1 Einführung

1.1 Überblick "Grundlagen der EDV"

Begriffe:

- "Elektronische Datenverarbeitung"
- "Informatik"
- "Informationstechnologie"
- "Computer Science"

Information: Wissen über Sachverhalte, Zustände, Vorgänge, ...

Daten: Dargestellte (aufgezeichnete) Informationen zum Zwecke der *Verarbeitung* (auch: Schrift, ..., aber hier: *Digitale* Daten)

- Aus Informationen werden durch Erfassung Daten
- Aus Daten werden durch Interpretation (Zuordnung einer Bedeutung) Informationen

Teilbereiche der Computer Science: (ohne Anspruch auf Vollständigkeit und systematisch korrekte Gliederung!)

- Hardware
- Logik (auch: Beweisen, ...)
- Theoretische Informatik
  (Berechenbarkeitstheorie, Komplexitätstheorie, Automatentheorie, ...)
- Algorithmen und Datenstrukturen
- Software-Engineering
- Anwendungen: Betriebl. EDV, phys. Anwendungen, Schachspielen, Grafik und vieles andere mehr!
- Datenbanken
- Programmiersprachen und Compilerbau
- Betriebssysteme
- Netzwerk- und Kommunikationstechnik
- Echtzeitsysteme, Automatisierungstechnik, Signalverarbeitung

Lehrstoff dieses Gegenstandes für *beide* Jahre:

- Grundbegriffe
- Datendarstellung
- Logik & Schaltalgebra

- Hardware
- Programmiersprachen
- Betriebssysteme
- Algorithmen und Datenstrukturen

1.2 Elektronische Datenverarbeitung

Daten:

Definition siehe 1. Kapitel:
erfassen (maschinenlesbar machen) $\Longrightarrow$
speichern / verarbeiten / übertragen $\Longrightarrow$
ausgeben (menschlich lesbar machen)

Unterscheidung:
analog (stufenlos, “kontinuierlich”) — **digital** (in Schritten/Stufen, “diskret”)
Von lateinisch *“digitus”* (“der Finger”)
Analoguhr / Digitaluhr
Beispiele für Analogsignale, Digitalsignale
Analogrechner, Rechenschieber

Händische / mechanische / elektronische Datenverarbeitung:

Im Prinzip **kein** Unterschied!!!
Alles, was der Computer kann, kann man grundsätzlich auch händisch machen.
(umgekehrt nicht, siehe unten)

Vorteile elektronischer Datenverarbeitung:

- Hohe Geschwindigkeit
- Großer Speicherplatz
- Schnelle, weltweite Vernetzung
- Exakte, reproduzierbare Arbeitsweise (immer gleich richtig oder gleich falsch)

Damit sind Aufgaben lösbar, die händisch praktisch nicht durchführbar sind (Wettervorhersage, Flugbuchung, ...).

Aber:
Ein Computer kann nicht denken. Er hat keine Intelligenz, keine Kreativität, keine Intuition. Er tut genau das, was ihm sein Programm vorschreibt. (hat der Programmierer eine Situation nicht vorgesehen, so kann der Computer nicht von sich aus reagieren oder Entscheidungen treffen. Beispiele: Nicht an volle Platte oder vollen Speicher gedacht, nicht auf unsinnige Eingaben vorbereitet, nicht auf zu lange Zeilen geprüft, ...)

- “Künstliche Intelligenz”: Meist “regelbasierte” Systeme: Leiten aus bekannten Fakten durch Anwendung fixer Regeln neue Fakten her (nicht wirklich intelligent). Beispiel: Medizinische Expertensysteme.
- “Neuronale Netze”: Reine Signalverarbeitung, nicht einmal ein Programm: Bestimmte Eingangssignale werden zu bestimmten Ausgangssignalen verknüpft. Beispiel: Mustererkennung.

- Selbstlernende Programme: Art des Probierens / Lernens fix vorgegeben. Beispiel: Golf-Roboter (“wenn der Ball links vorbeifliegt, ziele beim nächsten Schuß mehr nach rechts!”).

Probleme der EDV:

- Datenschutz:
  - Persönliche oder personenbezogene Daten gelangen in unbefugte Hände (private Dokumente, finanzielle oder medizinische Daten, ...).
  - Personenbezogene Daten werden unbefugt verknüpft, d. h. aus der Kombination mehrerer, unabhängiger Datenbestände über dieselben Personen werden neue Erkenntnisse gewonnen.
  - Elektronische Kommunikation wird belauscht.

  $\Longrightarrow$ *Datenschutzgesetz (regelt u. a. Speicherung personenbezogener Daten), Fernmeldegesetz (regelt Abhören)*
- Komplexe EDV-Systeme sind unüberschaubar, unbeherrschbar, fehlerhaft (weil sie einfach zu groß sind, weil sie jahrelang gewachsenes Flickwerk sind, weil zentrale Entwickler gewechselt haben, weil sie nie im Realeinsatz getestet wurden, ...). Besonders heikle Beispiele: Waffensysteme, Kernkraftwerke.
- Abhängigkeit von der EDV: Computerausfall oder -fehler gefährdet Menschenleben oder Sachwerte (Flugzeuge, Steuerungen), lähmt die Handlungsfähigkeit (Verkehrssteuerung), bringt finanzielle Verluste (Firmen-EDV).

Unterschied Taschenrechner / Computer:

Simpler Taschenrechner: Programm fix in Hardware gegossen, nicht änderbar, nicht selbst programmierbar: Wenn ich dieselbe Aufgabe für hundert verschiedene Werte lösen will (z. B. Fläche eines Kreises), muß ich die komplette Berechnung hundert Mal eintippen.

Programmierbarer Taschenrechner: Ich kann selbst eine Formel / eine Berechnungsvorschrift / ein Programm eingeben und immer wieder ausführen: Bei wiederholten Berechnungen brauche ich nur mehr die Werte einzutippen und die gespeicherte Formel zu starten, nicht mehr jedesmal die komplette Berechnung.

Computer: Wie programmierbarer Taschenrechner, aber universeller:

- Bearbeitet Zahlen, Texte, Bilder, ...
- Die “Formel” kann Abfragen und Wiederholungen enthalten: Der Rechenablauf ändert sich abhängig von Eingabe und Zwischenergebnissen.

1.3 Geschichte der EDV

1000 vor Chr.	Erste “Kugerlrechner”
ab 1600	Einfache mechanische Rechner (von Blaise Pascal) für Addition & Subtraktion, nicht programmierbar
ab 1800	Programmierbare mechanische Rechner (von Charles Babbage) für höhere Mathematik, alle Merkmale eines Computers!
ab 1890	Lochkartensysteme (von Hermann Hollerith, später IBM) für Volkszählung, Steuereintreibung konnten zählen, sortieren usw., aber nicht programmierbar
1936–1941 1937–1944	“Z3” (von Konrad Zuse, Deutschland) sowie “Mark1” (von Howard Aiken, USA) elektromechanische Rechner aufgebaut aus Relais und Drehwählern, teils dual! Massenspeicher: Alte Kinofilme als Lochstreifen Geschwindigkeit: 5 Sekunden (!) pro Multiplikation
1943	“ENIAC” (USA) elektronischer Rechner aufgebaut aus Elektronenröhren Geschwindigkeit: 3 ms pro Multiplikation Röhrentechnik blieb bis 1960!
1957	“Mailüfterl” (Österreich!) erster *volltransistorisierter* Computer (ohne Röhren)
ab 1964	Verwendung von IC’s (integrierten Schaltkreisen): mehrere Transistorfunktionen auf einem einzigen Stück Halbleiter (Chip) statt einzelner Transistoren
ab 1965	erste Minicomputer, Beginn der Dezentralisierung
ab 1975	erster Mikroprozessor: “intel 8080” gesamte CPU auf einem einzigen Chip
ab 1975	erste Arbeitsplatz- und Homecomputer

Weitere historisch bedeutende, heute ausgestorbene Technologien:

Magnettrommel: Rotierender, magnetisierbarer Metallzylinder mit längs entlangbewegtem Lese- und Schreibkopf. Vorgänger der Magnetplatte.

Kernspeicher: Arbeitsspeicher (RAM) aus rund 2 mm großen, auf einem quadratischen Gitter von Drähten aufgefädelten, magnetisierbaren Ferrit-Ringen (1 Bit = 1 Ring) (bis etwa 1975).

Fernschreiber und Kugelkopf-Schreibmaschinen: Als Terminals (vor Entwicklung der Bildschirmterminals), oft kombiniert mit Lochstreifengeräten.

Lochkarte und Lochstreifen: Massenspeicher (bis etwa 1980!).

Spulenband: 12.5 mm breites, loses, auf Spulen aufgewickeltes Magnetband zur Massendatenspeicherung (bis etwa 1985).

Mikrofilm: (oder Lochkarten mit Mikrofilm-Einsatz) Alternative zur magnetischen Massendatenarchivierung (Vorteil: Menschlich lesbar!).

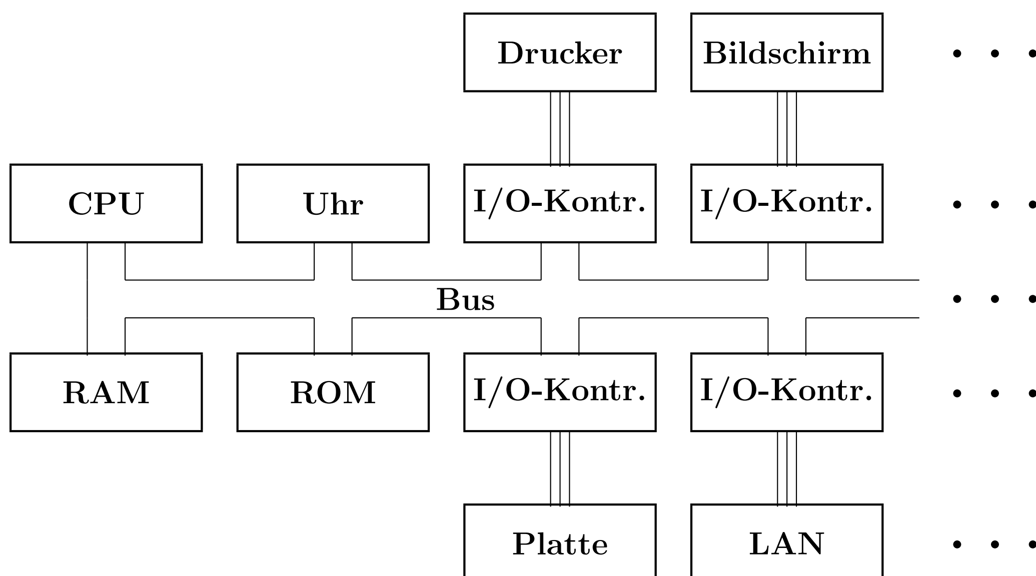
1.4 Hardware-Überblick

Wir unterscheiden:

Hardware: Die Geräte, "Zum Angreifen". Dumm (d. h. ohne Software nicht arbeitsfähig)!
Beispiele: Tastatur, Prozessor, Netzteil, CD-Laufwerk, ...

Software: Die Programme, nicht "Zum Angreifen" ("immateriell")
Software = "Arbeitsvorschrift" für Hardware, Software ist auf Hardware gespeichert
Beispiele: Windows (Betriebssystem), Word (Anwendung), Oracle (Datenbank), Grafikkarten-Treiber, ...
Sonderfall ***Firmware***: In Hardware "eingebrannte" Software
nicht oder schwer änderbar
Beispiele: BIOS, CD-Laufwerk-Firmware, ...

Grundsätzlicher Aufbau eines Rechners:



Prozessor (CPU, Central Processing Unit):

- Steuert alles andere
- Verarbeitet die Daten ("rechnet")

Arbeitsweise: Die CPU ...

1. Holt den nächsten Befehl aus dem Arbeitsspeicher
2. Holt die im Befehl angegebenen Daten aus dem Arbeitsspeicher
3. Führt die im Befehl angegebene Operation (z. B. +) auf den Daten aus
4. Schreibt das Ergebnis zurück in den Arbeitsspeicher

Wichtigste Bestandteile:

Rechenwerk (ALU, Arithmetic/Logic Unit): Addiert, vergleicht, ... Daten
Steuerwerk: Holt und dekodiert Befehle, steuert den Ablauf
Register: Sehr schnelle Zwischenspeicher für gerade bearbeitete Daten
(Operanden eines Befehls, Datenzeiger, Programmzeiger, ...)

Arbeitsspeicher (RAM, Random Access Memory): Speichert:

- Alle gerade in Ausführung befindlichen Programme
- Alle gerade in Bearbeitung befindlichen Daten

Les- und schreibbar, *Strom weg = Daten weg*

Direkt adressiert: Jedes einzelne Byte hat eine fortlaufende Adresse ("Hausnummer") und ist über diese direkt ansprechbar.
Bildlich: Arbeitsspeicher = viele Kästchen für je einen Buchstaben

Wichtig: Nicht verwechseln:

- Die ***Adresse*** einer Speicherzelle ist ihre "Hausnummer". Sie muß bei jedem Schreib- oder Lesezugriff auf den Speicher angegeben werden und legt fest, welche Speicherzelle gemeint ist. Die Adresse einer jeden Speicherzelle ist durch die Hardware festgelegt und unveränderlich.
- Der ***Inhalt*** einer Speicherzelle sind die dort gespeicherten Daten (1 Byte). Ein Schreibbefehl speichert die angegebenen Daten in der durch die angegebene Adresse festgelegten Speicherzelle (und ünerschreibt damit die bisher dort gespeicherten Daten), ein Lesezugriff liefert die in der durch die angegebene Adresse festgelegten Speicherzelle zuletzt geschriebenen Daten (die Daten bleiben dabei unverändert gespeichert). Nach dem Einschalten ist der Inhalt des RAM undefiniert, bis zum ersten Schreiben enthält es zufällige Daten.

Sonderfall *Cache:* Zwischenspeicher für häufig / zuletzt benutzte Daten:

- Befehls- und Datencache des Prozessors: Eigene Hardware im Prozessor
  Speichert Daten aus dem Arbeitsspeicher, kleiner und schneller als dieser
- Disk Cache: Bereich im Arbeitsspeicher oder eigener Chip auf der Plattensteuerung
  Speichert Daten von der Platte

Festwertspeicher (ROM, Read-Only Memory):
Vorprogrammiert, nur lesbar, hält ohne Strom, für Firmware / BIOS
Beim Einschalten: Initialisiert, macht Selbsttest, lädt Betriebssystem von der Platte

Technologien:

ROM: Maskenprogrammiert, unabänderlich
PROM: Ein Mal programmierbar ("Sicherungen", daher "brennen")
EPROM: Mit UV-Licht löschbar (nur komplett), neu programmierbar
EEPROM: Elektrisch löschbar und neu programmierbar ("Flash")

Zum ROM meist: Ganz kleines CMOS-RAM mit Batterie (für Konfigurationsdaten)

Bus: Verbindung zwischen allen Komponenten
Heute meist mehrere (PCI, AGP, RAM-Steckplätze, ...)

Enthält 3 Arten von Leitungen:

Datenleitungen: Übertragen die Daten (8/32/64/128 Stück)
Adressleitungen: Geben die Speicheradresse (oder den I/O-Controller) an, woher die Daten gelesen bzw. wohin sie geschrieben werden sollen (meist 32 Stück, reicht für 4 GB)
Steuerleitungen: Regeln, wer den Bus gerade benutzt, ob gelesen oder geschrieben wird, usw.

I/O-Controller, I/O-Devices:

- *Zum Menschen:*
  - ∗ Tastatur, Maus, Joystick, Touchscreen, Lightpen, Tablett
  - ∗ Display (Röhre / LCD / Plasma) (grafisch / alfanumerisch) (monochrom / färbig)
  - ∗ Drucker (Kettendrucker, Matrixdrucker, Laserdrucker, Tintendrucker, Thermo-Sublimationsdrucker, Thermodrucker)
  - ∗ Plotter (Flachbettplotter, Trommelplotter)
  - ∗ Scanner (Flachbettscanner, Trommelscanner)
  - ∗ Fotosatzbelichter, Fotobelichter, Filmbelichter
  - ∗ Audio (Mikrofon, Lautsprecher), Video (Kamera, TV-Tuner, Fernseher)
  - ∗ Leser:
    - ⋄ OCR-Leser (Optical Character Recognition) (Erlagscheine)
    - ⋄ Barcode-Leser (Supermarkt-Kasse)
    - ⋄ Markierungsleser (Lottoannahmeautomat)
  - ∗ Fingerabdruck- oder Iris-Scanner
- *Massenspeicher:* (bitte Technologien merken!)
  - ∗ Festplatte, Wechselplatte
  - ∗ Floppy
  - ∗ CD, CD-R, CD-RW, DVD, MO (Magneto-Optical)
  - ∗ Bandlaufwerke
  - ∗ Flash-Cards, Memory-Sticks, . . .

  Massenspeicher-Schnittstellen:

  - ∗ IDE
  - ∗ SCSI
  - ∗ FC (Fibre Channel)
- *Zu anderen Computern usw.:*
  - ∗ Serielle und parallele Schnittstelle
    (PS/2-Tastatur- und Maus-Stecker sind auch serielle Schnittstellen)
  - ∗ Modem, ISDN, ADSL
  - ∗ LAN (Ethernet, Token Ring, Funk, ATM, . . .)
  - ∗ USB, FireWire, IRDA
  - ∗ IEEE 488 (Meßgeräte, . . .)
  - ∗ Digital- und Analog-I/O (Motoren, Sensoren, Steuerungen, . . .)

Speichergröße:

Einheiten für Speichergröße:

1 Bit: 0/1
1 Byte: 8 Bits = 1 Buchstabe

1 Kilobyte (KB)		2^{10} Byte (1024)
1 Megabyte (MB)	2^{10} KB	2^{20} Byte $(1024 * 1024)$
1 Gigabyte (GB)	2^{10} MB	2^{30} Byte $(1024 * 1024 * 1024)$
1 Terabyte (TB)	2^{10} GB	2^{40} Byte $(1024 * 1024 * 1024 * 1024)$

Firmen rechnen oft mit 1000 / 1000000 / 1000000000 statt 1024 / . . . !!!

Speichereigenschaften: (Jahr 2001)

Speicher	Größe	Zugriffszeit	Transferrate
CPU-Register	einige 100 Byte	unter 1 ns	über 10 GB/s
Prozessor-Cache	einige 100 KB	1–3 ns	5–10 GB/s
RAM	einige 100 MB	20–50 ns	1–3 GB/s
Platte	max. 100 GB	8–20 ms	20–50 MB/s
Floppy	1.44 MB	0.5 s	100 KB/s
CD	700 MB	100-200 ms	1–5 MB/s
Band	max. 200 GB	einige Minuten	10–30 MB/s

Preis pro Byte sinkt von oben nach unten!!!

1.5 Software-Überblick

1.5.1 Das Schalenmodell

Hardware — Treiber — Betriebssystem — andere Systemsoftware — Anwendungssoftware

Idee:

- Jede Schicht nutzt nur Funktionen der unmittelbar darunterliegenden Schicht.
- Jede Schicht stellt der darüberliegenden Schicht bestimmte Funktionen zur Verfügung.

Sinn:

- Modularisierung
- "Abstraktion", Verstecken der Details

Beispiele (bitte selbst dazunotieren!):

- Schreiben einer Zahl in eine Datei
- Anzeigen von Text in einem Fenster

1.5.2 Systemsoftware

Treiber:
Hardware-spezifische Teile des Betriebssystems für jede einzelne Hardware-Komponente
Meist vom Hardware-Hersteller, nicht Betriebssystem-Hersteller
"Verstecken" die speziellen Eigenschaften der jeweiligen Hardware,
setzen allgemeine Operationen auf hardware-spezifische Operationen um
(z. B. die Handhabung von SCSI- oder IDE-Interface, oder die Befehle des Grafikchips)

Betriebssystem: Betriebssystem im engeren Sinn = ***Kernel***
Aufgaben:

- Initialisiert das System
- Verwaltet die Hardware, führt alle I/O-Operationen durch
- Verteilt die Hardware (CPU, RAM, Plattenplatz, ...) zwischen den Programmen und Benutzern, schützt Programme und Benutzer voreinander
- Startet alle anderen Programme

Der Kernel darf bestimmte Prozessorbefehle (z. B. zur Speicherverwaltung) ausführen, die alle anderen Programme nicht ausführen dürfen!

Der Kernel stellt allen anderen Programmen Systemfunktionen zur Verfügung, ist aber für den Benutzer nicht direkt sicht- oder greifbar.

Beispiele: Windows, Mac OS, Linux, OS/390

(Shared) Libraries:
Sammlungen immer wieder benötigter, von mehreren Programmen benutzter Funktionen: "Berechne Wurzel", "Vergleiche 2 Strings", "Schreibe Text in Fenster", ...

Benutzeroberfläche: Zur Interaktion mit dem Benutzer:

Textbasiert, Kommandozeilen-orientiert: DOS oder DOS-Fenster unter Windows, Shell unter Unix/Linux
Grafisch, Fenster-orientiert: Windows, X Window System unter Linux

Unter Windows Teil des Betriebssystems, unter Linux sauber getrennt

Hilfsprogramme:
... zur Systemkonfiguration und -administration
... zur Datenverwaltung (Windows-Explorer, Backup-Programm, ...)

Softwareentwicklungswerkzeuge:
Zum Schreiben und Testen neuer Programme
Compiler (oder Interpreter), Debugger, Versionsverwaltung, ...

1.5.3 Anwendungssoftware

Löst die Aufgabenstellungen des Benutzers

Standard-Software: "fertiges Packerl aus dem Regal"
Individual-Software: Auf Auftrag eigens angefertigt $\Longrightarrow$ Teurer!!!

Beispiele: Bürosoftware ("Word"), technische Software ("AutoCAD"), Datenbanken ("Oracle"), betriebliche Software ("SAP"), Spiele, ...

1.5.4 Die rechtliche Seite

Software-Entwicklung kostet viel Geld
Dieses Geld muß mit der Software wieder verdient werden
Die EDV-Branche macht viel mehr Umsatz mit Software als mit Hardware!

Software ist geistiges Eigentum des Entwicklers
unterliegt dem Urheberrecht (Copyright, wie Bücher, Musik, ...) und Patenten
gegen Bezahlung (einmalig oder laufend) erwirbt man ein durch den Lizenzvertrag geregeltes Nutzungsrecht (nicht die Software selbst!)

Sonderfall ***Shareware:***
Darf man kostenlos aus dem Internet laden und ausprobieren
zahlt man erst bei Gefallen (freiwillig oder wegen Ablaufdatum, Demoversion, ...)

Sonderfall ***Freeware:*** 3 mögliche Bedeutungen von "frei":

Frei von Kosten = Gratis: (z. B. Microsoft Internet Explorer) Meist Marketing-Strategie!
Frei von Rechten = ohne Lizenz nutzbar:
Der Autor verzichtet auf sein Copyright, jeder darf mit der Software machen, was er will
Frei von Geheimnissen = "Open Source":

- Der *Quelltext* der Software ist öffentlich verfügbar, für jeden einsehbar (bei kommerziellen Produkten ist der Quelltext das wichtigste Firmengeheimnis!)
- Das gesamte "Know How" des Programms steht damit der Allgemeinheit zur Verfügung
- *Nicht* frei von Copyright / Lizenzbestimmungen (typischerweise *GPL = "GNU Public License"*, verbietet z. B. Geheimhaltung, erlaubt Weiterentwicklung, Weiterverwendung)
- *Nicht* notwendigerweise kostenlos (aber meistens schon)
- Beispiele: Linux, Apache, GNU C Compiler, ...

2 Datendarstellung

2.1 Das Dualsystem

2.1.1 Codierung

Codierung: Hier: "Maschinengerechte" Darstellung von Daten.
Code: Vorschrift für eine ein-eindeutige (bijektive, umkehrbare) Zuordnung (Darstellung, Codierung) der Zeichen eines Zeichenvorrates zu den Zeichen eines anderen Zeichenvorrates.
Beispiele:

Morsealfabet: Buchstaben und Ziffern ⟶ Punkte und Striche
Blindenschrift: Buchstaben und Ziffern ⟶ fühlbare Punkte in einem Raster
ASCII-Code: Buchstaben und Ziffern ⟶ Zahlen 0–127

2.1.2 Zahlensysteme

Additionssysteme: *Wert einer Zahl = Summe der Werte ihrer Ziffern*
(*ohne* Rücksicht auf deren Position!)
Beispiel: Das römische Zahlensystem ⟹ *unpraktisch!*
Stellenwertsystem: *Wert einer Zahl = Summe von*
(Wert einer Ziffer) ∗ (Wert ihrer Position, "Stellenwert")
Beispiel: Unser Zehnersystem (= "Dezimalsystem")!

2.1.3 p-adische Systeme

p-adisches System = Stellenwertsystem zur "Basis" p
(p ... positive, ganze Zahl, $p > 1$)

- Die Stellenwerte sind ganzzahlige Potenzen von p, d. h. der Wert jeder Stelle ist das p-fache des Wertes der Stelle rechts daneben.
- Man braucht p verschiedene Ziffern mit den ganzzahligen Werten 0 bis $p - 1$. Zählt man zur größten Ziffer 1 dazu (d. h. erreicht man p), ergibt sich ein Übertrag bzw. eine zweistellige Zahl.

Berechnung des Wertes einer n-stelligen Zahl $x = a_{n-1}a_{n-2}\cdots a_1a_0$ (a_i = die i-te Ziffer, von 0 weg von rechts numeriert):

$$x = a_0 * p^0 + a_1 * p^1 + \cdots + a_{n-1} * p^{n-1} = \sum_{i=0}^{n-1}(a_i * p^i)$$

Für eine Kommazahl mit $n + m$ Stellen $x = a_{n-1}a_{n-2}\cdots a_1a_0.a_{-1}a_{-2}\cdots a_{-(m-1)}a_{-m}$:

$$x = a_{-m} * p^{-m} + \cdots + a_{-1} * p^{-1} + a_0 * p^0 + a_1 * p^1 + \cdots + a_{n-1} * p^{n-1} = \sum_{i=-m}^{n-1}(a_i * p^i)$$

(merke: $10^{-1} = \frac{1}{10^1} = \frac{1}{10}$, $10^{-2} = \frac{1}{10^2} = \frac{1}{100}$, ...)
Die Darstellung ist eindeutig, jede Zahl hat genau eine Darstellung!

2.1.4 Das Dualsystem

Dualsystem = p-adisches System zur Basis 2 ($p = 2$)
Nur 2 Ziffern: 0 und 1
Spannung hoch / niedrig, Loch / kein Loch, Licht / kein Licht, N–S magnetisiert / S–N magnetisiert (bitte nicht: Strom / kein Strom: Stimmt technisch nicht mehr!)
Daher

$$x = a_0 * 2^0 + a_1 * 2^1 + \cdots + a_{n-1} * 2^{n-1} = \sum_{i=0}^{n-1}(a_i * 2^i)$$

oder

$$x = a_{-m} * 2^{-m} + \cdots + a_{-1} * 2^{-1} + a_0 * 2^0 + a_1 * 2^1 + \cdots + a_{n-1} * 2^{n-1} = \sum_{i=-m}^{n-1}(a_i * 2^i)$$

2^0	1	2^4	16	2^8	256	2^{12}	4096	2^{16}	65532
2^1	2	2^5	32	2^9	512	2^{13}	8192	2^{20}	1048576
2^2	4	2^6	64	2^{10}	1024	2^{14}	16384	2^{32}	4294967296
2^3	8	2^7	128	2^{11}	2048	2^{15}	32768		

Dualzahl = Binärzahl (anderer Name für dasselbe Ding!)
Kennzeichnung (Unterscheidung von Dualzahl und Dezimalzahl) mit Index:

$1100_{\mathbf{2}}$	Dualzahl
$1100_{\mathbf{10}}$	Dezimalzahl
$1100_{\mathbf{16}}$	Hexadezimalzahl (Zahl im p-adischen System zur Basis 16)

2.1.5 Umwandlungen

Dezimal $\longrightarrow$ Dual:

- Dezimalzahl immer wieder durch 2 dividieren, bis Ergebnis 0 wird.
- Reste von rechts nach links anschreiben = Dualzahl

Beispiel:
$11/2 = 5$, Rest 1: 1
$5/2 = 2$, Rest 1: 11
$2/2 = 1$, Rest 0: 011
$1/2 = 0$, Rest 1: 1011
Ergebnis: $11_{10} = 1011_2$

Dezimal $\longrightarrow$ Dual, Nachkommateil:

- Nachkommateil der Dezimalzahl immer wieder mit 2 multiplizieren, bis Ergebnis 0 wird.
- Überträge von links nach rechts anschreiben = Nachkommateil der Dualzahl

Beispiel:
$0.6875 * 2 = 1.375$, Übertrag 1: .1
$0.375 * 2 = 0.75$, Übertrag 0: .10
$0.75 * 2 = 1.5$, Übertrag 1: .101
$0.5 * 2 = 1.0$, Übertrag 1: .1011
Ergebnis: $0.6875_{10} = 0.1011_2$
Achtung: Nichtperiodische Dezimalzahlen können periodische Binärzahlen ergeben!

Dual $\longrightarrow$ Dezimal: (Vor- und Nachkommateil)
In "Reihenschreibweise" anschreiben und ausrechnen (Tabelle verwenden!)[1].

2.1.6 Rechenregeln

Addition: Spaltenweise von rechts nach links:

- Nur 0: Ergebnis 0.
- Eine 1: Ergebnis 1.
- Zwei 1: Ergebnis 0, Übertrag 1.
- Drei 1 (zwei 1 plus Übertrag): Ergebnis 1, Übertrag 1.

Subtraktion: Spaltenweise von rechts nach links:

- $0 - 0$: Ergebnis 0.
- $1 - 0$: Ergebnis 1.
- $1 - 1$ oder $1 - (0 + \text{Ü})$: Ergebnis 0.
- $0 - 1$ oder $0 - (0 + \text{Ü})$: Ergebnis 1, Übertrag 1.
- $0 - (1 + \text{Ü})$: Ergebnis 0, Übertrag 1.
- $1 - (1 + \text{Ü})$: Ergebnis 1, Übertrag 1.

---

[1] Die Reihenschreibweise (Multiplikation der Ziffern mit ihren Stellenwerten) ist sinnvoll, wenn man die Stellenwerte auswendig weiß oder einer Tabelle entnehmen kann.

Ist das nicht der Fall, so ist die ***Horner-Methode*** günstiger, weil sie das Ergebnis mit ebensovielen Multiplikationen und Additionen, aber ohne Potenzrechnungen liefert. Sie beruht auf folgender Formel:

$$\sum_{i=0}^{n-1}(a_i * p^i) = a_{n-1} * p^{n-1} + a_{n-2} * p^{n-2} + \cdots + a_1 * p^1 + a_0 * p^0 = ((\cdots(a_{n-1} * p + a_{n-2}) * p + \cdots) * p + a_1) * p + a_0$$

Die Vorgangsweise ist daher folgende: Man beginnt mit der Ziffer ganz links, multipliziert diese mit der Basis, addiert die zweitlinkeste Ziffer dazu, multipliziert das Zwischenergebnis wieder mit der Basis, addiert die nächste Ziffer usw., bis man zuletzt die äußerst rechte Ziffer dazuaddiert und damit das Ergebnis erhält.

Multiplikation: Im Prinzip wie dezimal: Linken Operanden mit Ziffern des rechten Operanden multiplizieren, Ergebnisse entsprechend stellenverschoben untereinanderschreiben, am Schluß aufsummieren.
Vereinfachung: Man braucht nie eine echte Multiplikation: Ist die Ziffer rechts 0, ist die ganze Zeile 0. Ist die Ziffer rechts 1, ist die Zeile gleich der linken Zahl.

Division: Im Prinzip wie dezimal: Rechten Operanden immer wieder um eine Stelle verschieben und von linkem Operanden bzw. Rest abziehen probieren.
Vereinfachung: Man braucht nie ein "Wie oft geht ...": Ist der rechte Operand größer als der linke Operand oder Rest, ist die nächste Ziffer des Ergebnisses 0. Ist er gleich oder kleiner, wird er subtrahiert, und es kommt eine 1 ins Ergebnis.

2.1.7 Vor- und Nachteile des Dualsystems

\+ Die physikalische Darstellung von 0 und 1 ist technisch viel einfacher. Die Darstellung von 10 verschiedenen Werten pro Speicherstelle oder Datenleitung würde einen viel höheren technischen Aufwand erfordern!
\+ Die Rechenregeln sind viel einfacher (das kleine Einmaleins ist wirklich klein!).
– Eine Dualzahl erfordert mehr als drei Mal so viele Stellen wie die entsprechende Dezimalzahl. Für den Menschen zu unübersichtlich und fehleranfällig!
– Bei Zahlen mit periodischem Nachkommateil gibt es Rundungsverluste bei der Umrechnung.

2.1.8 Speicherung

Meist in 1, 2, 4 oder 8 Bytes (8, 16, 32 oder 64 Bits) gespeichert.
Achtung auf "Big Endian" / "Little Endian":
Legt die Reihenfolge fest, in der die Bytes einer 2, 4 oder 8 Byte langen Zahl im Hauptspeicher gespeichert werden:

- ***"Big Endian":*** So, wie man die Zahl schreibt: Das höchstwertige Byte kommt zuerst, d. h. in die Speicherstelle mit der niedrigsten Adresse.
- ***"Little Endian":*** Genau verkehrtherum, d. h. das niederwertigste, am weitesten rechts stehende Byte kommt in die erste Speicherstelle, das höchstwertige in die letzte.

Beispiel: Speichere die 4-Byte-Zahl $\texttt{78ABCDEF}_{16}$ an Adresse $\texttt{5010}_{16}$:

Adresse des Bytes	5010	5011	5012	5013
Inhalt bei "Big-Endian"	78	AB	CD	EF
Inhalt bei "Little-Endian"	EF	CD	AB	78

Wie eine Zahl im Speicher abgelegt wird, ist durch den Prozessor vorgegeben: Die Prozessoren der Pentium- und Alpha-Familien speichern aus mehreren Bytes bestehende Zahlen Little-Endian, Apple Mac's, IBM Großrechner usw. verwenden Big-Endian. Auch am Internet werden Zahlen Big-Endian verschickt: Das höchstwertige Byte fließt zuerst über die Leitung.

Begriffe:

- ***MSB:*** Most Significant Bit/Byte: Höchstwertiges Bit/Byte (ganz links).
- ***LSB:*** Least Significant Bit/Byte: Niederwertigstes Bit/Byte (ganz rechts).

2.2 Oktalzahlen, Hexzahlen, negative Zahlen

2.2.1 Das Oktalsystem

Oktalsystem = Stellenwertsystem zur Basis 8:
Ziffern 0–7, Stellenwerte 1, 8, $8^2 = 64$, $8^3 = 512$, ...

Praktische Abkürzung für Binärzahlen: Je drei Bits (von rechts beginnend!) werden zu einer Oktalziffer zusammengefaßt. Beispiel:

(00)1	111	000	011	100	110
1	7	0	3	4	6

Beispiele für Verwendung: (heute schon eher selten)

- Eine der möglichen Zahlendarstellungen in C (historisch bedingt): Zahlen mit führender `0` werden als Oktalzahl interpretiert!
- File-Berechtigungen in Unix/Linux (je 3 Bits für 3 Arten von Benutzern = 9 Bits = dreistellige Oktalzahl).

2.2.2 Das Hexadezimalsystem

Hexadezimalsystem = Stellenwertsystem zur Basis 16:
Ziffern 0123456789ABCDEF (oder ... abcdef), Stellenwerte 1, 16, $16^2 = 256$, $16^3 = 4096$, ...

Noch praktischere Abkürzung für Binärzahlen, weil sie mit Byte-Grenzen zusammenpaßt: Je vier Bits werden zu einer Hex-Ziffer zusammengefaßt, ein Byte entspricht daher genau einer zweistelligen Hexzahl, für 2 Bytes braucht man 4 Hexziffern, für 4 werden es 8 usw.

Dezimal	Dual	Hex	Dezimal	Dual	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Beispiele für Verwendung:

- *Adressen* (Zeiger auf Speicherstellen im Hauptspeicher) werden üblicherweise hexadezimal eingegeben und angezeigt.
- Zahlen, die *Bitmuster* darstellen sollen, werden ebenfalls üblicherweise hexadezimal angegeben. Bei `2147483647` kann man nicht erkennen, daß es sich um die 32 Bit Binärzahl handelt, bei der das vorderste Bit 0 und alle anderen Bits 1 sind, bei `7FFFFFFF` sieht man das auf einen Blick.

Umwandlung Dezimal $\longrightarrow$ Hexadezimal:
Fortlaufende Division durch 16, bis das Ergebnis 0 wird. Reste als Hexziffer von rechts nach links anschreiben.

Umwandlung Hexadezimal $\longrightarrow$ Dezimal:
Hexzahl in Reihenschreibweise (Summe aus Ziffernwert mal Stellenwert) anschreiben und ausrechnen (oder mit dem Horner-Verfahren).

Addition von Hexadezimalzahlen:
Ziffernwerte spaltenweise addieren[2]:

- Summe < 16: Summe als Hexziffer anschreiben.
- Summe ≥ 16: Summe $- 16$ als Hexziffer anschreiben, 1 Übertrag.

Subtraktion von Hexadezimalzahlen:
Ziffernwerte spaltenweise subtrahieren:

- Oben $\geq$ Unten: Differenz als Hexziffer anschreiben.
- Oben $<$ Unten: Differenz $+ 16$ als Hexziffer anschreiben, 1 Übertrag.

2.2.3 Ganze Zahlen mit Vorzeichen

Begriffe:

- ***Integer:*** Ganze Zahl (mit oder ohne Vorzeichen bleibt offen!).
- ***Unsigned Integer:*** Vorzeichenlose ganze Zahl.
- ***Signed Integer:*** Ganze Zahl mit Vorzeichen (siehe unten).

3 verschiedene Methoden:

Sign / Magnitude		**One's Complement**		**Two's Complement**	
Vorzeichen und Absolutwert		*"–" = Bitweise Inversion*		*"–" = Bitweise Inversion + 1*	
127	01111111	127	01111111	127	01111111
126	01111110	126	01111110	126	01111110
...		...		...	
1	00000001	1	00000001	1	00000001
0	00000000	0	00000000	0	00000000
−0	10000000	−0	11111111	−1	11111111
−1	10000001	−1	11111110	−2	11111110
...		...		...	
−126	11111110	−126	10000001	−127	10000001
−127	11111111	−127	10000000	−128	10000000

*Heute wird nur mehr **Two's Complement** verwendet, weil es am einfachsten zum Rechnen ist:*
Man braucht keine eigenen Operationen für Addition und Subtraktion von als Two's Complement dargestellten Signed Integers, es funktionieren die normalen Operationen für Unsigned Integers

[2] Bei mehreren zu addierenden Zahlen und beliebiger Basis p gilt allgemein: Summe aller Ziffernwerte einer Spalte aufaddieren und durch die Basis p dividieren: Das Ergebnis der Division ist der Übertrag in die nächste Spalte, und ihr Rest (als Ziffer im jeweiligen Zahlensystem geschrieben) ist das Ergebnis in dieser Spalte.

(gleicher Schaltkreis!)[3]. Alle anderen Darstellungen brauchen eigene Rechenoperationen für Signed Integers.

Merkhilfe für Two's Complement Signed Integers:

- Die "obere Hälfte" der gleichlangen unsigned Integers wird durch Subtraktion von 2^n (n = Anzahl der Bits) in den negativen Bereich "nach unten verschoben": Bei 8-Bit-Zahlen werden aus 128 bis 255 durch Subtraktion von 256 die Werte -128 bis -1.
- Stellt daher eine mit 1 beginnende Binärzahl als unsigned Integer betrachtet die positive Zahl z dar, so stellt dieselbe Binärzahl als signed Integer betrachtet die negative Zahl $z - 2^n$ dar (und umgekehrt).
- Man kann Two's Complement signed Integers auch als ein Stellenwert-Zahlensystem deuten, bei dem die vorderste Stelle nicht wie üblich den Stellenwert 2^{n-1}, sondern -2^{n-1} (also einen negativen Stellenwert!) hat.

Achtung:
Two's Complement signed Integers der Länge n gehen von -2^{n-1} bis $2^{n-1} - 1$:

Länge	Größte Unsigned-Zahl	Kleinste Signed-Zahl	Größte Signed-Zahl
8	255	−128	127
16	65535	−32768	32767
32	4294967295	−2147483648	2147483647

Die kleinste negative Zahl hat daher kein positives Gegenstück: $-(-128)$ *ergibt wieder* -128*!*
(dafür gibt es nur eine 0, nicht getrennte +0 und −0: −0 ergibt wieder 0)

Umwandlung negative Dezimalzahl $\longrightarrow$ *Two's Complement:* 2 Möglichkeiten:

- Absolutwert normal umwandeln, bitweise invertieren, 1 dazuzählen.
- Absolutwert von 2^n dezimal abziehen, Ergebnis normal umwandeln.

Umwandlung negatives Two's Complement $\longrightarrow$ *Dezimalzahl:* 2 Möglichkeiten:

- Bitweise invertieren, 1 dazuzählen, wie positive Binärzahl umwandeln, − davor.
- Umwandeln, als ob es vorzeichenlose Binärzahl wäre, vom Ergebnis 2^n abziehen.

Achtung:
Computer rechnen mit ganzen Zahlen fast immer ohne Berücksichtigung eines Überlaufes, ein Übertrag beim vordersten Bit wird einfach ignoriert:

- Addiert man zwei positive Zahlen, deren Summe s größer als $2^{n-1} - 1$ ist, so wird als Ergebnis $s - 2^n$ (d. h. eine negative Zahl!) gespeichert.
- Addiert man zwei negative Zahlen, deren Summe s kleiner als -2^{n-1} ist, so wird als Ergebnis $s + 2^n$ (d. h. eine positive Zahl!) gespeichert (analog für Subtraktionen usw.).

2.3 Andere Datentypen

2.3.1 BCD-Zahlen

BCD = Binary Coded Decimal (auch: "Packed Decimal")

[3] Für Multiplikation und Division sind hingegen eigene Befehle und zusätzliche Schaltungen notwendig.

Die Zahl wird als *Dezimalzahl* gespeichert, wobei jede Ziffer einzeln als Dualzahl `0000` bis `1001` codiert in 4 Bits abgelegt wird (d. h. 2 Dezimalziffern pro Byte): Die Speicherdarstellung einer BCD-Zahl angeschrieben als Folge von Hex-Ziffern entspricht der ursprünglichen Dezimalzahl. Die Dezimalziffern behalten also ihren Zehner-Stellenwert. Das Vorzeichen wird eigens abgespeichert.

Die Konvertierung zwischen Dezimalzahlen und BCD-Zahlen ist sehr einfach, aber dafür ist das Rechnen mit BCD-Zahlen für einen Computer wesentlich komplizierter als mit Binärzahlen (außerdem brauchen BCD-Zahlen mehr Platz als Binärzahlen).

BCD-Zahlen werden vor allem in kaufmännischen Anwendungen und hier vor allem für Fixkommazahlen (d. h. für Zahlen mit einer fixen Anzahl von Nachkommastellen, z. B. für Geldbeträge) verwendet: Dadurch umgeht man das Problem, daß aus nichtperiodischen Dezimalzahlen periodische und damit ungenaue Binärzahlen werden können.

2.3.2 Gleitkommazahlen

Der Physiker schreibt z. B.: $0,7255 * 10^{13}$
("wissenschaftliche Darstellung", "Exponentenschreibweise")

Allgemein kann jede Dezimalzahl x wie folgt geschrieben werden:

$$x = m * 10^e$$

m heißt Mantisse, e heißt Exponent ("wie viele Stellen muß ich das Komma verschieben?").

Weiters kann man eine solche Zahl immer *normalisieren*, sodaß gilt:
$-1 < m \leq -0,1$ oder $0,1 \leq m < 1$
(d. h. eine 0 vor dem Komma, eine von 0 verschiedene Zahl hinter dem Komma)
Normalisierte Gleitkommazahlen sind eindeutig (gleiche Werte haben gleiche Darstellung und umgekehrt), nicht normalisierte hingegen nicht (der gleiche Wert läßt sich auf mehrere Arten schreiben!).

Vorteile:

- Der darstellbare Zahlenbereich ist im Vergleich zu einer Festkommadarstellung gleicher Länge riesig:
  $-10^{\hat{e}} \cdots 10^{\hat{e}}$ (mit $\hat{e}$ ... größtes erlaubtes e)
- Die Genauigkeit entspricht immer der Anzahl der Stellen von m, unabhängig von der Größe der Zahl.

Man kann auch jede Dualzahl x in dieser Form darstellen:

$$x = m * 2^e$$

(wobei m und e ebenfalls Dualzahlen sind)

Es gab früher sehr viele Darstellungen basierend auf dieser Idee, die sich in Details unterschieden: Wie wird normalisiert? Wie werden negative Zahlen / negative Exponenten dargestellt? Echte Binärzahlen oder BCD-Darstellung für m und e? Wie wird 0, Überlauf und Unterlauf (heute: "Infinity" und "*NaN*" = "Not a Number") dargestellt?

Heute hat sich (außer bei IBM Großrechnern) weitgehend der Standard der ***IEEE*** ("Institution of Electrical and Electronical Engineers", sprich "Ei triple iee") durchgesetzt:

- Es wird Sign/Magnitude-Darstellung verwendet (ein Vorzeichenbit plus Darstellung des Absolutbetrags).
- m und e werden als echte, positive Binärzahlen dargestellt.

- Nach dem Vorzeichen kommt zuerst e, dann m.
- Es wird so normalisiert, daß immer genau eine 1 vor dem Komma steht. Diese 1 wird *nicht* mitgespeichert.
- Zum Exponenten e wird vor dem Speichern ein "Bias" dazuaddiert, damit der gespeicherte Wert immer positiv ist (auch bei negativem e).
- Der Wert 0 wird durch lauter 0-Bits (außer eventuell einem Vorzeichen, man unterscheidet $+0$ und -0) dargestellt.
- Einige Spezialwerte sind für die Darstellung von *"unendlich"* (*"infinity"*) und fehlerhaften Ergebnissen (*"NaN"* — *"Not a Number"*, z. B. Wurzel aus einer negativen Zahl oder 0/0) reserviert.

Zwei Varianten sind heute gebräuchlich:

	Single Precision	**Double Precision**
Bits insgesamt	32	64
Bits Exponent	8	11
Bits Mantisse	23	52
Bias	127	1023
Maximalwert, dezimal	10^{38}	10^{308}
Stellen, dezimal	6	16

Beispiel: Darstellung der Dualzahl $-1100,10111000100011_2 = -1,10010111000100011 * 2^3$ (dezimal $-12,72088623046875$):

1	10000010	10010111000100011000000

Vorzeichen	e mit Bias 127	Nachkommateil von m

Gerechnet wird mit dualen Gleitkommazahlen genauso wie mit Dezimalzahlen in Exponentenschreibweise:

Addition und Subtraktion: Zuerst muß die Mantisse der Zahl mit dem kleineren Exponenten so lange verschoben werden, bis beide Zahlen den gleichen Exponenten haben. Dann können die Mantissen normal addiert bzw. subtrahiert werden.

Multiplikation und Division: Hier ist kein Anpassen der Exponenten notwendig: Die Exponenten werden addiert bzw. subtrahiert, die Mantissen multipliziert bzw. dividiert.

In beiden Fällen muß das Ergebnis anschließend wieder normalisiert werden.

2.3.3 Zeichen

Der ASCII-Code

ASCII = American Standard Code for Information Interchange

Der ASCII-Code war der erste allgemein verwendete, herstellerunabhängige Code zur Darstellung von Zeichen (ursprünglich auf Fernschreibern und Lochstreifen: Die Löcher in einem Lochstreifen entsprechen direkt den Bits im ASCII-Code).

Er codiert Buchstaben, Ziffern, Sonderzeichen, den Zwischenraum und *Steuerzeichen* als ***7-Bit-Zahlen***, d. h. als Werte zwischen 0 und 127 (0–31 ($1F_{16}$) und 127 ($7F_{16}$) sind nicht druckbare Steuerzeichen, 32 (20_{16}) ist der Zwischenraum).

Dez	*0*	*16*	*32*	*48*	*64*	*80*	*96*	*112*	
0	*nul*	*dle*	␣	0	@	P	‘	p	*0*
1	*soh*	*dc1*	!	1	A	Q	a	q	*1*
2	*stx*	*dc2*	"	2	B	R	b	r	*2*
3	*etx*	*dc3*	#	3	C	S	c	s	*3*
4	*eot*	*dc4*	$	4	D	T	d	t	*4*
5	*enq*	*nak*	%	5	E	U	e	u	*5*
6	*ack*	*syn*	&	6	F	V	f	v	*6*
7	*bel*	*etb*	’	7	G	W	g	w	*7*
8	*bs*	*can*	(	8	H	X	h	x	*8*
9	*tab*	*em*	)	9	I	Y	i	y	*9*
10	*lf*	*sub*	*	:	J	Z	j	z	*A*
11	*vt*	*esc*	+	;	K	[	k	{	*B*
12	*ff*	*fs*	,	<	L	\	l	|	*C*
13	*cr*	*gs*	-	=	M	]	m	}	*D*
14	*so*	*rs*	.	>	N	^	n	~	*E*
15	*si*	*us*	/	?	O	_	o	*del*	*F*
	00	*10*	*20*	*30*	*40*	*50*	*60*	*70*	***Hex***

TABELLE 1: *Der ASCII-Zeichensatz*

Die ISO-Codes

ISO = International Standards Organization

Nachdem der ASCII-Code keinerlei nationale Sonderzeichen (Umlaute) enthielt und ein Byte 8 Bits bietet, also 256 Werte enthalten kann, entwickelte sich rasch ein Wildwuchs von 8-Bit-ASCII-Erweiterungen, die die Werte ab 128 mit verschiedensten Zeichen belegten (Umlaute, grafische Symbole, Striche zum Tabellen zeichnen, ...): Beinahe jeder Hersteller, jedes Betriebssystem, und jedes Land hatte bald seine eigenen ASCII-Codes (bekannt sind vor allem die IBM PC Extended ASCII Codepage 437 und 850, die “DOS-Zeichensätze”).

Inzwischen wurden einige dieser 8-Bit-Erweiterungen von ASCII offiziell als die sogenannten ISO-Zeichensätze standardisiert: ***ISO 8859-1*** (auch “ISO Latin 1”) ist der heute in den USA und Westeuropa übliche Zeichensatz, ISO 8859-15 ist das gleiche mit dem Euro-Symbol. Beide enthalten in den oberen 128 Zeichen vor allem die in westeuropäischen Sprachen vorkommenden Umlaute, die unteren 128 Zeichen sind ident zu ASCII.

Achtung:

- Umlaute stehen nicht unmittelbar nach den entsprechenden Buchstaben, sondern ganz wo anders im Zeichensatz.

  Einerseits erkennen dadurch Tests wie `char>='a' and char<='z'` (“Ist `char` ein Kleinbuchstabe?”) Umlaute nicht, und andererseits ergibt sich daraus eine falsche Sortierreihenfolge (alle Umlaute hinter `z`). Aus demselben Grund ist es nicht sinnvoll (in vielen Programmiersprachen auch gar nicht zulässig), Umlaute in Variablennamen zu verwenden.

- In einigen Programmiersprachen (vor allem C) ist nicht festgelegt, ob Buchstaben intern als Signed Integer oder Unsigned Integer betrachtet werden.

Verwendet die Implementierung einer Programmiersprache Signed Integers, so haben alle Zeichen über 127 (d. h. alle Umlaute und andere nicht-ASCII-Zeichen) in Wahrheit *negative* Werte!

Der Unicode

Um alle nationalen Zeichen (auch der fernöstlichen Kulturen) in einem einzigen Zeichensatz unterzubringen, verwendet der Unicode ***2 oder 4 Bytes*** (16 oder 32 Bits) pro Zeichen und kann damit 2^{16} bzw. 2^{32} verschiedene Zeichen darstellen, braucht aber dafür doppelt so viel Speicherplatz für den gleichen Text.

So wie bei den ISO-Codes sind die Werte 0–127 ident mit dem ASCII-Zeichensatz, die vorderen Byte von ASCII-Zeichen im Unicode sind also 0.

Um das Problem des verdoppelten Speicherplatzverbrauchs zu umgehen, wird für Unicode üblicherweise eine variabel lange Codierung verwendet: Die ASCII-Zeichen werden in einem Byte dargestellt, höhere und seltenere Zeichen in bis zu sechs Bytes. Diese Darstellung des Unicode-Zeichensatzes (inzwischen Default in Linux) heißt ***UTF-8***.

Beide Darstellungen erfordern gegenüber einem Ein-Byte-Zeichensatz wesentliche Umstellungen aller Programme.

Java war die erste Programmiersprache, die Strings in Unicode voll unterstützte. Auch manche Microsoft-Produkte verwenden intern schon Unicode.

Der EBCDIC-Code

EBCDIC = Extended Binary Coded Decimal Interchange Code

Der EBCDIC-Code wird auf IBM-Großrechnern verwendet. Auch er bildet Zeichen auf Bytes (d. h. 8-Bit-Zahlen) ab, aber in ganz anderer Anordnung als der ASCII-Code.

Wie beim ASCII-Code gibt es auch beim EBCDIC-Code viele nationale Varianten.

Achtung:
Im EBCDIC-Zeichensatz haben die Buchstaben keine aufeinanderfolgenden, fortlaufenden Werte! Mit anderen Worten: Zwischen `a` und `z` liegen nicht nur Buchstaben, sondern auch andere Zeichen!

Die ANSI-Terminal-Steuercodes

ANSI = American National Standards Institute

Weitgehend standardisiert sind inzwischen auch die Zeichenketten (meist beginnend mit den ASCII-Steuercodes *`esc`* oder *`csi`*), die — eingebettet in ASCII-Text — diverse Steuerfunktionen am Terminal bewirken (Fettschrift, Farbe, Cursorbewegung, …). Beispiele:

- *`esc`*`[2J` Bildschirm löschen.
- *`esc`*`[11;44H` Cursor in die 11. Zeile / 44. Spalte setzen.
- *`esc`*`[31m` Auf rote Schrift umschalten.

Auch die F-Tasten, Cursor-Steuer-Tasten usw. werden mit solchen ESC-Sequenzen codiert:

- *`esc`*`[A` Cursor-Up-Taste.
- *`esc`*`[21~` F10-Taste.

2.3.4 Strings

Strings bestehen aus nacheinander gespeicherten Buchstaben.

Das Problem dabei ist, daß in vielen Programmiersprachen für eine String-Variable ein Speicherbereich fixer Länge (nämlich der maximalen Länge, die der String haben darf) reserviert werden muß, der zur Laufzeit tatsächlich darin gespeicherte Text dann aber variable Größe hat.

Dafür gibt es folgende Lösungen:

- Der String wird intern immer in voller, fixer Länge verarbeitet, die nicht benötigten Zeichen werden beim Einlesen mit Füllzeichen (Zwischenraum oder Nullbyte) gefüllt und mitverarbeitet.

  Diese Darstellung war früher vor allem auf Großrechnern üblich, heute wird sie hauptsächlich noch bei der Speicherung von Daten in Files fixer Formatierung (Tabellen mit fixer Spaltenbreite) verwendet, aber kaum noch bei der Darstellung von Strings im Hauptspeicher.

- Das Ende des Strings wird durch einen speziellen Ende-Wert (ein Nullbyte) markiert, nur die Buchstaben bis zum ersten Nullbyte werden verarbeitet.

  Das ist die in der Programmiersprache C übliche und daher heute am weitesten verbreitete Darstellung von Strings im Hauptspeicher.

- Die Programmiersprache speichert automatisch bei jedem String einen Zähler mit, in dem gespeichert wird, wie viele Zeichen des Strings tatsächlich gerade benutzt sind (wie lang der Text ist): Der Zähler steuert, wie viele Zeichen bei der Verarbeitung berücksichtigt werden.

  Diese Methode ist in vielen anderen Programmiersprachen gebräuchlich.

Moderne Programmiersprachen reservieren zur Laufzeit dynamisch gerade so viel Speicherplatz, wie für die tatsächliche Länge des Strings notwendig ist. Auch dabei wird die Länge in einem intern angelegten Integer eigens mitgespeichert.

2.4 Programmcode, AV-Daten

2.4.1 Darstellung von Programmen

Maschinencode, Binärprogramm, ... = Aneinanderreihung von ***Maschinenbefehlen***
(*“Instructions”*)

Programme in Pascal, C, usw. (für den Computer normaler Text) werden vom ***Compiler*** in Maschinenprogramme übersetzt, diese bestehen aus einer Aneinanderreihung von Maschinenbefehlen.

1 Maschinenbefehl = Bitmuster bestimmter Länge (1–8 Byte)

Arten von Maschinenbefehlen:

- Lade- und Speicherbefehle, I/O-Befehle *bewegen Daten*
- Arithmetische und logische Operationen, Vergleichsbefehle *verknüpfen Daten*
- Bedingte und unbedingte Sprünge, Unterprogrammcalls und -returns *steuern den Ablauf*
- Stringbefehle, Gleitkommabefehle, Multimediabefehle *verarbeiten speziell codierte Daten*

Je nach Befehl sind die Bits des Befehls in mehrere Felder unterteilt:

Befehlsfeld: Gibt an, um welchen Befehl es sich handelt.
Operandenfeld oder -felder: Gibt an:

- Bei arith. Operationen, Lade- und Speicherbefehlen: Den / die Operanden:
  * Eine Integerkonstante
  * Eine Registernummer
  * Eine Adressierungsart, die bestimmt, wie die Adresse jener Hauptspeicherzelle berechnet wird, die den Operanden enthält festgelegten Adresse (absolute Adresse, Register, Adresse plus Register, ...)
- Bei Sprüngen und Unterprogrammaufrufen: Die Zieladresse (absolute Adresse im Programmcode oder Distanz relativ zur momentanen Programmadresse)

Darstellung von Maschinenbefehlen: Hexadezimal oder in ***Assembler-Notation***:

Label *Opcode* *Operanden* ; *Kommentar*

Label ... Sprungziel, Name einer Speicheradresse oder Variable (optional) (ergibt keinen Code!)
Opcode ... symbolische Abkürzung (***"Mnemonic"***) für den Befehl *Operanden* ... Label, Hex-Adresse, Int-Konstanten, Registername, ...
Kommentar ... selbstredend

Beispiele:

`ADD R3,(R1)` "Addiere den Inhalt der Speicherzelle, deren Adresse im Register 1 steht, zu Register 3."
`JNE LNEXT` "Wenn der letzte Vergleich 'ungleich' ergab, spring zu der Programmstelle, die mit dem Label `LNEXT` bezeichnet ist."

Programm zum Übersetzen von Programmen in Assembler-Notation in Binärprogramme:
Assembler

Menge aller Befehle, die ein bestimmter Prozessortyp kennt:
Befehlssatz des Prozessors (Instruction Set) (meist 100–300 Befehle)

Durch den Prozessor vorgegeben, bei jedem Prozessortyp unterschiedlich!!!
(intel x86, Alpha, Power-PC, S/390, ...)

2.4.2 Audiovisuelle Daten

Musik & Sprache, Bilder (Images), Filme, ...

2 Arten:

Digitalisierte physikalische Information:
Codierung der akustischen / optischen Signale (Schallwelle, Licht).

Beispiele:

`.wav`-Files: Stärke des Tonsignals (als Zahl) in regelmäßigen Abständen:
CD: Rund 44000 einzelne 16-bit-Zahlen pro Sekunde, bzw. 2 Zahlen für Stereo.
ISDN: 8000 8-Bit-Zahlen pro Sekunde.

Pixel- bzw. Rastergrafik (`.gif`, `.jpg`): Stärke der drei Grundfarben (Rot, Grün, Blau) in jedem Bildpunkt.

Vorteile:

- Einfach aufzunehmen und wiederzugeben (phys. Abtastung).
- Kann jedes Bild / jeden Ton darstellen.
- Einfach als Ganzes physikalisch transformierbar.

Nachteile:

- Große Datenmenge.
- Schwer "logisch" bearbeitbar.
- Schwer skalierbar.

Logische Information:

Beschreibung der Töne / der darzustellenden Objekte.

Beispiele:

`.mid`-Files: MIDI: Beschreibung der Instrumente und der Noten, die jedes Instrument spielt ("Klavier spielt ein kurzes A").

Vektorgrafik: Beschreibung der Bildelemente (Strich, Kreis, Text, ...).

Vorteile:

- Viel kleiner.
- Beliebig skalierbar (Qualität unabhängig von der Auflösung).
- Logisch bearbeitbar.

Nachteile:

- Nur für bestimmte Bild- und Toninhalte geeignet.
- Wiedergabe erfordert Rechenaufwand (MIDI-Synthesizer, Postscript-Interpreter, ...).

2.5 Prüfung, Komprimierung, Verschlüsselung

2.5.1 Prüfung von Daten

Zweck:

Erkennung und Korrektur von "umgefallenen Bits" durch elektrische oder magnetische Störungen, radioaktive Strahlung, mechanische Alterung, Verschmutzung, phys. Schwankungen, ...

Varianten:

- ***Parity-Bit:***

  Anwendung: Heute kaum noch. Früher Hauptspeicher, serielle Leitungen.

  Aufwand: 1 Bit pro 1 Byte

  Wirkung: Erkennt *einzelne* Bitfehler (keine mehrfachen), kann nicht korrigieren.

  Funktion: Das Parity-Bit wird so gesetzt, daß sich immer eine gerade (even) oder ungerade (odd) Anzahl von 1-Bits in Daten + Parity ergibt. Wenn nicht: Fehler!

- ***Hamming-ECC (Error Correcting Code):***

  Anwendung: Hauptspeicher, Cache, Bus, ...

  Aufwand: 7 Bits pro 8 Byte (deshalb ist Server-RAM 72 statt 64 Bits breit)
  Verzögerung beim Schreiben einzelner Bytes (erst read, dann write)!

Wirkung: Kann alle einzelnen Bitfehler korrigieren, erkennt alle Zweibit- und viele größeren Fehler.

Funktion: Besteht aus 7 einzelnen Parity-Bits über verschiedene Bits der zu sichernden Daten. Je nachdem, welche Quersumme richtig und welche falsch ist, läßt sich das umgefallene Bit eindeutig ermitteln (Vorläufer mit mehr Bits: Zeilen- und Spalten-Parity).

- ***CRC (Cyclic Redundancy Check), Solomon-Reed-ECC:***

  Anwendung: Platte, Floppy, Magnetband, CD, ... + Netzwerke

  Aufwand: Je nach Code: 16–512 Bits pro Block (512–32768 Byte)

  Wirkung: Kann (fast) alle Fehler erkennen, und je nach Code mehr oder weniger große (benachbarte) Fehler korrigieren (bis zu rund 8 Byte).

  Funktion: Technisch relativ einfache Hardware-Schaltung (Schieberegister), aber mathematischer Hintergrund schwierig (Polynomarithmetik, Division).

2.5.2 Komprimierung von Daten

Zweck:

- Einsparung von Speicherplatz
- Übertragung von Audio und Video (in Echtzeit) mit möglichst geringer Datenrate

Varianten:

	Verlustfreie Verfahren	***Verlustbehaftete Verfahren***
Ergebnis	Bitweise ident zum Original	*Original nicht wiederherstellbar!* *Datenverlust/Qualitätsverlust* (klingt ähnlich, schaut ähnlich aus)
Anwendung *Beispiele*	Für "echte" Daten, Grafiken `.zip`-Archive, `.gif`-Bilder	Für Audio & Video `.mp3`-Ton, `.jpg`-Bilder, `.mpg`-Video
Kompressionsfaktor	Von Daten abhängig (informationstheoret. Mindestgröße) manche Daten sind unkomprimierbar! typisch 1–3	Beim Komprimieren wählbar (höhere Kompr. $\Longrightarrow$ mehr Verluste) Alle Daten werden komprimiert typisch 3–100

Idee, Methoden:

- ***Verlustfreie Verfahren:*** Verschiedene Wege:

  Wiederholungskodierung: Aufeinanderfolgende gleiche Werte werden durch Wert + Wiederholungsanzahl ersetzt.

  Statistische Kodierung: Wie Morse-Code: Häufige Buchstaben oder Buchstabenfolgen werden durch kurze Bitfolgen ersetzt, seltene durch lange (z. B. *Huffman-Coding*).

  Wörterbuch-Verfahren: Es wird ein "Wörterbuch" vorkommender Zeichenketten aufgebaut, schon einmal vorgekommene Zeichenketten werden nur mehr durch ihre Position im Wörterbuch kodiert (z. B. *Lempel-Ziv-Komprimierung: LZW*).

- ***Verlustbehaftete Verfahren:***

  Analyse der menschlichen Wahrnehmung (Hören, Sehen) $\Longrightarrow$
  für den Menschen unwichtige / kaum wahrnehmbare / am wenigsten störende Informationsverluste und Verfälschungen werden in Kauf genommen (leise Töne, geringe Farbverschiebungen, ...).
  Bild/Ton wird physikalisch/mathematisch umgeformt (Fourier-Transformation, ...).
  Das Ergebnis enthält *keine* einzelnen Pixel- bzw. Abtast-Daten mehr!

2.5.3 Verschlüsselung von Daten (Kryptologie)

Zweck:

1. ***Geheime Übertragung von Daten:*** Jemand, der die Daten unbefugt liest, soll deren Inhalt nicht herausfinden können.
2. ***Erkennung von Fehlern und Verfälschungen:*** Es darf nicht möglich sein, die Daten gezielt unbemerkt zu verändern.
3. ***Eindeutigkeit des Absenders:*** Der Empfänger muß prüfen können, ob die Daten wirklich vom angegebenen Sender stammen: Es darf kein anderer Daten erzeugen können, die diese Prüfung bestehen. Umgekehrt darf der Sender nicht abstreiten können, daß er die Daten erzeugt hat.
4. ***Passwort-Prüfung:*** Passwörter sind so zu speichern, daß sie niemand (auch nicht der Systemverwalter!) im Klartext herausbekommt.

Idee:

- Bei 1. werden die Daten vom Sender verschlüsselt und vom Empfänger entschlüsselt. Dafür gibt es zwei grundsätzliche Methoden:

  Symmetrische Verfahren: Zur Kodierung und Dekodierung wird derselbe Schlüssel eingesetzt. Dieser muß geheim bleiben, wer ihn kennt, kann Nachrichten verschlüsseln und entschlüsseln. Problem daher: Wie kommt der Schlüssel von A nach B?
  Beispiel: *DES* (Data Encryption Standard).

  Asymmetrische Verfahren: Es gibt zwei verschiedene, zusammengehörige, voneinander abhängige Schlüssel: Einen öffentlichen, allgemein bekannten zum Verschlüsseln und einen privaten, geheimen zum Entschlüsseln (aus dem öffentlichen Schlüssel läßt sich nicht der private berechnen, umgekehrt schon).
  Beispiel: *RSA* (benannt nach den Erfindern)
  Damit braucht man nie einen Schlüssel geheim weitergeben: Nur der Empfänger braucht den geheimen Schlüssel wissen, und nur er kann Nachrichten entschlüsseln. Der zum Versenden notwendige Schlüssel braucht nicht geheimgehalten werden, weil man mit ihm keine Nachrichten entschlüsseln kann.

- Für 2. kommen sogenannte "one-way-hashes" (Beispiel: *MD5*) zum Einsatz, die zu gegebenen Daten eine Prüfsumme (auch Signatur genannt) berechnen. Diese ist üblicherweise kürzer als die Daten selbst (die Daten lassen sich daher nicht aus der Signatur berechnen, sondern werden unabhängig von der Signatur übertragen). Die wesentlichen Eigenschaften von "kryptografisch sicheren" Prüfsummenverfahren sind, daß

  * sich zu gegebener Prüfsumme keine Daten mit dieser Prüfsumme konstruieren lassen[4]
  * und zwei "ähnliche" Daten immer verschiedene Prüfsumme haben.

  Fehlerkorrektur-Bits oder einfache Prüfsummen erfüllen diese Forderungen *nicht*!

  Die Signatur wird entweder unabhängig von den Daten auf einem Server, auf den nur der Ersteller der Daten schreibenden Zugriff hat, für jeden lesbar zur Überprüfung angeboten, oder mit einer nur dem Ersteller bekannten Verschlüsselung (siehe 3.) erstellt und an die Daten angehängt.

---

[4] Das hat folgenden Sinn: Wenn jemand die ursprünglichen Daten und die dazugehörige Signatur kennt, und eine gezielte Manipulation in die Daten einbauen will, darf es nicht möglich sein, eine zweite, "unwichtige" Änderung der Daten zu berechnen, sodaß im Endeffekt die ursprüngliche Signatur wieder stimmt.

- Für 3. kommen asymmetrische Verfahren zum Einsatz, aber genau gegenteilig wie bei 1.: Der Sender verschlüsselt die für 2. berechnete Signatur (oder auch die gesamte Nachricht) mit seinem nur ihm bekannten privaten Schlüssel. Jeder kann die Signatur oder Nachricht mit dem öffentlichen Schlüssel entschlüsseln und prüfen.

  Wurde die Nachricht von einem anderen Sender erstellt oder manipuliert, der den privaten Schlüssel nicht kennt, wird die Entschlüsselung mit dem öffentlichen Schlüssel scheitern. Beispiel: *PGP*

  Zertifikate verfolgen eine ähnliche Idee.

- Für 4. kommen wie bei 2. nicht umkehrbare Einweg-Verschlüsselungen (wie MD5) zum Einsatz: Selbst wenn das verschlüsselte Passwort in unbefugte Hände gerät, kann daraus nicht das ursprüngliche Passwort berechnet werden. Beim Anmelden wird das eingegebene Passwort ebenfalls verschlüsselt und mit dem gespeicherten verglichen.

  Das direkte Knacken von Passwörtern ist daher mathematisch nicht möglich. Passwort-Knacker verschlüsseln viele tausend Passwörter pro Sekunde und vergleichen das Ergebnis. Da das Durchprobieren aller Möglichkeiten Jahre dauern würde, beginnen sie mit häufigen und naheliegenden Passwörtern. Gute und schlechte Passwörter unterscheiden sich also dadurch, wie leicht sie zu erraten sind bzw. ob sie in Wörterbüchern, Passwort-Sammlungen usw. vorkommen.

  Gegen Belauschen und Wiedereinspielen der Passwort-Übertragung helfen Verfahren, bei denen das übertragene verschlüsselte Passwort jedesmal anders aussieht: Einmal-Passwörter, Mitcodieren der Zeit, Challenge-Response-Verfahren, ...

Methoden: Verschiedenste, unter anderem:

Verfahren auf XOR-Basis: Die Bits der Nachricht werden möglichst trickreich durcheinandergewürfelt und mit den Bits des Schlüssels verknüpft (XOR = “Exklusives Oder”, eine Verknüpfung von Bits, siehe nächstes Kapitel). Beispiel: *DES* (Data Encryption Standard)

Verfahren mit langen Zahlen: Die Bits der Nachricht werden als sehr lange Integer-Zahlen (z. B. 512 oder 1024 Bits lang, das sind rund 300 Ziffern!) aufgefaßt; die Schlüssel sind ebenfalls lange Zahlen, und die Verschlüsselung / Entschlüsselung besteht aus Multiplikationen, Restbildungen usw. (die Verfahren beruhen auf mathematischen Eigenschaften von Primzahlen und der Tatsache, daß sich große Zahlen nicht in vernünftiger Zeit in Faktoren zerlegen lassen). Beispiel: *RSA*

3 Logik und Schaltalgebra

3.1 Logik — Grundlagen

Geschichtliches:

- Als erster Logiker gilt ***Aristoteles*** im alten Griechenland, ≈ 350 v. Chr..
- Die Logik in der Neuzeit begann mit ***George Boole***, ≈ 1850. Er etablierte die Logik als eigene Wissenschaft, sie trennte sich von der Philosophie und wurde ein Zweig der Mathematik.
- Der Österreicher ***Kurt Gödel*** (≈ 1935) erarbeitete logische Erkenntnisse, die sowohl für die mathematische Grundlagenforschung (Beweisbarkeit usw.) als auch für die theoretische Informatik (Berechenbarkeitstheorie usw.) von größter Bedeutung sind.

Eingrenzung:

Wir befassen uns hier mit der ***zweiwertigen Logik***, d. h. mit der Logik, die ***wahr*** und ***falsch*** kennt[5].

Aussage:

Eine ***Aussage*** ist ein Satz (umgangssprachlich oder in irgendeinem Formalismus), von dessen inhaltlicher Bedeutung eindeutig entschieden werden kann, ob sie *wahr* oder *falsch* ist (es genügt, wenn der Wahrheitsgehalt prinzipiell feststellbar ist, er muß nicht von vornherein bekannt sein).

Sätze, bei denen es nicht möglich ist, anzugeben, ob sie wahr oder falsch sind, sind *keine* Aussagen.

Beispiele:

- Holz schwimmt auf Wasser. (wahr)
- Ein Einser ist eine gute Note. (je nach Land!)
- Innsbruck liegt an der Donau. (falsch)
- 2 ist eine gerade Zahl. (wahr)
- 2100 ist ein Schaltjahr. (falsch) (sprachlich schlampig: '2100' = 'Das Jahr 2100')
- Morgen fällt Schnee. (?)
- Es gibt fremdes Leben im All. (?)

Beispiele für Sätze, die *keine* Aussagen sind:

- Geh nach Hause!
- Autos fahren schnell.

Logische Ausdrücke: (boolsche Ausdrücke)

Logische Ausdrücke sind Sätze, die Variablen enthalten, und nach Ersetzung aller Variablen durch Elemente aus deren Wertebereich eine Aussage darstellen.

Solange ein Satz Variablen enthält, ist er *keine* Aussage! Erst nach Belegung der Variablen läßt sich der Wahrheitsgehalt abklären!

Beispiele:

---

[5] Es gibt auch andere Logik-Systeme, z. B. mehrwertige Logik (wahr / falsch / vielleicht) oder Fuzzy Logik ("unscharfe Logik").

- Herr x wohnt in Wien.
- n ist eine Primzahl.
- `x=0 or x>=10 and x<100`

Zusammengesetzte Aussagen:

Eine ***zusammengesetzte Aussage*** besteht aus mehreren, miteinander verknüpften Aussagen.

Beispiele:

- Wenn ich Schule habe, und es regnet oder es ist schon knapp, dann fahre ich mit dem Auto in die Schule.
- `7=0 or 7>=10 and 7<100`

Aussagevariablen:

Eine ***Aussagevariable*** (boolsche Variable, Datentyp `bool` in Programmiersprachen) ist eine Variable, die mit den beiden Werten *wahr* oder *falsch* (bzw. `true` oder `false`) belegt werden kann. Sie kann daher als *Platzhalter für Aussagen* verwendet werden.

Aussagevariablen werden in der Logik üblicherweise mit Kleinbuchstaben ab "p" bezeichnet, in Programmen bitte sprechendere Variablennamen wählen (beispielsweise `ist_gerade` oder `fertig`).

Wichtige Aufgabe der Logik & unser Ziel in nächster Zeit:

Nicht Untersuchung elementarer Aussagen auf ihren Wahrheitswert,
sondern Untersuchung, Umformung, Vereinfachung, ... zusammengesetzter Aussagen!

Zu diesem Zweck werden alle elementaren Aussagen in einer zusammengesetzten Aussage durch Aussagevariablen ersetzt, sodaß nur die Struktur der zusammengesetzten Aussage erhalten bleibt! Wenn eine zusammengesetzte Aussage an mehreren Stellen die gleiche elementare Aussage enthält, so wird diese an allen Stellen durch die gleiche Variable ersetzt.

Beispiele:

- Wenn p, und q oder r, dann s.
- p `or` q `and` r

Aussageform:

Eine ***Aussageform*** (oder ***aussagenlogischer Ausdruck***) ist ein logischer Ausdruck, der nur Aussagevariablen (und keine anderen Variablen) enthält.

Eine Aussageform entsteht aus einer zusammengesetzten Aussage, wenn man die elementaren Aussagen durch Aussagevariablen ersetzt, und wird wieder zu einer Aussage, wenn man die Aussagevariablen alle durch *wahr* oder *falsch* ersetzt.

Beispiele für Aussageformen: Siehe oben.
Beispiele für daraus entstehende Aussagen:

- Wenn `W`, und `W` oder `F`, dann `W`.
  wahr!
- Wenn `W`, und `F` oder `F`, dann `W`.
  Nach den Gesetzen der Logik *wahr*, aber inhaltlich nicht das, was ich gemeint habe!
- Wenn `W`, und `F` oder `W`, dann `F`.
  falsch!

Die zentrale Frage:

Für welche Belegungen der Aussagevariablen mit *wahr* bzw. *falsch* wird die Aussageform zu einer wahren bzw. falschen Aussage?

Für eine Aussageform mit n Aussagevariablen gibt es 2^n Belegungsmöglichkeiten ...

Eine Lösungsmöglichkeit dazu:

Die ***Wahrheitstafel:*** Eine Tabelle mit vielen `W` ("*wahr*") und `F` ("*falsch*").

1. Man macht n Spalten, für jede Variable eine (alfabetisch geordnet).
2. Man macht 2^n Zeilen, für jede mögliche Kombination der Belegung der Variablen eine (systematisch, wie beim Hinaufzählen mit Binärzahlen!).
3. Man macht nach Bedarf weitere Spalten, von links nach rechts, für immer größere Teilausdrücke des gegebenen Ausdruckes, und zwar so, daß jeder Teilausdruck zwei schon links von ihm in der Tabelle enthaltene Ausdrücke kombiniert, bis schließlich in der äußerst rechten Spalte der Gesamtausdruck steht.
4. Man füllt die neuen Tabellenfelder von links nach rechts mit `W` und `F`, die man aus den weiter links stehenden `W` und `F` entsprechend den jeweiligen Ausdrücken berechnet.
5. Aus der fertigen Tabelle kann man direkt ablesen, bei welchen Belegungen der Aussagevariablen (bzw. bei welchen Ergebnissen der elementaren Aussagen, für die die jeweiligen Variablen eingesetzt wurden) (in den linken Spalten) die gesamte Aussage *wahr* oder *falsch* ist (gleiche Zeile, rechteste Spalte).

Beispiel:

p	q	r	s	q oder r	p und (q oder r)	Wenn p und (q oder r), dann s
F	F	F	F	F	F	W
F	F	F	W	F	F	W
F	F	W	F	W	F	W
F	F	W	W	W	F	W
F	W	F	F	W	F	W
F	W	F	W	W	F	W
F	W	W	F	W	F	W
F	W	W	W	W	F	W
W	F	F	F	F	F	W
W	F	F	W	F	F	W
W	F	W	F	W	W	F
W	F	W	W	W	W	W
W	W	F	F	W	W	F
W	W	F	W	W	W	W
W	W	W	F	W	W	F
W	W	W	W	W	W	W

Sonderfälle:

Ein aussagenlogischer Ausdruck heißt ***Tautologie*** ("*immer wahrer Satz*"), wenn er bei *jeder* möglichen Belegung der darin vorkommenden Aussagevariablen eine *wahre* Aussage ergibt.

Ein aussagenlogischer Ausdruck heißt ***Kontradiktion*** ("*immer falscher Satz*"), wenn er bei *jeder* möglichen Belegung der darin vorkommenden Aussagevariablen eine *falsche* Aussage ergibt.

Sonst (d. h. wenn er für mindestens eine Belegung *wahr* und für mindestens eine andere Belegung *falsch* ergibt) nennt man den Ausdruck ***"indeterminiert"*** ("*unbestimmt*").

Beispiele:

- Wenn ich Kopfweh habe, ändert sich das Wetter, oder es ändert sich nicht. (Tautologie)
- Wenn es regnet, regnet es. (Tautologie)
- Es gibt Leben auf dem Mars, und es gibt kein Leben auf dem Mars. (Kontradiktion)

3.2 Junktoren

Um

- Aussagen und logische Ausdrücke kurz schreiben zu können
- und Ungenauigkeiten und Zweideutigkeiten umgangssprachlicher Formulierungen zu vermeiden,

wurde in der Logik folgende Schreibweise eingeführt:

*Aussagen werden mittels aussagenlogischer Operatoren (**Junktoren**) zu zusammengesetzten Aussagen verknüpft.*

- Die ***Negation*** (***NOT***, $\neg$, Verneinung, umgangssprachlich "*... nicht ...*") kehrt den Wahrheitswert einer Aussage um. Sie ist der einzige *einstellige* junktor, alle anderen sind *zweistellig*.

- Die ***Konjunktion*** (***AND***, $\wedge$, Und-Verknüpfung, umgangssprachlich "*... und ...*") zweier Aussagen ergibt genau dann *wahr*, wenn beide Teilaussagen *wahr* sind.

  Merkregel für das Zeichen $\wedge$: Wenn man ein A wie "And" draus machen kann, ist es ein Und.

- Die ***Disjunktion*** (***OR***, $\vee$, Oder-Verknüpfung, umgangssprachlich "*... oder ... (oder beides)*") zweier Aussagen ergibt genau dann *wahr*, wenn mindestens eine Teilaussage *wahr* ist.

  Merkregel für das Zeichen $\vee$: Man denke an das Flußbett des deutschen Flusses "Oder": Wenn man in das Zeichen Wasser einfüllen kann, ist es ein Oder.

- Die ***Implikation*** (Folgerung, $\Rightarrow$, umgangssprachlich *"Wenn ..., dann ..."* oder *"Aus ... folgt ..."*) ist dann *falsch*, wenn die linke Seite *wahr* und die rechte Seite *falsch* ist, und in allen anderen Fällen *wahr*.

  Die linke Seite heißt *Prämisse (Voraussetzung)*, die rechte Seite *Konklusion (Schlußfolgerung)*: *"Unter der Voraussetzung ... gilt ..."*

  Wichtig: Wenn die Voraussetzung nicht erfüllt (d. h. *falsch*) ist, ist die ganze Aussage in jedem Fall *wahr*; es ist dann ganz egal, ob die rechte Seite *wahr* oder *falsch* ist!

  Beispiel: *"Wenn es regnet, ist die Straße naß."*
  Für den Fall, das es nicht regnet, ist diese Aussage *wahr*, ganz egal, ob die Straße naß oder trocken ist.

- Die ***Äquivalenz*** (***Bijunktion***, $\Leftrightarrow$, umgangssprachlich *"Genau dann, wenn ..., gilt ..."* oder *"Aus ... folgt ... und umgekehrt"*) ist genau dann *wahr*, wenn beide Seiten den gleichen Wahrheitswert haben (äquivalent = "gleichwertig").

- Die ***Antivalenz*** ($\not\Leftrightarrow$) ist die Umkehrung der Äquivalenz: Sie ist genau dann *wahr*, wenn die beiden Teilaussagen gegenteiligen Wahrheitswert haben.

  Das entspricht dem ***Exklusiven Oder*** (***XOR***, $|$, umgangssprachlich *"Entweder ... oder ... (aber nicht beides)"*).

- Das ***NAND*** ist die Negation des ***AND*** (umgangssprachlich *"Nicht beide"*).
- Das ***NOR*** ist die Negation des ***OR*** (umgangssprachlich *"Weder … noch …"*).

p	***NOT*** $\neg p$ (oder $\overline{p}$)
F	W
W	F

		AND	***OR***	***Impl.***	***Äquiv.***	***Antiv.***	***XOR***	***NAND***	***NOR***
p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$	$p \not\Leftrightarrow q$	$p \mid q$	$p \overline{\wedge} q$	$p \overline{\vee} q$
F	F	F	F	W	W	F	F	W	W
F	W	F	W	W	F	W	W	W	F
W	F	F	W	F	F	W	W	W	F
W	W	W	W	W	W	F	F	F	F

Aussagenlogische Ausdrücke lassen sich daher wie folgt definieren:

1. Die Zeichen `W` und `F` sind aussagenlogische Ausdrücke (*wahr* und *falsch*).
2. Einzelne Kleinbuchstaben sind aussagenlogische Ausdrücke (Aussagevariablen).
3. Sind $\mathcal{A}$ und $\mathcal{B}$ korrekte aussagenlogische Ausdrücke, so sind auch die folgenden Formeln korrekte aussagenlogische Ausdrücke[6]:
   - $(\mathcal{A})$
   - $\neg\mathcal{A}$
   - $\mathcal{A} \wedge \mathcal{B}$
   - $\mathcal{A} \vee \mathcal{B}$
   - $\mathcal{A} \Rightarrow \mathcal{B}$
   - $\mathcal{A} \Leftrightarrow \mathcal{B}$
4. Für die Junktoren werden folgende Vorrangregeln erklärt (die oberen haben Vorrang vor den unteren), dadurch überflüssige Klammern können weggelassen werden:
   - $\neg$
   - $\wedge$ $\vee$ [7]
   - $\Rightarrow$
   - $\Leftrightarrow$
5. Unnötige Klammern (ganz außen und um einzelne Buchstaben) kann man weglassen.
6. Alle anderen Zeichenketten sind *keine* aussagenlogischen Ausdrücke.

Eine ***Aussage*** ist ein aussagenlogischer Ausdruck ohne Kleinbuchstaben.

---

[6] $\not\Leftrightarrow$, $\mid$, $\overline{\wedge}$ und $\overline{\vee}$ lassen wir weg, weil sie nur eine Abkürzung für $\neg$ und einen anderen Junktor darstellen.

[7] Oft wird "$\wedge$ *vor* $\vee$" definiert, bei uns haben beide die *gleiche* Vorrangstufe: Zur leichteren und eindeutigen Lesbarkeit muß man Klammern setzen!

3.3 Gesetze und Umformungen

Anstatt mit Wahrheitstabellen die Lösungen zu finden, kann man versuchen, aussagenlogische Ausdrücke durch Anwendung von Gesetzen umzuformen. Dabei wird ein Teilausdruck, der von der Struktur her einer Seite des Gesetzes entspricht, durch die andere Seite des Gesetzes ersetzt (wobei für die Variablen im Gesetz entsprechend eingesetzt wird). Ein Gesetz ist dabei *in beide Richtungen* anwendbar.

Die Gesetze sind alle ***Tautologien*** in Form von ***Äquivalenzen***, d. h. die linke Seite hat bei allen möglichen Belegungen der Aussagevariablen den gleichen Wahrheitswert wie die rechte Seite. Es wird daher immer ein Teilausdruck durch einen äquivalenten Ausdruck ersetzt, und damit ist auch der gesamte umgeformte Ausdruck äquivalent zum ursprünglichen Ausdruck (d. h. er ist genau dann wahr, wenn der ursprüngliche Ausdruck wahr ist).

Das entspricht der Umformung mathematischer Ausdrücke durch Kommutativgesetz, Distributivgesetz, "Minus herausziehen" usw..

Gesetze für ¬:

$\neg\neg a$	$\Leftrightarrow a$	**Gesetz der doppelten Negation**

Gesetze für ∨ und ∧:

Idempotenzgesetz (IG)			
$a \vee a$	$\Leftrightarrow a$	$a \wedge a$	$\Leftrightarrow a$
Komplement (KO)			
$a \vee \neg a$	$\Leftrightarrow \mathrm{W}$	$a \wedge \neg a$	$\Leftrightarrow \mathrm{F}$
Neutrales Element (NE)			
$a \vee \mathrm{F}$	$\Leftrightarrow a$	$a \wedge \mathrm{W}$	$\Leftrightarrow a$
Verschmelzungsgesetz (VG)			
$a \vee \mathrm{W}$	$\Leftrightarrow \mathrm{W}$	$a \wedge \mathrm{F}$	$\Leftrightarrow \mathrm{F}$
Kommutativgesetz (KG)			
$a \vee b$	$\Leftrightarrow b \vee a$	$a \wedge b$	$\Leftrightarrow b \wedge a$
Assoziativgesetz (AG)			
$a \vee (b \vee c)$	$\Leftrightarrow (a \vee b) \vee c$	$a \wedge (b \wedge c)$	$\Leftrightarrow (a \wedge b) \wedge c$
Distributivgesetz (DG)			
$a \vee (b \wedge c)$	$\Leftrightarrow (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c)$	$\Leftrightarrow (a \wedge b) \vee (a \wedge c)$
Gesetz von De Morgan (DM)			
$\neg(a \vee b)$	$\Leftrightarrow \neg a \wedge \neg b$	$\neg(a \wedge b)$	$\Leftrightarrow \neg a \vee \neg b$
Absorptionsgesetz (AB)			
$a \vee (a \wedge b)$	$\Leftrightarrow a$	$a \wedge (a \vee b)$	$\Leftrightarrow a$

Gesetze für ⇒:

$a \Rightarrow a$	$\Leftrightarrow \mathrm{W}$	
$a \Rightarrow \neg a$	$\Leftrightarrow \neg a$	
$a \Rightarrow \mathrm{W}$	$\Leftrightarrow \mathrm{W}$	
$a \Rightarrow \mathrm{F}$	$\Leftrightarrow \neg a$	
$\mathrm{W} \Rightarrow a$	$\Leftrightarrow a$	
$\mathrm{F} \Rightarrow a$	$\Leftrightarrow \mathrm{W}$	
$a \Rightarrow b$	$\Leftrightarrow \neg a \vee b$	**Definition von ⇒**
$a \Rightarrow (b \Rightarrow c)$	$\Leftrightarrow b \Rightarrow (a \Rightarrow c)$	**Vertauschung der Prämissen**

Gesetze für $\Leftrightarrow$:

$(a \Leftrightarrow a)$	$\Leftrightarrow \mathrm{W}$	
$(a \Leftrightarrow \neg a)$	$\Leftrightarrow \mathrm{F}$	
$(a \Leftrightarrow \mathrm{W})$	$\Leftrightarrow a$	
$(a \Leftrightarrow \mathrm{F})$	$\Leftrightarrow \neg a$	
$(a \Leftrightarrow b)$	$\Leftrightarrow (a \Rightarrow b) \wedge (b \Rightarrow a)$	**Definition von $\Leftrightarrow$**
$(a \Leftrightarrow b)$	$\Leftrightarrow (a \wedge b) \vee (\neg a \wedge \neg b)$	**Definition von $\Leftrightarrow$**
$(a \Leftrightarrow b)$	$\Leftrightarrow (\neg a \vee b) \wedge (a \vee \neg b)$	**Definition von $\Leftrightarrow$**
$(a \Leftrightarrow b)$	$\Leftrightarrow (b \Leftrightarrow a)$	**Kommutativgesetz**
$(a \Leftrightarrow \neg b)$	$\Leftrightarrow (\neg a \Leftrightarrow b)$	
$(a \Leftrightarrow (b \Leftrightarrow c))$	$\Leftrightarrow ((a \Leftrightarrow b) \Leftrightarrow c)$	**Assoziativgesetz**

Das Dualitätsprinzip:

Wenn zwei aussagenlogische Ausdrücke a und b, in denen nur die Junktoren $\vee \wedge \neg$ vorkommen, äquivalent sind $(a \Leftrightarrow b)$, dann sind auch jene Ausdrücke $\hat{a}$ und $\hat{b}$ äquivalent $(\hat{a} \Leftrightarrow \hat{b})$, die wie folgt aus a und b gebildet werden:

- Alle $\vee$ werden durch $\wedge$ und alle $\wedge$ durch $\vee$ ersetzt.
- Alle `W` werden durch `F` und alle `F` durch `W` ersetzt.

Weiters gelten folgende wichtige ***Tautologien*** mit $\Rightarrow$. Diese eignen sich zwar ***nicht*** zum Umformen von Ausdrücken (weil sie nur in eine Richtung gelten und daher der umgeformte Ausdruck nicht für alle Belegungen äquivalent zum ursprünglichen Ausdruck wäre), sind aber wesentliche ***Schlußregeln*** beim Beweisen: Wenn man weiß, daß die linke Seite des $\Rightarrow$ gilt, kann man daraus schließen, daß auch die rechte Seite gilt.

$a \wedge b$	$\Rightarrow a$
a	$\Rightarrow a \vee b$
$a \wedge (a \Rightarrow b)$	$\Rightarrow b$
$(a \Rightarrow b) \wedge \neg b$	$\Rightarrow \neg a$
$(a \Rightarrow b)$	$\Rightarrow (\neg b \Rightarrow \neg a)$
$(a \Rightarrow b) \wedge (b \Rightarrow c)$	$\Rightarrow (a \Rightarrow c)$
a	$\Rightarrow (b \Rightarrow a)$
$(a \Rightarrow (b \Rightarrow c))$	$\Rightarrow ((a \Rightarrow b) \Rightarrow (a \Rightarrow c))$

3.4 Junktorenbasen

Es gibt 16 verschiedene Fälle, die bei der Verknüpfung zweier Aussagen für die 4 möglichen Kombinationen von `W` und `F` auftreten können:

a	b		$\overline{\vee}$					$\not\Leftrightarrow$	$\overline{\wedge}$	$\wedge$	$\Leftrightarrow$		$\Rightarrow$			$\vee$	
F	F	F	W	F	W	F	W	F	W	F	W	F	W	F	W	F	W
F	W	F	F	W	W	F	F	W	W	F	F	W	W	F	F	W	W
W	F	F	F	F	F	W	W	W	W	F	F	F	F	W	W	W	W
W	W	F	F	F	F	F	F	F	F	W	W	W	W	W	W	W	W

Dementsprechend gäbe es im Prinzip 16 verschiedene zweistellige Junktoren. Einige dieser Junktoren können durch andere Junktoren ausgedrückt bzw. ersetzt werden:

- $a \Leftrightarrow b$ entspricht $(a \wedge b) \vee (\neg a \wedge \neg b)$.
- $a \overline{\wedge} b$ entspricht $\neg a \vee \neg b$.

Es sind also nicht alle Junktoren notwendig, um alle Fälle ausdrücken zu können, die bei der Verknüpfung zweier Aussagen möglich sind.

Definitionen:

Junktorenbasis: Eine Junktorenbasis ist eine Menge von Junktoren, mit der alle anderen Junktoren ausgedrückt (äquivalent dargestellt) werden können.

Mit anderen Worten: Die Junktoren einer Junktorenbasis reichen aus, um alle Fälle auszudrücken, die bei der Verknüpfung zweier Aussagen möglich sind.

Beispiel: $\Leftrightarrow \Rightarrow \wedge \vee \neg$ ist eine Junktorenbasis, $\wedge \vee \neg$ auch.

Minimalbasis: Eine Minimalbasis ist eine Junktorenbasis mit möglichst wenig Junktoren.

Mit anderen Worten: Jeder Junktor in einer Minimalbasis ist notwendig. Läßt man einen Junktor weg, kann man nicht mehr alle anderen Junktoren bzw. alle Fälle ausdrücken.

Beispiele:

- $\wedge \vee \neg$ ist ***keine*** Minimalbasis, weil man durch die Gesetze von de Morgan ein $\vee$ durch ein $\wedge$ ersetzen kann oder umgekehrt. Man kann daher entweder das $\vee$ oder das $\wedge$ weglassen.
- $\vee\ \neg$ und $\wedge\ \neg$ sind beides Minimalbasen: Ohne $\vee$ (bzw. $\wedge$) könnte man überhaupt keine zwei Aussagen mehr verknüpfen, und ohne $\neg$ sind gewisse Junktoren (z. B. $\Leftrightarrow$ oder $\overline{\wedge}$ nicht mehr darstellbar.

Es gibt allerdings auch Minimalbasen mit nur *einem* Junktor: Einerseits $\overline{\vee}$, andererseits $\overline{\wedge}$.

Wir betrachten $\overline{\vee}$. Nachdem wir schon wissen, daß $\vee\ \neg$ eine Minimalbasis ist, müssen wir nur zeigen, daß wir $\neg$ und $\vee$ durch $\overline{\vee}$ ausdrücken können, damit wäre schon bewiesen, daß auch $\overline{\vee}$ eine Minimalbasis ist.

Das ist aber leicht:

- $\neg a \Leftrightarrow a \overline{\vee} a$
- $a \vee b \Leftrightarrow \neg(a \overline{\vee} b) \Leftrightarrow (a \overline{\vee} b) \overline{\vee} (a \overline{\vee} b)$

Weiters gilt:

- $a \wedge b \Leftrightarrow (a \overline{\vee} a) \overline{\vee} (b \overline{\vee} b)$
- $(a \Leftrightarrow b) \Leftrightarrow ((a \overline{\vee} a) \overline{\vee} b) \overline{\vee} ((b \overline{\vee} b) \overline{\vee} a)$

Diese Erkenntnis ist u. a. für den Entwurf integrierter Schaltkreise wichtig: Man möchte alle erdenklichen logischen Schaltungen aus möglichst wenig Grund-Schaltelementen zusammensetzen können. Manche Halbleitertechnologien erlauben technisch beispielsweise nur ein $\overline{\vee}$ oder ein $\overline{\wedge}$ als Grundfunktion.

3.5 Normalformen

Ziel dieses Kapitels ist es, bestimmte, einheitliche Darstellungsformen — sogenannte Normalformen — aussagenlogischer Ausdrücke zu betrachten. Diese sind aus mehreren Gründen von Interesse:

- Zwei beliebigen aussagenlogischen Ausdrücken sieht man normalerweise nicht auf Anhieb an, ob sie äquivalent sind. Sind sie hingegen beide in Normalform, läßt sich das normalerweise "auf einen Blick" feststellen.
- Bestimmte Typen von Logik-IC's (z. B. PAL's = Programmable Array Logic) erlauben nur die Realisierung bestimmter Verknüpfungen (eine Oder-Verknüpfung mehrerer Und-Verknüpfungen oder umgekehrt).
  Selbst wenn die Hardware flexibler ist, ist man an möglichst "flachen" (d. h. möglichst wenig tief geschachtelten) Verknüpfungen interessiert, um die Signallaufzeiten kurz zu halten.

Zuerst wie in der Mathematik üblich einige Definitionen:

Literal: Ein Literal ist eine unnegierte oder negierte Aussagevariable: p, $\neg q$, r, ...

Minterm: Ein Minterm ist eine Konjunktion ($\wedge$) von (nicht notwendigerweise verschiedenen) Literalen: $p \wedge \neg q$, $\neg p \wedge \neg q \wedge \neg p \wedge r \wedge \neg r$

Maxterm: Ein Maxterm ist eine Disjunktion ($\vee$) von (nicht notwendigerweise verschiedenen) Literalen: $r \vee q$, $q \vee \neg r \vee q \vee \neg p \vee s$

DNF: Eine DNF (disjunktive Normalform) ist eine Disjunktion von Mintermen (d. h. eine Oder-Verknüpfung von Und-Verknüpfungen von Literalen): $(p \wedge \neg q \wedge s) \vee (p \wedge r \wedge \neg r) \vee t$

KNF: Eine KNF (konjunktive Normalform) ist eine Konjunktion von Maxtermen (d. h. eine Und-Verknüpfung von Oder-Verknüpfungen von Literalen): $p \wedge (\neg q \vee \neg r) \wedge (p \vee r \vee s)$

Basiskonjunktion: Eine Basiskonjunktion ist ein Minterm, der alle im ursprünglichen aussagenlogischen Ausdruck vorkommenden Variablen genau ein Mal als Literal enthält, und zwar sortiert: $\neg p \wedge q \wedge \neg r$ (bei einem Beispiel mit p, q und r).

Basisdisjunktion: Eine Basisdisjunktion ist ein Maxterm, der alle im ursprünglichen aussagenlogischen Ausdruck vorkommenden Variablen genau ein Mal als Literal enthält, und zwar sortiert: $p \vee q \vee \neg r$ (bei einem Beispiel mit p, q und r).

KDNF: Eine kanonische (eindeutige) DNF ist eine Disjunktion von Basiskonjunktionen, wobei jede Basiskonjunktion höchstens einmal auftreten darf: $(p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r)$

KKNF: Eine kanonische (eindeutige) KNF ist eine Konjunktion von Basisdisjunktionen, wobei jede Basisdisjunktion höchstens einmal auftreten darf: $(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r)$

Es gibt zu jedem aussagenlogischen Ausdruck eine (bis auf Umsortierung) eindeutige, äquivalente KDNF und ebenso eine eindeutige, äquivalente KKNF (wohingegen eine "beliebige" KNF oder DNF *nicht* eindeutig ist: Zwei ganz verschieden aufgebaute KNF's oder DNF's können durchaus äquivalent sein!). Um festzustellen, ob zwei Ausdrücke äquivalent sind, reicht es also, deren KKNF oder KDNF zu bilden und zu vergleichen.

Die KDNF läßt sich wie folgt ermitteln:

1. ***Wahrheitstafel erstellen.***
2. `W`***-Zeilen suchen:*** Alle Zeilen mit `W` im Endergebnis markieren.
3. ***Basiskonjunktionen bilden:*** Aus jeder dieser Zeilen je eine Basiskonjunktion bilden: Jede Variable, die in dieser Zeile ein `W` hat, kommt unnegiert in die Basiskonjunktion, jede Variable mit einem `F` in dieser Zeile steht negiert in der Basiskonjunktion.
4. ***KDNF bilden:*** Die so erhaltenen Basiskonjunktionen alle mit $\vee$ verbinden.

Überlegungen zur Korrektheit:

- Eine Basiskonjunktion ist für genau die Zeile (Belegung der Variablen) wahr, für die sie gebildet wurde, und für keine andere Belegung: Nur in dieser Zeile sind alle unnegierten Variablen wahr und alle negierten Variablen falsch (d. h. deren Negation ebenfalls wahr), und nur wenn alle Literale wahr sind, ergibt deren Konjunktion wahr.

- Für jede Zeile mit Ergebnis W ist daher genau eine Basiskonjunktion wahr, für jede F-Zeile sind alle Basiskonjunktionen falsch.
- Die Oder-Verknüpfung aller Basiskonjunktionen ist genau dann wahr, wenn mindestens eine Basiskonjunktion wahr ist, d. h. genau in den Fällen, wo auch die ursprüngliche Aussage wahr ist, und sonst falsch.

Die KKNF wird analog ermittelt:

1. Wahrheitstafel erstellen.
2. Alle Zeilen mit F im Endergebnis markieren.
3. Basisdisjunktionen genau umgekehrt bilden: W-Variablen negiert, F-Variablen unnegiert.
4. Diese Basisdisjunktionen mit $\wedge$ verknüpfen.

Begründung:

- Jede Basisdisjunktion ist in "ihrer" Zeile falsch (weil nur dort alle Literale falsch sind) und in allen anderen Zeilen wahr (weil mindestens ein Literal wahr ist).
- In den W-Zeilen sind daher alle Basisdisjunktionen wahr.
- Folglich ist deren Und-Verknüpfung in den W-Zeilen wahr und in den F-Zeilen falsch.

Beispiel:
Ermittle die KDNF und KKNF zu $\neg((a \vee \neg b) \Rightarrow c)$:

			4	5	6	
a	b	c	$a \vee \neg b$	$4 \Rightarrow c$	$\neg 5$	
F	F	F	W	F	W	$\neg a \wedge \neg b \wedge \neg c$
F	F	W	W	W	F	
F	W	F	F	W	F	
F	W	W	F	W	F	
W	F	F	W	F	W	$a \wedge \neg b \wedge \neg c$
W	F	W	W	W	F	
W	W	F	W	F	W	$a \wedge b \wedge \neg c$
W	W	W	W	W	F	

Die KDNF lautet $(\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge \neg c)$.

			4	5	6	
a	b	c	$a \vee \neg b$	$4 \Rightarrow c$	$\neg 5$	
F	F	F	W	F	W	
F	F	W	W	W	F	$a \vee b \vee \neg c$
F	W	F	F	W	F	$a \vee \neg b \vee c$
F	W	W	F	W	F	$a \vee \neg b \vee \neg c$
W	F	F	W	F	W	
W	F	W	W	W	F	$\neg a \vee b \vee \neg c$
W	W	F	W	F	W	
W	W	W	W	W	F	$\neg a \vee \neg b \vee \neg c$

Die KKNF lautet $(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$.

Eine zu einem gegebenen aussagenlogischen Ausdruck äquivalente DNF läßt sich wie folgt durch Äquivalenzumformungen ermitteln:

1. ***Auf*** $\vee \wedge \neg$ ***bringen:*** Alle anderen Junktoren ($\Rightarrow$, $\Leftrightarrow$, $\overline{\vee}$, ...) durch $\vee \wedge \neg$ ersetzen.

2. ***Ausnegieren:*** Alle Negationen ganz nach innen ziehen: Auf alle Ausdrücke der Form $\neg(a \vee b)$ oder $\neg(a \wedge b)$ so lange die Gesetze von De Morgan anwenden ($\neg a \wedge \neg b$ bzw. $\neg a \vee \neg b$), bis die Negationszeichen nur mehr vor einzelnen Variablen stehen. Dabei doppelte Negationen streichen, sodaß vor jeder Variablen maximal eine Negation überbleibt.
3. ***Ausmultiplizieren:*** Alle $\vee$ nach außen ziehen: Auf alle Ausdrücke der Form $a \wedge (b \vee c)$ oder $(a \vee b) \wedge c$ so lange das Distributivgesetz anwenden ($(a \wedge b) \vee (a \wedge c)$ bzw. $(a \wedge c) \vee (b \wedge c)$), bis eine Disjunktion von Mintermen überbleibt.
4. ***Vereinfachen:***
   (a) ¬W durch F und ¬F durch W ersetzen.
   (b) Minterme, die $\cdots \wedge \mathrm{F} \wedge \cdots$ oder $\cdots \wedge a \wedge \cdots \wedge \neg a \wedge \cdots$ enthalten, komplett streichen. Fallen dadurch alle Minterme weg, ist der ursprüngliche Ausdruck äquivalent zu F (für alle Belegungen falsch)!
   (c) Mehrfache Auftreten desselben Literals in einem Minterm ($\cdots \wedge a \wedge \cdots \wedge a \wedge \cdots$ oder $\cdots \wedge \neg a \wedge \cdots \wedge \neg a \wedge \cdots$) auf ein einziges Auftreten dieses Literals zusammenkürzen.
   (d) Alle W aus den Mintermen streichen. Ausnahme: Besteht ein Minterm nur aus einem W und sonst nichts, ist der ursprüngliche Ausdruck äquivalent zu W (für alle Belegungen wahr)!

Das Verfahren zur Ermittlung einer äquivalenten KNF läuft wieder analog.

Beispiel:
Suche eine DNF zu $\neg(a \vee \neg(b \Rightarrow \neg(c \Rightarrow a)))$.

$\neg(a \vee \neg(b \Rightarrow \neg(c \Rightarrow a)))$	$\Leftrightarrow$	Definition von $\Rightarrow$
$\neg(a \vee \neg(\neg b \vee \neg(\neg c \vee a)))$	$\Leftrightarrow$	De Morgan außen
$\neg a \wedge \neg\neg(\neg b \vee \neg(\neg c \vee a))$	$\Leftrightarrow$	Doppel-¬, De Morgan innen
$\neg a \wedge (\neg b \vee (\neg\neg c \wedge \neg a))$	$\Leftrightarrow$	Doppel-¬
$\neg a \wedge (\neg b \vee (c \wedge \neg a))$	$\Leftrightarrow$	Distributivgesetz
$(\neg a \wedge \neg b) \vee (\neg a \wedge (c \wedge \neg a))$	$\Leftrightarrow$	Doppeltes $\neg a$
$(\neg a \wedge \neg b) \vee (\neg a \wedge c)$		Fertig!

3.6 KV-Diagramme

KV-Diagramme (***Karnaugh-Veitch-Diagramme***, benannt nach ihren Erfindern) sind ein Hilfsmittel, um zu einem gegebenen aussagenlogischen Ausdruck (bzw. einer Wahrheitstabelle) eine möglichst einfache DNF (disjunktive Normalform) zu finden.

Grundidee:

- Ein KV-Diagramm ist eine quadratische oder rechteckige Anordnung von Kästchen.
- Jedes Kästchen stellt genau eine Kombination von Belegungen der Variablen mit `W` oder `F` dar.
- Jedes Kästchen enthält den Wahrheitswert `W` oder `F` des Gesamtausdrucks für diese Belegung.
- Die Kästchen sind mit einer bestimmten Systematik angeordnet.
- Aus der Anordnung der `W`'s im Diagramm läßt sich eine DNF für den ursprünglichen Ausdruck ermitteln.

Vorgehen im Detail:

Ermitteln der Variablen: Die Größe des KV-Diagramms hängt von der Anzahl der Variablen im Ausdruck ab.

Konstruktion des Diagramms: Schrittweise, Variable für Variable, ausgehend von einem einzelnen Kästchen, bis man bei einem Diagramm mit 2^n Kästchen für n Variablen angekommen ist.

Dabei werden die Spalten und Zeilen am Rand mit Variablen oder Negationen von Variablen beschriftet. Diese gelten dann für die ganze Spalte bzw. Zeile. Für ein einzelnes Kästchen gilt die $\wedge$-Verknüpfung aller seiner Spalten- und Zeilenbeschriftungen.

- Um zu einem quadratischen KV-Diagramm eine Variable hinzuzufügen, spiegelt man das Diagramm (samt allen schon bestehenden Spalten-Beschriftungen!) nach rechts. Dann beschriftet man die Spalten des alten Diagramms alle mit der neuen Variable, die Spalten des gespiegelten Diagramms alle mit der Negation der neuen Variable. Die Zeilenbeschriftungen bleiben unverändert.
- Um zu einem rechteckigen KV-Diagramm eine Variable hinzuzufügen, spiegelt man das Diagramm (samt allen schon bestehenden Zeilen-Beschriftungen!) nach unten. Dann beschriftet man die Zeilen des alten Diagramms alle mit der neuen Variable, die Zeilen des gespiegelten Diagramms alle mit der Negation der neuen Variable. Die Spaltenbeschriftungen bleiben unverändert.

Beispiele:
(in jedem Kästchen steht, für welche Kombination von Literalen dieses Kästchen gilt)

$\boldsymbol{a}$	$\boldsymbol{\neg a}$
a	$\neg a$

	$\boldsymbol{a}$	$\boldsymbol{\neg a}$
$\boldsymbol{b}$	$a \wedge b$	$\neg a \wedge b$
$\boldsymbol{\neg b}$	$a \wedge \neg b$	$\neg a \wedge \neg b$

	$\boldsymbol{a}$	$\boldsymbol{\neg a}$	$\boldsymbol{\neg a}$	$\boldsymbol{a}$
	$\boldsymbol{c}$	$\boldsymbol{c}$	$\boldsymbol{\neg c}$	$\boldsymbol{\neg c}$
$\boldsymbol{b}$	$a \wedge b \wedge c$	$\neg a \wedge b \wedge c$	$\neg a \wedge b \wedge \neg c$	$a \wedge b \wedge \neg c$
$\boldsymbol{\neg b}$	$a \wedge \neg b \wedge c$	$\neg a \wedge \neg b \wedge c$	$\neg a \wedge \neg b \wedge \neg c$	$a \wedge \neg b \wedge \neg c$

		$\boldsymbol{a}$	$\boldsymbol{\neg a}$	$\boldsymbol{\neg a}$	$\boldsymbol{a}$
		$\boldsymbol{c}$	$\boldsymbol{c}$	$\boldsymbol{\neg c}$	$\boldsymbol{\neg c}$
$\boldsymbol{b}$	$\boldsymbol{d}$	$a \wedge b \wedge c \wedge d$	$\neg a \wedge b \wedge c \wedge d$	$\neg a \wedge b \wedge \neg c \wedge d$	$a \wedge b \wedge \neg c \wedge d$
$\boldsymbol{\neg b}$	$\boldsymbol{d}$	$a \wedge \neg b \wedge c \wedge d$	$\neg a \wedge \neg b \wedge c \wedge d$	$\neg a \wedge \neg b \wedge \neg c \wedge d$	$a \wedge \neg b \wedge \neg c \wedge d$
$\boldsymbol{\neg b}$	$\boldsymbol{\neg d}$	$a \wedge \neg b \wedge c \wedge \neg d$	$\neg a \wedge \neg b \wedge c \wedge \neg d$	$\neg a \wedge \neg b \wedge \neg c \wedge \neg d$	$a \wedge \neg b \wedge \neg c \wedge \neg d$
$\boldsymbol{b}$	$\boldsymbol{\neg d}$	$a \wedge b \wedge c \wedge \neg d$	$\neg a \wedge b \wedge c \wedge \neg d$	$\neg a \wedge b \wedge \neg c \wedge \neg d$	$a \wedge b \wedge \neg c \wedge \neg d$

Das Wesentliche bei den entstehenden Diagrammen ist, daß sich jedes Kästchen von jedem seiner Nachbar-Kästchen durch die umgekehrte Belegung genau einer Variable unterscheidet!

Ausfüllen der Kästchen: In jedes Kästchen wird nun der Wahrheitswert des Gesamtausdruckes eingetragen, der sich bei der für das Kästchen geltenden Belegung der Variablen ergibt. Für diesen Schritt muß man eventuell eine Wahrheitstabelle zu Hilfe nehmen und die Werte der Ergebnis-Spalte (ganz rechts) entsprechend in das KV-Diagramm eintragen.

Bilden von Schleifen: Alle `W` im Diagramm werden nach folgenden Regeln zu Schleifen zusammengefaßt:

- Es gelten nur quadratische oder rechteckige Schleifen, deren Seitenlängen Zweierpotenzen sind (1, 2, 4, ...), also ein einzelnes Kästchen, 2 benachbarte Kästchen, 4 Kästchen in einer Linie oder als $2*2$-Quadrat, ein $2*4$-Rechteck, aber *nicht* 3 Kästchen in einer Linie oder "um's Eck", ein $2*3$-Rechteck, usw..
- Es gelten auch Schleifen "außen herum" (über die Diagramm-Ränder hinweg von einem Kästchen in der ersten Zeile zum dazugehörigen Kästchen der letzten Zeile oder von einem Kästchen in der Spalte ganz links zum dazugehörigen Kästchen in der Spalte ganz rechts).
- Bei KV-Diagrammen mit einer Seitenlänge ≥ 8 gibt es Einschränkungen, wie Schleifen mit einer Seitenlänge ≥ 4 liegen dürfen, aber die wollen wir hier nicht weiter behandeln.
- Jede Schleife darf nur `W` enthalten, keine `F` (es können also nur benachbarte `W` zusammengefaßt werden).
- Jedes `W` muß in mindestens einer Schleife enthalten sein (es darf auch in mehreren Schleifen enthalten sein!). Ein alleinstehendes `W` bekommt daher seine eigene Schleife der Größe 1.
- Es werden möglichst große Schleifen gebildet.
- Es werden möglichst wenige Schleifen gebildet.

Bilden der Minterme: Jede Schleife entspricht genau einem Minterm ($\wedge$-Verknüpfung von Literalen). Bei einer Schleife der Größe 1 entspricht der Minterm genau der Zeilen- und Spaltenbeschriftung des jeweiligen Kästchens. Sonst enthält der Minterm genau jene Literale, die für alle Kästchen in der Schleife *gleich* sind.

Mit anderen Worten: Kommt eine Variable in irgendeinem Kästchen der Schleife negiert und in einem anderen unnegiert vor, fällt diese Variable im Minterm der Schleife weg. Der Minterm einer Schleife der Größe 2 hat daher eine Variable weniger, der Minterm einer Schleife der Größe 4 hat zwei Variablen weniger usw..

Bilden der DNF: Die Minterme der einzelnen Schleifen werden mit $\vee$ verbunden.

Beispiel: Ermittle zu $\neg(d \wedge a) \wedge \neg(d \wedge \neg b) \wedge (\neg(d \wedge c) \vee \neg(d \wedge \neg c))$ eine möglichst kurze DNF!

1	2	3	4	5	6	7	8	9	10
a	b	c	d	$\neg(d \wedge a)$	$\neg(d \wedge \neg b)$	$\neg(d \wedge c)$	$\neg(d \wedge \neg c)$	$7 \vee 8$	$5 \wedge 6 \wedge 9$
F	F	F	F	W	W	W	W	W	W
F	F	F	W	W	F	W	F	W	F
F	F	W	F	W	W	W	W	W	W
F	F	W	W	W	F	F	W	W	F
F	W	F	F	W	W	W	W	W	W
F	W	F	W	W	W	W	F	W	W
F	W	W	F	W	W	W	W	W	W
F	W	W	W	W	W	F	W	W	W
W	F	F	F	W	W	W	W	W	W
W	F	F	W	F	F	W	F	W	F
W	F	W	F	W	W	W	W	W	W
W	F	W	W	F	F	F	W	W	F
W	W	F	F	W	W	W	W	W	W
W	W	F	W	F	W	W	F	W	F
W	W	W	F	W	W	W	W	W	W
W	W	W	W	F	W	F	W	W	F

		a c	$\neg a$ c	$\neg a$ $\neg c$	a $\neg c$
b	d	F	W	W	F
$\neg b$	d	F	F	F	F
$\neg b$	$\neg d$	W	W	W	W
b	$\neg d$	W	W	W	W

Lösung: $\neg d \vee (\neg a \wedge b)$

3.7 Schaltalgebra

Digitale Schaltungen (sowohl auf einem Chip integrierte als auch diskrete, d. h. aus mehreren Bauteilen aufgebaute: Prozessoren, Grafikchips, Anlagensteuerungen, Telefonvermittlungen, ...) bestehen im Wesentlichen aus einigen wenigen Grundelementen, von denen je nach Anforderung Dutzende oder einige Millionen zusammengeschaltet werden.

Diese Grundelemente werden im Folgenden besprochen. Sie realisieren im Wesentlichen die Junktoren der Aussagenlogik sowie Möglichkeiten, Zustände (Wahrheitswerte) zu speichern, d. h. die Logik wird um eine zeitliche Komponente ergänzt.

Die beiden Wahrheitswerte `W` und `F` werden in Schaltungen meist durch zwei verschiedene Spannungen (z. B. +5 V und 0 V, Sprachgebrauch “Low” und “High”) realisiert und in der Schaltalgebra mit `1` und `0` bezeichnet.

3.7.1 Elektrische Schaltkreise

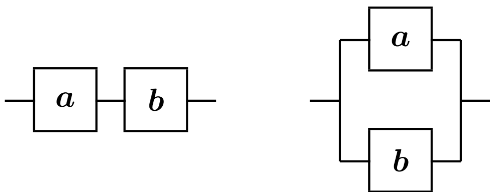
Eine aussagenlogische Variable entspricht einem Schalter[8]: Ist der Schalter ausgeschaltet / offen, so entspricht das `0` oder `F`, ist er eingeschaltet / geschlossen, bedeutet dies `1` oder `W`.

Eine aussagenlogische Formel entspricht einer Verbindung eines oder mehrerer Schalter zu einem Schaltkreis[9].

Das Ergebnis wird durch Stromfluß im Schaltkreis angezeigt: Fließt Strom (leuchtet die Lampe), entspricht das `1` bzw. `W`, fließt kein Strom, gilt das als `0` bzw. `F`.

Bei der ***Serienschaltung*** werden zwei oder mehr Schalter *hintereinander* in der gleichen Leitung angeordnet. Das entspricht einer $\wedge$-Verknüpfung (beide Schalter müssen eingeschaltet sein, damit Strom fließt).

Bei der ***Paralellschaltung*** werden zwei oder mehr Schalter *nebeneinander* in der gleichen Leitung angeordnet. Das entspricht einer $\vee$-Verknüpfung (mindestens ein Schalter muß eingeschaltet sein, damit Strom fließt).



Diese beiden Grundmuster können zur Realisierung komplizierterer Verknüpfungen beliebig kombiniert werden, aber mit Schaltern alleine kann *keine Negation* konstruiert werden.

In der Praxis handelt es sich meist nicht um mechanisch betätigte Schalter, sondern um Schalter, die elektrisch ein- und ausgeschaltet werden, sogenannte *Relais*: Der Schaltkontakt wird elektromagnetisch (d. h. durch Ein- und Ausschalten des Stromes in einer Spule) betätigt.

Da der Kontakt je nach Konstruktion beim Einschalten des Stromes in der Spule entweder geschlossen oder geöffnet werden kann, sind mit Relais auch *Negationen* möglich.

3.7.2 Elektronische Schaltkreise

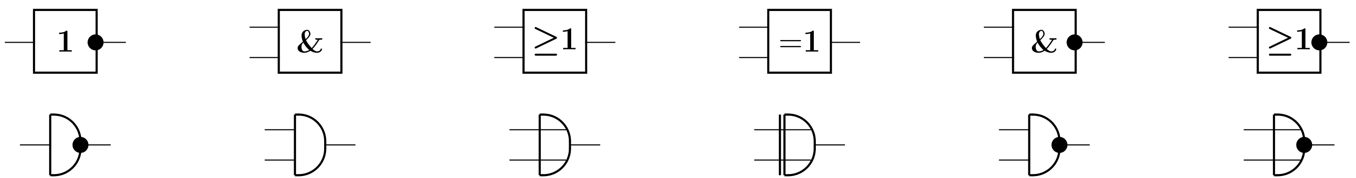
Elektronische Grundschaltkreise werden auch *Gatter* genannt. Sie verknüpfen ein oder mehrere Eingangssignale zu ein oder mehreren Ausgangssignalen. Sie haben daher entsprechend viele Ein- und Ausgangsleitungen, auf denen die Signale durch entsprechende Spannungen dargestellt werden.

Für die Darstellung von Gattern gibt es “alteingeführte” Symbole (die halbrunden in der unteren Zeile) und eine neue Norm (die quadratischen in der oberen Zeile): Nicht, Und, Oder, Exklusives Oder, NAND, NOR. Es gibt auch Gatter mit mehr als zwei Eingängen, sie werden analog dargestellt.

---

[8] Oft zeichnet man statt dem Schaltsymbol für einen Schalter auch einfach ein Kästchen in die Leitung, in das man den Namen der Variablen schreibt.

[9] Kommt eine Variable in der Formel mehrmals vor, braucht man entsprechend viele Kästchen, die alle mit dem gleichen Buchstaben beschriftet werden. Vorstellen muß man sich das als mehrere Schalter, die immer gleichzeitig betätigt werden, bzw. als einen Schalter mit mehreren Kontakten.



Der Punkt zur Darstellung der Negation kann nicht nur am Ausgang verwendet werden, sondern auch an Eingängen (wenn es sich um eine negierte Eingangsvariable handelt)[10]:

Ein Schaltnetz ist eine (sinnvolle) Verbindung mehrerer Gatter, d. h. der Ausgang eines Gatters wird mit ein oder mehreren Eingängen anderer Gatter verbunden. Die Eingangsleitungen des Schaltnetzes entsprechen den Variablen der Formel (und werden auch mit den Variablennamen beschriftet), die Ausgangsleitung des Schaltnetzes ihrem Ergebnis.

Dabei sollten:

- Niemals Eingänge frei bleiben (weil sich sonst Gatter in einem undefinierten Zustand befinden), d. h. jeder Eingang sollte entweder mit einem Gatter-Ausgang oder mit einer Eingangsleitung des Schaltnetzes oder mit einem konstanten 0- oder 1-Signal verbunden sein, wobei ein Ausgang bzw. eine Eingangsleitung auch mit mehreren Eingängen verbunden werden kann.
- Niemals zwei Ausgänge oder zwei Eingangsleitungen direkt miteinander verbunden werden[11].

Es ist erlaubt, Ausgänge auf Eingänge zurückzuführen, sodaß Kreise im Schaltnetz entstehen. Einige solcher Konstrukte sind sinnvoll und in der Praxis von großer Bedeutung (wir werden sie später betrachten), die meisten resultieren allerdings in einem Schaltnetz mit undefiniertem Verhalten.

Physikalische Eigenschaften solcher Netze (Laufzeit, undefinierte Zustände im Moment des Umschaltens) werden in der Schaltalgebra nicht betrachtet. Diese können zu verschiedenen Problemen führen:

- Kurze undefinierte Zustände im Umschaltbereich zwischen 0- und 1-Signalen.
- Kurze Ausgangsimpulse durch verschiedene Laufzeiten des Signals in verschiedenen Pfaden des Schaltnetzes.
- Instabile (schwingende) Schaltungen durch Rückkopplungen von Ausgängen an Eingänge.

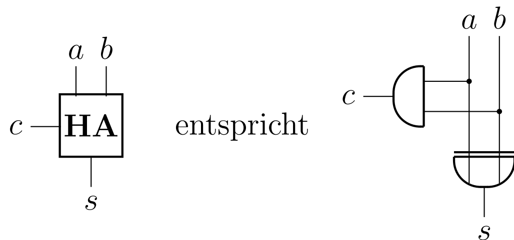
3.8 Halbaddierer und Volladdierer

Der Halbaddierer: Die Aufgabe, aus zwei Bits deren Summe s als $(a \wedge \neg b) \vee (\neg a \wedge b)$ (oder $a \not\Leftrightarrow b$) und deren Übertrag c ("carry") als $a \wedge b$ zu berechnen, kommt so oft vor, daß man für dieses Gatter ein eigenes Symbol eingeführt hat:

---

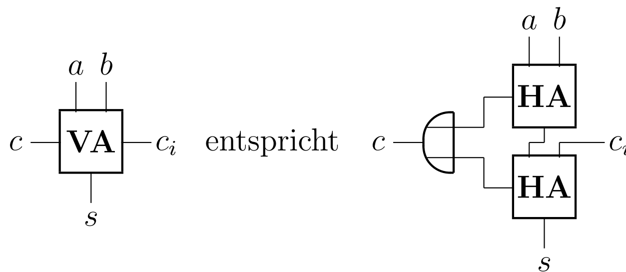
[10] In Schaltplänen bezeichnet ein Punkt bei Kreuzungen und Abzweigungen hingegen nur eine Verbindung (im Unterschied zu unverbundenen Kreuzungen), *keine* Negation!

[11] Es gibt zwar Baureihen von elektronischen Schaltelementen, wo das erlaubt und praktisch sinnvoll ist ("Wired And" bzw. "Wired Or" bei Open Collector Outputs), aber im Allgemeinen ist es technisch nicht sinnvoll (es bildet sich ein Kurzschluß) und logisch nicht definiert (was liefert die Verbindung zweier "gegenteiliger" Ausgänge?) und wird daher in der Schaltalgebra nicht behandelt.

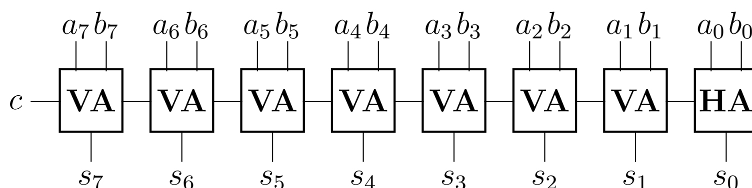


Der Volladdierer: Leider reicht es nicht, zwei Bits zu addieren: Man muß auch den Übertrag c_i ("carry in") der vorigen Stelle berücksichtigen.

Die Schaltung dazu setzt sich wie folgt zusammen (c_i ist ein Eingang, c ist ein Ausgang!):

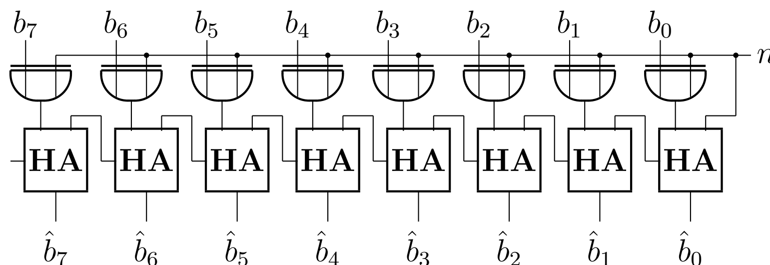


Damit kann man beliebig lange Binärzahlen nach folgendem Schema addieren:



Das Zweier-Komplement: Zum universellen Rechenwerk fehlt noch die Subtraktion. Man könnte dazu ebenso wie einen Halb- und Volladdierer auch einen Halb- und Vollsubtrahierer bauen. Man geht aber anders vor: Nachdem sich ein Integer auch durch die Addition seines Zweierkomplements subtrahieren läßt, muß man dem Addierwerk nur eine Schaltung voranstellen, die den zweiten Operanden abhängig von einem Input n ("negate") unverändert läßt (bei $n = \mathtt{0}$) oder in sein Zweierkomplement verwandelt (bei $n = \mathtt{1}$).

Das ist aber mit Halbaddierern einfach zu erreichen:



Look-Ahead-Carry: Wenn man einen Volladdierer betrachtet, so liegen zwischen c_i und c zumindest 2 Gatter. Für ein Addierwerk für zwei Zahlen mit n Bits bedeutet das, daß das Carry-Signal $2 * (n - 1)$ Gatter durchlaufen muß, bevor es beim c_i des vordersten Volladdierers ankommt: Jeder Volladdierer kann erst dann richtige Ergebnisse für s und c liefern, wenn der Volladdierer für das vorige Bit fertig gerechnet hat. Man nennt das "Ripple Carry": Der Übertrag "rieselt" von Bit zu Bit.

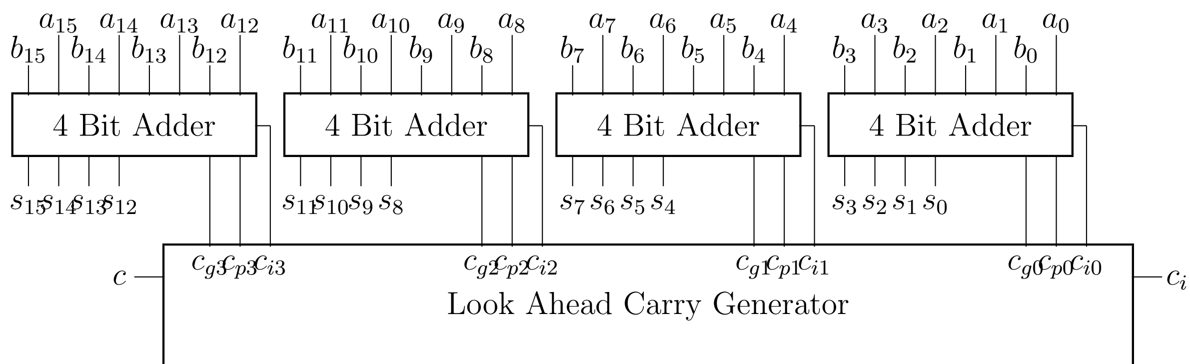
Das ist für 16 oder sogar 64 Bit lange Zahlen natürlich viel zu langsam. In der Praxis sind Addierwerke für Zahlen ab 8 oder 16 Bit Länge daher wie folgt aufgebaut:

- Sie bestehen aus voneinander unabhängigen Addierwerken, die je 4 Bits der beiden Zahlen addieren. Jedes dieser Addierwerke hat einen Eingang c_i ("carry in"), der dem Addierwerk anzeigt, ob von den Bits weiter rechts ein Übertrag daherkommt, aber keinen Ausgang c: Ein bei der Addition der 4 Bits links herauskommender Übertrag wird ignoriert.
- Zusätzlich hat jede 4-Bit-Gruppe eine Schaltung, die aus den $2 * 4$ Input-Bits die beiden Ausgangs-Signale c_g ("carry generate") und c_p ("carry propagate") berechnet: c_g zeigt an, daß für den gegebenen Input in dieser 4-Bit-Gruppe ganz links ein Übertrag entsteht, und c_p zeigt an, daß der Input so beschaffen ist, daß zwar kein neuer Übertrag ganz links entsteht, aber daß sich ein ganz rechts hereinkommender Übertrag nach ganz links fortpflanzen würde.
- Weiters gibt es eine Schaltung (die Look-Ahead-Carry-Logik), die mit den c_g und c_p der einzelnen 4-Bit-Blöcke gefüttert wird und daraus einerseits die c_i für jeden einzelnen Block und andererseits den Übertrag c aus der höchstwertigsten Stelle der gesamten Addition berechnet.

Zeitlich läuft eine Addition also wie folgt ab:

- Im ersten Schritt berechnet jede 4-Bit-Gruppe gleichzeitig für sich ihr c_g und c_p.
- Im zweiten Schritt berechnet die Look-Ahead-Carry-Logik daraus die c_i für jede Gruppe (sowie den Gesamt-Übertrag c).
- Im dritten Schritt addiert jede 4-Bit-Gruppe gleichzeitig für sich ihre Inputs (unter Berücksichtigung ihres hereinkommenden c_i).

Diese Schaltung ist zwar ungleich komplizierter, braucht aber pro Schritt nur in etwa 2–3 Gatterlaufzeiten.

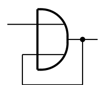


3.9 Flip-Flops

Rückgekoppelte Schaltkreise:

Durch Rückkopplungen erhält man Schaltungen mit "Gedächtnis", d. h. Speicherzellen, Zähler usw..

Der einfachste Fall ist ein rückgekoppeltes ODER-Gatter: Es merkt sich einen einmaligen 1-Impuls am Eingang. Es läßt sich aber nur durch Abschalten der Stromversorgung wieder löschen ...



Nicht jede rückgekoppelte Schaltung hat ein wohldefiniertes Speicherverhalten: Rückgekoppelte Schaltungen können auch instabil sein, d. h. der Ausgang wechselt ständig (ohne äußeres Zutun, d. h. ohne Änderung der Eingänge) seinen Zustand, die Schaltung "schwingt"[12].

Die Beschreibung solcher Schaltungen erfolgt entweder mit einem Zeitdiagramm (x-Achse: Zeit; y-Achse: Signalpegel auf den Ein- und Ausgängen), oder mit einer Wahrheitstabelle, wobei die Variablen mit einem Index n oder $n+1$ versehen sind: "Wenn die Ein- und Ausgänge zum Zeitpunkt n den angegebenen Zustand haben[13], so bekommen die Ausgänge zum Zeitpunkt $n+1$ (einen Takt oder eine Gatterlaufzeit später) den folgenden Zustand."

Das RS-Flip-Flop: ("bistabiler Multivibrator")

... hat 2 Eingänge:

- $\boldsymbol{s}$ ... "set" ("setzen") zum Einschalten
- $\boldsymbol{r}$ ... "reset" ("rücksetzen") zum Ausschalten

und meist 2 Ausgänge $\boldsymbol{q}$ und $\overline{\boldsymbol{q}}$, wobei $\overline{\boldsymbol{q}}$ einfach die Negation von $\boldsymbol{q}$ ist.

Verhalten:

- Ein 1-Signal auf s schaltet das Flip-Flop ein (q wird 1). Ist q ohnehin schon 1, ändert sich nichts.
- Ein 1-Signal auf r schaltet das Flip-Flop aus (q wird 0). Ist q ohnehin schon 0, ändert sich nichts.
- Ist sowohl r als auch s 0, so ändert sich nichts: Das Flip-Flop "merkt" sich den zuletzt erreichten Zustand (eingeschaltet oder ausgeschaltet), q bleibt unverändert (1, wenn zuletzt ein s-Impuls kam, 0, wenn zuletzt ein r-Impuls kam).
- Sind sowohl r als auch s auf 1, sind die Ausgänge undefiniert (man kann nicht zugleich ein- und ausschalten).

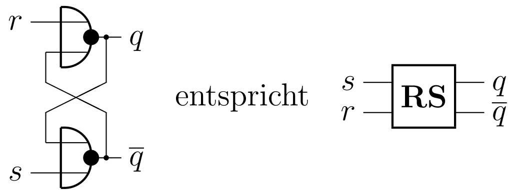
Die Wahrheitstabelle lautet daher:

s_n	r_n	q_n	q_{n+1}	$\overline{q}_{n+1}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	?	?
1	1	1	?	?

Dieses Verhalten läßt sich durch folgende Schaltung realisieren, man hat dafür auch ein Symbol eingeführt:

[12] Faustregel: Ist die Anzahl der Negationen in jedem Kreis der Schaltung gerade, handelt es sich vermutlich um eine stabile, sinnvolle Schaltung. Enthält ein Kreis eine ungerade Anzahl von Negationen, schwingt die Schaltung wahrscheinlich.

[13] Ausgänge mit Index n zählen in der Tabelle also als Eingänge!



Achtung:

- Beim Einschalten der Spannungsversorgung nimmt das RS-Flip-Flop ***zufällig*** einen der beiden Zustände ein. Man muß es daher initialisieren, bevor man das Ausgangssignal verwendet!
- Sind beide Eingänge `1`, so sind beide Ausgänge `0` (und nicht mehr wie definiert die Negation voneinander!). Gehen dann beide Eingänge genau gleichzeitig auf `0`, nimmt das Flip-Flop einen *zufälligen* Zustand ein!

Verwendet man zwei kreuzweise rückgekoppelte NAND-Gatter anstatt der NOR-Gatter, erhält man ein RS-Flip-Flop mit negierten Eingängen: Im Ruhezustand sind $\overline{r}$ und $\overline{s}$ auf `1`, ein `0`-Impuls auf $\overline{r}$ oder $\overline{s}$ bewirkt das Ein- oder Ausschalten, sind beide `0`, ist der Zustand undefiniert.

Varianten:

Getaktetes RS-Flip-Flop: Die beiden Eingänge r und s sind durch je ein UND-Gatter mit einem dritten Eingang c (“clock”, auch t für “time” oder “Takt”, oder e für “enable”) verbunden, werden also nur wirksam, wenn c auf `1` liegt.

Getaktetes D-Flip-Flop: Wie oben, aber mit einem einzigen Eingang: Der Eingang s wird d (“data”) genannt, und der Eingang r wird mit der Negation von d beschaltet.

Solange c `0` ist, wird d ignoriert (und darf sich beliebig ändern). Wenn c auf `1` liegt, wird der aktuelle Zustand von d im Flip-Flop gespeichert (und bleibt es auch, nachdem c wieder auf `0` gegangen ist).

Flankengesteuerte Flip-Flops: Bei den beiden soeben genannten Typen sollte man r und s bzw. d nicht ändern, solange c auf `1` liegt (sonst wechseln die Ausgänge ja wieder zu irgendwelchen Zeitpunkten anstatt wie gewünscht genau im Takt).

Die meisten heute verwendeten Flip-Flops reagieren daher nicht auf ein `1`-Signal am Takt- oder Enable-Eingang, sondern nur auf die steigende (oder fallende) Flanke, d. h. das Testen der Eingangssignale und bzw. das Umschalten der Ausgangssignale erfolgt genau in dem Moment, wo das Taktsignal von `0` auf `1` (oder von `1` auf `0`) springt. Den Rest der Zeit sind die Eingänge egal und die Ausgänge stabil.

Man erreicht das durch zusätzliche Gatter und Verzögerungsschaltungen (damit haben solche Schaltungen kein normales Gatterschaltbild und keine normale Wahrheitstabelle mehr!).

- Das (getaktete) JK-Flip-Flop verhält sich wie ein RS-Flip-Flop, prüft aber seine beiden Eingänge zum Setzen und Rücksetzen nur bei steigender Taktflanke. Zusätzlich ist definiert, daß dieses Flip-Flop bei jedem Taktimpuls in den jeweils anderen Zustand schaltet, wenn beide Eingänge auf `1` liegen.
- Verbindet man daher beide Eingänge mit einem konstanten `1`-Signal, erhält man ein Flip-Flop, daß bei jedem Taktimpuls seinen Zustand wechselt (bei einem normalen RS-Flip-Flop ist dieser Zustand undefiniert).

- Das zweistufige JK-Flip-Flop arbeitet genauso, ändert seine Ausgänge aber erst an der fallenden Taktflanke.
- Faßt man die beiden Eingänge daher wie oben zu einem einzigen d-Eingang zusammen, erhält man ein Schaltglied, das das Eingangssignal um einen halben Takt verzögert.

Der Sinn solcher Schaltungen ist Folgender:

- Bei steigender Taktflanke hat die Schaltung überall gültige, stabile Signalpegel, nichts ändert sich: Es werden in Ruhe die Eingangssignale geprüft und die neuen Zustände berechnet.
- Bei fallender Taktflanke werden dann alle neu berechneten Ausgangszustände aktiviert, die Signalpegel ändern sich, aber die Eingänge werden in diesem Moment ignoriert.

3.10 Anwendungen von Flip-Flops

Schieberegister:

Definition: Ein ***Register*** ist eine Schaltung zur kurzfristigen Speicherung einiger weniger Bits (z. B. 8 oder 32 Bits), sie besteht aus einem Flip-Flop pro Bit.

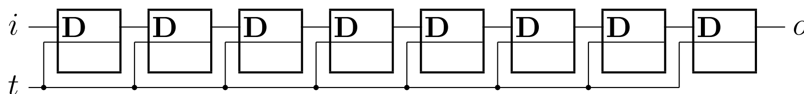
Zweck: Ein Schieberegister verschiebt die gespeicherten Bits bei jedem Takt um eine Stelle: Jedes gespeicherte Bit rutscht nach rechts ins nächste Flip-Flop. Ganz links hat das Schieberegister einen Eingang, bei jedem Takt wird das dort eingegebene Bit in das erste Flip-Flop gespeichert. Ganz rechts hat das Schieberegister einen Ausgang, auf dem bei jedem Takt das Bit des letzten Flip-Flops "herausfällt".

Anwendung: Schieberegister kommen in Computern an vielen Stellen vor:

- Verwandlung von parallelen in serielle Datenströme in seriellen oder USB-Schnittstellen, Netzwerkinterfaces, Speichermedien, digitalen Bildschirmschnittstellen, usw.: Die Flip-Flops werden alle gleichzeitig über ihre Setz- und Rücksetzeingänge mit den Daten geladen, und dann werden die Daten Bit für Bit auf den Ausgang geschoben.
- Rückverwandlung von seriellen in parallele Datenströme (am anderen Ende der Verbindung): Die Daten werden Bit für Bit in das Schieberegister geschoben, und wenn es voll ist, werden alle gleichzeitig an den Ausgängen der Flip-Flops ausgelesen.
- Zwischen- und Verzögerungsspeicher in der CPU, in Motherboard-Chipsets, usw.: Kommen mehrere Bytes (oder 32-bit-Zahlen, Maschinenbefehle o. ä.) schneller, als sie verarbeitet oder gespeichert werden können, so werden sie in aus Schieberegistern aufgebauten Speichern (FIFO's, ein Schieberegister pro Bit Datenbreite) zwischengespeichert und dann im Takt an die nächste Verarbeitungsstufe weitergegeben oder in den Speicher geschrieben.
- Schieberegister spielen eine zentrale Rolle bei der Berechung und Prüfung von CRC- und Solomon-Reed-Codes sowie bei bestimmten Verschlüsselungsverfahren und Zufallszahlen-Generatoren: Der zu prüfende bzw. zu verschlüsselnde Datenstrom wird durch ein Schieberegister geschoben, dabei werden die Daten durch trickreiche logische Verknüpfung mit den Ausgängen der Flip-Flops nach bestimmten mathematischen Regeln verändert.

Aufbau: Ein Schieberegister besteht aus getakteten D-Flip-Flops. Die Taktleitung aller Flip-Flops ist mit einem gemeinsamen Taktsignal verbunden, der D-Eingang jedes Flip-Flops ist mit dem Ausgang des vorherigen Flip-Flops verbunden.

Zusätzlich gibt es je nach Anwendung Setz- oder Rücksetzeingänge für die einzelnen Flip-Flops (um alle gleichzeitig zu löschen oder mit einem bestimmten Wert zu laden) und Ausgänge von jedem einzelnen Flip-Flop (um alle im Schieberegister gespeicherten Bits auf einmal auszulesen).

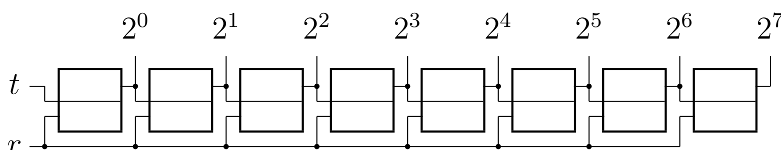


Zählwerke:

Zweck: Zähler zählen die Anzahl der Impulse an ihrem Eingang und liefern das Ergebnis als Dualzahl. Weiters werden sie dazu verwendet, um aus schnellen Taktsignalen langsamere Taktsignale (z. B. mit $\frac{1}{16}$ oder $\frac{1}{100}$ der Frequenz) zu erzeugen (in Quarzuhren, seriellen Schnittstellen, . . .).

Aufbau: Zum Aufbau eines Zählers verwendet man Flip-Flops, die bei jedem Taktimpuls (bzw. bei der steigenden oder fallenden Flanke des Takteingangs) den Zustand wechseln.

- Im einfachsten Fall verbindet man den Takteingang jedes Flip-Flops mit dem Ausgang des vorherigen Flip-Flops (bei Flip-Flops, die auf der *fallenden* Flanke schalten).
- Nutzt man die invertierten statt der normalen Ausgänge der Flip-Flops, erhält man einen Rückwärtszähler.
- Normalerweise laufen solche Zähler im Kreis, d. h. wenn der Zähler aus n Flip-Flops besteht und die Zahl $2^n - 1$ erreicht hat, springt er wieder auf 0. Mitunter nutzt man auch Setz- und Rücksetzeingänge der Flip-Flops, um den Zähler gezielt auf 0 zurücksetzen oder mit einer bestimmten Zahl laden zu können.
- Verknüpft man entweder die Takt-Eingänge oder die Rücksetz-Eingänge der Flip-Flops mit zusätzlichen Gattern mit den Ausgängen, kann man erreichen, daß der Zähler bei einem bestimmten Wert der Ausgänge auf 0 springt. Auf diese Weise erhält man beispielsweise *Dezimalzähler* (nach 9 kommt 0).
- Ein Problem bei derartigen "asynchronen" Zählern ist, daß nicht alle Ausgänge exakt gleichzeitig umspringen, sondern jeweils um eine Gatterlaufzeit verschoben. Bei *synchronen* Zählern wird das Eingangssignal (verknüpft mit den Ausgängen der vorigen Flip-Flops) auf die Eingänge aller Flip-Flops gleichzeitig geführt, damit alle gleichzeitig ihren Zustand ändern.



Serienaddierer:

Zweck: Der Serienaddierer hat zwei Eingänge. Auf jedem Eingang werden die Bits einer Dualzahl der Reihe nach (von rechts nach links, d. h. niederwertigstes Bit zuerst) eingegeben, ein Bit pro Takt. Der Ausgang soll die Summe der beiden Zahlen liefern, ebenfalls pro Takt ein Bit, von rechts nach links.
Wenn man die Eingänge und den Ausgang jeweils mit einem Schieberegister kombiniert, kann man mit dieser Schaltung Dualzahlen addieren.

Aufbau: Die Summe wird durch einen Volladdierer als Summe der beiden Zahlen plus des vom vorigen Takt gespeicherten Übertrags berechnet und sofort ausgegeben. Den Übertrag speichert man in einem getakteten D-Flip-Flop, das ihn mit einem halben Takt

Verzögerung wieder ausgibt und für die Addition der nächsten Bits an den Volladdierer liefert.