VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler
Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

INFORMATION ZUR VORLESUNG
Übungen, Literatur

IT
Informationstechnik

GESCHICHTLICHER HINTERGRUND
Geschichte der Computer
Pioniere der Informatik/Informationstechnik

INFORMATIONSTECHNISCHE GRUNDLAGEN
Informationstechnische Größen
Informationstechnische Präfixe

INFORMATION ZUR VORLESUNG

Übungen, Literatur

IT

Informationstechnik

GESCHICHTLICHER HINTERGRUND

Geschichte der Computer
Pioniere der Informatik/Informationstechnik

INFORMATIONSTECHNISCHE GRUNDLAGEN

Informationstechnische Größen
Informationstechnische Präfixe

Übungsaufgaben:

- Dieses Manuskript enthält in vielen Abschnitten Übungsaufgaben.
- Ein Teil davon wird in der Vorlesung besprochen, die anderen sind zur selbständigen Nachbereitung der Vorlesung und zur Prüfungsvorbereitung gedacht.
- Außerdem gibt es eine Sammlung von weiteren Aufgaben und Musterprüfungen in einem Archiv auf der Web-Seite zur Vorlesung.

Weiterführende Literatur:

Die Folien enthalten teilweise längere Texte als bei Vortragsfolien eigentlich üblich, um auch ein Selbststudium zu ermöglichen. Zusätzliche Informationen finden Sie u.a. in folgenden Büchern:

- H-P. Gumm, M. Sommer: *„Grundlagen der Informatik“*, De Gruyter Verlag, 2016
- R. Hellmann: *„Rechnerarchitektur - Einführung in den Aufbau moderner Computer“*, 2. Auflage, De Gruyter Verlag, 2016
- D. Hoffmann: *„Grundlagen der Technischen Informatik“*, 5. Auflage, Hanser Verlag, 2016
- H. Ernst, J. Schmidt, G. Beneken: *„Grundkurs Informatik“*, 6. Auflage, Springer Vieweg Verlag, 2016

INFORMATION ZUR VORLESUNG
Übungen, Literatur

IT
Informationstechnik

GESCHICHTLICHER HINTERGRUND
Geschichte der Computer
Pioniere der Informatik/Informationstechnik

INFORMATIONSTECHNISCHE GRUNDLAGEN
Informationstechnische Größen
Informationstechnische Präfixe

Informationstechnik
Irgendwas mit Computern ...

So sehen Computer aus ...

... oder so ...

Diese "Tankstelle" enthält auch einen Computer!

... und so ...

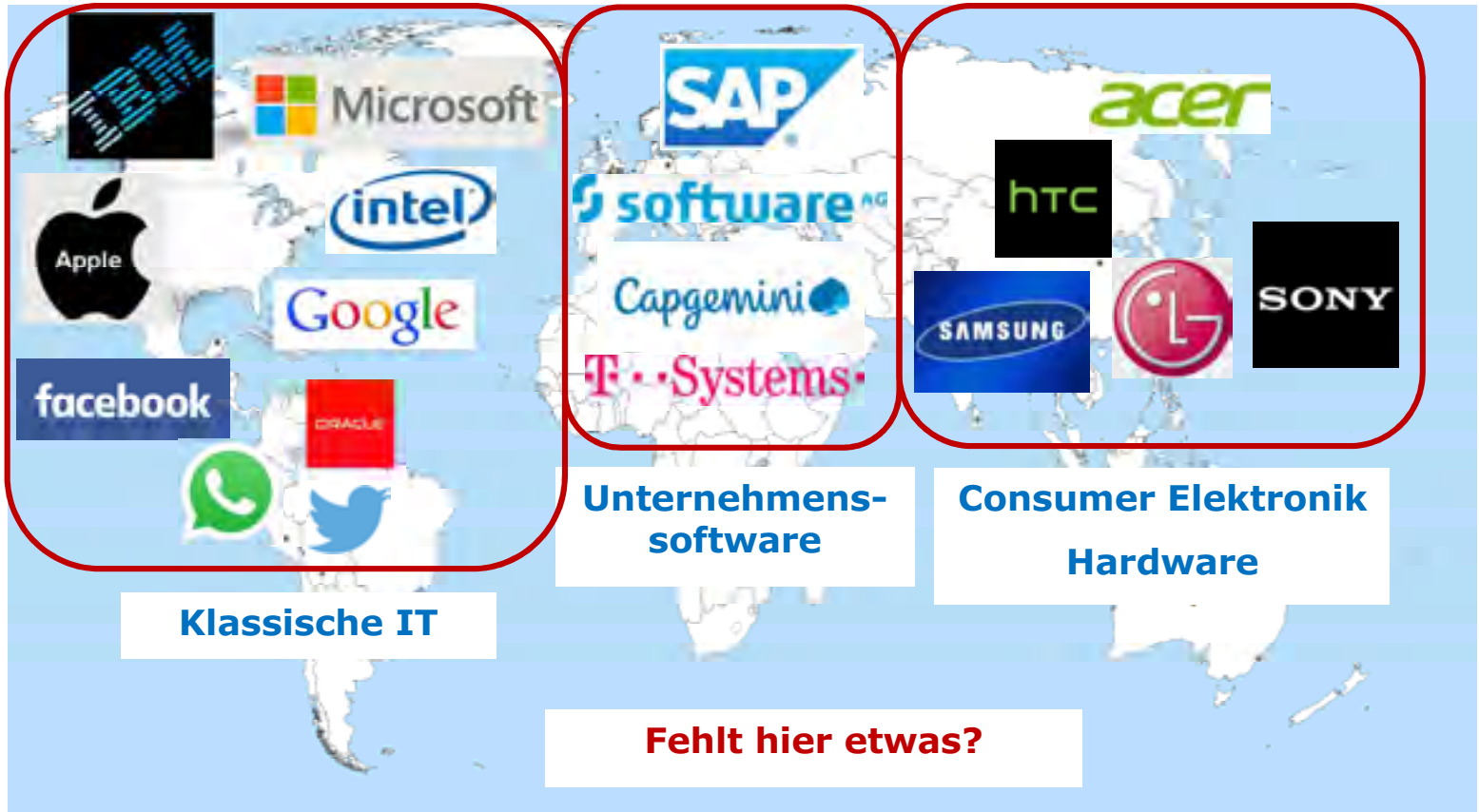
... es werden immer mehr: Internet of Things
Alles wird zum Computer in einem globalen Netzwerk

Wearables

Smart Home

Neue Mobilität und Autonomes Fahren

IT Welt-Champions ...



... und Europa

Informatik

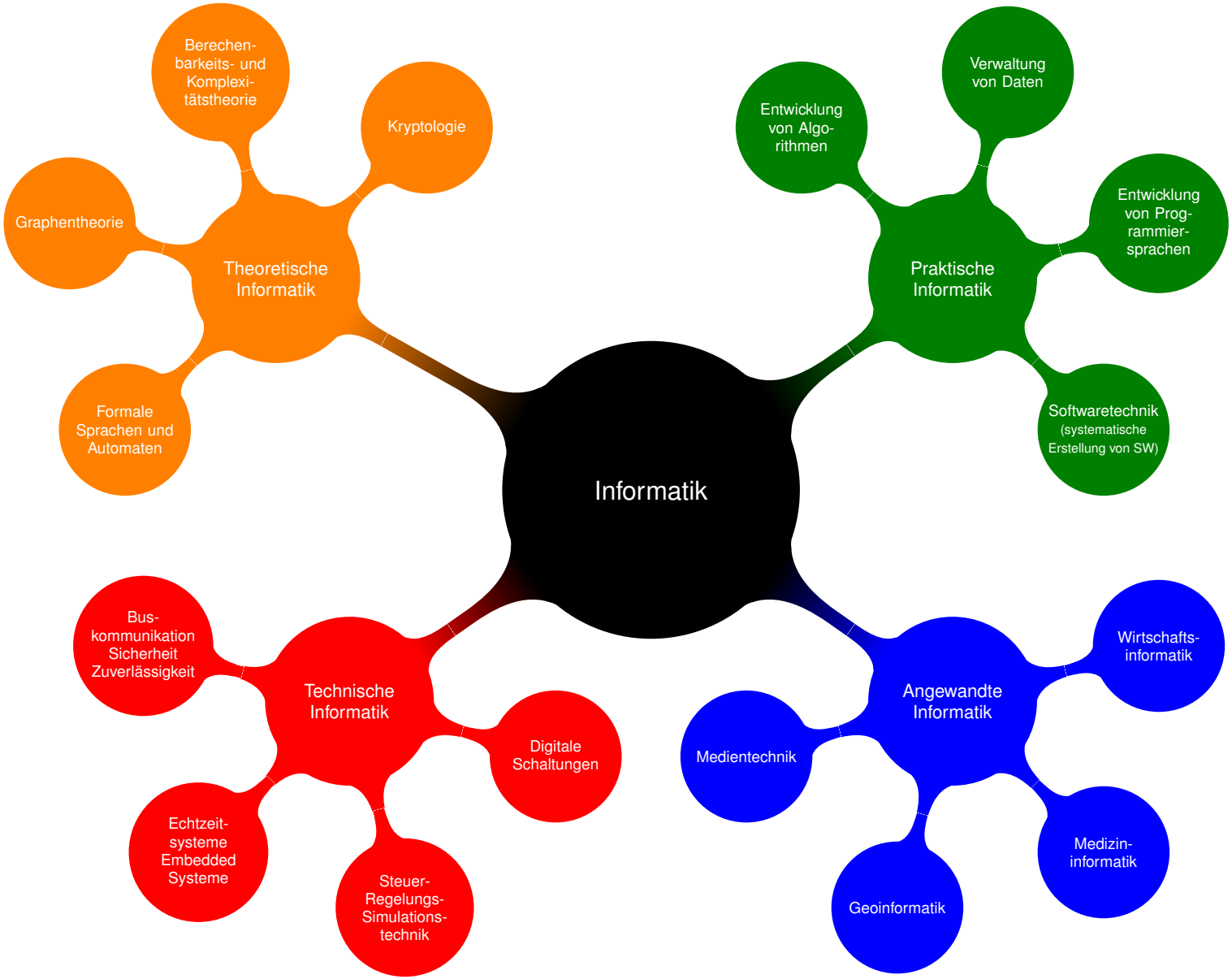
- Kunstwort aus *„INFORMation“* und *„MathemATIK“*
- Duden[1]: „Wissenschaft von den elektronischen **Datenverarbeitung**sanlagen und Grundlagen ihrer Anwendung“
- Englische Bezeichnung für Informatik: *„Computer Science“* - Wissenschaft, die sich mit Rechnern (engl.: Computer) beschäftigt

Informationstechnik

- Synonym: Datenverarbeitung. Bindeglied zwischen der klassischen Elektrotechnik und der Informatik.
- Duden[2]: „Technik der **Erfassung, Übermittlung, Verarbeitung und Speicherung von Informationen durch Computer und Telekommunikationseinrichtungen** (= Internet)“
- VDE[3]: „Die Informationstechnik befasst sich mit der Entwicklung von Methoden, Produkten, Anwendungen und Diensten zur Umsetzung, Verarbeitung, Übermittlung, Speicherung und Sicherheit von Information.“

[1] Quelle Duden: [1]
[2] Quelle Duden: [2]
[3] Quelle VDE-ITG: [3]

Informationstechnik
Teilgebiete der Informatik



Softwaretechnik (engl.: „Software Engineering“):

- Konzeption von Informationssystemen
- Modellierung und **Entwicklung komplexer Software-Systeme**
- Beispiele: Google–Suchmaschine, Amazon-Webshop
- Projekt-, Qualitäts- und Anforderungsmanagement
- Kundenbetreuung oder technischer Service / Vertrieb
- Tätigkeit als Software-Entwickler und Berater als Angestellter oder Selbständiger

Quelle: arbeits-abc [4]

Quelle: e-business [5]

Wirtschaftsinformatik (engl.: „Business Informatics“):

- Anwendungsorientierte integrative Querschnittsdisziplin zwischen **Betriebswirtschaftslehre und Informatik**
- Software zur **Gestaltung und Optimierung von Geschäftsprozessen**, Entwicklung von betriebswirtschaftlicher Standardsoftware
- Beispiel: Software zur Abwicklung des Bestell- und Rechnungswesen oder von Gehaltsabrechnungen
- Unterstützung bei der Analyse unternehmensrelevanter Daten (Business Intelligence), Beratung bei der Einführung IT-gestützter Projekte

Quelle: tse [6]

Quelle: SAP [7]

Technische Informatik (engl.: „Computer Engineering“):

- Software zur **Steuerung und Überwachung technischer Anlagen und Geräte** durch Computer **mit** Hilfe von **Aktuatoren und Sensoren**
- Beispiele: Selbstfahrende Autos, Patientenmonitore für Intensivstationen
- Nähe zur Elektrotechnik/Digitaltechnik/Mechatronik
- Embedded Systems, Bussysteme, Robotik, Simulationstechnik, Sicherheit und Zuverlässigkeit, Echtzeitsysteme

Quelle: KIT [8]

Quelle: Google/VW [9]

Quelle: Bosch [10]

INFORMATION ZUR VORLESUNG
Übungen, Literatur

IT
Informationstechnik

GESCHICHTLICHER HINTERGRUND
Geschichte der Computer
Pioniere der Informatik/Informationstechnik

INFORMATIONSTECHNISCHE GRUNDLAGEN
Informationstechnische Größen
Informationstechnische Präfixe

Historische Rechner Quiz

1. Rechenmaschine von Wilhelm Schickard (1623)
2. Analytical Engine von Charles Babbage (1837)
3. Zuse Z3 (1941)
4. Mark I (1944)
5. IBM PC (1981)
6. IBM Blue Gene/Q (2012)

Rechner A Zuse Z3 (1941):

- Erster universell programmierbarer Digitalrechner weltweit
- **Elektromagnetische Relaistechnik**:
  600 Relais für das Rechenwerk, 1600 Relais für das Speicherwerk
- Speicherkapazität 64 Worten mit je 22 Bit
- Arbeitet mit binärer Gleitkommaarithmetik

Rechner B Analytical Engine (1837):

- **Mechanische Rechenmaschine** zur Lösung polynomialer Funktionen
- Eingabe von Befehlen und Daten über Lochkarten bzw. Metallplatten
- Ausgabe über Drucker, Kurvenplotter und Glocke (als Signal an den Bediener).
- Speicherkapazität 1000 Worte zu 50 Dezimalstellen (ca. 21 kB).
- Recheneinheit („Mühle“ genannt) für vier Grundrechenarten, digitale Gleitkommaarithmetik
- Programmiersprache mit Schleifen und bedingten Verzweigungen

[4] Quelle Wikipedia, CMK: [11], [12], [13], [14], [15], [16]

Rechner C Personal Computer (1981):

- **Erster Computer für den Massenmark**t
- 16-Bit-Prozessor 8088 von Intel (8086 mit externem 8-Bit-Datenbus)
- 4,77 MHz CPU-Takt
- Bis zu 64 KB Arbeitsspeicher
- Ausstattung mit 5,25”-Diskettenlaufwerken mit 160 KB bzw. 320 KB
- Später auch mit 10 MB Festplatte

Rechner D Rechenmaschine von Wilhelm Schickard (1623):

- **Mechanische Rechenmaschine** für astronomische Berechnungen
- Addieren und Subtrahieren von bis zu sechsstelligen Zahlen
- „Speicherüberlauf“ durch Läuten einer Glocke signalisiert
- Berechnung mit Hilfe Napierischer Rechenstäbchen
- Automatische Zehnerübertrag

Rechner E Blue Gene/Q (2012):

- **Höchstleistungsrechner** mit bis zu 20 Petaflops
- Rechner für komplexe Simulationen, z.B. Nuklearreaktionen
- Ca. 50.000 Rechenknoten (16 Rechenkerne pro Knoten)
- 70 PB Plattenspeicher (mit 470 GB/s I/O-Bandbreite)
- Wasserkühlung

Rechner F Mark I (1944):

- **Elektromechanischer Rechner**
- Addition dauerte ca. 0,3 Sekunden
- Multiplikation zweier zehnstelliger Zahlen ca. 6 Sekunden, Division ca. 11 Sekunden
- Dateneingabe durch Lochstreifen und Lochkarten
- Datenausgabe Kartenlocher und elektrische Schreibmaschinen
- Programm dual codiert in einem 24 spurigen Lochstreifen

[5] Quelle Wikipedia, CMK: [11], [12], [13], [14], [15], [16]

Ordnen Sie die Beschreibung der Rechner A bis F und die Namen der Rechner 1 bis 6 den Bildern zu.

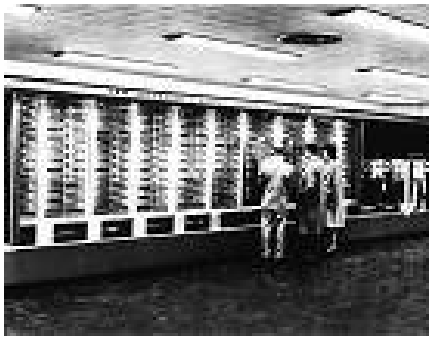
D 1: W. Schickard (1623)

Quelle: Wikipedia [13]

A 3: Zuse Z3 (1941)

Quelle: Wikipedia [11]

F 4: Mark I (1944)



Quelle: CMK [16]

B 2: Analytical Engine (1837)

Quelle: Wikipedia [12]

E 6: Blue Gene (2012)

Quelle: Wikipedia [14]

C 5: IBM PC (1981)

Quelle: Wikipedia [15]

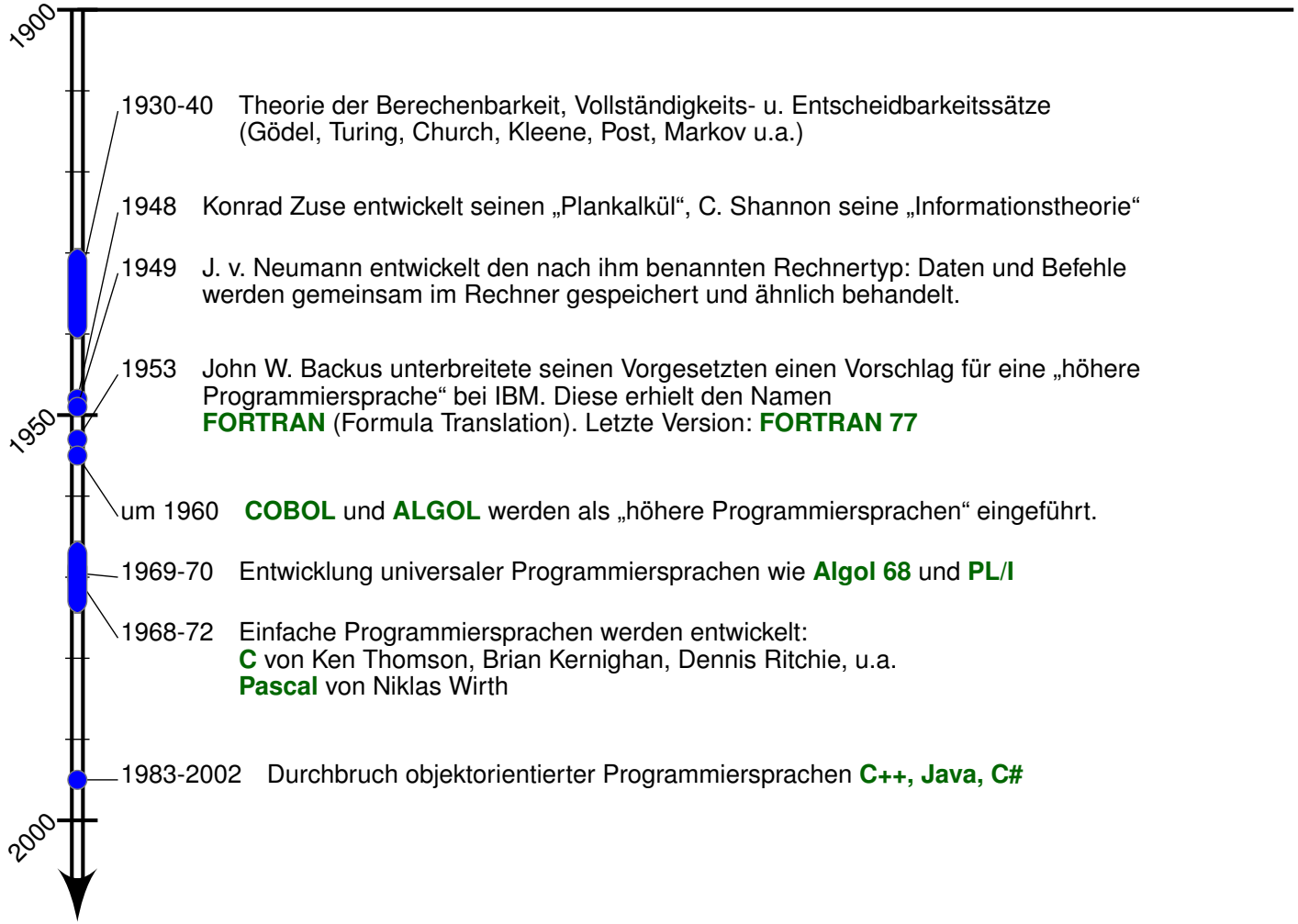
Geschichte der Computer
Zeitleiste 17. bis 19. Jahrhundert[6]

[6] Quelle zkm: [17]

Geschichte der Computer
Zeitleiste 20. und 21. Jahrhundert[7]

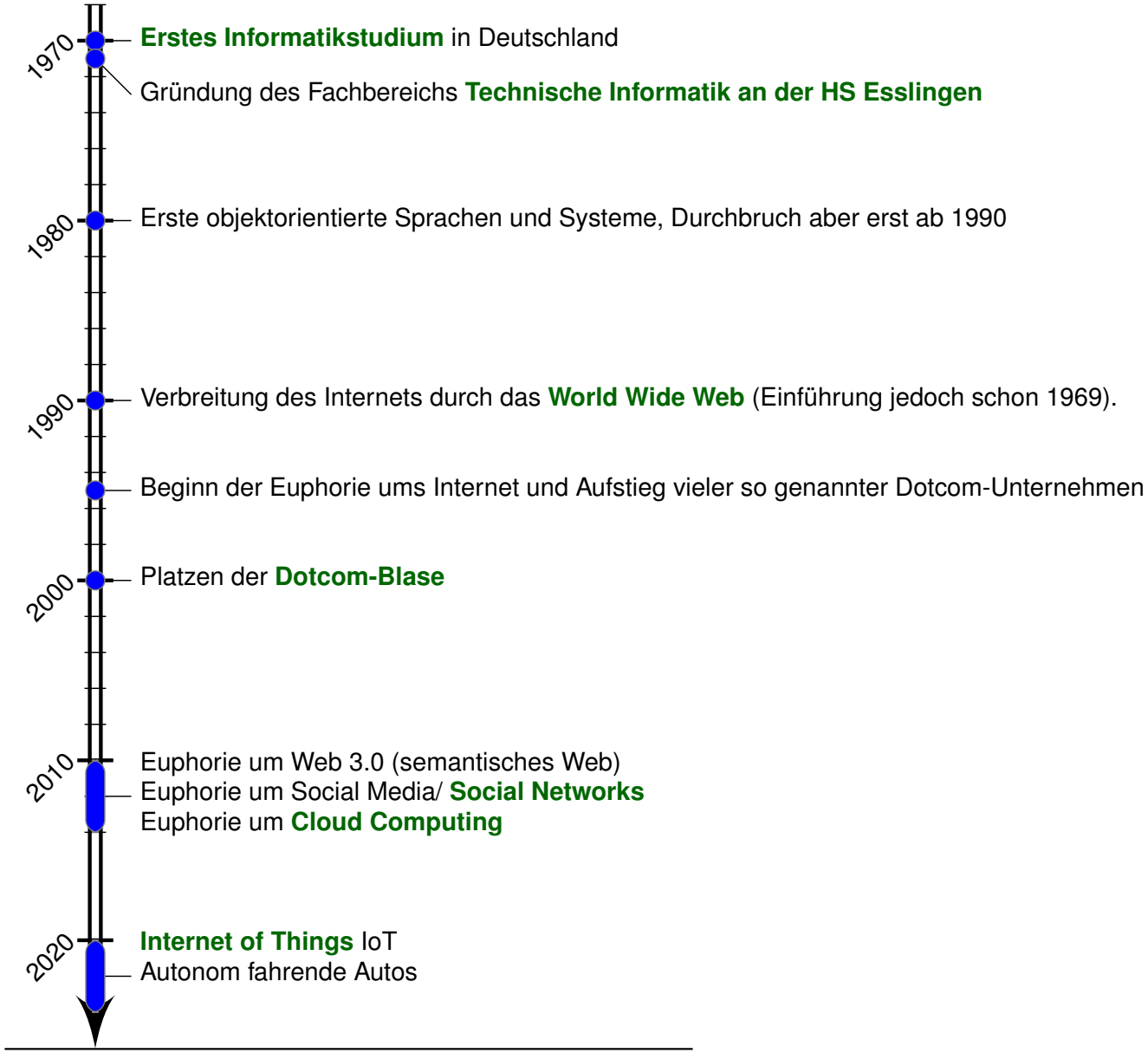
[7] Quelle zkm: [17]

Geschichte der Computer
Zeitleiste Programmiersprachen[8]



[8] Quelle Gumm, Sommer: [18]

Geschichte der Computer
Zeittafel weiterer Ereignisse[9]



[9] Quelle Gumm, Sommer: [18]

Wilhelm Schickard (1592-1635) - Astronom und Mathematiker:

Quelle: Wikipedia [13]

- Professor für Hebräisch und Mathematik (Astronomie) an der Universität Tübingen
- Baute die erste Rechenmaschine zur Berechnung astronomischer Formeln
- Die Maschine beherrschte das Addieren und Subtrahieren von bis zu sechsstelligen Zahlen

[10] Quelle Wikipedia: [13]

Charles Babbage (1791-1871) - Mathematiker, Philosoph, Erfinder, Ökonom

Quelle: Wikipedia [12]

- Professor für Mathematik an der Universität Cambridge
- Entwarf die mechanische Rechenmaschine **Analytical Engine** - der Vorläufer moderner Computer
- Enge Mitarbeiterin war **Ada Lovelace** (siehe unten)

Ada Lovelace (1815-1852) - Pionierin des Programmierens

Quelle: GI [19]

- Augusta Ada King Byron, Countess of Lovelace (Ada Lovelace) war die Tochter des berühmten Dichters Lord Byron
- Erste **zentrale Ideen zur Programmierung** (Testanweisungen, Nutzung von Variablen) für die **Analytical Engine**
- Nach ihr wurde in den 70-iger Jahren die Programmiersprache Ada benannt

Alan Turing (1912-54) - Pionier der Informatik und Kryptoanalytiker

Quelle: GI [19]

- Arbeitete am Design der Automatic Computing Engine (ACE) mit. Diese basierte auf der Analytical Engine des Mathematikers Charles Babbage, dessen Werk Turing zeitlebens bewunderte.
- Entwickelte das Berechenbarkeitsmodell der Turingmaschine, eines der Fundamente der theoretischen Informatik.
- War maßgeblich an der Entzifferung der mit der Enigma verschlüsselten deutschen Funksprüche während des Zweiten Weltkrieges beteiligt.

[12] Quelle GI: [19]

Konrad Zuse (1910-95) - Erfinder des modernen Computers

Quelle: GI [19]

- Studierte zuerst Maschinenbau, dann Architektur und schließlich Bauingenieurwesen
- In der elterlichen Wohnung baute er 1938 die erste binär arbeitende, programmierbare, mechanische Rechenmaschine - die **Z1**.
- 1941 baute er den **Z3**, den ersten vollautomatischen, frei programmierbaren Rechner auf Relaisbasis
- Mit dem **Plankalkül** konzipierte Konrad Zuse 1945 die erste höhere Programmiersprache.

[13] Quelle GI: [19]

Grace Hopper (1906-92) - Pionierin der Informatik

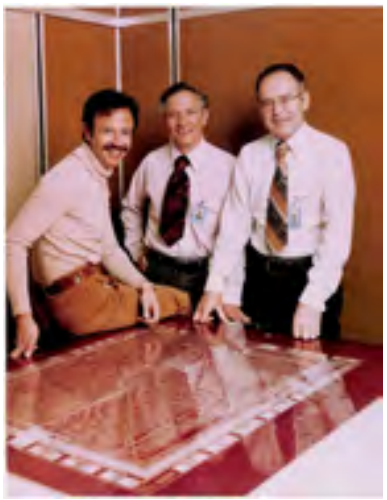
Quelle: GI [19]

- Studierte Mathematik und Physik am Vasser College und schloß Ihr Studium in Yale mit Auszeichnung ab.
- Mitarbeit am **Mark I**, **Mark II** und **UNIVAC I**
- 1952 entwickelte sie den ersten Compiler **(A-0)**.
- Von ihr stammt der Begriff „Bug“ für Softwarefehler. Nachdem eines Tages eine tote Motte in ihrem Logbuch klebte, versah sie dies mit dem Kommentar „First actual case of bug being found“.
- Entwickelte d. Programmiersprache **COBOL** (Common Business Oriented Language).

[14] Quelle GI: [19]

Robert Noyce (1927-1990), Gordon Moore (*1929), Andy Grove (1936-2016)

Gründer von Intel - Demokratisierung der Hardware



Quelle: GI [19]

- 1968 Gründung von Intel
- Erste Produkte: Speicher-Bausteine (RAM) für IBM, erstes EPROM
- Moore'sches Gesetz: Chip-Leistung verdoppelt sich alle zwei Jahre
- 1971 „Erfindung" des **1. Mikroprozessors** Intel 4004 (8008/8080) und des ersten Mikrocontrollers Intel 8048, viele Nachahmerprodukte durch andere Firmen
- 1981 Endgültiger Durchbruch: IBM entscheidet sich, den Intel 8086 im ersten IBM PC einzusetzen. **Demokratisierung der Hardware: Mikroprozessoren und Computer für den Massenmarkt**.
- Heute **größter Halbleiterhersteller der Welt**, durch hohen Forschungs- und Entwicklungsaufwand mit Abstand führende Halbleiterfertigungstechnik

Bill Gates (*1955) und Paul Allen (1953-2018)

Gründer von Microsoft - Demokratisierung der Software

Quelle: GI [19]

- 1975 Gründung von Microsoft
- Erstes Produkt: Basic-Interpreter für Kleincomputer, z.B. Apple I
- 1981 Endgültiger Durchbruch: IBM entscheidet sich für Microsoft DOS als Betriebssystem im ersten IBM PC.
- **Demokratisierung der Software: Software für den Massenmarkt**. **Windows** als Standardbetriebssystem, **Office** (Word/Excel/Powerpoint/Access/Outlook) als Standard-Bürosoftware, Web-Browser und Mediaplayer als Standard-Software für Internet-Clients. **Web und SQL Server**, starkes Cloud-Geschäft (**MS Azure**).
- **Größter Softwarehersteller der Welt mit genialem Marketing**, obwohl technologisch oft nur Nachahmerprodukte. Defizite bei Mobilgeräten (Smartphones, Tablets).

Steve Jobs (1955-2011) - Firmengründer und Visionär

Quelle: GI [19]

- 1976 gründete er mit Steve Wozniak und Ronald Wayne die Firma Apple
- 1984 stellte er den **Apple Macintosh** vor - der erste kommerziell erfolgreiche Computer mit einer grafischen Oberfläche
- 1985 verließ er Apple und gründete 1986 mit Edwin Catmull d. Computer-Zeichentrickfilm Studio Pixar Inc.
- Erste vollständig computeranimierte Kinofilme „Toy Story“, „Findet Nemo“ und viele weitere Oscar-normierte Filme
- 1996 kehrte er zurück um Apple zu retten.
- Innovationsführer bei **Mobilgeräten** mit iPod/iTunes (ab 2001), iPhone (ab 2007) und iPad (ab 2010)
- **Geniales Marketing der geschlossenen Apple Hardware-Software-Welt**: Alles aus einer Hand, Design und Usability vorrangig.

K. Tschirra (1940-2015), H.W. Hector (*1940), D. Hopp (*1940), H. Plattner (*1944), C. Wellenreuther (*1935) - Gründer von SAP

Quelle: Gründerszene

- Arbeiteten als Softwareentwickler bei IBM und gründeten 1972 SAP, weil IBM ihre Ideen für die Software für Lohnabrechnung und Buchhaltung nicht umsetzen wollte.
- SAP ist in allen Bereichen der **betriebswirtschaftlichen Software** aktiv und bietet neben der Anwendungssoftware auch eigene Datenbank- und Cloud-Lösungen. Zielgruppe sind große und mittelgroße Unternehmen, nicht der Endkundenmarkt.
- SAP ist heute nach Umsatz das **weltweit drittgrößte Softwareunternehmen** und nach Marktwert der größte deutsche DAX-Konzern.

[18] Quelle Wikipedia

Linus Torvalds (*1969) - Erfinder von Linux

Quelle: GI [19]

- Studierte Informatik an der Universität Helsinki
- Basierend auf dem Betriebssystem **Minix** (**Unix**-Klone von Andrew Tanenbaum) entwickelte er den **Linux**-Kernel
- Durch die Veröffentlichung des **Linux**-Quellcodes 1991 verbreitete sich dieses in kürzer Zeit weltweit.
- **Durchbruch der Open-Source Bewegung**. Heute Standard-Betriebssystem für Web-Server, Basis für das Smartphone-Betriebssystem Google Android

[19] Quelle GI: [19]

Larry Page und Sergey Brin (*1973) - Gründer von Google

Quelle: GI [19]

- Getrieben von der Idee das gesamte World Wide Web zu analysieren, entwickelten die beiden Doktoranden der Stanford University einen Web Crawler zum Suchen von Dokumenten im Web.
- Heute **weltweit führende Suchmaschine**, aggressives Sammeln von Nutzerdaten zu Werbe- und anderen Zwecken.
- Der Erfolg von Google machte die beiden zu Milliardären.
- Beide sind weiterhin im Unternehmen tätig, L. Page als CEO und S. Brin in der Entwicklung spezieller Projekte, z.B. bei „Google Glass“ oder beim autonomem Fahren von Autos
- Über das Smartphone-Betriebssystem Google **Android** und den Android Store massive Geschäftsinteressen im Mobilgerätemarkt.

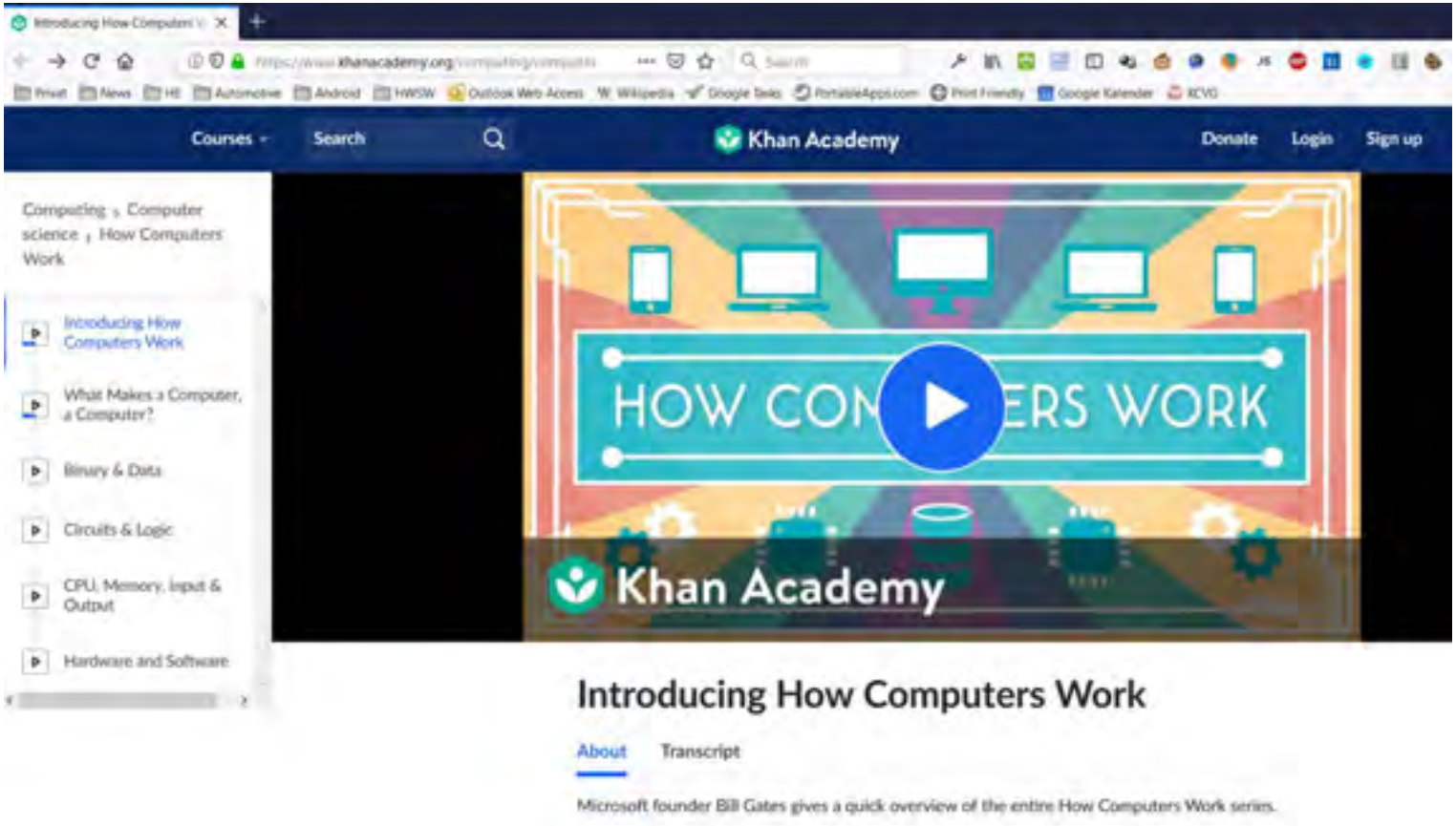
[20] Quelle GI: [19]

Mark Zuckerberg (*1984) - Gründer von Facebook

Quelle: GI [19]

- 2004 Entwicklung des **sozialen Netzwerks** Facebook während seines Studiums an der Harvard University.
- **Facebook** dient als Kommunikationsplattform und Tagebuch, Zukauf von **WhatsApp**
  → **„Dein ganzes Leben im Internet“**.
- Hohe Nutzerzahlen, da durch einfache Bedienung auch für Menschen ohne Computerkenntnisse leicht nutzbar.
- Wirtschaftlich erfolgreich, da Facebook ähnlich wie Google seine Dienste für Endverbraucher zwar kostenlos anbietet, aber deren Nutzerdaten und Nutzerinhalte analysiert und für Online-Werbung und andere kommerzielle Zwecke einsetzt.

[21] Quelle GI: [19]

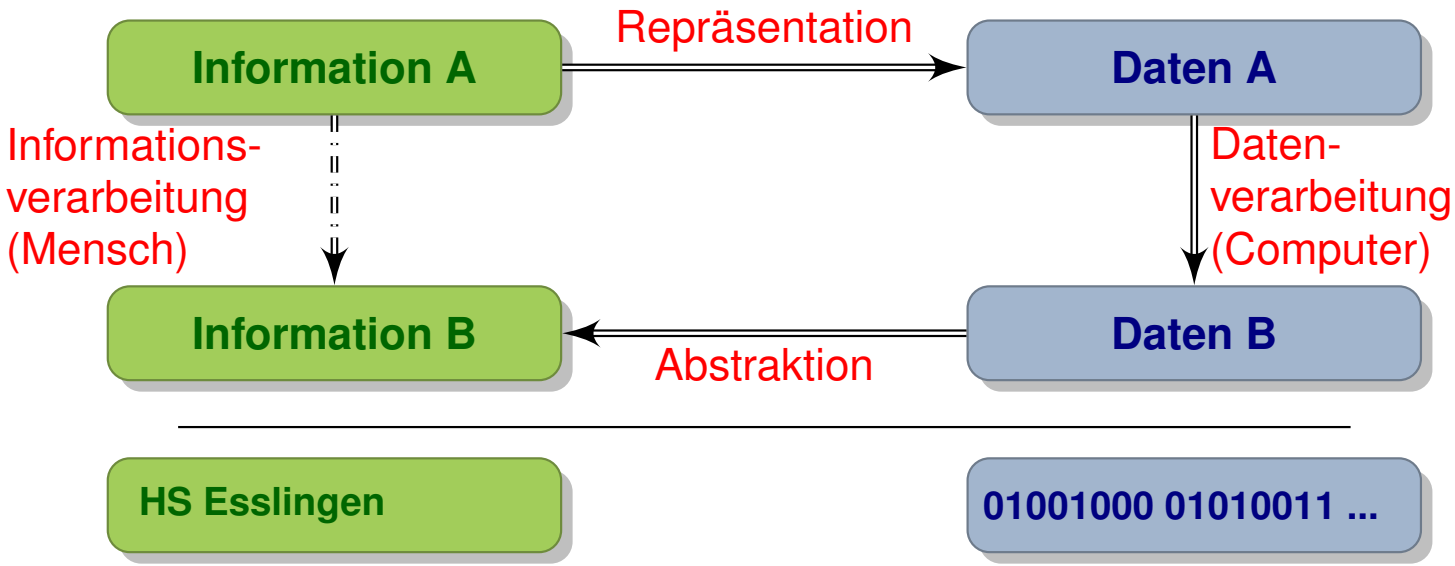


Eine kurze Video-Einführung in die Computertechnik:

`https://www.khanacademy.org/computing/computer-science/how-computers-work2/v/khan-academy-and-codeorg-introducing-how-computers-work`

Quelle: Khan Academy

- Als **Information** (lat.: in-form-are) bezeichnet man den Bedeutungsgehalt von Zeichen, Nachrichten, Mitteilungen etc.
- Zur Verarbeitung werden Informationen durch **Daten** repräsentiert und gespeichert (**Repräsentation**).
- Umgekehrt kann man Information aus Daten zurückgewinnen und interpretieren (**Abstraktion**).
- Rechner **lesen**, **schreiben** und **verknüpfen** Daten durch arithmetische und logische Operationen.
- Die **Datenverarbeitung** wird selbst durch Daten gesteuert, welche die Befehle des Programms codieren (Software = **Programmcode** + Daten).



[22] Quelle Gumm, Sommer: [18]

- Ein Bit ist die kleinstmögliche Einheit der Information und unterscheidet zwei Werte:
  - **ja** oder **nein**,
  - **wahr** oder **falsch**,
  - **schwarz** oder **weiß**,
  - **groß** oder **klein**,
  - . . .
- Zur Codierung genügt ein Code mit den zwei Zeichen mit den zwei Zeichen **1** und **0**. (**Binärer Code**)
- Technisch realisiert man diese Codierung z.B. durch
  - elektrische Ladungen: **0 = ungeladen**, **1 = geladen**
  - elektrische Spannungen: **0 = 0 Volt**, **1 = 5 Volt**
  - Magnetisierungen: **0 = unmagnetisiert**, **1 = magnetisiert**
  - Optische Marken: **0 = dunkel**, **1 = hell**

[23] Quelle Gumm, Sommer: [18]

Informationstechnische Grössen
Bitfolgen[24]

- Enthält eine Information mehr als zwei Werte, benötigt man mehrere Bits.
- Die Frage etwa, aus welcher Himmelsrichtung der Wind weht, lässt sich z.B. in **8** Richtungen unterteilen:
  Nord, **Süd**, **Ost**, **West**, **Südost**, **Nordwest**, **Nordost**, **Südwest**
- Jede beliebige eindeutige Zuordnung der Himmelsrichtungen zu einer Bitfolge kann als Codierung der Windrichtungen dienen, z.B.

  000 = **Nord** 001 = **Nordost**

  010 = **Ost** 011 = **Südost**

  100 = **Süd** 101 = **Südwest**

  110 = **West** 111 = **Nordwest**
- Jedes zusätzliche Bit verdoppelt die Anzahl der möglichen Bitfolgen:

Bei N Bits ergeben sich 2^N mögliche Bitfolgen (Werte/Datenworte)

[24] Quelle Gumm, Sommer: [18]

- Ein Rechner kann besser als ein Mensch mit Bitfolgen umgehen. Zur besseren Lesbarkeit ordnen wir Bitfolgen in Gruppen von 4 Bits, z.B.

  0100 1111 0110 0001 0110 1100 0110 1100_2

  4 F 6 1 6 C 6 C_{16}

- Eine Gruppe von **4 Bits** nennt man ein **Halb-Byte** oder **Nibble**. Da nur 16 verschiedene Nibbles möglich sind, kann man die Nibbles als Ziffern **'0'** bis **'9'** und Buchstaben **'A'** bis **'F'** kürzer schreiben: **Hex-Ziffern**

  $\mathbf{0000_2 = 0_{16}}$ ... $\mathbf{1001_2 = 9_{16}}$

  $\mathbf{1010_2 = A_{16}}$ ... $\mathbf{1111_2 = F_{16}}$

  Vollständige Tabelle siehe nächste Seite.

- Wenn nicht eindeutig klar ist, ob ein Wert dezimal, binär oder hexadezimal ist, sollte man ihn durch einen Index markieren:

  - Index 2 (oder *B*): binär
  - Index 16 (oder *H*): hexadezimal
  - Index 10 (oder *D* oder kein Index): dezimal

  z.B. $\mathbf{1010_B = A_H = 10_D}$ bzw. $\mathbf{1010_2 = A_{16} = 10_{10}}$
  (in C/Java: 0b1010 = 0x0A = 10)

Beispiel: n = 4 bit

Dezimal	Dual	Hexadezimal
0_{10}	0000_2	0_{16}
1_{10}	0001_2	1_{16}
2_{10}	0010_2	2_{16}
3_{10}	0011_2	3_{16}
4_{10}	0100_2	4_{16}
5_{10}	0101_2	5_{16}
6_{10}	0110_2	6_{16}
7_{10}	0111_2	7_{16}
8_{10}	1000_2	8_{16}
9_{10}	1001_2	9_{16}
10_{10}	1010_2	A_{16}
11_{10}	1011_2	B_{16}
12_{10}	1100_2	C_{16}
13_{10}	1101_2	D_{16}
14_{10}	1110_2	E_{16}
15_{10}	1111_2	F_{16}

Mehrstellige Werte werden von links nach rechts geschrieben:
MSB (Most Significant Bit) . . . vorderstes, d.h. höchstwertigstes Bit
LSB (Least Significant Bit) . . . hinterstes, d.h. niederwertigstes Bit

- Ein Rechner arbeitet stets mit Gruppen von Bits, z.B. mit **8 Bit**, **16 Bit**, **32 Bit** oder **64 Bit**.
- Eine Gruppe von **8 Bit** nennt man ein **Byte** (Abkürzung **B**).
- Es gibt $2^8 = 256$ verschiedene Bytes von **0000 0000**$_2$ bis **1111 1111**$_2$.
- Ein Byte kann verwendet werden, um Folgendes zu speichern:
  - Ein codiertes Zeichen in einem Zeichencode, z.B. ASCII,
  - eine natürliche Zahl zwischen 0 und 255,
  - eine ganze Zahl zwischen -128 und +127,
  - die Farbcodierung eines Punktes in einem Bild („Pixel“ = Picture Element).
- Für eine Gruppe von 2, 4 oder 8 Bytes sind auch die Begriffe **Word**, **Double Word** und **Quad Word** im Gebrauch, allerdings ist die Verwendung dieser Begriffe uneinheitlich.

Daten im Speicher

- Den Speicher eines Rechners kann man sich als große Tabelle vorstellen. Jede Zeile der Tabelle (**Speicher-Zelle**) enthält i.d.R. ein Byte.
- Damit der Rechner die Daten leicht findet, werden die Zeilen der Tabelle beginnend bei 0 durchnummeriert (**Adressen**).
- Die Anzahl m der Adressbits bestimmt die **Speichergröße**, z.B.
  16 Bit Adressen $\Rightarrow 2^{16} \cdot 1$ Byte $= 65\,536$ Byte $=$ **64 KiB**
  32 Bit Adressen $\Rightarrow 2^{32} \cdot 1$ Byte $= 4\,294\,967\,296$ Byte $=$ **4 GiB**
- Speichergrößen werden typischerweise mit binären Einheiten (2er Potenzen) in Byte angegeben, z.B. KiB, MiB, GiB, ...

- Eine **Datei** ist eine beliebig lange Folge von Bytes, die zusammen gehören. Z.B. der codierte Text eines Buches oder eines Softwareprogramms.
- Dateien werden meist auf Festplatten oder CD/DVDs gespeichert.
- Jede Information, mit der ein Rechner umgeht (Texte, Zahlen, Musik, Bilder, Programme, ...), muss sich auf irgendeine Weise als Folge von Bytes repräsentieren lassen und kann daher als Datei gespeichert werden.
- Unter der Größe einer Datei versteht man die Anzahl der darin enthaltenen **Bytes**.
- Man verwendet dafür die Abkürzung **B**. Eine Datei der Größe **756 B** enthält also **756 Byte**.
- In einigen Fällen wird die Abkürzung **b** auch für ein **Bit** verwendet. Bei Verwechslungsgefahr sollte man die Einheiten als Byte bzw. als Bit ausschreiben.

[25] Quelle Gumm, Sommer: [18]

- Für Längen- und Zeitangaben werden auch in der Informatik dezimale Einheiten benutzt.
- Ein **4 GHz** Prozessor ist mit $\mathbf{4 \cdot 10^9 = 4\,000\,000\,000\ Hertz}$ (Schwingungen pro Sekunde) getaktet. Ein Takt dauert folglich $\mathbf{\frac{1}{4 \cdot 10^9 Hz} = 0,25 \cdot 10^{-9} sec = 0,250 ns = 250 \cdot 10^{-12} = 250 ps}$
- Für Längenangaben wird neben den metrischen Maßen eine im Amerikanischen immer noch weit verbreitete Einheit verwendet: **1"= 1 in = 1 inch = 1 Zoll = 2,54 cm = 25,4 mm**.
- Für amerikanische Längenmaße hat sich nicht einmal das Dezimalsystem durchgesetzt. So gibt man Teile eines Zolls oft in Brüchen an, wie z.B. **2½"** oder **3½"** (Standarddurchmesser von Festplatten). Bildschirmgrößen dagegen werden in der Regel dezimal angeben, z.B. **15,6"**.

Informationstechnische Präfixe
Dezimalpräfixe[26]

Name	Symbol	Wert	Deutsch	US engl.
Kilo	k	$1000^1 = 10^3 = 1\,000$	Tausend	thousand
Mega	M	$1000^2 = 10^6 = 1\,000\,000$	Million	million
Giga	G	$1000^3 = 10^9 = 1\,000\,000\,000$	Milliarde	billion (!)
Tera	T	$1000^4 = 10^{12} = 1\,000\,000\,000\,000$	Billion	trillion (!)
Peta	P	$1000^5 = 10^{15} = 1\,000\,000\,000\,000\,000$	Billiarde	
Exa	E	$1000^6 = 10^{18} = 1\,000000\,000000\,000000$	Trillion	
Dezi	d	$10^{-1} = 0,1$	Zehntel	tenth
Zenti	c	$10^{-2} = 0,01$	Hundertstel	hundredth
Milli	m	$10^{-3} = 0,001$	Tausendstel	thousandth
Mikro	μ	$10^{-6} = 0,000\,001$	Millionstel	
Nano	n	$10^{-9} = 0,000\,000\,001$	Milliardstel	
Pico	p	$10^{-12} = 0,000\,000\,000\,001$	Billionstel	
Femto	f	$10^{-15} = 0,000\,000\,000\,000\,001$	Billiardstel	

Tabelle: Dezimalpräfixe

[26] Quelle PCHilfen.net: [20]

Informationstechnische Präfixe
Binärpräfixe[26]

Computer arbeiten intern mit Binärzahlen statt Dezimalzahlen. Dafür schlug IEC (International Electrotechnical Commission) folgende **Binärpräfixe** vor:

IEC Name	Symbol	Wert
Kibi	Ki	$1024^1 = 2^{10} = 1\,024$
Mebi	Mi	$1024^2 = 2^{20} = 1\,048\,576$
Gibi	Gi	$1024^3 = 2^{30} = 1\,073\,741\,824$
Tebi	Ti	$1024^4 = 2^{40} = 1\,099\,511\,627\,776$
Pebi	Pi	$1024^5 = 2^{50} = 1\,125\,899\,906\,842\,624$
Exbi	Ei	$1024^6 = 2^{60} = 1\,152\,921\,504\,606\,846\,976$

Zum Vergleich nochmals die **Dezimalpräfixe**:

Name	Symbol	Wert	Deutsch	US engl.
Kilo	k	$1000^1 = 10^3 = 1\,000$	Tausend	thousand
Mega	M	$1000^2 = 10^6 = 1\,000\,000$	Million	million
Giga	G	$1000^3 = 10^9 = 1\,000\,000\,000$	Milliarde	billion (!)
Tera	T	$1000^4 = 10^{12} = 1\,000\,000\,000\,000$	Billion	trillion (!)
Peta	P	$1000^5 = 10^{15} = 1\,000\,000\,000\,000\,000$	Billiarde	
Exa	E	$1000^6 = 10^{18} = 1\,000000\,000000\,000000$	Trillion	

Informationstechnische Präfixe
Missverständnisse[27]

- Hersteller von Festplatten verwenden meist 10er Potenzen für die Plattengröße, z.B. **512 GByte**.
- Betriebssysteme wie Windows oder Linux verwenden aber 2er Potenzen. Der Rechner wird für die „**512 GByte**“ Festplatte daher nur **477 GiByte** Speicherplatz anzeigen.
- Rechnung:

  $$\textbf{512 GByte} = 512 \cdot 10^9 \text{ Byte} = \frac{512 \cdot 10^9 \text{ Byte}}{2^{30} \text{ Byte / GiByte}} \approx \textbf{477 GiByte}$$

- Um solche Missverständnisse zu vermeiden schlug die IEC 1996 die neuen Abkürzungen KiByte, GiByte usw. vor. Sie haben sich aber bisher leider nicht durchgesetzt.

[27] Quelle Gumm, Sommer: [18]

- Ein User möchte eine **2 GiByte** große Datei aus dem Internet herunterladen.
- Die Download-Geschwindigkeit (Bitrate) sei **50 Mbit/s**.

  Hinweis: Der Overhead zur Übertragungssteuerung wird hier vernachlässigt. In der Praxis ist die tatsächliche Download-Geschwindigkeit etwas kleiner als die Bitrate. Außerdem können zusätzliche Verzögerungen durch den Server auftreten.
- Berechnen Sie die Download-Dauer.

$$\text{Datenmenge} = 2 \cdot 1024^3 \text{ Byte} \cdot 8 \frac{\text{Bit}}{\text{Byte}} = 16 \cdot 1024^3 \text{ Bit}$$

$$\text{Bitdauer} = \frac{1}{\text{Bitrate}} = \frac{1}{50 \text{ Mbit/s}} = 20ns$$

$$\text{Download-Dauer} = \text{Datenmenge in Bit} \cdot \text{Bitdauer} = \frac{\text{Datenmenge in Bit}}{\text{Bitrate}}$$

$$= \frac{16 \cdot 1024^3 \text{ Bit}}{50 \cdot 10^6 \frac{\text{Bit}}{\text{s}}} = \underline{\mathbf{344} \text{ s}} \approx 5,7 \text{ min} \quad \text{Best-Case, s.h. Hinweis.}$$

[28] Quelle Gumm, Sommer: [18]

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Logische Grundfunktionen
Rechenregeln
Weitere elementare Funktionen
Funktionen und Funktionsbündel
Disjunktive Normalform
Minimierung
KV-Diagramm
Konjunktive Normal- und Minimalform
Kombinatorische Schaltungen
Realisierungsformen

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

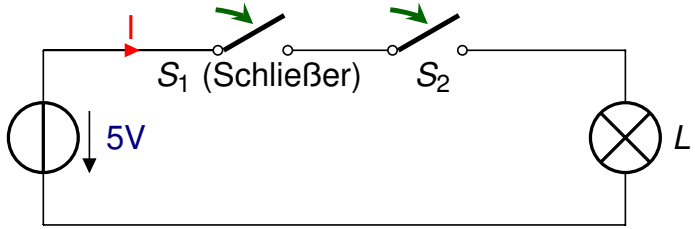
Logische Grundfunktionen
Rechenregeln
Weitere elementare Funktionen
Funktionen und Funktionsbündel
Disjunktive Normalform
Minimierung
KV-Diagramm
Konjunktive Normal- und Minimalform
Kombinatorische Schaltungen
Realisierungsformen

- Computer implementieren ihre Funktion auf Hardwareebene durch sogenannte „Ein/Aus-Schalter“.
- Jeder Schalter lässt sich durch ein Bit beschreiben:

  Beim Zustand **0** spricht man von **falsch (engl. false)**

  Beim Zustand **1** von **wahr (engl. true)**
- Durch die **boolesche Algebra** (George Boole, 1815-1864) lassen sich die Funktionen mathematisch beschreiben.
- Die **booleschen Algebra (Schaltalgebra)** besitzt drei logische Grundfunktionen:
  - Inversion bzw. NEGATION (engl.: **NOT**, in C/Java: ! bzw. ~)
  - Konjunktion bzw. UND (engl.: **AND**, in C/Java: && bzw. &)
  - Disjunktion bzw. ODER (engl.: **OR**, in C/Java; || bzw. |)
- **Aus den drei Grundfunktionen lassen sich alle denkbaren anderen logischen Funktionen aufbauen**.
- Die logischen Grundfunktionen werden auch in praktisch allen Programmiersprachen verwendet.

Elektr. Schaltung:



Math. Definition:

$$(S_1 \wedge S_2) = 1 :\Leftrightarrow (S_1 = 1) \wedge (S_2 = 1)$$

Gesprochen: $(S_1 \wedge S_2 = 1)$ wenn $(S_1 = 1)$ und $(S_2 = 1)$

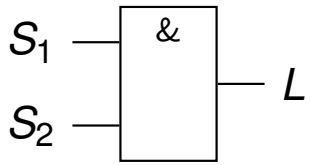
Notationen:

math.	tech.	ASCII	C/Java
$S_1 \wedge S_2$	$S_1 \cdot S_2$	$S_1 \& S_2$	&& bzw. &

Funktionstabelle:

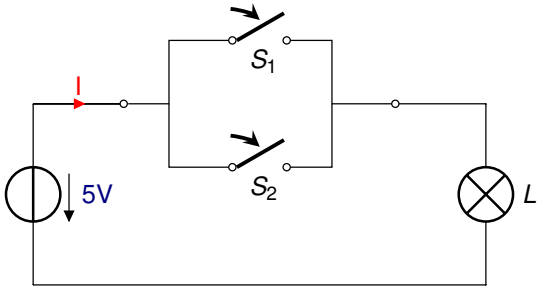
Eingang S_1	Eingang S_2	Ausgang $L = S_1 \wedge S_2$
0	0	0
0	1	0
1	0	0
1	1	1

Schaltzeichen:



AND „Gatter“

Elektr. Schaltung:



Math. Definition:

$$(S_1 \vee S_2) = 1 :\Leftrightarrow (S_1 = 1) \vee (S_2 = 1)$$

Gesprochen: $(S_1 \vee S_2 = 1)$ wenn $(S_1 = 1)$ oder $(S_2 = 1)$

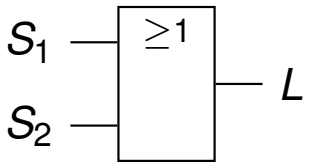
Notationen:

math.	tech.	ASCII	C/Java
$S_1 \vee S_2$	$S_1 + S_2$	$S_1 + S_2$	|| bzw. |

Funktionstabelle:

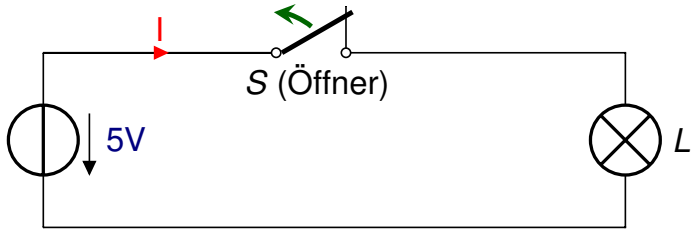
S_1	S_2	$L = S_1 \vee S_2$
0	0	0
0	1	1
1	0	1
1	1	1

Schaltzeichen:



OR „Gatter“

Elektr. Schaltung:



Math. Definition: $(\neg S = 1) :\Leftrightarrow (S = 0)$

Gesprochen: (Not $S = 1$) gilt genau dann, wenn $(S = 0)$

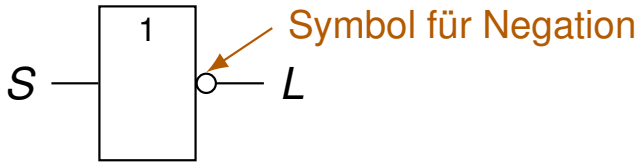
D.) Notationen:

math.	tech.	ASCII	C/Java
$\neg S$	$\overline{S}$	$/S$	! bzw. ~

Funktionstabelle:

Eingang S	Ausgang L
0	1
1	0

Schaltzeichen:



NOT „Gatter“

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Eine Menge **M** mit den drei Verknüpfungen **NEG($\neg$)**, **AND($\wedge$)** und **OR($\vee$)** heißt **Boolesche Algebra**, wenn für alle Elemente a, b, c der Menge **M** folgende **Grundgesetze (Axiome)** jeweils bzgl. AND und OR gelten:

Gesetze der „normalen“ Algebra

	AND	OR
Kommutativgesetze:	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
Vertauschungsgesetz	$\boldsymbol{a \cdot b = b \cdot a}$	$\boldsymbol{a + b = b + a}$
Assoziativgesetze:	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
Verknüpfungsregel	$\boldsymbol{(a \cdot b) \cdot c = a \cdot (b \cdot c)}$	$\boldsymbol{(a + b) + c = a + (b + c)}$
Absorptionsgesetze:	$a \wedge (a \vee b) = a$	$a \vee (a \wedge b) = a$
Distributivgesetze:	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Ausmultiplizieren	$\boldsymbol{a \cdot (b + c) = (a \cdot b) + (a \cdot c)}$	**gilt nicht!!!**
Existenz neutraler Elemente:	$a \wedge 1 = a$	$a \vee 0 = a$
	$\boldsymbol{a \cdot 1 = a}$	$\boldsymbol{a + 0 = a}$
Existenz komplementärer Elemente:	$a \wedge \overline{a} = 0$	$a \vee \overline{a} = 1$

TABELLE: Axiome der Booleschen Algebra

Notation:

Die verschiedenen Notationen können gleichberechtigt verwendet werden.
In der Vorlesung wird vorzugsweise folgende Konvention verwendet:

Negation:	$\overline{\mathbf{A}}$	(in Texten oft auch **/A** aus Zeichensatzgründen)
Konjunktion:	**A B**	(ohne Zeichen) oder $\mathbf{A \cdot B}$ (technische Notation)
Disjunktion:	$\mathbf{A \vee B}$	(math. Notation, damit Verwechslung mit Konjunktion ausgeschlossen werden kann)

Reihenfolge der Abarbeitung eines Ausdruckes:

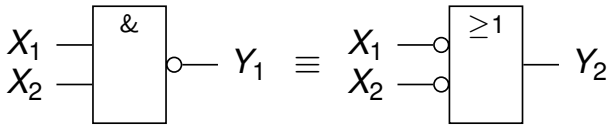
- Die Logik-Operatoren ($\neg$, $\wedge$ und $\vee$) haben theoretisch alle die gleiche Priorität.
- Damit aber nicht unnötig viele Terme eingeklammert werden müssen, verwendet man in der Praxis eine **„Punkt-vor-Strich“-Regel**:
- **„NOT vor AND vor OR“**.

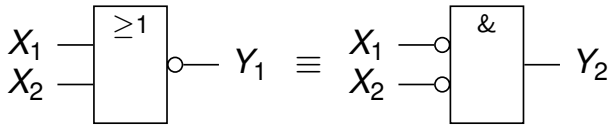
Beispiel: $\mathbf{E = \overline{A} \vee B\ C}$ entspricht $\mathbf{E = (\neg A) \vee (B \wedge C)}$

Umwandlung zwischen AND und OR: De Morgansche Gesetze

(i) $\overline{X_1 \cdot X_2} = \overline{X_1} \vee \overline{X_2}$ (ii) $\overline{X_1 \vee X_2} = \overline{X_1} \cdot \overline{X_2}$

bzw. $X_1 \cdot X_2 = \overline{\overline{X_1} \vee \overline{X_2}}$ bzw. $X_1 \vee X_2 = \overline{\overline{X_1} \cdot \overline{X_2}}$





- Die **De Morganschen Gesetze** gelten auch für n Operanden $X_1 \ldots X_n$.
- Mit den Gesetzen lassen sich **AND**- und **OR**-Ausdrücke in den jeweils anderen Ausdruck umwandeln.

Umwandlung beliebiger Logikfunktionen: Shannonsches Gesetz

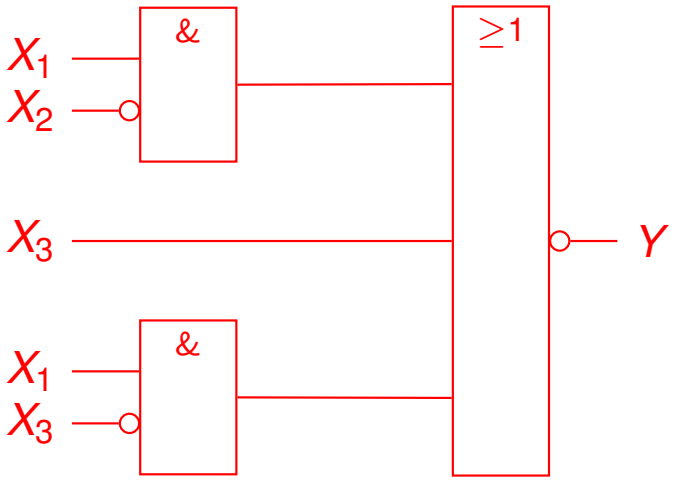
$$\overline{f(X_1, X_2, \ldots, X_n; \wedge, \vee)} = f(\overline{X_1}, \overline{X_2}, \ldots, \overline{X_n}; \vee, \wedge)$$

bzw. $$f(X_1, X_2, \ldots, X_n; \wedge, \vee) = \overline{f(\overline{X_1}, \overline{X_2}, \ldots, \overline{X_n}; \vee, \wedge)}$$

- **Erweiterung der De Morganschen Gesetze**
- Eine Boolesche Funktion mit n Variablen ist gleich der negierten Funktion, wenn man alle **Variablen negiert** und die Operatoren **AND** und **OR vertauscht**.
- **Beispiel:** $X_1\overline{X_2} \vee X_3 \vee X_1\overline{X_3} = \overline{(\overline{X_1} \vee X_2) \cdot \overline{X_3} \cdot (\overline{X_1} \vee X_3)}$

Zeichnen Sie die Schaltung, die durch die folgende Boolesche Gleichung beschrieben wird:

$$Y = \overline{X_1 \overline{X_2} \vee X_3 \vee X_1 \overline{X_3}}$$



Wie in der linearen Algebra können Boolschen Gleichungen durch Umformen mit Hilfe der Rechenregeln bewiesen werden.

Beispiel I:

Behauptung: $\mathbf{X_1\ X_2 \vee X_1\ \overline{X_2} = X_1}$

Beweis: $\mathbf{X_1\ X_2 \vee X_1\ \overline{X_2} = X_1} \underbrace{(X_2 \vee \overline{X_2})}_{=1} = \mathbf{X_1 \cdot 1 = X_1}$ q.e.d.

Beispiel II: **ÜBUNG**

Behauptung: $\mathbf{a\ b \vee \bar{a}\ b\ \bar{c} \vee \bar{a}\ b\ c = b}$

Beweis: $\mathbf{a\ b \vee \bar{a}\ b\ \bar{c} \vee \bar{a}\ b\ c = b}\ \underbrace{[\, a \vee \bar{a}\ \underbrace{(\bar{c} \vee c)}_{=1}\,]}_{=1} = \mathbf{b}$ q.e.d.

Für die endliche Anzahl der Elemente der Booleschen Algebra können im Gegensatz zur linearen Algebra **alle möglichen Belegungen** der Variablen eines Ausdrucks errechnet und **paarweise verglichen** werden (**Beweis durch Enumeration**). Ergeben sich für **zwei Ausdrücke** für alle möglichen Belegungen der Variablen **gleiche Werte**, so sind die Ausdrücke **äquivalen**t.

Beispiel:

Behauptung: $\overline{\mathbf{a}\,\mathbf{b}} = \overline{\mathbf{a}} \vee \overline{\mathbf{b}}$

Beweis:

a	b	$a\,b$	$\overline{a\,b}$	$\overline{a}$	$\overline{b}$	$\overline{a} \vee \overline{b}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Die beiden Spalten $\overline{a\,b}$ und $\overline{a} \vee \overline{b}$ sind identisch. q.e.d.

Beweisen Sie, die Richtigkeit der folgenden Booleschen Gleichung:

$Y = b\,c \vee a\,b\,\bar{c} \vee \bar{a}\,b\,\bar{c} = b$ **ÜBUNG**

1.) Durch Anwendung der Booleschen Rechenregeln:

$$b\,c \vee a\,b\,\bar{c} \vee \bar{a}\,b\,\bar{c} = b\,c \vee b\,\bar{c}\underbrace{(a \vee \bar{a})}_{=1} = b\,c \vee b\,\bar{c} = b\underbrace{(c \vee \bar{c})}_{=1} = b$$

2.) Durch Enumeration:

a	b	c	$b\,c$	$a\,b\,\bar{c}$	$\bar{a}\,b\,\bar{c}$	Y
0	0	0				
0	0	1				
0	1	0			1	1
0	1	1	1			1
1	0	0				
1	0	1				
1	1	0		1		1
1	1	1	1			1

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Exklusiv-ODER XOR (Antivalenz = Ungleichheit)

Math. Definition: $[(S_1 \nleftrightarrow S_2) = 1] :\Leftrightarrow [S_1 \neq S_2]$

Gesprochen: $(S_1 \nleftrightarrow S_2 = 1)$ gilt nur dann, wenn $(S_1 \neq S_2)$.

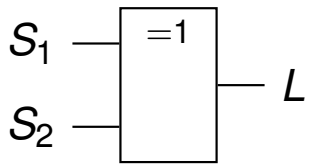
Notationen:

math.	tech.	ASCII	C/Java
$S_1 \nleftrightarrow S_2$	$\oplus$	#	^

Funktionstabelle:

S_1	S_2	$L = S_1 \nleftrightarrow S_2$
0	0	0
0	1	1
1	0	1
1	1	0

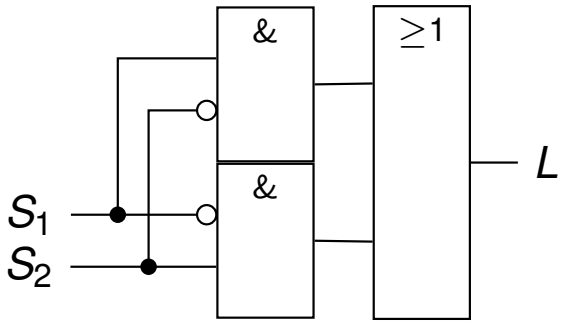
Schaltzeichen:



XOR

Formel: $L = S_1 \nleftrightarrow S_2 = S_1 \cdot \overline{S_2} \vee \overline{S_1} \cdot S_2$

Aufbau:



Rechenregeln:

1. **Kommutativgesetz**
   $A \nleftrightarrow B = B \nleftrightarrow A$
2. **Assoziativgesetz**
   $(A \nleftrightarrow B) \nleftrightarrow C = A \nleftrightarrow (B \nleftrightarrow C) = A \nleftrightarrow B \nleftrightarrow C$
3. **1. Distributivgesetz (bzgl. AND)**
   $A \cdot (B \nleftrightarrow C) = (A \cdot B) \nleftrightarrow (A \cdot C)$
4. **2. Distributivgesetz (bzgl. XOR)** (**nicht gültig!**)
   $A \nleftrightarrow (B \cdot C) \neq (A \nleftrightarrow B) \cdot (A \nleftrightarrow C)$

Beweisen Sie, die Richtigkeit durch Enumeration der folgenden Booleschen Gleichung:

$$\mathbf{a \cdot (b \nleftrightarrow c) = (a \cdot b) \nleftrightarrow (a \cdot c)}$$

ÜBUNG

a	b	c	$b \nleftrightarrow c$	$a\,(b \nleftrightarrow c)$	$a\,b$	$a\,c$	$(a\,b) \nleftrightarrow (a\,c)$
0	0	0					
0	0	1	1				
0	1	0	1				
0	1	1					
1	0	0					
1	0	1	1	1		1	1
1	1	0	1	1	1		1
1	1	1			1	1	

Beweisen Sie, die Richtigkeit durch Enumeration der folgenden Booleschen Gleichung:

$$\mathbf{a \nleftrightarrow (b \wedge c) \neq (a \nleftrightarrow b) \wedge (a \nleftrightarrow c)}$$

ÜBUNG

a	b	c	$b\,c$	$a \nleftrightarrow (b\,c)$	$a \nleftrightarrow b$	$a \nleftrightarrow c$	$(a \nleftrightarrow b) \wedge (a \nleftrightarrow c)$
0	0	0					
0	0	1				1	
0	1	0			1		
0	1	1	1	1	1	1	1
1	0	0		1	1	1	1
1	0	1		1	1		
1	1	0		1		1	
1	1	1	1				

XNOR (Äquivalenz = Gleichheit, NOT XOR)

Math. Definition:

$$[(S_1 \leftrightarrow S_2) = 1] :\Leftrightarrow [S_1 = S_2]$$

Gesprochen: $(S_1 \leftrightarrow S_2 = 1)$ gilt nur dann, wenn $(S_1 = S_2)$.

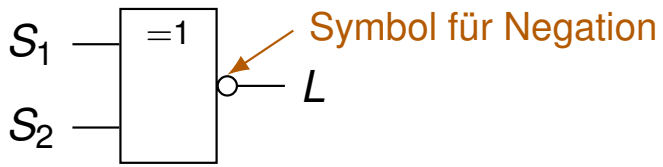
Notationen:

math.	tech.	ASCII	C/Java
$S_1 \leftrightarrow S_2$	$/\oplus$	$/\#$	$==$

Funktionstabelle:

S_1	S_2	$L = S_1 \leftrightarrow S_2$
0	0	1
0	1	0
1	0	0
1	1	1

Schaltzeichen:



XNOR

Negiertes UND : NAND = NOT AND

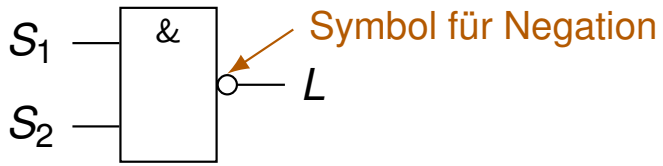
Math. Definition: $L = \overline{S_1 \wedge S_2} = \overline{S_1 \cdot S_2}$

Notationen: keine spezifische Notation

D.) Funktionstabelle:

S_1	S_2	$L = \overline{S_1 \wedge S_2}$
0	0	1
0	1	1
1	0	1
1	1	0

Schaltzeichen:



NAND

Negiertes ODER: NOR = NOT OR

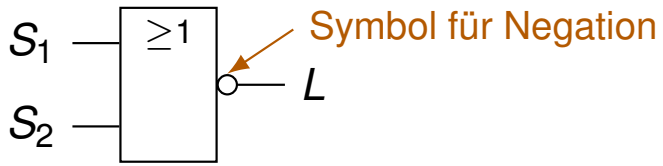
Math. Definition: $L = \overline{S_1 \vee S_2}$

Notationen: keine spezifische Notation

Funktionstabelle:

S_1	S_2	$L = \overline{S_1 \vee S_2}$
0	0	1
0	1	0
1	0	0
1	1	0

Schaltzeichen:



NOR

Aus NAND und NOR lässt sich jede logische Funktion aufbauen

- Aus den Gesetzen von De Morgan folgt:

  Jede **AND**- bzw. **OR**-Verknüpfung lässt sich aus der jeweils anderen Verknüpfung und der **NOT**-Verknüpfung aufbauen

- $C = A \cdot B \Rightarrow C = \overline{\overline{C}} = \overline{\overline{A \cdot B}} = \overline{\overline{A} \vee \overline{B}}$
- $C = A \vee B \Rightarrow C = \overline{\overline{C}} = \overline{\overline{A \vee B}} = \overline{\overline{A} \cdot \overline{B}}$
- Fasst man **AND** und **NOT** bzw. **OR** und **NOT** zu Basisfunktionen **NAND** und **NOR** zusammen, so kann man jede beliebige logische Funktion aus einer einzigen der beiden Basisfunktionen **NAND** bzw. **NOR** aufbauen.

Beispiel: NOT-Funktion aus NAND-Gattern aufgebaut

$B = \overline{A} = \overline{A \cdot A}$

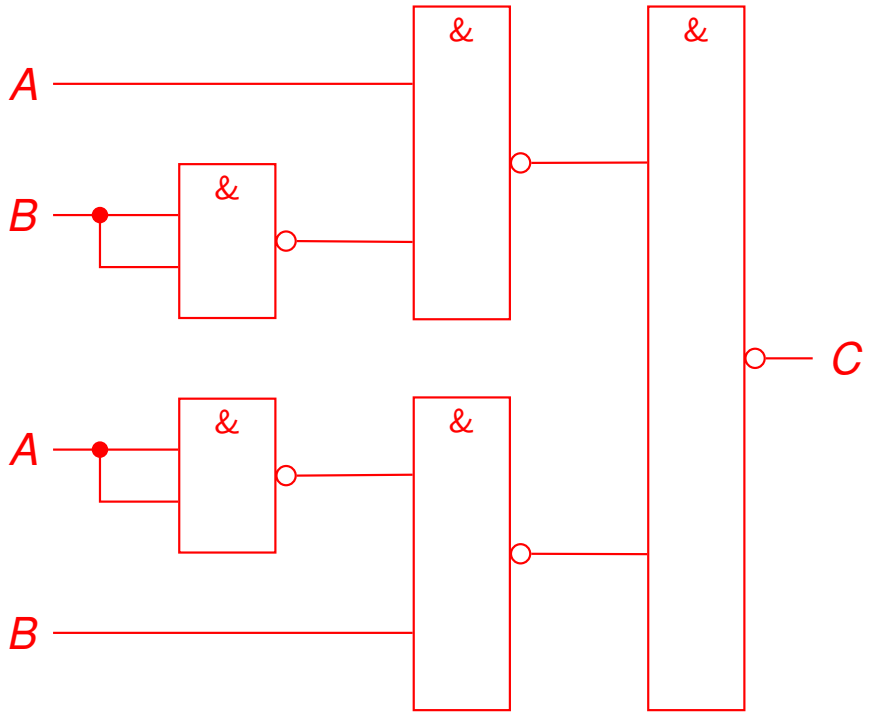
S_1	S_2	$L = \overline{S_1 \wedge S_2}$
0	0	1
0	1	0
1	0	0
1	1	0

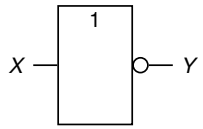
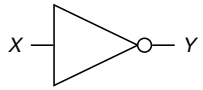
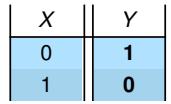
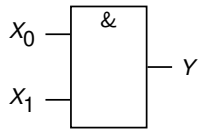
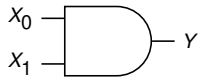
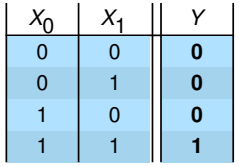
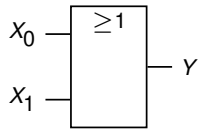
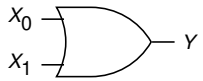
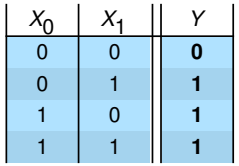
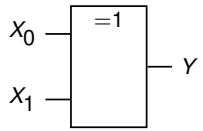
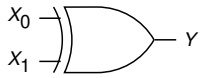
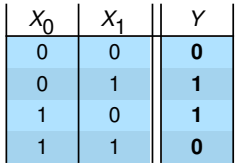
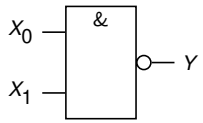
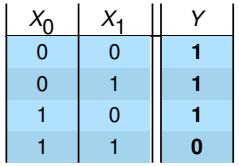
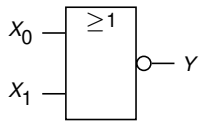
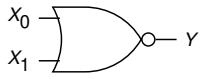
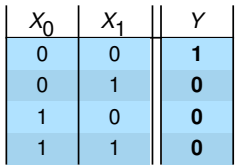
Beispiel: XOR-Funktion (Antivalenz) aus NAND-Gattern aufgebaut

Wandeln Sie die Funktion so um, dass sie durch NAND-Gatter implementiert werden kann und zeichnen Sie die Schaltung:

$$C = A \nleftrightarrow B = A \cdot \overline{B} \vee \overline{A} \cdot B$$

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



Name	Notation	Gatter Schaltzeichen IEC	Gatter Schaltzeichen US	Funktionstabelle
Not NICHT Negation	$Y = \overline{X}$ $Y = \neg X$ $Y = /X$			
AND UND Konjunktion	$Y = X_0 \cdot X_1 = X_0\ X_1$ $Y = X_0 \wedge X_1$ $Y = X_0\ \&\ X_1$			
OR ODER Disjunktion	$Y = X_0 \vee X_1$ $Y = X_0 + X_1$			
XOR Exklusiv-ODER Antivalenz	$Y = X_0 \leftrightarrow X_1$ $Y = X_0 \oplus X_1$ $Y = X_0\ \#\ X_1$			
NAND NICHT-UND	$Y = \overline{X_0 \cdot X_1}$ $Y = \neg(X_0 \wedge X_1)$ $Y = /(X_0\ \&\ X_1)$			
NOR NICHT-ODER	$Y = \overline{X_0 \vee X_1}$ $Y = \neg(X_0 \vee X_1)$ $Y = /(X_0 + X_1)$			

Weitere elementare Funktionen
Zusammenfassung: Rechenregeln

	AND	**OR**	**XOR**
Triviale Regeln	$A \cdot A = A$ $A \cdot \overline{A} = 0$ $A \cdot 1 = A$ $A \cdot 0 = 0$	$A \vee A = A$ $A \vee \overline{A} = 1$ $A \vee 1 = 1$ $A \vee 0 = A$	$A \nleftrightarrow A = 0$ $A \nleftrightarrow \overline{A} = 1$ $A \nleftrightarrow 1 = \overline{A}$ $A \nleftrightarrow 0 = A$
Kommutativgesetze	$A \cdot B = B \cdot A$	$A \vee B = B \vee A$	$A \nleftrightarrow B = B \nleftrightarrow A$
Assoziativgesetze	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A \vee B) \vee C = A \vee (B \vee C)$	$(A \nleftrightarrow B) \nleftrightarrow C = A \nleftrightarrow (B \nleftrightarrow C)$
Distributivgesetze	$A \cdot (B \vee C) = A\ B \vee A\ C$ $A \cdot (\overline{A} \vee B) = A\ B$	$A \vee (B\ C) = (A \vee B)(A \vee C)$ $A \vee (\overline{A}\ B) = A \vee B$	$A\ (B \nleftrightarrow C) = A\ B \nleftrightarrow A\ C$
Absorptionsgesetze	$A \cdot (A \vee B) = A$	$A \vee (A\ B) = A$	
De Morgansche Gesetze	$\overline{A \cdot B} = \overline{A} \vee \overline{B}$	$\overline{A \vee B} = \overline{A} \cdot \overline{B}$	
Shannonsches Gesetz	$\overline{\mathbf{f}(x_1, x_2, \ldots, x_n; \wedge, \vee)} = \mathbf{f}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}; \vee, \wedge)$		

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Funktionen mit 1 Eingangsvariablen: $y = f(x_0)$

(Boolesche Funktionen = Schaltfunktionen)

- Die **Funktion** $y = f(x)$ bildet einen binären Wert x auf einen binären Wert y ab: $f : x \mapsto y$ mit $x, y \in \{0, 1\}$
- Es gibt nur endlich viele Kombinationen der **Eingangswerte** x und der **Ausgangswerte** y und damit eine **begrenzte Anzahl Funktionen**.

Aufzählung aller möglichen Funktionen y = f(x) mit 1 Variablen:

x	$y = f_0(x)$	$y = f_1(x)$	$y = f_2(x)$	$y = f_3(x)$
0	0	0	1	1
1	0	1	0	1

Namen der 4 möglichen Funktionen:

f_0: $y = 0$ Kontradiktion (immer falsch)

f_1: $y = x$ Identität

f_2: $y = \overline{x}$ Negation

f_3: $y = 1$ Tautologie (immer wahr)

Funktionen mit 2 Eingangsvariablen: $y = f(x_1, x_0)$

- Die **Funktion** $y = f(x_1, x_0)$ bildet die zwei binären Werte (x_1, x_0) auf einen binären Wert y ab: $f : x \mapsto y$ mit $x_1, x_0, y \in \{0, 1\}$
- Zu den $2^2 = 4$ **eingangsseitigen** Werte-Kombinationen ergeben sich hier $2^{(2^2)} = 2^4 = 16$ **Kombinationsmöglichkeiten** für die Werte von y.
- Hinweis: Einige Kombinationen sind technisch nicht relevant.

Aufzählung aller möglichen Funktionen $y = f(x_1, x_0)$ mit 2 Variablen:

x_1	x_0	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Wichtige Funktionen:

f_8 ... AND, f_{14} ... OR, f_7 ... NAND, f_1 ... NOR, f_6 ... XOR, f_9 ... XNOR

Vollständige Auflistung siehe nächste Seite.

Funktionsgleichung	**Bezeichnung** (Aussagelogik)
$f_0(x) = 0$	**Kontradiktion** (immer falsch)
$f_1(x) = \overline{x_1 \vee x_0}$	**Neg. Disjunktion (NOR)**
$f_2(x) = \overline{x_1} \wedge x_0$	**Inhibition** (wenn nicht x_1, dann x_0)
$f_3(x) = \overline{x_1}$	**Negation x_1 (NOT)**
$f_4(x) = x_1 \wedge \overline{x_0}$	**Inhibition** (wenn nicht x_0, dann x_1)
$f_5(x) = \overline{x_0}$	**Negation x_0 (NOT)**
$f_6(x) = x_1 \nleftrightarrow x_0$	**Antivalenz (XOR)**
$f_7(x) = \overline{x_1 \wedge x_0}$	**Neg. Konjunktion (NAND)**
$f_8(x) = x_1 \wedge x_0$	**Konjunktion (AND)**
$f_9(x) = x_1 \leftrightarrow x_0$	**Äquivalenz (XNOR)**
$f_{10}(x) = x_0$	**Identität x_0**
$f_{11}(x) = \overline{x_1} \vee x_0$	**Implikation**
$f_{12}(x) = x_1$	**Identität x_1**
$f_{13}(x) = x_1 \vee \overline{x_0}$	**Implikation**
$f_{14}(x) = x_1 \vee x_0$	**Disjunktion (OR)**
$f_{15}(x) = 1$	**Tautologie** (immer wahr)

Funktionen mit n Eingangsvariablen: $y = f(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$

Es gibt 2^n verschiedene Eingangskombinationen und damit $2^{(2^n)}$ verschiedene Funktionen:

Anzahl Variablen	Eingangs-kombinationen	Anzahl Funktionen
1	2	4
2	4	16
3	8	256
...	...	...
8	256	$1,158 \cdot 10^{77}$
...	...	...
n	2^n	$2^{(2^n)}$

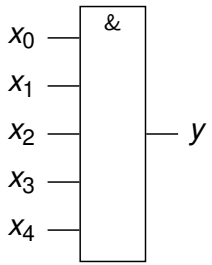
- Namen für Funktionen mit **n** Eingangsvariablen werden aufgrund der Vielfalt **funktionsspezifisch** vergeben.

  Beispiele: **Vergleicher**, **Adressdecoder**, **Multiplexer**, ...

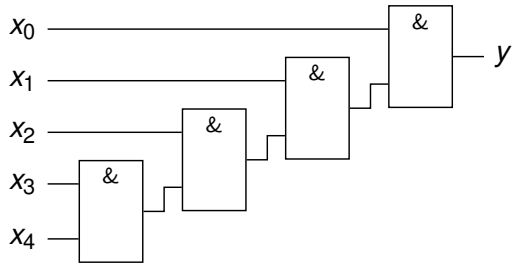
- Bei elementaren Logikfunktionen mit **n** Eingängen wird der Name der Logikfunktion beibehalten.

  Beispiel: AND-Gatter mit **n** Eingängen → Name: **AND(n)**

z.B. n = 5: **AND(5)**



logisch gleichwertig mit



Wenn Laufzeiten vernachlässigt werden können, ergeben sich in beiden Fällen identische Funktionen

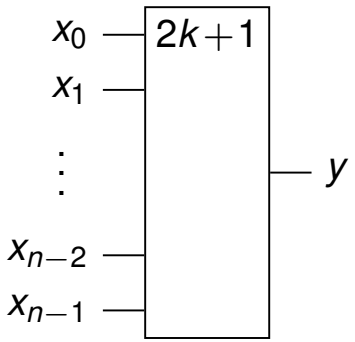
Da Assoziativ- und Kommutativgesetz gelten, spielt die Reihenfolge der Signale in der Eingangsbeschaltung hier keine Rolle.

Weitere Beispiele: **OR(3)**, **NAND(4)**, **XOR(8)** usw.

Antivalenz XOR(n) mit n Eingängen: Ungerade-Funktion 2k+1

- Da auch für die Antivalenz-Funktion Assoziativ- und Kommutativ-Gesetz gelten, gibt es auch eine **n**-stellige XOR-Funktion:
- **XOR(n) = 1 $\Leftrightarrow$ Anzahl der „1“** (Einsen) im Eingangswert ist **ungerade**.
- Funktionsname: „**2k+1**“ (**Ungerade Parität = Odd Parity**)
- Anwendung: Einfache Fehlerüberprüfung in der Datenübertragung

Schaltzeichen:



Beispiel:
(X) = (0,1,0,1,1) $\Rightarrow$ Y = 1
(X) = (1,1,0,1,1) $\Rightarrow$ Y = 0

Äquivalenzfunktion XNOR(n) mit n Eingängen: Gerade-Funktion 2k

- **XNOR(n) = 1 $\Leftrightarrow$ Anzahl der „1“** (Einsen) im Eingangswert ist **gerade**
- Funktionsname: „**2k**“ (**Gerade Parität = Even Parity**)
- Anwendung: Ebenfalls in der Datenübertragung.

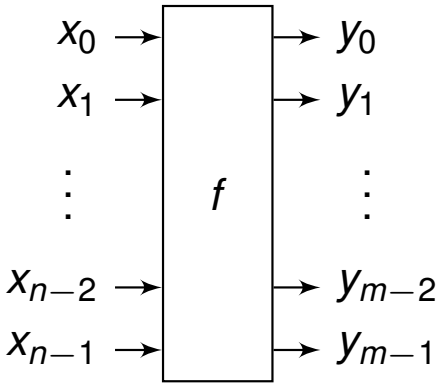
Funktionsbündel mit n Eingängen und m Ausgängen

- Häufig ergeben sich in Abhängigkeit derselben **n** Eingangsvariablen **mehrere Funktionen = Funktionsbündel f**

$$(\mathbf{y}) = (\mathbf{y_{m-1}}, \mathbf{y_{m-2}}, \ldots, \mathbf{y_1}, \mathbf{y_0}) = \mathbf{f}(\mathbf{x_{n-1}}, \mathbf{x_{n-2}}, \ldots, \mathbf{x_1}, \mathbf{x_0}) = \mathbf{f}(\mathbf{x})$$

- Die Eingangsvariablen $(\mathbf{x}) = (\mathbf{x_{n-1}}, \ldots, \mathbf{x_0})$ und die Ausgangsvariablen $(\mathbf{y}) = (\mathbf{y_{m-1}}, \ldots, \mathbf{y_0})$ werden wie in der Linearen Algebra als **Vektoren** oder aus Sicht eines Prozessors als **Datenworte** bezeichnet.
- Jede Funktion $\mathbf{y_i} = \mathbf{f_i}(\mathbf{x_{n-1}}, \ldots, \mathbf{x_0})$ mit $i = 1, \ldots m$ stellt eine eigenständige logische Funktion dar. Die technische Realisierung wird als **Schaltnetz oder kombinatorische Logik** bezeichnet.

Blockschaltbild Schaltnetz



n Eingänge m Ausgänge
m Funktionen

Beispiel Funktionstabelle zu einem Funktionsbündel

x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	1	1	0
0	0	1	1	1	0	0
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	1	0	0	1
1	0	1	1	1	0	1
1	1	0	1	1	1	0
1	1	1	0	0	1	1

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Aufgabenstellung:

Systematische Herleitung der **Funktionsgleichung** für eine logische Funktion bei gegebener **Funktionstabelle**.

Bezeichnungen:

- **Produktterm**: **UND** - Verknüpfung von logischen Variablen.
- **Minterm**: **UND** - Verknüpfung **aller n** (negierten oder nicht negierten) Variablen einer Funktion.

  Jede Zeile einer Funktionstabelle, die den **Ausgangswert 1** ergibt, kann mit einem Minterm beschrieben werden.
- **Maxterm**: **ODER** - Verknüpfung **aller n** (negierten oder nicht negierten) Variablen einer Funktion.

  Jede Zeile einer Funktionstabelle, die den **Ausgangswert 0** ergibt, kann mit einem Maxterm beschrieben werden.

Beispiel:

x_2	x_1	x_0	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Maxterme (ODER) für alle Zeilen mit $y = 0$

$$y = x_2 \vee x_1 \vee \overline{x_0} = 0$$

$$y = \overline{x_2} \vee x_1 \vee x_0 = 0$$

$$y = \overline{x_2} \vee \overline{x_1} \vee x_0 = 0$$

Minterme (UND) für alle Zeilen mit $y = 1$

$$y = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0} = 1$$

$$y = \overline{x_2} \cdot x_1 \cdot \overline{x_0} = 1$$

$$y = \overline{x_2} \cdot x_1 \cdot x_0 = 1$$

$$y = x_2 \cdot \overline{x_1} \cdot x_0 = 1$$

$$y = x_2 \cdot x_1 \cdot x_0 = 1$$

- 'Wird eine der Variablen des Maxterms **1**, wird der **Maxterm 1 (Maximum)**
- Wird eine der Variablen des Minterms **0**, so wird der **Minterm 0 (Minimum)**

- **Disjunktive Normalform DNF**:

  ODER-Verknüpfung aller **Minterme (=UND)** einer Funktionstabelle, für die die Ausgangsvariable den Wert **1** annimmt.

- Interpretation:

  Die **DNF** hat als resultierenden Wert **1** dann, wenn **mindestens** ein **Minterm**, d.h. eine Zeile der Funktionstabelle den Wert **1** annimmt.

Beispiel:

Gegeben: Funktionstabelle

x_2	x_1	x_0	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Gesucht: Disjunktive Normalform (DNF)

$$y_{DNF} = \overline{x_2}\ \overline{x_1}\ \overline{x_0} \vee \overline{x_2}\ x_1\ \overline{x_0} \vee \overline{x_2}\ x_1\ x_0$$

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Aufgabe: Vereinfachung einer logischen Funktion

Eine mit Hilfe der Normalformen gewonnene **Funktionsgleichung** kann meist **vereinfacht** werden.

Beispiel: $y = \overline{x_2}\, x_1\, \overline{x_0} \vee \overline{x_2}\, x_1\, x_0 = \overline{x_2}\, x_1\, (\overline{x_0} \vee x_0) = \overline{x_2}\, x_1$

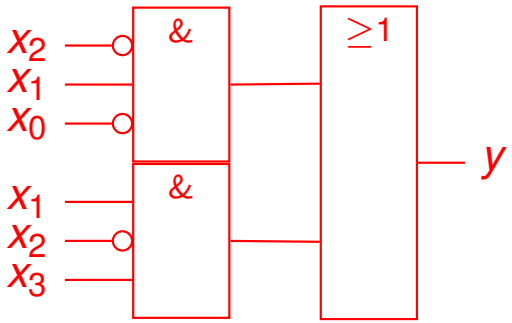
Durch die Vereinfachung soll der Realisierungsaufwand und/oder die Durchlaufzeit/Rechnenzeit minimiert werden.

Maß für den Schaltungs-/Programmieraufwand: Funktionslänge l

- Wird eine logische Funktion ausschließlich unter Verwendung der Grundelemente AND, OR und NOT realisiert, so wird die **Anzahl der Eingänge** aller verwendeten **AND** - und **OR** - Glieder (Gatter) als die **Funktionslänge l** bezeichnet.
- **Inverter** (NOT-Gatter, Negationen) werden dabei generell **nicht mitgezählt**.
- Dabei wird angenommen, dass die Funktion mit der **kleinsten Funktionslänge** den **geringsten Aufwand** in der Realisierung bedeutet.

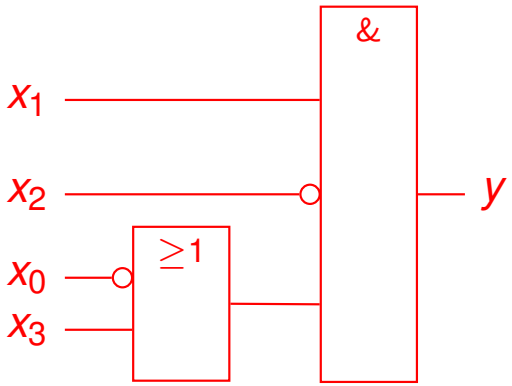
Beispiel A: $y = (\overline{x_2}\ x_1\ \overline{x_0}) \vee (x_3\ \overline{x_2}\ x_1)$ $l = 8$

Schaltung:



Beispiel B: $y = \overline{x_2}\ x_1\ (\overline{x_0} \vee x_3)$ $l = 5$ ÜBUNG

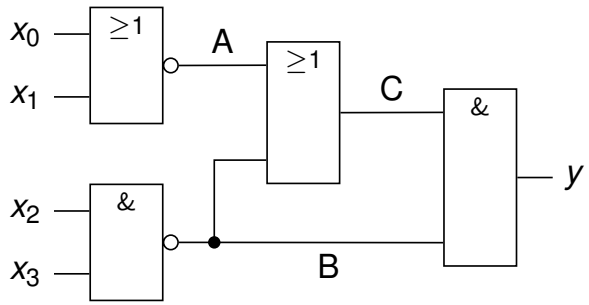
Schaltung:



Maß für die Durchlaufverzögerung: Schachtelungstiefe k

- Durchlaufen bei einer logische Funktion die Eingangssignale mehrere Gatter nacheinander, so wird die von einem Signal **maximal nacheinander durchlaufene Anzahl** der **Gatter** als die **Schachtelungstiefe k** bezeichnet.
- **DNF** (und KNF) haben stets die kleinstmögliche Schachtelungstiefe **$k \leq 2$**.
- **Inverter** (NOT-Gatter, Negationen) werden **nicht mitgezählt**!

Beispiel: Welche Schachtelungstiefe besitzt die Schaltung?



X1 oder X0 nach Y durchläuft 3 Gatter
X2 oder X3 über OR durchläuft 3 Gatter
X2 oder X3 direkt durchläuft 2 Gatter
Unterschiedliche Laufzeiten, aber relevante Schachtelungstiefe k = 3

- Der Aufwand für die Realisierung einer Logikfunktion kann über unterschiedliche Ansätze reduziert werden.
- Minimierung ist auch für Softwareimplementierungen wichtig, wenn z.B. eine if - else if - Kette vereinfacht werden soll.
- Alle Ansätze haben gemeinsam, dass mit einer steigenden Anzahl von Eingangsvariablen die Komplexität stark zunimmt.

Minimierung mit Hilfe der Booleschen Algebra

- Bei **wenigen Eingangsvariablen** kann durch Anwenden der Booleschen Algebra eine minimale Schaltung gefunden werden.

Rechnergestützte Verfahren

- Komplexe Algorithmen zur Schaltungssynthese.
- Beispiel: Verfahren nach **Quine und McCluskey (QMC)**

Graphische Verfahren zur Minimierung

- Anwendung des graphischen Verfahrens von **Karnaugh** und **Veitch** („**KV-Diagramm**“) ermöglicht eine Minimierung der zweistufigen logischen Funktion bei bis zu **5** Eingangsvariablen. → **wird besprochen**

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

Ziel der KV-Diagramme

- Aus einer Funktionstabelle wird graphisch die minimale disjunktive **zweistufige** Funktionsgleichung ermittelt.
- Minimierung der **disjunktiven Normalform** (DNF)
  $\Rightarrow$ **Disjunktive Minimalform** (DMF)
- KV-Diagramme sind für 2, 3 und 4 Eingangsvariablen geläufig und bis 5 Variable anwendbar.

Vorgehen

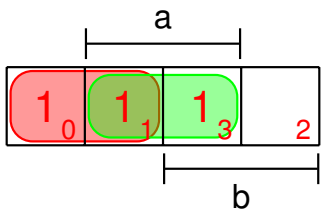
- I. **Mapping**: Abbildung der Funktionstabelle in ein KV-Diagramm
- II. **Übertrag**: Übertrag der Minterme (Zeilen mit y=1) in das KV-Diagramm
- III. **Blockbildung:** Benachbarte Felder mit Wert **1** bilden 2er-Blöcke, diese dann 4er-Blöcke, dann 8er-Blöcke, usw.
- IV. **Produktterm:** Aus den Blöcken werden die Produktterme der DMF aufgestellt

$\Rightarrow$ Vollständige Regeln siehe Seite 2.52

Beispiel: KV-Diagramme mit zwei Variablen

Bestimmen Sie die DMF:

	b	a	y
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	1



$y_{DMF} = \overline{b} \vee a$ mit $l = 2$

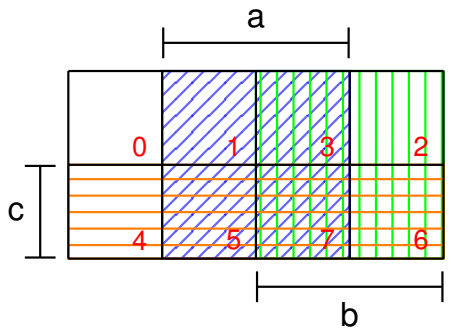
$y_{DNF} = \overline{b}\,\overline{a} \vee \overline{b}\,a \vee b\,a$ mit $l = 9$

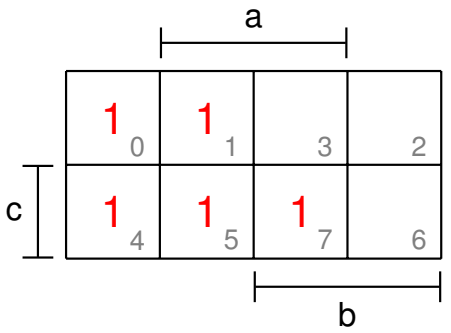
Zu Schritt I Mapping:
Das Mapping *Zeilen Funktionstabelle* $\rightarrow$ *Felder KV-Diagramm* ist nicht eindeutig (Bsp.: Vertauschen der Spalten a und b).
In der Vorlesung verwenden wir stets dasselbe Mapping.

Zu Schritt III Blockbildung:
Felder dürfen bei der Zusammenfassung mehrfach verwendet werden, wenn sich damit größere Blöcke bilden lassen.
hier z.B. Feld 1, damit zwei 2er-Blöcke möglich

Beispiel: KV-Diagramme mit drei Variablen

	c	b	a	y
0	0	0	0	1
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1





Zu Schritt IV: $y_{DMF} = \overline{b} \vee a \cdot c$

Alle Blöcke, auch Einzelblöcke in Produkttermen erfassen

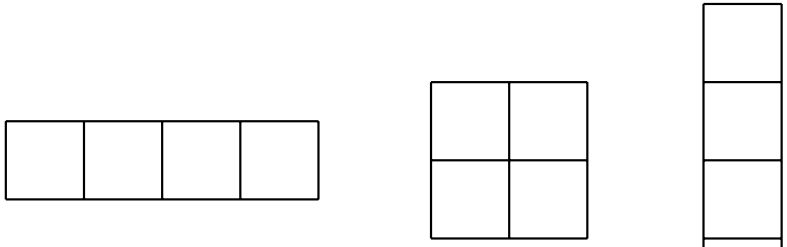
Zu Schritt III Blockbildung:
Blöcke sollten möglichst groß sein, Weitere Regeln siehe nächste Seite.

Zu Schritt III Blockbildung: Erlaubte Zusammenfassungen

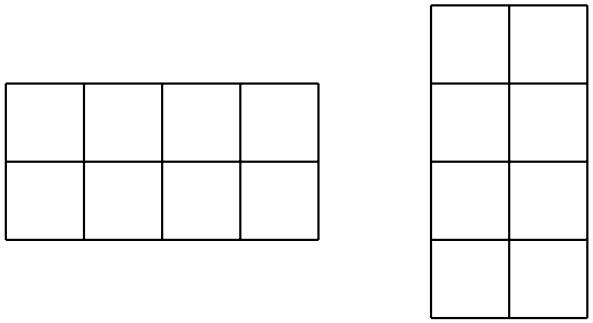
1er Block:

2er Blöcke:

4er Blöcke:

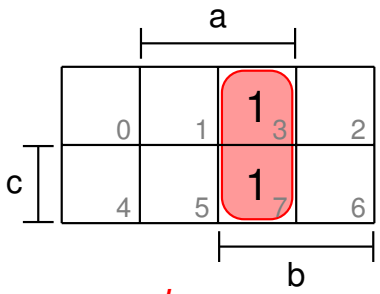


8er Blöcke:

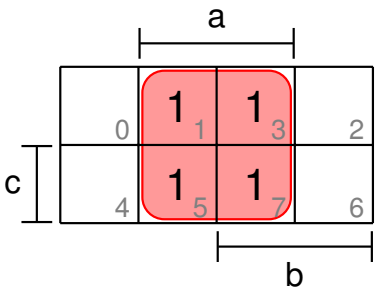


- Blöcke sollten möglichst groß sein, Mehrfachverwendung von Feldern erlaubt.
- Zusammenfassung von Diagonalblöcken nicht erlaubt.
- Felder an den Außenkanten (links - rechts bzw. oben - unten) sind benachbart.

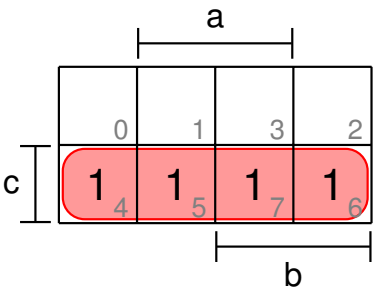
Beispiele von Blockbildungen in einem KV-Diagramme mit drei Variablen



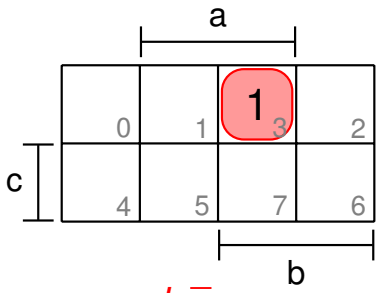
$y_{DMF} = a\,b$



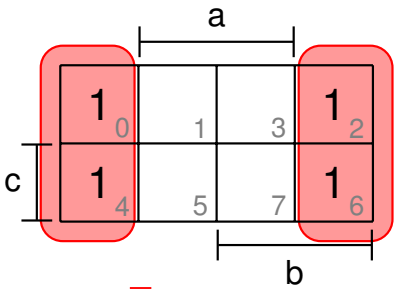
$y_{DMF} = a$



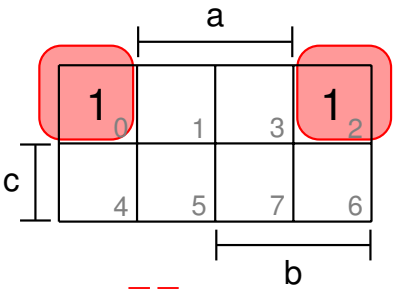
$y_{DMF} = c$



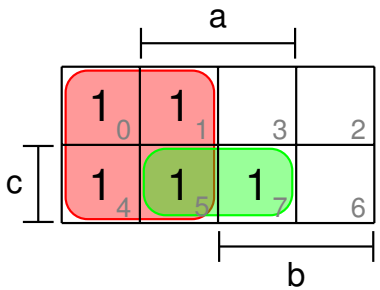
$y_{DMF} = a\,b\,\overline{c}$



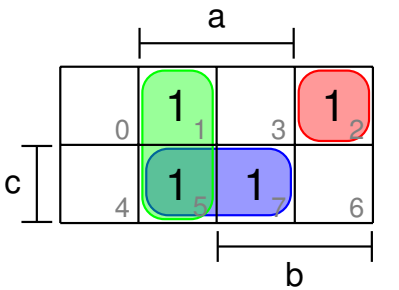
$y_{DMF} = \overline{a}$



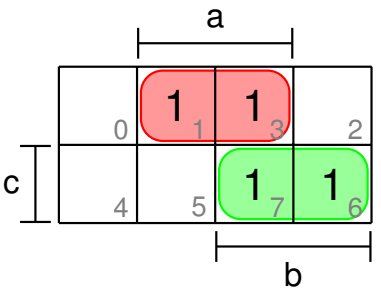
$y_{DMF} = \overline{a}\,\overline{c}$



$y_{DMF} = \overline{b} \vee a\,c$



$y_{DMF} = \overline{a}\,b\,\overline{c} \vee a\,\overline{b} \vee a\,c$

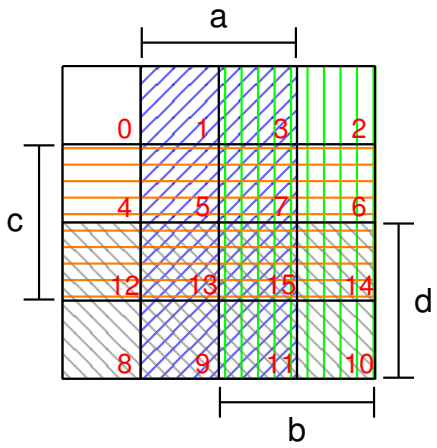


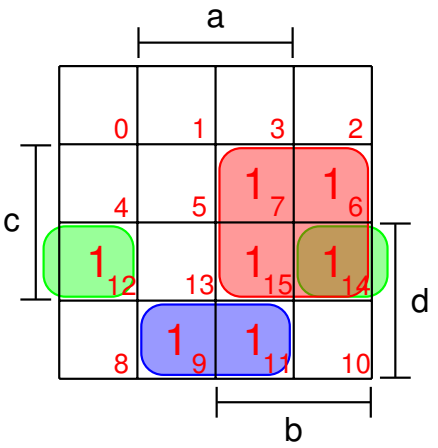
$y_{DMF} = a\,\overline{c} \vee b\,c$

Beispiel: KV-Diagramm mit 4 Variablen: Bestimmen Sie die DMF

ÜBUNG

	d	c	b	a	y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1





$$y_{DMF} = b\,c \vee \bar{a}\,c\,d \vee a\,\bar{c}\,d$$

Zusammenfassung der Regeln

- Schritt I. **Mapping**: Für jede **Zeile** der Funktionstabelle das zugeordnete Feld des KV-Diagramms ermitteln
- Schritt II. **Übertrag**: Alle **Zeilen** der Funktionstabelle mit **y=1** (und mit y=X) in die zugeordneten Felder des KV-Diagramms übertragen.
- Schritt III. **Blockbildung**:
  - **Horizontal** und **vertikal** benachbarte Felder dürfen zu Blöcken zusammengefasst werden. Möglich sind **1er-**, **2er-**, **4er-**, **8er-Blöcke** usw.
  - **Diagonal** nebeneinander liegende Felder sind **nicht benachbart**
  - Felder der **Aussenkanten** sind horizontal und vertikal ebenfalls benachbart und können zusammengefasst werden.
  - Mit der „**1**“ beginnen, die nur **eine Möglichkeit** zur Blockbildung erlaubt. Von hier aus den **größtmöglichen** Block bilden
  - Jedes Feld mit einer „**1**“ muss **mindestens einmal**, **kann** aber auch **mehrmals** erfasst werden
- Schritt IV. **Produktterm und DMF**:
  - Jeder Block liefert einen Produktterm, auch Einzelblöcke.
  - Zur Bildung der DMF werden die Produktterme **disjunktiv** (ODER) miteinander verknüpft

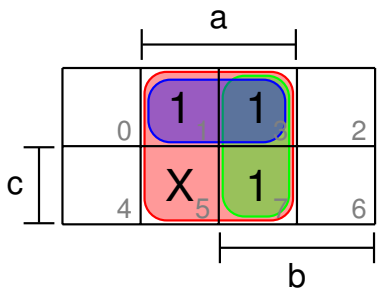
Don't Cares

- In Anwendungen treten häufig nicht alle Kombinationen der Eingangssignale auf.
- Das Ausgangssignal wird für solche Kombinationen dann **nicht definiert** und darf in der Realisierung einen **beliebigen Wert** (**0** oder **1**), engl.: **Don't Care**, annehmen.
- Die nicht definierten Ausgangssignale können zur Optimierung herangezogen werden und werden im KV-Diagramm mit „**X**" markiert.
- Ergibt sich für **X** = **1** eine **größere Gruppe**, so wird aus dieser Gruppe ein Produktterm gebildet.
- Die übrigen **X** werden nicht berücksichtigt, dies entspricht **X** = **0**.

Beispiel mit 3 Variablen:

	c	b	a	y	y_A	y_B
0	0	0	0	0	0	0
1	0	0	1	1	1	1
2	0	1	0	0	0	0
3	0	1	1	1	1	1
4	1	0	0	0	0	0
5	1	0	1	X	0	1
6	1	1	0	0	0	0
7	1	1	1	1	1	1

X...„don't care", Wert beliebig

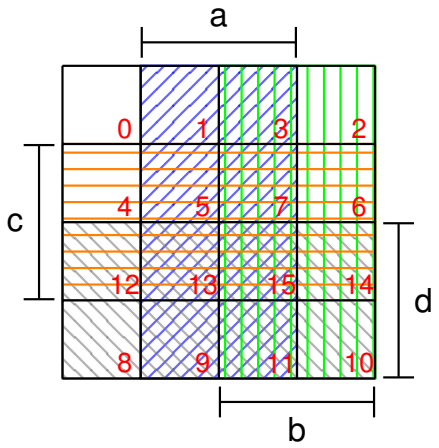


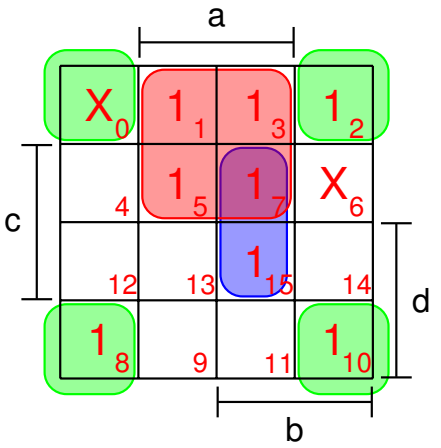
$$y_A = y_{DMF,X=0} = a\,b \vee a\,\overline{c}$$

$$y_B = y_{DMF,X=1} = a$$

Beispiel mit 4 Variablen: Bestimmen Sie die DMF der Funktion *y* **ÜBUNG**

	d	c	b	a	y
0	0	0	0	0	X
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	X
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1





$$y_{DMF} = a\,\overline{d} \vee \overline{a}\,\overline{c} \vee a\,b\,c$$

Durch die Zusammenfassung werden die X-Felder eindeutig auf 0 oder auf 1 gesetzt. D.h. in der realisierten Lösung gibt es keine Don't Cares mehr.

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

- Statt der Disjunktiven Normalform (ODER-Verknüpfung aller Minterme) ist nach DeMorgan auch eine **Konjunktive Normalform** möglich:

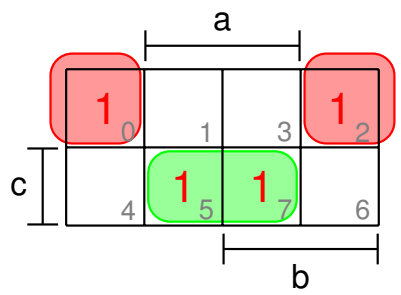
  UND-Verknüpfung aller **Maxterme (=ODER)** einer Funktionstabelle, für die die Ausgangsvariable den Wert **0** annimmt.

- **KNF** und **DNF** sind gleichwertige Normalformen.
- Analog zur DNF $\rightarrow$ DMF kann auch die KNF $\rightarrow$ KMF minimiert werden.
- In der Technik hat die **DNF bzw. DMF** die **größere Bedeutung**.

- Ausgangspunkt: Funktionstabelle für y
- Hinzufügen einer Spalte für das invertierte Ausgangssignal $\overline{y}$.
- Falls KNF gesucht: **DNF** für $\overline{y}$ aufstellen
- Falls KMF gesucht: **DMF** für $\overline{y}$ über KV-Diagramm ermitteln.
- Funktionsgleichung für $\overline{y}$ **negieren** und **De Morgansche Gesetze** anwenden.

Beispiel: Bildung der **KMF** mit drei Eingangsvariablen

	c	b	a	y	$\overline{y}$
0	0	0	0	0	1
1	0	0	1	1	0
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	1	0
7	1	1	1	0	1



$$\overline{y} = \overline{a}\,\overline{c} \vee a\,c$$

$$y_{KMF} = \overline{\overline{y}} = \overline{\overline{a}\,\overline{c} \vee a\,c} = (\overline{\overline{a} \cdot \overline{c}}) \cdot (\overline{a \cdot c})$$

$$y_{KMF} = (a \vee c) \cdot (\overline{a} \vee \overline{c})$$

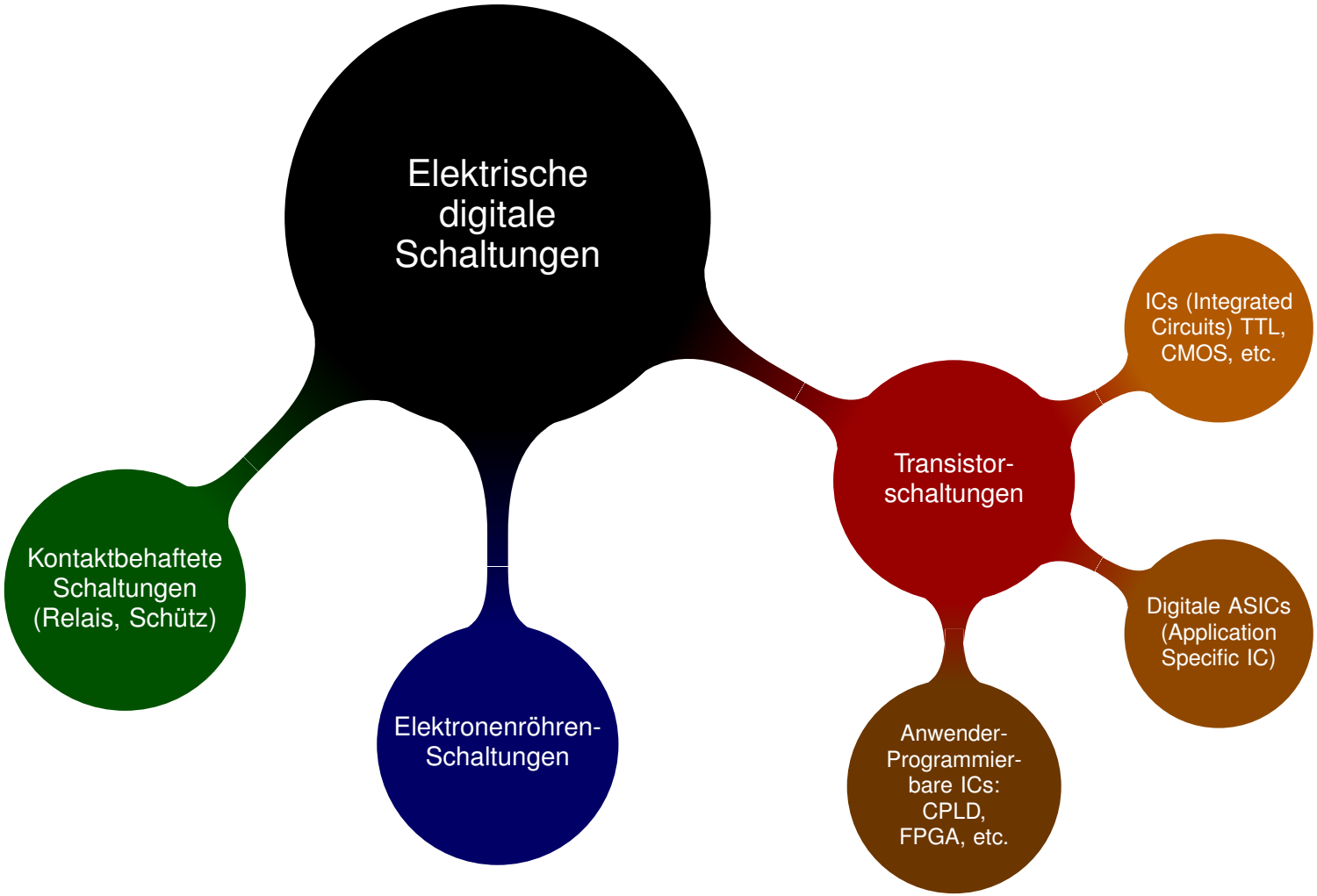
EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

- Die bisher betrachten Schaltungen sind **Kombinatorische Schaltungen** (auch **Schaltnetze** genannt) und besitzen folgende Eigenschaften:
  - Die **Ausgangssignale** sind ausschließlich **Funktionen der Eingangssignale**.
  - Wird also zu beliebigen Zeitpunkten eine bestimmte Kombination der Eingangssignale (X) angelegt, so ergibt sich für diese Eingangskombination stets die selbe Ausgangskombination (Y).
  - Insbesondere spielen **vorangegangene** Eingangssignale **keine Rolle**. Kombinatorische Schaltungen werden daher auch als „**gedächtnisfrei**“ bezeichnet.
- Beschreibungsformen für kombinatorische Schaltungen:
  - **Funktionstabelle**
  - **Boolesche Gleichungen**
  - **Schaltplan** (graphisch)

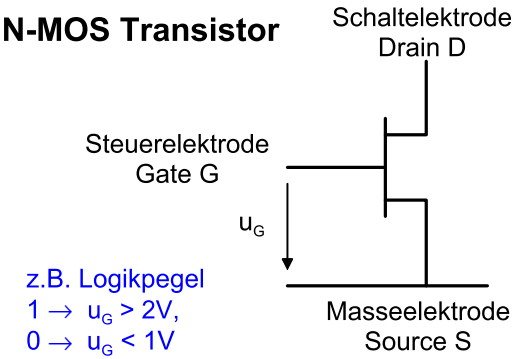
- Typische Schaltnetze:
  - Codeumsetzer: Codierer, Dekodierer
  - Arithmetik: Addierer, Subtrahierer, Multiplizierer, Vergleicher
  - Auswahlschaltungen: Multiplexer, Demultiplexer

EINFÜHRUNG IN DIE BOOLESCHE ALGEBRA

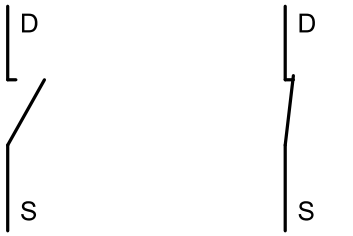
Realisierungsformen
Überblick elektrischer digitaler Schaltungen

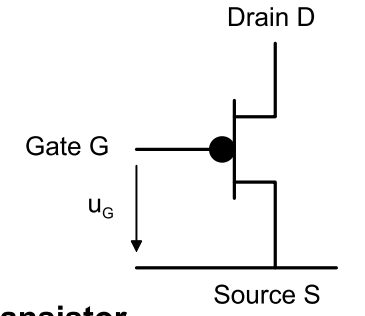


N-MOS Transistor

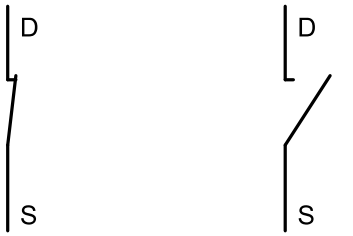


wirkt wie gesteuerter Schalter



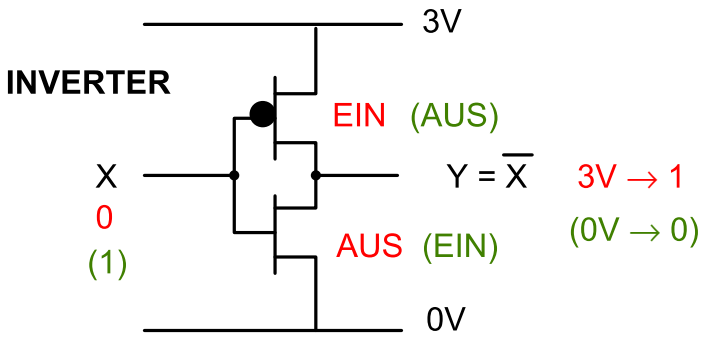


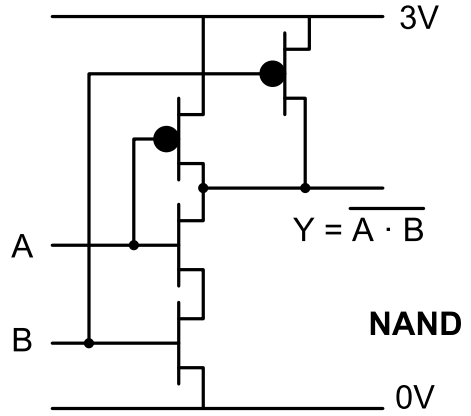
wirkt wie gesteuerter Schalter



P-MOS Transistor

Complementary MOS aus P- und N-MOS-Transistoren





- **Einzelgatter (NOT, AND, OR, XOR, Flipflops etc.)**
  Werden auch heute noch in Schaltungen auf Leiterplatten eingesetzt, v.a. wenn eine einzelne Logikfunktion benötigt wird.
- **(C)PLDs (Complex Programmable Logic Devices)**
  Bei höherer Komplexität werden **programmierbare Logikbausteine** eingesetzt. Der Aufbau von PLDs ermöglicht die direkte Umsetzung einer DMF. In CPLDs werden mehrere PLDs mit Verbindungskanälen kombiniert und jeder I/O-Pin besitzt ein **Flipflop** (1Bit Speicher).
- **FPGAs (Field Programmable Gate Array)**
  Ähnlich wie ein CPLD, jedoch sind die untereinander vernetzbaren Blöcke komplexerer. Ein Block beinhaltet **Flipflops** und **LUTs** (Look-up-Tables = programmierbare Funktionstabellen). Darüberhinaus sind auch oft schon **fertige Funktionsblöcke** wie RAM-Speicher oder CPU-Kerne enthalten.
- **ASICs (Application Specific ICs)**
  Erst bei **sehr hohen Stückzahlen** oder extremen Performance-Anforderungen ist es wirtschaftlich, ein IC applikationsspezifisch aufzubauen. Während CPLDs und FPGAs nur konfiguriert werden und alle Elemente fest platziert sind, werden bei ASICs geeignete Elemente **nach Bedarf platziert und verbunden**.
- **VHDL, Verilog und Silicon Compiler (Schaltungsynthese)**
  CPLDs, FPGAs und ASICS werden heute in einer Hardware-Programmiersprache (**VHDL, Verilog, SystemC**) entwickelt und durch ein **Schaltungsynthese**-Werkzeug (Silicon Compiler) in einen Gatter-Schaltplan (Netzliste) übersetzt.

Positive und negative Logik, Mehrwertige Logik

- Digitale Hardware codiert ein Bit typischerweise durch 2 Spannungspegel, z.B. $1 \Leftrightarrow u > 2V, \quad 0 \Leftrightarrow u < 1V$
- Da die absoluten Spannungswerte unwichtig sind, bezeichnet man die höhere Spannung als **H**, die niedrigere als **L**. Üblicherweise gilt die Zuordnung $1 \Leftrightarrow$ H, $0 \Leftrightarrow$ L, sogenannte **positive Logik**
- Die umgekehrte Zuordnung $1 \Leftrightarrow$ L, $0 \Leftrightarrow$ H heisst **negative Logik**.
- Verwendet man mehr als 2 Spannungspegel, kann man mehr als 1 Bit codieren. Dies wird bei modernen Speicherchips benutzt, um in einer Speicherzelle mehr als 1bit zu speichern (z.B. SSD mit TLC Flash ROM).

Logische Operationen in C/C++ und Java

- In C-artigen Programmiersprachen sind die logischen Operationen & AND, | OR, $\sim$ NOT, $\wedge$ XOR definiert. Diese Operationen arbeiten **bitweise**, d.h. Bit 0 des ersten Operanden wird mit Bit 0 des zweiten Operanden verknüpft, Bit 1 mit Bit 1 usw.
- In if- und while-Ausdrücken dagegen werden Bedingungen oft mit folgenden Operatoren verknüpft: && AND, || OR, ! NOT, z.B. `int i; if ((i>0) && (i<10)) ...` Hier erfolgt die Verarbeitung nicht bitweise, sondern die Bedingungen `(i>0)` und `(i<10)` werden als **true** oder **false** interpretiert und dann verknüpft.

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

GRUNDBEGRIFFE DER CODIERUNG
- Einführung in die Codierung
- Polyadische Zahlensysteme

CODIERUNG POSITIVER ZAHLEN
- Umrechnung von Zahlen zur Basis B ins Dezimalsystem
- Umrechnung von Dezimalzahlen in Zahlen zur Basis B
- Theoretische Grundlagen der Umrechnung
- Dual-, Oktal- und Hexadezimal-Code

CODIERUNG NEGATIVER ZAHLEN
- Codes für ganze Zahlen
- Vorzeichen–Betrags–Darstellung
- Offset–Code–Darstellung (Excess-Code, Stiplitz–Code)
- Komplement–Darstellung

CODIERUNG GLEITKOMMAZAHLEN
- Konzept
- IEEE 754 Format

WEITERE CODES
- Zeichen–Codes
- Gray, BCD und 1-aus-N Codes

FEHLERERKENNUNG UND -KORREKTUR
- Grundlagen
- Blockcodes

GRUNDBEGRIFFE DER CODIERUNG

Einführung in die Codierung
Polyadische Zahlensysteme

CODIERUNG POSITIVER ZAHLEN

Umrechnung von Zahlen zur Basis B ins Dezimalsystem
Umrechnung von Dezimalzahlen in Zahlen zur Basis B
Theoretische Grundlagen der Umrechnung
Dual-, Oktal- und Hexadezimal-Code

CODIERUNG NEGATIVER ZAHLEN

Codes für ganze Zahlen
Vorzeichen–Betrags–Darstellung
Offset–Code–Darstellung (Excess-Code, Stiplitz–Code)
Komplement–Darstellung

CODIERUNG GLEITKOMMAZAHLEN

Konzept
IEEE 754 Format

WEITERE CODES

Zeichen–Codes
Gray, BCD und 1-aus-N Codes

FEHLERERKENNUNG UND -KORREKTUR

Grundlagen
Blockcodes

Die Aufgabe der **Codierung** besteht darin, eine eindeutige (meist auch eineindeutige) **Zuordnung** zwischen Elementen **zweier Mengen** zu finden.

Beispiel: Sherlock Holmes: The Dancing Men

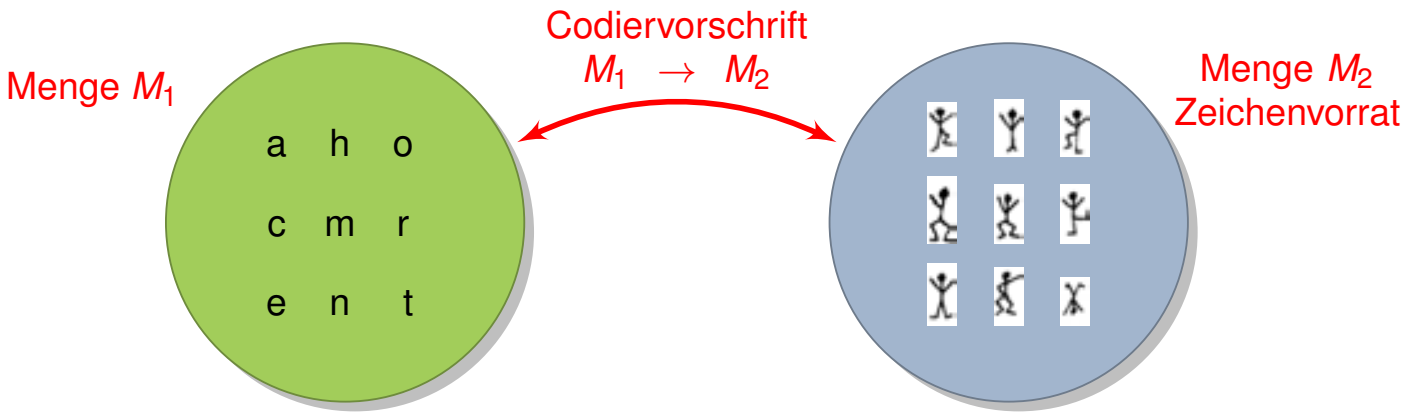
Verschlüsselung von Nachrichten durch Codierung der Zeichen a bis z mit Hilfe von Strichmännchen.

Zuordnungsbeispiel: ”e” $\longleftrightarrow$ 

Damit können Sätze wie folgt verschlüsselt werden:

”Come here at once” $\longleftrightarrow$ 

Durch die obige Codierung wird folgende eineindeutige Zuordnung festgelegt:



Zu codierende Datenmenge M_1:

Buchstaben: {A,B, . . .,Y,Z}

Ziffern (Dezimal): {0,1, . . .,8,9} (Hexadezimal): {0,1, . . .,9,A,. . .,F}

Natürliche Zahlen: $\{z \mid z \in \mathbb{N}_0\}$ Ganze Zahlen: $\{z \mid z \in \mathbb{Z}\}$

Zeichenvorrat M_2 eines **Binärcodes**:

n-stellige Binärworte: $\{0,1\}^n$ 2^n verschiedene Wörter

Eigenschaften von Codes:

Minimalcode: Ein Code, der **alle** zur Verfügung stehenden Elemente des Zeichenvorrats M_2 benutzt. **Bsp. S.3.6**

Redundanter Code: Ein Code, der **nicht alle** zur Verfügung stehenden Elemente des Zeichenvorrats M_2 benutzt.

Feste Wortlänge: Ein Code, bei dem alle Elemente des Zeichenvorrats M_2 **gleich lang** sind, d.h. z.B. bei Binärcodes, dass alle Binärworte des Zeichenvorrats aus genau n Bits bestehen. **Bsp. S.3.7**

Variable Wortlänge: Ein Code, bei dem es Elemente des Zeichenvorrats M_2 gibt, die **unterschiedlich lang** sind.

Gegeben seien Binärcodierungen mit fester Wortlänge: $M_1 = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$

Minimalcode	**Redundanter Code**
$M_2 = \{0,1\}^2$ $= \{(0,0);(0,1);(1,0);(1,1)\}$	$M_2 = \{0,1\}^3$ $= \{(0,0,0);(0,0,1);\ldots;(1,1,1)\}$
Anzahl $\lvert M_1\rvert = \lvert M_2\rvert = 2^2 = 4$	Anzahl $\lvert M_1\rvert = 4 < \lvert M_2\rvert = 2^3 = 8$

Zuordnungsvorschrift des Codes durch Codetabelle

Code C_1:

$\clubsuit$	00
$\diamondsuit$	01
$\heartsuit$	10
$\spadesuit$	11

Code C_r:

$\clubsuit$	010
$\diamondsuit$	101
$\heartsuit$	100
$\spadesuit$	111

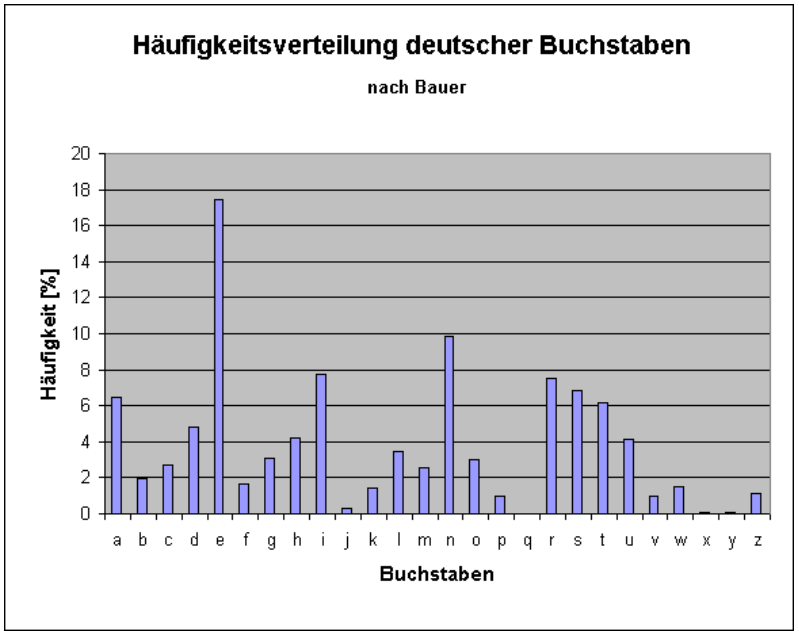
Anzahl möglicher Codierungen

$$4 \cdot 3 \cdot 2 \cdot 1 = 4! = 24$$

$$8 \cdot 7 \cdot 6 \cdot 5 = \frac{\lvert M_2\rvert !}{(\lvert M_2\rvert - \lvert M_1\rvert)!} = \frac{8!}{(8-4)!} = 1680$$

Morsealphabet: Code mit variabler Wortlänge

A	.–	**I**	..	**R**	.–.
Ä	.–.–	**J**	.– – –	**S**	...
B	–...	**K**	–.–	**T**	–
C	–.–.	**L**	.–..	**U**	..–
CH	– – – –	**M**	– –	**Ü**	..– –
D	–..	**N**	–.	**V**	...–
E	.	**O**	– – –	**W**	.– –
F	..–.	**Ö**	– – –.	**X**	–..–
G	– –.	**P**	.– –.	**Y**	–.– –
H	....	**Q**	– –.–	**Z**	– –..



Statistische Verteilung in Englisch ähnlich

Die Länge des zugewiesenen Codeworts richtet sich nach der Häufigkeit des Buchstabens. Dadurch ist es möglich die Gesamtlänge einer Nachricht zu minimieren.

GRUNDBEGRIFFE DER CODIERUNG

In diesem Kapitel werden Codierungen mit den folgenden Definitionsmengen betrachtet:

- $M_1 \subset \mathbb{N}_0$, **Natürliche** (nur positive) **Zahlen** inklusive 0
- $M_1 \subset \mathbb{Z}$, **Ganze** (positive und negative) **Zahlen**
- $M_1 \subset \mathbb{Q}$, **Rationale Zahlen** („Komma“-Zahlen mit Nachkommastellen)

Achtung:
Unterscheiden Sie zwischen der inhaltlichen Bedeutung von Zahlen (z.B. eins, zwei, drei) und deren Darstellung (Codierung)!

Arabische Zahl	Römische Zahl
1	I
2	II
3	III
4	IV
⋮	⋮
989	CMLXXXIX
⋮	⋮

Stellenwertsystem Additionssystem

Vorteil des arab. Zahlensystems:

- Codierung nicht willkürlich
- Abbildungsvorschrift kann **mathematisch** formuliert werden
- Einfaches Rechnen

Definition:
Polyadisches Zahlensystem = Zahlensystem mit fester **Stellenwertigkeit**

Beispiel:

$$(743,5)_{10} = (\overbrace{7}^{z_2} \cdot \overbrace{100}^{10^2} + \overbrace{4}^{z_1} \cdot \overbrace{10}^{10^1} + \overbrace{3}^{z_0} \cdot \overbrace{1}^{10^0} + \overbrace{5}^{z_{-1}} \cdot \overbrace{0,1}^{10^{-1}})_{10}$$

$$= (z_2\ z_1\ z_0\ ,\ z_{-1})_{10} = \sum_{\nu=-1}^{2} z_\nu \cdot 10^\nu$$

- **Dezimalsystem**: (Arabisches) Zahlensystem mit **Basis B=10**
- ν–te Stelle ($\nu \in \mathbb{Z}$) besitzt die Wertigkeit $\mathbf{10}^\nu$ mit $\nu = 0$...Einer-Stelle
- Ziffern z_ν können Werte von **0** bis $B - 1 = 10 - 1 = \mathbf{9}$ annehmen

Verallgemeinerung:

- **Polyadisches Zahlensystem** zur Basis $\mathbf{B} \in \mathbb{N} \geq 2$
- ν–te Stelle besitzt die Wertigkeit $\mathbf{B}^\nu$.
- Ziffern z_ν können ganzzahlige Werte zwischen **0** und $\mathbf{B}-\mathbf{1}$ annehmen.

Allgemeine Zahl zur Basis B:

$$(Z)_B = \pm(\ldots z_1 z_0\ ,\ z_{-1} z_{-2} \ldots)_B = \pm \sum_{\nu=-\infty}^{\infty} z_\nu \cdot B^\nu$$

$\nu \geq 0$: Stellen vor dem Komma, $\nu < 0$: Stellen nach dem Komma

B=10 . . . **D**ezimalzahlen, B=2 . . . **D**ualzahlen, B=16 . . . **H**exadezimalzahlen

Technische Realisierung nur mit **endlicher Stellenzahl n** möglich

Zunächst: Betrachtung von **positiven** Zahlen (inkl. 0): $\mathbf{Z \geq 0}$

- $Z = + \sum_{\nu=-s}^{l} z_\nu \cdot B^\nu$ mit $n = l + 1 + s$ Stellen

  B...Zahlenbasis, l+1...Stellen vor dem Komma, s...Stellen nach dem Komma

  Zahlen mit n=const. und s=const. werden als **Festkommazahlen** bezeichnet.

- Möglicher **Zahlenbereich**: $\mathbf{Z_{min} = 0} \leq Z \leq \mathbf{Z_{max} = (B^n - 1) \cdot B^{-s}}$
- Rechenregeln der klassischen Mathematik gelten wegen Über- und Unterläufen nicht mehr uneingeschränkt:

  Beispiel Assoziativgesetz: $\boxed{(Z_1 + Z_2) - Z_3 = Z_1 + (Z_2 - Z_3)}$

  Beispiele, die durch **Über- und Unterläufe** zu Problemen führen:

  $B = 10;\ l + 1 = 3;\ s = 0 \Rightarrow Z_{max} = 999;\ Z_{min} = 000$

  - $Z_1 = 900; Z_2 = 500; Z_3 = 450 \Rightarrow \mathbf{Z_1 + Z_2 = 1400 > Z_{max}}$ **Überlauf**
  - $Z_1 = 100; Z_2 = 500; Z_3 = 550 \Rightarrow \mathbf{Z_2 - Z_3 = -50 < Z_{min}}$ **Unterlauf**
  - **Erkennung und Behandlung von Über- und Unterläufen?**

Umrechnung von positiven Zahlen zur Basis B ins Dezimalsystem

- Direktes Auswerten der Summe: $Z = \sum_{\nu=-s}^{l} z_\nu \cdot B^\nu$

Beispiel: $Z = (1011)_2 \qquad B = 2, l+1 = 4, s = 0$

- $Z = \sum_{\nu=0}^{3} z_\nu \cdot 2^\nu = z_3 \cdot 2^3 + z_2 \cdot 2^2 + z_1 \cdot 2^1 + z_0 \cdot 2^0$
  $= 8 + 0 + 2 + 1 = (\mathbf{11})_{\mathbf{10}}$

Beispiel: $Z = (1,1101)_2 \qquad B = 2, l+1 = 1, s = 4$

- $Z = \sum_{\nu=-4}^{0} z_\nu \cdot 2^\nu = z_0 \cdot 2^0 + z_{-1} \cdot 2^{-1} + z_{-2} \cdot 2^{-2} + z_{-3} \cdot 2^{-3} + z_{-4} \cdot 2^{-4}$

  $= 1 + 0,5 + 0,25 + 0 + 0,0625 = (\mathbf{1,8125})_{\mathbf{10}}$

Zahlenbereich bei natürlichen Zahlen im n bit Dualcode (B=2)

- Min: $Z_{min} = (0 \cdots 0)_2 = 0$ **z.B. $n = 8 \rightarrow z_{min} = 0$**
- Max: $Z_{max} = (1 \cdots 1)_2 = 2^n - 1$ **z.B. $n = 8 \rightarrow z_{max} = 255_{10}$**

Rechnen Sie folgende Zahlen in das Dezimalsystem um: **Ü B U N G**

- $(1110)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 4 + 2 + 0 = (\mathbf{14})_{\mathbf{10}}$
- $(1010\ 0000)_2 = 1 \cdot 2^7 + 1 \cdot 2^5 = 128 + 32 = (\mathbf{160})_{\mathbf{10}}$
- $(0,011)_2 = 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0,25 + 0,125 = (\mathbf{0,375})_{\mathbf{10}}$
- $(120)_3 = 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0 = 9 + 6 + 0 = (\mathbf{15})_{\mathbf{10}}$
- $(231)_8 = 2 \cdot 8^2 + 3 \cdot 8^1 + 1 \cdot 8^0 = 128 + 24 + 1 = (\mathbf{153})_{\mathbf{10}}$
- $(77)_8 = 7 \cdot 8^1 + 7 \cdot 8^0 = 56 + 7 = (\mathbf{63})_{\mathbf{10}}$

Rechnen Sie folgende Zahlen in das Dezimalsystem um: **Ü B U N G**

- $(1,11)_2 = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + 0,5 + 0,25 = \mathbf{(1,75)_{10}}$
- $(101,01)_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-2} = 4 + 1 + 0,25 = \mathbf{(5,25)_{10}}$
- $(32,3)_4 = 3 \cdot 4^1 + 2 \cdot 4^0 + 3 \cdot 4^{-1} = 12 + 2 + 0,75 = \mathbf{(14,75)_{10}}$
- $(2,74)_8 = 2 \cdot 8^0 + 7 \cdot 8^{-1} + 4 \cdot 8^{-2} = 2 + 0,875 + 0,0625 = \mathbf{(2,9375)_{10}}$

Trick zur Umrechnung von Festkomma-Dualzahlen (mit s Stellen nach dem Komma) in Dezimalwerte:

- Dezimalwert $(Z)^*_{10}$ berechnen wie bei natürlichen Zahlen, d.h. Komma einfach weglassen **Bsp.: $(1,11)_2 \rightarrow (111)_2 = 7_{10}$**
- Tatsächlicher Dezimalwert $(Z)_{10} = (Z)^*_{10} \cdot 2^{-s} \rightarrow \mathbf{7_{10} \cdot 2^{-2} = 1,75_{10}}$

Umrechnung von Zahlen zur Basis B ins Dezimalsystem

Tabelle der 2er–Potenzen

2er Potenz	Dezimal	Dual	Hexadezimal
2^{-4}	$\frac{1}{16} = 0,0625$	$(0,0001)_2$	$(0,1)_{16}$
2^{-3}	$\frac{1}{8} = 0,125$	$(0,001)_2$	$(0,2)_{16}$
2^{-2}	$\frac{1}{4} = 0,25$	$(0,01)_2$	$(0,4)_{16}$
2^{-1}	$\frac{1}{2} = 0,5$	$(0,1)_2$	$(0,8)_{16}$
2^0	1	$(1)_2$	$(1)_{16}$
2^1	2	$(10)_2$	$(2)_{16}$
2^2	4	$(100)_2$	$(4)_{16}$
2^3	8	$(1000)_2$	$(8)_{16}$
2^4	16	$(1\ 0000)_2$	$(10)_{16}$
2^5	32	$(10\ 0000)_2$	$(20)_{16}$
2^6	64	$(100\ 0000)_2$	$(40)_{16}$
2^7	128	$(1000\ 0000)_2$	$(80)_{16}$
2^8	256		$(100)_{16}$
2^9	512		$(200)_{16}$
2^{10}	$1024 = 1$ Ki	$(100\ 0000\ 0000)_2$	$(400)_{16}$
2^{12}	$4096 = 4$ Ki		$(1000)_{16}$
2^{15}	$32768 = 32$ Ki		$(8000)_{16}$
2^{16}	$65536 = 64$ Ki		$(1\ 0000)_{16}$
2^{20}	$1\ 048\ 576 = 1$ Mi		$(10\ 0000)_{16}$
2^{30}	$1\ 073\ 741\ 824 = 1$ Gi		$(4000\ 0000)_{16}$
2^{32}	$4\ 294\ 967\ 296 = 4$ Gi		$(1\ 0000\ 0000)_{16}$
2^{40}	$1\ 099\ 511\ 627\ 776 = 1$ Ti		

Umrechnung durch Zerlegung in 2er-Potenzen

Beispiel: $Z = 149,5 = 128 + 16 + 4 + 1 + 0,5 = (1001\ 0101, 1)_2$

Vorgehen:

1. Größte 2er-Potenz $P_a < Z$ suchen $\rightarrow P_a = 128 = 2^7$
2. Größte 2er-Potenz P_b suchen, so dass $P_a + P_b < Z$
   $\rightarrow P_b = 16 = 2^4 \quad \rightarrow \quad P_a + P_b = 128 + 16 = 144$
3. Größte 2er-Potenz P_c suchen, so dass $P_a + P_b + P_c < Z$
   $\rightarrow P_c = 4 = 2^2 \quad \rightarrow \quad P_a + P_b + P_c = 128 + 16 + 4 = 148$
4. ... usw. ...

Übungen:

- $Z = 211 = 128 + 64 + 16 + 2 + 1 = (1101\ 0011)_2$
- $Z = 82,25 = 64 + 16 + 2 + 0,25 = (101\ 0010, 01)_2$
- $Z = 33,75 = 32 + 1 + 0,5 + 0,25 = (10\ 0001, 11)_2$

Für die Hardware-Implementierung im Computer ist dieses „Probierverfahren“ (Schleife mit Addition, Vergleich und Entscheidung) zu komplex.

Betrachtung von positiven Zahlen: Aufspaltung in ganzzahligen Teil (l+1 Stellen vor dem Komma) und echt-gebrochenen Teil (s Nachkomma-Stellen)

$$Z = + \sum_{\nu=-s}^{l} z_\nu \cdot B^\nu = \underbrace{\sum_{\nu=0}^{l} z_\nu \cdot B^\nu}_{\text{ganzzahliger Anteil: } Z_g} + \underbrace{\sum_{\nu=-s}^{-1} z_\nu \cdot B^\nu}_{\text{echt gebrochener Anteil: } Z_b} = Z_g + Z_b$$

Algorithmus für den ganzzahligen Teil Z_g: Fortgesetzte Division durch B

$$\begin{array}{lcllll} Z_g & : & B & = & q_0 & \text{Rest } z_0 \\ q_0 & : & B & = & q_1 & \text{Rest } z_1 \\ & & \vdots & & & \\ q_{l-1} & : & B & = & 0 & \text{Rest } z_l \end{array} \quad \Uparrow \quad \Rightarrow \boxed{Z_g = (z_l \ldots z_1 z_0)_B}$$

Algorithmus für den gebrochenen Teil Z_b: Fortgesetzte Multiplikation mit B

$$\begin{array}{lcllll} Z_b & \cdot & B & = & b_{-1} & + z_{-1} \\ b_{-1} & \cdot & B & = & b_{-2} & + z_{-2} \\ & & \vdots & & & \\ b_{-s+1} & \cdot & B & = & 0 & + z_{-s} \end{array} \quad \Downarrow \quad \Rightarrow \boxed{Z_b = (0\ ,\ z_{-1} z_{-2} \ldots z_{-s})_B}$$

Beispiel: Umrechnung der Zahl $Z = (28,8125)_{10}$ in die Basis 2

- Ganzzahliger Teil $Z_g = 28$ **vom Komma nach vorn**

28	:	2	=	14	Rest	0
14	:	2	=	7	Rest	0
7	:	2	=	3	Rest	1
3	:	2	=	1	Rest	1
1	:	2	=	0	Rest	1 MSB

$\Rightarrow$ $Z_g = (11100)_2$

- Gebrochener Teil $Z_b = 0,8125$ **vom Komma nach hinten**

0,8125	·	2	=	0,625	+	1
0,625	·	2	=	0,25	+	1
0,25	·	2	=	0,5	+	0
0,5	·	2	=	0,0	+	1 LSB

$\Rightarrow$ $Z_b = (0,1101)_2$

Probe: Umrechnung der Zahl $(11100,1101)_2$ in die Basis 10

$$(11100,1101)_2 = 1\cdot2^4 + 1\cdot2^3 + 1\cdot2^2 + 1\cdot2^{-1} + 1\cdot2^{-2} + 1\cdot2^{-4}$$
$$= 16 + 8 + 4 + 0,5 + 0,25 + 0,0625 = (\mathbf{28,8125})_{10}$$

Rechnen Sie folgende Zahlen aus dem Dezimalsystem in die angegebenen Zahlensysteme um:

ÜBUNG

- $(22)_{10} = (???)_2$

22	:	2	=	11	Rest	0
11	:	2	=	5	Rest	1
5	:	2	=	2	Rest	1
2	:	2	=	1	Rest	0
1	:	2	=	0	Rest	1

MSB

$\Rightarrow$ $Z = (1\,0110)_2$

- $(28)_{10} = (???)_4$

28	:	4	=	7	Rest	0
7	:	4	=	1	Rest	3
1	:	4	=	0	Rest	1

MSB

$\Rightarrow$ $Z = (130)_4$

Rechnen Sie folgende Zahlen aus dem Dezimalsystem in die angegebenen Zahlensysteme um:

ÜBUNG

- $(50)_{10} = (???)_2$

50	:	2	=	25	Rest	0
25	:	2	=	12	Rest	1
12	:	2	=	6	Rest	0
6	:	2	=	3	Rest	0
3	:	2	=	1	Rest	1
1	:	2	=	0	Rest	1

$\Rightarrow$ $Z = (11\ 0010)_2$

- $(79)_{10} = (???)_8$

79	:	8	=	9	Rest	7
9	:	8	=	1	Rest	1
1	:	8	=	0	Rest	1

$\Rightarrow$ $Z = (117)_8$

Rechnen Sie folgende Zahlen aus dem Dezimalsystem in die angegebenen Zahlensysteme um:

ÜBUNG

- $(0,375)_{10} = (???)_2$

$$
\begin{array}{rcrcrcl}
0,375 & \cdot & 2 & = & 0,75 & + & \boxed{0} \\
0,75 & \cdot & 2 & = & 0,5 & + & 1 \\
0,5 & \cdot & 2 & = & 0 & + & 1
\end{array}
\quad \Downarrow \text{LSB} \quad \Rightarrow \quad \boxed{Z = (0,011)_2}
$$

- $(0,8)_{10} = (???)_2$

$$
\begin{array}{rcrcrcl}
0,8 & \cdot & 2 & = & 0,6 & + & 1 \\
0,6 & \cdot & 2 & = & 0,2 & + & 1 \\
0,2 & \cdot & 2 & = & 0,4 & + & 0 \\
0,4 & \cdot & 2 & = & 0,8 & + & 0 \\
0,8 & \cdot & 2 & = & 0,6 & + & 1 \\
\vdots & \cdot & 2 & = & \vdots & + & \vdots
\end{array}
\quad \Downarrow \text{LSB} \quad \Rightarrow \quad \boxed{Z = \left(0,\overline{1100}\right)_2}
$$

Rechnen Sie folgende Zahl aus dem Dezimalsystem in das angegebene Zahlensysteme um:

ÜBUNG

- $(299,78125)_{10} = (???)_{16}$

$$\begin{array}{rcrcrl|c} 299 & : & 16 & = & 18 & \text{Rest} & 11 = B \\ 18 & : & 16 & = & 1 & \text{Rest} & 2 \\ 1 & : & 16 & = & 0 & \text{Rest} & 1 \end{array}$$

MSB

$$\Rightarrow Z_g = (12B)_{16}$$

$$\begin{array}{rcrcrcl|c} 0,78125 & \cdot & 16 & = & 0,5 & + & 12 = C \\ 0,5 & \cdot & 16 & = & 0 & + & 8 \end{array}$$

LSB

$$\Rightarrow Z_b = (0,C8)_{16}$$

$$Z = Z_g + Z_b = (12B,C8)_{16}$$

Statt der direkten Umrechnung kann die **Umrechnung von positiven Zahlen zur Basis B in Dezimalzahlen** auch nach folgender Umformung (**Horner-Schema**) erfolgen. Für den ganzzahligen Teil gilt:

$$Z_g = \sum_{\nu=0}^{l} z_\nu \cdot B^\nu = (((z_l \cdot B + z_{l-1}) \cdot B + z_{l-2}) \cdot B + \ldots + z_1) \cdot B + z_0$$

Beginn mit vorderster Stelle, dann wiederholen: Multiplikation x B plus nächstniedrigere Stelle → Vorteil: Berechnung der Potenzen $B^2, B^3, \cdots, B^l$ entfällt

Analog für den gebrochenen Teil positiver Zahlen:

$$Z_b = \sum_{\nu=-s}^{-1} z_\nu \cdot B^\nu$$
$$= \left(\left(\left(z_{-s} \cdot B^{-1} + z_{-s+1}\right) \cdot B^{-1} + z_{-s+2}\right) \cdot B^{-1} + \ldots + z_{-1}\right) \cdot B^{-1}$$

Beginn mit der hintersten Stelle, dann wiederholen: Multiplikation x $B^{-1} = \frac{1}{B}$ plus nächsthöhere Stelle. Abschließend nochmals Multiplikation x $B^{-1} = \frac{1}{B}$

Beispiel: $Z = (1011,1101)_2 \qquad B = 2, \;\; l+1 = 4, \;\; s = 4$

$$Z_g = ((z_3 \cdot 2 + z_2) \cdot 2 + z_1) \cdot 2 + z_0 = ((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 = \mathbf{(11)_{10}}$$

$$Z_b = \left(\left(\left(z_{-4} \cdot \tfrac{1}{2} + z_{-3}\right) \cdot \tfrac{1}{2} + z_{-2}\right) \cdot \tfrac{1}{2} + z_{-1}\right) \cdot \tfrac{1}{2}$$

$$= \left(\left(\left(\tfrac{1}{2} + 0\right) \cdot \tfrac{1}{2} + 1\right) \cdot \tfrac{1}{2} + 1\right) \cdot \tfrac{1}{2} = \mathbf{(0,8125)_{10}} \rightarrow \mathbf{Z = (11,8125)_{10}}$$

Die **Umrechnung von positiven Dezimalzahlen in Zahlen zur Basis B** erfolgt durch Umdrehen des Horner-Schemas. Für den ganzzahligen Anteil:

$$Z_g = z_0 + B \cdot (z_1 + B \cdot (z_2 + \ldots + B \cdot (z_{l-2} + B \cdot (z_{l-1} + B \cdot z_l))))$$

$$\Rightarrow \frac{Z_g}{B} = \frac{z_0}{B} + \underbrace{\left(z_1 + B \cdot \underbrace{\left(z_2 + \ldots + B \cdot (z_{l-2} + B \cdot (z_{l-1} + B \cdot z_l)) \right)}_{:= q_1 \in \mathbb{N}_0} \right)}_{:= q_0 \in \mathbb{N}_0}$$

→ Bestimmen von $z_0, z_1, \cdots, z_l$ durch fortgesetzte Division durch B

Für den echt-gebrochenen Anteil gilt:

$$Z_b = B^{-1} \cdot \left(z_{-1} + B^{-1} \cdot \left(z_{-2} + \ldots + B^{-1} \cdot \left(z_{-s+2} + B^{-1} \cdot \left(z_{-s+1} + B^{-1} \cdot z_{-s} \right) \right) \right) \right)$$

$$Z_b \cdot B = z_{-1} + \underbrace{B^{-1} \cdot \left(z_{-2} + \ldots + B^{-1} \cdot \left(z_{-s+2} + B^{-1} \cdot \left(z_{-s+1} + B^{-1} \cdot z_{-s} \right) \right) \right)}_{:= b_{-1} \in \mathbb{Q},\ d.h.\ 0 \leq b_{-1} < 1}$$

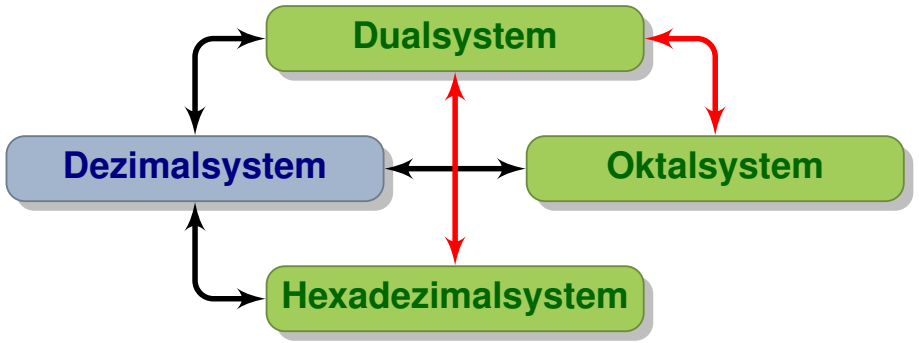
→ Bestimmen von $z_{-1}, z_{-2}, \cdots, z_{-s}$ durch fortgesetzte Multiplikation mit B

Durch schrittweise Anwendung ergeben sich damit die im vorigen Abschnitt auf S. 3.19 dargestellten Algorithmen.

Neben dem Dezimalsystem sind vor allem drei Zahlenbasen verbreitet:

Dualsystem	Ziffern: $z_\nu \in \{0, 1\}$.
B = 2	Die Ziffern werden hier auch Bit genannt.
	Vorteil: Einfache Darstellbarkeit $\rightarrow$ Digitaltechnik
Oktalsystem	Ziffern: $z_\nu \in \{0, 1, 2, 3, 4, 5, 6, 7\}$.
B = 8	*Vorteil*: Verwandschaft zum Dualsystem
Hexadezimalsystem	Ziffern: $z_\nu \in \{0, 1, 2, \ldots, 8, 9, A, B, C, D, E, F\}$.
B = 16	*Vorteil*: Verwandschaft zum Dualsystem

Umrechnung zwischen diesen Zahlensystemen und dem **Dezimalsystem** wie oben beschrieben.



Durch Verwandtschaft ist ein vereinfachtes Umrechnen zwischen Dual- und Hexadezimalsystem bzw. Dual- und Oktalsystem möglich.

- Beziehung zwischen dem **Dual–** und dem **Oktalsystem**:
  Da $\mathbf{8 = 2^3}$ ist, entsteht **eine Stelle des Oktalsystems aus** der Zusammenfassung von **drei Stellen des Dualsystems**.
- Beziehung zwischen dem **Dual–** und dem **Hexadezimalsystem**:
  Da $\mathbf{16 = 2^4}$ ist, entsteht **eine Stelle des Hexadezimalsystems aus** der Zusammenfassung von **vier Stellen des Dualsystems**.
- Hinweis:
  Zu einer **nicht–negativen Zahl** im polyadischen Zahlensystem, können **führende Nullstellen** sowie **Null–Ziffern als Nachkommastellen** beliebig **ergänzt** werden.

Beispiel: Dualsystem $\longleftrightarrow$ Oktalsystem

$$Z = (859,75)_{10} = (1101011011,11)_2 = (\underbrace{001}_{1}\underbrace{101}_{5}\underbrace{011}_{3}\underbrace{011}_{3,}\underbrace{110}_{6}{}_{8})_2$$

$$Z = (1533,6)_8 = 1\cdot 8^3 + 5\cdot 8^2 + 3\cdot 8^1 + 3\cdot 8^0 + 6\cdot 8^{-1} = 859,75$$

Beispiel: Dualsystem $\longleftrightarrow$ Hexadezimalsystem

$$Z = (859,75)_{10} = (1101011011,11)_2 = (\underbrace{0011}_{3}\underbrace{0101}_{5}\underbrace{1011}_{B,}\underbrace{1100}_{C}{}_{16})_2$$

$$Z = (35B,C)_{16} = 3\cdot 16^2 + 5\cdot 16^1 + 11\cdot 16^0 + 12\cdot 16^{-1} = 859,75$$

Rechnen Sie folgende Zahlen in die angegebenen Zahlensysteme um:

Ü B U N G

- $(100111001001,011)_2 = (???)_8 = \underbrace{100}_{4}\,\underbrace{111}_{7}\,\underbrace{001}_{1}\,\underbrace{001}_{1},\underbrace{011}_{3} = (\mathbf{4711,3})_{\mathbf{8}}$
- $(100111001001,011)_2 = (???)_{16} = \underbrace{1001}_{9}\,\underbrace{1100}_{C}\,\underbrace{1001}_{9},\underbrace{0110}_{6} = (\mathbf{9C9,6})_{\mathbf{16}}$
- $(37,F)_{16} = (???)_2 = \underbrace{0011}_{3}\,\underbrace{0111}_{7},\underbrace{1111}_{F} = (\mathbf{11\,0111,1111})_{\mathbf{2}}$
- $(27,5)_8 = (???)_{16} = \underbrace{010}_{2}\,\underbrace{111}_{7},\underbrace{101}_{5} = \underbrace{0001}_{1}\,\underbrace{0111}_{7},\underbrace{1010}_{A} = (\mathbf{17,A})_{\mathbf{16}}$

Rechnen mit Dualzahlen

- Wie mit Dezimalzahlen stellenweise von hinten nach vorne mit Übertrag
- $0+0=0, \quad 0+1=1+0=1, \quad 1+1=10_2$ (0 mit Übertrag 1)
- $1-0=1, \quad 0-0=1-1=0, \quad 0-1=11_2$ (1 mit Übertrag 1)

Beispiel: Dualzahlen mit **n = 4 Bit**

Addition:

	0011	$=(3)_{10}$		0011	$=(3)_{10}$
+	0100	$=(4)_{10}$	+	0101	$=(5)_{10}$
				1110	**= Ü**
	0111	$=(7)_{10}$		1000	$=(8)_{10}$

Subtraktion:

	0111	$=(7)_{10}$		1011	$=(11)_{10}$
−	0100	$=(4)_{10}$	−	0101	$=(5)_{10}$
				1000	**= Ü**
	0011	$=(3)_{10}$		0110	$=(6)_{10}$

Führen Sie die folgende Rechnungen im Dualsystem mit **n = 8 Bit** durch:

Ü B U N G

Addition:

	0001 1011	$= (27)_{10}$
+	0001 1110	$= (30)_{10}$
	0011 1100	$=$ *Uebertrag*
	0011 1001	$= (57)_{10}$

	1000 0100	$= (132)_{10}$
+	0001 0100	$= (20)_{10}$
	0000 1000	$=$ *Uebertrag*
	1001 1000	$= (152)_{10}$

Subtraktion:

	0001 1110	$= (30)_{10}$
−	0001 1011	$= (27)_{10}$
	0000 0110	$=$ *Uebertrag*
	0000 0011	$= (3)_{10}$

	1000 0100	$= (132)_{10}$
−	0001 0100	$= (20)_{10}$
	1110 0000	$=$ *Uebertrag*
	0111 0000	$= (112)_{10}$

Ein **Überlauf** tritt auf, wenn das Ergebnis einer Rechenoperation nicht mehr mit den verfügbaren **n-bit darstellbar** ist:

- Rechenwerke/Mikroprozessoren zeigen Überläufe bei Operationen mit **Betragszahlen** über ein Statusbit „**Carry-Flag**“(„**CF**“, „**CY**“ bzw. „**C**“) an.
- Ob auf den Überlauf reagiert wird, entscheidet nicht das Rechenwerk, sondern das Softwareprogramm, das das Überlaufflag abfragen kann.

Beispiele: $(S) = [(X) \pm (Y)] \bmod 2^n$ mit n = 4 bit Dualzahlen

Addition

X		0	1	1	1	7_{10}
Y		0	1	1	0	6_{10}
C	0	1	1	0	0	
S		1	1	0	1	13_{10}

Ergebnis: **korrekt**

Addition

X		1	1	1	0	14_{10}
Y		0	1	0	0	4_{10}
C	1	1	0	0	0	
S		0	0	1	0	2_{10}

Ergebnis: **falsch**

Überlauf $CF = 1$ ...

... bei Addition, wenn

$CF = \mathbf{C_n}$

Hinweis:
Subtraktionen von Betragszahlen werden von der Hardware durchgeführt, indem das Zweierkomplement des Subtrahenden addiert wird (siehe später). Dabei entstehende Überläufe werden durch $CF = \overline{C_n}$ erkannt.

Anforderungen an Codes für ganze Zahlen:

- Einfache Arithmetik, d.h. einfache Realisierung von Vorzeichenumkehr, Addition, Subtraktion, ...
- Eindeutige Codierung, z.B. nur ein Code für +0 und -0
- Symmetrischer Zahlenbereich, d.h. gleiche Anzahl von positiven und negativen Zahlen

Mögliche Codierungen:

- Vorzeichen-Betrag
- Dual-Offset
- Zweier-Komplement

Sign-Magnitude

- **Betrag** der Zahl mit **vorgestellten Vorzeichen**, wie beim Dezimalsystem.
- Betrag in Dual-, Oktal- oder Hexadezimalzahl-Darstellung.
- Vorzeichenbit wird der Zahl vorangestellt, d.h. das vorderste Bit (MSB) ist immer das Vorzeichenbit. Üblich: **0** für $+$ **1** für $-$

Beispiel: $Z = \mathbf{0}011_2 \rightarrow +3_{10}$ $Z = \mathbf{1}011_2 \rightarrow -3_{10}$

Achtung: Hier müssen **führende 0-Ziffern nach (!) dem Vorzeichenbit** eingefügt werden, weil sie sonst als Vorzeichen interpretiert werden.

Vorteile und Nachteile:

+ Sehr einfache Darstellungsform
− **Zwei Darstellungen für 0**: ”+0” und ”−0”
− **Zwei Rechenwerke** für $Z_1 \pm Z_2$ notwendig:

Addierwerk für:	pos.+pos.	$(+\lvert Z_1\rvert) + (+\lvert Z_2\rvert) = +(\lvert Z_1\rvert + \lvert Z_2\rvert)$
	neg.+neg.	$(-\lvert Z_1\rvert) + (-\lvert Z_2\rvert) = -(\lvert Z_1\rvert + \lvert Z_2\rvert)$
Subtrahierwerk für:	pos.−pos.	$(+\lvert Z_1\rvert) - (+\lvert Z_2\rvert) = +(\lvert Z_1\rvert - \lvert Z_2\rvert)$
	neg.−neg.	$(-\lvert Z_1\rvert) - (-\lvert Z_2\rvert) = -(\lvert Z_1\rvert - \lvert Z_2\rvert)$
	...	

Rechnen Sie die Vorzeichen–Betrags–Darstellungen in Dezimalzahlen um:

- $(1111)_2 = (-7)_{10}$
- $(0100\ 0001)_2 = +(2^6 + 1)_{10} = (+65)_{10}$
- $(1000\ 0001)_2 = (-1)_{10}$
- $(72)_8 = (111\ 010)_2 = -(2^4 + 2^3 + 2^1)_{10} = (-26)_{10}$
- $(A1)_{16} = (1010\ 0001)_2 = -(2^5 + 1)_{10} = (-33)_{10}$

Bestimmen Sie die Vorzeichen–Betrags–Darstellung:

- $+9_{10} = (???)_2 = (0\ 1001)_2$ (n = 5 bit)
- $+9_{10} = (???)_8 = (0\ 001\ 001)_2 = (011)_8$ (n = 7 bit)
- $-33_{10} = (???)_{16} = (100100001)_2 = (121)_{16}$ (n = 9 bit)

Zahlenbereich bei ganzen n bit Dualzahlen in Betrags-Vorzeichen-Darstellung

- Max: $Z_{max} = (0\ 1 \cdots 1)_2 = +(2^{n-1} - 1)$ **z.B. $n = 8 \rightarrow z_{max} = +127_{10}$**
- Min: $Z_{min} = (1\ 1 \cdots 1)_2 = -(2^{n-1} - 1)$ **z.B. $n = 8 \rightarrow z_{min} = -127_{10}$**

Grundidee des Dual–Offset–Codes:

Die ganzen Zahlen

$$Z_{min} = -2^{n-1} \quad \leq Z \leq \quad Z_{max} = +(2^{n-1} - 1)$$

werden durch eine Nullpunkt-Verschiebung (Offset-Addition) so verschoben, dass alle Werte positiv werden. Diese werden dann im n-bit Dualcode codiert:

$$Z_{min} \leq Z \leq Z_{max} \underbrace{\mapsto}_{\textit{Verschiebung}} 0 \leq \mathbf{Y = Z + |Z_{min}|} \leq 2^n - 1 \underbrace{\mapsto}_{\textit{Dualcodierung}} (Y)_2$$

Vorteile und Nachteile:

- \+ **Relativ einfache** Codierung
- \+ Genau **eine** Darstellung für **Null**
- – **Asymmetrie:** $|Z_{min}| \neq Z_{max}$ (prinzipbedingt immer vorhanden)
- – **Zwei verschiedene Rechenwerke** für $Z_1 + Z_2$ und $Z_1 - Z_2$ (wie bei Sign–Magnitude–Codierung)
- – Komplizierte Rechenoperationen: **Korrekturterme** für Addition und Subtraktion **notwendig**.

Beispiele:

Bestimmen Sie die Minimal- und Maximalwerte des 4 bit Dual-Offset-Codes:

$Z_{max} = +(2^3 - 1) = +7 \quad Y_{max} = Z_{max} + |Z_{min}| = 7 + 8 = 15 = (1111)_2$

$Z_{min} = -2^3 = -8 \qquad Y_{min} = Z_{min} + |Z_{min}| = -8 + 8 = 0 = (0000)_2$

Rechnen Sie die folgenden Dezimalzahlen in den 4 bit Dual-Offset-Code um:

- $Z = -6, \quad Y = Z + |Z_{min}| = -6 + 8 = (2)_{10} = (0010)_2 = (2)_{16}$
- $Z = +5, \quad Y = Z + |Z_{min}| = +5 + 8 = (13)_{10} = (1101)_2 = (D)_{16}$

Zahlenbereich bei n bit Dual-Offset-Zahlen

- Max: $Z_{max} = +(2^{n-1} - 1) \quad \rightarrow Y_{max} = (11\cdots1)_2$
- Min: $Z_{min} = -2^{n-1} \quad \rightarrow Y_{min} = (00\cdots0)_2$

Offset–Code–Darstellung (Excess-Code, Stiplitz–Code)

Beispiel vollständiger Offset–8–Code

Z Dezimal	Y Dezimal-Offset-8	$(Y)_2$ Dual–Offset–8–Code von Z	$(Y)_{16}$ Hex–Offset–8–Code von Z
−8	0	0000	0
−7	1	0001	1
−6	2	0010	2
−5	3	0011	3
−4	4	0100	4
−3	5	0101	5
−2	6	0110	6
−1	7	0111	7
+0	8	1000	8
+1	9	1001	9
+2	10	1010	*A*
+3	11	1011	*B*
+4	12	1100	*C*
+5	13	1101	*D*
+6	14	1110	*E*
+7	15	1111	*F*

Z_i seien Dezimalzahlen, die im Dual-Offset-Code als $Y_i = Z_i + C$ mit dem Offset $C := |Z_{min}|$ codiert sind:

Addition: $Z_A = Z_1 + Z_2 \longmapsto Y_A$

$$Y_1 = Z_1 + C$$
$$Y_2 = Z_2 + C$$
$$Y_1 + Y_2 = Z_1 + Z_2 + 2 \cdot C$$

$$Y_A = Z_1 + Z_2 + C = \underbrace{(\ Y_1 + Y_2\ )}_{\text{Ergebnis der Rechnung mit den Codeworten}} \underbrace{-\ C}_{\text{Korrektursub.}}$$

Subtraktion: $Z_S = Z_1 - Z_2 \longmapsto Y_S$

$$Y_1 = Z_1 + C$$
$$Y_2 = Z_2 + C$$
$$Y_1 - Y_2 = Z_1 + \cancel{C} - Z_2 + \cancel{C}$$

$$Y_S = Z_1 - Z_2 + C = \underbrace{(\ Y_1 - Y_2\ )}_{\text{Ergebnis der Rechnung mit den Codeworten}} \underbrace{+\ C}_{\text{Korrekturadd.}}$$

Gegeben sind Dual-Offset-Zahlen mit **n = 4 Bit** $\rightarrow C = (8)_{10} = (1000)_2$

Addition :

	0011	$=$	$(-5)_{10}$	$Z = Y - C$
$+$	0110	$= +(-2)_{10}$		
	1100	**Ü**		
	1001	$=$	$(+1)_{10}$	müsste aber $(-7)_{10} = 0001$ sein
$-$	1000	Korrektursubtraktion $-C = -(8)_{10}$		
	0001	$=$	$(-7)_{10}$	

Subtraktion :

0011	$=$	$(-5)_{10}$	
$-$ 0110	$= -(-2)_{10}$		
(1)1000	**Ü**		
1101	$=$	$(+5)_{10}$	müsste aber $(-3)_{10} = 0101$ sein
$+$ 1000	Korrekturaddition $+C = +(8)_{10}$		
(1)0101	$=$	$(-3)_{10}$	

Trick: Wenn $C = 2^{n-1}$ ist, kann die Korrekturaddition bzw. -subtraktion durchgeführt werden, indem im Ergebnis einfach das MSB invertiert wird.

Führen Sie die folgende Rechnungen im Dual-Offset-Code mit **n = 4 Bit** durch:

Ü B U N G

Addition :

$$\begin{array}{rll} & 0011 & = (-5)_{10} \\ + & 1010 & = +(+2)_{10} \\ & 0100 & \\ \hline & 1101 & \\ - & & \text{Korrektursubtraktion} \rightarrow \text{MSB invert.} \\ \hline & 0101 & = (-3)_{10} \end{array}$$

Subtraktion :

$$\begin{array}{rll} & 1101 & = +(+5)_{10} \\ - & 1001 & = -(+1)_{10} \\ & 0000 & \\ \hline & 0100 & \\ + & & \text{Korrekturaddition} \rightarrow \text{MSB invertieren} \\ \hline & 1100 & = (+4)_{10} \end{array}$$

Annahme: Alle Codeworte sollen n–stellige Binärworte sein

Ziel: Codierung so, dass nur **ein Rechenwerk** für Addition und Subtraktion

Beispiel: n=2 stellige Dezimalzahlen

61		61		
− 37		+ 63		63=100−37, 63 ist das 10^2er Komplement von −37
24		124		
		− 100		entspricht Weglassen der n+1. (=vordersten) Stelle
		24		

Addition : $Z_A = Z_1 + Z_2$ Mit Addierwerk berechenbar!!

Subtraktion : $Z_S = Z_1 - Z_2 = Z_1 + (-Z_2) = Z_1 + \underbrace{(C - Z_2)}_{\text{Komplement von } Z_2 \text{ bezüglich C}} \underbrace{- C}_{\text{Reduktion um C}}$

Anforderungen an Rechenwerk und Codierung:

- Allgemeines Addierwerk
- Spezielle Operationen: $\Rightarrow$ Komplementbildung: $-Z = C - Z$

  müssen einfach sein $\Rightarrow$ Reduktion um C: $Z - C$

Gut geeignet für Implementierung: **Zweier–Komplement mit $C_Z = 2^n$**

- Standarddarstellung für ganze Zahlen in Rechnern (z.B. int in C/Java)
- **Darstellung von Zahlen im n-bit Zweier-Komplement-Code**
  - positive Zahlen: $(Z)_{ZK} = (Z)_2$ **wie Dualcode**
  - negative Zahlen: $(Z)_{ZK} = (2^n - |Z|)_2$ **Komplement des Betrags dual cod.**

    Umrechnung Dezimal $\leftrightarrow$ Zweier-Komplement siehe nächste Seiten
- Zahlenbereich bei n bit Zweier-Komplement-Zahlen
  - Min: $Z_{min} = -2^{n-1} = (10\ldots0)_{ZK}$
  - Max: $Z_{max} = +(2^{n-1} - 1) = (01\ldots1)_{ZK}$

    Bsp.: $n = 8$ $Z_{min} = -2^7 = -128$, $Z_{max} = 2^7 - 1 = +127$

    Bsp.: $n = 16$ $Z_{min} = -2^{15} = -32768$, $Z_{max} = 2^{15} - 1 = +32767$
- Vorzeichenerkennung: $Z \geq 0 \rightarrow$ **MSB = 0**, $Z < 0 \rightarrow$ **MSB = 1**

Umrechnung Dezimalzahl Z in Zweier-Komplementzahl $(Z)_{ZK}$

- Positive Zahl $0 \leq Z \leq Z_{max} \rightarrow (Z)_{ZK} = (Z)_2 =$ normaler Dualcode
- Negative Zahl $Z_{min} \leq Z < 0$:
  - Algorithmus für Menschen: $(Z)_{ZK} = (2^n - |Z|)_2$
    Differenz dezimal berechnen → Ergebnis in Dualcode umrechnen
  - Algorithmus für Computer: $(Z)_{ZK} = \overline{(|Z|)_2} + 1$
    Betrag in Dualcode umrechnen → bitweise invertieren → +1 addieren

    Hinweis: $(Z)_{EK} = \overline{(|Z|)_2}$ wird auch als Einer-Komplement bezeichnet.

Bsp.: $n = 8$ **ÜBUNG**

$Z = +96_{10} \rightarrow (Z)_{ZK} = \mathbf{64 + 32 = (0110\ 0000)_2 = (0110\ 0000)_{ZK}}$

$Z = -96_{10} \rightarrow (Z)_{ZK} = \mathbf{2^8 - 96 = 160 = 128 + 32 = (1010\ 0000)_{ZK}}$

alternativ $\mathbf{(Z)_{ZK} = \overline{(0110\ 0000)_2} + 1 = 1001\ 1111 + 1 = (1010\ 0000)_{ZK}}$

$Z = +39_{10} \rightarrow (Z)_{ZK} = \mathbf{32 + 7 = (0010\ 0111)_2 = (0010\ 0111)_{ZK}}$

$Z = -39_{10} \rightarrow (Z)_{ZK} = \mathbf{\overline{(0010\ 0111)_2} + 1 = (1101\ 1001)_{ZK}}$

Umrechnung Zweier-Komplementzahl $(Z)_{ZK}$ in Dezimalzahl Z

- MSB = 0 → Positive Zahl $Z > 0 \rightarrow Z = +((Z)_2)_{10}$
  d.h. Umrechnung wie normale Dualzahl s.h. S. 3.13
- MSB = 1 → Negative Zahl $Z < 0$:
  - Algorithmus für Menschen: $Z = -(2^n - ((Z)_2)_{10})$

    Zahl wie normale Dualzahl in Dezimalzahl umrechnen s.h. S. 3.13
    → Ergebnis von 2^n abziehen → „−“-Zeichen davorstellen
  - Algorithmus für Computer: $Z = -(\overline{(Z)_{ZK}} + 1)_{10}$

    Bitweise invertieren → +1 addieren → Ergebnis wie normalen Dualcode in Dezimalzahl umrechnen s.h. S. 3.13→ „−“-Zeichen davorstellen

Bsp.: $n = 8$ **ÜBUNG**

$(Z)_{ZK} = (0110\ 0000)_{ZK} \geq 0 \rightarrow Z = 64 + 32 = (+96)_{10}$

$(Z)_{ZK} = (1010\ 0000)_{ZK} < 0 \rightarrow Z = -(2^8 - (128 + 32)) = (-96)_{10}$

alternativ $\overline{1010\ 0000}_2 + 1 = (0101\ 1111 + 1)_2 = (0110\ 0000)_2$
$= -(64 + 32) = (-96)_{10}$

.

$(Z)_{ZK} = (0010\ 0111)_{ZK} \geq 0 \rightarrow Z = +(0010\ 0111)_2 = 32 + 7 = (39)_{10}$

$(Z)_{ZK} = (1101\ 1001)_{ZK} < 0 \rightarrow Z = -(\overline{1101\ 1001}_2 + 1) = -(00100111)_2 = (-39)_{10}$

Komplement–Darstellung

Codetabellen von Betragszahlen und ganzen Zahlen für n=4

Dezimal	Betragszahl Dualcode	Ganze Zahl 2er-Komplement	Ganze Zahl Dual–Offset–Code	Ganze Zahl Signed–Magnitude
−8		1000	0000	
−7		1001	0001	1111
−6		1010	0010	1110
−5		1011	0011	1101
−4		1100	0100	1100
−3		1101	0101	1011
−2		1110	0110	1010
−1		1111	0111	1001
0	0000	0000	1000	+0 : 0000 und − 0 : 1000
1	0001	0001	1001	0001
2	0010	0010	1010	0010
3	0011	0011	1011	0011
4	0100	0100	1100	0100
5	0101	0101	1101	0101
6	0110	0110	1110	0110
7	0111	0111	1111	0111
8	1000			
9	1001			
10	1010			
11	1011			
12	1100			
13	1101			
14	1110			
15	1111			

Positive ganze Zahlen im 2er-Komplement-Code identisch zu Betragszahlen im Dualcode
MSB bei Dual-Offset-Code invertiert zu 2er-Komplement-Code, Rest identisch **!!!**

Rechnen mit Zahlen im Zweier-Komplement-Code

- **Additionen** $Z_A = Z_1 + Z_2$
  - Summanden und Ergebnis werden im Zweier-Komplement-Code dargestellt.
  - Addition erfolgt wie beim normalen Dualcode.
- **Subtraktion** $Z_S = Z_1 - Z_2 = Z_1 + (-Z_2)$
  - Statt Z_2 zu subtrahieren, wird $-Z_2$ addiert.
  - Z_1 und $-Z_2$ sowie das Ergebnis werden im Zweier-Komplement-Code dargestellt.
  - Addition erfolgt wie beim normalen Dualcode.
  - Die theoretisch notwendige Reduktion des Ergebnisses um 2^n besteht darin, dass man das Ergebnis auf n bit verkürzt, wenn es länger ist.

Beispiel 1 zum Rechnen mit 2er–Komplement–Darstellung:

gegeben: $Z_1 = 2 \quad Z_2 = 3, \quad n = 3$ gesucht: $Z_S = Z_1 - Z_2$

$$\begin{aligned} Z_1 &= (010)_2 \Rightarrow (Z_1)_{ZK} = (010)_{ZK} \\ -Z_2 &= -(011)_2 \Rightarrow +(-Z_2)_{ZK} = +(101)_{ZK} \\ &\text{Ü:} \quad \text{———————} \\ (Z_S)_{ZK} &= (111)_{ZK} = -(001)_2 = -1_{10} \end{aligned}$$

kein Überlauf möglich

Beispiel 2 zum Rechnen mit 2er–Komplement–Darstellung:

gegeben: $Z_1 = 3 \quad Z_2 = 2, \quad n = 3$ gesucht: $Z_S = Z_1 - Z_2$

$$\begin{aligned} Z_1 &= (011)_2 \Rightarrow (Z_1)_{ZK} = (011)_{ZK} \\ -Z_2 &= -(010)_2 \Rightarrow +(-Z_2)_{ZK} = (110)_{ZK} \\ &\text{Ü:} \quad \not{1}10 \\ (Z_S)_{ZK} &= (001)_{ZK} = +(001)_2 = +1_{10} \end{aligned}$$

Reduktion um 2^n kein ÜL möglich

Beispiel 3: **ÜBUNG**

Wandeln Sie die beiden Zahlen Z_1 und Z_2 in das 2-er Komplement um und führen Sie anschließend die jeweilige Operation durch:

Addition $Z_A = Z_1 + Z_2$: $Z_1 = 4_{10}$, $Z_2 = 2_{10}$, n=4 Bit

$$\begin{aligned} Z_1 &= (0100)_2 \Rightarrow (Z_1)_{ZK} = (0100)_{ZK} \\ +Z_2 &= +(0010)_2 \Rightarrow +(Z_2)_{ZK} = +(0010)_{ZK} \\ &\text{Ü:} \\ (Z_A)_{ZK} &= (0110)_{ZK} = +(0110)_2 = +6_{10} \end{aligned}$$

Subtraktion $Z_S = Z_1 - Z_2$: $Z_1 = 4_{10}$, $Z_2 = 2_{10}$, n=4 Bit

$$\begin{aligned} Z_1 &= (0100)_2 \Rightarrow (Z_1)_{ZK} = (0100)_{ZK} \\ -Z_2 &= -(0010)_2 \Rightarrow +(-Z_2)_{ZK} = +(1110)_{ZK} \\ &\text{Ü:} \quad \not{1}100 \\ (Z_S)_{ZK} &= (0010)_{ZK} = +(0010)_2 = +2_{10} \end{aligned}$$

Beispiel 4: **ÜBUNG**

Wandeln Sie die beiden Zahlen Z_1 und Z_2 in das 2-er Komplement um und führen Sie anschließend die jeweilige Operation durch:

Addition $Z_A = Z_1 + Z_2$: $Z_1 = 3_{10}$, $Z_2 = 7_{10}$, n=4 Bit

$$\begin{aligned} Z_1 &= (0011)_2 \quad \Rightarrow \quad (Z_1)_{ZK} = (0011)_{ZK} \\ +Z_2 &= +(0111)_2 \quad \Rightarrow \quad +(Z_2)_{ZK} = (0111)_{ZK} \\ & \qquad\qquad\qquad\quad \text{Ü:} \quad 011 \\ & \qquad\qquad\qquad\quad (Z_A)_{ZK} = (1010)_{ZK} = -\cancel{(0110)_2 = -6_{10}} \end{aligned}$$

Fehler durch Überlauf

Subtraktion $Z_S = Z_1 - Z_2$: $Z_1 = 3_{10}$, $Z_2 = 7_{10}$, n=4 Bit

$$\begin{aligned} Z_1 &= (0011)_2 \quad \Rightarrow \quad (Z_1)_{ZK} = (0011)_{ZK} \\ -Z_2 &= -(0111)_2 \quad \Rightarrow \quad +(-Z_2)_{ZK} = +(1001)_{ZK} \\ & \qquad\qquad\qquad\quad \text{Ü:} \quad 011 \\ & \qquad\qquad\qquad\quad (Z_S)_{ZK} = (1100)_{ZK} = -(0100)_2 = -4_{10} \end{aligned}$$

Beispiel 5: **ÜBUNG**

Wandeln Sie die beiden Zahlen Z_1 und Z_2 in das 2-er Komplement um und führen Sie dann die jeweilige Operation durch:

Subtraktion $Z_S = Z_1 - Z_2$: $Z_1 = 11_{10}$, $Z_2 = 6_{10}$, n=5 Bit

$$\begin{aligned} Z_1 &= (01011)_2 \Rightarrow (Z_1)_{ZK} = (01011)_{ZK} = +11_{10} \\ -Z_2 &= -(00110)_2 \Rightarrow +(-Z_2)_{ZK} = +(11010)_{ZK} = -6_{10} \\ \text{Ü:} & \quad \not{1}1010 \\ (Z_S)_{ZK} &= (00101)_{ZK} = +(00101)_2 = +5_{10} \end{aligned}$$

Subtraktion $Z_S = Z_1 - Z_2$: $Z_1 = (10001)_{ZK}$, $Z_2 = (11101)_{ZK}$, n=5 Bit

$$\begin{aligned} Z_1 &= (10001)_{ZK} \Rightarrow (Z_1)_{ZK} = (10001)_{ZK} = -15_{10} \\ -Z_2 &= -(11101)_{ZK} \Rightarrow +(-Z_2)_{ZK} = +(00011)_{ZK} = +3_{10} \\ \text{Ü:} & \quad 0011 \\ (Z_S)_{ZK} &= (10100)_{ZK} = -(01100)_2 = -12 \end{aligned}$$

Ein **Überlauf** (engl. **Overflow**), d.h. ein falsches Ergebnis, tritt auf, wenn das Ergebnis einer Operation nicht mehr mit den verfügbaren **n-bit darstellbar** ist, d.h. falls

- $Z_1 \pm Z_2 > Z_{max}$ oder $Z_1 \pm Z_2 < Z_{min}$ ist.

Mathematisch: Die Berechnung erfolgt Modulo 2^n, also $(Z_1 \pm Z_2)\ mod\ 2^n$.

Rechenwerke/Mikroprozessoren zeigen Überläufe durch Statusbits an:

- Bei **Operationen mit natürlichen Zahlen (Betragszahlen)**:

  Carry Flag CF$= 1$, wenn $C_n = 1$ (bei Addition) bzw. $\overline{C_n} = 1$ bei Subtraktion (siehe S. 3.34)
- Bei **2er-Komplement-Zahlen**: **Overflow-Flag OF** (bzw. OV).
- Die Statusbits können/müssen vom Softwareprogramm abgefragt werden.

Überläufe bei 2er-Komplement-Zahlen können nur auftreten, bei

- einer Addition, wenn beide Zahlen dasselbe Vorzeichen haben
- einer Subtraktion, wenn beide Zahlen unterschiedliches Vorzeichen haben

Beispiele für $(S) = [(X) \pm (Y)] \bmod 2^n$ mit **2er-Komplement–Zahlen** und **n = 4**:

Addition von zwei pos. Zahlen

X		0	1	0	0	$+4_{10}$
Y		0	0	1	1	$+3_{10}$
C	0	0	0	0	0	
S		0	1	1	1	7_{10}

Ergebnis: korrekt

Addition von zwei neg. Zahlen

X		1	1	0	0	-4_{10}
Y		1	1	0	0	-4_{10}
C	1	1	0	0	0	
S		1	0	0	0	-8_{10}

Ergebnis: korrekt

Addition je einer pos. und neg. Zahl

X		0	1	1	1	$+7_{10}$
Y		1	0	0	0	-8_{10}
C	0	0	0	0	0	
S		1	1	1	1	-1_{10}

Ergebnis: korrekt

X		0	1	0	0	$+4_{10}$
Y		0	1	0	0	$+4_{10}$
C	0	1	0	0	0	
S		1	0	0	0	-8_{10}

Ergebnis: falsch

X		1	1	0	0	-4_{10}
Y		1	0	1	1	-5_{10}
C	1	0	0	0	0	
S		0	1	1	1	$+7_{10}$

Ergebnis: falsch

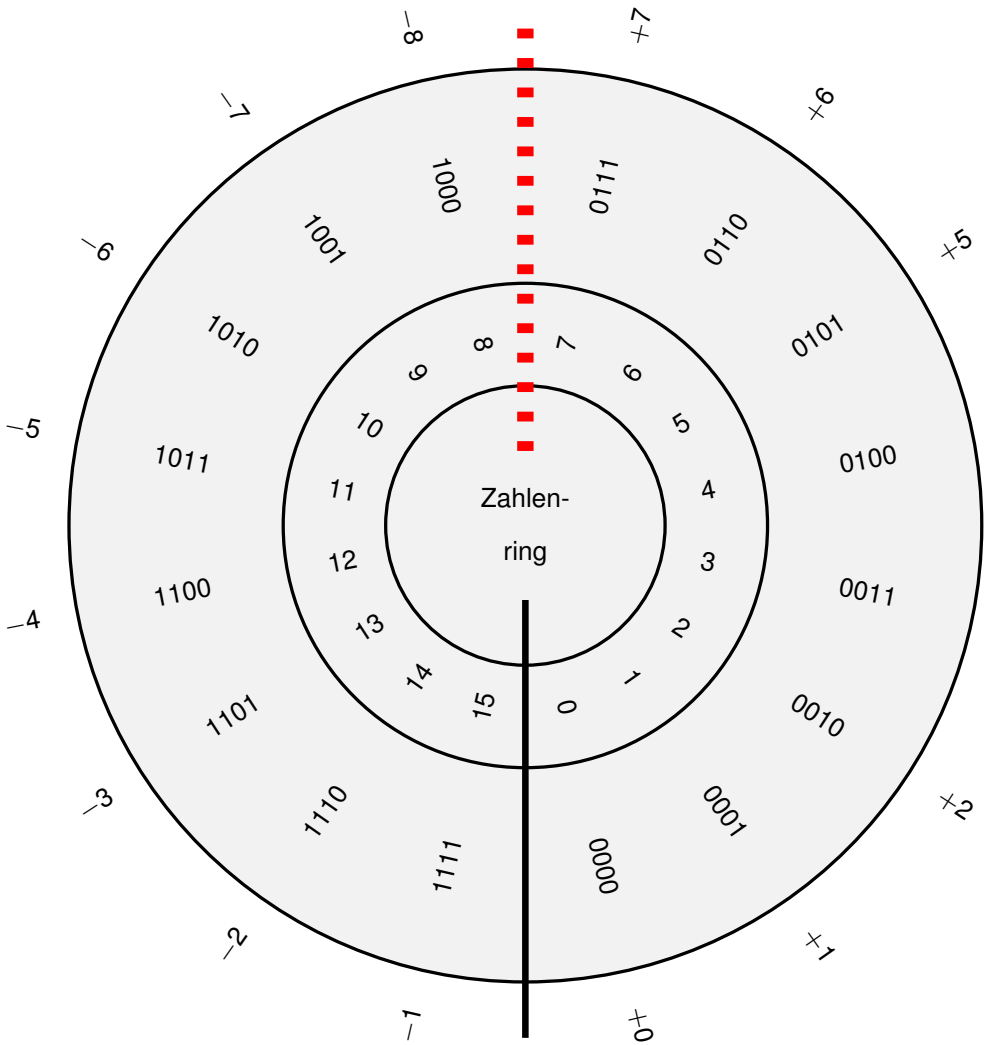
Überlauf OF=1 bei 2er-Komp.:

$$OF = C_n \leftrightarrow C_{n-1} \text{ XOR}$$

Bei Betragszahlen: Addition $CF = C_n$

Bei Betragszahlen: Subtraktion $CF = \overline{C_n}$

Andere Darstellung der Code-Tabelle von S. 3.52: **Zahlenring**



Ganze Zahlen: Über/Unterlauf bei Überschreiten der roten Linie !!!

Natürliche Zahlen: Über/Unterlauf bei Überschreiten der schwarzen Linie !!!

Innerer Ring: Natürliche Zahlen dezimal - Mittlerer Ring: Binärcode mit n=4 - Äußerer Ring: Ganze Zahlen dezimal

Achtung:
Bei der Umwandlung und Interpretation codierter Zahlen müssen immer folgende Informationen bekannt sein:

- Anzahl der Stellen n
- Position des Kommas Default: Ganz rechts
- Zahlenbasis B, z.B. $B = 2$ (binär), 16 (hexadezimal) oder 10 (dezimal)

  Wenn die Basis nicht explizit angegeben ist, ergibt sie sich oft aus dem Kontext, z.B. 32 ist definitiv nicht binär, könnte aber dezimal oder hexadezimal sein, 3A ist sicher hexadezimal.

  Im Computerbereich werden Zahlen statt binär oft hexadezimal geschrieben. Dies ist unabhängig von der verwendeten Codierung (nächster Punkt).
- Verwendete Codierung
  - Natürliche Zahlen oder positive rationale Zahlen: Dualcode (S. 3.11)
  - Ganze Zahlen oder positive und negative rationale Zahlen:
    * Vorzeichen-Betrag (S. 3.38),
    * Dual-Offset (S. 3.41),
    * Zweier-Komplement (S. 3.50, Default im Computerbereich)
  - Gleitkommazahlen: IEEE 754 (S. 3.66)

Wiederholung: Festkomma–Zahlen im Sign–Magnitude–Dualcode

$$Z = \pm \sum_{\nu=-s}^{l} z_\nu \cdot 2^\nu = \pm (z_l \dots z_1 z_0 \ , \ z_{-1} z_{-2} \dots z_{-s})_2$$

Abstand benachbarter Festkomma–Maschinenzahlen:

- absoluter Abstand: $\Delta Z = 2^{-s} = \text{konst.}$
- relativer Abstand = relativer Fehler: $\frac{\Delta Z}{|Z|}$ ist abhängig von $|Z|$

Probleme: Ein großer Zahlenbereich benötigt **sehr viele Stellen**.
Der **relative Fehler** hängt vom **Zahlenwert** ab! Je kleiner eine Zahl, desto größer ihr relativer Fehler.
schlecht, Wunsch: konstanter relativer Fehler!

Lösung: Gleitkomma–Darstellung: $\mathbf{Z = M \cdot B^E}$ (halblogarithm. Darstellung)

Beispiele:

B	M	E		Ergebnis Z		
10	$1,23$	3	$\Longrightarrow$	$1,23 \cdot 10^3$	$=$	1230
10	$-1,23$	3	$\Longrightarrow$	$-1,23 \cdot 10^3$	$=$	-1230
10	$-1,23$	-3	$\Longrightarrow$	$-1,23 \cdot 10^{-3}$	$=$	$-0,00123$

Halblogarithmische Darstellung: $\mathbf{Z = M \cdot B^E}$

- **B (Zahlenbasis)**: Hier wird nur der Sonderfall **B = 2** betrachtet.
- **M (Mantisse)**: **Festkommazahl** in **Sign–Magnitude–Dualcode** mit $l + 1 = 1$, d.h. genau eine Stelle vor dem Komma

$$M = \pm \sum_{\nu=-s}^{0} m_\nu \cdot 2^\nu = \pm (m_0 \ , \ m_{-1} \ldots m_{-s})_2$$

  Z heißt **normalisiert**, falls immer $m_0 = 1$ $\rightarrow$ eindeutige Darstellung

- **E (Exponent)**: **Ganze Zahl** codiert in einem **Dual–Offset–Code**

Folge: Der darstellbare Zahlenbereich wird durch die Stellenzahl des Exponenten, der relative Fehler durch die Stellenzahl der Mantisse festgelegt.

Bsp.: Mantisse mit 23 bit $\rightarrow log_{10}(2^{23}) \approx 7$ Dezimalstellen

Darstellung von Gleitkommazahlen im Standard IEEE 754:

Einfache Genauigkeit: 32 Bit (in C/Java: float)

Bits	1	8	23
	VZ von M	$E + 127$	$\|M\|$ ohne m_0

Doppelte Genauigkeit: 64 Bit (in C/Java: double)

Bits	1	11	52
	VZ von M	$E + 1023$	$\|M\|$ ohne m_0

- Die **Mantisse** wird **normalisiert im Sign-Magnitude Format** aber **ohne das Bit m_0** gespeichert.
- Nach allen Rechenoperationen wird das Ergebnis durch Rechts- bzw. Linksschieben der Mantisse und Anpassen des Exponenten wieder **normalisiert** ($m_0 = 1$).
- Die **Exponenten** sind in **Offset 127** bzw. **1023** codiert.
- Zahlenbereich (das Exponentenfeld darf nicht 0...0 oder 1...1 sein, siehe unten):

  einfache Genauigkeit: $2^{-126} \approx 10^{-38} \lesssim |Z| \lesssim 10^{38} \approx 2^{127}$

  doppelte Genauigkeit: $2^{-1022} \approx 10^{-308} \lesssim |Z| \lesssim 10^{308} \approx 2^{1023}$

- Für die Zahl **Null, die nicht normalisiert dargestellt werden kann** und für **weitere Ausnahmefälle** sind Sondercodierungen **0 . . . 0** und **1 . . . 1** im Exponenten- bzw. Mantissenfeld reserviert.

Null	$\pm$	0 . . . 0	0 . . . 0
Normalisierte Zahl	$\pm$	$0 < \text{E}+127 < 255$	Mantisse beliebig
Denormalisierte Zahl	$\pm$	0 . . . 0	Mantisse: nicht alle Bits 0
Unendlich Inf(inite)	$\pm$	1 . . . 1	0 . . . 0
Not a Number NaN	$\pm$	1 . . . 1	Mantisse: nicht alle Bits 0

Beispiel für Gleitkomma–Codierung mit einfacher Genauigkeit:

Bits	1	8	23
	VZ von M	$E + 127$	$\lvert M\rvert$ ohne m_0
binär	**0**	**1011 1010**	**0000 1111 0000 1111 0000 111**

- Vorzeichen von M ist $+$
- $E = \left(2^7 + 2^5 + 2^4 + 2^3 + 2^1\right) \boxed{-\mathbf{127}} = \left(11 \cdot 16^1 + 10 \cdot 16^0\right) \boxed{-\mathbf{127}} = 59$
- $|M| = \boxed{\mathbf{1}} + 0 \cdot 16^{-1} + 15 \cdot 16^{-2} + 0 \cdot 16^{-3} + 15 \cdot 16^{-4} + 0 \cdot 16^{-5} + 14 \cdot 16^{-6} = 1,0588\ldots$
- $Z \approx +1,0588\ldots \cdot 2^{59} \approx \underline{6,1037 \cdot 10^{17}}$

Gegeben ist die Gleitkommazahl: $(\mathbf{4258\ 0000})_{\mathbf{16}}$ nach IEEE 754 (einfache Genauigkeit).

Welcher Dezimalzahl entspricht diese Zahl?

$$\overbrace{0100}^{4}\ \overbrace{0010}^{2}\ \overbrace{0101}^{5}\ \overbrace{1000}^{8}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}$$

$$\underbrace{0}_{VZ}\ \underbrace{100\ 0010\ 0}_{Exp.}\underbrace{101\ 1000\ 0000\ 0000\ 0000\ 0000}_{Mant.}$$

Exp: $E + 127 = 1000\ 0100 = 132_{10} \rightarrow$ Korrektur: $E = 132 - 127 = 5$

VZ M: $0 \rightarrow$ „+“

Mant. **1,** $101\ 1000\ 0000\ 0000\ 0000\ 0000 \rightarrow 1,1011$

Dies ergibt: $+1,1011 \cdot 2^5 = +110110 = \underline{(+\mathbf{54,0})_{\mathbf{10}}}$

Gegeben ist die Gleitkommazahl: $(\mathbf{C302\ 8000})_{16}$ nach IEEE 754 (einfache Genauigkeit). **Ü B U N G**

Welcher Dezimalzahl entspricht diese Zahl?

$$\overbrace{1100}^{C}\ \overbrace{0011}^{3}\ \overbrace{0000}^{0}\ \overbrace{0010}^{2}\ \overbrace{1000}^{8}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}$$

$$\underbrace{1}_{VZ}\ \underbrace{100\ 0011\ 0}_{Exp.}\underbrace{000\ 0010\ 1000\ 0000\ 0000\ 0000}_{Mant.}$$

Exp: $E + 127 = 1000\ 0110 = 134_{10} \rightarrow$ Korrektur: $E = 134 - 127 = 7$

VZ M: $1 \rightarrow$ „-“

Mant. **1,** $000\ 0010\ 1000\ 0000\ 0000\ 0000 \rightarrow 1,00000101$

Dies ergibt: $-1,00000101 \cdot 2^7 = -10000010,1 = \underline{(-\mathbf{130,5})_{10}}$

Gegeben ist die Gleitkommazahl: $(\mathbf{7F80\ 0000})_{16}$ nach IEEE 754 (einfache Genauigkeit).

Ü B U N G

Welcher Dezimalzahl entspricht diese Zahl?

$$\overbrace{0111}^{7}\ \overbrace{1111}^{F}\ \overbrace{1000}^{8}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}\ \overbrace{0000}^{0}$$

$$\underbrace{0}_{VZ}\ \underbrace{111\ 1111\ 1}_{Exp.}\underbrace{000\ 0000\ 0000\ 0000\ 0000\ 0000}_{Mant.}$$

VZ: 0 $\rightarrow$ „+“

Exp: 1111 1111 $\rightarrow$ Sonderfall

Mant. 000 0000 0000 0000 0000 0000 $\rightarrow$ Sonderfall Unendlich

Dies ergibt $\underline{+\infty}$

Gegeben ist die Dezimalzahl $(-\mathbf{14,125})_{\mathbf{10}}$.

Welcher Gleitkommazahl nach IEEE 754 (einfache Genauigkeit) entspricht diese Zahl?

$-14,125 \rightarrow -1110,001 = -1,110001 \cdot 2^3$

VZ: „-“ $\rightarrow 1$

Exp: $3 + 127 = (130)_{10} = 10000010$

Mant. 110 0010 0000 0000 0000 0000

$$\underbrace{1}_{VZ}\ \overbrace{100\ 0001\ 0}^{Exp.}\ \overbrace{110\ 0010\ 0000\ 0000\ 0000\ 0000}^{Mant.}$$

$$\underbrace{1100}_{C}\ \underbrace{0001}_{1}\ \underbrace{0110}_{6}\ \underbrace{0010}_{2}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}$$

Ergebnis: $(\mathbf{C162\ 0000})_{\mathbf{16}}$

Gegeben ist die Dezimalzahl $(+\mathbf{5,625})_{\mathbf{10}}$. **Ü B U N G**

Welcher Gleitkommazahl nach IEEE 754 (einfache Genauigkeit) entspricht diese Zahl?

$+5,625 \rightarrow +101,101 \cdot 2^0 = +1,01101 \cdot 2^2$

VZ: „+“ $\rightarrow 0$

Exp: $2 + 127 = (129)_{10} = 10000001$

Mant. 011 0100 0000 0000 0000 0000

$$\underbrace{0}_{VZ}\ \underbrace{100\ 0000\ 1}_{Exp.}\ \underbrace{011\ 0100\ 0000\ 0000\ 0000\ 0000}_{Mant.}$$

$$\underbrace{0100}_{4}\ \underbrace{0000}_{0}\ \underbrace{1011}_{B}\ \underbrace{0100}_{4}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}$$

Ergebnis: $(\mathbf{40B4\ 0000})_{\mathbf{16}}$

Multiplikation mit Gleitkommazahlen:

$$Z_1 \cdot Z_2 = \left(M_1 \cdot 2^{E_1}\right) \cdot \left(M_2 \cdot 2^{E_2}\right) = (M_1 \cdot M_2) \cdot 2^{(E_1+E_2)}$$

Multiplikation von Festkommazahlen für die Mantissen und **Addition von ganzen Zahlen für die Exponenten**.

Division mit Gleitkommazahlen:

$$\frac{Z_1}{Z_2} = \frac{M_1 \cdot 2^{E_1}}{M_2 \cdot 2^{E_2}} = \left(\frac{M_1}{M_2}\right) \cdot 2^{(E_1-E_2)}$$

Division von Festkommazahlen für die Mantissen und **Subtraktion von ganzen Zahlen für die Exponenten**.

Addition/Subtraktion von Gleitkommazahlen:

$$Z_1 \pm Z_2 = \left(M_1 \cdot 2^{E_1}\right) \pm \left(M_2 \cdot 2^{E_2}\right) = \left(\widehat{M}_1 \pm \widehat{M}_2\right) \cdot 2^{\widehat{E}}$$

Addition/Subtraktion in zwei Schritten:

1. **Exponentenangleich**: $\widehat{E} = \max(E_1, E_2)$

   Mantisse des Summanden, der zum kleineren Exponenten gehört, um $|E_1 - E_2|$ Stellen nach rechts (= Komma nach links) schieben

   $\boxed{E_1 \geq E_2}$: $\widehat{E} = E_1;\quad \widehat{M}_1 = M_1;\quad \widehat{M}_2 = M_2 \cdot 2^{E_2 - E_1}$

   $\boxed{E_2 > E_1}$: $\widehat{E} = E_2;\quad \widehat{M}_2 = M_2;\quad \widehat{M}_1 = M_1 \cdot 2^{E_1 - E_2}$

2. **Addition/Subtraktion der Mantissen**

Beispiel:

$$\underbrace{1,00000100 \cdot 2^3}_{=Z_1} + \underbrace{1,11000000 \cdot 2^{-3}}_{=Z_2} = (1,00000100 + 0,00000111) \cdot 2^3 = 1,00001011 \cdot 2^3$$

$\Rightarrow$ Rechnen mit Gleitkommazahlen komplexer als mit Festkommazahlen

- In der Praxis werden fast ausschließlich Zahlenformate mit **8**, **16**, **32** oder **64** Bits eingesetzt.
- In den meisten Programmiersprachen gibt es vordefinierte ganzzahlige Datentypen
- Achtung: Die Datentypen haben in den Programmiersprachen **Java**, **C#** und **C/C++** unterschiedliche Bezeichnungen.

Bereich	Format	Typische Bezeichnungen
$0 \ldots 255$	8 Bit ohne VZ (dualcodiert)	**byte**, **unsigned char**
$-128 \ldots 127$	8 Bit mit VZ (2er-Komplement)	**byte**, **sbyte**, **char**
$0 \ldots 65535$	16 Bit ohne VZ (dualcodiert)	**ushort**, **unsigned short**
$-32768 \ldots 32767$	16 Bit mit VZ (2er-Komplement)	**short**, **short**, **short**
$0 \ldots 2^{32}-1$	32 Bit ohne VZ (dualcodiert)	**uint**, **unsigned int/long**
$-2^{31} \ldots 2^{31}-1$	32 Bit mit VZ (2er-Komplement)	**int**, **int**, **int/long**
$0 \ldots 2^{64}-1$	64 Bit ohne VZ (dualcodiert)	**ulong**, **unsigned _int64**
$-2^{63} \ldots 2^{63}-1$	64 Bit mit VZ (2er-Komplement)	**long**, **long**, **_int64**

[29] Quelle Gumm, Sommer: [18]

- Für die Darstellung von Gleitpunktzahlen in einem Text benutzt man z.B. die Notation „3.14 E 2“ oder „3.14E2“ . „**E**“ soll an „**Exponent**“ erinnern. Diese Notation ist in der Norm IEEE 754 festgelegt und in vielen Programmiersprachen üblich.
- Statt des deutschen Kommas wird im Englischen (und daher in fast allen Programmiersprachen) der Dezimalpunkt verwendet
- Die Bezeichnungen sind in den Programmiersprachen **Java**, **C#** und **C/C++** gleich.

Bereich	Format	Typische Bezeichnungen
± 1.5E-45 . . . 3.4E38	IEEE 754 einfache Genauigkeit 32 Bit	**float**, **float**, **float**
± 5.0E-324 . . . 1.7E308	IEEE 754 doppelte Genauigkeit 64 Bit	**double**, **double**, **double**

Single precision float wird gelegentlich auch als FP32, double precision als FP64 bezeichnet.

Bei der Implementierung von Machine Learning Algorithmen wird aus Geschwindigkeitsgründen neuerdings auch half precision FP16 eingesetzt (16 bit mit 5 bit Dual-Offset-15 Exponent und 10 bit Mantisse).

Definitionen

- **Alphanumerischer–Code** = Code für alphanumerische Zeichen (**Buchstaben und Ziffern**)
- **Zeichen–Code** = Code für alphanumerische Zeichen UND weitere **Symbole**, z.B. Satzzeichen.

ASCII–Code (American Standard Code for Information Interchange)

- Bekanntester Zeichen–Code.
- Ursprünglich Code mit **7**–bit $\rightarrow$ $\mathbf{2^7 = 128}$ verschiedene Zeichen, für US–Alphabet ausreichend.
- Erweitert auf **8**–bit $\rightarrow$ $\mathbf{2^8 = 256}$ verschiedene Zeichen, um zusätzlich auch europäische Zeichen, z.B. Umlaute, darstellen zu können.
- Regionale Varianten für kyrillische, griechische, und andere Zeichensätze. Hauptvariante für Mittel- und Nordeuropa **Latin–1**.
- Formal standardisiert: US-ASCII in ISO/IEC 646, Latin1-ASCII in ISO/IEC 8859
(ISO...International Standardization Organisation, IEC...International Electrotechnical Commission)

US–ASCII-Code in hexadezimaler Nummerierung:

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

8–bit = 2 Hex-Stellen $\rightarrow$ Zeile = 1. Hex-Stelle, Spalte = 2. Hex-Stelle

- Die original 7–bit ASCII-Zeichen entsprechen den Bytes **0000 0000** bis **0111 1111**, d.h. den Hex-Zahlen **00** bis **7F**$_{16}$.
- Steuerzeichen 0 - 31 = 00 - 1F$_{16}$ siehe nächste Seite
- Eine Datei, die nur ASCII-Zeichen enthält, nennt man **ASCII-Datei**.

Zeichen 0 bis 31 :
Nicht druckbare Steuerzeichen zur Datenübertragung und Textformatierung

Hex	Abk.	Bedeutung	Hex	Abk.	Bedeutung	Hex	Abk.	Bedeutung
0	**NUL**	**NULL**	1	SOH	Start of Heading	**2**	**STX**	**Start of Text**
3	**ETX**	**End of Text**	4	EOT	End of Transmission	5	ENQ	Enquiry
6	ACK	Acknowledge	7	BEL	Bell	**8**	**BS**	**Backspace**
9	**HT**	**Horizontal Tab**	**A**	**LF**	**Line Feed**	B	VT	Vertical Tab
C	FF	Form Feed	**D**	**CR**	**Carriage Return**	E	SO	Shift Out
F	SI	Shift in	10	DLE	Data Link Escape	11	DC1	Device Control 1
12	DC2	Device Control 2	13	DC3	Device Control 3	14	DC4	Device Control 4
15	NAK	Not Acknowledge	16	SYN	Synchronous Idle	17	ETB	End of Transmiss. Block
18	CAN	Cancel	19	EM	End of Medium	1A	SUB	Substitude
1B	**ESC**	**Escape**	1C	FS	File Separator	1D	GS	Group Separator
1E	RS	Record Separator	1F	US	Unit Separator			

Beispiele

- LF = Line Feed ... Cursor in neue Textzeile
- CR = Carriage Return ... Cursor auf Zeilenanfang
- BS = Backspace ... Cursor ein Zeichen zurück bewegen
- STX und ETX = Textanfang und Ende kennzeichnen

- Zur Codierung eines fortlaufenden Textes fügt man einfach die Codes der einzelnen Zeichen hintereinander.
- Eine Folge von Textzeichen heißt auch **Zeichenkette** (engl.: String).
- Der Text „Hallo Welt“ wird also durch die Zeichenfolge

  H e l l o W o r l d

  repräsentiert.
- Jedes dieser Zeichen, einschließlich des Leerzeichens „␣„ wird durch seine Nummer in der ASCII-Tabelle ersetzt:

  48 65 6C 6C 6F 20 57 6F 72 6C 64$_{16}$
- In C/C++ und vielen anderen Programmiersprachen werden Strings in Arrays gespeichert. In der Regel ist das Array länger als der String. Das Stringende wird daher durch ein NULL 00_H Byte nach dem letzten Zeichen markiert (**ASCIIZ = ASCII-Zero String**).
- Speicherplatzbedarf für ASCII-Zero String:
  1 Byte je ASCII-Zeichen + 1 NULL-Byte (als Stringende-Marke)

[30] Quelle Gumm, Sommer: [18]

- Erweiterung des ASCII-Codes: **Unicode** bzw. **Universal Character Set UCS-2** mit **16**–Bit, d.h. Zeichenvorrat $\mathbf{2^{16} = 65536}$ Zeichen
  - Die ersten 128 Zeichen des Unicodes sind identisch mit dem ASCII–Code.
  - Die zweiten 128 Zeichen sind identisch mit der Latin–1 ASCII–Erweiterung.
  - Die höherwertigen Codeworte stellen viele **nationale Sonderzeichen** bereit, z.B. griechische oder kyrillische Schriftzeichen.
- Für die Vielfalt der asiatischen und afrikanischen Schriften (China, Japan, Arabien, ...) reicht UCS-2 nicht aus, daher **UCS-4** mit **32**–Bit.

Probleme durch Unicode / UCS:

- **Speicherplatzbedarf**:
  - Datei 1 MByte in ASCII $\rightarrow$ 2 MByte in Unicode/UCS–2, 4 MByte in UCS–4
- **Kompatibilität**: Programmiersprachen und Editoren unterstützen meist nur eine Standard-Codierung (in C/C++ z.B. ASCII). Bei anderer Codierung Probleme, z.B. „komische“ Zeichen in Mails.

Abhilfe: **UTF–8**–Codierung (**UCS Transformation Format**):

- UTF-8 ist eine Codierung mit **variabler Wortlänge** von **1 bis 6 Bytes**.
- Alle originalen 128 ASCII-Zeichen werden mit **1 Byte codiert**, dabei immer MSB = 0. MSB = 1 markiert Zeichen mit mehr als einem Byte.
- UTF-8 heute **empfohlene Codierung im Web**, wird aber nicht von allen Programmiersprachen unterstützt.

Weitere Info: J. Spolsky, The absolute minimum every SW developer must know about character sets and Unicode.

Beispiel für unterschiedliche Textcodierungen

Text	S	c	h	ö	n	e		G	r	ü	ß	e
ASCII	53	63	68	F6	6E	65	20	47	72	FC	DF	65
Unicode	0053	0063	0068	00F6	006E	0065	0020	0047	0072	00FC	00DF	0065
UTF-8	53	63	68	C3B6	6E	65	20	47	72	C3BC	C39F	65

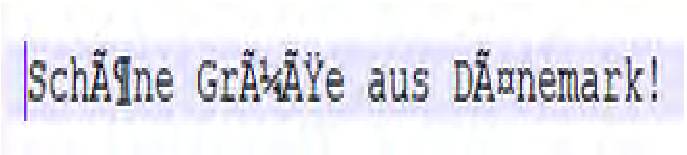
Alle Zahlenwerte in Hex

Speicherplatzbedarf:

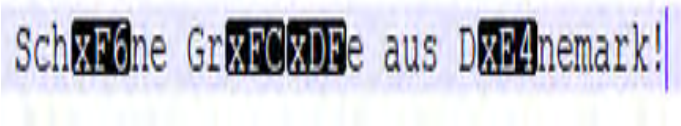
ASCII ISO 8859-Latin-1 : 12 Byte | Unicode UCS-2 BE : 24 Byte | UTF-8 : 15 Byte

Wenn eine Datei mit einem Programm in einer anderen Codierung geöffnet wird, kann die Darstellung fehlerhaft sein:

UTF-8-Mail in ASCII-Mail-Client



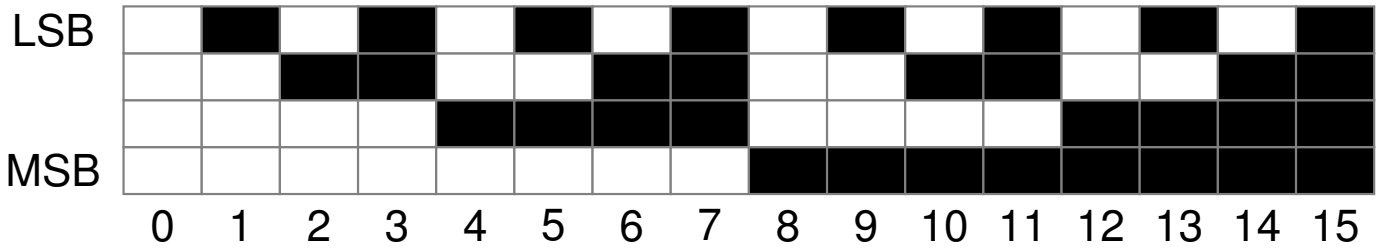
ASCII-Mail in UTF-8-Mail-Client



Häufige Aufgabenstellung:

- Optisches Abtasten codierter Information, z.B. QR- oder Bar-Code

- Ähnliche Anwendung: Codelineal zur Messung von Längen



Optische Marken im 4 bit-Dualcode (hell: 0; dunkel: 1)

- Mögliches Problem: Scanner wird leicht schräg über das Lineal geführt

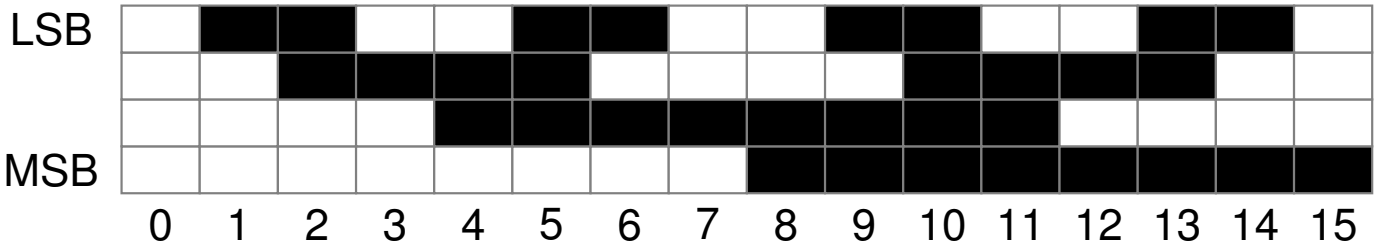
  Abtastzeile senkrecht über Feld 8 $\rightarrow 1000_2 = 8_{10} \rightarrow$ OK

  Abtastzeile linksschräg über Feld 7+8 $\rightarrow 1111_2 = 15_{10} \rightarrow$ FALSCH

- Abhilfe: Gray-Code

Gray-Code

- **Einschrittiger Code**, d.h. alle benachbarten Codeworte unterscheiden sich in genau einem Bit.



Optische Marken im 4 bit-Gray-Code (hell: 0; dunkel: 1)

- Falls der Scanner leicht schräg über das Lineal geführt wird, unterscheiden sich die Werte um maximal 1:

  Abtastzeile senkrecht über Feld 8 $\rightarrow 1100_G = 8_{10} \rightarrow$ OK

  Abtastzeile rechtsschräg über Feld 8+9 $\rightarrow 1101_G = 9_{10} \rightarrow$ OK

- Der Gray–Code besitzt keinen festen Stellenwert und ist damit für Rechenoperationen ungeeignet.
- Nach dem Messvorgang wird daher oft in den Dual–Code umgewandelt.

Allgemeines

- **Redundante Codes** benutzen **nicht alle möglichen Codeworte** zur Codierung, d.h. es gibt Codeworte in der Menge M_2, die nicht vorkommen können.
- Dadurch ergeben sich zwei Möglichkeiten:
  1. Falls ein nicht zugewiesenes Codewort erkannt wird, liegt ein Fehler vor $\Rightarrow$ **Fehlererkennung** (unter bestimmten Bedingungen auch Fehlerkorrektur)
  2. Bei Codeumsetzern treten Don't–Care Terme (X–Terme) auf, die zur **Minimierung von Code–Umsetzern** herangezogen werden können.

BCD–Codes (Binary–Coded–Decimal)

- Codiert werden die Dezimalziffern $\mathbf{M_1} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{8}, \mathbf{9}\}$ mit je 4 bit (Tetraden), d.h. es werden nur 10 von $|M_2| = 16$ möglichen Codeworten verwendet $\rightarrow$ Redundanz
- Anwendung der BCD–Codierung:
  - Vermeiden von Umwandlungen für Anzeige, z.B. bei einem Digitalmultimeter
  - Vermeiden von Rundungsfehlern $\left(\mathit{Grund} : (0{,}1)_{10}\right.$ ist nicht exakt im Dual–Code darstellbar!$\left.\right) \rightarrow$ wichtig für kaufmännische Anwendungen
  - Viele Rechner haben spezielle Rechenbefehle für Zahlen im BCD-Code.

Bespiel für einen BCD-Code: Dezimalziffern als 4 bit Dualzahlen codiert

	Dual–Coded–Decimal				
Ziffer	$(d)_2$	8	4	2	1
Z_1	Z_2	d_3	d_2	d_1	d_0
0	0	0	0	0	0
1	1	0	0	0	1
2	2	0	0	1	0
3	3	0	0	1	1
4	4	0	1	0	0
5	5	0	1	0	1
6	6	0	1	1	0
7	7	0	1	1	1
8	8	1	0	0	0
9	9	1	0	0	1

← Stellenwert

Beispiel: $34_{10} = (0011\ 0100)_{BCD}$

- Neben diesem Code gibt es $\frac{|M_2|!}{(|M_2|-|M_1|)!} = \frac{16!}{(16-10)!} \approx 2,9 \cdot 10^{10}$ andere mögliche BCD–Codes.

- Die Menge der zur Verfügung stehenden Codeworte bei einem n–stelligen Binärcode ist: $M_2 = \{0,1\}^n$
- Zulässige Codeworte $c = (c_{n-1}, \ldots, c_1, c_0) \in M_2$ eines

  W–aus–N–Codes enthalten genau **W 1en**: $w[(c)] = \sum_{\nu=0}^{n-1} c_\nu \stackrel{!}{=} \mathrm{W}$

  ($w[(c)]$ ist Gewicht des Codewortes (c))
- Häufige Verwendung findet der **1–aus–N–Code** z.B. in Anzeigelampen, Tastaturen und in der Zustandscodierung.

Beispiel: 1–aus–4–Code

$(c)_2$	c_3	c_2	c_1	c_0
1	0	0	0	1
2	0	0	1	0
4	0	1	0	0
8	1	0	0	0

Fehlererkennung: Falls mehr als ein Bit 1 ist oder alle Bits 0 sind, liegt ein Fehlerfall vor!

Hinweis: Falls das Codewort mit lauter 0–Belegungen zugelassen wird, bezeichnet man den Code als erweiterten 1–aus–N–Code.

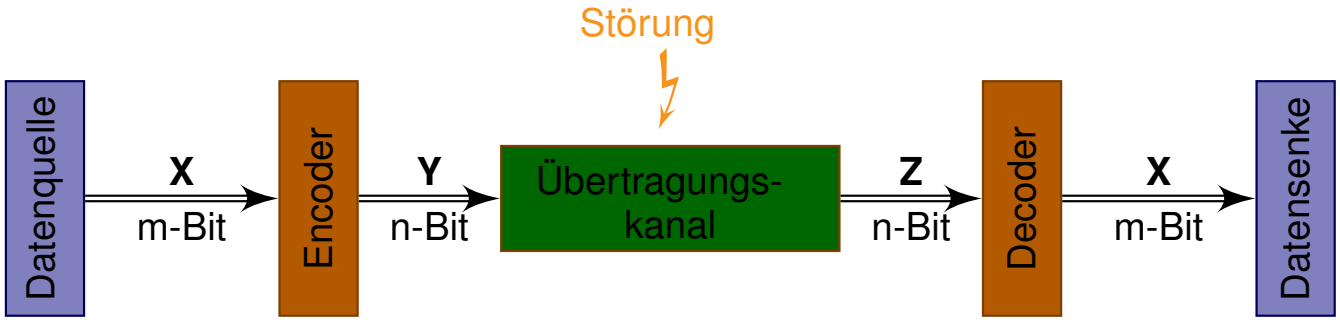
- **Menschliche Sprache** und Schrift ist durch eingefügte **Redundanz gegen Übermittlungsfehler** gesichert
- Beispiel:

D45 G3HT J4 W1RKL1CH!:
Ehct ksras! Gmäeß eneir Sutide eneir Uvinisterät,ist es nchit witihcg, in wlecehr Rneflogheie die Bstachuebn in eneim Wort snid, das ezniige was wcthiig ist, dsas der estre und der leztte Bstabchue an der ritihcegn Pstoiin snid. Der Rset knan ein ttoaelr Bsinöldn sein, tedztorm knan man ihn onhe Pemoblre lseen. Das ist so, weil wir nicht jeedn Bstachuebn enzelin leesn, snderon das Wort als gzeans enkreenn. Ehct ksras! Das ghet wicklirh! Und dfüar ghneen wir jrhlaeng in die Slhcue!

Und als absolute Steigerung dieses hier:
D1353 M1TT31LUNG Z31GT D1R, ZU W3LCH3N GRO554RT1G3N L315TUNG3N UN53R G3H1RN F43H1G 15T! 4M 4NF4NG W4R 35 51CH3R NOCH 5CHW3R, D45 ZU L353N, 483R M1TTL3W31L3 K4NN5T DU D45 W4HR5CH31NL1ICH 5CHON G4NZ GUT L353N, OHN3 D455 35 D1CH W1RKL1CH 4N5TR3NGT. D45 L315T3T D31N G3H1RN M1T 531N3R 3NORM3N L3RNF43HIGKEIT. 8331NDRUCK3ND, OD3R?

- Codierungen können durch gezieltes Hinzufügen von Redundanz so gestaltet werden, dass **Fehlererkennung** oder sogar (wie oben) **Fehlerkorrektur** beim Empfänger möglich wird.
- **Fehlerbehandlung**: Rückmeldung an Sender und **Wiederholung** der Übertragung (z.B. im Internet bei Dateidownload). Falls Wiederholung nicht möglich: Code mit **Fehlerkorrektur** (z.B. bei DVB-S/T oder Speicherung von Dateien auf Festplatten/DVD).
- Redundanz erreicht man durch Hinzufügen von Kontrollstellen:

- Darstellung eines Übertragungskanals:



Fehlerfreie Übertragung: $Z = Y$, Fehlerhafte Übertragung: $Z \neq Y$

- Einfachstes Beispiel: Einfügen eines **Paritätsbit** zur Fehlererkennung

- Bei einem Paritäts–Code wird ein Bit **p** (Paritätsbit) zu einem n–stelligen binären Codewort X hinzugefügt.
- Damit entsteht ein neues $(n+1)$–stelliges Codewort: $Y = (x_{n-1}, \ldots, x_1, x_0, \mathbf{p})$.
- Falls X ein Minimalcode ist, gibt es 2^n gültige Codeworte. Die Anzahl der möglichen Codeworte für Y mit dem Paritätsbit ist 2^{n+1}, d.h. es wird nur jedes zweite Codewort verwendet. $\Longrightarrow$ Redundanter Code
- Folge: 1 fehlerhaftes Bit kann bei der Übertragung des Codewortes erkannt werden.

Vorschrift zur Belegung des Paritätsbits:

Gerade Parität: (Even Parity) p so, dass das resultierende Codewort (y) ein **gerades Gewicht** $w\,[(y)]$ besitzt.

$$E = p = \leftrightarrow (x_{n-1}, \ldots, x_1, x_0)$$

Ungerade Parität: (Odd Parity) p so, dass das resultierende Codewort (y) ein **ungerades Gewicht** $w\,[(y)]$ besitzt.

$$O = p = \overline{\leftrightarrow (x_{n-1}, \ldots, x_1, x_0)}$$

Beispiel: Bildung eines Paritätsbit p bei einem Code mit m = 3

Paritätsbit (E = even = gerade):

X			p
x_2	x_1	x_0	E
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Paritätsbit (O = odd = ungerade):

X			p
x_2	x_1	x_0	O
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

- **Gewicht w(X)**: gibt die **Anzahl** der mit **1** belegten Stellen in einem Codewort des Codes X an
- **Stellendistanz / Distanz d** gibt die Anzahl der Stellen an, in denen sich zwei gültige Codewörter x_i, x_j voneinander unterscheiden

$$\boxed{d(x_i, x_j) = w[x_i \leftrightarrow x_j]}$$

- Beispiele:

1.

$$\begin{array}{rl} x_i = & \mathtt{1100} \\ x_j = & \mathtt{1000} \\ \hline x_i \leftrightarrow x_j = & \mathtt{0100} \end{array} \quad \Rightarrow d(x_i, x_j) = 1$$

2.

$$\begin{array}{rl} x_i = & \mathtt{1001\ 0011} \\ x_j = & \mathtt{0101\ 1101} \\ \hline x_i \leftrightarrow x_j = & \mathtt{1100\ 1110} \end{array} \quad \Rightarrow d(x_i, x_j) = 5$$

- **Hamming–Distanz** ist die geringste paarweise Stellendistanz **d** zwischen allen Codewörter eines Codes **C**, d.h. das **Minimum von d**.

$$h[C] = \min_{\text{alle } (x_i),(x_j) \text{ mit } i \neq j} \{ d(x_i, x_j) \}$$

- Beispiele: Bestimmen Sie die Hamming-Distanz für die folgenden Codes

1. $x_0 = 000$ $d(x_0, x_1) = 2$ $d(x_1, x_2) = 3$ $d(x_2, x_3) = 2$
   $x_1 = 011$ $d(x_0, x_2) = 1$ $d(x_1, X_3) = 1$
   $x_2 = 100$ $d(x_0, x_3) = 1$
   $x_3 = 010$ **h(C) = 1**

2. $x_0 = 000$ $d(x_0, x_1) = 2$ $d(x_1, x_2) = 2$ $d(x_2, x_3) = 2$
   $x_1 = 011$ $d(x_0, x_2) = 2$ $d(x_1, X_3) = 2$
   $x_2 = 101$ $d(x_0, x_3) = 2$
   $x_3 = 110$ **h(C) = 2**

Fehlererkennung

- Bei einem Minimalcode ($h = 1$), d.h. einem Code, bei dem alle Codeworte verwendet werden, führt jeder Bitfehler zu einem anderen zulässigen Codewort

  $\rightarrow$ Fehlererkennung (und Korrektur) nicht möglich.

- Eine Fehlererkennung ist möglich, wenn die Verfälschung eines oder mehrerer Bits auf ein unzulässiges Codewort führt

  $\rightarrow$ Redundanter Code mit $h > 1$ notwendig, d.h. es darf höchstens jedes zweite Codewort verwendet werden, z.B. Paritätscode

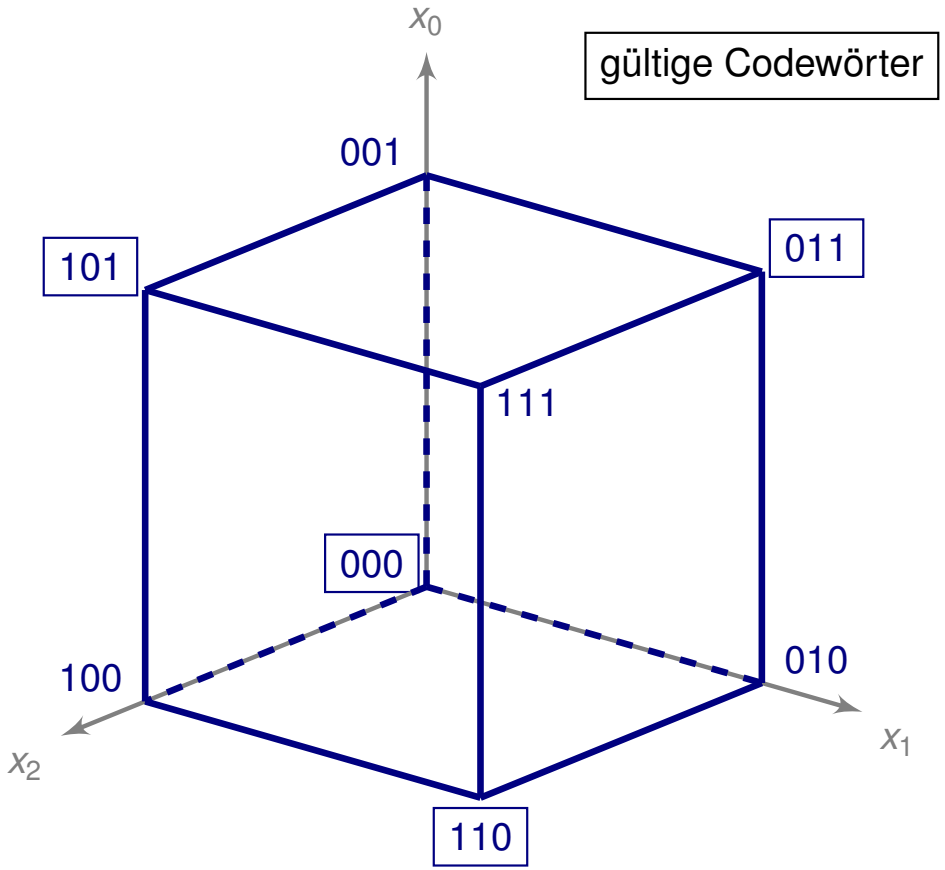
$\rightarrow$ **Anzahl der sicher erkennbaren Fehler** e^*: $\boxed{e^* = h - 1}$

Hamming Distanz h	1	2	3	4	5	...
Erkennbare Fehler e^*	0	1	2	3	4	...

$\rightarrow$ **Beispiel 1 s.h. nächste Seite**

Beispiel 1: 3-Bit Code mit $h = 2 \rightarrow$ **4 gültige** und **4 ungültige** Codewörter
(x_0 als gerades Paritätsbit für (x_2, x_1))

x_2	x_1	x_0	
0	0	0	gültige Codewörter
0	1	1	
1	0	1	
1	1	0	
0	0	1	ungültige Codewörter
0	1	0	
1	0	0	
1	1	1	



Bsp.: Übertragung 101 → Fehler im MSB → 001 → ungültig (erkannt)
Bsp.: Übertragung 101 → Fehler im LSB → 100 → ungültig (erkannt)
Bsp.: Übertragung 101 → Fehler im MSB und LSB → 000 → gültig (nicht erkannt)

Fehlerkorrektur:

- Zusätzlich zur Fehlererkennung ist eine **Fehlerkorrektur** möglich, wenn die Verfälschung eines oder mehrerer Bits zu einem unzulässigen Codewort führt, das immer noch „näher“ bei dem unverfälschten Codeworte liegt als bei allen anderen zulässigen Codeworten.

  $\rightarrow$ Redundanter Code mit $h \geq 3$ notwendig, d.h. zwischen 2 zulässigen müssen mindestens 2 unzulässige Codeworte liegen. Es darf also höchstens jedes dritte Codewort verwendet werden.

$\rightarrow$ **Anzahl der sicher korrigierbaren Fehler** e: $e = (\text{int})\ \frac{h-1}{2}$

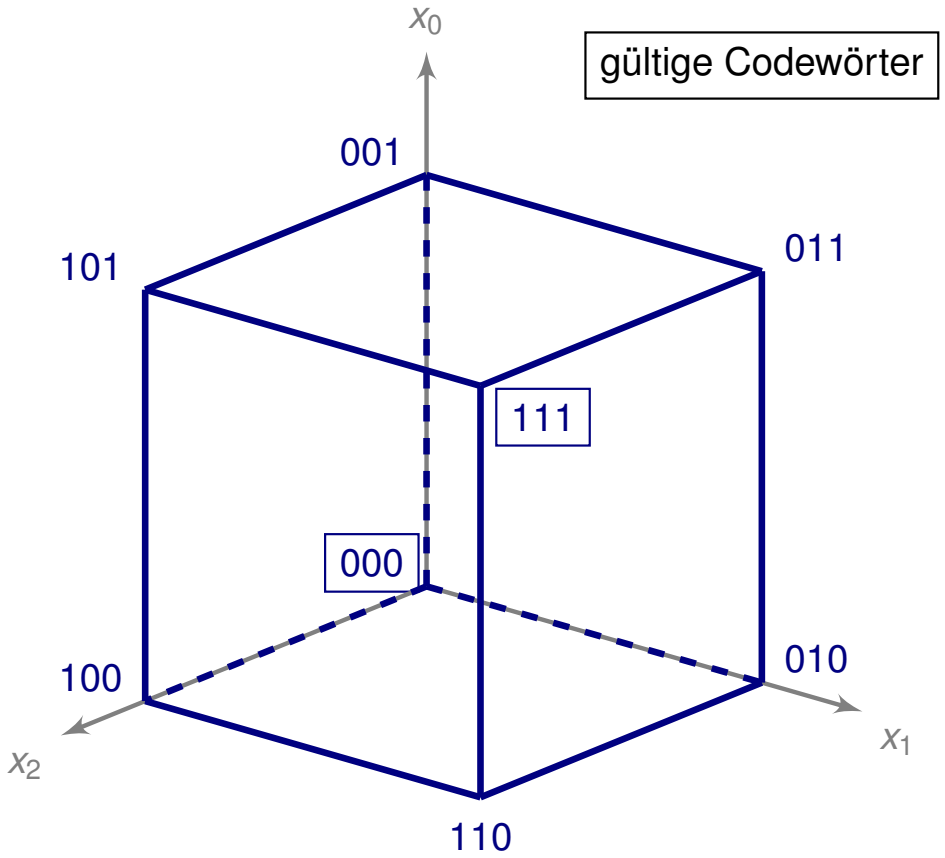
(int) . . . ganzzahlige Division, d.h. Quotient e ohne Nachkommastellen

Hamming Distanz h	1	2	3	4	5	6	. . .
Erkennbare Fehler e^*	0	1	2	3	4	5	. . .
Korrigierbare Fehler e	0	0	**1**	**1**	**2**	**2**	. . .

$\rightarrow$ **Beispiel 2 s.h. nächste Seite**

Beispiel 2: 3-Bit Code mit $h = 3 \rightarrow$ **2 gültige** und **6 ungültige** Codewörter

x_2	x_1	x_0	
0	0	0	gültige Code-wörter
1	1	1	
0	0	1	ungültige Codewörter
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	



Bsp.: Übertragung 111 → Fehler im MSB → 011 → ungültig (erkannt)
→ am nächsten bei 111 → korrigierbar
Bsp.: Übertragung 111 → Fehler im LSB → 110 → ungültig (erkannt)
→ am nächsten bei 111 → korrigierbar
Bsp.: Übertragung 111 → Fehler im MSB und LSB → 010 → ungültig (erkannt)
→ am nächsten bei 000 → nicht korrigierbar

Halbleiterspeicher mit Fehlererkennung und –korrektur

- Bei Servern mit sehr großem Speicher verwendet man ECC (Error Correction Code)-DRAM-Speicherbausteine. Dabei werden je 64 bit Speicherwort jeweils 8 Prüfbits gespeichert. Die Hamming-Distanz ist $h = 3$, d.h. es können 2 Fehler erkannt und 1 Fehler korrigiert werden.

Schnelle Berechnung von Prüfsummen für Festplatten und DVDs

- Für die schnelle Berechnung von Prüfsummen für Datenpakete, z.B. beim Senden von Daten über Ethernet-Netzwerke, gibt es verschiedene Algorithmen, die so festgelegt sind, dass möglichst viele Fehler erkannt werden können (**Cyclic Redundancy Check**).

Fehlererkennung bei IBAN-Kontonummern

- IBAN-Kontonummern bestehen in Deutschland aus 20 alphanumerischen Zeichen 0...9 und A...Z sowie 2 Prüfziffern.
- Der Algorithmus zur Berechnung der Prüfziffern aus der Kontonummer ist in ISO 7064 festgelegt (Wahl der Prüfziffer so, dass der Rest 1 ist, wenn die Zahl durch 97 dividiert wird).
- Der Empfänger berechnet die Prüfzeichen aus der Kontonummer ohne die Prüfzeichen und vergleicht sie mit den Prüfzeichen. Stimmen beide nicht überein, ist die IBAN ungültig.

- Bei der Übertragung großer Datenmengen, z.B. von Fernsehbildern, werden nicht einzelne Speicherworte, sondern große Datenblöcke durch sogenannte fehlerkorrigierende **Blockcodes** gesichert.
- Gegeben ist ein Nutzdatenblock $\underline{\boldsymbol{X}} = [x_{m-1}, x_{m-2}, \ldots, x_1, x_0]$ mit **m Bit**
- Der Nutzdatenblock wird durch $\underline{\boldsymbol{P}} = [p_{k-1}, \ldots, p_0]$ mit **k Bit** gesichert.
- Der gesicherte Datenblock $\underline{\boldsymbol{Y}} = [\underline{\boldsymbol{X}} \;\; \underline{\boldsymbol{P}}] = [x_{m-1}, \ldots, x_0, p_{k-1}, \ldots, p_0]$ mit **n = m + k Bit** wird übertragen.
- Da die Datenblöcke sehr groß sind, ist eine Codierung bzw. Dekodierung mit Codetabellen nicht möglich, sondern muss durch einen Algorithmus erfolgen, der einfach in Software bzw. Hardware implementiert werden kann, z.B. eine Matrixmultiplikation $\underline{\boldsymbol{Y}} = \underline{\boldsymbol{X}} \cdot \underline{\boldsymbol{G}}$ mit der Generatormatrix

$$\underline{\boldsymbol{G}}_{mxn} = \left[\underline{\boldsymbol{E}}_{mxm} \;\; \underline{\boldsymbol{A}}_{mxk}\right] = \begin{bmatrix} 1 & 0 & \ldots & 0 & a_{1,1} & \ldots & a_{1,k} \\ 0 & 1 & \ldots & 0 & a_{2,1} & \ldots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & 1 & a_{m,1} & \ldots & a_{m,k} \end{bmatrix}$$

- Der linke Teil der m x n Generatormatrix $\underline{\boldsymbol{G}}$ besteht aus einer m x m Einheitsmatrix $\underline{\boldsymbol{E}}$. Die rechte m x k Teilmatrix $\underline{\boldsymbol{A}}$ wird so gewählt, dass der Code eine möglichst große Hammingdistanz hat.

- Im Empfänger ist eine echte Dekodierung nicht notwendig, da das Nutzdatenwort $\underline{\boldsymbol{X}}$ in den ersten m bit des empfangenen Codeworts direkt enthalten ist: $\underline{\boldsymbol{Y}}^* = [\underline{\boldsymbol{X}}\ \ \underline{\boldsymbol{P}}] = [x_{m-1}, \ldots, x_0, p_{k-1}, \ldots, p_0]$
- Durch eine weitere Matrixmultiplikation $\underline{\boldsymbol{S}} = \underline{\boldsymbol{Y}}^* \cdot \underline{\boldsymbol{H}}^T$ wird überprüft, ob das empfangene Codewort $\underline{\boldsymbol{Y}}^*$ korrekt ist. Bei korrekter Übertragung wird das sogenannte **Fehlersyndrom** $\underline{\boldsymbol{S}} = 0$.
- Die sogenannte n x k **Prüfmatrix** $\underline{\boldsymbol{H}}^T$ ist

$$\underline{\boldsymbol{H}}^T_{nxk} = \begin{bmatrix} \underline{\boldsymbol{A}}_{mxk} \\ \underline{\boldsymbol{E}}_{kxk} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,k} \\ \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & \ldots & a_{m,k} \\ 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Bei den Matrixmultiplikationen wird bei allen Elementen stets modulo 2 gerechnet. In Hardware entspricht eine Multiplikation von zwei Bits einer AND-Verknüpfung, die modulo 2 Addition einer XOR-Verknüpfung.

Beispiel: Code mit m=2 Nutzdatenbits und k=3 Prüfbits $\rightarrow$ n=m+k=5 mit

$$\underline{\boldsymbol{G}} = \left[\underline{\boldsymbol{E}}_{2x2}\ \underline{\boldsymbol{A}}_{2x3}\right] = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}, \qquad \underline{\boldsymbol{H}}^T_{5x3} = \begin{bmatrix} \underline{\boldsymbol{A}}_{2x3} \\ \underline{\boldsymbol{E}}_{3x3} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Berechnung aller Codewörter $\underline{\boldsymbol{Y}}_{1x5} = \underline{\boldsymbol{X}}_{1x2} \cdot \underline{\boldsymbol{G}}_{2x5}$:

x_1	x_0	$y_4 = x_1$	$y_3 = x_0$	$y_2 = p_2$	$y_1 = p_1$	$y_0 = p_0$
0	0	0	0	0	0	0
0	1	0	1	1	0	1
1	0	1	0	0	1	1
1	1	1	1	**1**	**1**	**0**

Die übrigen $2^5 - 2^2 = 28$ Codewörter werden nicht verwendet. Dieser Code hat eine Hammingdistanz $h = 3$, d.h. es können $e^* = 2$ Fehler erkannt und $e = 1$ Fehler korrigiert werden.

Fortsetzung des Beispiels: Prüfmatrix $\underline{\underline{H}}^T_{5x3} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Berechnung des Fehlersyndroms $\underline{\underline{S}}_{1x3} = \underline{\underline{Y}}^*_{1x5} \cdot \underline{\underline{H}}^T_{5x3}$ bei Empfang korrekter und fehlerhafter Codewörter

y_4^*	y_3^*	y_2^*	y_1^*	y_0^*	S_2	S_1	S_0	Diagnose
0	0	0	0	0	0	0	0	korrekt
0	0	0	**1**	0	0	**1**	0	Fehler
0	1	1	0	1	0	0	0	korrekt
1	1	1	**0**	**1**	**0**	**1**	**1**	Fehler

...

Gegeben sei ein Code mit m=3 Nutzdatenbits und k=1 Prüfbit **Ü B U N G**

$\rightarrow$ n=m+k=4 mit $\underline{\boldsymbol{G}}_{3x4} = \left[\underline{\boldsymbol{E}}_{3x3}\ \underline{\boldsymbol{A}}_{3x1}\right] = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$

Bestimmen Sie sämtliche Codeworte $\underline{\boldsymbol{Y}}$ sowie die Prüfmatrix $\underline{\boldsymbol{H}}^T$ und überprüfen Sie die empfangenen Codeworte $(1001)_2$ und $(1\overline{000})_2$.

x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	1	0	0	1
1	0	1	1	0	1	0
1	1	0	1	1	0	0
1	1	1	1	1	1	1

$$\underline{\boldsymbol{H}}^T = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$\underline{\boldsymbol{S}} = (1001)_2 \cdot \underline{\boldsymbol{H}}^T = 0$ OK, $\underline{\boldsymbol{S}} = (1000)_2 \cdot \underline{\boldsymbol{H}}^T = 1$ Fehler

Leerseite

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

RECHNERARCHITEKTUR
- Rechnertypen
- Aufbau eines Rechners
- Klassische Architekturen
- Programmiermodell

SPEICHER: TYPEN, ORGANISATION UND HIERARCHIE
- Speicherorganisation
- Halbleiterspeicher
- Massenspeicher

STEUERWERKE
- Instruction Unit

RECHENWERKE
- Execution Unit
- Arithmetisch-Logische Einheit ALU
- Addierer und Subtrahierer
- Schiebeoperationen
- Multiplizierer und Dividierer
- Komplexere Funktionen

SPEICHERELEMENTE
- Flipflops
- Register
- Schaltwerke

RECHNERARCHITEKTUR

Rechnertypen
Alles Computer ...

Smartphone, PC

Mainframe

Embedded Systems

Server-Schränke

Rechnertypen
Personal Computer (PC) und Workstations, Virtuelle Maschinen[31]

- Ein **PC** ist ein individuell für den Benutzer vorgesehener Rechner für die **alltäglich anfallenden Aufgaben**
- Die mobilen Varianten sind **Notebooks** oder tastaturlose **Tablets**.
- **Leistungsfähige** PCs nennt man **Workstations**.
- Das installierte **Betriebssystem** (Windows, Mac OS, Linux) verwaltet die vorhandenen Hardware-Komponenten und stellt die Brücke zwischen Hardware und Benutzeroberfläche dar
- Ein heute durchschnittlicher PC hat eine mit ca. 3-4 GHz getaktete 64Bit-CPU, 8-16 GB Arbeitsspeicher, 2 TB Festplatte und kostet inkl. Tastatur, Maus und Flachbildschirm deutlich unter 1000 Euro.

- Um unterschiedliche Betriebssysteme auf demselben Rechner gleichzeitig auszuführen, kann man auf einer ausreichend leistungsfähigen Hardware (**Host**) mehrere Betriebssysteme jeweils in einer eigenen **virtuellen Maschine** (**Guest**), laufen lassen, z.B. VmWare bzw. VirtualPC.
- Wie bei einem physikalischen Rechner lässt sich die virtuelle Maschine **booten** und **herunterfahren** sowie **Anwendungen installieren**.

[31] Quelle R. Hellmann: [21]

- Ein **Server** stellt Daten für mehrere Benutzer zur Verfügung z.B. **Fileserver** oder **Datenbankserver** (Datenaustausch), **Webserver** (Bereitstellung Internetseiten), **Mailserver** (Postfächerverwaltung)
- Prinzipiell ist es möglich z.B. einen PC als Fileserver einzusetzen. Dadurch ist der Übergang zwischen Server und PC fließend.
- Oft wird Server-Hardware in **Racks** (Serverschränke) verbaut. Aufgrund der beträchtlichen **Wärmeentwicklung** müssen die Server gekühlt werden und stehen oft in separaten **klimatisierten Serverräumen**.

- Rechner oder Anwendungen, die die von einem Server bereitgestellten Daten nutzen, nennt man **Clients**.
- **Fat Client**: ist ein vollwertiger PC, der die vom Server **abgerufenen Daten selbst verarbeitet**. Die Anwendung ist auf ihm installiert.
  Bsp.: Microsoft Office (Word, Excel) auf einem PC.
- **Thin Client**: dient nur zur **Anzeige der Verarbeitungsergebnisse**. Verarbeitung und Speicherung finden auf dem Server statt.
  Bsp.: Google Docs auf Google Servern mit Zugriff über Web–Browser.

- Die Datenübertragung zwischen Server und Client erfolgt über ein **Rechnernetz**, z.B. ein Local Area Network LAN (Ethernet, WIFI/WLAN) oder ein Wide Area Network WAN („Internet“, DSL, UMTS/LTE) nach festen Regeln (**Protokolle** TCP/IP, HTTP).

- **Mainframes** bestehend aus einem **Zentralrechner** mit vielen **Terminals** haben bis in die 1980er Jahre die Datenverarbeitung bestimmt.
- Bis heute findet man Mainframes noch in **Banken und Versicherungen** oder als Abrechnungssystem bei der Telekom. Führende Hersteller sind IBM, Hitachi und Fujitsu. Der Preis liegt typischerweise bei **mehreren Millionen** Euro.
- Mainframes haben Vorteile bei der **Wartung**, der **Datensicherheit** und der Verarbeitung **großer Datenmengen**.
- Es können **mehrere Betriebssysteme** gleichzeitig betrieben werden und dem Benutzer als eigene **virtuelle Rechner** vorgegaukelt werden.

- Alternativ werden große Gruppen von Workstations als **Rechnercluster** eingesetzt, die über schnelle Netzwerke miteinander verbunden werden.
- Wenn das Rechnercluster nicht am Standort des Nutzers steht, sondern über das Internet von einem Fremdanbieter, z.B. Amazon Web Service, Microsoft Azure, betrieben wird, der Rechen- und Speicherkapazität als Dienstleistung verkauft, spricht man von einer **Cloud**.
- Für den Benutzer wirken Cluster und Cloud im Idealfall wie ein einziger großer Rechner.

[32] Quelle Gumm, Sommer: [18]

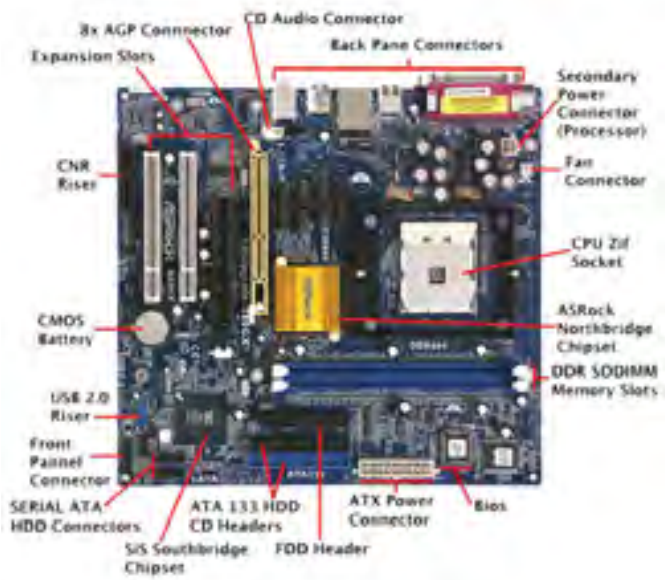
- **Supercomputer** sind Rechner mit **extrem hohen Rechenleistungen**. Sie werden z.B. für **FEM (Finite Elemente Methode)-Simulationen** wie die **Windkanalsimulation** im Automobilbereich oder die **Wettervorhersage** eingesetzt.
- Die hohe Rechenleistung erreicht man durch die Verwendung vieler **tausender Prozessoren** und einem **Terabyte großen Arbeitsspeicher**.
- Aktuell wird an Supercomputern im Bereich von **Exa-Flop** (10^{18} Fließkomma- Operationen pro Sekunde) gebaut. Zum Vergleich: Aktuelle PCs haben unter 100 Giga-Flops (10^{11}).
  Hinweis: 1 FLOP = 1 Floating Point Operation, z.B. Multiplikation von zwei IEEE754-Gleitkommazahlen.
- Die benötigte elektrische Leistung reicht bis in den **Megawatt-Bereich**.

- **Embedded Systems** (eingebettete Systeme) sind speziell an ein Einsatzgebiet angepasst Rechnersysteme, z.B. Steuerung von Fahrzeugen, Waschmaschinen, Fertigungsmaschinen aber auch SetTop-Boxen usw.
- Randbedingungen: Geringer Ressourcenverbrauch, d.h. **minimale Kosten**, **geringer Platz**-, **Energie-** und **Speicherverbrauch**, oft ohne Festplatte, Tastatur und Display.
- Wesentlich **höhere Zuverlässigkeits- und Zeitanforderungen** als bei normaler Computersoftware.
- Die Software wird **Firmware** genannt (ROM, Flash-Speicher).
- Man verwendet **spezielle Real-Time-Betriebssysteme** (AUTOSAR/OSEK OS, Embedded Linux, Windows Embedded, etc.)
- Programmiert wird in **C** oder **C++**, selten **Java**. Bei zeitkritischen Funktionen **Assembler**, z. T. mit dem Einsatz von Cross-Compilern
- Softwaretests finden entweder nicht auf der Zielhardware (Model in the Loop (MIL) / Software in the Loop (SIL)) oder auf der Zielhardware in Verbindung mit simulierten Peripheriekomponenten wie Sensoren/Aktoren statt (Hardware in the Loop (HIL)).
- Entwicklung der Software häufig **modellbasiert**.

RECHNERARCHITEKTUR

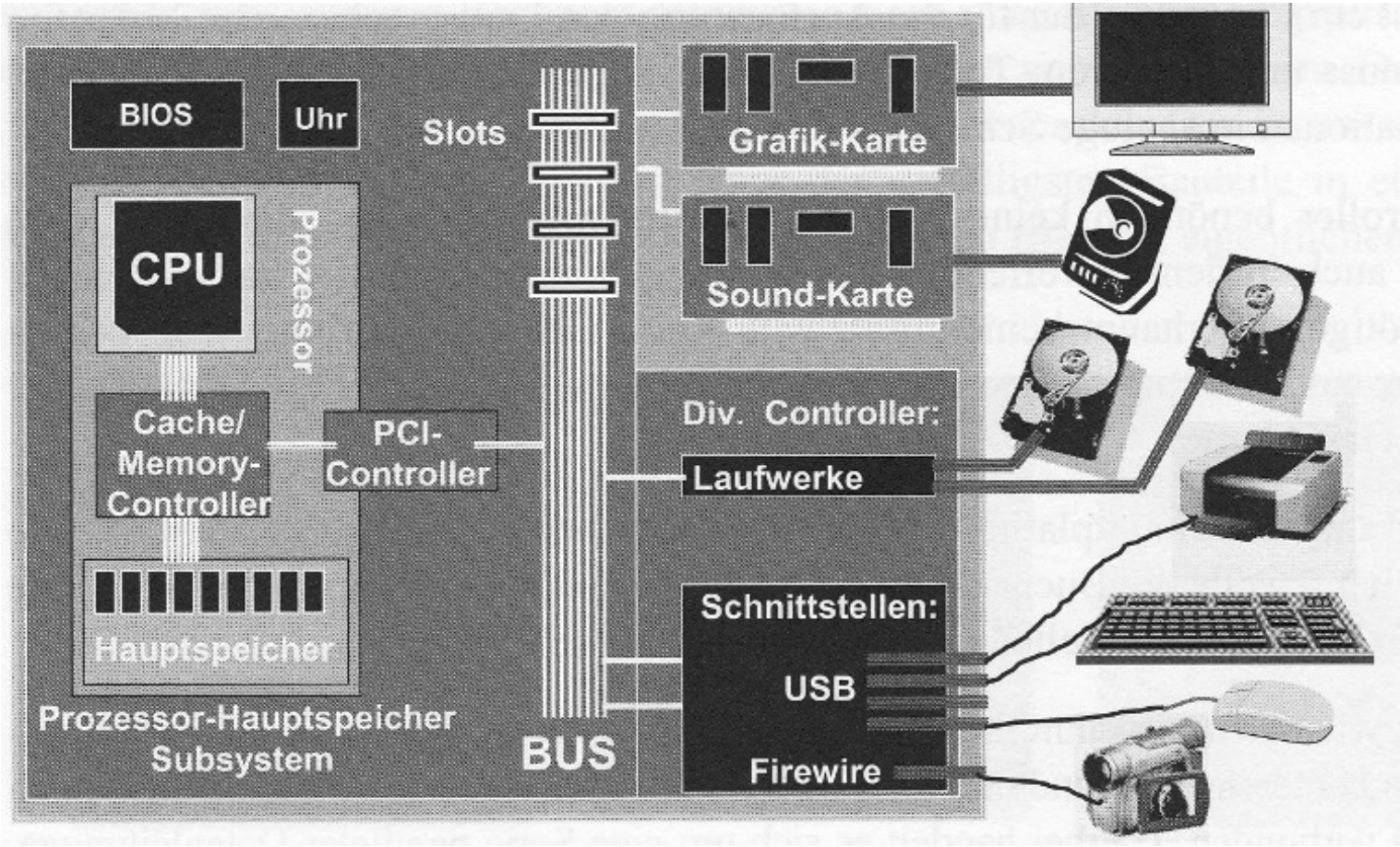
Typische Workstation (Quelle: Internet)

PC Innenansicht



PC Grundplatine (Motherboard)

Schematischer Aufbau der Grundplatine



Quelle Gumm, Sommer: [18]

Computer = Maschine zur Verarbeitung von **Daten** nach einem **Programm**

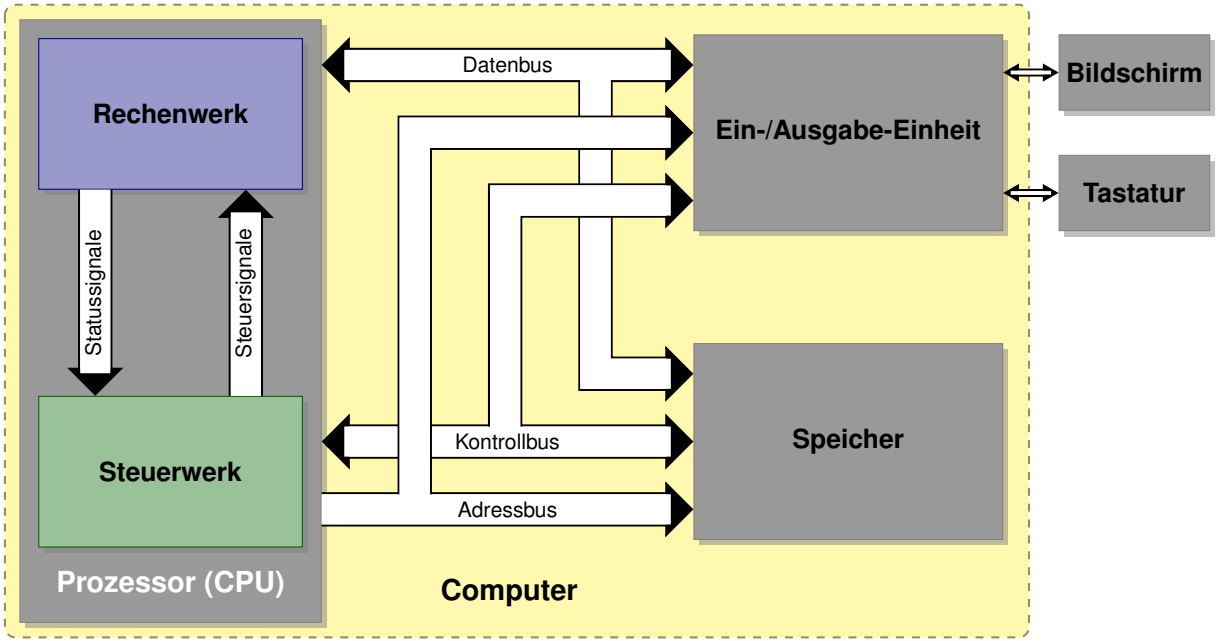
Beispiel eines einfachen Programms (C++):

```
int a, b, c;              // *** Daten (Variablen) ***

void main(void) {         // *** Programmcode ***
  cin >> a;               // Eingabe von der Tastatur
  b = 4;                  // Folge von Befehlen (Operationen)
  c = a + b;
  cout << c << endl;      // Ausgabe auf dem Bildschirm
}
```

Abbildung der Programmelemente in der Computer-Architektur:

- **Compiler übersetzt** Programm in binär codierte Maschinenbefehle (**Code**) und reserviert Platz für **Daten**
- Code und Daten werden im **Speicher** (Memory) abgelegt
- **Steuerwerk** der CPU (Central Processing Unit) holt die Befehle und die Daten aus dem Speicher und führt die Operationen im **Rechenwerk** aus
- Über **Ein- und Ausgaben** kommuniziert der Computer mit dem Benutzer



- **Steuerwerk (Instruction Unit)**: Holt **Programmbefehle** aus dem Speicher, dekodiert sie und erzeugt die **Steuersignale** für das Rechenwerk.
- **Rechenwerk (Execution Unit)**: Führt Operationen mit den **Programmdaten** aus und gibt dem Steuerwerk über **Statussignale (Flags)** Rückmeldungen.
- Programmbefehle und Programmdaten stehen im **Speicher (Memory)** und werden über den **Datenbus** transportiert. Die Auswahl und Steuerung erfolgt über den **Adress- und Kontrollbus** (= **Bussysteme**).
- Der Datenaustausch mit der Aussenwelt (Benutzer, andere Computer) erfolgt über die **Ein-Ausgabe-Einheit (Input/Output Unit I/O)**.
- **Multi Core CPU**: Heute werden oft mehrere Prozessoren (**Central Processing Unit CPU)** und weitere Komponenten auf einem **Mikroprozessor-Chip** integriert.

[33] Quelle R. Hellmann: [21]

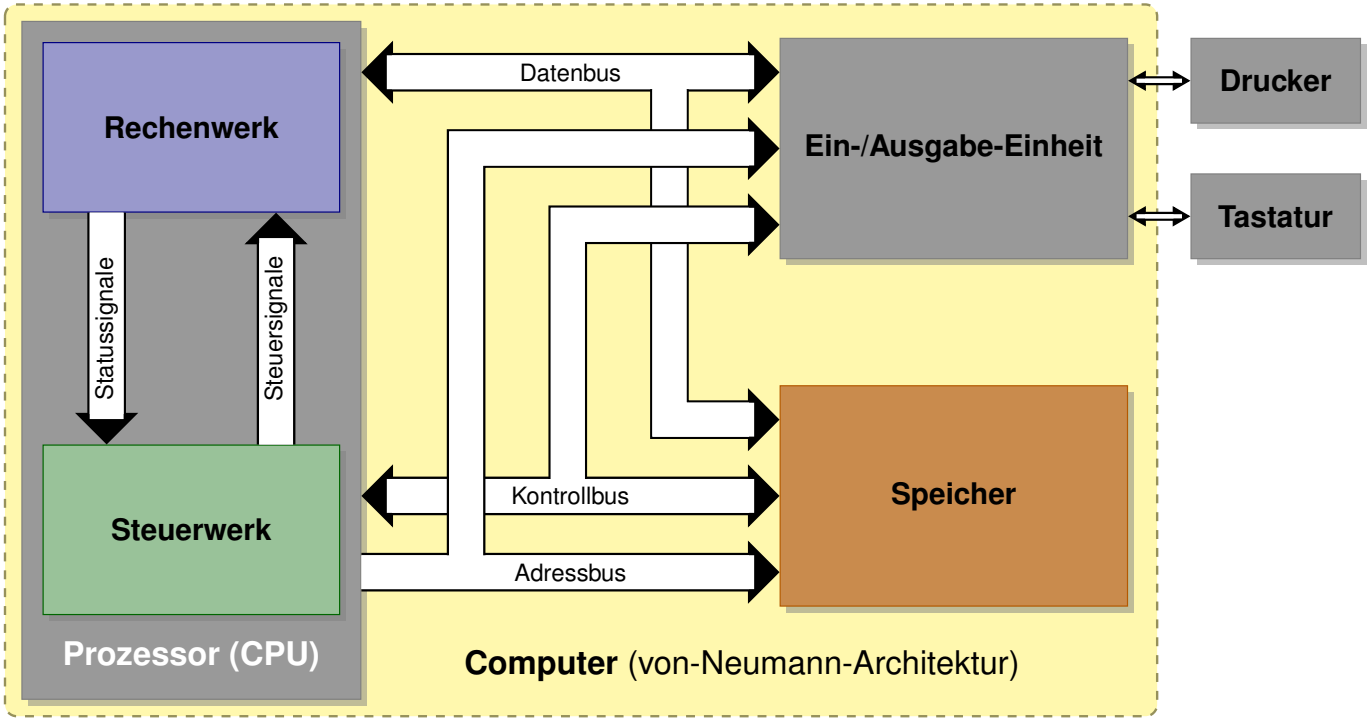
RECHNERARCHITEKTUR

SPEICHER: TYPEN, ORGANISATION UND HIERARCHIE

STEUERWERKE

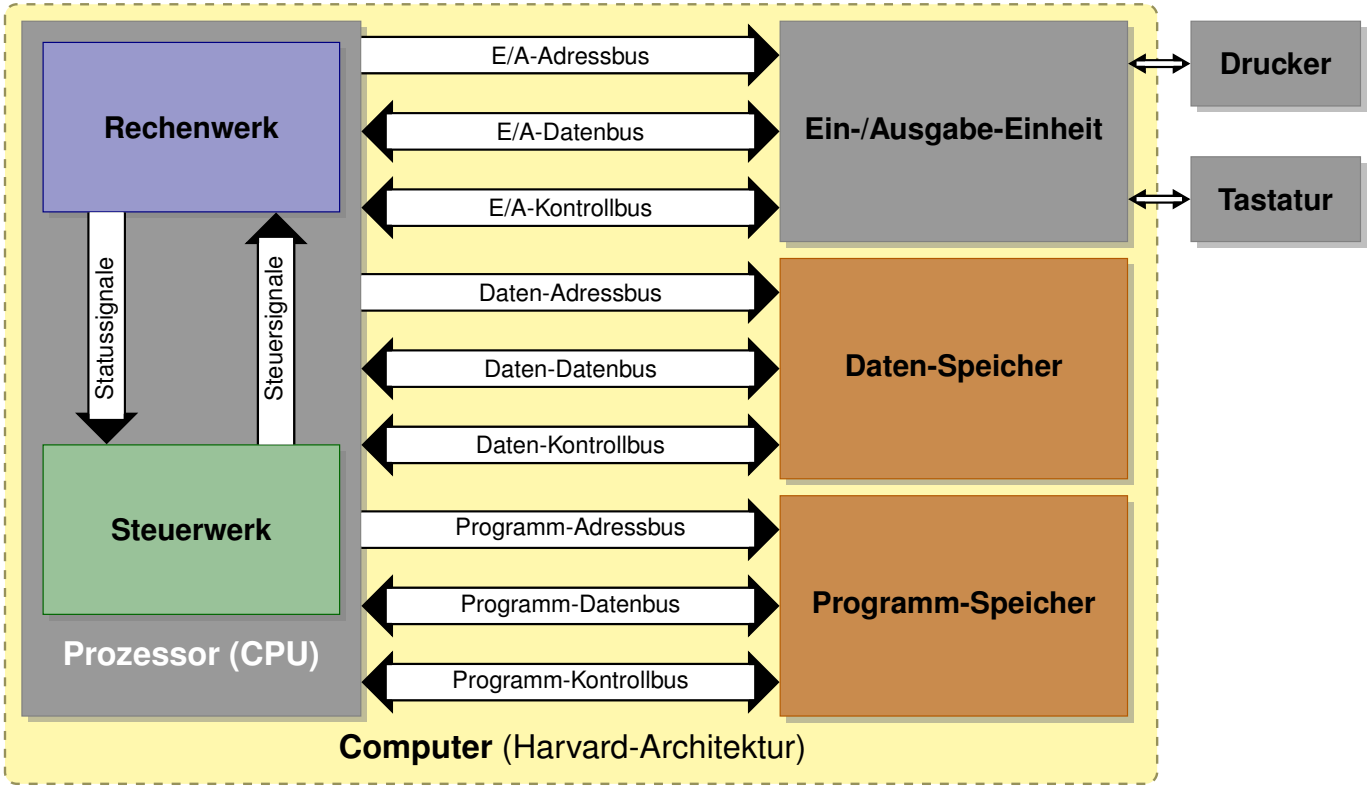
RECHENWERKE

SPEICHERELEMENTE



- Bei der **von-Neumann-Architektur** besitzt das Computersystem nur **einen gemeinsamen Speicher** und einen einzigen Adress/Daten/Kontrollbus für **Programmcode** und **Programmdaten**
- Vorteil: **Flexible Speicherausnutzung** bei Anwendungen mit unterschiedlich großer Menge an Programmcode und Daten
- Nachteil: Das **Bussystem wirkt als Flaschenhals**, d.h. es verlangsamt die Programmausführung, da der Speicherzugriff nur hintereinander erfolgen kann.

[34] Quelle R. Hellmann: [21]



- Bei der **Harvard-Architektur** besitzt das Computersystem **getrennte Speicher** sowie mehrere Adress/Daten/Kontrollbusse für **Programmcode** und **Programmdaten**
- Vorteil: Durch getrennte Busse **schnellere Programmausführung**
- Nachteil: **Unflexible Speicherausnutzung**
  Reale Harvard-Systeme implementieren daher oft eine zusätzlichen Busverbindung zwischen Daten-Speicher und Programm-Speicher.

RECHNERARCHITEKTUR

SPEICHER: TYPEN, ORGANISATION UND HIERARCHIE

STEUERWERKE

RECHENWERKE

SPEICHERELEMENTE

Programmiermodell: Der für den Softwareentwickler sichtbare Teil eines Rechners. Es besteht aus dem **Befehlssatz** und dem **Datenmodell** (**Registermodell, Adressierungsarten** für Daten im Speicher). Eine wichtige Kenngröße ist die **Breite der Daten- bzw. Adressworte**, z.B. 32 bit oder 64 bit.

CISC und RISC

- Die ersten CPUs waren **Complex Instruction Set Computer CISC**:
  - Viele Befehle (großer Befehlssatz) und komplexe Adressierungsmöglichkeiten, um das Programmieren zu vereinfachen.
  - Relativ kleine Registersätze und langsame Speicher aus Kostengründen.
- Da Programmierer und Compiler in der Praxis aber nur einen kleinen Teil der komplexen Befehle nutzen, sind neuere CPUs meist **Reduced Instruction Set Computer RISC**:
  - Einfacher (kleiner) Befehlssatz und wenige Adressierungsmöglichkeiten.
  - Relativ große Registersätze.
  - RISC ermöglicht einfachere, aber sehr viel schnellere Hardware. Das Problem der immer noch relativ langsamen großen Speicher löst man durch schnelle Zwischenspeicher (Caches).

Prozessorfamilien und Mikroarchitektur

- Damit ältere Software mit neuen CPUs kompatibel bleibt, wird das **Programmiermodell** nur selten geändert. Baureihen von CPUs mit aufwärtskompatiblem Programmiermodell bezeichnet man als **Prozessorfamilie**.
  Beispiele: Intel x86, ARM, Sun Sparc, MIPS, IBM Power, Arduino/Atmel AVR, ...
- Unter **Mikroarchitektur** versteht man die für den Softwareentwickler unsichtbaren Schaltungsinternas einer CPU. Die Fortschritte der Halbleitertechnik erlauben immer aufwendigere Mikroarchitekturen. **Hier findet der Fortschritt statt!**

Befehlssatz: Arten von Maschinenbefehlen

- **Datentransportbefehle**: Kopieren von Daten zwischen CPU $\rightarrow$ Speicher (Schreiben), Speicher $\rightarrow$ CPU (Lesen) oder zwischen Registern innerhalb der CPU
- **Arithmetisch-logische Operationen**, z.B. Addieren, Subtrahieren, Und, Oder usw. In der Regel können mit einem Befehl nur zwei Operanden verarbeitet werden, das Ergebnis überschreibt einen der beiden Operanden (**2-Adress-CPU**).
- **Programmablaufsteuerung**: Befehle zur Auswertung von Bedingungen und Programmverzweigungen und -schleifen (if-else, switch-case, for, while) sowie Unterprogrammaufrufe
- **Verwaltungsbefehle**

Befehlssatz: Aufbau von Maschinenbefehlen

- **Opcode**: Typ des Befehls binär codiert
- **Adressfeld**: Angabe zu den beiden Operanden (Registernummern, Speicheradressen)

Programmausführung: Prinzipiell arbeitet eine CPU in einer Endlosschleife

1. Befehl holen (1+2: **Instruction Phase**)
2. Befehl dekodieren
3. Operanden holen (3-5: **Execution Phase**)
4. Befehl ausführen
5. Ergebnis abspeichern

Zur Steigerung der Rechenleistung gibt es mehrere Ansätze:

- **Höhere Taktfrequenz**:
  Wird begrenzt durch Schaltgeschwindigkeit der Halbleiter, endliche Signallaufzeit (Lichtgeschwindigkeit!), Zugriffszeit der Speicherbauteile und Verlustleistung (Takt $\uparrow \rightarrow$ Energieverbrauch $\uparrow$)
- **Pipelining-Konzept** (Fließbandverarbeitung):
  Zeitverschachtelte Ausführung mehrerer Befehle, z.B. während das Rechenwerk den aktuellen Befehl ausführt, holt das Steuerwerk bereits den nächsten Befehl.
- **Superskalares Konzept** (Parallelverarbeitung): Das Steuerwerk wird so aufgebaut, dass in mehreren Rechenwerken mehrere Befehle gleichzeitig ausgeführt werden können.
- **Multi-Core-Konzept**:
  Möglichkeiten zur zeitverschachtelten oder parallelen Ausführung von Befehlen in einem Programm sind begrenzt, weil zwischen den Befehlen häufig **Datenabhängigkeiten** bestehen. Wenn z.B. der nachfolgende Befehl das Ergebnis des vorigen Befehls benötigt, kann dieser Befehl nicht gleichzeitig ausgeführt werden.
  Daher enthalten moderen Mikroprozessorchips heute mehrere CPUs (Multi-Core), die mehrere Programme oder Programmteile (**Multi-Tasking, Multi-Threading, Multi-Processing**) ausführen.
  Zwischen unterschiedlichen Programmen gibt es in der Regel keine Datenabhängigkeiten.

Aus Software-Sicht ist der Speicher eine große Tabelle. Der Inhalt einer Tabellenzeile wird als Speicherwort oder **Datum** bezeichnet. Die einzelnen Tabellenzeilen werden durchnummeriert (**Adressen**), damit die CPU die Daten findet. Typischerweise bekommt jedes Datenbyte eine eigene Adresse.

Beispiel:	**Adresse** (m bit)	**Daten** (n bit)
	0 0 ... 0 0	34
	0 0 ... 0 1	72
	...	90
variable1 →	0 1 ... 1 1	**22**
	...	41
variable2 →	1 0 ... 0 1	**22**
	...	31
	1 1 ... 1 1	14

Wenn wir in einem Programm **Variable** verwenden, reserviert der Compiler beim Übersetzen des Programms für jede Variable Speicherplatz und legt die zugehörige Adresse fest. Wenn im Programm eine Variable verwendet wird, wird die Variable von der CPU aus dem Speicher gelesen (**Read**) oder in den Speicher geschrieben (**Write**).

```
char variable1, variable2; //Definition der 8 bit Variablen
variable1 = 22;            //Schreiben variable1
variable2 = variable1; //Lesen variable1 -> Schreiben variable2
```

Der **Datentyp** legt die Größe der Variable in Byte fest, z.B. `char` = 8 bit, `short` = 16 bit, `int` = 32 bit usw. Die CPU muss zum Lesen bzw. Schreiben der Variable die Adresse des ersten Bytes und die Anzahl der Bytes kennen.

Speicher ist in der Regel **Byte-adressierbar**, d.h. jedes Byte bekommt eine eigene Adresse. Beim Speichern von Mehr-Byte-Werten, z.B. einem 32 bit Datenwort $(41\ 3F\ 2A\ 59)_{16}$, gibt es daher verschiedene Möglichkeiten:

- **Big Endian**: Abspeichern „von links nach rechts“, d.h. das höchstwertige Byte wird zuerst, also an der niedrigsten Adresse abgespeichert, das niederwertigste Byte an der höchsten Adresse.

	Adresse	Daten	
	...	...	
niedrigere Adresse →	2130_{16}	$\mathbf{41_{16}}$	← **Big Endian**
	2131_{16}	$\mathbf{3F_{16}}$	**typ. bei Freescale CPUs**
	2132_{16}	$\mathbf{2A_{16}}$	
höhere Adresse →	2133_{16}	$\mathbf{59_{16}}$	
	...	...	

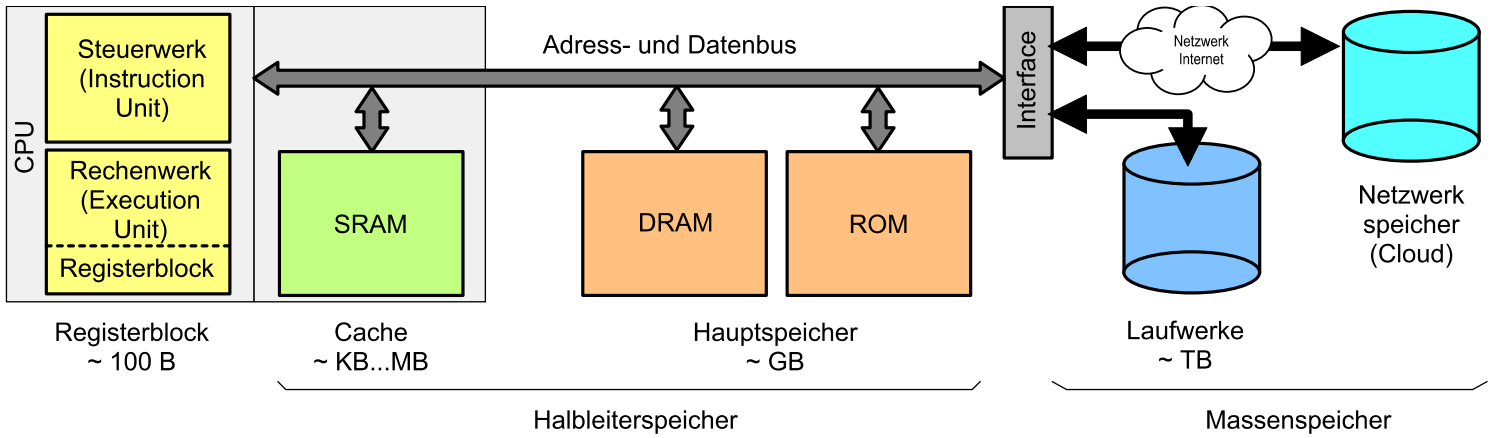
- **Little Endian**: Abspeichern „von rechts nach links“, d.h. das niederwertigste Byte wird zuerst, also an der niedrigsten Adresse abgespeichert, das höchstwertigste Byte an der höchsten Adresse.

	Adresse	Daten	
	...	...	
niedrigere Adresse →	2130_{16}	$\mathbf{59_{16}}$	← **Little Endian**
	2131_{16}	$\mathbf{2A_{16}}$	**typ. bei Intel CPUs**
	2132_{16}	$\mathbf{3F_{16}}$	
höhere Adresse →	2133_{16}	$\mathbf{41_{16}}$	
	...	...	

Kompromiss notwendig zwischen Leistungsfähigkeit (Zugriffszeit beim Lesen/Schreiben) und Aufwand (Speichergröße x Kosten je Bit)

⇒ **Hierarchische Speicherorganisation** mit verschiedenen Technologien

schnell	←	←	←	langsam
Register	SRAM	DRAM / ROM	Festplatten	DVD
teuer	→	→	→	billig



Aufgabe der Speicherverwaltung des Betriebssystems und der Hardware: Transparentes Kopieren der Daten zwischen den Speichern so, dass die jeweils benötigten Daten im schnellstmöglichen Speicher liegen.

DRAM

Flash-ROM

Speichertypen

- **ROM** Read-Only-Memory: Speicher, der nur gelesen werden kann (für Programme und konstante Daten). Der Speicherinhalt bleibt beim Abschalten der Spannungsversorgung erhalten.

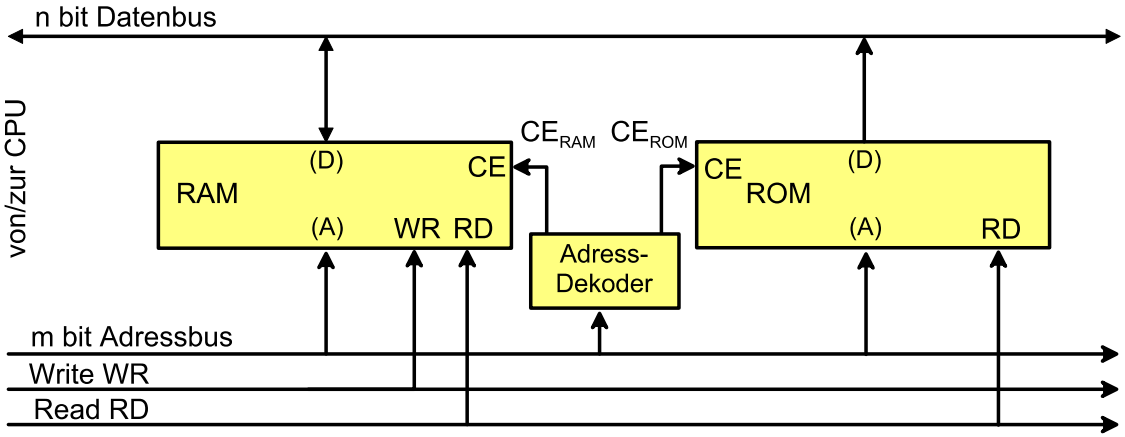
  Sogenannte **Flash-ROM**-Speicher können blockweise gelöscht und neu programmiert werden. Der Programmiervorgang (>1ms/Byte) ist extrem langsam im Vergleich zum Lesen von Daten (<2...50ns/Byte) und darf nur einige 100 000 mal wiederholt werden, weil die Halbleiterstruktur dabei stark gestresst wird.

  Geringer Schaltungsaufwand 1 Transistor je Bit.

- **RAM** Random-Access-Memory: Speicher, der gelesen und geschrieben werden kann (für variable Daten und Programme, die von einem Massenspeicher ins RAM kopiert wurden). Der Speicherinhalt geht beim Abschalten der Spannungsversorgung verloren.

  Übliche Typen sind **SRAM Statisches RAM** (<2...10ns/Byte), hoher Schaltungsaufwand 6 Transistoren je Bit) und **DRAM Dynamisches RAM** (30...60ns/Byte, geringer Schaltungsaufwand 1 Transistor je Bit).

- Um Leitungen und IC-Anschlusspins einzusparen, sind mehrere Speicherbausteine an einen **gemeinsamen Adressbus, Datenbus und Kontrollbus** (Schreib-Lese-Steuersignale) angeschlossen.
- **Adressen**, d.h. die laufenden Nummern der Speicherzellen, dienen zur Auswahl der Speicherzelle. Bei m bit Adressen kann die CPU 2^m Speicherzellen auswählen. Wenn mit jeder Adresse ein n bit Datenwort ausgewählt werden kann, spricht man von einem **Adressraum** von $N = 2^m \cdot n$, z.B. $n = 8$, $m = 32\ bit \Rightarrow N = 4\ GiByte$
- **Wahlfreier Zugriff**, d.h. Speicherworte bei beliebigen Adressen können in beliebiger Reihenfolge gelesen/geschrieben werden.
- Jedem Speicherbaustein wird ein zusammenhängender **Adressbereich** zugeordnet. Die Auswahl der einzelnen Bausteine erfolgt durch einen **Adressdekoder**.
- Die Übersicht über alle Adressbereiche im gesamten Adressraum wird als Adresstabelle (engl. **Memory Map**) bezeichnet.



D...Datenein/ausgänge, A...Adresseingänge, RD...Read, WR...Write, CE...Chip-Enable (Bausteinauswahl)

Beispiel:

- Ein Rechner verwende $m = 32$ bit Adressen. Der Rechner verwendet Byte-Adressierung.
- Der ROM-Bereich soll 256 MiByte groß sein und am Beginn des Adressbereichs liegen.
- Am Ende des Adressbereichs sollen 512 MiByte für den Bildspeicher der Grafikkarte reserviert werden.
- Der übrige Adressraum ist für RAM-Speicher vorgesehen.

Memory Map

Anfangsadresse	Endadresse	Speichertyp/größe
$0000\ 0000_{16}$	$\mathbf{0FFF\ FFFF_{16}}$	**ROM 256 MiByte**
$\mathbf{1000\ 0000_{16}}$	$\mathbf{DFFF\ FFFF_{16}}$	**RAM (4096-256+512) MiByte**
$\mathbf{E000\ 0000_{16}}$	$FFFF\ FFFF_{16}$	**Grafik-RAM 512 MiByte**

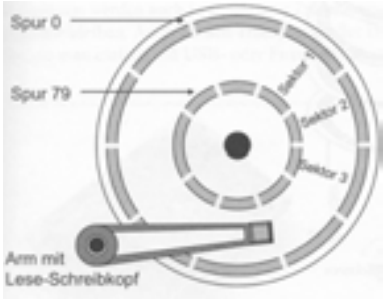
Nebenrechnung:

Adressraum 2^{32} Byte = $4 \cdot 2^{30}$ Byte = 4 GiByte = $2^{12} \cdot 2^{20}$ Byte

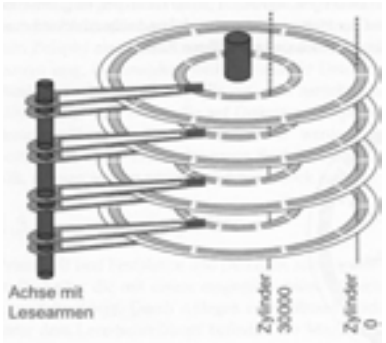
256 MiByte $= 2^8 \cdot 2^{20}$ Byte $= 2^{28}$ Byte = $1000\ 0000_{16} (= \frac{1}{16}$ GiByte)

Massenspeicher
Festplatten, DVD-Laufwerke, Memory Sticks

Magnetische Festplatte
(Harddisk HD)



Kapazität > 8 TB



+ kostengünstig
- langsam

Optische Speicher:
(CD, DVD, BlueRay)
+ sehr billig, - sehr langsam

Zukünftig: Halbleiterfestplatte
Solid State Disk SSD
(noch) teuer

USB-Stick

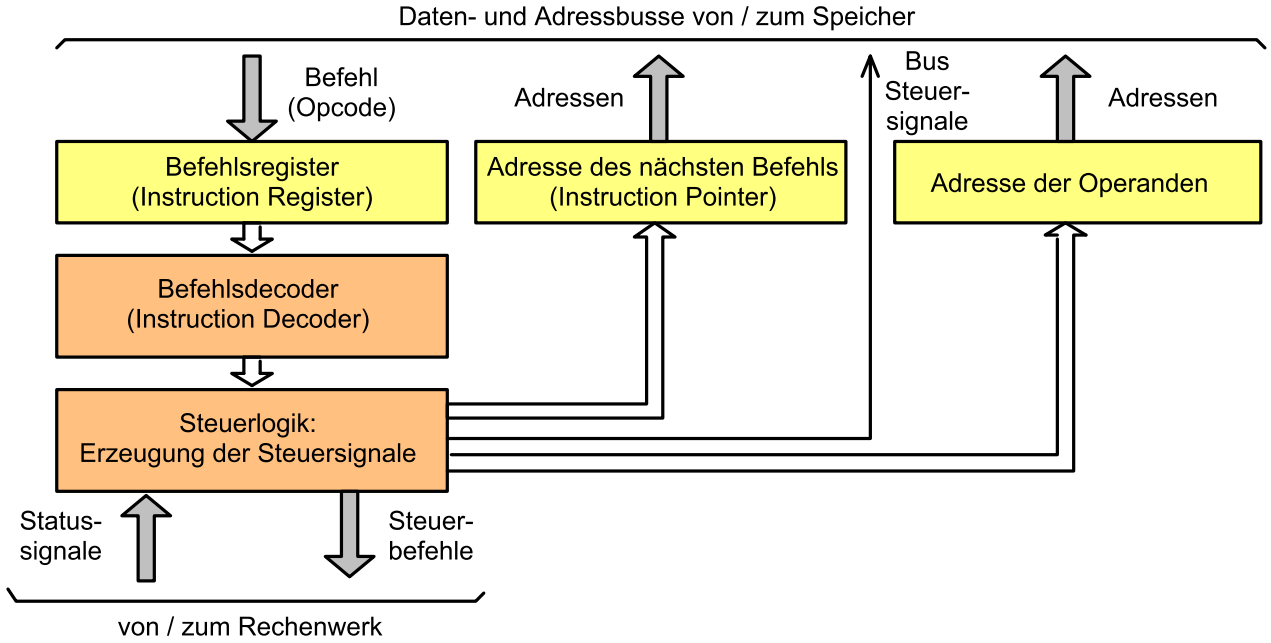
Massenspeicher

- Daten organisiert als Dateien durch das Betriebssystem
- **Lesen und Schreiben** erfolgt **blockweise**, typ. Blockgröße 512 Byte, Auswahl über Dateinamen (auf Anwenderebene) und Blocktabellen (auf Betriebssystemebene). Nur **sequentieller**, kein wahlfreier **Zugriff**.
- Nicht direkt an das parallele Adress-/Datenbussystem der CPU angeschlossen, sondern über schnelle serielle Verbindungen (SCSI, SATA, PCI-Express)

Typische Daten von Halbleiter- und Massenspeichern

Medium	Kapazität Obergrenze	Mittlere Zugriffszeit	Datentransfer-rate in MB/s
Register (Flipflops)	$\leq$ 1 KB	0,1 ns	70000
Cache (SRAM)	32 MB	2 ns	60000
Hauptspeicher (DRAM)	64 GB	10 ns	20000
Solid State Disk (SSD)	2 TB	0,25 ms	500
Flashkarten (ROM)	2 TB	1 ms	100
Magnet. Festplatten (HD)	16 TB	3,5 ms	150
optische Platten	60 GB	25 ms	10

Instruction Unit
Ablauf beim Ausführen von Befehlen

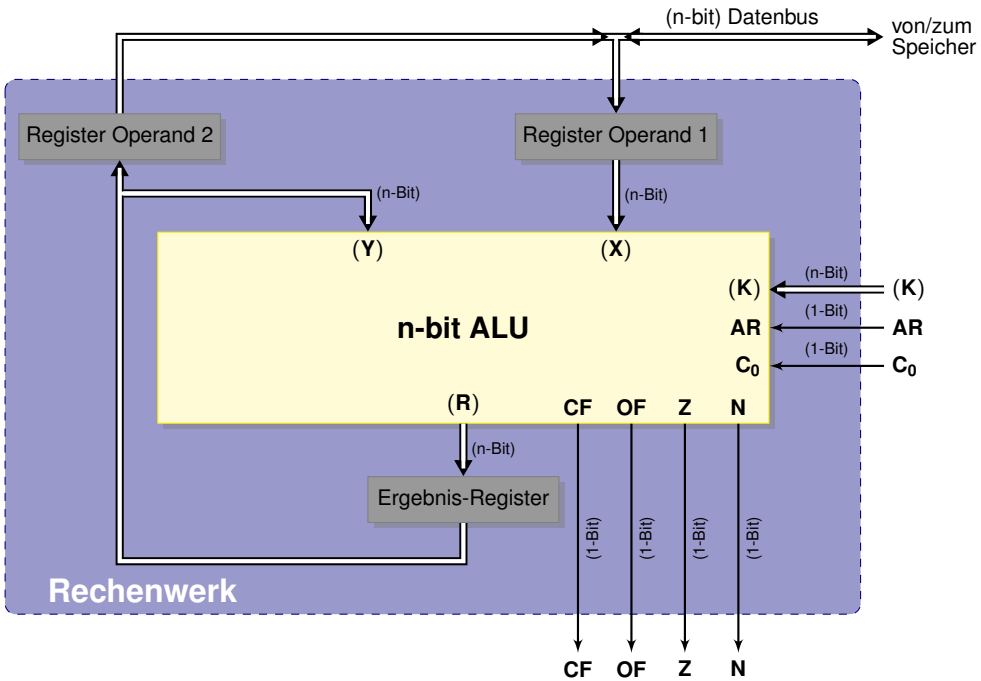


Schritt	**Instruction Phase**	
1	Befehl holen	Befehlsadresse → Adressbus, Lesesignal für Speicher, Befehl → Befehlsregister
2	Befehl dekodieren	Steuerbefehle erzeugen, Nebenaufgabe: Instruction Pointer auf nächsten Befehl setzen
	Execution Phase	
3	Operanden holen	Operandenadresse(n) → Adressbus, Lesesignal für Speicher Operanden → Operandenregister des Rechenwerks
4	Operation ausführen	Ausführung im Rechenwerk gemäß der Steuerbefehle des Steuerwerks
5	Ergebnis speichern	Ergebnisadresse → Adressbus, Schreibsignal für Speicher

Hinweis: Einzelne Phasen können entfallen bzw. in mehrere Teilschritte aufgeteilt werden, z.B. das Lesen von zwei Operanden aus demselben Speicher kann nur sequentiell erfolgen.

Execution Unit
ALU und Registerblock

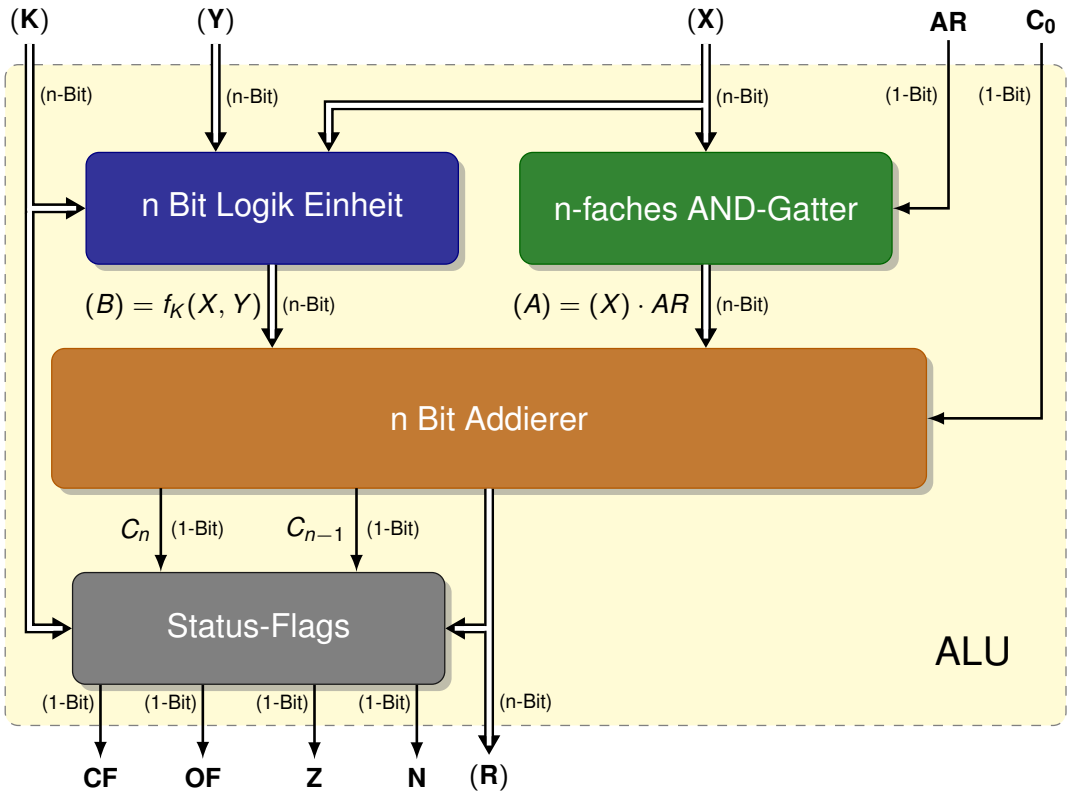
- Führt die **arithmetisch-logischen Befehle** eines Programms aus, besteht aus **Arithmetic Logic Unit ALU** und **Registerblock**.
- **Register** sind schnelle Zwischenspeicher für Operanden und Ergebnisse. Notwendig, da Speicher zu langsam.
- Manche Rechenwerke bestehen aus **mehreren ALUs** (z.B. einer ALU für die Mantissen und einer ALU für die Exponenten von Gleitkommazahlen)
- **Moderne CPUs** haben **mehrere Rechenwerke**, die unabhängig voneinander arbeiten (sogenannte **Superskalare Architekturen**)



(K),AR,C0 Steuersignale vom Steuerwerk. CF,OF,Z,N Statussignale zum Steuerwerk

Arithmetisch-Logische Einheit ALU
Beispielhafter Aufbau einer ALU

- Arithmetische und logische Operationen werden in Mikroprozessoren in einer **ALU** Arithmetic-Logic-Unit zusammengefasst.
- Die Einheit verknüpft zwei Operanden **(X)** und **(Y)** zu einem Ergebnis **(R)**.
- Steuersignale bestimmen die tatsächlich ausgeführte Operation und **Statusflags** zeigen die Gültigkeit und Eigenschaften des Ergebnisses an.



Steuersignale:

AR=1	schaltet Operand (X) an den Addierer durch, d.h. (A)=(X) · AR
(K):	Auswahl der Logik-Funktion $(B) = f_K(X, Y)$ durch $(K) = (k_3, k_2, k_1, k_0)$
(C_0):	Übertrag 1 kann voreingestellt werden

Statusflags:

CF:	Carry-Flag (Dualzahlen)
OF:	Overflow-Flag (2er-K-Zahlen)
Z:	Zero-Flag =1, falls (R)=0
N:	Negative-Flag =1, falls MSB von (R)=1

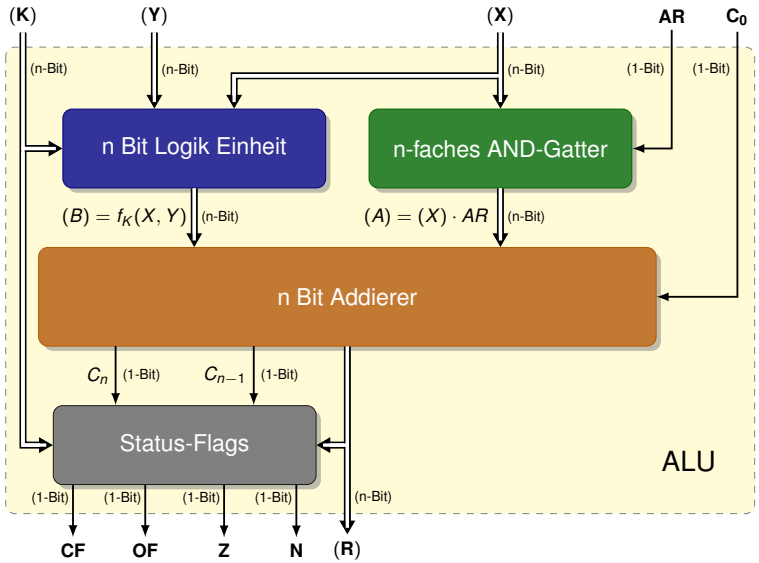
$$(R) = (B) + (A) + C_0 = f_K(X, Y) + (X) \cdot AR + C_0$$

- In der **Logik-Einheit** erfolgt eine Verknüpfung der Operanden **(X)** und **(Y)** zu einem Ergebnis **(B)** über eine Logik-Funktion, die mit dem Steuerwort (K) vorgegeben wird. Dabei wird immer **stellenweise** verknüpft.
- Beispiel-Tabelle der Steuerworte und der wichtigsten Funktionen einer Logik-Einheit:

Steuerwort (K)	**Ergebnis** $(B) = f_K(X, Y)$ Logikverknüpfungen erfolgen bitweise	**Logik-Funktion** f_K
$(0000) = 0_{16}$	$B = (0 \ldots 0)_2$	Kontradiktion
$(0001) = 1_{16}$	$B = \overline{(X) \vee (Y)}$	NOR
$(0011) = 3_{16}$	$B = \overline{(X)}$	Bitweise Invertierung X
$(0101) = 5_{16}$	$B = \overline{(Y)}$	Bitweise Invertierung Y
$(0110) = 6_{16}$	$B = (X) \nleftrightarrow (Y)$	XOR (Antivalenz)
$(0111) = 7_{16}$	$B = \overline{(X) \wedge (Y)}$	NAND
$(1000) = 8_{16}$	$B = (X) \wedge (Y)$	AND
$(1001) = 9_{16}$	$B = (X) \leftrightarrow (Y)$	XNOR (Äquivalenz)
$(1010) = A_{16}$	$B = (Y)$	Identität Y
$(1100) = C_{16}$	$B = (X)$	Identität X
$(1110) = E_{16}$	$B = (X) \vee (Y)$	OR
$(1111) = F_{16}$	$B = (1 \ldots 1)_2$	Tautologie

Arithmetisch-Logische Einheit ALU
Beispiele für Operationen

Mnemonic	Name Operation	Steuersignale (K)	AR	C_0	(B)	(A)	Operation (R)=
ADD	Addieren	A_{16}	1	0	(Y)	(X)	$(X)+(Y)$
SUB	Subtrahieren	5_{16}	1	1	$\overline{(Y)}$	(X)	$(X)-(Y)$
INCX	Inkrementieren	0_{16}	1	1	(0 . . . 0)	(X)	$(X)+1$
DECX	Dekrementieren	F_{16}	1	0	(1 . . . 1)	(X)	$(X)-1$
AND	Bitweise AND	8_{16}	0	0	$(X) \wedge (Y)$	(0)	$(X) \wedge (Y)$
NOTX	Bitweise Not	3_{16}	0	0	$\overline{(X)}$	(0)	$(\overline{X})$
NEGX	Negieren	3_{16}	0	1	$\overline{(X)}$	(0)	$-(X)$
LSL	Logic Shift Left	C_{16}	1	0	(X)	(X)	$(X) \ll 1 = X + X$



Steuerwort (K)	Ergebnis B = $f_K(X, Y)$	Logik-Funktion
$(0000)_2 = 0_{16}$	$(B) = (0\ldots0)$	Kontradiktion
$(0001)_2 = 1_{16}$	$(B) = \overline{(X) \vee (Y)}$	NOR
$(0011)_2 = 3_{16}$	$(B) = \overline{(X)}$	Bitweise Invertierung X
$(0101)_2 = 5_{16}$	$(B) = \overline{(Y)}$	Bitweise Invertierung Y
$(0110)_2 = 6_{16}$	$(B) = (X) \nleftrightarrow (Y)$	XOR (Antivalenz)
$(0111)_2 = 7_{16}$	$(B) = \overline{(X) \wedge (Y)}$	NAND
$(1000)_2 = 8_{16}$	$(B) = (X) \wedge (Y)$	AND
$(1001)_2 = 9_{16}$	$(B) = (X) \leftrightarrow (Y)$	XNOR (Äquivalenz)
$(1010)_2 = A_{16}$	$(B) = (Y)$	Identität Y
$(1100)_2 = C_{16}$	$(B) = (X)$	Identität X
$(1110)_2 = E_{16}$	$(B) = (X) \vee (Y)$	OR
$(1111)_2 = F_{16}$	$(B) = (1\ldots1)$	Tautologie

Rechenoperationen mit **2n bit Operanden** mit einer **n bit ALU**

- Vorgehen: Zerlegung der Berechnung in mehrere n bit Operationen
- Beispiel:
  8 bit Addition $(R) = (X) + (Y)$ mit 4 bit ALU durch Zerlegung in eine Addition der unteren 4 bit und eine Addition der oberen 4 bit

$(X) = 77_{10} = (0100\,1101)_2, \qquad (Y) = 110_{10} = (0110\,1110)_2$

(X)		0	1	0	0			1	1	0	1	77_{10}
(Y)		0	1	1	0			1	1	1	0	110_{10}
	0	1	0	0	1	←	1	1	0	0	0	
(R)	↓	1	0	1	1	↑	↓	1	0	1	1	187_{10}
	CF = 0					CF = 1						

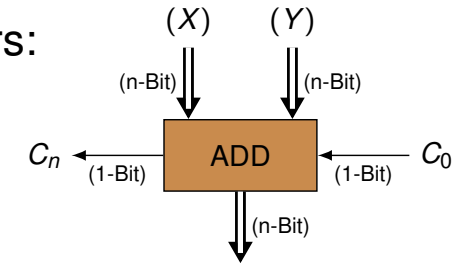
1. Schritt: **ADD** der unteren 4 bit (mit $C_0 = 0$)
   $\Rightarrow R_{low} = (X)_{low} + (Y)_{low}$ und Übertrag $CF = C_n$
2. Schritt: **ADDC** der oberen 4 bit (Add with Carry C_0 = CF aus 1.)
   $\Rightarrow R_{high} = (X)_{high} + (Y)_{high} + C_0$ und $CF = C_n$

Addition von Dualzahlen/2er-Komplementzahlen erfolgt nach dem Verfahren der stellenweisen Addition mit Übertrag (wie bei Dezimalzahlen):

- Beispiel: Addition zweier 8-stelliger Dualzahlen: $S = X + Y$.

		Binär								Dezimal
Operand 1	X		0	1	1	1	0	1	0	1	117_{10}
Operand 2	Y		0	0	0	0	1	0	0	1	9_{10}
Übertrag	C	0	0	0	0	0	0	0	1	0	
Ergebnis	S	0	0	1	1	1	1	1	1	0	126_{10} Dual oder 2K-Zahl

- Regeln für die Addition einzelner Stellen einer Dualzahl
  - Einzelne Stellen: 0+0 = 00; 0+1 = 01; 1+0 = 01, 1+1 = 10 (C S)
  - Mehrstellige Zahlen werden von rechts nach links unter Berücksichtigung der Überträge bearbeitet. Übertrag in die niederwertigste Stelle $C_0 = 0$
- Blockschaltbild eines n-Bit Addierers:



$(S) = [(X) + (Y) + C_0] \; mod \; 2^n$

Subtraktion $S = X - Y$ erfolgt durch Addition des 2er-Komplements $(Y)_{ZK}$

- Bildung des „2er-Komplements“:
  - Bitweise Invertieren des Subtrahenden Y $\rightarrow$ 1er-Komplement $(Y)_{EK} = \overline{(Y)}$
  - Das 2er-Komplement $(Y)_{ZK} = (Y)_{EK} + 1$ wird nicht explizit gebildet, sondern mittelbar durch einen Eingangsübertrag $C_0 = 1$ bei der Addition ersetzt:

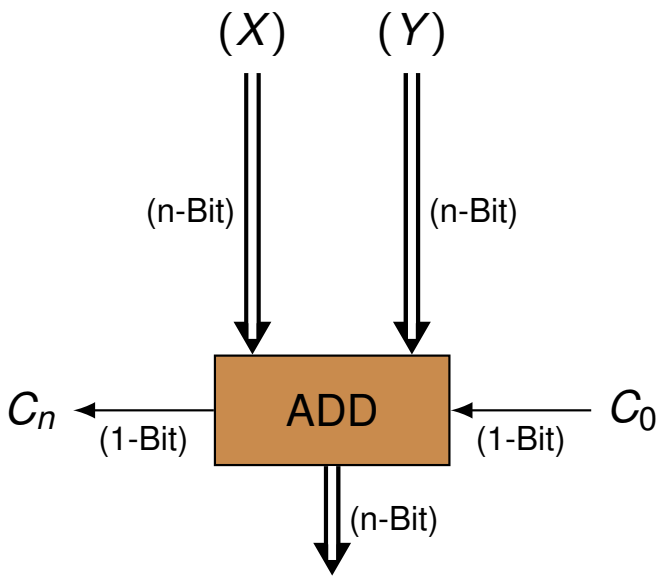
    $(S) = (X) + (Y)_{ZK} = (X) + (Y)_{EK} + 1 = (X) + \overline{(Y)} + 1$

- Beispiel: Subtraktion zweier n=8-stelliger Dualzahlen: $S = X - Y$.

		Binär									Dezimal
Operand 1	X		1	1	1	1	0	1	0	1	$245_{10} \mid -11_{10}$
Operand 2	Y		1	1	0	1	0	1	1	0	$214_{10} \mid -42_{10}$
Operand 1	X		1	1	1	1	0	1	0	1	
$\overline{\textbf{Operand 2}}$	$(Y)_{EK}$		**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	
Übertrag	C	1	1	1	1	0	0	0	1	**1**	
Ergebnis	S		0	0	0	1	1	1	1	1	31_{10} D | 2K

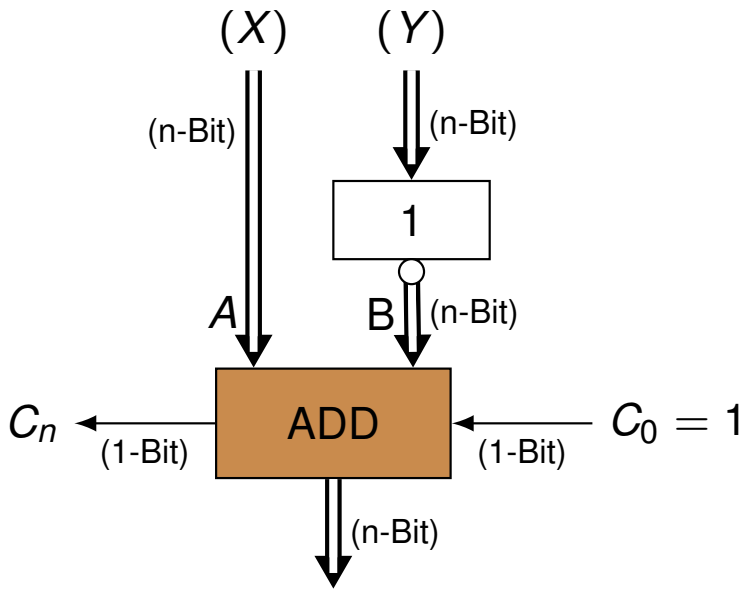
Blockschaltbilder:

n-Bit Addierer



$(S) = [(X) + (Y) + C_0] \; mod \; 2^n$

n-Bit Subtrahierer



$(S) = [(X) - (Y)] \; mod \; 2^n$

Überlauferkennung bei Addition und Subtraktion siehe Kapitel 3:

- Dualzahlen bei Addition $CF = C_n$, bei Subtraktion $CF = \overline{C_n}$
- 2er-Komplement-Zahlen bei Addition und Subtraktion $OF = C_n \leftrightarrow C_{n-1}$

Realisierungsidee: **Kaskadierung von einstelligen Addierern**

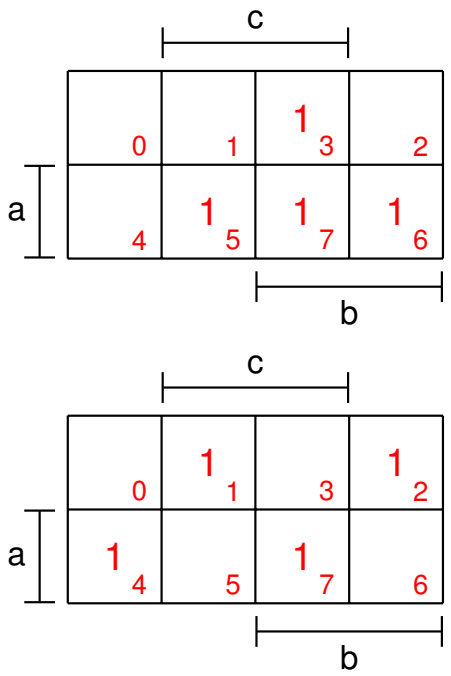
- Kriterien:
  1. Schaltungsaufwand: Metrik Funktionslänge
  2. Durchlaufzeit („Rechenzeit“): Metrik Schachtelungstiefe
- **Halbaddierer HA** Variante 1: 1 bit-Addierer **ohne** Eingang für Übertrag

a	b	U	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Funktionsgleichungen Halbaddierer Variante 1 in DMF:
  Übertrag $U_{HA} = a \cdot b$ AND Summe $S_{HA} = \overline{a} \cdot b \vee a \cdot \overline{b}$ XOR
- Funktionslänge (in DMF): $l_{HA,DMF} = l_U + l_S = 2 + 6 = 8$
- Durchlaufzeit (in DMF) für U: $t_p = 1 \cdot t_G$ für S: $t_p = 2 \cdot t_G$
  dabei t_G: Laufzeit eines einzelnen AND- bzw. OR-Gatters, Inverter zählen nicht

- **Volladdierer VA** Variante 1: 1bit-Addierer **mit** Eingang für Übertrag

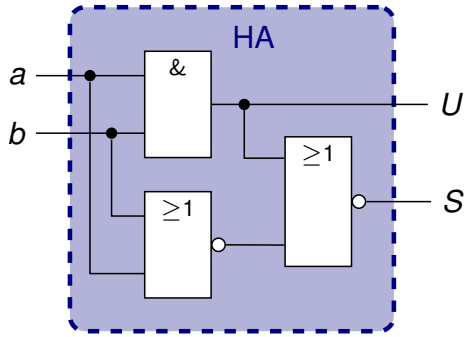
	a	b	c	U	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1



- Funktionsgleichung Volladdierer Variante 1 in DMF:
  Übertrag $U_{VA} = a \cdot b \vee a \cdot c \vee b \cdot c$
  Summe $S_{VA} = \overline{a} \cdot \overline{b} \cdot c \vee \overline{a} \cdot b \cdot \overline{c} \vee a \cdot \overline{b} \cdot \overline{c} \vee a \cdot b \cdot c$
- Funktionslänge (in DMF): $l_{VA,DMF} = l_U + l_S = 9 + 16 = 25$
- Durchlaufzeit (in DMF) für U und S: $t_{p,DMF} = 2 \cdot t_G$

Mehrstufige Realisierung mit reduziertem Schaltungsaufwand

- **Halbaddierer** Variante 2 aus AND + 2 NOR (Achtung: keine DMF-Struktur)



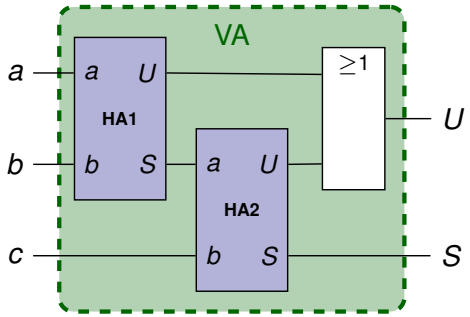
Funktionslänge: $l^*_{HA} = 6 \quad (l_{HA,DMF} = 8)$

Durchlaufzeit:

Pfad (a,b) $\rightarrow$ U: $t_p = 1 \cdot t_G$

Pfad (a,b) $\rightarrow$ S: $t_p = 2 \cdot t_G$

- **Volladdierer** Variante 2 aus 2 HA Variante 2 + OR



Funktionslänge: $l^*_{VA} = 2 \cdot l^*_{HA} + 2 = 14$

Durchlaufzeit: $\mathbf{(l_{VA,DMF} = 25)}$

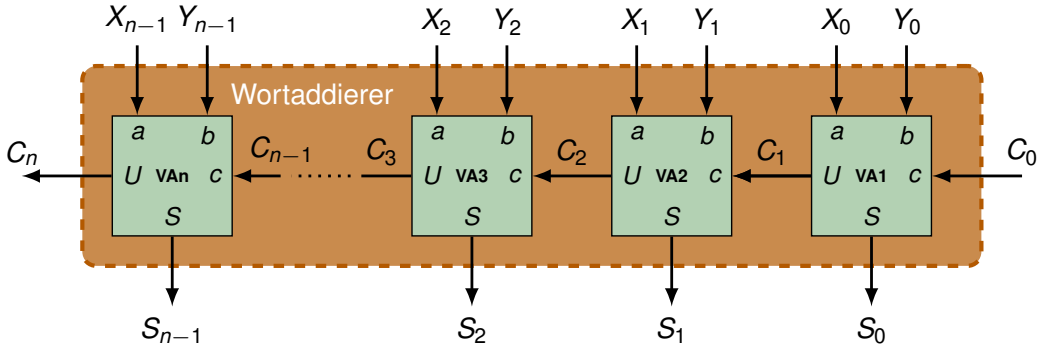
Pfad (a,b) $\rightarrow$ U,S: $t_p = 4 \cdot t_G$ (DMF: $2t_G$)

Pfad (c) $\rightarrow$ U,S: $t_p = 2 \cdot t_G$

Typisch für alle Hardware- und Softwarelösungen:

Kompromiss Aufwand $\Leftrightarrow$ Geschwindigkeit nötig

Kaskadierung von n Volladdierern: n Bit **Ripple-Carry Adder**



- Arithmetische Gleichungen:

  $(S) = [(X) + (Y) + C_0] \bmod 2^n \qquad C_n = [(X) + (Y) + C_0] \operatorname{div} 2^n$

- Bei Realisierung mit den vereinfachten Volladdieren Variante 2:

  Worst Case Durchlaufzeit $(x_0, y_0, c_0) \rightarrow (c_n, s_{n-1}) : \; t_P \leq (n+1) \cdot 2t_G$

  Schaltungsaufwand $I_{RippleCarry} = n \cdot I^*_{VA} = 14 \cdot n$ **beides ~n**

Carry-Look-Ahead Addierer für verbesserte Durchlaufzeit

- Reduzierung der Durchlaufzeit durch **direkte Berechnung der Überträge** mit einer 2-stufigen Logik **Ziel: Zeit unabhängig von n**
- Erfordert deutlich **höheren Schaltungsaufwand**, daher häufig Kompromiss: z.B. 32-bit Addierer aus Kaskade mit vier 8-bit Carry-Look-Ahead Addierern

Verschieben der einzelnen Stellen innerhalb eines Datenwortes.

- Wirkung der Schiebeoperationen bei **Dezimalzahlen**
- Beispiel $\mathbf{Z = 40302_{10}}$:
  - **Linksschieben** um k=2 Stellen: $Z \rightarrow Z' = 4030200_{10}$
    (Shift left) entspricht einer Multiplikation mit $10^k = 10^2$
  - **Rechtsschieben** um k=2 Stellen: $Z \rightarrow Z' = 403_{10}$
    (Shift right) entspricht einer ganzzahligen Division (DIV) durch $10^k = 10^2$
- Wirkung der Schiebeoperationen bei **binär codierten Zahlen** (X)
  - **Linksschieben** um k Stellen: $(X) \rightarrow (X)' = \left[(X) \cdot 2^k\right] \bmod 2^n$

    entspricht einer Multiplikation mit 2^k, evtl. Überlauf
    von rechts ist der Wert 0 nachzuschieben (LSB = 0)
  - **Rechtsschieben** um k Stellen: $(X) \rightarrow (X)' = (X) \text{ div } 2^k$

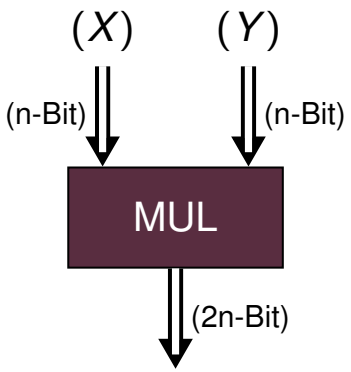
    entspricht einer ganzzahligen Division durch 2^k (DIV), kein Überlauf
    Bei Betragszahlen: MSB = 0
    Bei Ganzen Zahlen: MSB bleibt bestehen, d.h. MSB wird kopiert

- Beim Linksschieben kann es wie beim Addieren zu Überläufen kommen.
- Beim Rechtsschieben muss bei 2er-Komplement-Zahlen das MSB kopiert werden.
  CPUs haben unterschiedliche Befehle: Logic Shift Right LSR für Betragszahlen und Arithmetic Shift Right ASR für 2er-Komplement-Zahlen

Multiplikation von Betragszahlen und von Ganzen Zahlen im 2er-Komplement

- Die **Multiplikation** wird ausschließlich mit **positiven Werten** ausgeführt. Für **negative Faktoren** wird der Betrag (**2er-Komplement**) gebildet und das Vorzeichen separat behandelt.

  Daher kann für ganze Zahlen derselbe Algorithmus wie für Betragszahlen eingesetzt werden.
- **Separate Vorzeichenbetrachtung**:
  Ergebnis wird **negativ**, falls genau einer der **beiden Faktoren negativ**
  Falls die Vorzeichenbetrachtung ein **negatives Ergebnis** ergibt, wird das **2er- Komplement** zu dem Produkt der Betragsmultiplikation gebildet.
- Damit Überläufe grundsätzlich vermieden werden, wird das Ergebnis der n x n bit Multiplikation als 2n bit Ergebnis berechnet.
- Blockschaltbild eines n-Bit Multiplizieres:



Variante 1: Berechnung durch **mehrfache Summation**: $P = \sum_{i=1}^{X} Y$

- Beispiel: $4 \cdot 7 = ((7 + 7) + 7) + 7$

			1	1	1	= 7
+			1	1	1	= 7
		1	1	1	0	
		1	1	1	0	= 14
+			1	1	1	= 7
	1	1	1	0	0	
	1	0	1	0	1	= 21
+			1	1	1	= 7
	0	1	1	1	0	
	1	1	1	0	0	= 28

- Nachteil: **sehr langsam**

Variante 2: **Tabelle aller Produkte im Speicher Kleines Ein-mal-Eins**

- Bei n x n bit ist eine $2^{2n} \cdot 2n$ bit große Tabelle nötig.
  z.B. bei n=4: $2^8 \cdot 8$ bit = 2 KiBit, aber bei n=16: $2^{32} \cdot 32$ bit = 128 GiBit
- Für große n technisch nicht machbar

Variante 3: Stellenweise Multiplikation und stellenrichtige Addition

- In Anlehnung an das handschriftliche Multiplizieren von Dezimalzahlen

Beispiel:

5	3	·	2	3
			1	5	9
	1	0	6	
1	2	1	9	

- Beispiel: n x n - Multiplikation von Betragszahlen für n = 4

$(P) = (X) \cdot (Y)$ mit $(X) = 13_{10} = D_{16}$ und $(Y) = 11_{10} = B_{16}$

1	1	0	1	·	1	0	1	1
					1	1	0	1
				1	1	0	1	
			0	0	0	0		
		1	1	0	1			
	1	1	1	1	0	0	0	0
	1	0	0	0	1	1	1	1

Teilprodukt $(P_0) = (X) \cdot Y_0 \cdot 2^0$

Teilprodukt $(P_1) = (X) \cdot Y_1 \cdot 2^1$

Teilprodukt $(P_2) = (X) \cdot Y_2 \cdot 2^2$

Teilprodukt $(P_3) = (X) \cdot Y_3 \cdot 2^3$

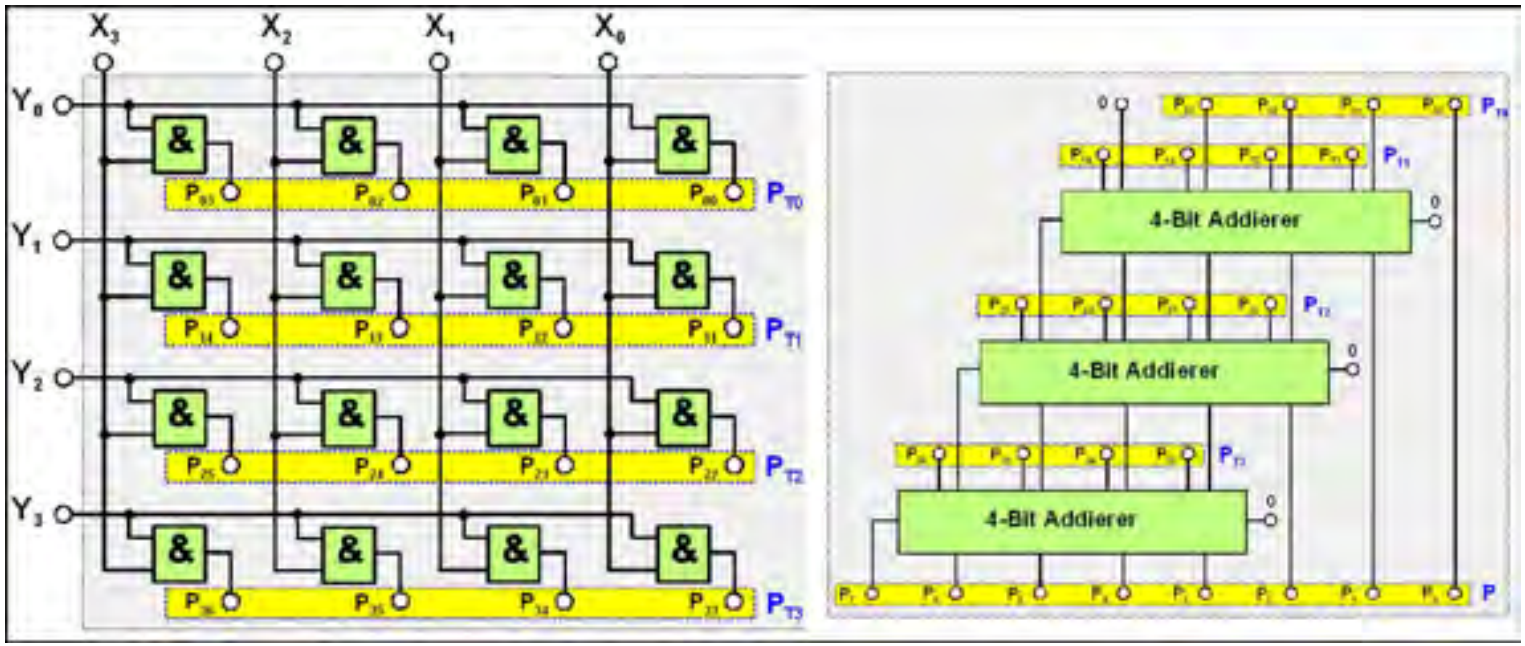
Überträge

$(P) = (P_0) + (P_1) + (P_2) + (P_3)$

$(P) = 8F_{16} = 143_{10}$

- Realisierung mit Schaltnetz (kombinatorische Schaltung) in 2 Schritten:
  1. stellenweise Multiplikation
  2. stellenrichtige Addition.
- Schaltbild für n=4:
  Einstellige Multiplikation, Ergebnis: Teilprodukte P_{Ti}
  Addition der Teilprodukte nach Stellenwertigkeit. Ergebnis: Produkt P
- Schaltungsaufwand: n x n AND-Gatter und (n-1) n-bit-Addierer

$$l_{Mult} = 2n^2 + (n-1) \cdot 14n = 16n^2 - 14n \approx 16n^2$$ **zum Vergl. ADD ~14*n***



- Wie bei der Multiplikation wird nur mit **Betragszahlen** gerechnet und das **Vorzeichen getrennt** betrachtet.

  Quotient: (Q) = (X) DIV (Y) (X): 2n-Bit Dividend (Y): n-Bit Divisor

  mit Rest: (R) = (X) MOD (Y) (Q): n-Bit Quotient (R): n-Bit Rest

  DIV ... ganzzahlige Division, MOD ... Rest der ganzzahligen Division

- Beispiel mit Dezimalzahlen:

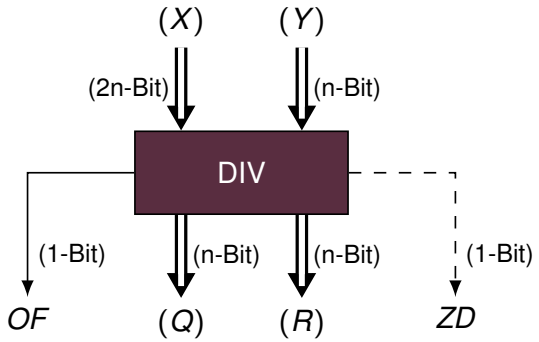
  17 : 3 = 5 Rest 2, Q= 17 DIV 3 = 5, R = 17 MOD 3 = 2

- Blockschaltbild eines n-Bit Dividierers:

Flags:

OF: Overflow

ZD: Zero Divide (optional)



- Falls das Ergebnis nicht mit n-Bit dargestellt werden kann, entsteht ein Überlauf, Anzeige durch OF
- Vorzugsweise wird wie bei der Multiplikation **stellenweise** vorgegangen:
  Der Quotient kann wie bei der handschriftlichen Division mit Dezimalzahlen von der höchsten Stelle beginnend ermittelt werden

Beispiel: $n = 4, \quad (X) = 36_{10}, (Y) = 7_{10} \rightarrow (Q) = 5_{10}, \quad (R) = 1_{10}$

	0	0	1	0	0	1	0	0	/	0	1	1	1	=	0	1	0	1
i=3	-	0	0	0	0	↓	|	|									
		0	1	0	0	1	|	|									
i=2		-	0	1	1	1	↓	|									
			0	0	1	0	0	|									
i=1			-	0	0	0	0	↓									
				0	1	0	0	0									
i=0				-	0	1	1	1									
					0	0	0	1									

Mikroprozessoren besitzen häufig getrennte Befehle für die Multiplikation bzw. Division von Betragszahlen und von Ganzen Zahlen

- Beispiel: Intel x86 (Core i3/i5/i7)

DIV	Assemblerbefehl für Division mit Betragszahlen
IDIV	Assemblerbefehl für Division mit Ganzen Zahlen (Integer Division)
MUL	Assemblerbefehl für Multiplikation mit Betragszahlen
IMUL	Assemblerbefehl für Multiplikation mit Ganzen Zahlen (Integer Multiply)

Rechenwerke können beliebig nichtlineare Funktionen in der Regel nicht direkt berechnen, sondern verwenden eine der folgenden Möglichkeiten:

- Alternative 1: Berechnung mittels **Potenzreihe** (Taylor-Reihe)

  Man kann zeigen, dass jede nichtlineare Funktion als unendliche Potenzreihe dargestellt werden kann:

  $y = f(x) = \sum_{i=0}^{\infty} {a_i}^i \cdot (x - x_0)^i \approx \sum_{i=0}^{n} {a_i}^i \cdot (x - x_0)^i$

  dabei sind $a_0, a_1, a_2, \cdots$ konstante Faktoren und x_0 ein konstanter Wert.

  Beispiele:

  $y = sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$

  $y = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

  Da man in der Praxis natürlich nicht unendlich viele Terme ausrechnen kann, muss man $i_{max} = n$ (und x_0) so wählen, dass man eine möglichst hohe Genauigkeit bei kleinstmöglichem Rechenaufwand erhält.

  Für viele Funktionen existieren außer der Potenzreihe auch andere Näherungsformeln, bei denen der Wert zum Teil auch in mehreren Schritten (iterativ) berechnet wird, z.B. der CORDIC-Algorithmus (siehe z.B. Wikipedia).

- Alternative 2: **Funktionstabelle** (Lookup-Table)

x	$y = sin(x)$
0°	0
30°	0.5
45°	0.7071
...	...

Nachteil:
Für ausreichende Genauigkeit feingranulare (= große) Tabelle nötig.

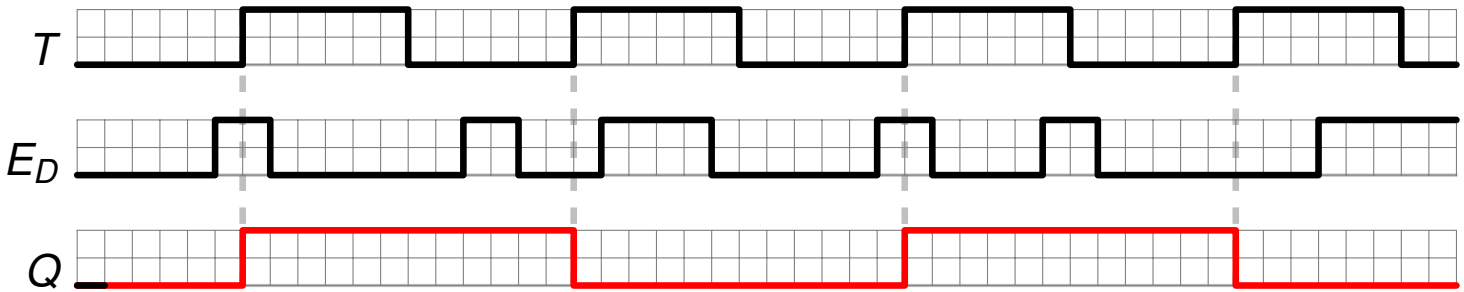
- Alternative 3: **Funktionstabelle und Interpolation** von Zwischenwerten
  - 1. Schritt: Stützstellensuche
    Gesucht werden die beiden Werte $x_1, y_1 = f(x_1)$ und $x_2, y_2 = f(x_2)$ aus der (groben) Funktionstabelle mit $x_1 \leq x \leq x_2$, d.h. die Werte in der Tabelle, die links bzw. rechts vom gesuchten Wert x liegen.
  - 2. Schritt: Interpolation zwischen den Stützstellen: $y \approx \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) + y_1$

Um eine höhere Genauigkeit zu erhalten, kann statt der linearen Interpolation auch ein Interpolations-Polynom verwendet werden.

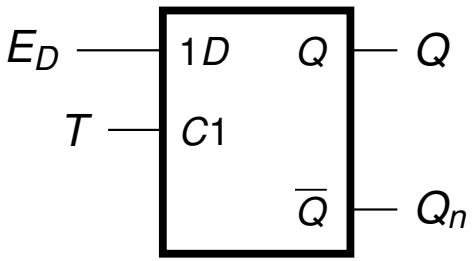
Aufgabe: Speichern eines Bits

- **Schreiben**: $Q^{k+1} = E_D^k$ (Übernahme $E_D \rightarrow Q$) wenn Takt T aktiv
- **Speichern**: $Q^{k+1} = Q^k$ unabhängig von E_D, wenn Takt T inaktiv
- Zwei unterschiedliche Konzepte der Taktsteuerung:
  - **Zustandssteuerung**: Takt aktiv $T = 1$, inaktiv $T = 0$
  - **Flankensteuerung**: Takt aktiv $T = 0 \rightarrow 1$ Übergang, sonst inaktiv

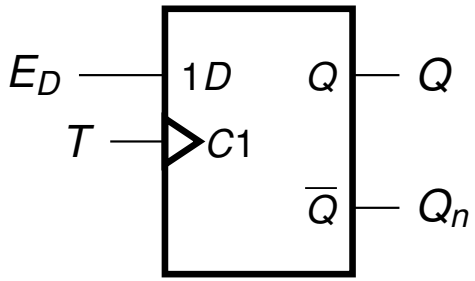
Beispiel: Impulsdiagramm für positiv taktflankengesteuertes D-FF



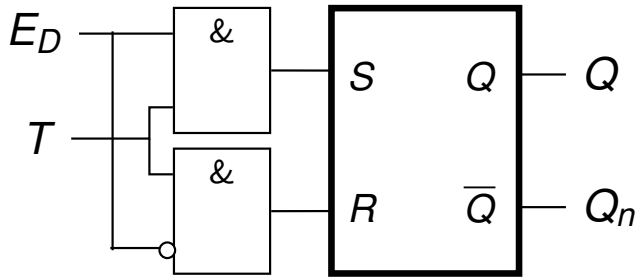
(Takt)-Zustandssteuerung



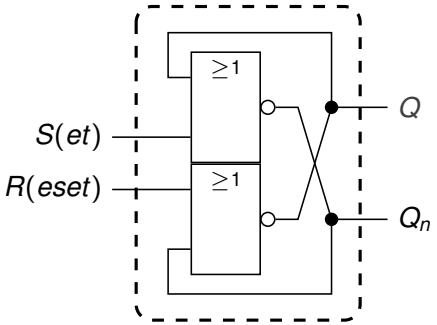
(Takt)-Flankensteuerung



Innerer Aufbau eines zustandsgesteuerten D(elay)-Flipflops:



mit dem RS-FF

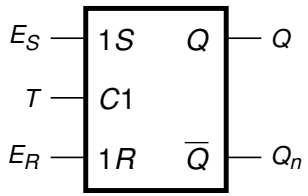
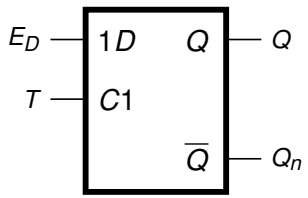
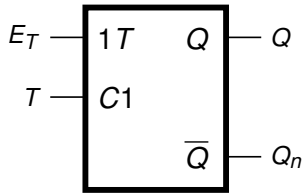
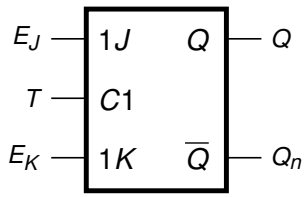


Die Funktion $Q^{k+1} = E_D^k$ wird als **Zustandsübergangsfunktion** bezeichnet. Sie beschreibt das Ausgangssignal Q^{k+1} in der Taktperiode $k + 1$ als Funktion des Eingangssignals E_D^k am Ende der Taktperiode k.

Diese 1 bit-Speicherzelle kann mit 6 CMOS-Transistoren implementiert werden und ist die **Grundspeicherzelle aller SRAMs** (statisches RAM).

Hinweis:
Neben dem D-FF gibt es eine Reihe weiterer Flipflop-Typen (siehe nächste Seite), die hier nicht besprochen werden sollen.

Durch **zusätzliche Beschaltung** aus dem RS-Flipflop abgeleitete FF-Typen:

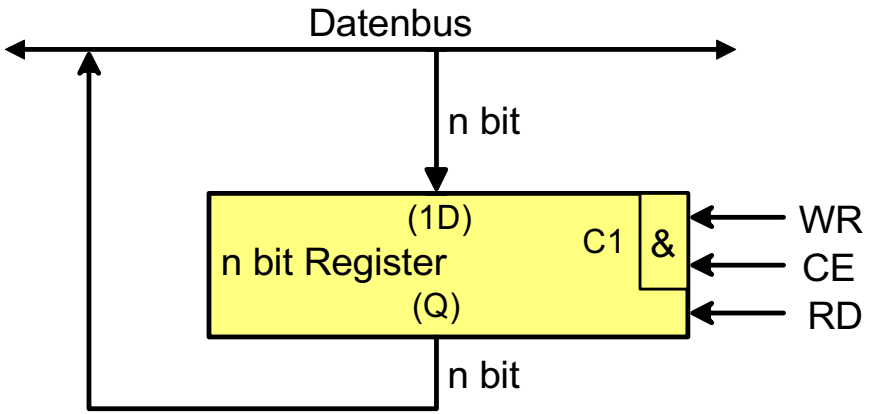
FF–Typ	Schaltbild	Vereinfachte Funktionstabelle	Aus RS-FF mit
RS–FF		E_R^k | E_S^k | Q^{k+1} 0 | 0 | $\mathbf{Q^k}$ 0 | 1 | **1** 1 | 0 | **0** 1 | 1 | **vermeid.**	
D(elay)–FF		E_D^k | Q^{k+1} 0 | **0** 1 | **1**	$S = E_D, R = \overline{E_D}$
T(oggle)–FF		E_T^k | Q^{k+1} 0 | $\mathbf{Q^k}$ 1 | $\overline{\mathbf{Q^k}}$	$S = E_T \cdot Q_n, R = E_T \cdot Q$
JK–FF		E_K^k | E_J^k | Q^{k+1} 0 | 0 | $\mathbf{Q^k}$ 0 | 1 | **1** 1 | 0 | **0** 1 | 1 | $\overline{\mathbf{Q^k}}$	$S = E_J \cdot Q_n, R = E_K \cdot Q$

In der Praxis ist das D-FF der wichtigste Flipflop-Typ.

Aufgabe: Schnelles Zwischenspeichern von n bit-Variablen

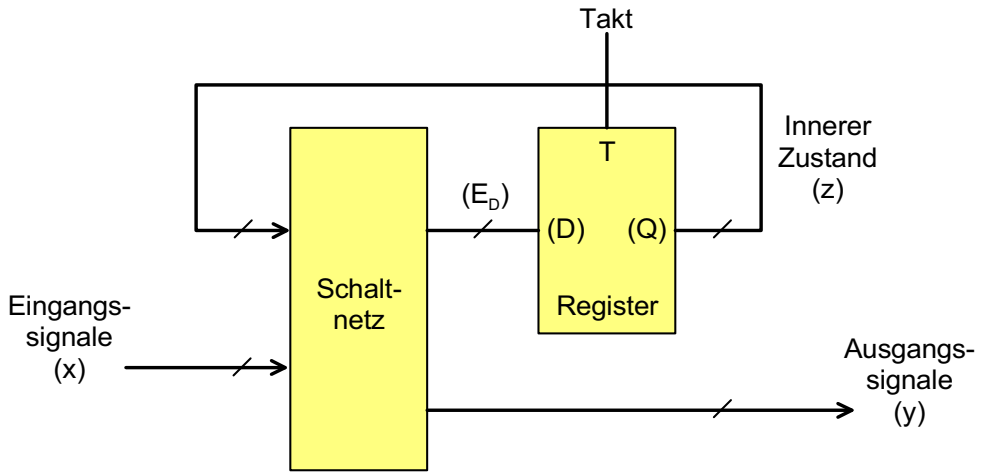
- **Register** werden aus einer Gruppe von zustandsgesteuerten D-Flipflops aufgebaut und durch Steuersignale der CPU angesteuert. Typischerweise haben die Register dieselbe Datenbreite wie die CPU und der Speicher, z.B. 8, 16, 32 oder 64 bit.
- Häufig werden mehrere Register in der CPU zu einem **Registerblock** zusammengefasst.
- Beispiel: Register mit Steuersignalen

WR...Write
RD...Read
(D)...Datenbus der CPU
CE...Chip-Enable Auswahlsignal



Aufgabe: Erzeugung von Steuersignalen

- Kombination eines Schaltnetzes und eines Registers
- Im Gegensatz zu einem einfachen Schaltnetz hat das Schaltwerk ein „Gedächtnis“, d.h. die Ausgangssignale (Y) hängen nicht nur von den aktuellen Eingangssignalen (X), sondern auch von der Vorgeschichte ab, die in den Registern als Zustand (Q) gespeichert wird.
- Andere Bezeichnungen: Schaltwerk, (Zustands)-Automat, Finite State Machine FSM, Ablaufsteuerung
- Schaltwerke werden z.B. in CPUs zur Erzeugung der Steuersignale beim Lesen und Schreiben des Speichers verwendet.



- Typische Schaltwerke:
  - Auf- und Abwärtszähler, Zeitgeber (Timer)
  - Schieberegister

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

EINFÜHRUNG
Struktur von Anwendungsprogrammen

COMPILER UND CO.
Programmiersprachen
Werkzeugkette
Elemente von Programmiersprachen
Strukturierte Datenablage

BETRIEBSSYSTEME
Schichtenmodell
Aufgaben
Übersicht Betriebssysteme

SOFTWAREENTWICKLUNG
Vorgehensmodelle
Unified Modeling Language UML

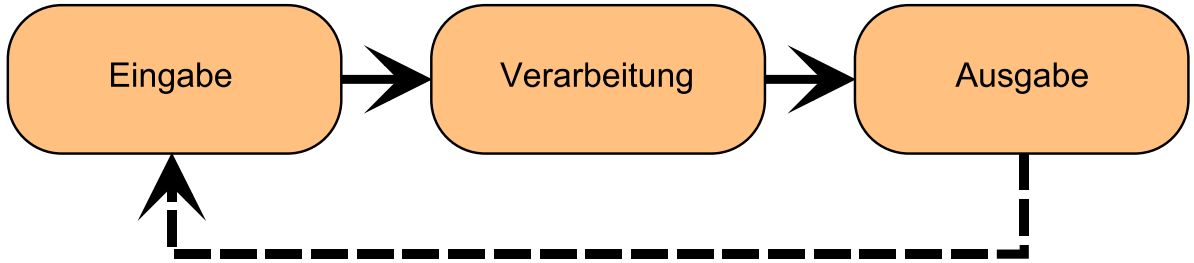
EINFÜHRUNG
Struktur von Anwendungsprogrammen

COMPILER UND CO.
Programmiersprachen
Werkzeugkette
Elemente von Programmiersprachen
Strukturierte Datenablage

BETRIEBSSYSTEME
Schichtenmodell
Aufgaben
Übersicht Betriebssysteme

SOFTWAREENTWICKLUNG
Vorgehensmodelle
Unified Modeling Language UML

Programme arbeiten einmalig oder zyklisch nach dem **EVA-Prinzip**:

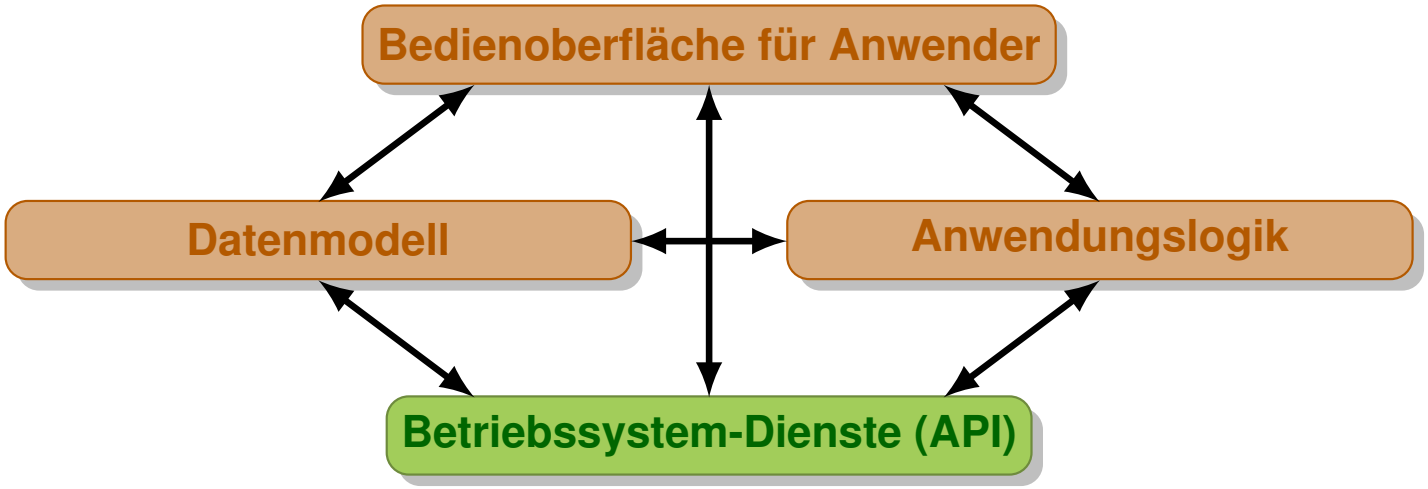


- **Eingabe** von Daten aus verschiedenen **Quellen**:
  - Benutzereingaben über Tastatur, Maus oder Touchpad
  - Dateien vom Massenspeicher oder über das Netzwerk
  - Datenbanken
  - Hardware-Schnittstellen, z.B. Soundkarte, Analog- oder Digitaleingänge von Sensoren (Messglieder), insbesondere bei Embedded Systemen
- **Verarbeitung** nach einem im Programm definierten **Algorithmus**
- **Ausgabe** der Ergebnisse auf verschiedene **Senken**:
  - Ausgabe über Bildschirm oder Drucker
  - Speichern in Dateien auf dem Massenspeicher oder über das Netzwerk
  - Datenbanken
  - Hardware-Schnittstellen, z.B. Soundkarte, Analog- oder Digitalausgänge zu Aktoren (Stellglieder), insbesondere bei Embedded Systemen

Anwendungsprogramme sollten in ihrer internen Struktur Folgendes trennen:

- **Benutzerschnittstelle**: Aus- und Eingabe des Programms zum Benutzer (oder zu anderen Computern oder technischen Systemen)
- **Anwendungslogik**: In diesem Teil des Programm findet die eigentliche Datenverarbeitung statt, z.B. bei einer Textverarbeitung die Formatierung des Textes.
- **Datenmodell**: In diesem Teil werden die Daten strukturiert und gespeichert. Je nach Aufgabenstellung und Datenmenge werden die Daten im Speicher, als Datei im Dateisystem des Computers, in einer Datenbank oder in einer Kombination gehalten.

Im Idealfall kann so z.B. die Datenquelle bzw. das Datenformat ausgetauscht oder eine andere Benutzeroberfläche eingeführt werden, ohne das gesamte Programm neu entwickeln zu müssen.



Entwicklung der Programmiersprachen *1

- Vor 1960: Maschinensprache (**Assembler**)
  Direktes Programmieren mit den CPU-spezifischen Maschinenbefehlen
- Seit 1960: 1. Generation der **„höheren“ Programmiersprachen**
  - **FORTRAN** für technisch-wissenschaftliche Anwendungen. Erste Bibliotheken für mathematische Algorithmen, z.B. Lösen von Gleichungssystemen.
  - **ALGOL** als Alternative zu FORTRAN.
  - **COBOL** für betriebswirtschaftliche Anwendungen.
- Seit 1980: Höhere Programmiersprachen für die **PC-Ära**
  - **C**: Effiziente Hochsprache mit standardisierten Bibliotheken, von Microsoft für die Implementierung und DOS/Windows ausgewählt. Standard für Unix.
  - **PASCAL**: Weiterentwicklung von ALGOL für die Informatik-Ausbildung.
  - **BASIC**: Interpretersprache zum Programmieren für Nicht-Informatiker.
- Seit 1990: Durchbruch der **Objektorientierten Programmierung (OOP)**
  - **C++**: Objektorientierte Version von C, wichtigste Sprache für Windows durch Microsoft Foundation. Class (MFC) Bibliothek.
  - **DELPHI**: Objektorientierte Version von PASCAL.
  - **Visual BASIC**: Entwicklungsumgebung zum Entwurf von Graphischen Bedienoberflächen (GUI) für Nicht-Informatiker.

*1

Die Jahreszahlen geben den Beginn der kommerziellen Nutzung an. Die Sprachen selbst sind zum Teil deutlich älter. Die Darstellung ist stark vereinfacht.

Entwicklung der Programmiersprachen (Fortsetzung)

- Seit 1995: CPU- und Betriebssystem unabhängige **„Write Once - Run Everywhere“** Software
  - **JAVA**: Angriff auf die Microsoft-Welt durch Sun/Oracle (und IBM) auf Grundlage einer vereinfachten C++-Syntax, Erfolg durch leistungsfähige Klassenbibliotheken für graphische Oberflächen, Internet-Kommunikation und Datenbankanbindung.
    Faktisch Nachfolger von COBOL für betriebswirtschaftliche Anwendungen.
  - **C#/.NET**: Microsoft’s Gegenangriff zu JAVA. Vorteil: Einfachere Integration von C++-“Altprogrammen“ und von vielen anderen Programmiersprachen.
- Seit 2000: Programmiersprachen für **„Web/Internet-Anwendungen“**
  - **JavaScript**: Programmausführung in Web-Seiten innerhalb des Web-Browsers. Heute mit Google’s NodeJS Laufzeitumgebung auch serverseitig bzw. auf dem Desktop. Diverse Weiterentwicklungen, z.B. Microsoft’s TypeScript.
  - **PHP, Perl, Ruby, ...**: Meist serverseitig eingesetzte Skriptsprachen.
  - **Kotlin**: Google’s Weiterentwicklung von JAVA für Android wegen Lizenzstreitigkeiten mit Sun/Oracle.
  - **Swift**: Apple’s Ergänzung zu **Objective-C** (einem C++-Dialekt) für iOS Smartphones.

Entwicklung der Programmiersprachen (Fortsetzung)

- Sprachen für spezielle Anwendungen
  - **Batch/Shell Skripte** zur Automatisierung von Betriebssystemen.
  - **Python, R, Matlab, ...**: Skriptsprachen mit leistungsfähigen Bibliotheken und Werkzeugketten (Frameworks) für Nicht-Informatiker für Maschinelles Lernen, Statistik, Simulationen und numerische Analyse aller Art.
  - **Structured Query Language SQL, LINQ, ...**: Abfragesprachen für Datenbanken.
  - **VHDL, Verilog, SystemC**: Sprachen für die Simulation und automatische Synthese von Digitaler Logik.
  - ...

Es werden ständig neue Programmiersprachen entwickelt, die nach kurzem *Hype* meist schnell wieder in Vergessenheit geraten. *Platzhirsche* sind (und bleiben wahrscheinlich) **C/C++, Java, C#, Javascript** und (vielleicht) **Python** (aktuelles Ranking z.B. www.tiobe.com/tiobe-index)

Praktisch alle Programmiersprachen verwenden dieselben Grundkonzepte und Sprachelemente (siehe übernächster Abschnitt).

Beispiele: Hello World in verschiedenen Programmiersprachen

C

```c
#include <stdio.h>

int main(void)
{   printf("Hello World\n");
    return 0;
}
```

C++

```cpp
#include <iostream>
using namespace std;

int main()
{    cout << "Hello, World!";
     return 0;
}
```

Java

```java
public class HelloWorld
{
    public static void main (String[] args)
    {   System.out.println("Hello World!");
    }
}
```

Pascal

```
program HelloWorld;
begin
    writeln('Hello, World!');
end.
```

C#

```
namespace HelloWorld
{
    class Hello
    {   static void Main(string[] args)
        {   System.Console.WriteLine("Hello World!");
        }
    }
}
```

Javascript in HTML

```
<!DOCTYPE HTML>
<html>
<body>
  <p>Hello World in HTML/Javascript</p>
  <script>
    alert( 'Hello, world!' );
  </script>
  <p>This message box is executed via Javascript.</p>
</body>
</html>
```

Python

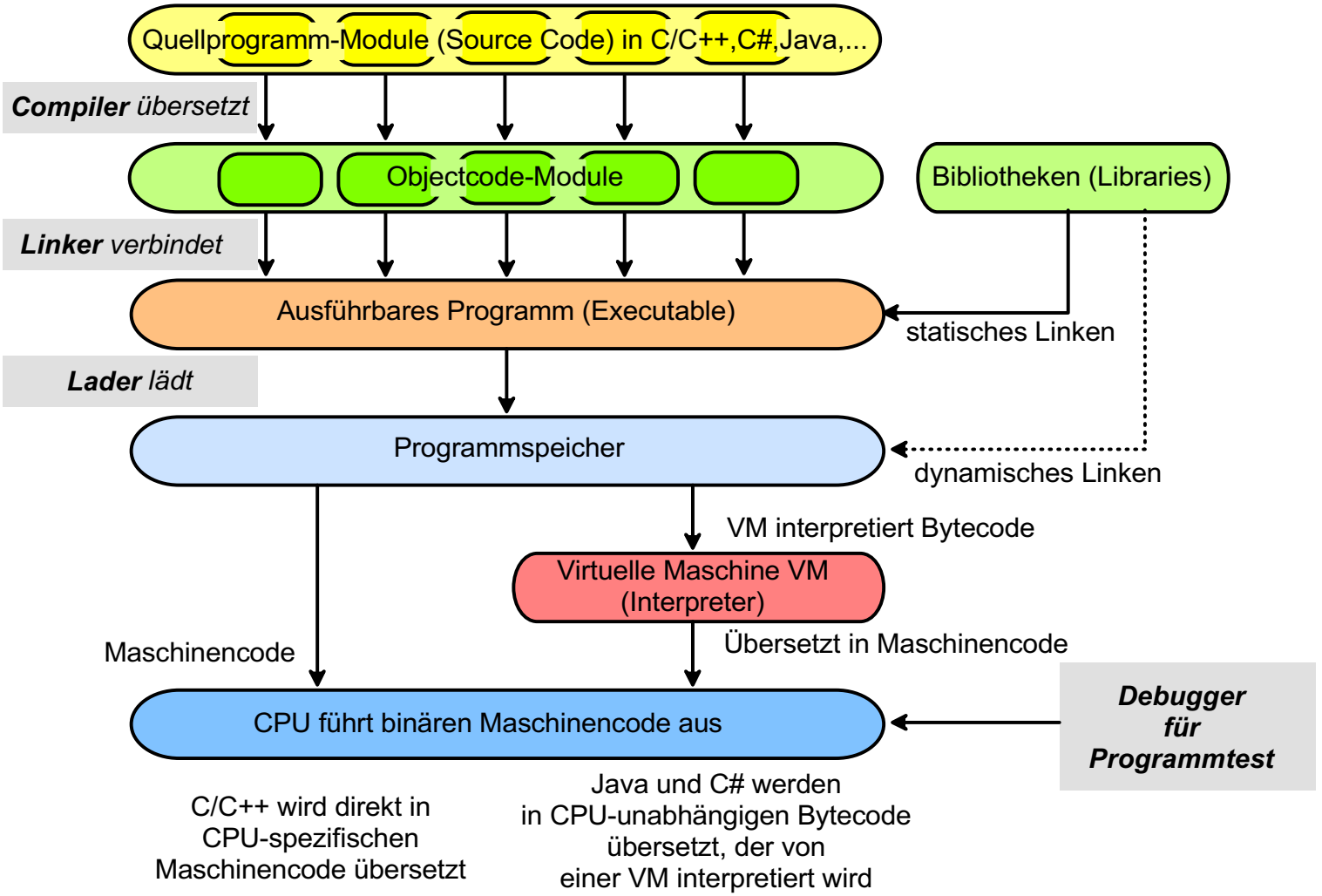
```
def main():
        print('Hello, world!')
```

Visual Basic Script

```
WScript.Echo "Hello world!"
```

Beispielcode aus den Tutorials von www.geeksforgeeks.org

Programme durchlaufen mehrere Schritte in einer Werkzeugkette (Toolchain), bevor sie von einer CPU ausgeführt werden können:



Erklärungstext siehe nächste Seite

Werkzeugkette
Übersicht 2

- **Quellprogramme (Source Code)** werden mit einem Editor als Textdateien gespeichert und durch den **Compiler** in maschinenlesbaren Code (Objektcode) übersetzt. Größere Programme bestehen in der Regel aus mehreren Dateien (Module), die unabhängig voneinander übersetzt werden.
- Die verschiedenen Objektcode-Module sowie die Bibliotheken (Libraries), die vordefinierte Hilfsfunktionen enthalten, werden durch den **Linker** zu einer **ausführbaren Datei (Executable)** zusammengebunden.
- Wenn der Benutzer das Programm startet, holt der **Lader des Betriebssystems** die ausführbare Datei von der Festplatte und kopiert sie in den Programmspeicher.
- Viele Programmiersprachen, z.B. C/C++, werden vom Compiler direkt in **CPU-spezifischen Maschinencode** übersetzt, die zugehörigen Bibliotheken sind betriebssystemabhängig. **Vorteil**: Schnelle Ausführung, oft auch kleine Speichergröße. **Nachteil**: Neu-Compilieren des Quellprogramms beim Wechsel des Betriebsystems oder der CPU-Familie notwendig (bei kommerziellen Programmen nur durch Hersteller möglich, d.h. Neukauf).
- **Interpreter-Sprachen** wie Java, Javascript oder C#/.NET werden für eine virtuelle Computerarchitektur in CPU- und Betriebssystem-*unabhängigen* Bytecode übersetzt. Beim Programmstart interpretiert diese **Virtuelle Maschine VM** den Bytecode und übersetzt ihn zur Laufzeit in den CPU- und Betriebssystem-*abhängigen* Maschinencode. **Vorteil**: Bytecode-Programm kann (theoretisch) auf jedem Computer ausgeführt werden, sofern es dafür eine VM gibt, **Nachteil**: Sehr langsamer Programmstart, langsame Ausführung, großer Speicherbedarf für VM.
- Code-Editor, Compiler, Linker sowie Debugger (Werkzeug für den Programmtest) sind meist Teil einer integrierten Entwicklungsumgebung (**Integrated Development Environment IDE**), z.B. Visual Studio oder Eclipse.

Datentypen (alle Beispiele in C/C++)

- **Skalare** (einfache) Datentypen für Zahlen, Buchstaben, z.B. `int`, `unsigned long`, `char`. Unterschied: Wertebereich → Länge in Byte
- **Felder (Arrays)** = Feste Anzahl von Elementen des selben Datentyps mit gemeinsamem Namen, die ein- oder mehrdimensional indiziert werden, z.B. Vektor oder Matrix, z.B. `int a[4], long b[3][8], char b[i]`
- **Zeichenketten (Strings)** für Texte (in C als Array von Buchstaben)
- **Strukturierte Datentypen (Struktur, Record)**, d.h. zusammengehöriger Datensatz unterschiedlicher Datentypen, z.B. Adresse aus Zeichenketten für Name, Straße, Ort und Zahlen für Postleitzahl, Hausnummer `struct {char Name[80]; int Plz; char Ort[80]; ...} adresse;`
- **Objekte (Class)** sind Strukturen, die zusätzlich auch Funktionen (**Methoden**) zum Bearbeiten der Datenelemente enthalten können.
- **Referenzen (Pointer)** sind Variablen, die auf andere Variablen zeigen, d.h. die Position eines Datums im Speicher oder in einer Datei angeben
- Moderne Sprachen stellen zusätzlich **abstrakte Datentypen** wie Enumerationen, Listen, Collections, Streams usw. bereit und erlauben **benutzerdefinierte Datentypen**.

Daten können als **Konstanten** oder als **Variablen** definiert werden.

Der Zugriff (**Sichtbarkeit, Scope**) kann im gesamten Programm (**globale Variable, Public**) oder nur innerhalb einer Funktion bzw. Klasse (**lokale Variable, Private**) erlaubt werden.

Programmkonstrukte (alle Beispiele in C/C++)

- Daten werden durch **Operationen** in **Ausdrücken (Expressions)** verknüpft und in der Regel einer Variablen zugewiesen (**Zuweisung, Statement**). Alle Programmiersprachen unterstützen die üblichen arithmetischen und logischen Operationen, z.B. `+, *, >, !, ...`
- Benutzerspezifische „Operationen“ werden durch **Funktionen (Unterprogramme)** realisiert. Funktionen erhalten beim Aufruf eine Liste von Werten (**Funktionsparameter**) und geben **Ergebnisse** zurück, z.B.
  `sortierterVektor = sortiere(vektorA, ASCENDING)`
- Sammlungen häufig benötigter Funktionen werden vom Compiler oder Betriebssystem als **Bibliothek (Library)** zur Verfügung gestellt.

  Entscheidender Unterschied zwischen den Programmiersprachen: Unterschiedlich umfangreiche und leistungsfähige **Bibliotheken**, insbesondere mächtige **Klassenbibliotheken**, die leicht für eigene Anforderungen modifiziert oder erweitert werden können.
- Im Übrigen stellen praktisch alle Sprachen dieselben Konstrukte bereit, wenn auch mit abweichender Syntax.

  Hauptunterschied: **Prozedurale Programmsprachen** wie C, Basic, ..., bei denen Code und Daten getrennt sind, und **Objektorientierte Sprachen** wie C++, Java, C#, ..., die Code und Daten zu leicht erweiterbaren Klassen zusammenfassen können.

Programmkonstrukte (Fortsetzung)

- Üblicherweise erfolgt der **Programmablauf sequentiell**, d.h. Statement für Statement. Der Beginn eines Programms (**Entry Point**) muss speziell markiert werden, z.B. muss jedes C/C++-Programm genau eine Funktion `main()` enthalten.
- Mit Hilfe von **Programmverzweigungen** kann der Ablauf gesteuert werden. Die wichtigsten Verzweigungskonstrukte sind:
  - **Funktionsaufrufe** und Rücksprünge
  - **Bedingte Ausführung** eines Blocks von Anweisungen in Abhängigkeit einer Bedingung (**Condition**), z.B.

    ```
    if (i > 0 ) { a= 1 / i; } else { a = 0;}.
    ```

    Bei einer diskreten Anzahl von möglichen Werten der Bedingung auch als

    ```
    switch ... case ... case ... case ...
    ```
  - **Wiederholte Ausführung** eines Blocks (**Schleife, Loop**) von Anweisungen mit Schleifenzähler oder solange eine Bedingung erfüllt ist, z.B.

    `for (i=0;i<3;i++){...}` oder `while(i<3){...}`
- Neben den Konstrukten, die ausführbaren Code erzeugen, existieren **Verwaltungsbefehle**, mit denen **Kommentare** oder Definitionen aus anderen Dateien (**Header-Files**) eingebunden werden können, z.B.

```
#include <stdio.h>
```

Schrittweise Übersetzung eines Programms durch den Compiler:

- **Lexikalische Analyse (Scanning)**: Wortanalyse $\Rightarrow$ Zerlegung des Quellprogramms in Token (Schlüsselworte wie `main()`, `for()`, $\cdots$), Variablen, Konstanten usw.
- **Syntaktische Analyse (Parsing)**: Grammatikprüfung $\Rightarrow$ Überprüfung des Programmtextes anhand Syntaxregeln, z.B. Erkennung von „Sätzen“ wie `if (bedingung) {`$\cdots$`} else {`$\cdots$`}`.
- **Type Checking**: Überprüfung der Regeln für Datentypen
- **Code Generation**: Umwandlung des Programms in die Maschinensprache der CPU und Optimierung des Codes bezüglich Speicherbedarf und Geschwindigkeit.

Das Ergebnis der ersten beiden Schritte (**Compiler Frontend**) sind die **Symboltabelle** und der **Abstract Syntax Tree**, eine baumartige Struktur, die den Programmfluss abbildet. Wenn man eine Programmiersprache in schematisierter Form (**Backus-Naur-Form**) beschreibt, kann man Werkzeuge für die lexikalische und syntaktische Analyse auch automatisch generieren, d.h. eigene Compiler bauen.

Die Erzeugung und Optimierung der Maschinensprache (**Code Generation and Optimization**) im **Compiler Backend** dagegen ist ein sehr komplexer Vorgang, für den hochspezialisierte Werkzeuge, z.B. GNU C Compiler, Microsoft Visual C++, Intel C Compiler, Sun Java Compiler notwendig sind.

Beispiel:
Ausschnitt aus der **Backus-Naur-Beschreibung** (BNF Reguläre Grammatik) einer Programmiersprache

```
Digit      ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
WholeNumber  ::= Digit [ Digit ]* ;

Letter     ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
               'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
               'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;

LiteralString::= "" [any character except double-quote]+ "" ;
Comparison  ::= '==' | '!=' | '<' | '<=' | '>' | '>=' ;
Separator   ::= ',' | ';' ;
AddSub      ::= '+' | '-' ;
MulDiv      ::= '*' | '/' ;
```

. . .

Aus einer BNF-Beschreibung mit Hilfe regulärer Ausdrücke lässt sich durch das Programms **lex(er)/flex** ein Scanner und mittels des Programms **yacc/bison** ein Parser, d.h. das gesamte Frontend eines Compilers, weitgehend automatisch erzeugen.

Große Datenmengen, die wiederverwendet und/oder schnell durchsucht werden sollen, werden meist separat zur Anwendung gespeichert:

- **Datenbanken**: Daten werden als Strukturen (Records) in miteinander verknüpften Tabellen gespeichert (Relationen).
  - Anbindung über Funktionsaufrufe (Application Programming Interface API) an Datenbank-Server über Internet-Kommunikation (Sockets).
  - Für die Abfrage und Verwaltung der Daten existieren Abfragesprachen wie SQL direkt in der Datenbank, so dass der Aufwand für den SW-Entwickler verringert wird.
  - Bekannte Datenbanken: Oracle DB, SAP Hana, IBM DB2, MySQL/MariaDB, SQLite, Microsoft Access und SQL Server sowie Datenbanken für nicht-strukturierte Daten wie MongoDB.
- Markup-Formate wie **JavaScript Object Notation JSON oder Extensible Markup Language XML**: Daten werden strukturiert in Textdateien gespeichert.
  - Die Datenformate enthalten nicht nur Informationen über die eigentlichen Daten sondern auch zu deren Struktur und Bedeutung.
  - Das Datenformat ist üblicherweise in einem Datenschema (einer Art Grammatik) formal festgelegt, so dass die Daten leicht durch Programm verarbeitet, geprüft und formatiert werden können.
  - Der bekannteste XML-Dialekt ist die **Hypertext Markup Language HTML** für Webseiten, deren Darstellung (Schriftgröße usw.) durch Cascade Style Sheets CSS unabhängig vom Inhalt verändert werden kann.

Strukturierte Datenablage
Beispiele

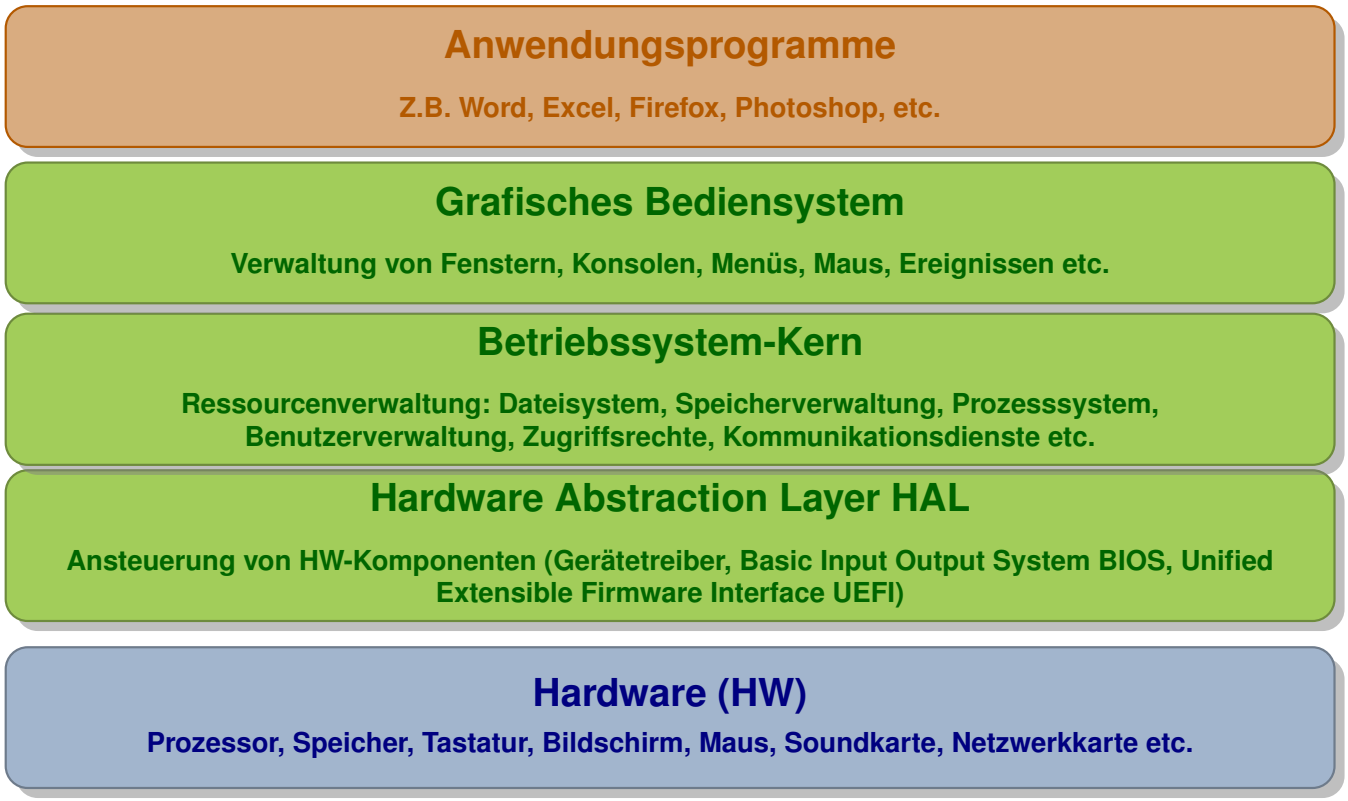
Datei im JSON-Format:
Inhalt besteht aus Key:Value-Paaren, die auch geschachtelt werden können.

```json
{
  "name": "Georg",
  "alter": 47,
  "verheiratet": false,
  "beruf": null,
  "kinder": [
    {
      "name": "Lukas",
      "alter": 19,
      "schulabschluss": "Gymnasium"
    },
    {
      "name": "Lisa",
      "alter": 14,
      "schulabschluss": null
    }
  ]
}
```

```html
<!doctype html>
<html lang="de">
  <head>
    <meta charset="utf-8">
    <title>Schreinerei Meier, Dingenskirchen</title>
    <link rel="stylesheet" href="formate.css">
  </head>
  <body>
    <nav>
    <ul>
      <li>Startseite</li>
      <li><a href="html/preise.html">Unsere Preise</a></li>
      <li><a href="html/bilder.html">Bilder von unseren Produkten</a></li>
    </ul>
    </nav>
    <h1>Willkommen bei der Schreinerei Meier im Internet!</h1>
    <p>Ma illo vostre instruction sed, …</p>
    <h2>Unsere Leistungen:</h2>
    <ul>
      <li>Möbel nach Ihren Wünschen
        <ul>
          <li>Küchenmöbel</li>
          <li>Regale und Schrankwände</li>
          <li>Badezimmermöbel</li>
        </ul>
      </li>
      <li>Haustüren</li>
      <li>Gartenzäune</li>
      <li>Reparaturen</li>
    </ul>
     <p>Web de major tentation primarimente, …</p>
     <p>Libro anglese denomination duo e …</p>
  </body>
</html>
```

Datei im HTML-Format
Inhalt wird über vordefinierte Tags (Elemente in spitzen Klammern) strukturiert.

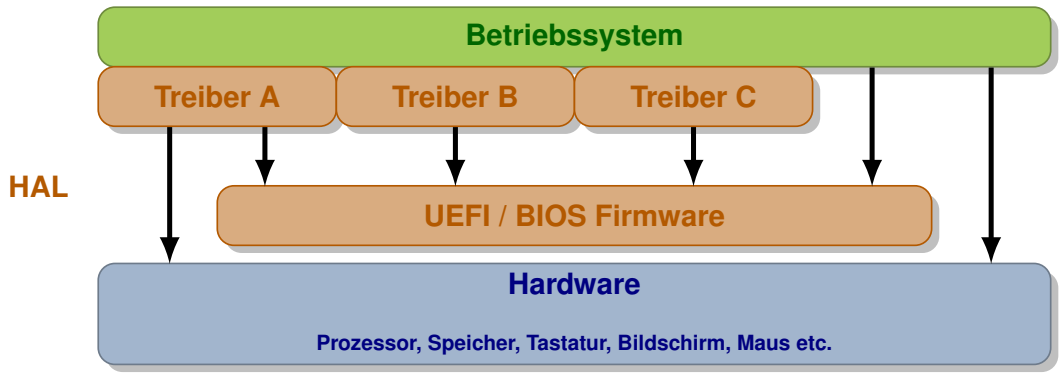
- Computersoftware wird in **Schichten** aufgeteilt. Jede Schicht ist für eine **klar definierte Aufgabenstellung** zuständig. Z.B. sorgt die HAL-Schicht dafür, dass die darüberliegende Software unabhängig von der darunterliegenden Hardware wird.
- Jede Schicht hat eine definierte Schnittstelle (**Application Programming Interface API**) zur darüberliegenden Schicht und eine weitere API zur unteren Schicht.
- Im Idealfall kann eine Schicht ohne Einfluß auf die Nachbarschichten modifiziert oder ausgetauscht werden, solange die Schnittstellen sich nicht ändern.



[35] Quelle Gumm, Sommer: [18]

Entkopplung der darüberliegenden Schichten von den Details der Hardware

- Die rudimentärsten Funktionen, die bereits für das Booten (Starten) des Computers notwendig sind, stehen als **Firmware** oder **Basic Input Output System BIOS** im ROM. Sie dienen dem Initialisieren der CPU und des Speichersystems, zum Ansteuern der Konsole (Textausgabe auf dem Bildschirm und Eingabe von der Tastatur) und zum Einlesen des eigentlichen Betriebssystems vom Massenspeicher (Festplatte). Außerdem führt das BIOS **Funktionstests** z.B. für den Speicher, und die **Erkennung von Hardwarekomponenten** durch.
- Komplexere Funktionen, z.B. zum Ansteuern des Grafikbildschirms oder der Netzwerkkarte, werden als **Geräte-Treiber-Programme** vom Massenspeicher nachgeladen und können bei vielen Betriebssystemen bei Änderungen der Hardware auch zur Laufzeit installiert werden (z.B. für Geräte mit USB-Anschluss).
- HAL-Funktionen können nicht nur von den darüberliegenden Softwareschichten aufgerufen werden, sondern können ihrerseits auch andere Programme unterbrechen und informieren (**Interrupts, Notification Messages, Events**).



[36] Quelle Gumm, Sommer: [18]

Ressourcen-Verwaltung CPU

- Auf modernen Computersysteme laufen in der Regel viele Programme **nebenläufig** (concurrent, parallel, **Multi-Tasking/Multi-Threading/Multi-Processing**), obwohl eine CPU zu einem Zeitpunkt (vereinfacht) nur einen Befehl ausführen kann.
- Eine Hauptaufgabe des Betriebssystems besteht darin, festzulegen, welches Programm welchen Speicherbereich (**Speicherverwaltung**) belegen und zu welchem Zeitpunkt tatsächlich die CPU verwenden darf (**Scheduling**).
- Die Zuordnung eines Programms zur CPU und zum Speicher wird als **Prozessverwaltung** bezeichnet. **Prozess** ist dabei (vereinfacht) ein Synonym für Programm. Sind innerhalb eines Programms Teile vorhanden, die untereinander parallel ablaufen sollen, spricht man auch von **Threads**. Wenn beides gemeint sein kann, wird auch der Begriff **Task** verwendet.
- Für das **Scheduling**, d.h. die Zuteilung der Rechenzeit der CPU auf die einzelnen Tasks, existieren verschiedene Verfahren:
  - **Round-Robin** (veraltet): Alle Tasks laufen der Reihe nach jeweils für eine bestimmte Zeit (Zeitscheiben). Wichtige Tasks bekommen mehr Zeit oder werden öfter berücksichtigt.
  - **Kooperatives Multi-Tasking** (heute selten): Tasks pausieren von Zeit zu Zeit freiwillig, damit andere Tasks ausgeführt werden können.
  - **Präemptives Multi-Tasking** (alle modernen Systeme): Jeder Task wird eine **Priorität** zugeordnet, die Task mit der höchsten Priorität wird ausgeführt. Tasks können für eine bestimmte Zeit pausieren oder auf bestimmte Ereignisse (Events) warten. Wenn eine Task mit höherer Priorität lauffähig wird, wird die laufende Task unterbrochen (preempt=verdrängt). Prioritäten können fest (statisch) oder zur Laufzeit veränderlich (dynamisch) sein.

- Je nach Einsatzfall kann das Scheduling unterschiedliche Schwerpunkte setzen. Endbenutzersysteme wie Windows und Linux optimieren die **Reaktionszeit**, d.h. sie versuchen, möglichst schnell auf Benutzereingaben zu reagieren und geben dem Programm im gerade vom Benutzer ausgewählten Fenster eine hohe Priorität.
- Die Server-Varianten von Windows und Linux optimieren den **Durchsatz**, d.h. sie versuchen alle Programme reihum möglichst oft zu bedienen.
- Echtzeit-Betriebssysteme (**Real-Time-Operating-System RTOS**) versuchen, einerseits die Reaktionszeit auf Ereignisse (Events, Interrupts) zu minimieren, andererseits Zeitschranken (**Deadline**), d.h. maximale Ausführungszeiten für Tasks und die rechtzeitige Bereitstellung von Ergebnissen zu gewährleisten.
- Damit Tasks Einfluss auf den Scheduler nehmen können und sich untereinander synchronisieren und Daten austauschen zu können, stellen Betriebsysteme entsprechende Dienste für die **Interprozess-Kommunikation** zur Verfügung.

Ressourcen-Verwaltung Speicher

- Jedem Programm werden vom Betriebssystem bestimmte Speicherbereiche exklusiv zugewiesen. Die Verwaltung erfolgt in der Regel getrennt für den **Code**, die **globalen Daten**, den **Stack** für lokale Variablen und den **Heap** für Daten/Objekte, die während des Programmlaufs erzeugt und wieder frei gegeben werden.
- Moderne CPUs verfügen über Hardware-Überwachungsmechanismen (**Memory Management**), die sicherstellen können, dass ein Programm nicht auf den Speicherbereich eines anderen Programms zugreifen kann (**Zugriffsschutz**).
- Falls der tatsächlich vorhandene Speicher nicht für alle gestarteten Programme ausreichen sollte, kopieren moderne Betriebssysteme den Speicherbereich derjenigen Programme, die im Augenblick ohnehin nicht die CPU verwenden dürfen, auf den Massenspeicher (**Virtueller Speicher**).

Verwaltung des Dateisystems

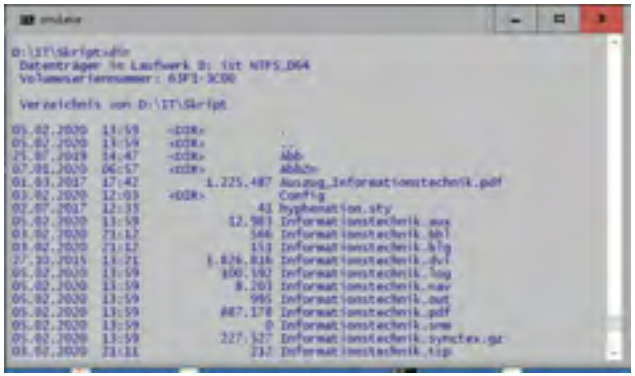
- Massenspeicher wie Festplatten speichern Daten in Blöcken von typ. 512 oder 1024 Byte. Eine **Datei (File)** ist ein zusammengehöriger Satz solcher Blöcke, die beliebig auf dem Massenspeicher verteilt sein können.
- Da eine Festplatte sehr viele Dateien enthalten kann, werden als übergeordnete Strukturen **Verzeichnisse (Directories)** eingeführt, die in einer **Baumstruktur** ausgehend von einem Wurzelverzeichnis (bei Windows c:\) organisiert werden.
- Das Betriebsystem verwaltet das **Dateisystem** mit Hilfe von Listen, die für jede Datei den **Dateinamen**, die Lage im Verzeichnisbaum (**Dateipfad**) sowie die Liste der zugehörigen Blöcke und deren Reihenfolge enthält. Außerdem verwaltet das Betriebssystem eine Liste aller freien Blöcke auf dem Massenspeicher, so dass schnell Platz für neue Dateien gefunden werden kann.
- Moderne Dateisysteme legen außerdem **Zugriffsrechte (Access Rights)** für Dateien fest. D.h. welche Benutzer auf eine Datei zugreifen dürfen und ob eine Datei nur gelesen oder auch geschrieben und gelöscht werden darf.
- Praktisch alle Dateisysteme unterstützen eine Reihe von elementaren Funktionen:
  - `fopen()`: Öffnen bzw. Neuanlegen einer Datei
  - `fread()`, `fwrite()`: Lesen/Schreiben einer Datei
  - `fclose()`: Schließen einer Datei
  - `fseek()`: Dateien können nur sequentiell, d.h. Byte für Byte, gelesen werden. Mit `fseek()` kann man sich in einer Datei vor- und zurückbewegen.
- In der Praxis gibt es verschiedene Dateisysteme, z.B. NTFS und VFAT für Windows oder EXT4 für Linux, die sich in den Verwaltungsstrukturen, der Geschwindigkeit und der Fähigkeit zur Fehlererkennung und -korrektur unterscheiden.

Kommunikationsdienste für Computernetze

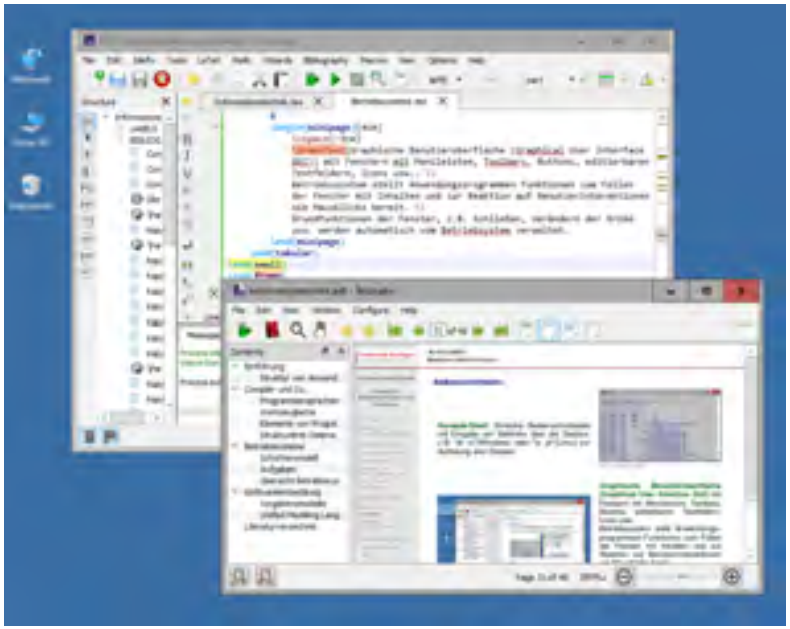
- Moderne Computersysteme sind vernetzt (**Netz, engl. Network**).
- Dazu ist sowohl eine standardisierte physikalische Verbindung, z.B. **Ethernet** (Kabelgebundenes lokales Netz), **Wireless Local Area Network WLAN** (auch WiFi, Kurzstrecken-Funknetz), **Digital Subscriber Line DSL** („Telefon“festnetz) oder **GSM/UMTS/LTE** (Mobilfunknetz) als auch ein standardisiertes Übertragungsprotokoll notwendig.
- Das **Übertragungsprotokoll** legt dabei fest, wie Verbindungen aufgebaut und werden und in welchem Format Anfragen und Antworten ausgetauscht werden.
- Übertragungsprotokolle sind ebenfalls in Schichten unterteilt (**Protokoll-Stack**), heute kommen meist die sogenannten **Internetprotokolle (**HTTP, TCP, IP usw.) zum Einsatz. Das Schichtenmodell stellt sicher, dass die einzelnen Schichten ausgetauscht werden können, ohne die anderen Schichten zu ändern, z.B. ist es für den Internet-Browser gleichgültig, ob der Computer über Ethernet oder ein WLAN angeschlossen ist.
- Computer, die anderen Systemen ihre Dienste anbieten, werden als **Server** bezeichnet. Computer, die diese Dienste nutzen, als **Client**.
- Um Kommunikationsdienste einfach in Programmen verwenden zu können, verwendet man häufig **Sockets**, eine standardisierte Programmierschnittstelle.

Bedienschnittstelle (User Interface)

Konsole/Shell: Einfache Bedienschnittstelle mit Eingabe von Befehlen über die Tastatur, z.B. "dir /s"(Windows) oder "ls -al"(Linux) zur Auflistung aller Dateien.



DOS Shell CMD



Windows Desktop und Anwendung

Graphische Benutzeroberfläche (Graphical User Interface GUI) mit Fenstern mit Menüleisten, Toolbars, Buttons, editierbaren Textfeldern, Icons, ...
Betriebssystem stellt Anwendungsprogrammen Funktionen zum Füllen der Fenster mit Inhalten und zur Reaktion auf Benutzeraktionen wie Mausklicks bereit.
Grundfunktionen der Fenster, z.B. Schließen, Verändern der Größe, werden automatisch vom Betriebsystem verwaltet.

Gestaltung der Bedienschnittstelle (**Usability**) ist ein wichtiger Schritt beim SW-Entwurf.

UNIX-artige Betriebssysteme: Linux, Mac OS

- Entstanden in den 1970er Jahren in den Bell-Labs zusammen mit der Programmiersprache C. War aber kein kommerzieller Erfolg und wurde daher als **Open Source** freigeben.
- Sehr beliebt für die Lehre an Hochschulen und bei kleineren Computer-Herstellern, weil es das einzige im Source Code frei verfügbare größere Betriebssystem war.
- Initialzündung für viele heute weit verbreitete Open-Source-Projekte, z.B. GNU C-Compiler, Apache Webserver, Open Office u.a.
- Kommerzielle Weiterentwicklungen mit proprietären Erweiterungen, z.B. Sun Solaris, IBM AIX.
- Von **Linus Torvalds** zunächst als Studentenprojekt in den 1990er Jahren auf PCs portiert, als **LINUX weit verbreitet als Web- und File-Server**.
- **Multi-Tasking** (mehrere Programme) und **Multi-User**-System (mehrere Benutzer gleichzeitig)
- Wildwuchs durch eine Vielzahl von **Benutzeroberflächen (KDE, Gnome, ...)** und eine Vielzahl von Herstellern/Open Source Gruppen, die Linux mit frei verfügbaren Anwendungsprogrammen zu **Distributionen, z.B. Fedora, Ubuntu, Debian, Suse, ...** bündeln, die nicht miteinander kompatibel sind. Daher geringe Verbreitung bei Endanwendern.
- Eine Unix-Variante ist auch Basis des Betriebssystems **Apple Mac OS X**, das aber eine proprietäre graphische Benutzeroberfläche hat.

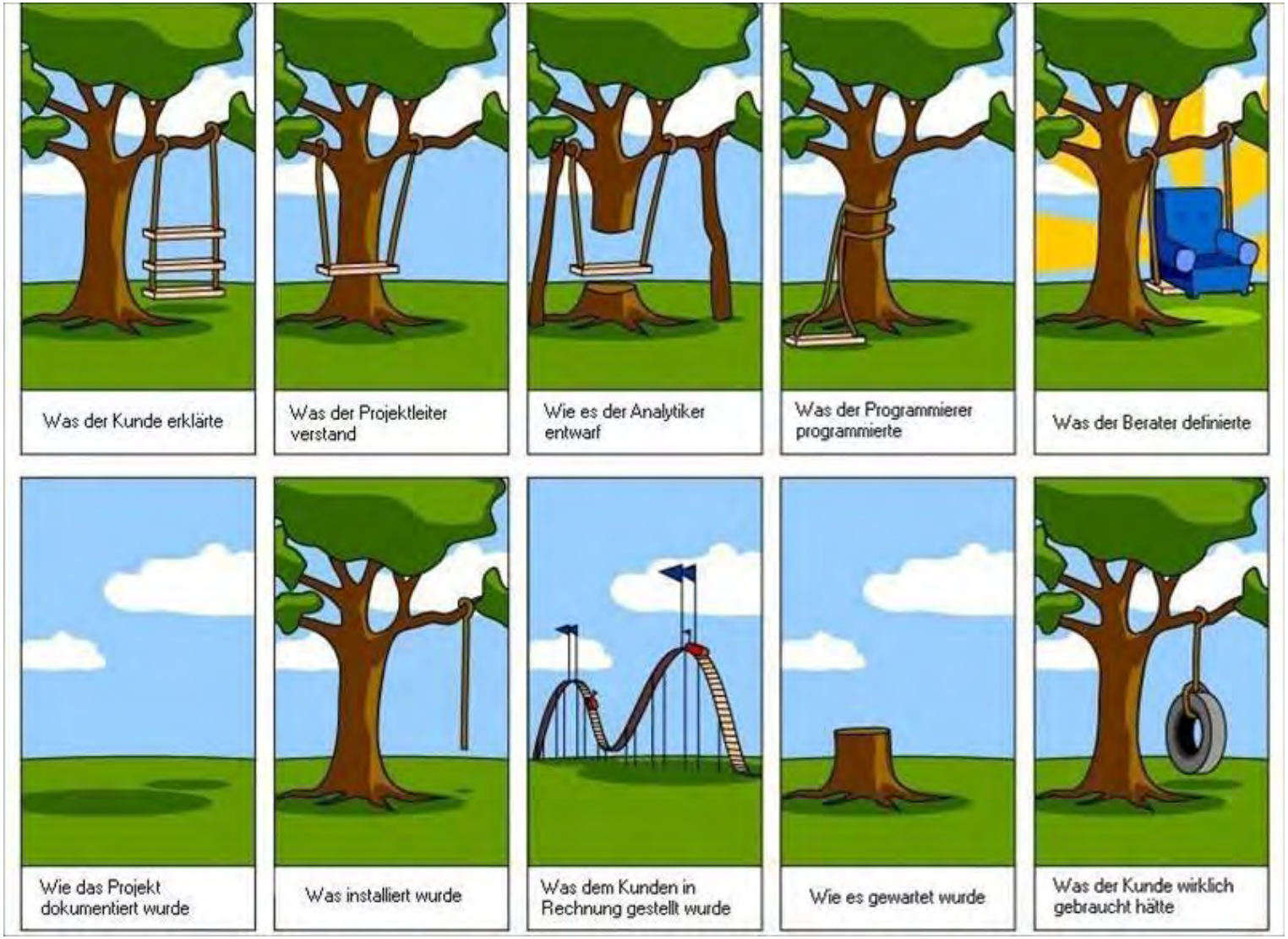
- Nachfolger des rein textbasierten Single-Tasking-/Single-User-Systems MS-DOS (Microsoft Disk Operating System) der ersten PC-Generationen in den 1980 Jahren
- Durchbruch um 1990 als 16-bit-System Windows 3, aufwärtskompatible 32-bit-Version NT/XP/Vista mit Unix-ähnlichen Konzepten, wird als 64-bit-Version **Windows** 7/8/10/... laufend weiterentwickelt.
- **Multi-Tasking**-System, in der Desktop-Version aber **kein echter Multi-User-Betrieb**, da mehrere Benutzer nicht gleichzeitig möglich.
- **Sehr hoher Marktanteil** (ca. 80% im PC-Markt, Quelle Statista.de, Nov. 2015) durch Microsoft-Lizenzmodell (Low-Cost-Lizenz für PC-Hersteller im Bundle mit jedem PC/Notebook sowie Microsoft-Office vorinstalliert).
- Ständig wachsende Bedeutung der Versionen für Server, Cluster, Cloud sowie High-Performance-Computing (Marktanteil bei Servern ca. 50%, Quelle Heise.de, 2014), ergänzt durch leistungsfähige Server-Anwendungen (Microsoft SQL Datenbank-Server, Microsoft Exchange-Mail-Server, Web-Server).
- **Erfolgsrezept**:
  - Aufwärtskompatible Versionen, d.h. Alt-Softwareanwendungen laufen auf neuen Versionen. Enge Abstimmung mit Intel-x86-CPU-Entwicklung.
  - Durchgängiges Bedienkonzept, wenig Wildwuchs (im Vergleich zu Linux)
  - Stark verbreitete Anwendungssoftware (Microsoft Office, Outlook)
  - Hoher Marktanteil führt zu großem Angebot von Fremdsoftware. Proprietäre Programmierschnittstellen (API) erhöhen den Aufwand für Softwarehersteller, ihre Anwendungssoftware auch für andere Betriebssysteme anzubieten.

Smartphone- und Tablet-Betriebssysteme

- **Google's Android**: Verwendet Linux als Betriebssystemkern, hat aber eine proprietäre Benutzeroberfläche. Anwendungsprogramme (Apps) werden in Java programmiert, verwenden aber proprietäre Bibliotheken.
- **Apple's iOS**: Verwendet eine vereinfachte Version von Apple's Mac OS X, eine Unix-Variante, als Betriebssystemkern. Anwendungsprogramme (Apps) werden in Objective C, einer C++-Variante, heute auch in Swift mit jeweils proprietären Bibliotheken programmiert.

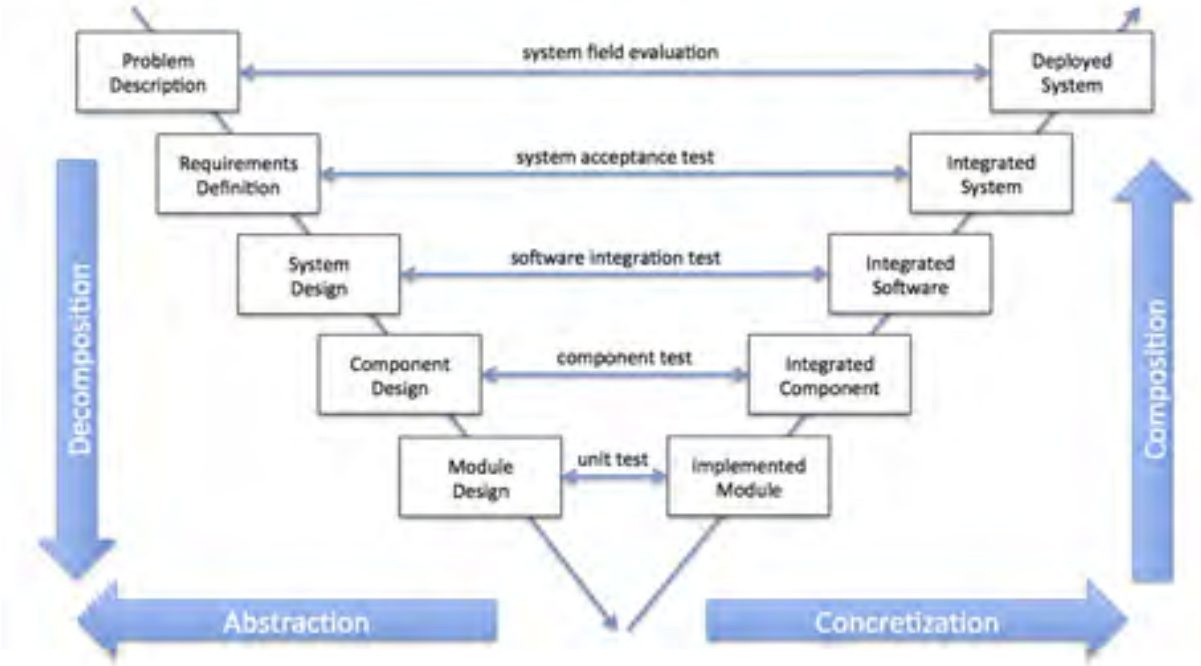
Echtzeit-Betriebssysteme

- Für Embedded Systems mit hohen Zeit- und Sicherheitsanforderungen existiert eine Vielzahl von Betriebssystemen, die einen deutlich kleineren Ressourcenbedarf (Anforderungen an CPU-Rechenleistung und Speicherplatz) haben als Windows oder Linux.
- **Echzeit (Real-Time)**: System, das die Einhaltung von Zeitbedingungen garantiert, z.B. die maximale Laufzeit eines Programms oder die maximale Reaktionszeit, z.B. dem Betätigen des Bremspedals und dem Ansprechen der Bremse bei einem Auto.
- Beispiele:
  - **OSEK/AUTOSAR OS**: Standard-Betriebssystem in der Automotive-Welt, z.B. für Motorsteuerungen oder ABS/ESP in Fahrzeugen.
  - **RT (Real Time) Linux**: Linux-Variante mit Echtzeiteigenschaften für die Automatisierungstechnik
  - . . .



Strukturierte Entwicklung nach dem **V-Modell** mit den Hauptschritten:

- Problemanalyse und **Aufstellen der Anforderungen (Requirements)**
  Kundenvorgaben (Lastenheft) $\Rightarrow$ (Entwicklungs)-Spezifikation
- **Entwurf (Design) der System-Architektur** und Zerlegen in Komponenten und Module
- **Entwickeln (Implementation) der Systemkomponenten**.



- **Top-Down-Vorgehen beim Entwickeln** (linker Zweig)
  Dieser linke Zweig entspricht dem älteren **Wasserfallmodell**, bei dem das Testen nicht explizit im Modell dargestellt wurde.
- **Bottom-Up-Vorgehen beim Integrieren und Testen** (rechter Zweig)

Die Bedeutung der Qualität von Softwaresystemen wurde lange Zeit unterschätzt:

- Heute testet man auf allen Ebenen des Entwicklungsprozesses und definiert beim Entwurf der Komponenten und Module die Testfälle gleich mit (**Design to Test**).
- Als Faustformel für den Aufwand gilt heute:
  1/3 für Anforderungsanalyse und Entwurf der Systemarchitektur
  1/3 für die Implementierung 1/3 für den Test.

Testen soll folgende Fragen klären:

- **Verifikation**: **"Do we build the system right?"** (Bauen wir das System richtig?)
  - Arbeitet das System entsprechend der Spezifikation? ⇒ **statisches und dynamisches Testen** V-Modell untere Ebenen
  - Werden alle Schritte des V-Modells in der richtigen Reihenfolge ausgeführt und dokumentiert? Setzen wir die richtigen Methoden und Werkzeuge ein?
- **Validierung**: **"Do we build the right system?"** (Bauen wir das richtige System?)
  - Sind die Anforderungen sinnvoll, vollständig und widerspruchsfrei?
  - ⇒ Kundenreviews, Protoyping, **Systemerprobung** V-Modell obere Ebene

Folgende Verfahren werden verwendet:

- **Dynamische Codeanalyse**: Ausführen des Programms und Überprüfen des Programmablaufs und der Ergebnisse (**Debugging**)
- **Statische Codeanalyse**: Überprüfen des Programms auf syntaktische Korrektheit und typische Fehlerquellen, z.B. nicht initialisierte Variable, Überläufe, ...
- **Review**: Diskussion der Anforderungen und des Designs, Begutachtung des Quellcodes (Code Inspection) und der Dokumente im Entwicklerteam und mit Kunden

In der Praxis wird selten geradlinig entwickelt, sondern iterativ und rekursiv, d.h. dass zum vorigen Schritt des V-Modells zurückgegangen oder eine Schrittfolge mehrfach durchlaufen werden muss. Das **Spiralmodell** nach Boehm stellt dies anschaulich dar:

B.W. Boehm: A Spiral Model of Software Development and Enhancement. IEEE Computer 5/1988, S. 61-72

Eine Extremform des Konzepts stellt die **Agile Entwicklung SCRUM** („Gedränge“) dar:

- Inkrementelle Entwicklung mit **extrem vielen, kurzen Schleifen** (2-4 Wochen)
- Ständige Verfügbarkeit eines funktionsfähigen Prototyps durch laufende Integration neuer/modifizierter Softwaremodule (**Continuous Integration**)
- Ständiges Überprüfen durch **automatisiertes Testen**

Vorgehensmodelle und Programmiersprachen sollten folgende Grundprinzipien berücksichtigen:

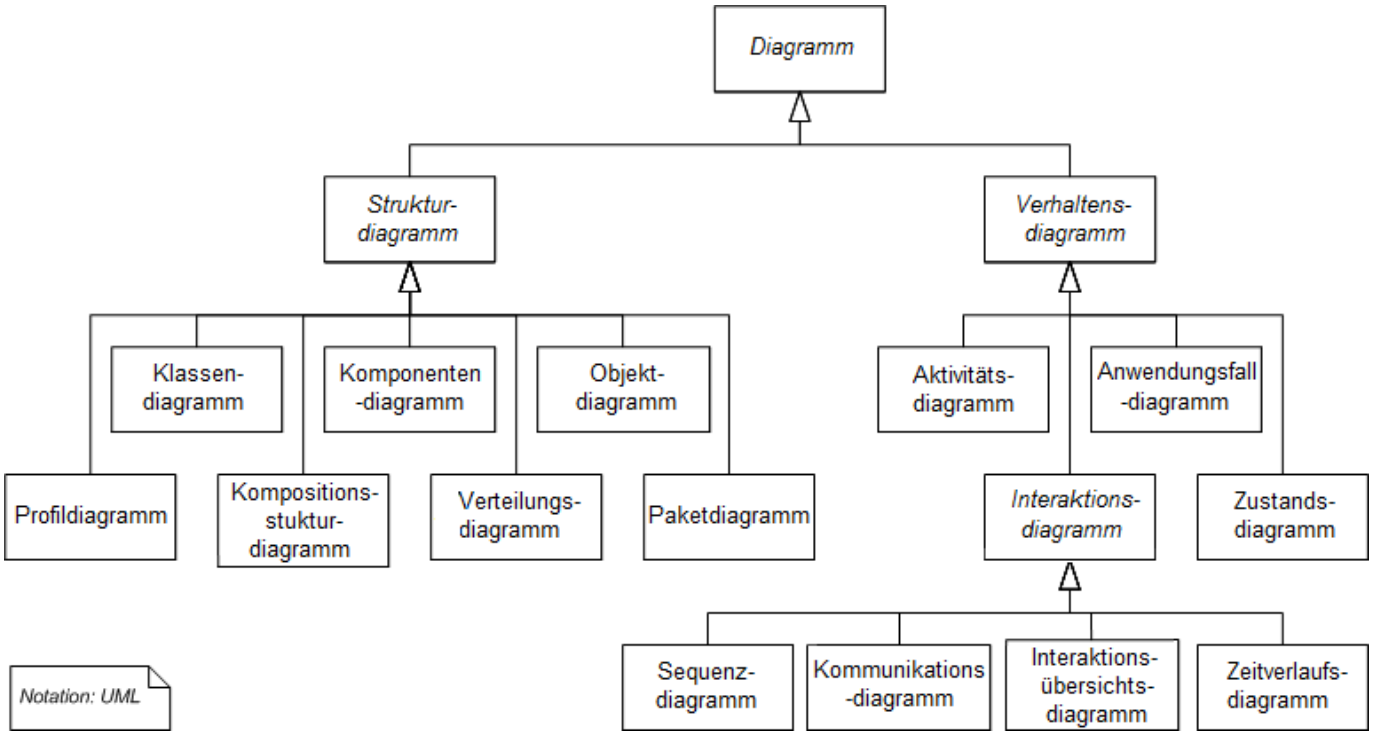
- **Separation of Concerns** (Concerns, hier: Aufgaben, Zuständigkeiten)
  - Trennung der Aspekte **was** entwickelt wird und **wie** entwickelt wird.
  - Trennung der **von außen sichtbaren Schnittstelle** und der **nicht sichtbaren Implementierung** (Interface and Implementation) $\Rightarrow$ **Information Hiding**.
  - Trennung der **Spezifikation** (Spezifikation $=$ Modell) und der **Realisierung** (Implementierung $=$ Produkt).
- **Schwache Kopplung** (Low Coupling) und **starke Kohäsion** (High Cohesion)
  - Schnittstellen zwischen den Teilen sollen so schlank wie möglich sein.
  - Ähnliche Konzepte sollen zusammengefasst werden.
- **Decomposition, Concretization, Composition, Validation**
  - Probleme werden in kleinere Einheiten aufgeteilt (**Decomposition**), die individuell einfacher zu lösen sind.
  - Die kleinen Einheiten werden realisiert (**Concretization**)
  - Die implementierten kleinen Einheiten werden zu größeren Einheiten zusammengesetzt (**Composition**)
  - Die konkrete Komposition wird mit der ursprünglichen Idee verglichen (**Validation**).

- Die Entwicklung komplexer Systeme erfolgt in Zusammenarbeit zwischen den Gruppen **Anwender** (User), Systemanalysten/**Systemarchitekten** und technischen Spezialisten (**Software- und Hardwareingenieure**).
- Der Austausch von Informationen erfolgt in Gesprächen und Dokumenten. Sprache ist aber keineswegs eindeutig, d.h. es kommt zu Missverständnissen und Widersprüchen.
- Die Umsetzung der Anforderungen und des Systementwurfs in die Implementierung (Programmcode) erfolgt bei Software manuell (**„Programmieren“**). Das ist fehlerträchtig und für den Anwender (aber auch für Fachkollegen) oft schwer nachvollziehbar.
- **Klassische Ingenieurdisziplinen**, z.B. Architekten oder Maschinenbauer, **beschreiben** ihre **Produkte graphisch**, d.h. durch Zeichnungen statt Sprache, weil dies kompakter, übersichtlicher und für andere leichter verständlich ist. Die graphische Darstellung ist meist durch Normung standardisiert.
- Mit der **Unified Modeling Language UML** wurde in den 1990er Jahren eine **grafische Sprache** zur **Spezifikation**, **Konstruktion** und **Dokumentation** von Softwaresystemen eingeführt.
- UML-Modelle sind sowohl für Menschen als auch für Computer lesbar. Mit Hilfe von geeigneten Werkzeugen kann ein UML-Modell auf Vollständigkeit und Widerspruchsfreiheit geprüft, simuliert (**Rapid Prototyping**) und schrittweise in eine Implementierung umgesetzt werden. Z.T. ist dies sogar automatisch möglich (**Automatische Codegenerierung**).

[37] Quelle: *Einführung in UML*, Dino Ahr, Uni Heidelberg

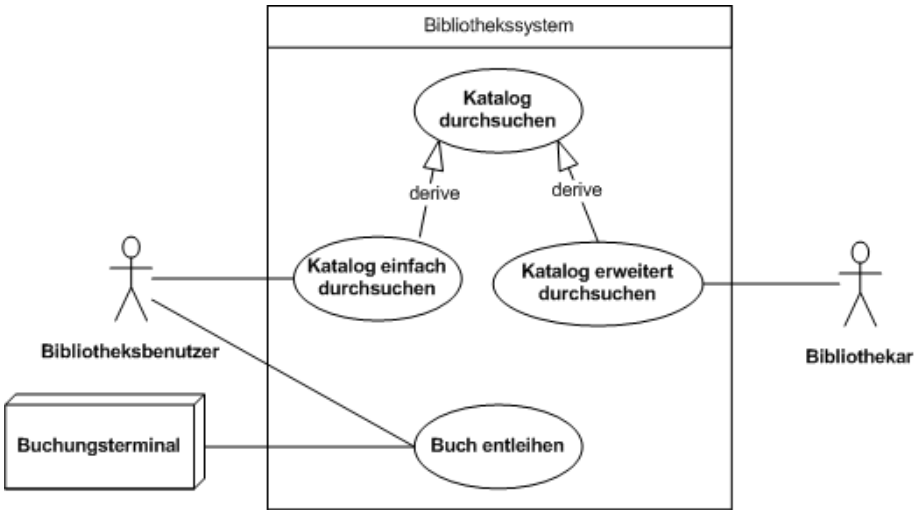
UML-Modelle beschreiben verschiedene Aspekte eines Systems:

- Anwendungsszenarien: **Use Case** **Bild Bibliothek**
- Systemarchitektur (Struktur): **Komponentendiagramm, Klassendiagramm, ...**
- Verhalten (Funktion) des Systems: **Aktivitätsdiagramm, Zustandsdiagramm, Sequenzdiagramm, ... Bild Familienstand**
- Lücken hat UML beim Design von Benutzeroberflächen oder von Signal-/Datenverarbeitungsalgorithmen. Dazu wird UML durch weitere Methoden und Werkzeuge ergänzt, z.B. GUI Designer oder Matlab/Simulink.

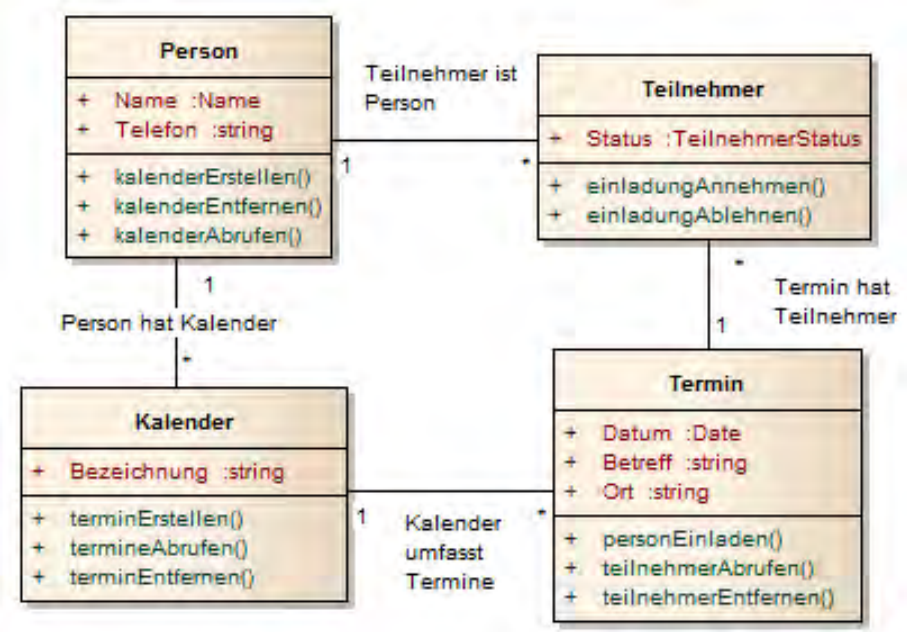


Quelle: Unified Modeling Language, Wikpedia Feb. 2016

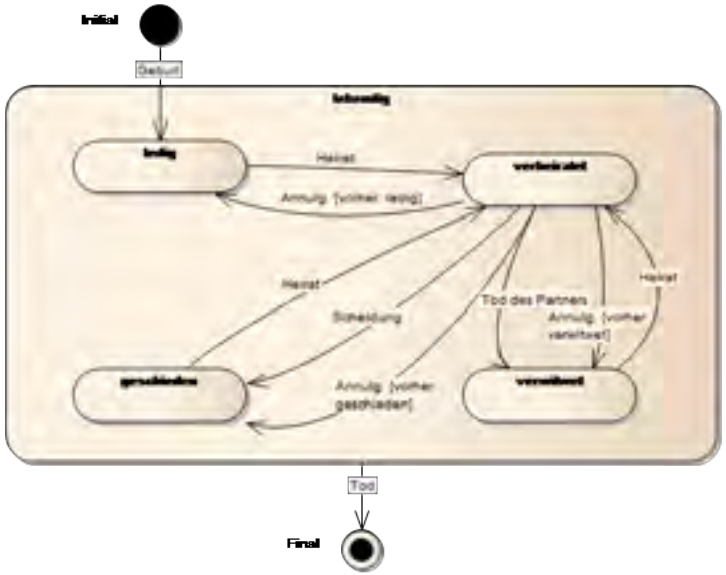
Use Case Diagramm:



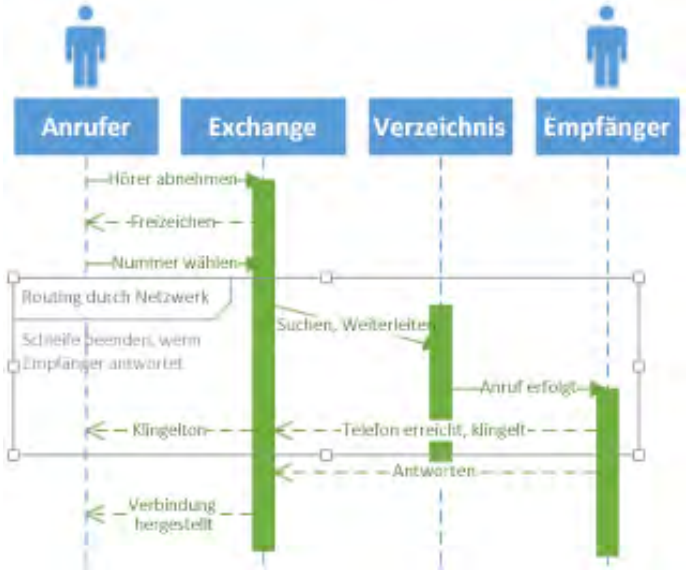
Klassendiagramm:



Quelle: www.sparxsystems.de



Zustandsdiagramm:



Sequenzdiagramm:

VORLESUNG INFORMATIONSTECHNIK

Prof. Dr.-Ing. R. Marchthaler

Prof. Dr.-Ing. W. Zimmermann

Webseite:
https://www.hs-esslingen.de/personen/werner-zimmermann

Moodle-Kurs:
https://moodle.hs-esslingen.de/moodle/course/view.php?id=29265

Hochschule Esslingen
Fakultät Informationstechnik

Sommer 2020

Version: 4. Juli 2021

LITERATURVERZEICHNIS

LITERATURVERZEICHNIS I

[1] WÖRTERBUCH DUDEN:
Informatik.
Online aufgerufen bei: http://www.duden.de/rechtschreibung/Informatik, 16. Juli 2019

[2] WÖRTERBUCH DUDEN:
Informationstechnik.
Online aufgerufen bei: http://www.duden.de /rechtschreibung/Informationstechnik, 16. Juli 2019

[3] VDE ITG-VORSTAND:
ITG-Definition Informationstechnik mit Bezug auf die ITG-Fachbereiche.
Online aufgerufen bei: http://www.vde.com/de/fg/ITG/Ueber%20Uns
/Documents/Definition_Informationstechnik%5B1%5D.pdf, 22. Mai 2013

[4] ARBEITS-ABC:
Fachberater/in für Softwaretechnik.
Online aufgerufen bei: https://www.ausbildung.de/berufe/themen/it/, 16. Juli 2019

[5] E-BUSINESS ; UNTERNEHMENSBERATUNG:
Beratung für Unternehmensberater. Die schlechten Webseiten der Consulting-Experten.
Online aufgerufen bei: http://www.e-business - unternehmensberatung . com / blog /
beratung-fuer-unternehmensberater - die - schlechten-webseiten-der-consulting-experten/, 16. Juli 2019

[6] GESELLSCHAFT FÜR TECHNOLOGIEBERATUNG UND SYSTEMENTWICKLUNG MBH:
Überblick SAP-System.
Online aufgerufen bei: http://www.tse.de /papiere/sap/ueberblick.html, 16. Juli 2019

[7] SAP:
Gemeiname Geschäftsprozesse.
Online aufgerufen bei: http://de.sap.info /gemeiname-geschaftsprozesse/1927, 23. Mai 2013

[8] KIT:
Industrierobotik.
Online aufgerufen bei: http://www.kit.edu/besuchen/11601.php/, 16. Juli 2019

[9] AUTOMOTOPORTAL:
3D Navigation - Google Earth.
Online aufgerufen bei: http: // www.automotoportal . com / photos / Volkswagen _ and _ Google _ develop _
revolutionary _ navigation _ system / 3, 16. Juli 2019

[10] ROBERT BOSCH AND AUTONOMES FAHREN, ROBOTER AUTOS - MAGAZIN FÜR AUTONOME AUTOS U. AUTONOME SYSTEME:
Bosch und die Fahrassistenten, Radarsensorik.
Online aufgerufen bei: http://www.autonomes-fahren.de/bosch-die-fahrassistenten-radarsensorik/, 16. Juli 2019

[11] WIKIPEDIA ONLINE-ENZYKLOPÄDIE:
Zuse Z3.
Online aufgerufen bei: https://de. wikipedia.org /wiki/Zuse_Z3, 16. Juli 2019

[12] WIKIPEDIA ONLINE-ENZYKLOPÄDIE:
Analytical Engine.
Online aufgerufen bei: https:// de. wikipedia.org/wiki/Analytical_Engine, 16. Juli 2019

[13] WIKIPEDIA ONLINE-ENZYKLOPÄDIE:
Wilhelm Schickard.
Online aufgerufen bei: https:// de. wikipedia.org/wiki/Wilhelm_Schickard, 16. Juli 2019

[14] WIKIPEDIA ONLINE-ENZYKLOPÄDIE:
Blue Gene.
Online aufgerufen bei: https:// de.wikipedia.org/ wiki/Blue_Gene#Blue_Gene.2FP, 16. Juli 2019

[15] WIKIPEDIA ONLINE-ENZYKLOPÄDIE:
IBM Personal Computer.
Online aufgerufen bei: http://de. wikipedia.org/wiki/IBM_PC, 16. Juli 2019

[16] COMPUTER MODELL KATALOG:
MARK I, II, III, IV.
Online aufgerufen bei: http:// computer-modell-katalog.de/mark.htm, 16. Juli 2019

[17] DOMINIKA SZOPE:
zkm Computergeschichte.
Online aufgerufen bei: http://www01.zkm.de/algorithmische-revolution/, 16. Juli 2019

[18] HEINZ-PETER GUMM ; MANFRED SOMMER:
Einführung in die Informatik.
10. Auflage.
München : Oldenbourg Verlag, 2012. –
ISBN 978–3–486–70641–3

LITERATURVERZEICHNIS III

[19] GI - GESELLSCHAFT FÜR INFORMATIK:
Bedeutende Informatikpersönlichkeiten.
Online aufgerufen bei: https://gi.de/persoenlichkeiten/, 16. Juli 2019

[20] GI - GESELLSCHAFT FÜR INFORMATIK:
PCHilfen.net.
Online aufgerufen bei: http:// pchilfen.net/elek/elektrik.php, 15. Juni 2013

[21] ROLAND HELLMANN:
Rechnerarchitektur - Einführung in den Aufbau moderner Computer.
2. Auflage.
München : Oldenbourg Verlag, 2016. –
ISBN 978–3110496659