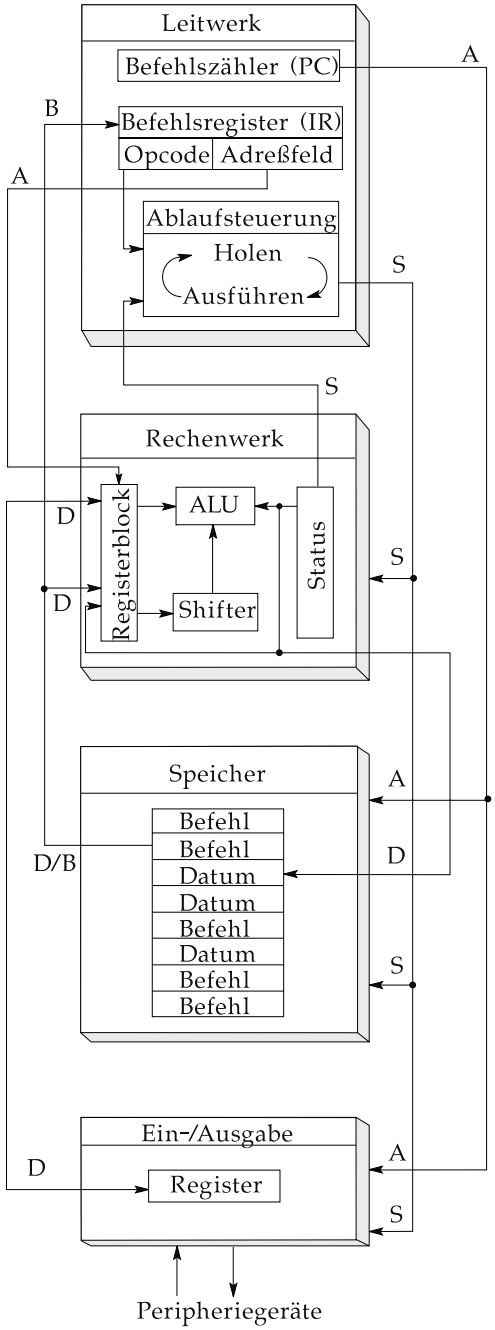


Abb. 2.1. Blockschaltbild eines von NEUMANN–Rechners

Speicher–, Verknüpfungs– und Verzweigungsbefehle. Im folgenden geben wir jeweils die notwendigen Verarbeitungsschritte für die Maschinenbefehle aus diesen Klassen an:

1. Datentransfer zwischen Speicher (bzw. Cache) und Prozessor

Hier muss zwischen Lese- und Schreibzugriffen unterschieden werden:

a) Lesen (load)

- Neuen PC-Wert bestimmen,
- Bestimmung der Speicheradresse des Quelloperanden,
- Lesezugriff auf den Speicher (Speicheradresse ausgeben),
- Speichern des gelesenen Datums in das Zielregister.

b) Schreiben (store)

- Neuen PC-Wert bestimmen,
- Bestimmung der Speicheradresse des Zieloperanden,
- gleichzeitig kann der Inhalt des Quellregisters ausgelesen werden,
- Schreibzugriff auf den Speicher (Speicheradresse und Datum ausgeben),
- Datum speichern.

Die Speicheradresse wird als Summe eines Registerinhalts und einer konstanten Verschiebung (displacement, offset) bestimmt. Beide Angaben sind Bestandteil des Maschinenbefehls und werden im *Adressfeld* (s.u.) spezifiziert.

2. Verknüpfung von Operanden

- Neuen PC-Wert bestimmen,
- Auslesen der beiden Operanden aus dem Registerblock,
- Verknüpfung in der ALU,
- Schreiben des Ergebnisses in den Registerblock.

Die Registeradressen sind Bestandteil des Maschinenbefehls und werden neben dem Opcode im Adressfeld angegeben.

als beim Entwurf mit einer Zustandstabelle. Trotzdem bleibt der Geschwindigkeitsvorteil eines festverdrahteten Leitwerks erhalten. Bei RISC–Prozessoren findet lediglich eine Umcodierung des Opcodes statt, d.h. es wird keine *zentrale* Ablaufsteuerung benötigt. Da RISC–Prozessoren nach dem Pipeline–Prinzip arbeiten (vgl. auch Kapitel 3, Abschnitt *Pipeline–Prozessoren*), kann man hier von einer *verteilten* Ablaufsteuerung sprechen.

2.6 Mikroprogrammierung einer RALU

Im folgenden wird das Programm **ralu** beschrieben, das ein Rechenwerk (Register + ALU = RALU) simuliert. Dieses Programm ist ähnlich aufgebaut wie das Programm **opw**, das bereits in Kapitel 1 vorgestellt wurde. Wir setzen die Kenntnis dieser Beschreibung hier voraus. Der Quellcode und ein ablauffähiges PC–Programm sind unter folgender Webseite abgelegt: **www.uni-koblenz.de/∼schiff/ti/bd2/**.

2.6.1 Aufbau der RALU

Die RALU besteht aus einer ALU und einem Registerfile mit 16x16–Bit Registern, von denen zwei Register als Operanden und ein Register für das Ergebnis einer ALU–Operation gleichzeitig adressiert werden können. Es handelt sich also um eine Drei–Adress Maschine wie in Abb. 2.7 dargestellt. Die ALU simuliert (bis auf zwei Ausnahmen) die Funktionen der integrierten 4–Bit ALU SN 74181. Man kann sich vorstellen, dass 4 solcher Bausteine parallel geschaltet wurden, um die Wortbreite von 16–Bit zu erreichen. Zwei weniger wichtige Operationen der ALU wurden durch Schiebe–Operationen ersetzt. Neben den (Daten–)Registern gibt es noch ein Statusregister, dessen Flags mit einer Prüfmaske getestet werden können. Der Aufbau der simulierten RALU ist in Abb. 2.15 dargestellt.

2.6.2 Benutzung des Programms

Das Programm wird wie folgt gestartet:

```
ralu [Optionen] [Dateiname]
```

Die möglichen Optionen wurden bereits in Kapitel 1 im Abschnitt *Simulationsprogramm eines Operationswerkes* beschrieben. Das Gleiche gilt für die möglichen Betriebsarten. Deshalb wird hier nicht näher darauf eingegangen.

Die RALU wird immer nach dem gleichen Schema gesteuert: Zu Beginn werden die Register mit Startwerten geladen. Dann wird ein Steuerwort vorgegeben, das eine ALU–Operation und die Operanden– und Ergebnisregister

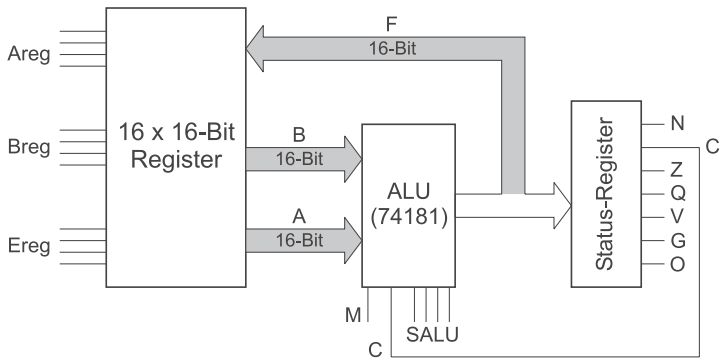


Abb. 2.15. Blockschaltbild der simulierten RALU

auswählt. Die so bestimmte Mikrooperation wird mit einem Takt–Befehl ausgeführt. Durch Hintereinanderschalten mehrerer Mikrobefehle kann schließlich die gewünschte Funktion schrittweise realisiert werden. Mit dem Befehl `quit` wird das Programm beendet.

2.6.3 Setzen von Registern

Mit dem Befehl `set` *Registernummer Konstante* können einzelne Register direkt mit Werten geladen werden. Die Registernummer kann hierbei Werte von 0 bis 15 annehmen und muss immer dezimal (ohne #) angegeben werden. Als Konstanten werden nur *positive* Zahlen in den drei möglichen Zahlendarstellungen akzeptiert. Die Zahlen müssen innerhalb des Wertebereiches 0 bis 65535 ($0000 bis $ffff) liegen.

Beispiel. Der Befehl `set 2 %10011` lädt Register 2 mit dem dezimalen Wert 19. Äquivalente Schreibweisen sind `set 2 #19` oder `set 2 $13`. Neben den Registern kann auch das Carry–Flag gesetzt werden. Hierzu dient der Befehl `carry 0` oder `carry 1`.

2.6.4 Steuerwort der RALU

Das Steuerwort der RALU wird mit dem Befehl `control` *Steuerwort* gesetzt. Das Steuerwort ist eine 17–Bit Zahl und setzt sich wie folgt aus 5 Teilwörtern zusammen:

M	SALU				Areg				Breg				Ereg			
M	S_3	S_2	S_1	S_0	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	F_3	F_2	F_1	F_0

Hierbei sind:

M: Modusbit der ALU. Ist M gelöscht, so werden arithmetische Operationen ausgeführt.

SALU: Steuerwort der ALU, laut der nachfolgenden Tabelle der ALU–Funktionen.

Areg: Adresse des Registers, dessen Inhalt dem Eingang A zugeführt werden soll.

Breg: dito, für den Eingang B.

Ereg: Adresse des Registers, in welches das Ergebnis der ALU geschrieben werden soll.

Steuerung	Rechenfunktionen		
SALU	M=1	M=0; Arithmetische Funktionen	
	Logische	C=0	C=1
$S_3S_2S_1S_0$	Funktionen	(kein Übertrag)	(mit Übertrag)
0 0 0 0	$F = \overline{A}$	$F = A$	$F = A + 1$
0 0 0 1	$F = \overline{A \vee B}$	$F = A \vee B$	$F = (A \vee B) + 1$
0 0 1 0	$F = \overline{A} \wedge B$	$F = A \vee \overline{B}$	$F = (A \vee \overline{B}) + 1$
0 0 1 1	$F = 0$	$F = -1$ (2erKompl.)	$F = 0$
0 1 0 0	$F = \overline{A \wedge B}$	$F = A + (A \wedge \overline{B})$	$F = A + (A \wedge \overline{B}) + 1$
0 1 0 1	$F = \overline{B}$	$F = (A \vee B) + (A \wedge \overline{B})$	$F = (A \vee B) + (A \wedge \overline{B}) + 1$
0 1 1 0	$F = A \oplus B$	$F = A - B - 1$	$F = A - B$
0 1 1 1	$F = A \wedge \overline{B}$	$F = (A \wedge \overline{B}) - 1$	$F = A \wedge \overline{B}$
1 0 0 0	$F = \overline{A} \vee B$	$F = A + (A \wedge B)$	$F = A + (A \wedge B) + 1$
1 0 0 1	$F = \overline{A \oplus B}$	$F = A + B$	$F = A + B + 1$
1 0 1 0	$F = B$	$F = (A \vee \overline{B}) + (A \wedge B)$	$F = (A \vee \overline{B}) + (A \wedge B) + 1$
1 0 1 1	$F = A \wedge B$	$F = (A \wedge B) - 1$	$F = A \wedge B$
1 1 0 0	$F = 1$	$F = A << B$ (rotate left)[12]	$F = A >> B$ (rotate right)
1 1 0 1	$F = A \vee \overline{B}$	$F = (A \vee B) + A$	$F = (A \vee B) + A + 1$
1 1 1 0	$F = A \vee B$	$F = (A \vee \overline{B}) + A$	$F = (A \vee \overline{B}) + A + 1$
1 1 1 1	$F = A$	$F = A - 1$	$F = A$

Beispiel. Soll die ALU die Register 3 und 4 UND–verknüpfen und dann das Ergebnis in Register 1 ablegen, so muss das Steuerwort wie folgt spezifiziert werden: `control %0101100110100000 1` oder `control $0b341` oder `control #45889` . Die letzte Darstellungsmöglichkeit des Steuerworts ist allerdings wenig übersichtlich.

2.6.5 Takten und Anzeigen der RALU

Nachdem das Steuerwort der RALU festgelegt und gesetzt worden ist, muss die RALU getaktet werden, um die gewünschten Funktionen auszuführen.

[12] Diese Notation bedeutet: Shifte den Eingang A um soviele Positionen nach links, wie der Wert am Eingang B angibt.

Dies geschieht mit dem Befehl `clock`. Dabei wird dann die gewählte Operation ausgeführt und das Ergebnis in den Registerblock übertragen. Außerdem wird das Statusregister dem ALU–Ergebnis entsprechend aktualisiert.

DUMP–BEFEHL. Mit dem `dump`–Befehl werden alle Registerinhalte in hexadezimaler Darstellung sowie Statusregister und Steuerwort in binärer Darstellung angezeigt.

2.6.6 Statusregister und Sprungbefehle

Das Statusregister der RALU beinhaltet verschiedene Status–Flags, die besondere Eigenschaften von ALU–Ergebnissen festhalten. Das Register besteht aus 8 Bit, von denen nur 7 Bit belegt sind:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
N	C	Z	Q	V	G	-	O

Diese 7 Flags sind:

- N: Negativ–Flag. Dieses Flag ist gesetzt, wenn das Ergebniswort der ALU in der Zweierkomplement–Darstellung eine negative Zahl ist.
- C: Carry–Flag. Das Flag ist gesetzt, wenn ein Übertrag vorliegt.
- Z: Zero–Flag. Gibt an, ob das Ergebnis gleich Null ist.
- Q: Equal–Flag. Gesetzt, wenn die Inhalte der Register A und B übereinstimmen.
- V: Overflow–Flag. Kennzeichnet einen Überlauf des darstellbaren Wertebereiches.
- G: Greater–Flag. Gesetzt, wenn der Inhalt von Register A größer als der von Register B ist.
- O: Odd–Flag. Gesetzt, wenn das Ergebnis der ALU eine ungerade Zahl ist.

Im Programm–Modus können die Inhalte der Flags getestet werden. Je nach ihrem Zustand verzweigt das Mikroprogramm zu einer Programm–Marke (Label). Auf diese Weise ist es z.B. möglich Schleifen zu programmieren. Steht zu Beginn einer Zeile das Symbol „>“ und unmittelbar danach ein Wort, so wird diese Stelle als Sprungmarke definiert.

Der Befehl `jmpcond` *Prüfmaske Marke* bildet eine UND–Verknüpfung aus dem Statusregister und der 8 Bit Prüfmaske. Ist das Ergebnis dieser Verknüpfung ungleich Null, so wird die Marke angesprungen. Ansonsten wird der dem `jmpcond` folgende Befehl ausgeführt. Bei `jpncond` *Prüfmaske Marke* wird ebenfalls das Statusregister mit der Prüfmaske UND–verknüpft. Es wird allerdings zur Marke gesprungen, wenn das Ergebnis gleich Null ist.

BEISPIELE. Bei `jmpcond $40 loop` wird die Marke `loop` angesprungen, wenn das Carry–Bit gesetzt ist. Nach dem Befehl `jpncond $88 ok` wird die

Programmausführung nur dann bei der Marke `ok` fortgesetzt, wenn weder das Negativ– noch das Overflow–Flag gesetzt sind. Die Befehle `jmpcond` und `jpncond` dürfen nur im Programm–Modus verwendet werden.

2.6.7 Kommentare und Verkettung von Befehlen

Mit diesen beiden Konstruktoren kann die Übersichtlichkeit von Mikroprogrammen verbessert werden. Mit dem Konstrukt „:“ werden Befehle verkettet, d.h. sie dürfen innerhalb einer Zeile stehen.

BEISPIEL. `clock : dump` taktet zuerst die RALU und zeigt dann die Registerinhalte an.

Das Konstrukt „;“ erlaubt es, die Mikroprogramme zu dokumentieren. Nach einem Semikolon wird der folgende Text bis zum Zeilenende ignoriert. Damit können Programme übersichtlicher und verständlicher werden.

BEISPIEL. `control $09120 ; Reg0=Reg1+Reg2` erläutert die Wirkung des angegebenen Steuerworts.

2.6.8 Beispielprogramme

Zum Schluss sollen drei Beispielprogramme vorgestellt werden. Das erste Mikroprogramm addiert einfach zwei Register und zeigt die Bildschirmausgabe, die während der Simulation erzeugt wird. Die beiden anderen Aufgaben sind uns schon aus dem Kapitel 1 bekannt. Ein Programm berechnet den ganzzahligen Teil des Logarithmus zur Basis 2 und das letzte Programm berechnet den ganzzahligen Teil der Quadratwurzel einer beliebigen Zahl aus dem Wertebereich $0 \ldots 32767$.

1. Programm: Addition zweier Register

```
;
;          ** Addition zweier Register **
;
set     1       #4      ; Reg1=Operand 1
set     2       #5      ; Reg2=Operand 2
;
dump                    ; Register ausgeben
control $09120          ; Reg0=Reg1+Reg2
carry   0               ; Carrybit loeschen
dump                    ; Register ausgeben
clock                   ; Takten->Ausfuehrung der Addition
dump                    ; Register ausgeben
quit                    ; Programm beenden
```

Dieses Mikroprogramm sei unter dem Namen `add.ral` abgespeichert. Nach dem Aufruf mit `ralu -aotx add.ral` erhält man dann folgende Ausgabe:

```
1.Befehl: add.ral

calling add.ral:

1.Befehl: set 1 #4 ; Reg1=Operand 1
2.Befehl: set 2 #5 ; Reg2=Operand 2
3.Befehl: dump ; Register ausgeben

RALU-Zustand nach dem 0.ten Taktimpuls:
--------------------------------------------------------------------------------
0.Register: $0000       1.Register: $0004       2.Register: $0005
3.Register: $0000       4.Register: $0000       5.Register: $0000
6.Register: $0000       7.Register: $0000       8.Register: $0000
9.Register: $0000      10.Register: $0000      11.Register: $0000
12.Register: $0000      13.Register: $0000      14.Register: $0000
15.Register: $0000

NCZQVG-O              M SALU              Areg Breg Ereg
Status: 00000000         ALU: 0 0000       Busse: 0000 0000 0000
--------------------------------------------------------------------------------

4.Befehl: control $09120 ; Reg0=Reg1+Reg2
5.Befehl: carry 0 ; Carrybit loeschen
6.Befehl: dump ; Register ausgeben

RALU-Zustand nach dem 0.ten Taktimpuls:
--------------------------------------------------------------------------------
0.Register: $0000       1.Register: $0004       2.Register: $0005
3.Register: $0000       4.Register: $0000       5.Register: $0000
6.Register: $0000       7.Register: $0000       8.Register: $0000
9.Register: $0000      10.Register: $0000      11.Register: $0000
12.Register: $0000      13.Register: $0000      14.Register: $0000
15.Register: $0000

NCZQVG-O              M SALU              Areg Breg Ereg
Status: 00000000         ALU: 0 1001       Busse: 0001 0010 0000
--------------------------------------------------------------------------------

7.Befehl: clock ; Takten->Ausfuehrung der Addition
8.Befehl: dump ; Register ausgeben

RALU-Zustand nach dem 1. Taktimpuls:
--------------------------------------------------------------------------------
0.Register: $0009       1.Register: $0004       2.Register: $0005
3.Register: $0000       4.Register: $0000       5.Register: $0000
6.Register: $0000       7.Register: $0000       8.Register: $0000
9.Register: $0000      10.Register: $0000      11.Register: $0000
12.Register: $0000      13.Register: $0000      14.Register: $0000
15.Register: $0000

NCZQVG-O              M SALU              Areg Breg Ereg
Status: 00000001         ALU: 0 1001       Busse: 0001 0010 0000
--------------------------------------------------------------------------------

9.Befehl: quit ; Programm beenden
```

2. Programm: Logarithmus zur Basis 2

```
;
;                       Beispielprogramm zu
;            W.Schiffmann/R.Schmitz: "Technische Informatik",
;        Band 2: Grundlagen der Computertechnik, Springer-Verlag, 1992
;
;     (c)1991 von W.Schiffmann, J.Weiland        (w)1991 von J.Weiland
;
;
set     0       #256    ; Operand, Wertebereich: 0-65535
;
set     1       $ffff   ; Ergebnis (vorlaeufig 'unendlich')
set     2       #1      ; Anzahl der Rotationen nach rechts
;
control $1f003          ; Reg[3]=Reg[0]
clock                   ; Reg[0] nach Reg[3] retten
;
; Solange wie es geht, wird durch 2 geteilt
;
>loop           control $1f000          ; mit Reg[0] den Status setzen
clock
jmpcond $20     end     ; Wenn A gleich Null, dann ende
;
; Division durch 2 durch Verschiebung um eine Position nach rechts
;
control $0c020          ; Reg[0]=Reg[0]>>Reg[2]
carry   1
clock
;
carry   1
control $00101          ; Reg[1]=Reg[1]+1
clock
jpncond $00     loop    ; unbedingter Sprung nach loop
;
; Ende der Berechnung, Operand nach Reg[0], Ergebnis in Reg[1]
;
>end            control $1f300          ; Reg[3] nach Reg[0]
clock
set     2       #0
set     3       #0
dump                    ; Ergebnis ausgeben
quit
```

3. Programm: Quadratwurzelberechnung nach dem Newtonschen Iterationsverfahren mit der Formel

$$x_{i+1} = \frac{x_i + \frac{a}{x_i}}{2}$$

```
;
; *** Quadratwurzelberechnung fuer die RALU-Simulation nach ***
; *** dem Newtonschen Iterationsverfahren: X(i+1)=1/2*(Xi+a/Xi) ***
;
;                       Beispielprogramm zu
;            W.Schiffmann/R.Schmitz: "Technische Informatik",
;        Band 2: Grundlagen der Computertechnik, Springer-Verlag, 1992
;
;     (c)1991 von W.Schiffmann, J.Weiland        (w)1991 von J.Weiland
;
```

```
;
set     0       #144    ; Radiant, Wertebereich: 0-32767
;
; Vorbereitungen
;
; Radiant bleibt in Reg[0]
; Der Iterationswert wird in Reg[1] gehalten
;
control $1f001          ; Radiant=0 ? (Und: Reg[1]=Reg[0])
clock
jmpcond $20     end
jpncond $80     prepare ; Radiant ok => Wurzel berechnen
;
; Negativer Radiant => beenden
;
set     1       0
dump
quit
;
; Berechnen der Wurzel
;
>prepare        set     4       #1      ; Anzahl der Rotationen nach rechts
; fuer die Division durch 2
>sqrloop        control $1f002          ; Reg[2]=Reg[0]
clock
control $1f105          ; Reg[5]=Reg[1]
clock
set     3       #-1     ; Reg[3]=$ffff
;
; Division Reg[3]=Reg[2]/Reg[1] durchfuehren
;
>divloop        control $00303          ; Reg[3]=Reg[3]+1
carry   1
clock
control $06212          ; Reg[2]=Reg[2]-Reg[1]
carry   1
clock
jpncond $80     divloop ; Ergebnis positiv ?
;
; X(i+1) durch 1/2*(Reg[1]+Reg[3]) ermitteln
;
control $09311          ; Reg[1]=Reg[3]+Reg[1]
carry   0
clock
control $0c141          ; Reg[1]=Reg[1]/2: Division durch shift
carry   1
clock
;
; Test, ob Ergebnis stabil bleibt. Wegen der Rechenungenauigkeit muss auch
; getestet werden, ob sich das Ergebnis nur um 1 nach oben unterscheidet.
;
control $1f516          ; Reg[5]=Reg[1] ?
clock
jmpcond $10     end
control $0f106          ; Reg[6]=Reg[1]-1
carry   0
clock
control $1f566          ; Reg[5]=Reg[6] ?
clock
jpncond $10     sqrloop
control $1f601          ; Reg[1]=Reg[6], Ergebnis setzen
clock
;
; Ende der Berechung: Radiant in 0, Ergebnis in 1, Rest loeschen
;
>end            set     2       0
set     3       0
```

```
set     4       0
set     5       0
set     6       0
dump                    ; Ergebnis ausgeben
quit
```

Siehe Übungsbuch
Seite 58, Aufgabe 86:
Fahrenheit nach Celsius

Siehe Übungsbuch
Seite 58, Aufgabe 87:
Briggscher Logarithmus

4. Prozessorarchitektur

Im Kapitel 2 haben wir den Aufbau des von Neumann Rechners kennengelernt. Er besteht aus Prozessor (Rechen- und Leitwerk), Speicher und der Ein-/Ausgabe. In den folgenden Kapiteln werden wir uns dem Entwurf von Prozessoren zuwenden, die als komplexes Schaltwerk den Kern eines von NEUMANN-Rechners bilden. Wir haben bereits gesehen, dass das Rechenwerk im Wesentlichen durch die Datenpfade bestimmt ist. Das Leitwerk steuert diese Datenpfade und die Operationen der ALU. Maschinenbefehle werden also durch die Prozessorhardware interpretiert.

Die Schnittstelle zwischen Hardware und *low–level* Software (Maschinenbefehle) wird durch die *Befehls(satz)architektur* (Instruction Set Architecture, ISA) festgelegt. Sie beschreibt den Prozessor aus der Sicht des Maschinenprogrammierers und erlaubt eine vollständige Abstraktion von der Hardware. Die Befehlsarchitektur beschränkt sich auf die Funktionalität, die ein Prozessor haben soll, d.h. sie spezifiziert das Verhaltensmodell des Prozessors. Der Rechnerarchitekt kann für eine gegebene ISA unterschiedliche *Implementierungen* in Hardware realisieren. Eine bestimmte Implementierung umfaßt die logische Organisation des *Datenpfads* (Datapath) sowie die erforderliche *Steuerung* (Control). Die Implementierung wird durch die Art und Reihenfolge der logischen Operationen beschrieben, die ausgeführt werden, um die gewünschte Befehlsarchitektur zu erreichen. Dabei bestimmt die Befehlsarchitektur den Hardwareaufwand, um den Prozessor zu implementieren. Im Laufe der Zeit entstanden vier Klassen von Prozessorarchitekturen, die sich wesentlich bzgl. ihrer Befehlsarchitektur und Implementierung unterscheiden:

- CISC–Prozessoren (Complex Instruction Set Computer) benutzen ein mikroprogrammiertes Leitwerk.
- RISC–Prozessoren (Reduced Instruction Set Computer) basieren auf einem einfachen Pipelining und dekodieren Befehle durch ein Schaltnetz.
- Superskalare RISC–Prozessoren benutzen mehrfaches Pipelining in Verbindung mit dynamischer Befehlsplanung.
- VLIW–Prozessoren (Very Large Instruction Word) benutzen die Datenparallelität von Funktionseinheiten in Verbindung mit statischer Befehlsplanung.

Datenpfad und Steuerung kennzeichnen die *logische* Implementierung eines Prozessors. Die „Bausteine“ der logischen Implementierung sind Register, Registerblöcke, Multiplexer, ALUs, FPUs (Floating Point Unit), Speicher, und Caches. Sie werden so zu einem Datenpfad und zugehöriger Steuerung zusammengefügt, dass sie die Funktionalität der Befehlsarchitektur implementieren. Die Umsetzung der logischen Organisation in einer bestimmten Technologie bezeichnet man als *technologische Implementierung* oder auch *Realisierung*. Im Laufe der technologischen Entwicklung standen verschiedene Bauelemente zur Realisierung von Prozessoren zur Verfügung:

1. Relais und elektromechanische Bauteile
2. Elektronenröhren
3. Einzelne (diskrete) Transistoren
4. Integrierte Schaltkreise unterschiedlicher Integrationsdichten
   - SSI = Small Scale Integration
   - MSI = Medium Scale Integration
   - LSI = Large Scale Integration
   - VLSI = Very Large Scale Integration

Die technologische Implementierung umfaßt die Platzierung der Transistoren auf einem Chip und die aufgrund der Schalteigenschaften und Laufzeiten erreichbaren Taktfrequenzen. Die o.g. Epochen der technologischen Entwicklung bezeichnet man auch als *Rechnergenerationen*.

Die Rechnerarchitektur kann grob in zwei Teilbereiche unterteilt werden: Befehlsarchitektur und Implementierung. Die Implementierung läßt sich weiter in logische Organisation und technologische Realisierung unterteilen (Abb. 4.1).

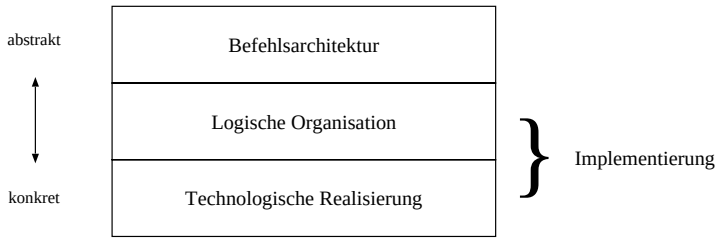


Abb. 4.1. Sichtweisen der Rechnerarchitektur

Die logische Organisation ist in hohem Maße von einer vorgegebenen Befehlsarchitektur abhängig. Je komplexer die Befehlsarchitektur, umso stärker wird der Spielraum für die logische Organisation auf mikroprogrammierte Lösungen beschränkt. Dagegen ist die technologische Realisierung weniger stark von

der logischen Organisation abhängig. Läßt man herstellungstechnische Gründe außer Betracht, so kann prinzipiell jeder logische Entwurf durch eine beliebige Schaltkreisfamilie realisiert werden.

Bisher sind wir von konkreten zu immer abstrakteren Betrachtungsweisen vorgegangen. In diesem und den beiden nachfolgenden Kapiteln werden wir von diesem Grundsatz abweichen und unsere Vorgehensweise umkehren. Im folgenden wird auf die drei Sichtweisen der Rechnerarchitektur näher eingegangen. Danach werden die Merkmale der verschiedenen Prozessortypen behandelt und deren (logische) Organisation diskutiert.

4.1 Befehlsarchitektur

Wenn man beim Entwurf von Prozessoren von einer einmal spezifizierten Befehlsarchitektur ausgeht, so können sich die Details der Implementierung ändern ohne dass sich diese auf die Software auswirken. Erfolgreiche Befehlsarchitekturen konnten mehrere Jahre oder gar Jahrzehnte bestehen, weil sie von technologisch verbesserten Nachfolge–Prozessoren weiter unterstützt wurden. Ein bekanntes Beispiel hierfür sind die INTEL Prozessoren aus der 80x86–Familie, deren Befehlsarchitektur 1980 eingeführt wurde und noch heute auf den modernen Pentium 4–Prozessoren unterstützt wird. Das bedeutet, dass noch heute 80x86 Objektcode auf technologisch verbesserten Prozessoren ausgeführt werden kann.

Die Befehlsarchitektur umfaßt alle Informationen, die man zur Programmierung in Maschinensprache benötigt. Hierzu gehören im Wesentlichen die Zugriffsmöglichkeiten auf die Operanden und die ausführbaren Operationen.

Die Operanden können innerhalb oder außerhalb des Prozessors gespeichert sein. Es gibt verschiedene Adressierungsarten, um auf sie zuzugreifen. Da es nicht sinnvoll ist, alle Daten mit Maschinenwortbreite darzustellen, werden verschiedene Operandentypen definiert.

Die durch eine Befehlsarchitektur bereitgestellten Operationen (Maschinenbefehle) sollten auf die Erfordernisse der Anwendungsprogramme abgestimmt sein. Hierzu wird die Nutzungshäufigkeit der geplanten Maschinenbefehle in repräsentativen Anwendungsprogrammen ermittelt. Maschinenbefehle, die vom Compiler nur selten genutzt werden, können möglicherweise gestrichen und durch eine Folge anderer Maschinenbefehlen nachgebildet werden. Man kann drei Gruppen von Maschinenbefehlen unterscheiden:

- Speicherzugriffe
- Verknüpfungen (ALU/FPU–Operationen)
- Verzweigungen

Maschinenbefehle sollten verschiedene Operandentypen verarbeiten können. Die Befehlsarchitektur muss festlegen, welche Operandentypen von welchen

einzelner Befehle durch das Pipelining sogar erhöht: einerseits durch redundante Teilschritte und andererseits durch zusätzliche Zeitverzögerungen der Pipelineregister. Würde man die Pipelineschaltnetze direkt hintereinander schalten, so würde die Latenzzeit der Befehle verringert — aber dann ist auch keine überlappte Verarbeitung mehr möglich.

Siehe Übungsbuch
Seite 60, Aufgabe 91:
CISC versus RISC

6.2 Aufbau eines RISC–Prozessors

In Abb. 6.2 ist der schematische Aufbau eines RISC–Prozessors dargestellt. In der IF–Stufe befindet sich der Programmzähler (PC–Register). Bei einem 32 Bit Prozessor mit byteweiser Speicheradressierung wird der Programmzähler in der IF–Stufe mit jedem Takt um 4 erhöht. Der aus dem Befehlsspeicher gelesene Befehl wird in das Pipelineregister zwischen IF– und ID–Stufe geschrieben. Der Opcodeteil wird dann in der ID–Stufe durch ein Schalt*netz* dekodiert. Die dabei erzeugten Steuersignale werden an die drei nachfolgenden Stufen EX, MEM und WB (über Pipelineregister gepuffert) weitergeleitet. Parallel zur Opcode–Dekodierung wird auf die Leseports des Registerblockes zugegriffen oder der Verschiebungsteil (displacement) des Adressfeldes wird für die Weitergabe an die EX–Stufe ausgewählt. Dort können arithmetische oder logische Verknüpfungen mit den Registern oder Adressrechnungen für Verzweigungs– bzw. LOAD/STORE–Befehle ausgeführt werden. Verzweigungsadressen werden in der ALU durch Addition der über die Pipelineregister verzögerten Folgebefehlsadresse ($PC + 4$) plus einer Verschiebung aus dem Adressfeld berechnet. Sie können abbhängig von dem Ausgang eines Registervergleichs auf Null in den Programmzähler zurückgeschrieben werden. In der MEM–Stufe kann auf den Datenspeicher zugegriffen werden. Es werden entweder die Daten aus einem Lesezugriff oder die von EX– über die MEM–Stufe weitergeleitete Ergebnisse einer ALU–Operation in den Registerblock geschrieben.

6.3 Pipelinekonflikte

Wenn die aufeinanderfolgenden Befehle beim Pipelining voneinander unabhängig sind, wird —bei gefüllter Pipeline— mit jedem Taktzyklus ein Befehl abgefertigt. Obwohl die Ausführung eines einzelnen Befehls fünf Taktzyklen dauert, kann man durch Befehlspipelining eine Verfünffachung des Befehlsdurchsatzes erreichen. Damit erhöht sich auch die Prozessorleistung um den

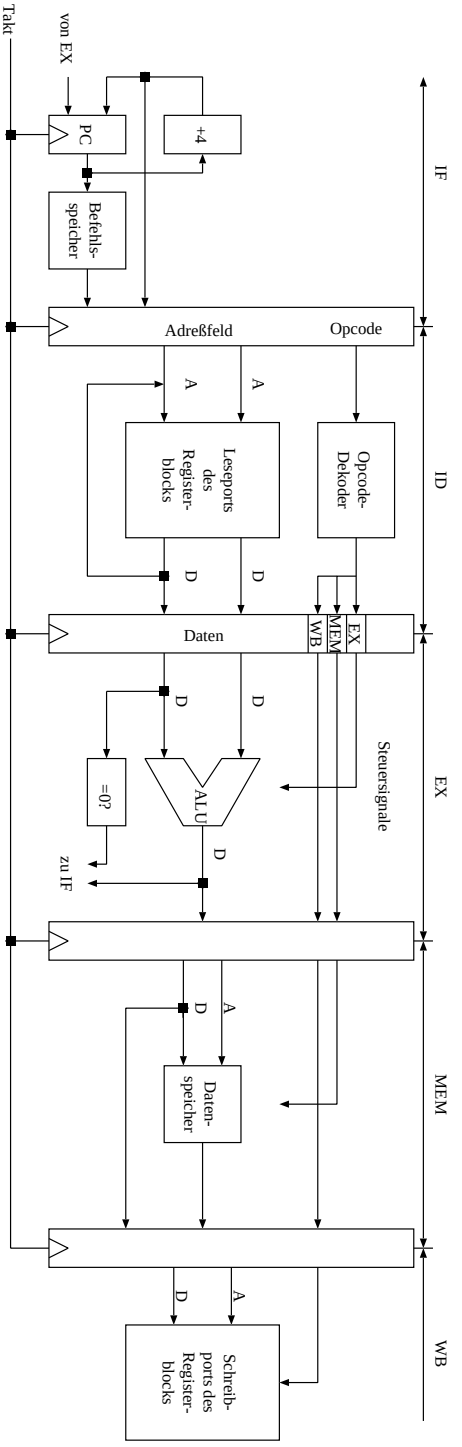


Abb. 6.2. Schematischer Aufbau eines RISC–Prozessors mit fünfstufigem Befehlspipelining und getrenntem Befehls– und Datenspeicher (Instruction/Data Memory IM/DM)

Faktor fünf. Im Idealfall ist beim Pipelining der CPI–Wert gleich eins und die Prozessorleistung erhöht sich entsprechend der Zahl der Pipelinestufen.

Leider sind bei realen Maschinenprogrammen die Befehle voneinander abhängig, d.h. es kommt zu Pipeline*konflikten*, die zu einem $CPI > 1$ führen und den Leistungsgewinn durch Befehlspipelining schmälern.

Wir können vier Arten von Pipelinekonflikten unterscheiden:

1. Strukturelle Konflikte
2. Datenflusskonflikte
3. Laufzeitkonflikte
4. Steuerflusskonflikte

Das Auftreten eines Konfliktes bewirkt, dass die Pipelineverarbeitung eines oder mehrerer nachfolgender Befehle solange eingestellt (stalled) werden muss, bis der Konflikt behoben ist. Wenn ein Konflikt vorliegt, so kann man ihn einerseits *beheben*, indem man durch den Compiler NOPs (No Operations) einfügt und somit die Pipelineverarbeitung nachfolgender Befehle verzögert. Andererseits kann man aber auch versuchen, Konflikte zu *beseitigen*. Bei der Konflikt*behebung* wird die Pipelineverarbeitung angehalten (stalled), d.h. die Pipelinestufen werden nicht optimal ausgelastet. Der CPI–Wert liegt daher deutlich über 1. Bei der Konflikt*beseitigung* wird entweder durch optimierende Compiler oder durch zusätzliche Hardware dafür gesorgt, dass der CPI–Wert möglichst nahe an 1 herankommt.

In den nächsten Abschnitten werden wir sehen, wie man Konflikte teilweise mit Hilfe optimierender Compiler, durch zusätzliche Hardwarelogik oder durch eine Kombination von beidem beseitigen kann. Zuvor sollen die Konflikte beschrieben und anhand von Beispielen erläutert werden.

Die Behebung von Pipelinekonflikten kann hardware– oder softwaremäßig erfolgen. Im erstgenannten Fall unterdrückt man einfach die Taktsignale der Pipelineregister für einen oder mehrere Taktzylen. Dadurch wird die Pipelineverarbeitung ausgesetzt (stalled). Strukturelle Pipelinekonflikte und dynamische Laufzeitkonflikte (z.B. Seitenfehler bei virtuellem Speicher) können nur hardwaremäßig behoben werden.

Die softwaremäßige Behebung von Konflikten erfolgt durch Einfügen von NOP–Befehlen, die nach ihrer Dekodierung alle nachfolgenden Pipelinestufen für einen Taktzyklus blockieren. Auch hier wird der Takt für die Pipelineregister unterdrückt. Die softwaremäßige Behebung bietet jedoch die Vorteile, dass der Hardwareaufwand gering ist und dass die NOP–Befehle mittels optimierender Compiler durch sinnvolle Operationen ersetzt werden können. Hierdurch wird es unter Umständen sogar möglich, die Pipelinekonflikte vollständig zu beseitigen. Zu den Pipelinekonflikten, die softwaremäßig behoben oder gar beseitigt werden können, zählen *statische* Laufzeitkonflikte (delayed load), unbedingte Verzweigungen (delayed branch) und Datenflusskonflikte.

steht eine Datenabhängigkeit (Data Hazard). Es gibt drei Arten von Datenabhängigkeiten, für die es eine Reihe von Synonymen gibt:

1. Read–After–Write (RAW) oder *echte* Datenabhängigkeit (essential dependence).
2. Write–After–Write (WAW) oder *Ausgabe* Datenabhängigkeit (output dependence).
3. Write–After–Read (WAR) oder *Anti/unechte* Datenabhängigkeit, Pseudoabhängigkeit (forward/ordering dependence)

RAW

Beim RAW liest der Befehl B_j den Wert von X, der zuvor durch den Befehl B_i verändert wird.

Beispiel: ADD R_3, R_1, R_2 ; $R_3 = R_1 + R_2$
ADD R_4, R_1, R_3 ; $R_4 = R_1 + R_3$

In einer Befehlspipeline mit den Stufen wie in Abb. 6.1 tritt durch RAW das Problem auf, dass die ID–Stufe des Befehls B_j den Inhalt von R_3 lesen will bevor die WB–Stufe das Ergebnis im Registerblock ablegt. Abb. 6.4 zeigt wie der RAW–Konflikt durch Einfügen von NOPs behoben werden kann. Der RAW–Konflikt kann durch einen sogenannten *Bypass* (auch *forwarding Hardware* genannt) vom Pipelineregister zwischen der EX/MEM–Stufe zurück zum ALU Eingang be*seitigt* werden (Abb. 6.5). Die Bypass Hardware befindet sich in der EX–Stufe und erkennt, dass das Ergebnis direkt aus dem Pipelineregister ausgelesen und über einen Multiplexer an den ALU Eingang weitergeleitet werden kann.

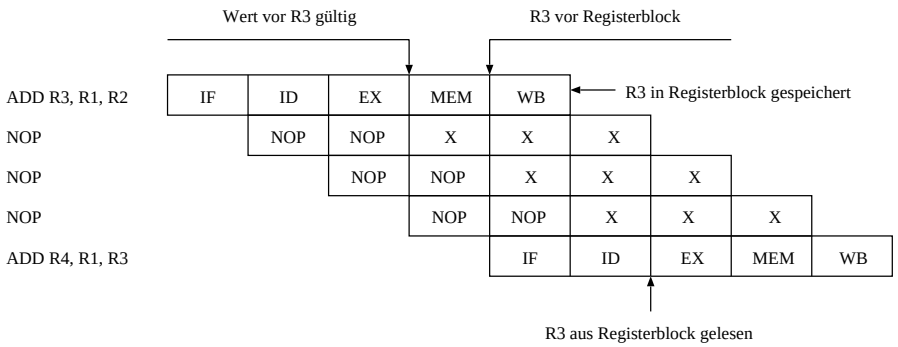


Abb. 6.4. Beispiel für die Behebung eines RAW Datenflusskonflikts durch Einfügen von NOPs

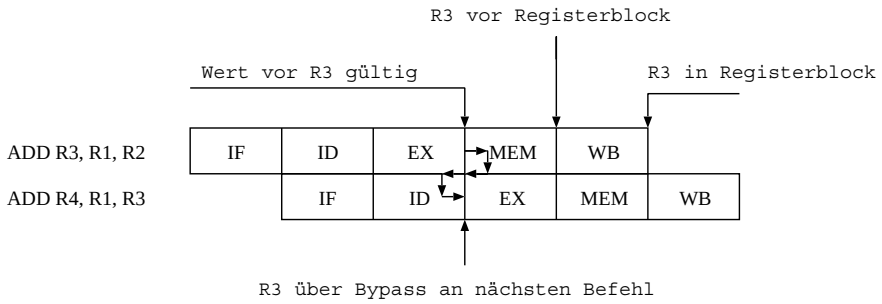


Abb. 6.5. Beseitigung des Datenflusskonflikts aus Abb. 6.4 mit Hilfe einer Bypass Hardware, die den Registerblock umgeht.

WAW

Beim WAW schreiben B_i und B_j in dasselbe Register. Wenn beide Befehle gleichzeitig gestartet werden, kann es vorkommen, dass B_j vor B_i die Variable beschreibt und danach ein falscher Wert abgespeichert ist.

Beispiel: ADD R_3, R_1, R_2 ; $R_3 = R_1 + R_2$
SUB R_3, R_4, R_5 ; $R_3 = R_4 - R_5$

WAR

Beim WAR schreibt B_j ein Register, das zuvor von B_i gelesen wird. Wenn beide Befehle gleichzeitig gestartet werden, kann es vorkommen, dass B_j schreibt bevor B_i den Wert gelesen hat. Dieser Fall tritt z.B. ein, wenn B_i eine zeitaufwendige Gleitkommaoperation und B_j eine Ganzzahloperation ist.

Beispiel: ADD R_3, R_1, R_2 ; $R_3 = R_1 + R_2$
SUB R_2, R_4, R_5 ; $R_2 = R_4 - R_5$

Während RAW lediglich bei skalaren RISCs von Bedeutung ist, müssen bei superskalaren RISCs auch WAW und WAR Datenabhängigkeiten erkannt und beseitigt werden, damit die Programmsemantik auch bei einer geänderten Ausführungsreihenfolge (out–of–order execution) erhalten bleibt.

6.3.3 Laufzeitkonflikte

Der Zugriff auf den Hauptspeicher erfordert normalerweise mehr als einen Taktzyklus. Die MEM–Phase muss daher bei den LOAD/STORE–Befehlen auf zwei oder mehrere Taktzyklen ausgedehnt werden. Dies kann entweder hardware– oder softwaremäßig erfolgen. Wenn im voraus bekannt ist, wie viele Taktzyklen zum Speicherzugriff erforderlich sind, kann das Problem softwaremäßig gelöst werden. Für dynamische Vezögerungen, z.B. infolge eines

Cache Miss (vgl. Kapitel 8) muss jedoch eine besondere Hardware vorhanden sein, um die Pipelineverarbeitung der nachfolgenden Befehle solange auszusetzen, bis der Laufzeitkonflikt behoben ist.

Statischer Laufzeitkonflikt

Betrachten wir den Befehl LOAD R_1,0(R_2) nach Abb. 6.6, der folgende Bedeutung hat: Register 1 soll mit dem Speicherinhalt geladen werden, der durch Register 2 adressiert wird. Am Ende der EX–Phase dieses Befehls wird R_2 (=0+R_2) als Speicheradresse ausgegeben. Das Datum für R_1 ist jedoch erst zwei Taktzyklen später gültig, d.h. die MEM–Phase umfaßt zwei Taktzyklen.[3] In Abb. 6.6 wird vorausgesetzt, dass der Registerblock durch einen Bypass umgangen werden kann. Ohne Bypass wären zwei weitere Taktzyklen nötig. Einer, um das gelesene Datum in das Register R_1 zu speichern (WB–Phase des LOAD–Befehls) und ein zweiter, um es wieder aus dem Registerblock auszulesen (ID–Phase des um vier NOP–Befehle verzögerten ADD–Befehls).

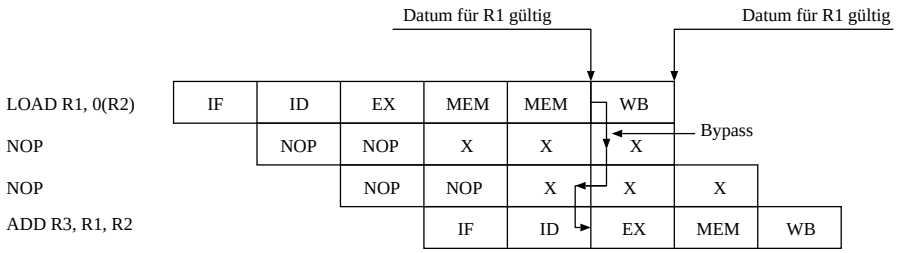


Abb. 6.6. Lesezugriff auf den Hauptspeicher, der zwei Taktzyklen erfordert. Der ADD–Befehl muss wegen der Datenabhängigkeit mit dem LOAD–Befehl ebenfalls um zwei Taktzyklen verzögert werden. Damit zwischen LOAD– und ADD–Befehl kein struktureller Pipelinekonflikt besteht, wird ein getrennter Daten– und Befehlsspeicher verwendet.

Um den Hardwareaufwand zu minimieren, setzen viele RISC–Prozessoren voraus, dass statische Laufzeitkonflikte bei LOAD–Befehlen softwaremäßig behoben werden. Ein Beispiel hierfür ist der MIPS–Prozessor, MIPS steht für Microprocessor without Interlocked Pipeline Stages. Ein LOAD–Befehl führt also immer zu einer Verzögerung bei der Bereitstellung der Daten. Man spricht daher von einem *delayed load* und bezeichnet den mit NOPs gefüllten Befehls„schlitz“ als *delay slot.*

[3] Man bezeichnet die Zeit nach Ende der EX–Phase bis zum Vorliegen eines gültigen Datums auch als *Latenzzeit*.

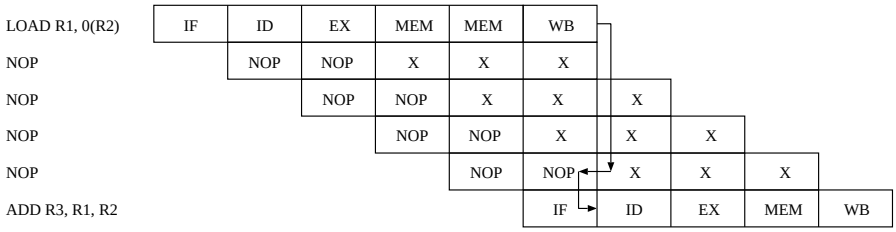


Abb. 6.7. Delayed Load aus Abb. 6.6, falls *kein* Bypass vorhanden wäre.

Dynamischer Laufzeitkonflikt

Sobald ein Befehl in die EX–Phase eintritt, ist seine Bearbeitung im Prozessor angestoßen. Sie kann nun nicht mehr gestoppt werden. Man bezeichnet diesen Vorgang als Befehlsausgabe (Intruction Issue). Während der ID–Phase muss daher geprüft werden, ob ein dynamischer Laufzeitkonflikt vorliegt. Im Fall eines Lesezugriffs bezeichnet man die zugehörige Hardware als LOAD Interlock. Sie überprüft den nachfolgenden Befehl auf eine Datenabhängigkeit mit dem LOAD–Befehl. Falls einer seiner Quelloperanden mit dem Zieloperanden des LOAD–Befehls übereinstimmt, muss die Pipelineverarbeitung für die Latenzzeit des Speicherzugriffs angehalten werden. Hierzu wird der Takt für die entsprechenden Pipelineregister (vgl. Abb. 6.8) ausgeblendet.

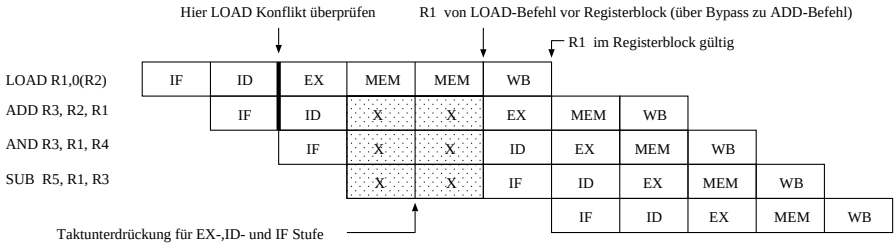


Abb. 6.8. Behebung eines dynamischen Laufzeitkonflikt mittels LOAD Interlock Hardware

6.3.4 Steuerflusskonflikte

Bei Verzweigungskonflikten muss entsprechend einer im Adressfeld angegebenen relativen Verschiebung zum Folgebefehl auf der Adresse $PC+4$ eine neue Zieladresse berechnet werden. Diese Berechnung geschieht —wie aus Abb. 6.2 ersichtlich— während der EX–Stufe mit Hilfe der ALU. Bei bedingten Verzweigungen wird gleichzeitig ein Registerinhalt auf Null getestet. Vor dem

Byte lang sein. Das Paket–Format (Abb. 7.33) besteht aus einem Header, den Nutzdaten und einer CRC–Prüfinformation. Im Header sind die Adressen von Sender– und Empfänger eingetragen. Da Pakete einander überholen können, wird jedem Paket eine Paketnummer zugeordnet. Ein IMP löscht ein weitergeleitetes Paket erst dann aus seinem Pufferspeicher, wenn der Nachfolgeknoten den fehlerfreien Empfang bestätigt hat. Im Fehlerfall, der bei ausbleibender Antwort durch den Ablauf eines Zeitzählers (Time out) erkannt wird, wird das Paket an einen anderen IMP übermittelt. Das ARPANET wurde unter Führung des amerikanischen Verteidigungsministerium DoD (Department of Defense) entwickelt und soll durch das DRI (Defense Research Internet) ersetzt werden.

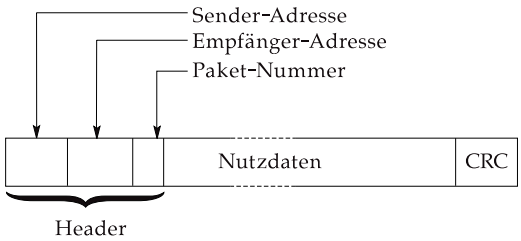
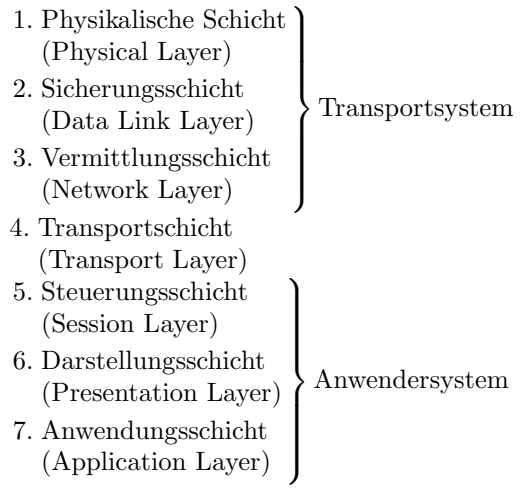


Abb. 7.33. Paketformat beim ARPANET

7.10 OSI–Modell

Das Open Systems Interconnection–Modell wurde von der ISO (International Organisation for Standardization) entwickelt. Es enthält ein hierarchisches Schichtenmodell mit sieben Schichten, die standardisierte Funktionen und Begriffe zur Kommunikation definieren. Ziel ist es, den Austausch von Daten zwischen Computern unterschiedlicher Hersteller zu ermöglichen. Daher wird das OSI–Modell vorwiegend im Zusammenhang mit WANs benötigt.

Das OSI–Modell stellt ein Rahmenwerk dar, um die zur Kommunikation unterschiedlicher Computersysteme benötigten Protokolle zu erarbeiten. Die Bedeutung des OSI–Modells liegt darin, dass es eine einheitliche Terminologie einführt und Schnittstellen identifiziert, für die standardisierte Protokolle entwickelt werden können. Die Spezifikation solcher Protokolle ist nicht im OSI–Modell enthalten. Sie ist Aufgabe anderer nationaler und internationaler Normungsgremien (z.B. ANSI = American National Standards Institute).
Das OSI–Modell besteht aus sieben Schichten:



Betrachten wir eine Punkt–zu–Punkt Verbindung zweier Computer (vgl. Abb. 7.34). In der Anwendungsschicht der beiden Maschinen laufen zwei Programme (Prozesse), die miteinander Nachrichten austauschen. Die Informationen von Programm A werden solange umgeformt, bis sie schließlich auf einem physikalisch existierenden Kanal übertragen werden können. Im Computer B wird die Reihenfolge der Umformungsschritte umgekehrt, so dass in allen Schichten im Computer B eine Informationsstruktur vorliegt, die zu den entsprechenden Schichten in Computer A passt. In jeder Schicht laufen Prozesse, die mit gleichgestellten Prozessen (*Peers*) in derselben Schicht kommunizieren. Diese Prozesse sind entweder durch Hardware oder durch Software realisiert. Da die Prozesse einer Schicht auf die Dienste benachbarter Schichten zurückgreifen, liegt eine vertikale Informationsübertragung vor. Ein realer (horizontaler) Datenaustausch erfolgt nur in der physikalischen Schicht. Aufgrund der Protokollhierarchie sind jedoch die einzelnen Schichten *logisch* miteinander gekoppelt. Sendeseitig erweitert jedes Schichtprotokoll die Datenpakete aus der nächst höheren Schicht durch Steuerinformationen, die für das entsprechende Schichtprotokoll auf der Empfängerseite bestimmt sind. Man kann sich vorstellen, dass jede Schicht ihre eigene Verpackung benutzt, die empfangsseitig von einer entsprechenden Schicht wieder entfernt wird. Zwei Schichten sind durch ein *Interface* miteinander verbunden. Mehrere Schichten können mit derselben Hardware bzw. durch ein Programm realisiert werden. Im folgenden werden die einzelnen Schichten kurz beschrieben.

Physikalische Schicht

Auf dieser Ebene werden die einzelnen Bits übertragen. Es muss festgelegt werden, wie eine 0 bzw. 1 repräsentiert werden soll und wie lang eine Bitzelle dauert. Das physikalische Medium zur Übertragung kann ein Kabel, ein Lichtwellenleiter oder eine Satellitenverbindung sein. Zur Spezifikation dieser

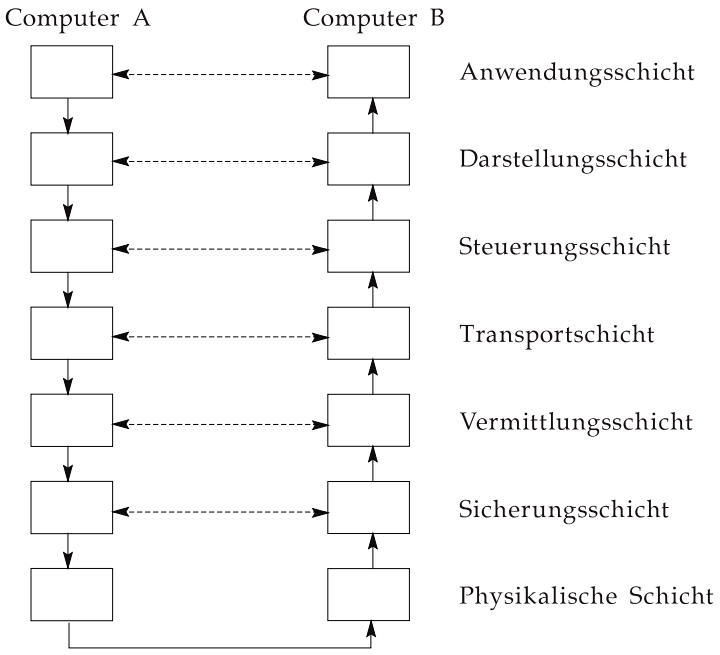


Abb. 7.34. Kommunikation zweier Computer im OSI–Modell

Schicht zählen z.B. auch der mechanische Aufbau und die Anschlussbelegung der Stecker.

Sicherungsschicht

Die Sicherungsschicht beseitigt Fehler, die auf der Übertragungsstrecke entstehen können. Dazu wird der Bitstrom in Übertragungsrahmen (Frames) zerlegt, denen man jeweils einen Prüfwert (meist nach dem CRC–Verfahren) zuordnet. Falls mit dem Prüfwert ein Übertragungsfehler entdeckt wird, sorgt das Protokoll der Sicherungsschicht für die Wiederholung des betreffenden Übertragungsrahmens. Im Allgemeinen wird nicht jeder einzelne Übertragungsrahmen durch den Empfänger bestätigt, sondern es wird eine vorgegebene Zahl von Rahmen gesendet.

Vermittlungsschicht

Die Aufgabe der Vermittlungsschicht ist es, in dem vorhandenen Netzwerk möglichst gute Wege zwischen den zwei Computern zu finden. Schlechtes Routing bewirkt Staus in den Netzwerkknoten. Außerdem werden die verfügbaren Kanäle nicht optimal ausgelastet. Routing ist nur dann erforderlich, wenn

Eine große Speicherkapazität, permanenter aber veränderlicher Speicherinhalt, niedrige Kosten pro Bit und geringe Zugriffszeiten sind Anforderungen, die nicht gleichzeitig mit einer *einzigen* Speicherart realisiert werden können. Trotzdem soll dem Benutzer ein scheinbar beliebig großer Speicher zur Verfügung stehen, ohne dass er die verschiedenen Speichermedien explizit kennen oder verwalten muss. Diese Aufgabe wird daher dem Betriebssystem übertragen, das i.a. durch geeignete Hardware unterstützt wird. Ein *virtueller* Speicher entsteht durch die Trennung von logischem und physikalischem Adressraum. *Eine* grundlegende Aufgabe von Speicherverwaltungseinheiten ist die schnelle Umsetzung dieser Adressen. Im letzten Teil dieses Kapitels werden wir Strukturierungselemente kennenlernen, um den virtuellen Adressraum auf eine (physikalisch vorhandene) Festplatte abzubilden. Da die Verwaltung von Caches sehr ähnlich organisiert ist, werden wir dabei auch auf Caches eingehen.

8.1 Halbleiterspeicher

Speicher können nach der verwendeten Zugriffsart eingeteilt werden. Wir unterscheiden Speicher mit *wahlfreiem, seriellem (zyklischem)* oder *inhaltsbezogenem* Zugriff (Abb. 8.2). Wahlfreier Zugriff (random Access) erfolgt unter Angabe einer Adresse, die genau einem Speicherplatz zugeordnet wird. Auf die einzelnen Speicherplätze kann in beliebiger Adressfolge zugegriffen werden. Beim seriellen oder zyklischen Zugriff ist die Adressfolge der Speicherworte fest vorgegeben, d.h. es können keine Speicherplätze übersprungen werden. Da die Speicherworte hintereinander abgelegt werden, müssen beim Zugriff auf nicht benachbarte Speicherplätze Verzögerungen in Kauf genommen werden. Halbleiterspeicher mit seriellem bzw. zyklischem Zugriff sind FIFOs, LIFOs und CCDs (Charge Coupled Devices). Beim inhaltsbezogenem Zugriff (Content Addressable Memory CAM) wird statt einer Adresse ein Suchwort benutzt, um Daten zu adressieren, die zusammen mit dem Suchwort eingespeichert wurden.

Hochintegrierte Halbleiterspeicher haben meist wahlfreien Zugriff. Sie können weiter unterteilt werden in *flüchtige* und *nichtflüchtige* Speicher. Nach dem Abschalten der Stromversorgung verlieren flüchtige Speicher ihren Inhalt. Zu den nichtflüchtigen Speichern zählen ROM (Read Only Memory), PROM (Programmable ROM), EPROM (Erasable PROM) und EEPROM (Electrically EPROM).

ROM und PROM können nicht gelöscht werden. Schreib/Lesespeicher werden durch die wenig treffende Abkürzung RAM für Random Access Memory bezeichnet. Wie aus Abb. 8.2 zu erkennen ist, könnten damit auch ROMs gemeint sein. Statische RAMs basieren auf bistabilen Kippstufen. Dynamische RAMs speichern die Information als Ladungspakete auf Kondensatoren.

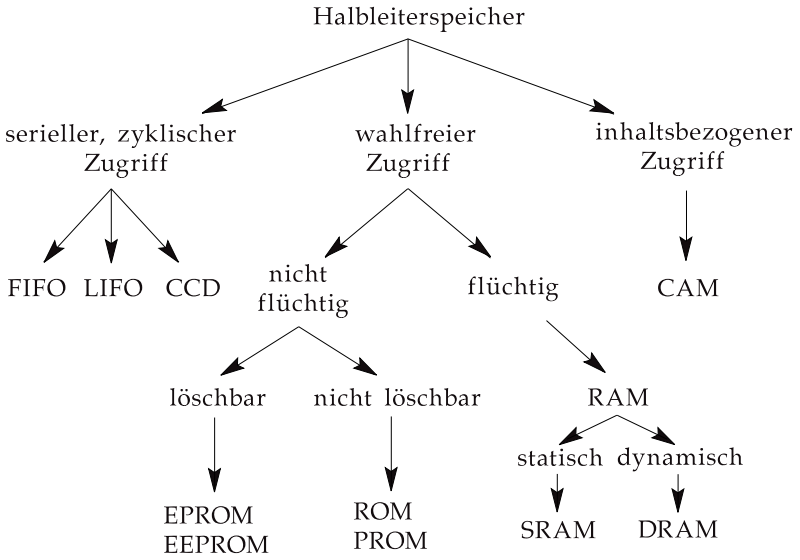


Abb. 8.2. Übersicht über Halbleiterspeicher

8.1.1 Speicher mit wahlfreiem Zugriff

Register bestehen aus Flipflops, die durch einen gemeinsamen Takt gesteuert werden — wir haben sie bereits weiter oben kennengelernt. Sie werden benötigt, um ein Datum kurzzeitig zu speichern. Man findet sie im Rechen– und Leitwerk eines Prozessors. Da sie direkt durch die Ablaufsteuerung angesprochen werden, ist die Zugriffszeit sehr gering. Halbleiterspeicher mit höherer Speicherkapazität müssen für den Zugriff auf die gespeicherten Informationen über eine geeignete Speicherorganisation verfügen.

Wir wollen im folgenden untersuchen, wie RAM– und ROM–Speicher organisiert sind, d.h. wie man die Speicherzellen anordnet und ihren Inhalt liest oder verändert. Eine Speicherzelle nimmt die kleinste Informationseinheit (1 Bit) auf. Je nach Herstellungstechnologie benutzt man unterschiedliche Speicherprinzipien, die in späteren Abschnitten erläutert werden. Allen gemeinsam ist die matrixförmige Anordnung der Speicherzellen, da hiermit die verfügbare Chipfläche am besten genutzt

Bitweise organisierte Halbleiterspeicher (Abb. 8.3) adressieren immer nur eine einzige Speicherzelle. Die Adressleitungen werden in zwei Teile aufgespalten, die man dann als *Zeilen*– und *Spalten*adresse benutzt. Die decodierten Zeilen– und Spaltenadressen dienen zur Auswahlsteuerung für die Speichermatrix. Jeder möglichen Adresse wird genau ein Kreuzungspunkt der decodierten Zeilen– und Spaltenauswahl–Leitungen zugeordnet. Sind beide auf 1–Pegel, so ist die zugehörige Speicherzelle aktiviert. Die Schreib–/Lesesteuerung, die

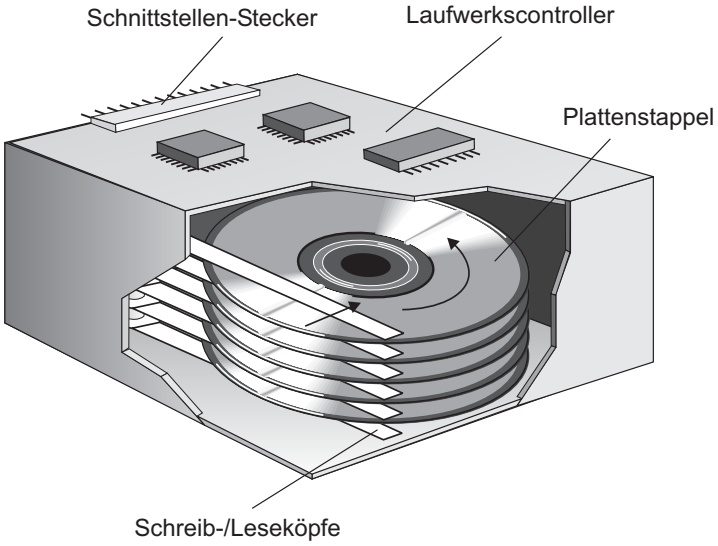


Abb. 8.21. Aufbau eines Festplattenlaufwerks.

Schreib–/Lesekopf sehr schnell geht, muss dieser erst einmal auf die angesteuerte Spur „einrasten“, indem er die Low–Level–Formatdaten ausliest und auswertet. Erst danach können Daten von dem angeforderten Sektor gelesen bzw. geschrieben werden.

Die *mittlere Datenzugriffszeit* einer Festplatte gibt an, wie lange es zwischen der Anforderung und der Ausgabe eines Sektors dauert. Diese Zeit wird durch die Positionierzeit, die Latenzzeit, die Einstellzeit und durch die Verarbeitungsgeschwindigkeit des Laufwerkcontrollers bestimmt. Eine geringe mittlere Zugriffszeit ist wichtig, wenn man viele kleinere Dateien verarbeitet. Bei Multimediaanwendungen kommt es dagegen vorwiegend auf hohe Datenraten an. Hier versucht man, die Daten möglichst zusammenhängend auf der Festplatte anzuordnen und durch eine hohe Rotationsgeschwindigkeit die Datentransferrate zu maximieren. Die mittlere Datenzugriffszeit spielt dann eher eine untergeordnete Rolle.

Die *Mediumtransferrate* gibt an, wie viele Daten pro Sekunde von oder zur Speicherschicht der Festplatte übertragen werden können. Sie wird in MByte pro Sekunde angegeben und hängt von folgenden Parametern ab:

Bitdichte gemessen in Bit per Inch (bpi). Diese Größe hängt von den physikalischen Eigenschaften des Schreib–/Lesekopfes und des verwendeten Speichermaterials ab. Um eine hohe Speicherdichte zu erreichen, sollte der Abstand zwischen Schreib–/Lesekopf und Speichermaterial möglichst klein sein.

Benutzte Codierung (MFM, RLL), die festlegt, wie viele Bitzellen im Mittel für die Speicherung eines Bits benötigt werden.

Lage der Spur, auf der der Sektor liegt. Bei konstanter Winkelgeschwindigkeit und konstanter Größe der Bitzellen ändert sich die Rate der Flusswechsel in Abhängigkeit von der jeweiligen Spur. Auf den inneren Spuren ist die Bahngeschwindigkeit und damit die Flusswechselrate geringer als auf den äußeren Spuren. Somit ergeben sich auch unterschiedliche Transferraten. Auf der innersten Spur ist die Datentransferrate am kleinsten, auf der äußersten Spur ist sie am größten.

Neben der *Mediumtransferrate* gibt es die *Datentransferrate*, die angibt, wie schnell Daten zwischen Hauptspeicher und Festplattencontroller übertragen werden können. Diese Kenngröße ist für die Praxis wichtiger als die Mediumtransferrate. Man beachte, dass die Datentransferrate einer Festplatte meist deutlich unter der Bandbreite des aktuellen Schnittstellenstandards liegt. So lag z.B. Ende 2001 die Datentransferrate von Festplatten bei ca. 7 MByte/s. Mit damaligen Schnittstellenstandards waren dagegen Datenraten von bis zu 20 MByte/s möglich.

Zum Abschluss soll noch auf die Größeneinheiten bei der Angabe der Speicherkapazität hingewiesen werden. Da alle Computer im Dualsystem rechnen, ist es sinnvoll, Größenangaben auch auf dieses Zahlensystem zu beziehen (Tabelle 8.1). Die Tatsache, dass die Größeneinheiten des Dezimalsystems deutlich geringeren Werten entsprechen, nutzen viele Hersteller bzw. Händler, um ihre Festplatten mit größeren Speicherkapazitäten anzupreisen, als diese tatsächlich besitzen. So hat z.B. eine Festplatte, die bezogen auf das Dezimalsystem eine Speicherkapazität von 160 GByte hat, in Wirklichkeit nur eine Speicherkapazität von 149 GByte.

Tabelle 8.1. Vergleich von Größeneinheiten im Dual– und Dezimalsystem.

Zeichen	Name	Wert Dual	Wert Dezimal
K	Kilo	2^{10}=1.024	1.000
M	Mega	2^{20}=1.048.576	1.000.000
G	Giga	2^{30}=1.073.741.824	1.000.000.000
T	Tera	2^{40}=1.099.511.627.766	1.000.000.000.000
P	Peta	2^{50}=1.125.899.906.842.624	1.000.000.000.000.000

8.4 Softsektorierung

Nachdem wir in Abschnitt 8.2 gesehen haben, wie man einzelne Bitmuster mehr oder weniger kompakt in ein Magnetisierungsmuster codieren kann,

wollen wir nun die Frage stellen, wie größere Datenmengen auf einer Festplatte organisiert werden. Die heute übliche Organisationseinheit ist eine *Datei*, die über einen Namen angesprochen werden kann. Eine Datei setzt sich aus Speicherblöcken zusammen, die auf der Festplatte als Sektoren abgelegt werden.

Die Oberfläche einer einzelnen Festplattenscheibe (Platter) wird in eine Menge konzentrischer Spuren aufgeteilt (Abbildung 8.22). Die Sektoren sind Kreisabschnitte auf einer Spur und bilden die kleinsten zugreifbaren Einheiten. Sie nehmen üblicherweise Datenblöcke von 512 oder 1024 Bytes auf.

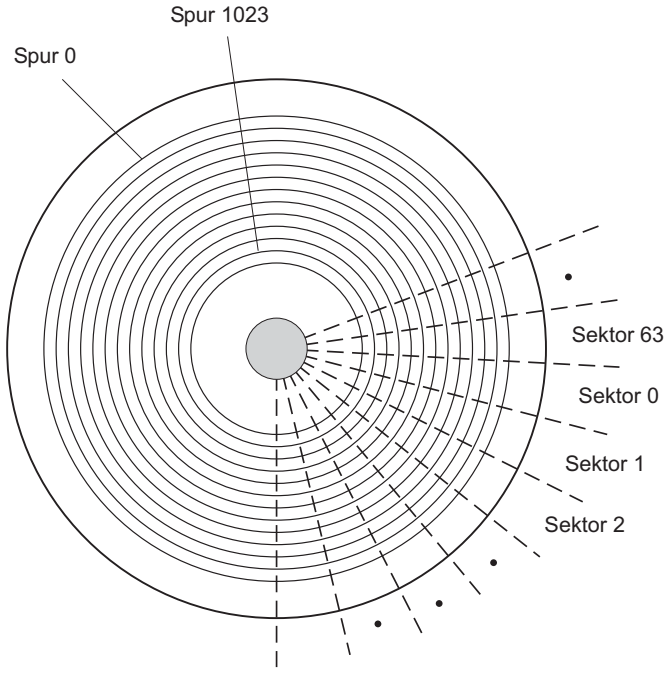


Abb. 8.22. Aufteilung der Plattenoberfläche.

Die Einteilung des Datenträgers in Spuren und Sektoren wird Softsektorierung oder auch *Low–Level–Formatierung* genannt. Früher benutzte man auch die Hardsektorierung, bei der in die Plattenoberfläche ein oder mehrere Indexlöcher gestanzt wurden, um die Sektorgrenzen zu markieren. Bei der Softsektorierung wird der Anfang eines Sektors durch bestimmte Magnetisierungsmuster markiert. Daran schließt sich dann ein Header an, in den die Sektornummer und weitere Verwaltungsinformationen (z.B. CRC–Prüfsummen zur Fehlererkennung) eingetragen werden. Dann folgen die eigentlichen Nutzdaten und die Prüfbits zur Fehlerkorrektur, die als ECC–Bits (Error Checking and Correcting) bezeichnet werden (Abbildung 8.23).

die Cluster unbelegt sind und dass sie zum Erzeugen neuer Dateien verwendet werden können. Die Sektoren auf der Festplatte bleiben beim Löschen einer Datei zunächst unverändert. Nur die Einträge in der Clusterkette werden also gelöscht. Es ist daher unter Umständen möglich, eine versehentlich oder „voreilig“ gelöschte Datei (oder ein gelöschtes Verzeichnis) wiederherzustellen. Ein weiterer Vorteil dieser Vorgehensweise beim Löschen von Dateien besteht darin, dass nur ein Bruchteil der Einträge auf der Festplatte gelöscht werden muss und dass dadurch der Löschvorgang beschleunigt wird.

Erzeugen einer Datei

Zum Erzeugen einer Datei wird zunächst im momentanen Arbeitsverzeichnis ein Verzeichniseintrag vorgenommen. Dann wird in der FAT ein unbelegter Cluster gesucht, dessen Index als Startcluster eingetragen wird. Nun werden entsprechend der gewünschten Dateilänge weitere unbelegte Cluster gesucht und in die Clusterkette aufgenommen. Schließlich wird unter dem Index des letzten Clusters der Wert für das Dateiende eingetragen (z.B. FFFF bei der FAT–16).

8.7.3 LINUX–Dateisystem

Mit jeder Datei bzw. mit jedem Verzeichnis assoziiert LINUX einen 64 Byte großen Datenblock, der zu dessen Verwaltung dient. Dieser Block heißt „Inode“. Die Inodes einer Festplatte befinden sich am Anfang einer Partition. Bei 1024 Byte großen Sektoren passen 16 Inodes in einen logischen Speicherblock (LBA). Anhand der Inode–Nummer kann LINUX den Speicherort finden, indem es die Inode–Nummer einfach durch 16 teilt.

Ein Verzeichniseintrag besteht aus einem Dateinamen und der zugehörigen Inode–Nummer. Beim Öffnen einer Datei sucht LINUX im angegebenen Verzeichnis nach dem Dateinamen und liefert die Inode–Nummer zurück. Nun wird der zugehörige Inode von der Festplatte in den Hauptspeicher gelesen. Mit den im Inode gespeicherten Informationen kann auf die Datei zugegriffen werden.

Man hat beim LINUX–Betriebssystem die Wahl zwischen mehreren Dateisystem–Varianten. Obwohl das Format eines Inodes vom jeweils ausgewählten Dateisystem abhängt, sind die folgenden Informationen stets vorhanden (vgl. Abbildung 8.28):

1. Dateityp und Zugriffsrechte,
2. Eigentümer der Datei,
3. Gruppenzugehörigkeit des Eigentümers,
4. Anzahl von Verweisen (Links) auf die Datei,
5. Größe der Datei in Bytes,

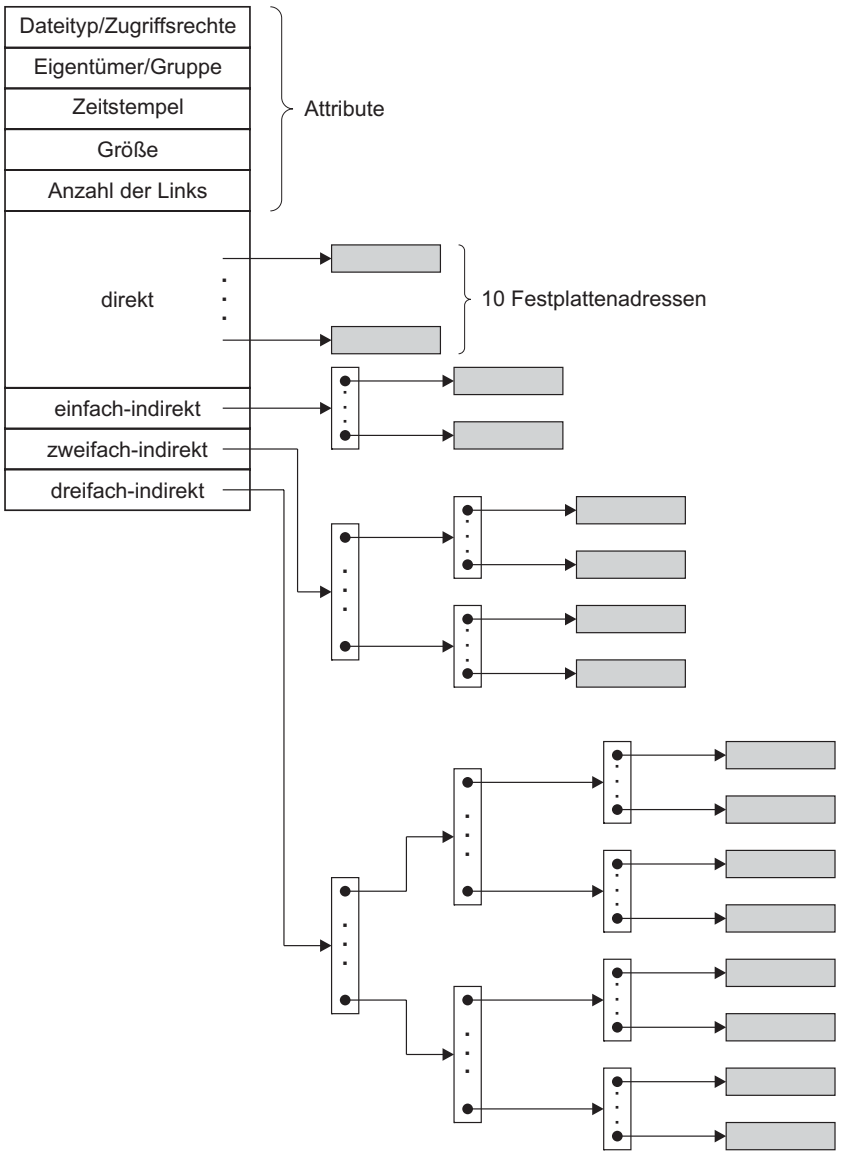


Abb. 8.28. Aufbau eines Inode.

6. 13 Festplattenadressen (LBAs),
7. Uhrzeit des letzten Lesezugriffs,
8. Uhrzeit des letzten Schreibzugriffs,
9. Uhrzeit der letzten Inode–Änderung.

Dateien und Verzeichnisse werden anhand des Dateityps unterschieden. Da LINUX alle Ein–/Ausgabegeräte als Dateien betrachtet, wird durch den Dateityp auch angezeigt, ob es sich um ein unstrukturiertes oder blockorientiertes Ein–/Ausgabegerät handelt.

Wichtig für die physische Abbildung der Datei auf die Festplatte sind die unter Punkt 6 aufgeführten Einträge für Festplattenadressen (LBAs). LINUX verwaltet die Festplattenadressen **nicht** wie die FAT durch Verkettung, sondern mittels Indexierung der Festplattenadressen einzelner Speicherblöcke.

Mit den ersten zehn Einträgen können bei einer Blockgröße von 1024 Byte Daten mit bis zu 10.240 Byte verwaltet werden. Wegen der direkten Indexierung ist – gegenüber dem FAT–Dateisystem – ein wahlfreier Zugriff möglich. So muss man beispielsweise zum Lesen des letzten Speicherblocks nicht sämtliche Vorgängerindizes durchlaufen.

Ist die Datei größer als 10.240 Byte, so geht man zur einfach indirekten Adressierung über. Die 11. Festplattenadresse verweist auf einen indirekten Block, der zunächst von der Festplatte gelesen werden muss und der auf weitere 256 (1024/4) Festplattenadressen verweist. Hier wird angenommen, dass eine Festplattenadresse 32 Bit lang ist. D.h. es können 2^{32} Speicherblöcke angesprochen werden. Bei 1024 Byte pro Speicherblock entspricht die Verwendung von 32–Bit–Werten für LBAs einer maximalen Speicherkapazität der Festplatte von 2^{42}= 4 TByte.

Die maximale Dateigröße bei einer Beschränkung auf 11 Festplattenadressen beträgt somit

$$(10.240 + 256 \cdot 1024)\ \text{Byte} = 272.384\ \text{Byte.} \approx 266\ \text{KByte}$$

Die 12. Festplattenadresse zeigt auf einen Block, der doppelt indirekt auf weitere Speicherblöcke zeigt. Wie beim einfach indirekt adressierten Block wird dieser Block zunächst von der Festplatte gelesen und in den Hauptspeicher gebracht. Jede der 256 in diesem Block stehenden 32 Bit (4 Byte) Werte wird wiederum als Speicherblockadresse auf einen Block mit weiteren 256 Festplattenadressen interpretiert. Somit können mit Hilfe der ersten 12 Festplattenadressen des Inodes insgesamt

$$(272.384 + 256 \cdot 256 \cdot 1024)\ \text{Byte} = 67.381.248\ \text{Byte.} \approx 64,26\ \text{MByte}$$

große Dateien verwaltet werden.

Falls die zu verarbeitende Datei noch größer sein sollte, nimmt man noch die 13. Festplattenadresse hinzu. Hiermit ist dann eine dreifach indirekte Adressierung möglich. Analog zu der obigen Darstellung können nun Dateien bis zu einer Größe von

$$(67.381.248 + 256 \cdot 256 \cdot 256 \cdot 1024)\ \text{Byte} = 17.247.250.432\ \text{Byte}$$

$$\approx 16,06\ \text{GByte}$$

Verfeinerung und benötigt statt des Zählers in Abb. 9.11 ein durch den Komparatorausgang gesteuertes Schaltwerk. Die Zahl der Schritte ist gleich der Bitzahl des A/D–Umsetzers. Im ersten Schritt erfolgt eine grobe Schätzung des Digitalwerts. Durch Setzen des höherwertigen Bits wird geprüft, ob die unbekannte Spannung U_x in der oberen oder unteren Hälfte des Eingangsspannungsbereichs liegt. Wenn der Komparatorausgang 0 ist, liegt U_X in der oberem Hälfte des Eingangsspannungsbereichs und das Bit bleibt gesetzt. Sonst wird es wieder zurückgesetzt. Auf die gleiche Art und Weise wird in jedem nachfolgenden Schritt der verbleibende Eingangsspannungsbereich halbiert und ein weiteres Bit bestimmt. Der Wert der unbekannten Spannung wird mehr und mehr eingeschränkt und liegt, nachdem das niederwertigste Bit erreicht ist, bis auf den Quantisierungsfehler genau im Ausgaberegister. Die typischen Umsetzzeiten, die mit der Methode der sukzessiven Approximation erreicht werden, liegen bei 12–Bit Auflösung in der Größenordnung von $2 \ldots 50$ μs.

Nachdem wir nun in die Grundlagen der Ein-/Ausgabe eingeführt haben, werden wir in den nächsten Abschnitten die Funktionsprinzipien ausgewählter Peripheriegeräte vorstellen.

9.5 Tastatur

Die Tastatur ist wohl das wichtigste Eingabegerät eines PCs. Sie wird über die so genannte PS/2– oder USB–Schnittstelle angeschlossen. Es gibt auch drahtlose Tastaturen, bei denen nur ein Funkmodul mit der Schnittstelle verbunden wird.

Tastaturen gibt es mit zeilenförmiger oder ergonomischer Anordnung der Tasten. Am häufigsten findet man jedoch zeilenförmig angeordnete Tasten, die wegen der Reihenfolge der ersten Buchstaben (von links oben nach rechts) auch als QWERTZ–Tastaturen bezeichnet werden. Dieses Layout der Tasten ist vor allem in Europa verbreitet. In Amerika findet man das QWERTY–Layout, bei dem im Wesentlichen die Z– und die Y–Taste vertauscht sowie einige Sonderzeichen anders platziert sind. Neben den Buchstaben– und Zifferntasten gibt es Funktionstasten (oberste Zeile), Steuertasten (Strg, Alt, Shift, AltGr), Cursortasten und oft einen numerischen Tastenblock. Das bei uns übliche Tastaturlayout MF II (Multi–Funktion II) hat 102 Tasten.

Wenn man eine Taste drückt, kommt es zu einem kurzzeitigen Schließen eines Kontakts, der sich im Kreuzungspunkt einer Matrix befindet, die aus Zeilen– und Spaltenleitungen besteht. Jede Taste kann über ihre Zeilen– und Spaltennummer eindeutig zugeordnet werden.

Auf der Tastaturplatine befindet sich ein Mikrocontroller (meist ein 8049), der die Leitungsmatrix so schnell abtastet, dass jeder einzelne Tastendruck getrennt registriert werden kann. Hierzu legt der Controller zyklisch 1–Signale auf die Zeilenleitungen. Wenn in einer angewählten Zeile eine Taste gedrückt

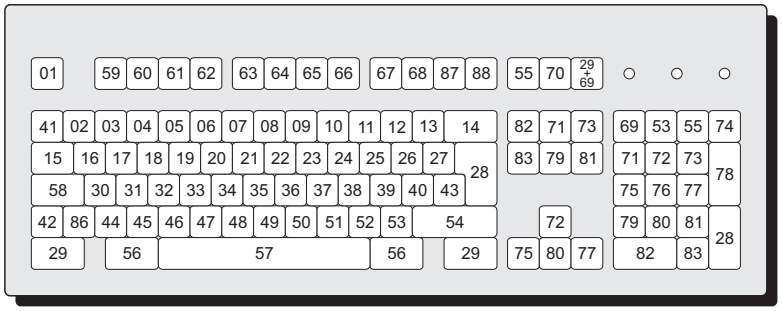


Abb. 9.12. Tastenlayout und Scancodes einer MF II–Tastatur

wird, kann der Controller die Dualcodes der Zeilen– und Spaltennummern der gedrückten Taste ermitteln. Eigentlich wird der Kontakt nicht nur einmal, sonderen mehrere Male hintereinander geschlossen. Man bezeichnet dies als *Tastenprellen*. Der Tastaturcontroller kann aber aufgrund der kurzen zeitlichen Abstände das Tastenprellen von aufeinanderfolgenden Tastendrücken unterscheiden. Früher benutzte man zum Zweck der Entprellung von Tastern elektronische Schaltungen.

Wegen der unterschiedlichen Größe und der von einer regelmäßigen Matrix abweichenden Anordnung der Tasten können aus dem Tastaturlayout die Spalten– und Zeilennummern nicht eindeutig zugeordnet werden. Aus der Abbildung 9.12 geht jedoch hervor, welcher *Scancode* beim Drücken einer Taste erzeugt wird. Dieser Scancode setzt sich wie oben beschrieben aus dem Dualcode der Spalten– und Zeilennummer zusammen, an der sich die gedrückte Taste in der Leitungsmatrix befindet. Er wird zunächst in einem FIFO–Pufferspeicher zwischengespeichert und dann vom Tastatur–Controller zur entsprechenden Schnittstelle (PS/2 oder USB) an den PC übermittelt. Dort befindet sich ein Schnittstellen–Controller (meist bereits im Chipsatz enthalten), der den seriellen Datenstrom wieder in eine Folge von (parallelen) Scancodes umwandelt (Abbildung 9.13). Die bitserielle Datenübertragung erfolgt synchron, d.h. neben dem Datensignal wird auch eine Taktleitung benötigt.

Sobald der Schnittstellen–Controller Daten vom Tastatur–Controller empfangen hat, erzeugt er einen Interrupt über die IRQ_1–Leitung. Damit informiert er den Prozessor, dass der Benutzer Daten eingegeben hat. Die Scancodes werden nun mit Hilfe des BIOS–Interrupthandlers (INT 9) vom Prozessor in ASCII–Zeichencodes umgewandelt und zusammen mit den Scancodes in einem Ringpuffer zwischengespeichert.

Zur Übertragung der Bilddaten wird entweder die Kamera über eine USB– oder FireWire–Schnittstelle mit dem PC verbunden oder man entnimmt die Speicherkarte aus der Kamera. Mit Hilfe eines Flash–Card–Lesegeräts, das meist mehrere verschiedene Kartenformate unterstützt, kann dann der Speicherinhalt ausgelesen werden. Die Lesegeräte werden meist über die USB–Schnittstelle angeschlossen. Außerdem bieten viele moderne Farb– und Fotodrucker die Möglichkeit, Bilder von Speicherkarten direkt auszudrucken.

9.8.2 Video– und Webkameras

Um bewegte Bilder aufzunehmen, verwendet man eine Video– oder Webkamera.[11] Digitale Videokameras enthalten spezielle Prozessoren, die eine Folge von Einzelbildern in Echtzeit in ein standardisiertes Videoformat umsetzen. Hierzu wird heute meist das *MPEG*–2 oder *MPEG*–4 Format verwendet, das auch bei DVD–Playern gebräuchlich ist. Durch die MPEG–Kompression werden die Datenrate und das Speichervolumen gegenüber einer Einzelbildfolge mit 25 Bildern pro Sekunde drastisch reduziert. Zur Wiedergabe müssen die komprimierten Daten dann wieder rechenaufwendig dekodiert werden. Um den PC–Prozessor hiervon zu entlasten, integriert man bei vielen Grafik– bzw. TV–Karten spezielle MPEG–Dekoder. Damit kann man dann auch bei weniger leistungsfähigen PCs MPEG–Videos „ruckfrei“ betrachten.

9.9 LCD–Bildschirm

In den letzten Jahren haben Flachbildschirme auf Basis von Flüssigkeitskristallen (Liquid Crystal Display, LCD) die klassischen Bildschirme mit Elektronenstrahlröhren (Cathode Ray Tube, CRT) fast völlig verdrängt. Obwohl CRTs wegen ihrer hohen Reaktionsgeschwindigkeit geschätzt werden[12], haben sie gegenüber LCD-Bildschirmen entscheidende Nachteile. CRT-Bildschirme sind nicht nur sehr sperrig und schwer, sondern sie haben auch deutlich schlechtere Darstellungseigenschaften. So treten selbst bei hochwertigen CRTs Randverzerrungen und Verzerrungen der Bildobjekte aufgrund von Konvergenzfehlern auf. Darüber hinaus ist bis heute nicht eindeutig geklärt, ob die von ihnen ausgehende elektromagnetische Strahlung gesundheitliche Schäden hervorrufen kann.

LCDs basieren auf den optischen Eigenschaften von Flüssigkeitskristallen, die aus durchsichtigen organischen Molekülen bestehen. Die stabförmigen Moleküle liegen als zähflüssige (viskose) Flüssigkeit vor und haben die Eigenschaft, Lichtwellen zu polarisieren. Dies bedeutet, dass die Schwingungsebene von Lichtwellen, die durch den Flüssigkeitskristall hindurchgehen, sich an der Orientierung der stabförmigen Moleküle ausrichtet.

[11] Letztere wird oft als Webcam bezeichnet.
[12] vor allem für Computerspiele wichtig.

In den 60ger Jahren entdeckte man, dass mit einem elektrischen Feld die Orientierung der Moleküle beeinflusst werden kann. Damit hat man die Möglichkeit, die *Polarisationsrichtung* des durch den Flüssigkeitskristall hindurchgehenden Lichts elektrisch umzuschalten. In Abbildung 9.18 ist der schematische Aufbau eines Pixels auf einem LCD–Bildschirm dargestellt.

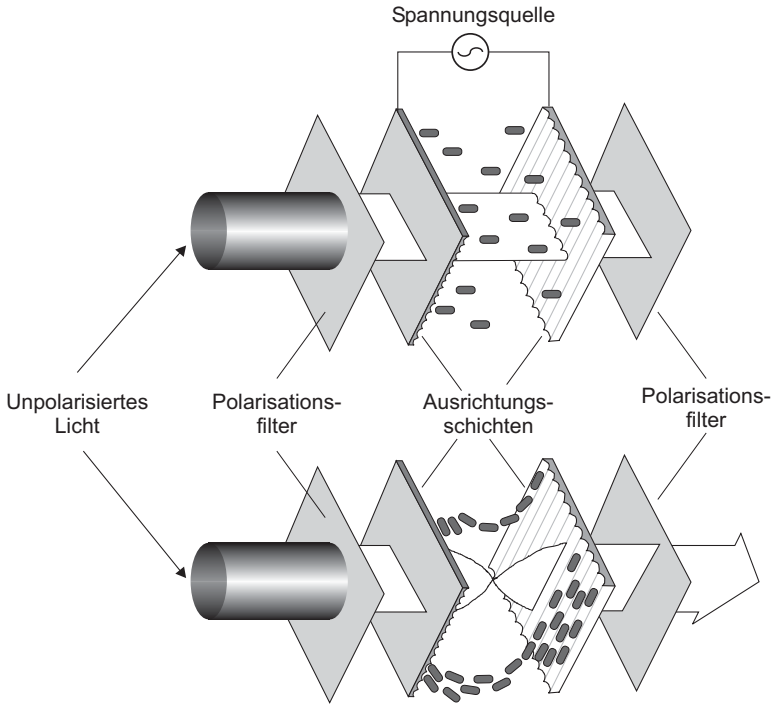


Abb. 9.18. Funktionsprinzip eines LCD–Bildschirms

Betrachten wir zunächst den unteren Teil der Abbildung. Von links kommt unpolarisiertes Licht aus einer Lichtquelle, die den gesamten Bildschirm eines LCD–Display von hinten gleichmäßig ausleuchtet. Die Hintergrundbeleuchtung wird durch Leuchtstoffröhren realisiert, die an der Seite des Displays angebracht sind. Die horizontale Schwingungsebene der Hintergrundbeleuchtung wird mit Hilfe einer Polarisationsfolie herausgefiltert. Die *Polarisationsfolie* deckt die gesamte Displayfläche ab, d.h. alle Pixel werden von hinten mit horizontal polarisiertem Licht angestrahlt.

Der eigentliche Flüssigkeitskristall befindet sich zwischen zwei durchsichtigen Folien (oder Glasplatten), in die winzige horizontale und vertikale Rillen eingeritzt sind. Diese Ausrichtungsschichten (alignment layer) sind um 90° gegeneinander verdreht und der Zwischenraum wird vom Flüssigkeitskristall

10. Aktuelle Computersysteme

In diesem Kapitel soll ein kurzer Überblick über aktuelle Computersysteme gegeben werden. Zunächst stellen wir die verschiedenen Arten von Computern vor. Dann betrachten wir am Beispiel von Desktop–Systemen deren internen Aufbau, der vor allem durch den Chipsatz geprägt wird. Dann werden die aktuellsten Desktop–Prozessoren der beiden führenden Hersteller AMD und Intel vorgestellt und miteinander verglichen. Danach beschreiben wir die Funktionsprinzipien der aktuellen Speichermodule sowie Ein– und Ausgabeschnittstellen. Schließlich gehen wir auch auf die Bedeutung von Grafikadaptern ein und geben einen Ausblick auf die künftige Entwicklung.

Die Entwicklung neuer Prozessorarchitekturen und Computersysteme ist rasant. Die Chiphersteller vermelden fast täglich neue technologische und architektonische Verbesserungen ihrer Produkte. Daher fällt es natürlich auch schwer, einen aktuellen Schnappschuss der Entwicklung wiederzugeben – zumal dieser dann nach kurzer Zeit wieder veraltet ist. Trotzdem wollen wir im Folgenden versuchen, den Stand im Winter 2004/05 zu erfassen.

10.1 Arten von Computern

Obwohl es uns meist nicht bewusst ist, sind wir heutzutage von einer Vielzahl verschiedenster Computersystemen umgeben. Die meisten Computer, die wir täglich nutzen, sind nämlich in Gebrauchsgegenständen eingebaut und führen dort Spezialaufgaben aus. So bietet uns beispielsweise ein modernes Handy die Möglichkeit, Telefonnummern zu verwalten, elektronische Nachrichten (SMS) zu versenden, Musik abzuspielen oder sogar Bilder aufzunehmen. Ähnliche Spezialcomputer findet man in Geräten der Unterhaltungselektronik (z.B. CD–, DVD–, Video–Recordern, Satelliten–Empfängern), Haushaltstechnik (z.B. Wasch– und Spülmaschinen, Trockner, Mikrowelle), Kommunikationstechnik (z.B. Telefon– und FAX–Geräte) und auch immer mehr in der KFZ–Technik (z.B. intelligentes Motormanagement, Antiblockier– und Stabilisierungssysteme). Diese *Spezialcomputer* oder so genannten *embedded systems* werden als Bestandteile größerer Systeme kaum als Computer wahrgenommen. Sie müssen jedoch ein weites Leistungsspektrum abdecken und insbesondere bei Audio– und Videoanwendungen bei minimalem Energiebedarf Supercomputer–Leistungen erbringen.

Solche Systeme basieren meist auf Prozessoren, die für bestimmte Aufgaben optimiert wurden (z.B. Signal– oder Netzwerkprozessoren). Aufgrund der immensen Fortschritte der Mikroelektronik ist es sogar möglich, Prozessorkerne zusammen mit zusätzlich benötigten digitalen Schaltelementen auf einem einzigen Chip zu realisieren (SoC, System on a Chip).

Neben diesen eingebetteten Systemen gibt es auch die so genannten *Universalcomputer*. Gemeinsames Kennzeichen dieser Computersysteme ist, dass sie ein breites Spektrum von Funktionen bereitstellen, die durch dynamisches Laden entsprechender Programme implementiert werden. Neben Standardprogrammen für Büroanwendungen (z.B. Schreib– und Kalkulationsprogramme) gibt es für jede nur erdenkliche Anwendung geeignete Software, die den Universalcomputer in ein anwendungsspezifisches Werkzeug verwandelt (z.B. Entwurfs– und Konstruktionsprogramme, Reiseplaner, Simulatoren, usw.).

Derartige Universalcomputer unterscheiden sich hinsichtlich der Größe und Leistungsfähigkeit. Die kleinsten und leistungsschwächsten Universalcomputer sind kompakte und leichte Taschencomputer (handheld computer), die auch als so genannte *PDA*s (Personal Digital Assistant) bekannt sind. Sie verfügen über einen nichtflüchtigen Speicher (vgl. Kapitel 8), der auch im stromlosen Zustand die gespeicherten Informationen beibehält. PDAs können mit einem Stift über einen kleinen berührungsempfindlichen Bildschirm (touchscreen) bedient werden und sind sogar in der Lage, handschriftliche Eingaben zu verarbeiten.

Notebooks sind ebenfalls portable Computer. Sie haben im Vergleich zu PDAs größere Bildschirme, eine richtige Tastatur und ein Sensorfeld, das als Zeigeinstrument (Maus–Ersatz) dient. Sie verfügen auch über deutlich größere Speicherkapzitäten (sowohl bzgl. Haupt– als auch Festplattenspeicher) und werden immer häufiger als Alternative zu ortsfesten *Desktop–Computern* verwendet, da sie diesen insbesondere bei Büro– und Kommunikationsanwendungen ebenbürtig sind. Um eine möglichst lange vom Stromnetz unabhängige Betriebsdauer zu erreichen, werden in Notebooks stromsparende Prozessoren eingesetzt.

Desktop–Computer oder *PC*s (Personal Computer) sind Notebooks vor allem bzgl. der Rechen– und Grafikleistung überlegen. Neben den typischen Büroanwendungen werden sie zum rechnergestützten Entwurf (CAD, Computer Aided Design), für Simulationen oder auch für Computerspiele eingesetzt. Die dazu verwendeten Prozessoren und Grafikadapter produzieren hohe Wärmeleistungen (jeweils in der Größenordnung von ca. 100 Watt), die durch große Kühlkörper und Lüfter abgeführt werden müssen.

Weitere ortsfeste Computersysteme sind die so genannten *Server*. Im Gegensatz zu den Desktops sind sie nicht einem einzelnen Benutzer zugeordnet. Da sie Dienstleistungen für viele über ein Netzwerk angekoppelte Desktops oder Notebooks liefern, verfügen sie über eine sehr hohe Rechenleistung (compute server), große fehlertolerierende und schnell zugreifbare Festplattensy-

steme[1] (file server, video stream server), einen oder mehrere Hochleistungsdrucker (print server) oder mehrere schnelle Netzwerkverbindungen (firewall, gateway).

Server–Systeme werden in der Regel nicht als Arbeitsplatzrechner genutzt, d.h. sie verfügen weder über leistungsfähige Grafikadapter noch über Peripheriegeräte zur direkten Nutzung (Monitor, Tastatur oder Maus).

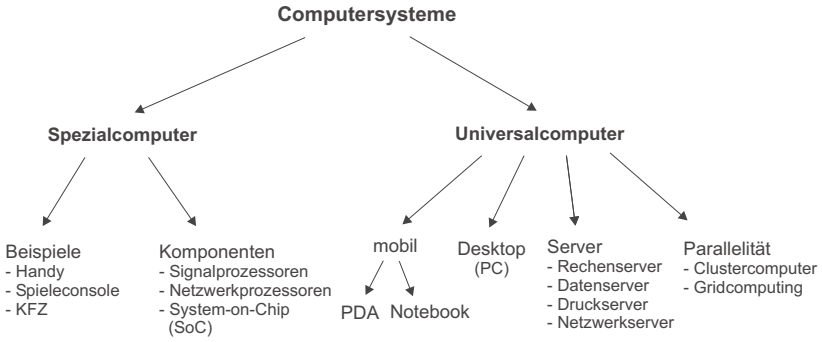


Abb. 10.1. Übersicht über die verschiedenen Arten von Computersystemen.

Um sehr rechenintensive Anwendungen zu beschleunigen, kann man mehrere Compute–Server zu einem so genannten *Clustercomputer* zusammenschalten. Im einfachsten Fall, werden die einzelnen Server–Systeme über einen *Switch*[2] mit Fast– oder Gigabit–Ethernet zusammengeschaltet. Über diese Verbindungen können dann die einzelnen Compute–Server Daten untereinander austauschen und durch gleichzeitige (paralllele) Ausführung von Teilaufgaben die Gesamtaufgabe in kürzerer Zeit lösen. Die maximal erreichbare Beschleunigung hängt dabei von der *Körnigkeit* (granularity) der parallelen Programme ab. Clustercomputer sind vor allem für grobkörnige (coarse grained) Parallelität geeignet. Hier sind die Teilaufgaben zwar sehr rechenintensiv, die einzelnen Programmteile müssen jedoch nur geringe Datenmengen untereinander austauschen.

Je feinkörniger ein paralleles Programm ist, desto höher sind die Anforderungen an das dem Cluster zugrundeliegende Netzwerk. Um eine hohe Beschleunigung der feinkörnigen Programme zu erreichen, muss die Netzwerkverbindung sowohl eine hohe Datenrate bereitstellen als auch möglichst geringe Latenzzeiten aufweisen. Ein Beispiel für ein derartiges (teures) Netzwerk ist das *Myrinet*.

[1] Meist so genannte RAID (Redundant Array of Independent Disks). Vgl. auch Kapitel 8.

[2] Vgl. Kapitel 7.

C. Abkürzungen

Abkürzung	Bedeutung
ATA	Advanced Technology Attachment
AGP	Accelerated Graphics Port
A/D	Analog/Digital
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ARPA	Advanced Research Project Agency Network
ASCII	American Standard Code for Information Interchange
ASM	Algorithmic State Machine
ATC	Address Translation Cache, vgl. TLB
BCD	Binary Coded Decimal
BIOS	Basic Input Output System
bpi	bit per inch, Aufzeichnungsdichte
CAD	Computer Aided Design
CAM	Content Addressable Memory
CCD	Charge Coupled Device
CD	Compact Disk
CDB	Common Data Bus
CISC	Complex Instruction Set Computer
CLAA	Carry Look Ahead Adder
CLAG	Carry Look Ahead Generator
CMAR	Control Memory Address Register
CMOS	Complementary Metal Oxide Semiconductor
CPI	Cycles Per Instruction
CPU	Central Processing Unit, Prozessor, Zentraleinheit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
D/A	Digital/Analog
DC	Directed Current, Gleichstrom
DDR	Double Data Rate

Fortsetzung auf der nächsten Seite