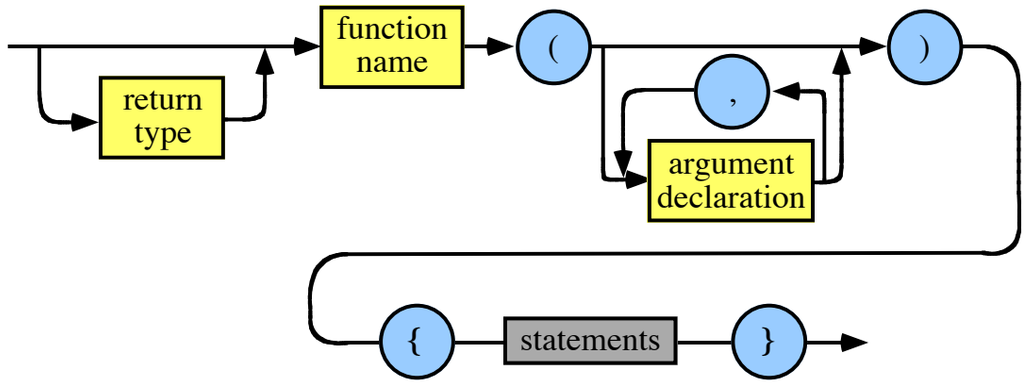
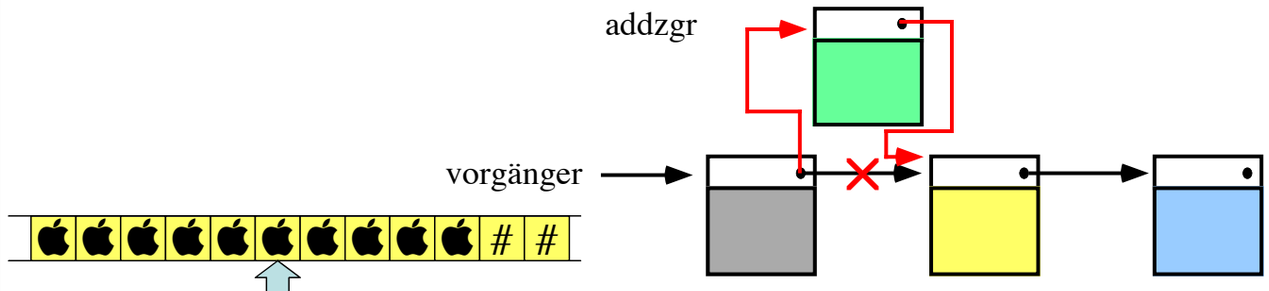
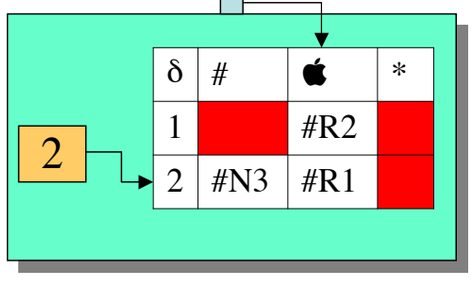
KONRAD FROITZHEIM

Grundlagen der Informatik







Technische Universität Bergakademie Freiberg

KAPITEL 1

Gebiet und Vorlesung

- Informatik - die Wissenschaft des Kommunikationszeitalters
- Vision
- Gebiete
- Formales zur Vorlesung
- End User License Agreement
- Literatur
- Inhaltsübersicht

I think there is a world market for maybe five computers.
[T.J. Watson, Chairman, IBM, 1943]

Informatik = Information + Automatik

Die Wissenschaft von der automatischen Verarbeitung von Informationen

- Information
  - lateinisch: Auskunft, Belehrung, Benachrichtigung
  - umgangssprachlich: Kenntnis über Sachverhalte und Vorgänge
  - inhaltliche Bedeutung eines Sachverhaltes (Semantik)
- Daten
  - Informationen, die zum Zweck der Verarbeitung eine bestimmte, vorher vereinbarte Darstellungsform (Syntax) haben
  - kontinuierliche Funktionen: "analoge Daten"
  - oder Zeichen: "digitale Daten"
- Automatik (griech.): selbsttätiger Ablauf
- Rechnen?

- Aufgabe der Informatik
  - Erforschung der grundsätzlichen Verfahrensweise der Informationsverarbeitung
  - Entwicklung allgemeiner Methoden
  - Anwendung in den verschiedensten Bereichen
- Informatik als Strukturwissenschaft
  - Struktur und Eigenschaften von Informationen
  - Struktur und Eigenschaften von Informationsverarbeitungssystemen
  - heute elektronische Datenverarbeitungsanlagen
  - Methoden: Analyse, formale Beschreibung, Konstruktion
- Informatik ist Ingenieurwissenschaft
  - Computer und Zubehör konstruieren
  - Verfahren und Algorithmen entwerfen
  - Programmieren
  - Pflegen und warten
  - aber: D. Knuth: The Art of Computer Programming

Vision

Vannevar Bush, 1945: Memex - a device in which one stores all his books, records, and communications, and which is mechanized so that it can be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.

- Memex
  - Internet + Web
  - Apple, 1988: Knowledge Navigator
- Holodeck
  - ca. 1990: Vision in Star Treck
  - Neal Stephenson, 2000: Snowcrash
  - ab 2004: 3D-Welten WoW, SecondLife, …
  - immersive 3D-Interfaces noch teuer -> X-Site
- Turing Test und künstliche Intelligenz
  - Google
  - IBM: Deep Blue
  - IBM: Watson
  - Robocup

Alan Turing, 1950:At the and of the century, the use of words and general educated opinion will have changed so much that one will be able to speak of "machines thinking" without expecting to be contradicted.

Gebiete der Informatik

- Theoretische Informatik
  - Berechenbarkeit
  - Aufwand für Berechnung
  - Beweisbarkeit von Programmen
  - Entscheidbarkeit
  - formale Methoden
  - Sprach-Klassen
- Algorithmen und Datenstrukturen
  - Formeln, Ablauf, Sequentialisierung
  - Daten darstellen und speichern
  - Zahldarstellung

- Programmieren
  - vom Problem zum Programm
  - ingenieurmässiges Gestalten (kreativ und systematisch)
  - Software Engineering
  - Programmiersprachen und Compiler
  - Objekte, Patterns und Frameworks
- Betriebssysteme
  - Abstraktion von Geräten und Computer-Teilen ("APIs")
  - Koordination der Programme
  - gerechte Verteilung der Ressourcen (Zeit, Platz, …)
  - Schutz und Sicherheit
- Rechnerarchitektur
  - vom Transistor zum Maschinenbefehl
  - Speicher, Festplatte, Eingabe, Ausgabe, …
  - Parallelrechner

There is no substitute for speed [E. McCreight, 1998]

- Technische Kommunikation
  - Signalübertragung und -Vermittlung
  - Protokolle
  - Kompression
  - ISDN, Rechnernetze, Internet, …
- Anwendungen
  - Bürosoftware
  - Datenbanken und Informationssysteme
  - Multimedia
  - Kommunikationsdienste (Telefon, TV, WWW, …)
  - CAD - Computer Aided Design
  - Simulation
  - Spiele
  - Steuerung und Robotik
  - Künstliche Intelligenz
  - Neuroinformatik

Formales

Termine:

Vorlesung: Dienstag 09:15 - 10:45, DBI-TZ
Dienstag 16:00 - 17:30, DBI-TZ

Dramatis Personae:

Prof. Dr. Konrad Froitzheim: frz@tu-freiberg.de
Professur Betriebssysteme und Kommunikationstechnologien

Dipl.-Ing. Mathias Buhr
MSc. BNC Frank Gommlich
Dipl.-Ing. Georg Heyne

Vorlesungsunterlagen (kein Skriptum):
http://ara.informatik.tu-freiberg.de/
kostenlos im itunes store: Konrad Froitzheim suchen

Diese Unterlagen zur Vorlesung sind weder ein Skriptum noch sind sie zur Prüfungsvorbereitung ausreichend.

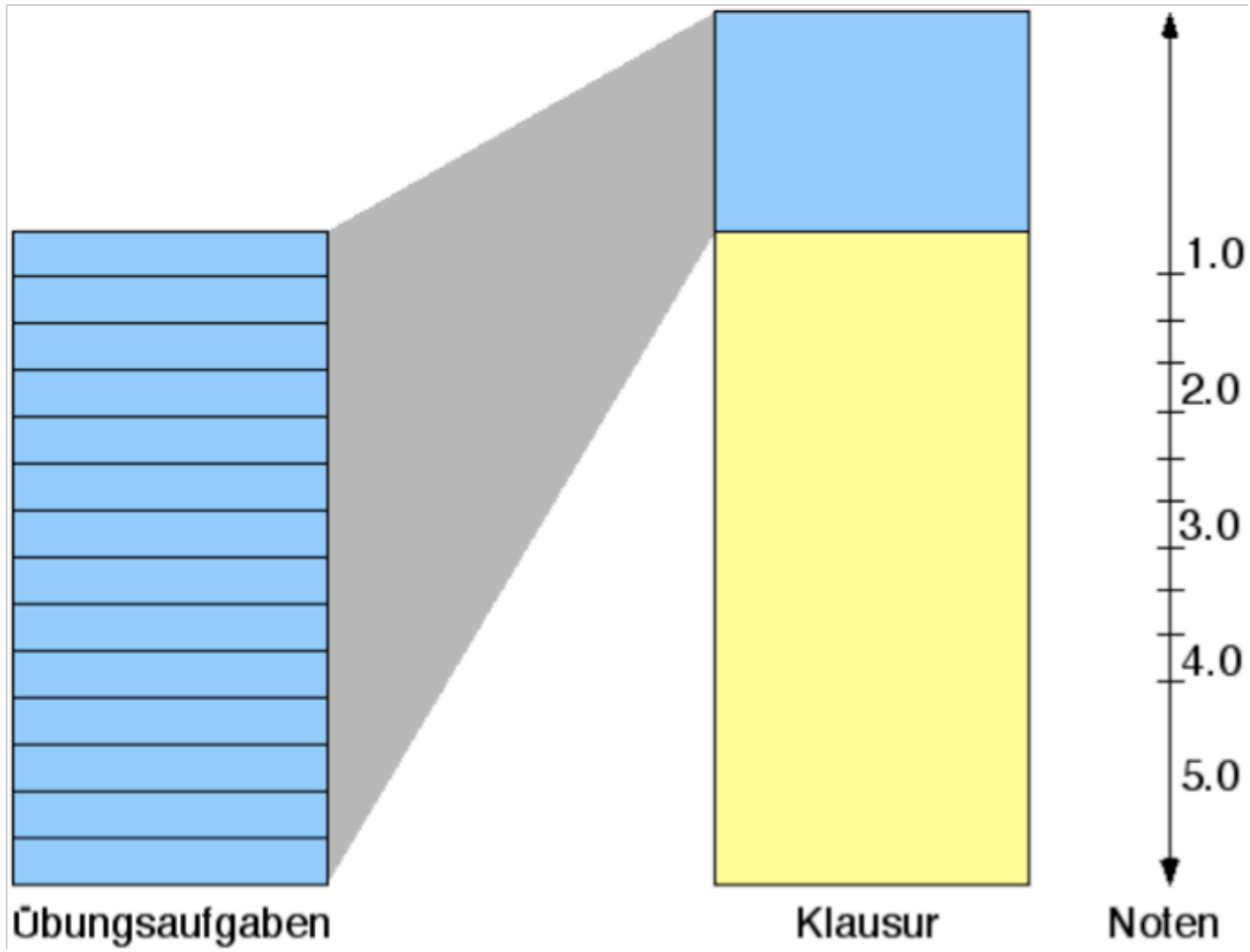
Ohne das gesprochene Wort in der Vorlesung sind sie unvollständig und können sich in Einzelfällen sogar als irreführend erweisen.

Sie dienen dem Vortragenden als Gedächtnisstütze und den Hörern als Gerüst für ihre Vorlesungsnotizen.

Zur Prüfungsvorbereitung ist die intensive Lektüre der Fachliteratur unumgänglich.

Die Unterlagen werden während des Semesters noch gändert oder ergänzt.

- Tutoren
  - Kleingruppen, Organisation im Bildungsportal Sachsen
  - erfahrene Studenten betreuen die Gruppen
  - insbesondere Programmieren
- Übungsaufgaben
  - in Kleingruppen bearbeiten
  - Bewertung durch Assistenten
  - Startkapital für Klausur
- Klausur
  - ca. 100 Punkte
  - 6-8 Aufgaben
  - Wissen und Fähigkeiten
  - bestanden mit ca. 40 Punkten
  - 1.0 ab ca 95 Punkten
  - Übungspunkte als Polster: 100+30+1 mögliche Punkte



Literatur

Balzert, H.: Lehrbuch Grundlagen der Informatik; Elsevier, 2005.

Conway, J., Hillegass, A.: iOS Programming; 2011.

Eernisse, M.: Build your own AJAX web application; Sitepoint, 2006.

Gamma, E. et al: Design Patterns; 1994.

Goll, Bröckl, Dausmann: C als erste Programmiersprache; Teubner, 2003.

Horn, Kerner: Lehr- und Übungsbuch Informatik; 1995.

Kernighan, B., Ritchie, D.: C.

Knuth, D.,E.: The Art of Computer Programming; 1973.

Mark, D., LaMarche, J.: Beginning iPhone 3 Development; 2009.

Post, E.: Real Programmers Don't Use PASCAL; Datamation, Vol. 29, Nr., 1983

Rembold, U.: Einführung in die Informatik; Hanser, 1987.

Schöning, U.: Theoretische Informatik - kurzgefaßt; Spektrum, 1995.

Sedgewick, R.: Algorithms in C; Addison Wesley, 1998 ...

Wirth, N.: Algorithmen und Datenstrukturen; Teubner, 1983 ...

Inhaltsübersicht

1. Einleitung

   …

2. Information und Daten

   2.1 Informationsbegriff

   2.2 Zahldarstellung

   2.3 Rechnen

   2.4 Datentypen

3. Vom Problem zum Programm

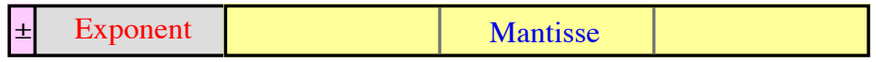
   3.1 Idee und Analyse

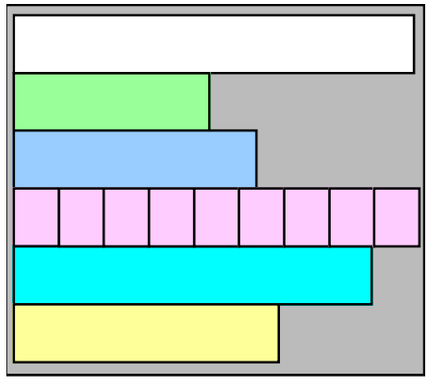
   3.2 Sequentialisierung

   3.3 Programmieren

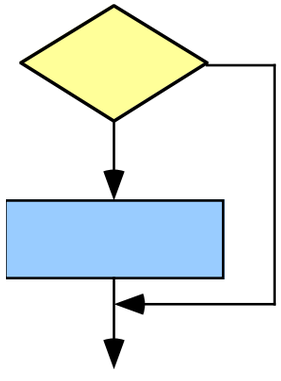
   3.4 Systematische Fehlersuche

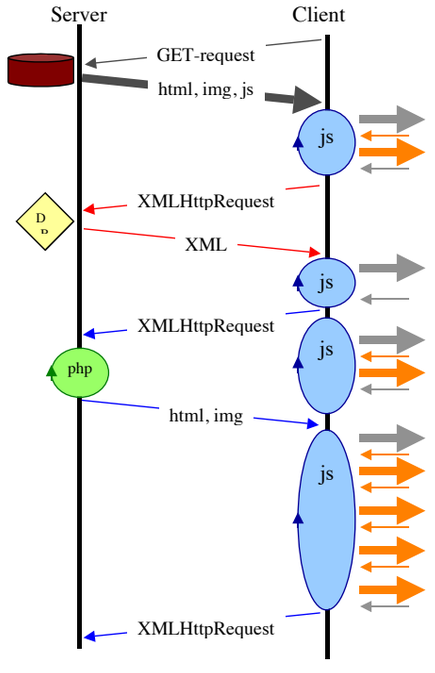
   3.5 Reaktives Programmieren



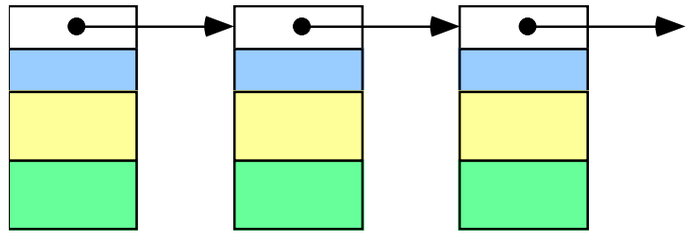


```
int square (int num)
{
      int result;
      result = num * num;
      return result;
}
```

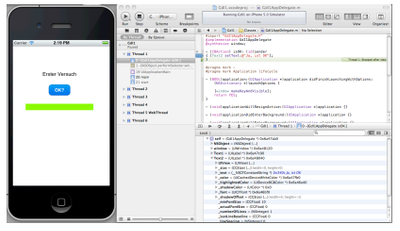




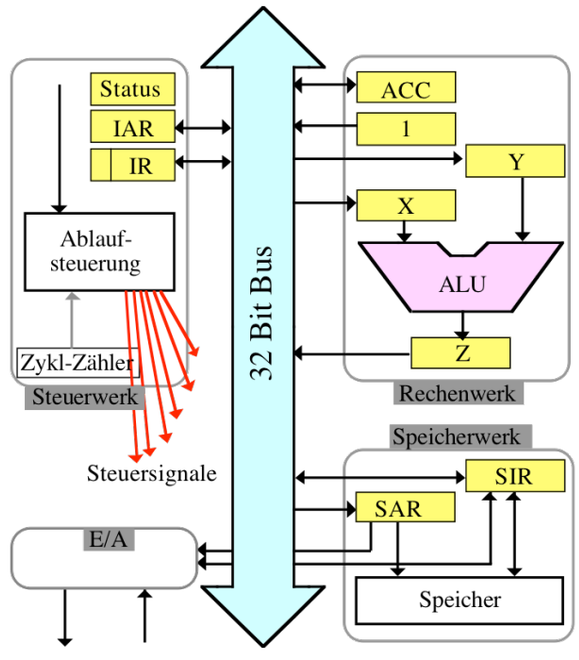
4. Algorithmische Komponenten
   - 4.1 Sortieren
   - 4.2 Listen
   - 4.3 Speicherverwaltung
   - 4.4 Rekursion
   - 4.5 Objekte, Patterns und Frameworks



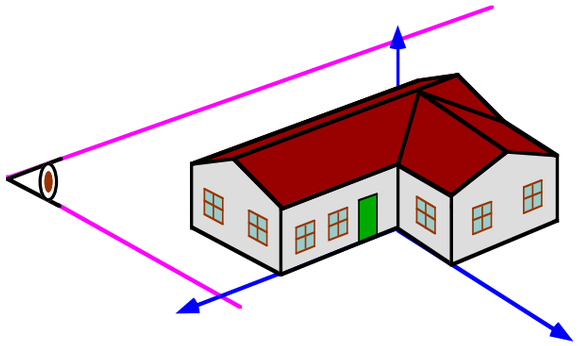
5. Betriebssystem: Abstrahieren und Koordinieren
   - 5.1 Verteilung der Ressourcen
   - 5.2 Struktureller Aufbau am Beispiel iOS
   - 5.3 App Programmieren



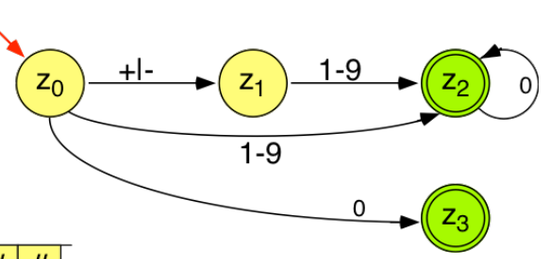
6. Rechnerarchitektur: Vom Transistor zum Programm
   - 6.1 Transistoren, Gates
   - 6.2 Rechnen und Speichern
   - 6.3 Funktionseinheiten: Speicher, Prozessor, Bus, …
   - 6.4 Maschinenbefehle
   - 6.5 Compiler

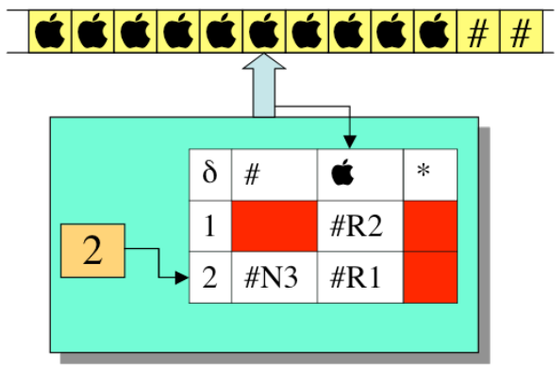


6. Medien
   - 6.1 Medien und Wahrnehmung
   - 6.2 Computergrafik
   - 6.3 Standbilder
   - 6.4 Video
   - 6.5 Audio



7. Theoretische Informatik
   - 7.1 Automaten und formale Sprachen
   - 7.2 Berechenbarkeit
   - 7.3 Komplexitätstheorie





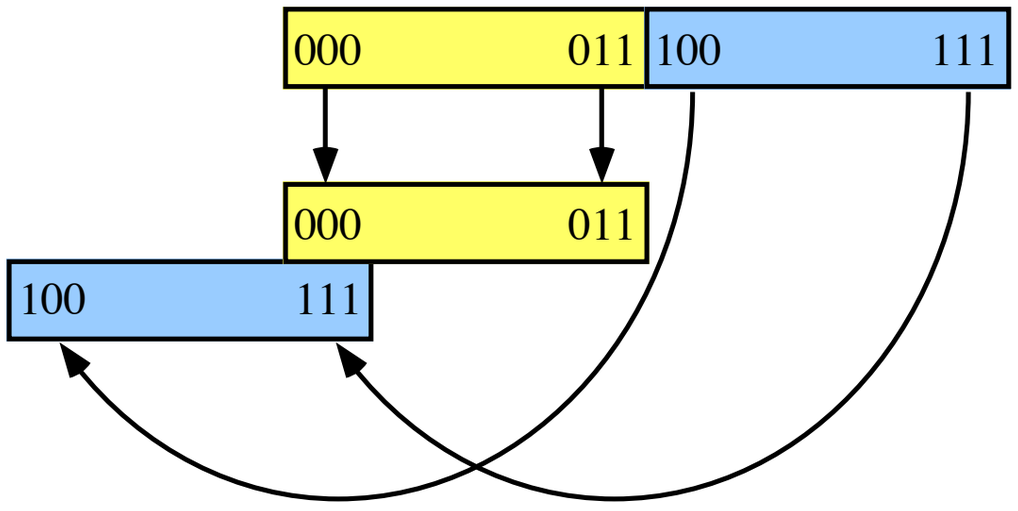
8. Anwendungsprogramme
   - 8.1 Datenbanken
   - 8.2 Bürosoftware
   - 8.3 Medienbearbeitung

KAPITEL 2

Information und Daten

- Information als mathematische Größe
- Zahlsysteme und Zahldarstellung
- Grundlegende Datentypen
- Ideen des maschinellen Rechnens

$$H = \sum_i p(s_i) \bullet ld \frac{1}{p(s_i)}$$

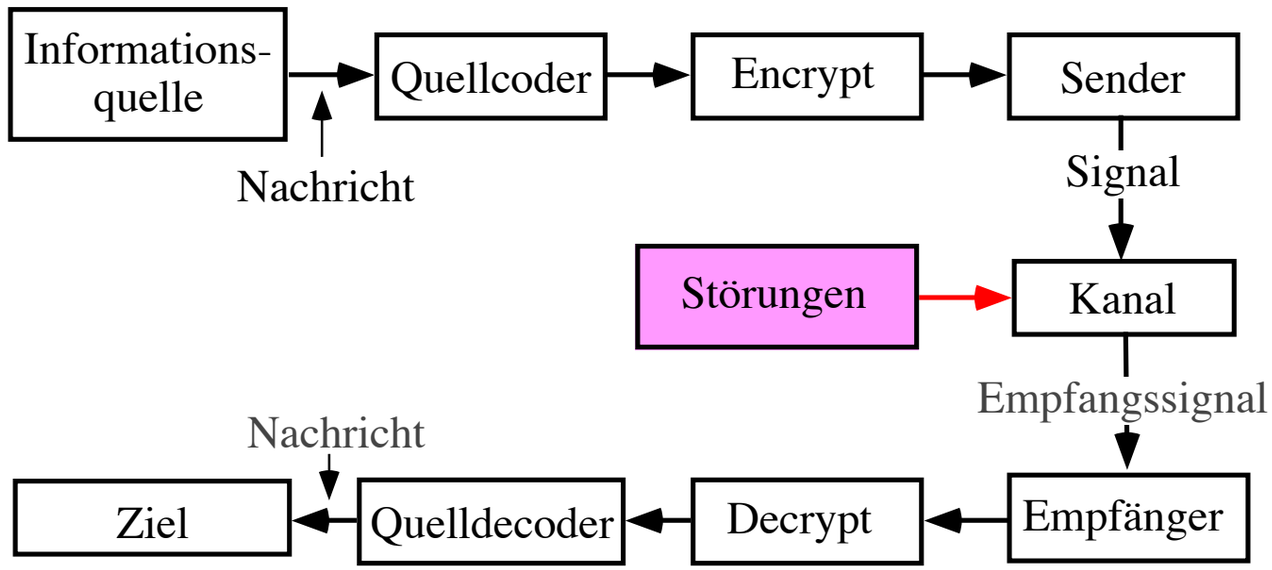


Informationsbegriff

- Shannon, 1948: Definition als mathematische Größe
- Nachrichtenquelle
  - Nachrichten mit Wahrscheinlichkeit $p(s_i)$
  - Wahrscheinlichkeit bestimmt Informationsgehalt
- Informationsgehalt eines Symbols: (Selbstinformation) $I_i = -ld\ p(s_i)$ bit
  - Entropie einer Nachrichtenquelle $H = \sum_i p(s_i) \bullet ld \frac{1}{p(s_i)}$
- Beispiel Münzwurf
  - Ereignisse Kopf und Zahl
  - p(Kopf) = 1/2, p(Zahl) = 1/2
  => H(Kopf) = -ld 1/2 bit = ld 2 = 1 bit
- Beispiel Würfel
  - Ereignisse 1, 2, 3, 4, 5, 6
  - p(i) = 1/6
  => H("3") = -ld 1/6 bit = 2,58… bit

- Wahrscheinlichkeit einer Symbolfolge $s_1 s_2 \ldots s_n$
  - Symbole haben Wahrscheinlichkeit $p(s_i) = p_i$
  - $p = p(s_1)*p(s_2)*\ldots p(s_n)$
  - p("the") = 0,076 * 0,042 * 0,102 = 3,25584 * 10^{-4}
- Wahrscheinlichkeit $p(s_i)$
  - allein ≠ im Kontext
  - $p(s_i='h') = 0{,}042$ => H('the') = 11,6 bits
  - $p(s_i='h' \mid s_{i-1} = 't') = 0{,}307$ => H('the') = 6,5 bits
- Informationsrate in
  - englische Buchstaben: 4,76 bit/Buchstabe
  - englischem Text: 4,14 bit/Buchstabe
  - englischem Text: 9,7 bit/Wort, ~ 2 bit/Buchstabe
- Modell entscheidend
- Satz von der rauschfreien Quellkodierung
  - H(N) = x bit => ∃ Kode K mit $\text{Länge}_K(N) = x+\varepsilon$ Bits
  - Mittlere Kodelänge kann Entropie annähern
- Kode = Nachricht + Redundanz

- Übertragung
  - zwischen Computern
  - im Computer
  - Speicherung



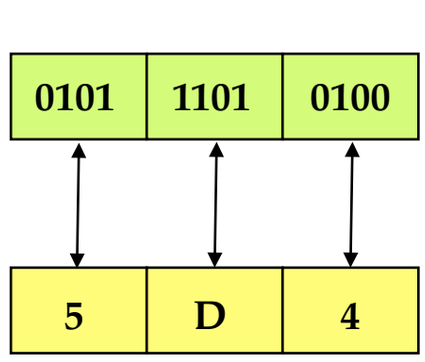
- Kommunikation [Shannon, Weaver]:
  - Lebewesen oder Maschine beeinflußt anderes Lebewesen oder Maschine
  - technisches Problem
  - semantisches Problem (Symbole und ihre Bedeutung)
  - Effektivitäts-Problem (Einfluß der Bedeutung)

Zahldarstellung

- Ganze Zahlen
- Polyadisches Zahlsystem

$$z = \sum_{i=0}^{n-1} a_i B^i$$

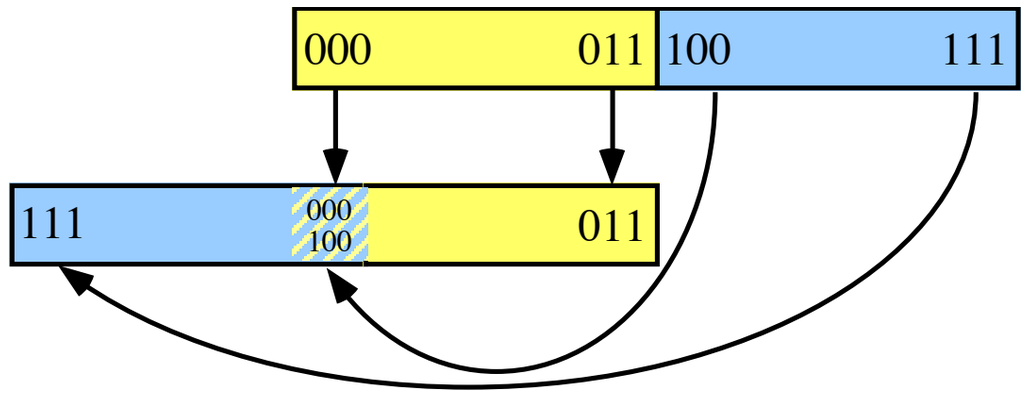
  - $0 \leq a_i < B$
  - Basis 10: $1492 = 2*10^0 + 9*10 + 4*100 + 1*1000$
  - Basis 2: $1492 = 1*1024+0*512+1*256+1*128+1*64$
    $+0*32+1*16+0*8+1*4+0*2+0*1 = 10111010100$
  - Basis 16: $1492 = \$5D4 = 5*256 + 13 * 16 + 4 * 1$

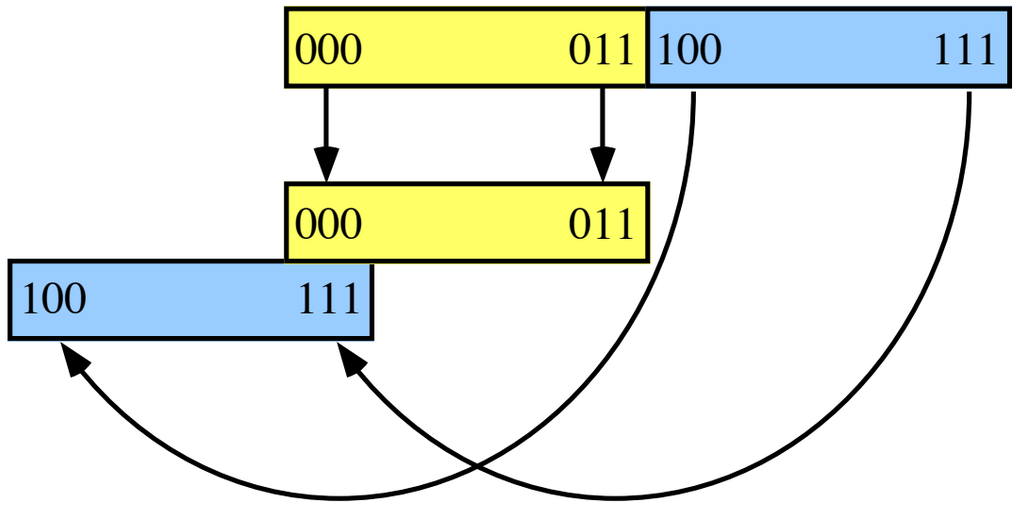


- Zahlenbereich
  - 10 Bit => 0..1023 = 1 'Kilo' -1
  - 20 Bit => 0..1024*1024-1 = 1 'Mega' -1
  - 32 Bit => 0..4 294 967 295 (2^{32}-1) = 4 'Giga' - 1
- negative Zahlen?

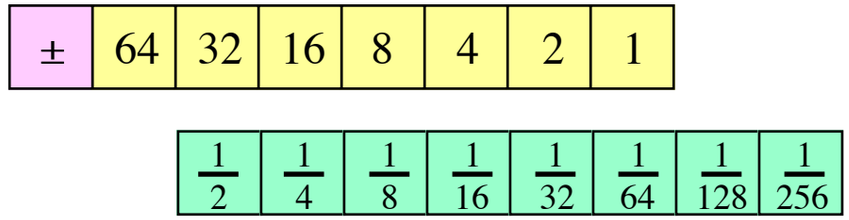
$$2^{10} = 1024$$

- Vorzeichen-Betrag (sign-magnitude)
  - 1 Bit Vorzeichen
  - 8 Bit (256 Werte): -127..127
  - komplexe Rechenregeln
- Stellenkomplement
  - jede Stelle 'negieren': 0->1, 1->0
  - 00111111 = 63
  - 11000000 = -63
  - negative Null: 00000000 und 11111111
  - -127..0,0..127
  - besondere Rechenregeln
- Basiskomplement
  - jede Stelle negieren, 1 addieren
  - 00111111 = 63
  - 11000000+1= 11000001= -63
  - nur eine Null
  - Umrechnung macht mehr Arbeit
  - Rechenregeln einfach





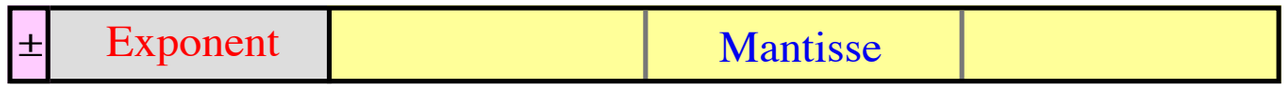
- Rationale Zahlen
  - Brüche: 1/2, 1/3, …
  - Dezimalschreibweise ggg,ddd: 0,5; 12,625
  - evtl unendlich viele Stellen: 16/3
  - ggggg,ddddd… : 1,33333333333333333333...
  - ggg,ddd = ggg + 0,ddd
  - ganzzahliger Teil + Bruchteil



- Näherungsdarstellung von reellen Zahlen
  - $\pi \approx 3{,}1415926$
  - $\sqrt{2} \approx 1{,}414213562$
  - wann sind diese Fehler nicht katastrophal?

=> numerische Mathematik

- Normalisieren
  - ggg,ddd = 0,gggddd * 10^{3}
  - 1,34567 = 0,134567 * 10^{1}
  - 0,012345 = 0,12345 * 10^{-1}
  - 0,0000001 = 0,1 * 10^{-6}
- Floating-Point Zahlen
  - 0 < Mantisse < 1
  - 10^{exp} "Exponent"
  - Vorzeichen



- Trick
  - normalisieren => 0,10111010101010 *2^{exp}
  - erstes Bit immer = 1 => weglassen
- typische Formate
  - ANSI/IEEE 754-1985
  - single: 32 bit - 23 bit Mantisse, 8 bit Exponent
  - double: 64 bit - 52 bit Mantisse, 11 bit Exponent
  - extended: 80 bit

Rechnen

- Addieren im Zehnersystem
  - stellenweise addieren

```
    1513
+   2112
------
    3625
```

  - Übertrag

```
    1523
+   2192
------
    3715
```

  - Rechenregeln für Übertrag 7+8 = 5 + Übertrag 1
- Binärer Fall
  - stellenweise addieren

```
    01001010
+   00101111
------------
    01111001
```

  - Rechenregeln einfach: 0+0=0, 0+1=1, 1+0=1, 1+1=0 Übertrag 1

- Beschränkte Stellenzahl
  - größte darstellbare Zahl
  - Ergebnis grösser: Überlauf

$$\begin{array}{rr} & 11001010 \\ + & 10101111 \\ \hline & 01111001 \end{array}$$

  - eventuell negative Zahl

$$\begin{array}{rr} & 01001010 \\ + & 01101111 \\ \hline & 10111001 \end{array}$$

- Subtrahieren
  - Komplement addieren: a-b = a+(-b)
  - besondere Regeln zur Ergebnisinterpretation

- Multiplikation
- Primitives Verfahren

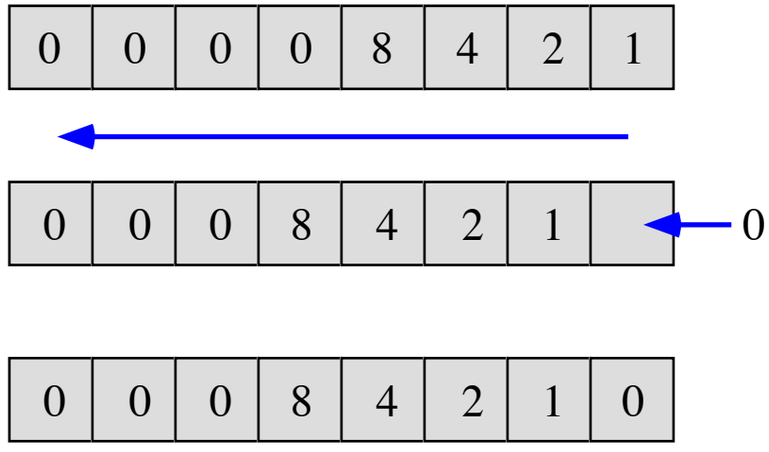
  - Multiplikand * Multiplikator

  1. erg auf Null setzen
  2. erg = erg + Multiplikand
  3. Multiplikator um 1 herunterzählen
  4. falls Multiplikator > 0: weiter mit 2

- Sonderfall

  - Verschieben um eine Stelle, Null nachziehen

  => Multiplikation mit Stellenwert

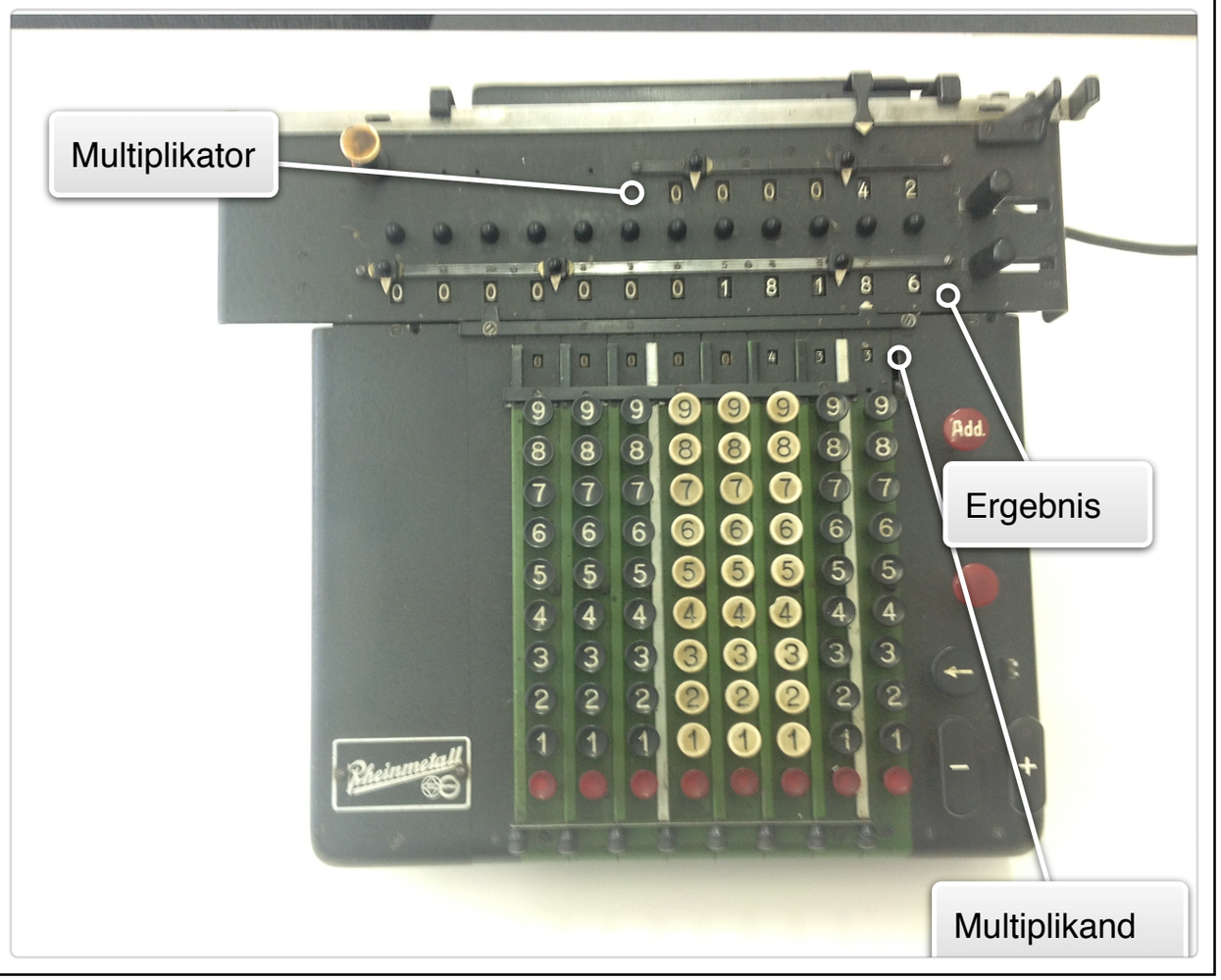


- Multiplizieren
  - Multiplikand * Multiplikator
    1. i auf Eins setzen
    2. Addieren von (Multiplikand * Stelle i des Multiplikators)
    3. Verschieben des Multiplikanden (mit Stellenwert multiplizieren)
    4. i hochzählen
    5. weiter mit 2, falls i ≤ Stellenzahl Multiplikator
  - im Zehnersystem:

```
1214   *   211
1214
12140
------------
242800
256154
```

- Trick:
  - Multiplikator in jedem Schritt eine Stelle nach rechts schieben
  - letzte Stelle in Schritt 2 verwenden

Interactive 2.1 Rechenmaschine ca. 1938



- binär Multiplizieren
  - ShiftLeft = Multiplikation mit 2
  - Stellen nur 0 oder 1 => Multiplikand addieren oder nicht
- Verfahren
  1. Ergebnis = 0; i = 0; a = Multiplikand; b = Multiplikator
  2. falls letzte Stelle von b = 1: Ergebnis = Ergebnis + a
  3. ShiftLeft(a)
  4. ShiftRight(b)
  5. Falls b>0 : weiter mit 2
- Beispiel: 12 * 5

Iteration	a	b	erg
0	0000 1100	0000 0101	0000 0000
			+ 0000 1100
1	0001 1000	0000 0010	0000 1100
			nix tun
2	0011 0000	0000 0001	0000 1100
			+ 0011 0000
3	0110 0000	0000 0000	0011 1100

Datentypen

- Buchstaben
  - 'A', 'B', …, 'Z', 'a', 'b', …, 'z', '.', ',', ';', …, '0', …, '9'
  - 65, 66, …
  - ASCII, EBCDIC
  - Sonderzeichen: .,!"§$%&/()=?@”#^\~.⁻¨*#;:_-<>
  - unsichtbare Zeichen: Space/Blank, CR

```
char <Name>;
```
A

- Zeichenketten (strings)
  - zusammengesetzt aus Buchstaben
  - 'Auto', 'Bergakademie', 'Klaus Dieter'

```
char <Name> [n];
```

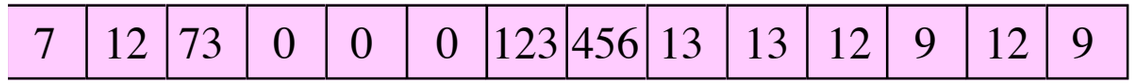
K	l	a	u	s	␣	D	i	e	t	e	r		

- Wie lang ist ein string?
  - Länge(string) = Anzahl Buchstaben
  - Länge('Auto') = 4
  - Länge('Klaus Dieter') = 12
  - Längenfeld oder besonderes Zeichen am Ende

- Boolsche Typen
  - 2 Werte: true, false
- Aufzählungen
  - frei gewählte Bezeichner
  - (rot, gruen, blau)
  - (Mercedes, BMW, VW, Ford, Opel, Audi, Porsche)
  - werden auf Zahlen abgebildet
  - `enum {red,green,blue} color`
- Zahlen (typische Länge …)
  - short, unsigned short (16 bit)
  - int, unsigned int (16, meist 32 bit)
  - long, unsigned long (32 oder 64 bit)
  - float (32 bit)
  - double (64 bit: 2.22 e-308 bis 1.79 e+308)

- Zusammengestzte Datentypen
  - aus mehreren elementaren Datentypen zusammengesetzt
  - gefühlter Sonderfall Zeichenkette siehe oben
- Vektoren und Matrizen
  - gleiche Datentypen

```
float vielstufig [n][m]…[k]
```

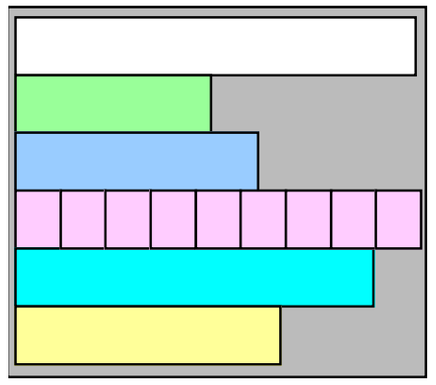


$$\begin{pmatrix}1\\2\\3\end{pmatrix} \qquad \begin{pmatrix}1 & 0 & 7 & 8\\0 & 1 & 9 & 2\\12 & 13 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix} \qquad \begin{pmatrix}\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 1\end{pmatrix} & \begin{pmatrix}1 & 2 & 4\\4 & 4 & 6\end{pmatrix} & \begin{pmatrix}7 & 2 & 5\\8\end{pmatrix}\end{pmatrix}$$

- Datenstrukturen (Records, zusammengesetzte Daten, …)
  - unterschiedliche Typen

```
struct Wagenbeschr
{  char     name[32];
   enum     {Schlaf,Abteil, …} Wagentyp ;
   int      Baujahr;
};
```

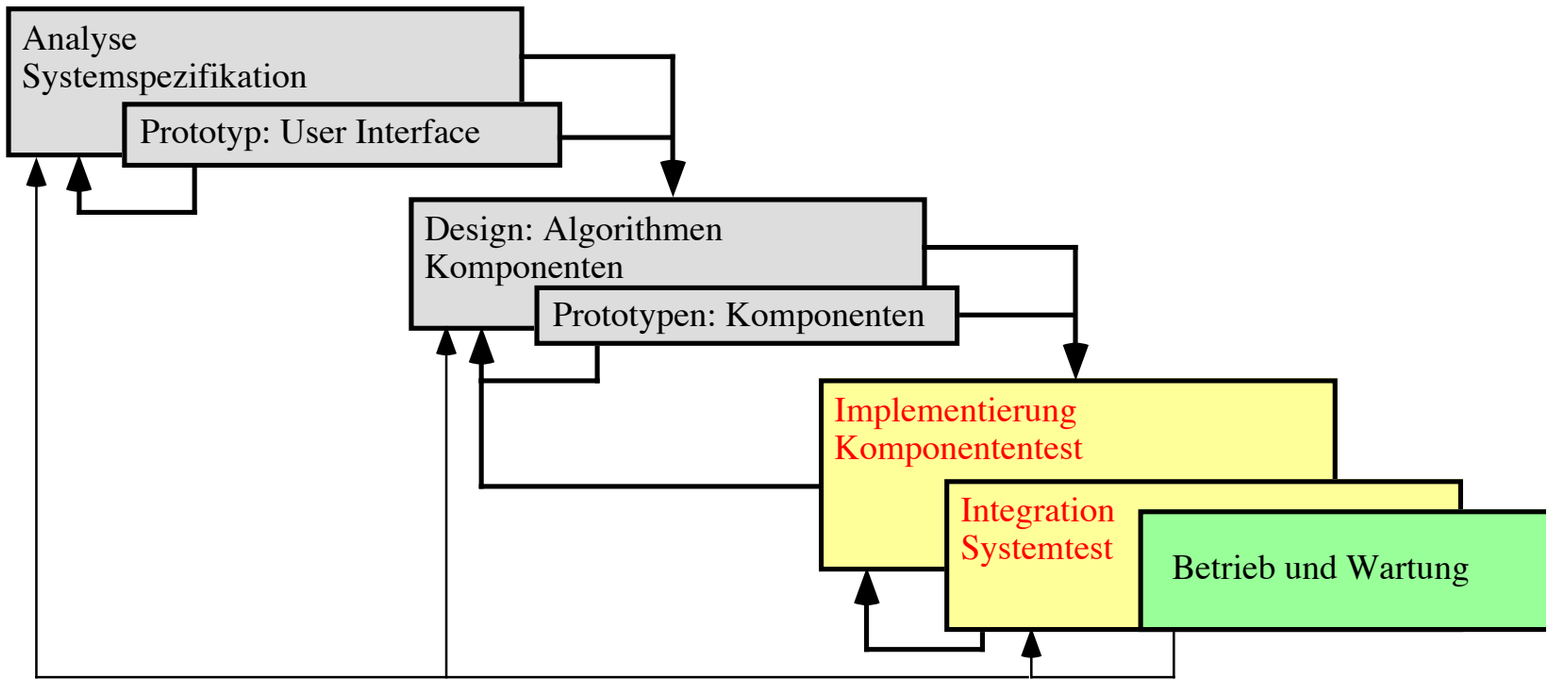
  - beschreiben Gegenstände, Personen, …



KAPITEL 3

Vom Problem zum Programm

- The Art of Computer Programming [D. Knuth, 1968]
- Programmieren ist Ingenieurtätigkeit
  - systematisch, planbar
  - reproduzierbar
  - dokumentiert
- Theoretischer Ablauf



- Computer Aided Software Engineering (CASE)
  - Computerprogramm zum Programmieren
  - grafische Hilfsmittel
  - formale Spezifikation
  - Versionsverwaltung
- Rollen
  - Projektleiter
  - Analytiker
  - Programmierer
  - Tester
- Phasenmodell
  - globale Anforderungen, Struktur der Arbeit
  - Untersuchung eines Geschäftsgebietes, Anforderungen
  - Systementwurf aus Benutzersicht
  - Technische Design
  - Codierung, Datenbank-Generierung
  - Systemeinführung, Test
  - Verwendung (Produktion)

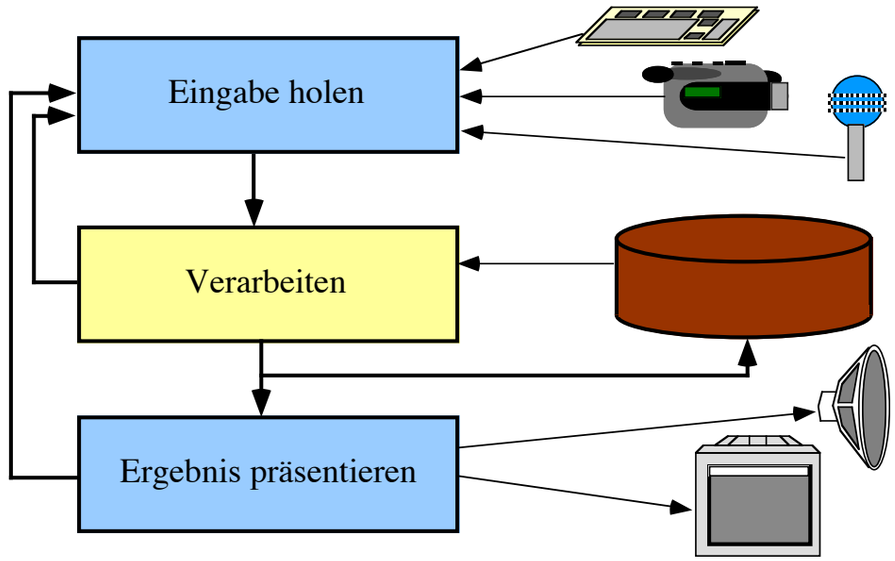
- Agile software development
  - kleine Gruppen
  - Selbstanordnung
  - ständige Kommunikation
  - intensive Reviews
  - Kunden integriert in Zyklus
- Extreme Programming

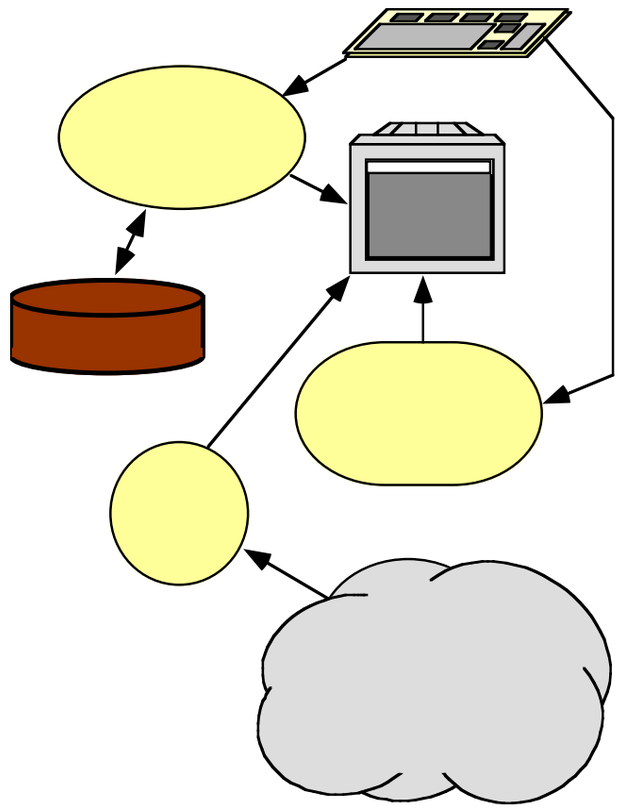
- Imperatives Programmieren
  - Programm entsprechend Datenfluss
  - Funktionen zur Daten- und Zustandänderung
  - Datenstrukturen
  - Pascal, C, Modula, Cobol, Fortran, …
- Objektorientiertes Programmieren
  - Datenstrukturen mit Transformationsfunktionen
  - Zustände in den Datenstrukturen
  - ereignisorientiertes Programmieren
  - Smalltalk, Java, C++, Oberon, …
- Funktionales Programmieren
  - Folgen mathematisch/logischer Funktionen
  - Lambda-Kalkül
  - ohne Zustände und änderbare Daten
  - APL, ML, **Haskell**, Lisp, Prolog, …

```
fac = 1; i = 1;
do
    {fac = fac * i;
     i = i+1;}
while (i <= n);
```

```
fac :: Integer -> Integer
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

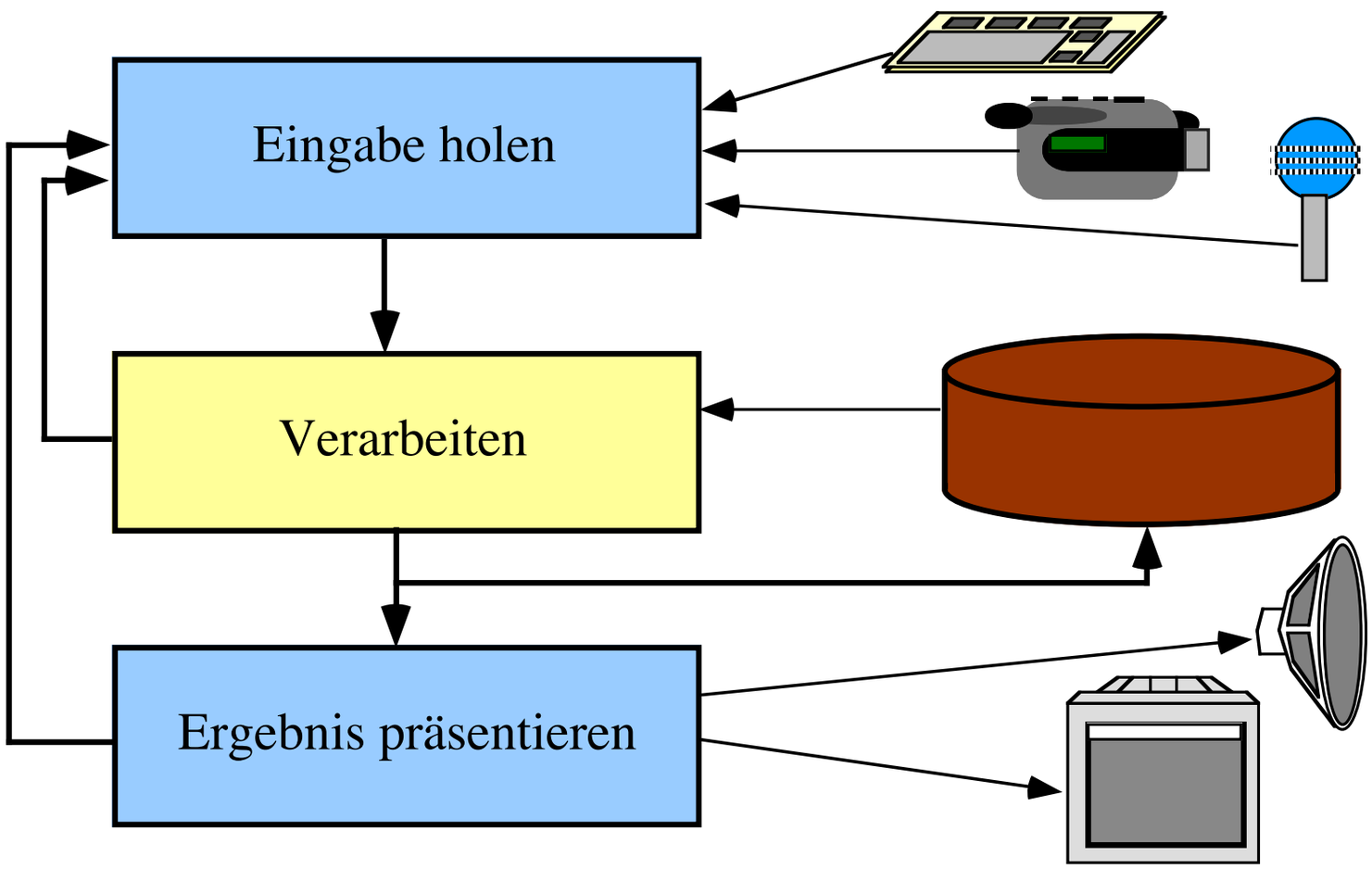
- Aktives Programmieren
  - Ablauf kontrollieren
  - Beginn,
    {Eingabe, Verarbeitung, Ausgabe},
    Ende
  - Programmierer plant Programmablauf
  - Programmiersprachen siehe oben
- Reaktives Programmieren
  - Komponenten bereitstellen
  - gekapseltes Verhalten
  - ereignisgesteuert
  - Anreiz -> Transformation -> Antwort
  - AJAX
  - php, Ruby, …





Idee und Analyse

- Aufgaben
  - Berechnen
  - Bearbeiten
  - Organisieren
  - Kommunizieren
  - Messen und Steuern
- Programmstruktur



- Problem definieren
  - intuitve Beschreibung
  - Eingabe detailliert aufschreiben
  - Resultate definieren
  - Ausgabe genau aufschreiben

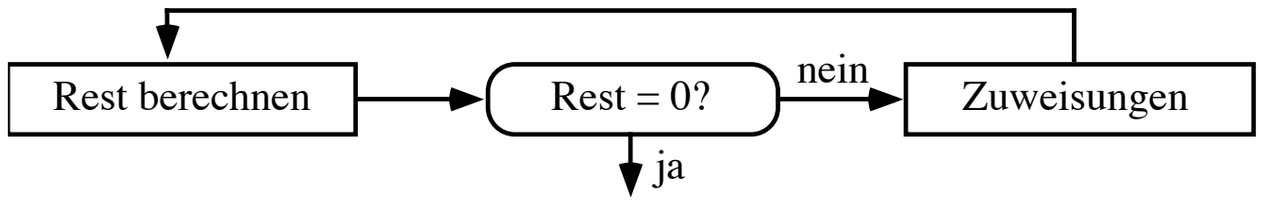
PROBLEM	VERFAHREN	TYPISCHE OPERATIONEN
Pullover anfertigen	Strickmuster	rechte Masche, linke Masche
Kuchen backen	Rezept	Mehl wiegen, rühren
Schrank aufstellen	Bauanleitung	schrauben, stecken, fluchen

- Wissen im Gebiet sammeln
  - Erfahrung der Mitarbeiter bzw. Kunden nutzen
  - klassiche Bearbeitung genau studieren
  - ähnliche Programme untersuchen
  - Gesetze bzw. Normen beachten
- Verfahren (er)finden
  - Formeln und Regeln
  - Menge von Operationen

- Berechenbarkeit prüfen
  - Eingabewerte erfassbar?
  - Ergebnis ausdrückbar?
  - Laufzeit (Komplexität) des Verfahrens
  - Ressourcen (Speicherplatz)
- Datenstrukturen bilden
  - Abbildung der Eingabe
  - Transformation in Ausgaben
  - Vollständigkeit
  - Speicherplatz
  - optimierter Zugriff (schnell und/oder einfach)
- Werkzeuge auswählen
  - Computertyp- und Ausstattung, Betriebssystem
  - Programmierwerkzeuge (Sprache, Prototyping-System, …)
  - Basissoftware (Datenbank, …)
- Kosten ermitteln
  - Arbeitszeit
  - Geräte und Material

Sequentialisierung

- Algorithmus
  - Abu Ja'far Mohammed ibn Mûsâ al-Khowârizm
  - Abfolge von Schritten
  - Rechenvorschrift
  - bis ca. 1950 nur im Zusammenhang mit Euklidschem Algorithmus



- Euklidscher Algorithmus
  - größten gemeinsamen Teiler ggt von (m,n) finden

```
1. setze rest = (m DIV n)
2. rest = 0? setze ggt = n; fertig
3. setze m = n und n = rest; weiter mit 1.
```

- Schrittweise Verfeinerung (step-wise refinement)
  - Zerlegung in Teilprobleme
  - Verfahren finden oder entwickeln

- Beispiel: Kaffee machen (Pseudocode)
- Grobverfahren

```
1. Wasser kochen
2. Nescafe in die Tasse
3. Wasser in die Tasse
```

- Einzelschritte verfeinern

```
1.1 Kessel mit Wasser füllen
1.2 Auf die Herdplatte stellen
1.3 Herdplatte anstellen
1.4 Warten bis Wasser kocht
1.5 Herdplatte abstellen

2.1 Glas öffnen
2.2 Menge Kaffeepulver auf den Löffel laden
2.3 Löffel in die Tasse leeren

3.1 Kessel von der Herdplatte nehmen
3.2 Tasse mit Wasser füllen
```

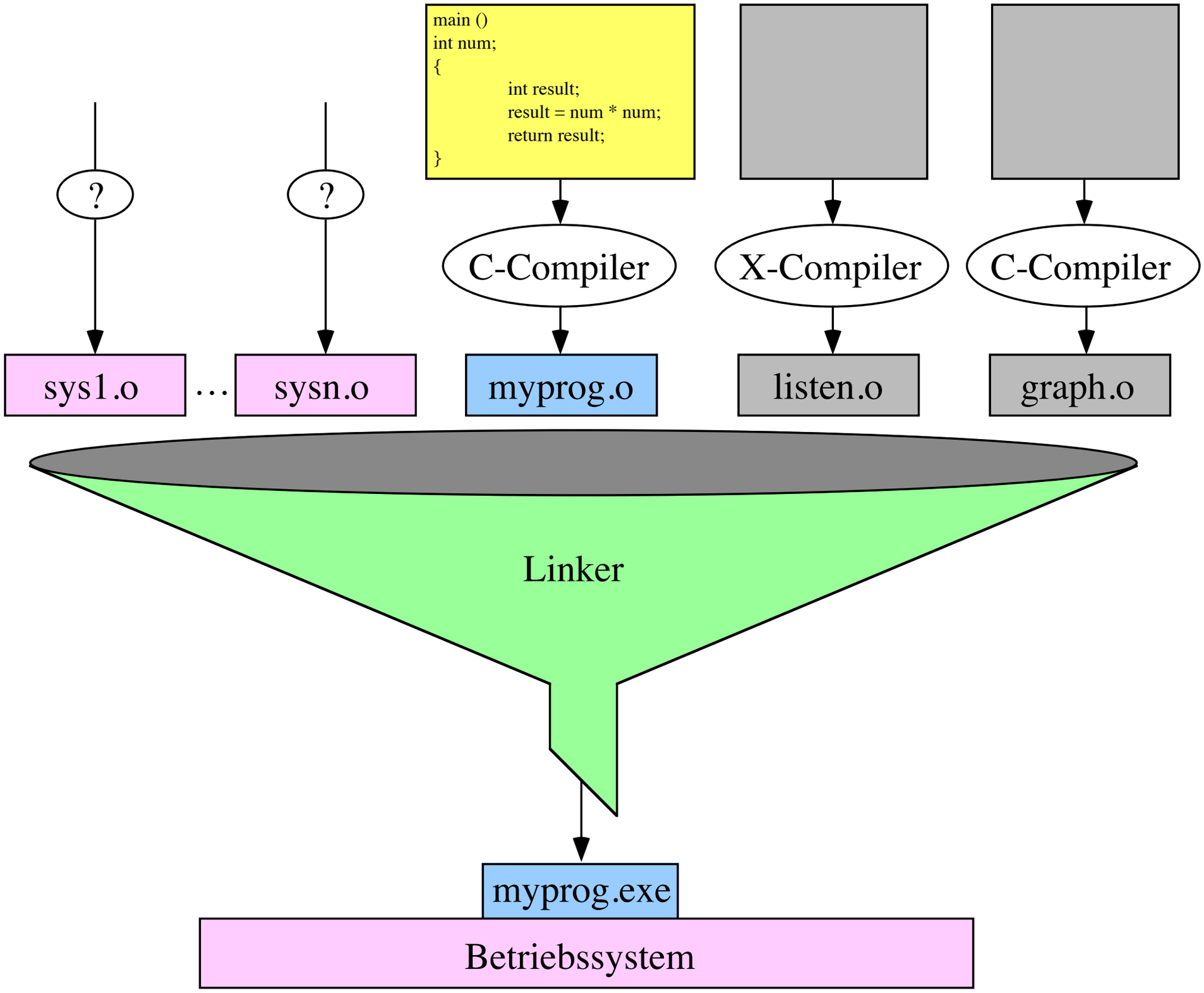
- Weiter verfeinern

- Methoden zur Strukturbeschreibung
  - Algorithmus beschreiben
  - Verfeinerungsstufen
- Grafische Darstellungsformen
  - Flußdiagramme (DIN 66 001)
  - Struktogramm (Nassi-Shneiderman-Diagramm, DIN 66 261)
  - Datenflußplan
- Sprachliche Beschreibung
  - Pseudocode
  - spezielle Entwurfssprachen
  - Spezifikationssprache
  - Elemente einer Programmiersprache
- Wesentliche Verarbeitungselemente
  - Sequenz
  - Selektion, Iteration
  - Gruppierung, Parallelität
  - Ein- und Ausgabe

Programmieren

- Viele Programmiersprachen
  - Allzwecksprachen: C, Pascal, Modula, Oberon, Ada
  - Spezialisiert: COBOL, FORTRAN
  - BASIC
  - objektorientiert: SmallTalk, Java, C++,
- Arbeitszyklus
  1. Editieren
  2. Übersetzen (Compiler)
  3. Zusammensetzen mit Standardteilen (Linker)
  4. Ausführen und Fehlersuchen
  5. if Fehler then 1.
- Fehlersuche
  - wesentlicher Bestandteil des Programmierens
  - Debugger
  - schrittweise ausführen
  - Werte und Zwischenergebnisse anzeigen

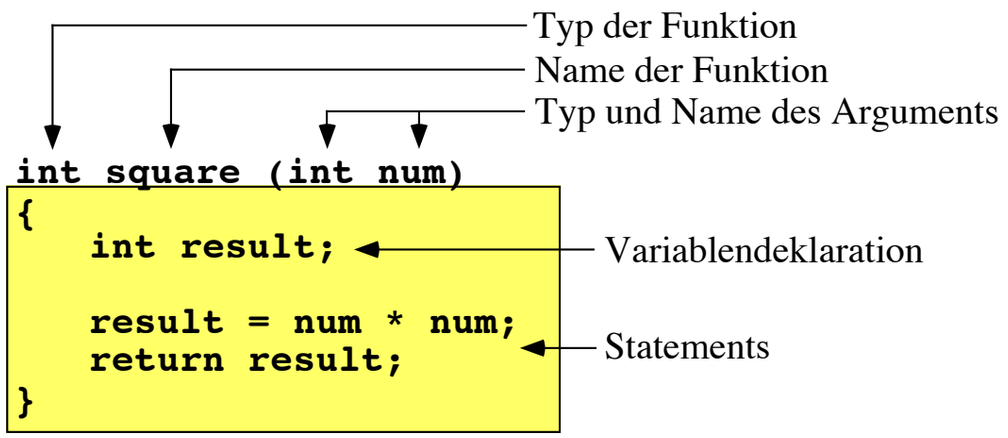
> It is practically impossible to teach good programming style to students that have had prior exposure to BASIC. As potential programmers, they are mentally mutilated beyond hope of regeneration. [E. W. Dijkstra]


main ()
int num;
{
int result;
result = num * num;
return result;
}
?
?
C-Compiler
X-Compiler
C-Compiler
sys1.o
…
sysn.o
myprog.o
listen.o
graph.o
Linker
myprog.exe
Betriebssystem

- C
  - von Dennis Ritchie und Ken Thompson
  - weit verbreitet, sehr portabel
  - eng verbunden mit dem Erfolg von UNIX
  - ANSI-Standard X3.159-1989, C89/C90
  - ISO-Standard ISO 9899:1999 (C99)
  - hat vielfältige Ausdrucksmöglichkeiten
  - effizientes Programmieren möglich
  - verleitet zum Tricksen

```
int square (int num)
{
  int result;
  result = num * num;
  return result;
}
```

- Zentrale Abstraktion: Funktionen
- Funktionskopf (Deklaration)
  - beschreibt die Schnittstelle zu den Benutzern
  - Parameteranzahl und -typ (Argumente)
  - Ergebnistyp
- Funktionsrumpf (Definition)
  - Variablen-Deklaration (Speicherplatz-Bestellung)
  - Verarbeitung
  - benutzt eventuell andere Funktionen



- 3 Funktionen die man immer braucht
- Einlesen von Eingabewerten: scanf()
  - beliebig viele Parameter
  - int, float, char, string…
  - von stdio - Standard-Eingabe
  - meist Tastatur

```
scanf("%d %d", &ersteZahl, &zweiteZahl);
```

- Eingabesteuerung
  - Formatstring mit Beschreibung der Eingabe
  - %d, %i integer
  - %e, %f, %g float
  - %c char
  - %s string
- Besondere Zeichen
  - space trennt und wird übersprungen
  - Zeilenende beendet Eingabe

```
123__456__<CR>  => ersteZahl=123; zweiteZahl=456;
```

- Eingabeanleitung muß mit printf() erzeugt werden

- Ausgabe: printf()

```
printf("Hello World\n ");
```

  - beliebig viele Parameter
  - Formatsteuerung wie scanf()
  - auf "stdout" - Standard Ausgabe
  - z.B. Bildschirm
  - mischt Formatstring und Variablen

```
printf("hier sind 3 Zahlen: %f %d %f",fl,gz,my);
```

- Kombinierte Ein/Ausgabe

  => Benutzungsoberfläche

```
printf("Buchstaben eingeben:"); /* scanf flushed */
scanf("%c",&zeichen); /* stdin */
printf("Der Zeichenwert ist: %d\n", zeichen)
```

```
Buchstaben eingeben:A
Der Zeichenwert ist: 65
_
```

- Spezial-Funktion main()
  - dahinter steckt das selbstgeschriebene Programm
  - gesamte Software ist eine Menge von Funktionen
  - Anwendungsprogramm ist auch eine Funktion

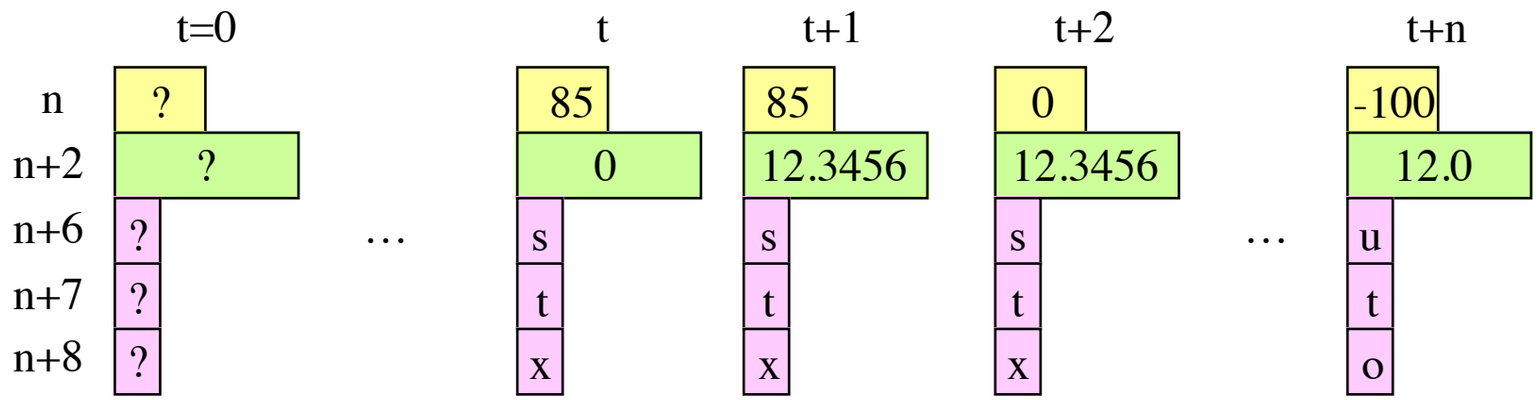
```
#include <stdio.h> /*das Noetigste importieren*/
int main(void)
{
   printf("kuckuck\n"); /* Verarbeitung */
   return 0;/* mit Rueckgabewert "kein Fehler" */
}
```

- Programm eintippen mit irgendeinem Editor
  - Notepad, TextEdit, emacs, vi, xedit, …
- Übersetzen und ausführen
  - cc ist der compiler, linkt auch

```
>cc -Wall -o myprogram myprogram.c
>./myprogram
kuckuck
>
```

Variablen-Deklaration: `int result;`

- Identifier
  - Namen, die der Programmierer erfindet
  - Buchstaben, Ziffern, "_", keine Ziffer am Anfang
  - case-sensitive
- Manche Zeichenketten bereits besetzt
  - Schlüsselworte: if, switch, while, …
  - vordefinierte Typen: int, float, …
- Variablen
  - Container zum Aufbewahren von Werten
  - werden vom Compiler als Speicherplatz angelegt
  - nach dem Programmende (...) verloren



- Mit Identifiern bezeichnet
  - Verwendung in Formeln, …
  - Speicherplatzadresse wird manchmal benötigt
  - &<ident> liefert Adresse
- Aussagekräftige Namen
  - dokumentieren Programm
  - lassen Typ erkennen
  - ungarische Notation [Symoni, 1999]
- Deklaration
  - Typ <Name>, …, <Name>;

```
int      i,j;
float    pi, psi, karl_otto;
char     zeichen;
char     name [32], name2 [16], _2ternname [31];
```

- Literale
  - Wert fest (Konstante?)
  - für Vergleiche, Initialisierung
  - Zahlen, Buchstaben, String

```
index = index + 1;
if (zeichen > 'Z') …;
zeichen = zeichen + 32; /* macht Kleinbuchst. */
pi = 3.1415926;
mpi = -3.1415926;
```

- Variablendeklaration mit Vorbesetzung
  - Typ <Name> = <Ausdruck>;

```
int     i = 12;
double  pi = 3.1415926;
double  pi1 = 4*atan(1);
double  pi2 = -mpi;
char    space = ' ';
```

- Typkonvertierung
  - zur Anpassung von Parametern
  - Verwendung in Ausdrücken; Bsp: `erg = i * pi;`
  - eventuell mit echter 'Umrechnung'
  - manche Konvertierung nur 'formal'
- Explizite Konvertierung (typecast)
  - `(<typname>) <ausdruck>`
  - `int n = 3; …; erg = sin((double) n);`
  - guter Stil
- Implizite Konvertierung
  - vom Compiler eingebaut
  - typisch in Ausdrücken und Funktionsaufrufen
  - Rückgabewerte
  - Typhierarchie
- Vorsicht mit der impliziten Konvertierung

```
i = 1.5;   /* i enthaelt nun 1 */
j = -5.8;  /* j enthaelt nun -5 */
f = 3;     /* f enthaelt nun 3.0*/
```

- Gültigkeit der Variablen
  - Ort der Vereinbarung
  - im 'Programm' => globale Variable
  - im Block => lokale Variable
  - in der Funktion => lokale Variable
  - entscheidet über Benutzbarkeit
- Lebenszeit von Variablen
  - lokal: in der Funktion - nur während des Funktionsaufrufes (Block)
  - global: während der gesamten Programmausführung
  - static macht auch lokale Variable permanent
  - siehe auch Abschnitt Funktionen

Operatoren, Ausdrücke und Anweisungen

- Formeln

```
dreiecksflaeche = grundlinie * hoehe / 2;
kreisflaeche = radius*radius*3.1415926;
umfang = 2*radius*pi;
```

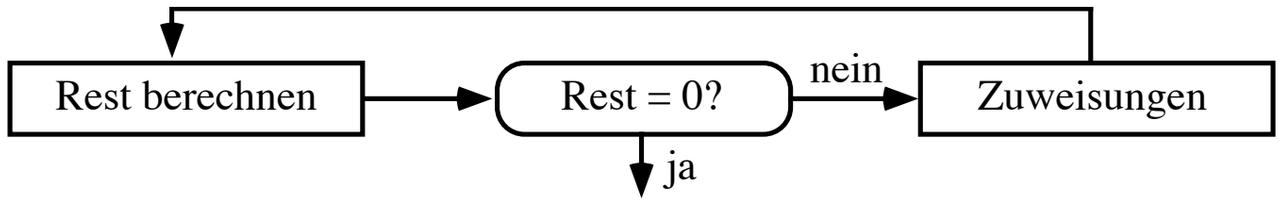
- Unäre (einstellige) Operatoren
- Binäre (zweistellige) Operatoren

Vorzeichen	+,-
Inkrement und Dekrement	++, --
Adresse	&
Negation	!

Multiplikation	*, /, % (Rest der int-Division)
Addition	+, -
Relation	<, <=, >, >=
Gleichheit	==, !=
logisches UND	&&
logisches ODER	ll
Zuweisung	=, +=, -=, *=, …

- %

```
do
{  rest = m % n;
   if (rest != 0)
   { m=n; n=rest;
   }
} while (rest>0);
```



- Vergleiche <, <=, >, >=
  - Zahlen 0 oder 1 als Ergebnis
  - 0 ist false, alles "ungleich 0" ist true

```
rest>0; m!=n;
```

- Gleich oder ungleich: ==, !=
  - geht oft nicht bei Gleitpunktzahlen
  - liefert 0 oder 1

```
-1 < 0, 0 == 0, 1 != -1     /* true */
0>1, 1>10                   /* 0 */
```

- Logische Operatoren: &&, ||

```
((a>b)&&(f>g))   /*(5>3) UND (7.3>-1.2) => true */
(((a>=b)||(f<=g)) && i)  /* i entspricht i!=0)*/
```

- Zuweisung =

```
links = <Ausdruck>;
```

- legt ausgewerteten Ausdruck in Variable links
- kann auch als boolsches Resultat verwendet werden

```
if (i=0) printf("ich werde nie gedruckt");
```

- Arithmetische Zuweisung: +=, -=, *=, /=, %=

```
<variable> op= <ausdruck>;
```

- entspricht `<variable> = <variable> op <ausdruck>;`
- natürlich auch mit Ergebnis

- Inkrement: ++, --
  - addiert bzw subtrahiert 1
  - liefert Inhalt von j vor dem Inkrement: `j++`
  - liefert Wert von j nach dem Inkrement: `++j`
  - liefert auch Ergebnis: `do … while (j--)`

- Klammerregeln
  - Klammer hat höchsten Vorrang
  - gruppiert Formelteile

```
(a+b)*c != a+b*c
1 + ((3+1)/(8-4)-5)
```

- Vorrang-Regeln (precedence)

Typ	Assoziativität
Klammern	links -> rechts
unäre Ops	rechts -> links
Multiplikation, Addition	links -> rechts
relational	links -> rechts
bitweises UND, ODER	links -> rechts
logisches UND, ODER	links -> rechts
Zuweisung	rechts -> links

- Wirkung auf verschiedene Typen

```
int j=5;
float f=5;
einint = j / 2;     /* einint  == 2 */
einfloat = f/2;     /* einfloat == 2.5 */
```

  - Typ des Ausdrucks wird durch Elemente des Ausdrucks bestimmt

```
einfloat = j/2;     /* einfloat == 2.0 */
```

- Elegant

```
aktuelles_zeichen = name[index++];
summe += element;
```

- Etwas kryptisch:

```
a = b = c;
if (a=b) …;
```

- Bitte nicht:

```
x = j * j++;
einfloat = einint = 3.5;
einint = einfloat = 3.5;
```

- Anweisungen (Statements)
  - `<Ausdruck>;`
  - Selektion, Iteration, Sprung
  - Increment
  - mehrere geklammerte Statements

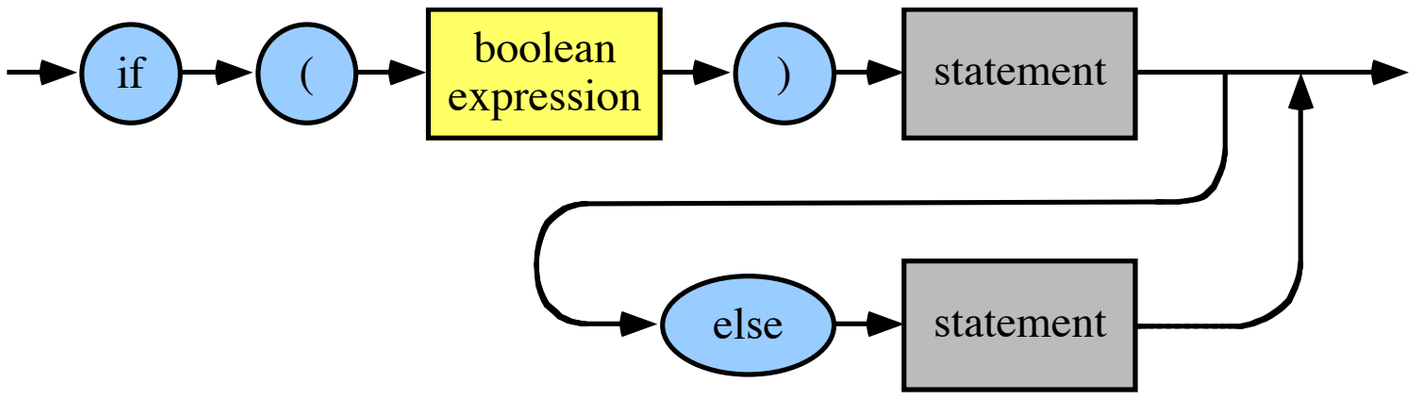
```
{  anz_gedr_zchn=printf("auch printf hat Ergebnis");
   gesZeichen += anz_gedr_zchn;
   if (gesZeichen > 80)
   {  printf("\n");
      gesZeichen =0;
   }
}
```

- {} macht aus mehreren Statements ein Statement (Block)
  - {…;…;…;}
  - macht Syntax-Beschreibung einfacher
  - Variablenvereinbarung möglich

```
if (a>b) <statement>;
else <statement>;
```

Kontrollfluß

- Verzweigung
  - then implizit, immer vorhanden



  - else optional
- Beispiele

```
if (a>b) max = a;
else max = b;

if (b>a) /* Werte tauschen */
   { swap = a; a = b; b = swap; }

if (divisor==0) printf("Fehler\n");
else quotient=dividend/divisor;
```

- Boolean Expression
  - eigentlich normaler Ausdruck mit Zahl als Resultat
  - true falls Resultat ≠ 0!

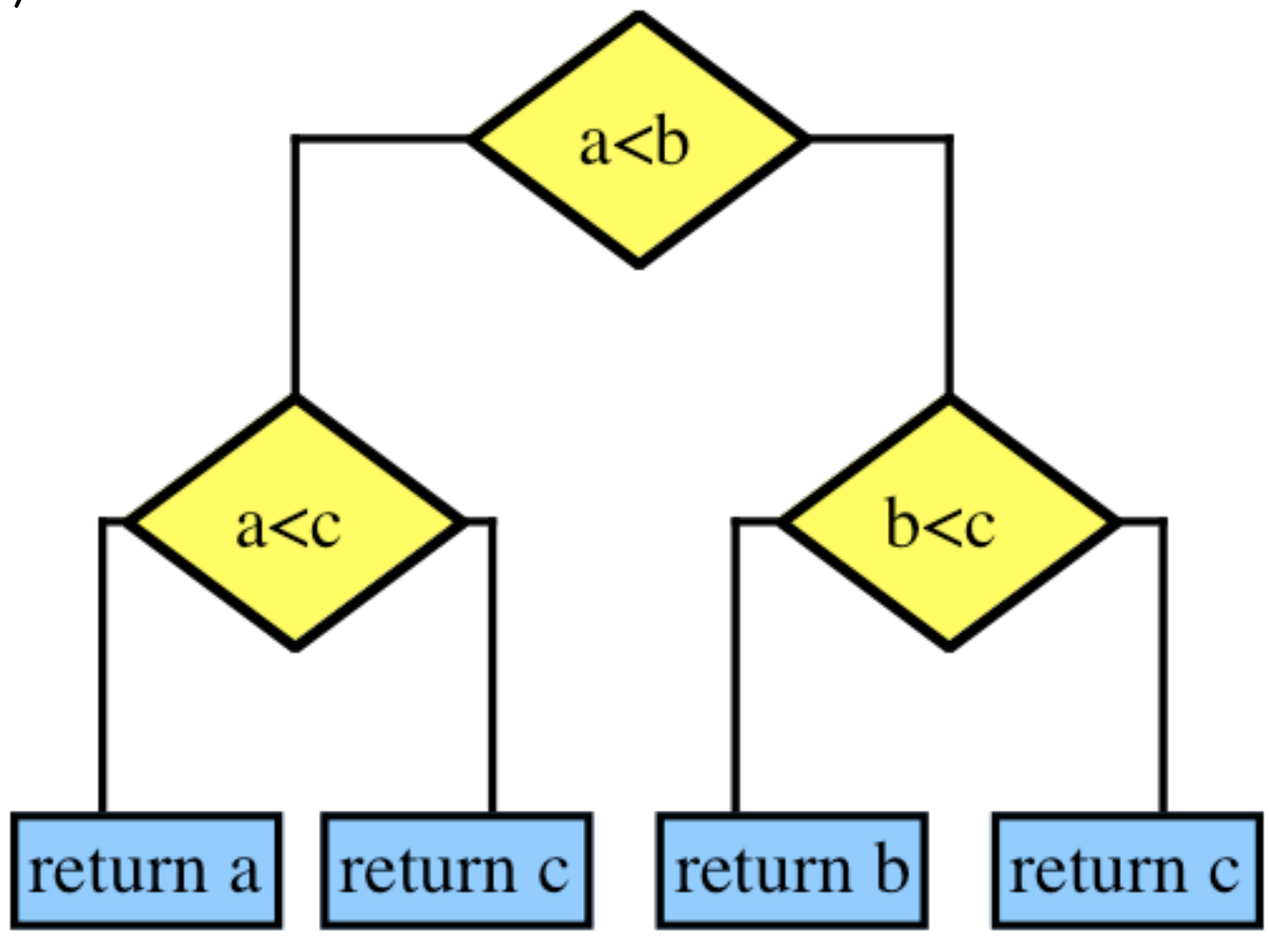
```
if (a-b) printf("a ist ungleich b");
```

  - klassischer Fehler: a=0

```
if (a=0) …;   /* immer falsch */
if (b=5) …;   /* immer wahr */
```

- geschachtelte Verzweigung

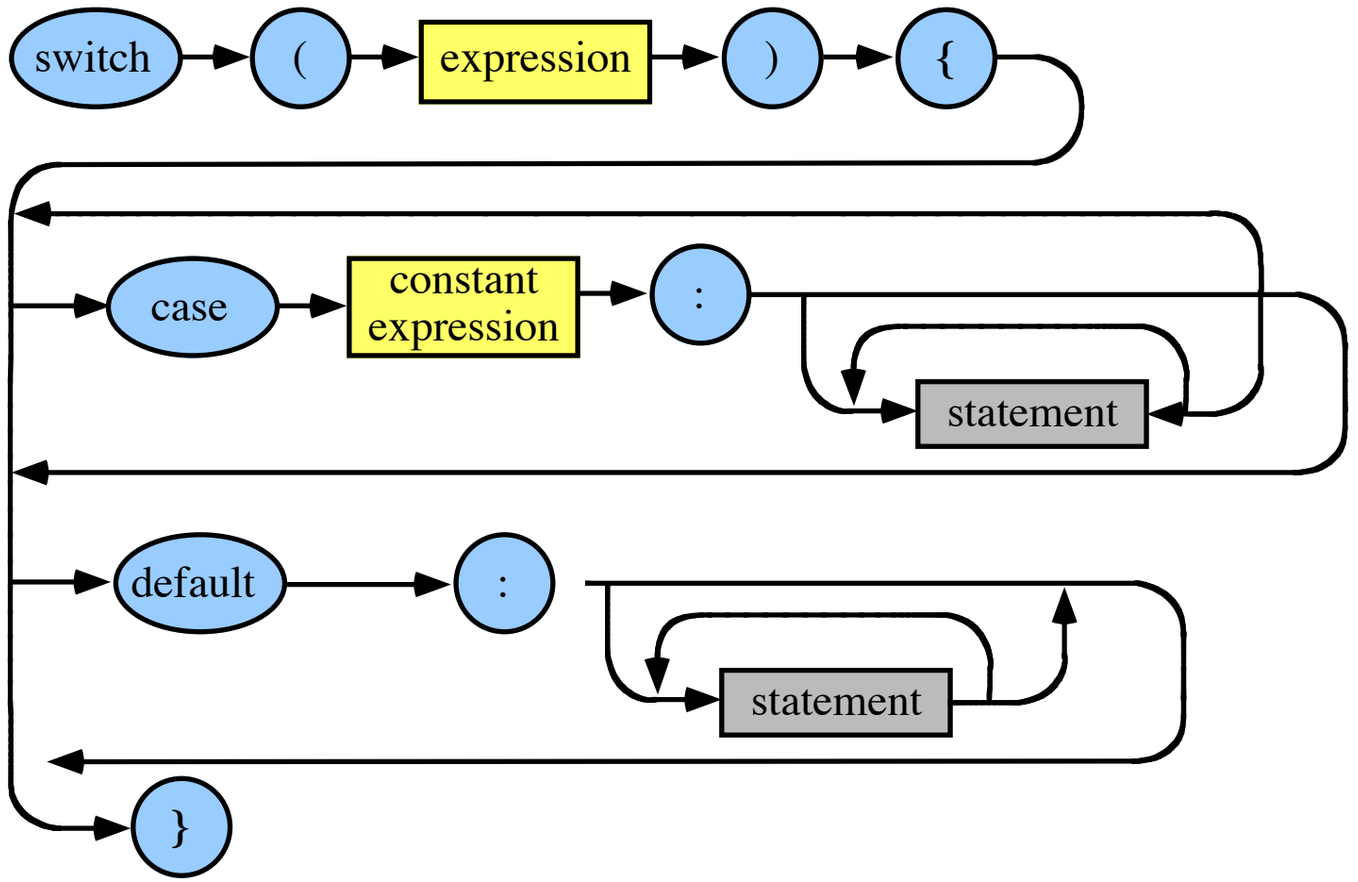
```
int min(a,b,c)
int a,b,c;
{  if (a<b)
      if (a<c) return a;
      else return c;
   else if (b<c)
      return b;
      else return c;
}
```



- Kaskadierte Verzweigung
  - mehr als 2 Fälle
  - nacheinander abfragen

```
int fallunterscheidung (eingabe)
char eingabe;
{
   if (eingabe == 'A')
      return 1;
   else if (eingabe == 'B')
      return 2;
   else if (eingabe == 'C')
      return 3;
   else if (eingabe == 'D')
      return 4;
   else
      return -1;
}
```

- Mehrfach-Verzweigung



- expression wird ausgewertet
- Resultat wird mit constant expressions verglichen
- Ausführung wird bei 'true'-Zweig fortgesetzt

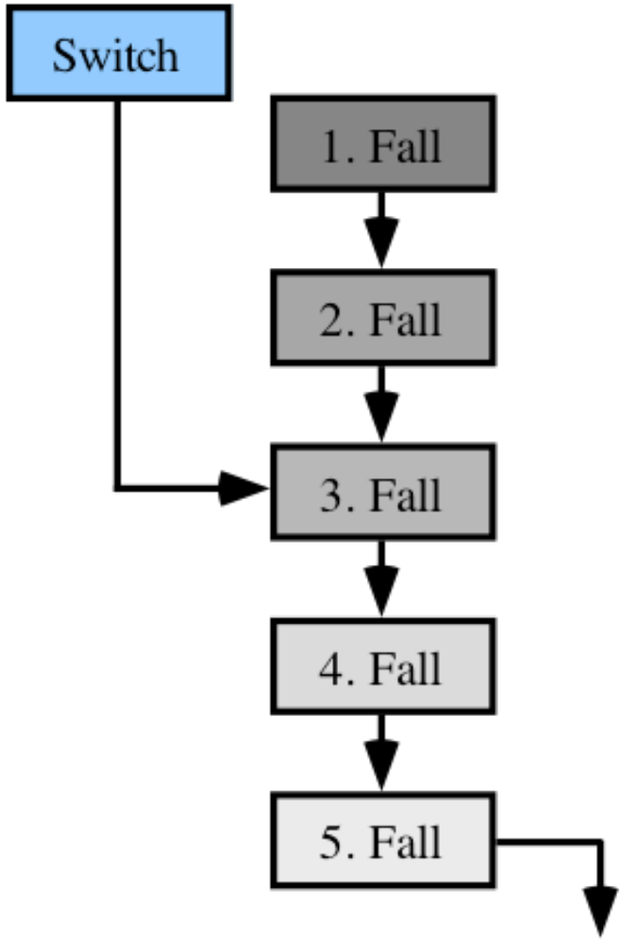
- Beispiel: Rechnen

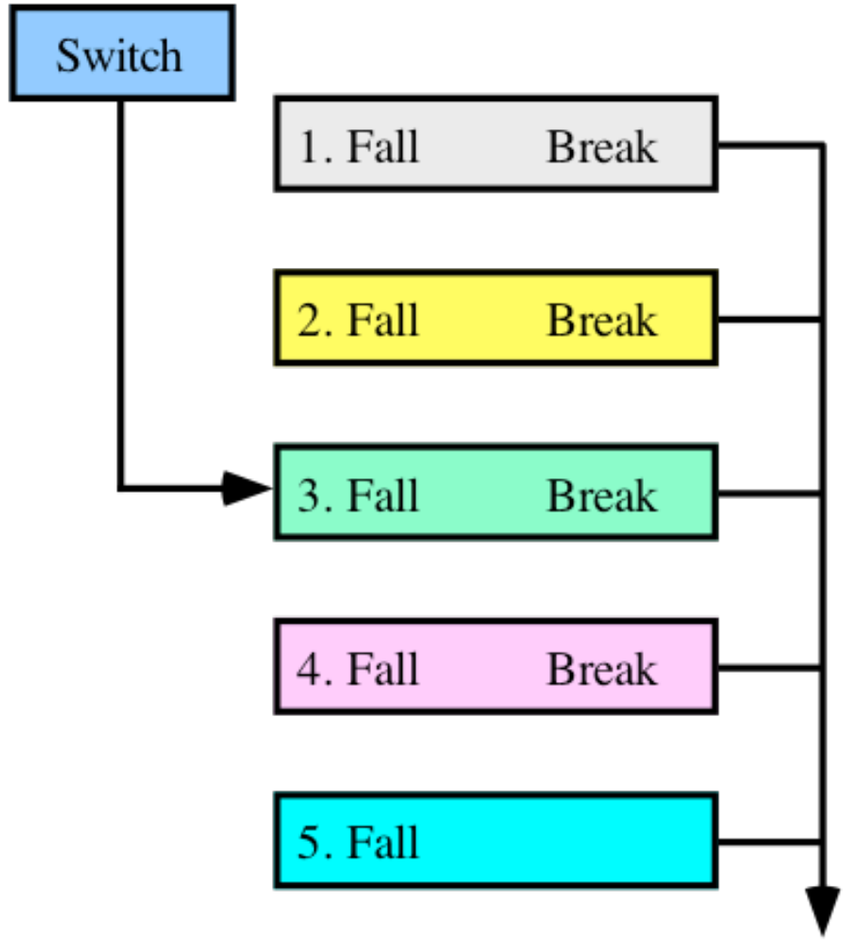
```
double evaluate(double op1,char operator,double op2)
{  switch (operator)
   {  case '+':  return op1 + op2;
      case '-':  return op1 - op2;
      case '*':  return op1 * op2;
      case '/':  return op1 / op2;
      default: printf("Aetsch, Fehler!!!");
}  return 0; }
```

- schlecht: alle folgenden Statements werden auch ausgeführt

```
double evaluate(double op1,char operator,double op2)
{  double erg=0;
   switch (operator)
   {  case '+':  erg = op1 + op2;
      case '-':  erg = op1 - op2;
      case '*':  erg = op1 * op2;
      case '/':  erg = op1 / op2;
      default: printf("Aetsch, Fehler!!!");
   }  return erg ; /* ist immer op1/op2 oder 0 */
}
```

- break ist wichtiges Element von switch
  - verhindert Ausführung der restlichen Fälle






```
switch (eingabe)
{  case 'A':  ausfuehren(); break;
   case 'B': beenden(); break;
   case 'D': drucken(); break;
   default : printf("Fehlerhafte Eingabe");
}
```

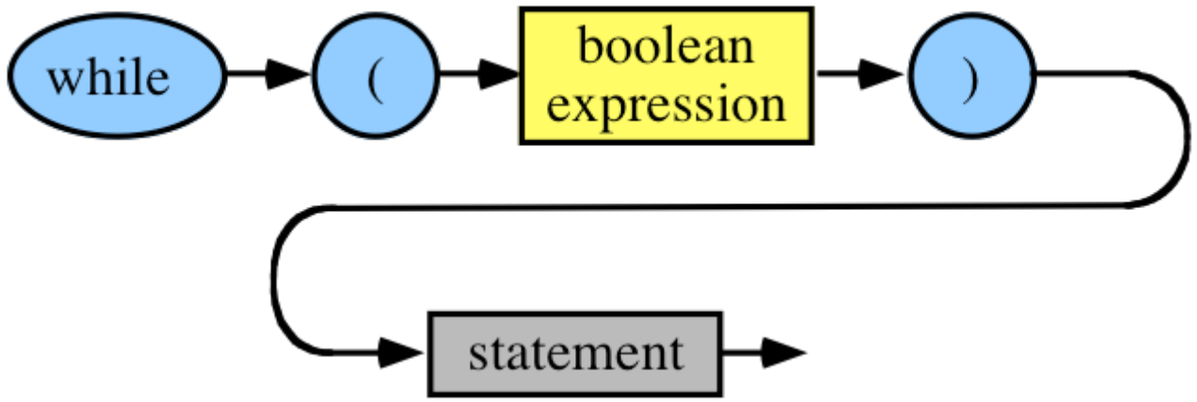
- Verbessertes Beispiel

```
double evaluate(double op1,char operator,double op2)
{  double erg=0;
   switch (operator)
   {  case '+':  erg = op1 + op2; break;
      case '-':  erg = op1 - op2; break;
      case '*':  erg = op1 * op2; break;
      case '/':  erg = op1 / op2; break;
      default: erg = 0; printf("Aetsch, Fehler!!!");
   }
   return erg;
}
```

- Beispiel: einfaches Menue

```
printf("Bitte waehlen Sie:\n S = Sichern\n \
           L = Laden\n N = Neu Anlegen");
scanf("%c",&eingabezeichen);
switch (eingabezeichen)
{  case 'S': case 's': savefile(); break;
   case 'L': case 'l': loadfile(); break;
   case 'N': case 'n': newfile();
}
```

- Schleife



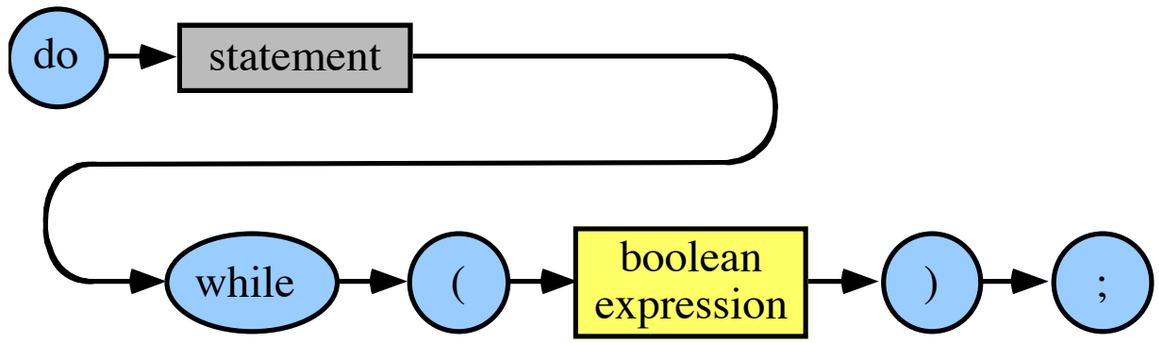
  - Bedingung wird am Anfang und nach jedem Durchlauf geprüft
- Beispiel n! (Fakultät, 2*3*4*5*…*(n-1)*n)

```
fak = 1; i = 1;
while (i <= n)
   {fak = fak * i; i = i + 1;}
```

- Beispiel: Leerstellen in der Eingabe zählen

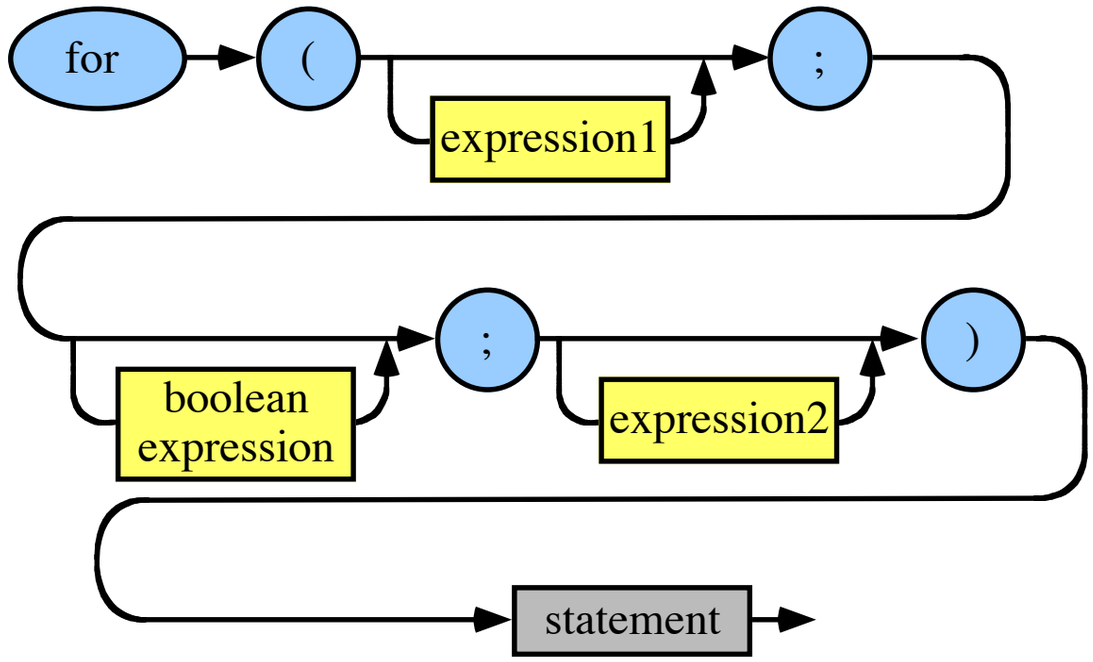
```
printf("Satz eingeben: \n");
ch = getchar();
while (ch != '\n')
{  if (ch ==' ') num_of_spaces++;
   ch = getchar();}
printf("Anzahl Leerstellen: %d",num_of_spaces);
```

- Schleife mit Test am Ende
  - erst ausführen, dann testen
  => 'statement' wird mindestens einmal ausgeführt



- Beispiel: nochmal n!

```
fak = 1; i = 1;
do
   {fak *= i; i ++;
   } while (i <= n);
```



- Zählschleife
  - expression1 initialisiert Schleifenzähler und andere Variable
  - boolean expression entscheidet über Ausführung
  - expression2 wird nach der Ausführung von Statement ausgeführt
  - sollte Schleifenzähler inkrementieren
- Beispiel: schon wieder n!

fak = 1;

```
for (i = 1; i <= n; i+=1) fak = fak * i;
/* oder */
for (fak=1,i = 1; i <= n; ) fak = fak * i++;
```

- 3 Expressions bieten viele Möglichkeiten

```
for (fak = i = 1; n-i++; fak *= i );
for (sum = i = 0; n-i++; sum += i );
/* hacker's paradise */
```

- Vorsicht mit der Anzahl

```
for (i=0; i<n; i++) /* n-mal ausgeführt*/
for (i=0; i<=n; i++) /* (n+1)-mal ausgeführt*/
```

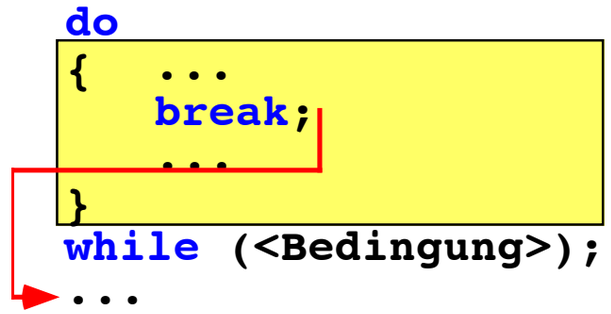
- Leerzeichen überspringen

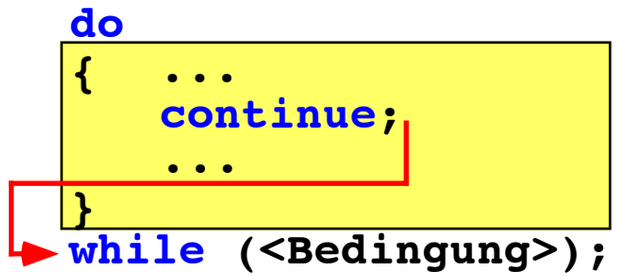
```
void skipspaces ()
{  int ch = ' '; /* kleiner Trick */
   for (; ch==' '; ch=getchar())
      ; /* Null Statement */
   ungetc (ch, stdin);
}
/* wollen wir nicht doch lieber while verwenden? */
```

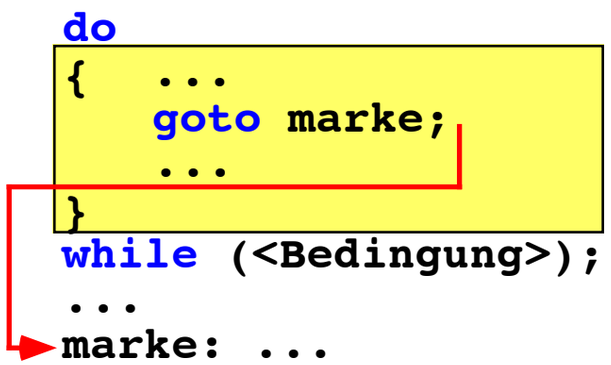
- Geschachtelte Schleifen
  - innere Schleife vollständig ausgeführt
  - mehrstufig

```
int main ()
{
   int j,k;
   printf("    1   2   3   4   5   6   7   8   9   10\n");
   printf("    ----------------------------------\n");
   for (j=1; j <= 10; j++)
   {
      printf("%5d|",j);
      for (k=1; k<=10; k++)
         printf("%4d", j*k);
      printf("\n");
   }
   return 0;
}
```

- Sprünge: the good, the bad and the ugly
  - break, continue, goto <label>
  - direkte Einflussnahme auf den Kontrollfluss
  - Programm kann auch anders formuliert werden
- break bricht ab
  - umfassende Schleife
  - switch
  - Ausführung mit Statement nach Schleife / switch fortsetzen
  - immer nur die jeweils innerste Schleife (bzw. switch)
- continue setzt fort
  - bleibt in der Schleife
  - Schleifentest + evtl. nächste Iteration
- goto springt zu einer Marke
  - Marke definiert: `ich_bin_ein_marke: <statement>`
  - `goto ich_bin_ein_marke;`



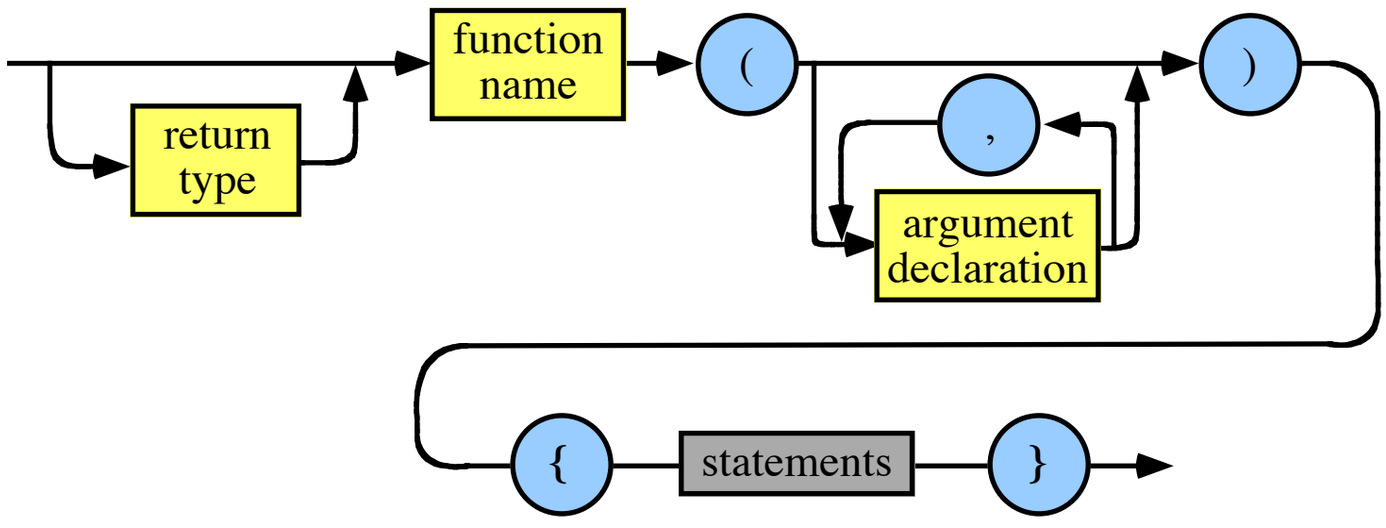




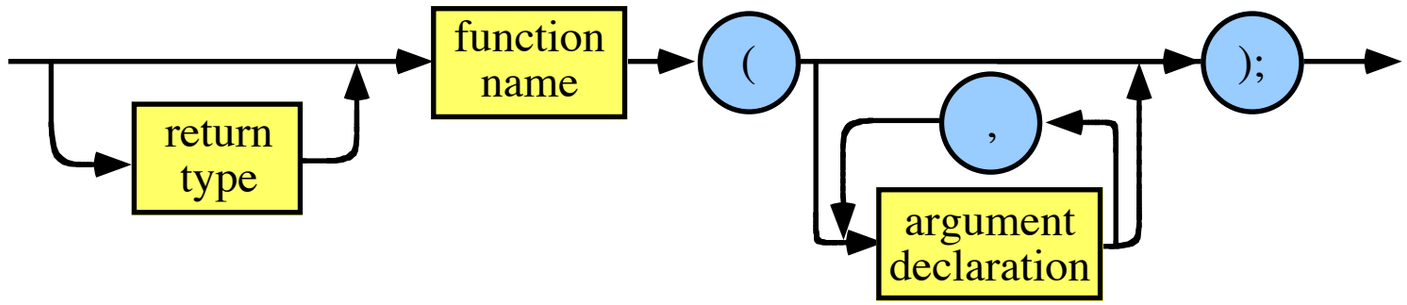
Funktionen im Detail

- Definierte Aufgaben gesondert programmieren
  - stepwise refinement
  - Problem zerlegen
  - häufig gebrauchte Komponenten
  - Werkzeugkasten
  - nur aktuelle Werte unterscheiden sich
  - Abstraktion
- Definition
  - Anzahl und Typ der Argumente
  - Statements im Rumpf der Prozedur

```
int max (int a, int b)
{
   if (a>b) return a;
   else return b;
}
```



- Keine Funktionen in Funktionen
- Deklaration vor der ersten Verwendung
  - falls Funktion vor der Deklaration aufgerufen wird
  - oder in Header-Dateien

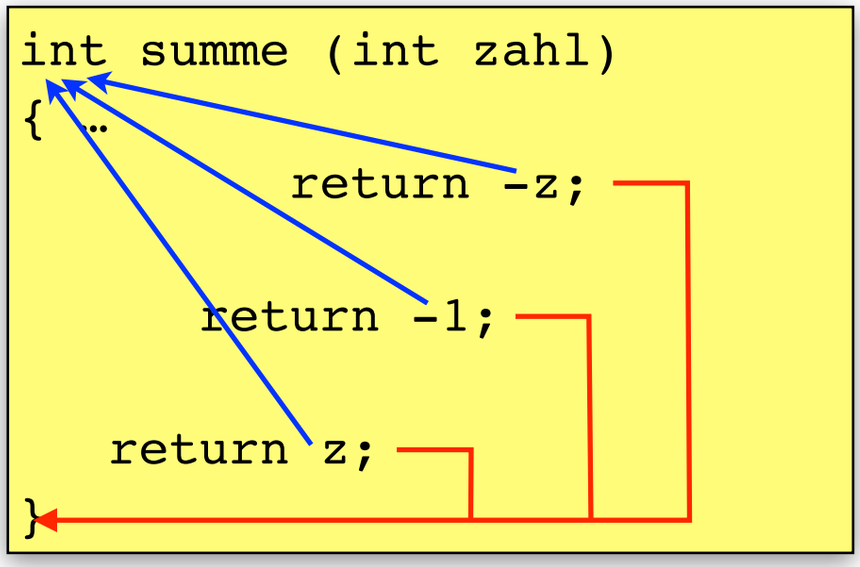



```
int f2 (int m, int n, int p); /* Deklaration */
...
...
int f1 (int a, int b)
{ ...
  f2(a,13,b+3);
  ...                    }
...
int f2 (int m, int n, int p) /* Definition */
{ ...
  f1(m,13);
  ...                    }
```

- Parameter
  - automatische Konvertierung ähnlich wie bei Zuweisung
  - es werden Werte übergeben
  - keine Variablen!
  - Werte werden in 'neue' Variablen gelegt
  - Trennung Wert-Variable
- Beispiel Werte-Parameter

```
void f(int argument)
{ argument = 3;}

int main ()
{  int a = 2;
   f(a);
   printf("%d\n",a);  /*wetten,dass 2 gedruckt wird?*/
   return 0;
}
```

- Rückgabewert
  - mit Funktionsresultat
  - return-Statement
  - Funktion wird sofort verlassen
  - mit Ergebnisrückgabe



- Resultate mit Seiteneffekt erzeugen
  - Parameter als Zeiger auf die Variablen
  - zeigen auf 'Behälter' für Rückgabewerte
- Beispiel: Vertauschen von 2 Variablen

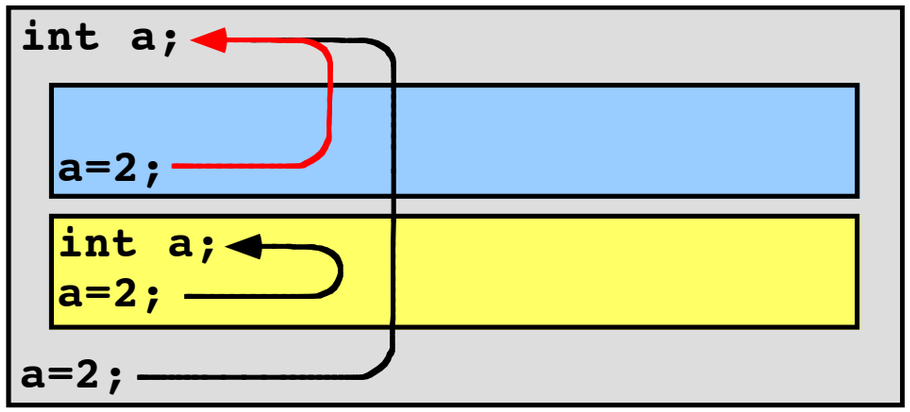
```
void swap (int *x,int *y)

{  int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

main()
{  int a=2, b=3;
   swap(&a,&b);
   printf("a ist %d\t b ist %d\n",a,b);
}
```

- Siehe scanf()

```
int f (int z)
{   z = 42;
}          z   42
```

```
int f (int *z)
{   *z = 42;
}

int main ()
{   int platz;
    …
    f(&platz);
    …
}        platz   42
```

- Gültigkeit der Namen
  - Parameter und lokale Variablen
  - lokal gültig
  - überdecken weiter aussen definierte Namen



- void
  - Funktion ohne Resultat
  - Spezialtyp als Füllelement

```
void licht_an(int welches)
{…}
```

- Ein komplettes Programm

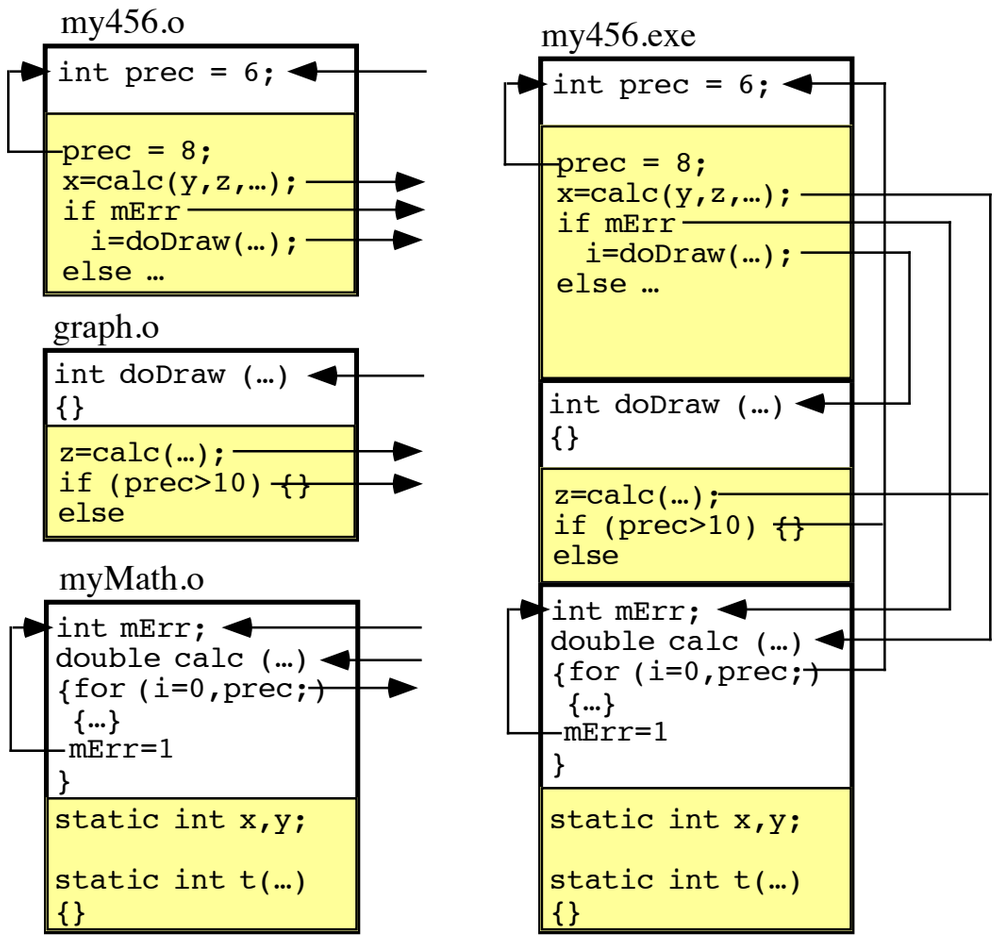
```
#include <stdio.h>

int summe (int zahl)
{  if (zahl >0) return zahl + summe(zahl-1);
   else return 0;
}
void allesummen (int max)
{ int lauf = 0;
   for (lauf = 1; lauf <= max; lauf++)
      printf("%3d: %6d\n",lauf,summe(lauf));
}
int einlesen()
{ int zahl;
   printf("Bitte Obergrenze eingeben:");
   scanf("%d",&zahl);
   return zahl;
}
int main ()
{  allesummen(einlesen()); return 0;}
```

Module

- Programme auf mehrere Dateien verteilen
  - Strukturierung des Problemes
  - Gruppenarbeit
  - getrennte Übersetzung
- Schnittstelle definiert
  - Leistungen des Modules
  - nach außen sichtbar
  - Typen und Variablen
  - Funktionen
  - Implementierung und Details versteckt
- Bekanntgabe an andere Module
  - Datei `regler.h` enthält Schnittstelle
  - Datei `regler.c` enthält Implementierung
  - Import mit include-Pseudobefehl: #include <modul.h>
- Übersetzung von Modulen
  - Parameter überprüfen,
  - Modul-Code und Referenzlisten erzeugen

- Zusammensetzen (Linken)
  - Zusammenkopieren der Module
  - Auflösen externer Referenzen: Einsetzen von Speicheradressen

Compiler Linker



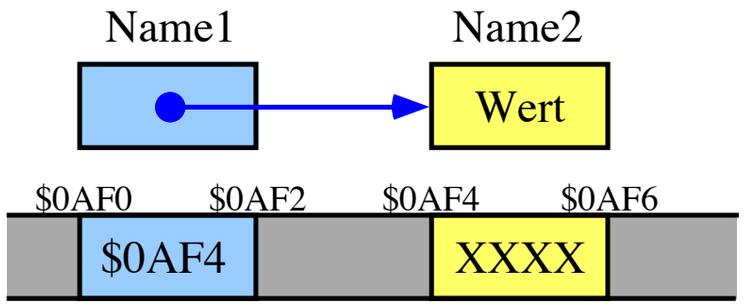
- Laufzeitumgebung: besondere Module
  - kommen mit dem Compiler
  - gehören oft zum Betriebssystem
  - standardisiert: ANSI, ISO, POSIX, …
- Vorgefertigte Funktionen
  - Standardwerkzeuge
  - Mathematische Funktionen
  - Ein/Ausgabe
  - String-Verarbeitung
  - Datum und Zeit
- Mathematische Funktionen: math.h

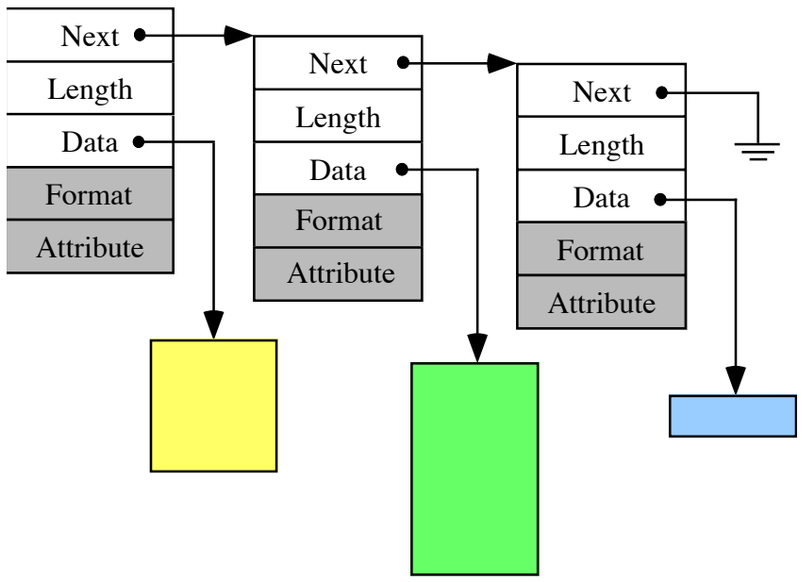
  sqrt(), log(), exp(), sin(), cos (), …
- Ein/Ausgabe: stdio.h
  - scanf(), printf(), getchar()
  - Datentyp FILE => Dateien
- String-Verarbeitung siehe unten

Arrays und Pointer

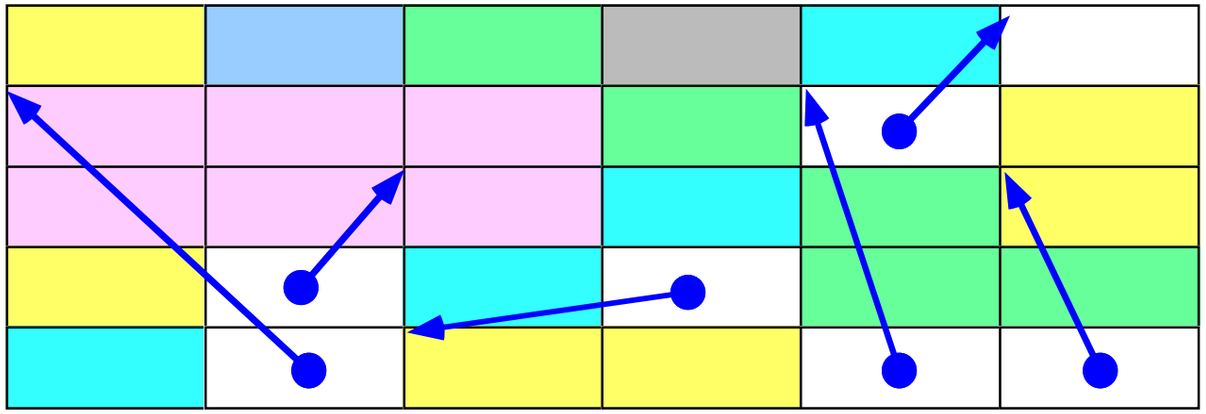
Zeiger (Pointer)

- Namen
  - identifizieren Objekte
  - von Menschen benutzbar
  - DNS: www.froitzheim.com
- Adressen
  - von Maschinen manipulierbar
  - schwer von Menschen verständlich
  - IP-Nummer: 134.60.77.64
  - 415-653-9516
- Referenzen
  - Verweis (Hinweis) auf andere Elemente
  - Bsp: Stellvertreter, Anrufumleitung
  - Hypertext-Referenzen
  - Listen, Bäume, …
  - Strukturinformation im Textprogramm
  - Alias/Verknüpfung (z.B. Windows 95, 98)





- Zeigervariablen
  - zeigen auf beliebige Datenobjekte

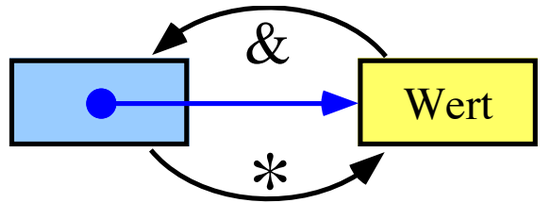


- '*' in Deklarationen und Definitionen

```
int *p; /* kein Speicherplatz für int! */
char ach, bch, *pch;
…
p = (int*)98;    /* Wo mag das wohl sein? */
p = p+1;       /* nun zeigt p auf Speicher 100 */
               /* oder auf 102? */
               /* Antwort: 98 + sizeof(int) */
```

- Address-of Operator '&'

```
char ach, bch, *pch;
ach = 'X';
pch = &ach;
scanf("%f",&messwert)
```



- Dereferenzierung mit '*'

```
printf("pch zeigt auf: %d",*pch); /* druckt X */
pch = (char*)100;
printf("Speicherzelle 100: %d",*pch); /* druckt ? */
```

- Bedeutung von '*' entgegengesetzt in:
  - Definition / Deklaration: 'pointer-to'
  - Expression: 'content-of'
- Leerer Zeiger == NULL

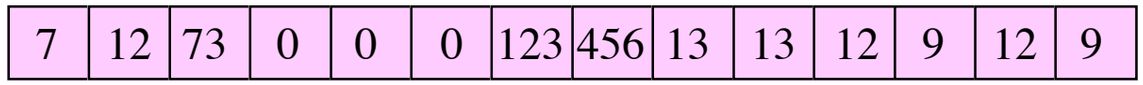
```
if (next == NULL) printf("Fertig");
```

- Pointer als Parameter
  - ermöglichen Rückgabe von Resultaten

```
int raten(int *wert, int min, int max)
{ int try = 0;
   do {  printf("\nBitte Eingabe: ");
         scanf("%d", wert);
         try = try + 1;}
   while (*wert < min || *wert > max);
   return try;
}
int eingabe;
/* Aufruf */
punkte = punkte + raten(&eingabe,17,99);
printf("der Kandidat hat %d geraten", eingabe);
```

- Übergabe großer Datenmengen ohne Kopie
  - Kopien kosten Zeit und Platz
  - aber: Veränderung in der Funktion evtl. unerwünscht

Arrays



…

- Arrays sind Vektoren, Matrizen,
- Definition

```
<typ> <name> "[" <dimension> "]"
            {"[" <dimension> "]"};
```

  - Beginn immer bei 0
  - Dimension ist Längenangabe

```
int vektor [100];
float matrix [zeilen][spalten];
```

- Initialisierung

```
int rot [3] = {255,0,0};
char name [32] = "Hallo"; /* 32 byte */
char name2 [] = "Hallo";  /* 6 byte */
```

- Verwendung
  - meist elementweise

```
vektor[index]
matrix[zeilenindex][spaltenindex]
```

- Dualität Array - Pointer
  - Array-Ident zeigt auf erstes Element
  - zunächst das Erste (array[0])
  - abgekürzte Schreibweise

  vektor[0] entspricht: *vektor
  &vektor[0] entspricht: vektor

  vektor + <expr> entspricht: &vektor[<expr>]
  *(vektor + <expr>) entspricht: vektor[<expr>]

  vektor + index
  &vektor[5]-2

  /* aequivalent */
  matrix[i][j]
  *(matrix[i] + j)
  ((matrix + i) + j)

- Länge des Grundtyps wird beachtet

- Als Parameter für Prozeduren
  - Prozedurdefinition ohne Wissen über Länge
  - Länge fest oder als Parameter

```
void feldlesen(char *dasFeld, int feldlength)
{  int index;
   for (index=0; index < feldlength; index++)
      {printf("Element %d eingeben: ",index);
       scanf("%c", &dasFeld[index]);}
}

char einVektor[100];

int main()
{
   feldlesen(einVektor,100);
   drucken(einVektor,100); /* noch programmieren */
   return 0;
}
```

- Strings
  - char-Array: `char name[32];`
  - Adressierung wie bei Arrays
  - Ende des Strings durch '\0' ('null terminated')
- Stringkonstanten in "…"

```
*"Konrad"           entspricht 'K'
*"Konrad"+2         entspricht 'M'
*("Konrad"+2)       entspricht 'n'
"Konrad"[6]         entspricht '\0'
"Konrad"            /* Adresse */
```

- Zuweisungen nur elementweise

```
name[0]='A'; name[1]='l'; name[2]='f';
name[3]='\0';
printf("%s\n",name);
```

```
...
Alf
```

- Funktionen zur Stringverarbeitung in string.h
  - strncpy, strlen, strcat, strcmp, …
- strncpy (strcpy gefährlich …)

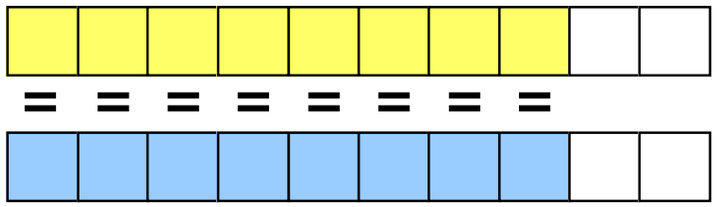
```
char name [32];
printf("%s",name);
strncpy(name, "Jeremias Jammermeier", sizeof(name)-1);
printf("%s\n",name+8);
```

```
...
Alf Jammermeier
```

- Idee für strlen

```
int strlen(char *theStr)
{ int length = 0;
   while (*theStr != '\0') /* 0 reicht auch */
      {length++; theStr++;}
   return length;
}
```

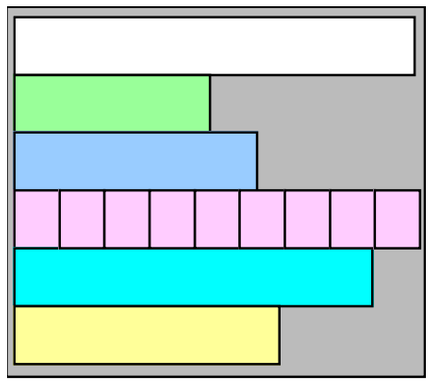
- Idee für strcmp



```
int strcmp(char *aStr, char *bStr)
{ if (strlen(aStr)!=strlen(bStr)) return 0;
   while ((*aStr == *bStr) && *aStr)
      {aStr++; bStr++};
   if (*aStr=='\0') return 1;
   else return 0;
}
```

Structures

- C-Form von Records
  - Objektbeschreibung, Personaldaten
  - heterogene Datenelemente
  - besonders in Datenbanken
  - verschiedene Felder
  - evtl. Zeiger



- Typ struct
- Strukturdeklaration

```
struct Wagenbeschr
   {  char name[32];
      enum helptype {Schlaf,Abteil, …, Kessel, …} Wagentyp;
      int Baujahr;
      struct Wagenbeschr *next;
   };
```

- Variablendefinition

```
struct Wagenbeschr ShellWagen, BPWagen, Aralwagen;
```

- Zuweisung

```
strncpy(ShellWagen.name,"02 17 850 0320-1",31);
ShellWagen.Wagentyp = Kessel;
ShellWagen.Baujahr = 1972;
ShellWagen.next = NULL;
Aralwagen = {"33 80 774 3128-0",Kessel, 1988, NULL};

BPWagen = Shellwagen;
```

- Vergleich nicht als Einheit

```
if (BPWagen == Shellwagen) /* falsch */
/* illegal structure operation */
```

- typedef macht eine bestimmte structure zum Typ

```
typedef struct Wagenbeschr WagenType;

WagenType EssoWagen;

/* natuerlich auch fuer Standardtypen */

typedef double Gewicht;
Gewicht maximal_zulaessiges_Gesamtgewicht;
```

- Initialisierung

```
Wagenbeschr SpeiWa = {"61 85 88-70 001-2",Speise, 1978, NULL};
Wagenbeschr PersWa = {.Wagentyp = Personen,
                      .baujahr =1979}; /* rest 0 */
```

- Funktionsaufruf
  - {} Auf Parameterposition möglich
  - mit Typecast

```
typedef struct {char *name; int number} X;

void fn1(const X *x);
{…;}

fn1(&a); /* benannte Variable */
fn1(&(X){"name", 42}); /* anonyme Variable */
fn1(&(X){.name="name", .number=42});
fn1(&(X){"name", random()});
fn1(&(const X){.name="name", .number=42});
        /* anonyme Konstante */
```

- Pointer auf structures

```
WagenType *wagenPtr;
wagenPtr = &EssoWagen;
*wagenPtr = BPWagen;
```

- Zugriff auf Komponenten
  - Problem: '.' bindet stärker als '*'

```
/* falsch: printf("%s",*wagenPtr.name) */
(*wagenPtr).Baujahr = 1990;
```

  - neuer Operator:

```
wagenPtr -> Baujahr = 1990;
printf("%s",wagenPtr -> name);
```

- Parameter und Resultat von Funktionen

```
WagenType *newcopy (Wagentype *derWagen)
{ WagenType *help;
   help = malloc(sizeof(Wagentype));
   if (help == NULL) printf("%s","Out of memory");
   else *help = *derWagen;
   return help;
}
```

Systematische Fehlersuche

- Software hat Fehler
  - komplexe logische Systeme
  - Programme mit 1 Million Zeilen und mehr
  - Interaktion mit anderen Programmen
  - unscharfe Eingabe
- Verifikation schwer evtl. unmöglich
  - unentscheidbare Probleme (=> Goedel)
  - vollständige Aufzählung der Randbedingungen
- Programm ordentlich aufschreiben
  - systematisch einrücken
  - aussagekräftige Namen
  - don't get fancy
- Programm oft ausführen
  - mit verschiedenen Parametern
  - auf verschiedenen Rechnern
  - andere Personen testen lassen

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. [B. Kernighan]

```
while (x == y) {
      something();
      somethingelse();
}
finalthing();
```

- Typische Probleme
- Bedingungen
  - Test auf Gleichheit: (a <= 0) oft besser als (a == 0)
  - "echt größer" (>) oder "größer-gleich" (>=)?
- while-Schleife
  - Endebedingung falsch
  - Seiteneffekte auf Endebedingung
- Variable nicht vorbesetzt
  - häufig bei erratischen Fehlern

```
int teiler;
scanf("%d",b);
a = b/teiler; /* Was steht wohl in teiler? */
```

  - Programm läuft nur einmal
  - Programm läuft nur nach bestimmten anderen Programmen
  - Programm läuft, wenn Variable eingefügt wird
- Pointer zeigt irgendwohin
  - NULL-Pointer, …
  - Speicher wird überschrieben

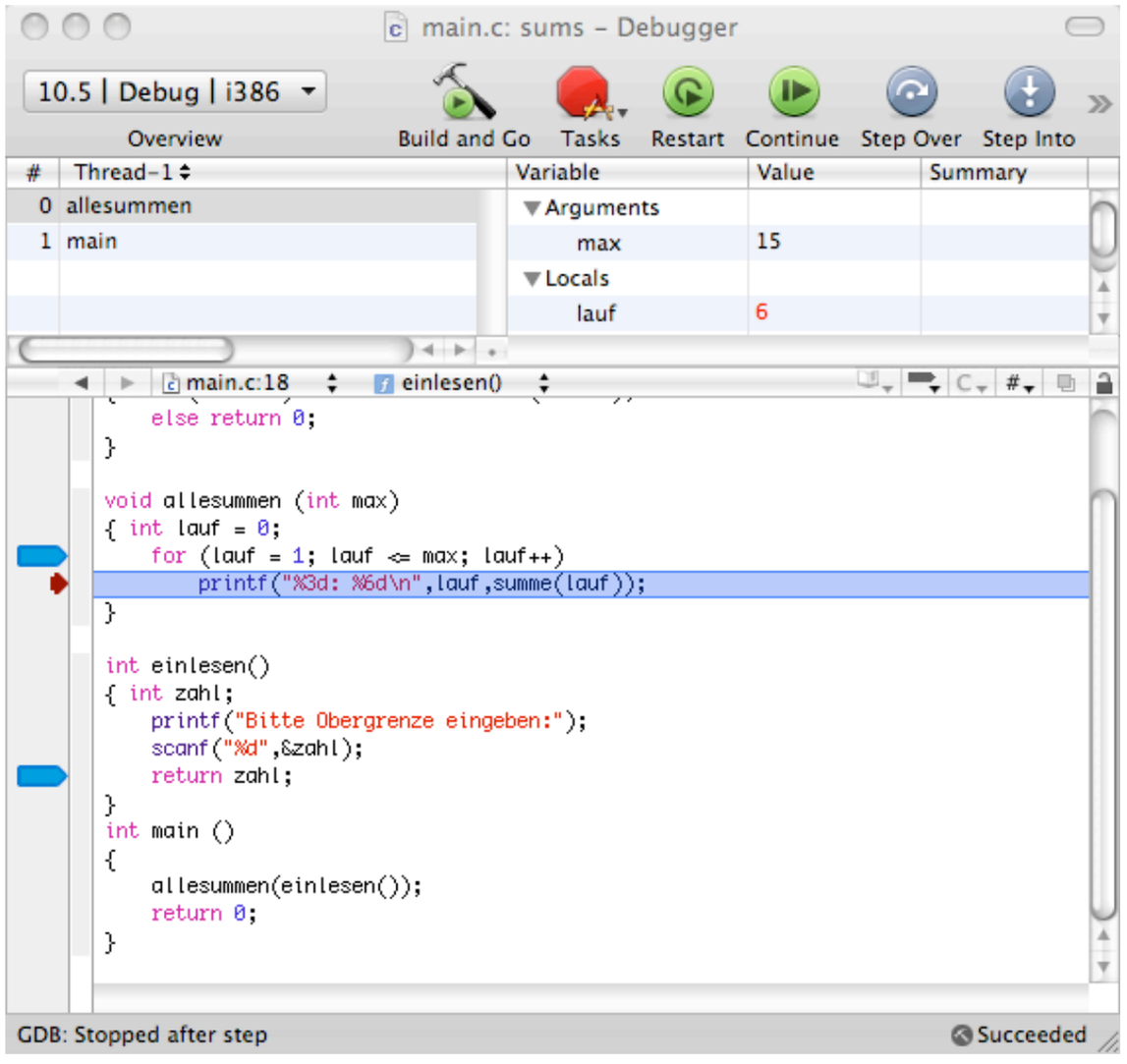
- Systematische Vorgehensweise
  - Beobachtung des Problemes (reproduzieren!)
  - Hypothesenbildung (warum?)
  - Messen (wenn - dann)
  - Beseitigen (programmieren)
- Wolfs-Zaun
  - Fehler eingrenzen
  - schrittweise eingrenzen mit printf()
- Haben die Variablen den richtigen Wert?
  - am Ende von Funktionen
  - nach jedem Statement
- Besondere Abfragen
  - Annahme: Variable sollte $\neq 0$ sein

```
if (<variable>==0) printf("Annahme verletzt\n");
```

oder: `assert(<variable>!=0); /* aus assert.h */`

  - Fehlermeldung und evtl. Abbruch
  - assertion (Zusicherung)

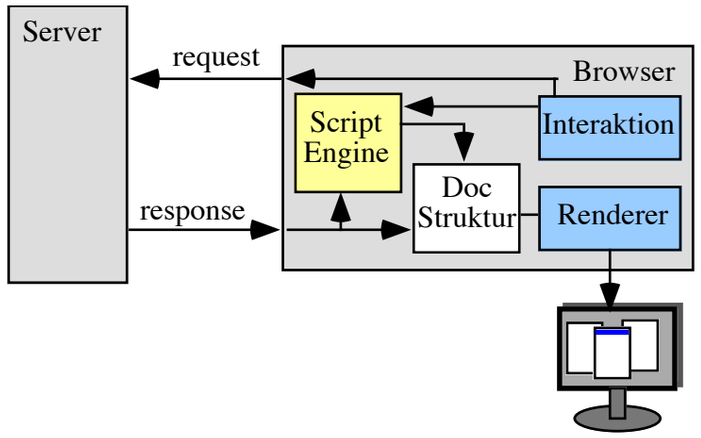
- Debugger
  - Variablen:Inspektion , Werte setzen
  - Aufrufkette
  - Breakpoints
  - trace
  - step

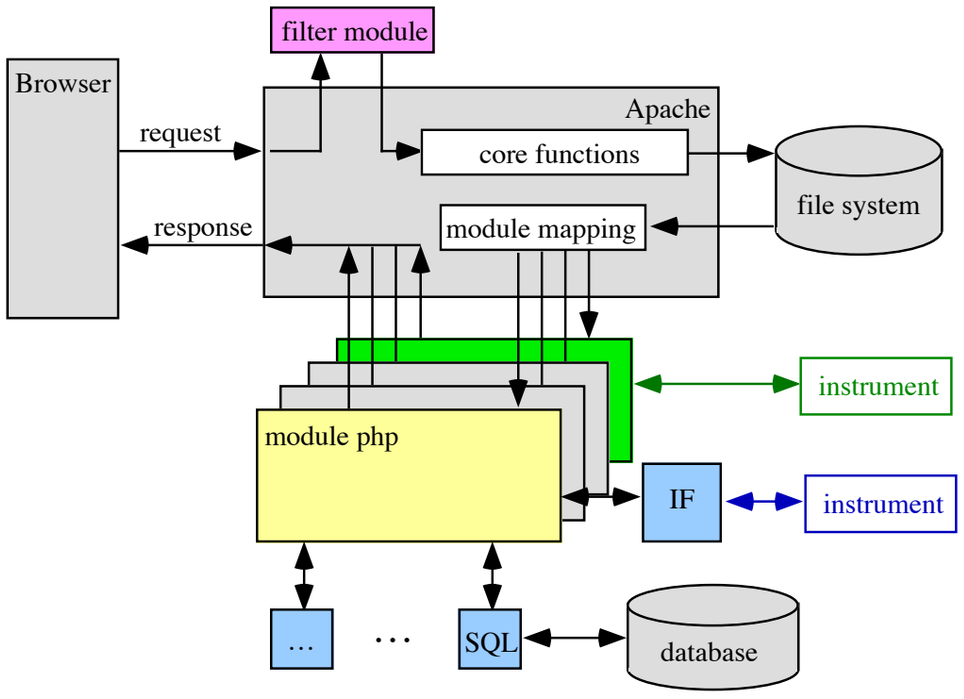


Reaktives Programmieren

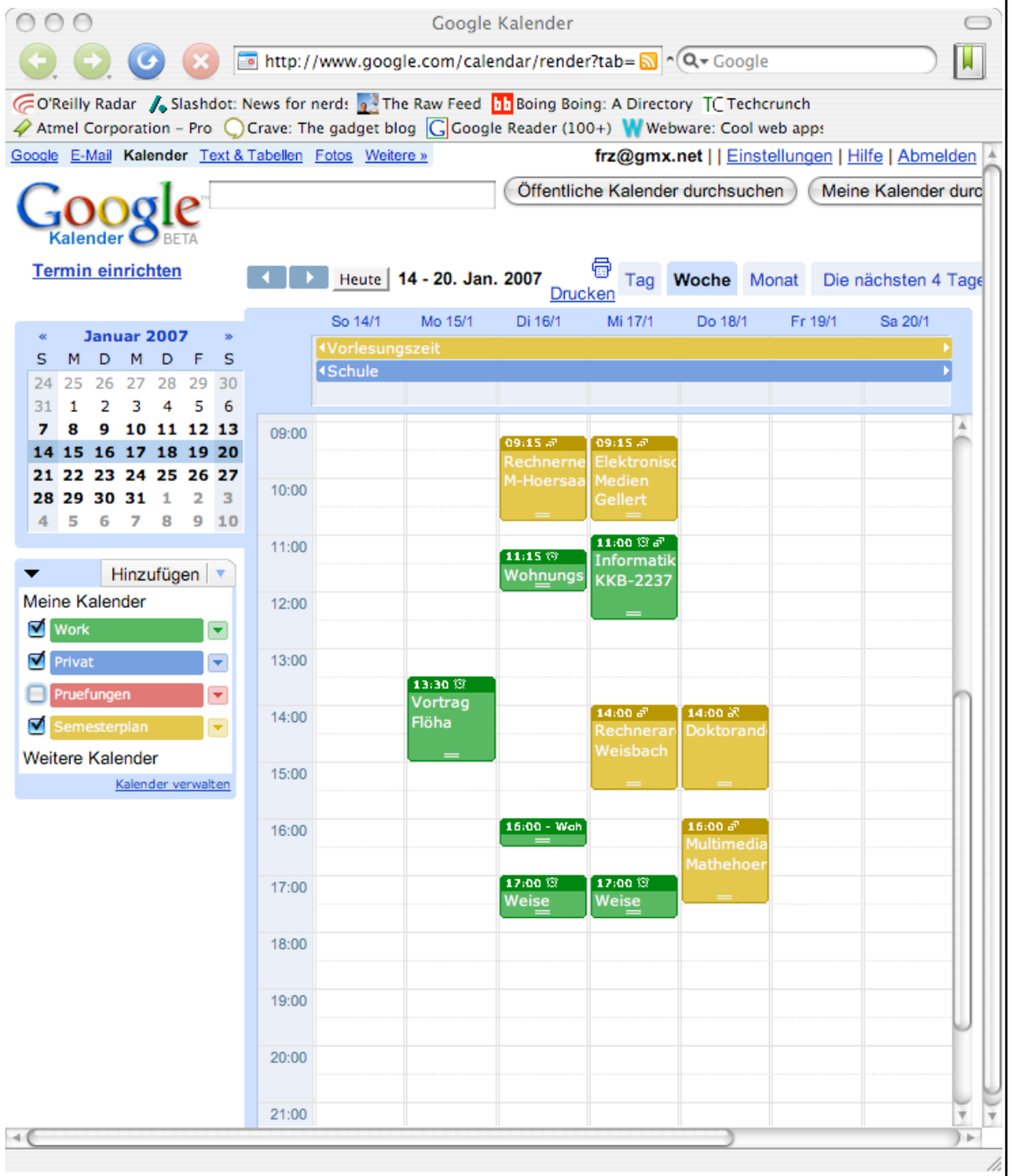
Ausführungsmodell WebApplikation

- Mikro-interaktiv
- Benutzeroberfläche
  - Browser
  - html mit Grafiken als Substrat
  - individuelles Verhalten: JavaScript
- Server
  - Datenbank + Geräte
  - Apache und Apache-Module
- Scripting
  - client-side: Javascript, Java Applets
  - server-side: ASP, php, Java und Beans
- Beispiele
  - http://rr.informatik.tu-freiberg.de
  - Shopping-Seiten



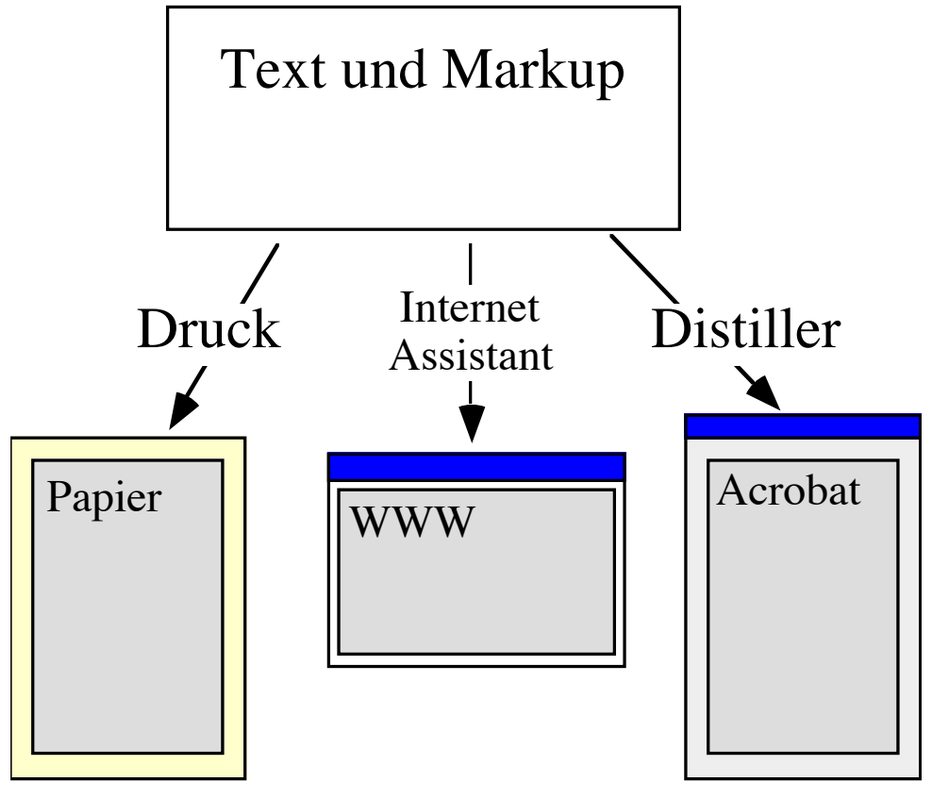


- Programmierparadigma
- Infrastruktur
  - Server:(Apache+php | JavaBeans)
  - Server:Datenbank
  - Client:(InternetExplorer | Firefox | Safari)
- Krümelware
  - kleine, vorgefertigte Bauteile (-> Objekte)
  - abgeleitete (erweiterte) Objekt
  - inkrementelles Programmieren
  - Einfügen und Erweitern statt Bauen
  - Gesamtkonzept? Architektur?
  - Datenfluß designen
- AJAX
  - Asynchronous JavaScript And XML
  - maps.google.com
  - flickr.com
  - docs.google.com, google calendar



HTML, die 'Sprache' des WWW

- Hypertext Markup Language
  - Berners-Lee 1989
  - Zweck: Verknüpfung von Dokumentation der Hochenergiephysik
  - keine Bilder - textbasierte Klienten
  - Standard Generalized Markup Language
  - Document Type Definition (DTD) von SGML
  - Hypertext-Referenzen: URLs eingebettet in das Dokument
  - http://www.w3.org/TR/REC-html40/
- Markup
  - logische Struktur für Text
  - Überschrift, normaler Paragraph, Zitat, …
  - Fußnote, Literaturverweis, Bildunterschrift, …
- Zuordnung der Attribute beim Satz
  - Autor produziert Inhalt und Struktur
  - Drucker setzt
  - Corporate Identity …
- HTML: ASCII-Text + `<A>`-Tag



- Beispieltext:

```
<HTML> <HEAD>
<TITLE> Ein HTML-Beispiel </TITLE>
</HEAD> <BODY>
<B>Dies</B> ist ein Hypertext Dokument.
<P>Mit einem Bild: <IMG SRC="bild.gif"> <BR> und einem
<A HREF="Beispiel1.txt"> Hyperlink </A> </P>
</BODY> </HTML>
```

Dies ist ein Hypertext Dokument.

Mit einem Bild:

und einem Hyperlink

- Weitere Elemente siehe z.B. http://www.selfhtml.org/
  - Listen, Stile
  - Formatierung

JavaScript

- Programmfragmente in HTML
  - Verbesserung von HTML-Seiten auf der Klienten-Seite
  - Fenstergröße und -Gestaltung
  - Menus, Effekte, …
  - Beispiel: http://maps.google.com
- Interpreter im Browser
- Eingebettet in HTML
  - script-Tag

```
<html><head><title>Test</title>
<script language="JavaScript">
<!--
  alert("Hallo Welt!");
//-->
</script>
</head><body>
</body></html>
```

- Oder in anderen HTML-Tags

```
<html> <head>
   <title>JavaScript-Test</title>
   <script language="JavaScript">
   <!--
    function Quadrat(Zahl)
     {var Erg = Zahl * Zahl;
      alert("Quadrat von " + Zahl + " = " + Erg);  }
   //-->
   </script> </head>
   <body> <form>
   <input type=button value="Quadrat von 6 errechnen"
      onclick="Quadrat(6)">
   </form> </body>
   </html>
```

- Eventhandler
  - Attribut in html-Tags
  - beschreiben Ausführungsbedingung
  - Aufruf einer JavaScript-Funktion
  - onload, onclick, onmouseover, onkeydown, …

- Sprache
  - Notation ähnlich Java
- Anweisungen
  - Zuweisungen

```
zahl = 0; zahl++; zahl+=1;
```

  - Bedingte Anweisungen und Schleifen

```
if (Zahl<0) zahl = 0;
while (idx<100) {…; idx++}
for(i = 1; i <= 100; i++)
      {…}
```

  - Funktionsaufrufe

```
alert("Und hier ein kleiner Hinweis");
```

  - Klammern mit {}

```
if (Ergebnis > 100)
{ Ergebnis =0; Neustart(); }
```

- Variablen: kein ordentliches Typenkonzept
  - Vereinbarung mit 'var'
  - Typ Zahl oder String
  - wird bei der ersten Zuweisung festgelegt

```
var Antwort = 42;
var Frage = "The Question for god …"
```

  - global oder in Funktion lokal
- Objekte: Eigenschaften und Methoden
  - selbstdefiniert oder vordefiniert in Umgebung
  - window, document, images, links, array, …
  - Objekt erzeugen mit `var einobjekt = new <object>;`

```
var einFenster = window.open(<ref>,<titel>,<parms>);
```

  - Datenstruktur (Felder = Eigenschaften)

```
einFenster.name = "Lustig";
```

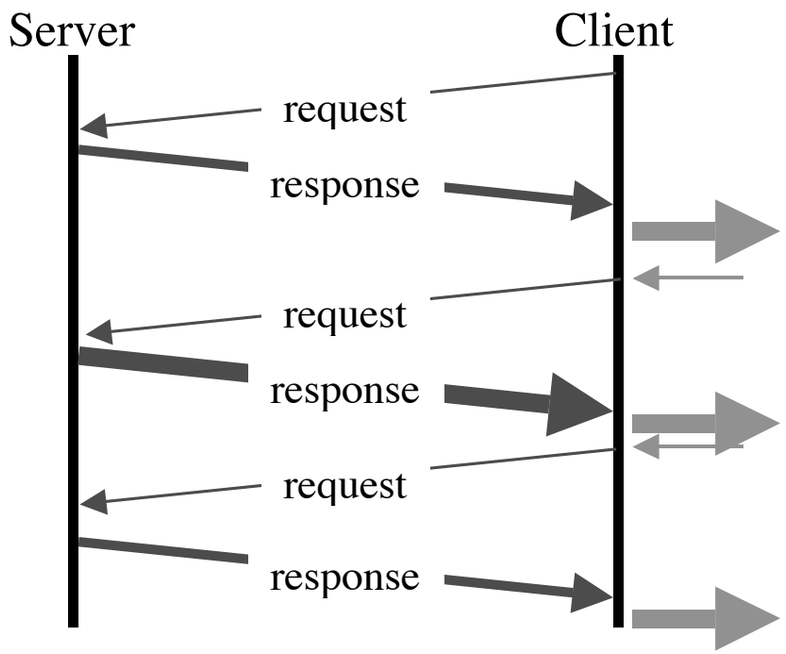
  - Methoden zur Manipulation: print, blur, moveTo, scrollBy, …

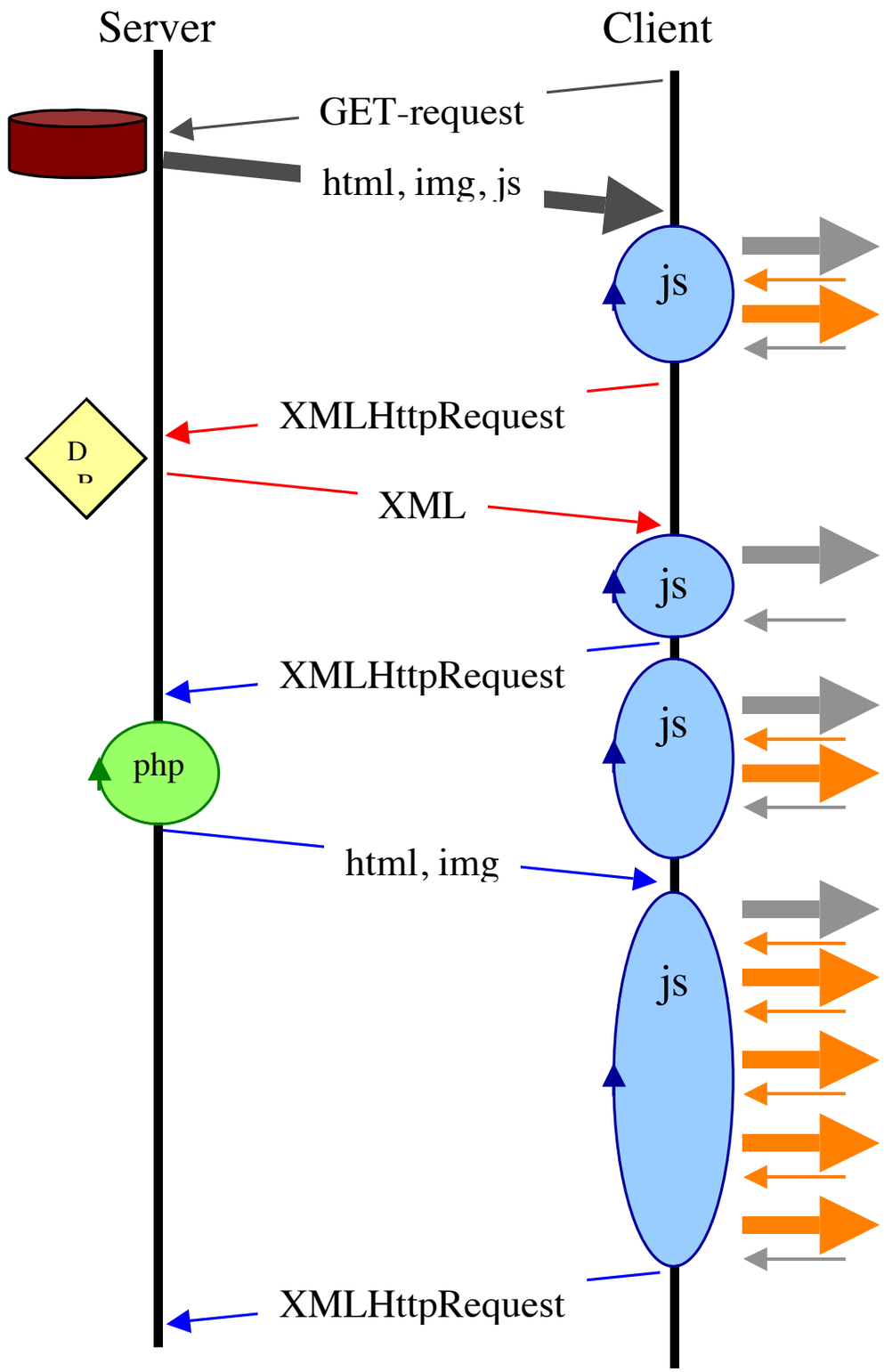
```
einFenster.resizeTo(500,400);
```

- Funktionen
  - Anweisungsblock mit Variablen
  - Aufruf aus anderen Funktionen und in Eventhandlern

AJAX

- Architektur von AJAX-Applikationen
- Programm im Browser (Client)
  - JavaScript
  - AJAX Libraries
- Server
  - Webserver, Datenbank
  - Serverside Scripte
- Leichtgewichtige Kommunikation
  - XMLHttpRequest
  - synchron und asynchron





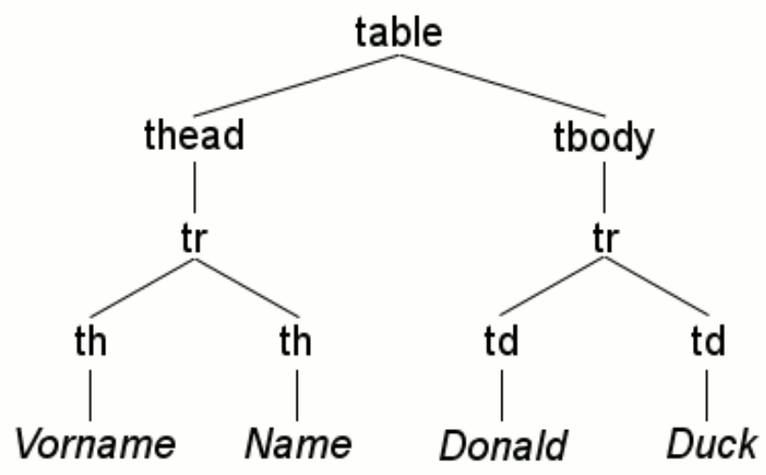
- XMLhttpRequest
- JavaScript Klasse
  - erstmals in IE 5
  - Interface für Http
  - ohne Benutzerinteraktion
  - synchron und asynchron
- Properties
  - readystate, status
  - onreadystatechange
  - responseText
- Methoden
  - open
  - send

```
function createXMLHttpRequest() {
 try {return new ActiveXObject(
                "Msxml2.XMLHTTP"); } catch(e) {}
 try {return new ActiveXObject(
                "Microsoft.XMLHTTP"); } catch(e) {}
 try {return new XMLHttpRequest(); } catch(e) {}
 alert("XMLHttpRequest not supported");
 return null;}
```

```
var xhReq = createXMLHttpRequest();
xhReq.open("get", "sumget.phtml?num1=10&num2=20", true);
xhReq.onreadystatechange = function() {
   if (xhReq.readyState != 4)  { return; }
   var serverResponse = xhReq.responseText;
   ...};
xhReq.send(null);
```

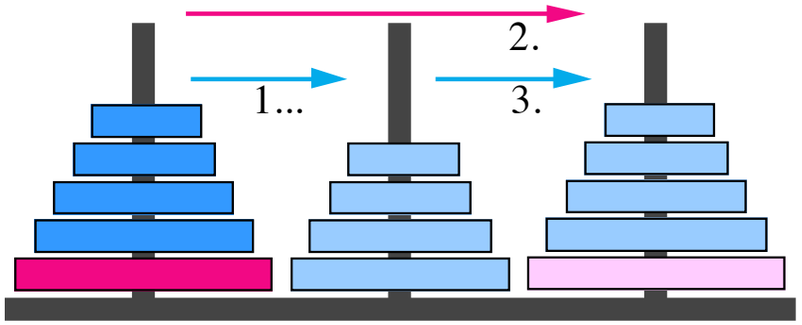
- Das X in AJAX
- Markup
  - Markup: Trennung Struktur - Inhalt
  - logische Struktur der Seite
  - Bsp: Überschriften, Absätze, Zitate, …
- XML: eXtensible Markup Language
  - Syntax für Markup
  - Semantik in XSL oder CSS
- Document Object Model DOM
  - baumartige Struktur der Dokumente
  - Zugriff auf Dokumenteninhalt (=Objekte)
  - Inhalt, Struktur, Stil
- AJAX
  - XML als ein Transfer-Format für Inhalt
  - Manipuliert DOM-Knoten
  - Einfügen, Löschen, Ändern
  - Browser 'rendert' Dokument

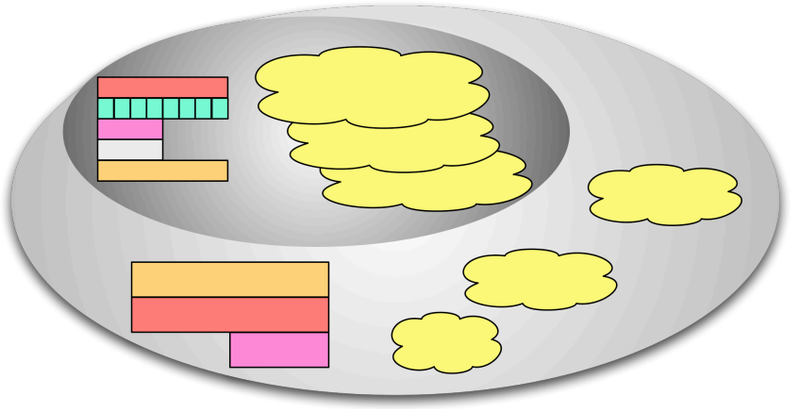
```
<table>
   <thead>
     <tr>
       <th>Vorname</th>
       <th>Name</th>
     </tr>
   </thead>
   <tbody>
     <tr>
       <td>Donald</td>
       <td>Duck</td>
     </tr>
   </tbody>
</table>
```



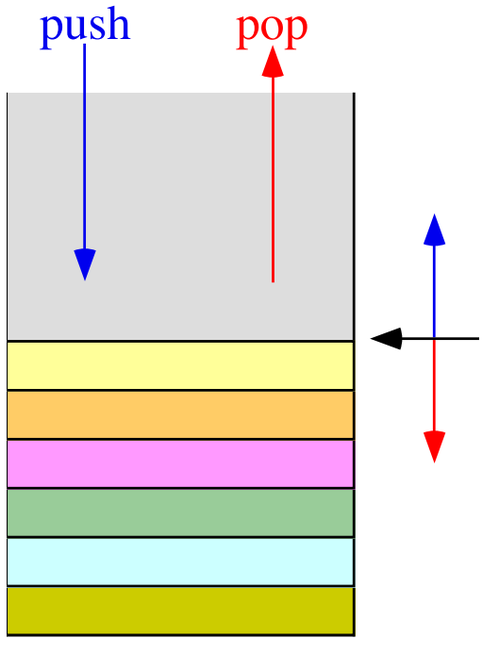
KAPITEL 4

Algorithmische Komponenten



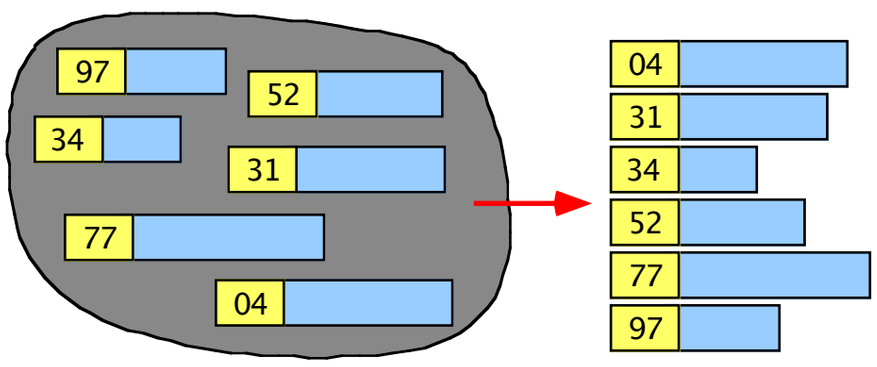


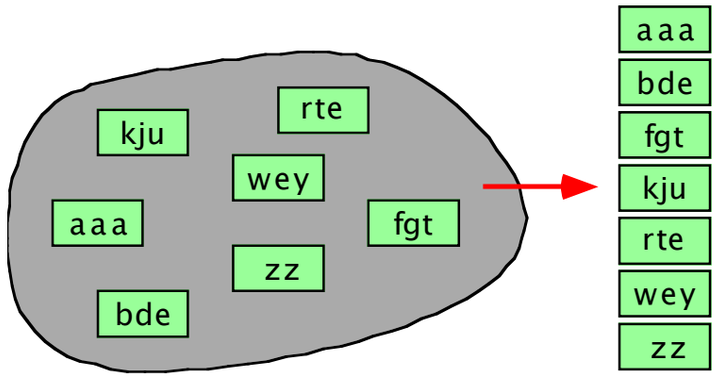
- Sortieren
- Listen
- Bäume
- Speicherverwaltung
- Objekte und Patterns



Sortieren

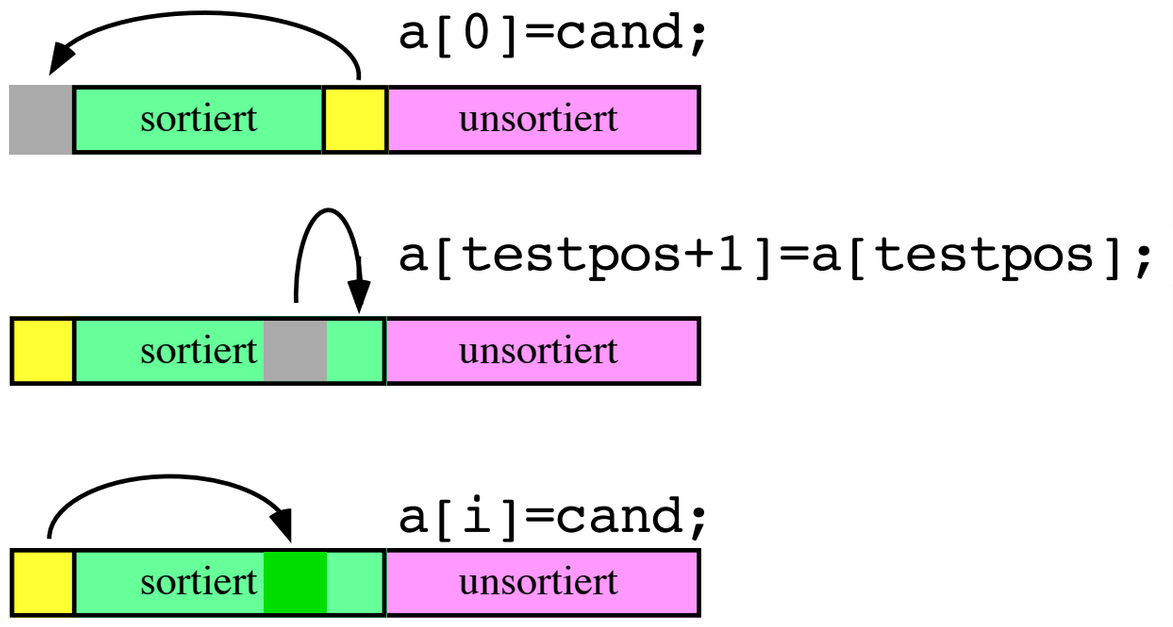
- Traditionelles EDV-Verfahren
  - algorithmische Fingerübungen
  - Abschätzungen für Rechenaufwand
  - Einfluss des Datenvolumens
- Problemstellung
  - ungeordnete Menge von Elementen,
  - evtl. nur sequentiell zugreifbar
  - Ordnung der Schlüssel gesucht





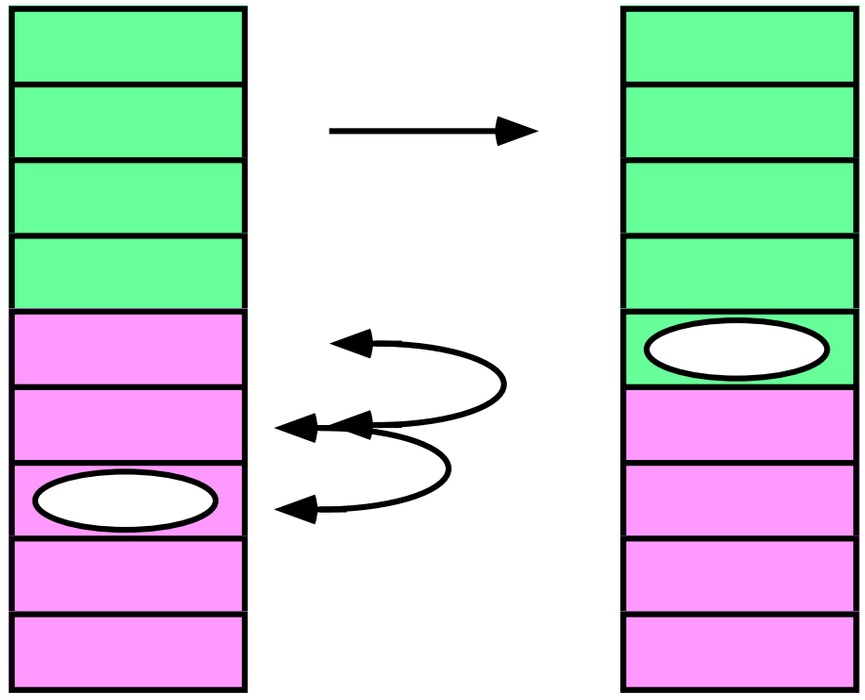
- Sortieren durch Einfügen
- Zu sortieren: item menge[N];
  - Kandidat im linken Teil einsortieren
  - entstandene Lücke mit schon sortierten Elementen auffüllen

```
void DirektesEinfügen(item *a, int N)
{
item cand;
int candpos, testpos;
for (candpos=2, candpos<=N, candpos++)
  {
     cand= a[candpos]; a[0]= cand;
     testpos=candpos-1;
     while (cand < a[testpos])
     {   a[testpos+1]=a[testpos];
        testpos= testpos-1;
     }
     a[testpos+1]=cand;
  }
}
```



- Sortieren durch Vertauschen ("Bubble Sort")
  - Benachbarte Elemente paarweise vertauschen, falls a[i]<a[i+1]
  - Analogie zu einer aufsteigenden Blase
  - bis zu der ihrem 'Gewicht' entsprechenden Höhe

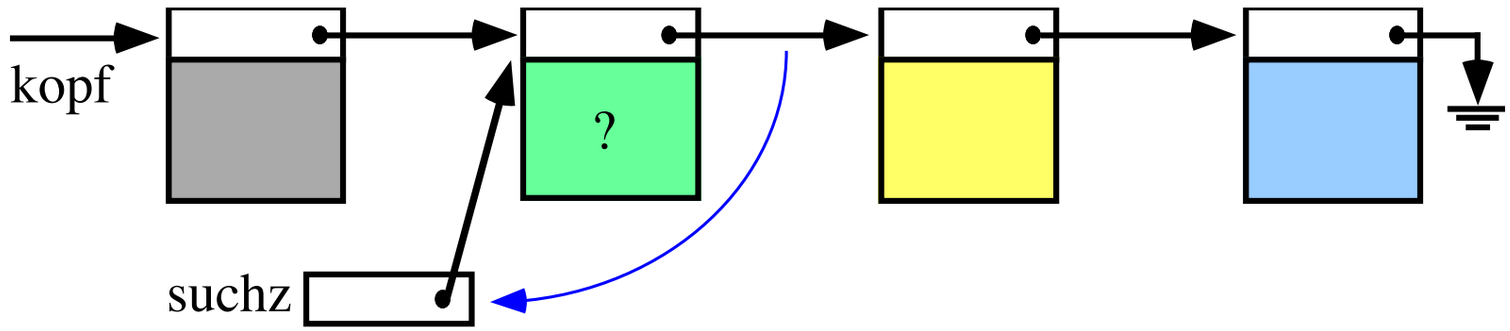
```
void bubblesort(item *a, int N)  {
item zwischen;
int fertig, cand;
for (fertig=2, fertig<N, fertig++)     {
   for (cand=N, cand>=fertig, cand--)  {
      if (a[cand-1]>a[cand])
         { zwischen=a[cand-1];
           a[cand-1]=a[cand];
           a[cand]=zwischen;
 }  }     }
```



- Verhalten der Elemente
  - leichtes Teilchen in einem Durchlauf ganz nach oben
  - schweres Teilchen sinkt mit jedem Durchlauf nur um eine Stelle
  - nur solange sortieren, bis kein Austausch mehr stattfindet
  - "Shakersort" als Variante mit abwechselnder Durchlaufrichtung
  - auch Bubblesort bewegt Elemente nur über eine kleine Distanz

Listen

- Liste: Verkettete Records
  - linear: Liste
  - mehrweg: Baum
- Durchsuchen einer Liste

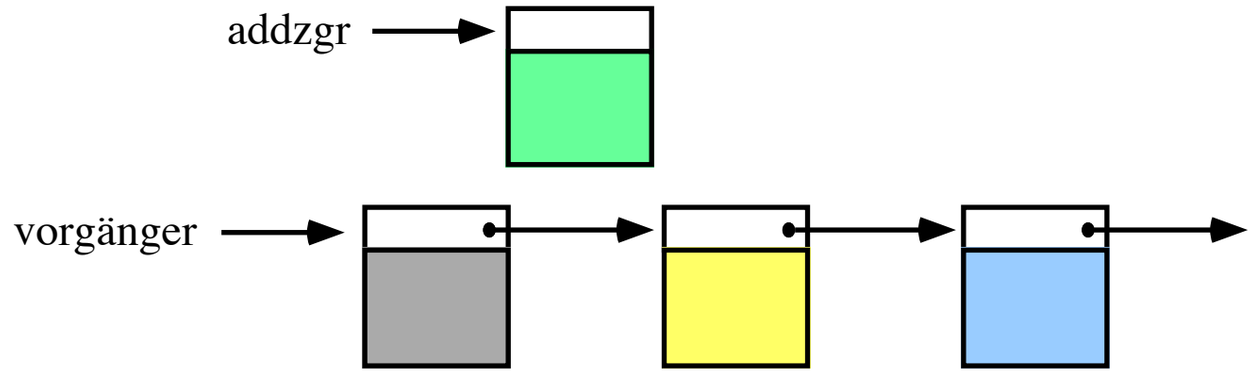



```
gefunden = NULL;
suchz = kopf;
while  (suchz != NULL)
{ if (suchz->data==gesucht)
  {
    gefunden:= suchz;
    break;
  }
  else  suchz = suchz->nxt;
}
```

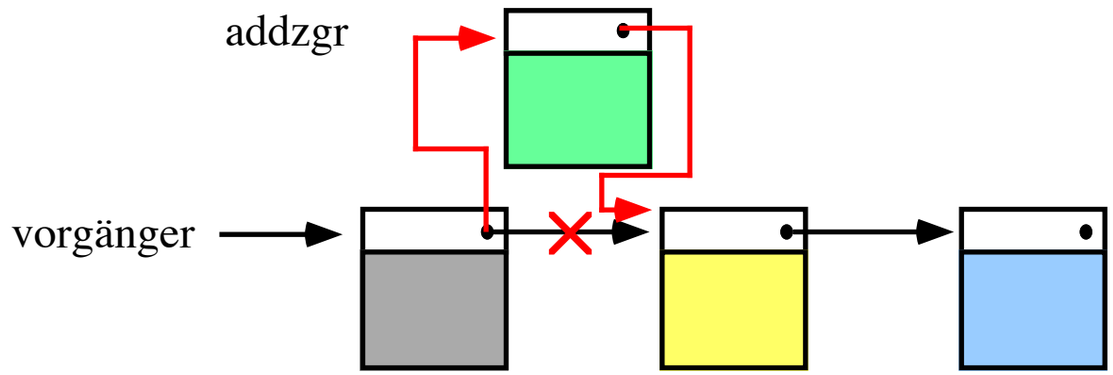
- Aufbau einer neuen Liste
  - Einlesen von content-Records
  - Einfügen am Kopf der Liste

```
listelem *kopf, *addzgr;
content eingabe;
...
kopf = NULL;  /* Liste leer */
while (elem_lesen(&eingabe))
{
  addzgr = elem_anlegen();
  addzgr->data = eingabe;
  addzgr->nxt = kopf;
  kopf = addzgr;
}
...
```

- Unmittelbar nach irgendeinem Vorgänger einfügen



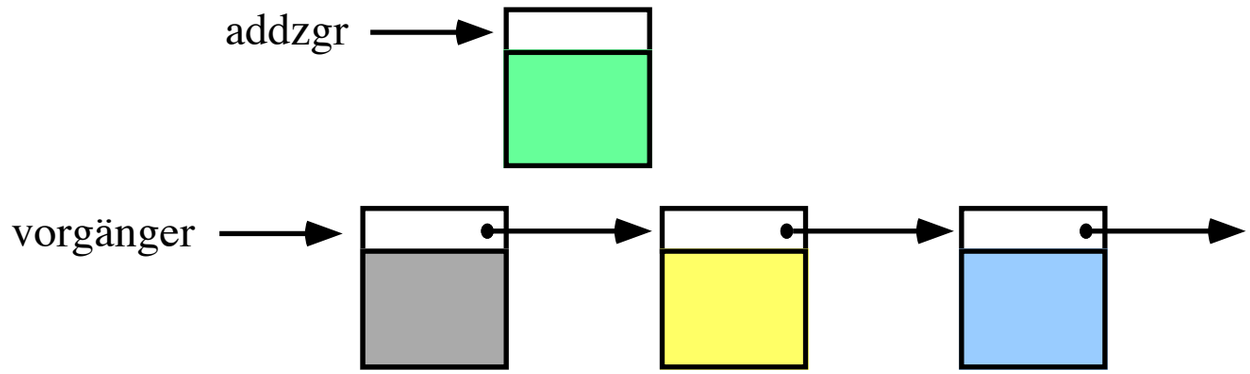

```
addzgr->nxt = vorgänger->nxt;
vorgänger->nxt   = addzgr;
```



- Zeiger auf Vorgänger wird gebraucht

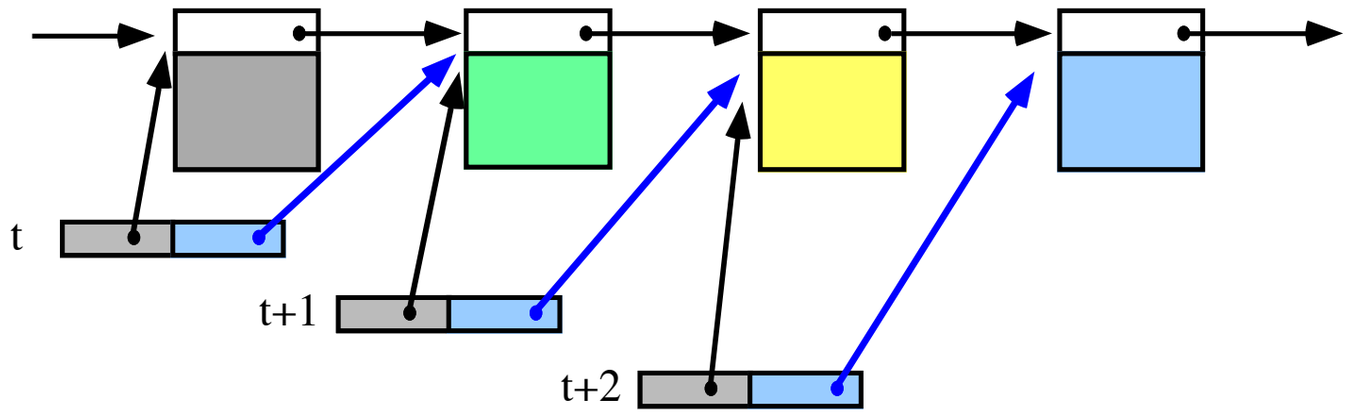
- Einfügen *nach* einem Element ist leicht
- *vor* einem Element schwierig

  => Trick: Nachher einfügen und altes Element nach vorne kopieren.

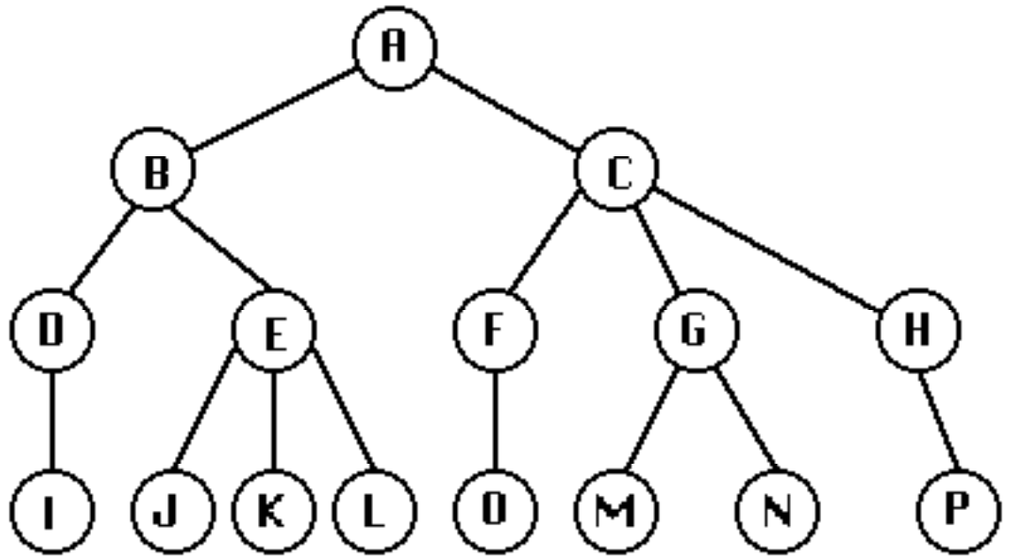


- Andere Lösung

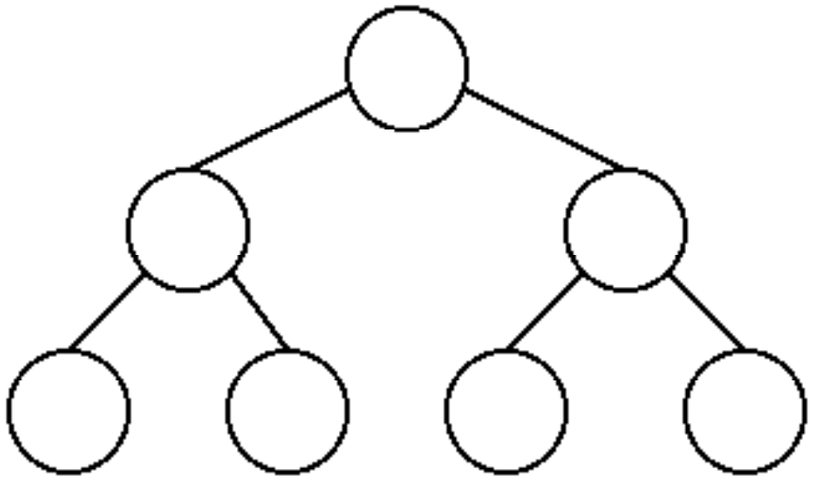
  Schleppzeiger auf Vorgänger immer mitführen



- Listen mit mehr als einem Nachfolger: Bäume



- Binärer Baum



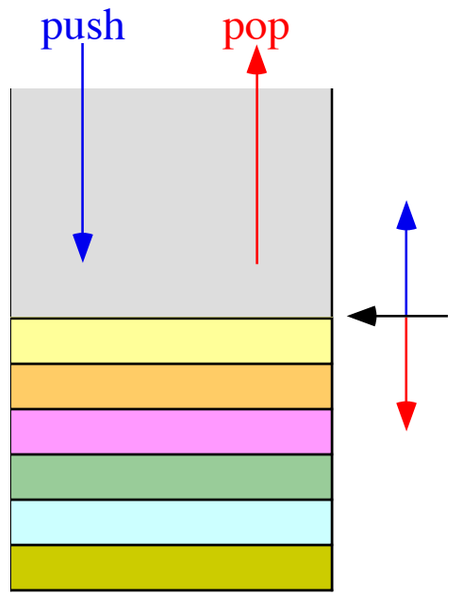
- Durchsuchen von Bäumen
  - linker Folge-Teilbaum
  - Knoten testen
  - rechter Folge-Teilbaum
  - => in-order
  - Aufhören bei Erfolg

```
knoten *teste(knoten *mitte, elt such)
{
  knoten *help;
  if (mitte == NULL) return NULL;
  else
  { help = teste(mitte->links);
    if (help != NULL) return help;
    if (mitte->inhalt == such) return mitte;
    return teste(mitte->rechts);
} }
```

- Andere Suchreihenfolge
  - per-order: Knoten, links, rechts
  - post-order: links, rechts, Knoten

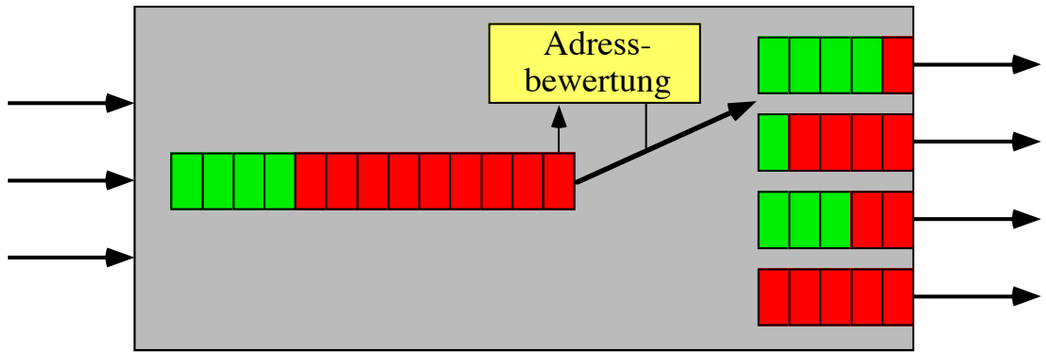
Speicherverwaltung

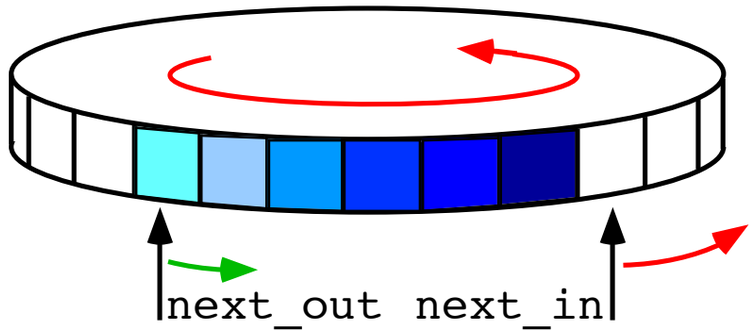
- Platz für dynamisch eintreffende Daten
- Verwaltung
  - freien Platz finden
  - gebrauchten markieren (lock)
  - nicht mehr gebrauchten freigeben
  - Fehlerroutinen
- Bearbeitungssemantik
  - Stack: last in - first out
  - Warteschlange: first in - first out
- Stack (Stapel)
  - Stack_Push legt Element in den Stack
  - Stack_Pop holt Element ab/heraus
  - Stack_Create
  - Implementierung mit Array oder mit Liste



```
#include <stdlib.h>
#include "item.h"

Item *stack;
int level, size;
Item badItem = {.text="stack_underrun"};
enum stack_error {stack_noError, stack_overflow};

void stack_Create(int maxElts)
   {stack=malloc(maxElts*sizeof(Item));
    level=0; size = maxElts;}

int stack_Push(Item theItem)
   {if (level<size)
      {stack[level++]=theItem; return stack_noError;}
    else return stack_overflow;}

Item stack_Pop()
   {if (level>0) return stack[--level];
    else return badItem;}
int main (){stack_Create(42);};
```

- Queue (Warteschlange)
  - FIFO
  - z.B. Netzwerkadapter, Router
  - in Array oder Liste
- Ringpuffer
  - 2 Zeiger: next_in, next_out
  - Schreiben inkrementiert next_in
  - Lesen inkrementiert next_out
  - inkrementieren modulo buffersize
  - am besten 8, 16, 32, 64, …

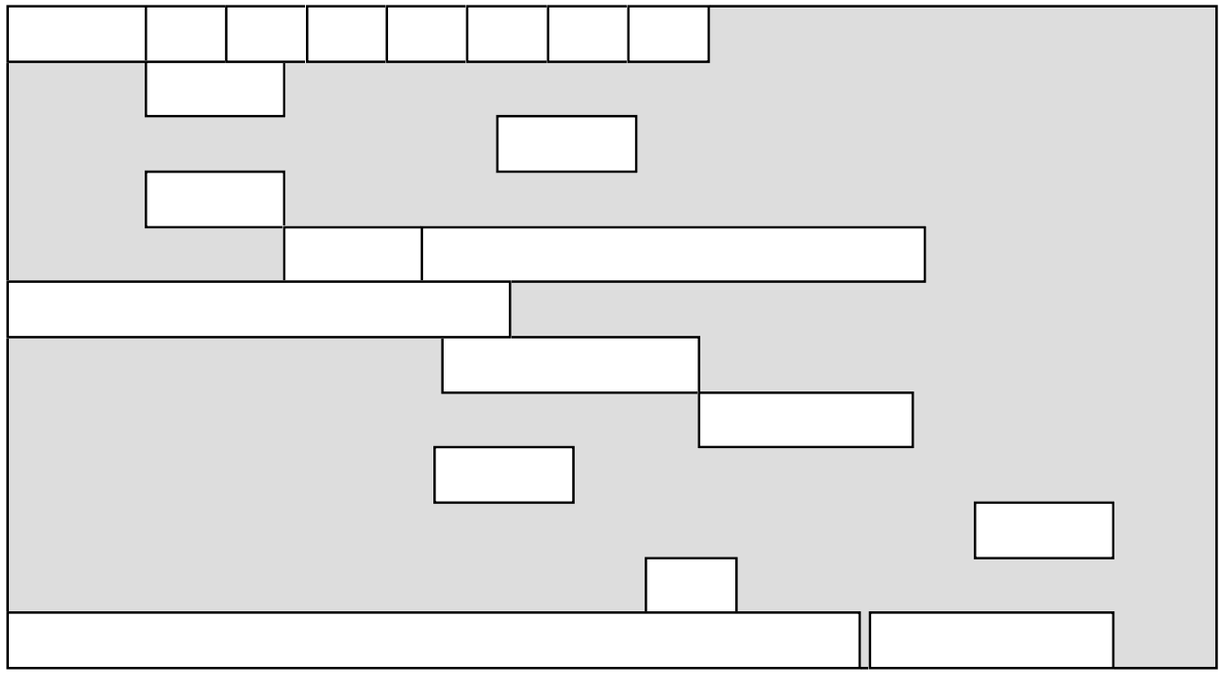





```
#include <stdlib.h>      /* nach Sedgewick */
#include "Item.h"

typedef struct QUEUEnode* link;
struct QUEUEnode { Item item; link next; };
link head, tail;

void QUEUEinit() { head = NULL; }
int QUEUEempty() { return head == NULL; }
```

```
link NEW(Item item, link next)
  { link x = malloc(sizeof *x);
    x->item = item; x->next = next;
    return x;
  }

enQueue(Item item)
  { if (head == NULL)
      { head = (tail = NEW(item, head)); return; }
    tail->next = NEW(item, tail->next);
    tail = tail->next;
  }
Item deQueue()
  { Item item = head->item;
    link t = head->next;
    free(head); head = t;
    return item;
  }
```

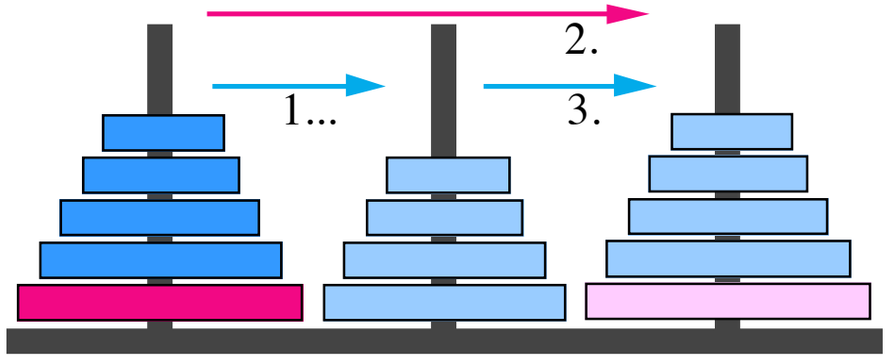
- Heap (Halde)
  - beliebige Speicherbereiche



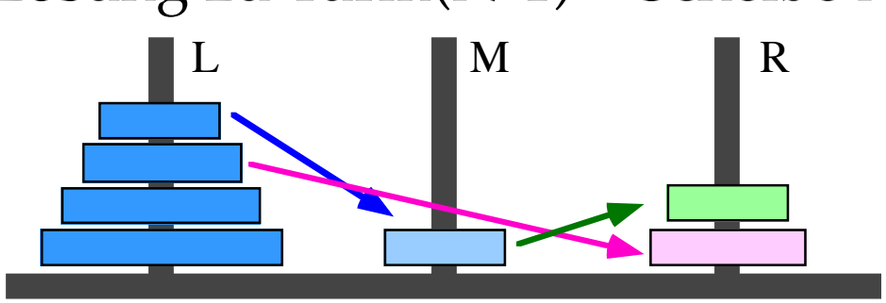
  - anlegen mit `void *malloc(size_t size)` aus stdlib.h
  - freigeben mit `free(void *pointer)`
- Implementierung
  - großer Bereich des Haupspeichers
  - Liste mit belegten Blöcken
  - Freispeicherliste
  - Fragmentierung sorgfältig managen

Rekursion

- Funktion wird in sich wieder aufgerufen
  - direkt oder indirekt
- Problem auf einfacheres zurückführen
  - wiederholen bis zum trivialen Problem
  - in jedem Schritt etwas leichtes machen
- wo haben wir das schon benutzt?
- Beispiel: Türme von Hanoi
- Regeln:
  - Scheibenturm von links nach rechts bringen
  - immer nur eine Scheibe bewegen
  - nur kleinere Scheiben auf größere legen



- Lösungsansatz:
  - Bewegung des Turmes mit Höhe N auf Turm mit Höhe N-1 zurückführen,
  - Turm(N-1) auf mittleren Stock bringen
  - verbleibende Scheibe nach rechts
  - Turm(N-1) nach rechts
- Rekursive Strategie:
  - triviale Lösung für Turm(0)
  - Lösung zu Turm(1) = Lösung zu Turm(0) + eine Scheibe bewegen
  - Lösung zu Turm(N) = Lösung zu Turm(N-1) + Scheibe N bewegen



- Züge für eine Turmhöhe von 4

```
L->M,L->R,M->R,L->M,R->L,R->M,L->M,L->R,
M->R,M->L,R->L,M->R,L->M,L->R,M->R,
```

```c
#include <stdio.h>

typedef char Stock;  /* 3 Stöcke: 'L','M','R' */

void BewegeScheibe(Stock von,Stock nach)
{ Printf("%c",von); Printf("->%c",nach);
  Printf("%c",',');
}

void  BewegeTurm(int restHoehe, Stock von,
                 Stock zwisch,Stock nach)
{ if (restHoehe<=0) return;
  BewegeTurm(restHoehe-1,von,nach,zwisch);
  BewegeScheibe(von, nach);
  BewegeTurm(restHoehe-1,zwisch,von,nach);
  printf("\n");
}

main()
{  BewegeTurm(4,'L','M','R'); }
```

Objekte, Patterns und Frameworks

Grundzüge des objektorientierten Programmierens

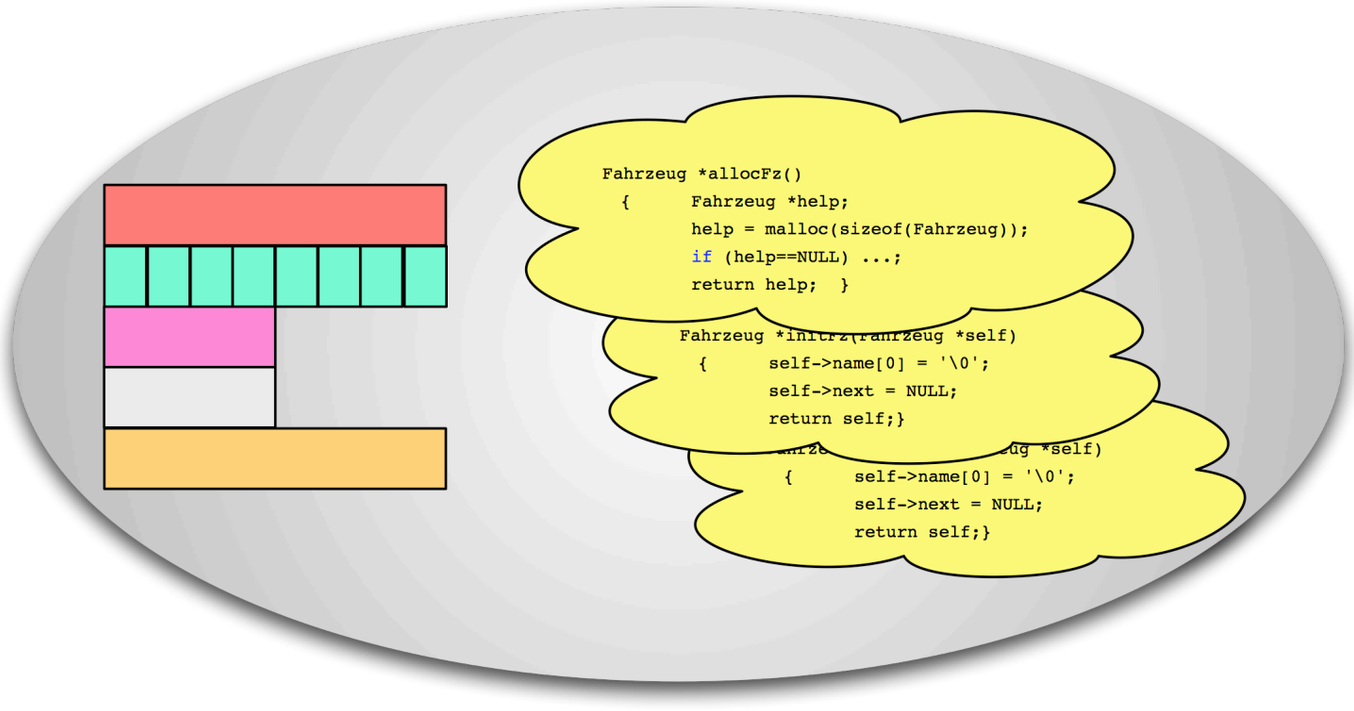
- Kay, Goldberg, Ingalls: Smalltalk
- Datenstrukturen
  - Inhalt als Attribute
  - Funktionen zur Inhaltsmanipulation
  - Semantik steckt in den Funktionen

```
typedef struct fzstruct
{  char name[32];
   enum {Lok,Perswg,Gueterwg} art;
   int Baujahr;
   struct Fahrzeug *next;
} Fahrzeug;
```

```
Fahrzeug *lok;
…
lok = initFz(allocFz());
lok->next = initFz(allocFz());
```

```
Fahrzeug *allocFz()
{  Fahrzeug *help;
   help = malloc(sizeof(Fahrzeug));
   if (help==NULL) ...;
   return help;
}

Fahrzeug *initFz(Fahrzeug *self)
{  self->name[0] = '\0';
   self->next = NULL;
   return self;
}
```

- Objekte: Struktur+Funktionen
  - als eigenständige Einheit des Programmierens
  - Klassiker: Initialisierung einer Struktur, Vergleich, …
  - datenstruktur-orientierte Modularisierung



- Beispiel Objective-C
  - Brad Cox, Tom Love, 1980 …
  - inspiriert von Smalltalk
  - NeXTStep, OpenStep, GNUStep, Cocoa, iPhone, iOS

- Klasse = Strukturdeklaration + Funktionen
  - Funktionen zum:
    - Initialisieren, Löschen, …
    - evtl. Felder zugreifen
    - Listenoperationen
  - heißen Methoden, Selektoren, Messages

  => Konzept

```
#import <Cocoa/Cocoa.h>
@interface fahrzeug : NSObject {
   char name[32];
   enum {Lok, Perswg, Gueterwg} Art;
   int Baujahr, Gewicht;
   fahrzeug *next; }
- (fahrzeug *)init;
- (void) add: (fahrzeug *)fz;
- (void) setweight:(int)wght;
- (int) getweight;
- (int) cmpwght;
- (fahrzeug *)getnext;
@end
```

```
#import "fahrzeug.h"
@implementation fahrzeug
- (fahrzeug *)init
{ [super init];
  name[0] = '\0';
  ...
}

- (void)add:(fahrzeug *)fz
  {next = fz; }

- (void) setweight:(int)wght
{ Gewicht = wght;}

- (int) getweight
{ return Gewicht}

- (fahrzeug *)getnext
{ return next; }

- (int) cmpwght
{ if (self == nil) return 0;
  else return Gewicht +
       [next cmpwght];       }
@end
```

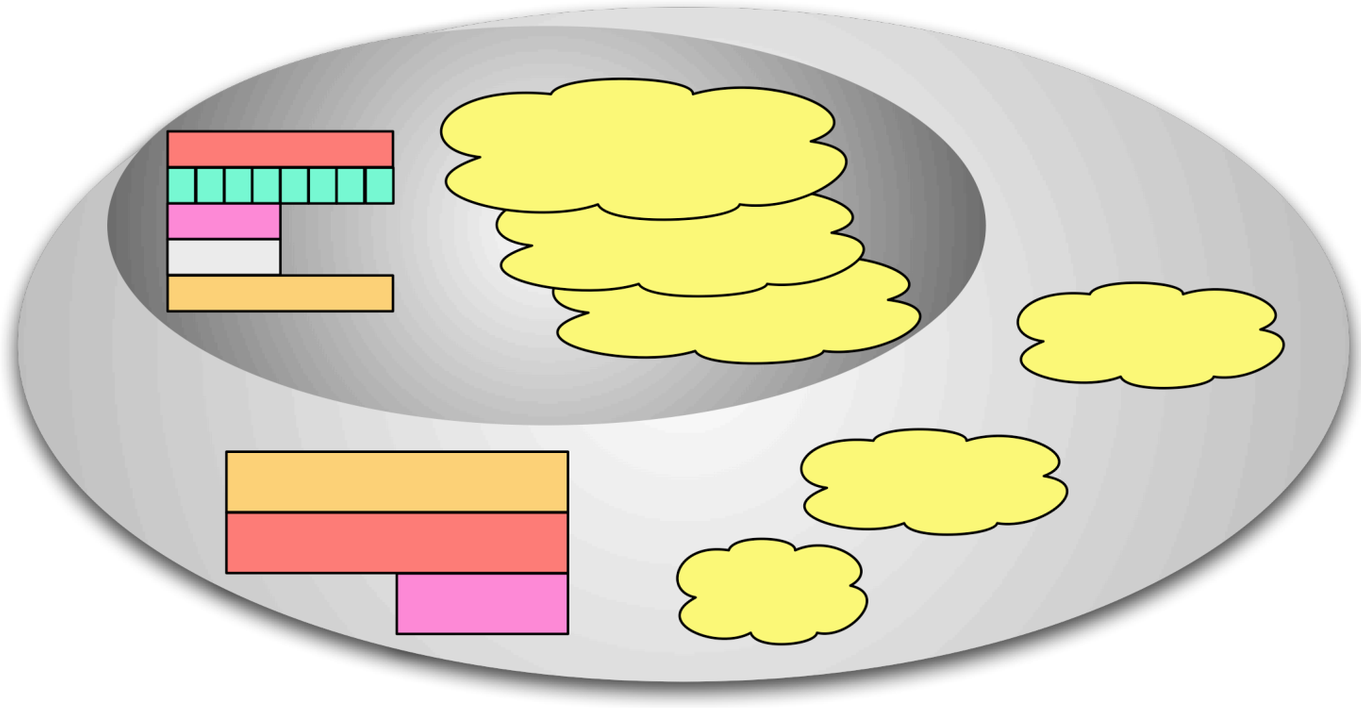
- Instanzen / Instantiierung
  - dynamisches Anlegen: alloc
  - meist Initialisieren: `init`
  - Referenzen in Zeigern halten
- Nachrichten senden
  - entspricht Funktionsaufruf
    `[<obj> <message>]`
  - mit Parametern:
    `[<obj> <message>:<parm>]`
    `[<obj> <message>:<parm> p2:<parm> p3: ...]`
  - Namenskonvention für Parameter 2 ... n: `andp2 …`
  - Abkürzung: [<obj> <message>:<parm>:<parm>:
- Accessor-Methoden
  - in den Methoden der Klasse mit Feldname
  - klassisch: `<instance>.<property>`
  - Accessor `[<object] <feldzugriff>]`
  - Namenskonvention für Methode `get<feld> bzw. set<feld>`

```
fahrzeug *zug, *z1;

zug = [[fahrzeug alloc]init];
z1  = [[fahrzeug alloc]init];
[zug add:z1];
```

```
- (fahrzeug *)init
{ [super init];
   name[0] = '\0';
   next = nil;
   Gewicht = 0;
   return self;     }
```

```
- (void)setweight:(int)wght;
- (int) getweight;
```

- Vererbung
  - Klassen erweitern
  - Basisklasse + neue Felder und Methoden



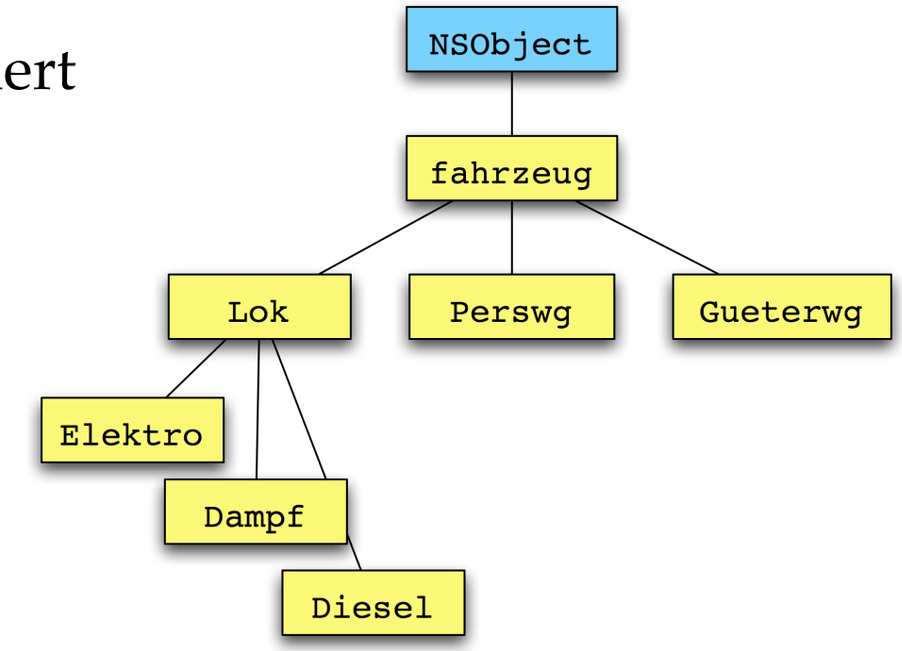
- Felder und Methoden der Basisklasse weiter verwendbar
- Begriff: Superklasse
- self und super

- Methoden
  - erben
  - neue hinzufügen
  - manche überschreiben
  - Scope: Blatt zur Wurzel

```objc
#import <Cocoa/Cocoa.h>
#import "fahrzeug.h";

@interface lok : fahrzeug
{
   enum {Elektro, Diesel, Dampf} prinzip;
   int leistung;
   int last;
}
- (lok *) init: (int) lst andlast: (int) load;
- (int) add: (fahrzeug *) fz;

@end
```

```objc
#import "lok.h"

@implementation lok
- (lok *)init:(int)lst andlast:(int)load
{
   [super init];
   prinzip = Elektro;
   leistung = lst;
   last = load;
   return self;              }

- (int) add:(fahrzeug *) fz
{
  if (last<[fz cmpwght]) return 0;
  else
  { [super add:fz];
    return 1;
  }}
@end
```

- Klassenbaum
  - Alle Klassen erben von einer Wurzelklasse, hier: NSObject
  - möglichst viele Methoden erben
  - abstrakte Klassen werden nicht *selbst* instantiiert
- Klassenmethoden
  - zum allgemeinen Klassenmanagement
  - z.B. Konstruktor (`alloc`)
  - `+ (<type>) <name>: ...`
- Polymorphismus?
  - von mehreren Klassen erben
  - nicht wirklich notwendig
  - nicht in allen OO-Programmiersprachen
  - in manchen Hilfskonstrukte



Frameworks am Beispiel

- Funktionen und Strukturen in Libraries
  - häufig verwendet
  - Listen, Dateizugriff, Ein/Ausgabe, Fenster, …
  - auch API genannt
  - kann zur Abstraktion verwendet werden
  - Portabilität
- Framework
  - Menge von Klassen
  - Wiederverwendbarkeit
  - Erweiterbarkeit: überschreiben, spezialisieren, …
  - gibt Programmstruktur vor
  - "Klasse Programm"
- Kontrollumkehr
  - Programm wird zur Funktionssammlung
  - Funktionen im Programm werden von außen gerufen
  - Programmierer *füllt* vorgegebene Funktionen

- Frameworks in iOS und MacOS
  - Core Image, OpenGL, Quartz, QuickTime, …
  - AppKit: Windows, Buttons, Menus, …
- Beispiel Foundation.h
  - aus NeXTStep: Namen NS*
- Numbers, Strings, Collections
  - NSData
  - NSString: suchen, vergleichen, konkatenieren
  - NSArray, NSDictionary, NSSet
  - NSCharacterSet, NSScaner (Zahlen aus Strings extrahieren)
- Operating System Services
  - NSFileManager, MSPathUtilities
  - NSThread, NSProcessInfo
- Memory Management für Objekte
  - NSAutoReleasePool
- URLs

- Beispiel NSString
  - auch für Unicode Char, UTF8 und UTF16
  - unveränderlich: NSString
  - mutable subclass: NSMutableString

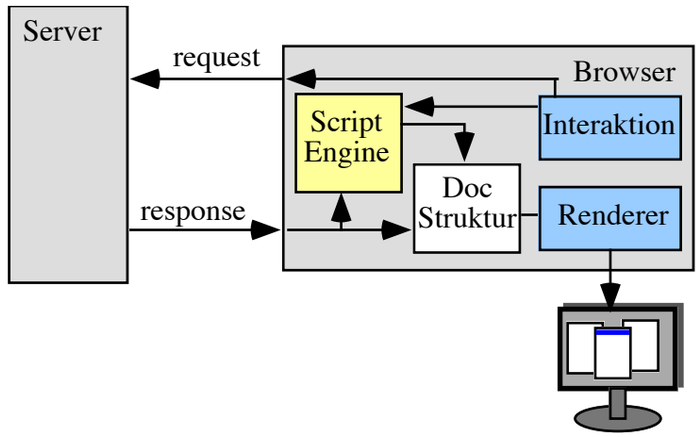
```
NSString *hString = @"Hello";
NSString *hwString = [hString stringByAppendingString:@", world!"];
```

- Methoden
  - Erzeugen (auch aus C-Strings, URLs oder Dateien)
  - Länge
  - Teilstrings suchen und herausholen
  - Teilen und Konkatenieren
  - Vergleichen
  - Pfade für Dateisystem

```
NSString *s10 = @"blue10";
NSString *s2 = @"blue2";
NSComparisonResult res;

res = [s10 compare:s2];
// res = -1 (NSOrderedAscending)

res = [s10 compare:s2 options:NSNumericSearch];
// res = 1 (NSOrderedDescending)
```

Patterns

- Gang of four: "Design Patterns: Elements of Reusable …" [1994]
- Lösungen für typische Problemstrukturen
  - generalisiert, wiederverwendbar
  - 'best practice'
  - "über" Datenstrukturen und Algorithmen
  - auch zur Prozesskommunikation
- Ein berühmtes architectural pattern: MVC
- Model
  - Verhalten und Daten
  - application domain
- View
  - Darstellung des Modells zur Interaktion
  - Display, Rendern
- Controller: reagiert auf Events
  - ändert Modell
  - auch nachrichtenbasiert



- Creational Patterns
  - Objekte erzeugen
  - Factory, Builder, Object Pool, Prototype
- Singleton
  - Menge mit genau einem Element
  - hier: nur eine Instanz einer Klasse
  - einziger Zugangspunkt
  - z.B. beim parallelen Programmieren
  - kapselt gemeinsam genutzte Resource

```
static Singleton *sharedSingleton = nil;

+ (Singleton*)sharedManager
{
    if (sharedSingleton == nil) {
        sharedSingleton = [[super alloc] init];
    }
    return sharedSingleton;
}
// Methoden zur Arbeit mit der Klasse folgen …
```

- Structural Patterns
  - Beziehungen zwischen Elementen
  - Bridge, Adapter
  - Composite und Aggregate
  - Proxy, Facade, Extensibility (Framework pattern)
- Behavioural Patterns
  - Chain of responsibility: Events erhalten und weitergeben
  - Observer (publish/subscribe): für Events registrieren
  - Iterator greift auf Felder nacheinander zu
- Concurrency Patterns
  - konzeptuell parallele Ausführung (threads)
  - Active Object, Monitor, Leader Follower, …
  - Reactor
  - Scheduler
  - Thread Pool

KAPITEL 5

Betriebssysteme

- Abstrahieren und Koordinieren
- Sammlung häufig gebrauchter Softwarekomponenten
  - Hardware-Abstraktion
  - essentielle Software
- Zuteilung der Ressourcen
  - CPU, Speicher, Bildschirm, …
- Oft mit Kommandointerpreter verwechselt
  - Shell, Command.COM, …
  - Explorer, Finder, …
- Dateisystem
  - ebenfalls nur Teil des Betriebssystemes
  - Verwaltung von Plattenplatz (Sektoren)
  - Zuordnung und Wiederverwendung
  - Abstraktion Sektor - Bytestrom

```
printf("%c, char);
```

```
MOVE char,$A7823A
```

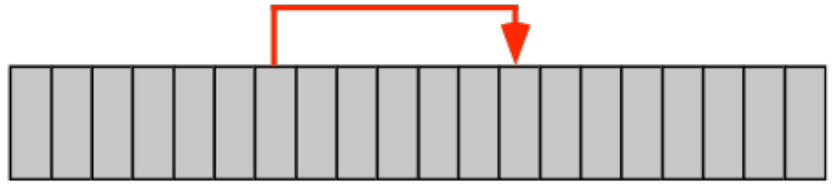
```
MOV   AX,[BP]
OUT   $3F8,AX
```

Verteilung der Ressourcen

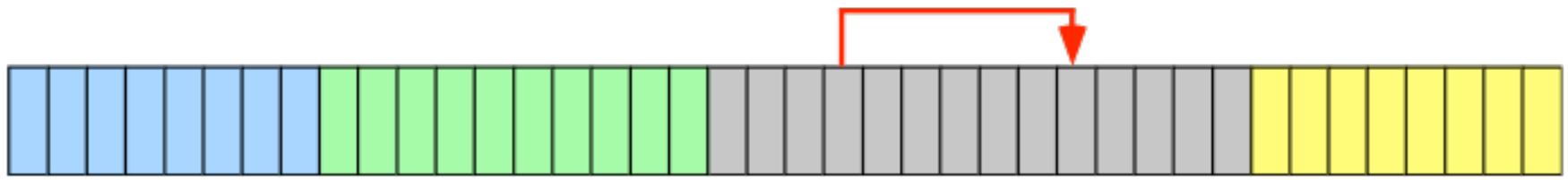
- Prozessor
  - Verteilung der Arbeit auf Prozessoren
  - Unterbrechungen
  - wartende Prozesse
- Speicher (RAM, Festplatte)
- Ein/Ausgabe
  - Bildschirm, Drucker
  - Audio-Ausgabe
- Verteilung des Mangels
  - Bildschirmfläche eindlich => Fensterkonzept
  - Tastatur, Maus
  - Audio-Ausgang => Mixer
  - Prozessorzeit => Zeitscheiben, Zeitüberwachung

Prozesse und Scheduling

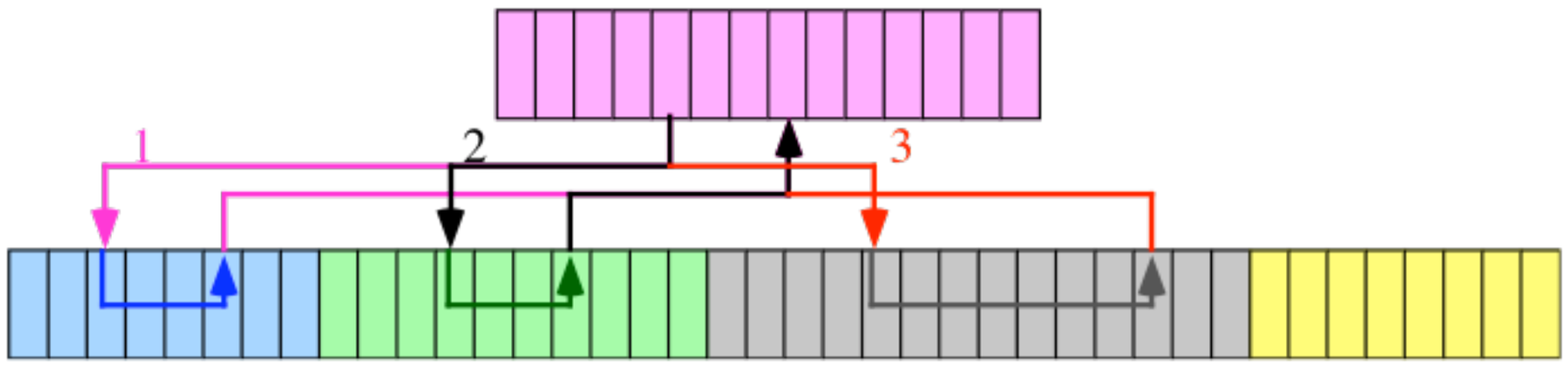
- Instruktionen werden sequentiell ausgeführt
  - in der Regel nicht gleichzeitig
  - ein Befehl nach dem anderen
  - Sprünge verändern nur die Reihenfolge



- Programm = ausführbares Objekt = Prozeß
  - mehrere Prozesse liegen im Speicher

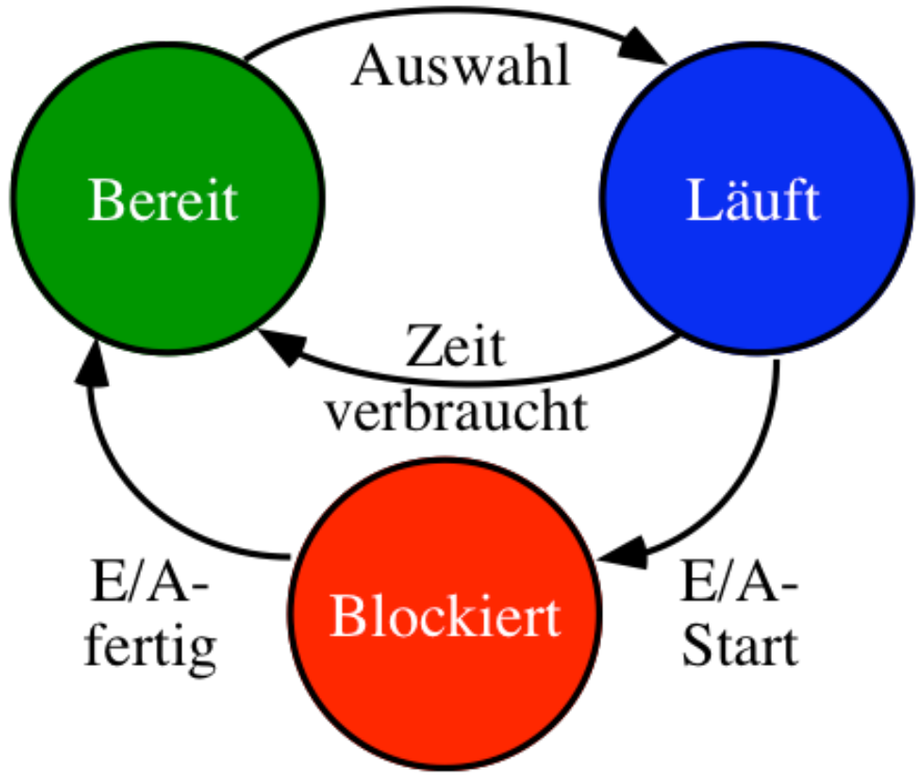


- Benutzerprozesse und ein besonderer Steuerprozeß



- Besonderer Steuerprozeß
  - gibt Ausführungsrecht befristet ab an Benutzerprozesse
  - verteilt Ausführungsrecht 'gerecht'
  - hat besondere Rechte
  - kennt alle anderen Prozesse
- Rückwechsel vom Benutzerprozeß zum Scheduler
  - Sprung in den Scheduler
  - freiwillig
  - erzwungen nach Fristablauf (=> Unterbrechung)
- Prozeßeigenschaften
  - Wichtigkeit (Priorität)
  - benutzte Ressourcen (Festplatte, Drucker, …)
  - verbrauchte Zeit
  - zugeordneter Speicher
- Auslagern (Swapping)
  - falls Hauptspeicher zu klein
  - exaktes Prozeßabbild auf Festplatte schreiben

- Prozesse
  - selbständige Codeeinheiten
  - Object-Code
  - Speicher
  - Attribute
- Multi-Tasking
  - mehrere Prozesse laufen 'verschachtelt'
  - "Zeitmultiplex"
  - für den Benutzer gleichzeitig
  - verschiedene Strategien des Wechsels



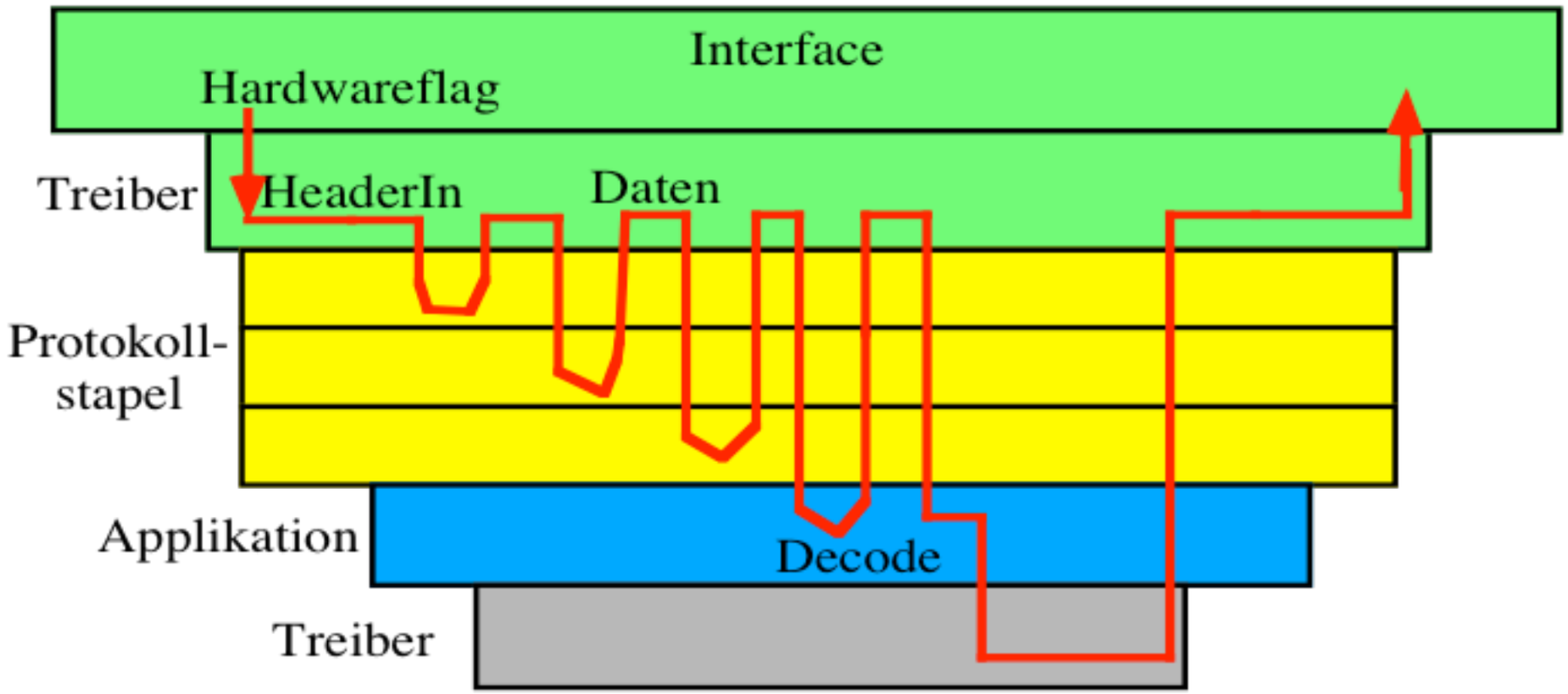
- Prozesswechsel (Umschalten zwischen Programmen)
  - Anhalten eines Prozesses
  - Speichern der Status-Information (PC, Register etc.)
  - Laden der Status-Information des neuen Prozesses
  - Wiederaufnahme der Ausführung bei der nächsten Instruktion
  - z.B. nach Zeit t, wenn gewartet werden muß, …
- Einplanung ≠ Scheduling

- Non-Preemptive Scheduling
  - Prozesse nicht unterbrechbar

  - Weniger Prozesswechsel
  - Kritische Prozesse können nicht unterbrochen werden
- Preemptive Scheduling
  - Prozesse durch andere Prozesse mit höherer Priorität unterbrechbar

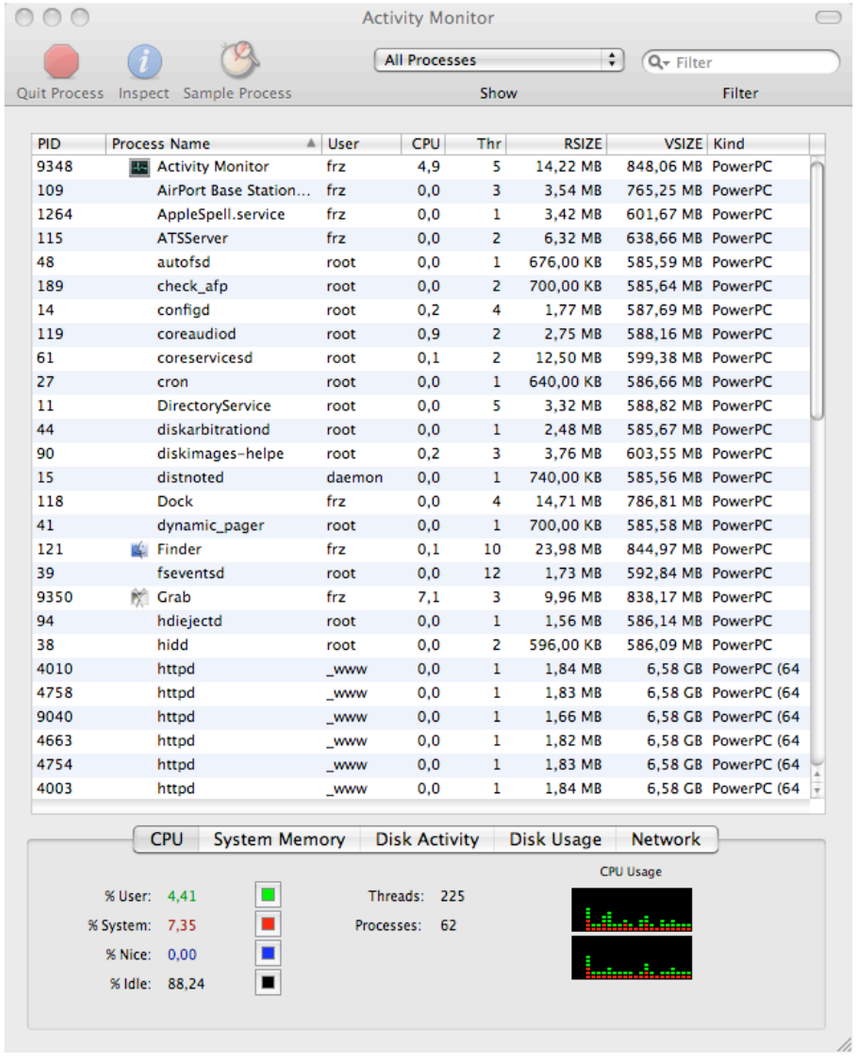
  - Oft in Betriebssystemen vorhanden für CPU
  - Prozesswechsel häufig und teuer
- Prioritätsfestlegung
  - Wichtigkeit
  - nahe 'Abgabe'-Termine
  - lange gelaufen => Priorität sinkt

- Unterbrechungsmanagement (Interrupts)
  - Interrupt-Service-Routine im Netzwerktreiber
  - Completion-Routine auf höheren Schichten (Call Back Routine)



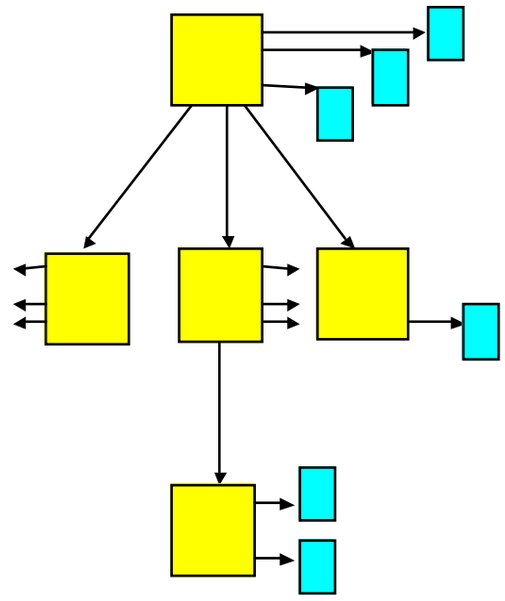
- Interrupt-Latenz durch nicht unterbrechbare Prozesse
- Interruptsperre, z.B. UNIX-Kern

- Steuerung der Prozesse
  - besonderes Steuerprogramm ('Shell')
  - kennt Scheduler
  - Starten eines Programmes => neuer Prozeß
  - Zwangs-Beenden entfernt Prozeß
  - besonders bekannt bei UNIX bzw. DOS
- Grafische Benutzeroberflächen für Steuerprogramm
  - integriert in File-System-Browser
  - Menubefehl 'Open'
  - Doppelklick als Abkürzung
  - Cntl-Alt-Del (Strg-Alt-Entf) zeigt Prozeßliste
  - Windows: Alt-Tab wechselt zwischen Programmen
- Eingebaute Kommandos
  - Prozessliste anzeigen
  - Geräteverwaltung
  - auch Dateiverwaltung auf der Festplatte

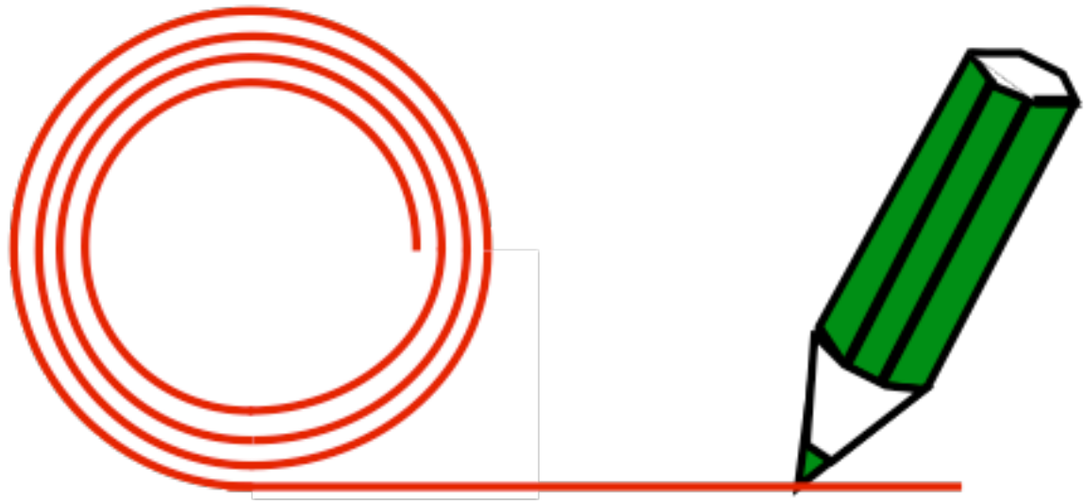


Verwaltung des Plattenplatzes

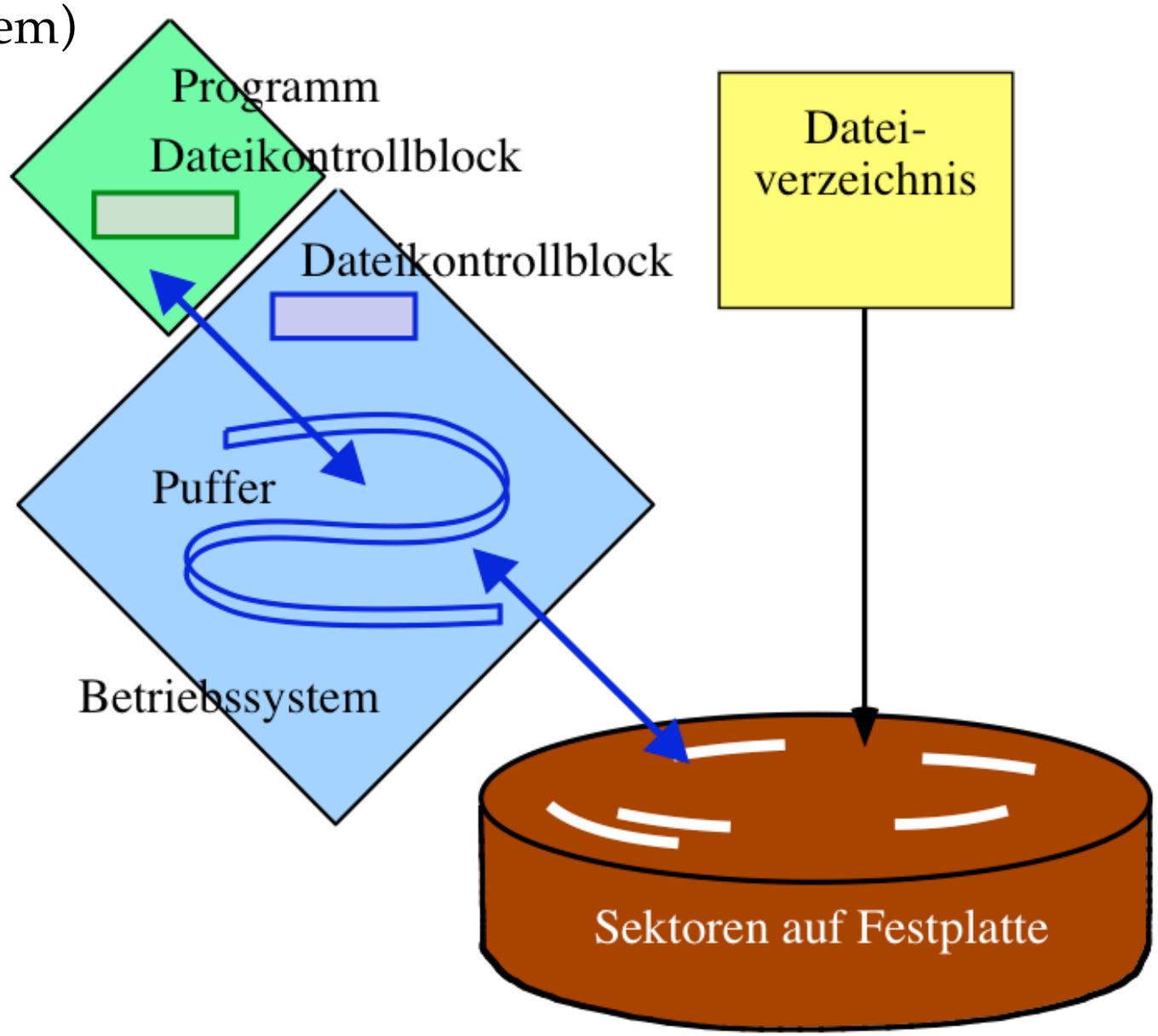
- Zentrale Abstraktion: Datei
  - Menge von Bytes auf der Platte
    - wird vom Betriebssystem zusammengehalten
    - für Programme und Daten
    - Daten: Briefe, Tabellen, Bilder, Datenbank, ...
- Verzeichnisse strukturieren Dateimenge
  - Verzeichnis enthält Dateien
    - Verzeichnis kann andere Verzeichnisse enthalten
    - Wurzelverzeichnis
- Pfad
  - Wurzel/Verzeichnis/Verzeichnis/Verzeichnis/Datei
    - c:\benutzer\froitzheim\daten\vorlesung.doc



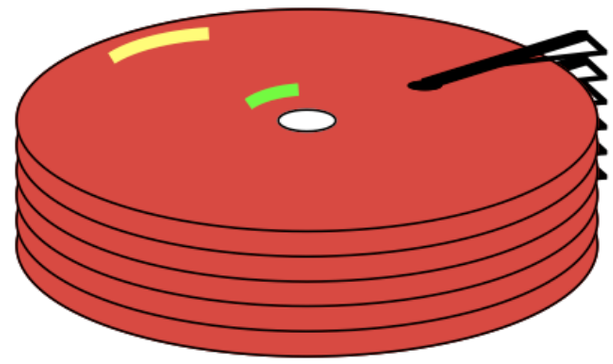
- Historisch gesehen ein Magnetband
  - mit einem Schreib-/Lesekopf
  - Datenblöcke sequentiell aneinanderreihen
  - wahlfreie Zugriffe teuer bzw. langsam
  - Einfügeoperationen sehr teuer
  - beinahe beliebiges Größenwachstum
- Aus der Sicht der Programmiersprache
  - Von Dateimodulen exportierter Typ "File"
  - sequentiell Lesen und Schreiben
  - Öffnen und Schliessen, Zugriffsrechte
- Aus der Sicht des Betriebssystems
  - benanntes Objekt ("WS1996PI.TXT")
  - Dateikontrollblock mit Puffer im RAM
  - Assoziation einer Dateivariablen mit externer Datenregion
  - Abbildung:
    blockweise adressierte Objekte auf Platte
    => byteweise adressierte Objekte im Hauptspeicher



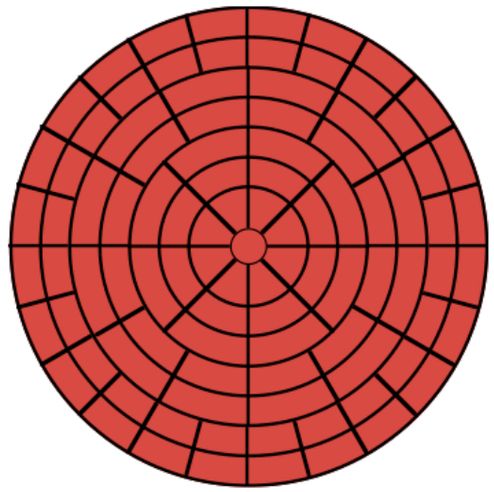
- Implementierung
  - Programm mit Zugriffsroutinen
  - Treiberroutinen im Betriebssystem
  - Dateivariable im Programm (file)
  - Dateikontrollblock (User & System)
  - Pufferbereich im Hauptspeicher
  - Dateiverzeichnis für Festplatte
  - Sektorzuordnung auf Festplatte



- Plattenformat
  - n Platten auf einer Spindel (z.B. 5)
  - 2n Oberflächen -> 2n Köpfe
  - viele Spuren pro Platte (z.B. 40.000)
  - Zylinder: Menge von Spuren im vertikalen Schnitt
  - viele Sektoren pro Spur (z.B. 1000 - 6000)

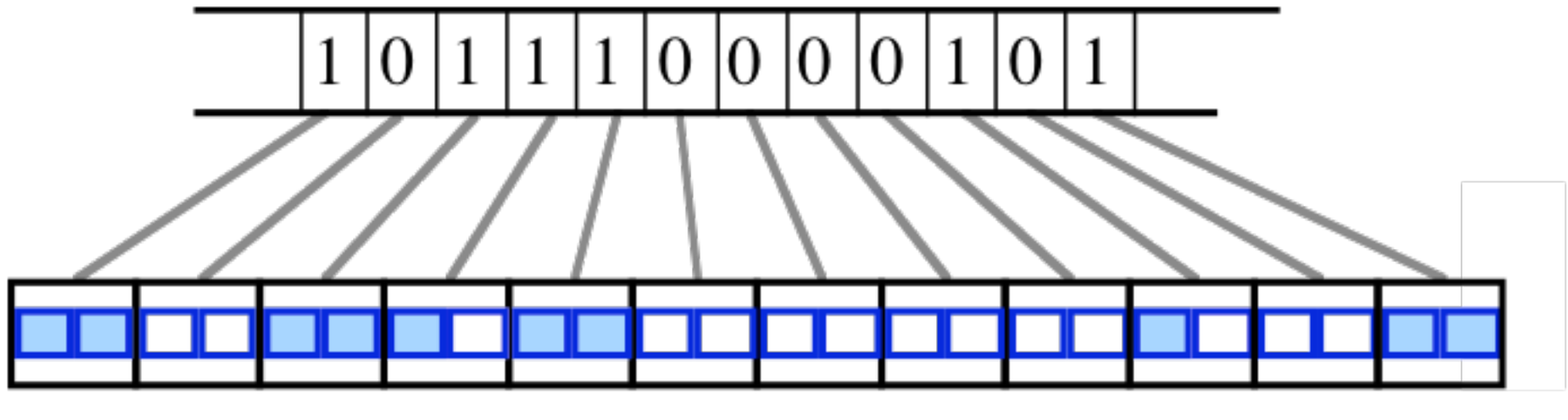


- Mehrere Festplatten-Partitionen
  - ***m*** Gbytes pro Platte
  - ***d*** Dateinamen pro Platte
  - ***s*** Dateistücke pro Platte
  - bessere Nutzung für kleine Dateien ...



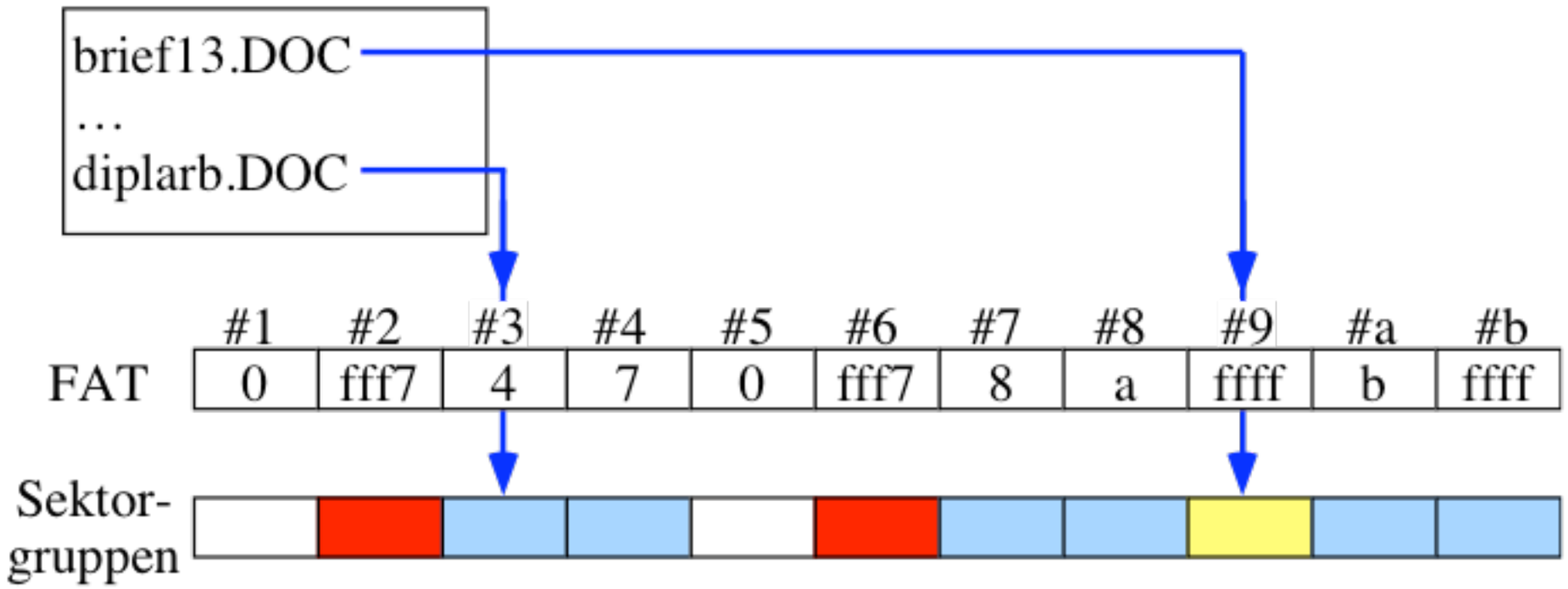
- SSD (Solid State Drive) auch blockorientiert
- Verschiedene Datei- & Betriebssysteme auf verschiedenen Partitionen
  - MacOS, MS-DOS
  - Unix, Linux
  - Windows (NTFS)
- Umschalten zwischen Betriebssystemen
  - Parameter RAM
  - Bootmanager

- Plattenspeichervergabe: Wo ist noch Plattenplatz frei?
  - Absuchen der FAT
  - kompakter mit Allozierungs-Bittabelle
  - Bitplätze entsprechen Platz auf der Platte



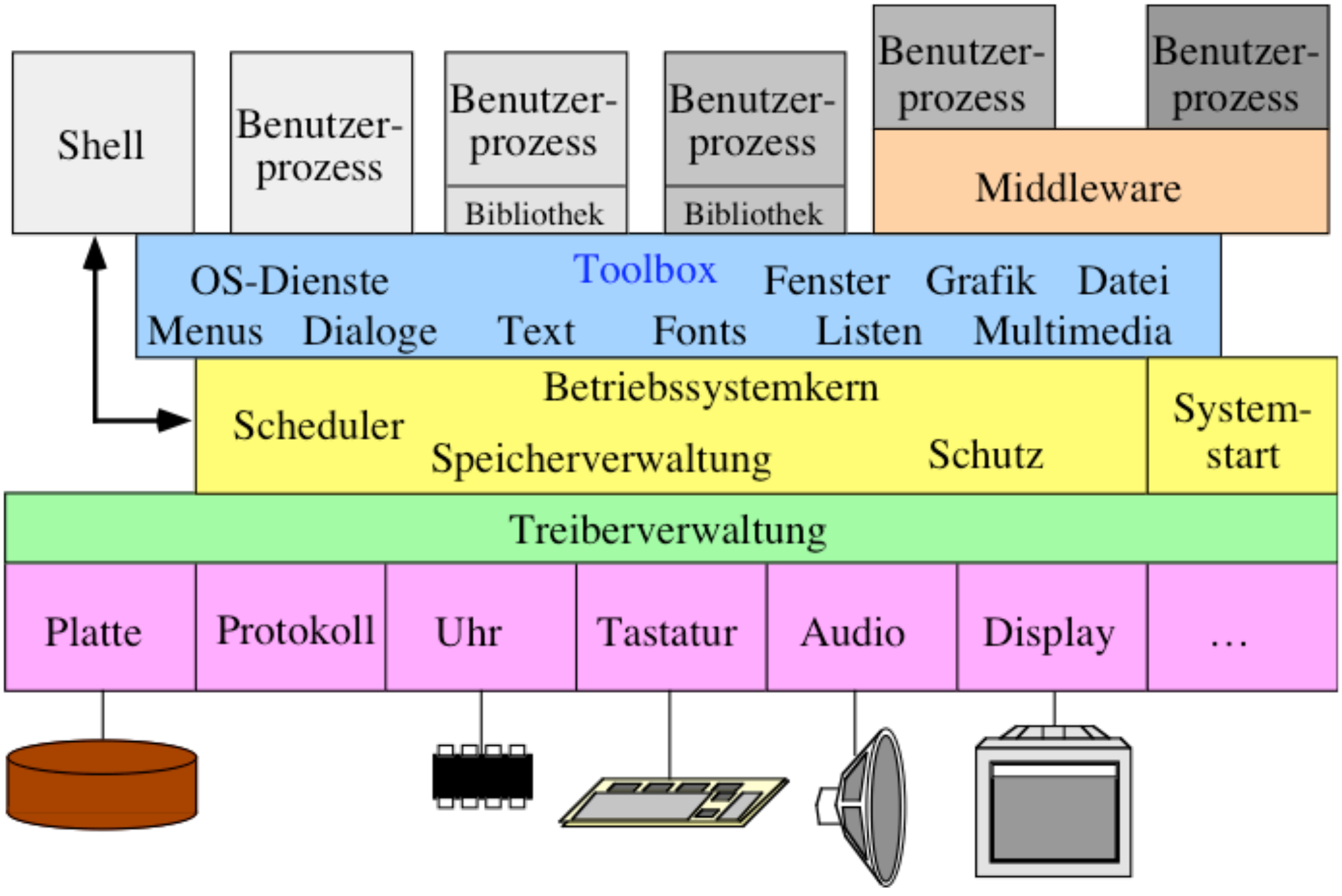
- Enthält ein Bit für jede Belegungseinheit (Allocation Block)
  - teilweise im Hauptspeicher
  - gesetzt, falls der Block belegt ist
  - z.B. 65536 Bit in der Tabelle
  - ab 64 Mbyte mehr als 2 logische Blöcke pro Bit
- Große Platten
  - viele Blöcke pro Eintrag (Cluster)
  - größere Bitmap
- Keine Aussage über Dateizugehörigkeit

- File Allocation Table (FAT)
  - Dateizuordnungstabelle auf den ersten Spuren einer Platte
  - Belegungseinheiten (Blöcke) der Datei über FAT verkettet
  - der letzte Block ist mit $FFFF markiert
  - schadhafte Blöcke sind mit $FFF7 markiert



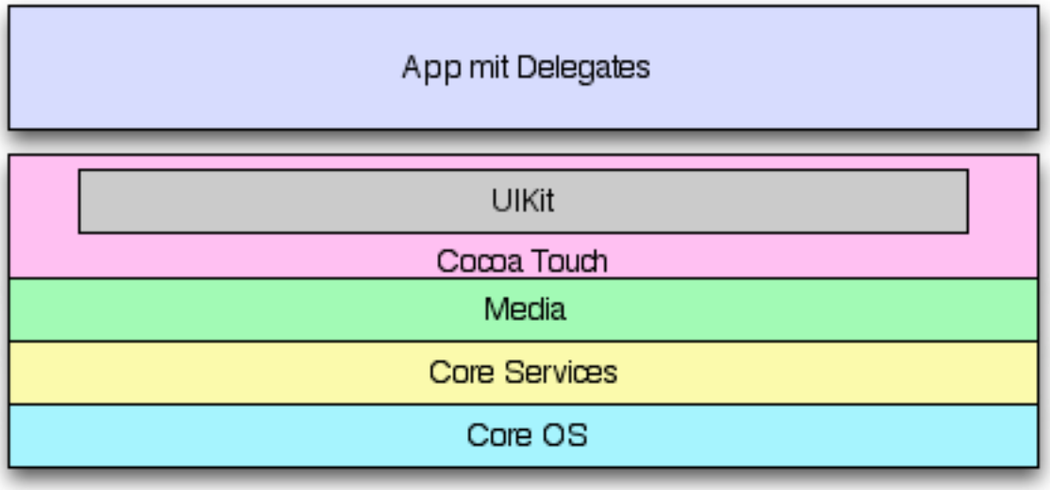
- Verzeichnis zeigt auf den ersten Block einer Datei
- Andere Dateisysteme
  - Windows NTFS komplexer
  - UNIX-NFS netzwerkfähig
- Virtual File System als Abstraktion

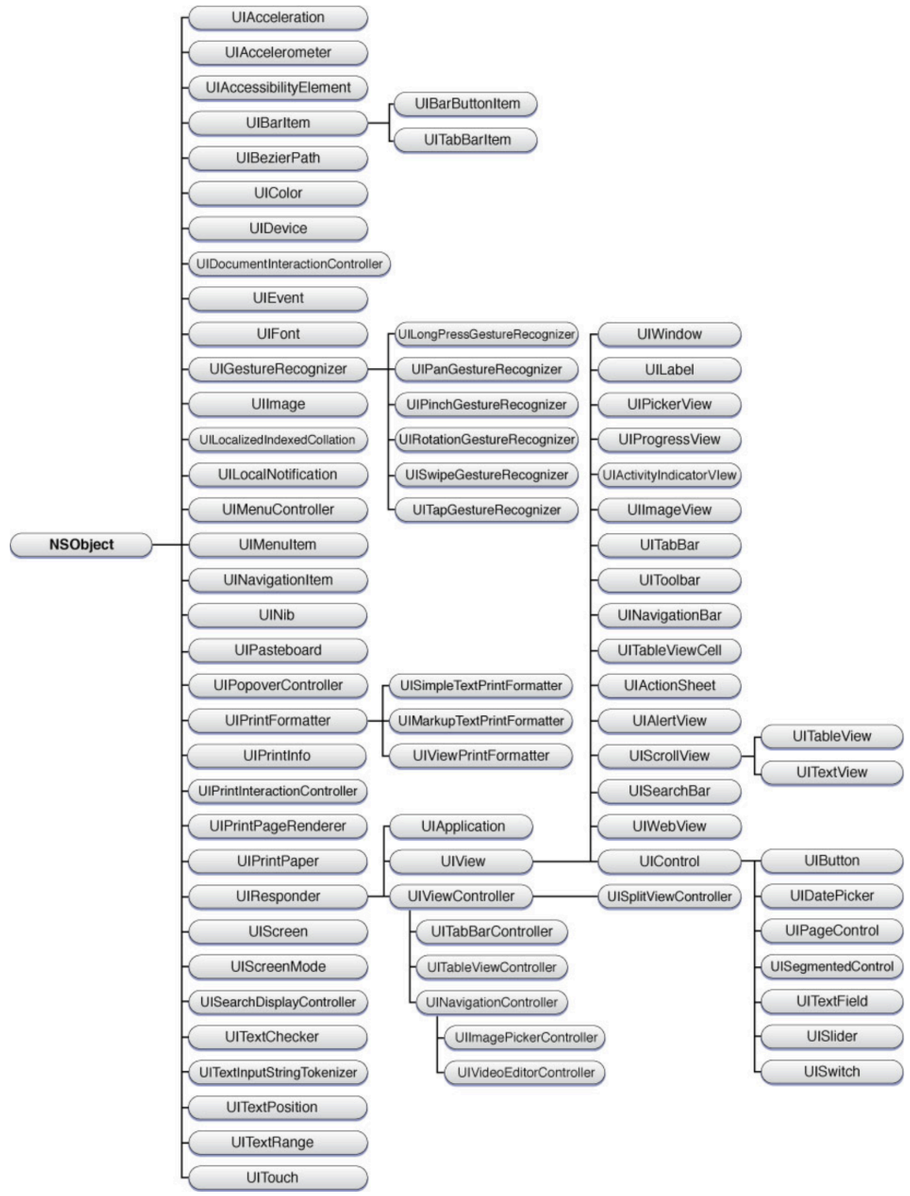
Struktureller Aufbau am Beispiel iOS



- Abstrakter Zugriff auf Hardware
  - Präsentation: Bildschirm, Drucker, Audio
  - permanenter Speicher: Festplatte, Diskette, CD/DVD
  - Systemressourcen: Uhr, FPU, Sensoren, …
  - Kommunikation: V.24, Centronics, SCSI, Ethernet, USB, SATA
  - Software: Dateien, Fonts, Menus, Fenster, …
- Verbergen von:
  - Speicheradressen
  - Registern
  - Kommandos
- Benutzung fertiger Komponenten
- Austausch der Implementierung
  - Beschleunigung
  - neuer Standard
  - Portabilität (WindowsNT auf IA, PowerPC, Alpha, …)
  - anderes Netzwerk (Bsp: Ethernet statt Token Ring)

- iOS
  - für iPhone, iPod touch, iPad





  - LCD, touch-Sensor-Overlay, nur Flash-Speicher
  - UIKit
- Komponenten des UIKit
  - UIApplication
  - UIWindow
  - UILabel
  - UITextView, UITableView
  - UIImageView
  - UISearchbar, UIWebView
  - UISlider, UISwitch

Beispiel: Applikation Framework UIKit

- App wird in iOS-Umgebung eingepasst
  - Events (Ereignisse) von Cocoa/UIKit angenommen
  - als Messages an App-Methoden geschickt

```
#import <UIKit/UIKit.h>
int main(int argc, char *argv[])
{ NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  int retVal = UIApplicationMain(argc, argv, nil, nil);
  [pool release];
  return retVal;                                          }
```

- Delegation
  - App soll auf Messages reagieren
  - klassisch: Callbacks bzw. Listeners, z.B. Eventhandler
  - iOS-Pattern 'Delegate'
  - Methoden erweitern, überschreiben oder hinzufügen
- Delegate Protocol
  - spezifiziert Vorgaben
  - Methoden und Parameter
  - manche müssen vorhanden sein, andere optional
  - UIApplicationDelegate

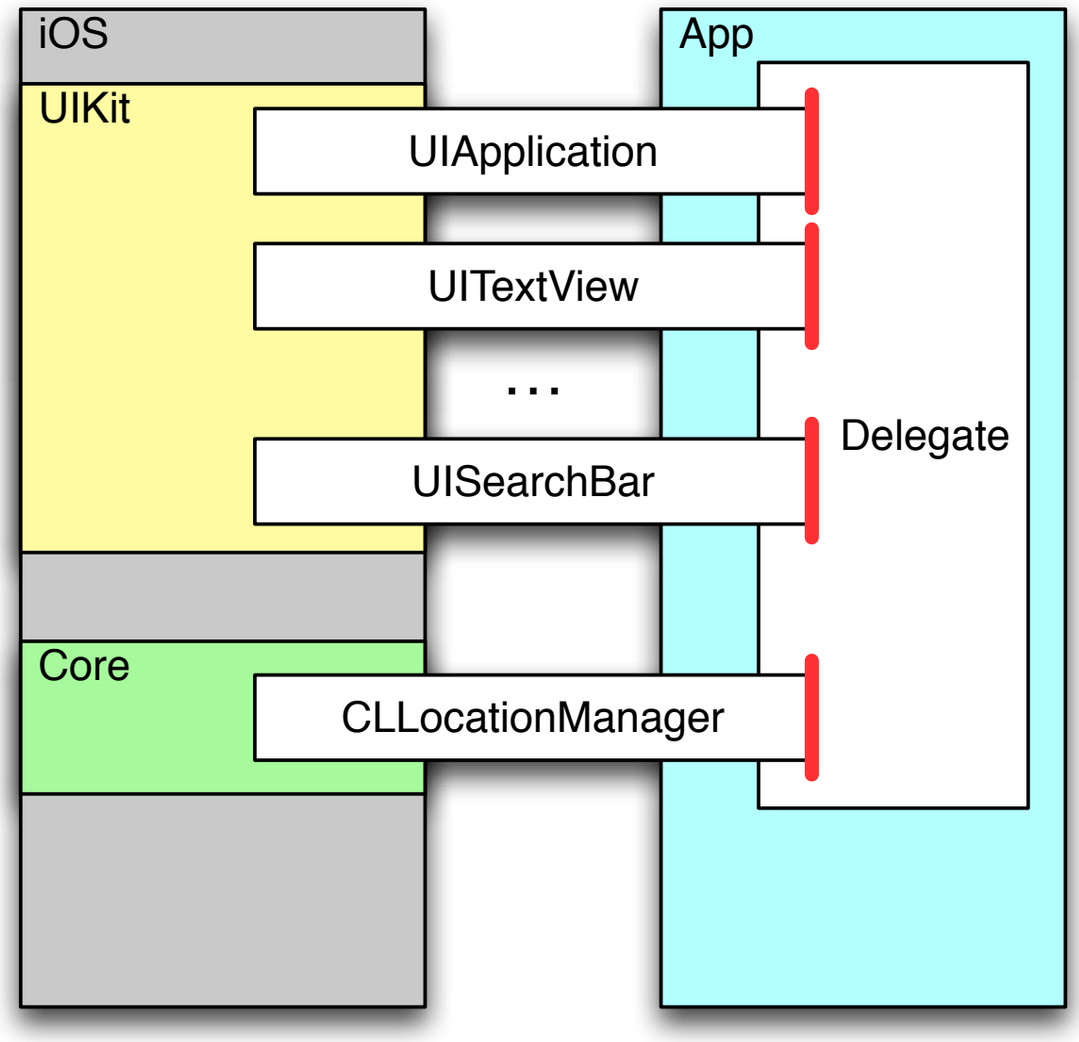
- Struktur einer App
  - AppDelegate mit mehreren Protokollen
  - `@interface windowappAppDelegate : NSObject <UIApplicationDelegate, UISearchBarDelegate, UITextViewDelegate>`
- Delegate
  - Interface zum Scheduler

```
application: didFinishLaunching…:
applicationWillResignActive:
applicationDidEnterBackground:
applicationWillEnterForeground:
applicationDidBecomeActive:
applicationWillTerminate:
applicationDidReceiveMemoryWarning:
```

  - UI-Responder

```
-(IBAction)goPrev : (id)sender;
-(IBAction)goNext : (id)sender;
```

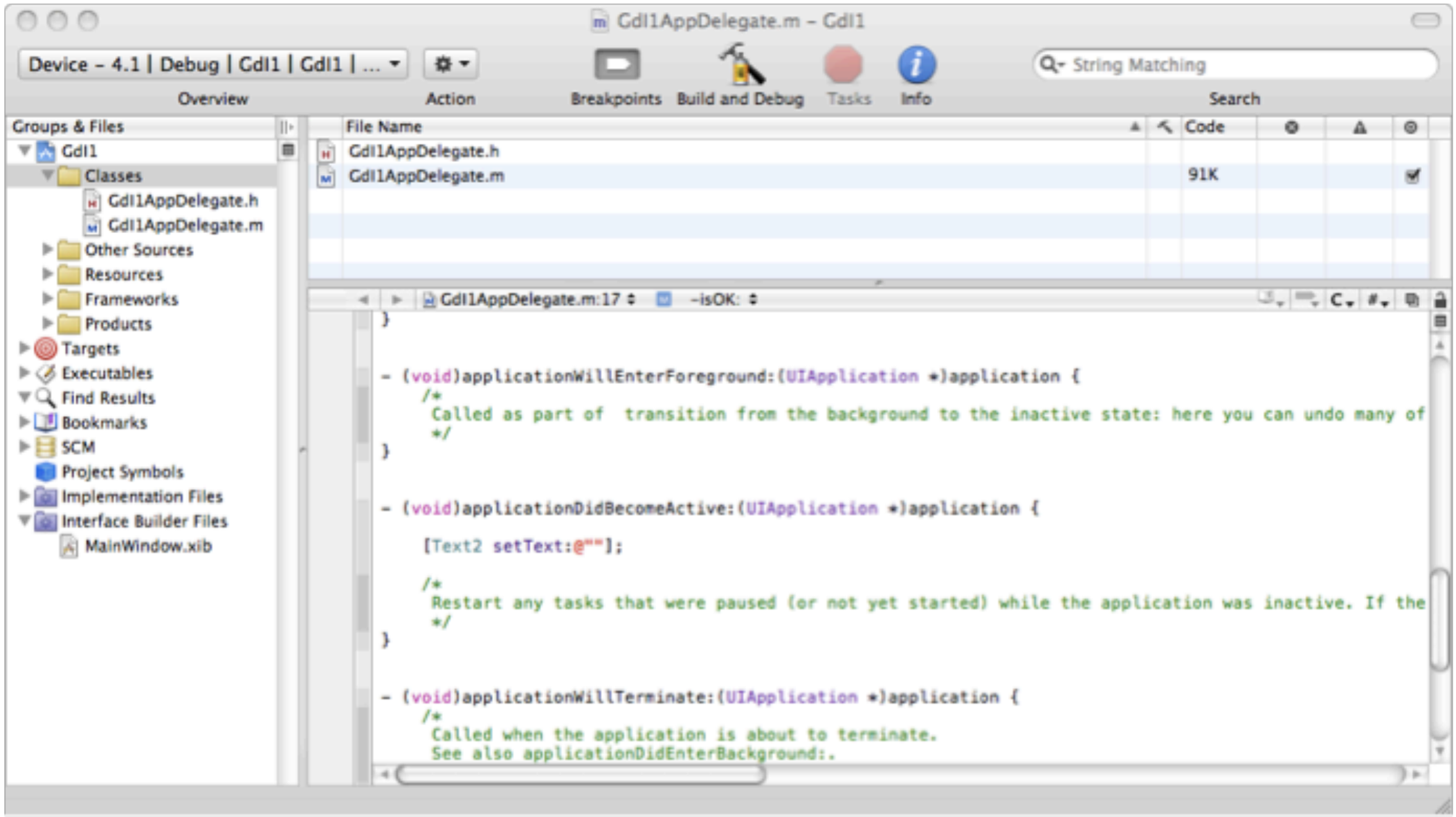
  - weitere (eigene) Methoden
  - importiert weitere (eigene) Klassen



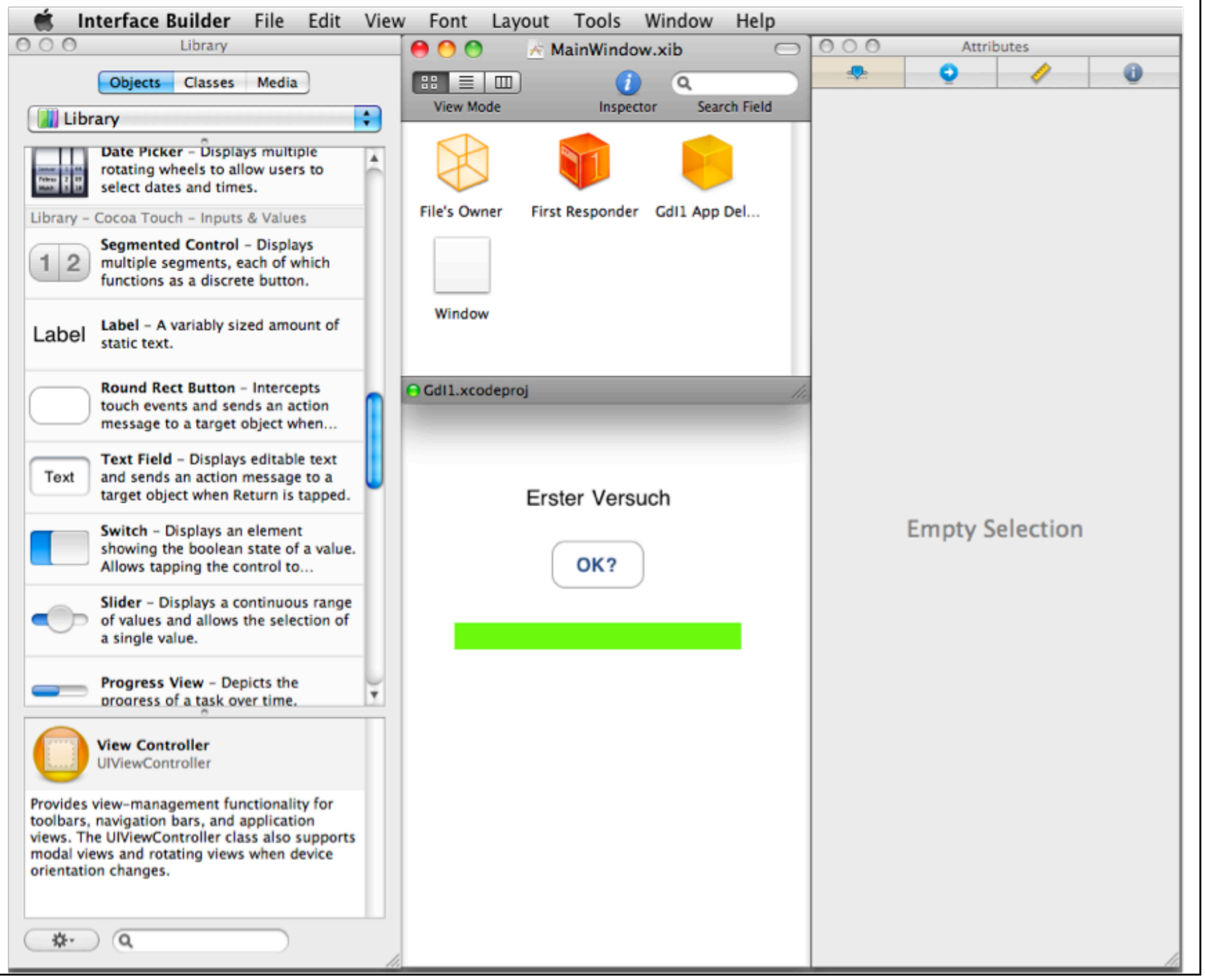
App Programmieren

- Xcode
  - nur auf Intel-Macs
  - Cross-Compiler für ARM
  - Debugger gdb
  - besondere Lizenz …
  - Mac mini im Pool
- Interface Builder
  - graphischer Entwurf der Oberfläche
  - Zuordnung UI-Elemente zu Methoden
- Simulator
  - auf dem Mac
  - interagiert mit Debugger: Breakpoints etc
  - simuliert multitouch
- Device
  - iPod Touch, iPad, iPhone
  - mit USB-Dock

- Xcode
  - Neues Projekt
  - Projekt-Vorlagen
  - generiert Delegate-Klasse
  - Resources: Dateien, Bilder, …



- Interface Builder
  - Tool zur grafischen Design der User Interface
  - viele Klassen in Library
  - Label
  - Button
  - Text-Feld
  - ScrollView
  - imageView
  - Searchbar
  - …
  - Eigenschaften
  - Verbindungen



- AppDelegate Interface
  - Ausgabeelemente: IBOutletq
  - Methoden für Eingabe-Elemente
  - (IBAction) als Resultat
  - evtl. weitere DelegateIF

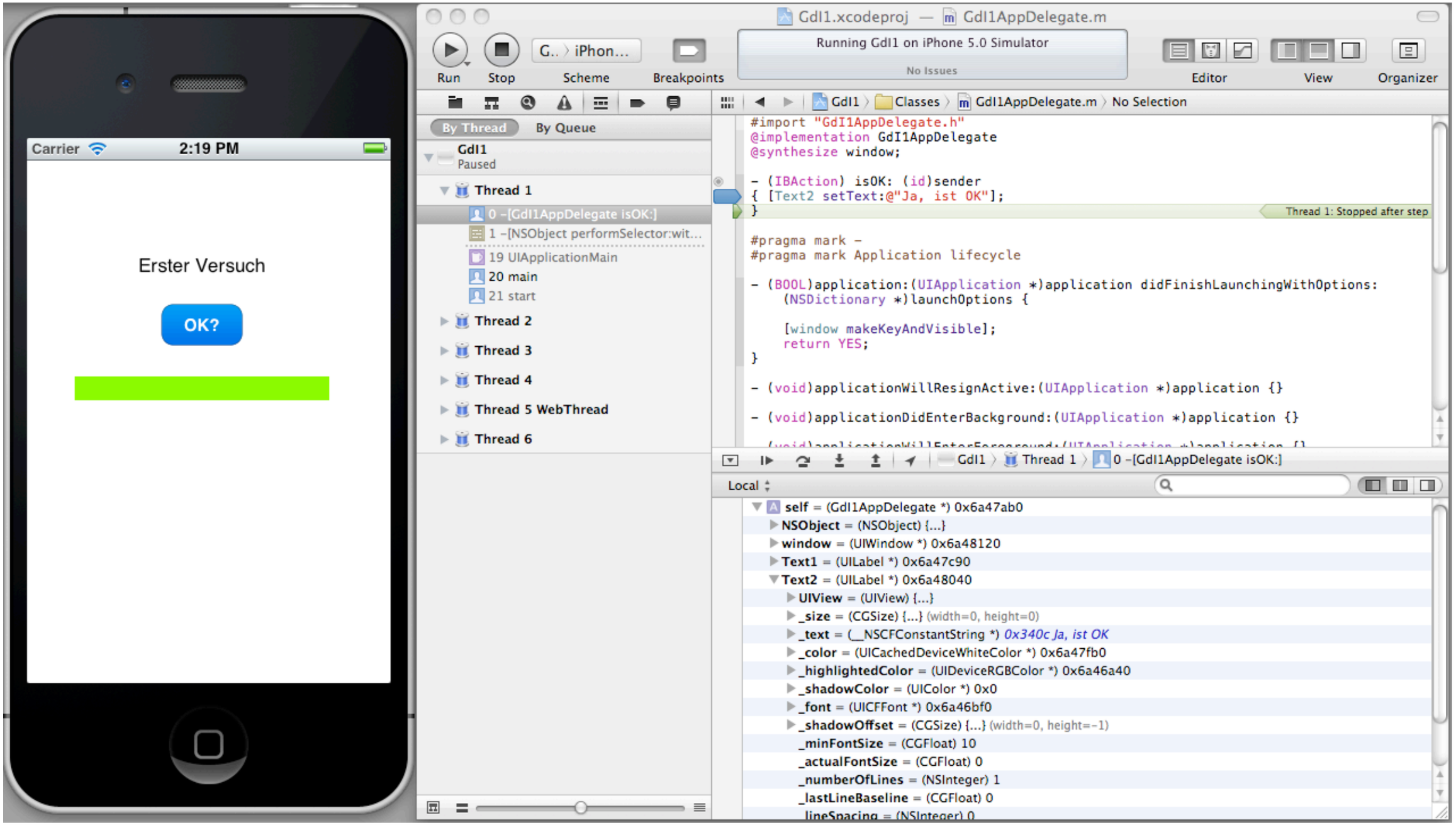
```
- (IBAction) isOK: (id)sender
{ [Text2 setText:@"Ja, ist OK"];
}
```

```
#import <UIKit/UIKit.h>

@interface GdI1AppDelegate :
      NSObject <UIApplicationDelegate> {
    UIWindow *window;

   IBOutlet UILabel *Text1;
   IBOutlet UILabel *Text2;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
- (IBAction) isOK: (id)sender;
@end
```

- AppDelegate Implementation
  - Ausgabe: Properties von IB-Elementen setzen
  - Methoden für Eingaben implementieren
  - Methoden überschreiben

```
- (void)applicationDidBecomeActive:(UIApplication *)application {

   [Text2 setText:@""];
}
```

- Debugger und Simulator



- Komplexere Apps
  - scrollView übernimmt pan & zoom mit Gesten
  - Eingabe in textView
  - searchBar, mapView, webView, …
  - 'Protokolle' werden implementiert

```
#import <UIKit/UIKit.h>
@interface windowappAppDelegate : NSObject <UIApplicationDelegate,
   UIScrollViewDelegate, UISearchBarDelegate, UITextViewDelegate>
{      int current;
//  weitere Variablen
       IBOutlet UISearchBar *searchBar;
//  weitere outlets
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UISearchBar *searchBar;
@property (nonatomic, retain) UIScrollView *scrollView;
@property (nonatomic, retain) UIImageView *imgBild;

- (IBAction)goPrev : (id)sender;
//  weitere Methoden
@end }
```

```
- (void) searchBarSearchButtonClicked:(UISearchBar *)aSearchBar
{  int oldcurrent = current;
   NSString *searchText = [searchBar text];
   if ([searchText length] != 0) {
       for (int i=0;i< [numbers count];i++)
           if ([searchText isEqualToString: [numbers objectAtIndex:i]]) current = i;
   if (current != oldcurrent) [self display :current];
}   }
```

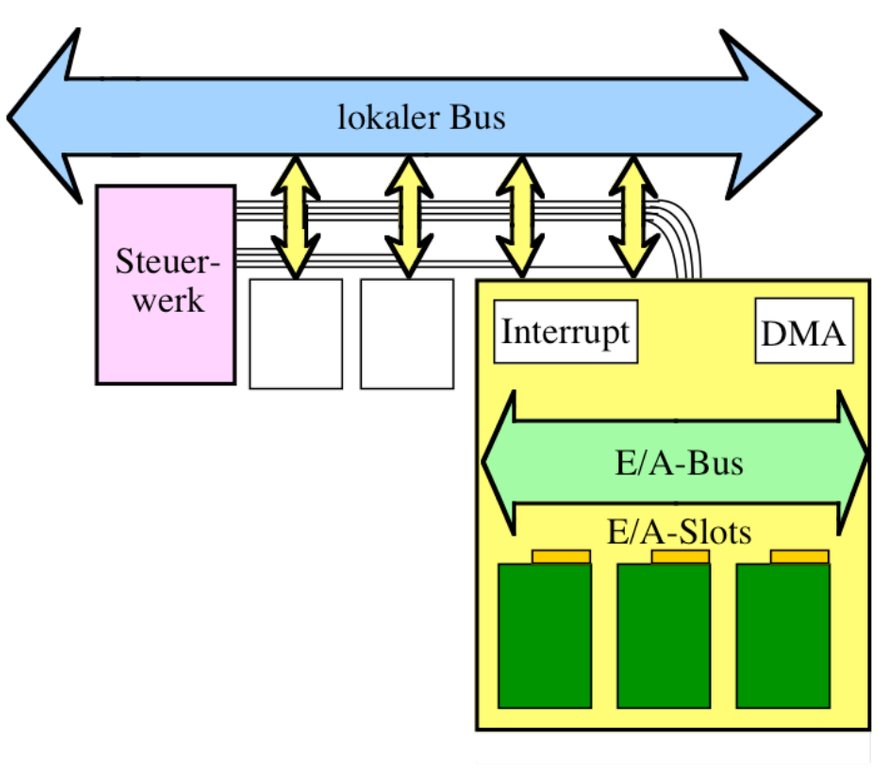
KAPITEL 6

Rechnerarchitektur

- Vom Transistor zum Programm
- Schaltfunktionen
- Hardwarestrukturen
- Transport, Speicherung, Rechnen

Hardware: The parts of a computer system that can be kicked. [Jeff Pesis]

Software wird schneller langsam als Hardware schneller wird [M. Reiser]



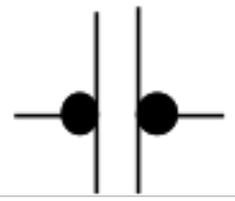
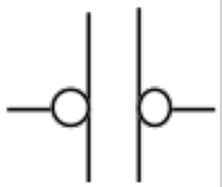
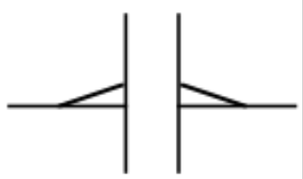
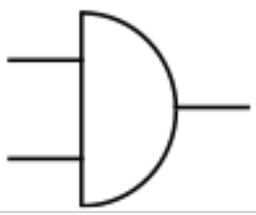
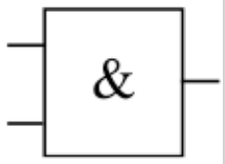
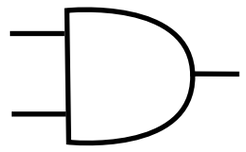
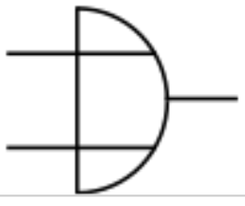
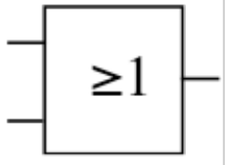
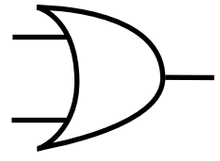
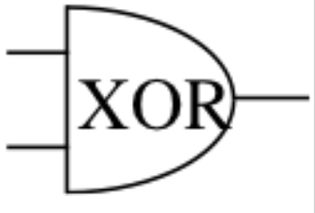
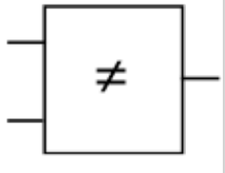
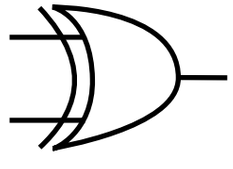
Transistoren, Gates

- Schaltfunktionen
  - 1..n Eingänge, 1..m Ausgänge
  - 2^n Argumentkombinationen
  - 2^{2^n} mögliche Funktionen
- 2-stellige Schaltfunktion

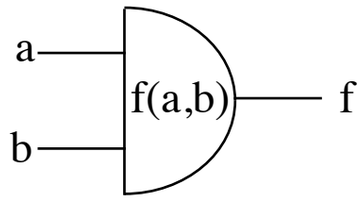
V_1	V_2	V_3	…	V_n	f_1	f_2	…	f_{max}
1	1	1		1	0	0		1
0	1	1		1	0	0		1
0	0	1		1	0	0		1
0	0	0		0	0	1		1

		A=1, B=1	A=1, B=0	A=0, B=1	A=0, B=0
f0		0	0	0	0
f1		0	0	0	1
…		…	…	…	…
f8	AND	1	0	0	0
f14	OR	1	1	1	0
f15		1	1	1	1

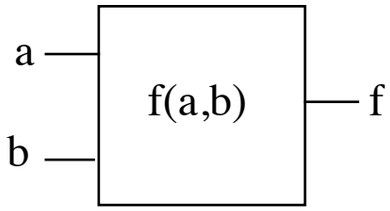
- Symbolische Darstellung von Schaltkreisen
  - Schaltplan
  - DIN 40700, IEEE / ANSI Y32.14, IEC

Funktion	Operator	graphisches Symbol		
		DIN	IEEE	IEC
Negation	$\neg$, $\overline{\ }$			
Und	$\cdot, \wedge, *$			
Oder	$+, \vee$			
Exklusiv Oder	$\oplus, \neq$			

- Allgemeine Schaltfunktionen:

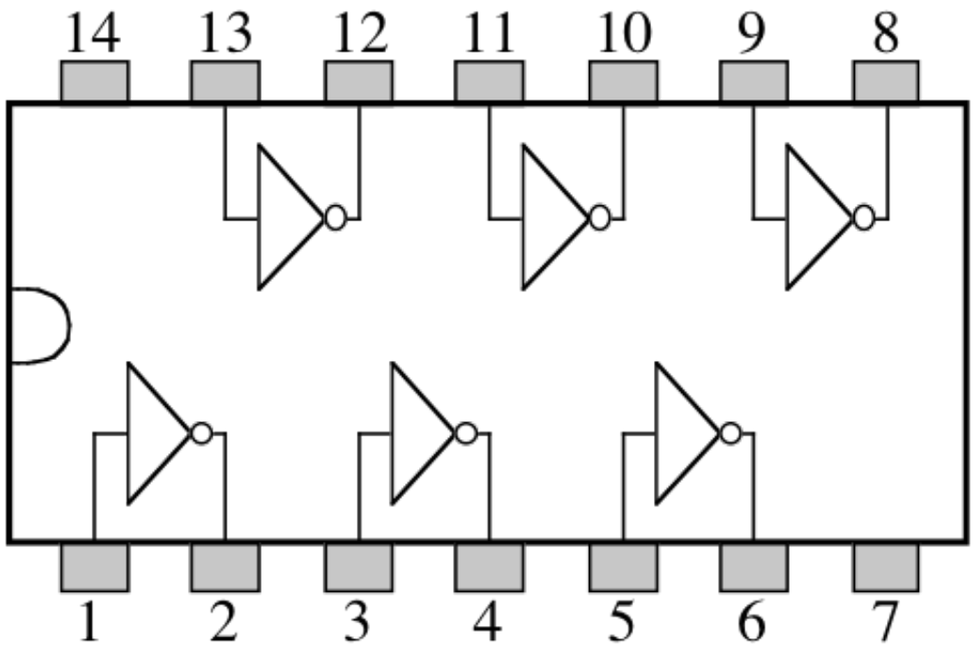


bzw.

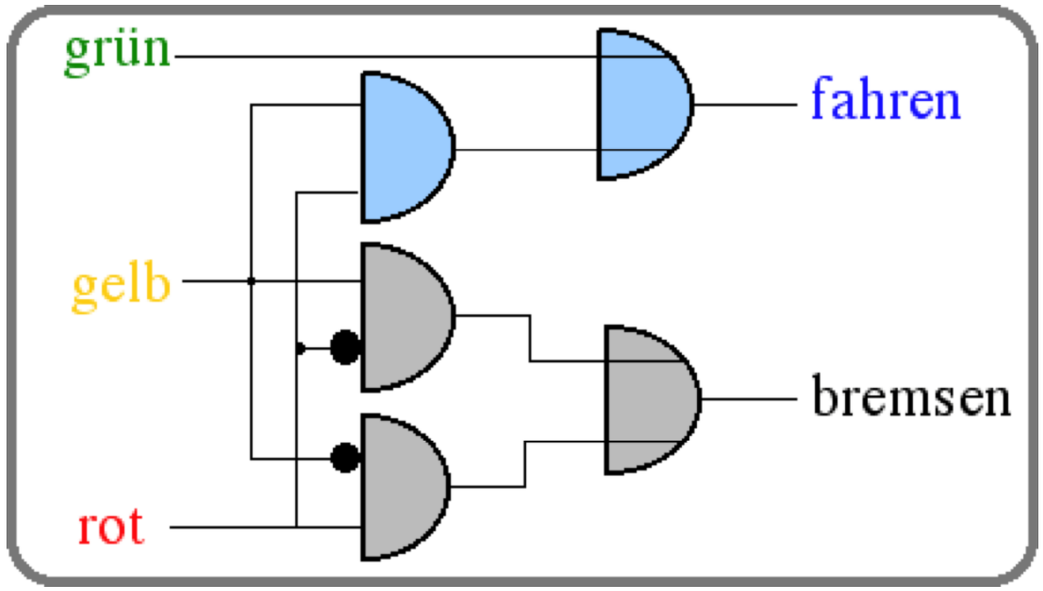


- Inverter, Verstärker:

- Hex Inverter 7404 als Chip:
  - Stift #7: 0 Volt, Erde, GND
  - Stift #14: z.B. +5 Volt:

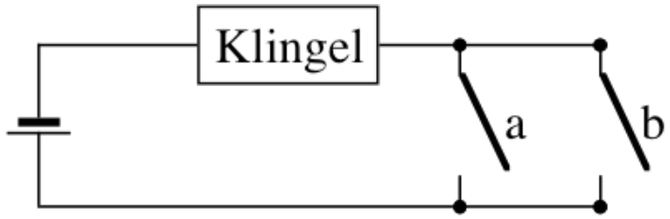


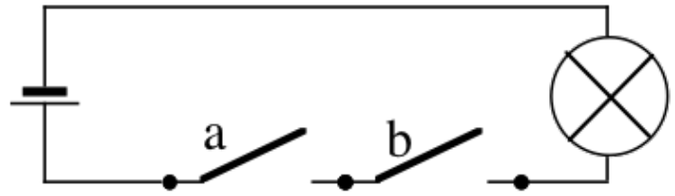
- Ampel-Reaktion

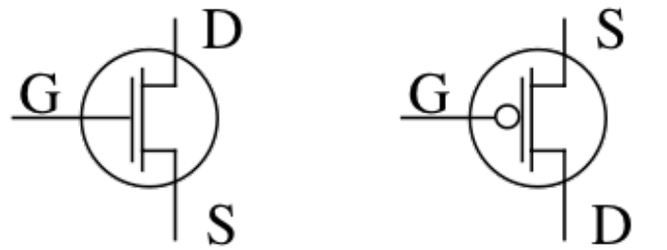


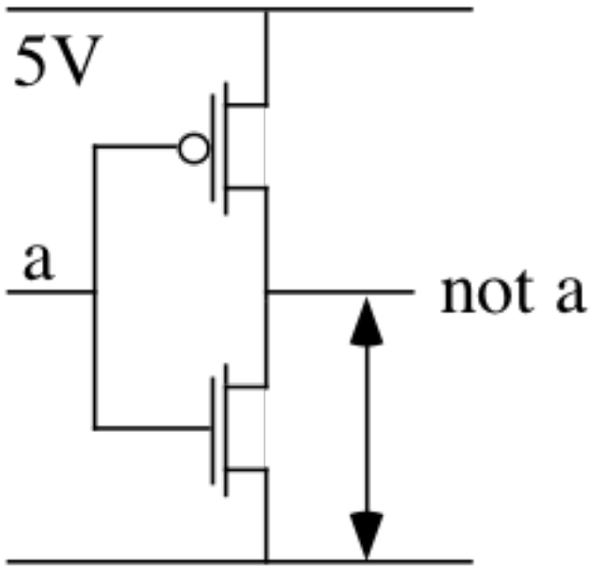
- Schaltalgebra
  - Schaltfunktionen
  - Rechenregeln
  - Normalform
  - Minimierung

- ODER (OR) Schaltung
  - a ∪ b
  - Disjunktion, logische Summe, Vereinigung
- UND (AND) Schaltung
  - a ∩ b
  - Konjunktion, logisches Produkt, Durchschnitt
- Feldeffekttransistor (FET)
  - Gate-Source-Spannung erzeugt Feld
  - Feld kontrolliert Stromfluss im Drain-Source-Kanal
  - U_{GS} steigt -> I_{DS} steigt exponentiell
- Negation (NOT)

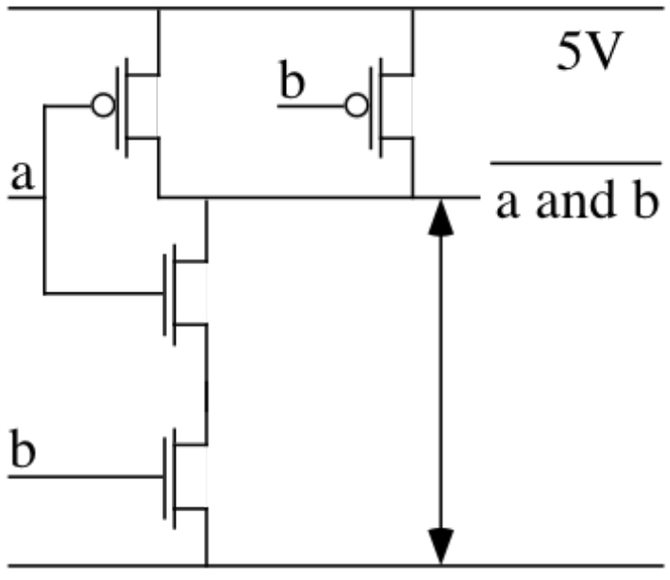


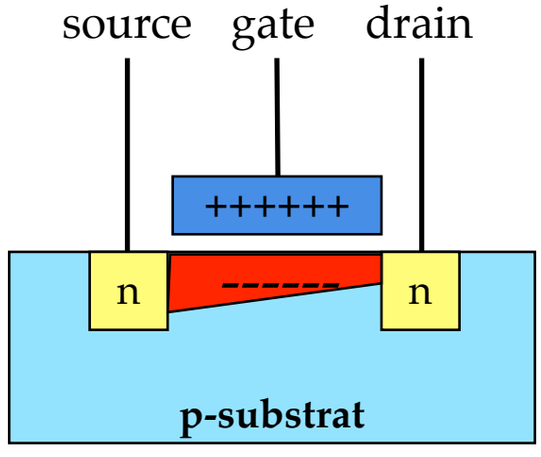






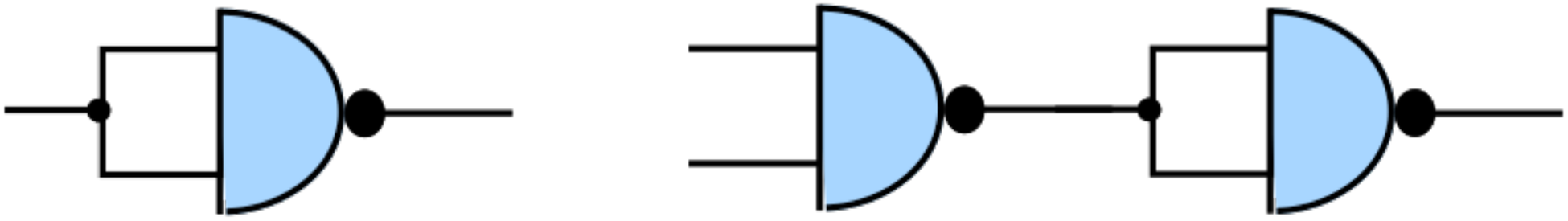
NAND





- Boolsche Algebra
- Kombination von NICHT mit UND / ODER
  - $\overline{A \wedge B} = \overline{A} \vee \overline{B}$
  - $A \vee B = \overline{\overline{A} \wedge \overline{B}}$
  - $A \wedge B = \overline{\overline{A} \wedge \overline{B}}$
- NAND und NOR einfacher zu bauen
  - CMOS: complementary Metal-Oxide-Silicon
  - 4 Transitoren in CMOS-Technik

  Inverter aus 1 NAND UND aus 2 NANDs



- 74xx, 74HCTxx, …
  - Einfache Logik-Chips
  - 7404: 6 Inverter
  - 7400: 4 NAND-Gatter
  - 7402: 4 NOR-Gatter
  - 7420: 2 NAND-Gatter mit je 4 Eingängen

Rechnen und Speichern

- Addition von zwei einstelligen Binärzahlen
  - Eingangsvariablen
  - Summenausgang
  - Übertrag zur nächsten Stufe

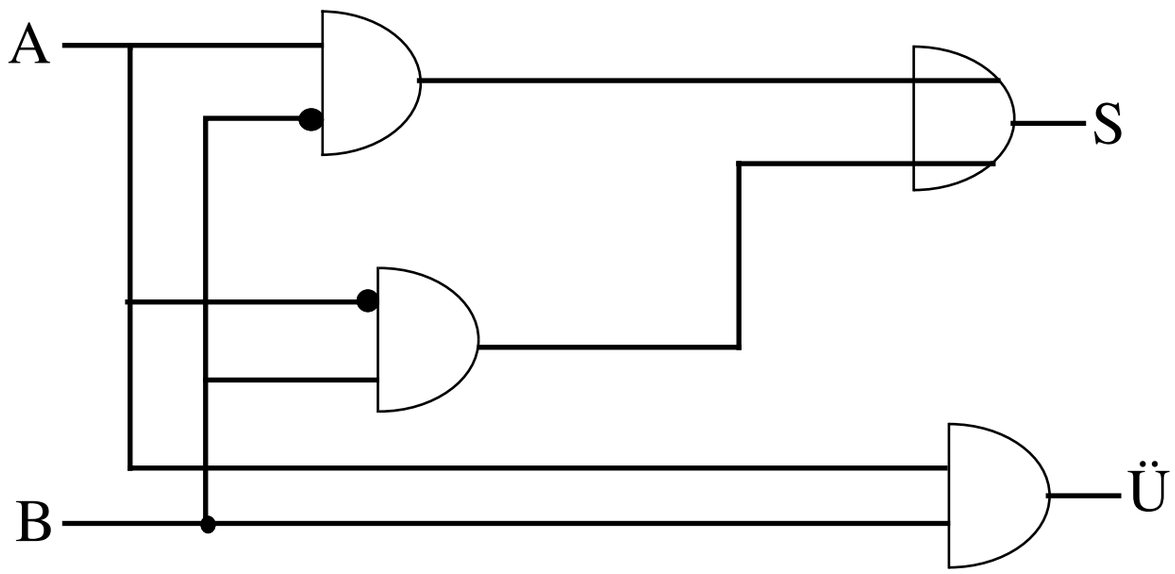
- Wahrheitstafel

A	B	S	Ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

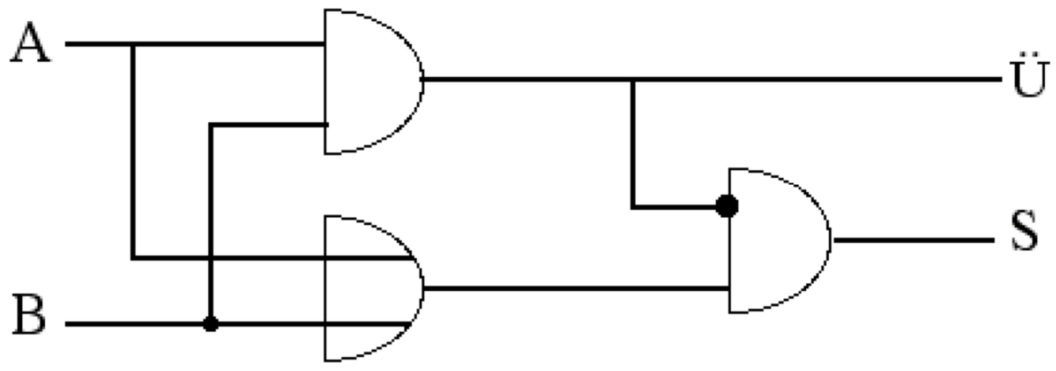
- Addierer kann aus sogenannten Halbaddieren zusammengesetzt werden
- Halbaddierer berücksichtigt nicht den Übertrag der früheren Stufe

- Halbaddierer

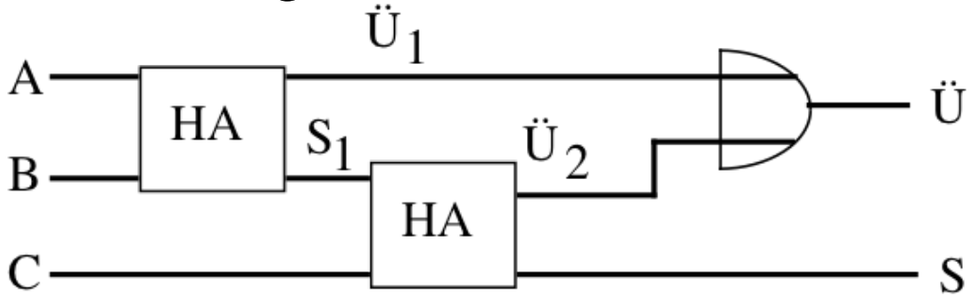
$S = A\bar{B} \vee \bar{A}B \qquad Ü = AB$



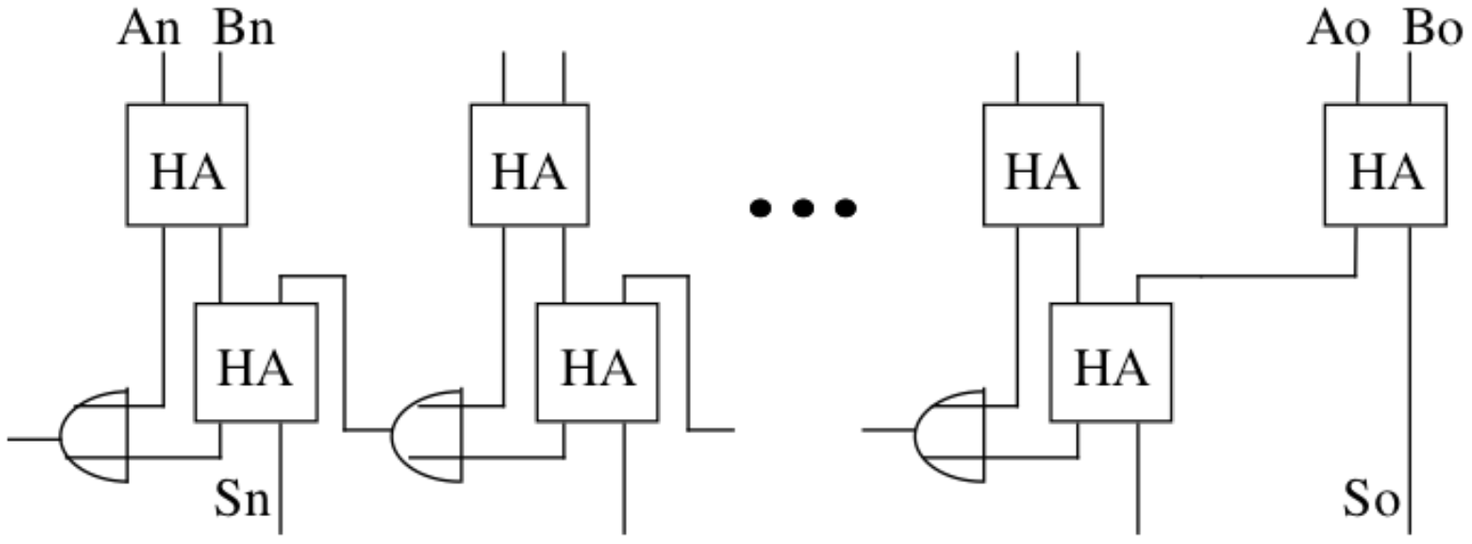
- Vereinfacht



- Volladdierer
  - für eine Binärstelle
  - berücksichtigt auch den Übertrag einer früheren Stufe

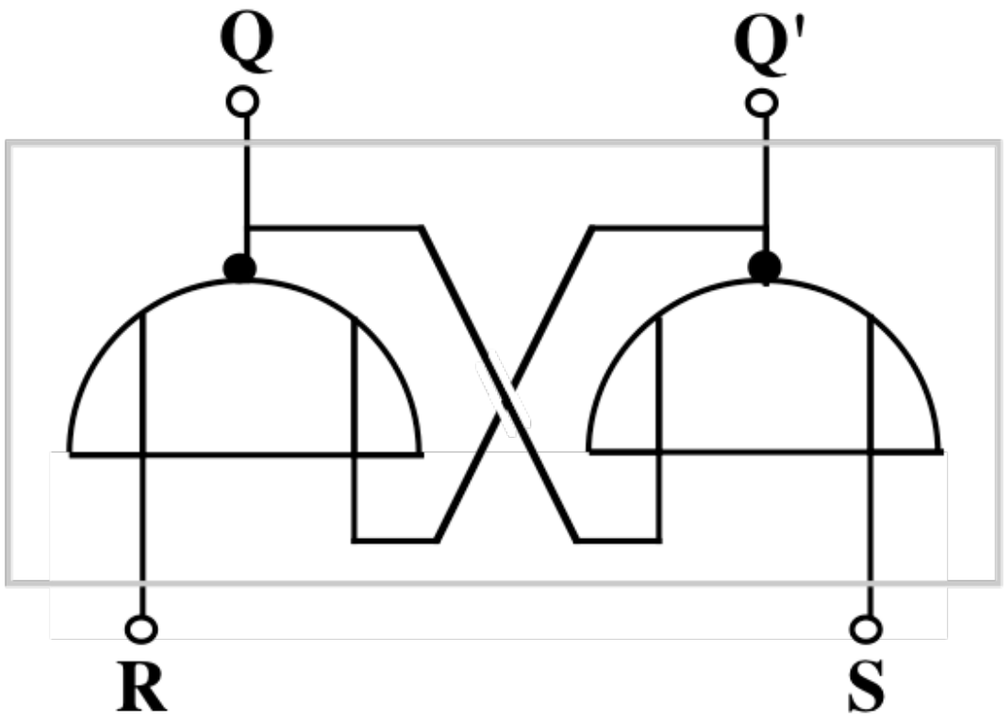


- Paralleladdierer
  - für die Addition eines Binärwortes
  - die Summen der jeweiligen Binärstellen parallel bilden
  - Übertrag durch die Stufen fortpflanzen lassen (Delay!)

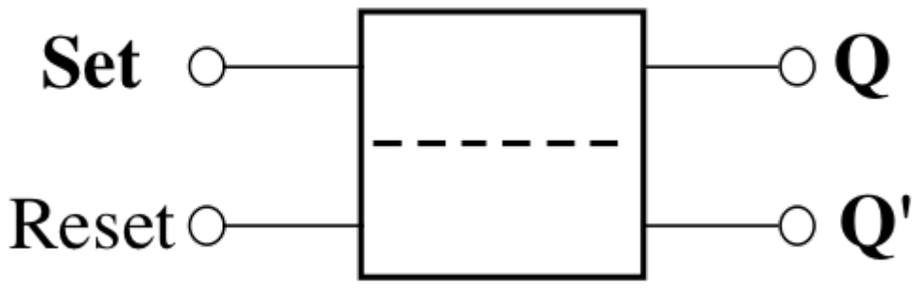


-> Carry Lookahead

- Einfache Speicherzelle: Flip-Flop
  - Speichern einer Binärstelle
  - die beiden Hälften halten sich gegenseitig



- sog. RS-Flip-Flop:
  - **S**etzen ("Set"),
  - **R**ücksetzen ("Reset")

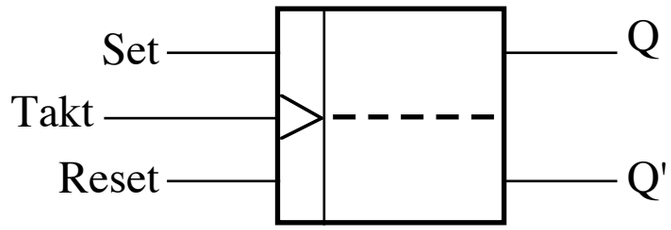


- Schaltfunktion für RS-Flip-Flop:
  - $\mathbf{Q = \overline{R \vee Q'} = \overline{R \vee \overline{(S \vee Q)}} = \overline{R} \wedge (S \vee Q)}$
  - $\mathbf{Q' = \overline{(S \vee Q)} = \overline{S} \wedge (R \vee Q')}$
- Wahrheitstafel:

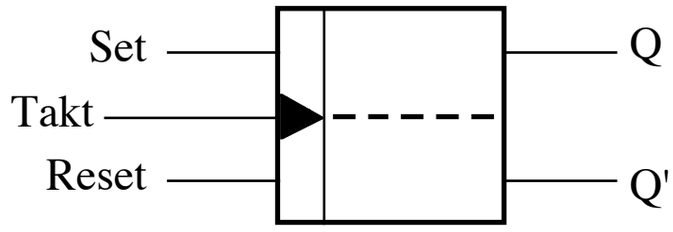
R	S	Q_n	Q'_n
0	0	Q_{n-1}	Q'_{n-1}
0	1	1	0
1	0	0	1
1	1	0	0

- Undefinierter Folgezustand für **R=1; S=1**
- Zwei Speicherungs-Zustände mit **R=0; S=0**:
  - $Q = 0; Q' = 1$
  - $Q = 1; Q' = 0$
  - fast immer $Q \neq Q'$!
- Zwischenzustände während Umschaltung

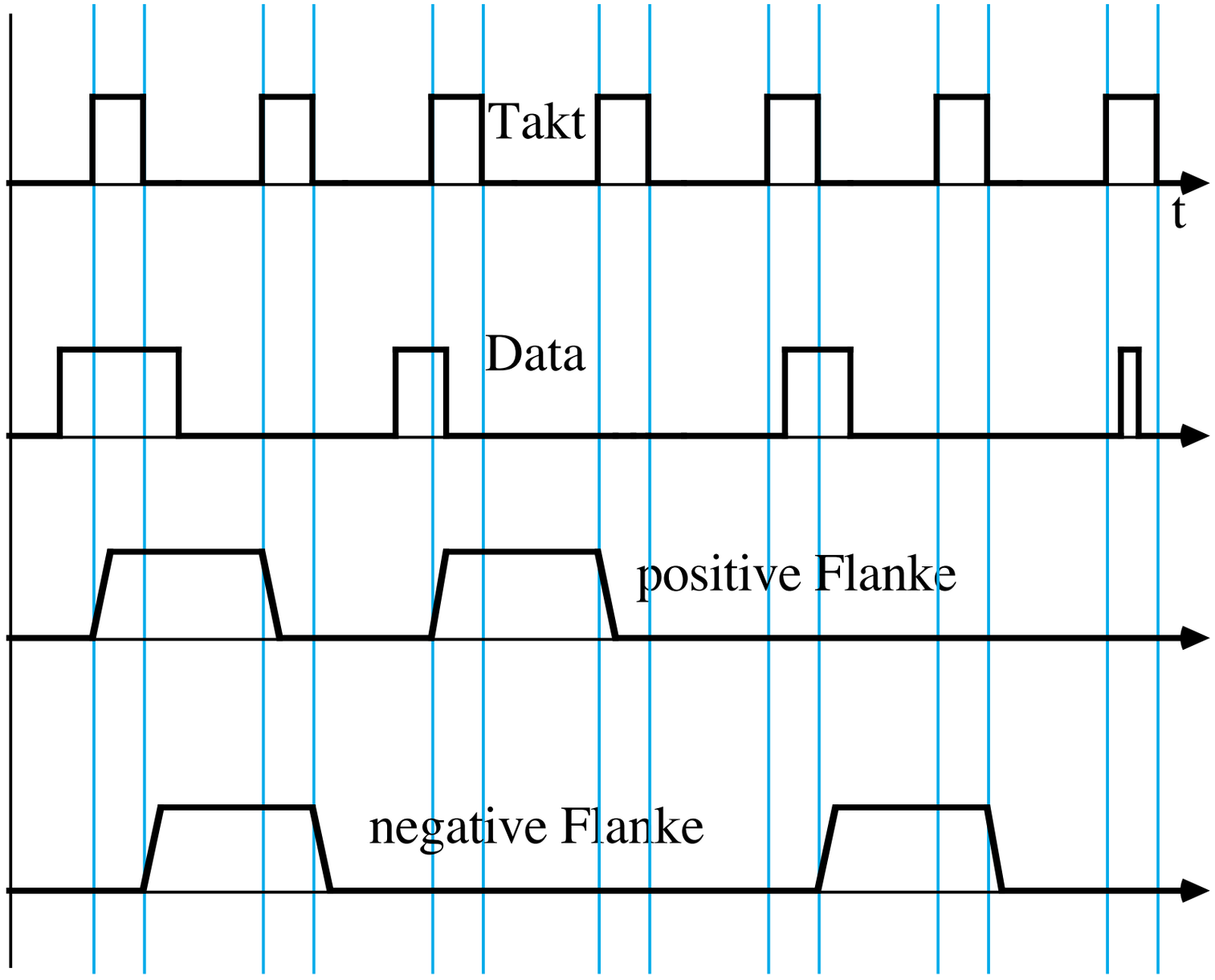
- Taktsteuerung
  - Eingangswerte stabilisieren lassen
  - dann Übernahmepuls
  - Eingangswerte werden nur zum Taktzeitpunkt berücksichtigt:
- Flankengesteuertes Flipflop (abfallend):



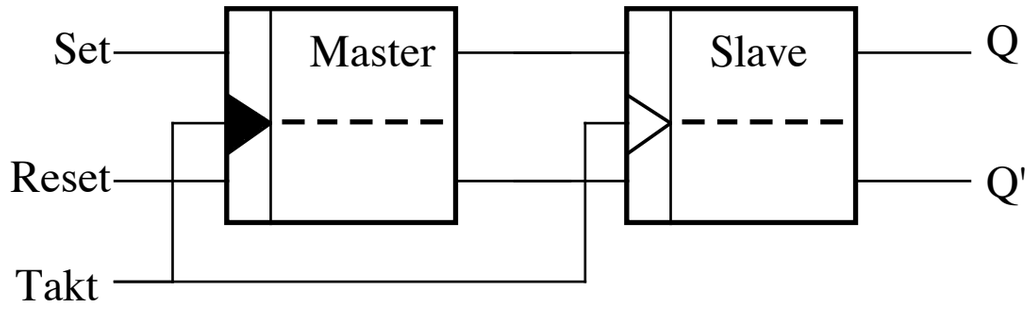
- Flankengesteuertes Flipflop (ansteigend):



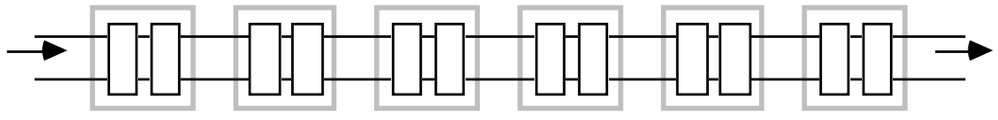
- Signalverlauf
  - erst bei entsprechender Taktflanke Eingabe einlesen
  - evtl. unterschiedliche Ergebnisse



- Master-Slave Flip-Flop
  - Zwei FF-Stufen arbeiten phasenverschoben
  - Master übernimmt Eingangswerte mit ansteigender Flanke
  - Slave gibt mit abfallenden Flanke Werte an Ausgang

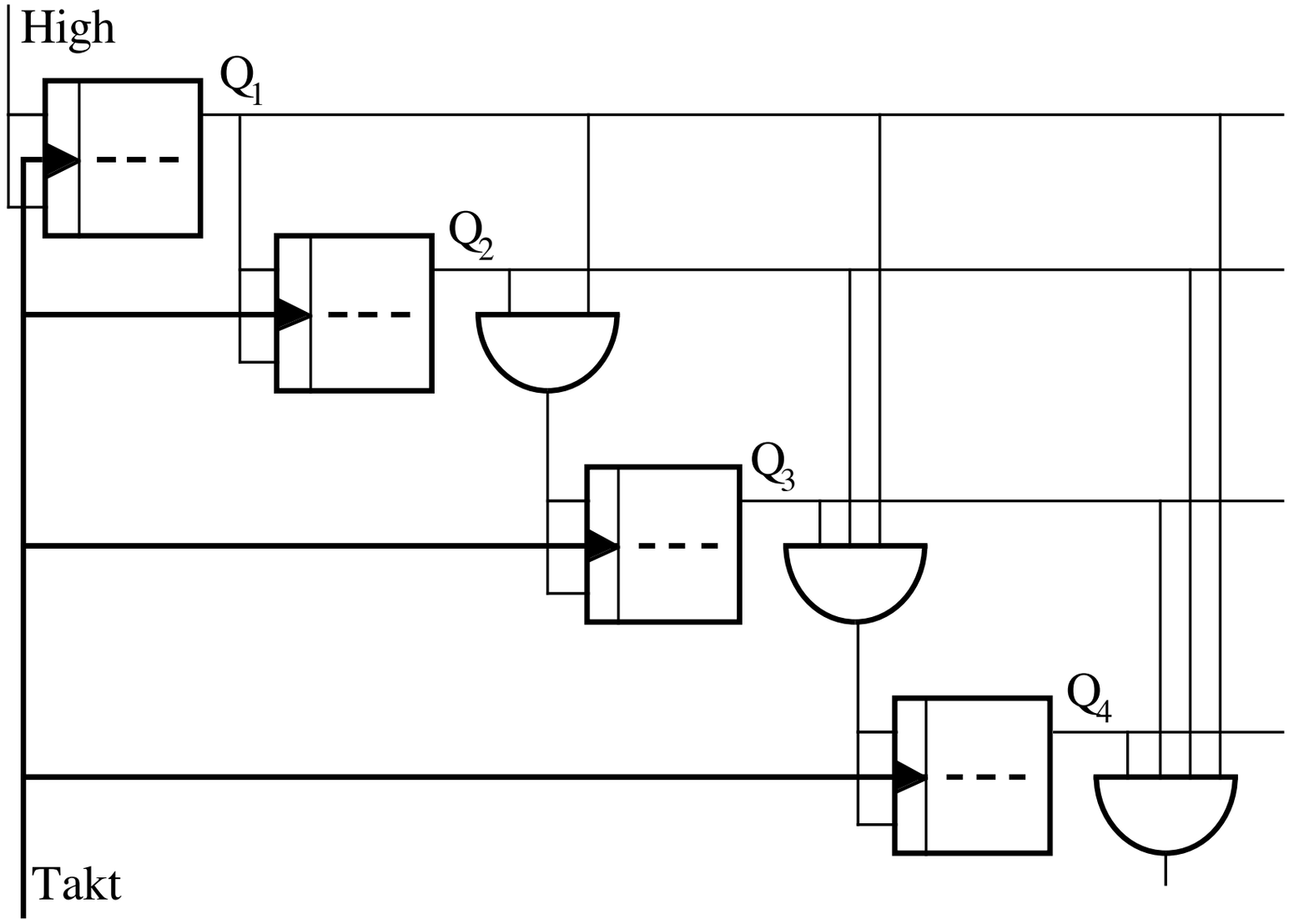


- Für Schaltungen mit heiklem Timing
  - Registertransfers.
  - Pufferregistern mit MS-FFs.
- MS-Schieberegister

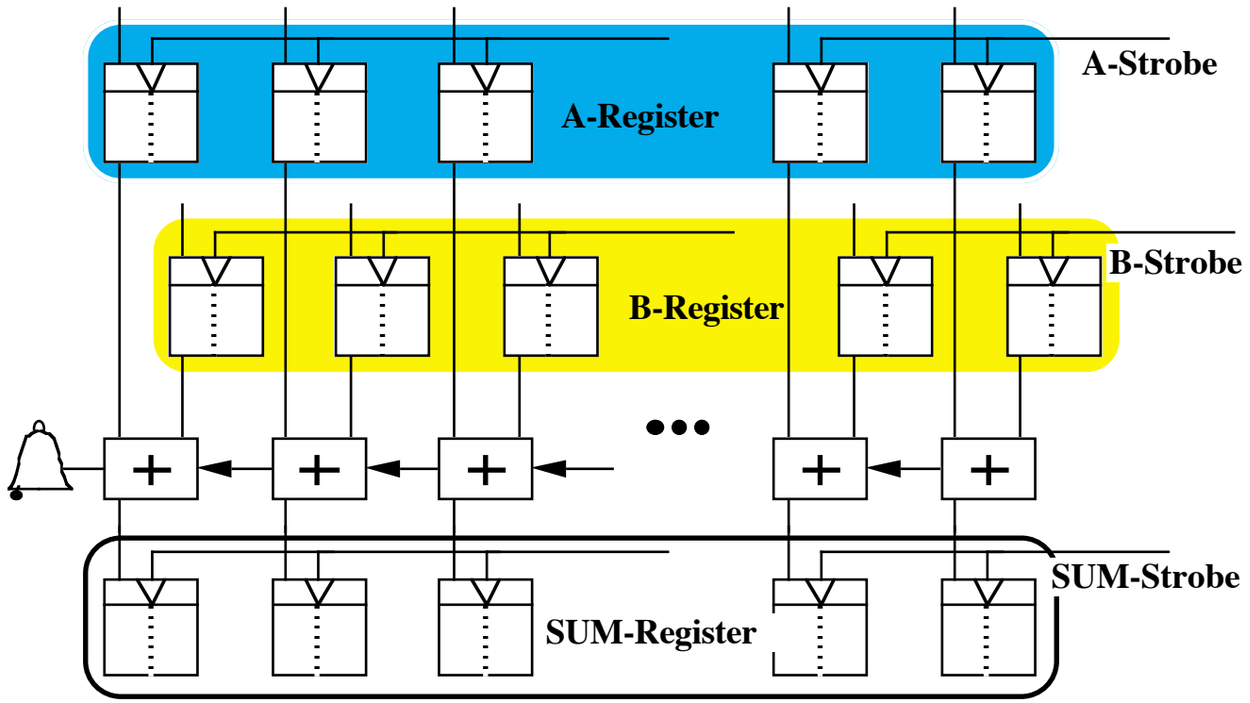


  - als digitale Verzögerungsleitung
  - für digitale Filter
  - als Rechenoperation

- Zähler
  - Jede Stufe erhält Takt
  - schaltet in Abhängigkeit vom Zustand aller vorherigen Stufen



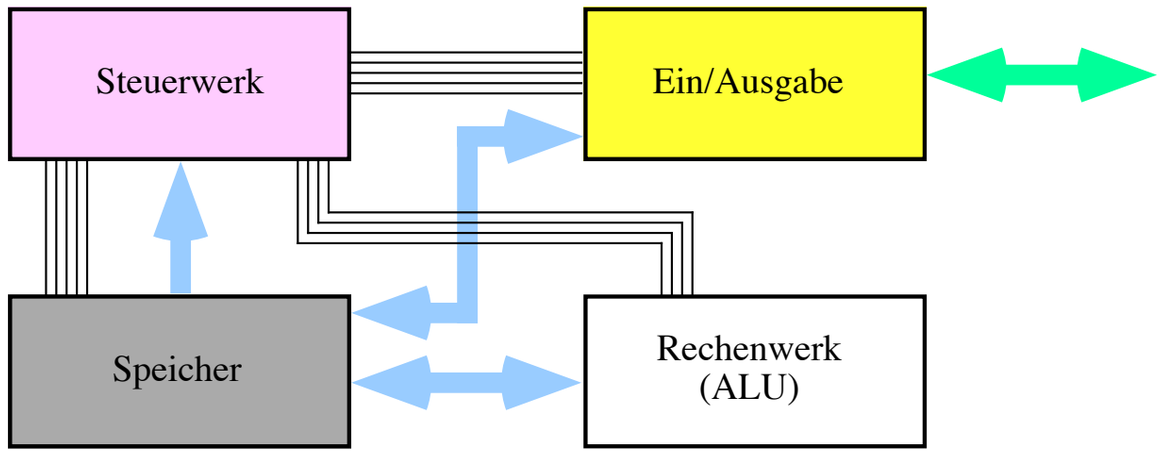
- Register speichert Zahlen
- Register in Verbindung mit Addierschaltung bzw. ALU



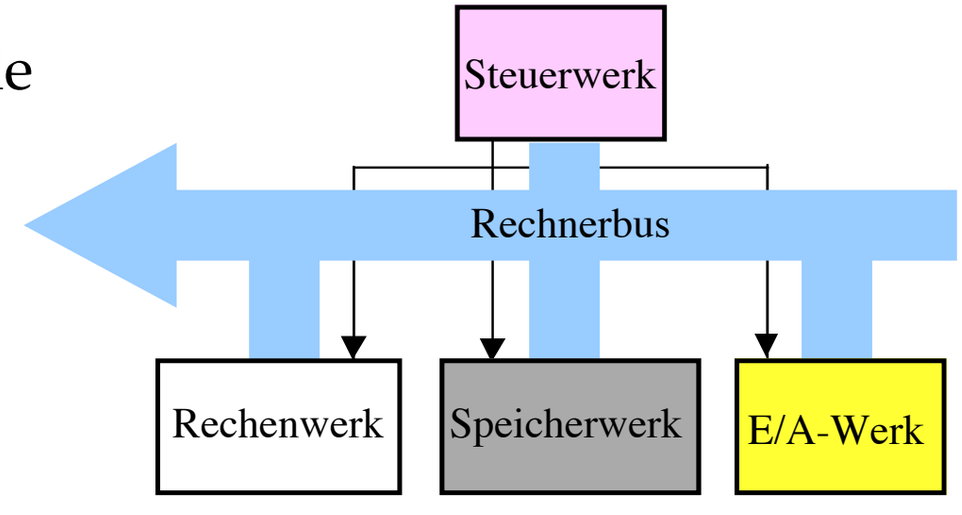
- Ablauf
  - A-Strobe zum Füllen des Registers A
  - B-Strobe zum Füllen des Registers B
  - Addition bzw. Übertrag abwarten,
  - Summe abholen mit SUM-Strobe
  - Überlauf

Funktionseinheiten: Speicher, Prozessor, Bus, …

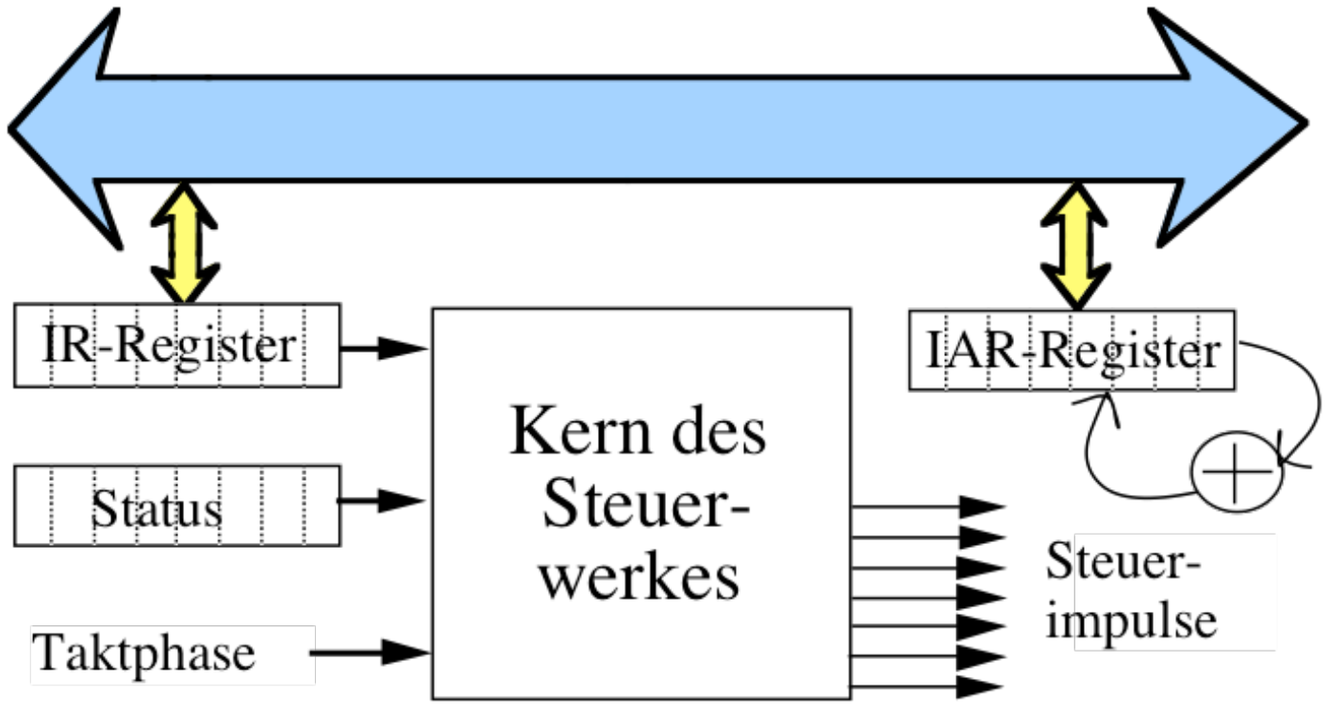
- Klassischer Rechner nach J. v. Neumann
  - Rechenwerk für arithmetische und logische Verknüpfungen
  - Universalspeicher für Daten und Programme



- Steuerwerk
  - sequentieller Ablauf des Programmes
  - Interpretation der Instruktionen => Steuerbefehle
- Busorientierter Rechner
  - universell verbunden
  - Schwerpunkt Transport und Verteilung
  - weniger Verbindungen

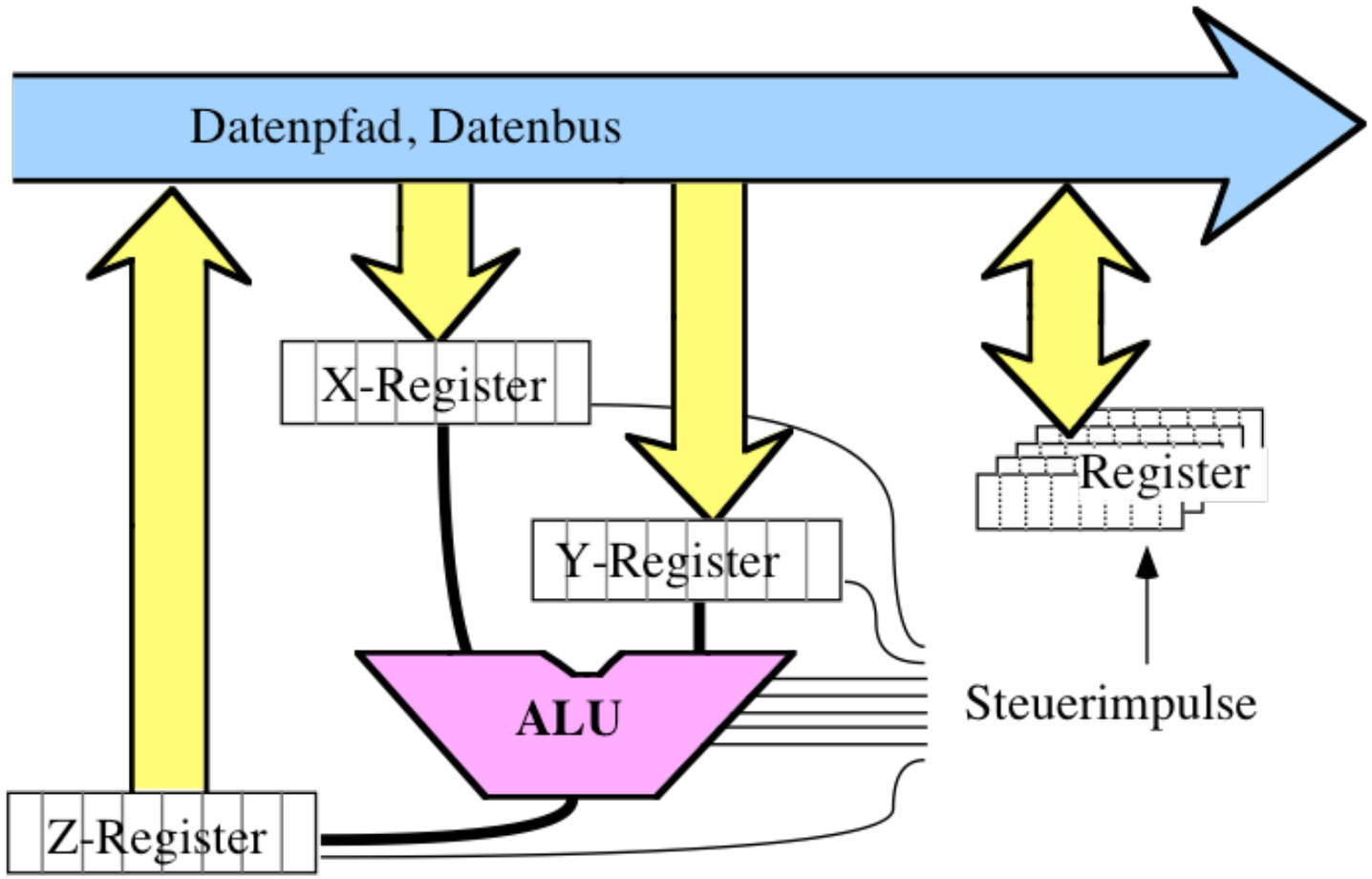


- Steuerwerk erzeugt Signale, die
  - Datentransfer auslösen
  - ALU-Operation auswählen
  - Speicheroperation auslösen



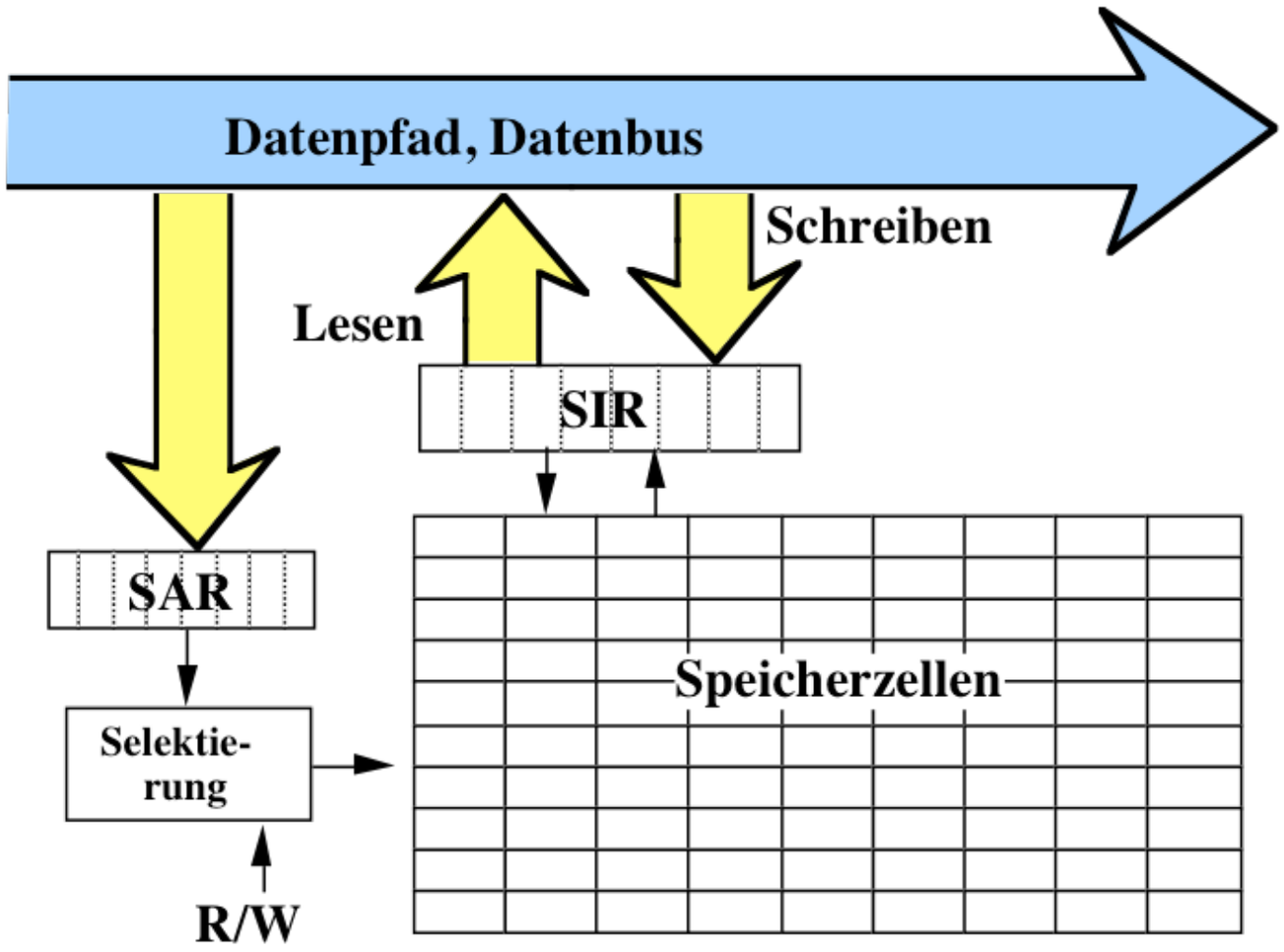
- Befehl wird in Sequenz von Kernzuständen umgesetzt
  - Kernzustand bestimmt Steuersignale
- Register
  - Instruktionsregister (IR)
  - Instruktionsadressregister (IAR) bzw. "Program Counter" (PC)
  - eventuell eigener Addierer für IAR

- Rechenwerk
  - kombinatorisches Schaltnetz
  - Addieren, Multiplizieren, UND, ODER, Vergleich, Shift, …
  - Ausgang der ALU liegt dauernd am Z-Register an
  - X-Register und Y-Register liegen dauernd am ALU-Eingang an



  - Zwischenresultate in Zusatzregistern
  - Steuerleitungen bewirken Datenübernahme = Transport

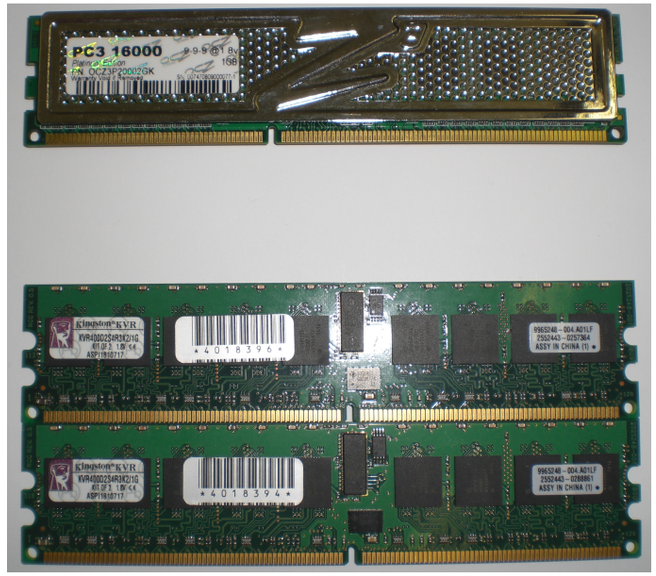
- Speicher



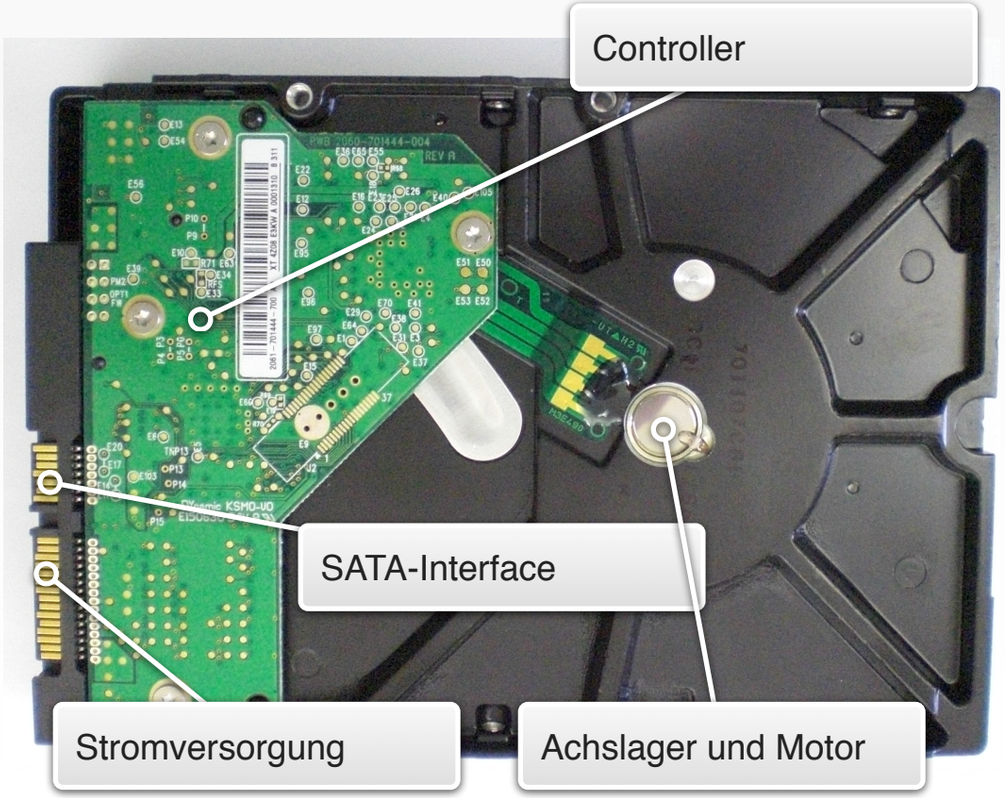
SAR = Speicheradressregister, SIR = Speicherinhaltsregister

- Random Access Memory (RAM)
  - Halbleiterspeicher halten Inhalt nur wenn sie "unter Strom" stehen
  - dynamische Speicher (DRAM) müssen periodisch aufgefrischt werden
- ROM: Festwertspeicher

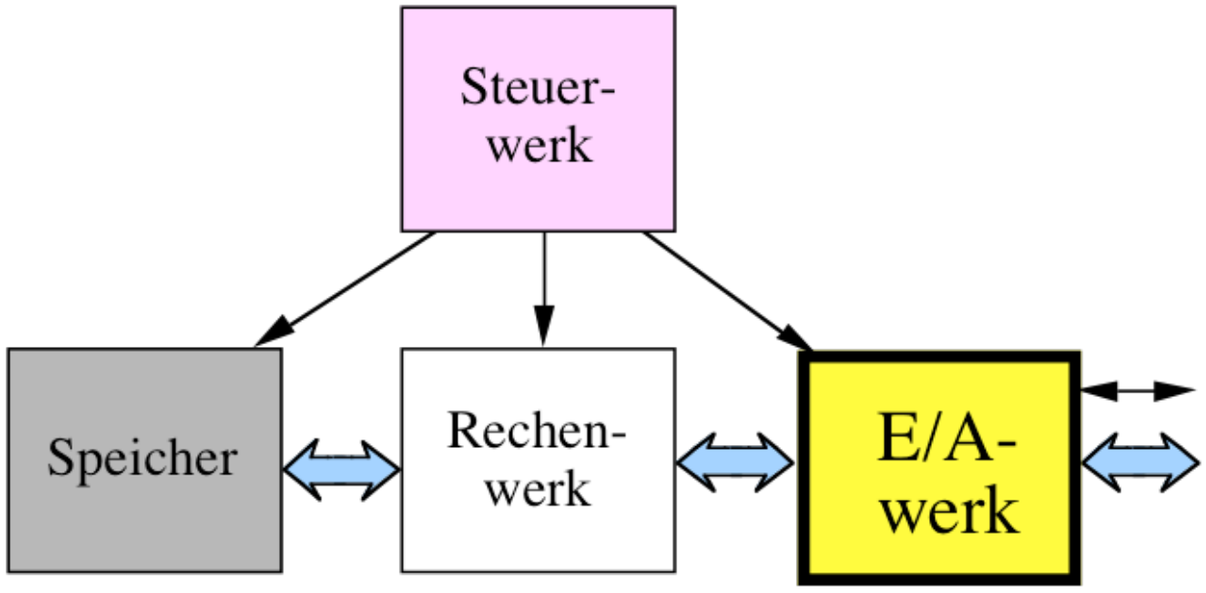
- Speicherhierarchie aus Kostengründen
- Register
  - 10-100 Wörter,< 1 Nanosekunde
  - kein separater Zyklus für die Adresse.
- Cache, Prozessorpufferspeicher
  - 8 KWörter bis 8 MBytes, < 10 Nanosekunden
  - häufig gebrauchte Daten und Codeteile
  - für Programmierer unsichtbar
- Hauptspeicher
  - 256 - 16384 Megabytes, ~ 10 - 50 Nanosekunden
  - evtl. inklusive Adressübersetzung
  - separater Zyklus für die Speicheradresse
- Hintergrundspeicher
  - Festplatte, Netz,
  - 40 - 3000 Gbytes, ~ Millisekunden
  - Zugriff über Betriebssystem
  - blockweise übertragen



Interactive 6.1 Sata-Festplatte 500 GB

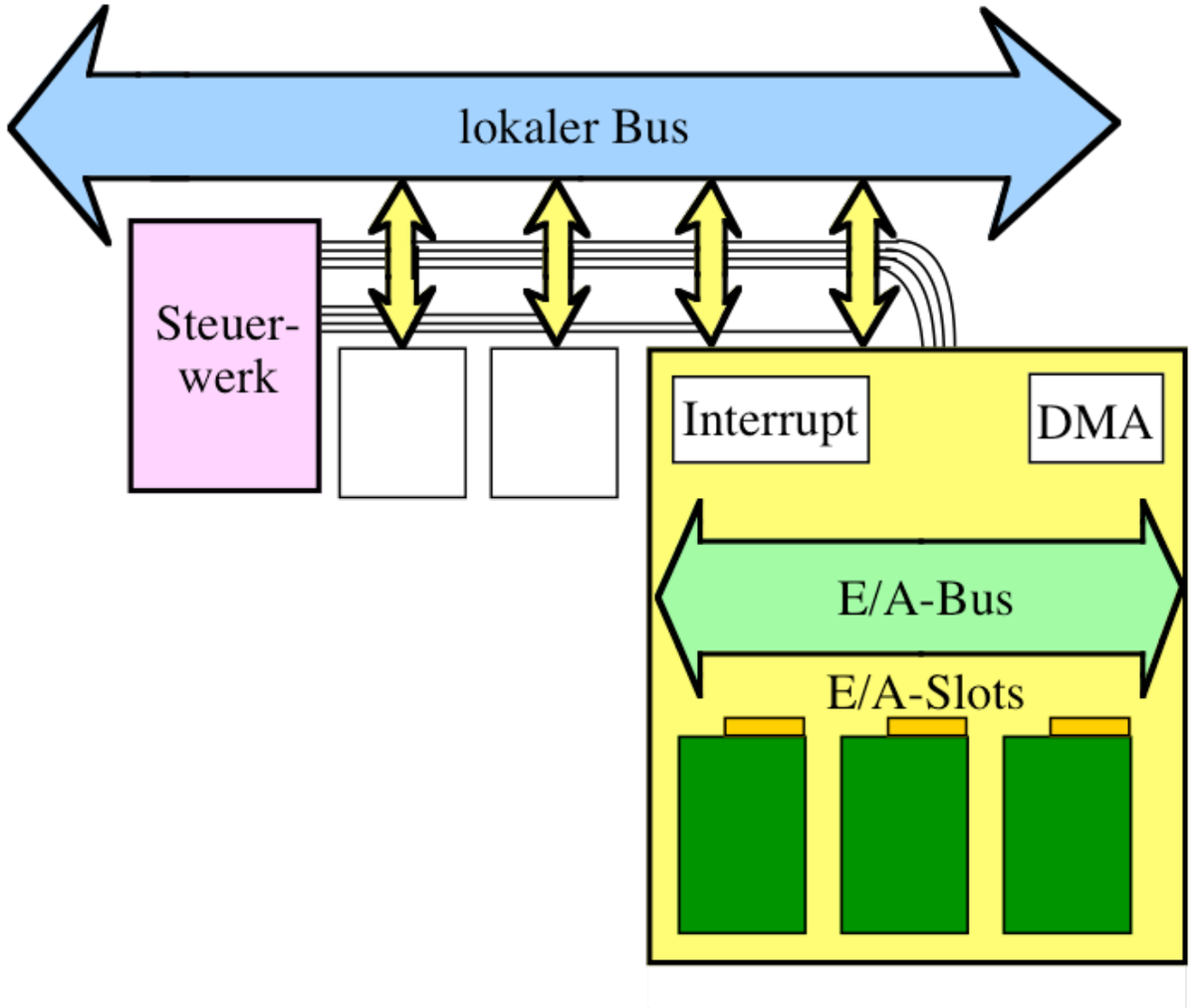


- Ein-/Ausgabewerk
  - kontrolliert durch Steuerwerk
  - Transport via Rechenwerk in den Speicher
  - Bedienungsleitungen zum Peripheriegerät
  - Keine Parallelarbeit von Rechenwerk & E/A
  - Missbrauch der Rechenregister
  - Wartezyklen der CPU auf Peripheriegeräte

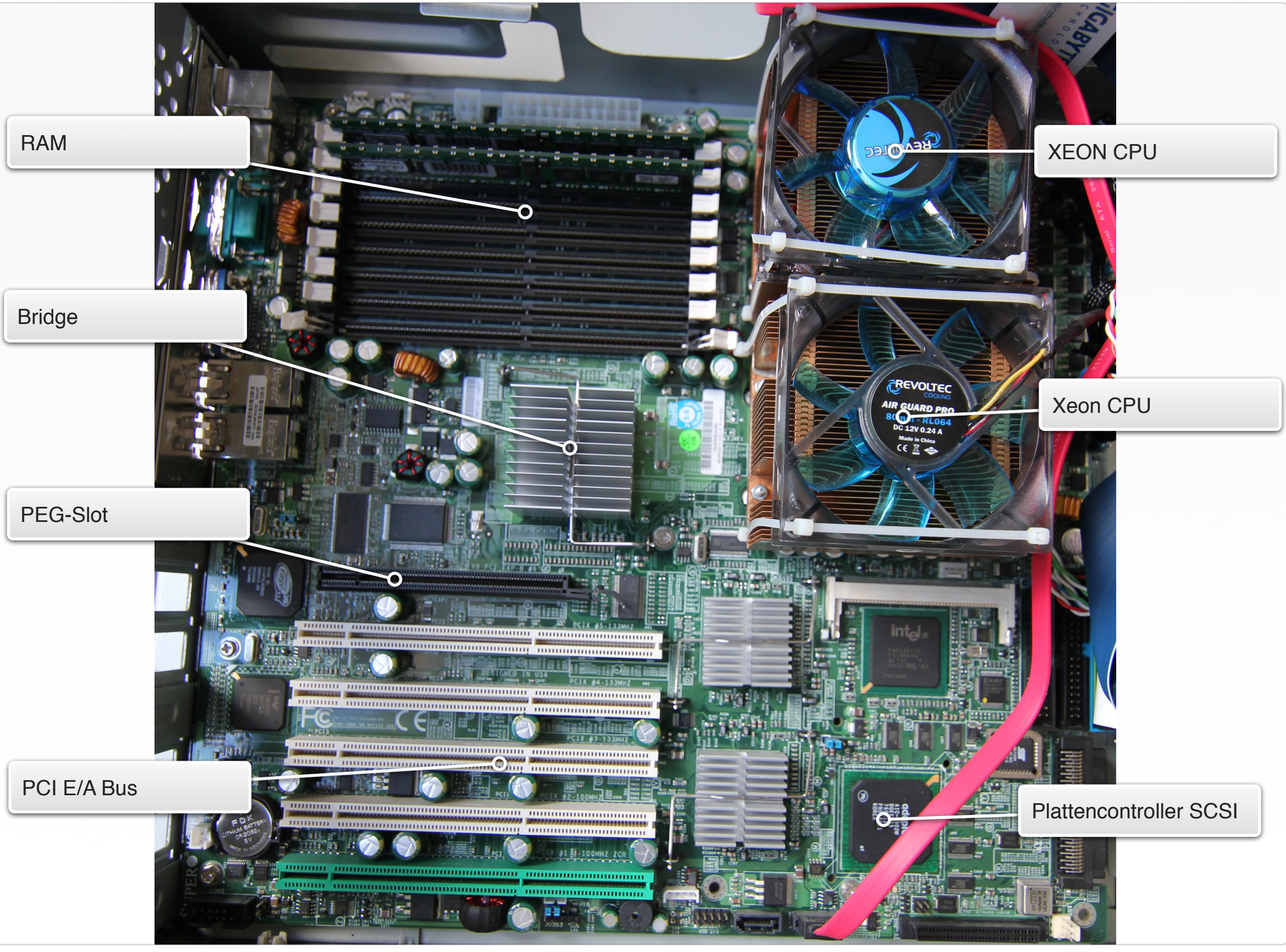


- Pfad in den Speicher als Flaschenhals
  - Resultate von Berechnungen
  - E/A-Übertragungen
  - Instruktionen

- Moderne Rechner
  - besonderer, standardisierter E/A-Bus
  - getrennt vom Prozessorbus
- autonomere E/A-Schnittstelle
  - externer Bus,
  - Abläufe parallel zum Rechenwerk
  - Interrupt-Funktion
  - DMA-Kanäle
    (direct memory access)
  - Display-Kontroller
  - Adapterplatinen
  - VLSI-Chips
- Grafik evtl. Extrabus
  - AGP
  - PCIe, PEG

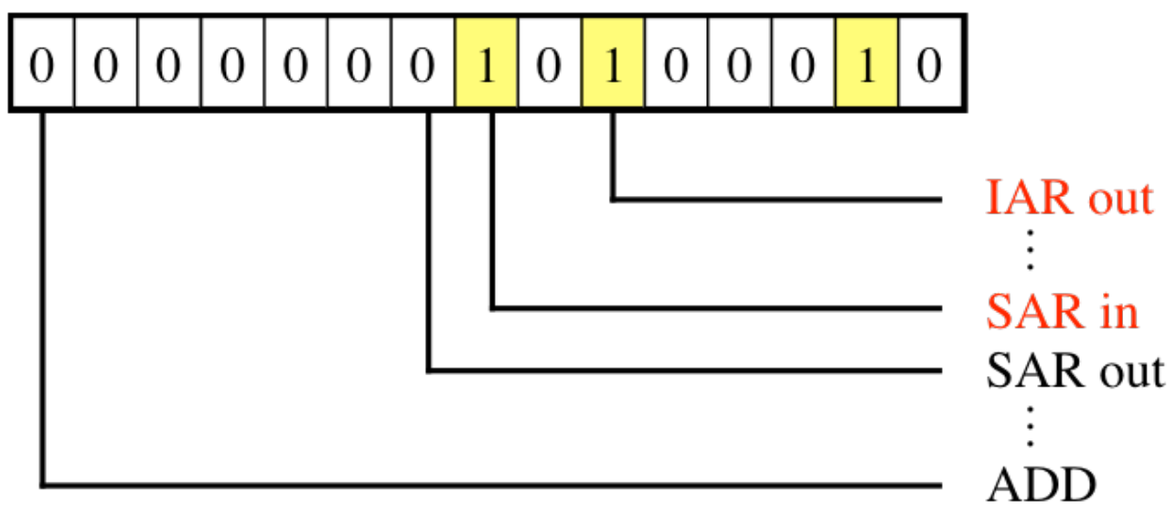


Interactive 6.2 Severboard Intel Xeon ca. 2009



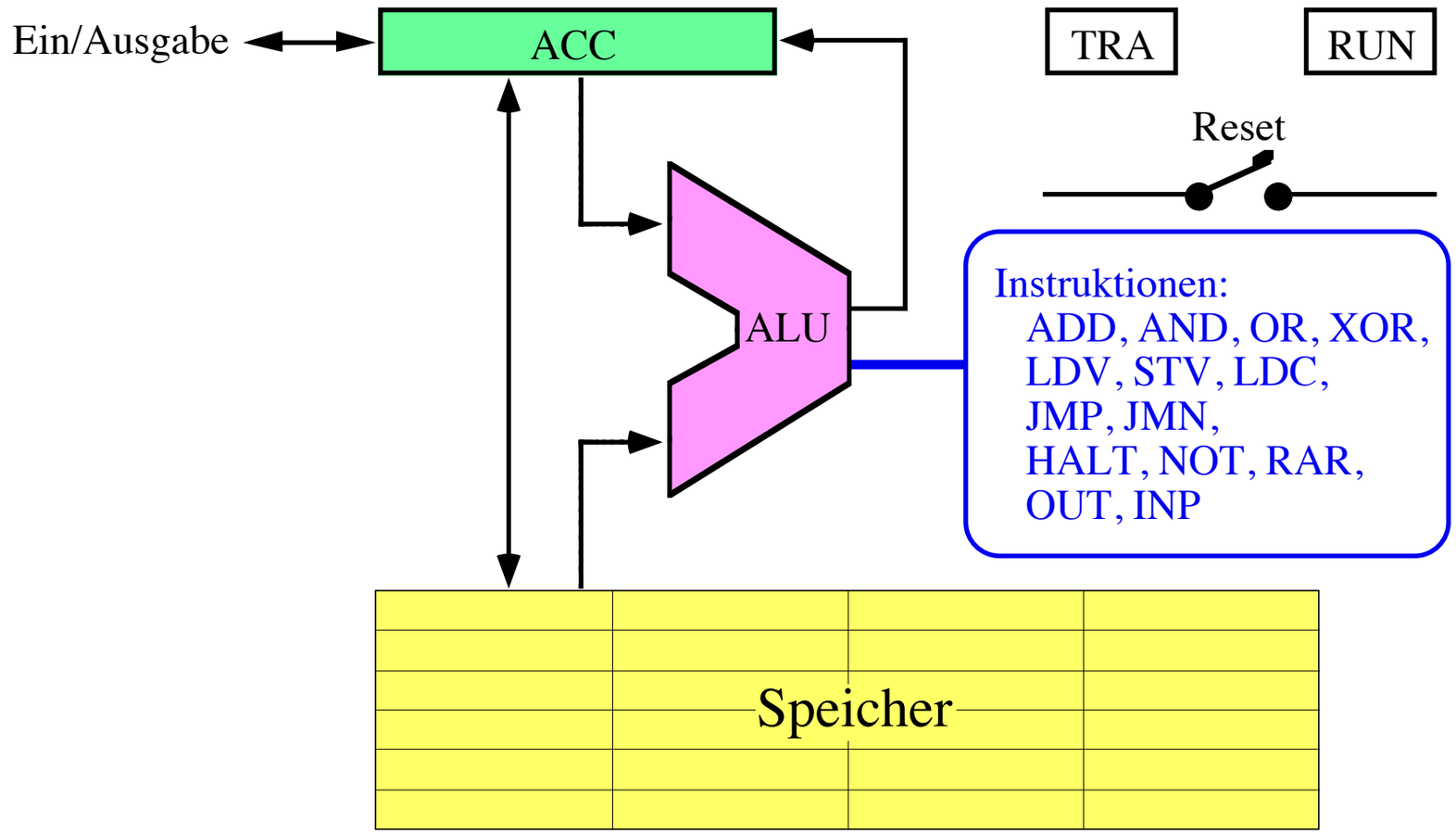
Maschinenbefehle

- Befehle liegen im Speicher
  - werden geholt wie Daten
  - kommen in Befehlsregister
  - Befehlszähler
- Befehl ⊗ Status => Sequenz von Steuerworten
  - Bits des Steuerwortes an Steuerleitungen anlegen
  - Speicher, E/A, ALU, … reagieren auf Steuerleitungen
  - in jedem Takt ein Steuerwort
  - Befehle können mehrere Steuerworte enthalten
  - Mikrobefehle



Minimalmaschine

- ein Register
- nur wenige, einfache Instruktionen
- TRA: Warten auf E/A
- 32-Bit Architektur



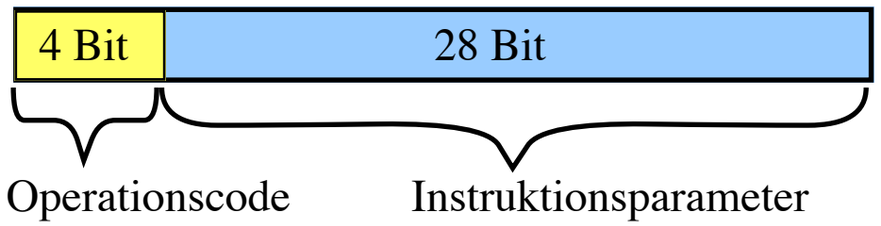
- Befehle zur Datenmanipulation

NOT		Komplementiere den Akkumulator, B-1 Komplement
RAR		Rotiere den Akkumulator um 1 Bit nach rechts
ADD	a	Addiere den Inhalt der Speicherzelle a zum Akkumulator
		ACC $\leftarrow$ ACC + Speicher[a]
AND	a	ACC $\leftarrow$ ACC $\wedge$ Speicher[a]
OR	a	ACC $\leftarrow$ ACC $\vee$ Speicher[a]
XOR	a	ACC $\leftarrow$ ACC $\otimes$ Speicher[a]
EQL	a	ACC<>Speicher[a]: ACC $\leftarrow$ 0
		ACC=Speicher[a]: ACC $\leftarrow$ 11..1
LDV	a	ACC $\leftarrow$ Speicher[a]
STV	a	Speicher[a] $\leftarrow$ ACC
LDC	c	ACC $\leftarrow$ c

- Kontrollfluß

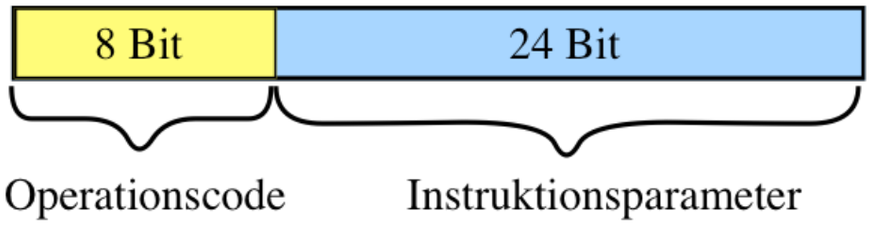
JMP	p	Nächste Instruktion in Speicher[p]
JMN	p	Sprung nach p falls ACC negativ
HALT		RUN $\leftarrow$ 0 Instruktionsausführung hält an
		Später Restart nur aus Speicher[0] möglich

- Basisformat



  - 0,1,2,3 ADD, AND, OR, XOR
  - 4, 5, 6 LDV, STV, LDC
  - 7, 8, 9 JMP, JMN, EQL
  - A .. E future use

- Erweitertes Format
  - mehr als 16 Instruktionen möglich,
  - 24 Bit Parameter & nicht immer benützt

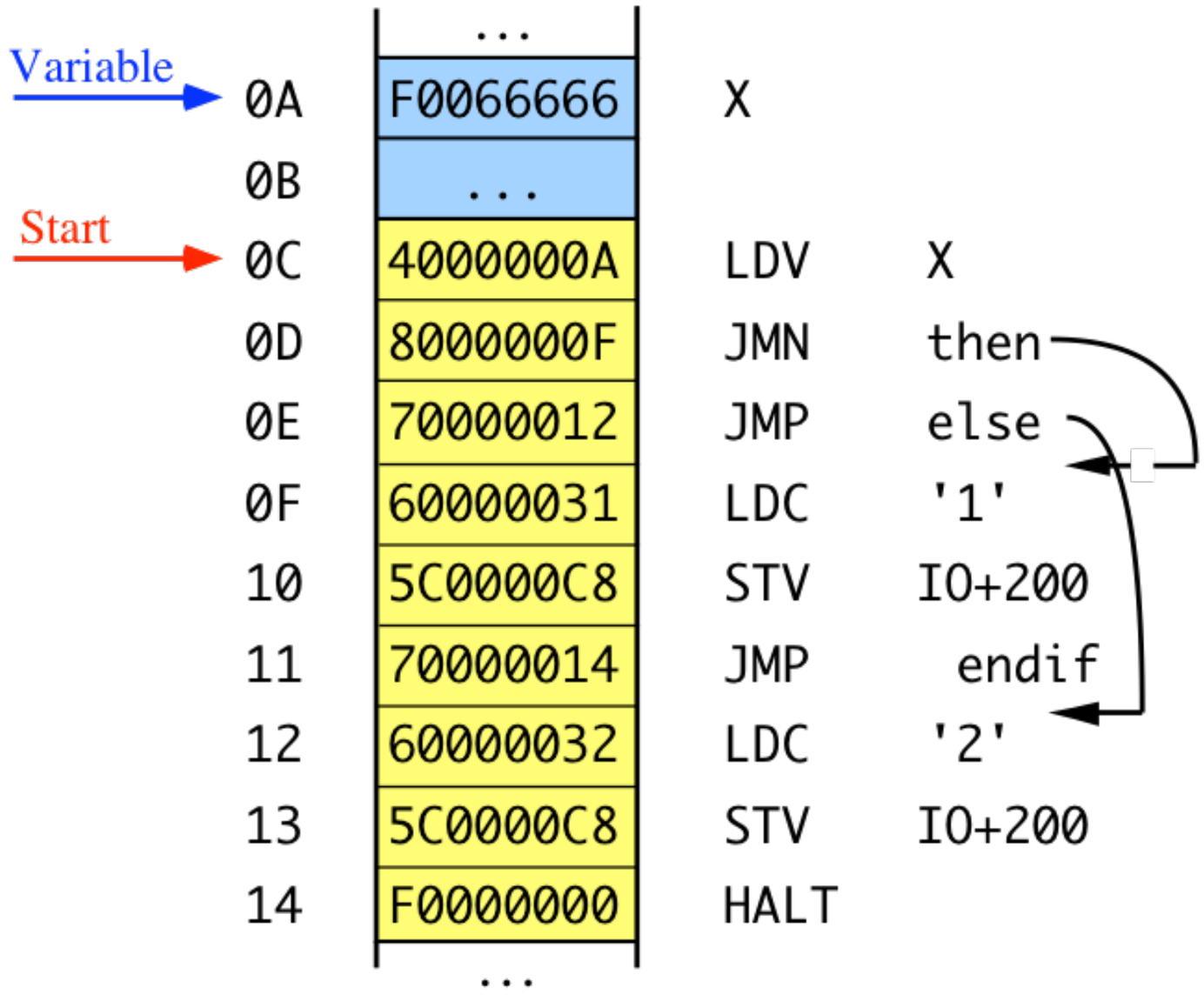


  - F0, F1, F2 HALT, NOT, RAR
  - F3, F4,… future use

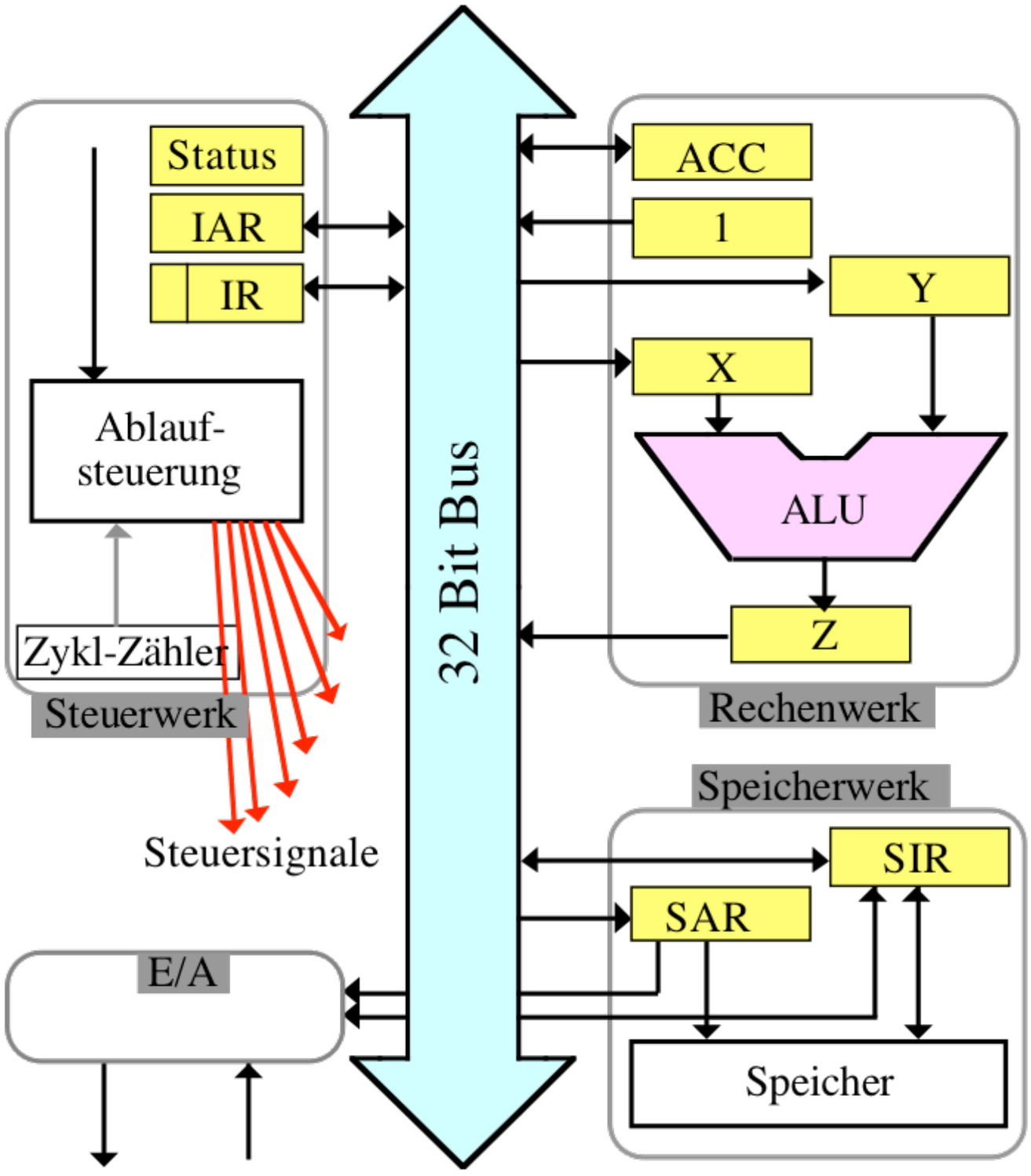
- Assemblerprogrammieren im Schnelldurchgang

```
if (X < 0)  printf("1")
            else printf("2");
```

- Hexadezimaler MiMa-Speicherauszug

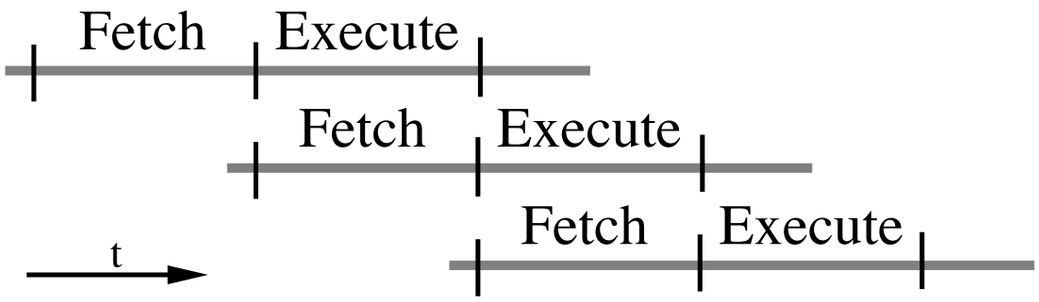


- Speicher
  - statisch
  - 24 Bit/Wort
- 1-Register
  - z.B. Inkrement
- Steuerwerkregister
  - Instruktionen
  - Instruktionsadresse (PC)
  - Status
- ALU
  - Eingang X,Y
  - Ergebnis Z
  - ADD, AND, XOR, OR
- Registertransfer
  - Quell-Register an Bus
  - Zielregister übernimmt
  - Steuerleitungen



Ablauf der Instruktionen

- Zweiteilung der Befehle
  - Fetch-Zyklus holt Befehl
  - Execute Zyklus führt Befehl aus
  - eventuell überlappend



- Unterzyklen im Steuerwerk
  - Mikrozyklus, "Minor Cycle", Taktphase
  - andere Steuerimpulse in jedem Unterzyklus
  - Registertransferebene
- Mima-Instruktionen
  - 12 Unterzyklen
  - 5 Unterzyklen Fetch
  - 7 Unterzyklen Execute

- Ablauf der Lade-Instruktion

  LDV a ; laden von Inhalt der Speicherzelle a in ACC

  ; ACC <= Speicher[a]

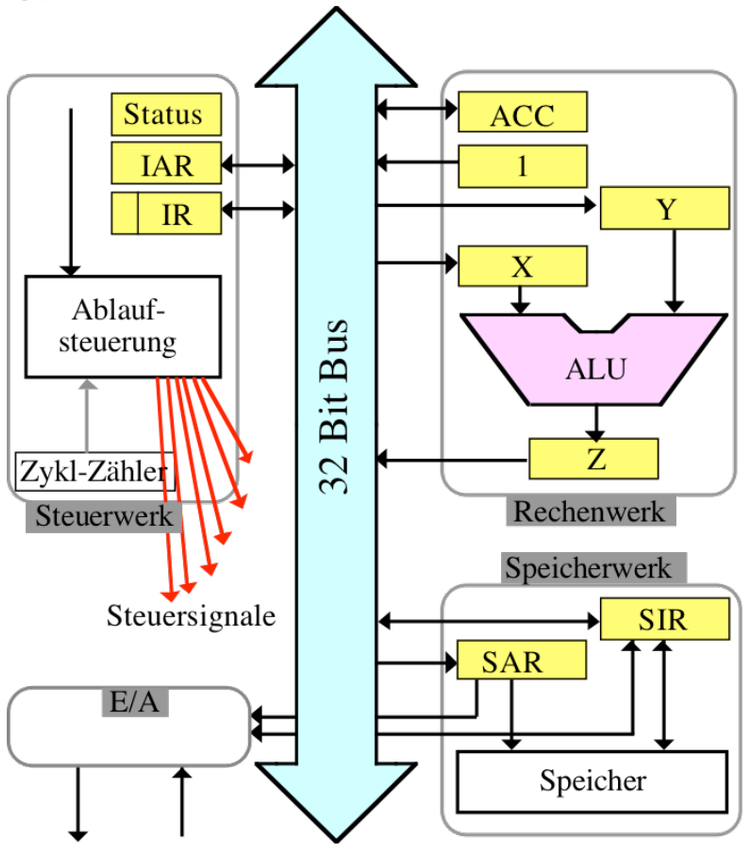
;Fetch-Zyklus

1. IAR -> (SAR, X),Leseimpuls an Speicher
2. 1 -> Y, ALU auf Addition schalten,
   Warten auf Speicher
3. Warten auf ALU, Warten auf Speicher
4. Z -> IAR, Warten auf Speicher
5. SIR -> IR

;Fetch-Ende, jetzt Execute-Zyklus

6. IR -> SAR, Leseimpuls an Speicher
7.,8.,9. Warten auf Speicher
10. SIR -> ACC
11.,12. leere Unterzyklen

;Execute-Ende, nächste Instruktion



- IAR -> (SAR, X)
  - IAR auf den Bus legen
  - SAR und X mit Steuerimpuls vom Bus lesen lassen

- Ablauf der ADD-Instruktion

  ADD a ; addieren von Inhalt der Speicherzelle a zum ACC

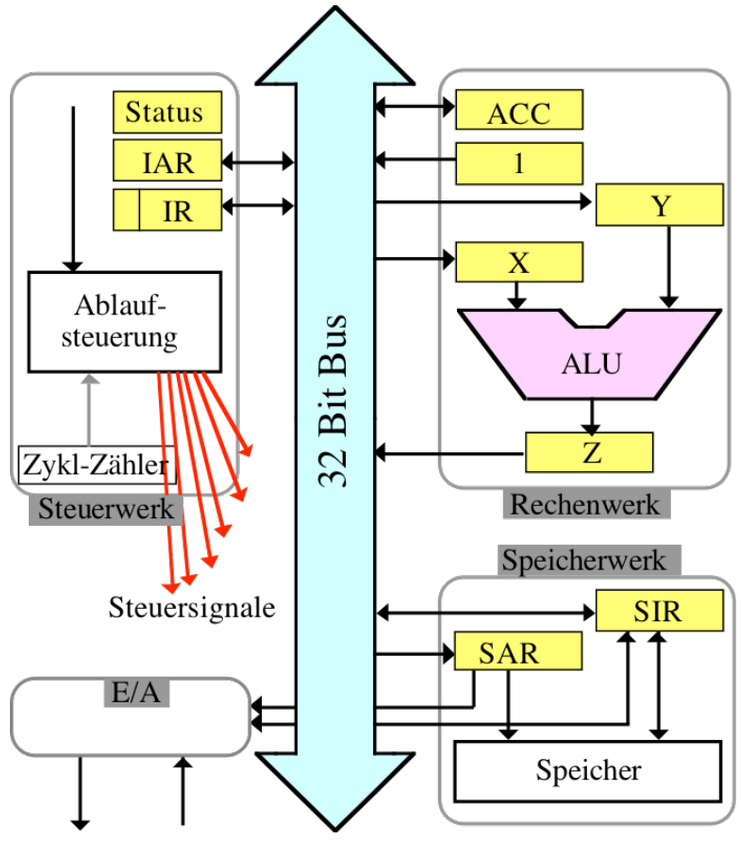
;Fetch-Zyklus

1. IAR -> (SAR, X),Leseimpuls an Speicher
2. 1 -> Y, ALU auf Addition schalten,
   Warten auf Speicher
3. Warten auf ALU, Warten auf Speicher
4. Z -> IAR, Warten auf Speicher
5. SIR -> IR

;Fetch-Ende, jetzt Execute-Zyklus

6. IR -> SAR, Leseimpuls an Speicher
7. ACC -> X, Warten auf Speicher
8., 9. Warten auf Speicher
10. SIR -> Y, ALU auf Addition schalten
11. warten auf ALU
12. Z -> ACC

;Execute-Ende, nächste Instruktion



- JMN p - Jump if Negative
  - Bedingter Sprung zur Instruktion im Speicherwort[p]
  - negativ bedeutet, das oberste Bit im Akkumulator ist 1
  - X, Y-Register und ALU werden ignoriert
  - kein Datenzugriff auf Speicher

;Fetch-Zyklus

1. IAR -> (SAR, X),Leseimpuls an Speicher
2. 1 -> Y, ALU auf Addition schalten,
   Warten auf Speicher
3. Warten auf ALU, Warten auf Speicher
4. Z -> IAR, Warten auf Speicher
5. SIR -> IR

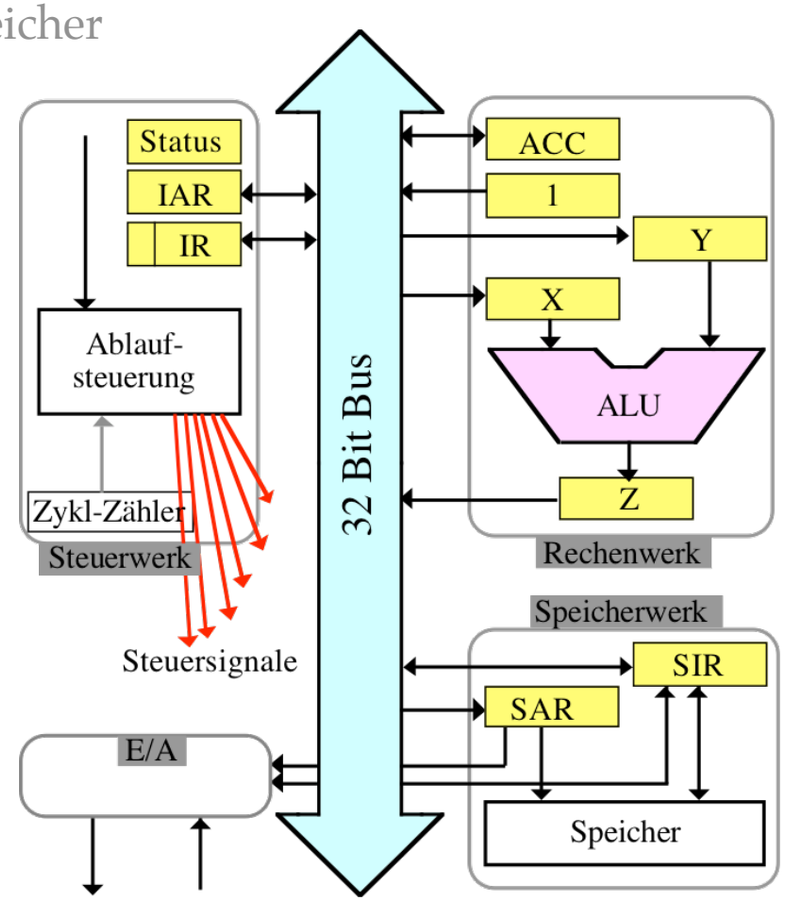
;Fetch-Ende, jetzt Execute-Zyklus

6a. falls vorderstes Bit in ACC = 1:
IR -> IAR

6b. falls vorderstes Bit in ACC = 0:
leerer Unterzyklus

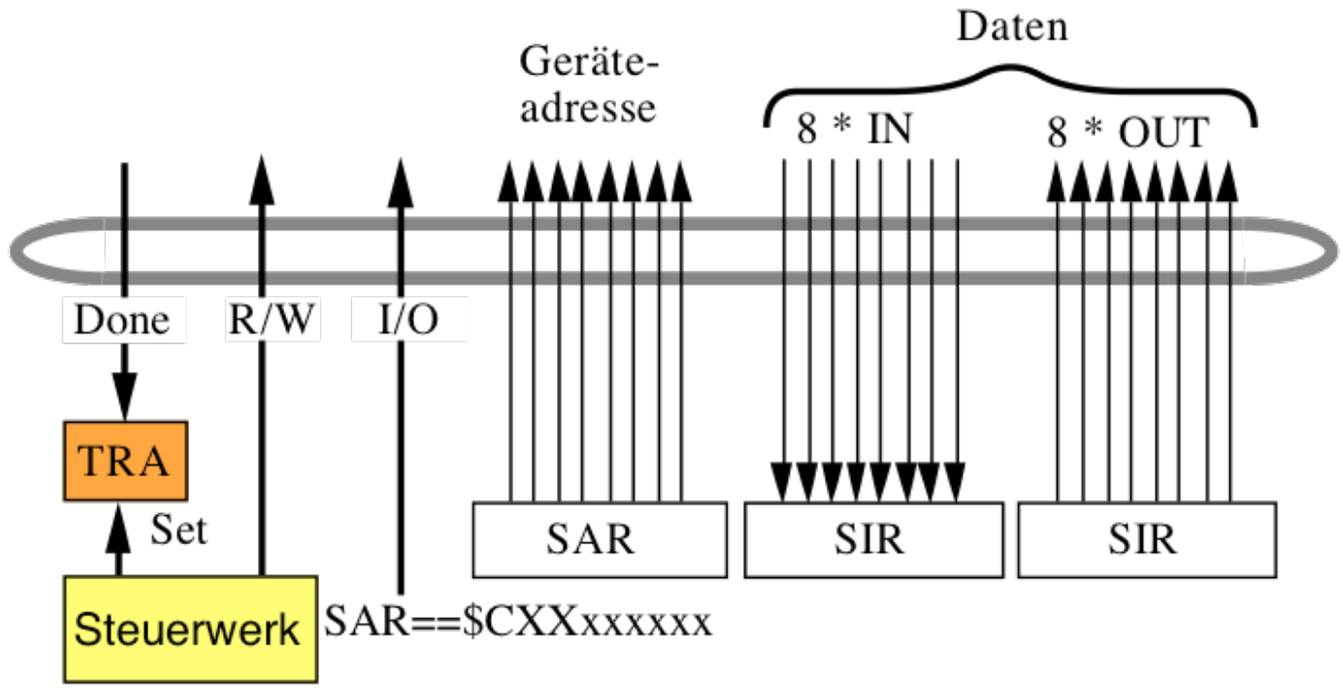
7. … 12. leere Unterzyklen

;Execute-Ende, nächste Instruktion

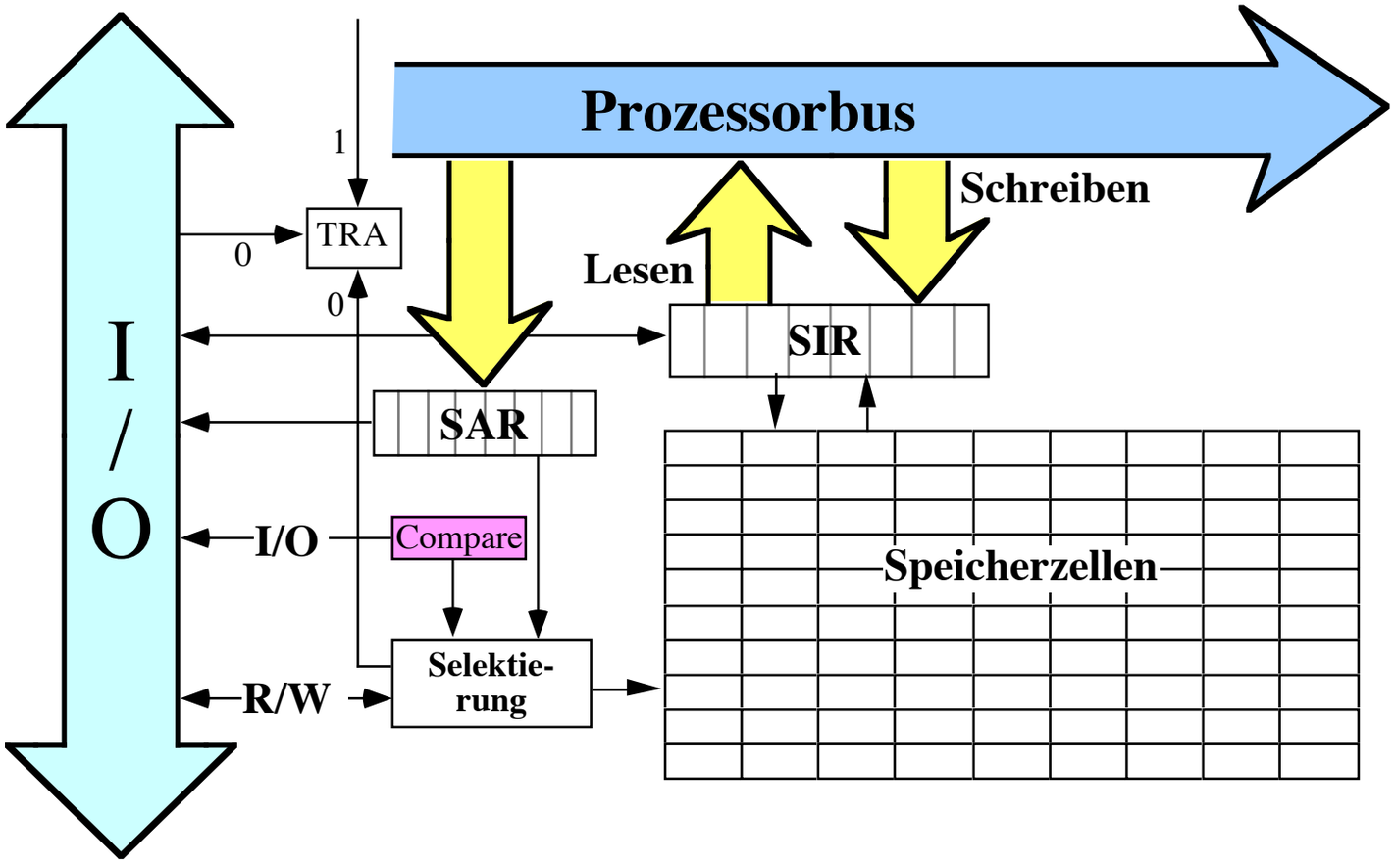


- Start der Mima
  - Drücken der Reset-Taste
    0 -> IAR
    0 -> TRA
    1 -> RUN
- Maschine beginnt ab Adresse 0 Instruktionen auszuführen
  - evtl. andere, festgelegte Adresse
- Urlader
  - einige Speicherzellen ab Adresse 0 sind Festwertspeicher
  - enthalten einen Urlader ("Boot-Strap Loader")
  - liest Programm von einem bestimmten Gerät in den Speicher
  - beginnt mit dessen Ausführung
- Schaltpult mit Schaltern und Lämpchen zum
  - Modifizieren und Inspizieren von Registern und Speicherinhalten
  - bei älteren Rechnern

- Integration der E/A in den normalen Adressbereich
  - memory mapped I/O
  - In - LDV, Out - STV
  - Geräte langsamer als Speicher
  - variable Geschwindigkeit
  - => Handshake
- Speicherwerk wertet Adresse aus
  - Adr == $CXXxxxxxx => E/A
  - Adr <> $CXXxxxxxx => Speicher
- TRA-Bit
  - vom Steuerwerk gesetzt
  - von Speichercontroller oder Peripherie zurückgesetzt
  - Steuerwerk wartet auf TRA=0
- Geräte beobachten die Geräteadresse im EA-Werk
- Als Bus ausgeführte E/A-Schnittstelle



- Speicher mit Memory-mapped I/O
  - TRA Bit wird vom Prozessor gesetzt
  - von der Selektionsschaltung zurückgesetzt
  - oder vom Gerätebus zurückgesetzt



SAR = Speicheradressregister, SIR = Speicherinhaltsregister

Erzeugung der Steuersignale

- Eingangssignale zum Steuerwerk:
  - Operationscode aus IR-Register (4/8 Bit)
  - Vorzeichenbit im Akkumulator
  - Mikrozyklus 1..12 aus externem Zähler
  - Status-Register = (RUN, TRA)

=> 8+1+4+2 = 15 Eingangsleitungen

- Ausgangssignale vom Steuerwerk
  - Rechteckige Steuer-Impulse
  - evtl. statische Steuerung (z.B. für ALU)
  - Takteingang für Flip-Flops
  - Out-Enable für Tristate-Ausgang am Bus

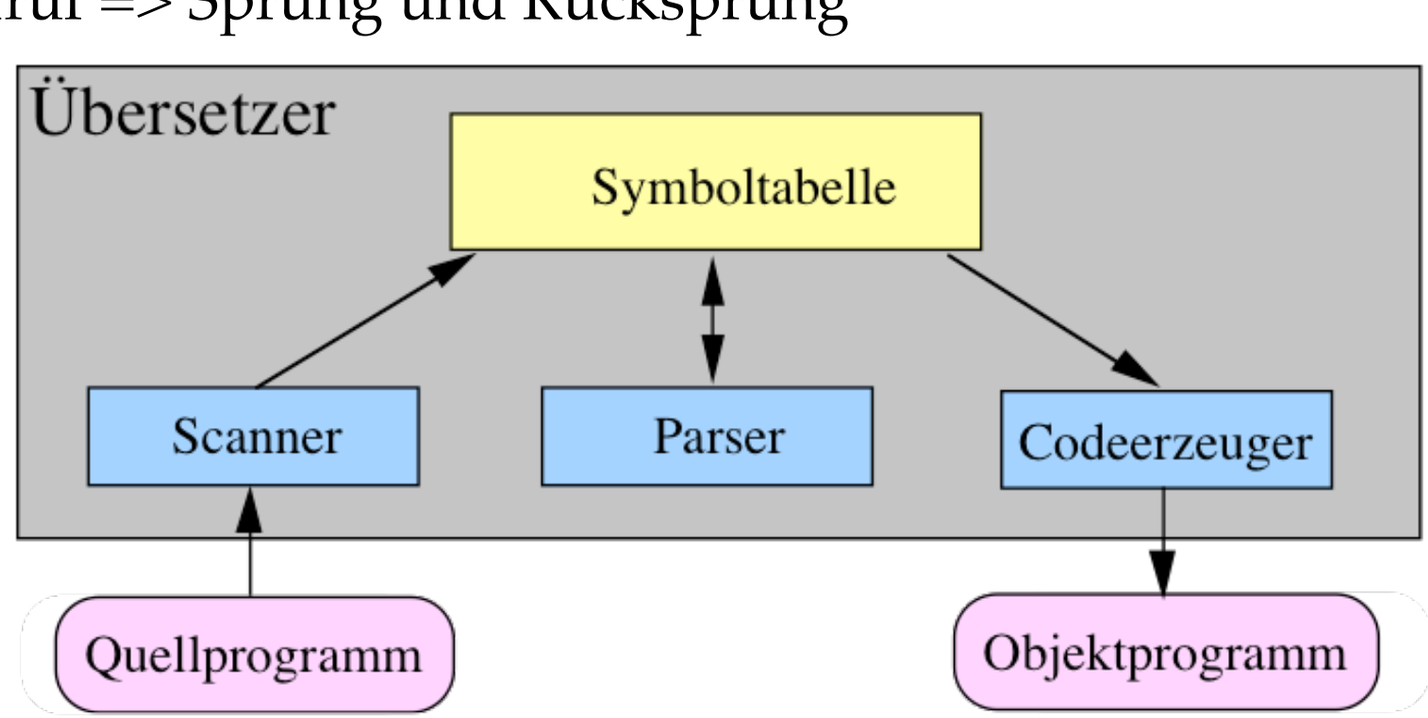
- RUN: 1 Leitung zum Clear-Eingang
- TRA: 1 Leitung zum Preset-Eingang
- ALU: 3 statische Pegel für die unterschiedlichen ALU-Funktionen
- Speicher: 2 Leitungen für Read & Write
- je 1 Input-Enable für Register
  - X, Y, ACC
  - IR, IAR, SIR, SAR
- zusätzlich Output-Enable für Register
  - ACC, 1, Z
  - IAR, SIR, IR (nur Bit [0..27])

=> 1+1+3+2+7+6 = 20 Steuersignale

- Steuerwerk kann realisiert werden:
  - als 20 Schaltfunktionen mit 15 Variablen
  - als ROM mit 2^{15} Wörtern à 20 Bit

Compiler

- Brücke Programmiersprache - Objektcode
  - C, Pascal, Modula, Fortran,
  - IA, 68000, PowerPC, 8051, Z80, DSPs, …
  - Name => Adresse
  - Statement => Instruktionen
  - Prozeduraufruf => Sprung und Rücksprung



- Einlesen des Programmes (Scanner)
  - findet Symbole
  - Identifier, Konstanten, …
- Syntaktische Analyse
  - zulässige Symbole werden verarbeitet ("Parsing")
  - für unzulässige Symbole Fehlermeldungen erzeugen
  - über "Look-Ahead" entschieden, welcher Pfad gewählt werden soll
  - bei schwierigen Programmiersprachen sehr weit vorausschauen
  - LL1 Programmiersprachen => maximal 1 Symbol Look-Ahead.
- Erzeugen der Maschinenbefehle (Codegenerierung)
  - syntaktische Prozeduren können auch die Instruktionen erzeugen
- Strategien
  - rekursive decent
  - bottom-up
  - top-down
  - Übersetzung für virtuelle Maschine besonders einfach
  - zeilenweise Übersetzung

- Beispiel: Ausdruck übersetzen

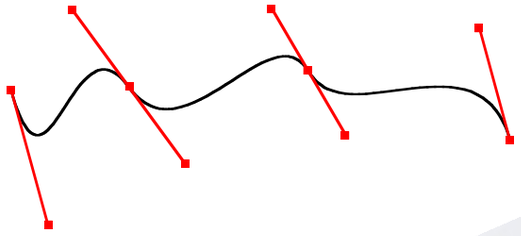
a=b-(c-((d*e)-g/h))

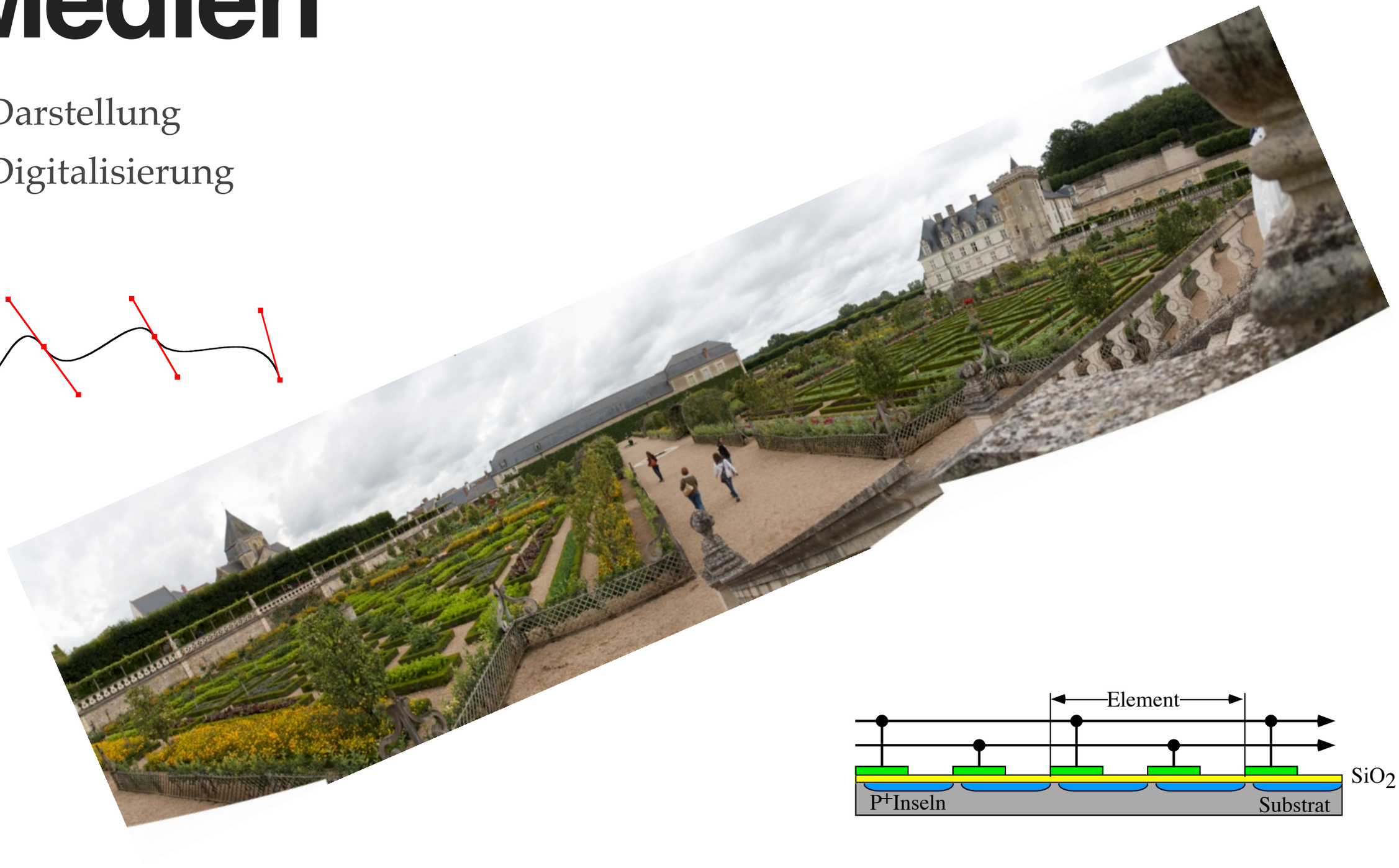
```
LDV     g
DIV     h       ; g/h
STV     hilf    ; optimiert wird nicht
LDV     d
MUL     e       ; (d*e)
SUB     hilf    ; -
STV     hilf
LDV     c
SUB     hilf
STV     hilf
LDV     b
SUB     hilf
STV     a       ;  a= ...
```

KAPITEL 7

Medien

- Darstellung
- Digitalisierung

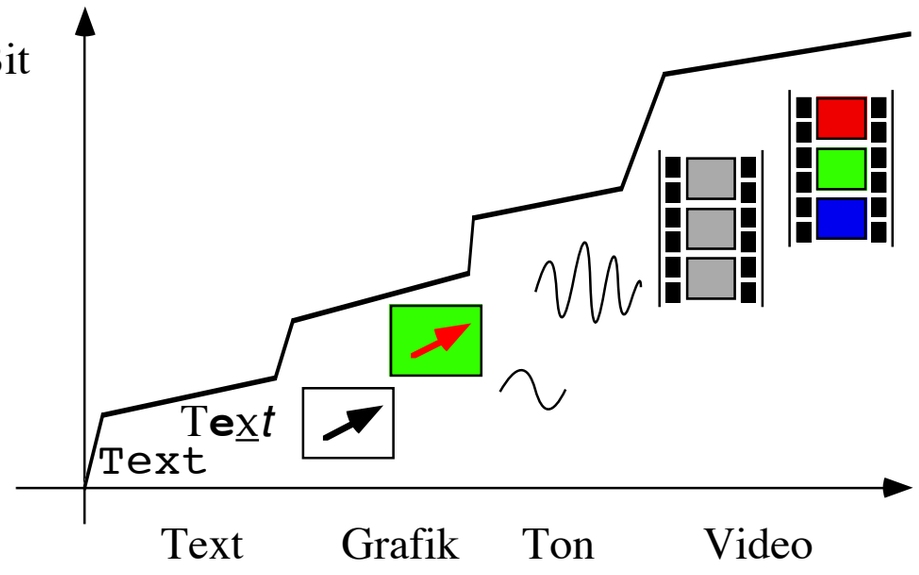




Medien und Wahrnehmung

- Nutzlast (Bit/bit)
  - Information wird in Bit gemessen, bit = Anzahl {0,1}

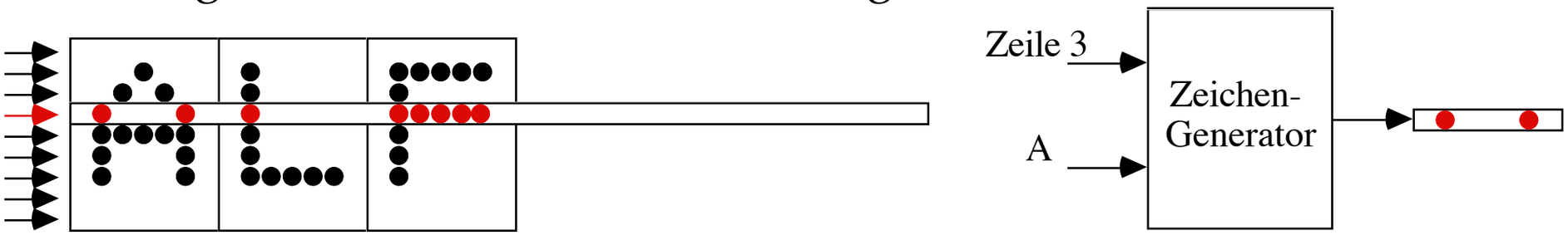
ASCII-Text	10 byte
Bitmap	1000 Punkte * 1 byte
Telefon	8.000 byte
Audio-CD	44.100 Samples * 2 byte * 2
Video	25 Bilder * 256 Spalten * 192 Zeilen * 3 byte/Punkt = 3.686.400 byte
TV	25 Bilder * 704 Spalten * 625 Zeilen * 3 byte/Punkt = 33.000.000 byte



- Aufnahmevermögen und Bitrate
- Dimensionen, räumliche Effekte
  - Menschen haben räumliches Empfinden (Sehen, Hören, Gleichgew.)
  - Raum und Zeit
  - Dimensionen werden vielfältig ausgewertet
- Diskrete und kontinuierliche Medien
  - Klassifikation entsprechend Auflösungsvermögen der Wahrnehmung
  - Im Raum Punkte oder Verläufe: Pixelmaps oder Photographien
  - In der Zeit Stilleben oder Bewegung
    Grafik oder Animation
    Bilder oder Video
  - Audio
    physikalisch immer kontinuierlich
    Psychisch auch diskret: Spracherkennung
    Sprache oder Musik
- Abschattungseffekte
  - in einem Medium
  - zwischen Medien

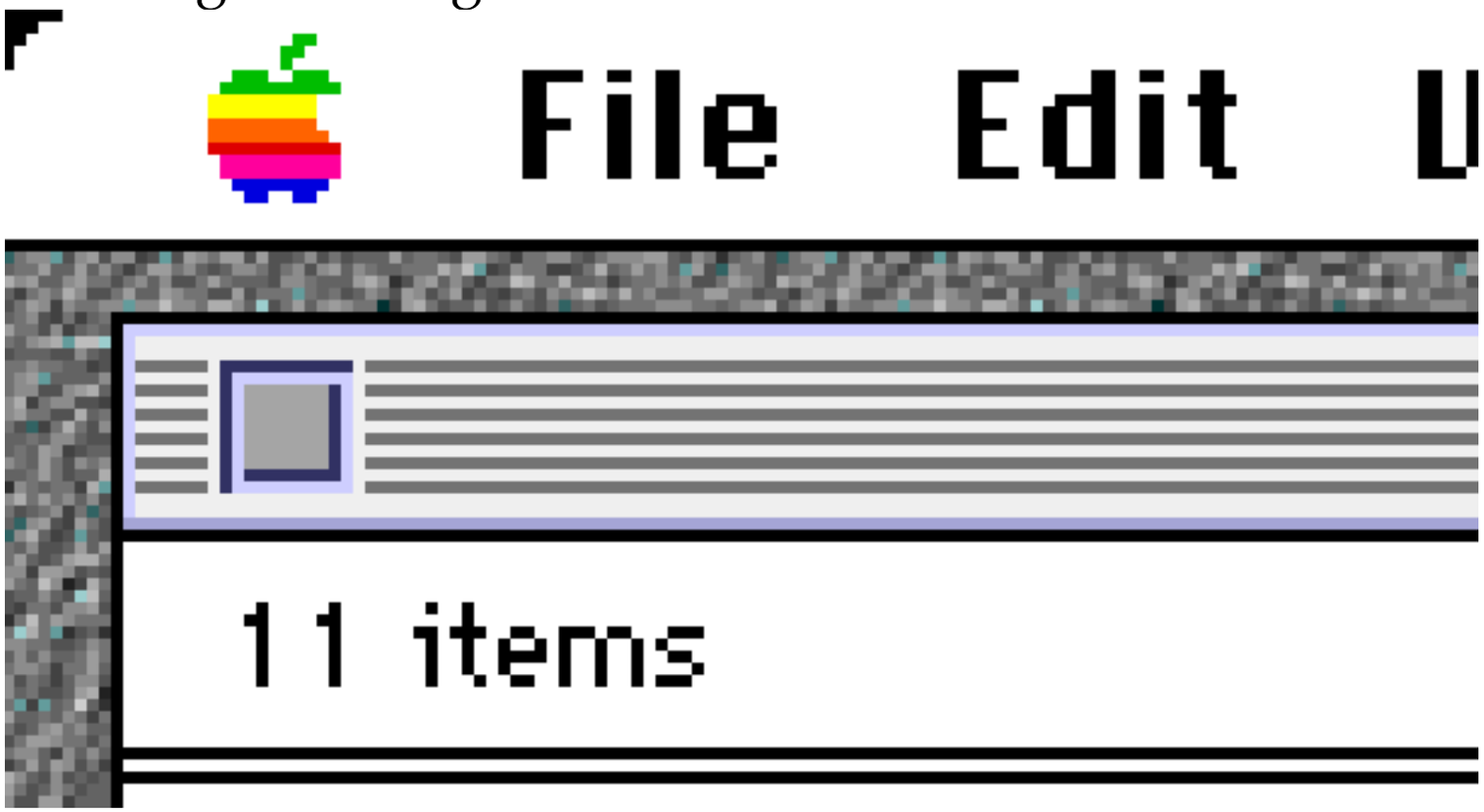
Computergrafik

- Darstellung visueller Objekte
  - Buchstaben und Zahlen,
  - geometrische Objekte (Gerade, Kreis, Rechteck, …)
  - Attribute (Farbe, Muster, Font, …).
- Bildspeicher
  - Hauptspeicher oder im Adapter,
  - eventuell mehrere Ebenen (Farbe, Graustufen, räumliche Position).
- Buchstabenbildschirme
  - nur Buchstaben darstellbar
  - oft als Rasterbildschirm, aber Punkte nur in Gruppen ansprechbar
  - Zeichengenerator: ROM zur Abbildung der Buchstaben auf Raste



- Vektorgrafik …

- Rasterbildschirm
  - jeder Punkt einzeln ansprechbar
  - uneingeschränkt grafikfähig

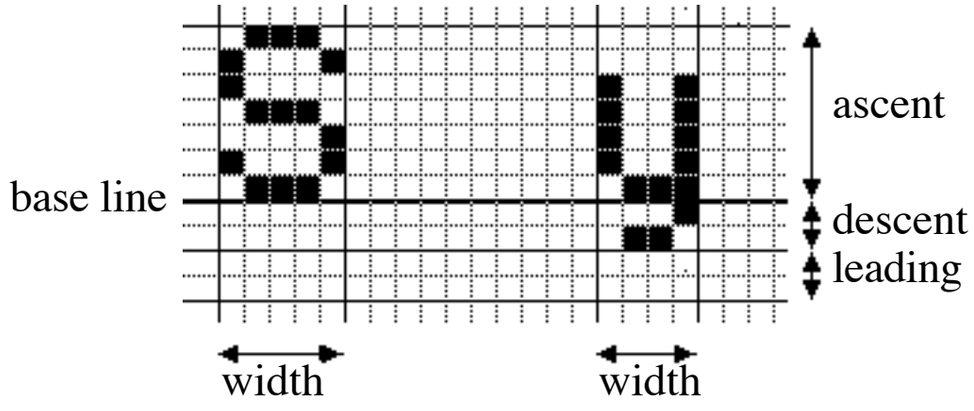


- Punkteanzahl typisch 1024*768 bis 2560*1600
- 72, 80 bis 240 Punkte/Zoll (dpi)
- 2560*1600*24 bit für 30" Farbmonitor -> 12.188.000 Byte
- Bildänderungsrate (Framerate, >25 Hz) und Bildwiederholrate (>70Hz)

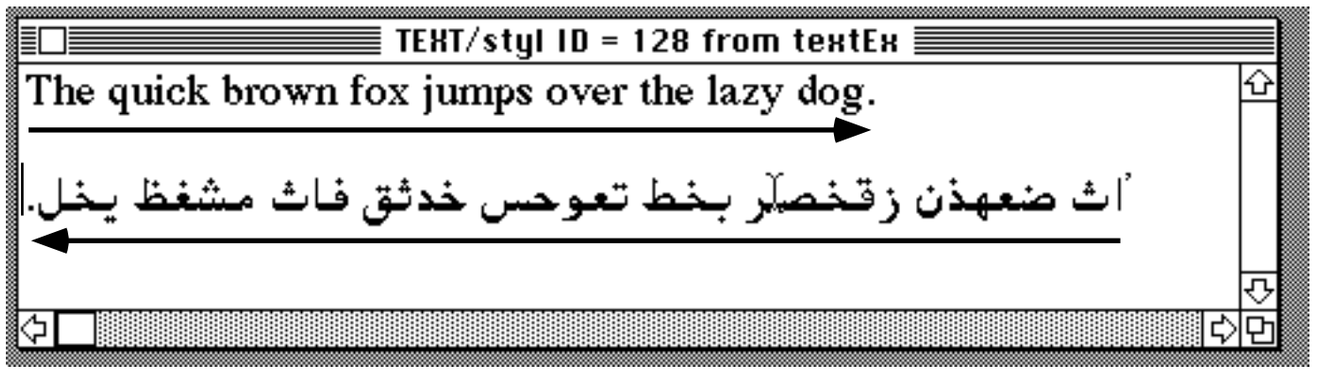
Text

- Zeichensatz
  - ASCII: American Standard Code for Information Interchange
    0 .. 31 Druckersteuerzeichen
    32 .. 127 druckbare Zeichen
    128 .. 255 nichtstandardisierte Erweiterungen
  - EBCDIC: Extended Binary Coded Decimal Interchange Code
  - ISO 8859-X
    Erweiterung von ASCII um länderspezifische Zeichen
    1, 2, 3, 4 und 9 für lateinische Zeichensätze
    5 kyrillisch, 6 arabisch, 7 griechisch und 8 hebräisch
  - Unicode
    Codes für alle Schriftzeichen der Welt
    16 Bit/Zeichen
    28.000 Codes für Ideographen (China, Korea, Japan)
    mehr Zeichen -> mehr Information/Zeichen: ae -> ä, ss -> ß

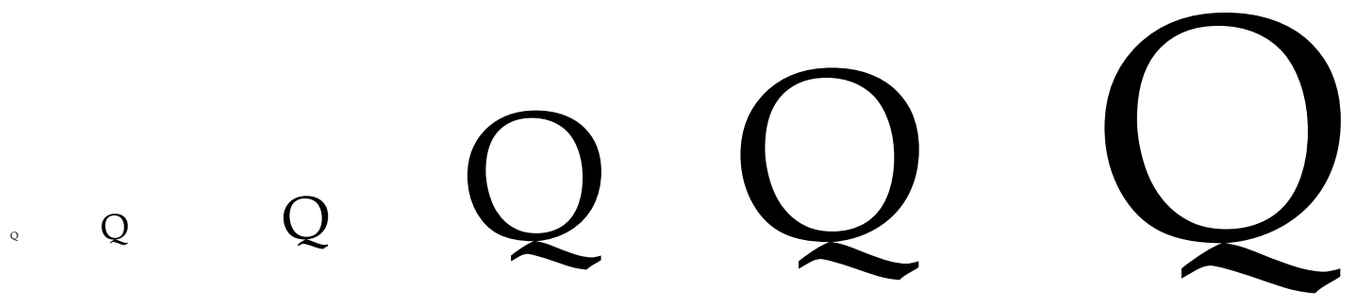
- kompaktes Medium
- Schriftattribute
  - **fett**, *kursiv*, Umriss, schattiert, …
  - Zeichengröße und -breite
  - Kerning und Ligaturen: fl statt fl
- Fontmetrik
  - beschreibt Laufeigenschaften des Textes
  - monospace vs. proportional



- Fontomania
  - tausende verschiedene Zeichensätze
  - Font-Beschreibungsalgorithmen siehe Kapitel 3
  - Helvetica
  - Palatino
  - Zapfino
- nicht-lateinische Schriften
  - andere Fonts
  - Hebräisch, Arabisch, Chinesisch, …
  - Schreibrichtung rechts -> links, vertikal

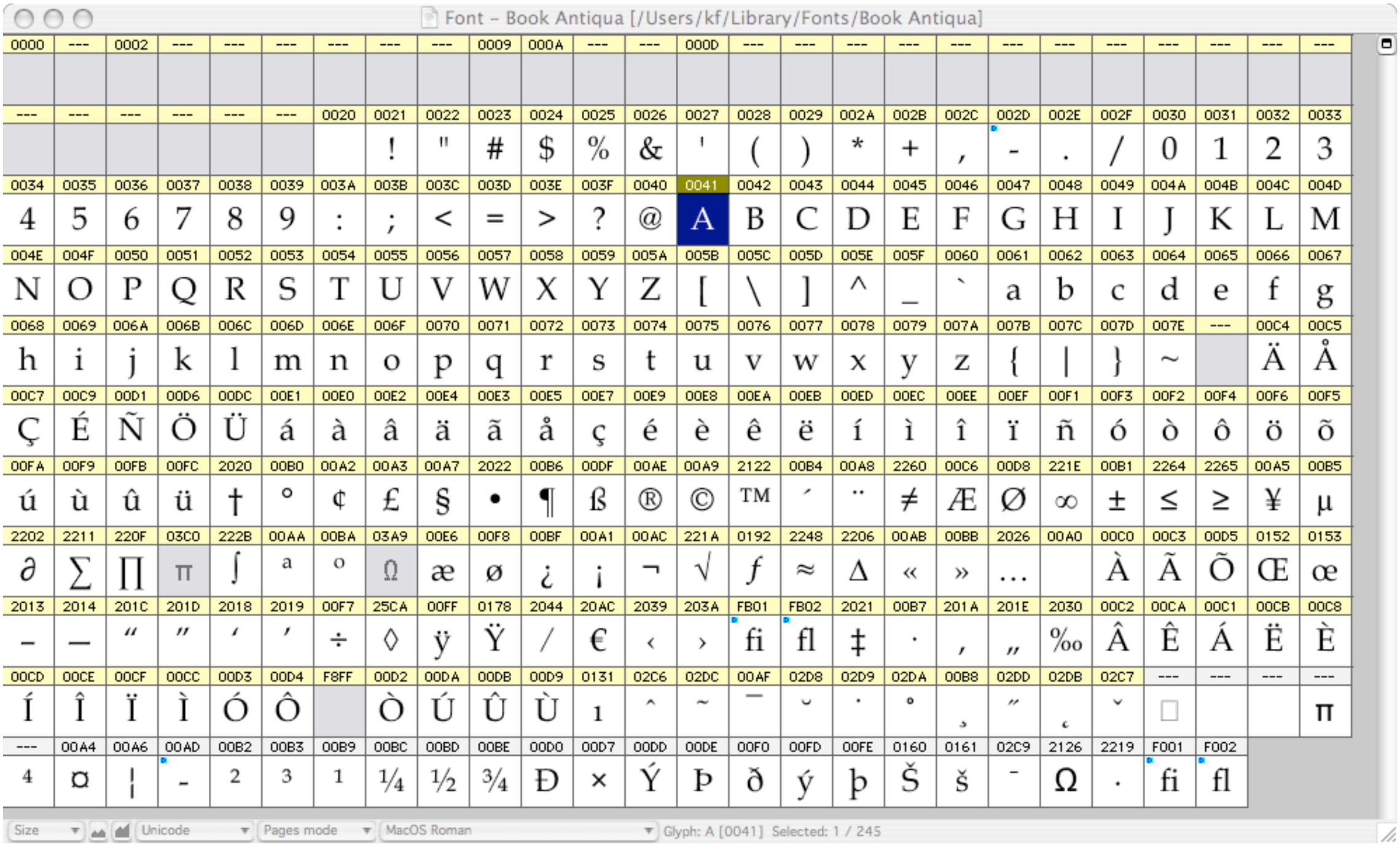


- Zeichendarstellung
- Bitmap-Fonts
  - werden entworfen, gezeichnet, gespeichert und fertig verteilt …
  - in verschiedenen Größen (z.B. 6 Punkte bis 127 Punkte)

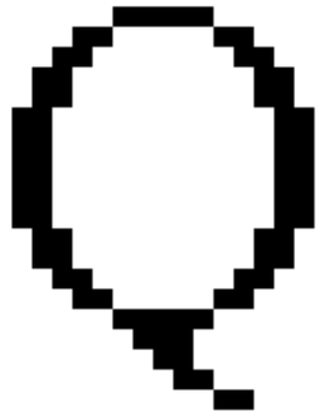


- und Formen

- Zeichensätze als Raster und Outline im System



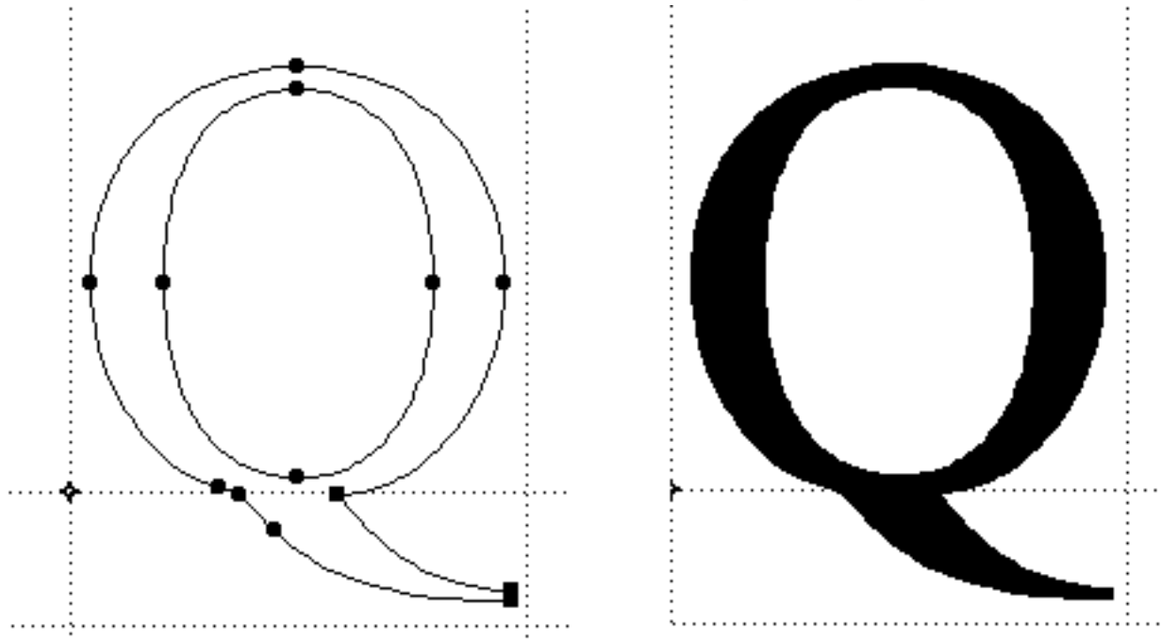
- Werden bei Bedarf in den Speicher geladen.
- Größe 24 Punkt (Vergrößerung *8)



Q

- Auflösungsabhängig, schlecht skalierbar
- Bitmap-Fonts werden bei zunehmender Zeichengröße Speicherfresser
- Bold, Italic, … müssen separat gespeichert werden

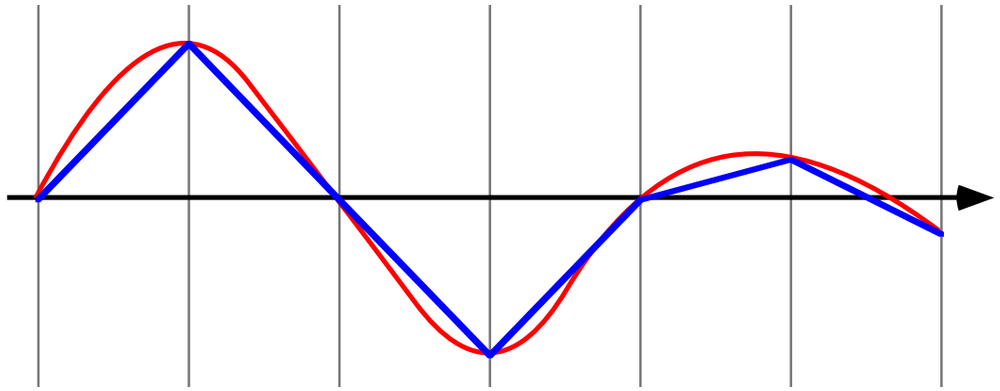
- Kurven zur Beschreibung von Fonts
- Die Umrisse der Zeichen werden als Kurvenzug angegeben



- Zur Darstellung wird dieser Kurvenzug ausgefüllt
  - unabhängig vom Koordinatensystem
  - affine Invarianz
  - möglichst einfach berechenbar

-> Stützpunkte und Interpolation

- Ähnlich Interpolation und Approximation mit Splines
  - stückweise linear: $f_i(x) = a_i\, x + b_i$
  - an den Stützpunkten stetig: $f_i(x) = f_{i+1}(x)$



  - stückweise kubisch: $f_i(x) = a_i\, x^3 + b_i\, x^2 + c_i\, x + d_i$

An den Stützpunkten:

a) stetig: $f_i(x_k) = s_k$, $f_i(x_{k+1}) = s_{k+1}$ => 2n Gleichungen

b) 'glatt': $f'_i(x) = f'_{i+1}(x)$ => 2(n-1) Gleichungen

=> Gleichungssystem 4n Unbekannte, 2n + 2n - 2 Gleichungen

je nach Randbedingungen versch. Approximationseigenschaften

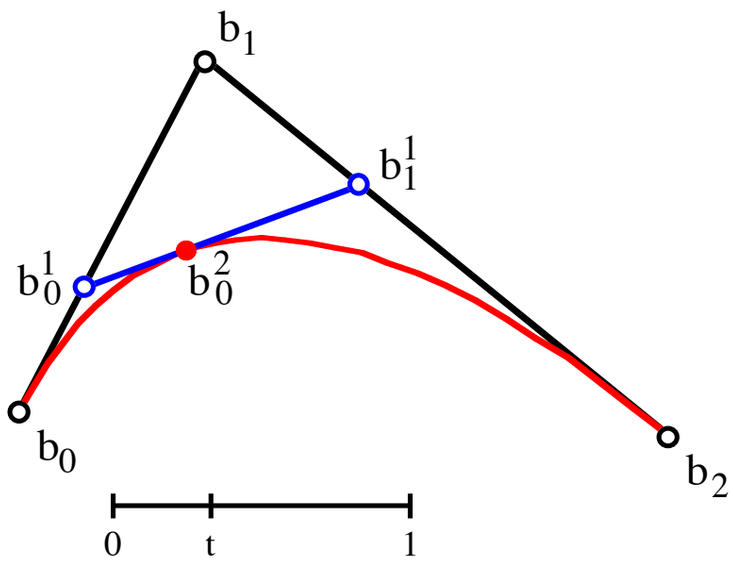
- Bézier-Kurven
  - Beispiel 2. Ordnung

    gegeben b_0, b_1, b_2

    $b_{1,0}(t) = (1 - t)\, b_0 + t\, b_1$

    $b_{1,1}(t) = (1 - t)\, b_1 + t\, b_2$

    $b_{2,0}(t) = (1 - t)\, b_{1,0}(t) + t\, b_{1,1}(t)$



- Algorithmus von de Casteljau

  gegeben b_0, b_1, …, b_n

$$b_i^r(t) = (1-t)\, b_i^{r-1}(t) + t\, b_{i+1}^{r-1}(t) \qquad r = 1, \ldots, n;\ i = 0, \ldots, n\text{-}r$$

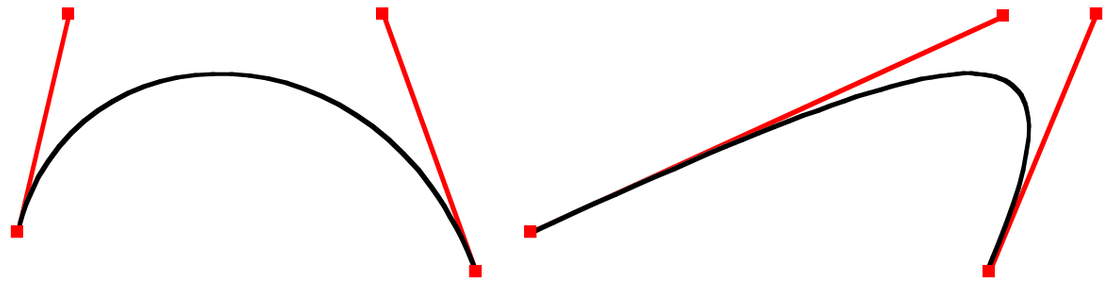
$$B_i^n = \binom{n}{i} t^i (1-t)^{n-i} \qquad b_0^n = \sum_{j=0}^{n} b_j\, B_j^n(t)$$

  - Explizite Darstellung mit Bernsteinpolynomen
  - Bézier-Kurven 3. Ordnung

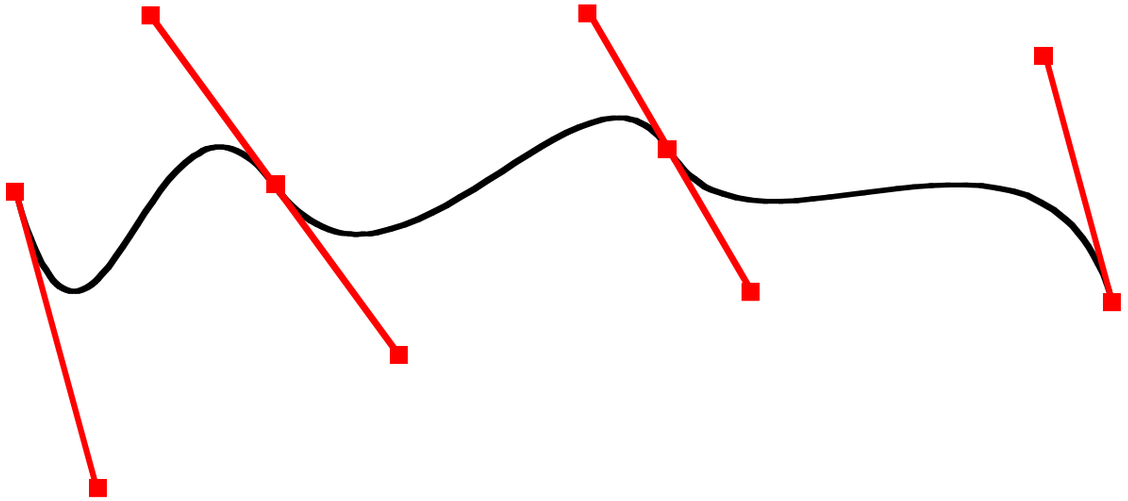
    Kontrollpolygon durch vier Punkte:

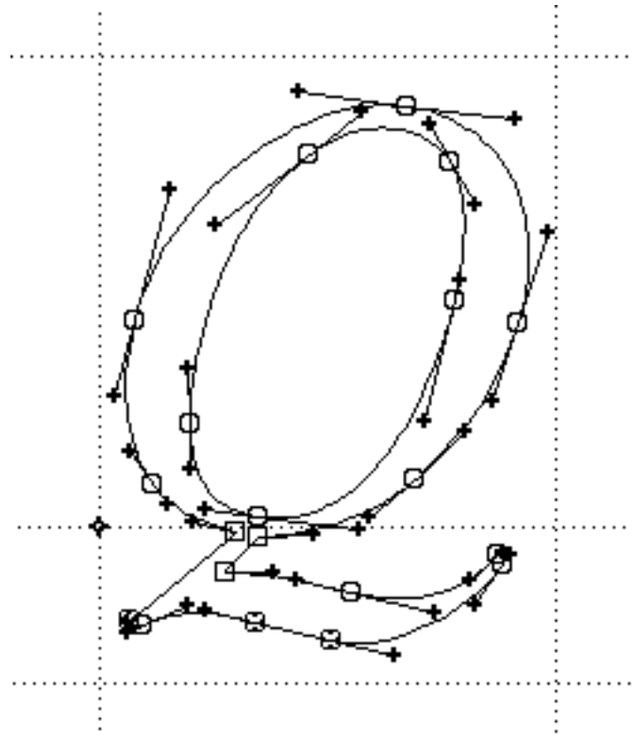
    Anfangspunkt (b_0) und Endpunkt (b_3)

    2 Kontrollpunkte (b_1, b_2)



- zusammengesetzte Kurve
  - mehrere Bézier-Splines zur Darstellung einer Kurve
  - Interpolationseigenschaft
  - Kontrollpunkte so legen, daß die Kurve glatt wird





- PostScript Type-1 Fonts
  - Fontparameter
  - Zeichenparameter
  - Bézier Kurven zur Beschreibung des Umrisses
  - 'Hints' zur Detailverbesserung
- TrueType oder andere Outline-Fonts benutzen ähnliche Kurven

3-D Grafik

- Modellieren
  - Topologie und Geometrie
  - geometrische Objekte erzeugen und anordnen
  - Attribute festlegen
  (Glanz, Farbe, Durchsichtigkeit)
  - Texturen bestimmen
  - Lichtquellen anordnen
- Rendering
  - Kameratyp und -position
  - Renderer wählen
  - Abbild berechnen
- Interagieren
  - Zeigemittel (Spacemouse, Handschuh, …)
  - Auswählen (picking)
  - Navigieren

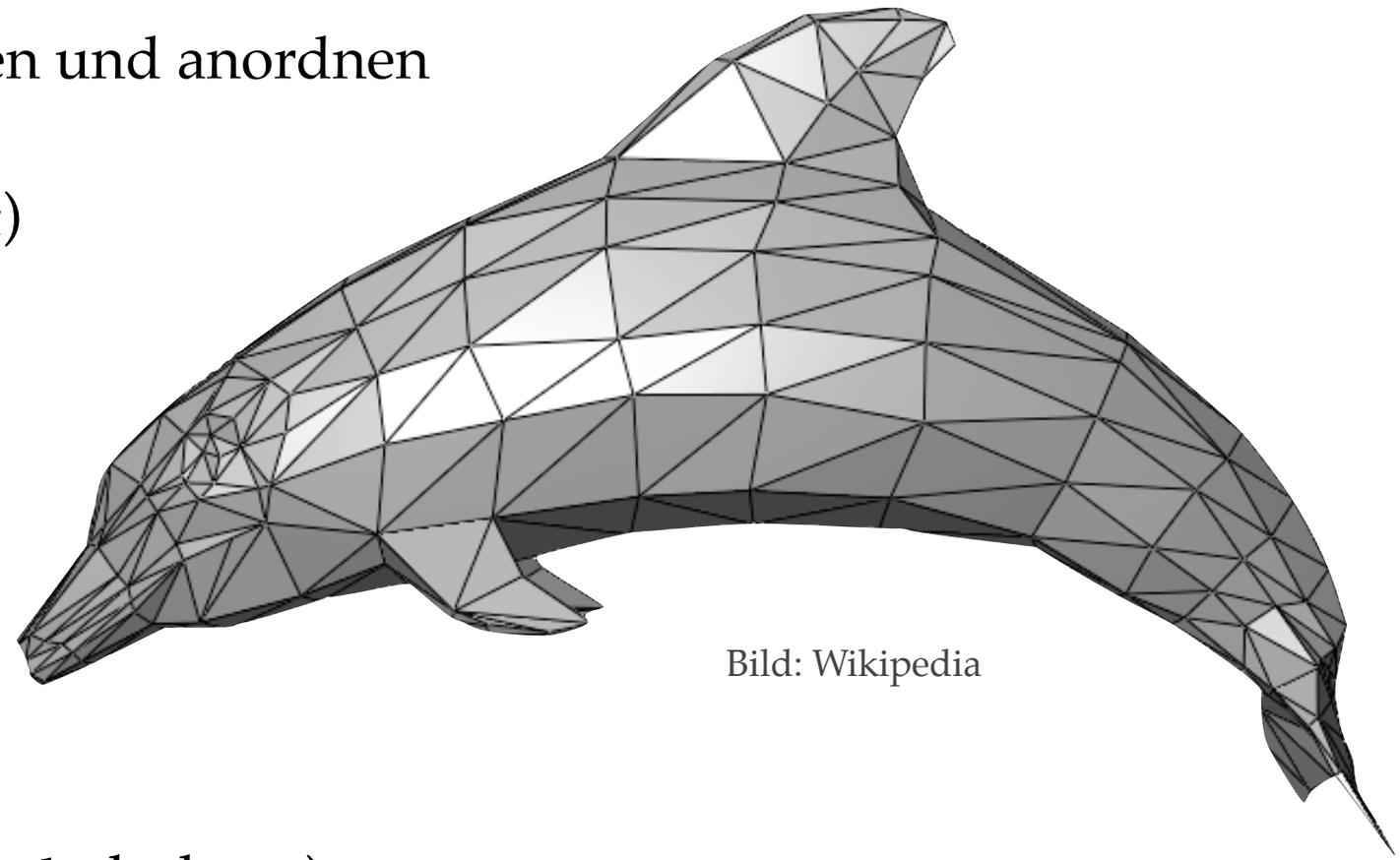
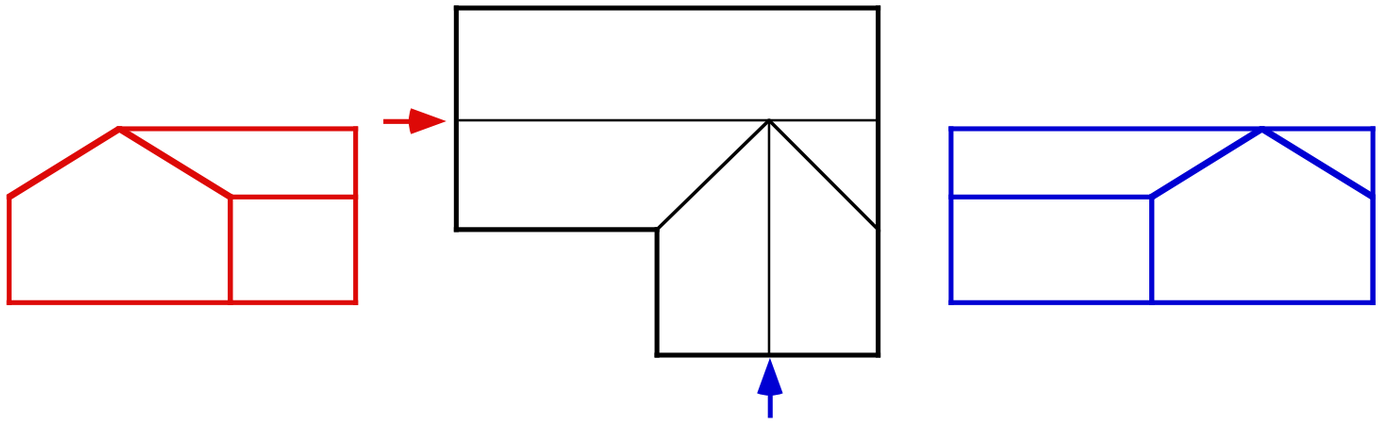
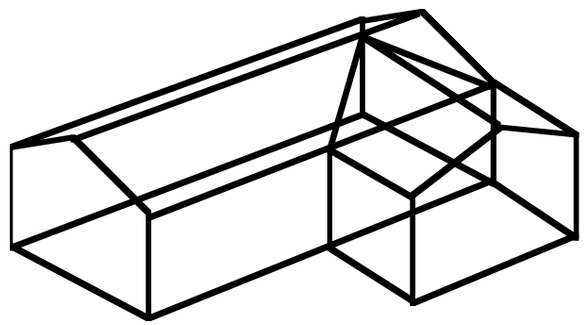


Bild: Wikipedia

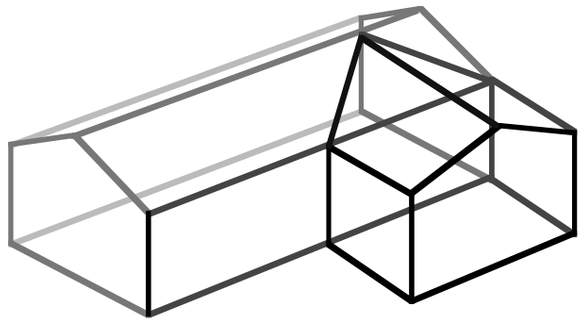
- Präsentation meist zweidimensional
  - Leinwand, Bildschirm, Papier
  - Projektion von 3-D Szenen auf 2-D Ebene
  - Tiefenhinweise gehen teilweise verloren
- Ansicht und Aufsicht

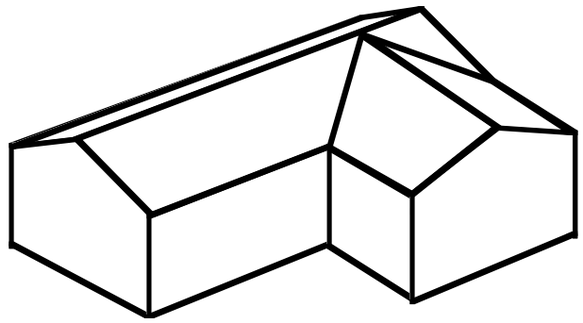


- Projektion und Drahtmodell
  - (fast) ohne Tiefeneindruck

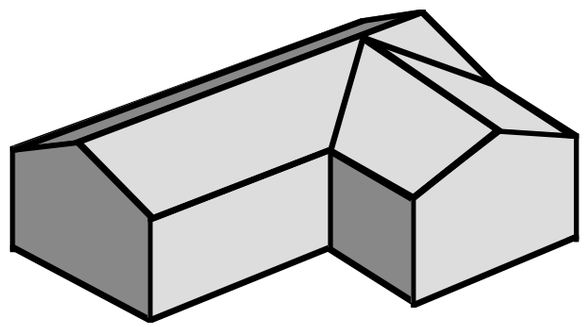


- Depth-Cueing
  - Linien 'vorne' hervorheben





  - Animation: Drehen um eine Achse
  - Entfernen verdeckter Linien
- Verbesserung der Darstellung
  - Füllen der Flächen

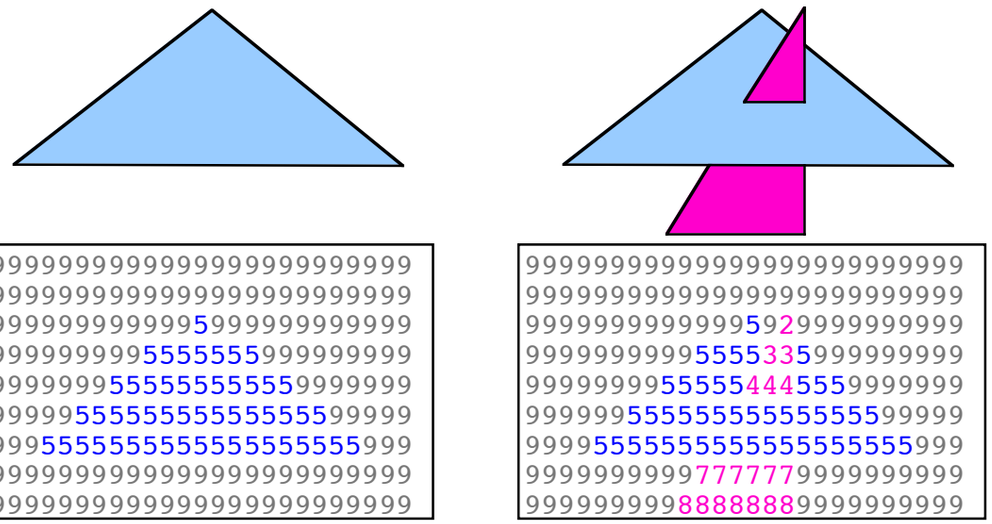


- Entfernen verdeckter Flächen

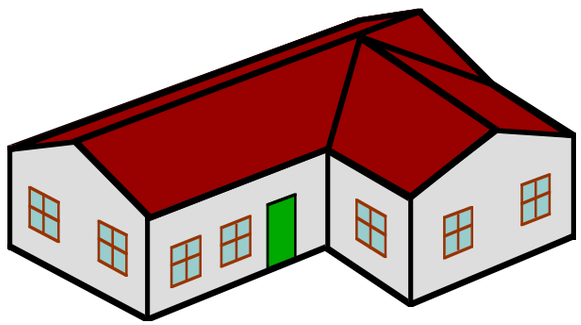
  Algorithmus von E. Catmull

  Tiefenpufferalgorithmus (z-buffer) Pixel = (R, G, B, Z)

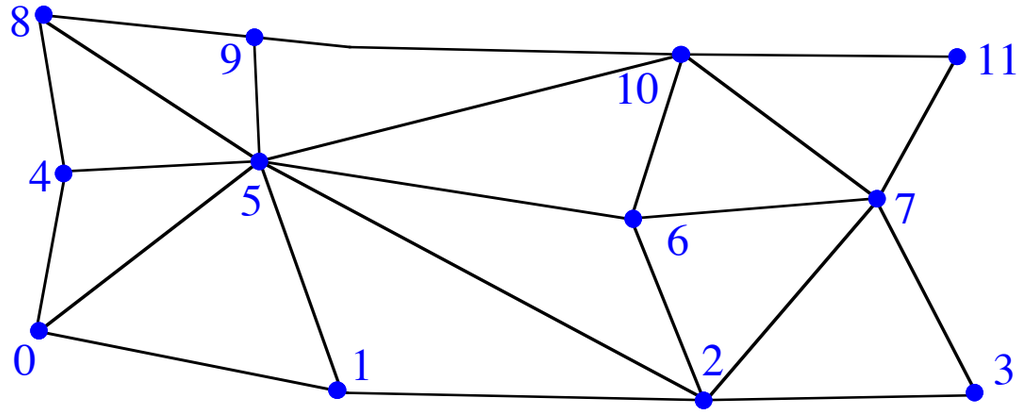
```
IF newpix.z<pixmap[x,y].z THEN pixmap[x,y]:= newpix;
```



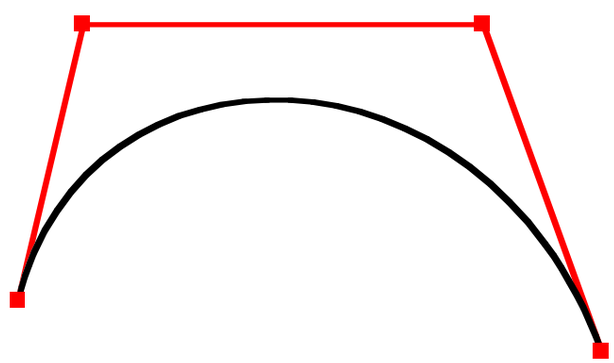
- Schattierungen simulieren Lichteinfall
- realistische Farben, Detail



- TriGrid: Gruppe von Dreieck-Facetten



  - vereinfachte Oberflächenbeschreibung
- Splines
  - stückweise definierte Kurve
  - Anpassung an vorgegebene Kurve
  - viele Spline-Typen mit besonderen Eigenschaften
  - Bézier, kubische Splines, deBoor
  - NURB: nonuniform rational B-spline



- Spline-Patches
  - stückweise Beschreibung von Oberflächen (patches)
  - Flächen als 3-dimensionales Analogon von Splines
  - Facetten sind Vierecke mit Splines als Kanten
  - NURB-patches

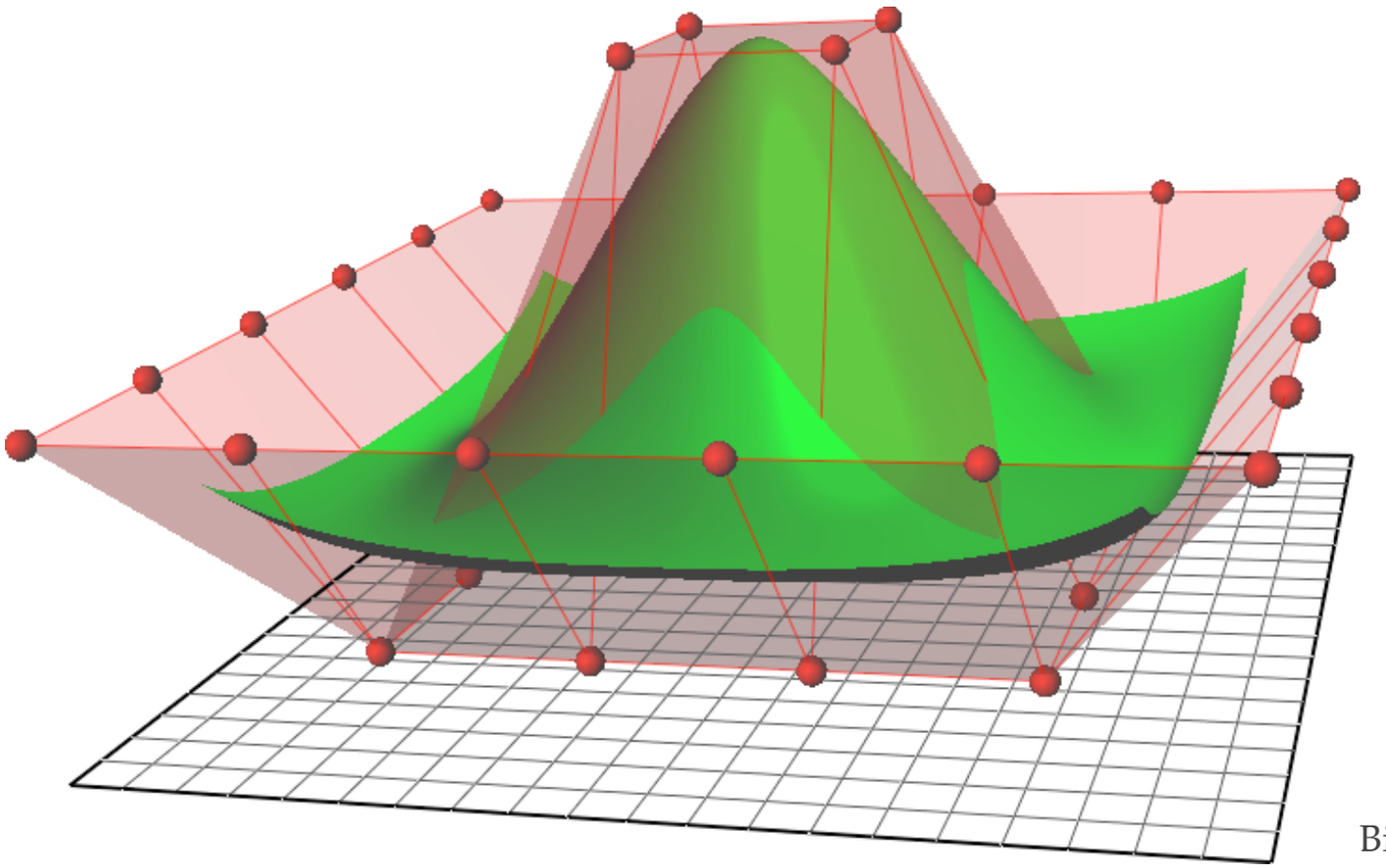
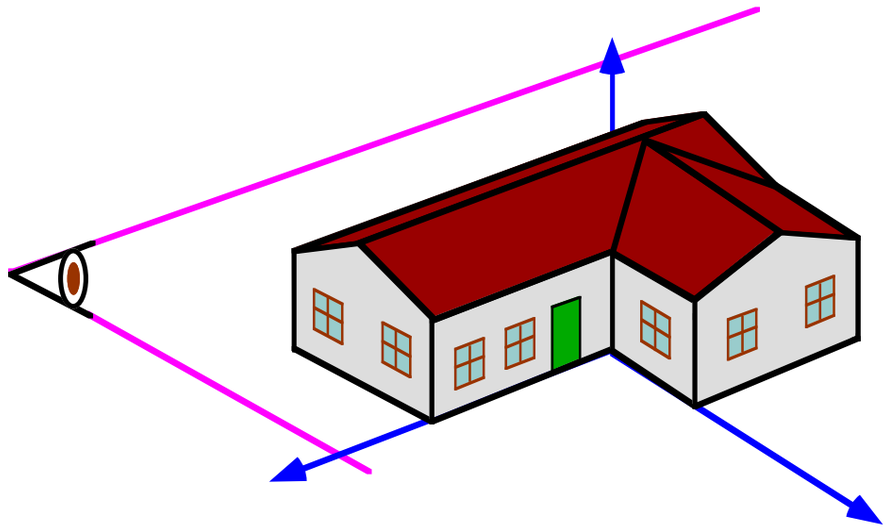
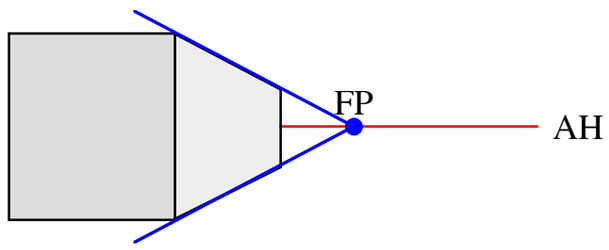


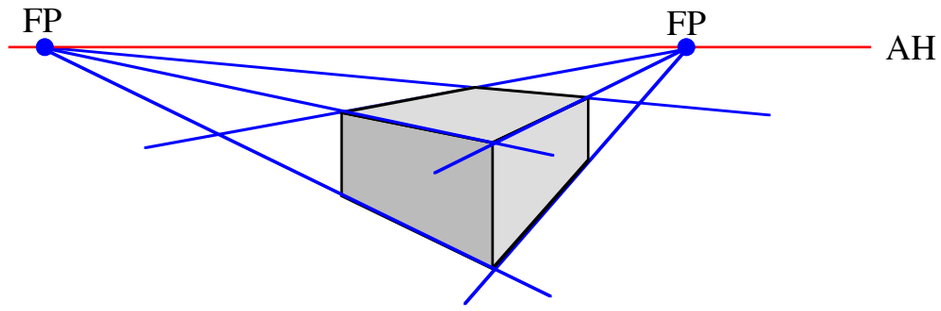
Bild: Wikipedia

- Kamera: Betrachtungsort, Blickwinkel, Öffnungswinkel
- Perspektivische Projektion

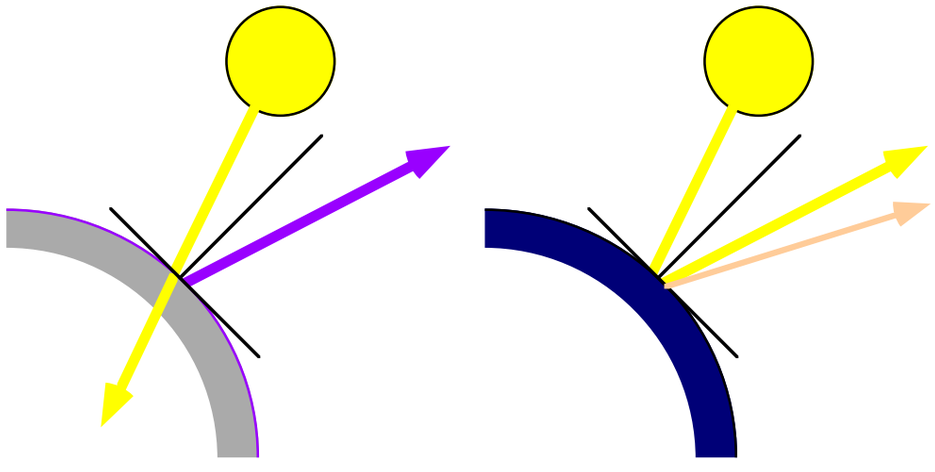


- Fluchtpunkt(e)
- Maße nicht korrekt ablesbar

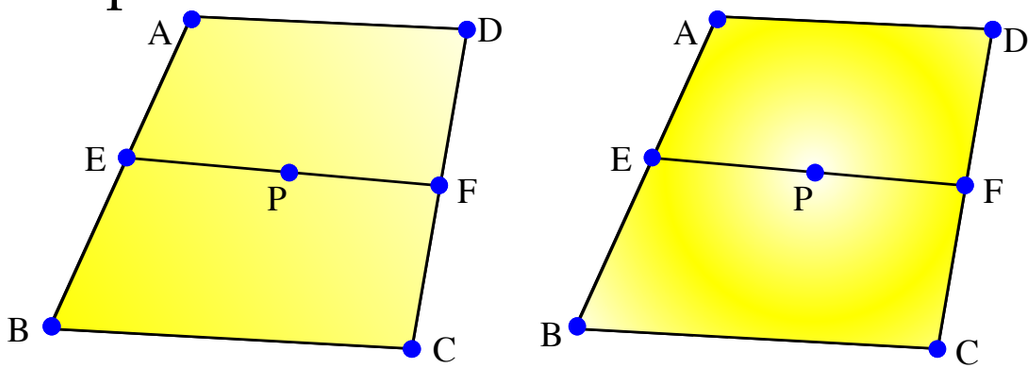




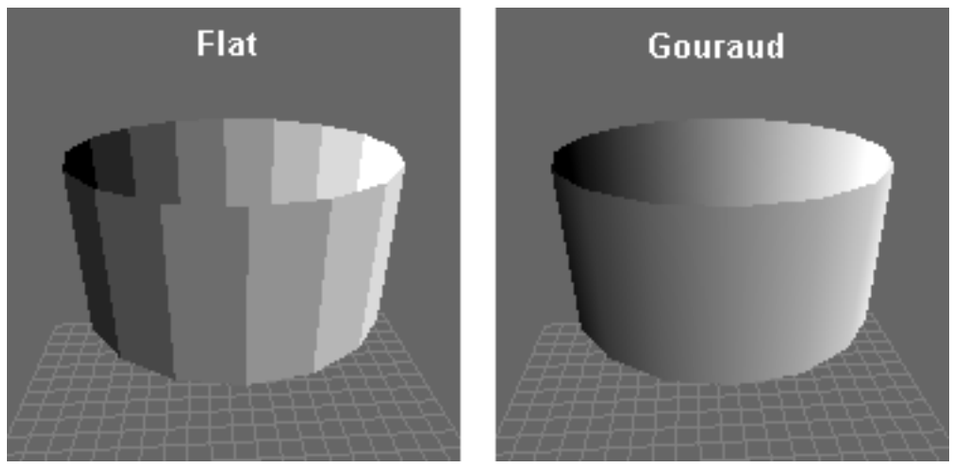
- Parallelprojektion
  - orthographische Projektion: Grundriß, Aufriß
  - schiefe (axonometrische) Projektion
  - isometrische Projektion
- Beleuchtung
  - Umgebungslicht (ambient), Punkt-Licht
  - diffuse Reflektion
  - Objekte werden von Lichtquelle angestrahlt
  - Licht wird teilweise reflektiert, teilweise durchgelassen
  - spiegelnde Reflektion
  - imitierendes Modell von Bui-Tung Phong, 1975



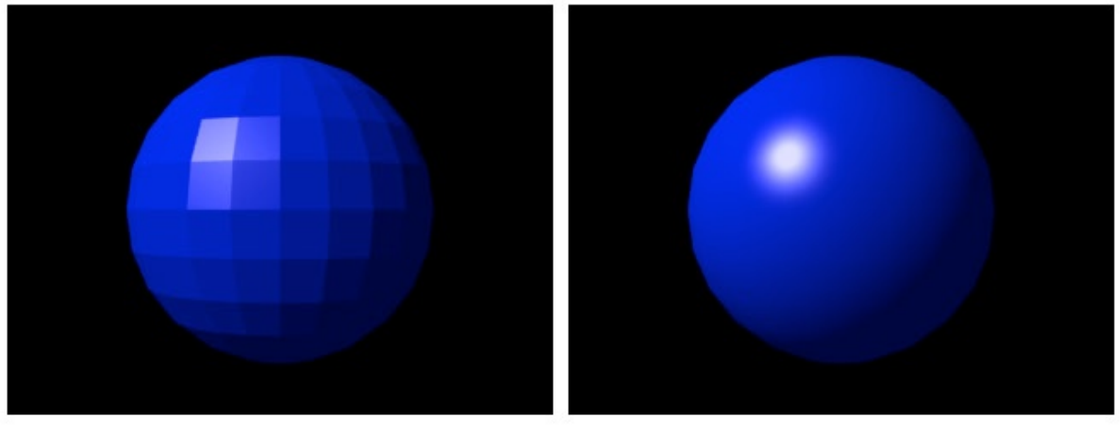
- Flat Shading: ein Farbwert pro Oberflächen-Facette
- Smooth Shading
  - Farbverlauf auf den Facetten
  - Gouraud-Shading: Interpolation zwischen Farbwerten an Eckpunkten



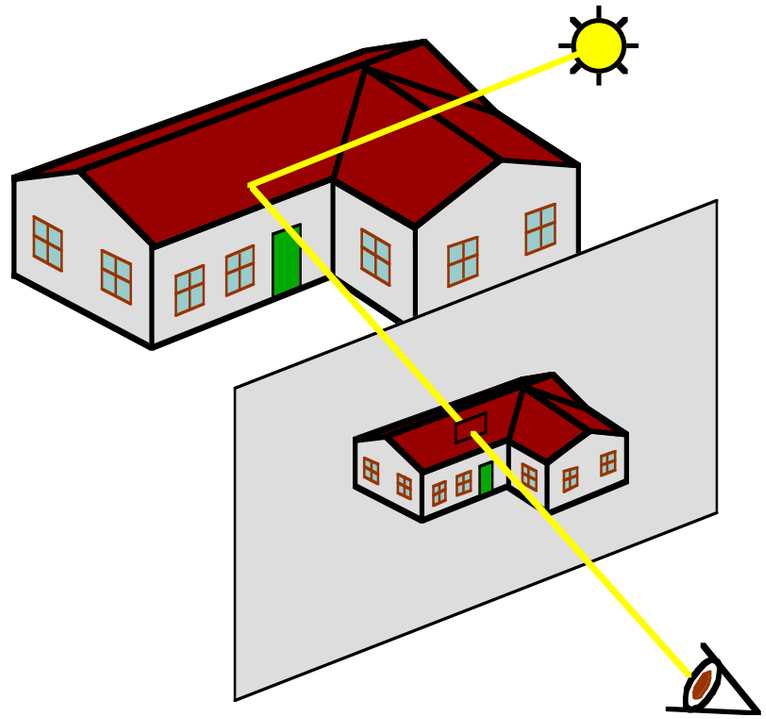
  - Phong-Shading: individuelle Intensitätsberechnung für Flächenpixel



Bilder: Wikipedia



- Ray-Tracing
  - Reflektionen von Reflektionen, mehrfache Spiegelung
  - Strahlpfad berechnen

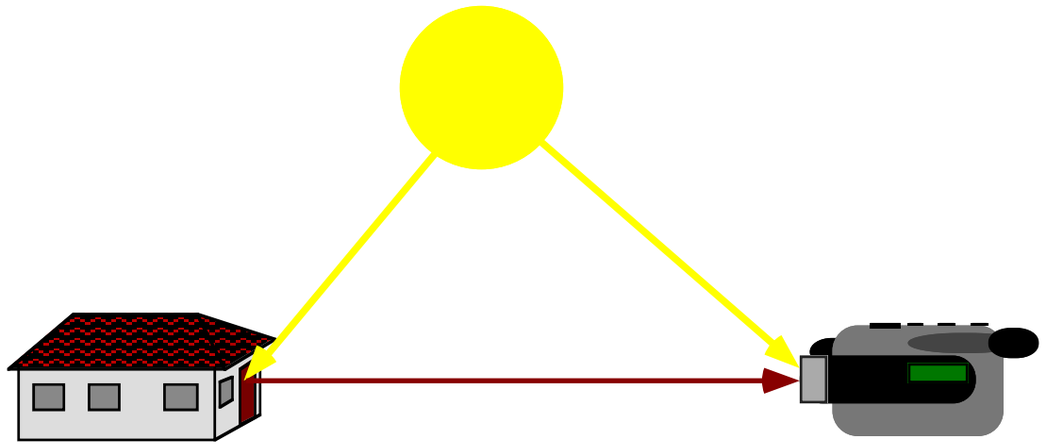


  - vorwärts und rückwärts (vom Auge zum Licht)
  - rechenintensiv
  - Berechnung blickpunktabhängig

- Radiosity
  - sichtunabhängige Berechnung
  - Einteilung der Oberflächen in patches
  - Emitter und Reflektor
  - Beleuchtungseinfluß auf alle anderen patches berechnen
  - Formeln aus der Wärmelehre
  - Abbrechen der Berechnung unter einem Grenzwert
- Textures
  - Oberflächenstruktur (Holz: Maserung)
  - 'Bekleben' der Oberflächen mit Muster
  - Bilder und Filme als Texturen
- VRML: Virtual Reality Markup Language
  - textuelle Beschreibung von 3D-Objekten und Szenen
  - primitive Objekte (cylinder, …)
  - Transformation, Gruppierung, Oberflächeneigenschaften
  - Texturen (MPEG-Filme)
  - Objekte und Hyperlinks
  - Sensoren erzeugen Events für andere Objekte

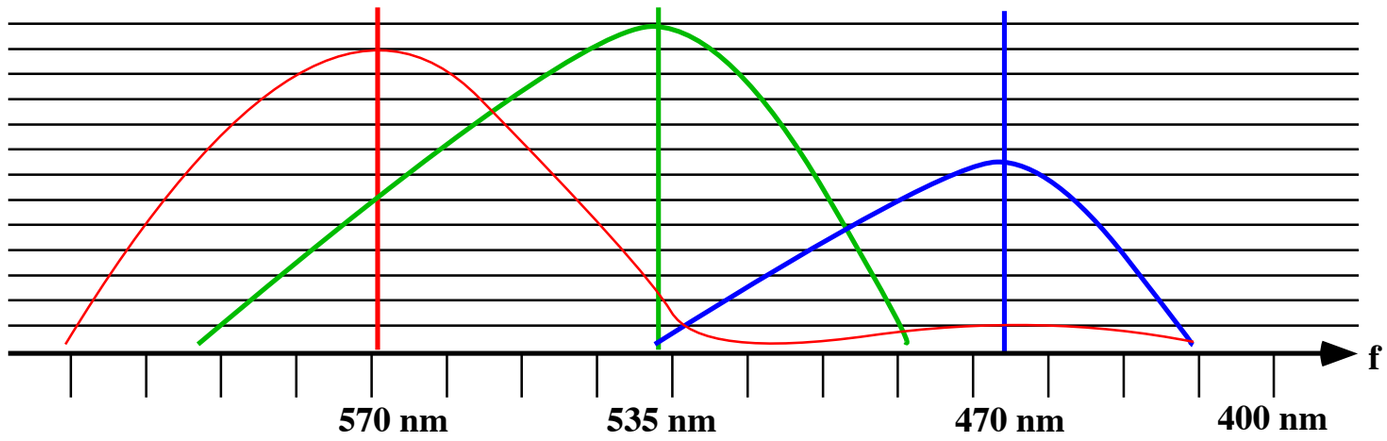
Standbilder

- Kontinuerliche Verläufe
  - Film hat höhere Auflösung als Auge
  - Abzüge, Bücher
  - Guter Druck typisch 2500 dpi
- Farbe
  - Lichtquelle (, Reflektion), Auge/Kamera/…:

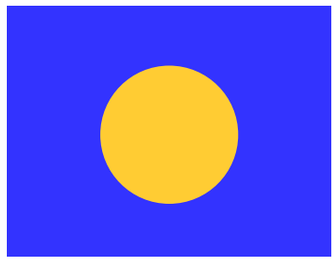


- Reflektiertes Licht = Licht - absorbiertes Licht = Oberflächenfarbe

- Spektrum und Empfindlichkeit des menschliche Sehapparates
  - 120 M Stäbchenzellen für Helligkeit in der Peripherie

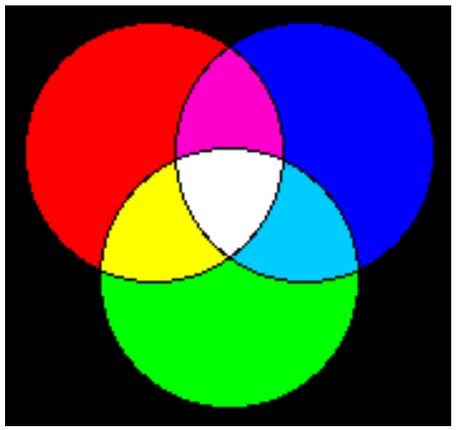


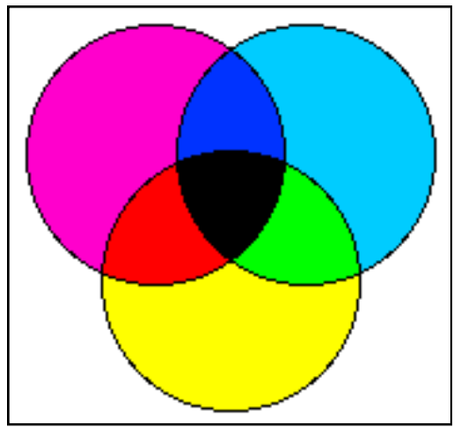
  - 7 M Zapfenzellen für Farbe (570, 535, 455 nm)
- Mensch sieht bis zu 350.000 Farbnuancen



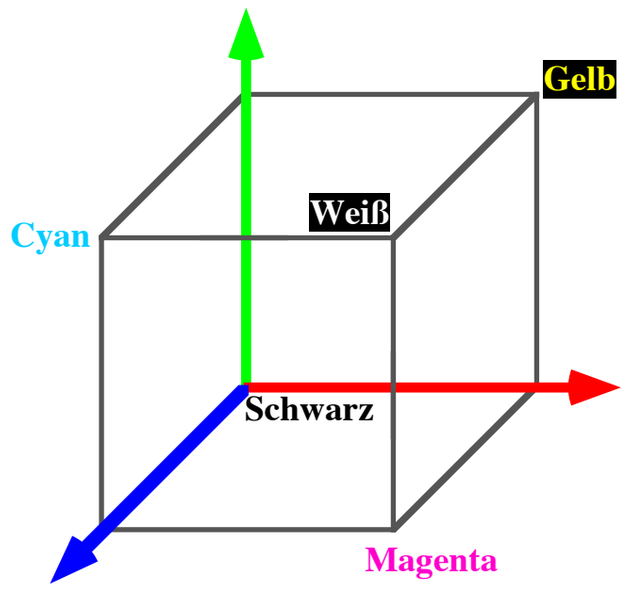
- Abschattung

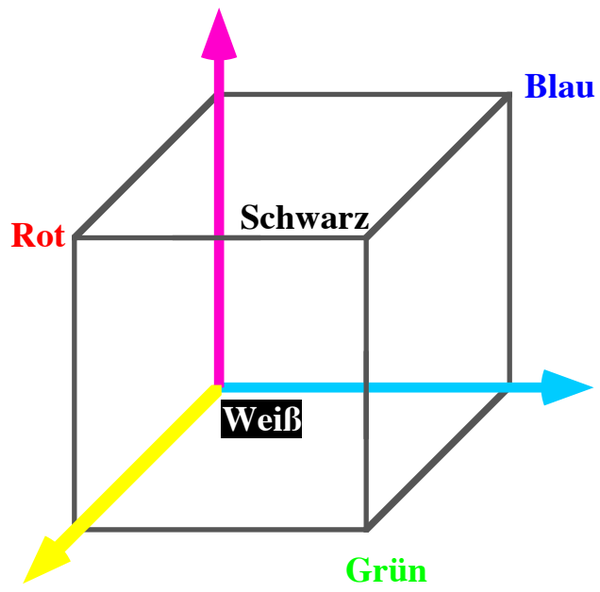
- Farbmischen: additiv subtraktiv





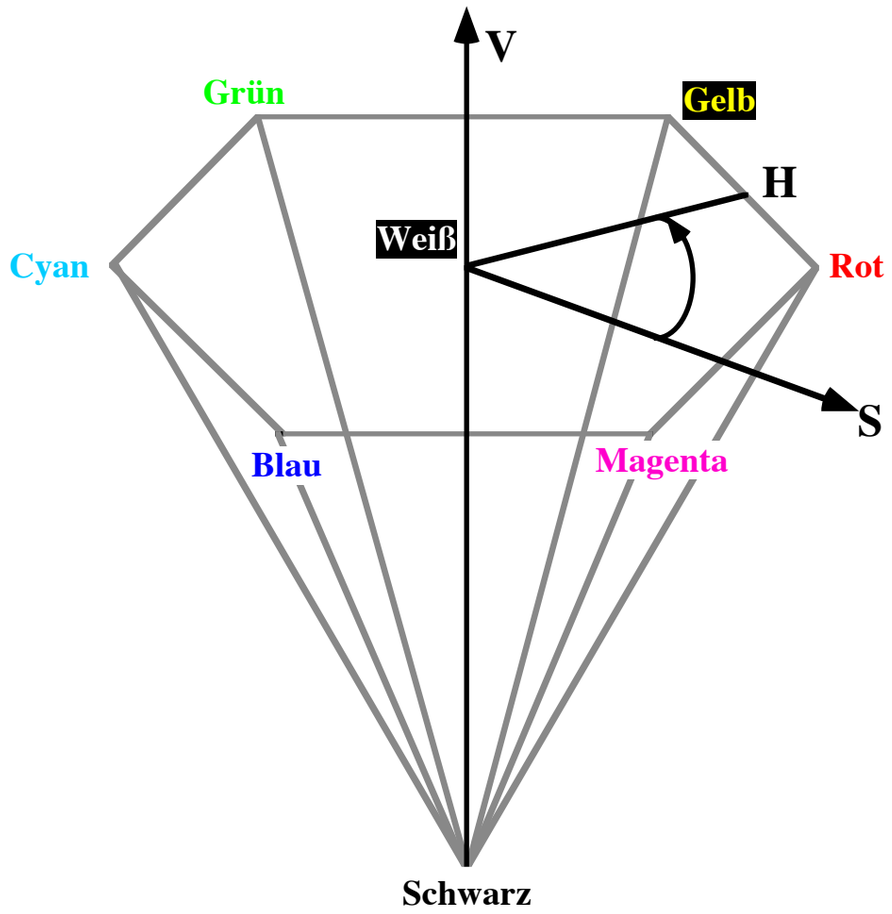
- Farbmodelle RGB CMY



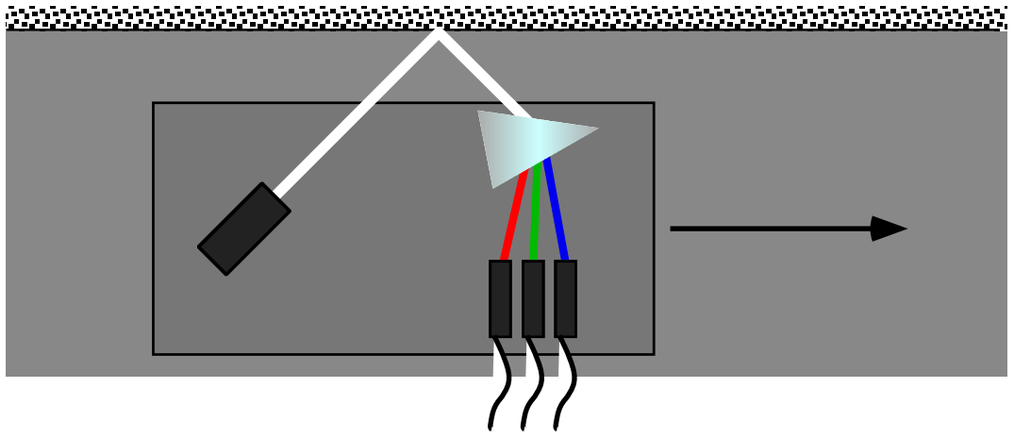


$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} Weiß \\ Weiß \\ Weiß \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

- HSV (Ton, Sättigung, Helligkeit)

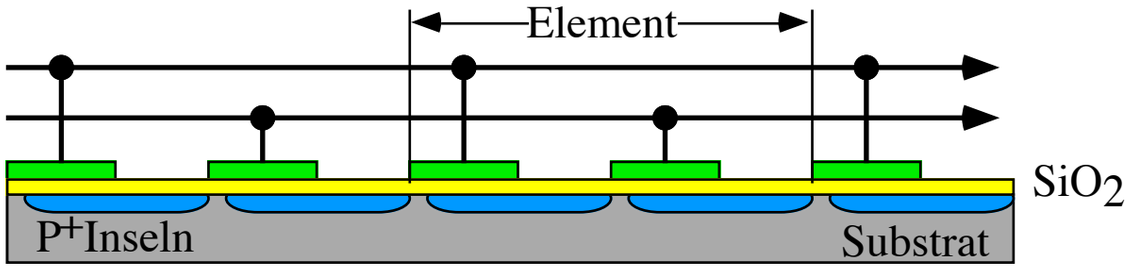


- Digitalisierung
  - Horizontale und vertikale Diskretisierung (Zerlegen in Pixel)
  - Diskretisierungsschritt entspricht Auflösung: 72 bis 6000 dpi
  - Bild wird angeleuchtet und Licht auf Detektor reflektiert
  - Quantisierung: 8 oder 12 Bit für Graustufen

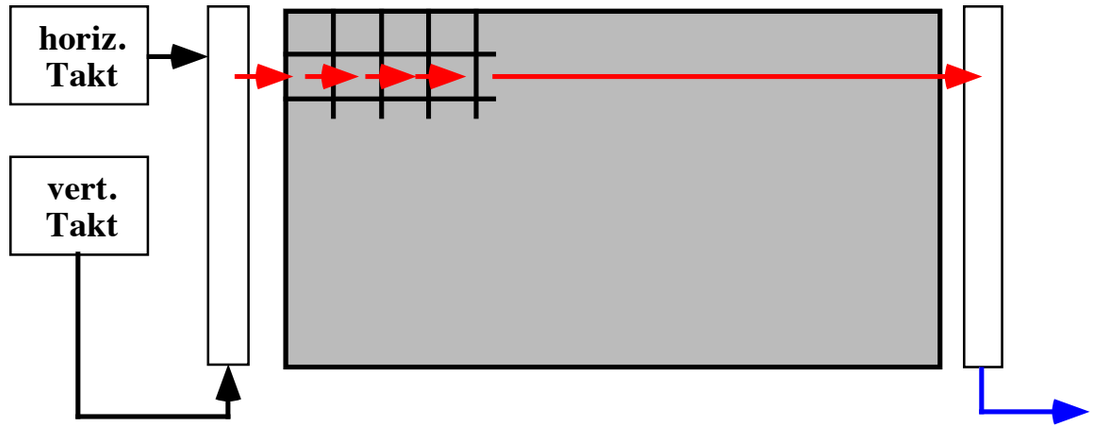


8, 16 oder 24 Bit für Farbe
eventuell mit Farbpalette

- CCD-Zeile

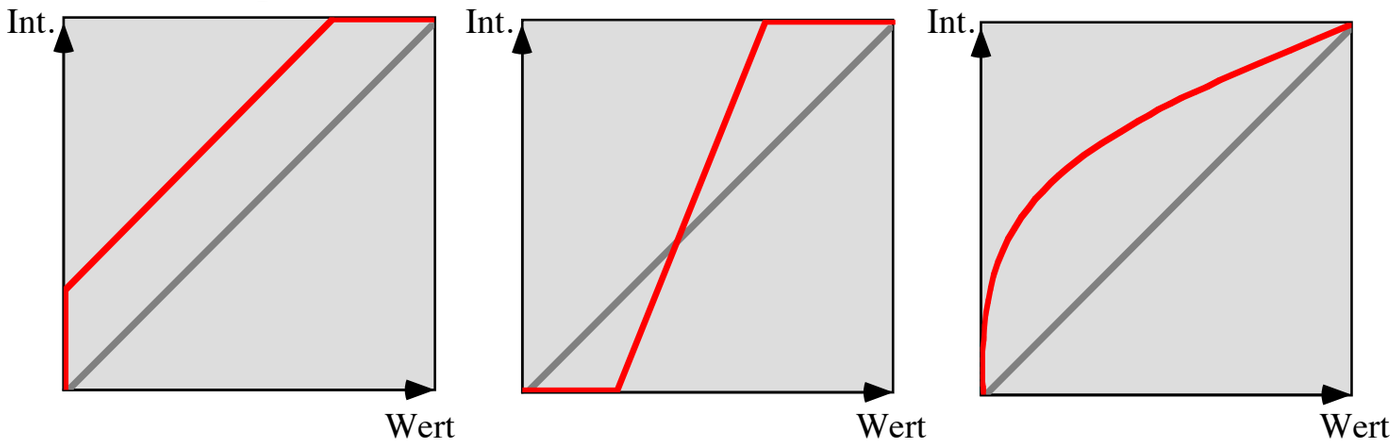


- Digitale Kameras benutzen CCD-Matrix



  - ca 3000-4000 ppi
  - Unterschiede zwischen Zellen
  - nur 50 - 80% der Chipfläche ist mit aktiven Elementen bedeckt
- Bildkodierung
  - RGB wird meist bei Computermonitoren verwendet
  - CMYK (Cyan, Magenta, Yellow, Schwarz) besonders für Druck
  - HSV für Fernsehen

- Aufbereitung nach der Digitalisierung
  - Kalibrierung der Farbwerte
  - Helligkeitsregelung, Kontrastverstärkung und 'Gamma' pro Farbkanal
  - Vorsicht: Color-Matching verwendet auch



Helligkeit Kontrast Gamma

- Datenmenge kann groß werden
  - Auflösung für Weiterverarbeitung wichtig (Druckgewerbe)
  - (300 * 4 [inch]) * (300 * 6 [inch]) * 3 Bytes = 6.480.000 Bytes
    => Kompression

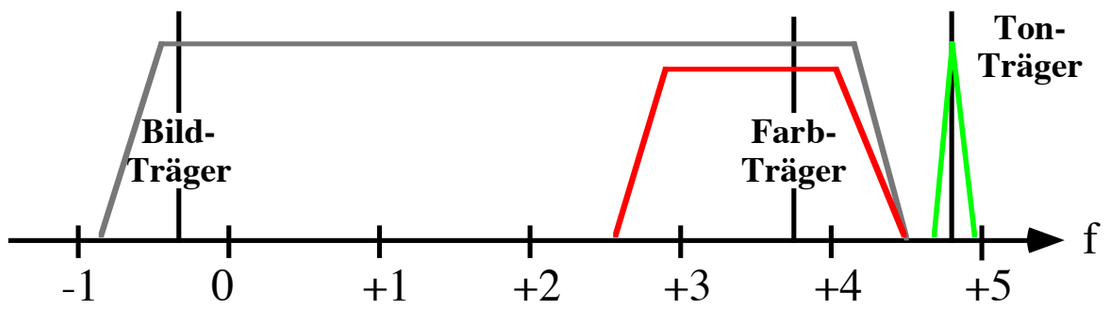
Video

- S/W Fernsehen (eigentlich Graustufen)
- Auflösung wesentlich geringer als bei Standbildern

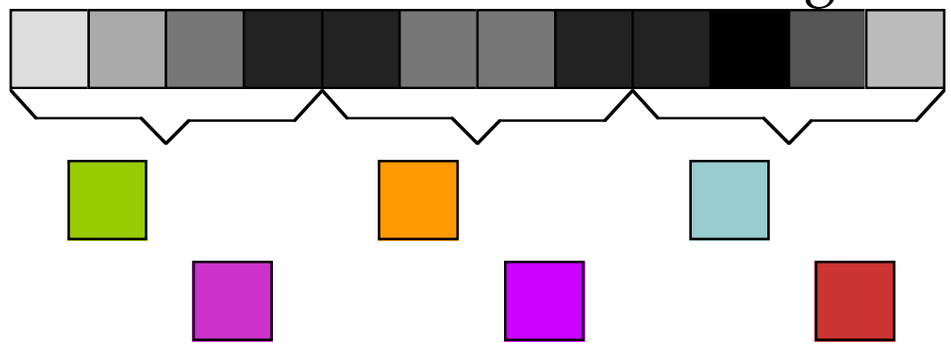
	Zeilen	Punkte/Zeile	Bilder/s	
CCIR 601	486	720	30	59,94 Hz
	586	720	25	50 Hz
CIF	288	352	25	Common Intermediate Format
QCIF	144	176	25	Quarter CIF
SIF	240	352	30	Standard Intermediate Format

- Fernsehnormen
  - Halbbilder (Felder, gerade/ungerade Zeilen) mit doppelter Frequenz
  - Farbfernsehen: PAL, SECAM: 50 Hz
    NTSC: 59,94 Hz (in Europa 50 Hz)
  - Bildwiederholrate = Übertragungsrate
- HDTV: High Definition TV
  - 1366-1920 * 720-1080; (Bildwiederholrate 50 oder 59,94 Hz)
  - Bildwiederholrate ≠ Übertragungsrate (24 Hz, 36 Hz, 72 Hz)
  - MPEG-2/4 zur Übertragung

- Kameras produzieren RGB
  - drei Übertragungskanäle
  - Synchronisation?
  => Mischsignal
- Composite
  - NTSC (National Television Systems Committee, …)
  - PAL (Phase Alternating Line)
  - SECAM (Sequentiel Couleur avec Memoire)
  - Grundidee: SW-Fernsehen + irgendwas = Farbe
  - Farbraum mit Luminance und Chrominanz
  - Luminance := SW-Signal
- Farbraum HSV
  - Chrominanzsignal mit niedrigerer Bandbreite
  - auf Subcarrier (3,58 MHz)



- Farbauflösung des Auges schlechter -> Unterabtastung



z.B.: 4:1:1 (YUV, PAL), 15:5:2 (YIQ, NTSC)

- Koeffizienten entsprechen Farbempfindlichkeit des Auges
- NTSC: YIQ (In-phase and Quadrature, I: 1,3 MHz, Q: 0,45 MHz)

  Y = 0,30 R + 0,59 G + 0,11 B;

  I = 0,60 R - 0,27 G - 0,32 B;

  Q = 0,21 R - 0,52 G + 0,31 B;
- PAL: YUV (U, V: 1,3 Mhz)

  Y = 0,30 R + 0,59 G + 0,11 B;

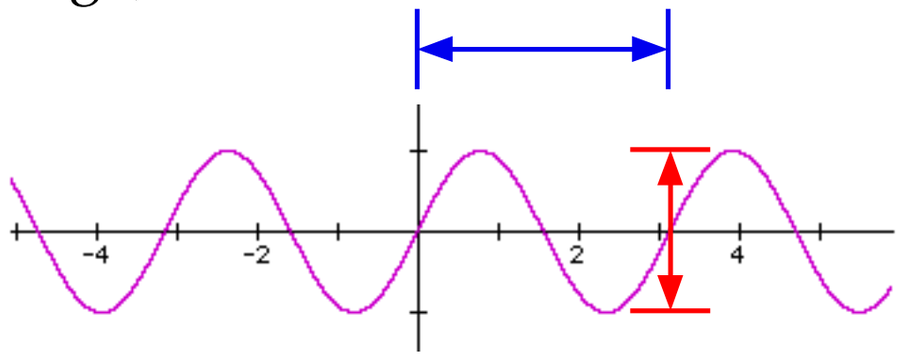
  U = (B-Y) * 0,493 = -0,15 R - 0,29 G + 0,44 B;

  V = (R-Y) * 0,877 = 0,61 R - 0,52 G - 0,10 B;
- VHS noch stärker analog komprimiert

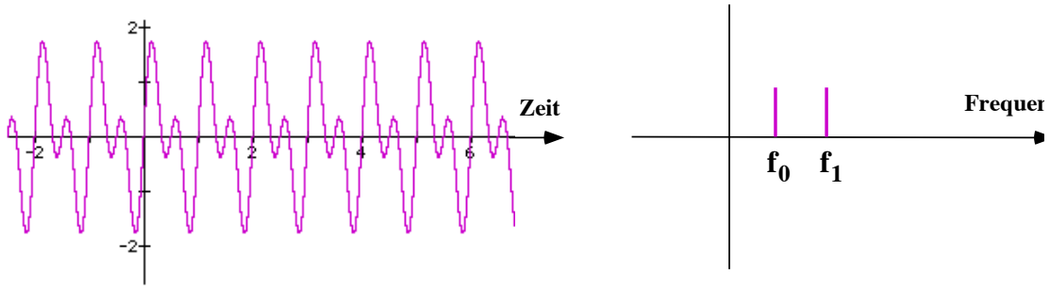
Audio

Audio-Eigenschaften

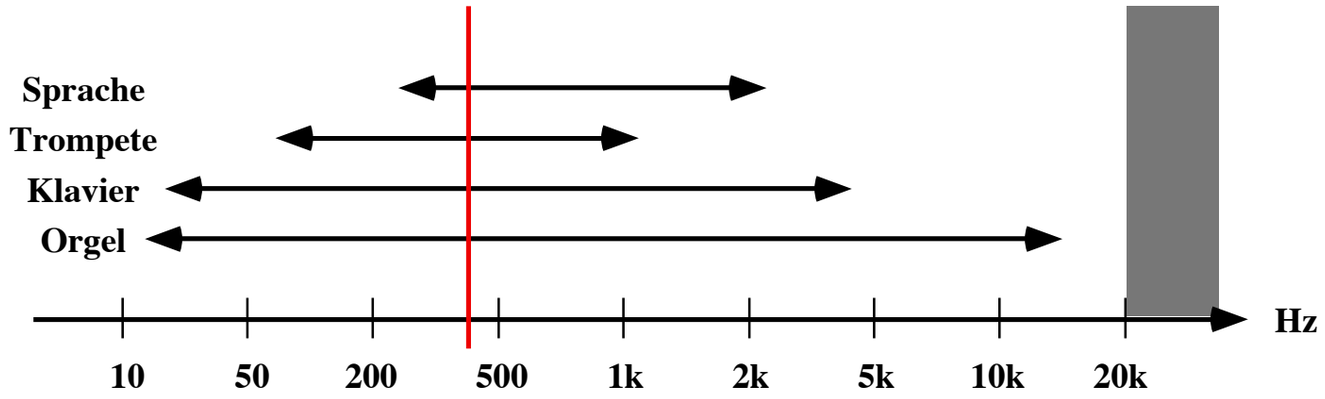
- Frequenz und Amplitude
  - Amplitude -> Lautstärke (gemessen in dB)
  - Frequenz (1m/Wellenlänge) -> Tonhöhe



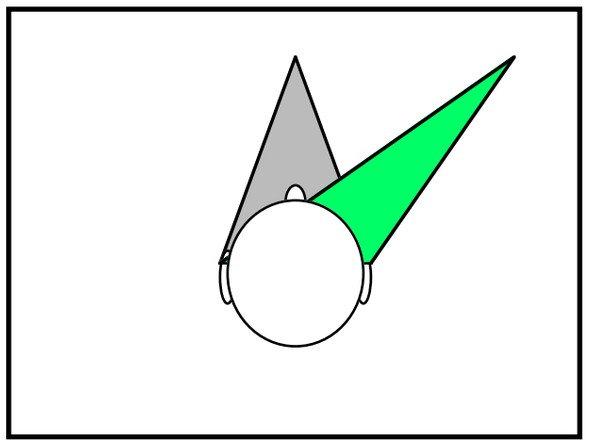
- Fourier: Jede Schwingung kann als Summe von Sinusschwingungen dargestellt werden:



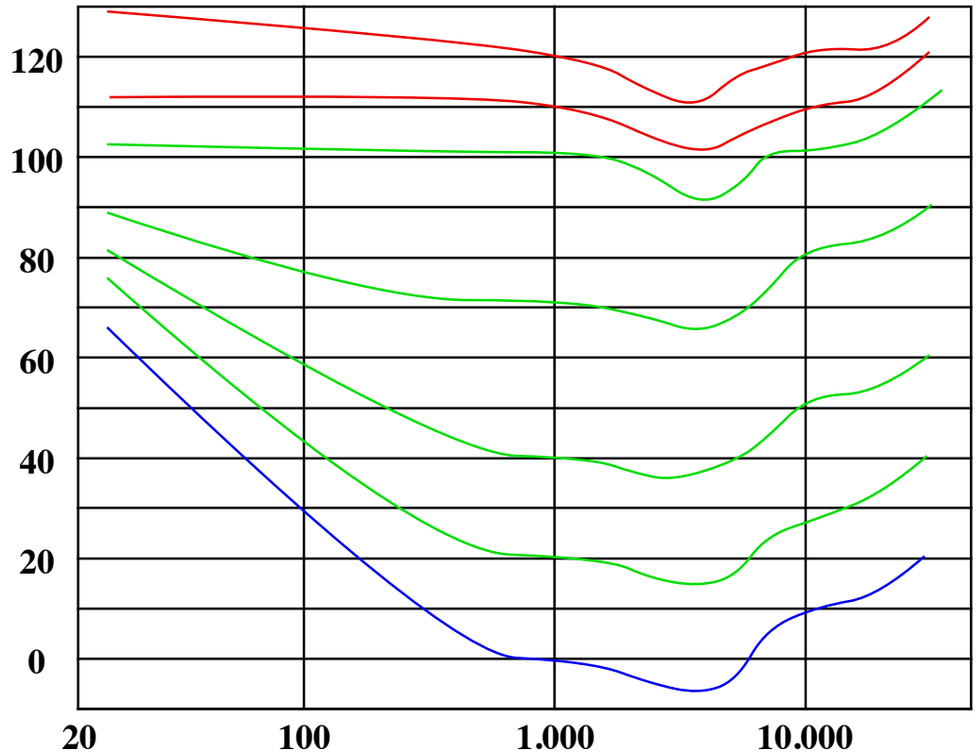
- Typische Frequenzbereiche
  - Telefon 300Hz - 3.400 Hz
  - Heimstereo 20 Hz - 20.000 Hz
  - UKW (FM) 20 Hz - 15.000 Hz



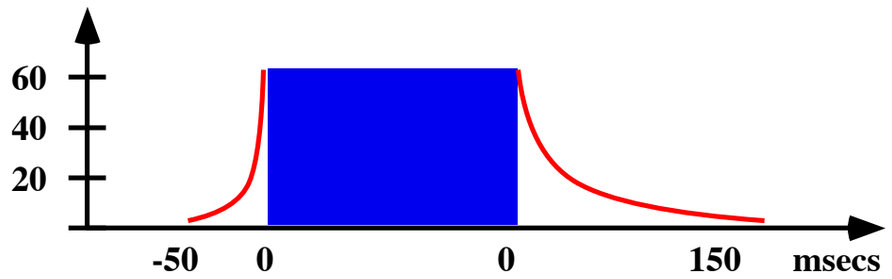
- Räumliches Hören
  - Lautstärke
  - Laufzeitunterschiede zu den Ohren
  - Spektrale Analyse nach Ohrposition
  - Filterfunktionen durch Außenohr
  - Echos



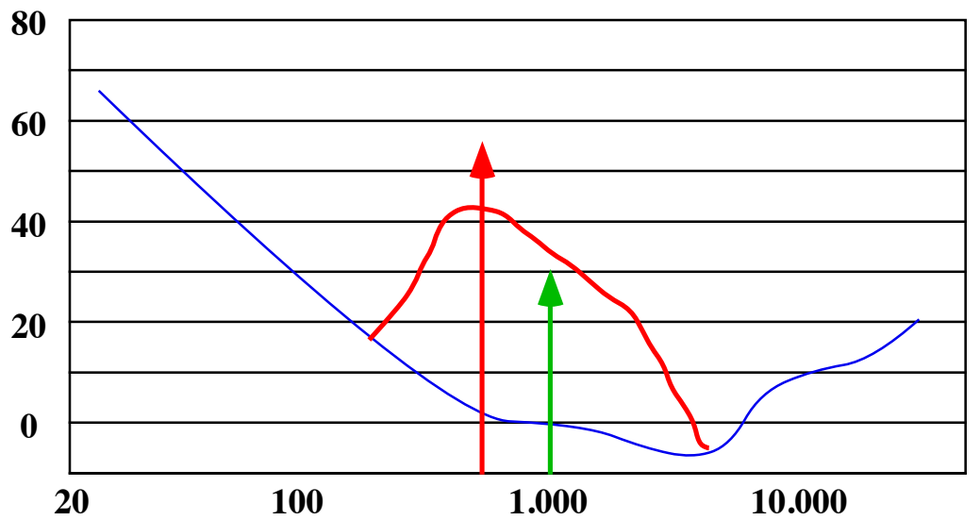
- Menschliches Hörvermögen
  - 20 - 20.000 Hz
  - hohes zeitliches Auflösungsvermögen
  - logarithmisch bezüglich Amplitude
- Lautstärkeempfinden nach Fletcher und Munson



- Abschattung
  - Zeit



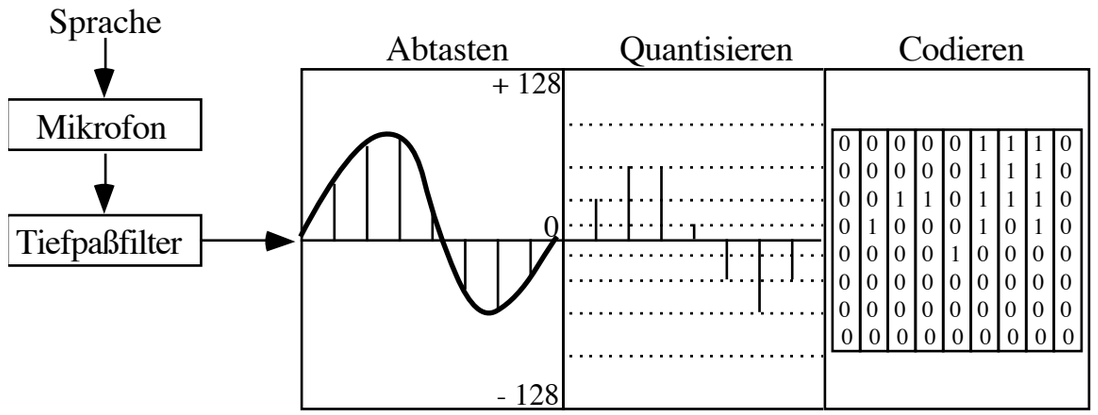
  - Frequenz



- Phase $y = \sin x + \sin(x + \pi)$
- zwei gleiche, phasenversetzte Schwingungen können sich auslöschen:

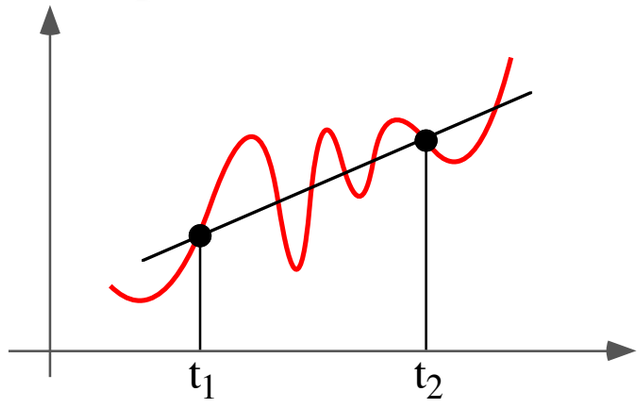
Digitale Repräsentationen (PCM, CD-Audio, DAT, ...)

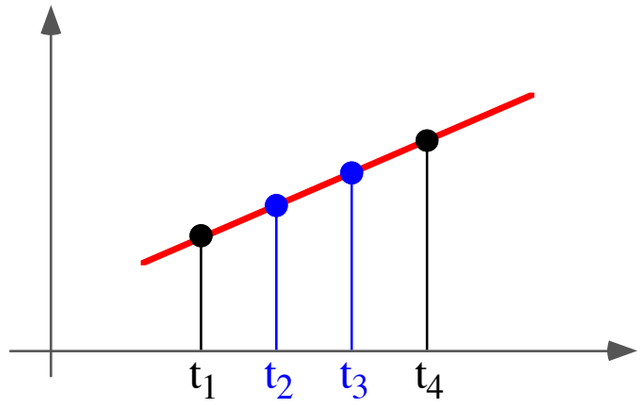
- Digitalisierung am Beispiel Telefon



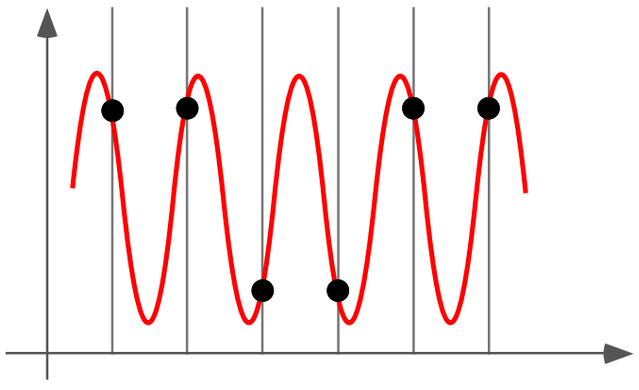
- Allgemein
  - zeitliche Diskretisierung (Abtasten, Sampling)
    Einteilung der Zeitachse in einzelne Stücke
  - Wert-Diskretisierung (Quantisierung)
    digitalen Näherungswert finden
    Reelle Zahl vs. Real/Integer

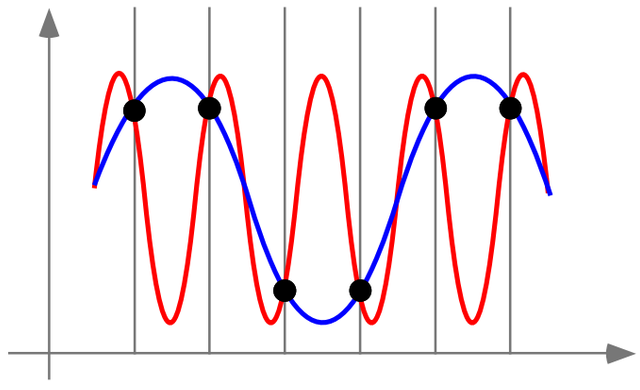
- Abtasttheorem
  - Anzahl Abtastwerte pro Zeiteinheit?





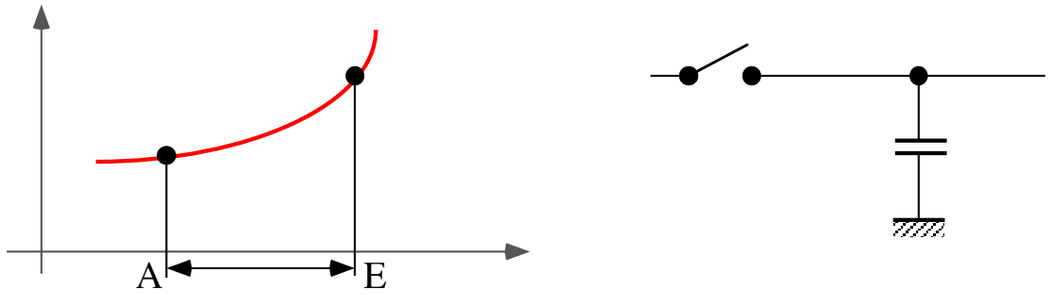
  - Abtastfrequenz > 2 * (höchste Frequenz)
  - [Whittaker 1915/1929, Borel 1897]
  - Aliasing bei zu niedrigen Abtastraten



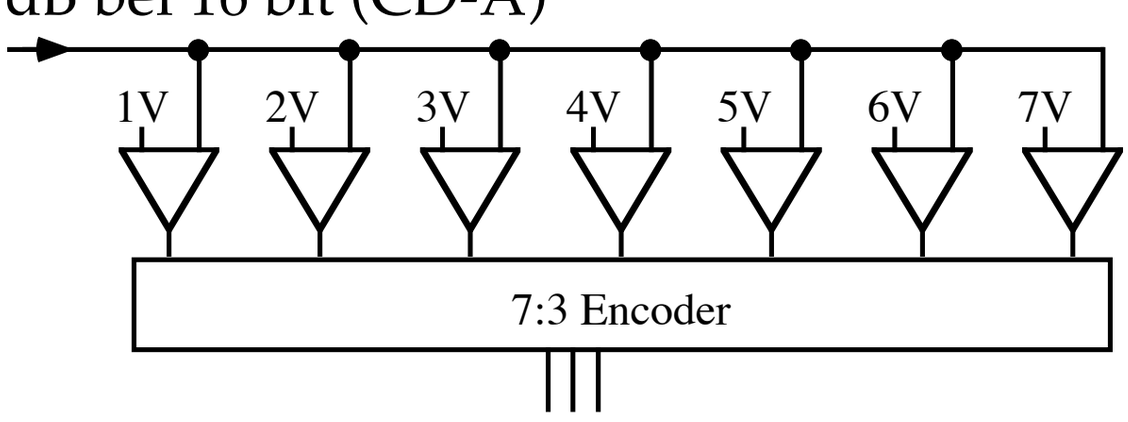


- Tiefpaß-Filter gegen Aliasing

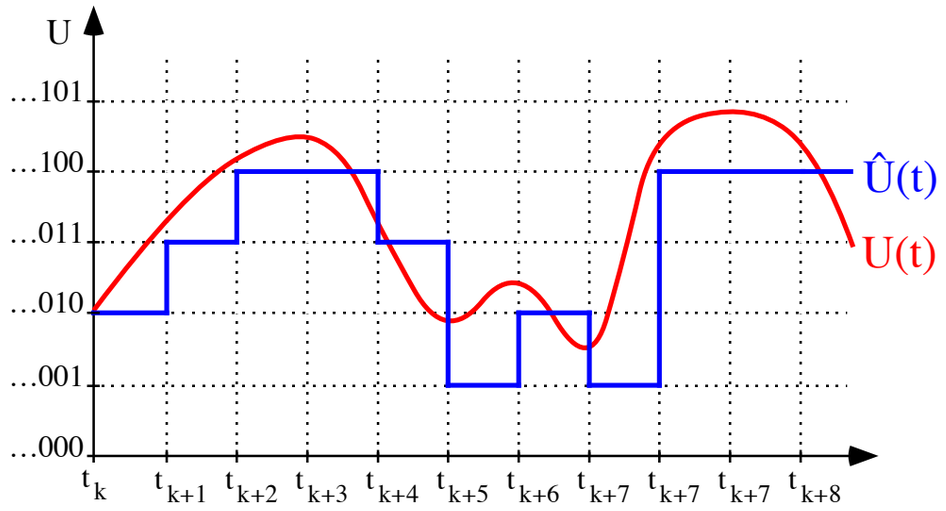
- Sample-and-Hold



- Quantisierung (ADC)
  - Wandlung des analogen Wertes in diskreten (digitalen) Wert
  - Quantisierungsfehler
  - 6 dB pro Bit => 96 dB bei 16 bit (CD-A)



- Diskretisierung und Quantisierung ergeben Treppenfunktion



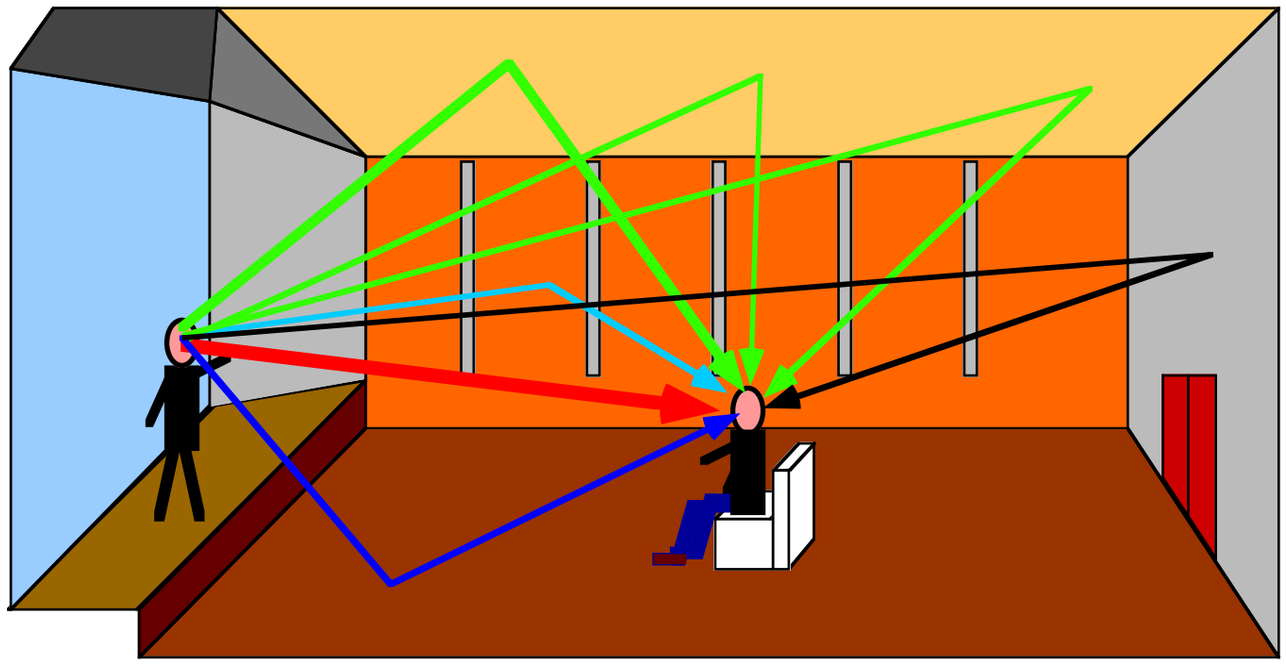
- Codierung
  - als Integerzahl (CD: 16 bit)
  - als Pseudo-Real (A-law, μ-law: 8 bit)
  - als Differenzen

±	E	x	p	M	a	n	t

- typische Verfahren

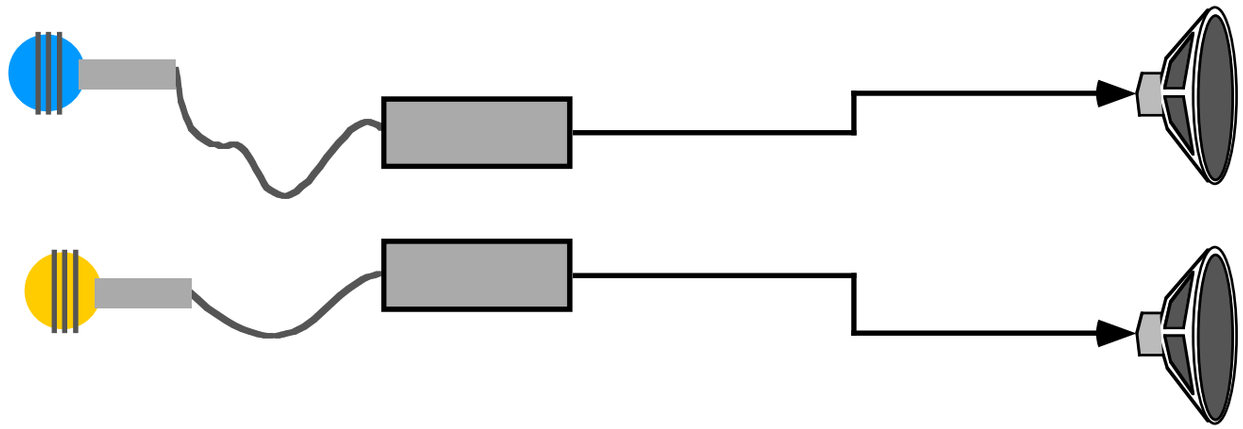
	bit	samples/sec	Kanäle	Datenstrom
CD-Audio	16	44.100	2	~1.4 Mbit/s
A-law (ISDN)	13 -> 8	8.000	1	64 kbit/s
ADPCM (Telefon)	13 -> 2	8.000	1	16 kbit/s

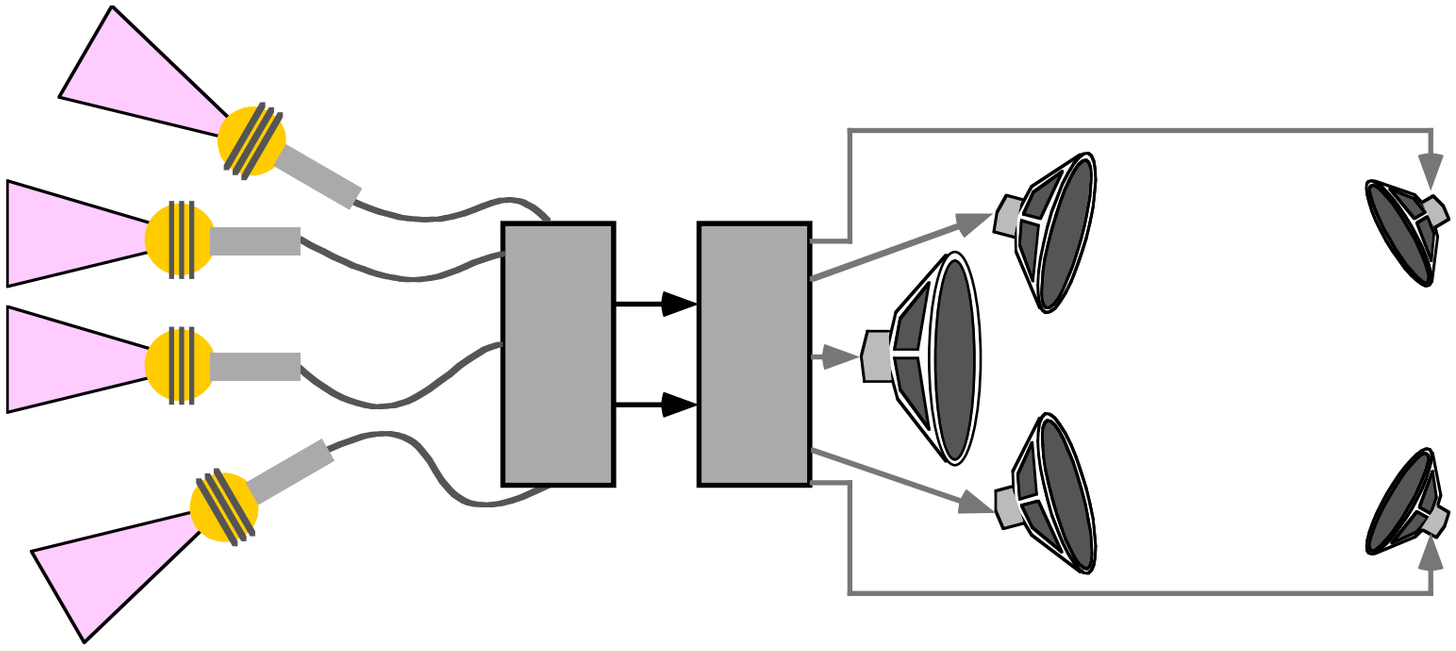
Raumton



- Reflexionen von Wänden, Decke, Boden, Gegenständen
- Wahrnehmung der
  - Signalstärke
  - Richtung der Quelle
  - Dämpfung durch Kopf in höheren Frequenzen
  - Laufzeitunterschiede (650 μsec hörbar)
- Simulation der Reflexionen durch Laufzeitunterschiede
- Kopfhörer
  - kontrollierte Umgebung
  - Bewegungssensor: Kopfdrehung, Ortsveränderung
  - keine Richtungsortung

- Stereo: zwei Kanäle, links und rechts
- Simulierter Raumklang
  - Reflektionen durch Verzögerung simulieren (-> Hall)
- Aufwendige Lautsprecheranordnung
  - Surround Sound (Dolby, DTS)
  - Raumaufnahme oder Simulation
  - Richtungsortung möglich

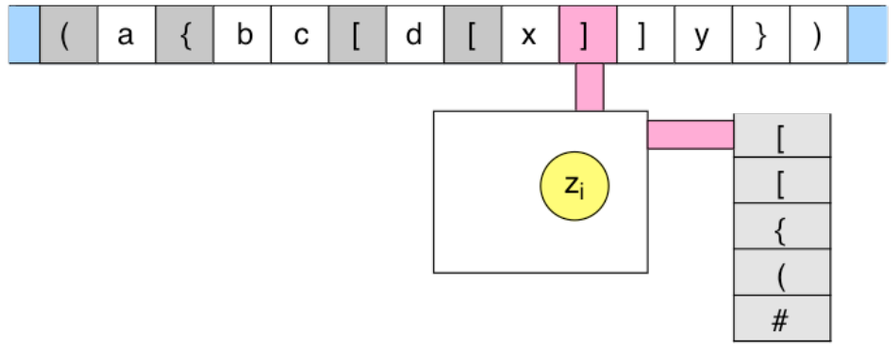


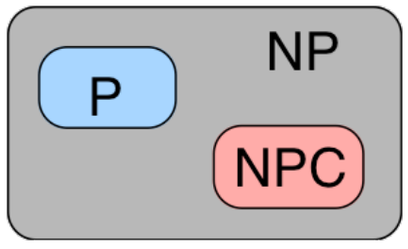


KAPITEL 8

Theoretische Informatik

- Grundlagen
- Erkenntnisse zur Berechenbarkeit
- Formale Sprachen
- Automaten
- Notation

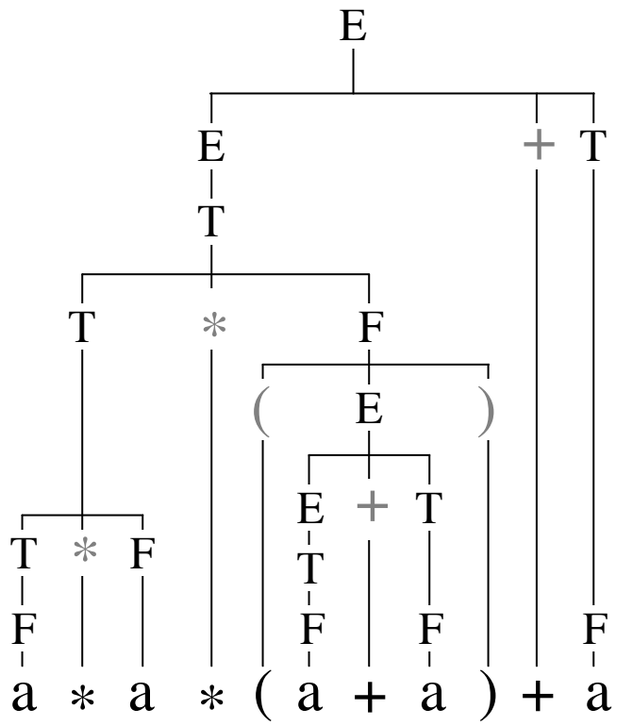




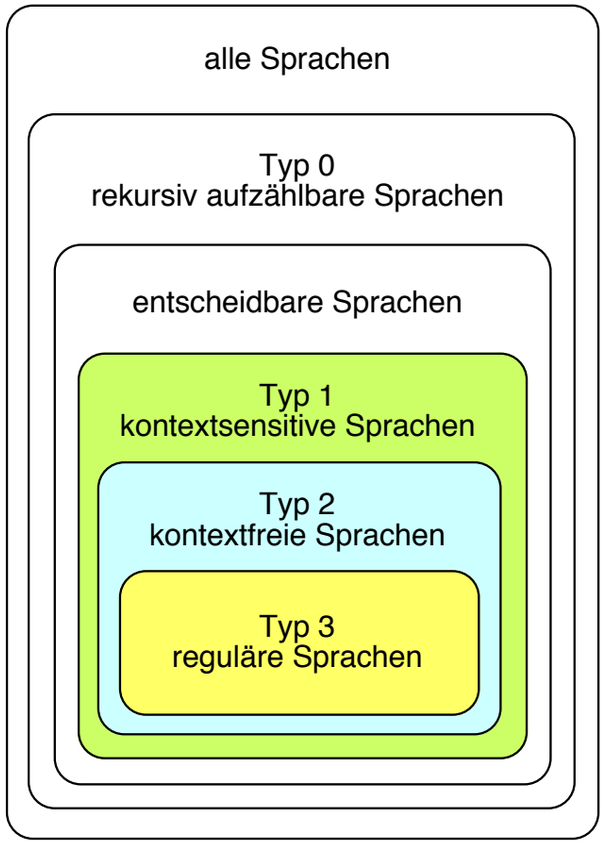
Automaten und formale Sprachen

- Formale Sprache
  - Alphabet $\sum$
  - $\sum^*$ Menge aller Worte über $\sum$
  - Formale Sprache ist Teilmenge von $\sum^*$
- Beispiel EXPR: *korrekt* geklammerte arithmetische Ausdrücke
  - $\sum$ = {(,),+,-,*,/,a} ; a kann Variable oder Konstante sein
  - (a-a)*a+a/(a+a)-a EXPR
  - ((((a+(a))))) EXPR
  - ((a+)-a(EXPR
  - wann ist ein Ausdruck w korrekt geklammert, also w EXPR?
- Grammatik ist 4-Tupel G=(V,$\sum$, P, S)
  - V endliche Menge von Variablen
  - $\sum$ endliche Menge von Symbolen (Terminalsymbole)
  - $V \cap \sum = \emptyset$
  - P Produktionen (Regeln)
  - $S \in V$ ist die Startvariable

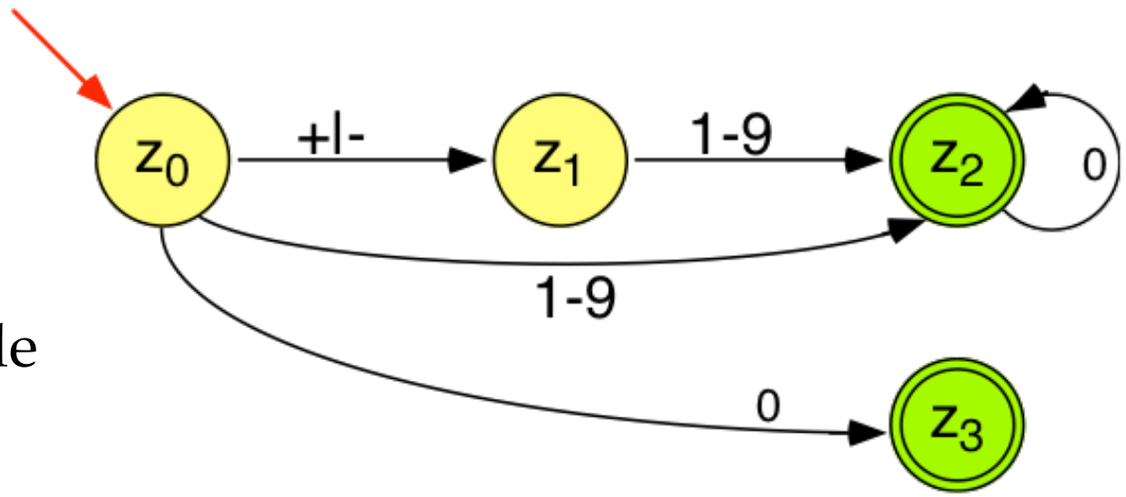
- Produktionen
  - $u, v \in (V \cup \sum)^*$
  - Relation $u \Rightarrow_G v$
  - $u = xyz$, $v = xy'z$; $x,z \in (V \cup \sum)^*$
  - $y \rightarrow y' \in P$
- Sprache $L(G) = \{w \in \sum^* \mid S \Rightarrow^*_G w\}$
- Ableitung von w_n
  - Folge $(w_0, w_1, w_2, \ldots, w_n)$
  - $w_0 = S$, $w_n \in \sum^*$
  - $w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$
  - nichtdeterministisch
- Ausdruck korrekt geklammert?
  - Ableitung existiert für Grammatik G
  - G = ({E,T,F}, {(,),a,+,*}, P, E)
  - P = { E → T, E → E+T,
    T → F, T → T*F,
    F → a, F → (E) }
  - a*a*(a+a)+a ∈ L(G) ?

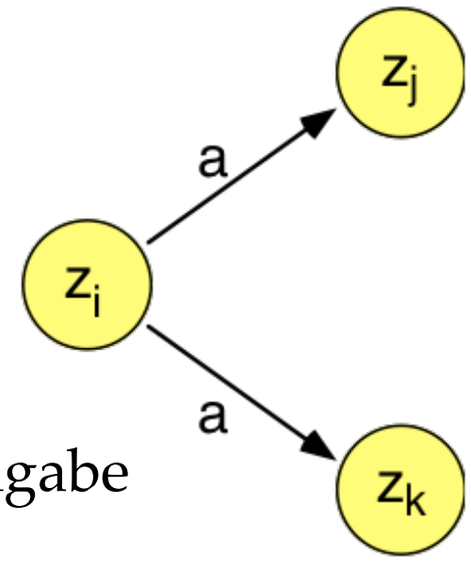


- Grammatik-Typen
  - Definition: jede Grammatik vom Typ 0
  - Typ 1 kontextsensitiv: $\forall\ w_1 \rightarrow w_2 \in P$ gilt $|w_1| \leq |w_2|$
  - Typ 2 kontexfrei: $\forall\ w_1 \rightarrow w_2 \in P$ gilt $w_1 \in V$
  - Typ 3 regulär: Typ 2 und $w_2 \in \sum \cup \sum V$
    (w_2 Terminalsymbol(+Variable))
- Chomsky-Hierarchie
  - Sprache L vom Typ x
    $\exists$ Grammatik G vom Typ x mit L(G) = L
  - kontextsensitiv: uAv → uxv
  - kontextfrei: auch ohne 'passenden' Kontext ersetzen
  - Syntaxanalyse für Programmiersprachen
  - Spezialklassen zwischen Typ 2 und 3: LL(k) und LR(k)
- Entscheidbarkeit
  - $\exists$ Algorithmus, der in endlicher Zeit feststellt, ob $w \in L(G)$
  - Typ 1,2,3 entscheidbar
  - Es gibt Typ 0 Sprachen, die nicht entscheidbar sind



- Automaten
  - akzeptieren ein Wort
  - Menge aller akzeptierten Wörter: Sprache
- Deterministischer endlicher Automat (DFA)
  - $M = (Z, \sum, \delta, z_0, E)$
  - Z endliche Menge der Zustände
  - $\sum$ endliches Eingabealphabet, $Z \cap \sum = \varnothing$
  - Überführungsfunktion δ: $Z \times \sum \rightarrow Z$
  - $z_0 \in Z$ ist Startzustand, $E \in Z$ Endzustände
- Darstellung typisch als Graph
  - gerichtet, beschriftet
  - Knoten sind Zustände
  - Pfeil auf Eingangsknoten
  - akzeptierende Endknoten als Doppelkreis
  - Kanten von z1 nach z2 mit a beschriftet: $\delta(z_0,a) = z_1$
- Durch DFA erkennbare Sprachen sind regulär (Typ 3)
- Nichtdeterministischer endlicher Automat (NFA)
  - Zustandsübergang in verschiedene Folgezustände bei gleicher Eingabe

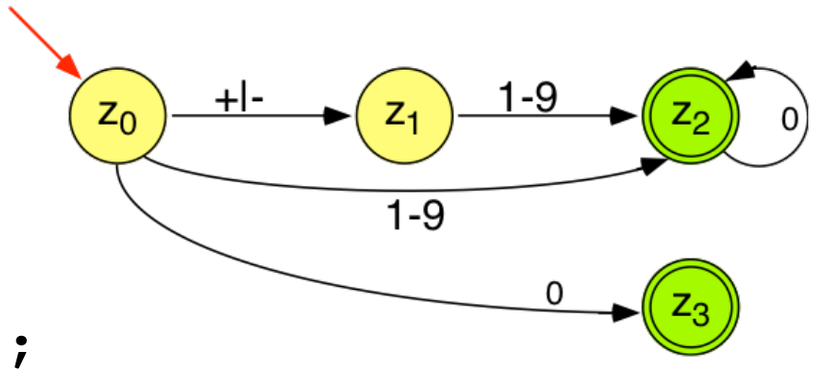




- Tabellenbasierte Implementierung von Automaten

z	0'	1'	2'	3'	4'	5'	6'	7'	8'	9'	+'	-'	…
0	3	2	2	2	2	2	2	2	2	2	1	1	-1
1	-1	2	2	2	2	2	2	2	2	2	-1	-1	-1
2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- Initialisierung der Tabelle




```
void tabinit(int tab[4][128])
{  int i,j = 0;
   for (i=0; i< 4; i++)
      for (j=0; j< 128; j++) tab [i][j]= -1;
   for (i='0'; i<='9'; i++) tab[0][i] = 2;
   tab[0]['+']= 1; tab[0]['-']= 1;
   tab[0]['0'] = 3;
   for (i='1'; i<='9'; i++) tab[1][i] = 2;
   for (i='0'; i<='9'; i++) tab[2][i] = 2;}
```

```
#include <stdio.h>

/* function tabinit einsetzen */

int istEndzustand(int zustand)
   {return zustand ==2 || zustand ==3;}

int main(void)
{  int zeichen, zustand = 0;
   int tabelle[4][128];

   tabinit (tabelle);

   while (zeichen=getchar()!='\n')
      if ((zustand = tabelle[zustand][zeichen])<0)
      break;

   if (istEndzustand(zustand))
      printf("akzeptiert\n");
      else printf("nicht akzeptiert\n");
   return 0;
}
```

- EBNF: erweiterte Backus-Naur-Form (ISO 14977)
  - kompakte Notation für kontextfrei Grammatiken
  - Regeln mit derselben linken Seite zusammenfassen

    $A \rightarrow \beta_1$

    $A \rightarrow \beta_2$

    …

    $A \rightarrow \beta_n$

    wird zu: $A ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  - $A ::= \alpha[\beta]\gamma$ steht für

    $A ::= \alpha\gamma \mid \alpha\beta\gamma$

    β *kann* einmal zwischen α und γ eingefügt werden
  - $A ::= \alpha\{\beta\}\gamma$ steht für

    $A ::= \alpha\gamma \mid \alpha B\gamma$

    $B ::= \beta \mid \beta B$

    β kann beliebig oft eingefügt werden
  - manchmal Minimum und Maximum durch Index

    $A ::= \alpha\{\beta\}^4\gamma$

  Gruppierung mit ()

$P = \{ \quad E \rightarrow T,$
$\quad E \rightarrow E+T,$
$\quad T \rightarrow F,$
$\quad T \rightarrow T*F,$
$\quad F \rightarrow a,$
$\quad F \rightarrow (E) \}$

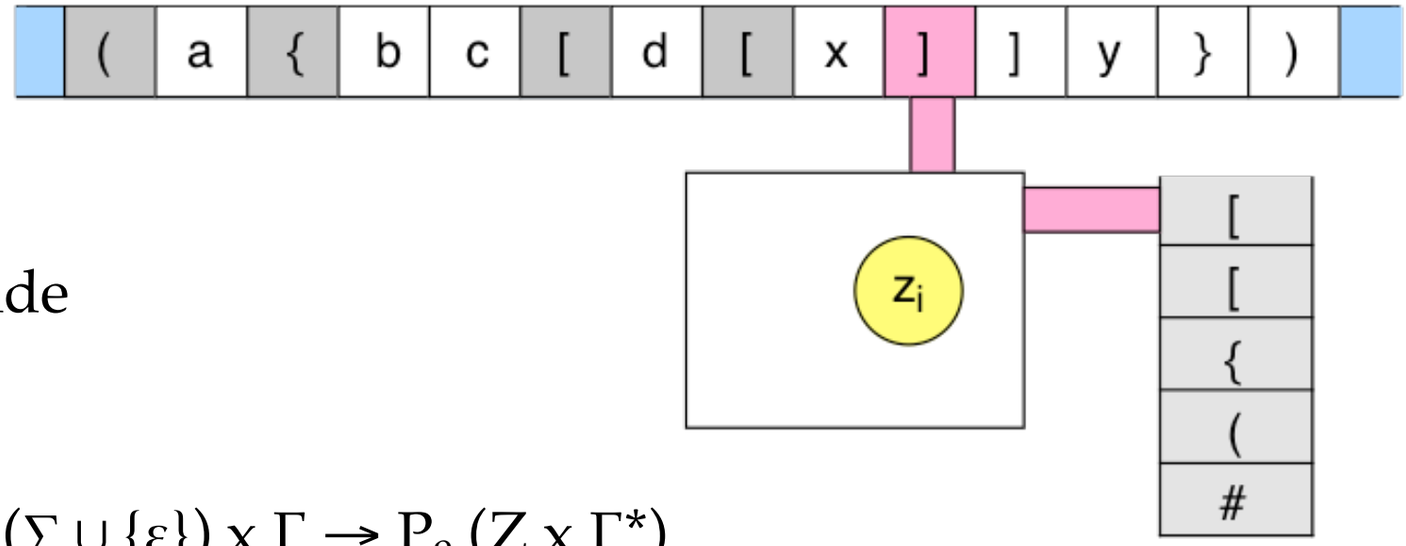
$P = \{ \quad E \rightarrow T \mid E+T,$
$\quad T \rightarrow F \mid T*F,$
$\quad F \rightarrow a \mid (E) \}$

- Beispiel einfache Programmiersprache [Wikipedia]

```
program ::= 'PROGRAM' ,
             wsp , ident , wsp,
            'BEGIN' , wsp,
            {assignment,";",wsp} ,
            'END.'
ident ::= alphachar , { alphachar | digit }
num ::= [ "-" ] , digit , { digit }
string ::= '"' , { anychar  } , '"'
assignment ::= ident , ":=" , ( num|ident|string )
alphachar ::= "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" ;
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
wsp ::= ? ASCII Character 32 ?
anychar ::= ? all visible characters minus " ?
```

```
PROGRAM DEMO1
BEGIN
  A0:=3;
  B:=45;
  C:=A;
  D123:=B34A;
  HANS:=WURST;
  TEXTZEILE:="Moin Moin";
END.
```

- Kontextfreie Sprachen (Typ 2)
  - werden nicht durch endlichen Automaten akzeptiert
  - Beispiel Klammerausrücke: Zählen der Klammern
  - Automat mit mehr 'Gedächtnis' als nur Zustand nötig



- Kellerautomat
  - $M = (Z, \sum, \Gamma, \delta, z_0, \#)$
  - Z endliche Menge der Zustände
  - $\sum$ Eingabealphabet
  - Γ Kelleralphabet
  - Überführungsfunktion : $Z \times (\sum \cup \{\varepsilon\}) \times \Gamma \rightarrow P_e (Z \times \Gamma^*)$
    (P_e ist Menge aller endlichen Teilmengen)
  - $z_0 \in Z$ ist Startzustand
  - $\# \in \Gamma$ das unterste Kellerzeichen
- PDA - Push Down Automaton
  - Keller ist Zusatzgedächtnis
  - nichtdeterministisch
  - akzeptiert genau die kontextfreien Sprachen
  - können zum Beispiel korrekt geklammerte Ausdrücke erkennen

Berechenbarkeit

- Intutive Berechenbarkeit
  - formale Definition von 'intuitiv berechenbar' schwer

> Eine Funktion f heisst berechenbar, falls ein Algorithmus (Programm) existiert, der (das) ausgehend von der Eingabe $(x_1, \ldots, x_n)$ in endlich vielen Schritte $f(x_1, \ldots, x_n)$ berechnet.

- Beispiele

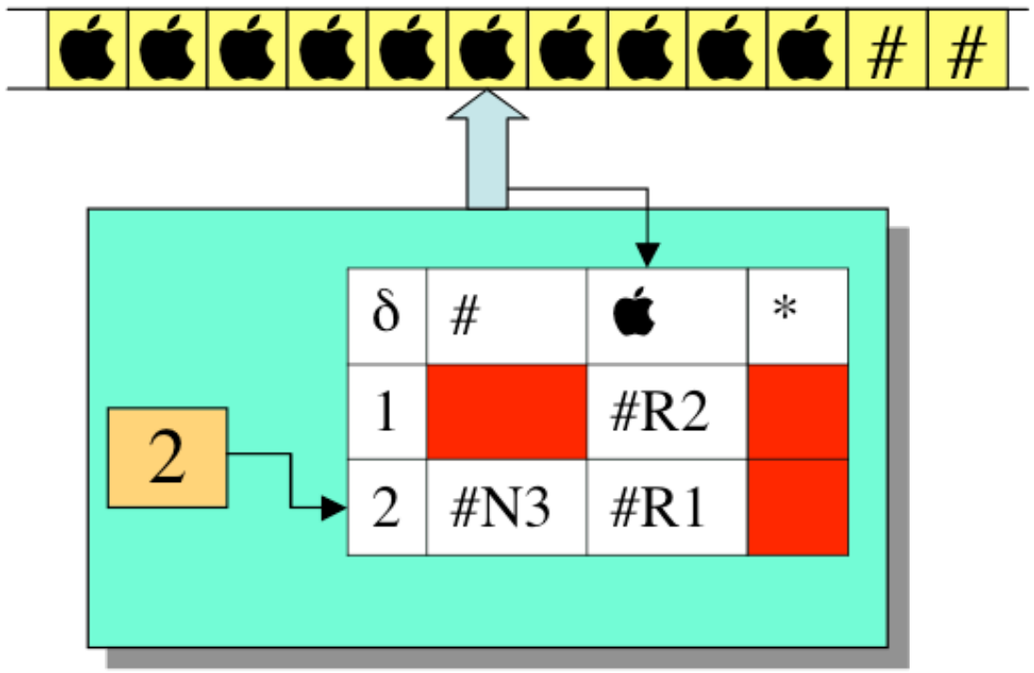
  $f(n) = 2*n$

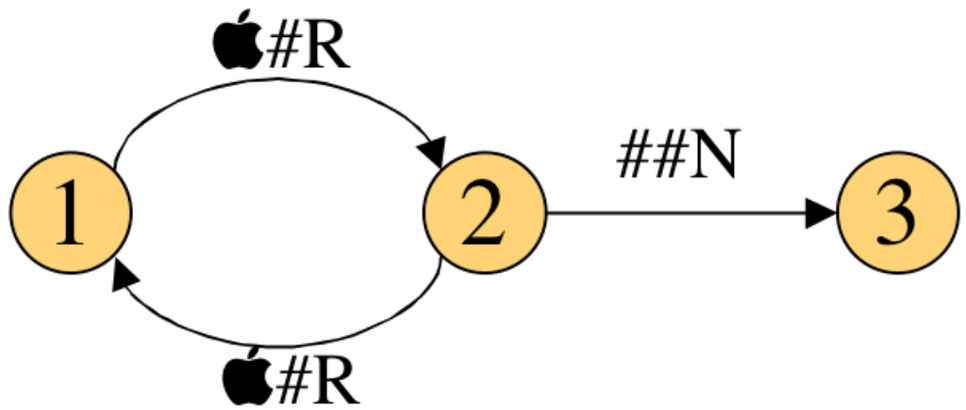
  $f_\pi(n) = 1$ falls n Anfang der Dezimalbruchentwicklung von π ist

  sonst 0

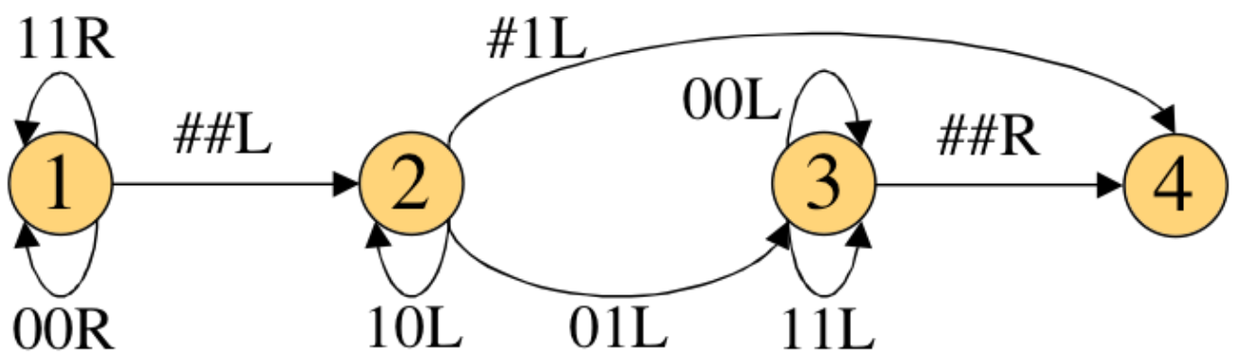
- Gibt es nicht berechenbare Funktionen?
  - ist für jede reelle Zahl r $f_r(n)$ berechenbar?
  - Nein: überabzählbar viele r (Cantorsches Diagonalschema …)
  - nur abzählbar viele Algorithmen
  - oder das Cantorsche Abzählschema modifizieren

- Turingmaschine [Alan Turing, 1936]
  - Automat
  - unbegrenztes Band mit Zeichen eines Arbeitsalphabets
  - Schreib/Lesekopf
  - Bandbewegung ein Feld nach rechts oder links
  - Zustand, gelesenes Zeichen und Regel
  - Zustandsübergang, Zeichen schreiben, Band bewegen





- Formale Beschreibung
  - Z endliche Menge von Zuständen
  - Γ Arbeitsalphabet
  - ∑ ⊂ Γ Eingabealphabet
  - Überführungsfunktion δ
    - δ: Z x Γ → Z x Γ x {L,R,N} (determinsitische TM)
    - δ: Z x Γ → *P*(Z x Γ x {L,R,N}) (nicht-determinsitische TM)
  - z_o Startzustand
  - # ∈ Γ -∑: Blank
  - E ⊆ Z Menge der Endzustände
- Beispiel: Binäre Addition von 1
- Turing-Vollständigkeit
  - ein Computer, der alle mit Turing-Maschinen berechenbaren Funktionen berechnen kann (Bsp: zellulärer Automat Rule 110)

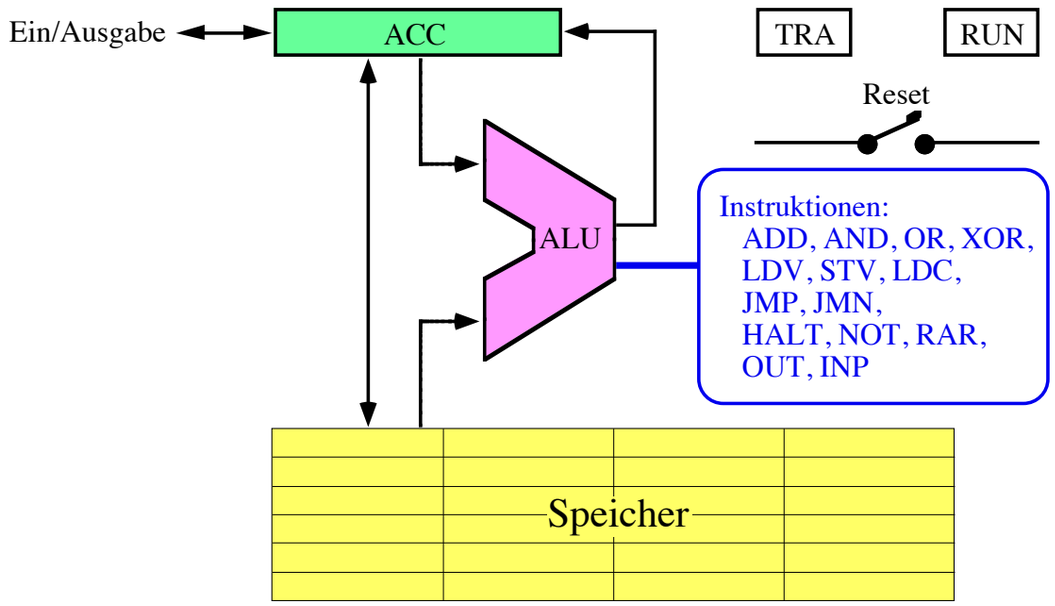


- Turingmaschinen akzeptierten Sprachen
  - $T(M) = \{x \in \sum^* \mid z_0x \rightarrow^* \alpha z \beta; \alpha, \beta \in \Gamma^*; z \in E\}$
  - allgemeine TM akzeptieren Typ 0 Sprachen
- Linear beschränkte Turing-Maschine: endliches Band
  - akzeptieren kontextsensitive Sprachen (Typ 1)
- Turing-Berechenbarkeit
  - $f : N \rightarrow N$ Turing-berechenbar falls ∃ TM
  - ∀ $n_1, \ldots, n_k, m \in N$ gilt
  - $f(n_1, \ldots, n_k) = m$ genau dann wenn $z_0n_{1b}n_{2b}\ldots n_{kb} \rightarrow^* \#\ldots\# z_e m_b \#\#\#$
- Churchsche These [Alonzo Church, 1936]

> Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funk-

  - eine These, von deren Richtigkeit fast alle Informatiker überzeugt sind
  - wahrscheinlich, hilfreich, 'funktioniert' gut

- Registermaschinen
  - RM oder auch RAM
  - unendlich viele Registerzellen (~Speicher)
  - Akkumulator und Einadressbefehle
  - Load, Store, Arithmetik, Sprünge
  - Adressierung: Konstante, direkt, indirekt
  - ähnlich MIMA
  - intuitiv: kann alle intuitiv berechenbaren Funktionen berechnen
  - kann Turingmaschinen simulieren



- While Berechenbarkeit
  - kann Touringmaschine simulieren und umgekehrt

```
Prog::= id := ausdruck
       | IF bed THEN Prog ELSE Prog
       | WHILE bed DO Prog DONE
       | Prog; Prog
bed::=     ausdruck <= ausdruck
ausdruck::=   id| 0 | succ(ausdruck)
```

- LOOP Berechenbarkeit schwächer

- Primitiv rekursive Funktionen
  - konstante Funktionen sind primitiv rekursiv
  - Identität und Nachfolgerfunktion primitiv rekursiv
  - $f(0, \ldots) = g(\ldots)$
  - $f(n+1, \ldots) = h(f(n, \ldots), \ldots)$
  - äquivalent zu LOOP-Berechenbarkeit
- Arithmetik als primitiv rerkursive Funktionen
  - add(0,x) = x /* Identität */
  - add(n+1,x) = succ(add(n,x))
  - mult(0,x) = 0
  - mult(n+1,x) = add(mult(n,x),x)
- μ-Rekursion
  - $g(x_1,\ldots,x_k) = \min\{n \mid f(n,x_1,\ldots,x_k) = 0, m<n\ f(m,x_1,\ldots,x_k)$ definiert$\}$
  - terminiert evtl. nicht

Die Klasse der μ-rekurisven Funktionen stimmt genau mit der Klasse der While-, RAM- und Touring-berechenbaren Funk-

- Ackermann-Funktion
  - berechenbar, nicht primitiv rekursiv
  - ack(0,n) = n+1
  - ack(m,0) = a(m-1,1) ;m>0
  - ack(m,n) = ack(m-1,a(m,n-1)); m,n>0

m/n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$265536-3$	a(3,(4,2))	a(3,a(4,3))

- Halteproblem
  - existiert ein Programm, das für jedes Paar von TM und Eingabe berechnet, ob die TM auf der Eingabe anhält?
  - Gegenbeispiel von Turing, 1936
  - semi-entscheidbar: TM hält entweder oder läuft endlos

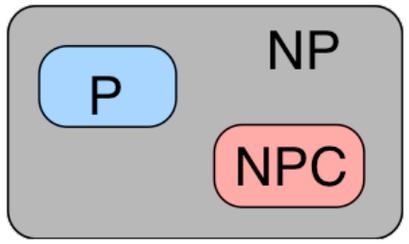
Jedes Beweissystem für die Menge der wahren arithmetischen Formeln ist notwendigerweise unvollständig

- Gödelscher Unvollständigkeitssatz [K. Gödel, 1931]
  - es blieben immer wahre arithmetische Formeln, die nicht beweisbar sind
  - in jedem logischen System existieren Aussagen, die weder bewiesen noch widerlegt werden können

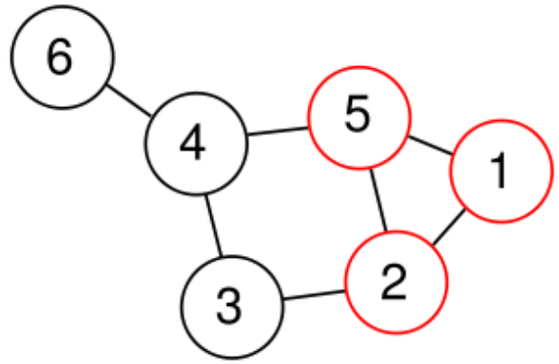
Komplexitätstheorie

- Berechenbar ist nicht unbedingt 'praktisch' lösbar
  - Ressourcen-Verbrauch: Rechenzeit, Speicher
  - stark wachsende Probleme
- Problem der Klasse P
  - in polynominaler Zeit lösbar
  - Suchen in Listen mit n Elementen O(n)
  - Sortieren O(n log n)
  - Sieb des Erathosthenes $O(n^2)$
- SAT-Problem (Satisfiability)
  - gegeben boolsche Gleichung g
  - SAT findet eine Variablenbesetzung x, so daß g(x) = true
  - $O(2^n)$
- Klasse NP
  - nichtdeterministisch polynomiale Zeit
  - polynomial entscheidbar, ob Kandidat Lösung ist
  - exponentielle Komplexität
  - jedes Problem in NP lässt sich auf SAT-Problem polynomial zurückführen

- NP-vollständig (NPComplete)
  - p ist NPC, falls SAT auf p zurückführbar ist
  - alle 'gleich schwer'
  - Tausende NPC-Problem bekannt
  - P=NP???



- Cliquen-Problem
  - Clique = Teilmenge von Knoten in Graphen paarweise verbunden
  - existiert k-Clique in einem Graphen?
- Hamilton-Problem
  - ein Weg der jeden Knoten genau einmal berührt?
  - Bsp: Springerproblem im Schach



- TSP - Travelling Salesman Problem
  - kürzester Hamilton-Kreis?
- Rucksack-Problem
- Anwendungen
  - Kryptographie, Optimierung
  - Stundenplanproblem, Verdrahtung in Chips und Platinen