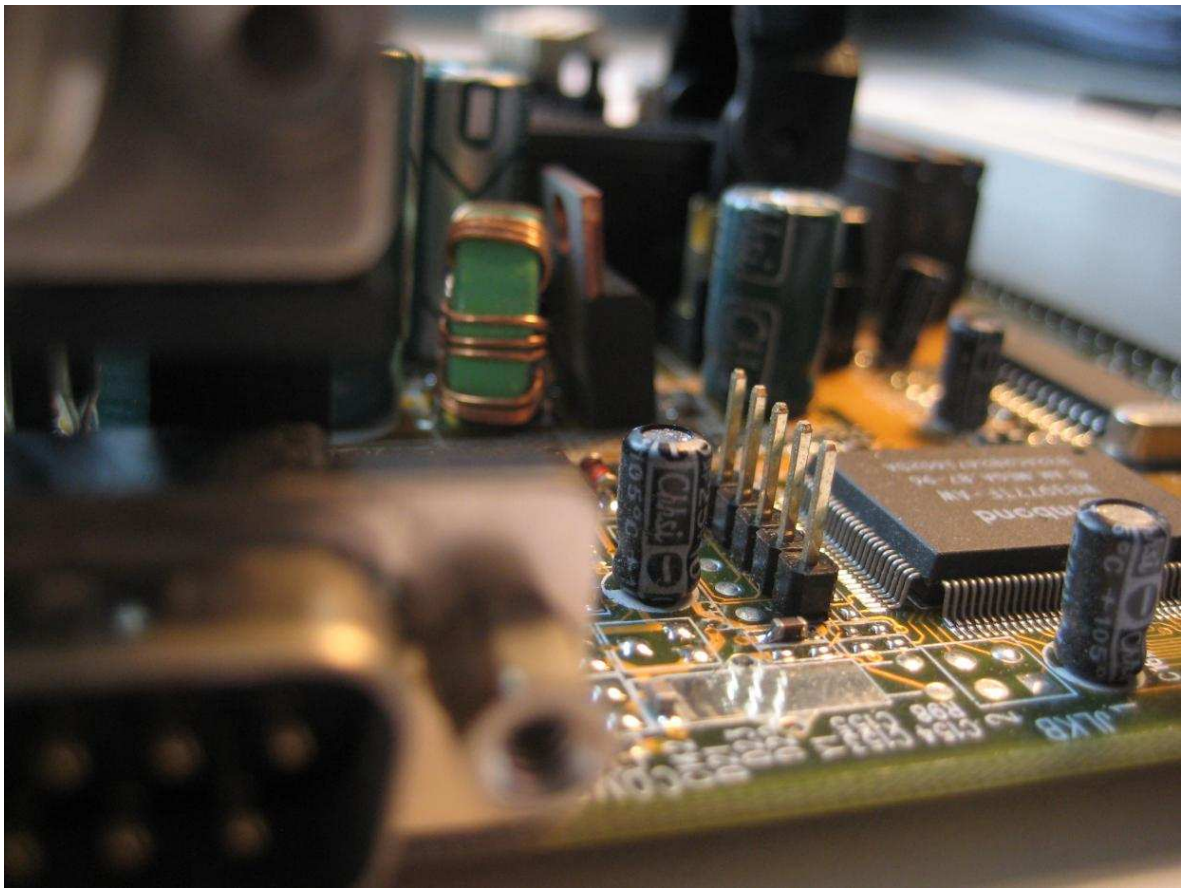
Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme



Rechnerarchitektur

Skript zur Vorlesung im Sommersemester 2008

Prof. Dr. Claudia Linnhoff-Popien

mobile and
distributed systems group

Unter Mitarbeit von: Ulrich Bareth, Caroline Funk, Orlin Gueorguiev,
Hans Dietmar Jäger, Georg Klein, Alex Mirsky, Diana Weiß

Bildnachweis Kapitel 1: MDSG / Ulrich Bareth

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Einführung in die Rechnerarchitektur

1

Motivation

Die verschiedenen Informatikvorlesungen beschäftigen sich mit der Erstellung von Software. Rechnerarchitektur dagegen umfasst die technischen Aspekte von Computersystemen. Nehmen wir also doch einmal einen Rechner und schauen hinein (siehe Abb. 1.1).

Abbildung 1.1: Normaler Arbeitsplatzrechner

Eine der wichtigsten Komponenten eines jeden Rechners ist die Hauptplatine (auch Mainboard oder Motherboard genannt), da sie sämtliche anderen Hardware-Elemente meist durch Steckverbindungen zu einer Einheit - dem Computersystem - miteinander verbindet (siehe Abb. 1.2).

Folgende Komponenten werden in aktuellen Rechnern auf die Hauptplatine gesteckt oder mit Kabeln verbunden, sofern sie nicht schon vom Hersteller direkt auf die Hauptplatine gelötet wurden. Dazu dienen international einheitlich festgelegte Schnittstellenstandards:

– Prozessor (hier nach "Slot 1"-Standard mit 242 Pins für Intel Prozessoren der entsprechenden PentiumII, Celeron und Pentium III Baureihen)

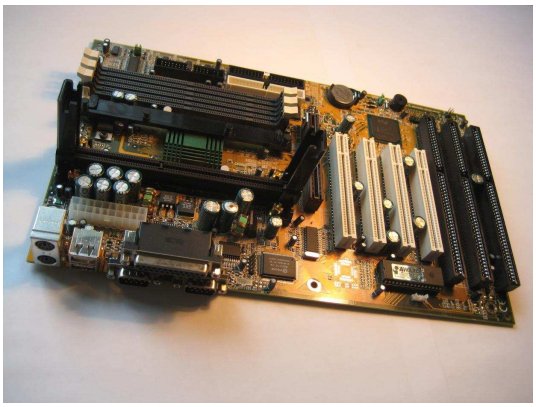


Abbildung 1.2: Hauptplatine eines Computers

- Arbeitsspeicher (hier 168 Pin SDRAM-Standard (Single Data Rate) mit bis zu 100MHz Frontside Bus)
- Laufwerke wie Festplatten, DVD-/CD- oder Disketten-Laufwerke (hier nach 40 Pin IDE-Standard)
- weitere Eingabe/Aausgabe-Geräte wie Grafikkarte, Netzwerkkarte, Sourndkarte (hier nach AGP- bzw. 32 Bit PCI- oder 16 Bit ISA-Standard)

Andere Komponenten werden üblicherweise schon ab Werk auf der Hauptplatine integriert ("onboard") ausgeliefert. Dazu gehören:

- Chipsatz bestehend aus:
  - Northbridge: koordiniert die Kommunikation zwischen der CPU, dem Hauptspeicher, der Southbridge und Hochgeschwindigkeits-Grafikschnittstellen wie AGP oder PCIe
  - Southbridge: koordiniert die Kommunikation zwischen der Northbridge und langsameren Komponenten wie PCI-Karten, Laufwerken, dem BIOS oder Eingabegeräten wie Keyboard oder Maus
- Controller für Laufwerke (bspw. IDE oder Floppy)
- BIOS-Chip mit integrierter Firmware (Basic Input/Output System)

Mittlerweile sind die Hardwarehersteller auch dazu übergegangen, Standardkomponenten wie eine Sound- oder Netzwerkkarte, manchmal auch eine Garfikkarte oder sogar Hauptspeicher und CPU onboard auf der Hauptplatine zu integrieren.

Hinter dieser Ansammlung von einzelnen Komponenten und ihren Verbindungen verbirgt sich doch eine ganz klare Systematik. Diese wollen wir uns im Rahmen dieser Vorlesung erschließen.

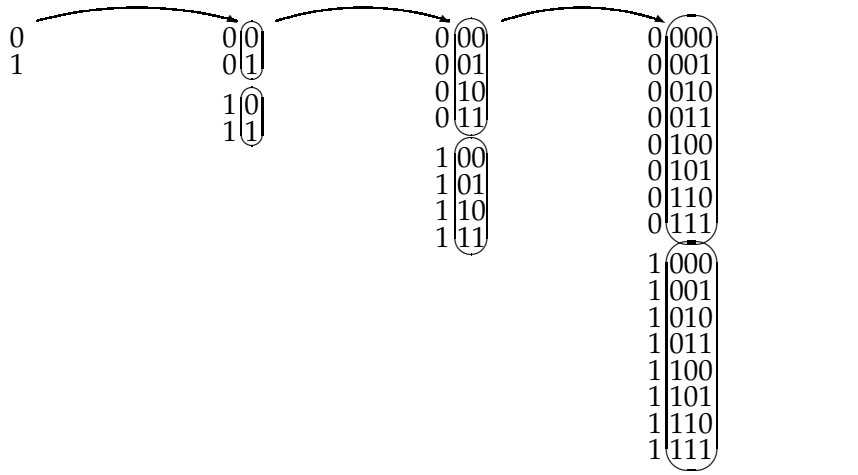
2

Darstellung von Informationen im Rechner

Die einfachste und fundamentalste Art der Darstellung von Informationen sind sogenannte **Wahrheitswerte**. An späterer Stelle werden wir formaler die Boolesche Algebra einführen. Im folgenden soll eine Menge B von Wahrheitswerten aus genau zwei Elementen bestehen: {wahr, falsch} oder {1, 0} oder {L, 0} oder ...
Solche Informationen können vom Rechner gespeichert werden. Dabei wollen wir eine Struktur zugrunde legen, bei der 8 Elemente, sogenannte Bits (Binary Digits) zu einer Information zusammengefaßt werden, zu einem **Byte**. Beispiele für Bytes sind 00000000 oder 01010101 oder auch 01000110.

Den beiden Zuständen eines Bits ordnet man die Werte 0 bzw. 1 zu. Jeder Zustand eines Bits kann unabhängig von den Zuständen der anderen Bits variieren. Damit lassen sich aus den 8 Bits eines Bytes $2^8 = 256$ verschiedene Bitmuster bilden.

Sollte man alle Bitmuster aufschreiben, so empfiehlt sich ein systematisches Vorgehen, bei dem man das letze Bit alterniert und bei Erschöpfung aller Bitmuster das Bit davor betrachtet.



Diesen Bitmustern werden dann Buchstaben, Zeichen oder Zahlen zugeordnet.
Z.B.

$0000\,0000 \mathrel{\widehat{=}} 1$
$0000\,0001 \mathrel{\widehat{=}} 1$
$0000\,0010 \mathrel{\widehat{=}} 2$
$0000\,0011 \mathrel{\widehat{=}} 3$
$0000\,0100 \mathrel{\widehat{=}} 4$
$0000\,0101 \mathrel{\widehat{=}} 5$
$0000\,0110 \mathrel{\widehat{=}} 6$
$0000\,0111 \mathrel{\widehat{=}} 7$
$0000\,1000 \mathrel{\widehat{=}} 8$
$0000\,1001 \mathrel{\widehat{=}} 9$
u.s.w.

Mit einem Byte lassen sich somit 256 verschiedene Informationen verschlüsseln. Bei mehreren Bytes ergeben sich noch mehr Informationen, z.B. bei 4 Bytes = 1 Wort sind das $2^{32} = 256^4$ also rund 10^{10} Informationen. Um eine solche Menge an Informationen kürzer und übersichtlicher darstellen zu können, wird eine hexadezimale Schreibweise eingeführt. Dabei werden jeweils 4 Bits gruppiert und Ihnen eine Zahl oder ein Buchstabe zugeordnet:

$0000 \Rightarrow 0$	$0100 \Rightarrow 4$	$1000 \Rightarrow 8$	$1100 \Rightarrow C$
$0001 \Rightarrow 1$	$0101 \Rightarrow 5$	$1001 \Rightarrow 9$	$1101 \Rightarrow D$
$0010 \Rightarrow 2$	$0110 \Rightarrow 6$	$1010 \Rightarrow A$	$1110 \Rightarrow E$
$0011 \Rightarrow 3$	$0111 \Rightarrow 7$	$1011 \Rightarrow B$	$1111 \Rightarrow F$

1 Byte kann so durch zwei hexadezimale Zeichen dargestellt werden, 1 Wort durch 8 hexadezimale Zeichen (vgl. auch später hexadezimale Zahlendarstellung).

Ein Byte ist in diesem Fall eine **adressierbare Speichereinheit**.

Bemerkung:
In anderen Architekturen können auch 4 Byte eine adressierbare Speichereinheit sein. Die

Anzahl ist weniger von Interesse als das Prinzip: alle Personen, die in einer Straße, in einem Haus wohnen, haben die selbe Adresse.

Wir betrachten nun unseren Speicher und geben den Bytes laufende Nummern, die auch Speicheradressen genannt werden; im folgenden gehen wir von 8 Bit großen Speicherzellen (vgl. Abbildung 2.1) aus.

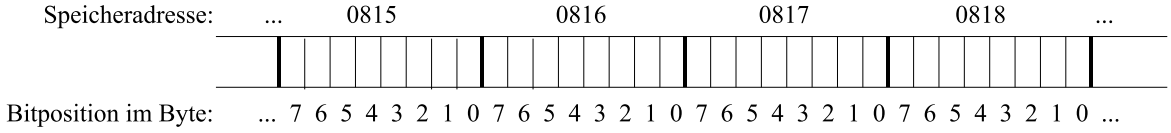


Abbildung 2.1: Speicheradressen

Die Adressierung beginnt bei 0 und endet bei einem 2^n Byte großen Speicher mit der Adresse $2^n - 1$. n ist dabei von der Größe der Hardware abhängig.

Die Größe eines Speichers wird üblicherweise wie folgt angegeben:

Kilobyte: 1 KB = 1024 Byte = 2^{10} Byte = 8.192 Bit $\approx 10^3$ Byte

Megabyte: 1 MB = 1024 KB = 2^{20} Byte = 1.048.576 Byte = 8.388.608 Bit $\approx 10^6$ Byte $\approx 10^6$ Byte

Gigabyte: 1 GB = 1024 MB = 2^{30} Byte =1.048.576 KB = 1.073.741.824 Byte = 8.589.934.592 Bit $\approx 10^9$ Byte

Terabyte: 1 TB = 1024 GB = 2^{40} Byte $\approx 10^{12}$ Byte

Petabyte: 1 PB = 1024 TB = 2^{50} Byte $\approx 10^{15}$ Byte

Wir wollen einen Speicher betrachten, der durch 32 Bit lange Adressen seine 1 Byte großen Speicherzellen adressiert. Wie groß kann der Speicher maximal sein?

32 Bit ermöglichen 2^{32} Bitmuster, d.h. $2^2 \cdot 2^{30}$ Byte = 4 GB können im Speicher adressiert werden.

3

Klassische Komponenten eines Computers

Die Grundlage heutiger digitaler Computer geht auf den Mathematiker **John von Neumann** zurück.

John von Neumann fiel auf, dass die Programmierung von Computern mit Unmengen von Schaltern und Kabeln sehr langsam, mühsam und unflexibel war. Statt dessen schlug er vor, das Programm mit den Daten im Speicher eines Computers in digitaler Form zu speichern und die serielle Dezimalarithmetik, bei der jede Ziffer mit 10 Vakuumröhren dargestellt wurde, durch parallele Binärarithmetik abzulösen.

In seiner Grundstruktur besteht ein Digitalrechner aus einem Verbundsystem von Prozessor, Speicher und Geräten für die Ein– und Ausgabe.
Dazu kommen noch Verbindungen zwischen diesen Einheiten, sogenannte Busse (siehe Abbildung 3.1).

Diese Bestandteile wollen wir im folgenden genauer betrachten.

3.1 Der Prozessor

Der **Prozessor** (Rechnerkern, Central Processing Unit (CPU), zentrale Recheneinheit) ist das Gehirn des Computers. Ein Programm muß sich im Hauptspeicher befinden, um von der CPU ausgeführt zu werden.
Der Prozessor ruft die Befehle (Instruktionen) der Programme ab, prüft sie und führt sie nacheinander aus.

Die CPU besteht aus Daten- und Befehlsprozessor.

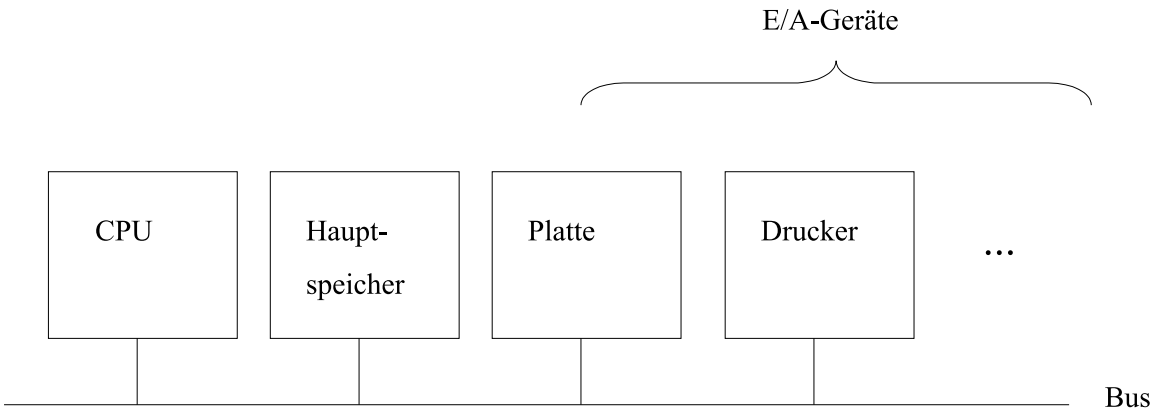


Abbildung 3.1: Komponenten eines Computers

Datenprozessor:

Der Datenprozessor ist zuständig für das klassische Verarbeiten von Daten und die Ausführung von Berechnungen. Der Datenprozessor enthält

- ein **Rechenwerk**, die sogenannte ALU (Arithmetic Logical Unit),
- und (mindestens) drei Speicherplätze (**Register**) zur Aufnahme von Operanden, die bezeichnet werden als **Akkumulator** (`A`), **Multiplikatorregister** (`MR`) und **Link-Register** (`L`).
  Hinzu tritt in einigen Fällen ein viertes Register, das sogenannte **memory buffer register** (`MBR`), das für die Kommunikation mit dem (Haupt-)Speicher notwendig ist.

Befehlsprozessor:

Der Befehlsprozessor entschlüsselt Befehle und steuert deren Ausführung. Dazu bedient er sich der folgenden Komponenten:

1. Der aktuell zu bearbeitende Befehl befindet sich im **Befehlsregister** (**i**nstruction **r**egister, `IR`).
2. Die Adresse des Speicherplatzes, der als nächstes angesprochen wird, ist im **Speicheradreßregister** (**m**emory **a**ddress **r**egister, `MAR`) vorhanden.
3. Die Adresse des nächsten auszuführenden Befehles wird im **Befehlszähler** (**p**rogram **c**ounter, `PC`) gespeichert.
4. Die Entschlüsselung eines Befehls erfolgt durch einen separaten **Befehlsdecodierer**.
5. Die Steuerung der Ausführung erfolgt durch das **Steuerwerk**.

Die technische Realisierung erfolgt über Chips. Das sind einzelne, dünne Träger aus kristallinem Silizium oder einem anderen Halbleiter. Darauf werden Schaltelemente untergebracht.

Die CPU ist ein Beispiel für einen Steuerungschip. Daneben gibt es auch Speicherchips (vgl. Abschnitt 3.2).

Fast alle modernen CPUs befinden sich auf einem einzigen Chip. Dadurch ist ihre Interaktion mit dem restlichen System gut definiert. Jeder CPU-Chip hat eine Reihe von sogenannten Pins, durch die seine gesamte Kommunikation mit der Außenwerlt erfolgt. Einige Pins geben Signale von der CPU nach draußen, andere Pins erhalten Signale von außen für die CPU. Und wieder andere können für die Ein- und Ausgabe verwendet werden.

Die Pins auf einem CPU-Chip können drei Klassen von Informationen übermitteln: Adressen, Daten und Steuerungsinformationen, auf die in einem späteren Kapitel der Vorlesung ausführlich eingegangen wird.

Die CPU kommuniziert mit dem Speicher und den E/A-Geräten, indem sie Signale auf ihre Pins legt und Signale von Pins entgegennimmt. Eine andere Kommunikation ist nicht möglich.

Zwei der wichtigsten Paramter, mit denen die Leistung einer CPU ermittelt werden kann, ist die Anzahl ihrer Adreß- und Datenpins. (Steuerpins sollen hier nicht betrachtet werden.)

Ein Chip mit m Adreßpins kann bis zu 2^m Speicherzellen adressieren. Übliche Werte von m sind $16, 20, 32$ und 64. Ein Chip mit n Datenpins kann ein n-Bit-Wort in einer Operation lesen oder schreiben. Übliche Werte von n sind $8, 16, 32$ und 64. Eine CPU mit 8 Datenpins benötigt folglich vier Operationen, um ein 32-Bit-Wort zu lesen, während eine CPU mit 32 Datenpins die gleiche Arbeit in einer Operation erledigt. Je mehr Datenpins ein Chip hat, desto schneller ist er also, aber er ist auch erheblich teurer.

Beispiel: Pentium II

Der Pentium II hat 242 Pins zur Außenwelt für 170 Signale, 27 Stromanschlüsse (in unterschiedlichen Spannungen), 35 Masseleitungen und 10 Reserven für eine evtl. künftige Verwendung. Ein Teil der logischen Signale belegt zwei oder mehr Pins, so dass es nur 53 verschiedene Pins gibt.

Die Adressen des Pentium II sind 36 Bit breit, die drei niederwertigen Bits müssen aber immer 0 sein, deshalb sind ihnen keine Pins zugewiesen. Das Adreßsignal A# (sprich: A Raute) hat demzufolge 33 Pins. Mit 36 Adreßbits beträgt der maximal adressierbare Speicher 2^{36}, also 64 GByte. Dabei kann aber nur jede achte Speicherzelle angesprochen werden, da jede Adresse mit 3 Nullen endet. Deshalb werden bei einem Datentransfer 8 Byte auf einmal geladen. Das Datensignal D# benötigt demzufolge 64 Pins, siehe Abbildung 3.2. Dabei entspricht das Adreßsignal A# dem Adreßbus und das Datensignal D# dem Datenbus.

Beispiel 2: UltraSparc II

Als zweites Beispiel soll die UltraSPARC-Familie von Sun betrachtet werden, die eine 64-Bit-SPARC-CPU benutzt. Diese Prozessoren werden in Sun-Workstations und -Servern benutzt.

Die UltraSPARC-II-CPU ist ein großer Einzelchip mit 787 Pins an der Unterseite. Es werden 64 Bits für Adressen und 128 Bits für Daten genutzt.

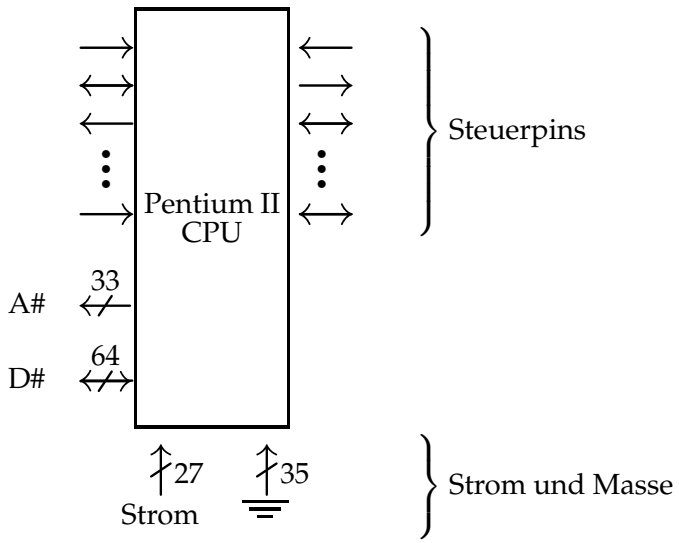


Abbildung 3.2: Logisches Pinout des Pentium II

Die Gesamtzahl von 787 Pins wurde sehr reichlich mit vielen unbenutzten und redundanten Pins gewählt. Außerdem glaubt die Industrie scheinbar, Glück zu haben, wenn sie über eine Primzahl von Pins verfügt.

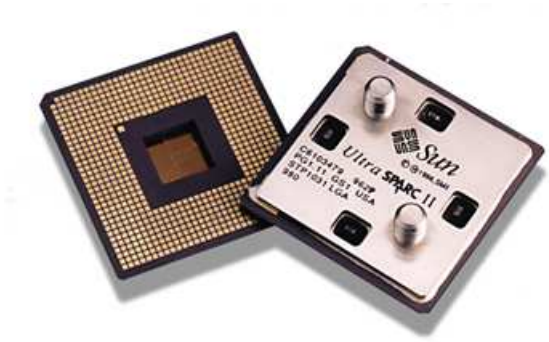


Abbildung 3.3: Der CPU-Chip der UltraSparc II

3.1.1 Die Arbeitsweise der CPU

Folgende Prinzipien charakterisieren die Arbeitsweise eines Rechners im von-Neumann-Modell:

– Zu jedem Zeitpunkt führt die CPU genau einen Befehl aus, dieser Befehl kann nur genau einen Datenwert bearbeiten. Allerdings kann bei Operationen wie der Addition ein

zusätzlicher Wert verwendet werden[1]. Dieser Aufbau wird auch bezeichnet als **s**ingle **i**nstruction, **s**ingle **d**ata (SISD).

– Alle Speicherwerte, d.h. alle Inhalte einer **Speicherzelle**, sind als Daten, Befehle oder Adressen brauchbar, die konkrete Bedeutung ergibt sich aus dem Kontext.

– Da also Daten und Programme nicht in getrennten Speichern untergebracht werden, besteht grundsätzlich keine Möglichkeit, die Daten vor ungerechtfertigtem Zugriff zu schützen.

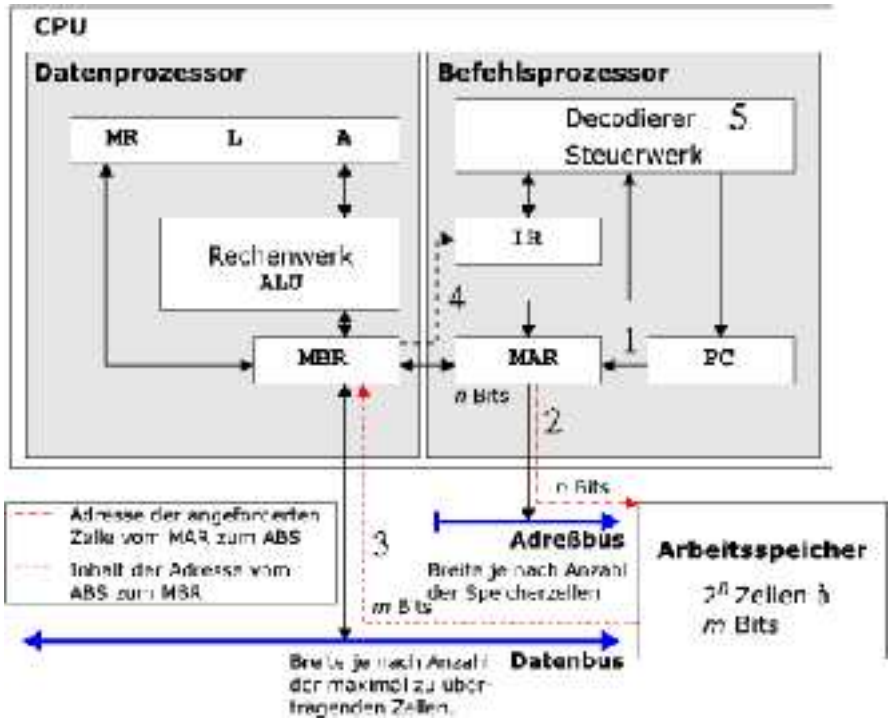


Abbildung 3.4: Aufbau einer CPU

Befehlszyklus:
Eine in Maschinencode vorliegende Befehlsfolge wird nun in zwei Phasen verarbeitet (Befehlszyklus)[2]:

1.Phase: **Fetch-Phase** (Interpretationsphase): [3] Der Inhalt des **PC** wird in das **MAR** geladen und der Inhalt dieser Adresse aus dem Speicher über das **MBR** in das **IR** geholt. Der Rechner geht zu diesem Zeitpunkt davon aus, daß es sich bei dieser Bitfolge um einen

[1]Einstellige Operationen (wie z.B. Negation, Quadratur) werden behandelt, als benötigten sie keinen Operator, da sich dieser bereits im Akkumulator-Register befindet. Bei zweistelligen Operationen reicht folglich die Angabe des zweiten Operanden, welcher mit dem Inhalt des Akkumulator-Registers gemäß der gewünschten Operation verknüpft wird. Das Ergebnis wird wiederum im Akkumulator-Register abgelegt (Ein-Adreß-Befehl).

[2]Es wird hier nur ein stark vereinfachtes Modell angegeben.

[3]Im folgenden stehen spitze Klammern für die Daten an der Adresse des angegebenen Registers.

Befehl handelt. Der Decodierer erkennt, um welchen Befehl und insbesondere um welchen Befehlstyp es sich handelt. Nehmen wir an, der aktuelle Befehl ist ein „Memory-Reference-Befehl“, welcher also – im Gegensatz etwa zu einem Halt-Befehl – einen zweiten Operanden aus dem Speicher benötigt, so weiß der Rechner, daß als nächstes dieser Operand aus dem Speicher geholt und im **MBR** abgelegt werden muß. Schließlich muß der Inhalt des **PC** aktualisiert werden, vgl. Abbildung 3.4.

```
1   MAR := PC;       (1)
2   MBR := S[MAR] ;  (2,3)
3   IR  := MBR;      (4)
4   decodiere(IR);   (5)
5   IF NOT Sprungbefehl THEN
6   BEGIN
7     <Stelle Operanden bereit>;
8     PC := PC + 1;    // bei 32-Bit Architektur PC := PC+4
9   ELSE
10    PC := <Sprungzieladresse>;
11  END
```

In Zeile 4 wird der Befehl dekodiert und im Falle eines Sprungbefehls das **PC**-Register in Zeile 10 auf die Sprungzieladresse gesetzt, andernfalls wird das **PC**-Register in Zeile 8 um eins erhöht (bei 32-Bit Architektur um vier).

2.Phase: **Execution-Phase** (**Ausführungsphase**): Hierbei wird die eigentliche Befehlsausführung erledigt, sowie die Initiierung der folgenden Fetch-Phase.

Als die von-Neumann-Architektur entwickelt wurde, stellte die Ausführung eines Befehls in der ALU den zeitintensivsten Teil der Operation dar. Heute jedoch ist die Zeit, die zum Lesen von Speicherinhalten aus dem Arbeitsspeicher, sowie zur Übertragung dieser Daten über den Datenbus benötigt wird, um ein Vielfaches höher als die für die eigentliche Ausführung der Befehle benötigte Zeit. Daher kommt es zwischen CPU und Arbeitsspeicher zu einem **bottleneck**, dem sogenannten **von-Neumann-Flaschenhals**.

3.2 Der Speicher

Der Speicher ist aufgebaut als Folge von Einheiten (bei der MIPS von 8 Bit Größe), sogenannten Speicherzellen. Diese Speicherzellen sind einzeln adressierbar über eine eindeutige Adresse. Es lassen sich zwei Kenngrößen für den Speicher definieren, nämlich

1. die **Breite** m einer Zelle und
2. die **Gesamtzahl** N = 2^n der Zellen, die von 0 bis 2^n -1 durchnummeriert sind.

Damit ergibt sich eine Speicherkapazität von $2^n \cdot$ m Bits.

Beispiel:

$m = 1$ Byte, $n = 32$ ergibt $2^{32} \cdot 1$ Byte = 4 GByte.

ODER

$m = 4$ Byte, $n = 40$ ergibt $2^{40} \cdot 4$ Byte = 4 TByte.

Beispiel 3.1 *(SPIM):* Die SPIM-Modellmaschine, die die Architektur einer MIPS R2000/R3000 simuliert, hat einen Arbeitsspeicher von $N = 2^{32}$ adressierbaren Speicherzellen, die jeweils ein Byte ($m = 8$) aufnehmen können.

Bei modernen Rechnern (wie der MIPS) lassen sich neben der adressierbaren Grundeinheit auch verschiedene Vielfache dieser Einheit ansprechen, z.B. 2 Bytes (oder Halbwort), 4 Bytes (oder Wort).

Um eine Zelle zu adressieren, benötigen wir n Bits (da $N = 2^n$), das **MAR** muß also n Bits aufnehmen können. Denn mit n Bits können wir 2^n Bitmuster erzeugen, d.h. 2^n Speicherzellen adressieren.

Beispiel 3.2 *(MIPS):* Die Adressen der MIPS sind 32 Bit lang (analog zu vorstehendem Satz, wobei die MI $N = 2^{32}$ Speicherzellen enthält). Eine Adresse von benötigten Daten ist 32 Bit lang, sie wird in das **MAR** geschrieben, das also ebenfalls 32 Bit Speicherplatz zur Verfügung stellen muß.

Speicherhierarchie

Wegen des oben beschriebenen von-Neumann-Flaschenhals ist eine schnelle Ausführung vor allem durch wenig Datentransport zwischen CPU und Arbeitsspeicher zu erreichen. Dazu wird eine **Speicherhierarchie** definiert, wie sie sich heute in beinahe jedem Rechner befindet (vgl. Abbildung 3.5). Dabei kommuniziert die CPU mit einer top-down-organisierten Folge von Speichern.

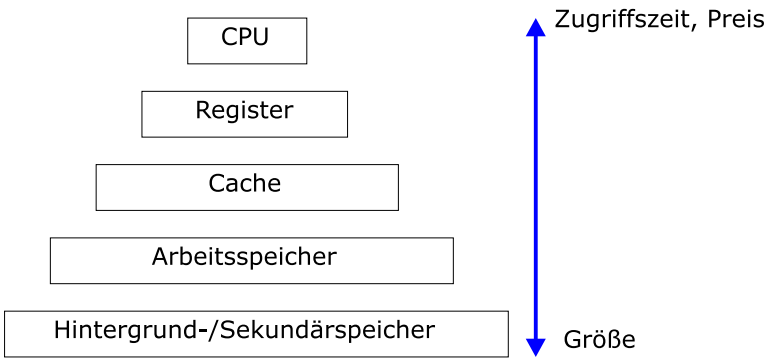


Abbildung 3.5: Speicherhierarchie

Je „näher“der Speicher an der CPU ist, desto teurer ist die Realisierung einer gewissen Speichergröße, da eine geringere Zugriffszeit erforderlich ist.

Register: Unmittelbar der CPU zugeordnet sind die Register (MIPS: 32-Bit-Register) zur Zwischenspeicherung von Werten und zum Ablegen von Informationen, die für die Befehlsausführung benötigt werden.

Cache: Auf der zweiten Hierarchieebene befindet sich der Cache-Speicher, d.h. das Bindeglied zwischen CPU und Arbeitsspeicher, in dem die als nächstes oder die am häufigsten benutzten Daten und Befehle zwischengespeichert werden. Die zugrundeliegende Idee ist die sogenannte **90:10 Regel**. Bei 90% aller Zugriffe werden nur 10% aller Daten, mit denen ein Programm insgesamt arbeitet, benötigt. (Datenlokalität!)

Hauptspeicher: Der Arbeitsspeicher (= Hauptspeicher) hat in der Regel (heute) eine Größe im Megabyte-Bereich. Wie beim Hintergrundspeicher gibt es Arbeitsspeicher im wesentlichen in zwei Typen:

ROM	***r**ead **o**nly **m**emory*, z. B. für Systemfunktionen
RAM	***r**andom **a**ccess **m**emory*

Hintergrundspeicher: Der Sekundär- oder Hintergrundspeicher dient zur Aufnahme von Daten und Programmen, auf die relativ selten zugegriffen werden muß. Außerdem stellt er die einzige Möglichkeit zur persistenten Speicherung, also auch für sichere **Langzeitarchivierung**, dar.

3.3 Die Busse

Busse stellen die Verbindungselemente zwischen den Hauptelementen eines von-Neumann-Rechners dar. Busse können außerhalb der CPU liegen und verbinden diese mit dem Speicher und der E/A-Einheit. Es gibt sie aber auch innerhalb der CPU um z.B. Daten von und zwischen ALU zu transportieren. Ein Bus besteht aus einer oder mehreren parallel verlaufenden Leitungen zur Übertragung von Adressen, Daten und Steuersignalen. Anders als bei einem Geräteanschluß (bei dem ein Gerät mit einem anderen über eine oder mehrere Leitungen verbunden ist), kann ein Bus verschiedene Peripheriegeräte über den gleichen Satz von Leitungen miteinander verbinden. Man unterscheidet generell zwei verschiedene Arten von Bussen:

Serielle Busse Ein serieller Bus besteht aus einer **Ein-Bit-Leitung**, d.h. zu einem Zeitpunkt kann nur eine Information abgefragt werden. Sie sind billig, aber langsam.

Beispiel: Universal Serial Bus (USB)

Der Universal Serial Bus (USB) ist ein Anschluß für periphere Geräte wie z.B. Maus, Modem, Drucker, Tastatur, Scanner oder Digitalkamera.

Die Übertragungsrate beträgt in der Version 1.1 zunächst 12 MBit/s, beim Standard USB 2.0 (ab Ende 2000) bis zu 480 MBit/s. Dabei kann ein „Plug and Play“ realisiert werden, das heißt, Peripheriegeräte können an den USB sogar bei laufendem Computerbetrieb angeschlossen und sofort genutzt werden. Der Computer braucht nicht neu konfiguiert und gestartet werden.

Parallele Busse Über n parallele Leitungen können n Bits gleichzeitig übertragen werden.

Die Busbreite ist ein sehr wichtiger Design-Parameter. Je mehr Adreßleitungen ein Bus hat, um so mehr Speicher kann die CPU direkt adressieren. Hat ein Bus n Adreßleitungen, so kann eine CPU über diesen Bus 2^n verschiedene Speicherzellen adressieren. Um größere Speicher zu ermöglichen, brauchen Busse viele Adreßleitungen.

Dabei erfordern breite Busse mehr Kupferdrähte als schmalere. Sie erfordern auch mehr Platz und sind teurer. Aus diesem Grund neigen viele Systemdesigner zu der Kurzsichtigkeit, Busse minimal zu designen.

Historisches Beispiel:
Der ursprüngliche IBM-PC enthielt eine 8088-CPU und einen 20-Bit-Adreßbus. Damit hatte der PC 20 Bits zur Adressierung eines Speichers von einem MByte.

Als der nächste CPU-Chip 80286 herauskam, entschloß sich Intel den Adreßraum auf 16 MByte zu vergrößern, so daß hier weitere Busleitungen hinzugefügt werden mußten, ohne die ursprünglichen 20 Leitungen aus Gründen der Aufwärtskompatibilität zu stören. Und es wurden neue Steuerleitungen zur Unterstützung der neuen Adreßleitungen hinzugefügt.

Als der 80386 herauskam, gab es weitere 8 Adreßleitungen und noch mehr Steuerleitungen. Das resultierende Design war der EISA-Bus, der **Extended Industry Standard Architecture** Bus. Dieser wäre viel strukturierter geworden, hätte man den Bus von Anfang an mit 32 Leitungen ausgestattet.

Theoretisch wäre ein einziger serieller Bus für den gesamten Rechner ausreichend, ein solcher Rechner würde jedoch unter permanentem "Datenstau" leiden. Daher verwendet man mindestens einen Adreß- und einen Datenbus, in der Regel auch noch Steuerbusse. Diese Teilung ist sinnvoll, da im allgemeinen die Länge einer Speicherzelle (MIPS: 8 Bit) ungleich der Länge ihrer Adresse (MIPS: 32 Bit) ist. Die Länge einer Speicheradresse entspricht also der Breite des Adreßbusses und damit der Länge des **MAR**, die Größe eines Speicherplatzes (Anzahl der angesprochenen Bits) der Breite des Datenbusses und damit der Länge des **MBR**, (vgl. Abbildung 3.6 auf der nächsten Seite).

Die oben beschriebene Vergrößerung des Speichers hatte primär Auswirkungen auf den Adreßbus. Allerdings kann man den Speicher eines Rechners auch dadurch erhöhen, daß man den Datenbus vergrößert.

Überlege:
eine Erweiterung des Adreßbusses um ein Bit bedingt eine Verdopplung der Speicherkapazität. Zum gleichen Ergebnis würde man mit einer Verdopplung der Breite des Datenbusses kommen.

Es gibt aber noch eine Alternative zur Erhöhung der Datenbusbreite (mehr Bit/Transfer), um die sogenannte Datenbandbreite eines Busses zu erhöhen. Dies ist eine Verringerung der Buszyklenzeit (mehr Transfers/Sekunde). Ein mögliches Problem bei einer solchen Beschleunigung des Busses wäre auch der Verlust der Aufwärtskompatibilität. Ältere Platinen, die für einen langsameren Bus ausgelegt sind funktionieren bei schnelleren Bussen nicht. Daher wird die zuvorgenannte Methode der Erhöhung von Datenleitungen in der Praxis normalerweise verwendet.

Praktisch kann es noch weitere rechnerinterne Busse geben, z.B. **Synchronisationsleitungen**, die die einzelnen Teile des Rechners mit der **Takt-Clock** verbinden, oder einen **I/O-Bus**,

welcher die Verbindung mit **Primärspeicher** und/oder **Sekundärspeicher** herstellt (vgl. Abbildung 3.6).

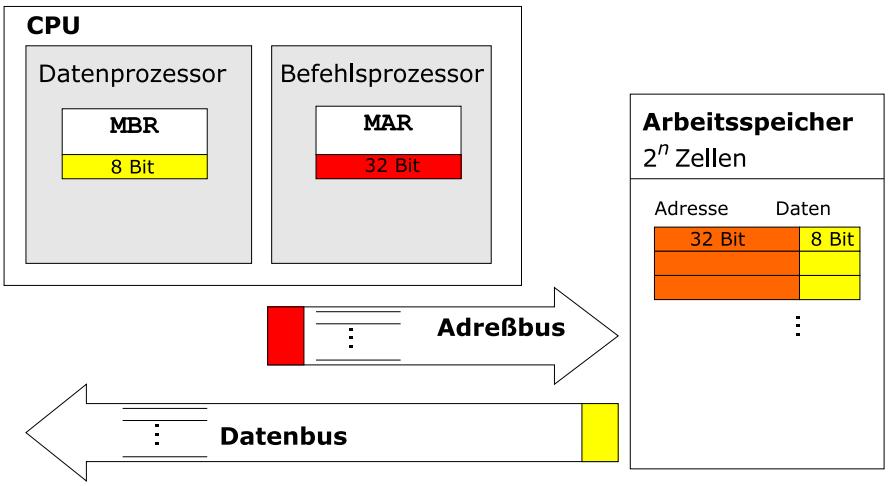


Abbildung 3.6: Transfer von Adressen und Daten zwischen Speicher und CPU

4

Klassifikation der von-Neumann-Rechner

Folgende Klassifikationen haben sich als nützlich erwiesen:

1. Klassifikation nach **Preis** und **Leistungsfähigkeit**:

   (a) **Personalcomputer** (PC): Rechner auf Mikroprozessorbasis, der im allgemeinen nur von einem Nutzer gleichzeitig genutzt wird. Moderne PCs sind netzwerkfähig, damit können Ressourcen auch von anderen Benutzern mitverwendet werden.

   (b) **Arbeitsplatzrechner** (**Workstation**): Sie werden in der Regel mit Mehrbenutzer-Betriebssystem und Netz ausgestattet.

   (c) **Großrechner** ("Mainframes"): Hochgeschwindigkeitsrechner, die von vielen Anwendern gleichzeitig benutzt werden.

   Die Grenzen zwischen den Kategorien sind fließend.

2. Klassifikation nach **Maschinenbefehlssatz:** Diese Klassifikation ist genauer und universeller.

   (a) Anfang der 70er Jahre wurden Prozessoren mit immer umfangreicheren Befehlssätzen ausgestattet, so daß mehr als 200 Befehle implementiert wurden (**CISC** – ***c**omplex **i**nstruction **s**et **c**omputer*). Die Komplexität des CISC-Befehlssatzes führte jedoch zu einer schlechten Geschwindigkeit, vor allem aufgrund folgender Tatsachen:

   – Geschwindigkeit für Speicherzugriffe ist niedriger als die komplexen CPU-Operationen von-Neumann-Flaschenhals.

- Gewisse Befehle werden nur in sehr speziellen Anwendungen verwendet, trotzdem existiert der **Mikrocode**.
- Die aufwendige Mikroprogrammierung des Steuerwerks ist deutlich langsamer als die direkte Verdrahtung der Befehle in Hardware.

(b) Seit Mitte der 70er Jahre werden CPUs mit **RISC**- oder ***r**educed **i**nstruction **s**et **c**omputer*-Befehlssatz gefertigt. Der Befehlssatz verfügt nur über wenig Instruktionen und Adressierungsarten, so daß die Instruktionen direkt in Hardware realisiert werden können und ein festes Format besitzen.

- Der Befehlssatz umfaßt in der Regel nur wenige Instruktionen, sowie Adressierungsarten und ist häufig sogar auf bestimmte Anwendungen hin optimiert.
- Diese wenigen Grundfunktionen können meist innerhalb eines Maschinen-Zyklus ausgeführt werden, d.h. ***c**locks **p**er **i**nstruction* CPI $= 1$.
- Auf den Hauptspeicher wird nur mit seriellen Befehlen (*load-and-store*-Befehle zugegriffen; alle anderen Befehle haben nur Register als Operationsraum.
- Die Befehlsausführung wird durch weitgehende Realisierung der Befehle direkt in der Hardware und eine große Anzahl von Registern beschleunigt.
- Befehlsdecoder und Steuerwerk sind fest verdrahtet, also nicht mikroprogrammiert.
- Alle Instruktionen haben *ein* festes Format.

Neben den von-Neumann-Rechnern lassen sich noch andere Arten von **Rechnerarchitekturen** finden. Diese werden in das Schema (Klassifikation nach Flynn) aus Tabelle 4.1 eingeordnet.

SISD	***s**ingle **i**nstruction – **s**ingle **d**ata*
SIMD	***s**ingle **i**nstruction – **m**ultiple **d**ata*
MISD	***m**ultiple **i**nstruction – **s**ingle **d**ata*
MIMD	***m**ultiple **i**nstruction – **m**ultiple **d**ata*

Tabelle 4.1: Übersicht Rechnerarchitekturen

Daneben kann man alternativ die Anzahl der zu verarbeitenden Befehle durch die Anzahl der zur Verfügung stehenden Datenleitungen beschreiben und diese mit der Anzahl der Prozessoren eines Rechners verknüpfen. Damit ergeben sich 4 mögliche Architekturen, die in Tabelle 4.2 auf der nächsten Seite dargestellt werden.

MISD macht keinen Sinn, da bei dieser Architektur ein Datensatz parallel von mehreren Prozessoren bearbeitet werden müßte.

		Anzahl Datenleitungen	
		1 Befehl	mehrere Befehle
Anzahl	1 Prozessor	SISD	SIMD
Prozessoren	mehrere Prozessoren	MISD	MIMD

Tabelle 4.2: Alternative Charakterisierung

Teil II

Verarbeitung von Bits

5

Arithmetik in Computern

Wir betrachten im folgenden eine Menge von Ziffern oder Zeichen, die wir als Alphabet Σ bezeichnen. Ein Wort ist dann eine Kombination von n Zeichen aus dem Alphabet Σ:

$$w \in \Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n \text{ wobei } \Sigma^n = \underbrace{\Sigma \times \Sigma \times \Sigma \times \cdots \times \Sigma}_{\text{n - mal}}$$

Zahlendarstellung

Als Einführung betrachten wir das Dezimalsystem mit der Basis $b = 10$. Diesem System liegt das Alphabet $\Sigma_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ zugrunde. Worte sind zum Beispiel 31, 2001, 10265. Feste Wortlängen lassen sich durch Aufstockung mit Nullen bilden: zum Beispiel Wortlänge $n = 8$: 00000031, 00002001, 00010265.

Andere Alphabete ergeben neue Zahlensyteme.
Mit der Basis $b = 2$ und $\Sigma_2 = \{0, 1\}$ erhalten wir das Dualsystem, mit der Basis $b = 8$ und $\Sigma_8 = \{0, 1, \ldots, 7\}$ das Oktalsystem und mit $b = 16$ und $\Sigma_{16} = \{0, 1, \ldots, 9, A, \ldots, F\}$ das Hexadezimalsystem, wobei $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$ und $F = 15$ entspricht.

Darstellung einer Zahl zur Basis b:
Sei $b \in \mathbb{N}$ mit $b > 1$. Dann ist jede natürliche Zahl z mit $0 \leq z \leq b^n - 1,\ n \in \mathbb{N}$, eindeutig als Wort der Länge n über Σ_b darstellbar durch

$$z = \sum_{i=0}^{n-1} z_i b^i$$

mit den Ziffern $z_i \in \Sigma_b$ für $i = 0, 1, \ldots, n-1$.
Vereinfachend schreibt man auch $z = (z_{n-1} z_{n-2} \ldots z_2 z_1 z_0)_b$.

Beispiel 5.1*:* Dezimaldarstellung: $z = 2006 = 2 * 10^3 + 0 * 10^2 + 0 * 10^1 + 6 * 10^0 = (2006)_{10}$
Dualdarstellung: $z = 2006 = 1 * 2^{10} + 1 * 2^9 + 1 * 2^8 + 1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = (11111010010)_2$
In diesem Dualzahlbeispiel ist $n = 11$. Damit sind die Zahlen von 0 bis $2^{11} - 1$ darstellbar.
Hexadezimaldarstellung: $z = 2006 = 7 * 16^2 + 13 * 16^1 + 6 * 16^0 = (\underbrace{\underbrace{7}_{0111}\ \underbrace{D}_{1101}\ \underbrace{1}_{0110}}_{0111.1101.0110})_{16}$

16·16		
16	16 ·13	1792
96	16	208
256 ·7	48	6
1792	208	2006

Bemerkungen:

1. Bei der Umwandlung von Hexadezimal– in Binärzahlen kann eine Transformation Stelle für Stelle erfolgen. Umgekehrt kann man Dualzahlen in 4er Blöcke (von hinten beginnend) einteilen und die Blöcke stellenweise in Hexadezimalzahlen transformieren (vergleiche **Kapitel 6**).

2. Im Rechner werden Zahlen u.U. als $(s_0 s_1 \ldots s_{n-1})$ dargestellt, so dass

$$\text{Wert}(s_0, \ldots, s_{n-1}) = \sum_{i=0}^{n-1} s_i 2^{n-1-i} = s_0 2^{n-1} + s_1 2^{n-2} + \ldots + s_{n-1} 2^0$$

   ist. Die Stellen werden damit wertmäßig entgegengesetzt zur Zweierpotenz angeordnet. Daraus resultieren die Begriffe "least significant bit" und "most significant bit".

5.1 Darstellung ganzer Zahlen

Mit n Bit kann man 2^n Bitmuster konstruieren, denen wir bislang lediglich positive Zahlen zugeordnet haben. Nun sollen die selben Bitmuster genutzt werden, um sowohl positive als auch negative Zahlen zu repräsentieren.

Bemerkung:
Statt für $n = 8$ die Zahlen $0 \ldots 2^8 - 1 = 255$ darzustellen, soll $-128 \ldots 127$ dargestellt werden.

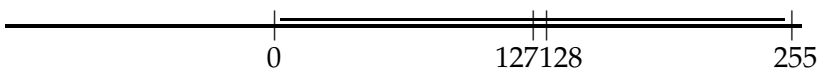


Abbildung 5.1: Positive Zahlen

Zur Darstellung gibt es verschiedene Möglichkeiten.

Wir verschieben also auf dem Zahlenstrahl den Wertebereich.

5.1.1 Vorzeichen/Betrag–Darstellung (sign/magnitude)

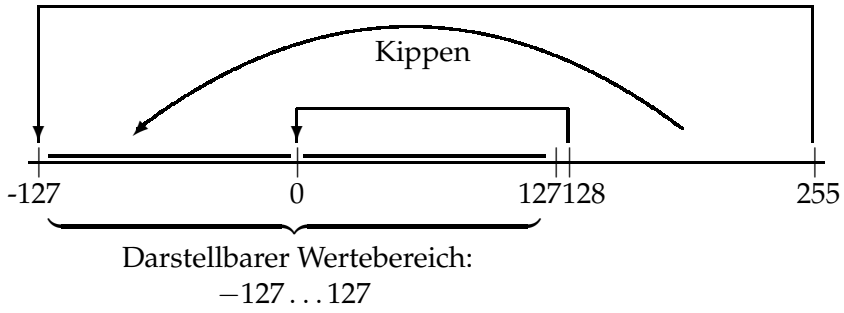


Abbildung 5.2: Vorzeichen/Betrag-Darstellung

Von den n Stellen einer Zahl wird eine Stelle zur Darstellung des Vorzeichens genutzt, z.B. s_0. Dann wird der Betrag durch die verbleibenden $(n-1)$ Stellen dargestellt und mit $(-1)^{s_0}$ multipliziert. Dies ergibt einen Wertebereich von $-2^{n-1}+1$ bis $2^{n-1}-1$, also für $n=8$ der Wertebereich -127 bis 127, für $n=11$ der Wertebereich -1023 bis 1023.

Nachteile:

- Die Null hat zwei Darstellungen (+0 und -0), womit man ein Bitmuster verschenkt.
- Zur Subtraktion braucht man eine eigene Implementierung (ein Subtrahierwerk), da die Addition technisch nicht auf die Subtraktion zurückgeführt werden kann.

5.1.2 Einerkomplement

Beim Einerkomplement entspricht $-x$ dezimal $99 \ldots 99 - x$ oder $x + (-x)_1 = 99 \ldots 99$.

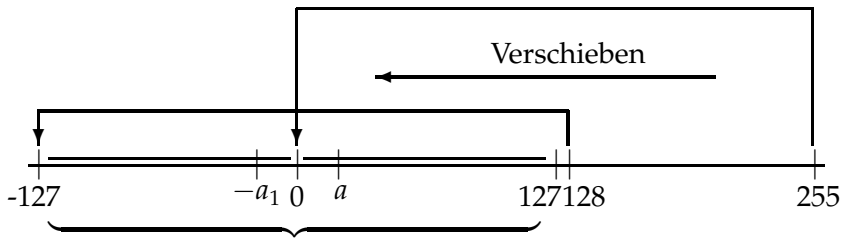
Ausgangspunkt war der Gedanke, was passiert, wenn man eine große Zahl von einer kleinen Zahl abzieht.

Z.B. $3 - 5 =$

$$\begin{array}{rr} & 00000011 \\ - & 00000101 \\ \hline & 11111110 \end{array}$$

Man wird so mit Überträgen rechnen, dass eine Folge von Einsen im Ergebnis der Subtraktion entsteht.

Definition 5.1 (1–Komplement): *Das 1–Komplement von x erhält man durch stellenweises Invertieren der Bits.*



Darstellbarer Wertebereich:
$-127 \ldots 127$

$$\begin{aligned} a + (-a)_1 &= a + 255 - a \\ &= 255 \\ &= 256 - 1 \\ &= 2^n - 1 \end{aligned}$$

Dezimal:
$a + (-a)_1 = 99 \ldots 99 = 10^n - 1$

Abbildung 5.3: Einerkomplement

Z.B.

$$\begin{array}{rclrclcl} k_1(5) & = & (0101)_2 & K_1(5) & = & (1010)_2 & := & -5 \\ k_1(13) & = & (01101)_2 & K_1(-13) & = & (10010)_2 & := & -13 \end{array}$$

Voraussetzung für die Anwendung dieser Methode ist eine feste Anzahl von Stellen.

Formal definieren wir:
Sei $x = (x_{n-1} \ldots x_0)$ eine n–stellige Dualzahl. Dann heißt

$$K_1(x) = (1 \not\leftrightarrow x_{n-1}, \ldots, 1 \not\leftrightarrow x_0)$$

oder

$$K_1(x) = (1 - x_{n-1}, \ldots, 1 - x_0)$$

das 1–Komplement von x.

$\not\leftrightarrow$ bezeichnet man als Antivalenz, $\not\leftrightarrow = \begin{cases} 1 & \text{falls } x_i = 0 \\ 0 & \text{falls } x_i = 1 \end{cases}$

Die Null hat weiterhin zwei Darstellungen: 000...0 und 111...1.

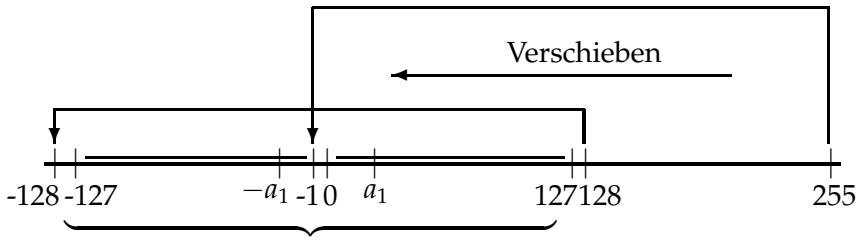
5.1.3 Zweierkomplement

Beim Zweierkomplement entspricht $-x$ dezimal $99 \ldots 99 - x + 1 = 100 \ldots 0 - x$ oder $x + (-x)_2 = 100 \ldots 0$.

Das höchstwertige Bit x_{n-1} einer n–stelligen Dualzahl geht mit dem Wert $-x_{n-1}2^{n-1}$ in den Wert der Zahl ein, die anderen Bits haben die üblichen Werte, d.h.

$$Wert(x) = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + x_{n-3}2^{n-3} + \cdots + x_1 2^1 + x_0 2^0$$

Damit repräsentieren die 2^n Bitmuster einer n–stelligen Dualzahl folgende Worte für n = 8: (vgl. Abbildung 5.5)



$$\begin{aligned} a + (-a)_2 &= a + 256 - a && \text{Dezimal:} \\ &= 256 && 100\ldots0 = 10^n \\ &= 2^n \end{aligned}$$

Abbildung 5.4: Zweierkomplement

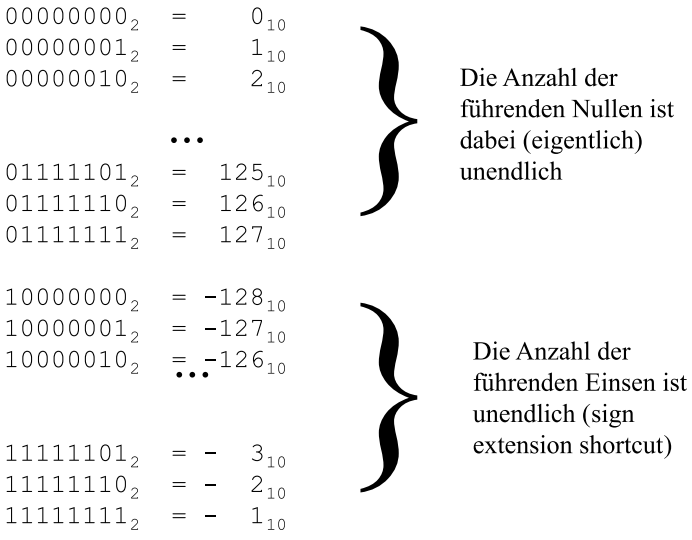


Abbildung 5.5: Dualzahldarstellung

Definition 5.2 (2–Komplement): *Sei $x = (x_{n-1}...x_0)$ eine n–stellige Dualzahl. Dann heißt $K_2(x) = K_1(x) + 1$ das* ***2–Komplement*** *von x. (Negation Shortcut)*

Beispiel:
$K_2(5) = 0101$
$K_2(-5) = 1010 + 1 = 1011$

Damit ergibt sich für $x = 50$ zusammengefasst folgende Darstellung:

Darstellung	+50	-50
Vorzeichen/Betrag	0011.0010	1011.0010
Einerkomplement	0011.0010	1100.1101
Zweierkomplement	0011.0010	1100.1110

Die Zahlen von 0 bis $2^{n-1} - 1$ benutzen die gleiche Darstellung wie in allen anderen Repräsentationen.
1000...0 stellt den größten negativen Wert, d.h. -2^{n-1} dar, der betragsmäßig jeweils um 1 verringert wird, wenn zur Dualzahl eins hinzuaddiert wird.

Beachte:
Bei dieser Darstellung gibt es eine negative Zahl, die in der Darstellung mit (n–1) Stellen keine betragsmäßige Darstellung als positive Zahl hat und die auch nicht durch o.g. Definition (Kippen der Bits und 1 hinzuaddieren) generiert werden kann — nämlich 10000000.

Diese Darstellung verfügt aber auch über Vorteile. Anhand der höchstwertigen Bits (zumindest des höchstwertigen Bits ganz vorne) kann getestet werden, ob es sich um eine positive oder negative Zahl handelt.

Problem der Praxis

Ein und dieselbe Folge von 32 Bits (die mit einer Eins beginnt) kann eine große positive oder auch eine betragsmäßig kleinere Zahl repräsentieren.

Beides macht Sinn:
Ganze Zahlen inklusive des negativen Wertebereichs benötigt man für zahlreiche Programmieraufgaben.
Bei der Adressierung von Speicherzellen benötigt man einen großen Wertebereich – und ausschließlich positive Zahlen.

Doch woran kann der Wert erkannt werden?

Eine Programmiersprache muß diesen Unterschied in ihrer Realisierung beachten.
Daraus resultiert ein Problem: wie vergleicht man Zahlen bzw. Bitstrings, wenn diese einmal negative und einmal positive Werte repräsentieren? (vergleiche Übung)

Allgemein kann man in jedem b-adischen Zahlensystem das $b-1$-Komplement einer Zahl bilden, wenngleich der Fall $b = 2$ für die Informatik mit Abstand am wichtigsten ist.

Wichtig ist in jedem Fall, dass eine Komplementdarstellung stets auf eine beliebige, aber fest vorgegebene Stellenzahl bezogen wird.

Wir wollen uns kurz den Fall $b = 10$ der Dezimalzahlen anschauen bei $n = 8$ Stellen ergibt sich als Neuner- bzw. Zehnerkomplement:

$$\begin{aligned} x &= 143 \\ K_9(x) &= 999.9856 \\ K_{10}(x) &= 9999.9857 \end{aligned}$$

Beachte: $\begin{aligned} x + K_9(x) &= 10^8 - 1 = 9999.9999 \quad \text{und} \\ x + K_{10}(x) &= 10^8 = 1.0000.0000 \end{aligned}$

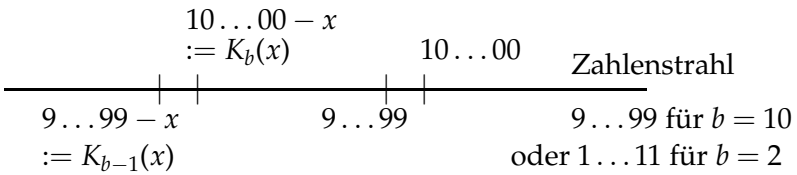
Möchte man nun $1000 - 143 = 1000 + (-143) = 857$ berechnen, so ergibt sich:

$$\begin{array}{r} 1000 \\ +\ 9999.9857 \\ \hline 1.0000.0857 \end{array} = K_{10}(143)$$

Die führende 1 ist ein Übertrag, der durch die Bildung des Komplements von 143 entstanden ist. Dieser wird nun weggelassen und 857 ist das Ergebnis.

Nun wollen wir zeigen, wie mittels des Zweierkomplements eine Subtraktion auf die Addition zurückgeführt werden kann.

Mit dem Einerkomplement berechnet man also die Differenz des Betrags von $(-x)$ und $9\ldots99 = b^n - 1$, mit dem Zweierkomplement die Differenz zu $100\ldots00 = b^n$.



Wegen $99\ldots99 + 1 = 100\ldots00$ ist $K_b(x) = K_{b-1}(x) + 1$, also $K_{10}(x) = K_9(x) + 1$ im Dezimalfall und $K_2(x) = K_1(x) + 1$ im Dualfall.

Beispiel 5.2*:* 1. 20 - 5 = 20 + (-5): vgl. Abbildung 5.6

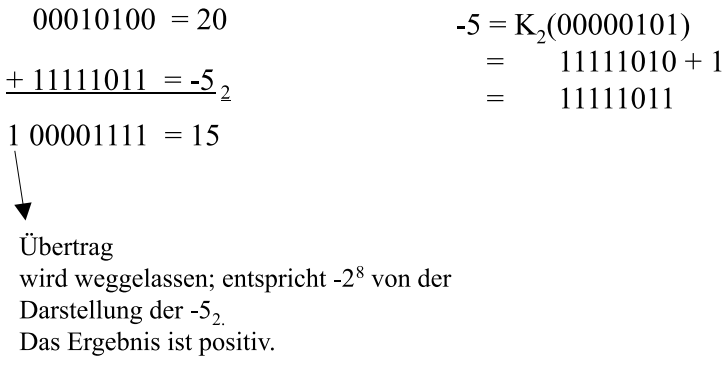


Abbildung 5.6: Beispiel 1

2. 5 - 20 = 5 + (-20): vgl. Abbildung 5.7

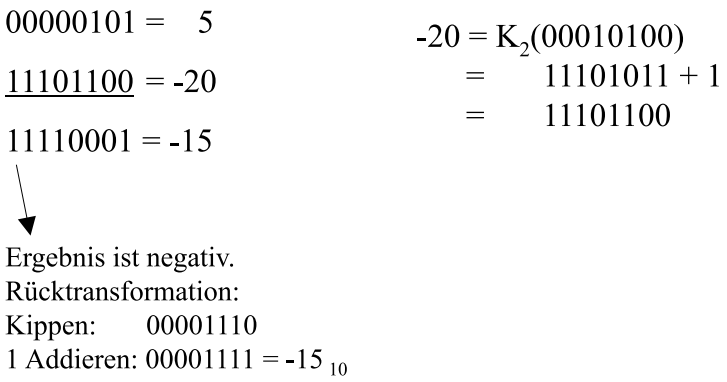


Abbildung 5.7: Beispiel 2

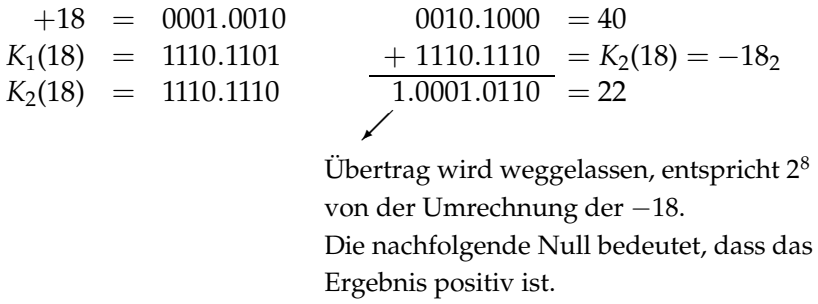


Abbildung 5.8: Beispiel 3

3. 40-18 = 22: vgl. Abbildung 5.8

4. 18-40 = -22, vgl Abbildung 5.9

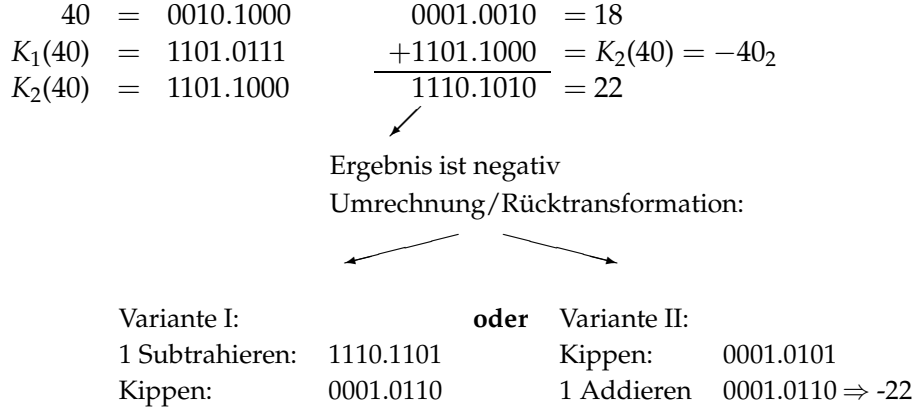


Abbildung 5.9: Beispiel 4

5. -18 + (-40) = -58, vgl. Abbildung 5.10

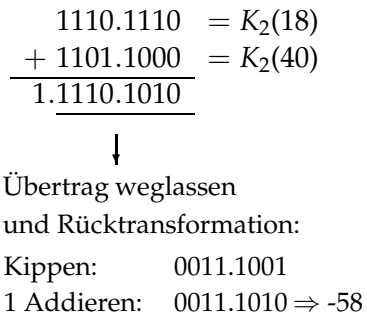


Abbildung 5.10: Beispiel 5

Allgemein passiert folgendes:

1. $z = (z_0, ..., z_{n-1})$
   $\bar{z}_1 = (1 - z_0, ..., 1 - z_{n-1})$
   $\bar{z}_2 = (1 - z_0, ..., 1 - z_{n-1}) + 1$
   $= (1, ..., 1) - (z_0, ..., z_{n-1}) + 1$
   $= 2^n - 1 - z + 1$
   $= 2^n - z$
   $\bar{z}_2 - 2^n = -z$

2. $x - z = x + (-z)$
   $= x + \bar{z}_2 - 2^n$
   $= x + \bar{z}_2 - \underbrace{(100\ldots0)}$
   Übertrag: (n + 1) Stellen

3. $x + \bar{z}_2 = x - z + \underbrace{(100\ldots0)}$
   Übertrag muss gestrichen werden!

Noch einige Beispiele:

1. 47 = 0010.1111
2. $k_2(-47) = k_1(-47) + 1 = 1101.0000 + 1 = 1101.0001$
3. 61 = 0011.1101
4. $k_2(-61) = k_1(-61) + 1 = 1100.0010 + 1 = 1100.0011$

1. 61 - 47 = 14:

$$\begin{array}{rl} 0011.1101 & = 61 \\ +\ 1101.0001 & = K_2(-47) = -47_2 \\ \hline 10000.1110 & = 14 \end{array}$$

2. 47 - 61 = -14:

$$\begin{array}{rl} 0010.1111 & = 47 \\ +\ 0011.1101 & = K_2(-61) = -61_2 \\ \hline 11111.0010 & = -14 \end{array}$$

3. -47 -61 = -108:

$$\begin{array}{rl} 1101.0001 & = k_2(-47) = -47_2 \\ +\ 1100.0011 & = K_2(-61) = -61_2 \\ \hline 11001.0011 & = -108 \end{array}$$

Abbildung 5.13: Beispiel 8

Beachte:
Im Gegensatz zu dem bereits betrachteten **Übertrag** (bei der 2–Komplement–Addition), bei dem die Länge der Operanden ausreicht, um das Ergebnis darzustellen, spricht man von einem **Überlauf** (overflow), wenn die Operandenlänge nicht mehr ausreicht, um den Wert der Zahl noch darzustellen.

5.2 Darstellung reeller Zahlen

5.2.1 Festkommazahlen

Das Komma steht an beliebiger, aber fester Stelle

$$z := (z_{n-1}z_{n-2}\ldots z_1 z_0 z_{-1}z_{-2}\ldots z_{-m})_2$$

z hat die Länge $n+m$, wobei n Stellen vor und m Stellen nach dem Komma gesetzt sind. Als Wert ergibt sich

$$z = \sum_{i=-m}^{n-1} z_i 2^i.$$

Positive und negative Zahlen müssen in dieser Darstellung noch unterschieden werden, z.B. durch ein Vorzeichenbit.

Beispiel 5.3:

$$\begin{aligned} 011,011 &= 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} \\ &= 2 + 1 + 0.25 + 0,125 \\ &= 3,375 \end{aligned}$$

Bei vielen Berechnungen ist der benutzte Zahlenbereich extrem groß oder klein. So ist die Masse eines Elektrons $9 \cdot 10^{-28}$ Gramm, die der Sonne $2 \cdot 10^{33}$ Gramm.

Damit ergibt sich Bedarf an mehr als 60-stelligen Zahlen zur Darstellung dieser Werte in folgender Form:

20.0000.0000.0000.0000.0000.0000.0000.0000,0000.0000.0000.0000.0000.0000.0000

00.0000.0000.0000.0000.0000.0000.0000.0000,0000.0000.0000.0000.0000.0000.0009

Dabei hätten wir 34 Ziffern links und 28 Stellen rechts vom Dezimalkomma. Würde man jedoch alle reellen Zahlen in derartige Darstellungen verpacken, so wären wertvolle CPU-Zeit und Speicherplatz verschwendet.

Probleme:

1. Man kann mit einer bestimmten Anzahl von Bits nur einen beschränkten Wertebereich abdecken.
   Überlege: Was ist die kleinste darstellbare Zahl? Wie sind alle darstellbaren Zahlen auf dem Zahlenstrahl verteilt?

2. Es muß separat gekennzeichnet oder allgemeingültig für alle Darstellungen vereinbart werden, an welcher Stelle sich das Komma befindet.

Daher wird ein System zur Darstellung von Zahlen benötigt, bei denen eine gewisse Variabilität im Bereich der ausdrückbaren Zahlen besteht.

5.2.2 Gleitkommazahlen

andere Bezeichnungen: Gleitpunktzahlen, Floating Point Representations, hier: IEEE 754 Floating Point Standard

Die hier betrachteten Darstellungen sind *nicht nur* für den MIPS-Prozessor gültig. Sie sind Bestandteil des IEEE 754 Floating Point Standards, der quasi in jedem Computer zu finden ist, der nach 1980 entwickelt wurde.

Allgemein geht man von folgender Darstellung aus:

$$(-1)^s \times F \times b^E \text{ und für } b = 2 \text{ ergibt sich: } (-1)^s \times F \times 2^E$$

dabei steht s für Vorzeichen (sign; 1 bedeutet gemäß o.g. Formel negativ), F für die Mantisse (fraction) und E für Exponent.

Die Darstellung $(-1)^s \times F \times 2^E$ ist nicht eindeutig wie auch die Darstellung zur Basis 10: $3,14159265... * 10^0 = 0,314159265... * 10^1 = \underbrace{0,00314159265...}_{\text{Mantisse } m} * \underbrace{10^3}_{\text{Exponent } E} = 314159,265... * 10^{-5}$ u.s.w.

Aus diesem Grund wird eine Darstellung ausgewählt, in die jede Gleitkommazahl überführt werden kann:
Eine Gleitkommazahl der Basis b heißt **normalisiert**, wenn

$$1 \leq |m| < b.$$

Für die Dualschreibweise ergibt sich

$$1 \leq |m| < 2.$$

Damit beginnt jede normalisierte Gleitkommazahl in dualer Darstellung mit 1,... d.h. hat die Form $1, xxxxx * 2^{yyyy}$.

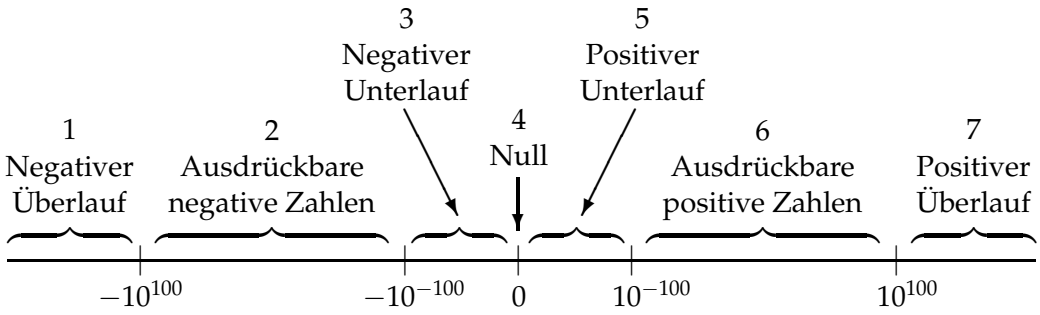
Die Mantisse ist in diesem Fall 1,xxxxx, der Exponent yyyy. Einzige Ausnahme ist bie E = 0. Um maximal viele Bits der Mantisse speichern zu können, wird die führende 1 nicht mitgespeichert.
Folglich werden in der einfachen Darstellung 24 Bits der Mantisse in den 23 Bit Significand abgelegt.
Bei Double können 53 Bit Mantisse in 52 Bit Significand gespeichert werden.

Mit dieser Darstellung von Gleitkomma-Zahlen ergibt sich auf dem Zahlenstrahl ein kleines „Loch“, welches insbesondere die Null selbst enthält. Diesen Sacherverhalt wollen wir uns im dezimalen Fall für eine vorzeichenbehaftete dreistellige Mantisse ($\pm 0, FFF$) und einen zweistelligen Exponenten mit Vorzeichen ($\pm EE$) anschauen.

Auf dem Zahlenstrahl lassen sich damit 7 Bereiche unterteilen:

1. Betragsmäßig große negative Zahlen kleiner als $-0,999 \times 10^{99}$
2. Negative Zahlen zwischen $-0,999 \times 10^{99}$ und $-0,100 \times 10^{-99}$

3. Kleine negative Zahlen mit Größen kleiner als $0,100 \times 10^{-99}$
4. Null
5. Kleine positive Zahlen mit Größen kleiner als $0,100 \times 10^{-99}$
6. Positive Zahlen zwischen $0,100 \times 10^{-99}$ und $0,999 \times 10^{99}$
7. Große positive Zahlen größer als $0,999 \times 10^{99}$



Die Bereiche 1,3,5 und 7 können mit derartigen Zahlen nicht angesprochen werden. Bei den betragsmäßig sehr großen Zahlen (Bereiche 1 und 7) ergibt sich ein Überlauffehler (Overflow Error), wenn z.B. $10^{60} \cdot 10^{60} = 10^{120}$ gerechnet werden soll. In den Bereichen 3 und 5 ergibt sich ein Unterlauffehler (Underflow Error). Dieser Unterlauffehler ist oft nicht so schwerwiegend, wenn man das Ergebnis von z.B.: $10^{-60} \cdot 10^{-60} = 10^{-120}$ durch Angabe einer null annähert.

Wie wird die 0 dargestellt?
Ist der Exponent 0, so wird die implizite 1 der Mantisse nicht weggelassen.

Damit erhält man als Wert einer Gleitkommazahl

$$z = \begin{cases} (-1)^s \times (1 + \text{ Significand}) \times 2^E & \text{, falls } E \neq 0 \\ (-1)^s \times \text{ Significand} \times 2^E & \text{, falls } E = 0 \end{cases}$$

Wird ein Wort (=32 Bit) für die Darstellung einer Gleitkommazahl genutzt, so werden die Bits fest aufgeteilt:

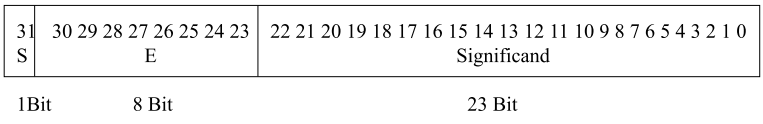


Abbildung 5.14: Bitaufteilung bei Gleitkommazahlen

Der Wertebereich wird deutlich größer als bei Festkommazahlen. Trotzdem können **Überläufe (Overflows)** auftreten, falls bei dem Ergebnis einer Operation der Exponent zur Darstellung nicht ausreicht.

Auf der anderen Seite kann es vorkommen, dass ein Exponent zu klein ist, um noch dargestellt werden zu können. Dann spricht man von einem **Unterlauf (Underflow)**.

Um die Gefahr von Über- und Unterläufen zu reduzieren, können größere Anzahlen von Bits für Gleitkommazahlen bereitgestellt werden. In der Programmiersprache C spricht man dann auch vom Double–Format.
Die Operationen auf Double–Zahlen werden auch Double Precision Floating Point Arithmetics genannt – im Gegensatz zu der Single Precision Floating Point Arithmetic für das vorangegangene Format.

Für Double werden in der MIPS 2 Wörter (= 64 Bit) verwendet.

1.Wort	63 S	62 61 60 59 58 57 56 55 54 53 52 Exponent	51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 Significand
	1 Bit	11 Bits	20 Bits

2.Wort	Significand (continued)
	32 Bits

Abbildung 5.15: Bitaufteilung bei Double

Single ermöglicht Wertebereiche von etwa $2 * 10^{-38} \ldots 2 * 10^{38}$, Double von etwa $2 * 10^{-308} \ldots 2 * 10^{308}$.

Neben der Darstellung größerer Exponenten besteht der Vorteil in einer genaueren Angabe der Mantisse, wodurch akuratere Rechnungen möglich werden.
Beispiel 5.4 *(Gleitkommadarstellung):*

$$\frac{1}{2} = 1,0 * 2^{-1}$$

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Abbildung 5.16: Bitaufteilung für 0,5

$$2 = 1,0 * 2^{1}$$

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Abbildung 5.17: Bitaufteilung für 2

Dabei sieht die größere Zahl (d.h. 2) kleiner aus als die andere, obwohl dies nicht der Fall ist.

Bemerkung:
Bislang wurde in diesen Beispielen die Darstellung des Exponenten im Zweierkomplement vorgenommen. Das ist nicht IEEE 754-konform und hat folgende Nachteile:

Die Größe, d.h. Dimension einer Zahl spielt eine entscheidende Rolle für die Addition zweier Zahlen, weshalb der Exponent auch vor der Mantisse steht.

Um anhand des Exponenten bereits zwei Zahlen vergleichen zu können, wird von der Zweierkomplementdarstellung abgewichen.

Der kleinste Exponent soll durch 00...0, der größtmögliche durch 11...1 dargestellt werden. Diese Konvention wird als **Biased Notation** bezeichnet.

Der Bias ist dabei die Zahl, die subtrahiert werden muß, um den wirklichen Wert zu erhalten.

Für die einfache Darstellung verwendet IEEE 754 den Bias 127, für die Darstellung doppelter Genauigkeit den Bias 1023.
Beispiel 5.5 *(Die Darstellung des Exponenten bei* $\frac{1}{2}$ *und 2):* $\frac{1}{2} = 1,0 * 2^{-1}$, d.h. -1 + 127 = 126 = 0111 1110
$2 = 1,0 * 2^{1}$, d.h. 1 + 127 = 128 = 1000 0000

Damit ergibt sich als allgemeiner Wert einer Gleitkommazahl

$$\boxed{(-1)^s * (1 + \text{Significand}) * 2^{(\text{Exponent - Bias})}}$$

Beispiel 5.6*:* Darstellung des Wertes -0,75 in einfacher und doppelter Genauigkeit:

$$-0,75_{10} = -0,11_2 \underbrace{=}_{\text{Normalisierung}} -1,1 * 2^{-1}$$

Darstellung in einfacher Genauigkeit:

$$(-1)^s * (1 + \text{Significand}) * 2^{(\text{Exponent - 127})}$$

ergibt:

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1	0 1 1 1 1 1 1 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Abbildung 5.18: Bitaufteilung für -0,75

Darstellung in doppelter Genauigkeit:

$$(-1)^s * (1 + \text{Significand}) * 2^{(\text{Exponent - 1023})}$$

ergibt:

1. Wort

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2. Wort

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 5.19: Bitaufteilung für -0,75

Rückrechnung einfache Genauigkeit:

$$z = (-1)^1 * (1 + 0,1) * 2^{(126-127)} = -1,1 * 2^{-1} = -0,11_2,$$

d.h. $-0,75_{10}$

Rückrechnung doppelte Genauigkeit:

$$z = (-1)^1 * (1 + 0,1) * 2^{(1022-1023)} = -1,1 * 2^{-1} = -0,11_2,$$

d.h. $-0,75_{10}$

Beispiel:

Gleitkommadarstellung für -5:

$-5_{10} = -101_2 = -1,01 \cdot 2^2 = (-1)^1 \cdot (1 + 0,01) \cdot 2^{129-127}$

Diese Darstellung ergibt $s = 1$, Significand = 0,01 und Exponent = 129.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

oder in doppelter Genauigkeit:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Beachte, dass nur dann zwei Zahlen addiert oder subtrahiert werden können, wenn beide Darstellungen sich auf denselben Exponenten beziehen.

6

Logischer Entwurf von Computern

Ziel dieses Abschnittes ist es, die Grundlagen des Logic Designs zu vermitteln. Dabei soll verstanden werden, worauf das Grundprinzip der Implementierung eines Computers basiert.

Aus grundlegenden logischen Bausteinen, den **Gattern (Gates)** werden einfache, kombinatorische Systeme konstruiert, in denen noch keine Informationen gespeichert werden können.

Ausgangssituation

Die Elektronik in einem modernen Computer arbeitet *digital*.
Damit kann auch im Alltag alles andere, das digital funktioniert mittels Prozessoren gesteuert werden: Uhren, Telefon (ISDN), Mikrowelle, Tiefkühlschrank, Heizung, Thermometer, Fernseher...

Was heißt digital?
Digital Electronic arbeitet nur mit zwei Spannungspegeln (Andere Spannungswerte treten nur temporär auf oder beim Umschalten zwischen hoch und niedrig).

Dies ist der Schlüssel, warum Computer binäre Zahlen zur Darstellung und Übertragung von Informationen verwenden. Ein binäres System erfüllt genau die Anforderungen, die der digitalen Elektronik abstrakt gesehen zugrunde liegen.

Wir wollen nun von den Voltbereichen weggehen und statt dessen von Signalen sprechen:
Signale sind entweder wahr/1/positiv/asserted oder falsch/0/negativ/deasserted.
Dabei sind 0 und 1 komplementäre oder inverse Werte des jeweils anderen.

Im folgenden wollen wir die Transformation solcher Signale betrachten bzw. die Berechnung von neuen Signalen ausgeben.

6.1 Boolesche Algebra

George Boole war ein englischer Mathematiker, der sich Mitte des 19. Jahrhunderts – d.h. bereits vor 150 Jahren – mit der formalen Sicht heutiger digitaler Strukturen beschäftigte.

Im folgenden betrachten wir ein Alphabet $\Sigma_2 = \{0, 1\}$ für das wir ab jetzt die Bezeichnung B verwenden.

mathematische Herangehensweise:

Erklärt man auf B zwei zweistellige Operationen "$\nleftrightarrow$" (Antivalenz) und "*" (Multiplikation) durch

$$\begin{aligned} 0 \nleftrightarrow 0 &= 1 \nleftrightarrow 1 = 0, \\ 1 \nleftrightarrow 0 &= 0 \nleftrightarrow 1 = 1, \\ 0 * 0 &= 0 * 1 = 1 * 0 = 0, \\ \text{und} \quad 1 * 1 &= 1 \end{aligned}$$

so ist B ein Körper der Ordnung 2 mit dem Nullelement 0 und dem Einselement 1.

technische Herangehensweise:

Man betrachte die Variablen a, b $\in$ B und definiere damit auf B drei Operatoren:

- Der **OR–Operator** wird geschrieben als + oder $\vee$. Das Ergebnis einer OR-Operation ist 1, falls mindestens eine der Variablen in $a \vee b$ den Wert 1 besitzt.
  Diese OR-Operation wird auch als **logische Summe** bezeichnet.
  Wahrheitstafel (truth table):

a	b	$a\ OR\ b$
0	0	0
0	1	1
1	0	1
1	1	1

- Der **AND-Operator** wird geschrieben als * oder $\wedge$. Das Ergebnis einer AND-Operation ist 1, falls beide Eingaben bei $a \wedge b$ der Wert 1 haben.
  Diese AND-Operation wird auch als **logisches Produkt** bezeichnet.
  Wahrheitstafel (truth table):

a	b	$a\ AND\ b$
0	0	0
0	1	0
1	0	0
1	1	1

- Der **NOT-Operator** wird geschrieben als $\overline{a}$ oder $\neg a$. Das Ergebnis einer NOT-Operation ist genau dann 1, falls der Operator auf 0 angewandt wurde.
  D.h. dieser Operator bewirkt die **Invertierung** des Eingabewertes.
  Wahrheitstafel (truth table):

a	$NOT\ a$
0	1
1	0

Eine Boolesche Menge B zusammen mit diesen 3 Operatoren wird als **Boolesche Algebra** (B, AND, OR, NOT) oder (B, *, +, $\neg$) bezeichnet. Alternativ kann man auch $B(\wedge, \vee, -)$ schreiben.

In einer Booleschen Algebra gelten verschiedene Gesetze, die zur Manipulation logischer Gleichungen hilfreich sind:

- Kommutativgesetz: $a \vee b = b \vee a$ und $a \wedge b = b \wedge a$
- Assoziativgesetz: $(a \vee b) \vee c = a \vee (b \vee c)$ und $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
- Distributivgesetz: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ und $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
- Identitätsgesetz: $a \vee 0 = a$ und $a \wedge 1 = a$
- Null– und Eins–Gesetz: $a \wedge 0 = 0$ und $a \vee 1 = 1$
- Komplementärgesetz: $a \vee \overline{a} = 1$ und $a \wedge \overline{a} = 0$
- Verschmelzungsgesetz: $(a \vee b) \wedge a = a$ und $(a \wedge b) \vee a = a$
- de Morgansche Regeln: $\overline{a \vee b} = \overline{a} \wedge \overline{b}$ und $\overline{a \wedge b} = \overline{a} \vee \overline{b}$

Ausgehend von den betrachteten drei Operatoren wollen wir allgemein definieren, was eine Boolesche Algebra ist:

Definition 6.1 (Boolesche Funktion): *Eine Funktion $f : B^n \to B$ heißt **n–stellige Boolesche Funktion**.*

Ist $n = 1$, so kommt man insgesamt auf 4 mögliche einstellige Funktionen: f(x)=0; f(x)=1; f(x)=x; f(x)=$\overline{x}$.

x	0	x	$\overline{x}$	1
0	0	0	1	1
1	0	1	0	1

Im Fall $n = 2$ kommt man auf 16 zweistellige Boolesche Funktionen, insbesondere gehören dazu auch AND und OR.

x	y	0	AND	$x\overline{y}$	x	$\overline{x}y$	y	$\leftrightarrow$	OR	NOR	$=$	$\overline{y}$	$\overline{\overline{x}y}$	$\overline{x}$	$\overline{x\overline{y}}$	NAND	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	1	1	0	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	0	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

Allgemein gilt, dass es für jedes beliebige $n \in \mathbb{N}$ mit $n \geq 1$ genau 2^{2^n} n–stellige Boolesche Funktionen gibt.

6.2 Logische Bausteine

Im folgenden wollen wir logische Bausteine zur Realisierung Boolescher Funktionen betrachten.

6.2.1 Gatter

Die logischen Funktionen, die bisher betrachtet wurden, werden nun als Gatter implementiert.
Ein AND–Gatter implementiert beispielsweise die AND–Operation, ein OR–Gatter implementiert die OR–Operation. Diese beiden Gatter realisieren 2–stellige Boolesche Funktionen. Damit haben sie zwei Eingänge und einen Ausgang.
Die logische Funktion NOT ist einstellig. Sie wird als Inverter mit einem Eingang und einem Ausgang implementiert.

Wir verwenden im folgenden gemäß IEEE die Symbole wie in Abbildung 6.1, eine Übersicht

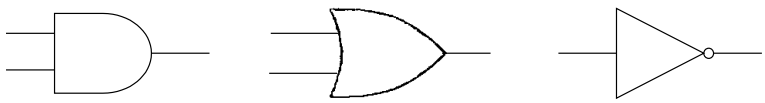


Abbildung 6.1: Gatter AND, OR und NOT

über andere Symbole findet sich unter http://www.du.edu/ etuttle/electron/elect13.htm.

Damit kann z.B. die Funktion $\overline{\overline{a} + b}$ dargestellt werden als

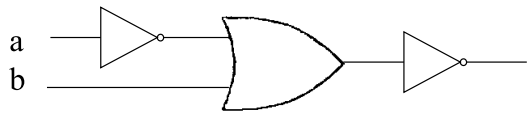


Abbildung 6.2: Gatterverknüpfung

oder vereinfacht als

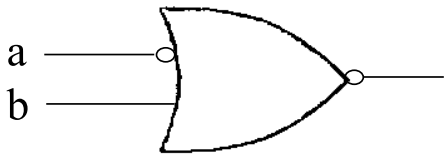


Abbildung 6.3: Gatterverknüpfung

Betrachte NAND und NOR

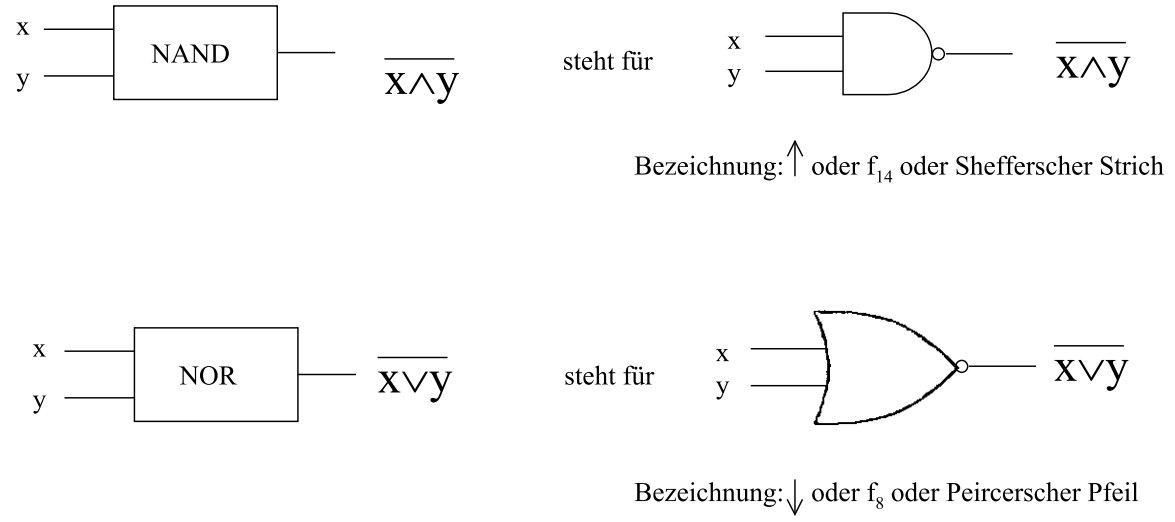


Abbildung 6.4: Gatter NAND und NOR

Es läßt sich zeigen, dass $\{\uparrow\}$ und $\{\downarrow\}$ funktional vollständig sind.
Es reicht bereits *einer* dieser beiden Bausteine prinzipiell aus, um *jede* Boolesche Funktion durch eine Schaltung zu realisieren.

Zur Realisierung einer vorliegenden Antivalenz würde sich folgender Baustein eignen:

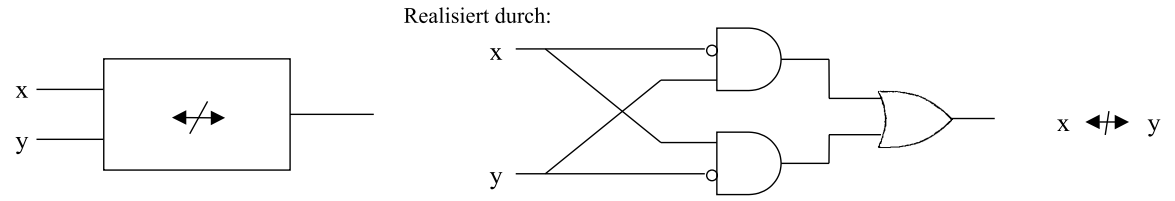


Abbildung 6.5: Antivalenz

In gleicher Weise wollen wir nun Bausteine betrachten, die mehr Ausgänge haben.

Schaltelemente mit mehreren Ausgängen können durch eine Boolesche Funktion nicht mehr beschrieben werden. Aus diesem Grund wird der Begriff der Schaltfunktion eingeführt.

Definition 6.2 (Schaltfunktion): *Eine Funktion $F : B^n \to B^m$ mit $n, m \in \mathbb{N}$ und $n, m \geq 1$ heißt* ***Schaltfunktion****.*

D.h. eine Schaltfunktion besitzt ein n–Tupel von Bits als Eingabe und ein m–Tupel von Bits als Ausgabe.
Der Zusammenhang zwischen einer Schaltfunktion und einer Booleschen Funktion besteht darin, dass man eine Schaltfunktion $F : B^n \to B^m$ durch m Boolesche Funktionen $f_i : B^n \to B$, i = 1,...,m darstellen kann.
Dann ist

$$F(x_1, ..., x_n) = (f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n))$$

für alle $x_1, \ldots, x_n \in B$.
Jede Schaltfunktion ist also durch eine Folge von Booleschen Funktionen beschreibbar.

6.2.2 Decoder

Der Decoder hat n Eingänge und $2^n = m$ Ausgänge, wobei es für jede Eingabekombination genau einen Ausgang gibt, der "wahr" ist, d.h. an dem ein Signal anliegt. Alle anderen Ausgänge sind "falsch".

Beispiel 6.1 *(3–to–8–Decoder):* Wir wollen als Beispiel den 3–to–8–Decoder betrachten, der auch 3–Bit–Decoder genannt wird. Dazu wollen wir die Wahrheitstabelle in Abbildung 6.6

Eingaben (Inputs)			Ausgaben (Outputs)							
I_2	I_1	I_0	Out_7	Out_6	Out_5	Out_4	Out_3	Out_2	Out_1	Out_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Abbildung 6.6: DecoderEin– und Ausgaben

betrachten.

Ist also der Wert der durch die Eingänge (I_2 I_1 I_0) dargestellten Dualzahl i, so ist die Ausgabe Out i wahr, alle anderen Ausgaben sind falsch.

Darstellung: siehe Abbildung 6.7.

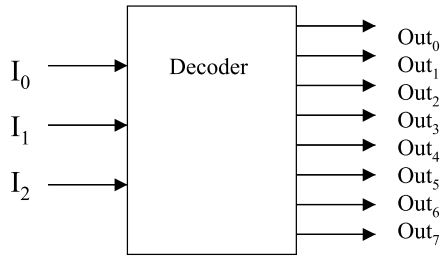
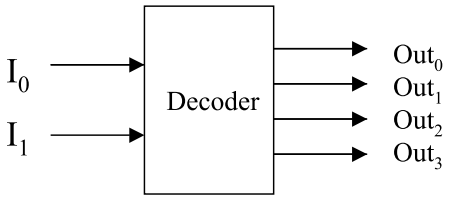


Abbildung 6.7: Decoder

Damit ergeben sich folgende Funktionen für die Realisierung:

$$Out_0 = \overline{I}_0\overline{I}_1\overline{I}_2 \qquad Out_4 = \overline{I}_0\overline{I}_1 I_2$$
$$Out_1 = I_0\overline{I}_1\overline{I}_2 \qquad Out_5 = I_0\overline{I}_1 I_2$$
$$Out_2 = \overline{I}_0 I_1\overline{I}_2 \qquad Out_6 = \overline{I}_0 I_1 I_2$$
$$Out_3 = I_0 I_1\overline{I}_2 \qquad Out_7 = I_0 I_1 I_2$$



I_0	I_1	O_0	O_1	O_2	O_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Wird realisiert als

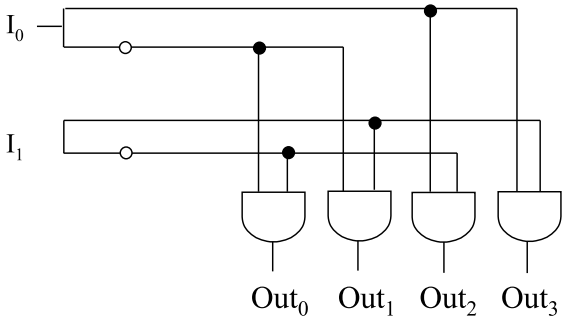


Abbildung 6.8: 2–to–4–Decoder

Beispiel 6.2 *(2–to–4–Decoder):* Betrachte als Beispiel den 2–to–4–Decoder

Bemerkung:
Häufig werden die Bezeichnungen Decoder und DeMUX synonym verwendet. Durch n Inputs wird einer von 2^n Outputs adressiert, der auf 1 gesetzt wird.
Beispiel 6.3*:* Im Falle unseres 3–to–8 Decoders wären n = 3 und m = 8.

6.2.3 Encoder

Die zum Decoder inverse Funktion wird durch das logische Element des sogenannten Encoders realisiert. Der Encoder hat 2^n Eingänge, von denen genau einer wahr sein sollte und produziert einen n–Bit Output.

Im folgenden soll der 8-to-3-Encoder betrachtet werden.

Die zugehörige Wertetabelle lautet:

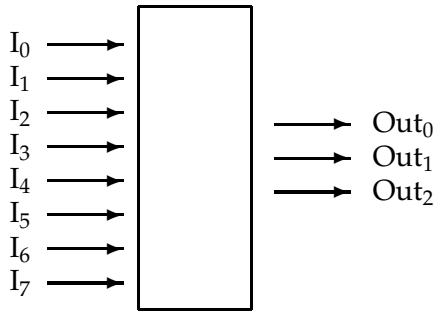


Abbildung 6.9: 8-to-3-Encoder

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Out_2	Out_1	Out_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Allgemein kann man für die Ausgaben entsprechende Funktionen aufstellen, etwa für Out_0:

$$Out_0 = \overline{I}_0 I_1 \overline{I}_2 \overline{I}_3 \overline{I}_4 \overline{I}_5 \overline{I}_6 \overline{I}_7 \vee \overline{I}_0 \overline{I}_1 \overline{I}_2 I_3 \overline{I}_4 \overline{I}_5 \overline{I}_6 \overline{I}_7 \vee \overline{I}_0 \overline{I}_1 \overline{I}_2 \overline{I}_3 \overline{I}_4 I_5 \overline{I}_6 \overline{I}_7 \vee \overline{I}_0 \overline{I}_1 \overline{I}_2 \overline{I}_3 \overline{I}_4 \overline{I}_5 \overline{I}_6 I_7$$

Bestimme Eingänge (wie z.B. $I_0 = I_1 = I_2 = 1$ und $I_3 = I_4 = I_5 = I_6 = I_7 = 0$) sind jedoch nicht belegt. Damit läßt sich die Formel für Out_0 wie folgt vereinfachen:

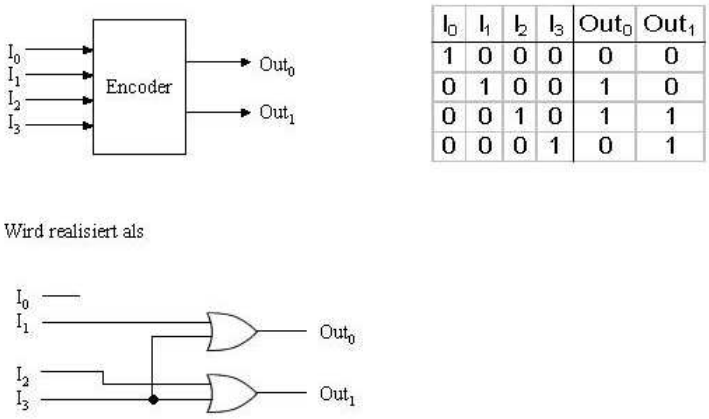
$$Out_0 = I_1 \vee I_3 \vee I_5 \vee I_7$$

Diese Vereinfachung erfhält man auch durch Anwendung, mathematischer Gesetzmäßigkeiten, wie wir in Abschnitt 6.5 sehen werden.

Analog ergibt sich für Out_1 und Out_2:

$$\begin{aligned} Out_1 &= I_2 \vee I_3 \vee I_6 \vee I_7 \\ Out_2 &= I_4 \vee I_5 \vee I_6 \vee I_7 \end{aligned}$$

Der 4–to–2–Encoder (vgl. Abbildung 6.10) besitzt die umgekehrte Funktionalität bezüglich des Decoders. Er hat 2^n Eingänge, von denen *genau einer* mit einer 1 belegt ist, und n Ausgänge.



I_0	I_1	I_2	I_3	Out_0	Out_1
1	0	0	0	0	0
0	1	0	0	1	0
0	0	1	0	1	1
0	0	0	1	0	1

Abbildung 6.10: 4–to–2–Encoder

6.2.4 Multiplexer

Multiplexer werden in der Literatur oft auch als Selektoren bezeichnet, d.h. sie wählen Signale aus.
Ein Multiplexer hat mehrere Eingänge und einen Ausgang, wobei dieser einem der Eingänge entspricht, der durch eine Steuerung ausgewählt wird.

Beispiel 6.4 *(2–Eingaben–Multiplexer):* Als Beispiel wollen wir den 2–Eingaben–Multiplexer betrachten.
Darstellung:

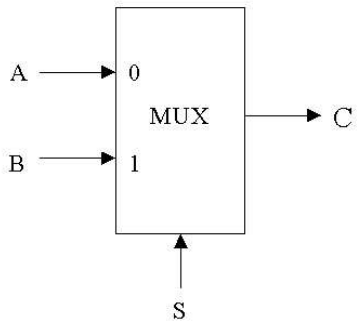


Abbildung 6.11: MUX

Als Funktionstabelle ergibt sich vereinfacht:

S	Out
0	A
1	B

oder in ausführlicher Form (C entspricht hier Out):

A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

A und B stehen dabei für Eingabewerte, S steht für einen Selektor, d.h. Steuerwert (Control value). Dieser Steuerwert bestimmt, welcher der Eingabewerte zum Ausgabewert wird. Die Funktionalität des 2–Eingaben–Multiplexers kann als Boolesche Funktion beschrieben werden:

$$C = (A \cdot \overline{S}) + (B \cdot S)$$

Diese Boolesche Funktion ist 3-stellig, d.h. wir haben 2 Nutz- und 1 Steuereingabe. Durch Verknüpfung von Gattern kann er wie folgt realisiert werden:

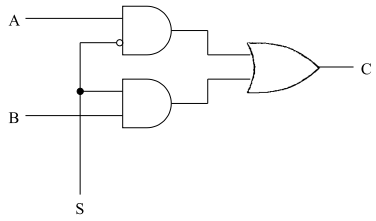


Abbildung 6.12: 2–Eingaben–Multiplexer

Wir wollen nun den 2–Eingaben–Multiplexer verallgemeinern. Multiplexer können mit jeder beliebigen Anzahl von Eingaben realisiert werden. Im Falle von n Eingaben werden jedoch $\lceil \log_2 n \rceil$ Selektoreingaben benötigt.

Bei einem 4-Eingaben-Multiplexer haben wir folgendes Prinzip (vgl. Abbildung 6.13):

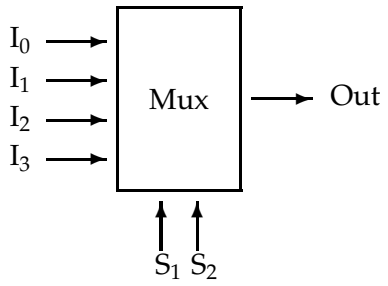


Abbildung 6.13: 4-Eingaben-Multiplexer

Die Funktionstabelle lautet in Kurzform:

S_1	S_2	Out
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Die Funktionalität kann durch folgende Boolesche Funktion beschrieben werden:

$Out = I_0 \cdot \overline{S_1} \cdot \overline{S_2} + I_1 \cdot \overline{S_1} \cdot S_2 + I_2 \cdot S_1 \cdot \overline{S_2} + I_3 \cdot S_1 \cdot S_2$

Die zugehörige Boolesche Funktion ist dabei 6-stellig.

Bei einem 3-Eingaben-Multiplexer würden auch 2 Steuereingänge benötigt werden, wobei eine Bitmusterkombination der Steuerwerte nicht benötigt werden würde. Die zugehörige Boolesche Fünktion wäre 5-stellig.

Nun wollen wir den allgemeinsten Fall eines Multiplexers mit n Eingaben (im Sinne von Nutzeingaben) und $\lceil \log_2 n \rceil$ Steuer- oder Selektereingaben anschauen.

Dann besteht ein Multiplexer aus drei Teilen (vgl. Abbildung 6.14).

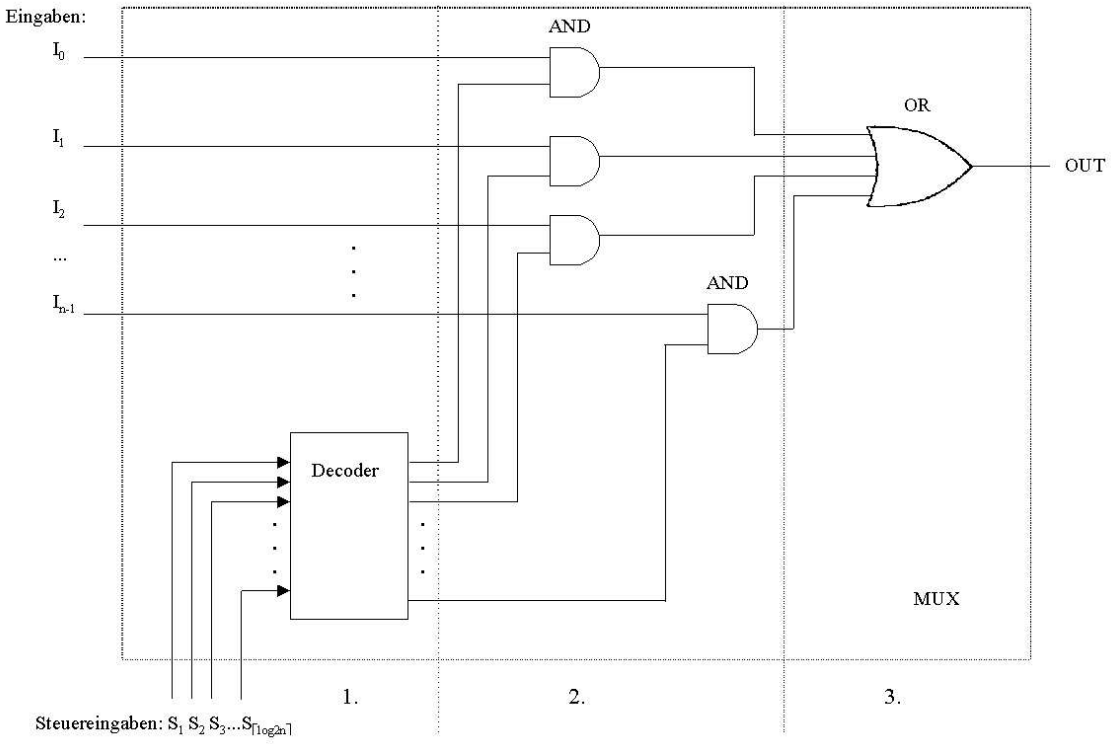


Abbildung 6.14: Die drei Teile des Multiplexers

1. Einem Decoder, der aus den $\lceil \log_2 n \rceil$ Selektoreingaben n Signale erzeugt, die jeweils einen anderen Eingabewert repräsentieren.

2. Einer Ansammlung von n AND–Gattern, die jeweils ein Signal vom Decoder mit einem Eingabesignal kombinieren.

3. Einem OR–Gatter mit n Eingängen (bzw. (n - 1) hintereinandergeschalteten OR–Gatter mit je zwei Eingängen), das die Ausgaben der AND–Gatter miteinander verknüpft.

Die zugehörige Boolesche Funktion wäre $n + \lceil \log_2 n \rceil$-stellig. Unter Umständen kann es aus Herstellersicht kostengünstiger sein, möglichst gleichartige Bauteile in Massenproduktion herzustellen und zu komplexeren Schaltnetzen zusammenzusetzen.

In diesem Sinne würde sich ein 4-Nutzeingaben-Multiplexer auch aus 3 einfachen Multiplexern mit je 2 Nutzeingaben zusammensetzen lassen.

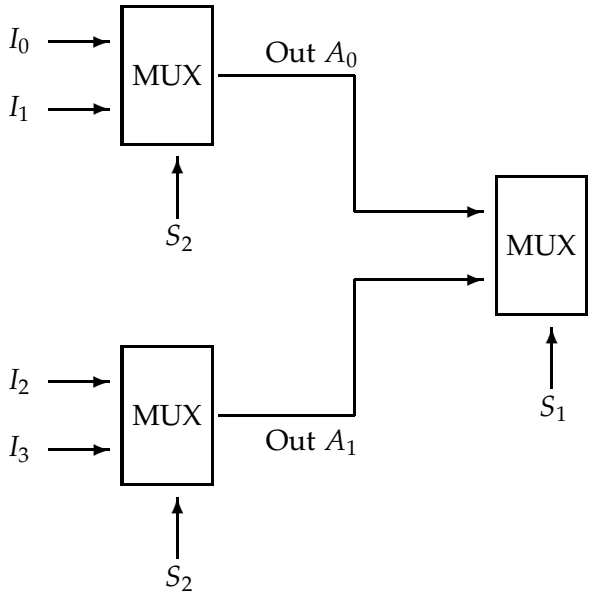


Abbildung 6.15: 4-Eingaben-Multiplexer durch 3 2-MUX realisiert

Zur Begründung der Richtigkeit wollen wir uns zunächst die Ausgänge der beiden erstgeschalteten Multiplexer anschauen:

S_2	Out A_0
0	I_0
1	I_1

S_2	OutA_1
0	I_2
1	I_3

Damit ergibt sich für den letzen Multiplexer

S_1	S_2	Out
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Damit erhalten wir die gleiche Funktionalität wie bei der vorangegangenen Realisierung.

Eine Variante dieses Multiplexers kann auch darin bestehen, nicht decodierte Selektorsignale zu verwenden. Dann muss die Anzahl der Eingabesignale sinnvollerweise gleich der Anzahl der Selektorsignale sein. Die zugehörige Boolesche Funktion wäre dann 2n-stellig.
Wie in diesem Beispiel des n–Eingaben–Multiplexers zu sehen ist, können logische Bausteine zusammengesetzt werden, um neue Schaltfunktionen zu ergeben.

Damit wollen wir zu Schaltnetzen übergehen.

6.3 ALU

6.3.1 Halb- und Volladdierer

Im folgenden wollen wir Schaltungen betrachten, mit denen später eine Arithmetisch–logische Einheit (ALU) für den von–Neumann–Rechner realisiert werden kann. Insbesondere müssen die grundlegenden arithmetischen Operationen auf binären Zahlen ausgeführt werden können.

Ziel ist es, die Addition von Zahlen zu realisieren. Dazu wollen wir noch einmal überlegen, welche Schritte ausgeführt werden, wenn eine Addition gemäß der "Schulmathematik" erfolgt:

$$\begin{array}{r} 102 \\ +\ 39 \\ \hline 141 \end{array} \quad \text{oder dual} \quad \begin{array}{r} 110.0110 \\ +\ \ 10.0111 \\ \hline 1000.1101 \end{array}$$

Besteht die Aufgabe darin, zwei 16–stellige Dualzahlen zu addieren, so ist das Ergebnis maximal 17–stellig.

Ansatz 1: Entwurf eines Schaltnetzes über die Normalformen

Hierzu benötigt man 16 * 2 = 32 Inputs und erhält bei der Addition 17 Outputs. Die Addition zweier 16–stelligen Zahlen ist also eine Abbildung Add: $B^{32} \rightarrow B^{17}$. Diese Schaltfunktion würde mittels 17 Boolescher Funktionen realisiert werden, die jeweils pro Ergebnisstelle Addstelle: $B^{32} \rightarrow B$ abbilden.

Wie realisiert man eine solche Boolesche Funktion?
Jede der 17 Funktionen hat $2^{32} = 4 * (2^{10})^3 \approx 4 * 1000^3 = 4 * 10^9$ Eingabemöglichkeiten. Unter der Annahme, dass die Hälfte dieser Eingaben die Ausgabe 1 erzeugen und diese Eingaben dann nach dem Darstellungssatz für Boolesche Funktionen als Mintermsumme realisiert werden müßten, bräuchte man ungefähr $2 * 10^9$ Minterme mit je 16 AND–Gattern mit je 2 Eingängen für jede der 17 Booleschen Funktionen, um die Schaltfunktion zu realisieren.

Daher wählt man einen anderen Ansatz.

Ansatz 2: Entwurf eines Schaltnetzes mittels elementarer Bausteine

Wir wollen intuitive, zugrundeliegende Gesetzmäßigkeiten der Addition ausnutzen:
Die letzte Stelle (die Einerstelle) hat keinen Übertrag zu berücksichtigen. Bei allen anderen Stellen kann zusätzlich zur Addition der beiden Ziffern noch ein Übertrag hinzukommen.

Schritt 1:
Man betrachte die letzte Stelle. Die Eingabewerte sind zwei Dualziffern x und y. Ausgabewerte sind ein Resultat R und ein Übertrag U für die nächste Stelle. Damit ergibt sich folgende Wahrheitstafel:

x	y	R	U
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Offensichtlich gilt: $R = \overline{x}y + x\overline{y} = x \nleftrightarrow y$ (exclusives oder) und $U = xy$.
Als Schaltnetz erhalten wir (vgl Abbildung 6.16):

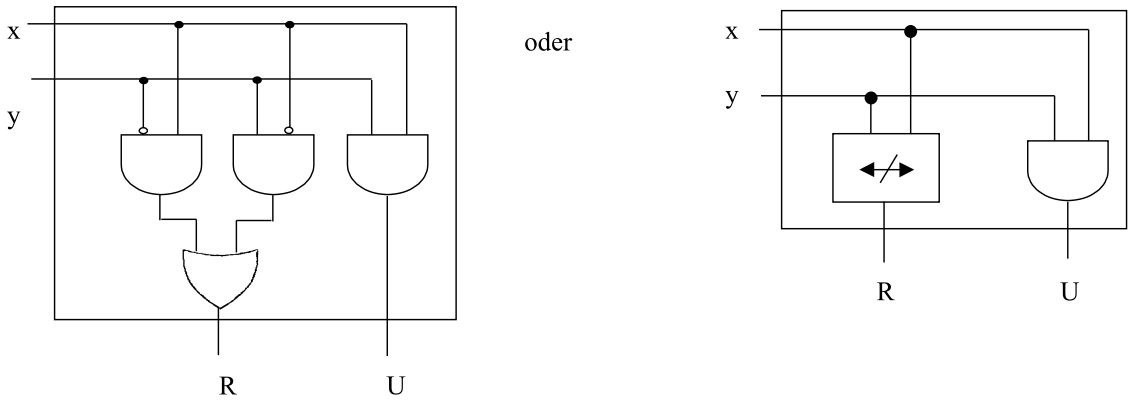


Abbildung 6.16: Schaltnetz des Halbaddierers

Dieses Schaltmodul betrachtet man als neuen, elementaren Baustein und nennt ihn Halbaddierer. Vereinfacht stellt man seine Funktionalität wie folgt dar:

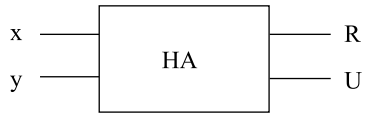


Abbildung 6.17: Halbaddierer

Schritt 2: Betrachte eine beliebige andere Stelle der Addieraufgabe. Jede Stelle ungleich der letzten hat 3 Eingaben: zwei duale Ziffern x, y und einen dualen Übertrag u. Ausgaben sind

wieder das Resultat R und der Übertrag U. Als Wahrheitstabelle ergibt sich:

x	y	u	R	U
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Offensichtlich gilt: $R = x \leftrightarrow y \leftrightarrow u$ d.h. bei ein oder drei Eingaben einer 1 ist auch das Ergebnis 1, und $U = xy + xu + uy = xy + (x \leftrightarrow y)u$.
Dieses Schaltnetz zur Addition zweier beliebiger Ziffern und eines Übertrags wird folgendermaßen realisiert:

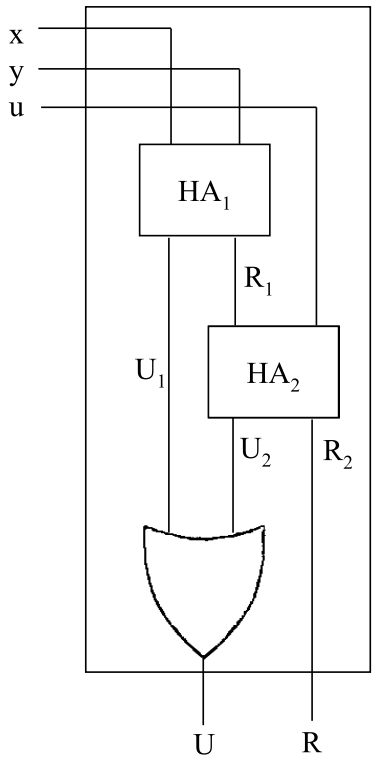


Abbildung 6.18: Schaltnetz des Volladdierers

Dabei ergeben sich folgende Zwischwerte:

$$\begin{aligned} U_1 &= xy \\ R_1 &= x \nleftrightarrow y \\ U_2 &= (x \nleftrightarrow y)u \\ R_2 &= x \nleftrightarrow y \nleftrightarrow u = R \\ U &= xy + (x \nleftrightarrow y)u \end{aligned}$$

Dieses Schaltnetz nennt man **Volladdierer**. Man betrachtet ihn ebenfalls als elementaren Baustein und stellt ihn wie in Abbildung 6.19 dar.

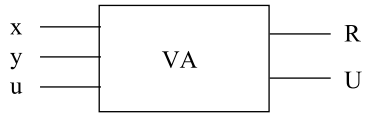


Abbildung 6.19: Symbol des Volladdierers

6.3.2 Ripple-Carry-Addiernetz

Nach diesen beiden Schritten können wir zur Addition n–stelliger Dualzahlen zurückkehren, die wir exemplarisch für n = 4 betrachten wollen. Seien zwei 4–stellige Dualzahlen $x_3x_2x_1x_0$ und $y_3y_2y_1y_0$ so gegeben, dass ihre Indizes den zugehörigen 2-er-Potenzen entsprechen. Dann läßt sich ihre Addition mittels dem Addiernetz in Abbildung 6.20 realisie-

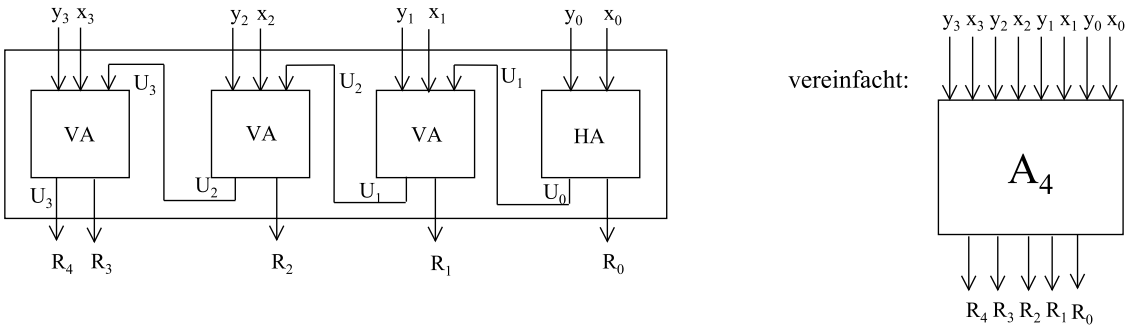


Abbildung 6.20: Addiernetz

ren.

Durch Hinzunahme weiterer Volladdierer läßt sich dieses Schaltnetz beliebig erweitern, so dass auch die Addition beliebiger n–stelliger Dualzahlen möglich wird.

Beispiel 6.5*:* Für die Addition zweier 16–stelliger Dualzahlen genügt es, anstelle der Realisierung von $2 * 10^9 * 17 = 34 * 10^9 = 3,4 * 10^{10}$ Mintermen lediglich einen Halb– und 15 Volladdierer zu verwenden.

Verallgemeinert kann man ein Addiernetz zur Addition zweier n–stelliger Dualzahlen auch mittels n Volladdierern realisieren. Dann muß nur sichergestellt sein, dass am u–Eingang des ersten Volladdierers stets eine 0 anliegt.

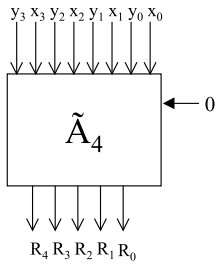


Abbildung 6.21: Ripple–Carry–Addiernetz mit 4 Volladdierern

Solche Addiernetze werden auch als **asynchrone Addiernetze** bezeichnet oder als **Ripple–Carry–Adder**, da bei derartigen Addiernetzen der endgültige Übertrag (von rechts nach links) durch das Schaltnetz rieselt.

Das Problem bei einer hohen Stellenanzahl der zu addierenden Dualzahlen besteht darin, dass viele Schaltstufen vorhanden sind. D.h. es dauert eine gewisse Zeit, bis die Volladdierer nacheinander ihre Operationen ausgeführt haben. Die Idee zur Beschleunigung eines Schaltnetzes ist daher eine Verringerung der Anzahl der Schaltebenen.

6.3.3 Carry-Look-Ahead-Addiernetz

Am Beispiel der dualen Addition wollen wir nun auf das Problem der *Beschleunigung von Schaltnetzen* eingehen.

Beispiel 6.6*:* Wann kann bei einem 4-stelligen Ripple-Carry-Addiernetz mit dem vollständigen Ergebnis gerechnet werden?

Nehmen wir eine Schaltzeit von 10 psec = 10^{-11} sec pro Gatter inkl. Negation an, so liefert ein Halbaddierer beide Outputs nach 30 psec, ein Volladdierer nach 70 psec. Nehmen wir beim 4-stelligen Ripple-Carry-Addiernetz den Worst Case an, dass z.B. bei der Addition

$$\begin{array}{r} 1111 \\ +\ 0001 \\ \hline 1.0000 \end{array}$$

ein Übertrag das gesamte Netz durchläuft, so ist i.A. erst nach dem Durchlaufen des Halb- und aller 3 Volladdierer mit dem vollständigen Ergebnis zu rechnen. Dies wäre nach $30 + 3 \cdot 70 = 240$ psec.

Diese Zeit soll nun durch Verwendung zusätzlicher Hardware verkürzt werden.

Die Idee zur Beschleunigung eines Addiernetzes ist die Verringerung der Anzahl der Schaltebenen durch Zusammenfassung von Bit-Gruppen z.B. der Größe $g = 4$.

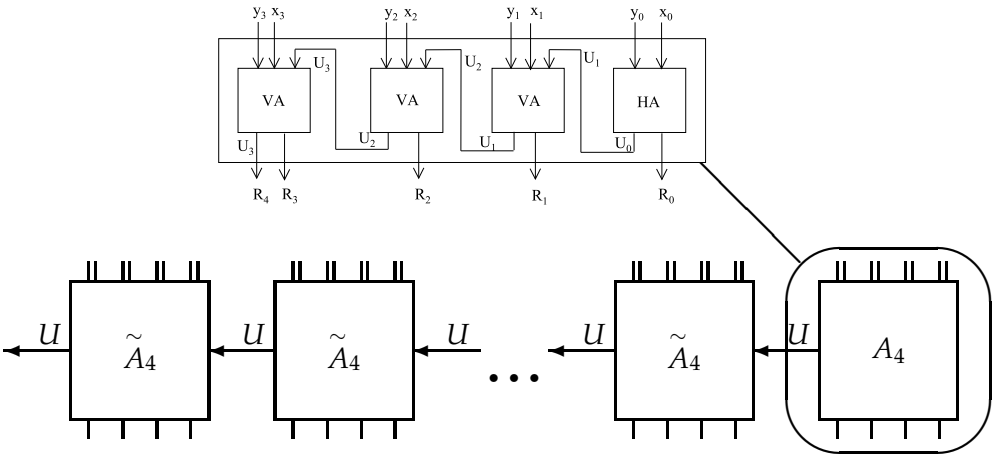


Abbildung 6.22: Prinzip eines n-stelligen Addierwerkes

Dabei besteht A_4 aus drei Volladdierern und einem Halbaddierer, $\tilde{A}_4$ besteht aus vier Volladdierern. Der zeitliche Engpaß besteht offensichtlich an den Übergangsstellen zwischen den einzelnen Modulen, d.h. bei U. Würde U also schneller zur Verfügung stehen, könnte im jeweils nächsten Modul bereits weiter gerechnet werden.

Dies führt zur Entwicklung einer Schaltung zur schnelleren Bestimmung von U, dem Carry-Look-Ahead-Adder, d.h. einem Addierwerk, das vorausschauend den Übertrag berechnet, bevor das eigentliche Ergebnis zur Verfügung steht.

Zum Entwurf eines solchen Netzes wollen wir zunächst die Frage stellen, warum ein Übertrag $U = 1$ bei der Rechnung

$$\begin{array}{r} x_3x_2x_1x_0 \\ +y_3y_2y_1y_0 \\ \hline Uz_3z_2z_1z_0 \end{array}$$

auftritt? Dies ist auf jeden Fall so, falls sowohl x_3 also auch y_3 jeweils den Wert 1 besitzen, d.h. falls $x_3y_3 = 1$.

Ferner ist dies der Falls, falls x_2 und y_2 jeweils eins sind und ferner x_3 oder y_3, d.h. falls $(x_3 + y_3)x_2y_2 = 1$.

Dann haben wir noch den Fall, dass x_1 und y_1 und in jedem Vorgängerpaar mindestens eine eins auftritt, also $(x_3 + y_3)(x_2 + y_2)(x_1 + y_1)x_0y_0 = 1$, oder mit bestehendem Übertrag aus dem Vormodul $(x_3 + y_3)(x_2 + y_2)(x_1 + y_1)(x_0 + y_0)U = 1$.

Damit ergibt sich

$$\begin{aligned} U &= x_3y_3 \\ &+ (x_3 + y_3)x_2y_2 \\ &+ (x_3 + y_3)(x_2 + y_2)x_1y_1 \\ &+ (x_3 + y_3)(x_2 + y_2)(x_1 + y_1)x_0y_0 \\ &+ (x_3 + y_3)(x_2 + y_2)(x_1 + y_1)x_0y_0 \\ &+ (x_3 + y_3)(x_2 + y_2)(x_1 + y_1)(x_0 + y_0)U \end{aligned}$$

Diese Formel sieht recht kompliziert aus, allerdings läßt sie sich sehr gut durch Parallelisierung in 3 Stufen berechnen.

1. Berechne alle Terme der Form $(x_i + y_i)$ und $(x_i * y_i)$
2. Multipliziere alle Faktoren einer Zeile
3. Berechne die Summe über alle Zeilen

Als Schaltung ergibt sich demnach das Addiernetz in 6.23.

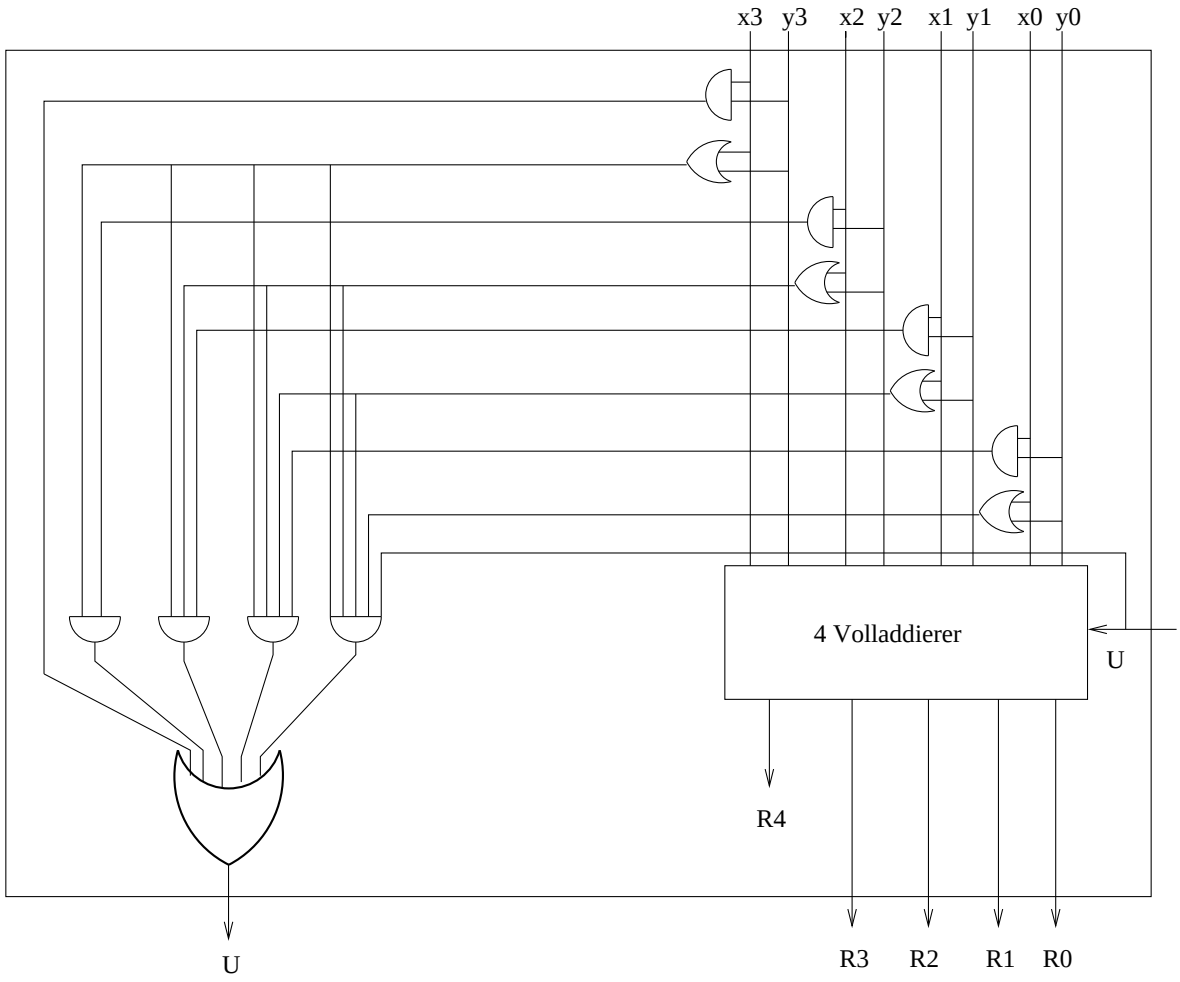


Abbildung 6.23: Carry-Look-Ahead-Addiernetz

Dabei wird vorausgesetzt, dass es auch sehr schnelle AND- bzw.OR-Gatter für mehr als 2 Inputs gibt. Mit der Annahme, dass diese ebenfalls 10 psec für die Ausführung benötigen, würde U nach 30 psec vorliegen, wenn jeweils pro Stufe die o.g. 10 psec veranschlagt werden. Damit ist der Carry-Look-Ahead-Adder mit 30 psec um den Faktor 8 schneller als der Ripple-Carry-Adder mit den 240 psec.

Es sei bemerkt, dass dieses dreistufige Prinzip auch mit Gruppengrößen $g > 4$ funktioniert. Allerdings werden die großen AND-und OR-Gatter dann noch mehr Inputs haben und zwar jeweils $g + 1$. Für höhere Stellenzahlen ist dies schwierig zu realisieren.

Allerdings kann durch eine andere Art der Modularisierung Abhilfe geschaffen werden.

6.3.4 Carry-Select-Addiernetze

Nun wollen wir die **Carry–Select–Addiernetze** betrachten. Durch redundante Hardware wird die Verarbeitung noch weiter beschleunigt.

Veranschaulichung durch ein Beispiel mit zwei 8–stelligen Dualzahlen.

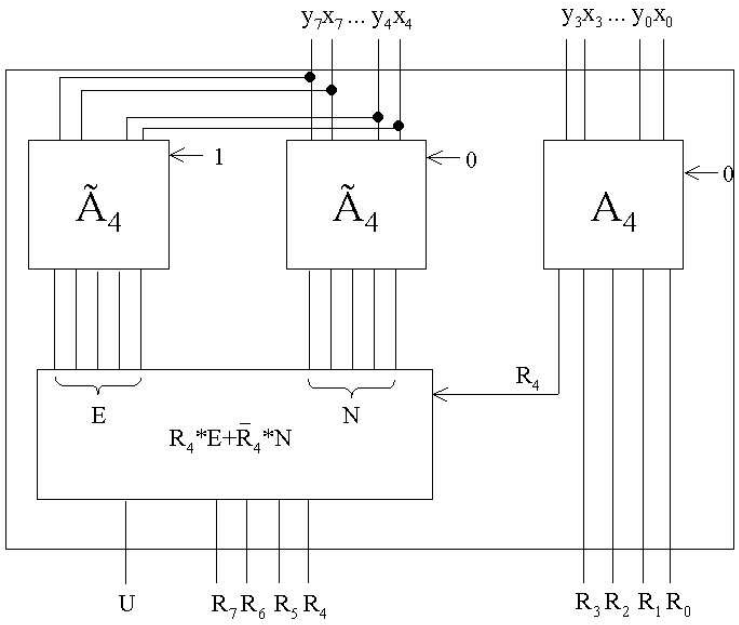


Abbildung 6.24: 8–stelliges Carry–Select–Addiernetz

Die niedrigwertige Hälfte der Inputoperanden ($x_3x_2x_1x_0$ und $y_3y_2y_1y_0$) wird normal mit einem Addiernetz für 4–stellige Dualzahlen verarbeitet. Die obere Hälfte wird dagegen zweimal addiert: Einmal für einen möglicherweise auftretenden Übertrag 0, einmal für einen möglicherweise auftretenden Übertrag 1. Steht dieser Übertrag nach der Berechnung der unteren 4 Stellen fest, so wird das entsprechende Ergebnis (E oder N) aus den Summen der oberen Operanden ausgewählt.

6.3.5 Carry-Save-Addiernetz

Für den Fall, dass mehr als 2 Summanden addiert werden müssen, wollen wir noch die **Carry–Save–Addiernetze** betrachten.

Idee: in einer ersten Stufe werden 3 Summanden stellenweise zu einer Summe mit Übertrag zusammenaddiert, in weiteren Stufen kommt zu der Summe und dem Übertrag je ein neuer Summand hinzu.

Beispiel 6.7 (Carry–Save–Addiernetz): 4 je vierstellige Summanden: x = 0101, y = 0011, z = 0100, w = 0001

x	0101		Summe	0010		Addiernetz :	
y	0011		Ubertrag	1010		n S	1001
z	0100	$\Rightarrow$	w	0001	$\Rightarrow$	n U	0100
Summe	0010		neue Summe	1001		1101	
Ubertrag	1010		neuer Ubertrag	0100			

Für diese Berechnung verwendet man sogenannte Carry–Save–Addierbausteine (CSA), die drei Summanden auf zwei Summanden, d.h. eine "Summe" und einen Übertrag reduzieren.

Schreibweise:

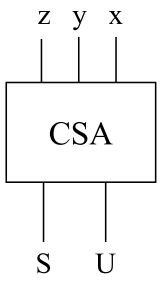


Abbildung 6.25: Carry–Save–Addiernetz

Damit kann man m je n–stellige Dualzahlen mit (m-2) CSA–Bausteinen realisieren, die jeweils durch nebeneinandergeschaltete Volladdierer realisiert werden. Am Ende wird ein Addiernetz benötigt.

Für 8 n–stellige Dualzahlen benötigt man folglich 6 CSA–Bausteine und ein Addiernetz (vgl. Abbildung 6.26).

Bemerkung:
Eine andere Realisierung würde in Abbildung 6.27 bestehen. Dabei würde jedoch jeder Addierbaustein pro Dualstelle einen Ausführungsschritt benötigen. Damit stehen den $(4 + n)$-Stufen beim Carry-Save-Addiernetz die $(3 \cdot n)$-Stufen bei klassischen Ripple-Carry Addiernetzen gegenüber. Bei $n = 8$-stelligen Dualzahlen würde sich demnach mit 24 zu 12 Stufen ein deutlicher Vorteil bei den Carry-Save-Addiernetzen ergeben.

6.4 Grundlagen der Schaltnetze

Werden logische Bausteine zusammengeschaltet, so spricht man von **Schaltnetzen**.

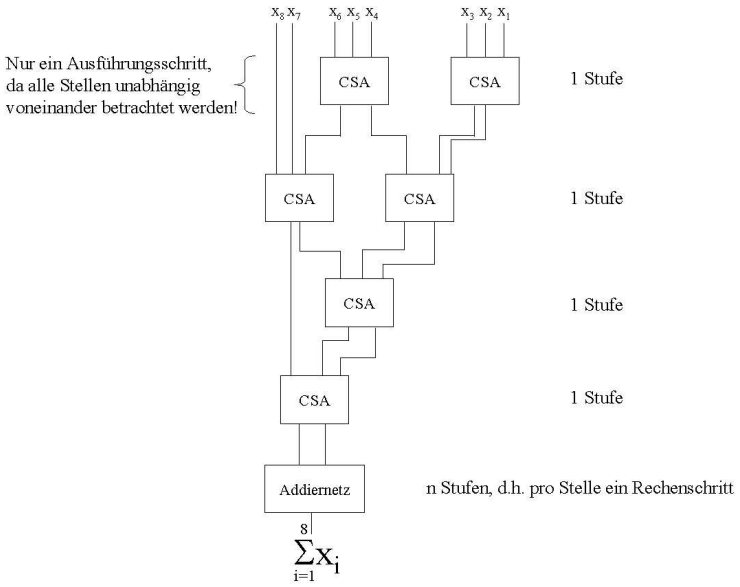


Abbildung 6.26: Beispiel für 8 n–stellige Dualzahlen

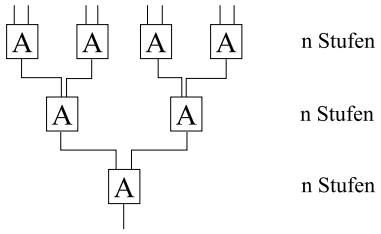


Abbildung 6.27: n Stufen

Formaler:
Ein Schaltnetz ist ein **gerichteter zyklusfreier Graph (Directed Acyclic Graph, DAG)**, dessen Knoten aus logischen Bausteinen, Eingaben und Ausgaben bestehen. Eingaben sind die Punkte, in die keine Kante hineinführt, Ausgaben die Punkte, aus denen keine Kanten herausführen.

Uns interessieren im folgenden nicht die technische Realisierung und Details solcher Bausteine, sondern wir gehen davon aus, dass sie als Bausteine zur Verfügung stehen. In diesem Sinne wollen wir uns mit ihrem *logischen Aufbau* entsprechend der zugehörigen Booleschen Funktionen beschäftigen. Außerdem wollen wir beachten, dass jedes Schaltelement Kosten verursacht (Material, Zeit für Ausführung, Platz,...). Diese Kosten sollen so niedrig wie möglich gehalten werden.

Bezüglich der Kosten wollen wir zwei Aspekte beachten: Zeit und Raum.

Geschwindigkeit (Zeit):
Jedes Gatter hat eine gewisse Verzögerung bzw. Schaltzeit, die von der Aktivierung des Inputs bis zum Bereitstellen des Outputs vergeht. Diese Zeit kann für einzelne Gatter im Bereich weniger Picosekunden liegen. Die Verzögerung eines Schaltnetzes hängt jedoch davon ab, wieviele Stufen von Gattern die Eingabesignale insgesamt zu durchlaufen haben. Beispielsweise wird eine dreistufige Schaltung immer langsamer sein, als eine zweistufige Schaltung. Daher wird man versuchen, Schaltungen mit möglichst wenigen Stufen zu realisieren.
Größe (Raum):
Die Herstellungskosten eines Schaltnetzes sind im wesentlichen proportional zur Anzahl der verwendeten Gatter, so dass eine geringe Gatterzahl das Ziel ist. Die Anzahl wiederum beeinflußt die Chip–Fläche und damit die Schaltgeschwindigkeit. Je größer andererseits ein Chip wird, desto höher ist auch die Wahrscheinlichkeit, dass sich bei seiner Herstellung Produktionsfehler einschleichen. Außerdem erfordern große Chips längere Verbindungen zwischen ihren Schaltelementen.

Zu bemerken ist allerdings, dass eine geringe Stufenanzahl nicht immer mit einer minimalen Größe zu vereinbaren ist.

6.4.1 Normalformen von Schaltfunktionen

Ausgangssituation:

Sei $f : B^n \to B,\ n \geq 1$ eine n–stellige Boolesche Funktion. Dann kann f durch eine Wahrheitstafel mit 2^n Zeilen dargestellt werden, wobei die Argumente so angeordnet seien, dass in der i–ten Zeile (beachte: $0 \leq i \leq 2^n - 1$) gerade die *Dualdarstellung von i* steht. i heißt dann ein *Index zu f*.

Beispiel 6.8*:* Betrachte n = 3, d.h. $f : B^3 \to B$. Eine boolesche Funktion sei gegeben durch:

i	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Definition 6.3 (einschlägiger Index): *Sei i eine Zeilennummer der Wahrheitstafel, und sei $i_1, \ldots, i_n$ die Ziffernfolge der Dualdarstellung von i. Dann heißt i* ***einschlägiger Index*** *zu f, falls $f(i_1, \ldots, i_n) = 1$ ist.*

Beispiel 6.9*:* In oben genannter Wahrheitstafel sind 3, 5 und 7 einschlägige Indizes zu f, alle anderen Indizes sind nicht einschlägig.

Die Menge aller einschlägigen Indizes zu einer n-stelligen Booleschen Funktion f wird mit I bezeichnet, dabei gilt: $I \subseteq \{0, 1, \ldots, 2^n - 1\}$.
Das heißt: zu einer n-stelligen Booleschen Funktion gibt es maximal 2^n einschlägige Indizes.
Und: Eine n-stellige Boolesche Funktion hat genau dann 2^n einschlägige Indizes, wenn es die Konstante 1 ist.
Umgekehrt hat eine Boolesche Funktion genau dann gar keine einschlägigen Indizes, wenn es die Konstante 0 ist.

Definition 6.4 (Minterm): *Sei i ein Index von $f : B^n \to B$ und $(i_1, \ldots, i_n)$ die Dualdarstellung von i. Dann heißt die Funktion $m_i : B^n \to B$* ***i–ter Minterm*** *von f, falls*

$$m_i(x_1, \ldots, x_n) := {x_1}^{i_1} * {x_2}^{i_2} * \ldots * {x_n}^{i_n}$$

mit

$$x_j^{i_j} := \begin{cases} x_j & falls\ i_j = 1\ und \\ \overline{x_j} & falls\ i_j = 0. \end{cases}$$

Beispiel 6.10*:* Im oben genannten Beispiel sind die Minterme wie folgt definiert:

$$m_0 := m_0(x_1, x_2, x_3) = \overline{x_1} * \overline{x_2} * \overline{x_3}$$

$$m_1 := m_1(x_1, x_2, x_3) = \overline{x_1} * \overline{x_2} * x_3$$

$$m_2 := m_2(x_1, x_2, x_3) = \overline{x_1} * x_2 * \overline{x_3}$$

$$m_3 := m_3(x_1, x_2, x_3) = \overline{x_1} * x_2 * x_3 \text{ u.s.w.}$$

Aus $m_3 = \overline{x_1} * x_2 * x_3$ folgt, dass der Minterm m_3 genau dann den Wert 1 annimmt, wenn das Argument $(x_1, \ldots, x_n)$ die Dualdarstellung von i annimmt, d.h.

$$m_3 = 1 \Leftrightarrow (\overline{x_1} = 1 \wedge x_2 = 1 \wedge x_3 = 1) \Leftrightarrow (x_1 = 0 \wedge x_2 = 1 \wedge x_3 = 1)$$

und 011 ist die Dualdarstellung von 3.

Damit können wir Minterme zur Beschreibung Boolescher Funktionen verwenden.
Es gilt der **Darstellungssatz für Boolesche Funktionen:**

Jede Boolesche Funktion $F : B^n \to B$ ist eindeutig darstellbar als Summe der Minterme ihrer einschlägigen Indizes.

Oder mit anderen Worten:
Ist $I \subseteq \{0, ..., 2^n - 1\}$ die Menge der einschlägigen Indizes von f, so gilt $f = \sum_{i \in I} m_i$ und keine andere Mintermsumme stellt f dar.

Definition 6.5 (Disjunktive Normalform): *Diese Darstellung $f = \sum_{i \in I} m_i$ nennt man auch **Disjunktive Normalform (DNF)** der booleschen Funktion $f : B^n \to B$.*

Beispiel 6.11*:* Betrachte die oben durch die Wahrheitstafel gegebene Funktion f. Dann lautet die Disjunktive Normalform:

$$f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_3$$

Bemerkung:
Ist f = 0, so existiert keine DNF.

Aus dem Darstellungssatz für Boolesche Funktionen ergibt sich insbesondere, dass jede n–stellige Boolesche Funktion mittels der Operationen +, * und – bzw. praktisch durch AND–, OR– und NOT–Gatter realisierbar ist. Ein solches System von Booleschen Funktionen (d.h. hier +, *, –) mittels derer sich jede Boolesche Funktion darstellen läßt, heißt **funktional vollständig**. Mittels der de Morganschen Regeln läßt sich leicht überlegen, dass auch +, – und *, – vollständig sind, denn $x * y = \overline{\overline{x} + \overline{y}}$ und $x + y = \overline{\overline{x} * \overline{y}}$.

Übungsaufgabe:
Zeige, dass NAND = $\overline{x * y}$ bzw. NOR = $\overline{x + y}$ als jeweils eindeutige Systeme funktional vollständig sind.

Nun wollen wir eine andere Normalform kennenlernen. Dazu benötigen wir zunächst wieder Terme.

Definition 6.6 (Maxterm): *Sei i ein Index von $f : B^n \to B$, und sei m_i der i–te Minterm von f. Dann heißt die Funktion $M_i : B^n \to B$ **i–ter Maxterm** von f, falls $M_i(x_1, ..., x_n) := \overline{m_i(x_1, ..., x_n)}$.*

Beispiel 6.12*:* Für unser Beispiel ergibt sich mittels der de Morganschen Regeln:
$M_3 = \overline{m}_3 = \overline{\overline{x}_1 x_2 x_3} = x_1 + \overline{x}_2 + \overline{x}_3$
$M_4 = \overline{m}_4 = \overline{x_1 \overline{x}_2 x_3} = \overline{x}_1 + x_2 + \overline{x}_3$

Ein Maxterm M_i nimmt genau dann den Wert 0 an, wenn das Argument $(x_1 ... x_n)$ die Dualdarstellung von i ist. Damit ist jede Boolesche Funktion $f : B^n \to B$ eindeutig als Produkt der Maxterme ihrer nichteinschlägigen Indizes darstellbar.

Diese Darstellung heißt auch **Konjunktive Normalform (KNF)**.

Beispiel 6.13*:* $f(x_1 x_2 x_3) = M_0 * M_1 * M_2 * M_4 * M_6$
$= (x_1 + x_2 + x_3) * (x_1 + x_2 + \overline{x_3}) * (x_1 + \overline{x_2} + x_3) * (\overline{x_1} + x_2 + x_3) * (\overline{x_1} + \overline{x_2} + x_3)$

Wir haben gelernt, dass eine Funktion, die eine BlackBox realisieren soll, stets als Summe von Produkten (sum–of–products–form) in der DNF und als Produkt von Summen (product–of–sums–form) in der KNF dargestellt werden kann.

Interessanter ist für uns die Darstellung der Summe von Produkten in der DNF. Jede logische Funktion kann so dargestellt werden, indem man von der Wahrheitswertetafel ausgeht und dabei die Eingabekombinationen als Produkte betrachtet, die die Ausgabe 1 ergeben.

Diese Herangehensweise führt zu einer Zwei–Level–Repräsentation, die im folgenden betrachtet werden soll.

6.4.2 Programmable Logic Array (PLA)

Im folgenden soll eine Technik betrachtet werden, die auf folgender Idee beruht:

Für verschiedene Schaltfunktionen soll ein universell einsetzbarer Einheitsbaustein entworfen werden, der eine möglichst homogene Struktur hat und für unterschiedlichste Anwendungen einsetzbar ist.

Ein solcher Baustein wird etwas aufwändiger sein als eine Schaltung, die nur hinsichtlich einer Anwendung entworfen wurde. Duch den großen Bedarf eines solchens Bausteins wird dieser sich jedoch konstengünstig in Massenproduktion herstellen lassen. Außerdem wird er einen übersichtlichen Aufbau haben und sehr wartungsfreundlich sein. D.h. Herstellung, Test und Betrieb lassen sich einfach realisieren.

Ein solcher Einheitsbaustein ist das Programmierbare Logische Feld (Programmable Logic Array, PLA). Es hat prinzipiell folgenden Aufbau, siehe Abbildung 6.28.

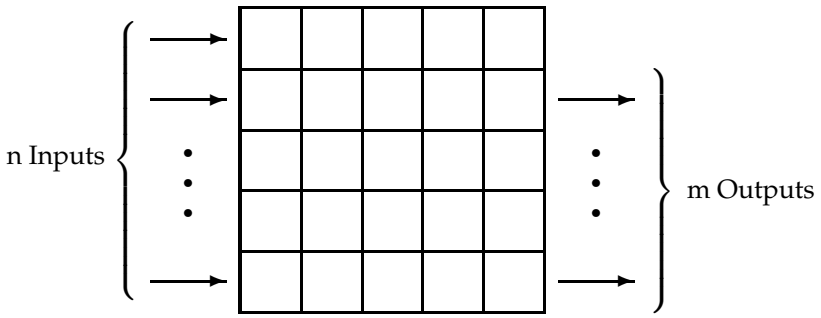


Abbildung 6.28: Grundprinzip eines PLA's

Intern ist ein PLA gitterförmig verdrahtet, wobei sich aus jedem Kreuzungspunkt von 2 Drähten ein einheitlich formatierter Baustein befindet, siehe Abbildung 6.29 auf der nächsten Seite

Die Funktionalität solcher Gitterpunkte kann sich unterscheiden. Wir wollen im folgenden 4 verschiedene Typen betrachten, die mit 0,1,2 und 3 durchnummeriert werden.

Alle vier Typen haben eine Eigenschaft gemeinsam: Mindestens einer der beiden Inputs wird am Ausgang unverändert weitergegeben. Alle Bausteine sind leicht durch Gatter beschreibbar.

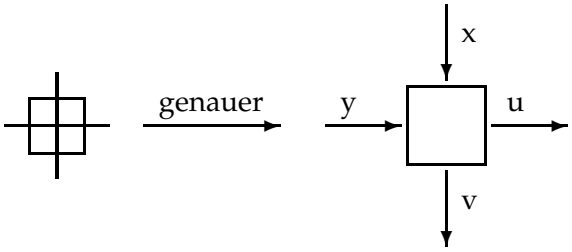


Abbildung 6.29: Gitterpunkt eines PLA's

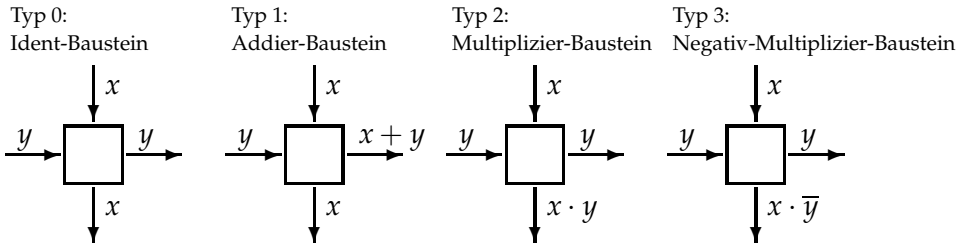


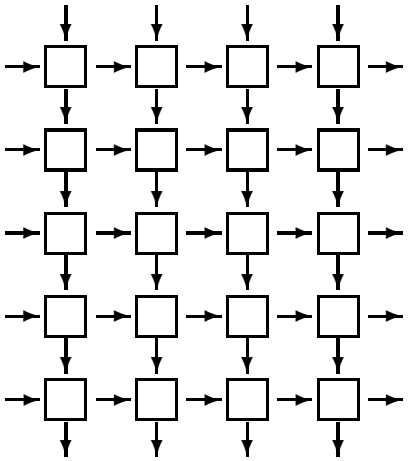
Abbildung 6.30: verschiedene Gitterpunkt-Typen

Beispiel 6.14*:* Mittels eines PLA's soll die Schaltfunktion

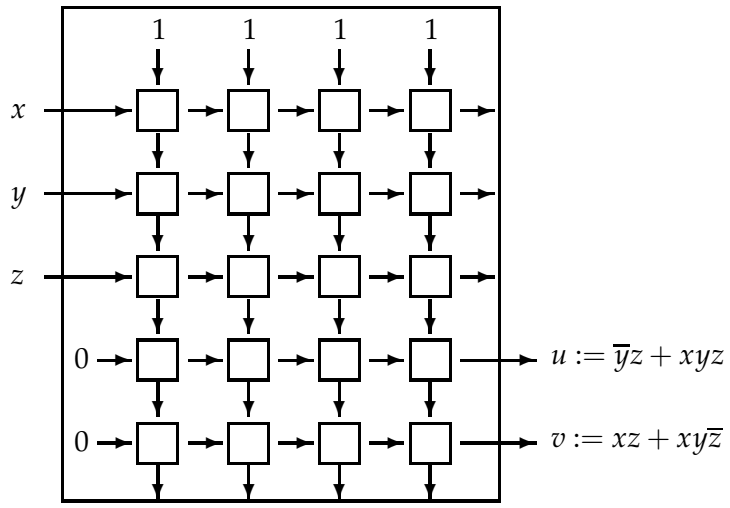
$$f : B^3 \to B^2 \text{ mit } f(x,y,z) = (\overline{y}z + xyz, xz + xy\overline{z})$$

realisiert werden.

Dazu betrachten wir ein PLA mit $n = 5$ Inputs an der linken Seite, $m = 5$ Outputs an der rechten Seite und $k = 4$ Spalten.

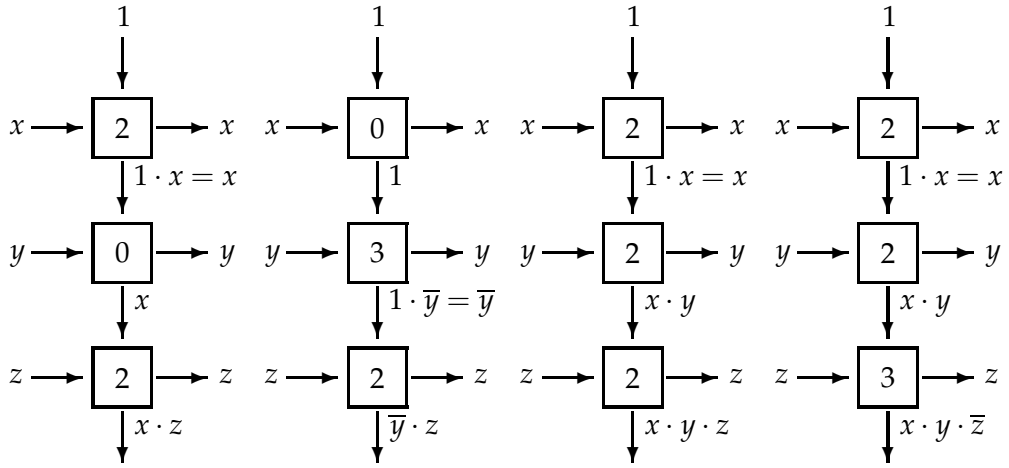


Da die gegebene Schaltfunktion 3-stellig ist und damit nur drei der fünf Inputs auf der linken Seite benötigt werden, „sperren" wir die beiden unteren Inputs durch Anlegen einer 0. Die vier oberen Inputs „neutralisieren" wird durch Anlegen von Einsen. Von den Outputs benötigen wir genau zwei, und zwar die unteren beiden auf der rechten Seite.

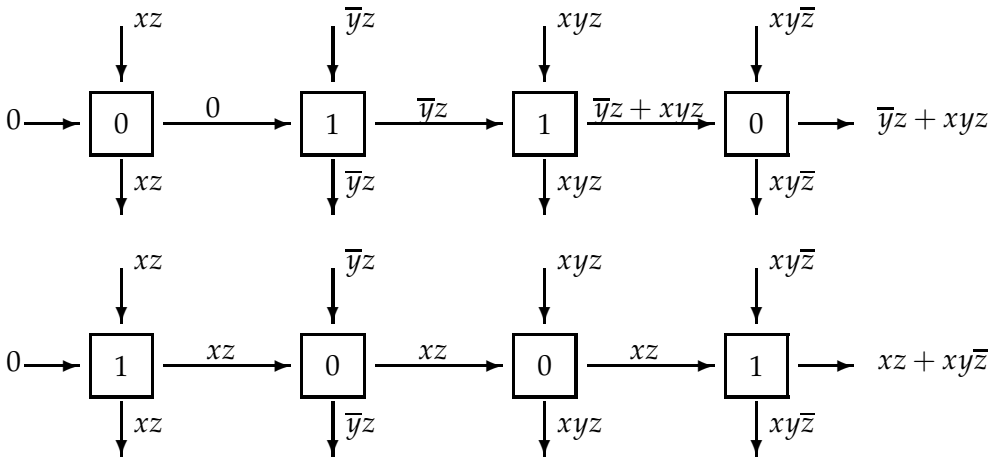


Die vier Spalten des Felds werden nun zur Erzeugung der vier Produkte $\overline{y}z, xyz, xz$ sowie $xy\overline{z}$ verwendet. Diese sind dann noch geeignet zu summieren und an die Ergebnis–Ausgänge weiterzuleiten.

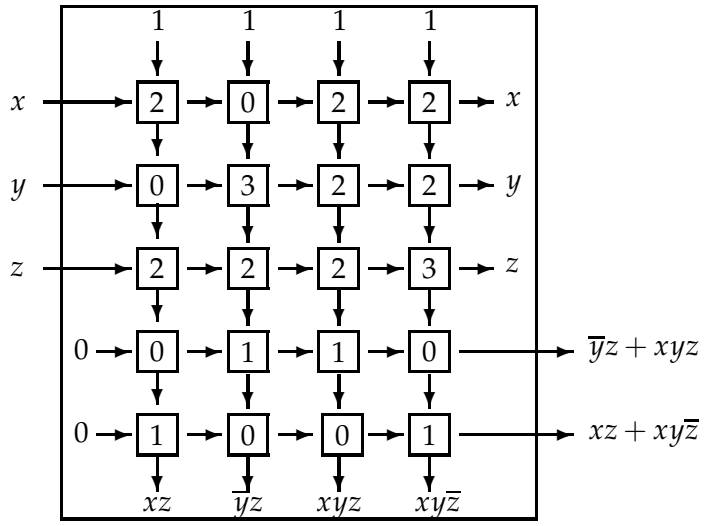
Die einzelnen Produktterme erhält man dabei wie folgt:



Die geeignete Zusammenfassung der Produktterme zu den Booleschen Funktionen u und v ergibt sich durch:



Damit ergibt sich zur Realisierung des angegebenen Beispiels folgende PLA:



Diese PLA läßt sich verkürzt durch folgende Matrix beschreiben:

2	0	2	2
0	3	2	2
2	2	2	3
0	1	1	0
1	0	0	1

Allgemein ergibt sich folgendes Prinzip: Ein PLA hat eine Menge von Inputs (bzw. komplementären Werten) als Eingabe und zwei Stufen von Logiken: ein Feld von AND's, das eine Menge von Produkten generiert, d.h. die Minterme, und als zweite Stufe ein Feld von OR's.

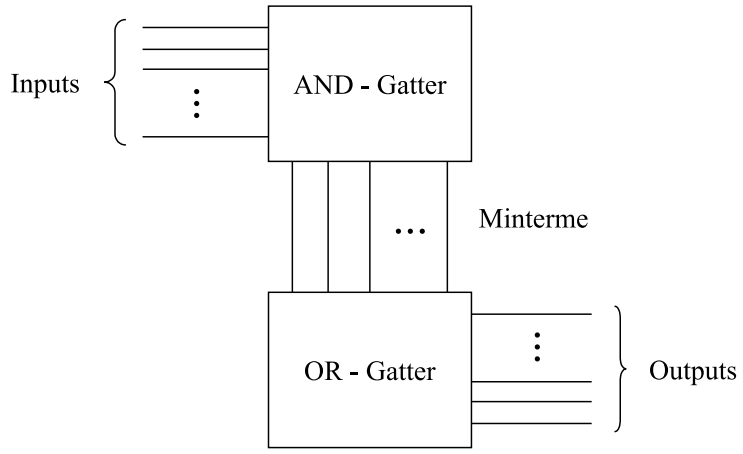


Abbildung 6.31: PLA

Dabei läßt sich eine typische Systematik erkennen: Im Feld der AND-Gatter werden nur Bausteine vom Typ 0, 2 oder 3 verwendet, im Feld der OR-Gatter nur Bausteine vom Typ 0 oder 1.

Dabei vereinbaren wir, daß „von oben“ Einsen eingespeist werden, bei den OR-Gattern von links Nullen als Input genutzt werden.

Durch geeignete Eintragung in eine PLA-Matrix kann jede Schaltfunktion $f : B^r \to B^s$ realisiert werden. Dazu benötigt man r Zeilen im Feld der AND-Gatter und s Zeilen im Feld der OR-Gatter. Ist f in disjunktiver Form gegeben und kommen in den Summen insgesamt t verschiedene Produktterme vor, so muß das PLA t Spalten haben oder mehr.

Beachte: Bei f wird im allgemeinen von einer disjunktiven Form ausgegangen, *nicht* von einer disjunktiven Normalform. Eine disjunktive Normalform würde zum gleichen Ergebnis führen, jedoch unter Umständen etwas aufwändiger sein.

Beispiel:

$$u = (x + \overline{x})\overline{y}z + xyz = \underline{x\overline{y}z} + \overline{x}\overline{y}z + \mathrm{xyz}$$

$$v = x(y + \overline{y})z + xy\overline{z} = \mathrm{xyz} + \underline{x\overline{y}z} + xy\overline{z}$$

Damit ergibt sich das PLA:

2	2	2	3
2	2	3	3
2	3	2	2
1	0	1	1
1	1	1	0

Aus der Sicht der Historie ist die Anzahl der Produktterme möglichst zu minimieren, da PLA's beschränkte Möglichkeiten bieten. In der zweiten Hälfte der 70er Jahre wurde in den USA häufig das PLA vom Typ DM 7575 der Firma National Semiconductor verwendet. Dies

hatte 14 Inputs, 8 Outputs und 96 Spalten. Damit waren $14 \times 96 = 1344$ Bausteine in der AND-Ebene plus $8 \times 96 = 768$ Bausteine in der OR-Ebene nötig. Eine 14-stellige Boolsche Funktion kann bis zu 16.384 Minterme besitzen, jedoch waren nur 96 davon in den Spalten des DM 7575 generierbar. Trotzdem war durch Optimierung der Funktionen eine Fülle von Anwendungen abdeckbar.

Programmierung von PLAs

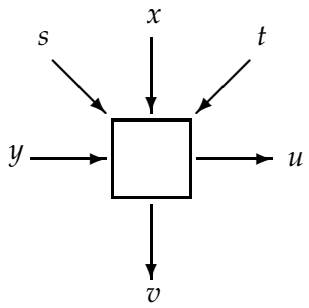
Wie können nun die Eintragungen in eine PLA-Matrix für eine konkret gegebene Schaltfunktion vorgenommen werden? Dazu bieten sich zwei Möglichkeiten an:

I. Hardwaremäßige Programmierung

Durch eine geeignete Ätz-Maske können Ätz-Muster entsprechend der Bausteintypen 1,2 oder 3 erzeugt werden. Die regelmäßige Geometrie eines PLAs erleichtert ein präzises Arbeiten dieser Art erheblich. Aber ein PLA ist dann nur noch für eine bestimmte Anwendung einsetzbar, da es extrem aufwändig ist, derartige Ätzungen rückgängig zu machen.

II. Softwaremäßige Programmierung

Die softwaremäßige Eintragung der Baustein-Typen ist wesentlich flexibler. Zunächst wollen wir uns einen einzelnen Baustein betrachten. 4 Bausteintypen lassen sich durch $\log_2 4 = 2$ Informationen unterscheiden. Deshalb versehen wir jeden Baustein mit zwei programmierenden Zuleitungen s und t, über die dann eingegeben werden kann, wie sich der Baustein verhalten soll.



Daraus ergibt sich folgende Funktionalität:

Bausteintyp	s	t	u	v
0	0	0	y	x
1	0	1	$x+y$	x
2	1	0	y	xy
3	1	1	y	$x\overline{y}$

mit

$$\begin{aligned} u &= y + \overline{s}tx \\ v &= \overline{s}x + s\overline{t}xy + stx\overline{y} \end{aligned}$$

Für eine Funktion $f : B^n \to B^m$ müssen dann im zugehörigen PLA mit $M = (n+m) * k$ Bausteinen alle M Bausteine mit je zwei Zuleitungen versehen werden.

Die dabei benötigten $2M$ binären Informationen können z.B. in einem sogenannten Festwertspeicher (Read-Only-Memory, ROM) abgelegt sein. Dieser Speicher ist dadurch charakterisiert, daß sein einmal z.B. durch Ätzen abgelegter Inhalt nicht mehr veränderbar ist, also nicht mehr durch einen neuen Inhalt überschrieben werden kann.

Ein ROM läßt sich also zur Programmierung eines PLAs verwenden, und durch Austausch des ROM läßt sich das PLA leicht umprogrammieren. Dadurch wird das PLA zu einem universell einsetzbaren Baustein.

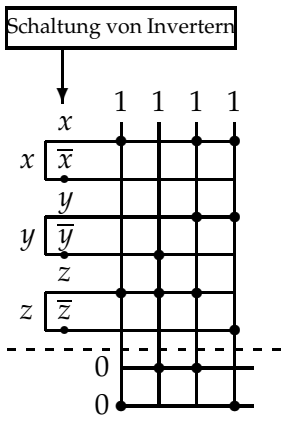
Beispiel 6.15*:* Beim DM7575-PLA mit 2112 Bausteinen benötigen wir ein ROM mit $2 \cdot 2112 =$ 4224 Bit = 528 Byte Speicherkapazität. Dies ist relativ viel, wobei die Preise für derartrige Speicher jedoch jährlich um bis zu 30% gefallen sind und noch weiter fallen.

Alternativ wollen wir noch den sogenannten **punktorientieren Ansatz** eines PLAs betrachten.

Die Idee ist folgende: wir wollen jeden Baustein mit nur einer Zuleitung steuern. Im Feld der OR-Gatter kommen nur Bausteine vom Tp 0 oder 1 vor, damit genügt eine Zuleitung bereits. Im Feld der AND-Gatter gibt es Bausteine vom Typ 0, 2 und 3. Diese werden dadurch reduziert, daß die Anzahl der Inputs verdoppelt wird – zu jedem Input wird auch noch der negierte Input hinzugenommen. Dadurch verdoppelt sich im Feld der AND-Gatter die Anzahl der Inputs und auch der Bausteine, aber das gesamt PLA läßt sich mit einer Zuleitung (einem Punkt) pro Baustein steuern.

Ohne Punkt gehen wir jeweil von Bausteinen des Typs 0 aus, mit Punkt im Feld der AND-Gatter von Bausteinen des Typs 2, im Felder der Or-Gatter von Bausteinen des Typs 1.

Damit ergibt sich für unser Beispiel:



6.4.3 Read Only Memory (ROM)

Im folgenden wollen wir ein ROM designen, das über n-Bit-Adreßleitungen insgesamt 2^n Speicherzellen der Größe m Bit ansprechen kann. Der sich ergebende Festwertspeicher kann aufgefaßt werden als eine $m \times 2^n$-Matrix, deren Spalten die Inhalte der Speicherzellen mit den Adressen $0 \ldots 2^n - 1$ sind.

Beispiel 6.16*:* $n = 3, m = 4$. Gegeben seien 8 Speicherzellen mit je 4 Bit wie folgt:

1	0	0	1	1	1	1	0
0	0	1	1	1	1	1	1
0	0	1	1	0	0	0	0
0	0	1	1	1	0	0	1

Nun soll durch Angabe der Adresse einer Speicherzelle der zugehörige Inhalt der Speicherzelle gelesen werden.

Dazu fassen wir das ROM als OR-Gatter eines PLAs auf und erweitern dieses durch Hinzunahme eines AND-Gatters der Größe $n \times 2^n$ zu einem PLA der Größe $(n + m) \times 2^n$, so daß die n Bit der Adresse der Input sind und die m Bit des Inhalts der Speicherzelle der Output sind. Das AND-Gatter dient dabei als Adreß-Decodierer, das OR-Gatter ist der eigentliche Speicherinhalt.

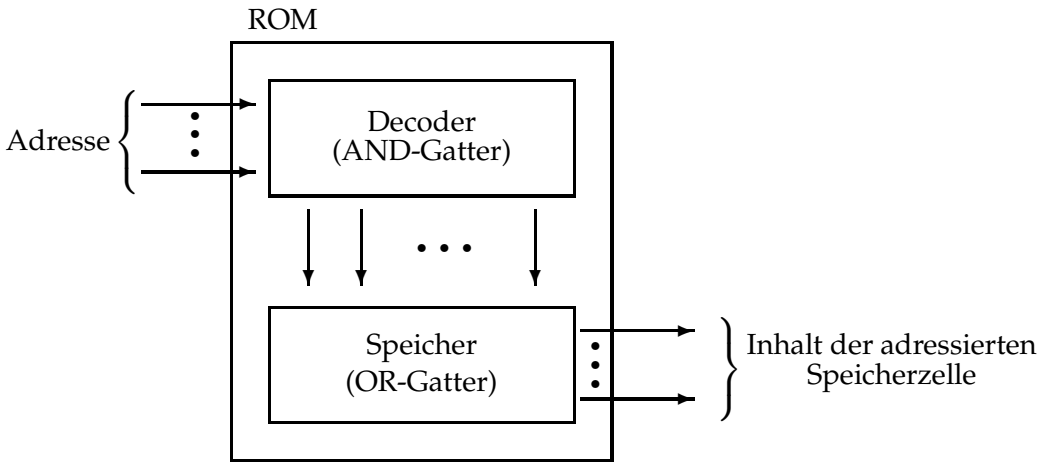
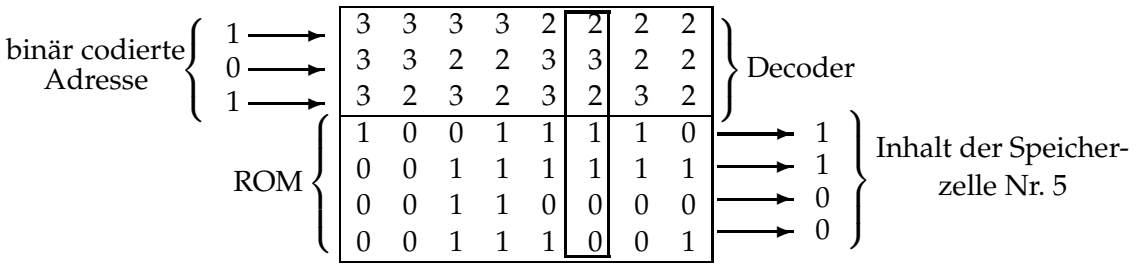
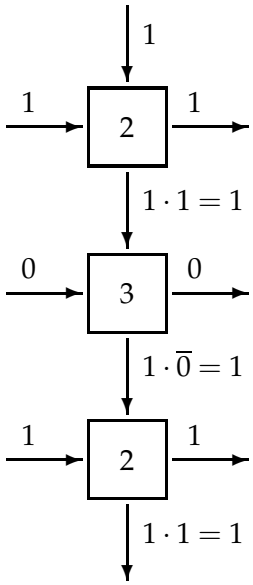


Abbildung 6.32: Anwendung eines PLA als ROM

Beispiel 6.17*:* Realisierung eines ROM mit dem in Beispiel 6.16 gegebenen Speicherinhalt, das den Inhalt der Speicherzelle mit Adresse 5 liest.



Durch Eingabe von $5_{10} = 101_2$ wird in der Spalte Nr. 5 der AND-Gatter eine Eins erzeugt und an den unteren Teil der PLA weitergeleitet. Im Decoder passiert folgendes:



In allen anderen Spalten wird mindestens ein Baustein den Wert 0 erzeugen, und damit ist das ganze Produkt gleich Null. Damit wird genau der Inhat der zugehörigen darunterliegenden Speicherzelle als Ergebnis ausgegeben. Alle anderen Inhalte der Speicherzellen werden mit dem Faktor 0 aufaddiert, d.h. spielen keine Rolle.

Das durch ein PLA realisierte ROM hat noch eine interessant Eigenschaft: während im OR-Gatter wie immer nur Nullen und Einsen auftreten, kommen hier im AND-Gatter nur Bausteine vom Typ 2 und 3 vor.

Damit kann das gesamte PLA mit einer Zuleitung pro Gitterpunkt auskommen, da in dieser speziellen Anwendung keine Bausteine vom Typ 0 im AND-Gatter benötigt werden.

6.4.4 Very Large Scale Integration (VLSI)

1946 wurde an der University of Pennsylvania der Rechner ENIAC (Electronic Numerical Integrator And Computer) fertiggestellt. Er besaß 18.000 Röhren, benötigte eine Standfläche von 300 m^2, wog 30t, hatte eine Leistungsaufnahme von 50.000 W und kostete 500.000 Dollar.

Anfang der 60er Jahre wurde die erste integrierte Schaltung (Integrated Circuit, IC) vorgestellt. Dabei werden alle Schaltungselemente wie Gatter, Delays und deren Verbindungsdrähte in *einem* gemeinsamen Herstellungsprozeß auf einem sogenannten *Chip* gefertigt.

Dieser rechteckig formatierte Chip ist in der Regel ein Siliziumplättchen, das sich zum Zeitpunkt der Herstellung auf einer größeren Siliziumscheibe, dem sogenannten *Wafer* befindet. Der Wafer hat einen Durchmesser zwischen 8 und 20 cm so daß bei einer Fläche von 20 bis 30 mm^2 pro Chip im allgemeinen mehrere hundert Chips gleichzeitig aus einem Wafer hergestellt werden können.

Mit fortschreitender Minituarisierung konnte eine immer größer Anzahl von Bauelementen auf einem Chip integriert werden. Je nach Anzahl der logischen Gatter pro Chip unterscheidet man vier Stufen der Integration, wobei die Grenzen je nach verwendeter Literatur auch differieren können.

SSI Small Scale Integration ≤ 10 Gatter pro Chip

MSI Medium Scale Integration 10 bis 10^2 Gatter pro Chip

LSI Large Scale Integration 10^2 bis 10^5 Gatter pro Chip

VLSI Very Large Scale Integration $> 10^5$ Gatter pro Chip

Manche Autoren verwenden als Fortsetzung des VLSI auch den Begriff ULSI (Ultra Large Scale Integration), jedoch verhindern physikalische Gesetzmäßigkeiten eine Miniaturisierung „ad infinitum“.

Zur technischen Realisierung

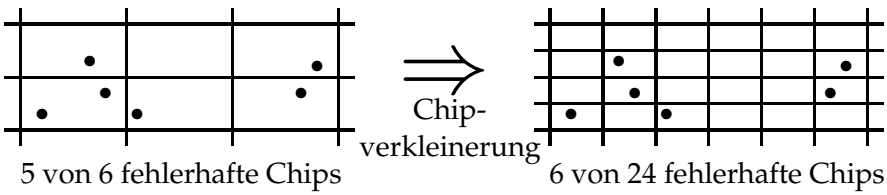
VLSI-Chips werden in unserer Betrachtungsweise so realisiert, daß die Integration von Gattern in einer PLA-ähnlichen Struktur erfolgt. Dann ist ein VLSI-Chip logisch gesehen ein rechteckiges Gitter, welches aus äquidistanten, aufeinander senkrecht stehenden Gitterlinien erzeugt wird.

Von solchen Gittern können mehrere übereinander liegen. Drähte, die zur Verbindung von Schaltelementen benötgt werden, verlaufen nur in horizontaler oder vertikaler Richtung. Durch die einzelnen Ebenen läßt sich dabei vermeiden, dass sich Drähte unerwünscht berühren. Drähte dürfen sich kreuzen, sofern sie in verschiedenen Ebenen verlaufen. Drähte in verschiedenen Ebenen dürfen ferner nicht stückweise direkt aufeinander liegen, da eine induktive Beeinflussung zu starke Störungen hervorrufen würde.

Zum Übergang von einer in eine andere Ebene gibt es sogenannte Kontakte.

Technologisch gesehen muß man bei der Positionierung eines Drahts auf einer Gitterlinie mit einer gewissen Streuung rechnen. D.h. es kann passieren, dass ein Draht nicht genau auf einer Linie angebracht wird, sondern mit einem Fehler, der durch ein bestimmtes Herstellungsverfahren bedingt ist. Ferner ist die Breite des Drahts von Bedeutung. Daraus resultiert die Fläche eines Chips und folglich auch die Höhe seiner Herstellungskosten.

Die für eine Chip benötigte Fläche ist auch noch aus einer anderen Sicht von Bedeutung. Durch Staubkörner bedingt liegt der Anteil fehlerbehafteter Chips pro Wafer bei bis zu 90%. Durch eine Verkleinerung der Chips sinkt der Anteil der fehlerbehafteten Teile.



Layout von VLSI-Schaltungen

Im folgenden wollen wir das Beispiel eines ROMs noch einmal aufgreifen. Die Realisierung kann auf 2 verschiedene Arten erfolgen. Zum einen hatten wir im vorangegangen Abschnitt bereits eine Realisierung mittels PLA kennengelernt, wobei die AND-Ebene die Funktionalität eines Decoders wahrnimmt und die OR-Ebene den eigentlichen Speicherinhalt beinhaltet.

Die Wirkungsweise der AND-Ebene war dabei wie folgt: Hat ein ROM 2^n Speicherplätze, so sind diese bekanntlich durch n Bits adressierbar. Die AND-Ebene muß erkennen, welcher Speicherinhalt gelesen werden soll. Soll beispielsweise die Adresse i angesteuert werden, so muß die i-te Spalte des ROM in der AND-Ebene gerade den Minterm erzeugen, der durch i binär codiert wird.

Statt dieser PLA-Struktur betrachten wir als alternatives Layout zur Generierung der 2^n Minterme in der AND-Ebene einen binären Baum.

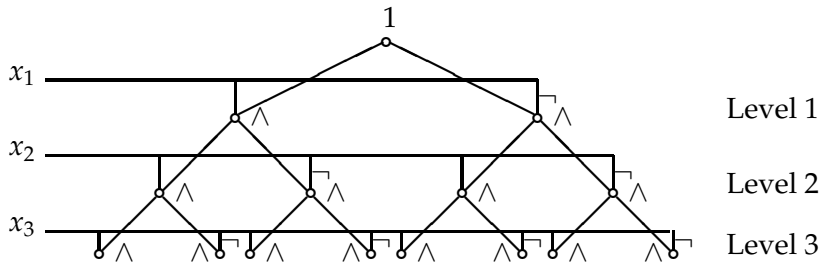


Abbildung 6.33: Realisierung als Minterm-Baum

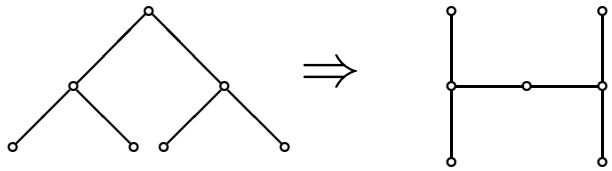
Beginnend mit x_1 und $\overline{x_1}$ in Level 1 generieren wir in Level 2 die Produkte $x_1x_2, x_1\overline{x_2}, \overline{x_1x_2}$ (von links nach rechts in dieser Reihenfolge) und auf Level 3 alle $2^3 = 8$ Minterme.

Diese Konstruktion ist unmittelbar auf mehr als 3 Level erweiterbar, so daß mit jedem binären Baum mit n Level 2^n Minterme generierbar sind.

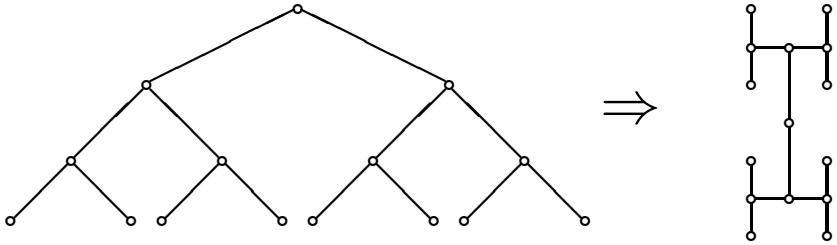
Ferner ist jede Boolesche Funktion $f : B^n \to B$ mit einem n-Level-Baum darstellbar, indem man auf dem n-ten Level alle Minterme der zugehörigen DNF auswählt und in einer OR-Funktion oder einem OR-Baum verknüpft – ganz analog zur OR-Ebene eines PLA, welcher die entsprechenden Minterme summiert.

Das PLA hatte den Vorteil, dass es auf einer gitterförmigen Struktur realiserbar war. Diesen Vorteil wollen wir auch bei den Minterm-Bäumen nutzen. Dazu werden die Punkte des Baumes so auf einem Gitter verteilt, dass sie ein Muster in der Form eines „H“ bilden.

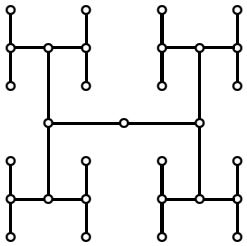
Für einen binären Baum mit 2 Level erhält man:



Ein binärer Baum mit 3 Leveln ergibt:



Bei 4 Level ergibt sich:



Wie der Minterm-Baum, so ist auch der H-Baum für beliebige Stellenzahlen verwendbar. Der Flächenbedarf ist dabei linear zur Anzahl der Blätter des Baums.

Prinzipiell gibt es zwei Herangehensweisen, Schaltnetze zu entwerfen:

- *Bottom–Up–Entwurf*:
  Komplexe Schaltungen werden aus elementaren Bausteinen sukzessive zusammengesetzt.

- *Top–Down–Entwurf*:
  Die Schaltung wird in wohldefinierte Teilaufgaben zerlegt; die Schaltung ergibt sich aus einer Realisierung der Komponenten.

Die Idee des Bottom–Up–Entwurfs findet bereits dann Anwendung, wenn aus den elementaren Gattern komplexere Schaltungen aufgebaut werden, die ihrerseits wieder als logische Bausteine betrachtet werden. Beispiele dafür sind die bereits betrachteten Decoder, Encoder und Multiplexer. Ein solches Zusammenfassen von Bausteinen oder Modulen zu neuen Bausteinen bezeichnet man als **Integration**.

6.5 Optimierung von Schaltnetzen

Hinsichtlich unserer Überlegungen am Anfang von Abschnitt 6.4 verursacht jedes Schaltnetz Kosten. Diese sollen so gering wie möglich gehalten werden.
Möchte man die Ausführungszeit einer Schaltung optimieren, so kann es günstig sein, zusätzliche Hardware einzufügen, um weniger Schaltstufen ausführen zu müssen (vgl. Carry–Select–Addiernetze).

Im folgenden wollen wir (quasikomplementär zur Zeit) Platz und Material einsparen, ohne das Verhalten eines Schaltnetzes zu ändern. Dazu gehen wir zu den Gesetzmäßigkeiten der Booleschen Algebra zurück und schauen uns an einem Beispiel eine Vereinfachung an:

Beispiel 6.18 *(Resolutionsregel):* Gegeben sei die Boolesche Funktion

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x_1} x_2 x_3 + x_1 x_2 x_3 \\ &= (\overline{x_1} + x_1) x_2 x_3 \quad \text{nach Komplementärgesetz gilt: } \overline{x} + x = 1 \text{ und } 1 \wedge x = x \\ &= x_2 x_3 \end{aligned}$$

Diese Vereinfachung ist auch unter dem Namen **Resolutionsregel** bekannt, d.h. kommen in einer disjunktiven Form zwei Summanden vor, welche sich in *genau einer* komplementären Variablen unterscheiden, so können diese beiden Summanden durch den ihnen gemeinsamen Teil ersetzt werden.

Beispiel 6.19 *(mehrfache Anwendung der Resolutionsregel):* Betrachte

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= x_1 \bar{x}_2 x_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_4 + x_1 x_2 x_3 x_4 + \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 + \bar{x}_1 \bar{x}_2 x_3 x_4 \\ &= x_1 \bar{x}_2 x_4 + x_1 x_3 x_4 + \bar{x}_2 \bar{x}_3 x_4 + \bar{x}_1 \bar{x}_2 x_4 \\ &= \bar{x}_2 x_4 + x_1 x_3 x_4 + \bar{x}_2 \bar{x}_3 x_4 \end{aligned}$$

Die mehrfache Anwendung beruht dabei auf der Gesetzmäßigkeit x+x=x, die das Verdoppeln von Summanden erlaubt.

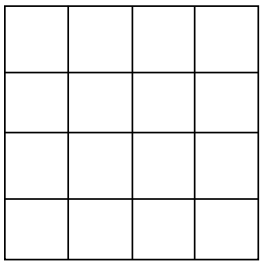
Um die Übersicht über alle möglichen Resolutionen zu behalten, wollen wir ein graphisches Verfahren betrachten, mit dem eine Schaltfunktion recht leicht vereinfacht werden kann.

6.5.1 Das Karnaugh–Diagramm

Das Karnaugh–Diagramm (engl.: Karnaugh map) einer booleschen Funktion $f : B^n \to B,\ n \in \{3, 4\}$ ist eine graphische Darstellung der Funktionstafel von f durch eine 0–1–Matrix z.B. der Größe 2×4 für n = 3 oder der Größe 4×4 für n = 4.

Beispiel 6.20*:* Das Karnaugh–Diagramm für die zuletzt betrachtete Funktion $f(x_1, x_2, x_3, x_4)$ sieht für n = 4 wie in Abbildung 6.34 aus.

Die Spalten der Matrix werden mit den möglichen Belegungen der Variablen x_1 und x_2 beschriftet; die Zeilen werden mit den möglichen Belegungen der Variablen x_3 (im Falle n = 3)

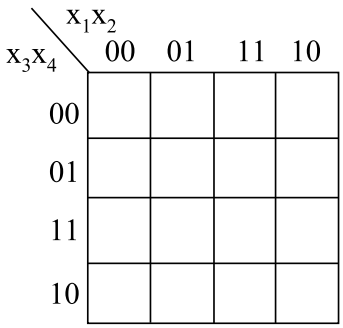


n = 4 heißt:
4x4 Matrix, deren 16 Felder in
geeigneter Weise mit Nullen
und Einsen belegt werden

Abbildung 6.34: 4x4 Matrix

bzw. x_3 und x_4 (im Falle n = 4) beschriftet.
Die Reihenfolge der Beschriftung erfolgt dabei so, dass sich zwei zyklisch benachbarte Spalten oder Zeilen nur in genau einer Komponente (Variable) unterscheiden. "Zyklisch benachbart" heißt dabei, dass auch die unterste und oberste Zeile bzw. die ganz linke und ganz rechte Spalte als benachbart angesehen werden.

Beispiel 6.21*:* Hier ein Beispiel:



Beschriftung des 4 x 4
Karnaugh-Diagramms
für n = 4

Abbildung 6.35: Beschriftung des KV-Diagramms für n = 4

In die entsprechenden Felder der Matrix werden nun die Funktionswerte von f eingetragen, wobei i.d. Regel nur genau die Einsen eingezeichnet werden. Jedem Minterm von f mit einschlägigem Index entspricht damit genau eine Eins im Karnaugh–Diagramm von f und umgekehrt.

Beispiel 6.22*:* Betrachte das vorangegangene Beispiel:

$$f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3x_4$$

Nun entsprechen zwei zyklisch benachbarte Einsen zwei Mintermen welche sich in genau einer komplementären Variablen unterscheiden. Auf diese Terme kann somit die Resolutionsregel angewandt werden.

Zwei solche zyklisch benachbarte Einsen bilden einen sogenannten Zweierblock. Der durch die Resolutionsregel entstehende Term hat gerade eine Variable weniger als jeder, der ihm zugrundeliegenden Minterme.

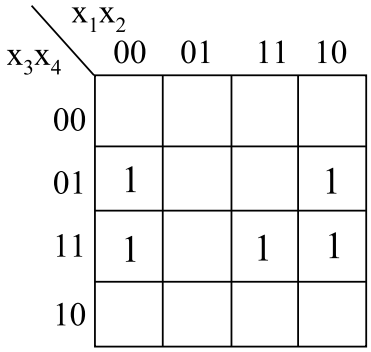


Abbildung 6.36: Beispiel

Diese Beobachtung läßt sich für Vierer–, Achter– und Sechzehner–Blöcke wie folgt verallgemeinern:
Rechteckige $2^r \times 2^s$–Blöcke, mit $r, s \in \{0, 1, 2\}$, von zyklisch benachbarten Einsen entsprechen $2^r * 2^s$ Mintermen, welche sich *paarweise* in $r + s$ Variablen unterscheiden, wobei alle Möglichkeiten des negierten bzw. nicht negierten Auftretens dieser Variablen vorkommen. Folglich läßt sich die Gesamtheit, d.h. die Summe dieser Minterme durch wiederholte Resolution zu dem Term vereinfachen, welcher gemeinsamer Bestandteil dieser Minterme ist. Er hat $(n - r - s)$ Variablen. D.h. je größer ein Block ist, desto weniger Variablen braucht man.

Für das Karnaugh–Diagramm bedeuted dies:

Man sollte alle im Diagramm auftretenden Einsen durch möglichst große Resolutionsblöcke der Form $2^r \times 2^s$ überdecken und dazu so viele Blöcke nutzen, dass jede 1 mindestens in einem Block vorkommt.

Beispiel 6.23*:* Hier ein Beispiel: Der Viererblock hängt weder von x_1 noch von x_3 ab, da

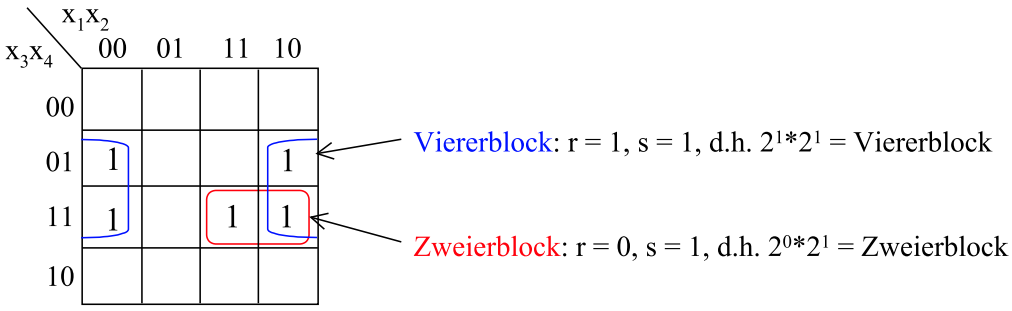


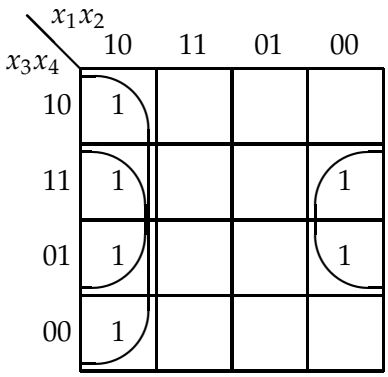
Abbildung 6.37: Viererblock

seine Einsen sowohl dort stehen, wo auch x_1 als auch x_3 jeweils die Werte 0 und auch 1 annehmen. Der ihm entsprechende Term enthält folglich nur x_2 und x_4 und hat gemäß der Spalten/ Zeilenbeschriftung den Wert $\bar{x}_2 x_4$. Analog ergibt sich für den Zweierblock der Wert $x_1 x_3 x_4$.

Die Rücktransformation der Blöcke in Terme ergibt den vereinfachten Wert der Booleschen Funktion.

Beispiel 6.24*:* Betrachte

$f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1\bar{x}_2\bar{x}_3x_4$
= gemäß Karnaugh–Diagramm = $x_1\bar{x}_2 + \bar{x}_2x_4$



Bemerkung:

1. Beachte, dass auch im Falle der Karnaugh–Diagramms in Abbildung 6.38 alle vier

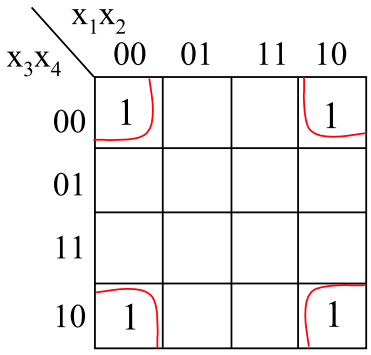


Abbildung 6.38: Ecken

   Ecken zu einem Viererblock zusammengefaßt werden können.

2. In seltenen Fällen ist es sinnvoll, *nicht* unbedingt von den größten Blöcken auszugehen (vgl. Abbildung 6.39). Auch die Minimierung der Anzahl der Blöcke ist relevant.

6.5.2 Don't–Care–Argumente

Bisher sind wir bei der Behandlung von Schaltnetzen immer davon ausgegangen, dass die zu realisierende Boolesche Funktion total war. D.h. für $f : B^n \to B$ umfaßte der Definitionsbereich von f ganz B^n. Es waren alle 2^n Elemente von B^n als Argumente für f möglich.

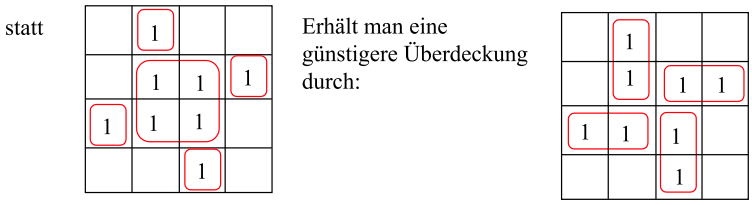


Abbildung 6.39: Verbesserung

Beispiel 6.25*:* Aus den 10 Dezimalziffern sollen z.B. vier Werte ausgewählt werden, für die eine Lampe leuchtet. Alle anderen 6 Dezimalziffern sollen auf 0 abgebildet werden.
Schalttechnisch benötigt man eine Funktion $f : B^4 \to B$ mit $f(x) = 1$ für $x = i, j, k, l$ und $f(x) = 0$ für $x = m,\ m \neq i, j, k, l$ und $i, j, k, l, m \in \{0, ..., 9\}$.

Es kann folglich der Fall auftreten, dass nur gewisse der 2^n Inputs möglich sind, alle anderen Argumente sind nicht festgelegt.

Beispiel 6.26*:* Im Beispiel 6.25 sind von den 16 Eingängen 10 als Ausgänge definiert und 6 nicht.

Diese restlichen Argumente werden als **Don't Cares** bezeichnet. Eine solche Boolesche Funktion f nennt man partiell. Die Don't–Care–Argumente können mit willkürlichen Funktionswerten belegt werden. Ist f drei– oder vierstellig, so erhalten wir mittels Karnaugh das einfachste Schaltnetz, wenn Don't–Cares mit dem Funktionswert 1 belegt werden, wenn dadurch bereits vorhandene Blöcke vergrößert werden können. Selbstverständlich brauchen Don't–Care–Blöcke jedoch nicht überdeckt zu werden.

Beispiel 6.27*:* Sei f für $x \in \{0, 1, 2, ..., 9\}$ definiert durch

$$f(x) := \begin{cases} 1\ falls\ x \in \{4, 5, 8, 9\} \\ 0\ sonst,\ d.h.\ x \in \{0, 1, 2, 3, 6, 7\} \end{cases}$$

Zur Binärcodierung verwenden wir vierstellige Dualzahlen, mit denen $s^4 = 16$ Argumente codiert werden könnten. Wir erhalten folglich $16 - 10 = 6$ Don't–Care–Argumente, die im

folgenden durch D gekennzeichnet werden.

	x_1	x_2	x_3	x_4	
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
A	1	0	1	0	*D*
B	1	0	1	1	*D*
C	1	1	0	0	*D*
D	1	1	0	1	*D*
E	1	1	1	0	*D*
F	1	1	1	1	*D*

Dies ergibt das Karnaugh–Diagramm in Abbildung 6.40. Mit den größtmöglichen Über-

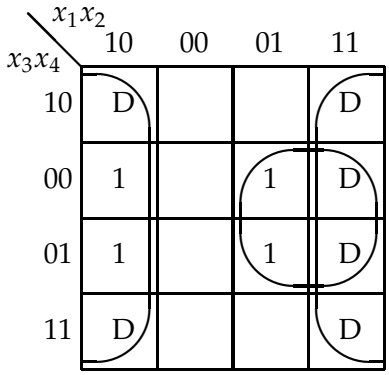


Abbildung 6.40: Don't Care

deckungen durch einen Achter– und einen Viererblock erhalten wir $f(x_1, x_2, x_3, x_4) = x_1 + \overline{x_3}x_4$.
Kosten: 2

Zum Vergleich:
ohne Ausnutzung der Don't–Cares wäre diese Funktion mit $f(x_1, x_2, x_3, x_4) = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3$ nicht so einfach ausgefallen.
Kosten: 5

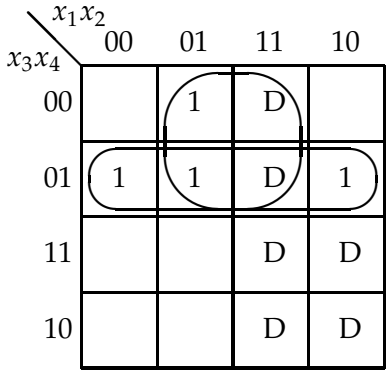
Beispiel 6.28: Als weiteres Beispiel betrachten wir:
Sei f definiert durch:

$$\begin{cases} 1 \text{ falls } x \in \{1,4,5,9\} \\ 0 \text{ sonst, d.h. } x \in \{0,2,3,6,7,8\} \end{cases}$$

Dies ergibt folgende Tabelle:

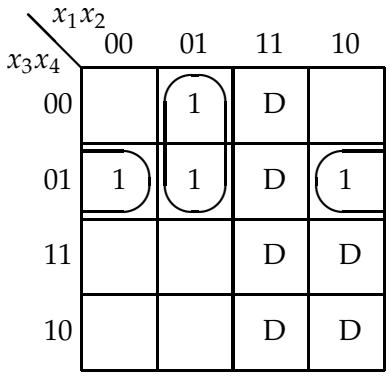
	x_1	x_2	x_3	x_4	
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
A	1	0	1	0	*D*
B	1	0	1	1	*D*
C	1	1	0	0	*D*
D	1	1	0	1	*D*
E	1	1	1	0	*D*
F	1	1	1	1	*D*

Das Karnaugh-Diagramm lautet:



Damit ergibt sich $f(x_1, x_2, x_3, x_4) = x_2\overline{x_3} + \overline{x_3}x_4$.

Ohne Dont't Cares ergibt sich:



und die Funktion $f(x_1, x_2, x_3, x_4) = x_1x_2\overline{x_3} + \overline{x_2}x_3x_4$

6.5.3 Quine–McCluskey–Verfahren

Während das Verfahren von Karnaugh nur zur Vereinfachung Boolescher Funktionen mit wenigen Argumenten geeignet ist, da es sonst sehr unübersichtlich wird, wollen wir im folgenden ein anderes Vereinfachungsverfahren kennenlernen, das für Boolesche Funktionen beliebiger Stellen geeignet ist.

Dazu benötigen wir zunächst einige Begriffe.

Definition 6.7 (disjunktive Form): *Eine Boolesche Funktion $f : B^n \to B$ liegt in **disjunktiver Form (DF)** vor, wenn f darstellbar ist als*

$$\sum_{i=1}^{k} M_i,\ k \geq 1.$$

Dabei ist M_i ein Term der Form

$$\prod_{j=1}^{l} x_{i_j}^{\alpha_j},\ l \geq 1$$

wobei

$$x_{i_j}^{\alpha_j} = \begin{cases} x_{i_j},\ falls\ \alpha_j = 1 \\ \bar{x}_{i_j},\ falls\ \alpha_j = 0. \end{cases}$$

Beachte
Im Gegensatz zur Disjunktiven Normalform haben wir hier Terme M_i statt m_i, wobei die M_i nicht alle möglichen n Faktoren enthalten müssen.

M_i darf jedoch **nicht** mit dem in Kapitel 6.4.1 definierten Maxterm verwechselt werden.

Beispiel 6.29*:* n = 4:
Ein Summand der DNF wäre z.B. $m := x_1\bar{x}_2x_3x_4$. Dies kann auch ein Summand der DF sein.
Dagegen wäre $M_i = x_1\bar{x}_3$ ein Summand der DF, jedoch kein Summand der DNF.
DF enthalten i.d.R. Terme mit weniger als n Variablen.

Die DNF jeder Booleschen Funktion $f : B^n \to B$ ist eine DF. Eine DF einer Booleschen Funktion ist i.d.R. jedoch keine DNF. Man kann aber eine beliebige DF in eine DNF umformen. Umgekehrt gibt es zu einer DNF i.d.R. viele DF.

Disjunktive Formen werden durch zweistufige Schaltungen realisiert: die erste Stufe berechnet mittels AND–Gattern die einzelnen Produkte, die zweite Stufe verknüpft diese Ergebnisse mit einem großen OR–Gatter. Damit besteht eine geringe Signallaufzeit und eine praktische Realisierung ist einfach und schnell z.B. mit einer PLA-ähnlichen Realisierung.

Wir wollen eine Schaltung bewerten. Dazu benötigen wir ihre **Kosten**.

Definition 6.8 (Kosten): *Sei $f : B^n \to B$ eine Boolesche Funktion und d eine der disjunktiven Darstellungen. Dann ergeben sich die Kosten K(d) wie folgt:*
Für

$$d \equiv x_{i_1}^{\alpha_1} * x_{i_2}^{\alpha_2} * \cdots * x_{i_t}^{\alpha_t} \;\; gilt: \;\; K(d) = t - 1$$

(da man t-1 AND–Gatter zur Realisierung dieses Produktes braucht) und für

$$d \equiv M_1 + M_2 + \cdots + M_k \;\; gilt: \;\; K(d) = (k-1) + \sum_{i=1}^{k} K(M_i)$$

(da man jedes Produkt einzeln realisieren muß und für k Terme k-1 OR–Gatter für deren Verknüpfung benötigt). D.h. wir nehmen an, dass Inverter keine Kosten verursachen und ansonsten die Anzahl der AND– und OR–Gatter gezählt wird.

Beispiel 6.30*:*

$f = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_3x_4 + x_1x_2\bar{x}_3x_4 + \bar{x}_1x_2x_3x_4 + x_1\,x_2x_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2x_3\bar{x}_4$

verursacht die Kosten $K(f) = (8-1) + 8 * (4-1) = 7 + 24 = 31$.

Mittels Karnaugh–Verfahren kann man diesen Term zu einer disjunktiven Form

$$f' = x_2x_4 + \bar{x}_2\bar{x}_4$$

vereinfachen, der die Kosten

$$K(f') = (2-1) + 2 * (2-1) = 1 + 2 * 1 = 3$$

hat, also wesentlich billiger zu realisieren ist (mehr als Faktor 10).

Bemerkung:
Sei allgemein $f : B^n \to B$ in DNF dargestellt durch $d \equiv M_1 + M_2 + ... + M_k$. Dann ist jeder Term Minterm und d hat stets die Kosten

$$K(d) = k - 1 + k * (n-1) = k - 1 + k * n - k = k * n - 1.$$

Das Karnaugh–Verfahren vereinfacht Schaltfunktionen und reduziert ihre Kosten. Diese Problematik soll nun auch auf Funktionen $f : B^n \to B$ mit $n > 4$ angewandt werden. Wir formulieren diese Problematik in ihrer allgemeinsten Form wie folgt:

Vereinfachungsproblem Boolescher Funktionen:
Bestimme zu einer gegebenen Booleschen Funktion $f : B^n \to B$ eine sie darstellende disjunktive Form d mit minimalen Kosten $K(d)$.

Für n = 3, 4 kann das Karnaugh–Verfahren angewandt werden. Für größere Dimensionen benötigen wir weitere Begriffe.

Definition 6.9 (Implikant/Primimplikant): *Sei $f : B^n \to B$ eine Boolesche Funktion. Ein Term M heißt* ***Implikant*** *von f, kurz $M \leq f$, falls $M(x) \leq f(x)$, d.h. $M(x) = 1 \Rightarrow f(x) = 1 \; \forall x \in B^n$ gilt. Ein Implikant M von f heißt* ***Primimplikant*** *von f, falls es keine echte Verkürzung von M gibt, die noch Implikant von f ist.*
Eselsbrücke: *Es gilt $g \leq h$, falls die Einsen von g im Karnaugh-Diagramm eine Teilmenge der Einsen von h sind.*

Bemerkung:

- f′ ist eine Verkürzung von f, falls $K(f') < K(f)$ und $f \leq f'$ gilt.

- Im Karnaugh–Diagramm entsprechen rechteckige Blöcke von Einsen (Länge und Breite der Blöcke sind Zweierpotenzen) den Implikanten und maximale derartige Blöcke den Primimplikanten, d.h. beim Optimieren suchen wir Primimplikanten.

- Ist M Implikant von f, und ist m ein Minterm von f derart, dass M eine Verkürzung von m ist, so gilt $m \leq M$, d.h. m ist Implikant von M.

- Minterme zu einschlägigen Indizes von f sind Implikanten von f.

Es läßt sich beweisen, dass folgender Satz gilt:

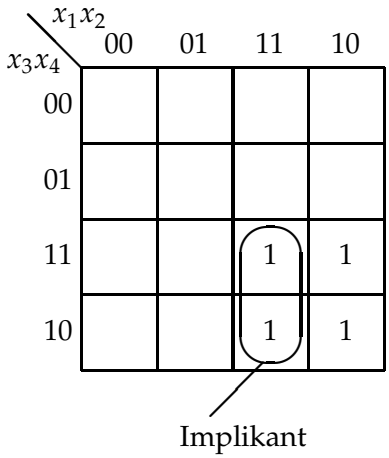
Sei $f : B^n \to B$ eine Boolesche Funktion und $d \equiv M_1 + M_2 + \ldots + M_k$ eine Darstellung von f als disjunktive Form mit minimalen Kosten. Dann sind die $M_1, \ldots, M_k$ Primimplikanten von f. $m \leq M \leq f$, M Verkleinerung von m.

Beispiel 6.31*:*

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= x_1\overline{x}_2x_3 + x_1x_2x_3 \\ M_0 &= x_1x_2x_3x_4 \text{ und} \\ M_1 &= x_1\overline{x}_2x_3 \text{ und} \\ M_2 &= x_1x_2x_3 \end{aligned}$$

sind **keine** Primimplikanten.

$M_p = x_1x_3$ ist Primimplikant.



$M_2 = x_1x_2x_3$

$m = x_1x_2x_3x_4$ ist Minterm und Implikant von $M_2 = x_1x_2x_3$

M_2 ist Verkürzung von m

$$\begin{aligned} \Rightarrow x_1x_2x_3x_4 &\leq x_1x_2x_3 \\ m &\leq M_2 \end{aligned}$$

Nach diesen betrachteten Grundlagen wollen wir nun zu einem Verfahren übergehen, das 1952 von W. Quine angegeben wurden und 1956 von E. McCluskey verbessert wurde.
Es besteht darin,

1. alle Primimplikanten zu bestimmen und
2. eine kostenminimale Auswahl von Primimplikanten vorzunehmen.

Das eigentliche Verfahren wollen wir mittels eines Beispiels veranschaulichen.

Beispiel 6.32*:* Sei $f : B^n \to B$ gegeben durch die DNF–Darstellung:
$f = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_3\bar{x}_4 + x_1\bar{x}_2x_3x_4 + x_1x_2\bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3x_4 + x_1x_2x_3\bar{x}_4$

Schritt 1: Bestimmung aller Implikanten, dann aller Primimplikanten.
Nach der Resolutionsregel suchen wir Terme, die sich lediglich in der Komplementarität einer Variablen unterscheiden. Dazu teilt man alle Minterme anhand der Anzahl vorkommender Negationszeichen in Gruppen ein:

Gruppe	Minterm	Einschlgiger Index
1	$x_1\overline{x}_2x_3x_4$	$1011 = 11$
	$x_1x_2\overline{x}_3x_4$	$1101 = 13$
	$x_1x_2x_3\overline{x}_4$	$1110 = 14$
2	$\overline{x}_1x_2x_3\overline{x}_4$	$0110 = 6$
	$x_1x_2\overline{x}_3\overline{x}_4$	$1100 = 12$
3	$\overline{x}_1x_2\overline{x}_3\overline{x}_4$	$0100 = 4$
4	$\overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4$	$0000 = 0$

Alle Paare von Mintermen, auf welche die Resolutionsregel anwendbar ist, findet man somit durch Betrachtung aller Paare benachbarter Gruppen.

Alle gewonnenen verkürzten Implikanten werden mit den einschlägigen Indizes der "verbrauchten" Terme und den nicht verbrauchten, d.h. den Implikanten, auf die die Resolutionsregel nicht mehr anwendbar ist, in eine neue Tabelle eingetragen.
Dies ergibt:

Gruppe	Implikant	Index und verbrauchte Mintermnummern	
1	$x_1x_2\overline{x}_3$	$110* = 12,\ 13$	
	$x_2x_3\overline{x}_4$	$*110 = 6,\ 14$	
	$x_1x_2\overline{x}_4$	$11*0 = 12,\ 14$	
	$x_1\overline{x}_2x_3x_4$	$1011 = 11$	(wurde noch nicht verbraucht)
2	$\overline{x}_1x_2\overline{x}_4$	$01*0 = 4,\ 6$	
	$x_2\overline{x}_3\overline{x}_4$	$*100 = 4,\ 12$	
3	$\overline{x}_1\overline{x}_3\overline{x}_4$	$0*00 = 0,\ 4$	

Das "*" beim Index kennzeichnet dabei durch die Resolutionsregel herausgefallene Variablen.

Man iteriert dieses Verfahren sooft, bis sich keine Vereinfachung der Tabelle mehr ergibt.

Gruppe	Implikant	Index und verbrauchte Mintermnummern
1	$x_2\overline{x}_4$	$*1*0 = 4,\ 6,\ 12,\ 14$
	$x_1x_2\overline{x}_3$	$110* = 12,\ 13$
	$x_1\overline{x}_2x_3x_4$	$1011 = 11$
3	$\overline{x}_1\overline{x}_3\overline{x}_4$	$0*00 = 0,\ 4$

Dies ist die letzte Tabelle. Sie besteht aus *allen* Primimplikanten.

Schritt 2: Kostenminimale Auswahl von Primimplikanten
Dazu stellen wir zunächst die sogenannte Implikationsmatrix auf, die den Zusammenhang zwischen Primimplikanten und Mintermen festhält:

Primimplikant\Minterm	0	4	6	11	12	13	14
$x_1\overline{x}_2x_3x_4$				1			
$x_1x_2\overline{x}_3$					1	1	
$x_2\overline{x}_4$		1	1		1		1
$\overline{x}_1\overline{x}_3\overline{x}_4$	1	1					

Ein Matrixelement a_{ij} wird gesetzt, wenn der Minterm $j \leq$ zur Verkürzung von Primimplikant i herangezogen wurde. Sonst werden die Matrixelemente 0 gesetzt. Bei diesen Überlegungen hilft die letzte Spalte der in Schritt 1 zuletzt aufgestellten Tabelle, d.h. dass der j–te Minterm an der Bildung des i–ten Primimplikanten beteiligt war.

In dieser aufgestellten Matrix hat man nun noch eine Auswahl von Primimplikanten, d.h. Zeilen so zu treffen, dass die von diesen Zeilen gebildete Teilmatrix in jeder Spalte

mindestens eine Eins enthält und andererseits die Gesamtkosten dieser Primimplikanten minimal sind.

Im oben genannten Beispiel benötigt man alle Primimplikanten. Ferner ist $d \equiv x_1\bar{x}_2x_3x_4 \vee x_1x_2\bar{x}_3 \vee x_2\bar{x}_4 \vee \bar{x}_1\bar{x}_3\bar{x}_4$ mit $K(d) = 11$.

Dieser Schritt 2 kann unter Umständen einen sehr hohen Aufwand erfordern, wenn man viele Alternativen bzgl. ihrer Kosten miteinander vergleichen muß! Dazu liefert Quine McCluskey jedoch keinen Algorithmus.

Beispiel 6.33*:* Wir betrachten noch ein weiteres Beispiel:
f(x) = $x_1x_2x_3x_4 + x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_3x_4$

1. Bestimmung aller Implikanten, dann der Primimplikanten

Gruppe	Minterm	Einschlägiger Index
0	$x_1x_2x_3x_4$	1111 = 15
1	$x_1\bar{x}_2x_3x_4$	1011 = 11
2	$x_1\bar{x}_2\bar{x}_3x_4$	1001 = 09
	$\bar{x}_1\bar{x}_2x_3x_4$	0011 = 03
3	$\bar{x}_1x_2\bar{x}_3\bar{x}_4$	0100 = 04
	$\bar{x}_1\bar{x}_2x_3\bar{x}_4$	0010 = 02
	$\bar{x}_1\bar{x}_2\bar{x}_3x_4$	0001 = 01

2. Verkürzung der Implikanten

Gruppe	Implikant	Index und verbrauchte Mintermnummern	
0	$x_1x_3x_4$	1*11	= 11, 15
1	$x_1\bar{x}_2x_4$	10*1	= 9, 11
	$\bar{x}_2x_3x_4$	*011	= 3, 11
2	$\bar{x}_2\bar{x}_3x_4$	*001	= 9, 1
	$\bar{x}_1\bar{x}_2x_3$	001*	= 2, 3
	$\bar{x}_1\bar{x}_2x_4$	00*1	= 1, 3
3	$\bar{x}_1x_2\bar{x}_3\bar{x}_4$	0100	= 4

3. Am Ende bekommen wir:

Gruppe	Implikant	Index und verbrauchte Mintermnummern	
0	$x_1x_3x_4$	1*11	= 11, 15
1	$\bar{x}_2x_4$	*0*1	= 1, 3, 9, 11
2	$\bar{x}_1\bar{x}_2\bar{x}_3$	001*	= 2, 3
3	$\bar{x}_1x_2\bar{x}_3\bar{x}_4$	0100	= 4

Primimplikanten/Minterm	15	11	9	3	4	2	1
$x_1x_3x_4$	1	1					
$\bar{x}_2x_4$		1	1	1			1
$x_1x_3x_4$				1		1	
$x_1x_3x_4$					1		

Teil III

Speicherung

7

Schaltwerke

Im Kapitel 6 haben wir Möglichkeiten kennengelernt, das Verhalten einer "Black Box" logisch durch Schaltfunktionen zu beschreiben. Ferner haben wir erläutert, wie sich vorgegebene *Schaltfunktionen – zusammengesetzt aus Booleschen Funktionen* – durch Schaltnetze realisieren lassen.

Wir sind dabei davon ausgegangen, dass sich *nach dem Anlegen von Input–Signalen quasi unmittelbar*, d.h. nur ganz kurze Zeit später, *Signale an den Ausgängen einstellen. Diese Zeit* vom Anlegen der Inputs bis zum Ablesen der Outputs wurde dabei *vernachlässigt*.

Außerdem war für die Berechnung eines Outputs nur der aktuelle Input maßgebend, nicht jedoch frühere Eingaben.

Nun wollen wir ein Beispiel betrachten.

Beispiel 7.1 *(Ringzähler):* Gesucht ist ein Ringzähler für 4–stellige Dualzahlen, der ausgehend von einer beliebigen Zahl diese um 1 erhöht und das Ergebnis dann wieder als Input betrachtet und erneut um 1 erhöht u.s.w., d.h. gesucht ist die Realisierung der Funktion

$$R : B^4 \to B^4 \; definiert \; durch \; R(d(i)) := d((i+1) mod 16)$$

wobei d(i) die 4–stellige Dualdarstellung von $i \in \{0, \ldots, 15\}$ sei.
Gibt man z.B. $x_3 = 0,\ x_2 = 1,\ x_1 = 1,\ x_0 = 0$ ein, so erhält man den Output $y_3 = 0,\ y_2 = 1,\ y_1 = 1,\ y_0 = 1$.

Problematisch wird es nun, diesen Output als neuen Input zu nehmen und erneut die Funktion darauf anzuwenden, d.h. den Output als nächsten Input aufzufassen, also eine Art *Rückkopplung* herzustellen.

Schaltnetze – als zyklusfreie Graphen – lassen eine derartige Konstruktion nicht zu, im Gegensatz zu *Schaltwerken, die gerichtete zyklische Graphen darstellen*.

Abhilfe können wir durch die Einführung einer Kontrollinstanz schaffen, welche durch eine zentrale Uhr (clock) Taktimpulse ausgibt.

7.1 Delay

Als neues Bauteil verwenden wir dazu ein sogenanntes **Delay**, welches in der Lage ist, ein Bit zu speichern.

Aus logischer Sicht besteht ein Delay aus einem Vorspeicher V und einem Speicher S. Ein Delay ist nun in der Lage, ein Bit zu speichern. Die Arbeitsweise ähnelt einer Schleuse, die in 2 Phasen arbeitet:

Abbildung 7.1: Beispiel einer Schleuse

1. Arbeitsphase: Der Inhalt von S wird nach rechts abgegeben, es steht also als Signal y_i für eine längere Zeit zur Verfügung. Ein Signal x_i wird in V abgelegt, wobei V und S durch eine Sperre getrennt sind (vgl. Abbildung 7.1).

2. Setzphase Eine zentrale Synchronisation (Clock = Uhr, die Taktimpulse erzeugt), hebt die Sperre kurzzeitig auf und bewirkt dadurch die Abgabe des Inhaltes von V an S. Diese Abgabe wird als **Setzen** bezeichnet.

Die Setzphase ist i.A. wesentlich kürzer als die Arbeitsphase. In dieser Setzphase werden keine Signale von außen aufgenommen oder nach außen abgegeben.

Befindet sich zum Zeitpunkt i der Wert x_i im Vorspeicher und y_i im Speicher, so wird beim nächsten Takt der Wert x_i in den Speicher geschrieben und vom Zeitpunkt i zum Zeitpunkt i + 1 übergegangen. Formal schreibt man $y_{i+1} \Leftarrow x_i$, $y_{i+1} \leftarrow x_i$ oder $y_{i+1} := x_i$.

Damit läßt sich ein funktionstüchtiger Ringzähler implementieren.

Beispiel 7.2: Wir führen 4 Delays ein, die einen gerade erzeugten Output speichern, damit er im nächsten Takt als Input dient (vgl. Abbildung 7.2).

In der Praxis wird dabei die Clock so beschaffen sein müssen, dass ein neuer Taktimpuls erst dann erzeugt wird, wenn man sicher sein kann, dass der vom Ringzähler erzeugte Output die gesamte Schaltung durchlaufen hat.

Bei der Verwendung von Delays nehmen wir ein getaktetes synchronisiertes Arbeiten an. Solche Schaltungen mit Delays nennen wir **Schaltwerke**.

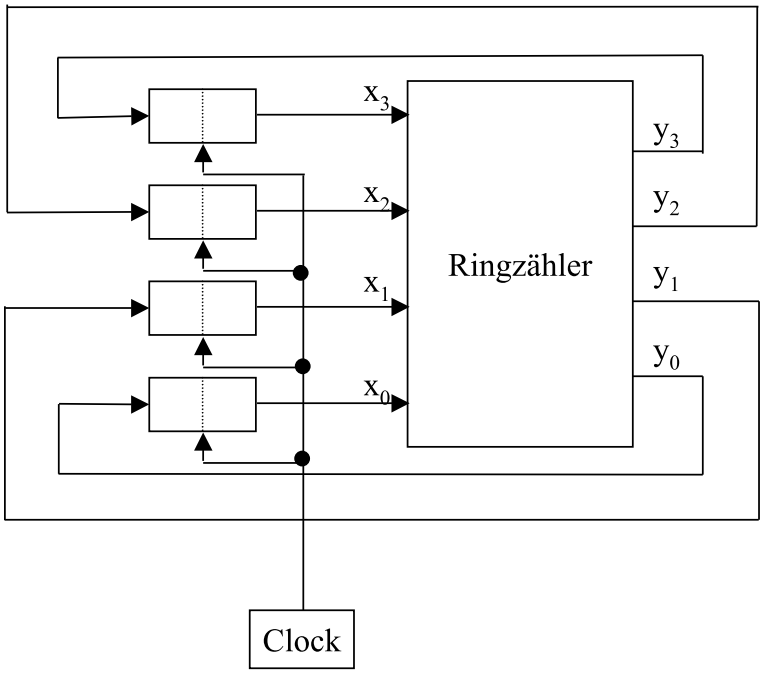


Abbildung 7.2: Beispiel eines Ringzählers

Zu Speicherungszwecken im Computer werden wir i.d.R. nicht mit einem Delay auskommen. Meist benötigt man **Folgen von Delays**. Diese werden dann auch als **Register** bezeichnet.

Im Beispiel des Ringzählers haben wir ein 4–stelliges Register verwendet, dessen Komponenten paarweise voneinander unabhängig waren.

Wir verwenden dafür eine vereinfachte Darstellung wie in Abbildung 7.3. Dabei ist nur der Speicherteil nicht jedoch der Vorspeicher dargestellt. *Register sind nun in der Lage, Worte der Länge n über der Booleschen Menge B zu speichern*. Bei einem n–stelligen Register bezeichnet man **n** auch als **Wortlänge**. Gängige Wortlängen sind z.B. n = 8, 16, 32, 48, 64 also z.B. 1, 2, 4, 6, 8 Byte oder 2, 4, 8, 12, 16 Hexadezimalziffern.

Alle Delays eines solchen Registers und prinzipiell auch alle Delays eines Schaltwerkes bzw. Rechners werden insgesamt *gleichzeitig getaktet*, wobei zwischen zwei aufeinanderfolgenden Taktimpulsen in jedem Delay sowohl Arbeits– als auch Setzphase ablaufen. Die *Zeit*, die *zwischen zwei Taktimpulsen* vergeht, nennt man die *Taktzeit* oder *Taktzykluszeit* eines Rechners, *den die Clock, d.h. die Rechneruhr erzeugt*.

Taktzeiten liegen heute in der Größenordnung 10^{-9} bis 10^{-8} Sekunden. In diesem Rhythmus vollziehen sich synchron alle rechnerinternen Abläufe. Aber nicht alle Abläufe können innerhalb eines Takts realisiert werden. So kann eine Addition zweier Dualzahlen z.B. 12 Takte dauern. Innerhalb eines Taktes können parallel jedoch mehrere Ereignisse ablaufen.

Sollen diese Ereignisse innerhalb eines Taktzyklus in einer bestimmten Reihenfolge ablaufen, muß der Taktzyklus in Teilzyklen aufgeteilt werden. Ein übliches Vorgehen, eine feinere Auflösung als der einfache Taktgeber bietet, ist die Anzapfung der primären Taktleitung und

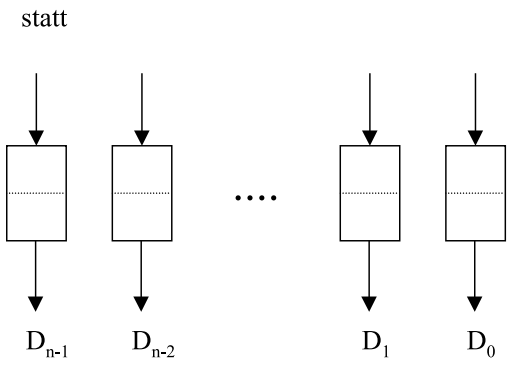


Abbildung 7.3: Darstellung eines n–stelligen Registers

das Einfügen einer Schaltung mit einer Verzögerung, so dass zum primären ein sekundäres, phasenverschobenes Taktsignal erzeugt wird (vgl. Abbildung 7.4).

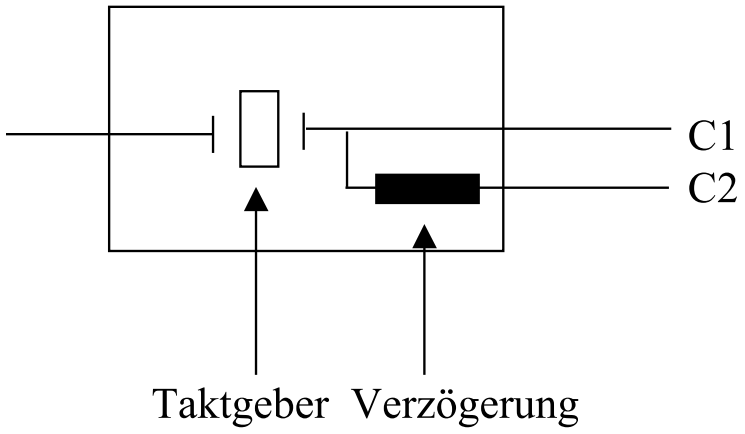


Abbildung 7.4: Taktgeber und Verzögerung für ein sekundäres phasenweise verschobenes Taktsignal

Daraus ergibt sich das Taktdiagramm in Abbildung 7.5.

Bemerkung: Taktgeber (Clocks) sind immer symmetrisch, d.h. die im High-Zustand ablaufende Zeit ist gleich der im Low-Zustand ablaufenden Zeit.

Ist eine noch stärkere Verfeinerung erforderlich, so kann man mehr sekundäre Leitungen von der primären aus anzapfen und mit verschiedenen Verzögerungen belegen.

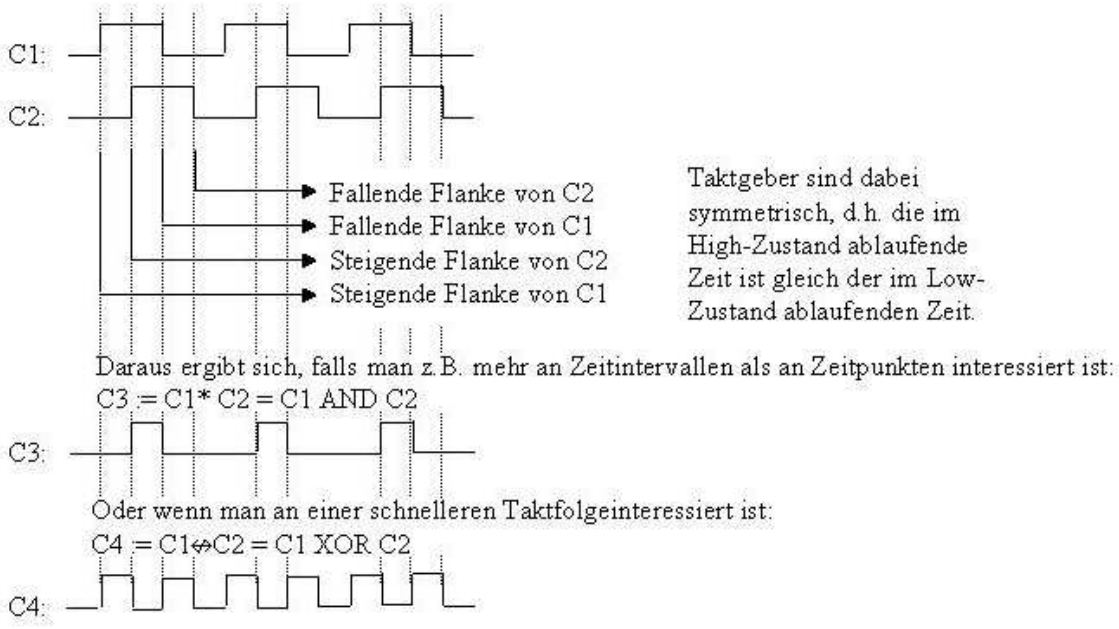


Abbildung 7.5: Taktdiagramm

Nun wollen wir uns mit der Realisierung von Speicherelementen wie Delays oder Registern beschäftigen.

7.2 Realisierung von 1-Bit-Speichern

Ein Speicher dient der Speicherung auszuführender Instruktionen und Daten. Auf der Gate–ebene beginnend wollen wir im folgenden sehen, wie solche Speicher funktionieren.

Um einen 1–Bit–Speicher zu erstellen, brauchen wir eine Schaltung, die sich Eingabewerte merkt. Eine solche Schaltung läßt sich aus zwei NOR–Gattern bauen.

7.2.1 Latches

Wir betrachten nun einen **SR–Latch** (vgl. Abbildung 7.6).
Diese Schaltung hat zwei Eingänge: S (Set) zum Setzen und R (Reset) zum Löschen. Ferner hat sie zwei komplementäre Ausgänge Q und $\bar{Q}$.

Fall 1: $S = R = 0$

Im Falle $Q = 0$ ist am oberen NOR–Gatter der Ausgang $\bar{Q} = 1$ und damit am unteren NOR–Gatter der Ausgang $Q = 0$.
Umgekehrt ergibt $Q = 1$ den Ausgang $\bar{Q} = 0$ und damit bleibt auch hier der Wert für $Q = 1$ durch die Rückführung.
$\Rightarrow$ Diese beiden Zustände sind konsistent.

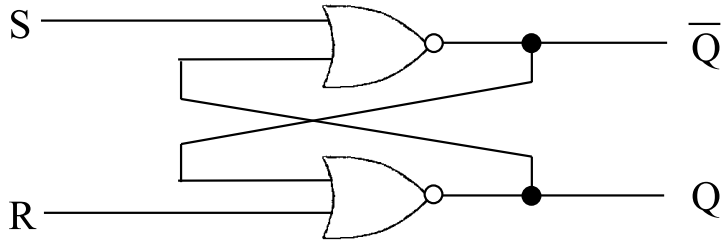


Abbildung 7.6: SR–Latch

Fall 2: $S = 1$ bei $Q = 0$ (oder $Q = 1$)
Dadurch wird am oberen Gatter $\bar{Q} = 0$ erzwungen. Bei $R = 0$ und dem zurückgeführten $\bar{Q} = 0$ ergibt sich $Q = 1$.
$\Rightarrow$ Setzt man also S, so setzt man Q.

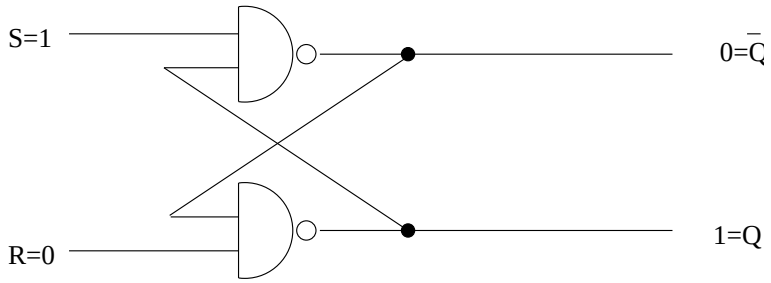
Fall 3: $R = 1$ bei $Q = 0$ (oder $Q = 1$)
Dadurch wird am unteren Gatter $Q = 0$ erzwungen. Bei $S = 0$ und dem zurückgeführten $Q = 0$ ergibt sich $\bar{Q} = 1$.
$\Rightarrow$ Setzt man R, so löscht man Q.

Fall 4: $S = R = 1$
Dieser Fall ist unzulässig. Es ergibt sich $Q = \overline{Q} = 0$. Das nachfolgenden Anlegen beider Eingänge auf 0 führt zu einem Undeterminismus, d.h. es gewinnt der Eingang, der am längsten in 1 bleibt.

Wir können zusammenfassen, dass bei $S = R = 0$ der aktuelle Zustand erhalten bleibt. Bei einem vorübergehenden Setzen von S oder R endet unser Latchbauelement in einem bestimmten Zustand – und zwar ungeachtet vorheriger Zustände.

Die Schaltung merkt sich also, ob S oder R zuletzt an war. Anhand dieser Eigenschaft können wir Computerspeicher entwickeln.

Bemerkung: Ein äquivalentes Verhalten eines 1-Bit-Speichers läßt sich auch durch ein Latch realisieren, das anstelle der beiden NOR-Gatter je ein NAND-Gatter hat.



Soll ein solches Bauelement nur zu ganz bestimmten Zeiten seinen Zustand ändern, so ist ein **getaktetes SR–Latch (Clocked SR Latch)** (vgl. Abbildung 7.7) zu realisieren.

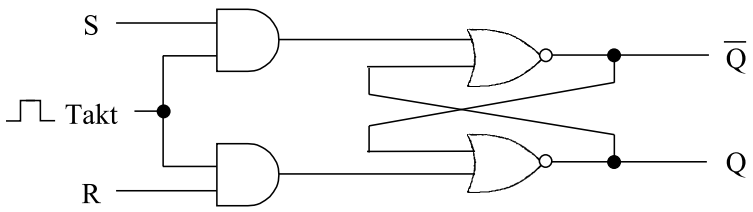


Abbildung 7.7: Getaktetes SR–Latch

Ist der Takt 0, dann reagiert die Schaltung gar nicht auf Eingänge. Der Zustand bleibt. Ist der Takt 1, so ist die Wirkung wie oben.

Wollen wir nun die Undeterminiertheit von $S = R = 1$ beseitigen, so empfiehlt sich ein getaktetes **D–Latch** (vgl. Abbildung 7.8).

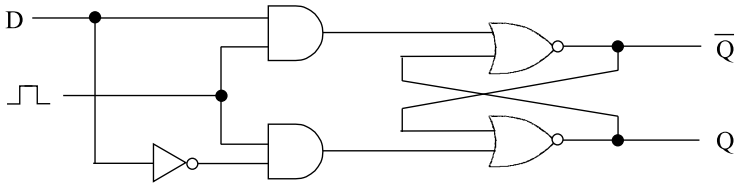


Abbildung 7.8: D–Latch

Der untere Eingang ist dabei immer das Komplement des oberen Eingangs.

Dieses Bauelement ist ein **echter 1–Bit–Speicher**, d.h. um den aktuellen Wert von D in den Speicher zu laden, wird ein positiver Impuls auf die Taktleitung gelegt.

7.2.2 Flip-Flops

Eine Variante der Latch–Bauelemente geht davon aus, dass der Zustandsübergang nicht eintritt, wenn der Taktgeber 1 ist, sondern von 0 auf 1 (steigende Flanke) oder von 1 auf 0 (falende Flanke) übergeht. Solche Elemente heißen **Flip–Flops**.

Eine *Flip–Flop*–Schaltung wird *flankengesteuert* genannt (edge–triggered), eine *Latch*–Schaltung wird *pegelgesteuert* (level–triggered) genannt. In der Literatur werden diese Begriffe aber auch umgekehrt gebraucht.

Die Realisierung eines D–Flip–Flops mit steigender Flanke geht nun davon aus, dass statt des Takts ein Bauelement mit der Ergänzung in Abbildung 7.9 verwendet wird.

Damit haben wir den Pegel in Abbildung 7.10 an den verschiedenen Stellen.

Und wir erhalten eine D–Flip–Flop–Schaltung wie in Abbildung 7.11.

Als abkürzende Darstellung für D-Latches und Flip-Flops wollen wir folgende Symbole verwenden, wobei Ck für den Taktgeber (Clock) steht, siehe Abbildng 7.12.

a D–Latch

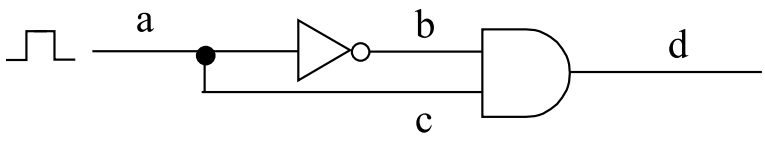


Abbildung 7.9: Ergänzung des Steuertakts zur Flankensteuerung

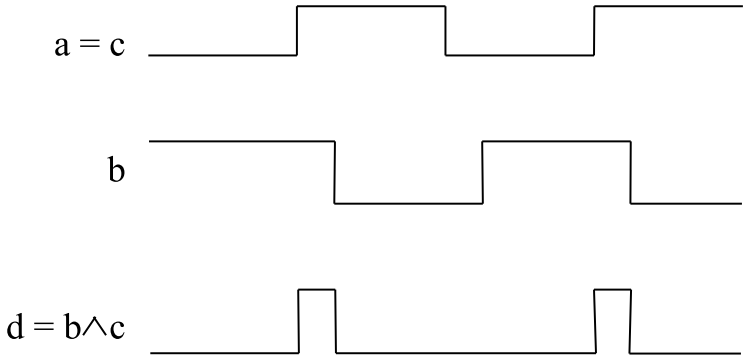


Abbildung 7.10: Pegelstände

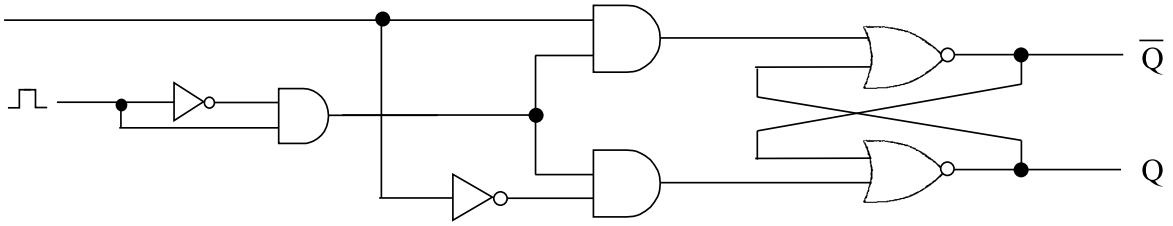


Abbildung 7.11: D–Flip–Flop–Schaltung

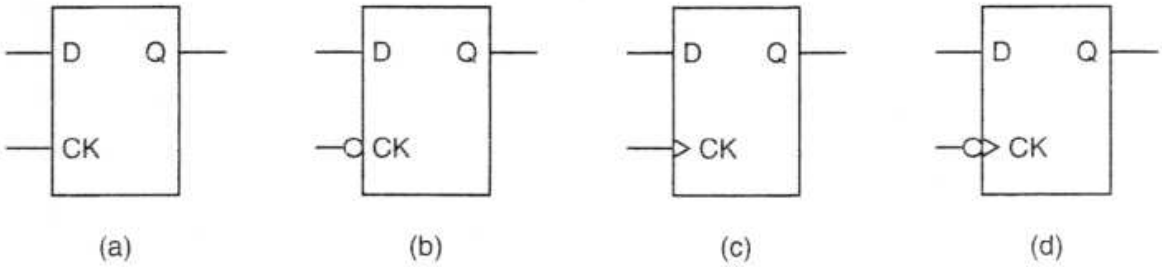


Abbildung 7.12: Standardsymbole für D-Latches und Flip-Flops

b 0-gesteuerter D–Latch

c Flip–Flop mit steigender Flanke

d Flip–Flop mit fallender Flanke

Der 0-gesteuerte D-Latch hat einen Taktgeber, der normalerweise 1 ist, aber auf 0 abfällt, um den Zustand von D zu laden. Der Flip Flop mit steigender Flanke lädt den Zustand von D bei Übergang des Taktimpulses von 0 auf 1. Der Flip Flop mit fallender Flanke lädt den Zustand von D entsprechend bei Übergang des Taktimpulses von 1 auf 0.

Das Symbol kann um einen Ausgang $\overline{Q}$ erweitert werden, sowie zwei zusätzliche Eingänge Preset (PR), der den Zustand $Q = 1$ setzt sowie Clear (CLR), der den Zustand $Q = 0$ setzt Für den Flip Flop mit steigender Flanke ergibt sich damit das Symbol:

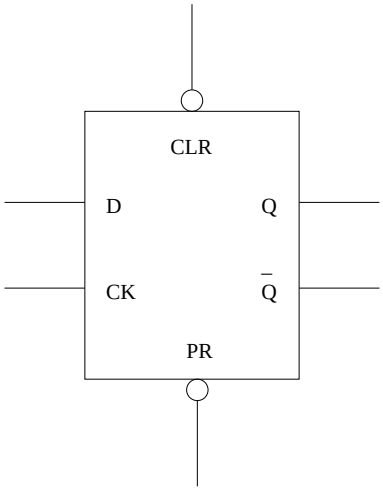


Abbildung 7.13: Um CLR, PR und $\overline{Q}$ erweitertes Flip Flop

7.3 Realisierung von Registern

Klassische Flip-Flop-Schaltungen ermöglichen die Speicherung *von einem Bit über die Dauer eines Takts*. Zur Speicherung *von mehreren Bits* gibt es verschiedene Konfigurationen, von denen wir zwei prinzipiell verschiedene betrachten wollen.

2-Bit-Register

Zunächst wollen wir eine einfache Schaltung mit *zwei* unabhängigen D-Flip-Flop-Bauelementen *inclusive Clear und Presetsignalen* betrachten.

Obwohl sich die Flip-Flop-Bauelemente gemeinsam auf einem 14-Pin-Chip befinden, haben beide Bauelemente keinen Zusammenhang, sondern arbeiten *völlig unabhängig voneinander*, indem pro Bauelement 6 Pins mit den Ein- bzw. Ausgängen der Flip-Flops verbunden sind:

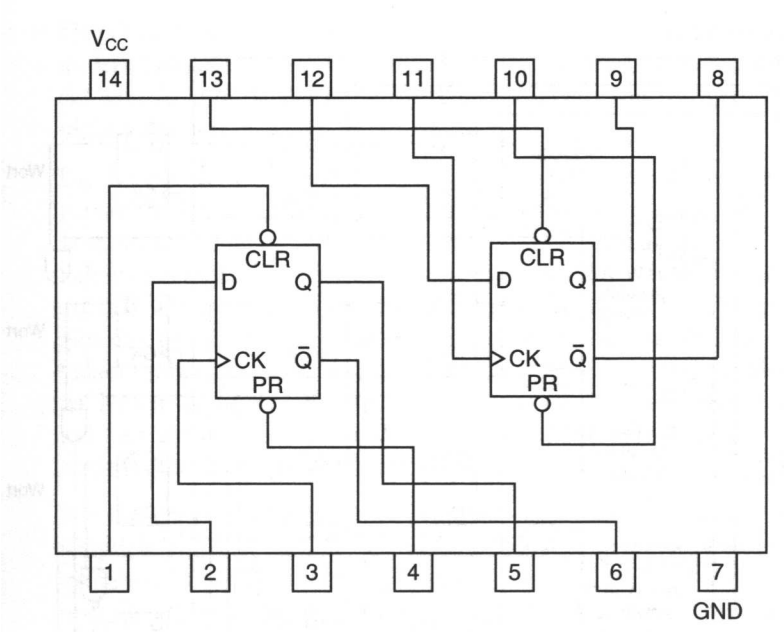


Abbildung 7.14: 2–Bit–Flip-Flop

einmal die PINs 1-6 mit einem Flip-Flop, und einmal die Pins 8-13 mit dem anderen Flip-Flop. Zusätzlich haben wir einen *Pin (Vcc) für Strom* und einen *Pin (GND) für Masse*.

8–Bit–Register

Im folgenden wollen wir 8 Flip-Flop-Bausteine so kombinieren, dass sie *synchron über einen gemeinsamen Takt* gesteuert werden können. Um das sich ergebende 8-Bit-Register schlank und kostengrünstig zu gestalten, wird auf die $\overline{Q}$-und Preset-Leitungen verzichtet, so dass sich die Anzahl der PINs reduziert und die Schaltung vereinfacht.

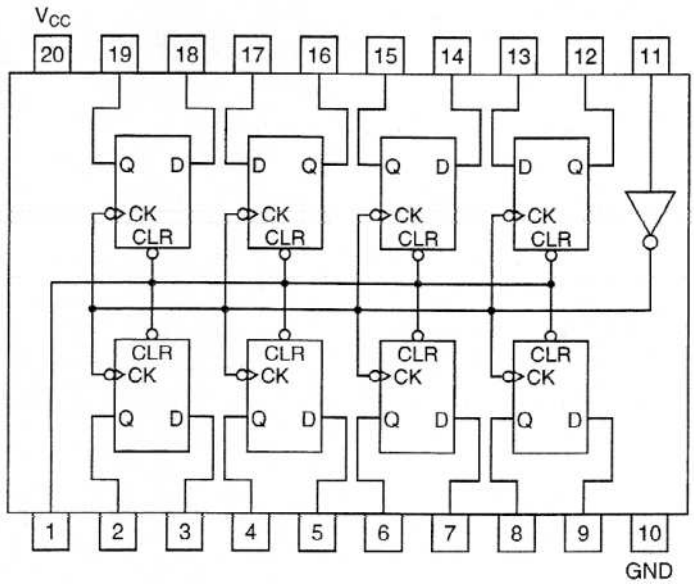


Abbildung 7.15: 8–Bit–Flip-Flop

Die Frage ist, warum in der Schaltung Flip-Flops *mit fallender Flanke* verwendet werden, wenn gleichzeitig der Takt an Pin 11 negiert wird. Dies liegt daran, dass möglicherweise nicht genug Strom bereitsteht um für alle 8 Flip-Flops eine ausreichende steigende Flanke als Taktgeber bereitzustellen.

Insgesamt erhalten wir ein zusammenhängendes 8–Bit–Register. Zwei solche Chips können durch Verbindung der jeweiligen PINs 1 und 11 zu einem 16-Bit-Register zusammengeschlossen werden.

7.4 Realisierung von Speicherchips

Um größere Speicher zu konstruieren, ist eine andere Organisation erforderlich, mit deren Hilfe man *einzelne Wörter adressieren* kann. Dazu wollen wir uns die Schaltung in Abbildung 7.16 anschauen.

Dabei handelt es sich um einen Speicher für vier 3–Bit–Wörter, bei dem jede Operation ein volles 3–Bit–Wort schreibt oder liest. Die gesamte *Speicherkapazität* beträgt *12 Bit*, was kaum mehr als beim 8–Bit–Register ist. Das Design läßt sich jedoch leicht auf große Speicher erweitern (ist gut skalierbar) und benötigt weniger PINs.

Realisirung:

1. CS: Der Chip Select steuert der Wahl von dem genauen Chip. Bei Wert 0 wird, unabhängig von allen anderen Eingaben, nichts gespeichert und die Ausgabe wird 0 sein.
2. RD: RD kommt von READ(Lesen). Falls RD = 1, dann wird es gelesen, sonst wird geschrieben.

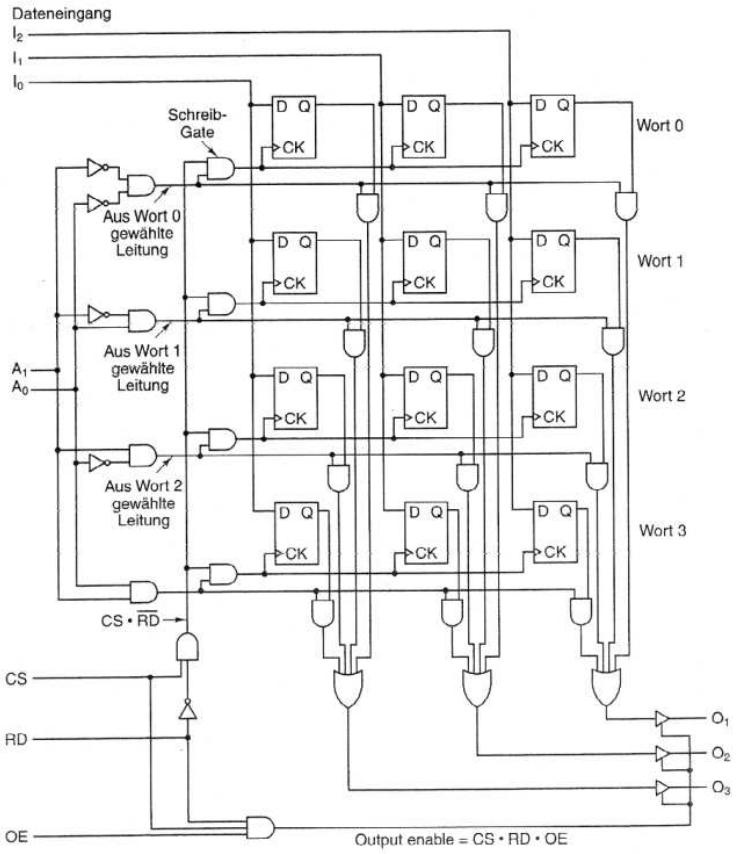


Abbildung 7.16: Speicher für vier 3-Bit-Wörter

3. OE: Output Enabled (Ausgabe freigegeben) ist nur dann 1, wenn der Baustein bereit ist ein Antwort zu geben. Falls OE 0 ist, dann wird das Ausgabe auch 0 sein.

4. A_0 und A_1 zeigen, auf welches Wort wird geschrieben, bzw. von welchem Wort wird gelesen.

A_0	A_1	Wort
0	0	Wort 0
1	0	Wort 1
0	1	Wort 2
1	1	Wort 3

5. I_0, I_1 und I_2: Dateneingänge, biem Schreibmodus werden diese Bits auf den entsprechenden Platz geschrieben

6. O_0, O_1 und O_2: Datenausänge, biem Lesemodus werden diese Bits die gelesenen und als Antwort gegeben, falls CS.RD.OE = 1

Zusammenfassung:

1. Schreiben: $CS * \overline{RD}$

2. Lesen: CS.RO.OE

3. Speicherdauer bis über $CS * \overline{RD}$, d.h. beliebig lange

4. Strom Masse 13 Pin's

5. Speicherinhalt hat Adresse

6. gut Skalierbar

Wir wollen die Schaltung nun ein wenig genauer betrachten. Insgesamt haben wir drei Dateneingänge I_0, I_1 und I_2, zwei Adreßeingänge A_0 und A_1 sowie drei Steuerungseingänge CS (Chip Select) für die Auswahl genau dieses Chips, RD (Read) für die Anzeige, dass gelesen wird im Falle RD=1 und sonst bei RD=0 geschrieben wird, und *OE (Output Enable)* für die Bereitschaft von ggf. anderen Bauteilen einen Output zu empfangen. Demzufolge ist eine **Schreiboperation** möglich bei

$$CS \cdot \overline{RD}$$

und eine **Leseoperation** kann stattfinden bei

$$CS \cdot RD \cdot OE$$

Mit den drei Datenausgängen $0_0, 0_1$ und 0_2 kann der Speicherbaustein inclusive Strom und Masse in einer *14–PIN–Einheit* untergebracht werden, wie wir sie beim 2–Bit–Register betrachtet hatten, während beim 8–Bit–Register bereits 20 PINs erforderlich waren.

Die Funktionsweise des Speichers geht zunächst bei den *Adressen* von einem *Decoder* aus, der durch die vier AND–Gatter unmittelbar hinter den Eingaben von A_0 und A_1 realisiert ist.

Wenn zudem der Schreibmodus angesteuert wird, wird das entsprechende *Schreibgate* aktiviert, das die durch A_0 und A_1 codierte Speicherzelle adressiert. Bei den 3 Flip-Flops der Speicherzelle wird ein Taktsignal angelegt, so dass die 3 bei jeweils Eingang D anliegenden Inputwerte gespeichert werden.

Im Lesemodus wird jeweils der Ausgang Q der Flip-Flops durch ein AND-Gatter mit einem Signal von Dekodierer verknüpft, so dass die Speicherinhalte der ausgewählten Adresse von O_1, O_2 und O_3 anliegen.

Angenehm an diesem Speicher ist, dass er von der Größe her *mühelos erweiterbar* ist. Möchten wir vier Wörter á 8 Bit speichern, so müssen lediglich fünf weitere Spalten von je 4 Flip-Flops und je 5 Eingangs- sowie Ausgangsleitungen hinzugefügt werden. Bei einer Verdoppelung der Wortanzahl ist lediglich eine Adreßzeile hinzuzufügen und der Decoder entsprechend der Stelligkeit anzupassen.

Da sich die *integrierte Schaltkreistechnik* gut zur Herstellung von Chips eignet, deren interne Struktur ein wiederkehrendes, zweidimensionales Muster aufweist, sind *Speicherchips* ein *idealer* Anwendungsbereich. Im Zuge der Verbesserung dieser Technik steigt die *Bitanzahl, die man auf einen Chip packen kann alle 18 Monate um den Faktor zwei*. Diese Bitverdoppelung in $1\frac{1}{2}$ Jahren nennt man auch **Mooresches Gesetz**.

7.5 Random Access Memory (RAM)

Die gerade betrachteten Speicherbausteine und –chips können lesen und schreiben. Sie werden als RAMs (Random Access Memories) bezeichnet, was ein etablierter, wenn auch unglücklicher Begriff ist um auszudrücken, dass die Zugriffszeit für alle Speicherzellen sowohl beim Lesen wie auch beim Schreiben in etwa gleich lang ist (random access).

RAMs gibt es in zwei Varianten: statisch und dynamisch

Statische RAMs (SRAMs)

werden intern mit Schaltungen realisiert, die strukturell unserem *Flip-Flop-Speicherbaustein* entsprechen. Diese Speicher haben die Fähigkeit, ihre *Inhalte zu behalten, solange der Strom fließt:* Sekunden, Minuten, Stunden oder gar Tage. SRAMs sind *sehr schnell*. Die typischen Zugriffszeiten betragen nur wenige Nanosekunden. Aus diesem Grund sind SRAMs als *Cache-Speicher* der Ebene 2 sehr beliebt.

Dynamische RAMs (DRAMs)

bestehen demgegenüber nicht aus Flip-Flops, sondern aus einer Reihe von *Zellen*, die *jeweils einen Transistor und einen winzigen Kondensator enthalten*. Kondensatoren können be– und entladen werden, so dass sich Nullen und Einsen speichern lassen. Da elektrische Ladung jedoch zu Kriechverlusten neigt, muß jedes Bit in einem DRAM *alle paar Millisekunden aufgefrischt*, d.h. nachgeladen werden.

Um die Auffrischung muß sich eine *externe Logik* kümmern, so daß DRAMs *komplexere Schnittstellen* erfordern als SRAMs. Dafür haben DRAMs jedoch größere Kapazitäten und eine sehr hohe Dichte, also viele Bits pro Chip. Aus diesem Grund werden *Hauptspeicher* in der Regel als DRAM realisiert und optimalerweise *mit SRAM Cache kombiniert*.

Die große Kapazität hat jedoch ihren Preis: DRAMs sind viel *langsamer* als SRAMs. Zig Nanosekunden werden als Zugriffszeit benötigt.

DRAMs sind in verschiedenen Typen erhältlich: Der ältere **FPM-DRAM** (Fast Page Model) basiert auf der Anordnung der RAM-Speicherzellen in einer Matrix bzw. Tabelle. Dadurch läßt sich die Adressierung sehr einfach gestalten. Nach jedem Takt wird zwischen der Angabe von Spaltenadresse (RAS, Row Address Signal) und Zeilenadresse (CAS, Column Adress Signal) hin und her geschaltet.

Die Optimierung dieses RAS/CAS - Verfahrens erlaubt einen bis zu dreimal schnelleren Zugriff auf die Daten als bei herkömmlichen DRAMs. Während eines fortlaufenden Speicherzugriffs wird das Anlegen der immer gleichen Zeilenadresse gespart. Es genügt, die Zeilenadresse einmal anzugeben und die jeweilige Spaltenadresse anzugeben und auszuwerten. Der Zugriff erfolgt erheblich schneller. Diese Speicherart wird zunehmend durch den **EDO-DRAM** (Extended Data Output) abgelöst. Hinter der Bezeichnung EDO steckt eine Technik, mit der die Spannung in den Kondensatoren länger aufrecht erhalten bleibt. Dadurch kann der Speicherzustand einer Speicherzelle länger beibehalten werden. Damit stehen die Daten am Ausgang länger bereit und die Häufigkeit des Speicherrefreshs verringert sich. Während die Daten noch gelesen werden, wird bereits die nächste Adresse an den Speicherbaustein angelegt. Bei aufeinanderfolgenden Lesezugriffen wird somit eine höhere mittlere Lesegeschwindigkeit erreicht.

Dadurch ist zwar eine einzelne Speicherreferenz nicht schneller, allerdings verbessert sich die Speicherbandbreite, d.h. es können mehr Wörter pro Zeiteinheit durchgeschleust werden.

Während FPM- und EDO- Chips asynchron – also ohne einheitlichen Taktgeber – arbeiten, ist **SDRAM** (Synchronous DRAM) eine Mischung aus statischem und dynamischem RAM. Er wird durch einen einzelnen, synchronen Taktgeber gesteuert. Intern besteht der SDRAM aus zwei Speicherbänken. Der Zugriff erfolgt abwechselnd, so dass die benötigte Erholzeit zwischen den Zugriffen scheinbar entfällt. Einen zusätzlichen Geschwindigkeitsvorteil bringt das Pipeline–Verfahren. Darüber hinaus ist SDRAM programmierbar und so universell einsetzbar. SDRAMs werden häufig *in großen Cache-Speichern* benutzt und gelten als vielversprechende Technologie für Hauptspeicher der Zukunft, auch wenn aktuell übliche Anwendungen eher im Bereich der Mainframes und PDA's liegen.

Eine weitere Verbesserung hinsichtlich der Datenrate ermöglicht der **DDR-SDRAM** (Double Data Rate SDRAM). Das Verfahren ist ganz einfach: es wird sowohl die aufsteigende als auch die absteigende Flanke des Takts für Speicherzugriffe genutzt, so dass sich eine Bandbreitenverdopplung ergibt.

In einer weiteren Entwicklungsstufe wurde der **DDR–II–SDRAM** konzipiert, der pro Taktschritt sogar auf 4 Datenworte zugreifen kann.

Statt DDR-SDRAM verwendet man häufig auch nur die Abkürzung **DDRAM** (Double Data Random Access Memory). Während dieses Wiederaufladens hat die CPU keinen Zugriff auf den DRAM. Dadurch ist der DRAM langsamer als der SRAM.

RAMs haben den entscheidenden Nachteil, dass die **Daten bei ausgeschaltetem Strom verloren gehen**. In vielen Anwendungen wie z.B. Spielzeug, Elektrogeräten und Autos, PDAs und Handys sollten das Programm und ein Teil der Daten *auch bei ausgeschaltetem Strom gespeichert bleiben*. Außerdem werden weder das Programm noch die Daten nach der Installation jemals geändert.

Diesen Anforderungen werden die **ROMs** (Read-Only Memories) gerecht, die *weder absichtlich noch aus Versehen geändert oder gelöscht werden können*. Die Daten werden bei der Herstellung auf der Oberfläche eingebrannt. Bei einem Änderungswunsch muß man den gesamten Chip ersetzen.

ROMs sind zudem *viel billiger* als RAMs, wenn man sie in großen Mengen herstellt, weil sich dann die Kosten für eine notwendige Maske armotisieren. Problematisch ist jedoch, dass zwischen Herstellung und Auslieferung oft viele Wochen vergehen und Daten dann mitunter *unaktuell* sind.

Um es den Unternehmen zu vereinfachen, neue ROM-basierte Produkte zu entwickeln, wurde der **PROM** (Programmable ROM) erfunden. Dieser Chips arbeitet wie ein ROM, das man *einmal* im Feld programmieren kann durch Durchbrennen winziger Sicherungen wodurch sich die Vorlaufzeit erheblich verkürzt.

Das nachfolgende entwickelte **EPROM** (Erasable PROM) kann man *sowohl progammieren als auch löschen*. Wird das Quarzfenster eines EPROMs 15 Minuten lang einem *starken ultravioletten Licht ausgesetzt*, sind alle Bits auf 1 gesetzt. Im Falle eines iterativen Designzyklus kann ein EPROM dann weitaus *wirtschaftlicher als ein PROM sein*, da es wiederverwendbar ist. Von der Organisation her entsprechen EPROMs den *statischen RAMs*.

Eine nochmalige Verbesserung ist der **EEPROM** (Extended EPROM), der statt ultraviolettem Licht durch Impulse gelöscht werden kann. Er kann vor Ort byteweise gelöscht und neu programmiert werden, ohne dass man ihn aus der Schaltung herausnimmt und muß auch nicht – wie der EPROM – in ein spezielles EPROM–Programmiergerät eingelegt werden. Dafür ist er allerdings wesentlich *kleiner* (nur $\frac{1}{64}$) und *langsamer* (nur halb so schnell) als EPROMs. Wäre der Vorteil der Nicht-Flüchtigkeit nicht gegeben, so könnten EEPROMs auch nicht mit DRAMs oder SRAMs konkurrieren, da sie zehnmal langsamer, 100 mal kleiner und viel teurer sind.

Schließlich wollen wir noch den **Flash-Speicher** (Flash Memory) betrachten. Der Flash-Speicher kann *blockweise gelöscht* und *wieder programmiert* werden, auch ohne Herausnehmen aus der Fassung. Anwendungsgebiet ist beispielsweise das Speichern von Bildern in Digitalkameras, wozu Flash-Speicher als kleine gedruckte Schaltkarte mit etlichen MByte als eine Art „Film“ zur Verfügung steht.

Eines Tages lösen Flash-Speicehr möglicherweise Platten ab. Dies wäre angesichts der Zugriffszeit von 100 ns eine Verbesserung. Problematisch ist jedoch, dass sie verschleißen und nach etwa 10.000 Löschungen unbrauchbar sind, während Platten unabhängig von der Anzahl der Lösch- und Schreibvorgänge Jahre überdauern.

Eine Gegenüberstellung der Speichermedien wird abschließend in Tabelle 7.1 vorgenommen.

Speichertyp	**Nutzungsart**	**Löschung**	**Bytes änderbar**	**flüchtig**	**übliche Verwendung**
SRAM	Lesen/Schreiben	Elektrisch	Ja	Ja	Level–2–Cache
DRAM	Lesen/Schreiben	Elektrisch	Ja	Ja	Hauptspeicher
ROM	Nur Lesen	Nicht möglich	Gar nicht	Nein	Großvolumige Geräte
PROM	Nur Lesen	Nicht möglich	Einmal programmierbar, aber nicht löschbar	Nein	Kleinvolumige Geräte
EPROM	Vorweigend Lesen, Schreiben sehr aufwändig	In spezieller Kammer mittels UV-Licht	Wiederverwendbar durch Neuprogrammierung	Nein	Herstellung von Geräteprototypen
EEPROM	Vorwiegend Lesen, Schreiben aufwändig	Elektrisch mittels Impulsen, innerhalb der Schaltung byteweise löschbar	Mit speziellem Gerät	Nein	Herstellung von Geräteprotoypen
Flash-Speicher	Lesen/Schreiben	Elektrisch blockweise löschbar	Blockweise änderbar	Nein	Speichermedium für Digitalbilder

Tabelle 7.1: Gegenüberstellung von RAM und ROM-Typen

8

Darstellung von Speicherinhalten

Der Speicher (Memory) ist der Teil des Computers, in dem die Programme und Daten gehalten werden. Im Zeitalter der Digitalrechner mit gespeicherten Programmen muß der Speicher von den Prozessoren gelesen und beschrieben werden können.

Zunächst sollen die **Grundlagen kurz wiederholt** werden:

Wir wissen bereits, daß die Grundeinheit des Speichers die binäre Ziffer ist, genannt **Bit**. Mit dieser Untersuchung von nur 2 Werten liegt – physikalisch gesehen – die zuverlässigste Methode zur Kodierung digitaler Daten vor.

Speicher bestehen aus einer Reihe von **Zellen**. Jede Zelle hat eine Adresse, auf die das Computerprogramm Bezug nehmen kann. Hat ein Speicher *n Zellen*, verfügt er über die *Adressen 0 bis* $n-1$. *Alle Zellen* in einem Speicher *enthalten die gleiche Bitanzahl*. Besteht eine Zelle aus *k Bits*, so kann sie eine von 2^k *verschiedenen Bitkombinationen* aufnehmen. Und: beträgt die *Länge der Adresse m Bit*, so können 2^m *Zellen direkt adressiert werden*. Damit hat die Anzahl der Bits einer Adresse einen direkten Zusammenhang zur maximalen Anzahl von direkt adressierbaren Zellen im Speicher aber hängt nicht von der Anzahl der Bits pro Zelle ab.

Die übliche Zellgröße ist derzeit 8 Bit, d.h. ein Byte. Diese Größe geht ursprünglich auf die Darstellung der ASCII-Zeichen zurück.

8.1 Zeichencodes

Jeder Computer verfügt über einen Zeichensatz, mit dem er arbeitet. Dieser umfaßt zumindest die 26 Groß- und 26 Kleinbuchstaben, die Ziffern 0 bis 9 und Symbole wie Leerzeichen, Punkt, Komma, +, -, @, ENTER, u.s.w.

Um *diese Zeichen* im Computer darstellen zu können, wird ihnen *jeweils eine Zahl zugeordnet*, z.B. $a = 1, b = 2, \ldots, z = 26, + = 27, - = 28$ usw. *Diese Zuordnung von Zeichen zu natürlichen*

Zahlen ist der Zeichencode (Character Code). *Computer, die miteinander kommunizieren, müssen den gleichen Code verwenden. Deshalb wurde dieser Code standardisiert.*

8.1.1 ASCII

ASCII steht für American Standard Code for Information Interchange. Jedes ASCII-Zeichen, hat 7 Bit, was insgesamt $2^7 = 128$ Bitmuster ergibt. Hexadezimal werden die Zeichen von 0 bis 7F durchnummeriert, was die $8 \cdot 16 = 128$ Zeichen ergibt.

Die Codes 0 bis 1F sind Steuerzeichen, die nicht ausgedruckt werden. Zum großen Teil dienen sie der Datenübertragung. So kann eine Nachricht bestehen aus:

SOH <Header> STX <Text> ETX EOT

Dabei bedeutet SOH „Start of Header“, STX „Start of Text“, ETX „End of Text“und EOT „End of Transmission“.

Heute werden die in der Praxis über Telefonleitungen und Netze gesendeten Nachrichten aber anders formatiert, so dass die *ASCII-Steuerzeichen kaum noch benutzt werden*.

Dieser überwiegend in den USA entwickelte Zeichensatz hat jedoch den *Nachteil*, dass französische *Akzente*, deutsche *Umlaute u.s.w. nicht realisierbar sind.*

Der **erste Versuch**, *ASCII zu erweitern*, war der IS 646, wodurch *weitere 128 Zeichen realisierbar wurden*. Dadurch entstand ein 8-Bit-Code mit der Bezeichnung **Latin-1**, der überwiegend lateinische Buchstaben mit Akzenten hinzunahm.

Mit dem **zweiten Versuch** IS 8859 namens **Code Page** (Codeseite) werden bestimmte Sprachgruppen unterstützt, wobei IS 8859-1 das Latin-1 ist.

IS 8859-2 versorgt osteuropäische Sprachen wie Polnisch, Tschechisch und Ungarisch mit einem erweiterten Lateinischen Alphabet.

IS 8859-3 enthält Zeichen für Türkisch, Maltesisch, Esperanto u.s.w.

Wichtig ist dabei nur, dass die Software weiß, mit welcher Codeseite sie gerade arbeitet. Außerdem kann man keine Sprachen von verschiedenen Codeseiten miteinander kombinieren. Und: Japanisch und Chinesisch werden ganz ignoriert.

8.1.2 Unicode

Einige Computerunternehmen beschlossen, ein Konsortium zu bilden und mit der *Schaffung eines neuen Systems* als internationalen Standard IS 10646 *namens Unicode* zu beauftragen. *Unicode wird heute von verschiedenen Programmiersprachen (z.B. Java), verschiedenen Betriebssystemen (z.B. Windows NT) und zahlreichen Anwendungsprogrammen unterstützt*.

In Unicode wird jedem Zeichen und Symbol ein eindeutiger permantenter 16-Bit-Wert zugeordnet, der als **Code Point** (Codepunkt) bezeichnet wird. Damit gibt es $2^{16} = 65.536$ Codepunkte. Da die Sprachen der Welt jedoch zusammen ca. 200.000 Symbole benötigen (50.000 Kanji-Zeichen dargestellt durch Han-Ideogramme allein für ein halbwegs vollständiges japanisches Wörterbuch), sind Codepunkte eine knappe Ressource.

Hex	Name	Bedeutung	Hex	Name	Bedeutung
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of Text	12	DC2	Device Control 2
3	ETX	End Of Text	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELl	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	Hat	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

Hex	Zeichen	Hex	Zeichen	Hex	Zeichen	Hex	Zeichen	Hex	Zeichen	Hex	Zeichen
20	(Leer-zeichen)	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Abbildung 8.1: Der ASCII-Zeichensatz

Die Hälfte aller Codepunkte ist zur Zeit aufgebraucht. Dabei entspricht *0 bis 255 dem Latin–1–Zeichencode, um die Konvertierung zwischen ASCII und Unicode zu erleichtern* und die Akzeptanz von Unicode zu erhöhen.

Für Latein sind ingesamt 336 Codepunkte zugewiesen, für Griechisch 144, für Kyrillisch 256, für die im Chinesischen und Japanischen gebrauchten Han-Ideogramme 20.992 Codepunkte, für die koreanischen Hangul-Silbenzeichen 11.156.

Um die Verschwendung von Codepunkte zu verhindern hat *jedes von insgesamt 112 diakritischen Zeichen (Akzente, Ö-Punkte, . . .) seinen eigenen Codepunkt. Es ist dann Sache der Software, dieses mit anderen Elementen zu neuen Zeichen zu kombinieren*. Ferner werden für Zeichen die nahezu gleich ausschauen aber eine unterschiedliche Bedeutung haben auch dieselben Codepunkte verwendet.

Ferner gibt es 112 Satzzeichen, 48 Währungssymbole, 256 mathematische Symbole, 96 geometrische Formen und 192 Ornamente. 6400 Codepunkte wurden für den lokalen Gebrauch freigehalten, damit sich die Benutzer Sonderzeichen für Spezialanwendungen ausdenken können.

Zwei Probleme gibt es bei Unicode.

1. Währende das lateinische Alphabet alphabetisch angeordnet ist, liegen z.B. die Han-Schriftzeichen *nicht in der Reihenfolge des Wörterbuchs vor*. D.h. sie können nicht durch einfache Vergleiche der Unicode-Werte alphabetisch sortiert werden. Ein japanisches Programm benötigt dazu *externe Tabellen*.

2. Ständig entstehen *neue Wörter* wie z.B. Applet, Gigabyte, Smiley. Im japanischen werden dazu *neue Codepunkte* benötigt.

Daneben muß das Unicode-Konsortium ständig entscheiden, welchen Anträgen auf Zuweisung von Unicode stattgegeben werden soll, z.B. Blindenschrift und weitere Codepunkte für japanische Zeichen.

8.2 Byteanordnung

Die Bytes eines Wortes können von links nach rechts oder rechts nach links durchnummeriert werden. Wir wollen zunächst die Numerierung von links nach rechts betrachten.

Diese Darstellung wird bei SPARC– oder IBM–Großrechnern verwendet. *Da die Zählung am großen, d.h. höherwertigen Ende beginnt*, wird diese Darstellung in Anlehnung an Jonathan Swift's "Gullivers Reisen" als **Big Endian** bezeichnet.

Dem steht das **Little Endian**–Format gegenüber. Dieses Format ist beispielsweise in der Intel–Familie gebräuchlich.

Beachte, dass die numerischen Werte in beiden Fällen mit dem niedrigen Bits ganz rechts dargestellt werden. Das Wort, das die ganze Zahl 1 erhält hätte in beiden Fällen die Adresse 8. Probleme bereitet die Kombination von Zahlen und Zeichenketten, insbesondere wenn eine Little Endian–Maschine einen Datensatz über das Netz an eine Big Endian–Maschine schickt. Hier gibt es keine einfache Lösung. Notfalls muß bei jedem Datensatz ein Header eingefügt werden, der Art und Länge der nachfolgenden Daten ausgibt.

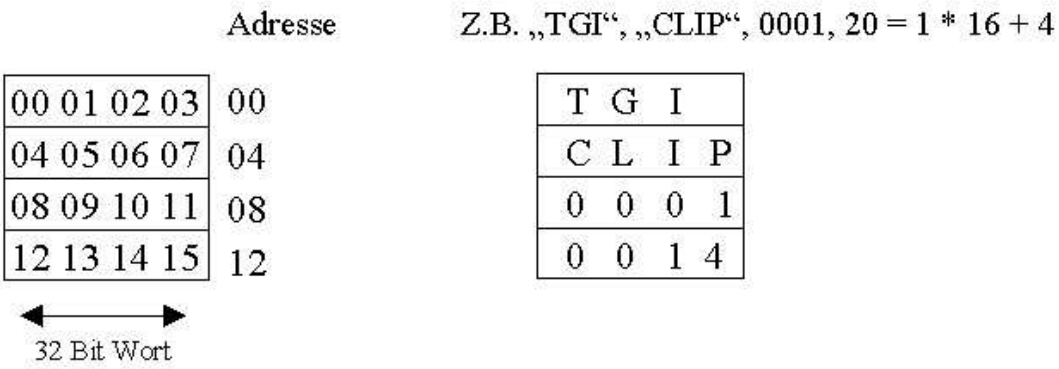


Abbildung 8.2: Big–Endian–Darstellung

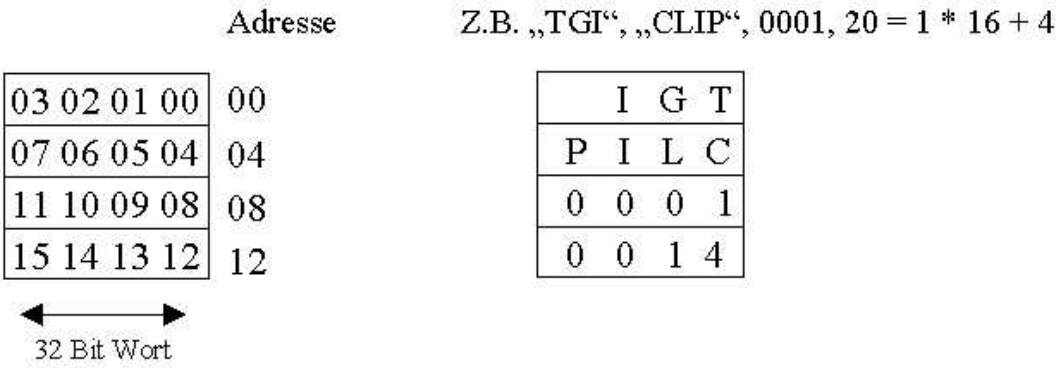


Abbildung 8.3: Little–Endian–Darstellung

8.3 Darstellung von Arrays

Nun soll analog die Speicherung von Datentypen behandelt werden, die mehrdimensionalen ARRAY-Deklarationen in höheren Programmiersprachen entsprechen.

Eine mehrdimensionale Reihung sei (mit r als Dimension des Feldes)

```
1   VAR a: ARRAY[ug1 .. og1, ug2 .. og2, ..., ugr .. ogr] OF INTEGER
```

Wiederum benötigen wir eine Speicherabbildungsfunktion für die Adressierung.

Beispiel 8.1*:* Zunächst betrachten wir $r = 2$ und setzen $ug = m$, $og = n$. Dann erhalten wir als hochsprachliches Konstrukt

```
1 VAR a: ARRAY[m1 .. n1, m2 .. n2] OF INTEGER
```

Wir erhalten also eine Matrix

$$\begin{pmatrix} m_1m_2 & m_1(m_2+1) & \cdots & m_1n_2 \\ (m_1+1)m_2 & (m_1+1)(m_2+1) & \cdots & (m_1+1)n_2 \\ \vdots & \vdots & & \vdots \\ n_1m_2 & n_1(m_2+1) & \cdots & n_1n_2 \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n_2-m_2} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n_2-m_2} \\ \vdots & \vdots & & \vdots \\ a_{n_1-m_1,0} & a_{n_1-m_1,1} & \cdots & a_{n_1-m_1,n_2-m2} \end{pmatrix}$$

Die Anordnung der Feldelemente wird im Arbeisspeicher „linearisiert“, d.h. die Elemente werden lexikographisch abgespeichert:

$$\begin{array}{ccccccccc} & (m_1,m_2) & < & (m_1,m_2+1) & < & \ldots & < & (m_1,n_2) \\ < & (m_1+1,m_2) & < & (m_1+1,m_2+1) & < & \ldots & < & (m_1+1,n_2) \\ & \vdots & & \vdots & & & & \vdots \\ < & (n_1,m_2) & < & (n_1,m_2+1) & < & \ldots & < & (n_1,n_2). \end{array}$$

Das Array wird also zeilenweise abgespeichert.

Es ergeben sich einige zu betrachtende Fragen bei der Adreßberechnung eines beliebigen Elements x:

1. Wie ist die Adresse des ersten Elements?
2. Wieviele Vorgänger hat ein Element x?
3. Welche Größe hat der Operand ($|LOP| =$?).

Nun kehren wir zur Betrachtung des allgemeinen Falles für $r \in \mathbb{N}$. Sei $a[i_1, i_2, \ldots, i_r]$ ein beliebiges Element aus dem Array mit $ug_j \leq i_j \leq og_j$ für $1 \leq j \leq r$ zurück. Dann erhalten wir die folgende Speicherabbildungsfunktion:

$$\text{adresse}(a[i_1, i_2, \ldots, i_r]) = \text{adresse}(a[ug_1, \ldots, ug_r]) + \text{Anzahl Vorgänger}(a[i_1, i_2, \ldots, i_r]) \cdot |LOP|$$

Obwohl die Funktion auf den ersten Blick recht einfach erscheint, ergibt sich das Problem, wieviele Vorgänger $x := a[i_1, \ldots, i_r]$ in der lexikographischen Abspeicherung hat:

1. Man betrachte den ersten Index. Dann sind alle $a[j_1, j_2, \ldots, j_r]$ mit $j_1 < i_1$ Vorgänger von x.

   j_1 kann in diesem Fall die Werte $ug_1, ug_1 + 1, \ldots, i_1 - 1$, also $i_1 - ug_1$ verschieden Werte, annehmen.

   Somit erhalten wir als Anzahl der Vorgänger in diesem Fall

   $$(i_1 - ug_1) \cdot s_2 \cdot s_3 \cdot \ldots \cdot s_r$$

   mit $\boxed{s_i = og_i - ug_i + 1 \text{ (Länge der } i\text{-ten Felddimension).}}$

2. Man betrachte nun den zweiten Index. Dann sind alle $a[i_1, j_2, \ldots, j_r]$ mit $j_2 < i_2$ Vorgänger von x.

   Wiederum kann j_2 die Werte $ug_2, ug_2 + 1, \ldots, i_2 - 1$ annehmen, also $i_2 - ug_2$ verschiedene.

   Somit erhalten wir als Anzahl der Vorgänger von x in diesem Fall

   $$(i_2 - ug_2) \cdot s_3 \cdot \ldots \cdot s_r.$$

3. Dieser Vorgang wird fortgesetzt bis zum r-ten Index. Dann sind alle $a[i_1, i_2, \ldots, i_{r-1}, j_r]$ mit $j_r < i_r$ Vorgänger von x.

   Wiederum kann j_r die Werte $ug_r, ug_r + 1, \ldots, i_r - 1$ annehmen, also $i_r - ug_r$ verschiedene.

   Somit erhalten wir als Anzahl der Vorgänger von x in diesem Fall

   $$i_r - ug_r.$$

Somit hat $a[i_1, i_2, \ldots, i_r]$ also

$$(i_1 - ug_1) \cdot s_2 \cdot s_3 \cdot \ldots \cdot s_r + (i_2 - ug_2) \cdot s_3 \cdot \ldots \cdot s_r + \ldots + (i_{r-1} - ug_{r-1}) \cdot s_r + (i_r - ug_r)$$

Vorgänger. Dies ergibt einfacher:

$$\begin{aligned} & i_1 \cdot s_2 \cdot s_3 \cdot \ldots \cdot s_r + \ldots + i_{r-1} s_r + i_r - ug_1 \cdot s_2 \cdot s_3 \cdot \ldots \cdot s_r - ug_{r-1} s_r - ug_r \\ & \qquad =: h(i_1, i_2, \ldots, i_r) - h(ug_1, ug_2, \ldots, ug_r) \\ & = (((i_1 \cdot s_2 + i_2) s_3 + i_3) \cdot s_4 + \ldots \cdot s_r) + i_r - \left[(((ug_1 \cdot s_2 + ug_2) \cdot s_3 + ug_3) \cdot s_4 + \ldots \cdot s_r) + ug_r\right] \end{aligned}$$

Dabei ist $h(i_1, i_2, \ldots, i_r)$ variabel, während $h(ug_1, ug_2, \ldots, ug_r)$ konstant ist bezüglich $i_1, \ldots, i_r$.

Damit ergibt sich

$$\text{adr}(a[i_1, \ldots, i_r]) = \text{adr}(a[ug_1, ug_2, \ldots, ug_r]) + [h(i_1, \ldots, i_r) - h(ug_1, \ldots, ug_r)] \cdot |LOP|.$$

Sortiert man diese Gleichung nach konstanten und variablen Termen, so erhält man:

$$\text{adr}(a[i_1, \ldots, i_r]) = \text{adr}(a[ug_1, ug_2, \ldots, ug_r]) - h(ug_1, \ldots, ug_r) \cdot |LOP| + h(i_1, \ldots, i_r) \cdot |LOP|$$

Nimmt man nun die fiktive Anfangsadresse $a[0, 0, \ldots, 0]$, also $ug_1 = ug_2 = \ldots = ug_r = 0$ an, so vereinfacht sich die Formel schließlich zu

$$\text{adr}(a[i_1, \ldots, i_r]) = \text{adr}(a[0, 0, \ldots, 0]) + h(i_1, \ldots, i_r) \cdot |LOP|.$$

Beispiel 8.2*:* Sei folgendes Feld gegeben `a: ARRAY[1..7, 2..8,0..3] OF INTEGER`. Dann hat `a` die Dimension $r = 3$ und die Kennung `W`, also ist $|LOP| = 4$. Sei weiterhin die Anfangsadresse von `a` $a[1,2,0] = 2100$ (absolute Anfangsadresse , AA).

Ferner ist $s_1 = og_1 - ug_1 + 1 = 7$, $s_2 = og_2 - ug_2 + 1 = 7$ und $s_3 = og_3 - ug_3 + 1 = 4$.

Wir wollen nun die fiktive Anfangsadresse adr($a[0,0,0]$) berechnen: Die obige Gleichung zur Berechnung von adr($a[i_1,\ldots,i_r]$) mit Hilfe der fiktiven Anfangsadresse läßt sich umformen zu (für $i_i = ug_i$ im zweiten Schritt):

$$\begin{aligned}\text{adr}(a[0,0,0]) &= \text{adr}(a[i_1,\ldots,i_r]) - h(i_1,\ldots,i_r)\cdot|LOP| \\ &= \text{adr}(a[ug_1,\ldots,ug_r]) - h(ug_1,\ldots,ug_r)\cdot|LOP| \\ &= 2100 - [((ug_1\cdot s_2)+ug_2)\cdot s_3 + ug_3]\cdot|LOP| \\ &= 2100 - [((1\cdot 7)+2)\cdot 4+0]\cdot 4 \\ &= 2100 - 36\cdot 4 = 1956.\end{aligned}$$

Mit Hilfe der fiktiven Anfangsadresse lassen sich nun die Adressen der Feldelemente leicht berechnen:

Beispiel 8.3*:* Es soll die Adresse des Elements $a[7,8,3]$ berechnet werden:

$$\begin{aligned}\text{adr}(a[7,8,3]) &= \text{adr}(a[0,0,0]) + h(7,8,3)\cdot|LOP| \\ &= 1956 + 924 = 2880.\end{aligned}$$

Die Adresse des letzten Feldelements ist also 2880.

Als Probe soll schließlich noch die Adresse des ersten Feldelements, also die absolute Anfangsadresse des Feldes, berechnet werden:

$$\begin{aligned}\text{adr}(a[1,2,0]) &= \text{adr}(a[0,0,0]) + h(1,2,0)\cdot|LOP| \\ &= 1956 + 144 = 2100.\end{aligned}$$

Bemerkungen zu mehrdimensionalen Reihungen

1. Die fiktive Anfangsadresse $a[0,\ldots,0]$ ist tatsächlich *fiktiv*, d.h. an dieser Adresse wird im allgemeinen kein Feldelement stehen (außer wenn alle Felddimensionen bei 0 beginnen).

2. Alternativ kann auch die absolute Anfangsadresse (z.B. adr($[1,2,0]$)) als Basisadresse benutzt werden, zu der relativ die anderen Feldelemente berechnet werden.

3. Zur internen Beschreibung eines Feldes wird in der SPIM ein sogenannter **Felddeskriptor** verwendet. Dieser kann verschiedene Darstellungen besitzen, hier soll die in Abbildung 8.4 auf der nächsten Seite gezeigte Struktur verwendet werden.

   Es ist möglich, den Felddeskriptor direkt vor das Feld zu schreiben, dann ist die absolute Anfangsadresse nicht notwendigerweise zu speichern.

r	ug_1 s_1	ug_2 s_2	$\cdots$	ug_r s_r	LOP	AA	$A0$

Abbildung 8.4: Beschreibung des Felddeskriptors. Dabei ist AA *die absolute Adresse* des ersten Feldelements und $A0$ *die fiktive Anfangsadresse.*

8.4 Fehlererkennung und –korrektur

Wir wissen bereits, dass Zellen derzeit 8 Bit groß sind. Dies resultiert aus der ASCII–Darstellung. Um 128 Zeichen darstellen zu können, werden 7 Bit benötigt. Zuzüglich einem **Paritätsbit (Parity Bit)** ergibt sich ein Byte als Zellgröße.

Beispiel 8.4 *(ASCII–Darstellung):* - "A" hat die ASCII–Darstellung "65"

- "f" hat die ASCII–Darstellung "102"
- "8" hat die ASCII–Darstellung "56"
- "+" hat die ASCII–Darstellung "43"

Wozu sind **Paritätsbits** nötig?
In Computerspeichern können aufgrund von Spannungsspitzen in der Stromleitung oder aus anderen Gründen gelegentlich Fehler auftreten. Als Schutz vor solchen Fehlern setzen die meisten Speicher Codes für die Fehlererkennung und ggf. auch **Fehlerkorrektur** ein. Bei diesen Codes werden auf bestimmte Weise jedem Speicherwert zusätzliche Bits hinzugefügt. Beim Lesen eines Wortes aus dem Speicher werden diese Zusatzbits geprüft, um festzustellen, ob ein Fehler aufgetreten ist.

Ein Speicherwort bestehe im folgenden aus **m- Datenbits**, denen wir r redundante **Prüfbits** hinzufügen. Dies ergibt eine n–Bit–Einheit (oder n–Bit–Codewert) mit $n = m + r$. Bezüglich zweier beliebiger Codewörter, z.B. 10001001 und 10110001 kann die Anzahl sich unterscheidender Bits bestimmt werden (hier 3). Dies wird **z.B. mittels bitweiser Anwendung eines booleschen XOR durchgeführt**, wobei die Anzahl der 1–Bits im Ergebnis gezählt wird. Die Anzahl der Bits, die mindestens geändert werden müssen, um von einem gültigen Code um zum nächsten überzugeben, wird **Hamming–Abstand (Hamming Distance)** nach Hamming (1950) genannt.

Bei einem m–Bit–Speicherwert sind alle 2^m Bitmuster zulässig. Bezogen auf die n–Bit–Codewörter sind 2^m der 2^n Codewörter zulässig. Taucht bei einer Speicheroperation ein ungültiges Codewort auf, so erkennt der Computer, dass ein **Speicherfehler** aufgetreten ist.

Zurück zu unseren Codes für die Fehlererkennung.
Ein Fehlererkennungscode hat den **Abstand (Distance)** d, falls d **Einzelbitfehler** oder Vielfache davon erforderlich sind, um von einem gültigen Codewort zu einem anderen zu gelangen. Um d Einzelbitfehler erkennen zu können, benötigt man einen **Fehlererkennungscode** mit dem Abstand $d + 1$, weil es bei einem solchen Code möglich ist, dass d Einzelbitfehler ein gültiges Codewort in ein anderes gültiges Codewort überführen können. Für eine **Korrektur** von d Einzelbitfehlern sind Codes mit Abstand $2d + 1$ erforderlich, weil dann die zulässigen Codewörter so weit voneinander entfernt sind, dass bei d Fehlern ein Codewort immer noch näher ist als irgendein anderes.

Beispiel 8.5 ***(Fehlererkennungscodes)**:* Als Beispiel eines Fehlererkennungscodes wollen wir einen Code betrachten, bei dem den Daten ein einzelnes Bit, ein sogenanntes Paritätsbit, angehängt wird. Dieses Bit wird so gewählt, dass die Anzahl der Einsen eines Codes immer gerade ist. Dieser Code hat den **Hamming–Abstand** 2, da jeder **Einzelbitfehler** ein Codewort mit falscher Parität erzeugt. Sobald ein Wort mit falscher Parität aus dem Speicher gelesen wird, wird ein Fehlerzustand signalisiert. Das Programm kann nicht weiterlaufen, aber zumindest auch keine falschen Ergebnisse berechnen.

Beispiel 8.6 ***(Fehlerkorrekturcodes)**:* Als einfaches Beispiel eines Fehlerkorrekturcodes betrachten wir einen Code mit nur vier gültigen Codewörtern:

$$00000\,00000,\ 00000\,11111,\ 11111\,00000,\ 11111\,11111$$

Dieser Code hat den Hamming–Abstand 5. Gemäß $5 = 2d + 1$ kann er maximal Doppelfehler korrigieren. Kommt das Wort 00000 00111 an, so muß das Original 00000 11111 gewesen sein. Sollte ein dreifacher Fehler vom Original 00000 00000 aus diesen Wert 00000 00111 erzeugt haben, so wird dieser Fehler nicht korrigiert.

Wir wollen nun überlegen, wieviele **Prüfbits** r notwendig sind, um für m Datenbits einen Code zu entwickeln, der alle Einzelbitfehler korrigieren kann.

Wir gehen wieder von m Datenbits und r Prüfbits aus, was zu 2^m gültigen Speicherwörtern führt. Zu jedem Codewort mit $n = m + r$ Bits gibt es n ungültige Codewörter mit einem Abstand 1. Man erhält diese, indem man systematisch jedes der n Bits im n–Bit–Codewort invertiert. Für die 2^m gültigen Speicherwörter benötigen wir also zusätzlich $n \cdot 2^m$ Bitmuster für mögliche Fehler, also $(n+1) \cdot 2^m$ Bitmuster insgesamt. Diese notwendige Anzahl von Bitmustern darf natürlich den Wertebereich der mit n Bits darstellbaren Bitmuster, also 2^n nicht überschreiten.
Es ergibt sich:

$$\begin{array}{lll} (n+1)\cdot 2^m & \leq 2^n & mit\ n = m + r \\ (m+r+1)\cdot 2^m & \leq 2^{m+r} & \mid : 2^m \\ m+r+1 & \leq 2^r & \end{array}$$

Bei gegebenem m ergibt sich damit eine minimale Anzahl r von **Prüfbits** wie folgt:

Speicher–größe m	Prüfbit–größe r	Gesamt–größe n	Overhead der Prüfbits bzgl. der **Speicherbits**
4	3	7	75
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	256	4
512	10	522	2

Beispiel 8.7*:* Für den einfachsten dargestellten Fall $m = 4$ und $r = 3$ soll eine Idee skizziert werden, wie Fehler korrigiert werden können. Wir gehen von der Struktur $m_1 m_2 m_3 m_4 r_1 r_2 r_3$

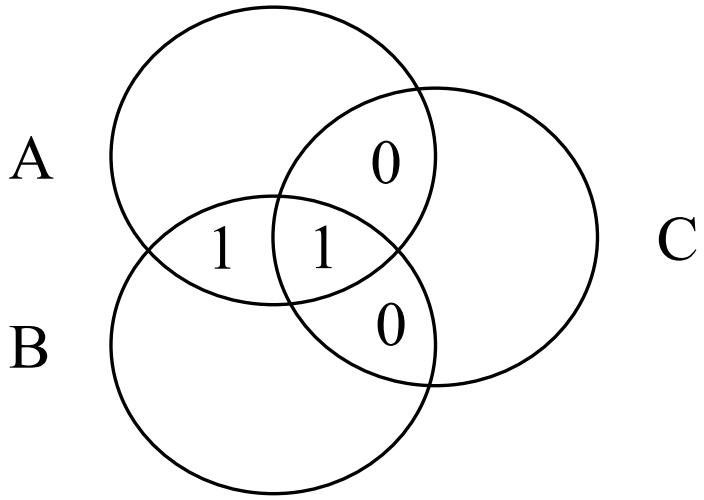


Abbildung 8.5: Venn–Diagramm

aus und verwenden die Prüfbits als Paritätsbits in Mengen A, B und C in einem sogenannten Venn–Diagramm, in dem m_1, m_2, m_3 und m_4 in Mengen AB, ABC, AC und BC (in alphabetischer Reihenfolge) codiert werden. Wir wollen sie mittels des Speicherwortes 1100 in Abbildung 8.5 veranschaulichen.

Daraus folgen die 3 Paritätsbits in A, B und C wie in Abbildung 8.6.

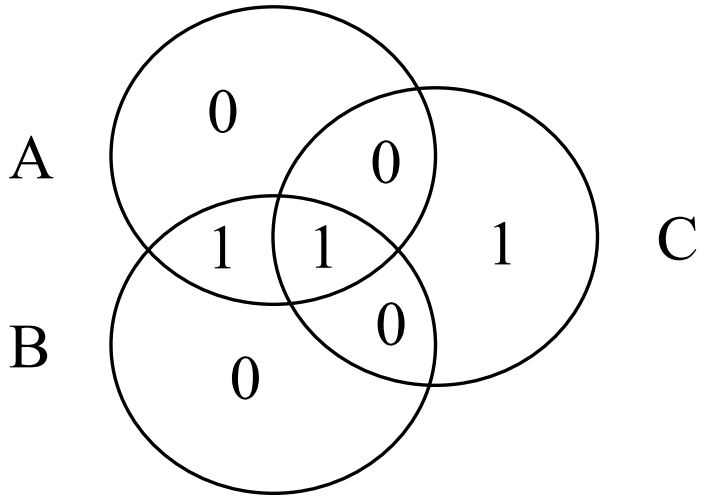


Abbildung 8.6: Venn–Diagramm

Es ergibt sich das zulässige Codewort 1100001.

Im Falle des Codewortes 1110001 erhalten wir das Bild in Abbildung 8.7, also Fehler in den Mengen A und C, und folglich schließen wir auf einen Fehler in A und C. Die **Fehlerkorrektur** ergibt das vorausgegangene Beispiel.

Fazit der Paritätsbits:

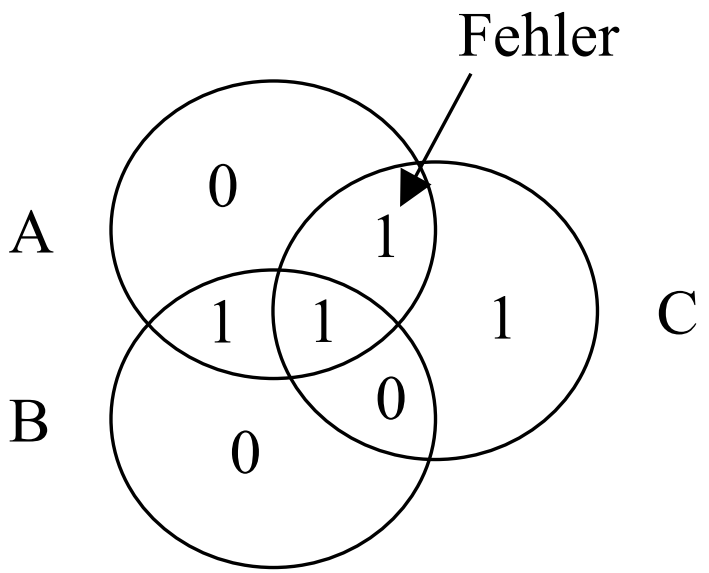


Abbildung 8.7: Venn–Diagramm

Mit dem Paritätsbit, das an die mit 7 Bit dargestellten ASCII–Zeichen angehängt wird, haben wir einen Code mit **Hamming–Abstand** 2. Daraus folgt, im Falle eines einzelnen Bitfehlers können wir diesen erkennen, nicht jedoch korrigieren. Dazu wären 4 Bits notwendig.

Nun wollen wir zur Entwicklung von **Fehlerkorrektur-Algorithmen** für Speicherwörter beliebiger Länge übergehen.

Hamming-Algorithmus Wir wollen den Hamming-Algorithmus zur Fehlerkorrektur von Einbitfehlern am Beispiel der Speichergröße m=8 und der zugehörigen Prüfbitgröße r=4 betrachten. Daraus ergibt sich eine Gesamtlänge von 12 Bit. Diese Bits werden von links beginnend mit Eins angefangen durchnummeriert:
$x_1, x_2, x_3, x_4 \ldots x_{12}$
Alle Bits, deren Index eine Zweierpotenz ist, sind Paritätsbits (hier: x_1, x_2, x_4, x_8), alle anderen ($x_3, x_5, x_6, x_7, x_9, x_{10}, x_{11}, x_{12}$) sind Speicherbits. Wir wählen im folgenden eine gerade Parität. Das Prinzip des Hamming-Algorithmus besteht nun darin, dass jedes Paritätsbit bestimmte Positionen prüft und zwar:

1. Bit 1 prüft die Bits 1, 3, 5, 7, 9 und 11
2. Bit 2 prüft die Bits 2, 3, 6, 7, 10 und 11
3. Bit 4 prüft die Bits 4, 5, 6, 7 und 12
4. Bit 8 prüft die Bits 8, 9, 10, 11 und 12

Generell wird Bit b = $b_1 + b_2 + \cdots + b_i$ von den Bits $b_1, b_2, \ldots b_i$ geprüft, wobei die $b_1, b_2, \ldots, b_i$ Zweierpotenzen sind.

Wir wollen das 8-Bit-Speicherwort 11011111 betrachten. Das zugehörige 12-Bit-Codewort ist ◡◡1◡101◡1111 wobei die Paritätsbits einzeln berechnet werden und das 12-Bit-Codewort wie folgt ergeben: **101110101111**.
Nun nehmen wir eine fehlerhafte Codewortspeicherung an, bei der der Einbitfehler an der Stelle 5 vorliegt, d.h. wir betrachten das Codewort **101100101111**.
Die 4 Paritätsbits werden geprüft und ergeben die Ergebnisse, dass genau Bit 1 und Bit 4 falsch sind. Das falsche Bit liegt somit in der Schnittmenge der von Bit 1 und Bit 4 geprüften Bits, d.h. es kann Bit 5 oder Bit 7 sein. Im Falle von Bit 7 wäre Bit 2 auch fehlerhaft, daher muß Bit 5 falsch sein. Auf dieses Ergebniss kommt man einfacher, wenn man 1 und 4 addiert. Durch Invertierung des fünften Bits ergibt sich das korrigierte Codewort.

9

Primäre Speicher

Eine große Herausforderung im Computer-Design ist die Bereitstellung eines Speichersystems, das in der Lage ist, den Prozessor mit Operanden zu versorgen. Logisch gesehen befindet sich ein Cache zwischen CPU und Hauptspeicher.

Der Primärspeicher enthält das Programm – in der Regel der Hauptspeicher –, das gerade ausgeführt wird. Dabei ist eine kurze Zugriffszeit sehr von Bedeutung – maximal einige Dutzend Nanosekunden. Diese Zeit hängt nicht von der Adresse ab. Die Bereitstellung der Daten ist aber von größter Bedeutung für einen schnellen Gesamtablauf.

Die hohe Wachstumsrate der Prozessorgeschwindigkeit kann man beim Speicher leider nicht erreichen. Daraus resultiert der von-Neumannsche-Flaschenhals. Dieser Effekt wird insbesondere dadurch verstärkt, dass die Speichermenge auch immer größer wird.

Dieses Problem ist eigentlich nicht von der Technik verursacht, sondern hat mit der Wirtschaftlichkeit zu tun. Ingenieure könnten durchaus einen Speicher bauen, der so schnell wie eine CPU es benötigt, ist. Weil der Weg über den Bus zum Speicher sehr langsam ist, müßte sich dieser Speicher allerdings auf dem CPU-Chip befinden. Bestückt man aber einen CPU-Chip mit einem großen Speicher, so wird dieser größer und teurer. Und selbst unabhängig von den Kosten gäbe es Grenzen für die Größe eines CPU-Chips.

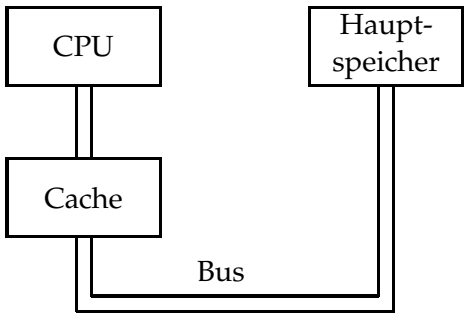
Trotz Wunsch nach einem großen schnellen, billigen Speicher muß man letzlich zwischen einem großen langsamen und einem kleinen schnellen Speicher wählen.

9.1 Cache

Es wird die Technik gewählt, einen kleinen schnellen mit einem großen langsamen Speicher zu kombinieren. Dadurch erhält man die Geschwindigkeit eines relativ schnellen und die Größe eines relativ großen Speichers zu einem angemessenen Preis.

Der kleine schnelle Speicher heißt Cache (vom französischen cacher - verstecken). Der Cache-Speicher beruht auf dem Grundkonzept, dass die am häufigsten gebrauchten Speicherwörter im Cache stehen sollen. Wenn die CPU ein Wort benötigt, wird dort zuerst nachgesehen. Nur wenn sie hier das Wort nicht findet, geht sie an den Hauptspeicher. Sofern ein merklicher Anteil der benötigten Wörter im Cache steht, ergibt sich eine erheblich reduzierte durchschnittliche Zugriffszeit.

Logisch gesehen befindet sich ein Cache zwischen CPU und Hauptspeicher.



9.1.1 Lokalitätsprinzip

Dieser Erfolg des Cache basiert auf dem sogenannten **Lokalitätsprinzip**. D.h. Programme greifen in der Regel nicht beliebig auf ihre Speicher zu. Wird die Adresse A refenziert, so liegt die nächste Speicherreferenz meist in der Nähe von A: Bei der Abarbeitung von Befehlen eines Programms, die nicht Sprünge oder Prozeduraufrufe sind, werden Befehle aus aufeinanderfolgenden Speicherzellen geholt. Zudem geht der größte Teil der Abarbeitungszeit auf Schleifen. Und bei Matrizen wird ebenfalls immer wieder dieselbe Matrix referenziert.

Zwei Eigenschaften der Abarbeitung eines Programms, die unter dem Begriff Lokalitätsprinzip zusammengefaßt werden, sind wichtig:

1. **Temporale Lokalität** (Lokalität der Zeit oder zeitliche Lokalität) wird ein bestimmter Befehl, d.h. eine gewisse Stelle des Speichers referenziert, so wird genau diese Stelle bald wieder gebraucht. Diese Tatsache ist z.B. durch Schleifen eines Programms oder Prozeduraufrufe bedingt.

2. **Spatiale Lokalität** (Lokalität des Raumes oder räumliche Lokalität) wird ein Befehl benötigt und damit eine gewisse Speicheradresse angesprochen, so werden Speicherstellen mit benachbarten Speicheradressen nachfolgend auch referenziert. Dieser Sachverhalt resultiert z.B. aus der sequentiellen Abarbeitung von Programmcode oder dem hintereinanderfolgenden Aufrufen von Feldern eines Arrays.

Alle Cache-Speicher basieren auf dem Lokalitätsprinzip. Dabei wird beim Aufruf eines Speicherworts nicht nur dieses Wort, sondern auch ein Teil seiner Nachbarn in den Cache geladen.

9.1.2 Leistungsbewertung

Wird ein Speicherwort in einem kurzen Zeitintervall k mal gelesen oder geschrieben, so benötigt der Computer *eine* Referenz auf den langsamen Hauptspeicher und $k-1$ Referenzen auf den schnellen Cache. Je größer k umso kürzer ist die mittlere Bereitstellungszeit.

Bei einer Cache-Zugriffszeit c und einer Hauptspeicherzugriffszeit m erhalten wir eine **Trefferate (Hit Ratio)** h als Anteil der Referenzen, die aus Zugriffen aus dem Cache erfüllt werden können von

$$h = \frac{k-1}{k}.$$

Entsprechend ergibt sich die **Fehlschlagrate (Miss Ratio)** e.

$$e = 1 - h = \frac{1}{k}$$

Daraus ergibt sich eine mittlere Zugriffszeit

$$\overline{t} = \frac{(k-1)\cdot c + 1\cdot(c+m)}{k} = \frac{k\cdot c - c + c + m}{k} = c + \frac{1}{k}\cdot m = c + (1-h)\cdot m$$

Bei sehr vielen Zugriffen auf dieselbe Speicherzelle, d. h. $k \to \infty$, geht $h \to 1$ und $\overline{t} \to c$, d.h. die Zugriffszeit nähert sich c an. Bei $h \to 0$ ist jedes mal eine Referenz auf den Hauptspeicher nötig, nachdem der Cache (vergeblich) angefragt wurde. Dann nähert sich $\overline{t} \to c + m$ an.

Bei machen Systemen kann parallel zur Cacheanfrage eine Suche im Hauptspeicher gestartet werden. Tritt ein Cache-Fehlschlag ein, so ist der Speicherzyklus schon im Gang. Allerdings setzt diese Strategie voraus, dass bei einem Cache-Treffer die Hauptspeicheranfrage sofort gestoppt werden kann, was die Implementierung verkompliziert.

9.1.3 Funktionsweise des Cache

Mit dem Lokalitätsprinzip als Leitlinie werden Haupt– und Cachespeicher in **Blöcke fester Größe** unterteilt. Diese Blöcke heißen auch **Cache-Zeilen (Cache Lines)**. Bei einem Cache-Fehler wird die gesamte Cache-Zeile aus dem Hauptspeicher geladen, nicht nur die benötigte Speicherzelle.

Beispiel 9.1*:* Bei einer Zeilengröße von 64 Byte wird bei einer Referenz auf die Speicheradresse 260 die ganze Zeile der Bytes 256 bis 319 in eine Cache-Zeile geladen.

Diese Vorgehensweise ist effizienter als der sequientielle Abruf von Speicherwörtern. Außerdem entsteht weniger Overhead, d.h. Verwaltungsaufwand.

Im folgenden wollen wir einen sehr einfachen Cache betrachten, bei dem ein Prozessor jeweils 1 Wort verarbeitet und ein Block auch einem Wort entspricht.

Man nehme folgende Cachebelegung an, und dass das Wort x_n referenziert werden soll:

x_2
x_4
x_{n-1}
...
...
x_1
x_3

Dann ergibt sich z.B. folgende Belegung nach Feststellung des "Miss" und Laden von x_n in den Cache:

x_2
x_4
x_{n-1}
...
x_n
x_1
x_3

Es ergeben sich jedoch 2 prinzipielle Fragen:

- Woran erkennen wir, ob sich Daten im Cache befinden oder nicht?
- Wie finden wir die Daten, die wir suchen?

Wir wollen zunächst von einem sehr einfachen Ansatz ausgehen.

Direkte Abbildung von Speicher auf den Cache
Jede Speichereinheit wird an genau einer Stelle im Cache gespeichert. Dazu wird in der Regel folgender Zusammenhang verwendet:

(Blockadresse im Speicher) modulo (Blocknummer im Cache)

Basierend auf der Zweierpotenzrechnung läßt sich so recht einfach das Mapping realisieren (vgl. Abbildung 9.1).

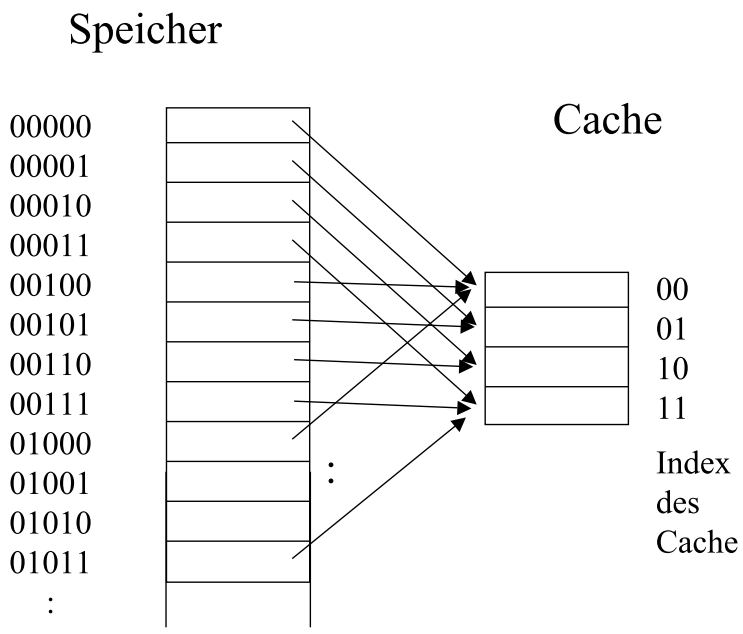


Abbildung 9.1: Mapping

Es bleibt jedoch das Problem, wie ich erkenne, zu welchem Speicherplatz ein belegter Cacheblock gehört. Z.B. könnten Daten in Cacheblock 10 sowohl zum Speicherplatz 00110 als

auch zu 10010 oder vielen anderen Plätzen gehören. Aus diesem Grund fügen wir den Cacheblöcken je ein sogenanntes "Tag" hinzu. Dieses Tag erhält den höherwertigen Teil der Speicheradresse. Für das gerade angeführte Beispiel also 001 bzw. 100. Damit kann man bestimmen, welcher Speicheradresse der Inhalt eines Cacheblockes zugeordnet ist.

Außerdem benötigen wir ein sogenanntes "valid bit" V pro Cacheplatz, das anzeigt, ob das entsprechende Tag eine gültige Adresse enthält. Dies ist z.B. beim Starten eines Prozessors von Bedeutung, wenn z.B. alle Speicherplätze mit Nullen belegt sind.

Den Zugriff und das Arbeiten mit dem Cache wollen wir nun an einem Beispiel durchdenken.

Beispiel 9.2*:* Wir nehmen einen Speicher mit 32 Plätzen an, also Adressen der Länge 5 Bit. Unser Cache habe 8 Blöcke, also dreistellige Nummern. Wir benötigen nun nacheinander folgende Speicherzugriffe (bei anfangs leerem Cache):

dezimale Adresse	Binär–adresse		resultierendes hit/miss	zugeordneter Cacheblock
22	10110		miss	10110 mod 8 = 110
26	11010		miss	010
22	10110		hit	110
26	11010	$\Rightarrow$	hit	010
16	10000		miss	000
3	00011		miss	011
16	10000		hit	000
18	10010		miss	010

Wir wollen nun von einem leeren Cache ausgehen und nacheinander die Modifikation der einzelnen Daten betrachten, die sich durch Belegung der Cacheblöcke ergeben.

Index	V	Tag	Daten
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

22 ⇒

Index	V	Tag	Daten
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Inhalt (10110)
111	N		

26 ⇒

Index	V	Tag	Daten
000	N		
001	N		
010	Y	11	Inhalt (11010)
011	N		
100	N		
101	N		
110	Y	10	Inhalt (10110)
111	N		

22, 26 ändert nichts, dann 16 ⇒

Index	V	Tag	Daten
000	Y	10	Inhalt (10000)
001	N		
010	Y	11	Inhalt (11010)
011	N		
100	N		
101	N		
110	Y	10	Inhalt (10110)
111	N		

3 ⇒

Index	V	Tag	Daten
000	Y	10	Inhalt (10000)
001	N		
010	Y	11	Inhalt (11010)
011	Y	00	Inhalt (00011)
100	N		
101	N		
110	Y	10	Inhalt (10110)
111	N		

16 ändert nichts, dann 18 ⇒

Index	V	Tag	Daten
000	Y	10	Inhalt (10000)
001	N		
010	Y	10	!! Inhalt (10010)
011	Y	00	Inhalt (00011)
100	N		
101	N		
110	Y	10	Inhalt (10110)
111	N		

Bei der letzten Belegung der Cacheblöcke ist zu beachten, dass sich der 3. Block (mit dem Index 010) hinsichtlich seines Wertes und seines Tags ändert. Beide Einträge werden einfach überschrieben. Der ursprüngliche Wert der Speicherzelle 26 ist dann nicht mehr im Cache, sondern lediglich im Hauptspeicher vorhanden. Seine erneute Referenzierung würde ein "miss" ergeben.

Zusammenfassend wollen wir analysieren, wieviel Speicherplatz erforderlich ist, um einen Direct–mapped Cache zu realisieren, der 64 KByte Daten zwischenspeichern kann. Außerdem gehen wir von 32–Bit–Adressen (d. h. der Speicher umfasst $2^{32} = 2^2 \cdot 2^{30} = 4$ GByte) und Blöcken der Größe eines Wortes, also 32 Bit = 4 Byte aus. In der Analyse betrachten wir zunächst den Datenanteil von 64 KByte und danach die Größe des gesamten Cache.

Der Datenanteil von 64 KByte wird in Worte zu je 32 Bit zerlegt:

64 KByte = $64 \cdot 1024$ Byte = $2^6 \cdot 2^{10}$ Byte = 2^{16} Byte
4 Byte = 1 Wort, d. h. wir benötigen $2^{16} : 4 = 2^{16} : 2^2 = 2^{14}$ Wörter.

Betrachtungen zum Cache: Da 1 Block = 1 Wort groß ist besteht unser Cache aus 2^{14} Blöcken; dass heißt, die Anzahl der Index-Bits ist 14. Jeder Block hat 32 Bit Daten, ein Tag und ein Valid Bit (vgl. Abb. oben). Das Tag benötigt folgenden Speicherplatz: Aus 2^{32} Byte Speicher und 2^{16} Byte Cache ergibt sich für das Tag eine Länge von $32 - 16 = 16$ Bit[1]. Das Valid Bit ist 1 Bit groß. Der Index ergibt sich aus der Position – er wird nicht noch einmal mitgespeichert.

[1]Die Anzahl der Tag-Bits hängt **nur** von der Datengröße des Cache ab im Vergleich zum Gesamtspeicher; und nicht davon, wie viele Byte pro Cache-Block enthalten sind. Die Anzahl der pro Cache-Block enthaltenen Byte bestimmt die Anzahl der Index-Bits. Es gilt die Beziehung zwischen Adress-, Tag-, Index-Bits und Größe eines Cache-Blocks: Anzahl der Adress-Bits = Anzahl der Tag-Bits + (Anzahl der Index-Bits + 2er-log der Anzahl pro Cache-Bock enthaltenen Byte) Diese letzte Summe ist konstant bei konstanter Größe des Daten-Cache.

Insgesamt benötigen wir zur Realisierung des Cache:

$$\begin{aligned} 2^{14} \cdot (32 + 16 + 1) &= 2^{14} \cdot 49 &&= 802816\ Bit \\ &= 2^4 \cdot 49 \cdot 2^{10} &&= 784 \cdot 2^{10} \\ &&&= 784\ KBit \\ &&&= 98\ KByte \end{aligned}$$

D.h. zur Realisierung eines 64 KByte Cache werden 98 KByte Speicherplatz benötigt. Es liegt also ein Faktor 1,53 vor, der durch die Verwaltung dieser Struktur bedingt ist.

9.1.4 Cache-Ebenen

Durch flexiblere Zuordnung der Speicherinhalte zu Cacheblöcken läßt sich die Anzahl der Cache Misses reduzieren. Allerdings wird das Suchen nach Speicherinhalten im Cache und die Verwaltung der Cacheblöcke dann entsprechend komplexer.

Anforderung an ein modernes Speichersystem ist:

- eine Verkleinerung der **Latenz**, d.h. die Verkürzung der Verzögerung bei der Lieferung von Speicherinhalten wie z.B. Operanden und
- eine Vergrößerung der **Bandbreite**, d.h. der Datenmenge, die pro Zeiteinheit geliefert wird.

In der Regel schließen sich diese Anforderungen gegenseitig aus. Die wirksamste Technik zur Verbesserung der Latenz und Bandbreite ist die Verwendung mehrerer Caches.

Eine Technik besteht darin, einen jeweils separaten Cache für Befehle (Instruktionen) und Daten einzuführen. Diese Technik wird auch als **Split-Cache** bezeichnet. Dabei können Speicheroperationen auf beiden Caches unabhängig voneinander durchgeführt werden, so dass sich die Bandbreite effektiv verdoppelt. Diese Technik bedingt jedoch, dass zwei getrennte Speicherports am Prozessor bereitstehen, denn jeder Port hat seinen eigenen Cache. Ferner muß jeder Cache einen unabhängigen Zugriff auf den Hauptspeicher haben.

Instruktion– und Datencache sind in der Regel direkt auf dem CPU-Chip angeordnet und haben eine Größe von je 8 KByte bis 64 KByte.

Heute sind viele Speichersysteme viel komplizierter: Zwischen dem Instruktions– und Datencache und dem Hauptspeicher kann ein zusäztlicher Cache, der Level-2-Cache oder L2-Cache angesiedelt werden. Dieser befindet sich nicht auf dem CPU-Chip. Er ist aber in der CPU-Baueinheit direkt neben dem CPU-Chip über einen Hochgeschwindigkeitspfad an die CPU angeschlossen. In der Regel ist er vereint (unified), d.h. er umfasst eine Mischung aus Daten und Instruktionen. Typische Größen für den L2-Cache reichen von 256 KByte bis 1 MByte.

Möclicherweise gibt es auch drei oder mehr Cache-Ebenen. Der Cache der dritten Ebene würde sich dann auf der Prozessorkarte befinden und besteht aus ein paar MByte SRAM, was viel schneller als der DRAM Hauptspeicher ist. L3 ist auch vereint.

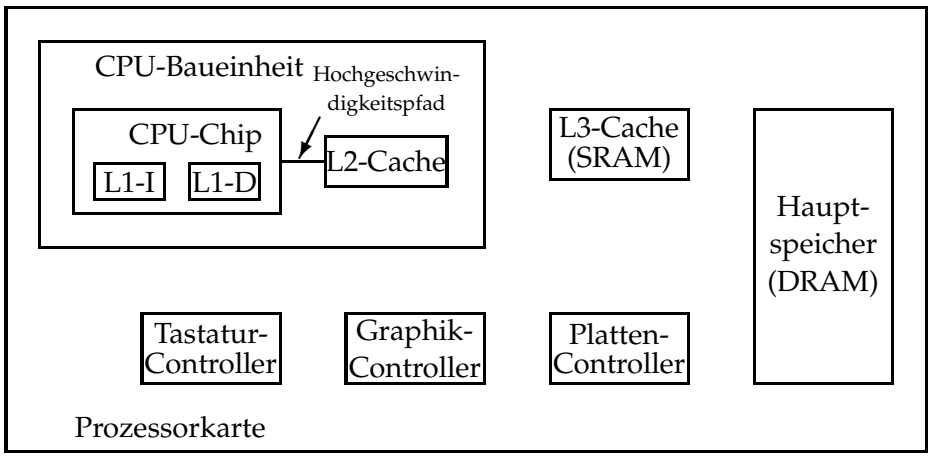


Abbildung 9.2: Prozessorkarte mit drei Cache-Ebenen

Caches sind im allgemeinen inklusive. D.h. der volle Inhalt des Level-1-Cache ist im Level-2-Cache enthalten, und der volle Inhalt des Level-2-Cache ist im Level-3-Cache enthalten.

Zur Historie: In der Intel-CPU-Familie war der 80486 (April 1989) der erste Prozessor mit eingebautem 8K-Cache-Speicher. In der übernächsten Generation (da man Zahlen nicht als Wahrenzeichen schützen dar, hieß die nächste Generation nicht 80586, sondern Pentium nach dem Griechischen Wort für Fünf, die übernächste Generation dann Pentium Pro) ab März 1995 wurden zwei eingebaute Cache-Ebenen verwendet. In dieser Generation sind L1-I und L1-D jeweils 8KByte groß. In der gleichen Vertiefung der Karte, nicht aber auf dem Chip selbst, befindet sich der L2 mit 256 KByte.

9.2 Speichermodule SIMM und DIMM

Seit Entstehung der Halbleiter-RAMs bis Anfang der 90er Jahre wurden Speicher als einzelne Chips produziert, gekauft und installiert. Die Speicherkapazität stieg von 1 Kbit auf 1 Mbit und mehr. Jeder Chip war eine separate Komponente, die in Sockel von PCs eingesteckt werden konnte, wenn der Käufer sie braucht.

Derzeit nutzt man ein anderes Prinzip. Üblicherweise 8 oder 16 Chips werden auf einer kleinen Leiterplatte montiert und als Baugruppe verkauft. Abhängig davon, ob sie nur auf einer oder aber auf beiden Seiten der Leiterplatte einen Kontaktstreifen aufweisen, werden sie als **SIMM (Single Inline Memory Module)** oder **DIMM (Dual Inline Memory Module)** bezeichtnet.

Das Prinzip eines SIMMs ist in Abbildung 9.3 dargestellt.

Eine übliche SIMM-Konfiguration hat beispielsweise acht Chips mit je 32 MBit d.h. 4 MByte. Das Modul verfügt über insgesamt 32 MByte. Viele Computer haben Platz für vier Module, so dass sich eine Gesamtkapazität von maximal 128 MByte ergibt.

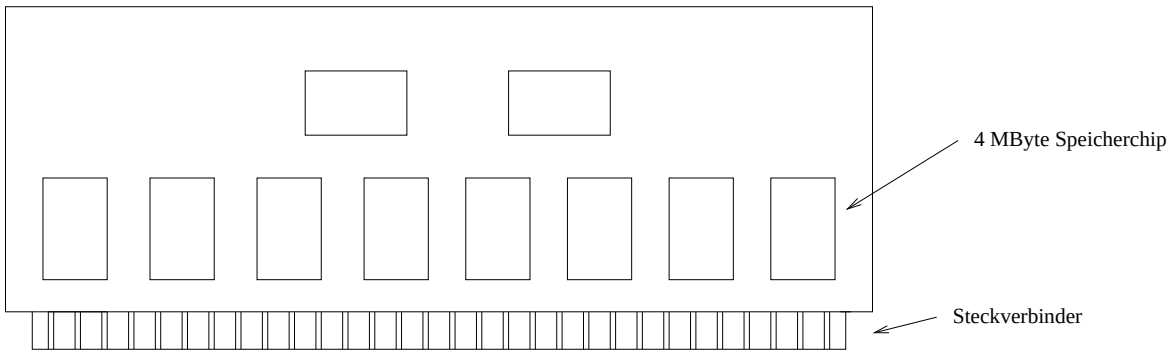


Abbildung 9.3: SIMM von 32 MByte – zwei zusätzliche Chips steuern das SIMM

Bei Bedarf können die 32 MByte SIMMs aber auch durch 64 MByte SIMMs oder größer ersetzt werden, so dass sich die Gesamtkapazität noch einmal verdoppelt wird bzw. noch weiter steigt.

Die ersten SIMMs hatten 30 Steckverbinder und konnten jeweils 8 Bits gleichzeitig abgeben. Die übrigen Steckverbinder dienten zur Adressierung und Kontrolle. Spätere SIMMs wiesen 72 Steckerverbinder auf und lieferten jeweils 32 Bits gleichzeitig. Für Maschinen wie den Pentium, die gleichzeitig 64 Bit erwarten, wurden SIMMs mit 72 Steckverbindern paarweise angeordnet, wobei jedes SIMM je eine Hälfte der benötigten Bits lieferte.

Derzeit sind DIMMs die standardmäßige Bestückungsart. Ein DIMM verfügt auf jeder Seite über 84 vergoldete Steckverbinder, also insgesamt 168. Er kann gleichzeitig 64 Datenbits liefern. In Notebook-Computern kommt ein kleineres DIMM mit der Bezeichnung **SO-DIMM (Small Outline DIMM)** zum Einsatz.

SIMMs und DIMMs können auch über ein zusätzliches Paritätsbit bzw. eine Fehlerkorrektur verfügen. Da die durchschnittliche Fehlerrate bei einem SIMM oder DIMM jedoch bei zehn Jahren (!) liegt, wird bei gewöhnlichen Computern auf eine derartige Fehlererkennung und –korrektur verzichtet.

10

Sekundäre Speicher

Kleine Register, verschiedene Cache-Ebenen und ein zeitgemäßer Hauptspeicher als primäre Speichermedien: egal, wie große dieses System auch sein mag, es ist immer zu klein. Allein durch das ständige Wissenswachstum der Menschheit und neue platzraubende Formate wie Audio, Bilder und insbesondere Video steigen die Anforderungen an ein RAM ständig.

10.1 Speicherhierarchien

Die traditionelle Lösung zum Speichern großer Datenmengen ist die Speicherhierarchie.

Im folgenden soll der Speicher als Hierarchie mit mehreren Speicherebenen, die unterschiedliche Speichergrößen und Zugriffszeiten aufweisen, betrachtet werden. Je kürzer die Zugriffszeit eines Speicherbausteins sein soll, desto teurer ist heutzutage die Herstellung. Heute werden 3 grundlegende Technologien für Speicherhierarchien verwendet.

- Hauptspeicher wird als **Dynamic Random Access Mamory (DRAM)** implementiert
- schnelle Speicher, die "näher" an der CPU sind, nutzen **Static Random Access Memory (SRAM)**. Diesen Speicher haben wir als Cache bezeichnet. SRAM ist teurer als DRAM, dafür aber schneller. SRAM benötigt ferner mehr Platz – hat damit also weniger Speicherplatz bezogen auf dieselbe Menge Silicon.

  Die Implementierung von SRAM und DRAM ist im Abschnitt B.5 im Anhang von Patterson Hennessy beschrieben.
- Schließlich werden **Magnetplatten** (Festplatten) verwendet, um den größten, billigsten und langsamsten Speicher zu realisieren, der zusätzlich die persistente Speicherung von Daten übernimmt.

Mit Werten aus dem Jahre 1997 wollen wir diese 3 Technologien vergleichen:

	Typische Zugriffszeit	Preis ($ pro MByte)
SRAM	5 – 25 ns	100 – 250 $
DRAM	60 – 120 ns	5 – 10 $
Magnetplatte	10 – 20 Mio. ns	0,10 – 0,20 $

Daraus resultiert der Ansatz, Speicher als Hierarchie aufzubauen (vgl. Abbildung 10.1).

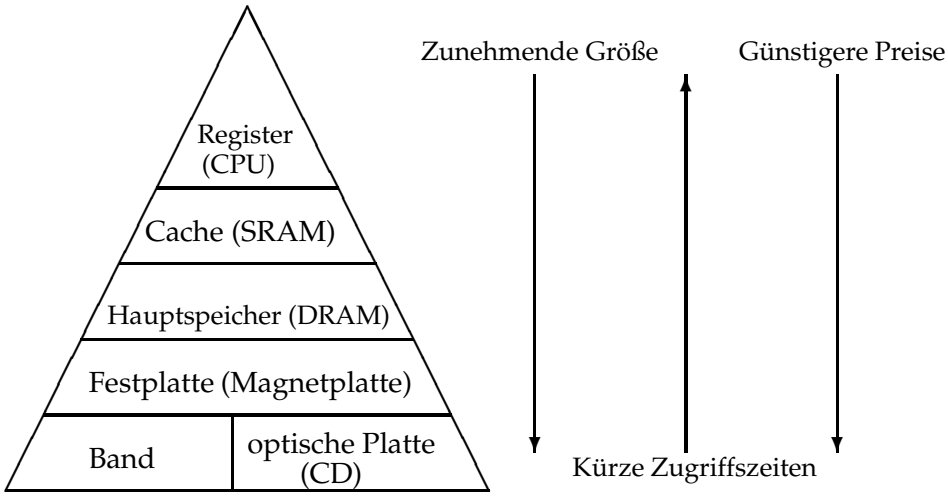


Abbildung 10.1: Speicherhierarchie

Durch geeignete Zugriffsverfahren stehen der CPU in vielen Fällen die Daten aus dem SRAM zur Verfügung. Alle Daten werden in der niedrigsten Ebene gespeichert. Die höheren Ebenen enthalten dann jeweils eine Teilmenge von Daten der darunterliegenden Ebene (Ausschnittshierarchie).

In den meisten Systemen ist der Speicher eine echte Hierarchie in dem Sinne, dass Daten nicht in einer Ebene i vorhanden sein können, wenn sie nicht bereits in der Ebene i+1 vorhanden sind.

Dabei werden Daten zu einem Zeitpunkt nur zwischen jeweils benachbarten Ebenen kopiert, dabei wird die kleinste Einheit von Daten, die zwischen 2 benachbarten Ebenen auf einmal kopiert werden kann als Block bezeichnet. Werden Daten von einer untenliegenden Einheit benötigt, so spricht man im Falle des Vorhandenseins von einem "Hit", im Falle des Nichtvorhandenseins von einem "Miss".

Register, Cache– und Hauptspeicher haben wir bereits unterschiedlich detailliert als primären Speicher behandelt. Im folgenden wollen wir uns mit den größeren und in der Hierarchie weiter unten angesiedelten sekundären Speichermedien befassen.

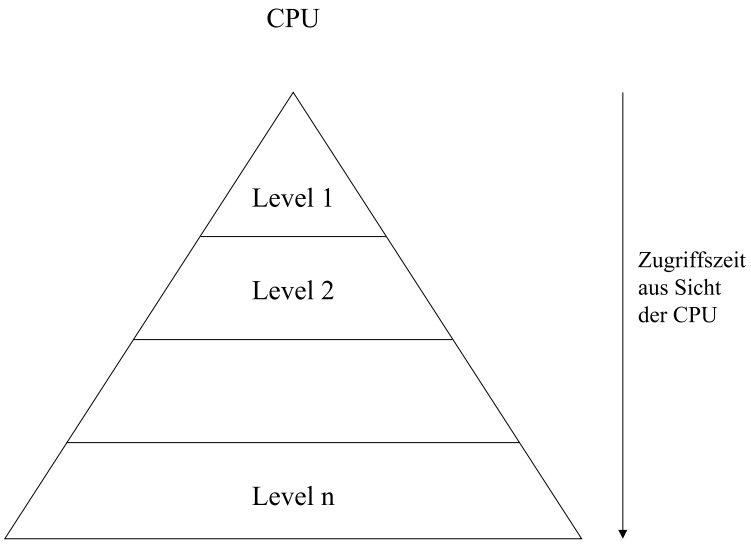


Abbildung 10.2: Zugriffszeit

10.2 Festplatten

Aus technischer Sicht ist eine Festplatte eine Magnetplatte, die aus einer oder mehreren runden Aluminium– oder Glasscheiben mit magnetisierbarer Oberfläche besteht.

Aus ursprünglich bis 50 cm Durchmesser sind heute 3-12 cm Durchmesser geworden. Festplatten für Notebook liegen sogar unter 3 cm Durchmesser.

Ein Festplattenkopf mit einer Induktionsspule schwebt auf einem Luftkissen knapp über der Oberfläche. Wenn positive oder negative Ladung durch den Lopf fließt, magnetisiert er die Fläche direkt darunter (Schreiben). Läuft unter dem Kopf ein magnetisierter Bereich vorbei, wird eine positive oder negative Ladung induziert (Lesen).

Die kreisförmige Abfolge von Bits, die gelesen oder geschrieben werden, während die Platte eine Umdrehung zurücklegt, wird **Spur (Track)** genannt. Bei jedem radialen Abstand des Kopfs kann eine andere Spur beschrieben werden. Die Spuren bilden also konzentrische Kreise um den Plattenmittelpunkt, an dem sich die **Spindel** befindet. Heute haben Platten bis zu 50.000 Spuren pro cm Radius, wobei auf jeder Spur 50.000 - 100.000 Bits/cm gespeichert werden. Die Datendichte wird meistens in GBit/Zoll^2 angegeben, bei aktuellen Platten liegt dieser Wert über 60 GBit/Zoll^2. Die Rotationsgeschwindigkeit der Platten beträgt 3600, 5400, 7200, 10.000 oder 15.000 **Umdrehungen pro Minute (UpM)**, das sind bis 180 Umdrehungen pro Sekunde. Dies ergäbe einen Durchsatz von 5 bis 20 MByte/s, wobei jedoch Drehverzug, Armbewegung und Suchzeit in erheblichem Maße dazukommen.

Die meisten Festplatten bestehen aus mehreren vertikal übereinander angebrachten Scheiben. Für jede Oberfläche ist dabei ein eigener Arm mit Lese/Schreibkopf vorgesehen, siehe Abbildung 10.3.

Alle Arme sind mechanisch gekoppelt und bewegen sich daher gleichzeitig in eine jeweilige radiale Position. Ein Spurensatz bei einer gegebenen radialen Position wird als **Zylinder**

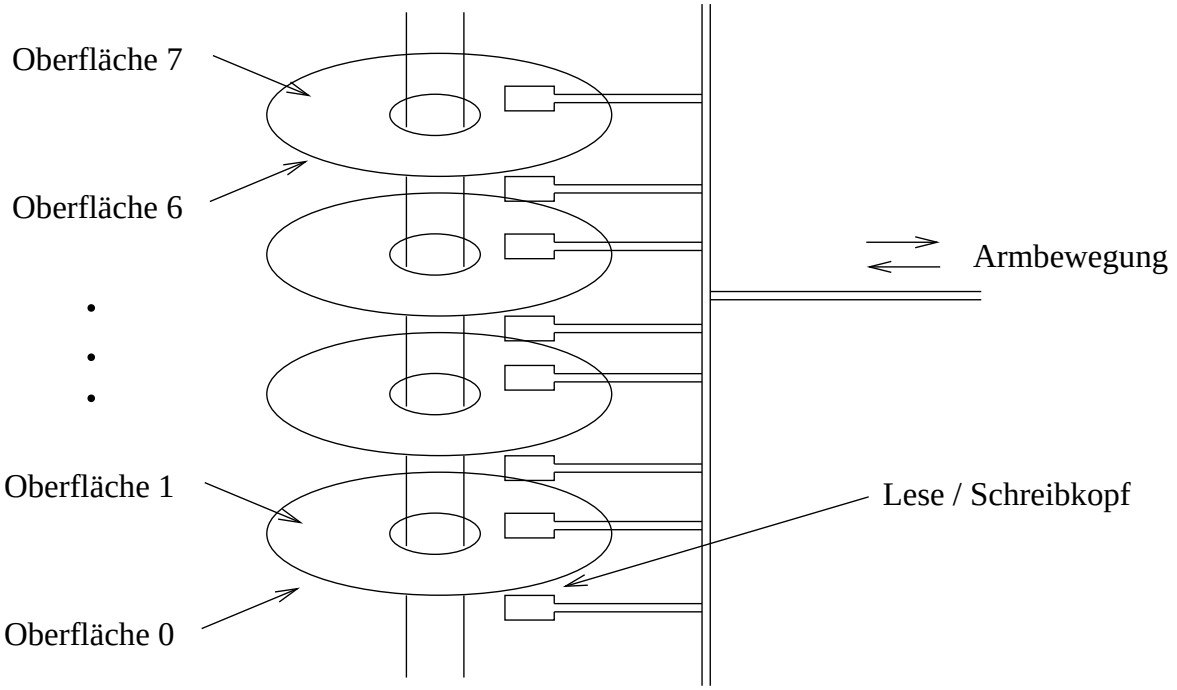


Abbildung 10.3: Eine Festplatte mit vier Scheiben

bezeichnet. Bei nur einer Oberfläche entspricht die Anzahl der Zylinder damit der Anzahl der Spuren.

Jede Spur ist in **Sektoren (Sectors)** mit fester Länge von z.B. 512 Nutzbytes unterteilt, denen aus organisatorischen Grnden eine Päambel vorausgehet. Den Daten folgt ein Fehlerkorrekturcode (*Error Correcting Code, ECC*). Dies kann ein Hammingcode sein oder ein sogenannter Reed-Solomon-Code, der mehrere Fehler korregieren kann. Zwischen aufeinanderfolgenden Sektoren befindet sich eine kleine Zwischensektor-Lücke (**Intersector Gap**), siehe Abbildung 10.4.

Beachte: Manche Hersteller geben die Kapazität ihrer Festplatten in unformatiertem Zustand an, als ob jede Spur nur Nutzdaten enthielte. Präambeln, ECC und Lücken reduzieren die Festplattenkapazität jedoch um ca. 15%.

Probleme bei Festplatten bestehen daran, daß diese bei Geschwindkeiten von 60 bis 180 UpM heiß werden und sich ausdehnen. Dadurch müssen Positionierungsmechanismen neu kalibriert werden.

Ferner sind gemäßt $U = 2 \cdot \pi$ die äußeren Spuren länger als die inneren, die Rotation erfolgt jedoch mit konstanter Winkelgeschwindigkeit. Früher hat man innen die größtmögliche Speicherdichte verwendet und ist nach außen mit gleicher Bogengradzahl linear größer geworden. Heute erhöht man nach außen hin die Anzahl der Sektoren pro Spur, so daß bei identischer Sektorengröße außen mehr Sektoren die Kapazität der Spur erhöhen.

Zu jedem Festplattenlaufwerk gehört noch ein Festplatten-Controller (Disk Controller), d.h. eine Leiterplatte voller Chips, die das Laufwerk steuern. Zu den Aufgaben des Controllers gehören u.a.

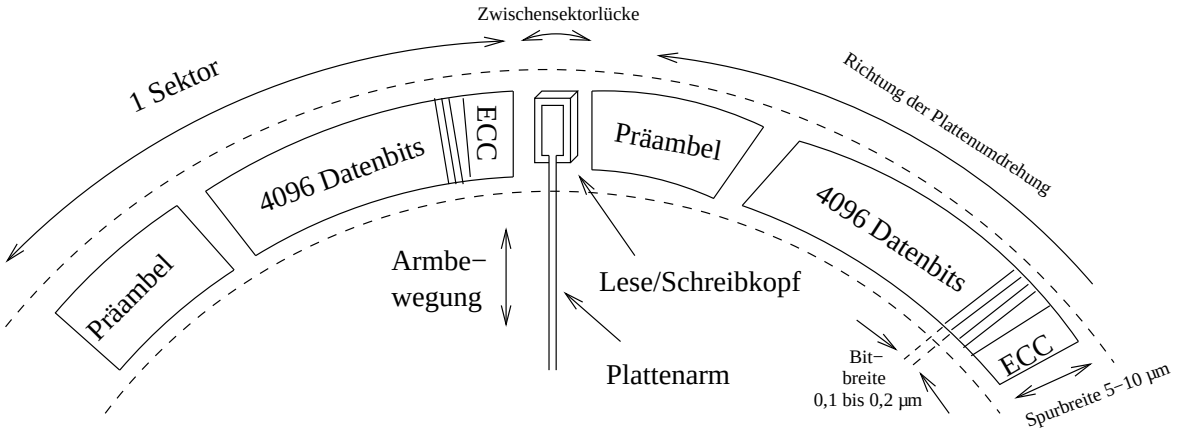


Abbildung 10.4: Zwei Sektoren auf einer Plattenspur

- die Annahme von Befehlen von der Software wie z.B. READ, WRITE, FORMAT
- die Steuerung der Armbewegung,
- das Erkennen und Korrigieren von Fehlern und
- die Umwandlung von Bits aus dem Speicher in einen seriellen Bitstrom.

10.3 IDE-Festplatten

Diese Festplatte der Firma Seagate wurde für moderne PCs entwickelt. Sie hatte zunächst 4 Köpfe, 306 Zylinder und 17 Sektoren pro Spur. Damit haben wir insgesamt $4 \cdot 306 \cdot 17 = 20.808$ Sektoren mit insgesamt 10 MByte. Der Controller kann 2 Festplatten steuern. Der Controller wurde Mitte der 80er Jahre nicht mehr auf einer separaten Leiterplatte untergebracht, sondern bei sogenannten **IDE**-Laufwerken (**Integrated Drive Electronics**) in die Festplattenlaufwerke integriert. Die maximale Kapazität beträgt 528 MByte bei einer Erweiterung auf 16 Köpfe, 63 Sektoren und 1024 Zylinder. Denn: 16 Köpfe ·1024 Zylinder = 16.384 Spuren à 63 Sektoren macht 1.032.192 Sektoren je 512 Byte sind 528.482.304 Byte = 528.482,304 KB = 528,482304 MB.

Eine Weiterentwicklung ist unter dem Namen **Extended IDE (EIDE)** bekannt. Sie unterstützt die logische Blockadressierung (LBA), die in Kopf-, Sektor- und Zylinderadressen umgerechnet werden muß. Ferner sind 4 Laufwerke ansteuerbar.

IDE- und EIDE-Festplatten werden aufgrund ihres günstigen Preises nicht mehr nur für Intelsysteme eingesetzt, sondern auch in anderen Computer genutzt.

10.4 SCSI-Festplatten

Bezüglich Organisation der Zylinder, Spuren und Sektoren liegt bei den SCSI-Festplatten (**Small Computer System Interface**, sprich „skasi“) dasselbe Prinzip wie bei IDE vor. Der

Unterschied besteht in einer anderen Schnittstelle. Diese ermöglicht einerseites einen viel höheren Durchsatz, andererseits aber auch größere Kabellängen und auch den Anschluß größerer Anzahl von Geräten. Diese Schnittstelle ist deshalb der Standard in UNIX-Workstations von Sun, HP, SGI und Server-Systemen.

Die SCSI-Schnittstelle ist ein Bus, an den ein SCSI-Controller und bis zu sieben Geräte angeschlossen werden können. Dabei kann es sich um eine oder mehrere SCSI-Festplatten, CDROM-Laufwerke, CD-Recorder, Scanner, Bandlaufwerke und andere SCSI-Peripheriegeräte handeln.

Jedes SCSI-Gerät hat eine eindeutige Nummer von 0 bis 7 (oder bei „Wide SCSI“ bis 15) und hängen an einem Kabelstrang (Bus). Hierbei handelt es sich um ein meist 68-poliges Kabel, an dem die jeweiligen Geräte angeschlossen sind. Externe Geräte verfügen über einen Ein- und Ausgang und der Bus wird durch das Gerät durchgeschleift. Bus-Systeme werden in der Elektrotechnik auch für viele andere Anwendungen verwendet (ISDN, ältere Netzwerke, u.s.w.). Ein Bus muß immer mit einem Terminator (Endwiderstand) abgeschlossen (terminiert) werden, sonst entstehen an Kabelenden Signalreflexionen, die zu Übertragungsfehlern führen. Die Terminierung erfolgt entw. durch Setzen einer Steckbrücke (Jumper) oder mittels einem speziellen Terminator-Stecker. In den letzten Jahren wurden unterschiedliche SCSI-Standards entwickelt, die unterenander abwärtskompatibel sind. Bei aktuell verwendeten Ultra-320-Standard beträgt die maximale Datenrate 320 MB/s. Für die Datenübertragung werden 68-adrige Flachbandkabel verwendet, hierbei werden 16 Bit Daten gleichzeitig übertragen. Bei Serversystemen finden meist SCA-Systeme verwendung, hierbei handelt es sich um übliche Ultra-320-Geräte, die für Hot-Swap-Betrieb ausgelegt sind (d. h. Austausch der Festplatte im laufenden Betrieb). In den für diese Technologie verwendeten 80-poligen SCA-Steckern werden neben den 68 Adern des Ultra-320-Interfaces zusätzliche Adern für die Stromversorgung der Festplatte benötigt.

Alle Geräte an einem Bus können gleichzeitig den Bus für die Übertragung nutzen. Über Steuersignale werden Phasen gekennzeichent, die über den Buszugriff entscheiden, hierfür wird sogenannte Konkurrenzbereinigung (Arbitration) verwendet. Während IDE und EIDE jeweils nur ein aktives Gerät zulassen, wird dadurch bei SCSI die Systemleistung erheblich verbessert, wenn mehrere Prozesse gleichzeitig aktiv sind.

Vergleich IDE/SCSI-Technologie:

Technologie	IDE	SCSI
Kosten	gering (in jedem PC vorhanden)	mittel / hoch
Geschwindigkeit d. Schnittstelle	hoch bei Systemen mit nur 1 - 2 Festplatten, gering bei Systemen mit mehr angeschlossenen Geräten	hoch
Erweiterbarkeit	für Privatgebrauch ausreichend	hoch
Anzahl der Geräte	2 je IDE-Kanal (Standard-PCs haben meist 2 IDE-Kanäle)	hoch (15 Geräte je Kanal)
CPU-Last	hoch	gering
Hot-Swap (Austausch der Laufwerke im lfd. Betrieb)	nur mit Speziallösungen	möglich, bei SCA-Systemen vorgesehen
Kabellänge / Gerätetyp	max. 60 cm, deshalb Nutzung nur für im PC eingebaute Laufwerke wie Festplatten, CD/DVD-Lw., Bandlaufwerke	mehrere Meter, deshalb Verwendung auch Anschluß von unterschiedlichsten externen Geräten mit hohem Datenaufkommen möglich (externe Laufwerke, externe Speichersysteme, Scanner)

10.5 RAID-Systeme

Um die Geschwindigkeit von Festplatten weiter zu erhöhen liegt den RAID-Systemen die Idee zugrunde, die Ein- und Ausgabe bei Festplatten zu parallelisieren. In diesem Sinne werden alle bisherigen Architekturen als **SLED** (**S**ingle **L**arge **E**xpensive **D**isk; Große und teuere Einzelfestplatte) bezeichnet. Unter der Bezeichnung **RAID** (**R**edundant **A**rray of **I**nexpensive **D**isks, Redundante Anordnung billiger Festplatten) werden sechs genau festgelegte Plattenarchitekturen vorgestellt, wobei die Industrie aus der ursprünglich wissenschaftlichen Bezeichung des „I“ im Sinne von Inexpensive (billig) in Independent (unabhängig) machte.

Das Grundkonzept von RAID ist es, neben einem i.d.R. großen Server einen Kasten voller Festplatten aufzustellen und den Festplatten-Controller durch einen RAID-Controller zu ersetzen. RAID erscheint dem Betriebssystem dabei wie ein SLED, so dass keine Softwaremodifikationen nötig sind.

Im folgenden wollen wir uns die 7 Architekturen unter der Originalbezeichnung Level 0 bis Level 5 sowie Level 10 kurz anschauen.

Level 0

Die von RAID simulierte virtuelle Einzelplatte wird in gleichgroße Streifen von je k Sektoren unterteilt. Dabei liegen die Sektoren 0 bis $k-1$ als Streifen 0 vor, die Sektoren k bis $2k-1$ als Streifen 1 u.s.w.
k gibt dabei die Anzahl der Sektoren pro Streifen an.

Die Streifen werden dann gemäß ihrer Nummer modulo der Plattenanzahl auf die Einzelplatten verteilt. Für beispielsweise 4 Einzelplatten ergibt sich das in Abbildung 10.5 dargestellte RAID-System. Diese Art der Datenverteilung über mehrere Plattenlaufwerke wird als

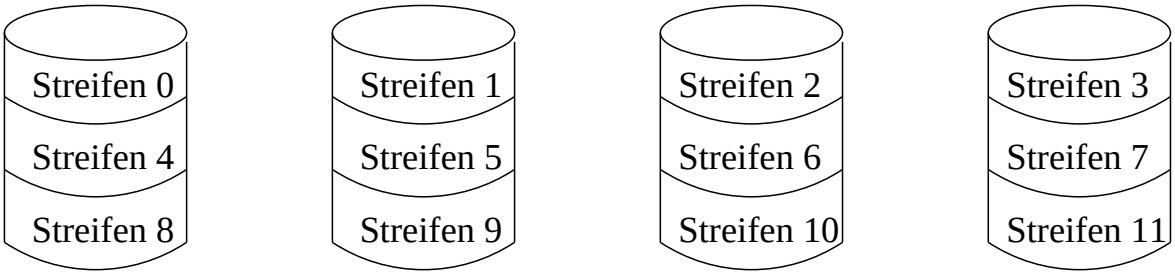


Abbildung 10.5: RAID-System für Level 0, das aus 4 Einzelplatten besteht

Striping bezeichnet. Von Vorteil ist: Wenn der RAID-Controller den Befehl erhält, einen aus z.B. vier aufeinanderfolgenden Streifen bestehenden Datenblock zu lesen, teilt er diesen in vier separate Befehle auf – einen für jede der Festplatten – und läßt sie parallel laufen. Damit ist RAID Level 0 besonders bei großen Datenabfragen von Vorteil.

Auf der anderen Seite gibt es einen Nachteil: die Ausfallhäufigkeit. Besteht das RAID aus 4 Festplatten mit Ausfallrate 20.000 Stunden, dann kommt es beim RAID System im Mittel alle 5.000 Stunden zu einem Ausfall. Level 0 darf nicht mit JBOD („Just A Buch of Disks“) vertauscht werden. JBOD kombiniert ebenso wie Level 0 mehrere Festplatten zu einem großen virtuellen Festplattenlaufwerk, ermöglicht jedoch Nutzung von unterschiedlich großen Festplatten, da die Daten auf die Festplatten nicht nach dem Modulo-Verfahren verteilt werden. JBOD bietet deshalb kaum Performance-Vorteile im Gegensatz zur Nutzung einer einzigen Platte.

Level 0 - RAID wird meist für Anwendungen verwendet, bei denen große Datenmengen anfallen, die jedoch ohne großen Aufwand wiederhergestellt werden können (z. B. temporäre Laufwerke, Datenpartitionen von Proxy-Servern u.s.w.)

Level 1

Eine ausgezeichnete Fehlertoleranz ergibt sich durch Duplikation aller vier Festplatten. Man erhält vier Primär- und vier Backupplatten. Beim Schreiben werden alle Daten doppelt geschrieben. Beim Lesen kann jede Kopie genutzt werden und damit ggf. doppelt so schnell gelesen werden.

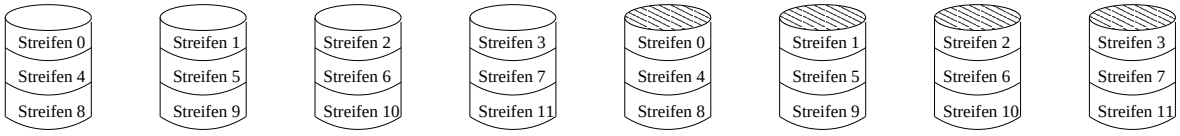


Abbildung 10.6: RAID-System für Level 1 mit 4 Primär und 4 Sekundärplatten

Wenn ein Laufwerk versagt, wird statt dessen einfach die Kopie benutzt. Bei der Wiederherstellung wird ein neues Laufwerk installiert und eine Kopie des Original- oder Backup-Laufwerks erstellt.

Level 1 - RAID wird meist bei günstigen Systemen mit nur 2 Festplatten verwendet, bei denen mittlere Performance ausreicht, hohe Datensicherheit jedoch erforderlich ist. Hierbei kommt oft IDE-Technologie zum Einsatz.

Level 2

Level 2 arbeitet auf Wort- oder Bytebasis. Im Falle eines Bytes wird dieses in 2 Halbbytes á 4 Bit aufgeteilt, denen je ein Hamming-Code hinzugefügt wird, wodurch 7 Bit erhalten werden. Die 7 Laufwerke sind hinsichtliche Arm- und Rotationsposition synchronisiert, so daß kein Striping realisiert wird, sondern auf jedes Laufwerk ein Bit verteilt wird.

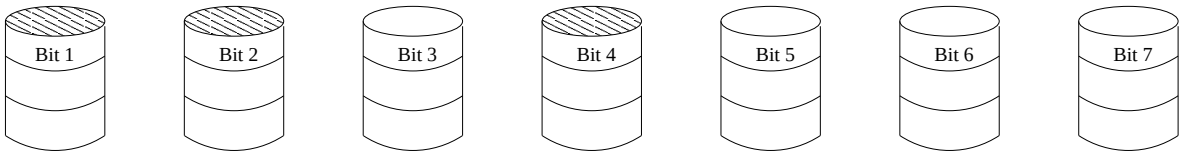


Abbildung 10.7: RAID-System für Level 2 mit 3 Paritätsbits

Die Bits 1,2 und 4 sind Paritätsbits. Dabei wird dem Controller viel abverlangt, da ständig die Paritäten ausgewertet (beim Lesen) bzw. berechnet werden müssen (beim Schreiben).

Level 3

Das RAID Level 3 ist eine vereinfachte Version von RAID Level 2 in dem Sinne, daß für jedes Datenwort nur ein einzelnes Paritätsbit berechnet und auf ein Paritätslaufwerk geschrieben wird, siehe Abbildung 10.8. Damit ist auf jeden Fall Fehlererkennung möglich. Da i.a. jedoch die Position des fehlerhaften Bits bekannt ist, kann auch eine Fehlerkorrektur erfolgen.

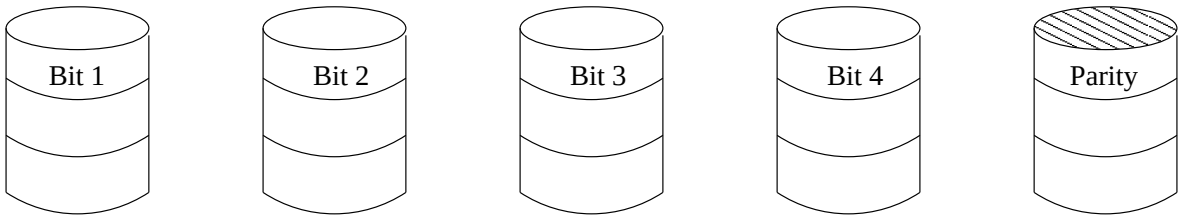
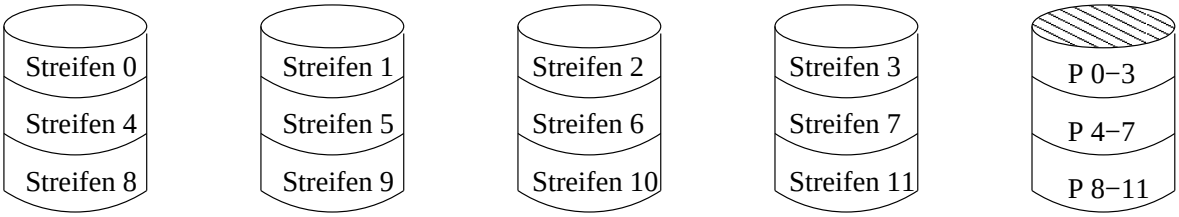


Abbildung 10.8: RAID-System für Level 3 mit einem Paritätsbit

Die RAID-Systeme des Level 2 und 3 bieten sehr hohe Datenraten, jedoch ist keine echte Parallelität von Anforderungen der Ein- und Ausgaben möglich wie Level 0 und 1.

Level 4

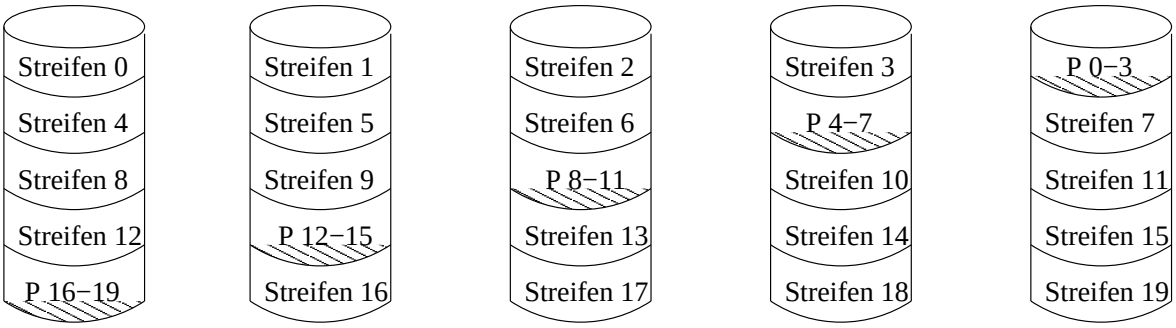
Nun wird wieder mit Streifen gearbeitet. Daher sind keine synchronisierten Laufwerke erforderlich. RAID Level 4 entspricht RAID Level 0 um ein weiteres Laufwerk ergänzt, auf das die Parität geschrieben wird, ggf. als XOR-Funktion.



Das Design bietet Schutz bei Datenverlust, doch die Leistung ist bei kleinen Aktualisierungen ungünstig. Wenn nur ein Sektor geändert wird, müssen alle Laufwerke gelesen werden, um die Parität neu berechnen zu können. Alternativ könnte man die alten und neuen Daten vergleichen und die Parität neu berechnen, aber das ist gleichermaßen aufwändig. Folglich wird das Paritätslaufwerk ein Flaschenhals, der mit Level 5 behoben wird.

Level 5

Dieses Level entspricht dem Prinzip von Level 4, jedoch mit verteilten Paritäten.



In der Praxis ist Level 5 der am meisten verwendete RAID-Level. Dieser kombiniert hohe Performance und hohe Datensicherheit.

Level 10

Bei Level 10 handelt es sich um eine Kombination von Level 1 und Level 0 - RAID. Hierbei werden mehre Festplatten zu einer großen virtuellen Festplatte zusammengefügt (RAID-0) und diese virtuelle Festplatte wird anschließend auf eine 2. identische virtuelle Festplatte gespiegelt (RAID-1).

RAID-Systeme werden entweder über Software-Treiber (sog. Software-RAID) oder Hardware (sog. Hardware-RAID) realisiert. Beim Software-RAID übernimmt die CPU die Vertreilung der Daten, eine Hardware-RAID-Lösung beinhaltet eigenen Prozessor, der diese Aufgabe wahrnimmt. Die meisten RAID-Lösungen sind für Hot-Swap-Einsatz (Austausch der Festplatten im laufwenden Betrieb) ausgelegt, oft wird eine zusätzliche leere Festplatte

als "hot-spare" - Festplatte eingebaut: beim Ausfall einer der Festplatten bei einem redundanten RAID-Level (d. h. alle RAID außer RAID-0) erkennt der Kontroller automatisch den Fehler und bindet die zusätzliche Festplatte anstelle des defekten Laufwerkes automatisch ein.

10.6 Disketten

Im Zeitalter der PC's wurde es nötig, Software zu verteilen. Dazu bot die Diskette ein nützliches Hilfsmittel. Die Bezeichnung Floppy Disk: floppy = schlapp, wabbelig geht darauf zurück, dass die ersten dieser Datenträger biegsam waren.

Die Eigenschaften entsprechen denen von Festplatten mit einem Unterschied: Bei einer Festplatte schwebt der Kopf knapp über der Oberfläche auf einem Kissen von sich schnell bewegender Luft. Bei der Magnetscheibe in der Diskette wird diese vom Kopf berührt. Daher hat eine Diskette einen höheren Verschleiß von Medium und Kopf. Um diesen Verschleiß zu reduzieren, wird der Kopf bei Personalcomputern abgehoben und die Umdrehung angehalten, wenn das Laufwerk weder liest noch schreibt. Beim nächsten Schreib- oder Lesevorgang gibt es jedoch erst einmal eine Verzögerung von etwa einer halben Sekunde, bis der Diskettenmotor hochgefahren ist.

Disketten gibt es in zwei Größen und jeweils als Low-Density (LD) und High-Density(HD). Im Gegensatz zu den älteren 5 $\frac{1}{4}$-Zoll-Disketten stecken die 3 $\frac{1}{2}$-Zoll-Disketten in einer festen Schutzhülle und zählen damit wörtlich genommen nicht mehr zur Gattung der Floppys. Sie sind deshalb auch besser geschützt und können zudem mehr Daten aufnehmen.

Art	$5\frac{1}{4}$**" LD**	$5\frac{1}{4}$**" HD**	$3\frac{1}{2}$**" LD**	$3\frac{1}{2}$**" HD**
Größe (Zoll)	$5\frac{1}{4}$	$5\frac{1}{4}$	$3\frac{1}{2}$	$3\frac{1}{2}$
Kapazität (Byte)	360 KByte	1,2 MByte	720 KByte	1,44 MByte
Spuren	40	80	80	80
Sektoren / Spur	9	15	9	18
Köpfe	2	2	2	2
Umdrehungen / Minute	300	360	300	300
Datenrate (Kbps)	250	500	250	500
Typ	biegsam	biegsam	fest	fest

10.7 CD-ROM

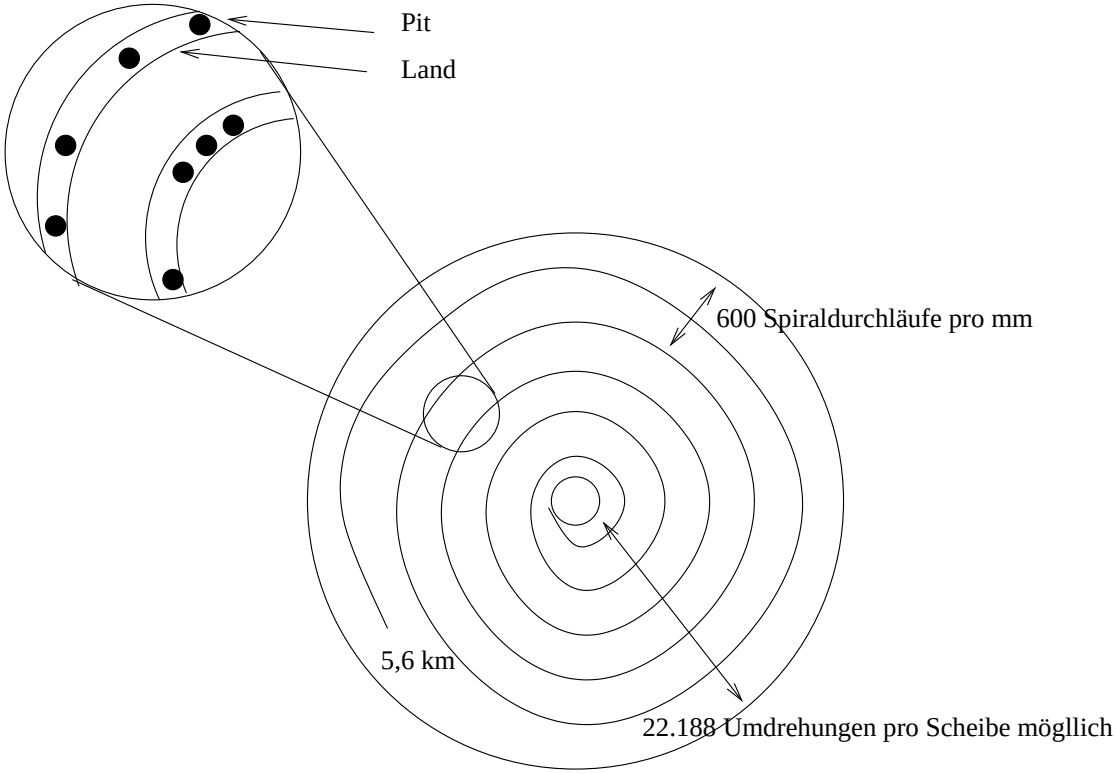
Im Gegensatz zu den bislang magnetischen Datenträgern wollen wir nun die – in den letzen Jahren entwickelten – optischen Platten betrachten. Sie weisen viel höhere Speicherdichten auf.

Die optischen Platten der ersten Generation wurden vom niederlängischen Elektronikkonzern Philips zur Speicherung von Filmen und Aufzeichnung von Fernsehprogrammen erfunden. Sie hatten einen Durchmesser von 30cm und wurden unter der Bezeichnung Laser Vision vermarktet.

1980 entwickelte Philips zusammen mit Sony die CD (Compact Disc), die schnell die Schallplatten aus Vinyl verdrängte. Platten- und Laufwerkspezifikation wurden als Internationaler Standard veröffentlicht, so dass CDs verschiedener Musikgesellschaften und Abspielgeräte verschiedener Elektronikhersteller problemlos zusammenarbeiten können.

Alle CDs haben einen Durchmesser von 12 cm, sind 1,2 mm dick und haben ein 1,5 cm großes Loch. Sie sollen 100 Jahre halten. (Bitte im Jahr 2080 unbedingt prüfen, ob noch alle CDs in Ordnung sind!)

CDs werden – in der Nähe des Mittellochs beginnend – in einer einzigen nach außen fortlaufenden Spirale beschrieben. Diese Spirale macht 22.188 Umdrehungen um die Scheibe (ca. 600 pro mm) und ist abgewickelt 5,6 km lang.



Zur Herstellung einer CD werden mit einem Hochleistungslaser Löcher von 0.8 Mikron Durchmesser in eine beschichtete Glas-Masterdisk gebrannt. Von diesem Master wird eine Negativ-Form hergestellt, die mit Polykarbonatharz ausgespritzt wird. So entsteht eine CD mit gleichem Lochmuster wie die Masterdisk. Auf das Polykarbonat wird eine sehr dünne Schicht aus reflektierendem Aluminium, eine Lack-Schutzschicht sowie ein Etikett aufgebracht.

Die Vertiefungen im Polykarbonatsubstrat heißen **Pits** und die nicht gebrannten Flächen zwischen den Pits heißen **Lands**.

Beim Abspielen wirft eine Laserdiode Infrarotlicht auf die Pits und Lands von der Polykarbonatseite her. Damit erscheinen dem Laser die Pits als Erhebungen, die eine Höhe von einem Viertel der Wellenlänge des Laserlichts haben. Bei der Hin-und-Zurück-Reflektion ergibt sich eine halbe Wellenlänge Unterschied. Dadurch kann das Abspeilgerät Pits von Lands unterscheiden.

In der Praxis wird ein Pit/Land - oder ein Land/Pit-Übergang als 0 definiert. Die Abwesenheit eines Übergang bedeutet eine 1. Damit die Musik gleichmäßig schnell abgespielt wird, müssen die Pits und Lands mit konstanter linearer Geschwindigkeit verbeoströmen. Aus diesem Grund muß die Drehzahl der CD kontinuierlich verringert werden, während der Lesekopf sich von innen nach außen über die CD bewegt.

Um auf 120 cm/s zu kommen schwankt die Drehgeschwindigkeit zwischen 200 UpM (außen) und 530 UpM (innen).

Damit unterscheidet sich ein CD-Laufwerk prinzipiell von einem Festplattenlaufwerk, das sehr schnell mit konstanter Winkelgeschwindigkeit arbeitet. Ferner sind die Festplattenlaufwerke mit 3600 und 7200 UpM deutlich schneller.

1984 wurde von Philips und Sony ein Standard für CDs zur Speicherung von Computer-Daten veröffentlicht – die sogenannten CD-ROMs (Comapt Disk – Read Only Memory). Diese sind genauso groß wie Audio-CDs sowie mechanisch und optisch kompatibel. Neu ist die Formatierung sowie Möglichkeiten der Fehlerkorrektur.

Bei einer CD gibt es Frames von 588 Bits. Diese sind Datenübertragungsblöcke, die aus 192 Bit = 24 Byte Nutzdaten und 396 Bits für die Fehlerkorrektur und Steuerung bestehen.

Bei den CD-Roms werden 98 Frames zu einem CD-ROM-Sektor gruppiert, der mit einer 16-Byte-Präambel beginnt. Die ersten 12 Byte sind hexadezimal immer 00.FF.FF.FF.FF.FF.FF.-FF.FF.FF.FF.00. Damit erkennt das Abspielgerät den Start einers CD-ROM-Sektors. Es folgen 3 Byte für die Sektornummer und ein Byte für einen **Modus**.

Durch die nur ein Spur auf der CD ist die Kopfpositionierung viel komplizierter als bei Festplatten mit konzentrischen Spuren. Bei der Positionierung berechnet die Laufwerk-Software den ungefähren Ort, bewegt den Kopf dorthin und wertet dann die Präambel aus – insbesondere deren Sektornummer.

Für CD-ROMs sind zwei Modi definiert: Modus 1 arbeitet mit 2048 Datenbytes und 288 Fehlerkorrekturbytes. Modus 2 kombinert dies zu 2336 Datenbytes. Damit ergibt sich für den Modus 1 eine Datenrate von 153.600 Byte/s und für Modus 2 175.200 Byte/s. Im Vergleich zu einer SCSI-2-Festplatte mit 10 MByte/s sind CD-ROMs damit extrem langsam. Dafür haben sie jedoch mit einer Kapazität von 650 MByte in Modus 1 eine sehr hohe Speicherkapazität im Verhältnis zu den extrem niedrigen Herstellungskosten.

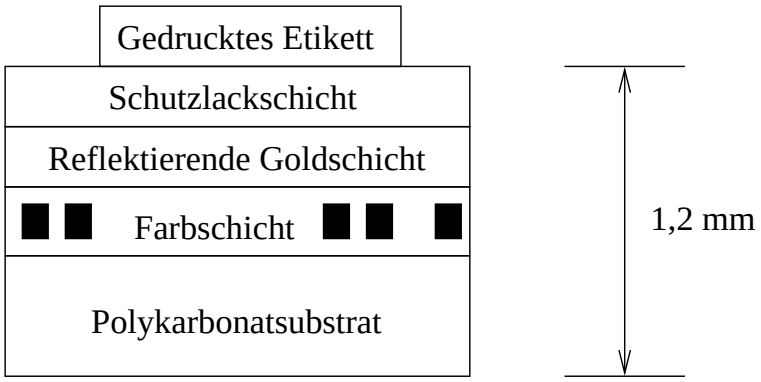
10.8 Beschreibbare CDs

Beschreibbare CDs kamen Mitte der 90er Jahre auf. Sie unterteilten sich in einmal beschreibbare CDs und wiederbeschreibbare CDs.

Einmalbeschreibbare CDs

Im Gegensatz zu den Festplatten ist bei diesen CDs kein Löschen und Neu-Beschreiben möglich.

Sogenannte CD-Rs (CD-Recordables) sehen wie gewöhnliche CD-ROMs aus. Sie sind jedoch an der Oberfläche golden statt silbern. Unterhalb der reflektierenden Goldschicht – also zwischen Lesegerät und Goldschicht – befindet sich eine Farbschicht, in die beim Schreiben eine dunkle Stelle eingebrannt wird. Damit kann das Lcht an der Goldschcht nicht mehr reflektiert werden und der Unterschied zwischen Pits und Lands wird simuliert.



1989 wurde ein neues Format der CD-R publiziert, die CD-ROM XA, die es erlaubt CD-Rs nach und nach sektorenweise zu beschreiben. Eine Gruppe aufeinanderfolgender beschriebener Sektoren wird CD-ROM-Spur (CD-ROM-Track) genannt.

Zu den frühen Einsatzarten der CD-Rs gehörte die Kodak Photo CD.

Bei CD-Rs gibt es ein Problem: jede Spur muß ohne Halt in einem einzigen zusammenhängenden Vorgang beschrieben werden. Daher muß die Festplatte, von der die Daten kommen, schnell genug sein, um diese Daten auch rechtzeitig auszuliefern, ohne dass Suchzeiten dadurch entstehen, dass Daten über die Festplatte verstreut sind. Solche Pufferunterläufe werden dadurch vermieden, dass CD-R-Software die Möglichkeit bietet, vor dem Brennen der CD-R alle Eingabedaten zu einem einzigen zusammenhängenden 650 MByte-CD-Rom-Dokument zusammenzufassen. Dieser Vorgang hat zwei Nachteile: er benötigt 650 MByte freien Plattenspeicher und verdopplet quasi die tatsächliche Schreibzeit.

Aus Sicht der Wahrnehmung von Urheberrechten gibt es vielfältigste Bestrebungen, das einfache Kopieren von CD-ROMs und Audio-CDs auf CD-Rs zu vermeiden bzw. maximal zu erschweren.

Wiederbeschreibbare CDs

Die CD-RW (Rewritable) arbeitet mit einer Legierungsschicht aus z.B. Silber, die zwei Zustände mit unterschiedlicher Reflexionsfähgiketi kennt: Kristallin und amorph. CD-RW-Laufwerke nutzen drei Leistungsstufen: bei hoher Leistung schmilzt der Laser die Legierung, so dass der niedrigreflektierende amorphe Zustand entsteht, der ein Pit repräsentiert. Bei mittlerer Leistung schmilzt die Legierung und stellt den natürlichen kristallinen Zustand wieder her, der einem Land enstpricht. Und bei niedriger Leistung wird der Zustand des Materials zum Lesen abgegriffen.

Warum hat die CD-RW die CD-R noch nicht vollständig abgelöst?

CD-RW-Blanks sind teuerer als CD-R-Blanks. Außerdem gibt es Anwendungen wie z.B. das Sichern von Daten, die auch versehentlich nicht gelöscht werden sollen.

10.9 DVD

DVDs (Digital Video Disk oder Digital Versatile Disk) sind generell wie CDs aufgebaut, die auch mit Vertiefungen (Pits) und ebenen Stellen (Lands) arbeiten. Auch sie werden von einer Laserdiode angestrahlt und von einem Fotodetektor gelesen.

Im Unterschied zu den CDs haben DVDs folgende Eigenschaften:

- kleinere Pits (0,4 Mikron statt 0,8)
- engere Spiralen (0,74 Mikron Spurenabstand statt 1,6)
- Übergang zu Rotlaser (wie er auch an Kassen von Supermärkten verwendet wird)

Vorteile von DVDs im Gegensatz zu CDs sind darin zu sehen, dass die Kapazität von 650 MByte auf 4,7 GByte steigt (das 7,2-fache). Außerdem steigt die Bandbreite bei DVD-Geräten durch den Übergang zu Rotlaser bei einfacher Lesegewschwindigketi auf 1,4 MByte/s im Gegensatz zu 150 KByte/s bei CDs.

Der Nachteil der DVD-Technik ist darin zu sehen, dass herkömmliche CDs und CD-ROMs in einem DVD-Spieler nur mit einem zweiten Laser oder einer sehr aufwändigen Konvertireungsoptik abzuspielen sind.

Welche Bedeutung haben 4,7 GByte Speicherkapazität?
Mit MPEG-2-Komprimierung kann eine DVD so 133 Minuten Videofilm in Vollbildgröße bei hoher Auflödung (720 x 480) einschließlich Tonspuren in 8 Sprachen sowie Untertiteln in weiteren 32 Sprachen aufnehmen. Damit sind 92% aller Hollywood-Filme speicherbar, da diese maximal 133 Minten lang sind.

Für die restlichen 8% Filme, für Multimedia-Spiele oder Nachschlagewerke gibt es eine Klassifikation in insgesamt 4 DVD-Formate, von denen drei mehr Speicherkapazität haben:

	Eine Schicht (Single-layer)	Zwei Schichten (Dual-layer)
Einseitig (Single-sided)	4,7 GByte	8,5 GByte
Zweiseitig (Double-Sided)	9,4 GByte	17 GByte

Bei der Zwei-Schichten-Technik befindet sich eine reflektierende Schicht am Boden und darüber eine semi-reflektierende Schicht. Da die untere Schicht geringfügig größere Pits und Lands benötigt, ist ihre Kapazität etwas kleiner als die der oberen Schicht.

Durch das Zusammenkleben von zwei einseitigen 0,6 mm-Platten Rücken an Rücken werden zweiseitige Platten produziert, die vom Nutzer umgedereht werden müssen.

Teil IV

Ein- und Ausgabe

11

Ein- und Ausgabe

In Kapitel 3 wurde die grundlegende Struktur eines Computers mit den drei Hauptkomponenten Prozessor, Speicher und Geräten für die Ein- und Ausgabe (Input/Output, I/O) eingeführt. In diesem Kapitel werden nun die Ein- und Ausgabegeräte (E/A-Geräte) genauer betrachtet. Diese sind für die Übertragung von Daten in und aus dem Computer zuständig und ermöglichen die Kommunikation mit Benutzern und anderen Maschinen. In den meisten Fällen müssen die Daten bei der Ein- und Ausgabe aufbereitet werden, da die interne Darstellung von der externen abweicht. E/A-Geräte sind mit Prozessor und Speicher über einen oder mehrere Busse verbunden, sie enthalten auch in zunehmendem Maße eigene Prozessoren und Speicher. Einen Überblick zu Ein- und Ausgabegeräten gibt Tabelle 11.1, Beispiele, die hier genauer betrachtet werden, sind Tastaturen, Mäuse, Monitore, Drucker und Modems.

11.1 Tastatur

Tastaturen gibt es in mehreren Varianten. Preiswerte Tastaturen von heute haben Tasten, die beim Drücken einen mechanischen Kontakt herstellen. Bei hochwertigeren Modellen liegt zwischen Taste und der darunterliegenden Leiterplatte eine Folie aus gummiartigem Material. Unter jeder Taste befindet sich eine kleine kuppelartige Wölbung, die nachgibt, wenn man die Taste drückt. Ein Stück leitfähiges Material in der Wölbung schließt den Stromkreis. Bei manchen Tastaturen liegt unter jeder Taste ein Magnet, der beim Anschlag der Taste durch eine Spule bewegt wird und einen Strom induziert, der erkannt werden kann. Es sind auch noch weitere mechanische und elektromagnetische Methoden in Gebrauch.

Beim Anschlag einer Taste am Computer wird eine Unterbrechung ausgelöst und der Tastatur-Interrupt-Handler übernimmt. Der Handler liest ein Hardware-Register im Tastatur-Controller

Gerät	**Verhalten**	**Partner**	**Art**
Tastatur	Eingabe	Mensch	
Maus	Eingabe	Mensch	
Trackball	Eingabe	Mensch	
Trackpad	Eingabe	Mensch	
Trackpoint	Eingabe	Mensch	
Touchpad	Eingabe	Mensch	
Joystick	Eingabe	Mensch	
Gamepad	Eingabe	Mensch	
Lightpen / Lightgun	Eingabe	Mensch	
Grafiktablett	Eingabe	Mensch	
Scanner	Eingabe	Mensch	
Digitalkamera	Eingabe	Mensch	
Datenhandschuh / Datenanzug	Eingabe	Mensch	
Mikrofon	Eingabe	Mensch	
Touchscreen	Eingabe	Mensch	
Bildschirme	Ein-/Ausgabe	Mensch	visuell (flüchtig)
Beamer	Ausgabe	Mensch	visuell (flüchtig)
Head-Mounted Display	Ausgabe	Mensch	visuell (flüchtig)
Drucker	Ausgabe	Mensch	visuell (dauerhaft)
Plotter	Ausgabe	Mensch	visuell (dauerhaft)
Lautsprecher	Ausgabe	Mensch	akustisch
Braillezeile	Ausgabe	Mensch	haptisch
Modem	Ein-/Ausgabe	Maschine	
Netzwerk	Ein-/Ausgabe	Maschine	

Tabelle 11.1: Überblick über E/A-Geräte

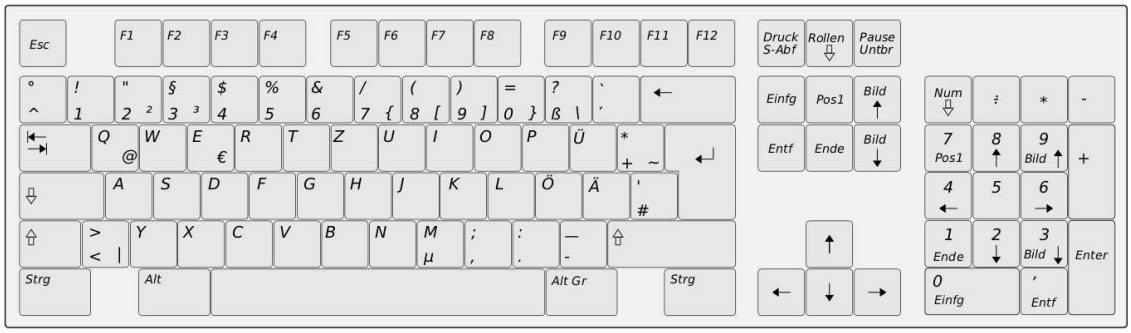


Abbildung 11.1: Deutsche Tastatur

aus, um die Nummer der gedrückten Taste abzufragen, den so genannten Scancode. Das Loslassen der Taste bewirkt eine weitere Unterbrechung. Dies ist besonders wichtig für die Kombination von Tasten, z.B. Shift und ein Buchstabe um einen Großbuchstaben zu erhalten. Angeschlossen werden Tastaturen heutzutage meist über eine PS/2- oder USB-Schnittstelle. Bei drahtlosen Tastaturen findet die Übertragung meist per Funk oder Infrarot statt und es wird nur das Empfangsmodul mit einer Schnittstelle am Computer verbunden. In Deutschland werden vor allem Tastaturen mit dem so genannten QWERTZ-Layout eingesetzt (siehe Abb. 11.1), in den USA wird das QWERTY-Layout genutzt. Nach heutigem Kenntnisstand entsprechen beide nicht modernen ergonomischen Anforderungen, dahingehend optimierte Tastaturen setzten sich bisher aber nicht durch.

11.2 Maus

Mäuse sind üblicherweise aus Kunststoff und besitzen neben 1 bis 3 Tasten heutzutage oft auch ein sogenanntes Scrollrad. Es gibt im Wesentlichen drei verschiedene Arten: mechanische, optomechanische und optische Mäuse. Bei der mechanische Maus steuert eine Kugel, die aus der Maus unten herausragt, im Inneren zwei Rädchen, ein horizontales und ein vertikales. Anhand der Fortbewegung der Rädchen wird ermittelt, wie stark die Bewegung in eine bestimmte Richtung ausgeführt wurde. Die optomechanische Maus funktioniert sehr ähnlich, hier wird die Rollbewegung der Kugel über zwei orthogonal zueinander stehende Achsen an Kodierer weitergeleitet. Durch Schlitze an den Kodierern dringt Licht, und wenn die Maus bewegt wird rotieren die Achsen und Lichtimpulse treffen bei den Detektoren ein. Bei optischen Mäusen sitzen an der Unterseite eine Leuchtdiode (LED, light emitting diode) und ein Photodetektor. Dieser Sensor macht permanent Bilder von der Oberfläche, auf der die Maus bewegt wird und schließt so auf die Bewegung. Probleme haben optische Mäuse auf glatten, unstrukturierten Oberflächen. Neueste Entwicklungen gehen in Richtung Lasermaus, diese kommen mit glatten Oberfläche besser zurecht als klassische optische Mäuse. Alternativ zu einer Maus werden v.a. bei Notebooks auch Touchpads, Trackpads oder Trackpoints eingesetzt; auch ein Trackball kann die Maus ersetzen, statt das Gerät über die Oberfläche zu bewegen wird hierbei eine Kugel vom Daumen in einer Halterung bewegt.
Mäuse senden die ermittelte Bewegung dann üblicherweise in einer Dreier-Sequenz an den Computer. Das erste Byte gibt die Bewegung in Richtung der x-Achse in den letzten 100 ms als vorzeichenbehaftete Ganzzahl an, das zweite die Bewegung in Richtung der y-Achse und das dritte steht für den Status der Maustasten. Manchmal werden auch 2 Byte für jede Information verwendet.

11.3 Monitor

Ein Monitor ist ein Ausgabegerät, die Bildschirmgröße wird in Zoll diagonal von Ecke zu Ecke gemessen.

11.3.1 CRT-Bildschirm

Ein CRT-Bildschirm (cathode ray tube, Kathodenstrahlröhre) enthält eine Elektronenkanone, die einen Elektronenstrahl auf einen phosphoreszierenden Schirm dicht an der Vorderseite der Bildröhre schießen kann (siehe hierzu Abbildung 11.2). Farbmonitore haben drei Elektronenkanonen, je eine für rot, grün und blau (RGB) und ein Pixel ist in drei Leuchtpunkte aufgeteilt. Damit die Leuchtpunkte genau getroffen werden, gibt es in der Lochmaske für jeden Pixel ein Loch und sie verhindert, dass benachbarte Bildpunkte mitleuchten. Beim Rasterbildschirm streicht der Elektronenstrahl während des horizontalen Durchlaufs über den Schirm und zeichnet eine annähernd waagrechte Linie. Danach wird ein Zeilenrücklauf durchgeführt, so dass wieder vom linken Rand gestartet werden kann. Die horizontale Bewegung wird hierbei von einer linear ansteigenden Spannung gesteuert, die an die links und rechts neben der Elektronenkanone angebrachten horizontalen Ablenkungselektroden angelegt wird. Die vertikale Bewegung wird ebenfalls durch eine - deutlich langsamer - linear ansteigende Spannung gesteuert, die an den vertikelen Ablenkungselektroden angelegt wird. Wenn der Strahl in der unteren rechten Ecke angekommen ist, erfolgt ein Rücklauf nach links oben und das Verfahren beginnt von vorne. Das Abtastmuster wird in Abbildung 11.3 dargestellt. Ein Bild wird 70 bis 120 Mal in der Sekunde neu gezeichnet und etwa ab 80 Hz Bildwiederholfrequenz erscheint es als flimmerfrei.

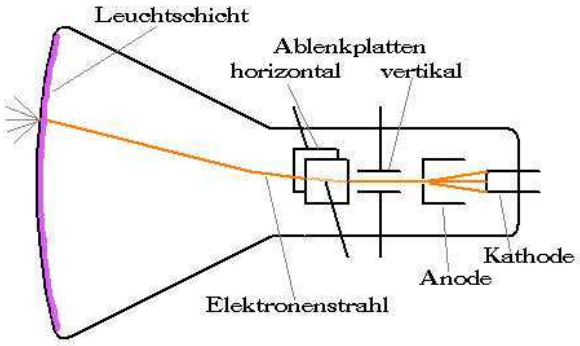


Abbildung 11.2: CRT-Bildschirm

11.3.2 Flachbildschirm

Die leichtere, platzsparende und teurere Variante sind LCDs (liquid crystal display, Flüssigkristallanzeige). Flüssigkristalle sind viskose organische Moleküle, die wie eine Flüssigkeit fließen können, aber auch wie ein Kristall eine Raumstruktur aufweisen. Sind alle Moleküle in der gleichen Richtung angeordnet, hängen die optischen Eigenschaften des Kristalls von Richtung und Polarisation des einfallenden Lichtes ab. Wird ein elektrisches Feld angelegt, so kann die Molekülausrichtung beeinflußt werden. Abbildung 11.4 zeigt den Aufbau eines Flachbildschirmes. Ein LCD besteht aus zwei parallelen Glasplatten, zwischen denen

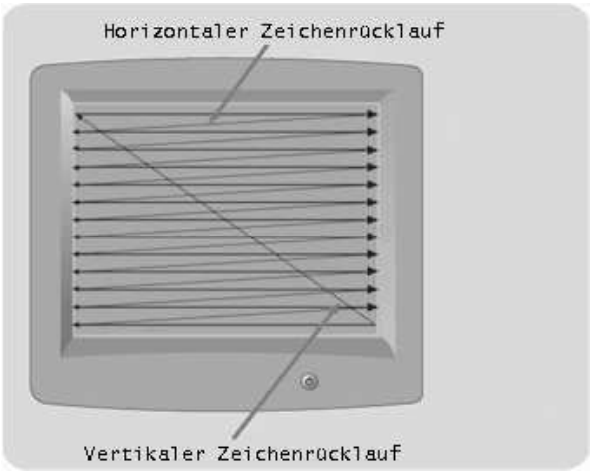


Abbildung 11.3: Abtastmuster

sich ein hermetisch abgeschlossener Flüssigkristall befindet. An beiden Platten liegen transparente Elektroden an, mit denen elektrische Felder im Flüssigkristall erzeugt werden. Der Bildschirm wird durch natürliches oder künstliches Licht von hinten durch die rückwärtige Platte beleuchtet. Verschiedene Teile des Bildschirms haben eine unterschiedliche Spannung, wodurch das Bild bestimmt wird. Vorn und hinten auf dem Bildschirm befinden sich Polarisationsfilter, da die Anzeigetechnik polarisiertes Licht erfordert. Einfallendes Licht wird also vor dem Eintritt in die Flüssigkeit polarisiert. Durch die Verdrillung der Molekülflächen folgt eine Drehung der Polarisationsrichtung des Lichts. Dies hat wiederum zur Folge, dass das Licht den gegenübergesetzten Filter passieren kann und die Zelle hell erscheint. Im Ruhezustand ist das Display durchsichtig, diese Anordnung wird auch Normally-White-Mode genannt. Legt man eine elektrische Spannung an die Elektroden an, so tritt unter dem Einfluss des elektrischen Feldes eine Drehung der Moleküle ein, sodass sie sich senkrecht zu den Elektrodenoberflächen ausrichten. Die Verdrillung ist damit aufgehoben, die Polarisationsrichtung des Lichts wird nicht mehr geändert und damit kann es den zweiten Polarisationsfilter nicht mehr passieren. Die Funktion ist auch umkehrbar: ordnet man die Polarisationsfilter parallel an, dann ist die Zelle ohne Spannung dunkel und mit Spannung hell. Man spricht vom Normally-Black-Mode, welcher wegen des schlechteren Kontrastes selten verwendet wird.

Man unterscheidet zwei verschiedene Verfahren, wie die Bildpunkte eines LC-Monitors angesteuert werden, Passiv-Matrix und Aktiv-Matrix. Bei einem Passiv-Matrix-Bildschirm weisen beide Elektroden parallele Drähte auf. Bei einem Monitor mit 800x600 Bildpunkten könnte beispielsweise die hintere Elektrode aus 800 vertikalen Drähten und die vordere aus 600 horizontalen bestehen. Wenn man nun Spannung an einen der vertikalen Drähte anlegt und einen Impuls auf einen horizontalen Draht gibt, so ändert sich die Spannung in einem bestimmten Bildpunkt, so dass er kurzzeitig dunkel wird. So werden zur Steuerung von 480.000 Bildpunkten nur 1.400 Leitungen benötigt. Analog zur Kathodenstrahlröhre kann man so eine dunkle Linie zeichnen. Ein Monitor zeichnet üblicherweise 60 Mal in der Sekunde den Bildschirm neu und spiegelt dem menschlichen Auge so ein konstantes Bild vor. Das bessere und teurere Aktiv-Matrix-Display hat an jedem Bildpunkt einen aktiven Ver-

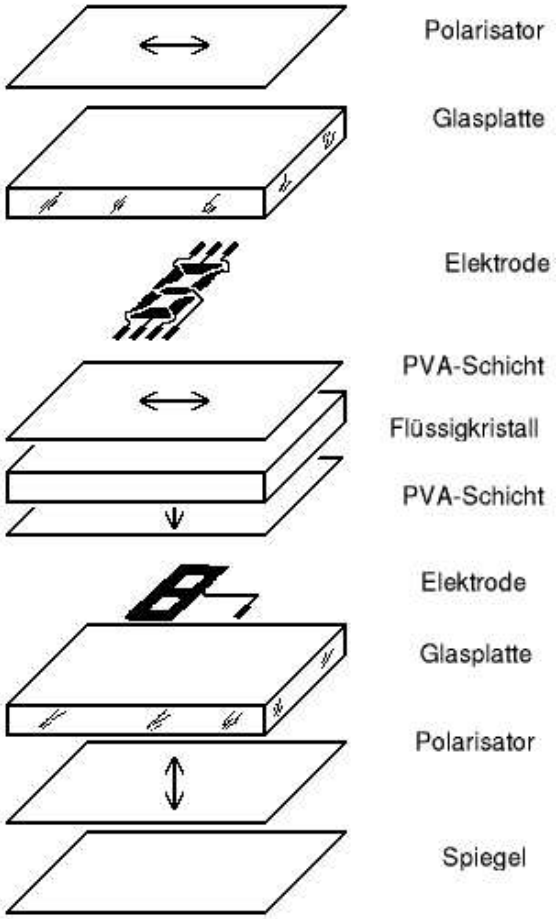


Abbildung 11.4: Aufbau eines LCD

stärker und einen Stromversorgungsanschluß. Das Bild ergibt sich hierbei aus dem Muster der anliegenden elektrischen Spannung. Bei einer Auflösung von 1024x768 werden dabei insgesamt 786.432 Bildpunkte angesteuert. Wichtigster Vertreter von Aktiv-Matrix-Displays sind TFT-Displays (thin-film transistor). Dabei wird der Transistor auf das Glassubstrat direkt aufgedampft.

11.4 Drucker

11.4.1 Monochromdrucker

Matrixdrucker

Beim Matrixdrucker streicht ein Druckkopf mit 7 bis 24 elektromagnetisch aktivierbaren Nadeln über jede Druckzeile. Ein Drucker mit 7 Nadeln kann beispielsweise 80 Zeichen in einer 5x7-Matrix auf eine Zeile bringen. Jede Druckzeile hat dann 7 Zeilen, die aus 5x80 = 400 Zeichen besteht. Jeder Punkt kann entweder gerduckt oder nicht gedruckt werden.

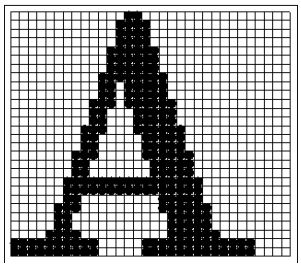


Abbildung 11.5: Matrixdrucker

Die Druckqualität kann durch zwei Techniken verbessert werden:

1. Verwendung von mehr Nadeln

2. Drucken von überlappenden Punkten

Da beim zweiten Verfahren mehrere Durchgänge über jede Zeile gemacht werden müssen, ist dies auch mit einer längeren Druckzeit verbunden. Die meisten Matrixdrucker können daher in verschiedenen Betriebsmodi arbeiten. Matrixdrucker sind günstig, aber auch laut und langsam und kommen daher immer weniger zum Einsatz. Sie bieten allerdings eine gute Möglichkeit, mehrere Durchschläge gleichzeitig auszudrucken, was mit den anderen Verfahren so nicht möglich ist.

Tintenstrahldrucker

Bei Tintenstrahldruckern streicht der bewegliche Druckkopf mit der Tintenpatrone horizontal über das Papier und versprüht aus kleinen Düsen die Tinte. Hierzu gibt es zwei populäre Verfahren: Bubble-Jet-Drucker und Piezo-Drucker. Beim Bubble-Jet-Drucker wird in jeder Düse dazu ein Tintentropfen bis zum Siedepunkt erhitzt, so dass er dann durch die Düse spritzt. Durch Abkühlung der Düse wid ein weiterer Tintentropfen angesaugt und das Verfahren beginnt von vorn. Die Geschwindigkeit eines Tintenstrahldruckers hängt also eng davon ab wie schnell dieser Zyklus wiederholt werden kann, einzelne Heizelemente arbeiten mit einer Frequenz bis 10.000 Hz. Bekannte Vertreter sind beispielsweise Drucker von Canon und Hewlett-Packard.

Piezo-Drucker nutzen die Eigenschaften von Piezokristallen, die sich unter elektrischer Spannung verformen, um die Tinte durch eine feine Düse zu pressen. Es erfolgt eine Tropfenbildung der Tinte, deren Tropfenvolumen sich über den angelegten elektrischen Impuls steuern lässt. Die Arbeitsfrequenz eines Piezokristalls reicht bis zu 16.000 Hz. Bekannte Vertreter kommen hier von Hewlett-Packard, Epson und Lexmark. Die Auflösung von Tintenstrahldruckern variiert stark, heutzutage werden oft 1200 dpi (dots per inch) angeboten. Tintenstrahldrucker sind günstig und bieten eine gute Qualität, aber das Verfahren ist langsam und die Tintenpatronen sind teuer.

Laserdrucker

Im Inneren eines Laserdruckers dreht sich eine Trommel (oder ein Band), welche am Anfang jeder Seite auf 1000 Volt aufgeladen und mit fotoempfindlichem Material überzogen wird (vgl. Abb. 11.6). Dann streicht das Laserlicht horizontal über die Trommel, die waagrechte Ablenkung des Lasers geschieht hierbei durch einen drehbaren achteckigen Spiegel über der Trommel. Der Strahl verursacht ein Muster aus hellen und dunklen Punkten, die Stellen, auf die er auftrifft, verlieren ihre elektromagnetische Ladung. Nach jeder Zeile dreht sich die Trommel ein winziges Stück weiter, so dass die nächste Linie gezeichnet werden kann. Die Linien aus Punkten erreichen nacheinander den Toner, gefüllt mit elektrostatisch empfindlichem, schwarzen Pulver. Der Toner wird von den elektrisch geladenen Punkten angezogen und bildet so das Muster ab. Die Trommel wird dann gegen ein Blatt Papier gedrückt, welches anschließend durch aufgeheizte Rollen geführt wird. Hierbei wird der Toner auf dem Papier fixiert. Die Trommel wird dann entladen und vom überschüssigen Toner befreit, bevor der Zyklus von Neuem beginnt.

Neben dem eigentlichen Drucksystem besteht ein Drucker außerdem noch aus einer CPU und Speicher für die Zwischenspeicherung empfangener Druckaufträge. Druckaufträge können z.B. in der Sprache PostScript an den Drucker gegeben werden. Laserdrucker bieten eine hohe Qualität, verbunden mit einer höheren Geschwindigkeit und annehmbaren Kosten. Die Technik ist der der Kopierer sehr ähnlich, weswegen viele Hersteller auch Kombinationsgeräte anbieten. Das beschriebene Verfahren kann nun zwar Schwarz-Weiß-Seiten in einer bestimmten Auflösung drucken, nicht aber Graustufen. Dieses Problem löst man mit dem sogenannten Half Toning oder Punktschattierung. Dabei wird ein Bild in Halbtonzellen aufgelöst, welche in der Regel aus 6x6-Pixeln bestehen. Jede Zelle kann also zwischen 0 und 36 schwarzen Punkten enthalten und sie wirkt je dunkler, desto mehr schwarze Punkte

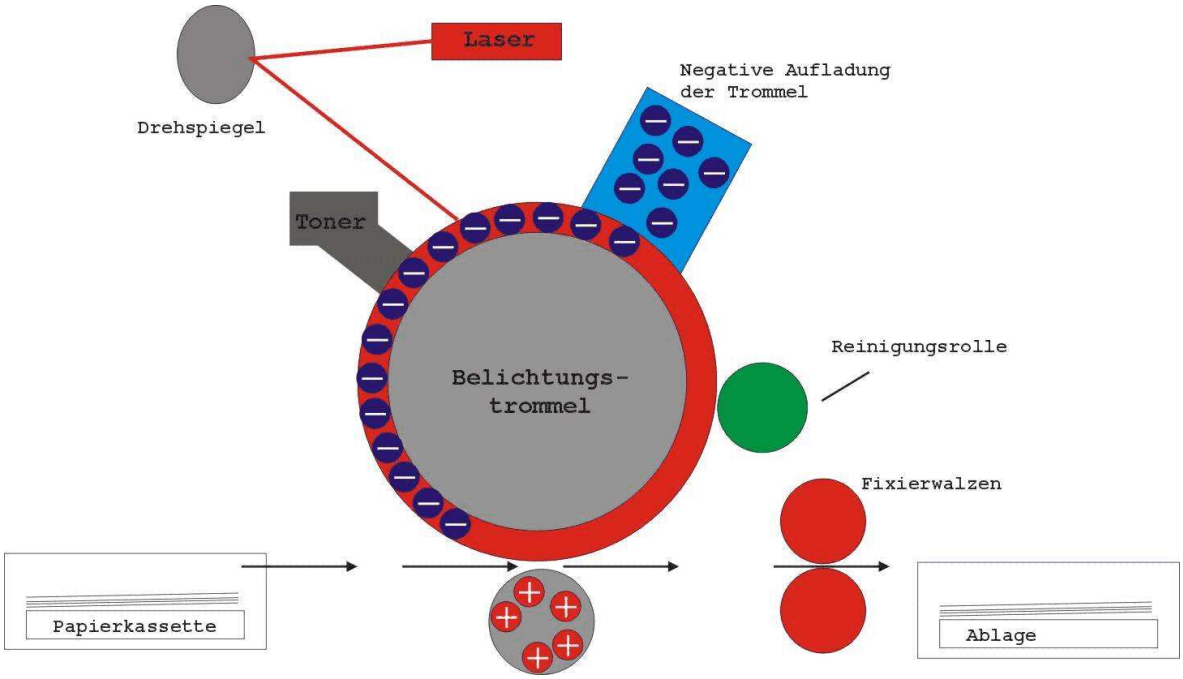


Abbildung 11.6: Aufbau eines Laserdruckers

sie hat. Grauwerte im Bereich von 0 bis 255 werden durch Aufteilung dieses Bereichs in 37 Zonen dargestellt und je nach Zone wird die Halbtonzelle mit einem anderen Punktmuster bedruckt. Die Auflösung verringert sich dabei natürlich und man nennt dies Halbtonrasterfrequenz, sie wird in lines per inch (lpi) gemessen.

11.4.2 Farbdrucker

Bilder mit reflektiertem Licht, wie etwa Farbfotos, absorbieren bestimmte Wellenlängen und reflektieren den Rest. Sie werden durch eine lineare Überlagerung der drei subtraktiven Primärfarben aufgebaut:

- Zyan absorbiert rot
- Gelb absorbiert blau
- Magenta absorbiert grün

Theoretisch kann jede Farbe durch geeignete Kombination dieser drei erzeugt werden, in der Praxis arbeitet aber fast jeder Farbdrucker noch zusätzlich mit schwarer Tinte - CMYK-Drucker. Die Abbildung eines Farbbildes auf dem Monitor in ein identisches Druckbild ist alles andere als trivial, da z.B. Monitore mit durchscheinendem Licht und der RGB-Farbpalette, Drucker hingegen mit reflektiertem Licht und der CMYK-Farbpalette arbeiten. Hier ist viel Erfahrung und Geduld gefragt.

Farbtintenstrahldrucker

Ein Farbtintenstrahldrucker arbeitet wie der monochrome Tintenstrahldrucker, nur dass er vier Tintenpatronen trägt. Man unterscheidet zwei Tintentypen:

1. Flüssigfarbtinte (dye-based ink) besteht aus in einer Trägerflüssigkeit aufgelöstem Farbstoff. Sie liefern leuchtende Farben und fliessen leicht, verblassen aber in ultraviolettem Licht.

2. Pigmenttinte (pigment-based ink) enthält feste Pigmentpartikel, die in einer Trägerflüüssigkeit schweben. Diese verdampft und läßt die Pigmente auf dem Papier zurück. Diese Tinte verblasst nicht, ist aber auch nicht so kräftig in der Farbe.

Feststoffdrucker

Ein Feststoffdrucker arbeitet mit vier festen Blöcken einer speziellen, wachsartigen Tinte. Die Vorlaufzeit dieser Drucker ist hoch, da die Tinte eingeschmolzen werden muss, bevor sie auf das Papier gesprüht wird. Auf dem Papier wird sie wieder fest und durch Walzen fixiert.

Farblaserdrucker

Farblaserdrucker funktionieren wie ihre monochromen Artgenossen, es werden aber Bilder in den vier Grundfarben im Speicher abgelegt und ein Drucker hat vier verschiedene Toner. Der Speicherbedarf ist hier offensichtlich höher.

Wachsdrucker

Ein Wachsdrucker hat ein breites Band mit Wachs in vier Farben, das in seitengroße Streifen unterteilt ist. Heizelemente schmelzen das Wachs auf das vorbeiziehende Papier auf. Das Verbrauchsmaterial ist hier besonders teuer.

Farbsublimationsdrucker

Im Farbsublimationsdrucker streicht ein Träger mit den Farbstoffen über einen thermischen Druckkopf mit programmierbaren Heizelementen. Die Farbstoffe gehen vom festen direkt in den gasförmigen Zustand über (Sublimation), sie verdampfen sofort und werden von Spezialpapier absorbiert. Jedes Heizelement kann 256 verschiedene Temperaturen erzeugen und so wird die Menge an aufzubringendem Farbstoff definiert. Je heisser, desto mehr Farbstoff wird aufgebracht und desto intensiver ist die Farbe. Hiermit ist eine nahezu kontinuierliche Abstufung möglich und man erzeugt sehr realistische Bilder.

11.5 Modem

Es ist heutzutage üblich, dass Computer zur Kommunikation miteinander vernetzt sind, beispielsweise in einem lokalen Netzwerk oder über ein Modem mit einem entfernten Computer. Oft wird dazu die Telefonleitung als zugrundeliegender Kommunikationsweg genutzt. Diese ist als solche allerdings noch nicht geeignet, Computer-Signale zu übertragen. Signale mit zwei Stufen erleiden erhebliche Verzerrungen, wenn sie über eine Telefonleitung übertragen werden, wodurch es zu Übertragungsfehlern kommt. Eine reine Sinusschwingung mit einer Frequenz von 1000 bis 2000 Hz, der sogenannte Träger (Carrier), lässt sich jedoch mit relativ geringen Verzerrungen übertragen. Die Impulse einer Sinuswelle sind gleichmäßig und übertragen dadurch noch keine Information. Diese können erst durch eine Veränderung von Amplitude, Frequenz oder Phase auf den Träger moduliert werden, siehe dazu Abbildung 11.7.

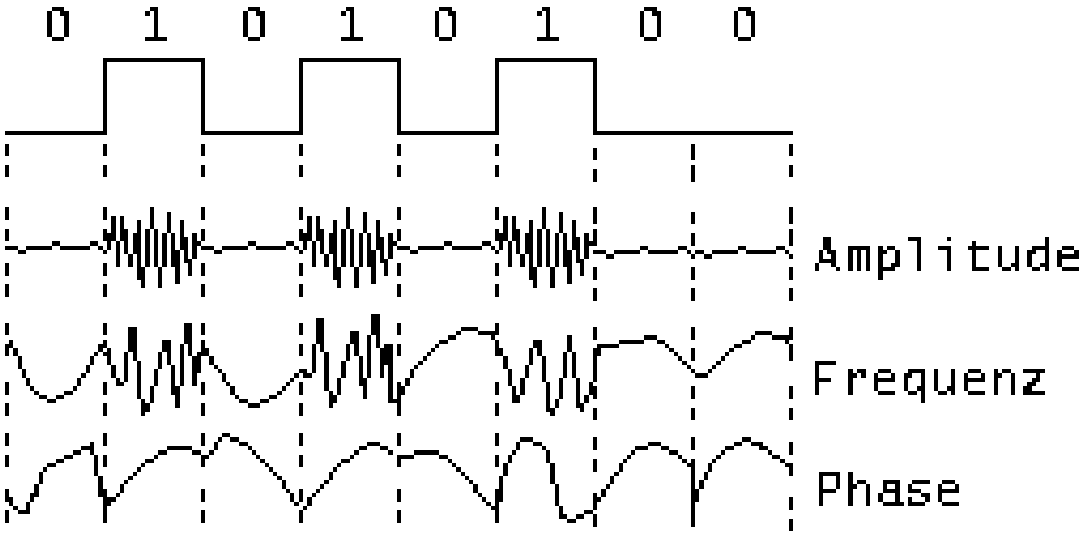


Abbildung 11.7: Übertragung über Telefonleitung mit Amplituden-, Frequenz- und Phasenmodulation

Bei der Amplitudenmodulation arbeitet man mit zwei verschiedenen Spannungen für 0 und 1. Bei einer sehr langsamen Übertragung digitaler Daten würde man die 1 aus lautes Geräusch und die 0 als Stille wahrnehmen. Hingegen ist der Spannungspegel bei der Frequenzmodulation gleichbleibend, aber 0 und 1 werden als unterschiedliche Trägerfrequenzen dargestellt. Hört man sich diese Signale an, so unterscheiden sie sich in der Tonhöhe. Bei der einfachen Phasenmodulation bleibe Amplitude und Frequenz unverändert, während bei einem Signalübergang von 0 auf 1 oder von 1 auf 0 die Phase des Trägersignals umgekehrt (um 180 Grad verschoben) wird. In komplexeren phasenmodulierten Systemen

wird bei Beginn eines jeden unteilbaren Zeitintervalls die Phase der Trägerfrequenz um 45, 135, 225 oder 315 Grad verschoben und so die gleichzeitige Übertragung von 2 Bits ermöglicht (Dibit-Kodierung). In anderen Schemata können durch ähnliche Änderungen auch 3 oder mehr Bits pro Zeitintervall übertragen werden. Die Anzahl der Zeitintervalle (d.h. die Anzahl potentieller Signalveränderungen pro Sekunde) ist die Baudrate. Die Baudrate beschreibt die Anzahl der Signalcodes (Symbol), die pro Sekunde übertragen werden können. Bei zwei oder mehr Bits pro Zeitintervall ist die Bitrate größer als die Baudrate.
8-Bit-Zeichen können nun über eine Telefonleitung nur seriell übertragen werden, daher liefert der Computer die Zeichenreihen bitweise als zweistufige Signale an das Modem. Dieses übernimmt die Modulation (Amplituden-, Frequenz- oder Phasenmodulation) auf das Trägersignal. Beginn und Ende eines Zeichens werden dabei jeweils durch ein weiteres Bit (Startbit und Stopbit) gekennzeichnet, so ergeben sich für die Übertragung jedes 8-Bit-Zeichens 10 Bit. Das sendende Modem schickt die einzelnen Bits in gleich großen Zeitintervallen ab. Das empfangende Modem am anderen Ende der Verbindung wandelt die modulierte Trägerfrequenz wieder in eine Binärzahl um. Durch das Startbit weiß das emfangende Modem, wann ein neues Zeichen beginnt und da die Zeitintervalle auf beiden Seiten gleich groß sind, ist durch den eingebauten Taktgeber auch bekannt wann es aus den eintreffenden Daten die einzelnen Bits lesen kann. Moderne Modems arbeiten mit einer Datenrate von bis zu 57600 Bit pro Sekunde. Zum Einsatz kommt dabei eine Kombination der verschiedenen Modulationsverfahren um mehrere Bits pro Baud zu senden. Fast alle Modems arbeiten im Vollduplex-Betrieb, d.h. es können gleichzeitig Daten in beiden Richtungen übertragen werden, dies wird durch die Nutzung unterschiedlicher Frequenzen ermöglicht. Modems, die nur jeweils in eine Richtung gleichzeitig senden können arbeiten im Halbduplex-Betrieb. Kann überhaupt nur in eine Richtung gesendet werden, so nennt man das Simplex-Betrieb.

11.5.1 ISDN (Integrated Service Digital Network)

ISDN wurde Anfang der 80er Jahre von den europäischen Post- und Fernmeldegesellschaften entwicklet und führte zunächst ein stiefmütterliches Dasein. Erst mit dem Aufbruch in das Internetzeitalter wurde es als Netz zur Datenübertragung populär. Für einen ISDN-Anschluß muss nicht die Leitung selbst, sondern nur die Geräte an beiden Enden ausgetauscht werden. Ein ISDN-Anschluß besteht aus zwei unabhängigen Kanälen mit 64000 Bit/s und einem Signalkanal mit 16000 Bit/s, welche mit vielen Geräten auch zusammen geschaltet werden können. ISDN hat neben der höheren Geschwindigkeit auch den Vorteil, dass es zuverlässiger ist als eine analoge Datenübertragung und der Verbindungsaufbau deutlich schneller zustande kommt.

11.5.2 DSL (Digital Subsciber Line)

DSL ist eine Entwicklung der letzten Jahre, die relativ breitbandige Übertragung über das herkömmliche Telefonleitungssystem (POTS - plain old telephone service) mit Kupferkabeln ermöglicht. Es wird ein Frequenzspektrum genutzt, das bisher brach lag, so dass gleichzeitig die gewohnten Telefoniedienste und additiv dazu die Datendienste verwendet werden können. Es handelt sich bei DSL um eine Familie von Protokollen, xDSL, mit unterschiedlichen

Charakteristika. Um die hohe Datenrate zu erreichen werden geeignete Modulationsverfahren eingesetzt, die je nach DSL-Variante variieren können. Bis 1996 war CAP (carierless amplitude modulation) der de-facto Standard, mittlerweile wird DM/OFDM (discrete multitone technology / orthogonal frequency-division multiplexing) eingesetzt. Meist wird ADSL (asymmetric digital subscriber line) angeboten, bei dem der upstream deutlich schmalbandiger ist als der downstream zum Nutzer. Eine typische Konfiguration bietet Downstream-Raten zwischen 256kbit/s und 8Mbit/s innerhalb einer Distanz von einigen huntert Meter um die Vermittlungsstelle, der Upstream liegt dabei zwischen 64kbit/s und 1024kbit/s und liegt üblicherweise bei 256kbit/s.

Teil V

Abarbeitung von Maschinenbefehlen

12

Vom Programm zum Maschinenprogramm

12.1 Einführung

In der Schule, im Selbststudium, in Info I und Info II war die Erstellung von Computerprogrammen bereits oft betrachtet worden.

Computerprogramme bestehen aus einer Abfolge von Befehlen, die dem Programmierer ein geeignetes Mittel geben, ein vorhandenes Problem recht übersichtlich in seinen Denkstrukturen zu lösen. Solche Computerprogramme wollen wir im folgenden als **höhersprachliche Programme** oder auch **Programme in Hochsprache** bezeichnen. Die eigentliche Erstellung dieser Programme soll als bekannt vorausgesetzt werden – sie ist Gegenstand anderer Lehrveranstaltungen. Im folgenden wollen wir vielmehr kurz anschauen, wie wir zu solchen Programme kommen (Abschnitt 12.1.1). Danach soll betrachtet werden, wie derartige Programme im Rechner abgearbeitet werden (Abschnitt 12.1.2).

12.1.1 Entwicklung eines Programms

Programme werden immer größer und komplexer. Daher gibt es seit den 70er Jahren eine Disziplin innerhalb der Informatik, die sich mit der wissenschaftlichen Entwicklung großer Programmsysteme beschäftigt: das **Software Engineering.**

Die Bearbeitung von Softwareprojekten wird dabei in mehrere voneinander abgegrenzte Phasen unterteilt. Man erhält einen Softwarelebenszyklus, einen sogenannten **Software Life Cycle**. Der Vorteil dieser Vorgehensweise besteht in der Definition von Meilensteinen. Dadurch kann der Projektfortschritt ständig kontrolliert werden, insbesondere bevor man zur nächsten Phase übergeht.

Prinzipiell ist der Ablauf eines Software Life Cycle zwar eindeutig, es gibt jedoch keine einheitliche Form der Darstellung. D.h. in der Literatur existieren verschiedene Ausprägungen dieses Software Life Cycle hinsichtlich der Gliederung der Phasen in weitere Unterphasen und Iterationen von Phasen.

Im folgenden soll das Grundprinzip vorgestellt werden.

1. **Problemanalyse** auch Anforderungsanalyse oder Systemanalyse genannt.
   erfolgt in enger Zusammenarbeit mit dem Auftraggeber (Benutzer, Anwender). Es werden Vorstellungen über die Funktion entwickelt, die das System erbringen soll. Unter Umständen werden Leistungsdaten (Geschwindigkeit, Antwortzeit, ...) oder Entwicklungskosten mit einbezogen.
   Das Ergebnis ist eine informelle Problemspezifikation, die auch Anforderungsbeschreibung oder Pflichtenheft genannt wird.

2. **Systementwurf**
   teilt die zu lösenden Aufgaben in Module auf. Dies erleichtert die Übersichtlichkeit, verbessert die Korrektheit und Zuverlässigkeit. Von Vorteil ist auch, dass verschiedene Programmierteams später an genau festgelegten und gegeneinander abgegrenzten Teilaufgaben parallel arbeiten können.
   Das *Ergebnis* ist eine Systemspezifikation, also eine formale Spezifikation, die als Grundlage für die Implementierung gilt.

3. **Programmentwurf**
   baut die einzelnen Module weiter aus und verfeinert sie. Dabei erfolgt der Übergang vom "Was?" zum "Wie?". Datenstrukturen werden festgelegt und Algorithmen entwickelt.
   Das *Ergebnis* besteht in mehreren Programmspezifikationen.

4. **Implementierung und Test**
   beinhaltet, dass jedes Modul für sich programmiert wird und anhand seiner Spezifikation getestet (verifiziert) wird.
   Das *Ergebnis* besteht in einem Programm, das sich durch Zusammensetzen der einzelnen Module ergibt.

5. **Betrieb und Wartung**
   umfassen die Pflege der Software. Auf gegebenenfalls erweiterte oder geänderte Ansprüche des Benutzers sowie entdeckte Fehler ist geeignet zu reagieren. Unter Umständen führt dies auf Aktionen der Problemanalyse zurück, wodurch ein Zyklus entsteht.

Diese Problematik des Softwarelebenszyklus soll als bekannt vorausgesetzt werden und im Rahmen dieser Vorlesung nicht weiter vertieft werden. Statt dessen wollen wir uns nun mit dem Produkt, einem korrekten Programm, beschäftigen. Insbesondere wollen wir nachvollziehen, wie dieses im Computer abgearbeitet wird.

12.1.2 Verarbeitung eines Programms im Rechner

Wir wollen die Ebene der höhersprachlichen Programme nun zunächst vollständig verlassen und den Computer als Maschine betrachten, die ein Problem löst, in dem sie Befehle

ausführt.

Der Computer wird dabei durch elektronische Schaltungen realisiert, die zwei Einschränkungen haben:

– die Menge der ausführbaren Befehle ist begrenzt und

– es können nur sehr einfache Befehle ausgeführt werden

Solche grundlegenden Befehle sind zum Beispiel:

– Addiere zwei Zahlen oder

– Prüfe eine Zahl, um festzustellen, ob sie Null ist.

Derartige primitive Befehle eines Computers bilden eine sogenannte Sprache, die ein Computer verarbeiten kann. Sie wird **Maschinensprache (machine language)** genannt. Da sehr wenige unterschiedliche Befehle zur Verfügung stehen, braucht man in der Regel sehr viele Befehle, um eine bestimmte Aufgabe zu lösen. Die Gesamtheit der Befehle, die der Ausführung der Aufgabe dient, nennt man dann **Maschinenprogramm**.

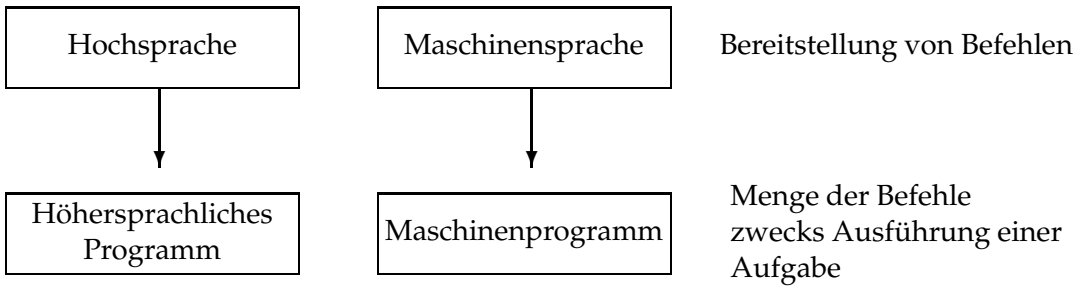


Abbildung 12.1: Sprachen und Programme

Je nach dem, wie ein Rechner gebaut ist, ergibt sich daraus die Menge der Maschinenbefehle, die er verarbeiten kann. Das heißt, Personen, die einen Computer entwerfen, müssen entscheiden, welche Befehle in seine Maschinensprache einzubeziehen sind. Kriterien für diese Entscheidung sind:

- Die Befehle sollen die beabsichtigte Funktionalität und die Leistungsanforderungen realisieren, aber

- sie sollen so einfach wie möglich gehalten werden, um die Komplexität zu reduzieren.

- Die erforderliche Elektronik soll so kostengünstig wie möglich realisierbar sein.

Die resultierende Maschinensprache ist für den Menschen sehr unübersichtlich und daher schwierig zu benutzen. Im Gegensatz dazu sind Programme in Hochsprache für den Programmierer wesentlich übersichtlicher jedoch für die CPU zu komplex und daher umständlich!

Die Lösung des Konflikts besteht darin, den Programmierer in einer Hochsprache arbeiten zu lassen und das Ergebnis, d.h. sein höchersprachliches Programm, auf ein Maschinenprogramm abzubilden. In der Regel wird dabei jeder Befehl der Hochsprache durch eine Folge von Befehlen in Maschinensprache ersetzt. Das reslutierende Programm besteht dann nur aus Befehlen der Maschinensprache und kann vom Computer ausgeführt werden.

Der Übergang von einem höhersprachlichen Programm zu einem Maschinenprogamm kann auf zweierlei Arten erfolgen:

Die erste Variante besteht darin, das vollständige höhersprachliche Programm in eine entsprechende Folge von Maschinenbefehlen zu übersetzen. Diese Technik des Übergangs von der Programmiersprache zur Maschinensprache heißt **Übersetzung** oder auch Compilierung. Innerhalb des Rechners geschieht dies in der Regel mit Hilfe anderer Programme, sogenannter Compiler.

Auch bei der zweiten Lösungsmöglichkeit wird das Programm in einer Maschinensprache ausgeführt. Jedoch werden alle Befehle nacheinander geprüft und direkt als entsprechende Folge von Maschinenbefehlen ausgeführt. Diese Technik, bei der zunächst kein neues Programm in der Maschinensprache erzeugt werden muß, heißt **Interpretation**. Das ausführende Programm wird **Interpreter** genannt.

Compilierung und Interpretation ähneln sich. Programme werden mittels Befehlen einer Programmiersprache geschrieben, aber Folgen von Befehlen einer Maschinensprache werden abgearbeitet.

Der *Unterschied* besteht jedoch darin, dass bei der Compilierung zuerst das gesamte Programm konvertiert wird. Das Programm in Programmiersprache wird zur Ausführung nicht mehr benötigt. Das Programm in der Maschinensprache wird in den Speicher geladen und ausgeführt.

Bei der Interpretation wird jeder Befehl der Programmiersprache nach der Prüfung und Dekodierung unmittelbar ausgeführt. Es wird also kein explizit übersetztes Programm generiert.

Zum Vergleich von Übersetzer und Interpreter soll ein Quellprogarmm aus einer linearen Folge von k Schritten bestehen. Jeder Programmschritt i erfordert einen Decodierungsschritt D_i und einen Ausführungsschritt A_i.

Bei der Compilierung ergibt sich der Ablauf $D_1, D_2, \ldots, D_k, A_1, A_2, \ldots, A_k$. Bei der Interpretation ergibt sich der Ablauf $D_1, A_1, D_2, A_2, \ldots, D_k, A_k$.

Aus diesen unterschiedlichen Verarbeitungsabläufen lassen sich folgende allgemeine Eigenschaften ableiten:

- Bei einem linearen Programm, bei denen sequentiell jeder Befehl genau einmal ausgeführt wird, ist der gesamte Zeitbedarf der beiden Verfahren etwa gleich.

- Falls ein Programm eine Schleife enthält, also Schritt i n-mal ausgeführt wird, so ist der Zeitbedarf beim Interpreter $(n-1) \cdot D_i$-mal höher als beim Compiler.

  Falls ein Programm eine Auswahl enthält, also Schritt i aus n Alternativen besteht, so ist der Zeitbedarf beim Compiler $(n-1) \cdot D_i$-mal höher als beim Interpreter.

- Compilierung ist notwendig oder von Vorteil, wenn eine statische Korrektheit von Programmen vor der Ausführung sichergestellt werden soll, da nur so eine vollständige Decodierung erfolgt.

  Bei einer geforderten hohen Laufzeiteffizienz ist ebenfalls eine Compilierung erforderlich, da diese in der Regel vor der eigentlichen Programmausführung erfolgen kann.

- Interpretation ist notwendig, wenn das Programm selbst während der Laufzeit geändert werden soll oder eine große Interaktivität gefordert ist.

Ein Compiler wandelt das Quellprogramm in mehreren Phasen in das Zielprogramm um. Jede Phase bearbeitet eine abgeschlossene Teilaufgabe. Die Phasen sind durch einen Datenfluß voneinander abhängig.

Jede Compilierung hat zwei grundlegende Phasen:

Analyse: Das Quellprogramm wird in seine Bestandteile zerlegt und es wird eine Zwischendarstellung des Quellprogramms, ein sogenannter **Parse–Baum** erzeugt.

Synthese: Aus dem Parse–Baum wird das gewünschte Zielprogramm konstruiert (in der Regel aufwendiger als die Analyse)

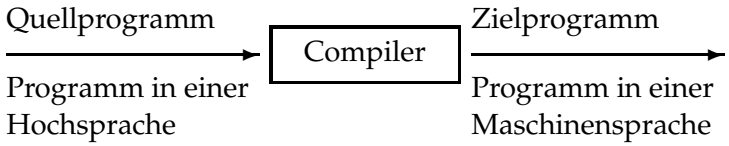


Abbildung 12.2: Compilierung

Besonderheiten der Compilierung können dabei sein:

- vor der Compilierung kann das Quellprogramm in Module zerlegt sein, die sich in verschiedenen Dateien befinden
- ist das Zielprogramm ein Assemblercode, so muß eine weitere Übersetzung in Maschinencode erfolgen
- der Maschinencode muß gegebenenfalls mit Bibliotheksroutinen zusammengebunden werden

Daraus ergeben sich folgende Phasen der Umwandlung:

Die Funktionsweisen des Compilers, Assemblers und Linkers sollen in den folgenden Abschnitten behandelt werden.

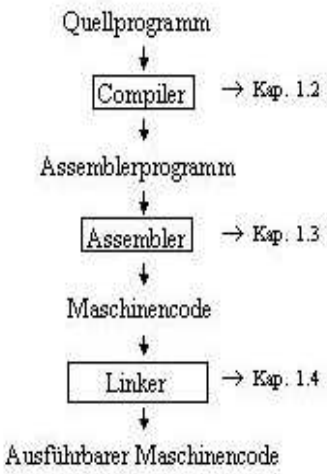


Abbildung 12.3: Compilierungsphasen

12.2 Funktionsweise des Compilers

Das Quellprogramm wird zunächst in seine Bestandteile zerlegt, und es wird eine Zwischendarstellung, ein sogenannter Parse-Baum erzeugt (Analysephase). Danach wird aus dem Parse-Baum das gewünschte Zielprogramm konstruiert (Synthesephase). Diese beiden Phasen wollen wir im folgenden betrachten.

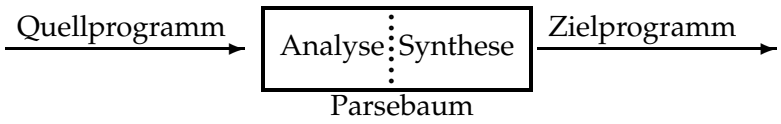


Abbildung 12.5: Grundstruktur eines Compilers

I. Analysephase

Die Analyse des Quellprogramms besteht selbst wieder aus drei Teilen.

1. Lexikalische Analyse wird auch lineare Analyse oder Scanning genannt.
Der Zeichenstrom des Quellprogramms wird von links nach rechts gelesen und in **Symbole (tokens)** aufgeteilt. Ein Symbol ist dabei eine Folge von Zeichen, die zusammen eine bestimmte Bedeutung haben.
Z.B. `dauer := sekunden + minuten * 60` ergibt die Symbole:

−Bezeichner :	″dauer″
−Zuweisungssymbol :	″ := ″
−Bezeichner :	″sekunden″
−Operationszeichen :	″ + ″
−Bezeichner :	″minuten″
−Operationszeichen :	″ ∗ ″
−Zahl :	″60″

Dabei werden die Leerzeichen, eventuelle Kommentare und die die Zeichen dieser einzelnen Symbole voneinander trennen, entfernt.

2. Syntaktische Analyse wird auch hierarchische Analyse oder Parsing genannt.
Dabei werden Symbole hierarchisch zu semantisch sinnvollen Teilbäumen zusammengefaßt. Symbole einer Gruppe haben immer eine bestimmte Bedeutung. So entstehen **grammatikalische Sätze**, die der Compiler später benutzt. Solche Sätze werden durch einen Parse–Baum (vgl. Abbildung 12.6) dargestellt.

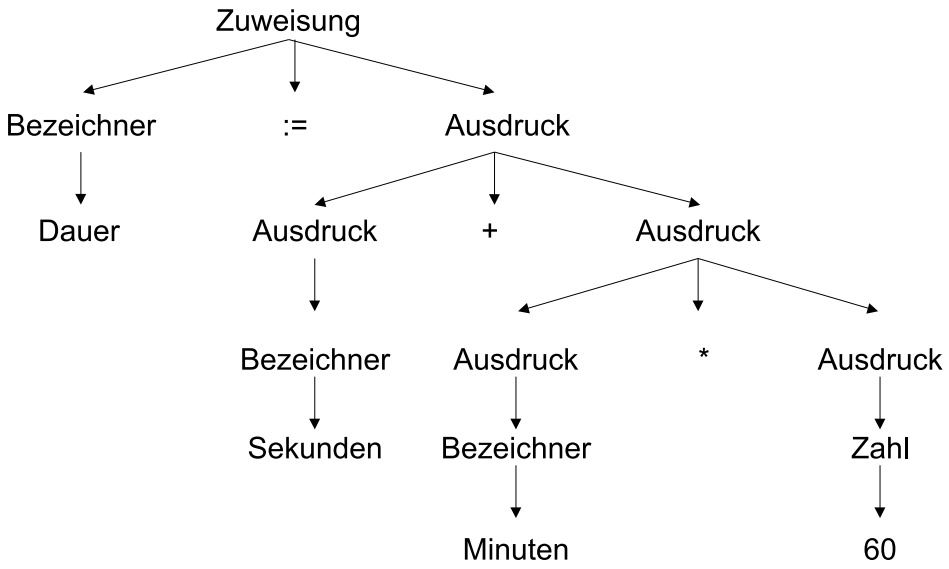


Abbildung 12.6: Parsebaum

In dem grammatikalischen Satz "sekunden + minuten * 60" ist "minuten * 60" eine logische Einheit, "sekunden + minuten" jedoch nicht.

Die hierarchische Struktur wird in der Regel durch rekursive Regeln ausgedrückt, z.B.

- jeder Bezeichner ist ein Ausdruck
- jede Zahl ist ein Ausdruck
- wenn a_1 und a_2 Ausdrücke sind, dann auch $a_1 + a_2,\ a_1 * a_2,\ (a_1)$
- "Bezeichner := Ausdruck" ist eine Anweisung
- "if (Ausdruck) then Anweisung" ist eine Anweisung

3. Semantische Analyse Um sicherzustellen, dass die Bestandteile eines Programms sinnvoll zusammenpassen, werden Prüfungen durchgeführt. Bei Typprüfungen überprüft der Compiler z.B. ob die Operanden eines Operators von der Spezifikation der Quellsprache zugelassen sind, d.h. ob entsprechende Regeln erfüllt sind.
Gegenbeispiel:

```
< Ausdruck > + < Anweisung >
if <Ausdruck> then <Zahl>
```

wären **nicht** zugelassen.

II. Synthesephase

Nach diesen Schritten der Analyse folgt die Synthese, in der aus dem Parse–Baum das gewünschte Zielprogramm konstruiert wird.

4. Zwischencodeerzeugung
Der Parsebaum soll nun in ein Zielprogramm übersetzt werden. Dazu gibt es verschiedene Möglichkeiten. Wir wollen im folgenden den sogenannten **"Drei–Adress–Code"** betrachten, welcher der Assemblersprache für eine Maschine ähnelt. Drei–Adress–Code ist eine Folge von Instruktionen, bei denen jede Instruktion höchstens drei Operanden und neben der Zuweisung ":=" höchstens einen Operator (*, +, ...) hat. Z.B.

- $temp_0 := sekunden$
- $temp_1 := minuten$
- $temp_2 := temp_1 * 60$
- $temp_2 := temp_0 + temp_2$
- $dauer := temp_2$

$temp_0$, $temp_1$ und $temp_2$ sind dabei temporäre Namen für Speicherplätze, auf die der Rechner sehr einfach zugreifen kann (Register – werden später behandelt).

5. Code–Optimierung
Der Zwischencode kann gegebenenfalls verbessert werden, es entsteht effizienter Assemblercode.

Hinsichtlich der Code–Optimierung gibt es bei den Compilern große Unterschiede, auf die hier nicht eingegangen werden soll.

6. Code–Erzeugung
Nun wird der Zielcode erzeugt, der bei uns aus Assemblercode besteht. Wichtig ist dabei, den Variablen jeweils Speicherplätze – sogenannte Register – zuzuordnen. Z.B.:

- lw $t0, sek (lade Wort)

- lw $t1, min
- mult $t2, $t1, 60
- add $t2, $t0, $t2
- sw $t2, dauer (speichere Wort)

Parallel zu diesen 6 Phasen hat der Compiler noch die Aufgaben der Fehlerbehandlung und Symboltabellenverwaltung.

12.3 Funktionsweise des Assemblers

Wir gehen davon aus, dass unser Compiler Assemblercode erzeugt hat. Dieser wird einem Assemblerprogramm zur weiteren Verarbeitung übergeben, so dass Maschinencode entsteht.

Zusammenhang zwischen Assembler– und Maschinencode
Assemblercode ist eine für den Nutzer leichter verständliche Version des Maschinencodes, in der Namen und Zahlen vorkommen. Im Maschinencode haben wir nur noch Nullen und Einsen, die für den Rechner verständlich sind.

12.4 Funktionsweise des Linkers

Laden und Binden sind zwei Funktionen, die im allgemeinen von *einem* Programm ausgeführt werden, dem sogenannten **Linker**. **Laden** bedeutet, die Adressen im Maschinencode ggf. geeignet um eine Stellenanzahl L zu verändern und im Speicher abzulegen.

Das **Binden** ermöglicht es, verschiedene Dateien, die jeweils Maschinencode enthalten, zu einem Programm zusammenzufassen.

13

Einführung in den SPIM Simulator

Ziel der Vorlesung ist das Kennenlernen von Aufbau und Funktion der Rechner. Dabei ist das Herzstück - der Prozessor - von besonderer Bedeutung. Wie in Kapitel 12 gelernt, arbeitet dieser Maschinencode ab, der aus Assemblercode generiert wurde.

Im folgenden wird ein spezieller Prozessor betrachtet, der MIPS R2000. Dieser zeichnet sich durch eine klare Architektur und einen übersichtlichen Befehlssatz aus.

Der MIPS R2000 ist ein RISC-Prozessor. RISC steht für **R**educed **I**nstruction **S**et **C**omputer. Das Gegenstück sind die sogenannten **C**omplex **I**nstruction **S**et **C**omputer (CISC). RISC-Prozessoren haben vergleichsweise wenig Befehle implementiert, die aber sehr einfach sind und daher sehr schnell ausgeführt werden können. Dafür müssen in der Regel mehr RISC-Befehle verwendet werden als bei einem vergleichbaren CISC-Computer.

Zur Info: RISC- und CISC-Architekturen werden zu einem späteren Zeitpunkt in einem separaten Kapitel vertieft. Es sei an dieser Stelle jedoch angemerkt, dass gängige Prozessoren, wie der Intel 8086 und der Pentium CISC-Rechner sind.

1980 bereits entwickelte David A. Patterson an der Berkeley-Universität den Begriff RISC und den RISC-Prozessor RISC I, der für die SPARC-Prozessoren der Firma SUN Pate stand. 1984 wurde das Grundmodell der SPIM von John Hennesy an der Universität Stanford entwickelt. In der Praxis werden die MIPS-Prozessoren u.a. von DEC verwendet.

13.1 Einsatz von Simulatoren

Schon bei kleinen Kindern beobachtet man gerne, dass das Spiel die Form des Lernens ist, die am liebsten angenommen wird und noch dazu Spaß machen kann. Spielen oder eine Situation durchspielen, das führt direkt zur Simulation.

Ist ein System sensibel oder teuer (man denke an das Beispiels eines Flugzeugs), so soll der wissbegierige Nutzer nicht am eigentlichen Objekt dessen Verhalten erforschen und herumexperimentieren, sondern sich zunächst am Modell üben. Hier können ungehemmt alle denkbaren Systemzustände interaktiv erprobt oder auch wieder rückgängig gemacht werden.

So verhält es sich auch mit der Assemblerprogrammierung. Diese ist auf einem Simulator einfacher zu erlernen als direkt auf einem Rechner. Ein Simulator kann bei Programmierfehlern leicht zurückgesetzt werden, bei einem abgestürzten Rechern dauert dies wesentlich länger. Außerdem erlaubt der Simulator eine bessere Einsicht in wichtige Teile des Rechners als dies bei professionellen Assemblern der Fall ist. Ein weiterer Vorteil der Verwendung von Simulatoren liegt in der Möglichkeit der Simulation von fremden Prozessoren. Der in der Vorlesung verwendete SPIM-Simulator kann so z.B. auf allen Unix-Rechnern, bei Windows ab Version 3.11 und auf Macintosh-Rechnern verwendet werden.

13.2 SPIM-Ausführung eines Beispielprogramms

Im folgenden wollen wir das Programmbeispiel aus Abschnitt 12.2 betrachten, das um zwei Anweisungen für die Beendigung des Programms ergänzt wird. Damit erhalten wir:

```
  lw   $t0, sek
2 lw   $t1, min
  mul  $t2, $t1, 60
4 add  $t2, $t0, $t2
  sw   $t2, dauer
6
  li   $v0, 10
8 syscall
```

Dieses Programm wird als Textsegment in den SPIM-Simulator eingegeben.

Die Ausführung des Programms soll für sek= 12 und min= 18 erfolgen. Diese beiden Zahlen rechnen wir zunächst in Binär- und Hexadezimaldarstellung um:

Basis 10	12	18
Basis 2	0000 1100	0001 0010
Basis 16	0c	12

In den Registern R8(t0) und R9(t1) im oberen Teil des Simulators sind die Hexadezimalzahlen als Registerwerte zwischengespeichert. Dabei liegt jedoch keine 8-Bit-Darstellung zugrunde, sondern eine 32-Bit-Darstellung, wodurch sich für 4 Bit je eine Hexadezimalzahl ergibt, d.h. es sind 8 Hexadezimalzahlen bei den Registerwerten sichtbar.

Wir wollen das Ergebnis Hexadezimal per Hand nachrechnen:

Schritt 1 (Zeile 3)

$18_{10} \cdot 60_{10}$ mit $60_{10} = 3 \cdot 16 + 12 = 3c_{16} = 12_{16} \cdot 3c_{16}$

$$\begin{array}{r} 12{\cdot}3c \\ \hline 36 \\ d8 \\ \hline 438 \end{array}$$

Schritt 2 (Zeile 4)

$438_{16} + c_{16}$

$$\begin{array}{r} 438 \\ c \\ \hline 444 \end{array}$$

Dieses hexadezimale Ergebnis 444 ist in Register R10 (t2) nach Ausführung des Programmlaufs sichtbar.

Nun wollen wir uns anschauen, wie ein Assemblerbefehl wie z.B.

```
add  $t2, $t0, $t2
```

in Maschinencode umgewandelt wird. Dazu betrachten wir zunächst einen MIPS-Befehl in seiner allgemeinsten Struktur aus 6 Bestandteilen, siehe Abbildung 13.1.

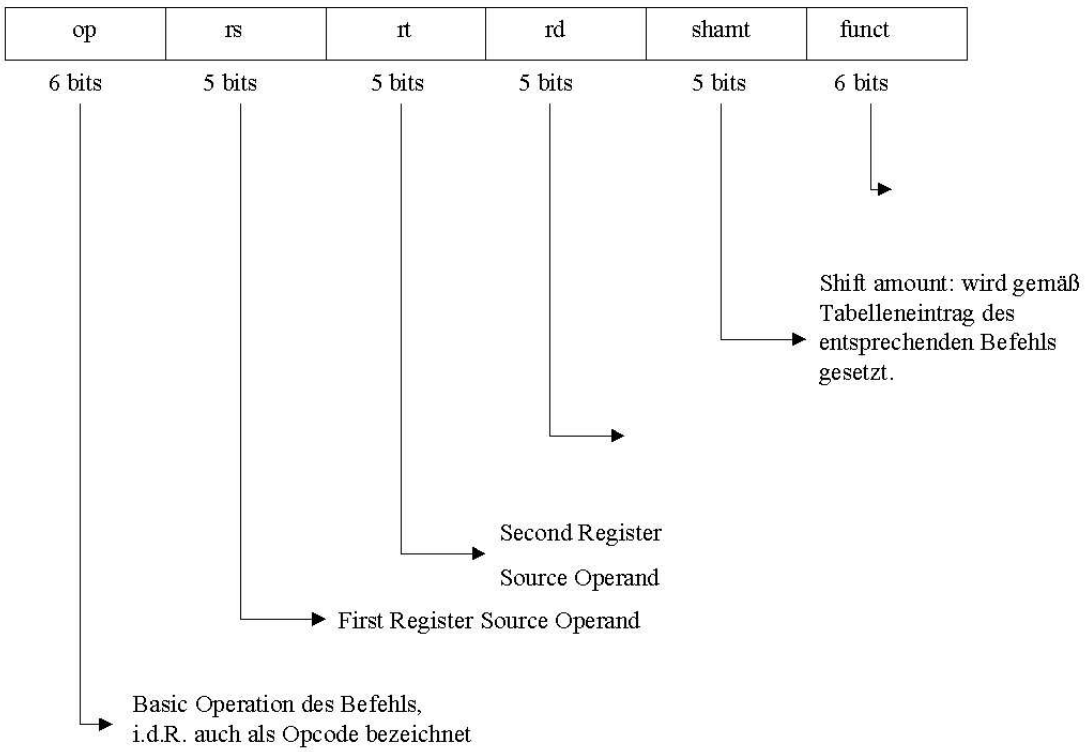


Abbildung 13.1: MIPS–Befehl

Für die einzelnen Befehle bzw. Operationen ist die Belegung der einzelnen Bestandteile aus Tabellen ersichtlich:

Befehl	*op*	*rs*	*rt*	*rd*	*shamt*	*funct*
add	0	*reg*	*reg*	*reg*	0	32
sub	0	*reg*	*reg*	*reg*	0	34

reg steht dabei für den Wert des speziellen Registers, der aus anderen Tabellen ersichtlich ist:

$$\begin{array}{llll} \$t0 \Rightarrow \$8 & \$s0 \Rightarrow \$16 & & \\ \$t1 \Rightarrow \$9 & \$s1 \Rightarrow \$17 & \Rightarrow & add\$t2, \$t0, \$t2 \\ \vdots & \vdots & & \Downarrow \\ \$t7 \Rightarrow \$15 & \$s7 \Rightarrow \$23 & & add\$10, \$8, \$10 \end{array}$$

Damit können wir unseren Addierbefehl codieren: (vgl. Abbildung 13.2)

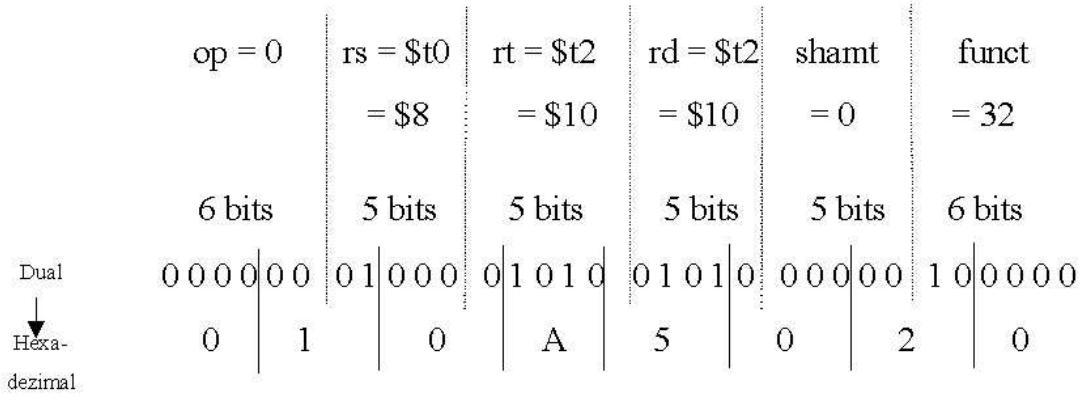


Abbildung 13.2: Codierung

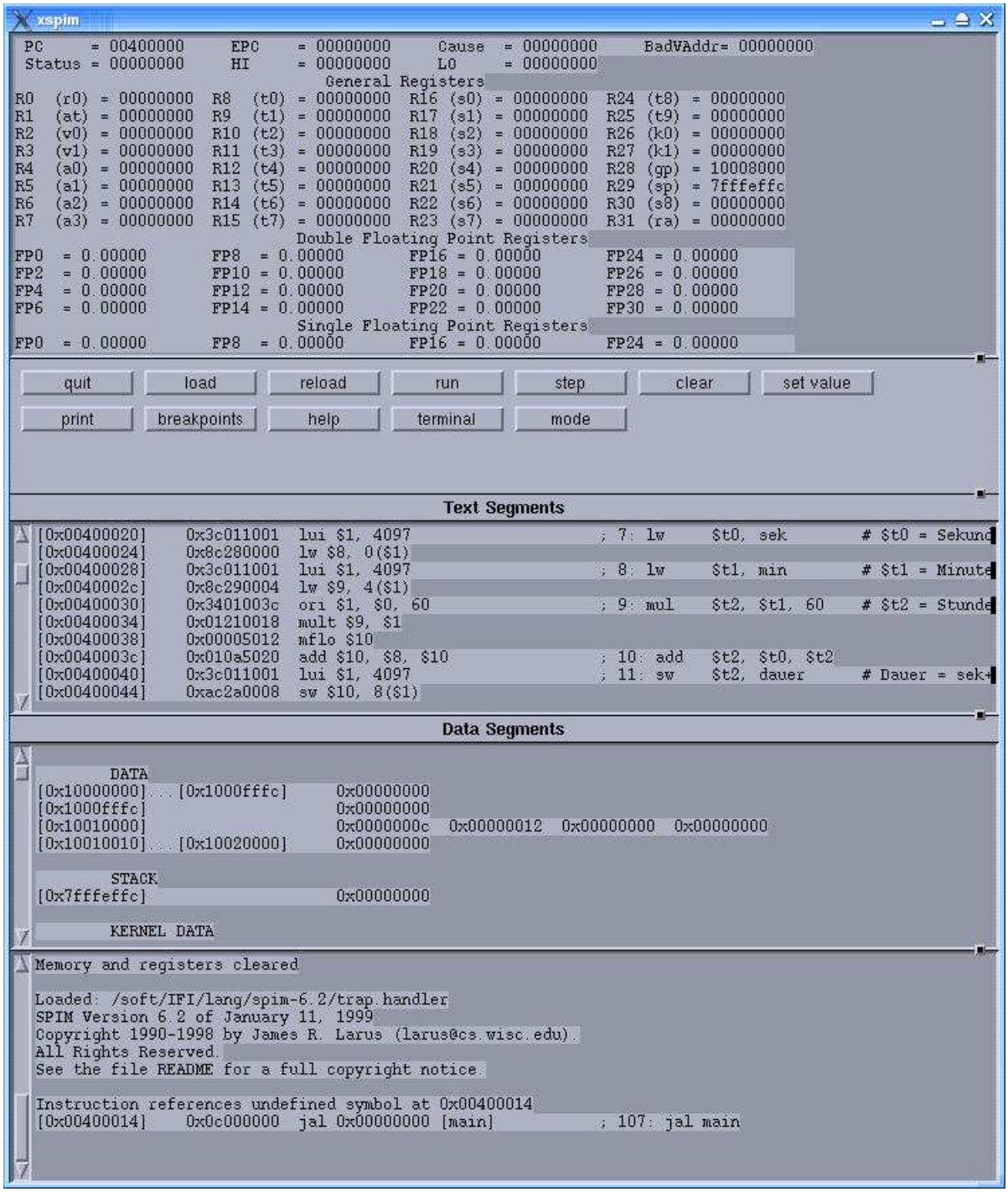
Diese hexadezimale Darstellung entspricht der numerischen Codierung 0x010a5020 (⇒ Maschinencode) (vgl Beispiel).

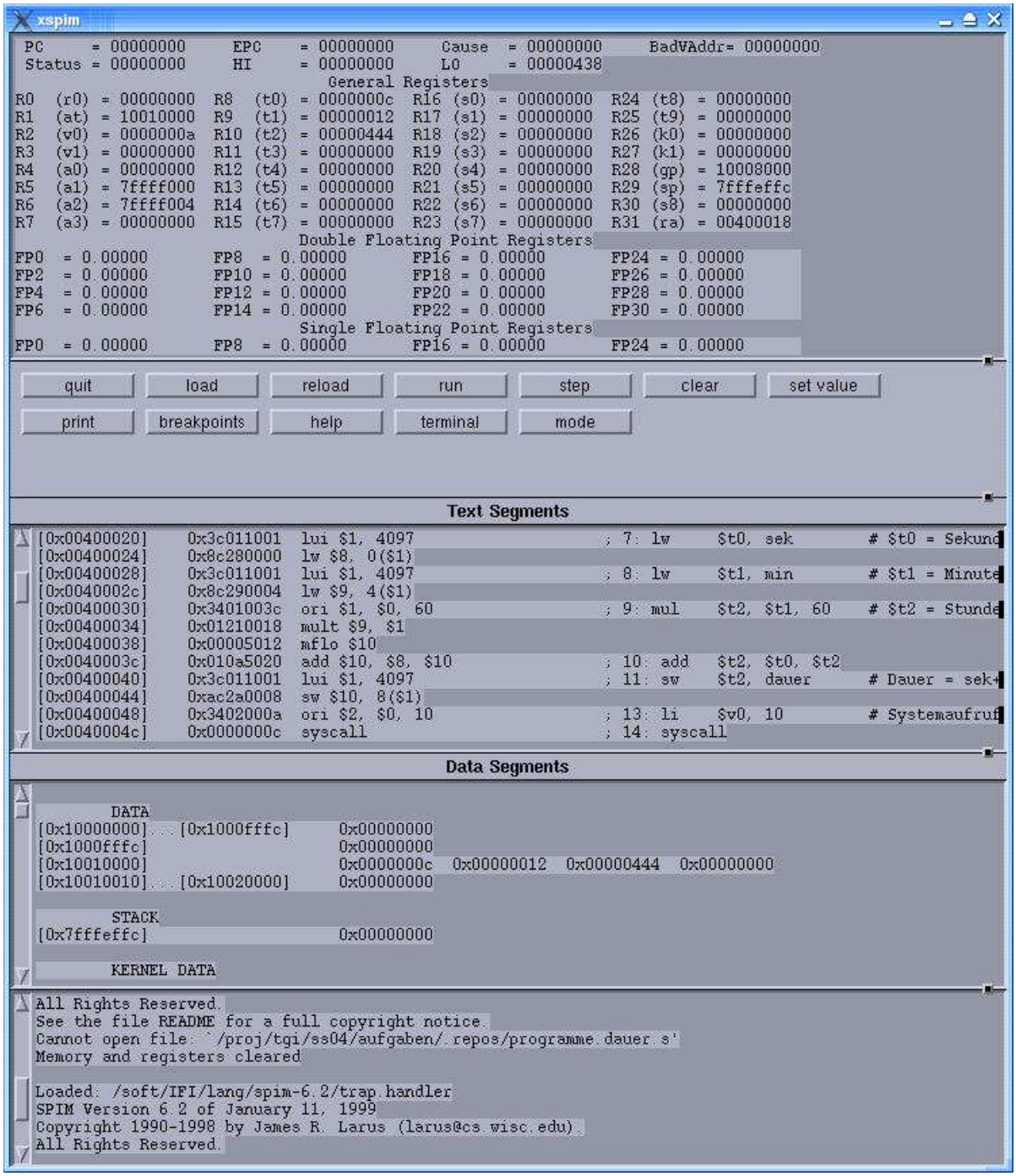
Ein weiteres Problem besteht in der Adressierung. Wie bereits erwähnt, sind die von MIPS verwendeten Befehle 32 Bit lang, d.h. benötigen 4 Byte, d.h. 4 Speicherzellen zum Speichern der Daten.

Angenommen der Speicherbereich des Programms beginnt mit der Adresse 0040.0000 zur Basis 16. Dann hat der nächste Befehl die Anfangsadresse $0040.0000 + 4 = 0040.0004$. Durch weiteres Aufaddieren der Zahl 4 ergibt sich die Folge von Adressen

0040 0000
0040 0004
0040 0008
0040 000*c*
0040 0010
0040 0014
0040 0018
0040 001*c*
0040 0020
0040 0024 u. s. w.

Damit der Prozess weiß, welchen Befehl er als nächstes abarbeiten muß, nutzt er ein spezielles Register, das die Adresse des nächsten abzuarbeitenden Befehls speichert, den sogenannten Befehlszähler (englisch: Programme Counter, PC). Dieser wird nach jeder Befehlsabarbeitung um 4 inkrementiert bzw. bei Sprungbefehlen auf die Sprungadresse gesetzt.


xspim
PC = 00400000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 00000000 HI = 00000000 LO = 00000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000
R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 00000000 R28 (gp) = 10008000
R5 (a1) = 00000000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (sp) = 7fffeffc
R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00000000
Double Floating Point Registers
FP0 = 0.00000 FP8 = 0.00000 FP16 = 0.00000 FP24 = 0.00000
FP2 = 0.00000 FP10 = 0.00000 FP18 = 0.00000 FP26 = 0.00000
FP4 = 0.00000 FP12 = 0.00000 FP20 = 0.00000 FP28 = 0.00000
FP6 = 0.00000 FP14 = 0.00000 FP22 = 0.00000 FP30 = 0.00000
Single Floating Point Registers
FP0 = 0.00000 FP8 = 0.00000 FP16 = 0.00000 FP24 = 0.00000
quit
load
reload
run
step
clear
set value
print
breakpoints
help
terminal
mode
Text Segments
[0x00400020] 0x3c011001 lui $1, 4097 ; 7: lw $t0, sek # $t0 = Sekund
[0x00400024] 0x8c280000 lw $8, 0($1)
[0x00400028] 0x3c011001 lui $1, 4097 ; 8: lw $t1, min # $t1 = Minute
[0x0040002c] 0x8c290004 lw $9, 4($1)
[0x00400030] 0x3401003c ori $1, $0, 60 ; 9: mul $t2, $t1, 60 # $t2 = Stunde
[0x00400034] 0x01210018 mult $9, $1
[0x00400038] 0x00005012 mflo $10
[0x0040003c] 0x010a5020 add $10, $8, $10 ; 10: add $t2, $t0, $t2
[0x00400040] 0x3c011001 lui $1, 4097 ; 11: sw $t2, dauer # Dauer = sek
[0x00400044] 0xac2a0008 sw $10, 8($1)
Data Segments
DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc] 0x00000000
[0x10010000] 0x0000000c 0x00000012 0x00000000 0x00000000
[0x10010010]...[0x10020000] 0x00000000
STACK
[0x7fffeffc] 0x00000000
KERNEL DATA
Memory and registers cleared
Loaded: /soft/IFI/lang/spim-6.2/trap.handler
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Instruction references undefined symbol at 0x00400014
[0x00400014] 0x0c000000 jal 0x00000000 [main] ; 107: jal main


xspim
PC = 00000000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 00000000 HI = 00000000 LO = 00000438
General Registers
R0 (r0) = 00000000 R8 (t0) = 0000000c R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 10010000 R9 (t1) = 00000012 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 0000000a R10 (t2) = 00000444 R18 (s2) = 00000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000
R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 00000000 R28 (gp) = 10008000
R5 (a1) = 7fffff000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (sp) = 7fffeffc
R6 (a2) = 7ffff004 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000
R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00400018
Double Floating Point Registers
FP0 = 0.00000 FP8 = 0.00000 FP16 = 0.00000 FP24 = 0.00000
FP2 = 0.00000 FP10 = 0.00000 FP18 = 0.00000 FP26 = 0.00000
FP4 = 0.00000 FP12 = 0.00000 FP20 = 0.00000 FP28 = 0.00000
FP6 = 0.00000 FP14 = 0.00000 FP22 = 0.00000 FP30 = 0.00000
Single Floating Point Registers
FP0 = 0.00000 FP8 = 0.00000 FP16 = 0.00000 FP24 = 0.00000
quit
load
reload
run
step
clear
set value
print
breakpoints
help
terminal
mode
Text Segments
[0x00400020] 0x3c011001 lui $1, 4097 ; 7: lw $t0, sek # $t0 = Sekund
[0x00400024] 0x8c280000 lw $8, 0($1)
[0x00400028] 0x3c011001 lui $1, 4097 ; 8: lw $t1, min # $t1 = Minute
[0x0040002c] 0x8c290004 lw $9, 4($1)
[0x00400030] 0x3401003c ori $1, $0, 60 ; 9: mul $t2, $t1, 60 # $t2 = Stunde
[0x00400034] 0x01210018 mult $9, $1
[0x00400038] 0x00005012 mflo $10
[0x0040003c] 0x010a5020 add $10, $8, $10 ; 10: add $t2, $t0, $t2
[0x00400040] 0x3c011001 lui $1, 4097 ; 11: sw $t2, dauer # Dauer = sek+
[0x00400044] 0xac2a0008 sw $10, 8($1)
[0x00400048] 0x3402000a ori $2, $0, 10 ; 13: li $v0, 10 # Systemaufruf
[0x0040004c] 0x0000000c syscall ; 14: syscall
Data Segments
DATA
[0x10000000]...[0x1000fffc] 0x00000000
[0x1000fffc] 0x00000000
[0x10010000] 0x0000000c 0x00000012 0x00000444 0x00000000
[0x10010010]...[0x10020000] 0x00000000
STACK
[0x7fffeffc] 0x00000000
KERNEL DATA
All Rights Reserved.
See the file README for a full copyright notice.
Cannot open file: `/proj/tgi/ss04/aufgaben/.repos/programme.dauer.s'
Memory and registers cleared
Loaded: /soft/IFI/lang/spim-6.2/trap.handler
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.

Teil VI

Zusammenspiel der unteren Ebenen eines Computers

14

Struktur von Computern

Wir gehen noch einmal zurück zu Teil V, speziell in den Abschnitt 12.1.2 (Verarbeitung eines Programms im Rechner).

Die primitiven Befehle eines Computers, die dieser verarbeiten kann, nannten wir Maschinensprache – für den Menschen war diese eher schwierig zu benutzen. Eine neue, von Menschen leichter zu benutzende Befehlsmenge hatten wir Programmiersprache genannt, wobei eine Compilierung für die Übersetzung der Programmierbefehle in Maschinenbefehle sorgte. Die Befehlsfolgen zweier solcher Sprachen können dieselbe Semantik haben und denselben Algorithmus implementieren – sie widerspiegeln jedoch unterschiedliche Sichten auf die Verarbeitung eines Problems. Man sagt, sie repräsentieren unterschiedliche **Schichten (layers)** oder **Ebenen (levels)**. Diese Begriffe werden anwendungsspezifisch verwendet, stehen jedoch für den selben Sachverhalt.

14.1 Das Schichtenprinzip der Informatik

Der Chef gibt der Poststelle einer Firma – sagen wir mal in München – einen Brief zum Weiterleiten an eine Kollegin in der Aussenstelle – sagen wir mal in Aachen–, für ihn ist lediglich interessant, dass die Arbeit gemacht wird, nicht aber wie.

Die Poststelle kann nun beispielsweise ein Telefax schicken, sie kann den Brief elektronisch versenden – aber nehmen wir einmal an, dass das Dokument physisch am Ort B ankommen soll und sie entscheidet sich für die Post. Die Post erbringt in diesem Fall einen Beförderungsdienst und ist damit eine untenliegende Schicht.

Also sucht die Postbearbeiterin einen Dienstzugangspunkt der Post – z.B. einen Briefkasten und nimmt damit einen Postdienst in Anspruch. Wie die Post den Brief befördert, ist egal – wichtig ist lediglich, dass der Brief in der Poststelle der Aussenstelle ankommt.

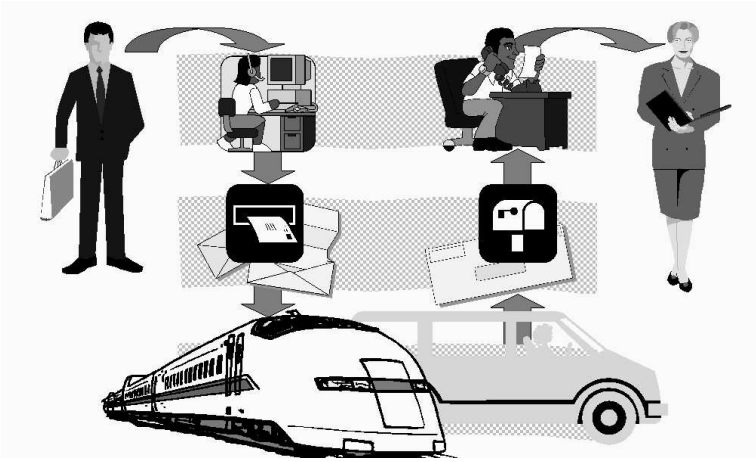


Abbildung 14.1: Das Schichtenprinzip der Informatik

Die Post stempelt den Brief und sortiert, sie muss zur eigentlichen Beförderung aber einen Fahrdienst in Anspruch nehmen, beispielsweise die Deutsche Bahn oder auch einen Paketdienst.

Diese wieder untenliegende Schicht befördert den Brief vom Ort München an den Ort Aachen, dort übergibt die Bahn den Brief der Post, die Post übergibt ihn dem Adressaten in Form der Poststelle der Aussenstelle, und innerhalb der Poststelle wird der Brief sortiert und dem Empfänger übergeben.

Wir sehen, dass wir in diesem Beispiel 2 Anwedungskomponenten haben, die beiden kooperierenden Angestellten, die mittels ihrer Poststelle und den darunterliegenden Schichten einen Brief versenden können.

Jede Schicht nutzt dabei einen Dienst der darunterliegenden Schicht.

Jeder Schicht bleibt die Wirkungsweise der darunterliegenden Schicht verborgen, wie der Dienst erbracht wird, interessiert nicht, nur dass er in einer spezifierten Form erbracht wird.

Und – um einen Dienst der darunterliegenden Schicht zu nutzen, sind sogenannte Dienstzugangspunkte (Service Access Points) notwendig, an denen der unterliegenden Schicht die notwendigen Mittel (Brief) übergeben werden kann bzw. abgeholt werden kann.

14.2 Mehrschichtige Computer

Die meisten Computer bestehen aus zwei oder mehr **Ebenen**. Dabei sind – je nach Modell – bis zu sechs oder sieben Ebenen üblich. Computer werden in mehreren Ebenen betrachtet, die jeweils auf der vorhergehenden, niedrigeren Ebene aufbauen. Jede Ebene stellt dabei eine bestimmte Abstraktion mit unterschiedlichen Objekten und Operationen dar. Durch diese Art der Entwicklung und Analyse von Computern sind wir in der Lage, irrelevante Details zu ignorieren, und somit ein komplexes Thema beherrschbar zu machen.

Anwendungsprogrammierer, die mit einer Maschine arbeiten, die aus n Ebenen besteht, interessieren sich immer nur für die oberste Ebene. Die darunterliegenden Ebenen werden ggf. für Maschinenprogrammierer oder Systemprogrammierer (vgl. Info III) interessant. Noch tiefere Ebenen interessieren Systemarchitekten oder Designer eines Computers.

Wir wollen die Ebenen einzeln betrachten und dabei bei der einfachsten, unteren Ebene beginnen.

Ebene 0: Die Ebene 0 ist die **Hardware** der Maschine oder die **Ebene der digitalen Logik (Digital Logic Level)**.
Der Vollständigkeit halber sei erwähnt, dass noch darunter die sogenannte **Gerätebene (Device Level)** liegt. Der Computer–Designer beschäftigt sich dann mit einzelnen Transistoren. Dies wird jedoch nicht zum Bereich der Informatik gezählt, sondern ist Hoheitsgebiet der Elektrotechniker. Und was sich im Inneren der Transistoren abspielt, ist wiederum Gegenstand der Festkörperphysik.

Für uns ist jedoch die Ebene der digitalen Logik die unterste. Auf ihr betrachten wir Gatter (Gates), die zwar aus analogen Komponenten aufgebaut sein können, z.B. aus Transistoren, die aber aus einem oder mehreren digitalen Eingangssignalen digitale Ausgangssignale berechnen. Aus einer kleinen Anzahl von Gates konnten wir einen 1–Bit–Speicher konstruieren, der eine 0 oder 1 speichern konnte, mehrere 1–Bit–Speicher wurden zu Registern gruppiert. Jedes Register kann dem entsprechend eine maximale Anzahl Binärzahlen aufnehmen.

Ebene 1: Die Ebene 1 ist die **Mikroarchitekturebene (Microarchitecture Level)**. Auf dieser Ebene befinden sich in der Regel 8 bis 32 Register, die einen lokalen Speicher und ein Schaltnetz besitzt, die man **Arithmetisch Logische Einheit (Arithmetic Logic Unit, ALU)** nennt.
Die ALU kann einfache arithmetische Operationen durchführen. Mindestens ein oder zwei Register sind direkt mit der ALU verbunden, über diese Verbindung (**Datenweg** = **Data Path**) fließen Daten, so dass die ALU z.B. eine Addition der beiden in den Registern gespeicherten Werte vornehmen kann und das Ergebnis in einem dieser Register abspeichert. Dieser Vorgang – z.B. die Addition – muß nun gesteuert werden. Dies wird entweder von einem **Mikroprogramm** durchgeführt oder heute vielfach auch direkt in Hardware realisiert.

Erfolgt die Steuerung durch Software, so ist das Mikroprogramm ein Interpreter für Anweisungen der Ebene 2. Nacheinander liest es Instruktionen ein, prüft sie und führt sie aus, wobei es dafür den Datenweg verwendet.

Beispiel: Eine ADD–Instruktion wird eingelesen. Ihre Operanden werden gesucht und in die Register gebracht. Dann wird die Summe durch die ALU berechnet und schließlich das Ergebnis zurückgeschrieben.

Ebene 2: Die Ebene 2 ist die **ISA–Ebene**, wobei ISA für **Instruction Set Architecture** steht. Jeder Prozessorhersteller veröffentlicht ein Handbuch für die von ihm verkauften Prozessoren mit dem Titel "Leitfaden der Maschinensprache" oder "Betriebsprinzipien des Prozessors 0815" o.ä. Diese Handbücher behandeln genau die ISA–Ebene. Genauer: die darin enthaltenen Beschreibungen der Instruktionsmenge des Prozessors betreffen die Instruktionen, die vom Mikroprogramm interpretiert werden.

Bemerkung: stellt ein Hersteller für seinen Prozessor zwei Interpreter zur Verfügung, um zwei verschiedene ISA–Ebenen zu interpretieren, so müßte er für beide ein eigenes Handbuch bereitstellen.

Ebene 3: Die Ebene 3 wird auch **Operating System Machine Level** oder **Ebene der Betriebssystemmaschine** genannt. Die auf dieser Ebene neu hinzukommenden Merkmale werden von einem Interpreter, d.h. dem Betriebssystem ausgeführt. Einige Anweisungen können auch vom Mikroprogramm auf Ebene 2 ausgeführt werden – deshalb ist diese Ebene i.d.R. hybrid (d.h. eine Instruktion einer Ebene kann auch auf anderen Ebenen vorhanden sein). Auf Ebene 3 gibt es jedoch eine Reihe neuer Instruktionen ggf. eine andere Speicherorganisation und ggf. die Möglichkeit zwei oder mehr Programme gleichzeitig auszuführen. Programme auf dieser Ebene werden von Systemprogrammierern geschrieben. Die Funktionsweise des Betriebssystems ist Gegenstand der Vorlesung Info III.

Da die weiteren Ebenen im kommenden Verlauf dieser Vorlesung weniger von Bedeutung sind bzw. bereits betrachtet wurden, wollen wir sie nur kurz vorstellen.

Zwischen Ebene 3 und 4 besteht ein fundamentaler Bruch:

- die unteren Ebenen 1 – 3 sind nicht für die Modifikation durch den Anwendungsprogrammierer gedacht,
- die unteren Ebenen 1 – 3 werden immer interpretiert, die höheren können durch Compiler unterstützt werden und
- die Sprachen der unteren Ebenen 1 – 3 sind numerisch, d.h. bestehen aus langen Zahlenreihen wohingegen Sprachen der Ebene 4 und 5 für den Menschen leichter verständlich sind, d.h. aus Wörtern und Abkürzungen bestehen.

Ebene 4: Dies ist die **Ebene der Assemblersprache**. Das Programm, das die Übersetzung ausführt, ist ein Assembler.

Ebene 5: Diese Ebene befaßt sich mit einer höheren **Programmiersprache** (High–Level Language), z.B. C, C++, Java,..., die vom Anwendungsprogrammierer zwecks Lösung eines spezifischen Problems ausgewählt wurden. I.A. werden die Programme von einem Compiler (Übersetzer) in Code der Ebene 4 und 3 übersetzt. Gelegentlich werden sie auch interpretiert, wie z.B. Java–Programme.

Die auf jeder Ebene verfügbaren Datentypen, Operationen und Merkmale nennt man Architektur. D.h. die Architektur betrifft die Aspekte, die für den Benutzer der jeweiligen Ebene sichtbar sind – z.B. verfügbare Speichermenge, nicht z.B. Chiptechnik.

Es gibt einen Bereich der Informatik, der sich damit beschäftigt, wie die für den Programmierer sichtbaren Teile eines Computers ausgelegt werden sollen. Dieser Bereich heißt **Computerarchitektur** (oder synonym: **Rechnerarchitektur**, **Rechnerorganisation**, **Computerorganisation**).

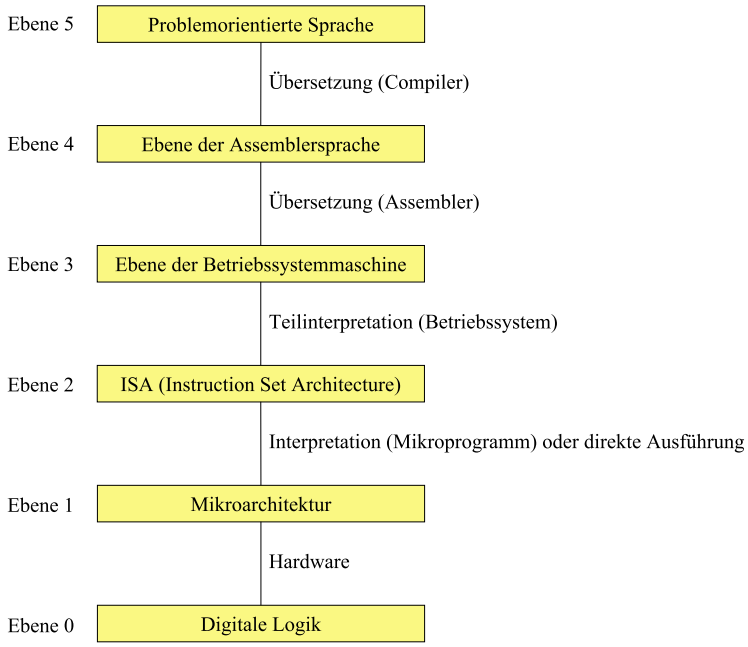


Abbildung 14.2: Die Struktur eines Computers mit 6 Ebenen

14.3 Entwicklung mehrschichtiger Computer

Ebenen 0 und 2: Die ersten digitalen Computer in den vierziger Jahren hatten nur 2 Ebenen: die **ISA–Ebene**, auf der die gesamte Programmierung erfolgte, und die **digitale logische Ebene**, die diese Programme ausführte. Dies war kompliziert, kaum zu verstehen, noch schwieriger zu bauen und unzuverlässig.

und Ebene 1: 1951 : Maurice Wilkes (Uni Cambridge) kam auf die Idee, einen Computer mit 3 Ebenen zu entwickeln, um die Hardware zu vereinfachen. Dazu wurde ein **Mikroprogramm** als eingebauter Interpreter entwickelt. 1970 hatte sich dieses 3–schichtige Konzept schließlich durchgesetzt.

und Ebene 3: Etwa 1960 wurde der Vorgang, dass jeder Programmierer den Computer selbst bedienen mußte automatisiert. Daraus entstand ein Programm namens **Betriebssystem**. Der Programmierer lieferte Steuerinformationen (anfangs in Form von Steuerkarten) zusammen mit dem Programm. Nachfolgend konnte eine CPU von vielen Terminals aus in **Time–Sharing–Betrieb** bedient werden.

und Ebenen 4 und 5: Mit der Entwicklung von höheren Programmiersprachen entwickelten sich die Ebenen 4 und 5.

14.4 Die Instruction Set Architecture

Wir wissen bereits, dass

- die ISA–Ebene zwischen der Mikroarchitektur– und der Betriebssystemebene liegt.
- historisch gesehen die ISA–Ebene vor den anderen beiden genannten Ebenen entwickelt wurde, d.h. ursprünglich war sie die einzige Ebene, die zusätzlich zur digitalen logischen Ebene existierte.

Diese Ebene wird heute häufig auch als Architektur einer Maschine bezeichnet. Ihre Bedeutung resultiert daraus, dass sie für den Systemarchitekten die Schnittstelle zwischen Soft– und Hardware ist.

Diese Anforderungen werden durch die **ISA–Ebene** gelöst. Programme werden – bei dem von nahezu allen Systemdesignern übernommenen Konzept der ISA – von verschiedenen Hochsprachen compiliert. Unabhängig davon wird **Hardware** entwickelt, die Programme der ISA–Ebene direkt ausführen kann. → Die ISA–Ebene definiert so die Schnittstelle zwischen den **Compilern** und der Hardware. Diese Schnittstelle beinhaltet eine Sprache, die beide gemeinsam verstehen müssen.

Das Prinzip ist aus Abbildung 14.3 ersichtlich.

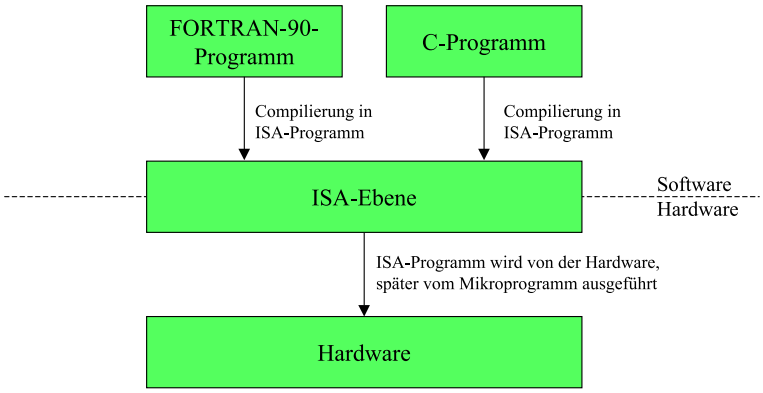


Abbildung 14.3: Software–Hardware Ebene

Die **Theorie** sieht wie folgt aus. Bei der Entwicklung einer neuen Maschine sprechen die Architekten idealerweise mit

- den Compilerprogrammierern
  Diese wünschen sich i.d.R. Merkmale, z.B. eine Instruktion mit Sprung, die von den Architekten – sofern möglich – kostengünstig implementiert werden müssen.
- den Hardwarespezialisten
  Diese wünschen sich ebenfalls Merkmale, die seitens der Softwarespezialisten codiert werden müssen, z.B. Speicher, der im Falle von Adressen mit bestimmten Eigenschaften (z.B. Primzahlen) schnell arbeitet.

In der **Praxis** ist es jedoch so, dass der Nutzer einer neuen Maschine zuerst danach fragen wird, ob diese mit dem Vorgängermodell **kompatibel** ist, ob sie das alte Betriebssystem ausführen kann und ob sie vorhandene Anwendungsprogramme ausführen kann. Dies sollte

der Fall sein und diese Überlegungen ergeben eine ganz neue Sicht bezüglich der Anforderungen für eine ISA–Entwicklung. Ziel ist es, eine ISA–Ebene bei neuen Computerarchitekturen entweder beizubehalten oder **abwärtskompatibel** zu machen, d.h. eine neue Architektur für eine neue und bessere Maschine zu konstruieren, die alte Programme ohne Änderungen ausführen kann.

Die Motivation zur Schaffung besserer ISAs ergibt sich in jedem Fall aus der Sicht, dass die ISA die Schnittstelle zwischen Hard– und Software ist. Für den **Hardware**–Designer soll sie sich effizient implementieren lassen, für den **Software**–Designer soll sich entsprechender Code dafür leicht erzeugen lassen. Die ISA soll eine **maximale Leistung** bei **minimalen Kosten** aufweisen.

14.4.1 Bestandteile der ISA–Ebene

Was ist die ISA–Ebene eigentlich? Was beinhaltet sie?

Die ISA–Ebene definiert, wie sich die Maschine dem Maschinenprogrammierer darstellt. Um der Hardware eines Computers etwas zu befehlen, muß man dessen Sprache sprechen. Die Wörter der Maschinensprache werden in diesem Sinne Befehle oder Instructions genannt, das Vokabular heißt Befehlsmenge oder Instruction Set.

Aus der Sicht, dass heute kein Programmierer mehr in Maschinencode programmiert, können wir anwendungsspezifischer auch sagen: Der Code auf ISA–Ebene ist das, was ein Compiler ausgibt.

Genauer heißt das: um Code für die ISA–Ebene zu produzieren, muß der Compilerprogrammierer wissen,

- um welches **Speichermodell** es sich handelt,
- welche **Register** vorhanden sind,
- welche Datentypen verfügbar sind,
- welche Instruktionen verfügbar sind.

Diese Informationen definieren zusammen die ISA–Ebene.

Bemerkungen:

1. Fragen, wie z.B., ob die **Mikroarchitekturebene** mittels Software oder Hardware realisiert ist, spielen auf der ISA–Ebene keine Rolle, da sie für den Compilerprogrammierer nicht sichtbar sind. Eine Einschränkung dieser Sichtbarkeit wäre lediglich dahingehend zu machen, dass Eigenschaften der untenliegenden Ebenen, wie z.B. Schnelligkeit, sich direkt auf oberer Ebene auswirken.

2. Bei einigen Architekturen wird die **ISA–Ebene** in einem formellen Dokument spezifiziert, das zum Teil von einem Industriekonsortium erarbeitet wird.

Beispiel 14.1*:* Für Sun SPARC, Version 9 und Java Virtual Machine (JVM) wurden 1994 bzw. 1997 solche offiziellen Definitionen erarbeitet. Im Fall der SPARC V9 soll es mehreren Chipanbietern ermöglicht werden, funktionell identische SPARC–Chips herzustellen, die sich nur in Leistung und Preis unterscheiden. Das Dokument spezifiert dabei, um welches Speichermodell es sich handelt, welche Register vorhanden sind, was die Instruktionen bewirken u.s.w. Es behandelt jedoch nicht, wie die Mikroarchitektur aussieht, d.h. eine entsprechende Implementierung zu erfolgen hat.

Es gibt jedoch keine öffentliche Definition für die ISA–Ebene des Pentium–II–Chips, weil Intel es anderen Herstellern nicht leicht machen will, Pentium–II–Chips herzustellen.

3. In der Vorlesung Systemprogrammierung (INFO III) wird behandelt, dass die meisten Computer zwei **Arbeitsmodi** aufweisen – den **Nutzermodus** zur Ausführung von Anwendungsprogrammen und den **Systemmodus**, in dem sogenannte privilegierte Befehle abgearbeitet werden und das Betriebssystem ausgeführt wird. Beide Modi müssen von der ISA unterstützt werden. Wir wollen uns jedoch auf den Nutzermodus beschränken.

14.4.2 Das Registermodell

Alle Computer haben einige auf der ISA–Ebene sichtbare Register. Diese Register auf ISA–Ebene werden grob in zwei Kategorien eingeteilt:

- *Register für spezielle Zwecke:*
  Dazu gehört der ***Programm Counter (PC)*** und der ***Stack Pointer (SP)***, sowie weitere Register mit speziellen Funktionen.
- *Register für allgemeine Zwecke:*
  Diese nehmen lokale Variablen und Zwischenergebnisse von Berechnungen auf. Ihre Hauptfunktion ist die Bereitstellung eines schnellen Zugriffs auf häufig benutzte Daten und damit letztendlich eine Reduzierung von Speicherzugriffen. Moderne Maschinen mit schnellen CPUs und relativ langsamen Speichern haben normalerweise mindestens 32 Allzweckregister, der Trend geht noch zu größeren Anzahlen.

Zusätzlich zu den Registern auf **ISA–Ebene**, die für Anwendungsprogramme sichtbar sind, gibt es immer eine gewisse Anzahl von speziellen Registern, die nur im **Systemmodus** verfügbar sind. Sie werden nur vom Betriebssystem benutzt, so dass der Compiler und letztendlich der Nutzer nichts darüber wissen müssen. In INFO III wird hier mehr ins Detail gegangen.

14.4.3 ISA–Instruktionen

Das Hauptmerkmal der **ISA–Ebene** sind ihre Maschineninstruktionen. Sie steuern die Aktionen der Maschine.

Wir wollen 4 verschiedene architekturelle Stile der Instruction Sets betrachten, die sich dadurch unterscheiden, wo die Operanden einer Instruktion herkommen.

Akkumulator Die ersten Computer hatten nur ein Register, das als **Akkumulator (Accumulator)** bezeichnet wurde. Die ADD–Instruktionen addierte z.B. immer ein Speicherwort zum Akkumulator, so dass nur ein Operand – der im Speicher – spezifiziert werden mußte. Bei einfachen Berechnungen funktionierte diese Technik gut. Bei mehreren Zwischenergebnissen mußte der Akkumulator diese in den Speicher zurückschreiben und später wiederholen.

Memory–memory Wir wollen nochmals eine ADD–Instruktion betrachten, für die drei Operanden spezifiziert werden müssen – zwei Quellen und ein Ziel. Bei diesem Stil sind alle drei Operanden jeder Instruktion im Speicher enthalten. D.h. es wird aus dem Speicher gelesen und das Ergebnis wird wieder in den Speicher geschrieben.

Stack Alle Operationen werden an der Spitze eines Kellerspeichers abgearbeitet. Dabei arbeiten push und pop als Speicherzugriffsoperationen, alle anderen Instruktionen arbeiten nur auf dem Kellerspeicher.

Load–Store Alle Operationen werden auf Registern abgearbeitet. Dieser Stil wird uns im folgenden beim MIPS–Prozessor beschäftigen.

- Zum Bewegen von Daten zwischen Speicher und Registern gibt es LOAD– und STORE–Instruktionen.
  Im MIPS–Prozessor wären zugehörige Instruktionen z.B.
  lw $\$s1, 100(\$s2)$ für load word, d.h. einen Datentransfer vom Speicher zum Register und
  sw $\$s1, 100(\$s2)$ für store word, d.h. einen Datentransfer vom Register zum Speicher.
- Zum Kopieren von Daten zwischen Registern gibt es MOVE–Instruktionen.
  Im MIPS–Prozessor z.B.
  mov.s fd, fs für ein Kopieren eines single floatingpoint Wertes vom Register fs zum Register fd
- Ferner sind arithmetische Instruktionen vorhanden, boolesche Instruktionen zum Vergleichen von Datenelementen und für Sprünge. Solche Instruktionen sind für den MIPS–Prozessor bereits aus zahlreichen Übungen bekannt. Zu ihnen gehören z.B.
  add $\$t0, \$a0, \$a1$ für die Addition der Inhalte von Register a0 und a1 und die Ablegung der Summe in Register t0,
  sub $\$s0, \$t0, \$t1$ für eine entsprechende Subtraktion von Registerinhalten,
  beq $\$s1, \$s2, L$ für einen Vergleich der Registerinhalte von s1 und s2 und bei Gleichheit einen Sprung gemäß GOTO L,
  slt $\$s1, \$s2, \$s3$ für if($s2 < $s1) then $s1 := 1 else $s1 := 0.

Bezüglich der Instruktionen sind die ISA–Ebenen einzelner Rechnerarchitekturen sehr unterschiedlich.

14.5 CISC– versus RISC–Architekturen

Wir wollen im folgenden zwei grundlegende **Rechnerarchitekturen** in ihrer historischen Entwicklung und ihren Charakteristiken betrachten.

14.5.1 Migration von CISC zu RISC

Ende der 70er Jahre waren höhere Programmiersprachen entwickelt worden. Man sagt: es bestand eine semantische Lücke, die geschlossen werden mußte. Die Folge war, dass mit sehr komplexen Instruktionen experimentiert wurde. Diese komplexen Instruktionen wurden auf ISA–Ebene in Mikrooperationen zerlegt und sequentiell interpretiert.

Ein gegenläufiger Trend basierte auf Ideen von Seymour Cray, die in einer FBM–Forschergruppe in einen Hochleistungsrechner eingebracht werden sollten. Anfang der 80er Jahre entstanden daraus Veröffentlichungen zu einem experimentellen Minicomputer unter der Bezeichnung 801, der jedoch nie auf den Markt kam.

1980 begann eine Gruppe an der Berkley–Universität unter der Leitung von David Patterson und Carlo Sequin mit der Entwicklung von VLSI–Chips, die ohne Interpretation auskamen, d.h. direkt in Hardware–Schaltungen implementiert wurden. Sie prägten den Ausdruck **Reduced Instruction Set Computer (RISC)** für diese Konzept und nannten ihn CPU–Chip RISC I, dem ein RISC II folgte.

Ein Jahr später entwickelte John Hennesey an der Stanford–Universität einen etwas anderen Chip mit der Bezeichnung MIPS. Dieser Chip wurde dann Bestandteil kommerzieller Produkte, insbesondere der SPARC.

Wichtig: Die Entwickler wählten völlig neue Befehlssätze. Aus diesem Grund waren die neuen Prozessoren *nicht* **abwärtskompatibel** zu bestehenden Architekturen. Ziel war eine maximale Systemleistung durch Instruktionen, die sich schnell ausführen ließen. Das Ergebnis waren einfache Prozessoren mit relativ wenigen Instruktionen, in der Regel etwa 50. Damit unterschieden sich RISC–Rechner grundlegend von den damals etablierten **Complex Instruction Set Computern (CISC)**, wie z.B. der DEC VAX, Intel und den IBM–Großrechnern, die 200 bis 300 Instruktionen ausführen konnten.

Die Bezeichnungen RISC und CISC sind bis heute erhalten geblieben, obwohl heutzutage die Größe des **Befehlsvorrats** keine große Rolle mehr spielt.

Vorteil der RISC–Architekturen ist, dass die Instruktionen so einfach sind, dass sie in einem einzigen Taktzyklus abgearbeitet werden können. Dabei werden meistens zwei Register aufgerufen, die Registerinhalte kombiniert (Addition, AND,...) und das Ergebnis wieder in ein Register geschrieben. Eine RISC–Maschine braucht folglich i.d.R. mehr einfache Instruktionen (z.B. 4 bis 5) um das gleiche zu tun, was eine CISC–Maschine mit einer einzigen ggf. jedoch viel komplexeren Instruktion erledigt. Dass die RISC–Instruktionen jedoch direkt durch Hardware–Schaltungen realisiert werden – man sagt auch interpretiert werden – ist eine RISC–Instruktion etwa 10mal schneller als eine CISC–Instruktion.
Fazit: RISC ist offensichtlich schneller.

Das interpretieren verlor erheblich an Attraktivität, und es sprach immer mehr für RISC–Maschinen. Man könnte nun glauben, dass RISC–Maschinen wie etwa die DEC Alpha angesichts der Leistungsvorteile die CISC–Maschinen wie den Intel Pentium aus dem Markt verdrängt haben müßten.

Warum ist dies jedoch nicht geschehen?

- Die fehlende Abwärtskompatibilität und die in Intel–Software bereits investierten Milliarden hielten viele Unternehmen von einem Wechsel der Hardware ab.

- Und: Intel war überraschender Weise in der Lage zwar an der CISC–Architektur festzuhalten, ab dem 486er jedoch einschlägige Ideen der RISC–Architektur zu übernehmen. Intel–CPUs enthalten heute einen RISC–Kern, der die einfachsten und i.d.R. häufigsten Instruktionen in einem Zyklus ausführt, während die komplexeren Instruktionen in der üblichen CISC–Weise interpretiert werden. Man spricht von einem Hybridansatz. Damit sind häufige Instruktionen also schnell und weniger häufige Instruktionen langsam. Dieser Ansatz ist zwar weniger schnell als ein reines RISC–Design, liefert aber immer noch eine konkurrenzfähige Gesamtperformance, wobei alte Software ohne Modifikation weiter benutzt werden kann.

14.5.2 RISC–Designprinzipien

Externe Randbedingungen wie neue Technologien, die z.B. bestimmte Abläufe verkürzen, oder die bereits erwähnte Abwärtskompatibilität zu einer bestehenden Architektur, verlangen in der Regel Kompromisse bei der Entwicklung von Computerarchitekturen. Dennoch gibt es einen Bestand an Designprinzipien, um deren Einhaltung sich die Designer von CPUs bemühen. Diese werden heute als RISC–Designprinzipien bezeichnet, und einige dieser Prinzipien wollen wir im folgenden betrachten.

- **Instruktionen werden direkt von der Hardware ausgeführt**
  Alle gängigen Instruktionen werden direkt von Hardware ausgeführt, d.h. nicht durch Mikroinstruktionen interpretiert. Der Wegfall dieser Zwischenebene sorgt meist für eine hohe Geschwindigkeit.

  Bei CISC–Architekturen müssen komplexe Instruktionen in kleinere Anweisungen aufgebrochen werden, die dann als Abfolge von Mikroinstruktionen ausgeführt werden. Dieser zusätzliche Schritt verlangsamt die Maschine, was man bei weniger häufig vorkommenden Befehlen aber in Kauf nehmen kann.

- **Instruktionen werden mit maximaler Rate ausgegeben**
  Die Einheit mit der die Rate des Ausgebens von Instruktionen gemessen wird, ist MIPS, was für Millionen Instruktionen pro Sekunde steht und gleichzeitig das Akronym für den MIPS–Prozessor ist. Wenn man z.B. 500 Millionen Instruktionen pro Sekunde starten kann, so hat man einen 500–MIPS–Prozessor, egal, wie lange die Ausführung dieser Instruktionen dauert. Realisiert werden solche Technologien durch Parallelität (**Pipelining**), d.h. man kann nur dann eine große Anzahl langsamer Instruktionen in kurzer Zeit starten, wenn man mehrere davon gleichzeitig ausführen kann. Um Probleme beim Zugriff auf Register zu vermeiden, ist eine Synchronisation von Abläufen erforderlich.

- **Befehle müssen leicht dekodierbar sein**
  Dazu gehört, Instruktionen gemäß bestehender Regeln auszulegen und ihnen eine feste Länge mit möglichst wenigen Feldern zu geben. Je weniger verschiedene Formate es für Instruktionen gibt, desto besser.

- **Der Speicher ist nur bei Load– und Store–Vorgängen involviert**
  Da Speicherzugriffe viel Zeit beanspruchen können und die Verzögerungen unvorhersehbar sind, sollen die Operanden für die meisten Instruktionen aus den Registern

kommen und dort wieder abgelegt werden. Operanden aus dem Speicher in Register zu verschieben ist ein Vorgang, der in separaten Instruktionen erfolgen kann. Solche Speicherzugriffsinstruktionen eignen sich – falls möglich – besonders gut für das Überlappen mit anderen Instruktionen, die von diesen Werten unabhängig sind.

- **Stelle ausreichend Register bereit**
  Diese Anforderung ergibt sich aus der vorangegangenen: Damit ein einmal abgerufenes Wort in einem Register gehalten werden kann, bis es nicht mehr benötigt wird, sollten viel, meist mindestens 32 Register bereitgestellt werden.

14.5.3 Praktische Beispiele für Computerarchitekturen

Wir wollen nun ein paar spezifische Eigenschaften der ISA von bestimmten Prozessoren betrachten. Für detaillierte Ausführungen siehe Tanenbaum, Computerarchitektur, 4. Auflage, Kapitel 5.1.4 – 5.1.7.

- **ISA–Ebene des Intel Pentium II**
  Der 80386 war die erste 32–Bit–Maschine der Intel–Familie. Die Nachfolger (80486, Pentium, Pentium Pro) bis zum Pentium Pro waren CISC, ab dem Pentium II RISC–Maschinen. Damit der Pentium II abwärtskompatibel ist, verhält er sich nach außen wie ein CISC–Rechner.

- **ISA–Ebene der Ultra SPARC II**
  die Ultra SPARC II wurde erstmals 1987 von Sun Microsystems vorgestellt. Während die ursprüngliche SPARC eine 32–Bit–Architektur war, ist die Ultra SPARC II eine 64–Bit–Maschine. Der adressierbare Speicher ist ein lineares Array von 2^{64} Byte. Dieser Speicher ist jedoch derart groß ($> 18 \cdot 10^{18}$ Byte), so dass ihn keine heute erhältliche Maschine implementieren kann. Zur Zeit werden maximal 2^{44} Byte implementiert, was sich bei künftigen Modellen erhöhen wird. Der Standard ist hier big–endian–Format, das sie durch setzen eines Bits auf little–endian–Format umschalten läßt.

  Die Ultra SPARC II hat 32 sichtbare 64–Bit–Allzweckregister (R0 bis R31), die durch Load– und Store–Instruktionen geschrieben und gelesen werden können. Außerdem gibt es 32 Gleitkommaregister für entweder 32–Bit–Werte (einfache Genauigkeit) oder 64–Bit–Werte (doppelte Genauigkeit). Paare von Registern können auch 128–Bit–Werte (vierfache Genauigkeit) unterstützen. In Wirklichkeit gibt es jedoch noch mehr Register. Die für das Programm sichtbaren Register nennt man deshalb Registerfenster. Die Ultra SPARC ist eine Load/Store–Architektur. Das heißt, das nur Load– und Store–Operationen direkt auf den Speicher zugreifen können, also Instruktionen, die Daten zwischen Speicher und Registern hin– und herschieben, d.h. alle Operanden für arithmetische und logische Instruktionen müssen von Registern kommen und all Ergebnisse in einem Register, d.h. nicht dem Speicher gespeichert werden.

15

Kontroll– und Datenpfad

Die Leistung eines getakteten Prozessors hängt von 3 Faktoren ab:

- instruction count
  d.h. der Anzahl von Befehlen, die von einem Programm ausgeführt werden
- clock cycle time
  d.h. die Sekunden oder eine andere Zeiteinheit, die ein Taktzyklus benötigt
- clock cycles per instruction (cpi)
  d.h. die mittlere Anzahl von Taktzyklen pro Instruktion

Daraus ergibt sich die Abarbeitungszeit eines Programmes als

$$CPU\ time\ =\ Instruction\ count \cdot cpi \cdot clock\ cycle\ time.$$

Die Größe instruction count hängt vom Compiler bzw. der ISA ab. Die anderen beiden Größen, d.h cpi und clock cycle time werden dagegen durch die Implementierung des Prozessors bestimmt.

Vor diesem Hintergrund wollen wir die Prinzipien der Implementierung eines Prozessors betrachten.

15.1 Prinzip der Prozessorarbeitsweise

Unabhängig von der Art des Befehls beginnt jeder Befehl mit 2 Schritten:

1. Hole den Befehl mit der Adresse des PC (Programm Counter) aus dem Speicher.

2. Lies ein oder zwei Register, die in Relation zu dem entsprechenden Befehl stehen.

Alle weiteren Schritte hängen dann von der Befehlsklasse ab (memory–reference, branch oder arithmetic–logical).

Trotz unterschiedlicher Befehlsausprägungen haben Befehle verschiedener Klassen Gemeinsamkeiten.
So nutzen z.B. alle Befehlsklassen die ALU:

- memory–reference–Instruktionen für die Adreßberechnung
- arithmetisch–logische Instruktionen für die auszuführende Operation
- branch–Instruktionen zum Vergleich von Größen

Inwiefern danach Daten gelesen oder geschrieben werden bzw. an eine bestimmte Befehlsadresse gesprungen wird, hängt von dem jeweiligen Befehl ab.

Das grundlegende Prinzip der Abarbeitung dieser drei Befehlsklassen wollen wir auf dem momentanen, noch sehr abstrakten Niveau, wie in Abbildung 15.1 zu sehen, zusammenfas-

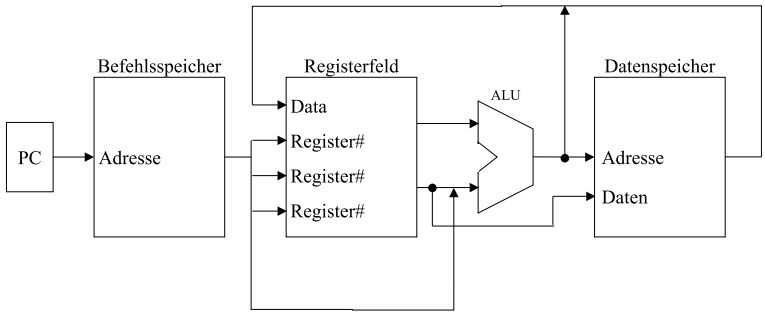


Abbildung 15.1: logische Funktionsweise des Prozessors

sen.

Alle Befehle beginnen mit der Nutzung des Befehlszählers (PC), der die Adresse des nächstauszuführenden Befehls an den Befehlsspeicher gibt. Der zugehörige Befehl wird geholt und in seine Felder zerlegt.

Insbesondere können – z.B. bei arithmetischen Operationen – bis zu drei Register spezifiziert sein.
In Abhängigkeit der spezifischen Operationen gehen Daten ganz spezifische Wege:

- Bei Load– und Store–Operationen werden aus Registeroperanden Speicheradressen berechnet, in die oder von denen Registerinhalte in den/von dem Datenspeicher kopiert werden.
- Bei arithmetisch–logischen Operationen wird das Ergebnis einer ALU–Berechnung in ein Register zurückgeschrieben.

- Bei Branch–Operationen führt die ALU einen Vergleich aus, ihr Output bestimmt die nächste Befehlsadresse. Für die Realisierung einer solchen Operation ist jedoch zusätzliche Kontrolllogik erforderlich.

Später werden zusätzliche Verbindungen zwischen den einzelnen Komponenten entstehen und neue funktionale Komponenten benötigt werden, die wir bei verschiedenen Detaillierungen hinzunehmen.

15.2 Logischer Entwurf und Taktung

Bei den funktionalen Einheiten des MIPS–Prozessors müssen wir zwei Typen unterscheiden:

- Elemente, die Daten verarbeiten, d.h. deren Output vom aktuell anliegenden Input abhängt, z.B. die ALU. Mit dem gleichen Input erzeugen diese Elemente stets den gleichen Output. Diese Elemente haben keinen internen Zustand. Man spricht auch von kombinatorischen Elementen.
- Elemente, die einen Zustand enthalten. Solche Elemente haben intern einen Speicher. Sie werden auch Zustandselemente genannt. Merkt man sich die Zustände von diesen Elementen und lädt man sie entsprechend wieder, so kann die Abarbeitung erfolgen, als wäre sie an dieser Stelle nie unterbrochen worden.

  Zustandelemente haben Ein– und Ausgaben. Zu den Eingaben gehören dabei Datenwerte und Taktsignale. Ein Beispiel für ein ganz einfaches Zustandselement ist der D–Flip–Flop aus Kapitel 7. Er hatte genau einen Dateneingang und einen Taktsignaleingang sowie einen Ausgang.

  Bezogen auf unseren logischen Prozessor gehören zu den Zustandselementen der Daten– und Befehlsspeicher sowie das Registerfeld. Solche Zustandselemente werden auch sequentielle Elemente genannt. Ihr Output hängt dabei sowohl von Ihrem Input als auch ihrem internen Zustand ab.

Wichtig ist zunächst eine präzise Synchronisation zwischen lesenden und schreibenden Zugriffen zu erreichen, so dass diese in einer gewissen Ordnung oder Reihenfolge auftreten. Um dies sicherzustellen, wird eine Taktung eingeführt.

Wie erfolgt die Taktung von sequentiellen Elementen?
Wir gehen von einer flankengesteuerten Taktung aus, nehmen dabei steigende Flanke an. Alle kombinatorischen Elemente müssen damit die Verarbeitung ihrer Daten abgeschlossen haben, bevor die steigende Flanke anliegt. Denn nur sequentielle Elemente können Daten speichern und daher sollten kombinatorische Elemente i.d.R. ihre Inputs von einer Menge sequentieller Elemente beziehen und ihre Outputs an eine Menge sequentieller Elemente weitergeben. Inputs eines kombinatorischen Elements sind gewöhnlich in einem vorhergehenden Takt generiert worden. Outputs werden entsprechend in nachfolgenden Takten weiterverarbeitet. Damit ergibt sich eine zeitliche Abhängigkeit wie in Abbildung 15.2.

Berücksichtigt man diese zeitliche Abhängigkeit, so kann auch eine Kombination von nur einem Zustandselement und einem kombinatorischen Element wie in Abbildung 15.3 erfol-

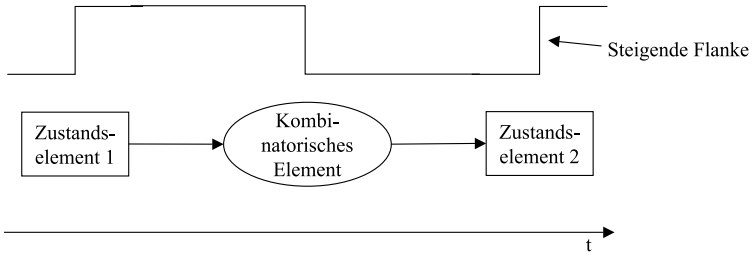


Abbildung 15.2: Zeitliche Abhängigkeit

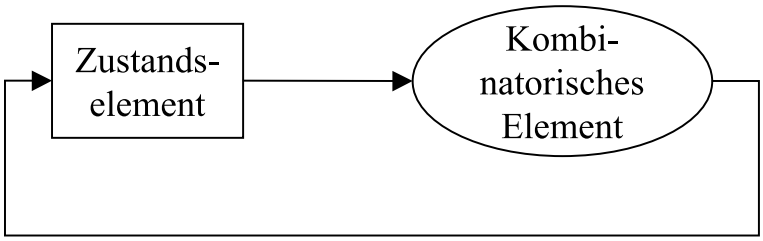


Abbildung 15.3: Kombination

gen. Wichtig ist dabei in jedem Fall, dass die Verarbeitung der Daten im kombinatorischen Element innerhalb des einen Taktzyklus abgeschlossen ist und nicht länger dauert! Oder umgekehrt definiert man: die Dauer eines Taktzyklus muß mindestens so lang sein, wie die Verarbeitung der Daten durch das kombinatorische Element dauert.

15.3 Der Datenpfad

Wir wollen die Komplexität des Datenpfades zunächst dadurch reduzieren, dass wir nur die Komponenten betrachten, die von jeder Klasse von MIPS–Instruktionen benötigt werden.

1. In jedem Fall müssen die Instruktionen eines Programms gespeichert werden können. Dazu brauchen wir ein Zustandselement, das wir als **Befehlsspeicher** bezeichnen.

2. Die Adresse eines jeden Befehls wird ebenfalls als Zustandselement gespeichert, das wir als Program Counter (PC) bezeichnen.

3. Schließlich benötigen wir ein Addierwerk, um den PC geeignet erhöhen zu können, damit er auf den nächsten Befehl zeigt. Diese Addierwerk wollen wir als Bestandteil der ALU betrachten.

Da wir Zustandselemente als Rechtecke darstellen, ergeben sich die Elemente wie in Abbildung 15.4.

Um eine beliebige Instruktion auszuführen, müssen wir diese zunächst aus dem Befehlsspeicher holen. Um das Holen der nächsten Operation gleichzeitig vorzubereiten, muß der PC

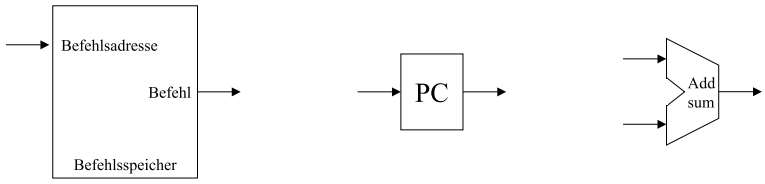


Abbildung 15.4: Zustandselemente

inkrementiert werden. D.h. bei 32 Bit langen Maschinenebefehlen muß er um 4 erhöht werden, da eine Speicherzelle 1 Byte umfaßt. Damit können die oben ausgeführten 3 Elemente zu folgendem Datenpfad für das Holen von Instruktionen und Erhöhen des Befehlszählers zusammengesetzt werden (vgl. Abbildung 15.5).

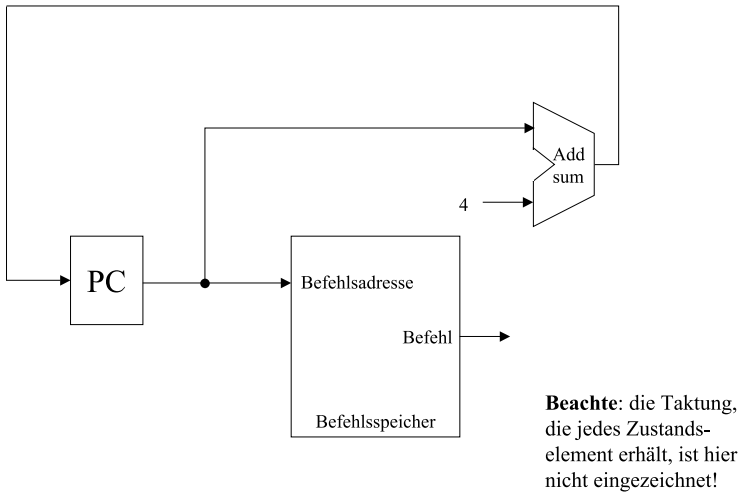


Abbildung 15.5: Datenpfad zum Holen von Instruktionen und Erhöhen des PC

Die gelesene Instruktionen wird dann in den nachfolgenden Teilen des Datenpfads weiterverarbeitet. Sie hat die Struktur, die wir aus Abschnitt 12.3 kennen, d.h. sie besteht aus maximal 6 Feldern op, rs, rt, rd, shamt, func.

Nun wollen wir die arithmetisch–logischen Operationen, die sogenannten R–Instruktionen betrachten.

Zur Wiederholung: wir hatten als Felder

6 Bits	5 Bits	5 Bits	5 Bits	5 Bits	6 Bits
op	rs	rt	rd	shamt	func

z.B. für add $t1, $t2, $t3 im Sinne von $t1 := $t2 + $t3.

Wir wollen nun die 32 Register des MIPS–Prozessors als Zustandselemente eines Registerfeldes betrachten. Da R–Format–Instruktionen 3 Felder für Registernummern haben, brauchen wir diese auch als Inputs für das Registerfeld. 2 Register für einen Input und ein Register

für den Output der Operation, die mittels der ALU berechnet wurde. Zum Schreiben von Daten benötigen wir neben der Spezifikation des Registers noch einen Input, über den der eigentliche Datenwert übergeben werden kann. Als Output des Registerfeldes können die Werte der im Input spezifierten Register angegeben werden.

Wir müssen noch zwischen verschiedenen Zugriffen auf dieses Registerfeld unterscheiden, ob z.B. der o.g. R–Befehlscode hinsichtlich der 3 Registerbezeichnungen eingegeben wird oder das Ergebnis einer Addieroperation in das Destinationsregister geschrieben wird. Letztere Operation auf dem Registerfeld wird durch ein Write Control Signal angezeigt, das mit RegWrite bezeichnet wird.

Daraus ergibt sich das Zustandselement in Abbildung 15.6.

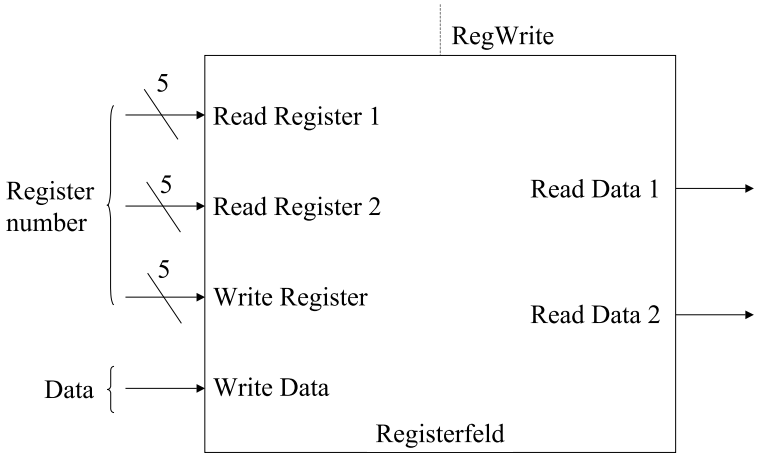


Abbildung 15.6: Zustandselement Registerfeld

Beachte:
Das Zeichen $\overset{5}{\nrightarrow}$ steht für einen Eingabebus mit 5 parallelen Leitung. Mit diesen 5 Bits ergeben sich $2^5 = 32$ Bitmuster.

Ähnlich wie das Registerfeld hat auch die ALU mehrere Möglichkeiten der Operationsausführung, z.B. add und sub. Im 6. Feld der Befehlscodierung hat die jeweilige Funktion eine Kodierung z.B.

Addition	= 32 =	100000
Subtraktion	= 34 =	100010
AND	= 36 =	100100
OR	= 37 =	100101
slt	= 42 =	101010

Der MIPS–Prozessor benutzt eine ALU, die in der 1–Bit–Form den Aufbau wie in Abbildung 15.7 hat.

Daraus ergibt sich, dass ein Kontrollbit zwecks Steuerung der Invertierung notwendig ist, zwei extra Kontrollbits zur Steuerung der Operation. Für diese 3 Bits ergibt sich für die wichtigsten Operationen folgender ALU control input:

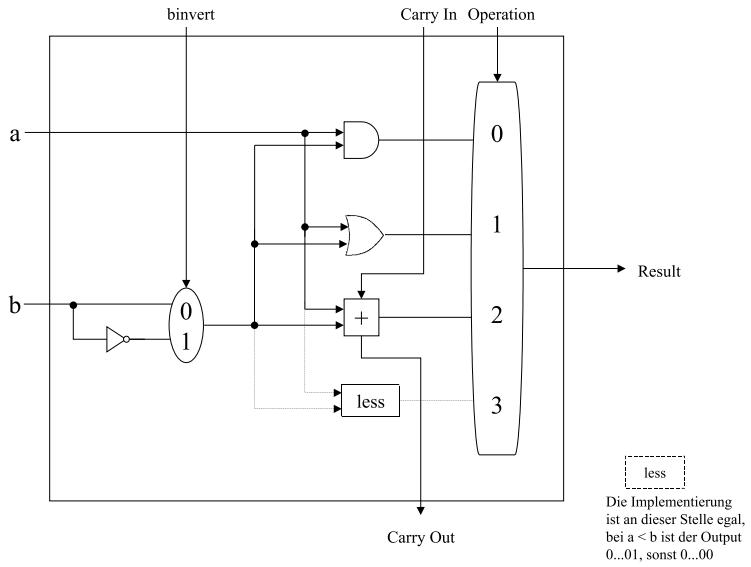


Abbildung 15.7: Aufbau der ALU beim MIPS

Addition	$a + b$:	010
Subtraktion	$a - b$:	110
AND	$a \wedge b$:	000
OR	$a \vee b$:	001
slt:		111

Weiterhin wären $a \wedge \neg b$ mit 100 sowie $a \vee \neg b$ mit 101 realisierbar.

Eine 32–Bit ALU wird durch ein geeignetes Zusammenspiel von 32 1–Bit ALUs realisiert (vgl. Abbildung 4.18. in Patterson Hennesy). Sie kann zusätzlich mit der Verknüpfung aller 32 Resultatausgänge durch einen sogenannten "Zero detecter" erhalten (vgl. Abbildung 4.19.

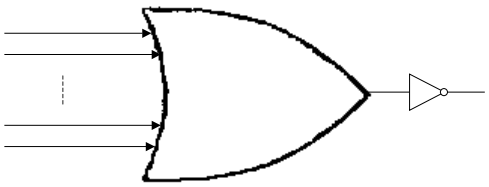


Abbildung 15.8: Verknüpfung

in Patterson Hennesy).

Die interne Realisierung dieses Bausteins soll uns im folgenden nicht weiter interessieren. Statt dessen führen wir für unseren Datenpfad den Baustein in Abbildung 15.9 ein.

Dabei erhalten wir 2 Inputs à 32 Bit und einen Output für ein 32–Bit–Result.

Damit ergibt sich für R–Type–Instruktionen der Datenpfad in Abbildung 15.10. Wir wollen nun von den R–Type–Instruktionen übergehen zu den I–Type–Instruktionen. Dieses Format wird für den Datentransport genutzt.

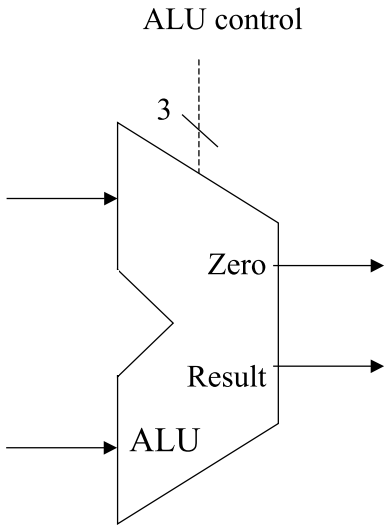


Abbildung 15.9: ALU

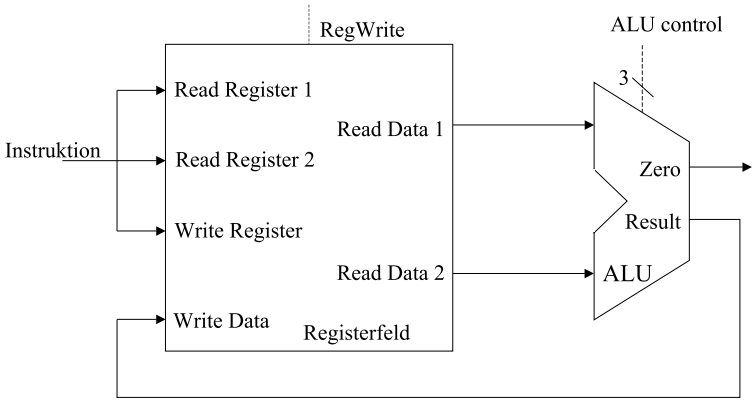


Abbildung 15.10: Datenpfad für R–Type

Da alle Instruktionen die gleiche Gesamtlänge haben, reicht die Feldlänge hier nicht aus, um z.B. eine Konstante im Sinne einer Adresse zu speichern. Ein gutes Design erfordert nun gute Kompromisse. Beim I–Type wird aus den letzten 3 Feldern ein Feld gemacht, das dann in der Summe 16 Bit groß ist

6 Bits	5 Bits	5 Bits	16 Bits
op	rs	rt	Adresse

z.B. ergibt sich für **lw** $s1, 100 ($s2) die Codierung
35 18 17 100
oder dual
100011 10010 10001 0000000001100100 .

Allgemein haben diese I–Type–Instruktionen die folgende Form:
lw $*s*1, offset-value ($*s*2) oder
sw $*s*2, offset-value ($*s*2).
Für die Abarbeitung muß zunächst die Speicheradresse für den lesenden oder schreibenden Zugriff berechnet werden. Dazu wird zunächst der Offset von einem 16–Bit–Wert in einen 32–Bit–Wert transformiert, indem ein Sign–Extension–shortcut ausgeführt wird. Dann wird zu diesen erweiterten Offset das Basisregister $s2 hinzuaddiert.

Im Falle einer Store–Instruktion wird der zu speichernde Wert aus einem Register gelesen, d.h. aus $s1. Im Falle einer Load–Instruktion wird der Datenwert von der berechneten Speicheradresse gelesen und in das Register $s1 geschrieben.

Damit benötigen wir die bereits betrachteten Bauelemente: das Registerfeld und die ALU sowie als neue Elemente:

- ein Zustandselement für den Datenspeicher (vgl. Abbildung 15.11)

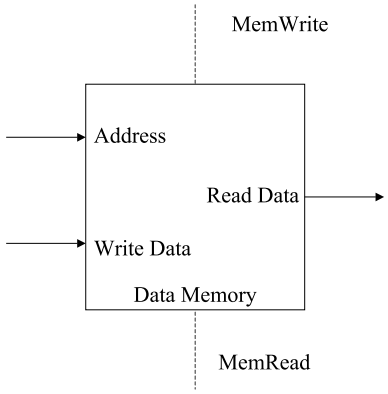


Abbildung 15.11: Datenspeicher

- und ein kombinatorisches Element für die Sign–Extension (vgl. Abbildung 15.12).

Dies ergibt für die Load– und Store–Instruktionen den Datenpfad in Abbildung 15.13.

Analog werden auch Branch–Instruktionen realisiert – dazu reichen nahezu die bereits behandelten Bauelemente aus. Der resultierende Datenpfad ist in Patterson Hennesy Seite 350 ersichtlich.

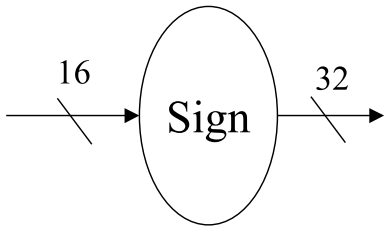


Abbildung 15.12: Element für Sign–Extension

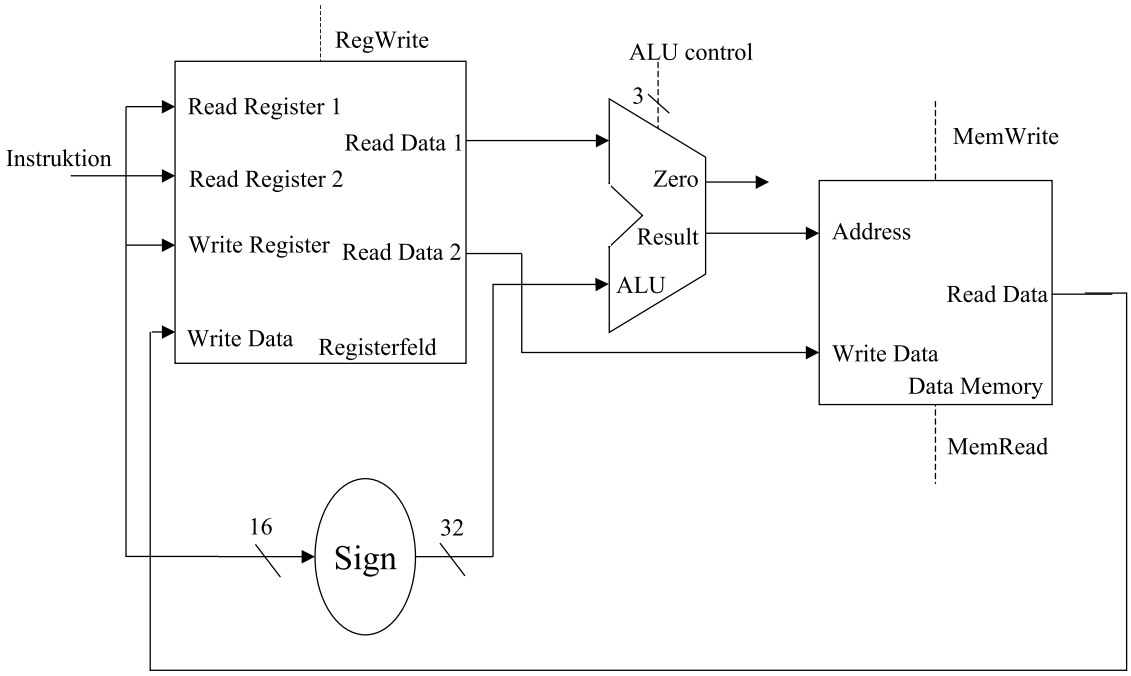


Abbildung 15.13: Datenpfad für Load– und Store–Instruktionen

Aus diesen Kombinationen von Bauelementen soll nun eine Gesamtkombination erfolgen.

Generierung eines Single Datapath

Der einfachste Datenpfad – ein single datapath – führt alle Instruktionen in je einem Taktzyklus aus. Dies bedingt, dass jede Ressource innerhalb dieses einen Taktzyklus nur maximal einmal benutzt werden kann. Wird ein Element mehr als einmal benötigt, so muß dieses so oft dupliziert werden, wie es benötigt wird. Daraus resultiert auch die Trennung von Befehls– und Datenspeicher.

Beachte jedoch umgekehrt: einige Elemente, die z.B. von den arithmetisch–logischen und den Speicher–Referenz–Instruktionen benötigt werden, können von unterschiedlichen Instruktionsflüssen gleichermaßen genutzt werden. Dies betrifft z.B. das Registerfeld und die ALU.

Um solche Elemente für verschiedene Instruktionsklassen gleichermaßen zu nutzen, müssen verschiedene Inputs möglich sein. Ferner muß ein Kontrollsignal die Inputs entsprechend steuern. Dies wird über einen Multiplexer geregelt. Er wählt Daten aus.

Wir wollen uns exemplarisch die Kombination der R–Type–Instruktionen mit den Speicherzugriffsinstruktionen anschauen. Dabei sind zwei Dinge zu beachten:

- Der 2. Input der ALU ist entweder ein Registerinhalt (im Falle einer R–Type–Instruktion) oder die sign–extension der Speicheradresse (im Falle einer Speicherreferenzinstruktion).
- Der in das Registerfeld zu schreibende Wert kommt in Falle der R–Type–Instruktion von der ALU, im Falle der Speicherreferenzinstruktion von Speicher.

Daraus ergibt sich:

15.4 Kontrolle eines Prozessors

Wir betrachten zunächst noch einmal den Aufbau der Maschinenbefehle:

R–Type:	op–code	rs	rt	rd	shamt	func
Bit–Position:	31–26	25–21	20–16	15–11	10–6	5–0
Load–and–Store:	op–code	rs	rt	address		
	31–26	25–21	20–16	15–0		
Branch:	op–code	rs	rt	address		
	31–26	25–21	20–16	15–0		

Dabei ist im Feld op–code im Falle der R–Type–Instruktionen der Wert 0 enthalten, bei Load und Store der Wert 35 bzw. 43 und bei Branch–Instruktionen der Wert 4. Die zu lesenden Register sind immer an den Positionen 25–21 bzw. 20–16 zu finden, das Basisregister für Load und Store ist jeweils rs mit der Bit–Position 25–21. Das Zielregister bei Load kann auch in 20–16 stehen, bei den R–Type–Instruktionen steht es in 15–11. Um auszuwählen,

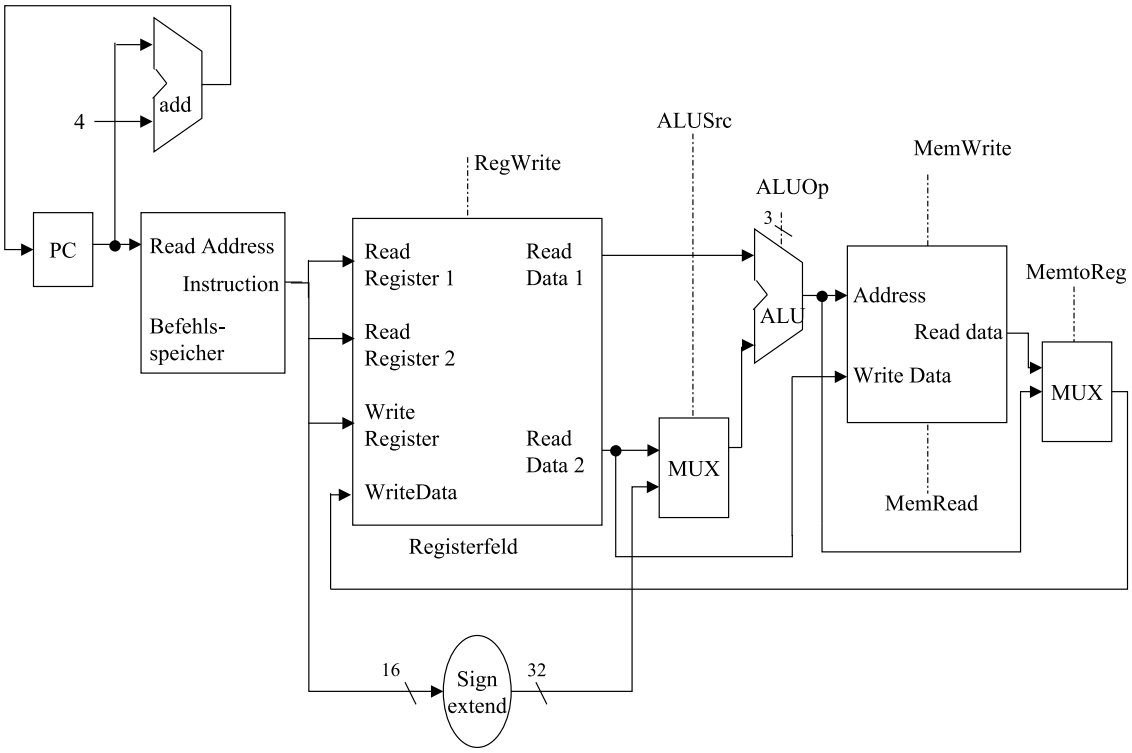


Abbildung 15.14: Datenpfad

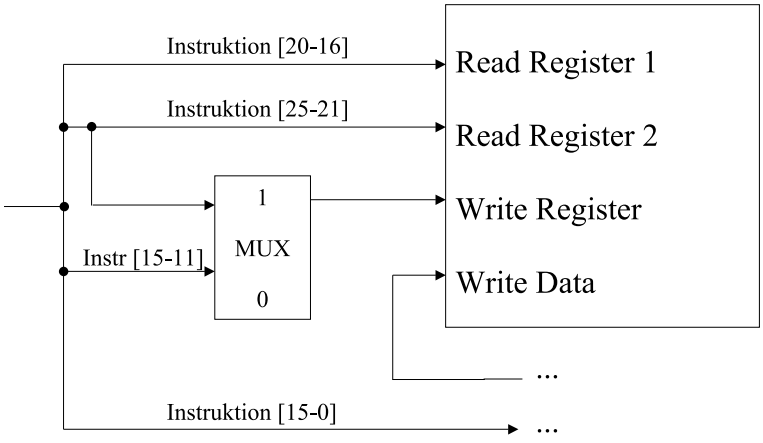


Abbildung 15.15: Einfügen des Multiplexers

welches Feld für das zu schreibende Register genutzt wird, muß folglich ein Multiplexer für die "Register Destination" ergänzt werden (vgl. Abbildung 15.15).

Der Multiplexer benötigt damit einen einfachen Eingang für die Steuerung. Insgesamt haben wir 7 einfache Steuersignale und das Steuersignal für die ALU, das auf ein 2–faches reduziert wird. Damit kann für den Kontrollpfad eine Kontrollfunktion ergänzt werden, deren Output in die verschiedenen Multiplexer mündet (vgl. Abbildung 15.16). Die Kombinati-

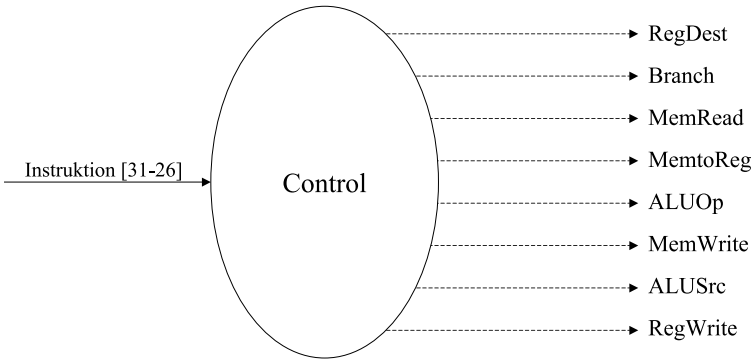


Abbildung 15.16: Control Element

on von Kontroll– und Datenpfad ergibt die in Patterson Hennesy auf Seite 360 abgebildete Darstellung.

15.5 Single Cycle versus Multiple Cycle Implementation

Die bislang betrachtete Single Cycle Inmplementation wird in modernen Rechnerarchitekturen nicht benutzt, da sie ineffizient ist. Man betrachte die Formel von Anfang von Kapitel 15:

$$CPU\ time\ =\ Instruction\ count \cdot cpi \cdot clock\ cycle\ time.$$

Jede Instruktion wird innerhalb eines einfachen Zyklus abgearbeitet (singel cycle), daher ist cpi = 1. Die "clock cycle time" wird durch den längsten denkbaren Datenpfad bestimmt. Dies ist eine Load–Instruktion, die nacheinander 5 funktionale Einheiten benutzt:

Befehlsspeicher – Registerfeld – ALU – Datenspeicher – Registerfeld

Man nehme dabei folgende Ausführungszeiten und prozentualen Anteil für die einzelnen Instruktionsklassen und an:

	Befehlsspeicher	Registerfeld	ALU	Datenspeicher	Registerfeld	Σ	%
R–Type	2	1	2	–	1	6ns	44%
Load	2	1	1	2	1	8ns	24%
Store	2	1	1	2	–	7ns	12%
Branch	2	1	2	–	–	5ns	18%
Jump	2	–	–	–	–	2ns	2%

In der Praxis würde jede Klasse von Befehlen 8ns für dne Zyklus brauchen. Im Mittel wären im theoretischen Fall 6,34ns nötig. Damit könnte die CPU theoretisch $\frac{8ns}{6{,}34ns} = 1,2618$ mal schneller arbeiten. Eine analoge Rechnung für Floating Point Instruktionen ergibt einen Verbesserungssfaktor von 2,9.

Die Ausführung jeder Instruktion wird nun in eine Folge von Stufen aufgespalten, wobei eine Stufe der Abarbeitung einer funktionalen Einheit entspricht. Daraus resultiert der Ansatz der **Multicycle Implementation**.

Jede Stufe, d.h. jeder Schritt (step) benötigt genau einen Taktzyklus. Neu ist nun, dass eine Instruktion in der Regel mehr als einen Taktzyklus benötigt und jede Klasse von Instruktionen kann unterschiedliche Anzahlen von Taktzyklen benutzen. Auch kann eine funktionale Einheit mehr als einmal pro Zyklus benutzt werden. Dadurch reduziert sich die benötigte Hardware.

Es ergibt sich das Grundprinzip der Multicycle–Architektur wie in Abbildung 15.17.

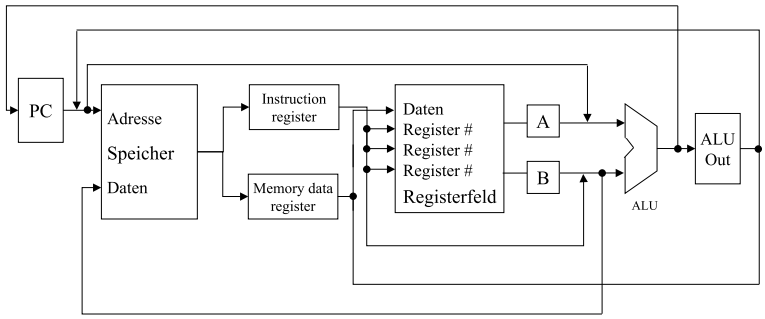


Abbildung 15.17: Grundprinzip der Multicycle Architektur

Damit ergeben sich grundlegende Unterschiede zum single cycle Datenpfad:

- Ein gemeinsamer Speicher kann sowohl für Daten als auch Befehle genutzt werden.
- Anstelle der verschiedenen ALUs und Addierwerke (zum Inkrementieren des PC, für arithmetische Instruktionen,...) wird nur noch eine ALU verwendet.
- Es werden zusätzliche Register benötigt, um einen Wert für den nachfolgenden Taktzyklus zwischenzulagern: das Instruction Register (IR), das Memory Data Register (MDR), sowie die Register A, B und ALUOut.

Daten, die innerhalb einer Instruktion am Ende eines Taktzyklus für einen nachfolgenden Taktzyklus bereitgestellt werden müssen, müssen in einem Zustandselement gespeichert werden.

Daten, die für nachfolgende Instruktionen benötigt werden und dann auch in nachfolgenden Taktzyklen weiterverarbeitet werden, müssen in Zustandselementen abgelegt werden, die für den Programmierer sichtbar sind, z.B. das Registerfeld, der PC oder der Speicher. Die von der selben Instruktion in nachfolgenden Zyklen benutzten Werte werden i.d.R. in den anderen Zustandselementen abgelegt.

Durch diese neue Architektur wird nun allerdings die Steuerung entsprechend komplexer – die resultierende Architektur ist in Patterson Hennesy Seite 380 dargestellt, mit entsprechender Steuerung auf Seite 383. Damit lassen sich die einzelnen Phasen eines Befehlszyklus für spezielle Prozessorarchitekturen jetzt nachvollziehen.

16

Pipelining

Unter dem Pipelining verstehen wir eine Implementierungstechnik, bei der sich die Ausführung mehrerer Instruktionen überlappt. Dabei wird die Parallelität zwischen Befehlen in einem sequentiellen Befehlsstrom ausgenutzt. Pipelining ist für den Programmierer transparent, also nicht sichtbar. Wir wollen uns das Grundprinzip zunächst am Beispiel des Wäschewaschens veranschaulichen.

Wir sprechen nicht von Pipelining, wenn wir folgende 4 Schritte mehrmals sequentiell hintereinander ausführen:

1. Waschen von Wäsche in der Waschmaschine
2. Trocknen der nassen Wäsche in Trockner
3. Bügeln der trockenen Wäsche
4. In den Schrank legen der gebügelten Wäsche

Bei z.B. 4 Waschladungen kann man die resultierenden 16 Schritte schneller ausführen, wenn man parallel zum Trocknen der ersten gewaschenen Wäsche bereits die 2. Maschine laufen läßt. Sind diese Arbeitsgänge beendet, so kann man mit dem Bügeln beginnen, während die 2. Ladung im Trockner ist und die 3. Waschmaschine läuft. Bei hypothetisch angenommenen Zeitdauern von je 1 Stunde pro Arbeitsgang ist man somit bereits nach 7 Stunden statt nach 16 Stunden fertig.

Bei zunehmender Anzahl von Waschmaschinenläufen wird dieses Verhältnis noch extremer: bei 20 Waschgängen benötigt man sequentiell 80 Stunden, parallel geschachtelt 23 Stunden.

16.1 Prinzip des Pipelinings

Dieses Prinzip wollen wir nun auf einen Prozessor anwenden, in den die Ausführung der Instruktionen "gepipelined" wird. Dazu unterteilen wir die Befehlsausführung in 5 Schritte:

1. Hole die Instruktion aus dem Speicher (Fetching)
2. Dekodiere die Instruktion und lies die entsprechenden Register (die MIPS–Befehlsformate erlauben es, Instruktionen zu dekodieren und simultan dazu Register zu lesen)
3. Führe eine Operation mittels ALU aus bzw. berechne mittels der ALU eine Adresse
4. Greife auf einen Operanden im Datenspeicher zu
5. Schreibe das Ergebnis in das entsprechende Register

Wir wollen nun den Single Cycle Datapath mit dem Pipelining vergleichen.

16.2 Leistungsbewertung

Als Beispielszenario betrachten wir noch einmal 8 Operationen hinsichtlich der Dauer der 5 genannten Schritte: lw, sw, add, sub, and, or, slt und beq.

Instruktionsklasse	Instruktion holen	Register lesen	ALU–Operation	Daten–zugriff	Register schreiben	Gesamtzeit
lw	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
sw	2 ns	1 ns	2 ns	2 ns	–	7 ns
R–Type:						
add, sub, and, or, slt	2 ns	1 ns	2 ns	–	1 ns	6 ns
beq	2 ns	1 ns	2 ns	–	–	5 ns

Dabei ist recht schön zu sehen, dass die Ausführungszeiten der einzelnen Schritte den Zugriffszeiten auf die Bausteine des Datenpfads entsprechen.

In der Single Cycle Implementierung brauchte jede Instruktion genau einen Zyklus. Dieser Zyklus dauerte so lange, wie die langsamste Instruktion d.h. 8 ns. Eine sequentielle Ausführung von 4 Ladebefehlen braucht bei einer Implementierung wie in Abbildung 16.1 32 ns. Verwendet man jedoch Pipelining, so benötigt man lediglich 15 ns (vgl. Abbildung 16.2).

Der Geschwindigkeitsvorteil hängt nun von der Anzahl der eingefüllten Schritte ab. Unter idealen Bedingungen ist das Pipelining so oft schneller, wie wir Schritte einführen. Bezogen auf o.g. Beispiel müßten dazu aber alle Schritte gleich lang sein und es müßten sehr viele hintereinander betrachtet werden, um die Anfangsphase des Auslastens nicht aller Einheiten auszugleichen.

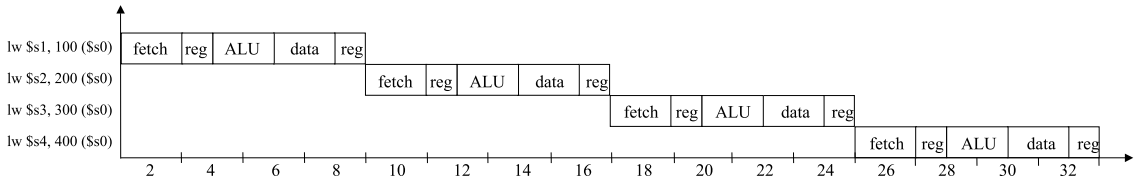
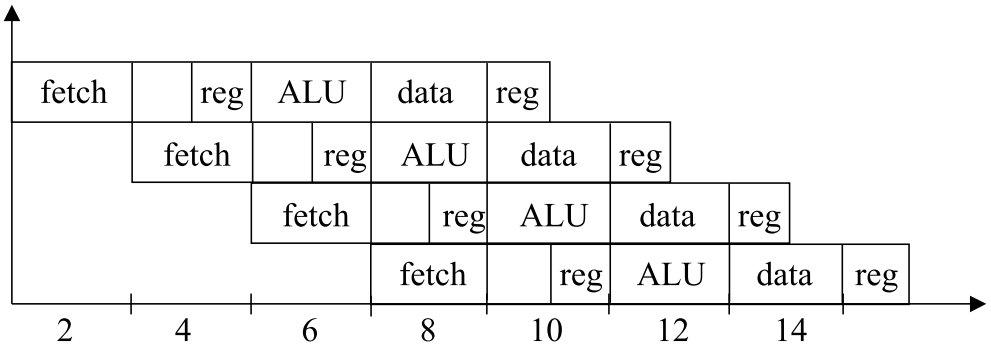


Abbildung 16.1: sequentielle Ausführung



Beachte: auch kürzere Schritte wie z.B. Registerzugriffe müssen sich an die längeren Schritte anpassen

Abbildung 16.2: mit Pipelining

Beispiel 16.1*:* Bei 1000 Instruktionen:
ohne Pipelining: 8000 ns;
mit Pipelining: 2007 ns,
d.h. etwa Faktor 4, da die beiden Registerzugriffe für Overhead zeitlich gesehen sorgen.

Dieser zeitliche Gewinn resultiert aus der Erhöhung des Durchsatzes, d.h. in der gleichen Zeit werden mehr Befehle abgearbeitet. Da große, reale Programme Milliarden von Instruktionen bearbeiten, ist dieser Durchsatz von großer Bedeutung.

Aus diesen einführenden Bemerkungen soll auf Designkriterien für den Prozessor geschlossen werden:

- Alle MIPS–Instruktionen haben die gleiche Länge. Dies erleichtert das uniforme Holen von Befehlen (Fetch Phase).
- Der MIPS–Prozessor hat eine geringe Anzahl von Instruktionstypen, die aber ähnlich aufgebaut sind. Lesende und schreibende Register sind nach Möglichkeit analog angeordnet. Damit kann ein Lesen von Registern parallel zum Bestimmen der Art einer Instruktion erfolgen. Wäre diese sogenannte Symmetrie nicht vorhanden, so müßte das Pipelining in 6 Schritten erfolgen, d.h. der Schritt 2 aufgespaltet werden in das Dekodieren und in das Lesen der Register.
- Die Load– und Store–Architektur des MIPS–Prozessors hat den Vorteil, dass nie sowohl eine Adreßberechnung als auch eine Operationsausführung in einer Instruktion

auftauchen. Bei Architekturen, die nicht nach dem Load–Store–Prinzip arbeiten, wären daher zusätzliche Schritte beim Pipelining erforderlich.

Der Durchsatz ist die Anzahl der abgearbeiteten Instruktionen pro Zeiteinheit. $\approx \frac{1}{ZAZ}$

ZAZ:
Zwischenankunftszeit von Instruktionen, d.h. Zeit zwischen dem Eintritt aufeinander folgender Instruktionen in die 1. Stufe der Pipeline.

$ZAZ_{pipelined} = \frac{ZAZ_{nonpipelined}}{K}$ (im Idealfall)

im Beisspiel: $ZAZ_{pipelined} = \frac{8ns}{5} = 1,6ns \neq 2ns$

Die Ursache dafür ist die schlechte Balance der Pipeline–Stufen.(Varianz der Ausführungszeiten bei den Basisoperationen; hier: Registerzugriff = 1ns, alle anderen = 2ns) Aber Pipelining erhöht den Durchsatz der bearbeiteten Instruktionen.

Durchsatz = $\frac{1}{CPI}$

7 Taktzyklen, 3 Befehle: $CPI = \frac{7}{3} = 2,\bar{3}$

1000 Befehle $\Rightarrow$ Taktzyklen $\frac{(n-1)+k}{n} \rightarrow 1$

Pipelining reduziert nicht die Ausführungszeit einer Instruktion.

Die Berechnung des Speedup zum Vergleich der Gesamtausführungszeit steht im folgenden Beispiel.

Beispiel 16.2*:* Speedup = $\frac{12}{7} = 1,7$ (für 3 Instruktionen)

aber Speedup = $\frac{8000}{2006}$ = 3,98 (für 1000 Instruktionen)

$\Rightarrow$ Speedup ist also unabhängig von n

Speedup = $\frac{n\dot{K}}{n-1+K}$

n= Anzahl der Instruktionen, K= Anzahl der Stufen

$\Longrightarrow$ Pipelining reduziert die Gesamtausführungszeit eines Programms, obwohl die Ausführungszeit einer Instruktion unverändert bleibt. (oder aufgrund von x sogar ansteigt)

Gründe für die Begrenzung des Speedup / Erhöhung des Durchsatzes:

- Balance der Pipeline–Stufen
- Overhead verursacht durch Verzögerungseinflüsse der Pipeline–Implementierung (ca. 10% in jeder Pipelinestufe)
- Abhängigkeiten zwischen Instruktionen (Hazards)

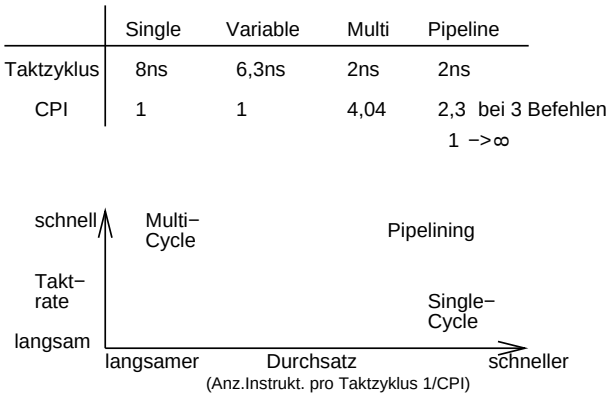


Abbildung 16.3: Vergleich: Taktrate / Durchsatz (CPI)

16.3 Pipeline Hazards

Es gibt Situationen, bei denen die nächste Instruktion nicht im nächsten Taktzyklus ausgeführt werden kann. Solche Ereignisse werden als Hazards (Beeinträchtigung) bezeichnet. Wir wollen 3 Typen von Hazards betrachten.

Strukturelle Hazards (structural hazards): Die Hardware ist nicht in der Lage, bestimmte Teile von unterschiedlichen Instruktionen in einem Zyklus auszuführen.

Ursachen:

- Elemente sind nicht vollständig pipeline implementiert d.h. Element wird in allen Stufen der Abarbeitung einer Instruktion genutzt (z.B. Main Control beim Single Cycle Datapath)
- Elemente sind nicht ausreichend dupliziert um alle Befehlskombinationen in der Pipeline ausführen zu können (z.B. Speicherelement im Multi Cycle Datapath)

Bezogen auf unsere Wäsche wäre eine solche Situation durch ein kombiniertes Wasch–maschine/Trockner–Gerät gegeben. Damit kann man nicht parallel Wäsche waschen und gewaschene Wäsche trocken.

Unsere vorgestellte Instruktionsmenge ist jedoch speziell auf das Pipelining zugeschnitten, daher ergeben sich solche strukturellen Hazards bei uns nicht. Z.B. hätten wir allerdings einen strukturellen Hazard, wenn Daten– und Befehlsspeicher eine Einheit bilden würden, d.h. nur von einer Instruktion genutzt werden könnten.

Steuer–Hazards (control hazards): Diese Hazards resultieren aus der Situation, Entscheidungen erst basierend auf einem späteren Ausführungsschritt oder dem noch ausstehenden Ergebnis einer vorangegangenen Instruktion treffen zu können.

Ein Wäschebeispiel wäre hierfür, wenn man eine Reihe stark verschmutzter Stücke hat, die mehrmals hintereinander heiß gewaschen werden müssen. Ist nun ein Teil bei der Wäsche stark eingegangen, so sollte es im nächsten Waschgang nicht mehr so heiß gewaschen werden. Ob ein Wäschestück jedoch eingegangen ist, kann man erst nach dem Trocknen feststellen.

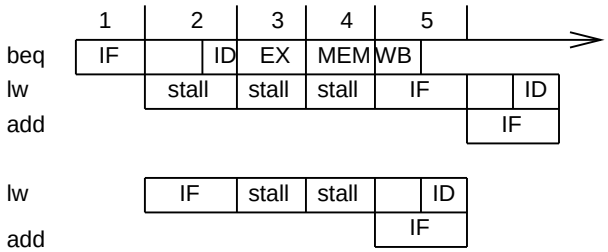


Abbildung 16.4: sequentielle Ausführung

Beim Prozessor könnte dies im Fall einer Branch–Instruktion auftreten. Basierend auf dem Vergleich zweier Zahlen wird die Entscheidung für einen Sprung getroffen. Hierbei kann nicht das Holen zweier Instruktionen unmittelbar aufeinanderfolgen, sondern es muß gewartet werden, bis entsprechende Registerwerte gelesen und ausgewertet wurden. Daraus ergibt sich eine Verzögerung der Taktzyklen, die verschenkt werden müssen, um entsprechende Werte bereitzustellen. Entsprechende Leistungsbetrachtungen sind in Patterson Hennesy Seite 442ff dargestellt.

Das Problem dabei ist, dass die Verzweigung erst am Ende der 2. Stufe als solche erkannt wird. Daraus folgt, dass mit der Bearbeitung der folgenden Instruktion zwangsläufig begonnen wird und ab Erkennung bis zur Entscheidung Wartezyklen eingelegt werden.

Eine Alternative dazu is die Verzweigungsvorhersage, d.h. man nimmt an, dass die Bedingung nicht eintritt. Die nachfolgenden Instruktionen werden verworfen, wenn die Annahme nicht eintritt.

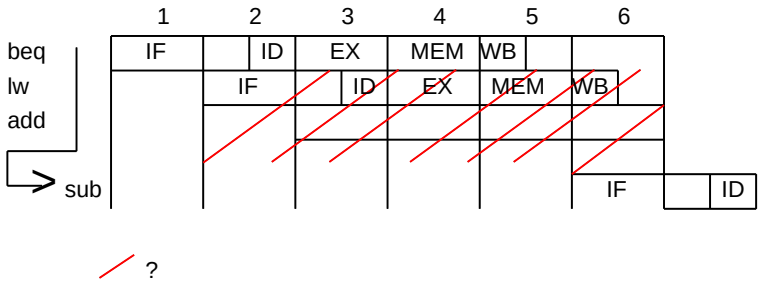


Abbildung 16.5: Verzweigungsvorhersage

Hierbei ist wiederum das Problem, dass die fortgeschrittene Bearbeitung der Instruktionen nach Beginn kann zu unerwünschten Nebeneffekten führen, falls die Verzweigung durchgeführt wird. (z.B. bei SW)

Die Lösung sind zusätzliche Elemente und eine Kontrolllogik, die die Verzweigungsbedingung früher (in der 2. Stufe) prüfen.

Daten–Hazards (data hazards): Diese Hazards treten auf, wenn die Reihenfolge des Operandenzugriffs durch die Pipeline gegenüber der normalen, durch sequentielle Ausführung der Instruktionen bestimmte Reihenfolge geändert wird.

Im Waschbereich kann dies auftreten, wenn man von einem Paar Socken erst eine gewaschen hat. Man muß warten, bis beide Socken getrocknet sind, bevor man sie in den Schrank legen kann.

Beispiel 16.3*: add* $1, $2, $3
sub $4, $1, $5
Der add–Befehl schreibt ein Register, das Quelloperand für den sub–Befehl ist, aber add beendet das Schreiben erst drei Taktzyklen nachdem sub darauf zugreift.

Lösung:
ALU–Ergebnis wird vorzeitig in Pipeline–Registern zwischengespeichert.

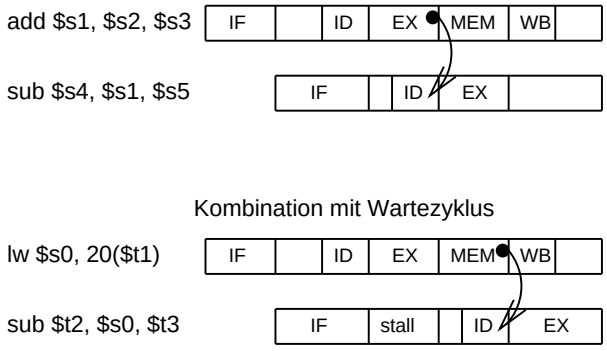


Abbildung 16.6: Auflösung des Datenhazard beim Pipelining

Wir wollen an dieser Stelle nicht die verschiedenen Optimierungsmöglichkeiten betrachten, die es neben dem Anhalten von Befehlen gibt, sondern die Merkmale des Pipelining noch einmal kurz zusammenfassen:

- Pipelining erhöht die Anzahl simultan ausgeführter Instruktionen. Damit erhöht sich auch die Rate, mit der Instruktionen gestartet und beendet werden und folglich auch der Durchsatz.
- Pipelining reduziert jedoch nicht die Bearbeitungszeit einzelner Instruktion: eine 5–stufige Pipeline benötigt immer 5 Taktzyklen für die Beendigung der Ausführung.
- Pipelining erhöht damit den Durchsatz der Instruktionen, es verbessert jedoch nicht die Ausführungszeit.

Teil VII

Parallele Rechnerarchitekturen

17

Designkriterien für Parallelrechener

Immer höhere Anforderungen an Computer bedingen, dass diese immer schneller werden. Obwohl die Taktraten ständig steigen, kann die Geschwindigkeit von Schaltungen nicht unendlich erhöht werden.
Aus diesem Grund wird als Alternative das Konzept der Parallelrechner betrachtet. D.h. während es wahrscheinlich unmöglich ist, einen Computer mit einer CPU und einer Zykluszeit von 0,001 ns zu bauen könnte es sehr gut möglich sein, einen mit 1000 CPUs mit jeweils einer Zykluszeit von 1ns zu konstruieren. Die Rechenleistung wäre theoretisch die gleiche.
Wir wollen **zwei Anwendungen für Parallelrechner** betrachten:

- Bei Banken (z.B. als automatisierte Schalterterminals) oder bei Fluggesellschaften (z.B. als Reservierungssysteme) sind Transaktionsverarbeitungssysteme und große Webserver üblich, die bspw. als Computer mit 8 bis 64 CPUs für ein großes Timesharingsystem unter UNIX realisiert werden. Dabei sind Tausende von räumlich entfernten Benutzerstationen denkbar.
- Alternativ dazu gibt es Parallelrechner, die für einen einzigen Auftrag benutzt werden, der sich aus vielen parallelen Prozessen zusammensetzt. Ein Beispiel dafür wäre ein Schachprogramm, das ein bestimmtes Brett analysiert, indem es eine Liste mit zulässigen Zügen generiert und dann parallell Prozesse einteilt, um jedes neue Brett rekursiv parallel zu analysieren.

Das erste Beispiel dient dabei der Bedienung durch viele Nutzer, wobei mehrere unabhängige Jobs gleichzeitig ausgeführt werden. Da diese Jobs nichts miteinander zu tun haben, kommunizieren sie auch nicht. Das zweite der Beschleunigung einer einzelnen Problemlösung.
Parallelität ist bereits vom Pipelining bekannt. Bei einer Leistungssteigerung um den Faktor 100 oder 1000 müssen jedoch ganze CPUs oder zumindest ein grosser Teil davon wiederholt werden, damit sie effizient zusammenarbeiten.

17.1 Designkriterien für Parallelrechner

Beim Design gibt es drei grundlegende Fragen:

1. Von welcher Art, Größe und Anzahl sind die Verarbeitungselemente (CPU, ...)?
2. Von welcher Art, Größe und Anzahl sind die Speicherelemente?
3. Wie sind Verarbeitungs- und Speicherelemente zusammengeschlossen?

Verarbeitungselemente

Sie reichen von kleinen ALUs bis zur vollständigen CPU. Sofern dies ein kleiner Chip oder ein "Chipteil" ist, kann ein Computer bis zu einer Milion solcher Bauteile bekommen.
Handelt es sich beim Verarbeitungselement um einen vollständigen Computer mit eigenem Speicher und eigenen E/A-Geräten, so sind die Anzahlen geringer, obwohl schon Systeme mit fast 10.000 CPUs installiert worden sind.
Parallelrechner werden auch vermehrt aus kommerziell erhältlichen Teilen - insbesondere CPUs - gebaut.

Speicherelemente

Sie werden oft in Module unterteilt, die unabhängig voneinander aber parallel arbeiten, so dass von vielen CPUs gleichzeitig zugegriffen werden kann.
Diese Module können klein (KByte) oder groß (MByte) sein. Sie können eng mit den CPUs integriert sein oder sich auf einer anderen Schaltkarte befinden.
Oft werden auch Cache-Systeme mit 2, 3 oder sogar 4 Ebeen benutzt.

Verbindung von Verarbeitungs- und Speicherelementen

Bei den Verbindungsmethoden unterscheidet man statische und dynamische.
Bei den statischen Methoden werden alle Komponenten fest miteinander verbunden - als Stern, Ring oder Gitter.
Bei den dynamischen Methoden werden alle Teile zu einem Vermittlungsnetz verbunden, das Nachrichten zwischen den Komponenten dynamisch weiterleiten kann.
Ausserdem unterscheidet man lose und fest gekoppelte Systeme:
Systeme mit einer kleinen Zahl großer, unabhängiger CPUs die bei niedriger Geschwindigkeit miteinander verbunden sind, nennt man lose gekoppelt (**loosely coupled**).
Eng gekoppelte (**tightly coupled**) Systeme sind solche, bei denen die Komponenten im allgemeinen kleiner sind, eng zusammen und meist über Verbindungsnetze mit hoher Bandbreite interagieren.

Kommunikationsmodelle

Prinzipiell werden zwei verschiedene Designs unterschieden:

Mehrprozessorsyteme sind Systeme, bei denen sich alle CPUs einen gemeinsamen physischen Speicher teilen. Sie heißen auch **Shared Memory Systems.**

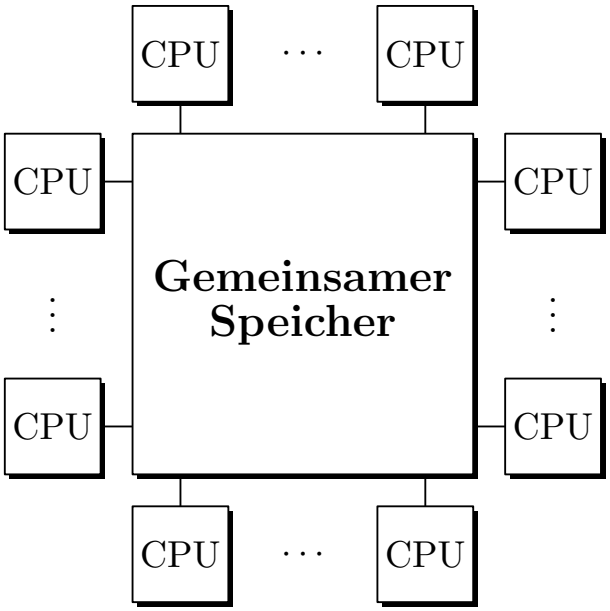


Abbildung 17.1: Shared Memory Systems

Die Kommunikation zwischen den Prozessen erfolgt über das Schreiben und Lesen von Daten in bzw. aus dem Speicher mittels LOAD- oder STORE-Instruktion.
Das ist für den Programmierer leicht verständlich und anwendbar und daher sind diese Systeme sehr beliebt.
Ein Anwendungsbeispiel wäre eine Bildverarbeitung, bei der jede CPU einen eigenen Bildausschnitt analysiert.
Mehrprozessorsysteme werden von vielen Anbietern vertrieben. Einige Beispiele sind SUN Enterprise 10.000, SGI Origin 2000 und HP/Convex Exemplar.

Mehrrechnersysteme sind ein Design für parallele Architekturen, bei denen jede CPU einen eigenen Speicher hat, auf den nur sie zugreifen kann. D.h. keine der anderen CPUs kann darauf zugreifen. Dieses Design wird auch als **Distributed Memory System** bezeichnet, in der Regel ist die Architektur lose gekoppelt.

Das System unterscheidet sich vom Mehrprozessorsystem vor allem darin, dass jede CPU einen privaten, lokalen Speicher hat, auf den sie mittels LOAD- und STORE- Instruktionen zugreifen kann. Dieser ist aber - wie gesagt - für keine anderen Instruktionen zugänglich. Damit ist ein physischer Adreßraum pro CPU vorhanden, nicht aber für alle CPUs gemeinsam wie bei den Mehrprozessorsystemen.
Zwecks Kommunikation tauschen die CPUs Nachrichten über ein Verbindungsnetz untereinander aus. Dazu stehen die Kommunikationsprimitiven SEND und RECEIVE zur Verfügung. Damit ist die Software viel komplizierter als bei Mehrprozessorsystemen - auch

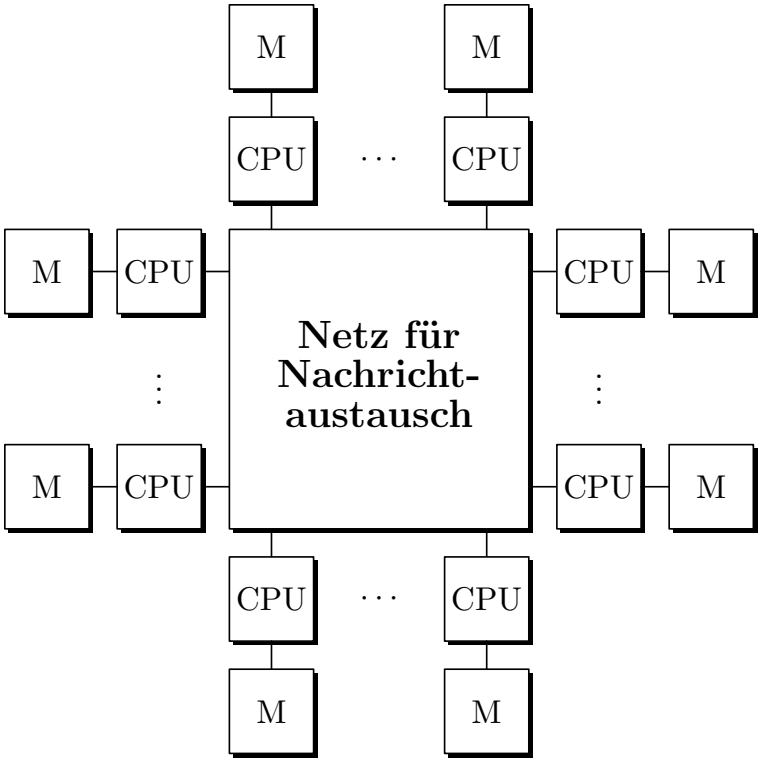


Abbildung 17.2: Netz für Nachrichtenaustausch

hinsichtlich der Entscheidung, welche Daten wo abgelegt werden.
Mehrrechnersysteme werden ebenfalls von verschiedenen Herstellern angeboten - Beispiele sind IBM SP/2, Intel/Sandia Option Red und Wisconsin COW.
Warum werden trotzdem Mehrrechnersysteme gebaut, wenn doch die Mehrprozessorsysteme viel einfacher und leichter programmierbar sind?
Große Mehrrechnersysteme lassen sich bei gleicher CPU- Anzahl einfacher und billiger bauen als Mehrprozessorsysteme. Dies liegt insbesondere an der Implementierung des von z.B. 100 CPUs gemeinsam genutzten Speichers.
Damit haben wir auf der einen Seite schwer baubare aber leicht programmierbare Mehrprozessorsysteme - auf der anderen Seite bei den Mehrrechnersytemen das Gegenteil.
Diese Beobachtung hat zu umfangreichen Bemühungen um die Konstruktion von Hybriden Systemen geführt, die sich sowohl leicht bauen als auch leicht programmieren lassen. Ein Ansatz der Konstruktion der Hybridsysteme besteht darin, moderne Rechner geschichtet zu entwickeln und den gemeinsamen Speicher auf einer beliebigen Schicht zu implementieren.

Als zweiter Ansatz wird aus der Kopplung von Mehrprozessorsystemen (Shared Memory Systems) und Mehrrechnersystemen (Distributed Memory Systems) die sogenannte Methode der **Distributed Shared Memory Systems (DSM)**, die Ende der 80er Jahre entwickelt wurde.
Dabei wird Mehrrechner-Hardware benutzt, und das Betriebssystem simuliert den gemeinsamen Speicher, indem es einen systemweiten virtuellen Adreßraum zur gemeinsamen Nutzung bereitstellt.

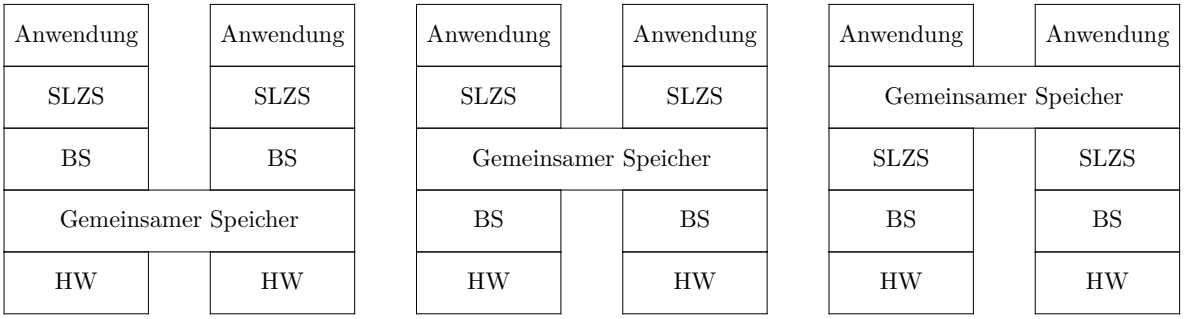


Abbildung 17.3: Hybrides System

Führt eine CPU eine LOAD- oder STORE-Operation auf Speicher aus, den sie nicht hat, so erfolgt ein automatischer Prozeduraufruf (Trap) zum Betriebssystem. Dieses sucht die Seite in seiner Seitentabelle und fordert von der CPU, die diese Seite hat eine Freigabe, um vom entfernten Speicher die Seite zu bearbeiten.

Verbindungsnetze
Sowohl bei Mehrrechner- als auch bei Mehrprozessorsystemen sind Verbindungsnetze erforderlich.
Verbindungen sind physische Kanäle, über die Bits fließen.
Sie können:

- elektrisch oder optisch arbeiten
- seriell oder parallel sein
- simplex (unidirektional), halbduplex (jeweils eine Richtung) oder Vollduplex (beide Richtungen gleichzeitig) arbeiten.

Jede Verbindung wird durch eine maximale Bandbreite (Bits pro Sekunde) charakterisiert.

Ein Switch ist ein Gerät mit mehreren Ein- und Ausgabeports. Kommt ein "Datenpaket" an einem Eingabeport eines Switches an, so kann aus einigen Bits des Pakets der Ausgabeport hergeleitet werden, über den das Paket weiter versendet wird.
Die Topologie eines Verbindungsnetzes bestimmt, wie die Verbindungen und Switches angeordnet werden. Topologien werden in der Regel als Graphen modelliert, wobei die Verbindungen als Kantenlinien und die Switches als Knotenpunkte dargestellt werden.
Im folgenden wollen wir verschiedene Topologien betrachten.

Taxonomie paralleler Computer

Die vielen Varianten von Parallelrechnern, die im Laufe der Jahre entstanden sind, müssen kategorisiert werden. Nach Flynn, 1972 gibt es folgende Taxonomie:

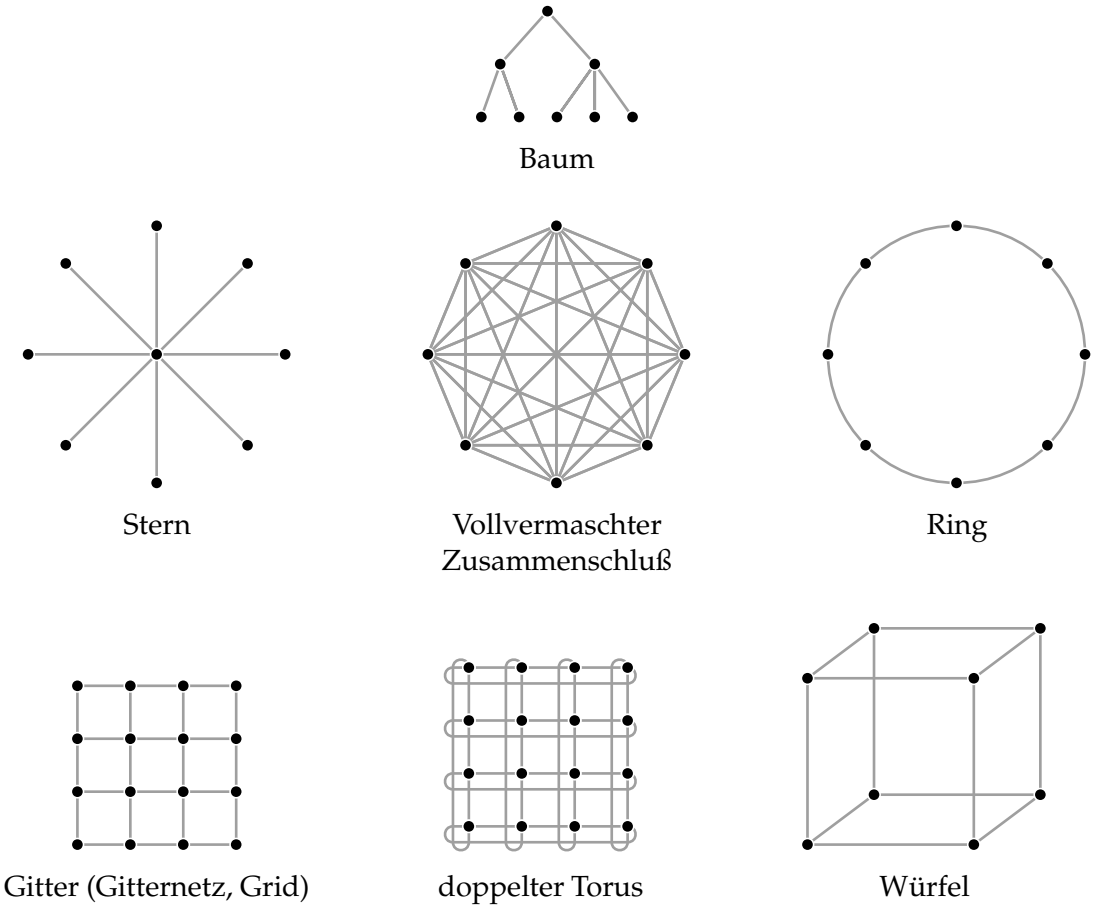


Abbildung 17.4: Topologien von Verbindungsnetzen

Instruktionsströme	Datenströme	Bezeichnung	Beispiele
1	1	SiSD	Klass. von Neumann Maschine
1	n	SiSD	Vektor-Supercomputer Array-Prozessor für wiss. Anwendungen
n	1	MISD	-
n	m	MISD	Mehrprozessor Mehrrechner

Instruktions- und Datenströme sind unabhängig. Daher gibt es vier Varianten. Pro Instruktionsstrom gibt es im System einen Programmzähler. Ein Datenstrom beschreibt die Operandenmenge. Daraus ergibt sich:

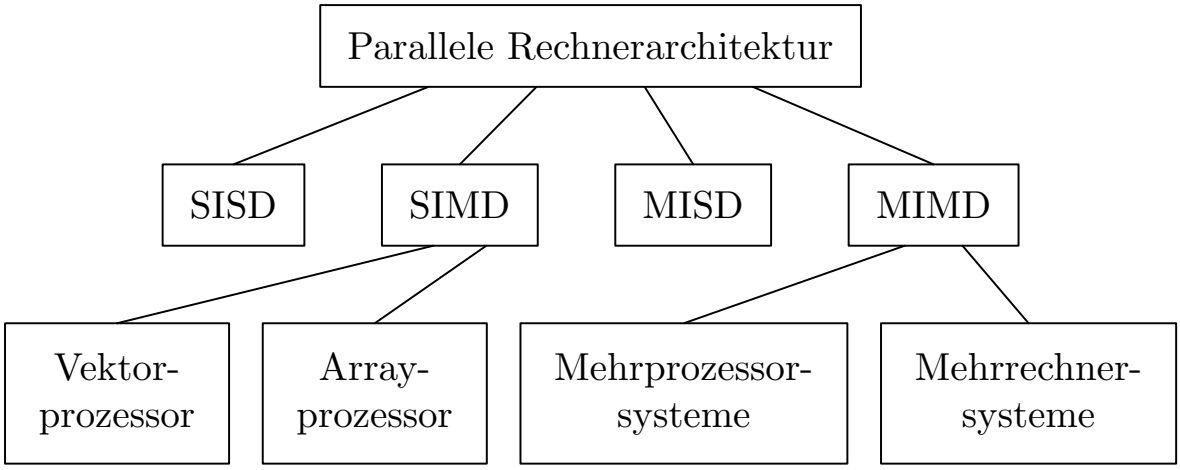


Abbildung 17.5: Parallele Rechnerarchitektur

17.2 SiMD-Computer

Single Instruction Single Data Computer werden zur Lösung rechenintensiver Probleme in Wissenschaft und Ingenieurwesen eingesetzt wenn regelmäßige Datenstrukturen zugrundliegen, z.B. Vektoren und Arrays.
Diese Maschinen zeichnen sich dadurch aus, dass eine einzige Steuereinheit Instruktionen einzeln ausführt, wobei jede Instruktion mit mehreren Datenelementen arbeitet.

Arrayprozessoren (Matrizenrechner)

Das Konzept wurde 1958 entwickelt. 10 Jare später wurde die erste Maschine, der ILLIAC IV tatsächlich gebaut und für die NASA eingesetzt.
Der Rechnertyp eignet sich für Anwendungen auf Matrizen (Array), wenn gleichartige Berechnungen zur selben Zeit auf viele verschiedene Datenmengen angewendet werden.
Ein Arrayprozessor (auch Array Computer) besteht aus einer großen Zahl identischer Prozessoren, die die gleiche Instruktionsabfolge auf verschiedenen Datenmengen anwenden.

Die ILLIAC IV bestand aus vier Quadranten, wobei jeder Quadrant ein quadratischer Gitter aus 8x8 Prozesser- und Speicherelementen enthält.

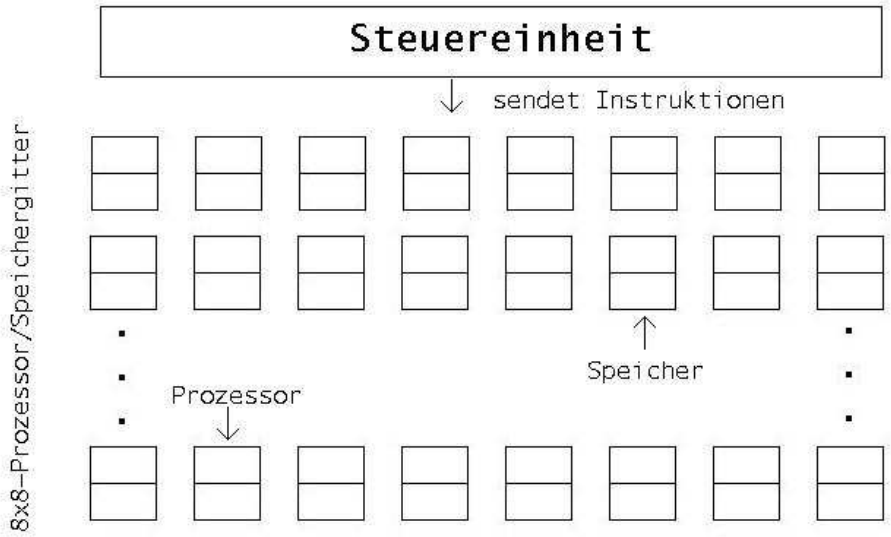


Abbildung 17.6: Arrayprozessor

Je eine Steuereinheit pro Quadrant gab die Instruktionen aus, die dann im Gleichschritt von allen Prozessoren ausgeführt wurden, wobei jeder Prozessor, der in einer Initialisierungsphase geladenen Daten aus dem eigenen Speicher bearbeitete.
Wegen mehrfacher Kostenüberschreitung wurde nur ein Quadrant jemals konstruiert, doch dieser erreichte eine Leistung von 50 Megaflops (Millionen Gleitkommaoperationen in der Stunde).

Obwohl alle Arrayprozessoren diesem allgemeinen Modell folgen, gibt es eine Reihe von Designvarianten.
Dies betrifft einfache bzw. komplexe Verarbeitungselemente (1-Bit-ALU bis 32-Bit-ALU) und auch die Verbindung der Verarbeitungselemente miteinander. Dabei sind quasi alle betrachteten Topologien (Stern, Baum, Ring, . . .) potentiell denkbar, wobei rechteckige Gitter besonders beliebt sind.
Die dritte Designfrage lautet, wieviel lokale Autonomie die Verarbeitungselemente haben. Bei dem beschriebenen Design bestimmt die Steuereinheit, welche Instruktion auszuführen ist. Bei vielen Arrayprozessoren kann aber jedes Verarbeitungselement selbst entscheiden, ob es eine Instruktion ausführen kann oder nicht.
Aus praktischer Sicht haben Arrayprozessoren eine ungewisse Zukunft.

Vektorprozessoren (Vektorrechner)

Kommerziell viel erfolgreicher ist der Vektorprozessor. Der Markt wurde jahrelang von einer Maschinenfamilie beherrscht, die 1976 mit der Cray-1 startete. Sie war von Seymour Cray für Cray Research (jetzt Teil von Silicon Graphics) entwickelt worden. Spätere Maschinen

waren die Modelle C90 und T90.
Eine Anwendung wäre eine Menge von Anweisungen wie:
a[i] = b[i] + c[i], wobei a, b und c Vektoren sind.
Eine mögliche SiMD-Architektur für solche Anwendungen wäre:

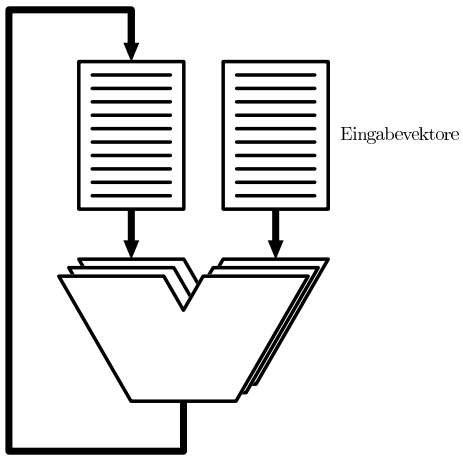


Abbildung 17.7: Steuereinheit

Diese Maschine nimmt zwei Vektoren mit je n Elementen als Eingabe und arbeitet mit den entsprechenden Elementen parallel mit Hilfe einer Vektor-ALU, die alle Elemente gleichzeitig behandeln kann.
Als Ergebnis produziert sie wieder einen Vektor. Die Ein- und Ausgabevektoren können im Speicher oder in speziellen Vektorregistern gespeichert werden.
In der Praxis werden nur wenige Supercomputer so gebaut wie in der Abbildung gezeigt. Dies hat unterschiedliche Gründe: ein Design mit 64 ALUs mit sehr hohen Geschwindigkeiten wäre auch für Supercomputer zu teuer. Deshalb wendet man normalerweise eine Kombination von Vektorverarbeitung und Pipelining an.
Abschließend noch eine Bemerkung:
Die Cray 1 war mit 80 MHz getaktet und hatte einen Arbeitsspeicher von 8 MByte (1976). Damals war sie der leistungsfähigste Computer der Welt. Heute gibt es keinen PC mehr mit einer derart niedrigen Taktrate und einem so kleinen Speicher. D.h. die Computerindustrie entwickelt sich enorm schnell weiter.

17.3 Mehrprozessoren mit gemeinsamem Speicher

Mehrprozessorsysteme sind bereits aus 17.1 bekannt als Rechner mit mehreren CPUs und einem einzigen Adreßraum, der für alle CPUs sichtbar ist.
Er führt eine Kopie des Betriebssystems mit einer Reihe von Tabellen aus. Insbesondere sind von Bedeutung die Tabellen, die verwalten, welche Speicherseiten belegt und welche frei sind. Oder blockiert ein Prozeß, so speichert die CPU seinen Zustand in den Betriebssystemtabellen und sucht gleichermaßen einen anderen Prozeß zum Ausführen. Wie alle Computer, braucht auch ein Mehrprozessor E/A-Geräte.
Hat jede CPU gleichrangingen Zugriff auf alle Speichermodule und alle E/A-Geräte und

gilt sie für das Betriebssystem als austauschbar durch eine der anderen CPUs, so wird das System als **Symetrischer Multi Prozessor** (SMP) bezeichnet.
Das Problem besteht in folgendem:
Viele Speichermodule halten jeweils einen Teil des physischen Speichers.
Mehrere CPUs können dabei versuchen, ein Speicherwort gleichzeitig zu lesen, während mehrere andere CPUs das gleiche Wort schreiben und ggf. sich die Reihenfolge der Nachrichtenzustellung ändert (z.B. durch Nachrichtenüberholung).
Und es sind ggf. mehrere Kopien von Speicherblöcken vorhanden (Caching).

Beispiel 17.1*:* CPU 0 schreibt Wert 1
CPU 1 schreibt Wert 2 in gleiches Wort
CPU 2 liest Wort und erhält Wert 1

Um unter diesen Umständen kein Chaos zu bekommen, muß die Software sich an bestimmte Regeln halten. Dazu braucht man Konsistenzmodelle. D.h. man vereinbart einen Vertrag, der die "Strenge" der Aktualität beschreibt.

17.4 Mehrrechnersysteme mit Nachrichtenaustausch

17.4.1 Massiv parallele Prozessorsysteme (MPP)

Dies sind riesige, sehr teure Supercomputer, wie z.B. Cray T3E und Intel/Sandia Option. Sie kosten im Millionenbereich.
Anfangs wurden MPPs im wissenschaftlichen Bereich eingesetzt, heute auch in kommerziellen Anwendungen.
Diese Rechner setzen Standard-CPUs als Pozessoren ein, insbesondere aus der Intel-Familie, SUN Ultra SPARC, IBM RS/6000 und DEC Alpha.
MPPs sind dadurch charakterisiert, dass sie ein herstellerspezifisches Verbindungsnetz mit hoher Leistung nutzen, um Nachrichten mit niedrigen Latenz und hoher Bandbreite auszutauschen.

17.4.2 Cluster of Workstations (COW)

Die zweite Art von Mehrrechnersystemen wird auch als COW bzw. Network of Workstations (NOW) bezeichnet (Andersen et. al. 1995).
Ein solches System setzt sich im typischen Fall aus ein paar hundert PC's oder Workstations zusammen die über eine handelsübliche Netzwerkkarte verbunden sind.
Im Vergleich zum MPP lässt sich ein COW leichter bauen, weil er gänzlich aus kommerziellen Komponenten an Ort und Stelle zusammengesetzt werden kann.
Diese Teile werden in großen Stückzahlen produziert, dadurch sind sie relativ preisgünstig.
Als COW-Varianten gibt es zentrale und dezentrale.
Ein zentraller COW ist ein Workstation- oder PC-Cluster, der sich i.d.R. in einem großen Schrank befindet. Die Baueinheit ist sehr kompakt um Stellfläche und Kabellänge zu reduzieren. Typischerweise sind sie homogen.
Dezentrale COW-Systeme bestehen aus Workstations oder PCs, die beliebig in einem Gebäude oder Campus verteilt sind. Sie sind meist heterogen und über ein LAN verbunden.

Index

Literaturverzeichnis

[1] **David A. Patterson & John L. Hennessy**
Computer Organization and Design: The Hardware/Software Interface
Morgan Kaufmann Publishers, 2nd Edition, 1997

[2] **Andrew S. Tanenbaum, James Goodman**
Computerarchitektur: Struktur, Konzepte, Grundlagen
Markt und Technik, 1999

[3] **Oberschelp, Vossen**
Rechnerarchitektur
Oldenbourg-Verlag, 1990