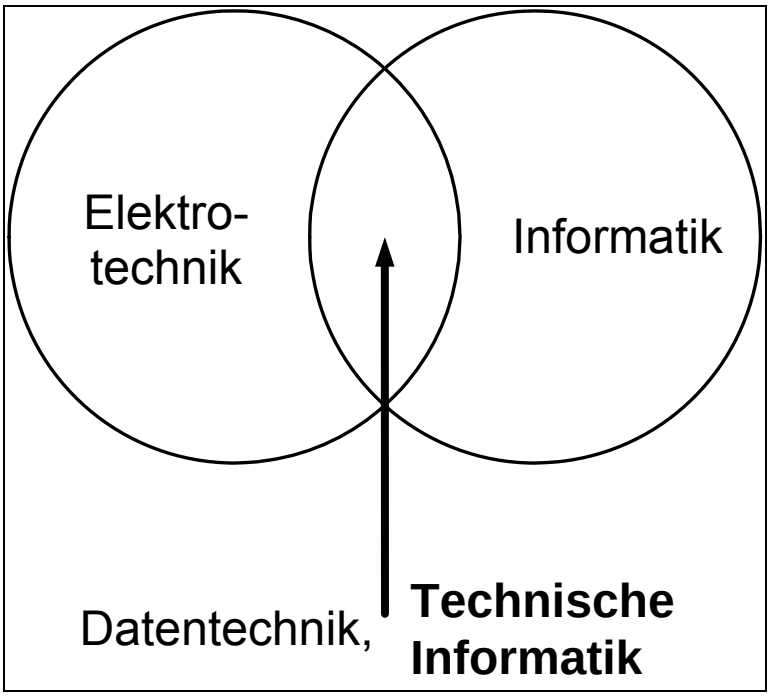
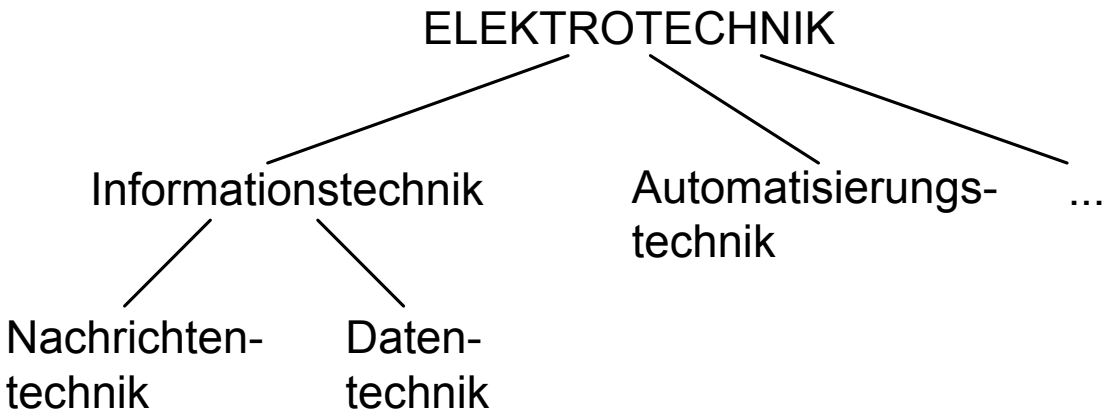
1. EINFÜHRUNG

1.1 Einleitung

Was ist Technische Informatik?

Einordnung im Überlappungsbereich zwischen Elektrotechnik und Informatik





Nachrichtentechnik:
Schwerpunkt auf der *Codierung* und *Übertragung* von Information (Nachrichten)

Datentechnik/Technische Informatik:
Schwerpunkt auf der *Speicherung* und *Verarbeitung* von Information (Daten)

***Informatik* ist**
die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.

Informatik ist als eine umfassende *Basis- und Querschnittsdisziplin* zu verstehen, die sich sowohl mit technischen als auch mit organisatorischen und sozialen Phänomenen und Problemen bei der Entwicklung und Nutzung informationsverarbeitender Systeme beschäftigt [GI 1989].

Informatik *umfasst*

- Theorie
- Methodik
- Analyse und Konstruktion
- Anwendung
- Auswirkung des Einsatzes

von informationsverarbeitenden, insbesondere computergestützten Systemen [GI 1989].

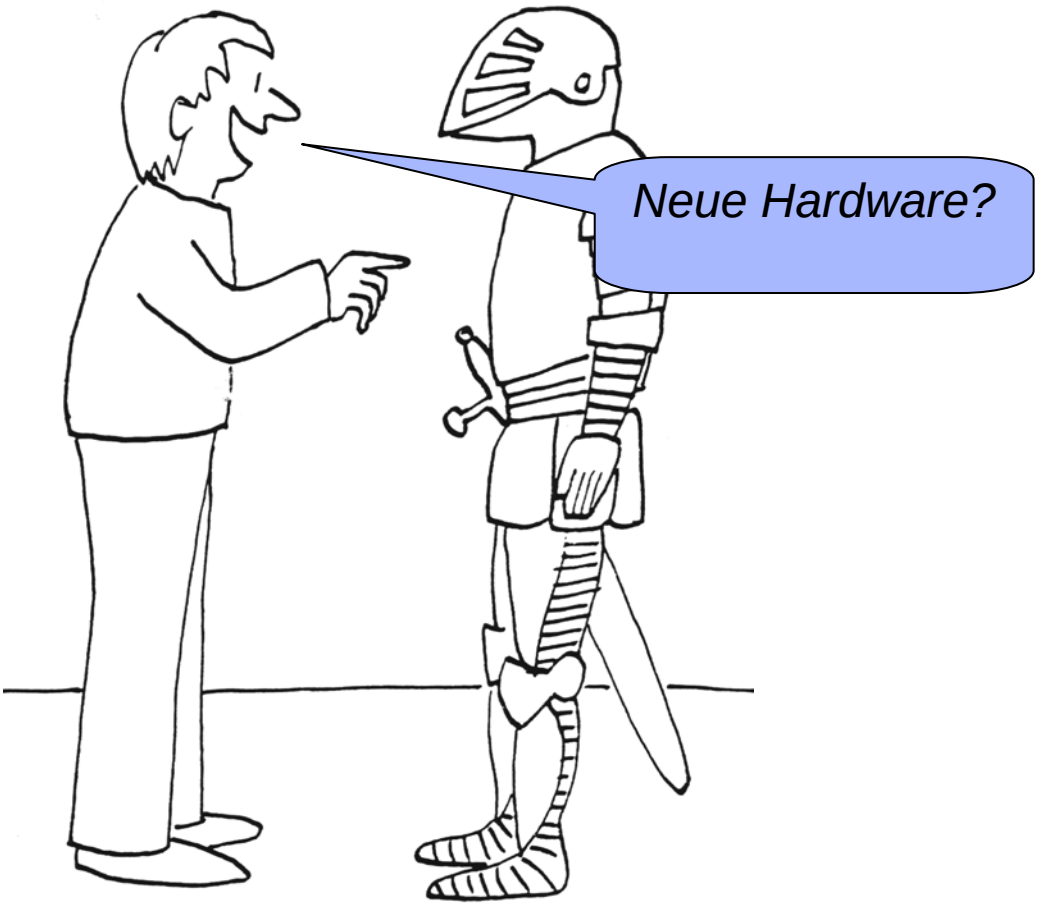
Rechner "optimal" für eine Zielanwendung zu konstruieren oder auch nur einzusetzen, wird in zunehmendem Maße in den Aufgabenbereich von Informatikern fallen. Ein tieferes Verständnis der *Arbeitsweise und Architektur von Rechnern* ist damit unerlässlich [Becker2005].

Was ist „*Technische Informatik*“?

Die ***Technische Informatik*** beschäftigt sich als eines der Hauptgebiete der Informatik mit der Architektur, dem Entwurf, der Realisierung, der Bewertung und dem Betrieb von Rechner-, Kommunikations- und eingebetteten Systemen sowohl auf der Ebene der Hardware als auch der systemnahen Software. [Wikipedia]

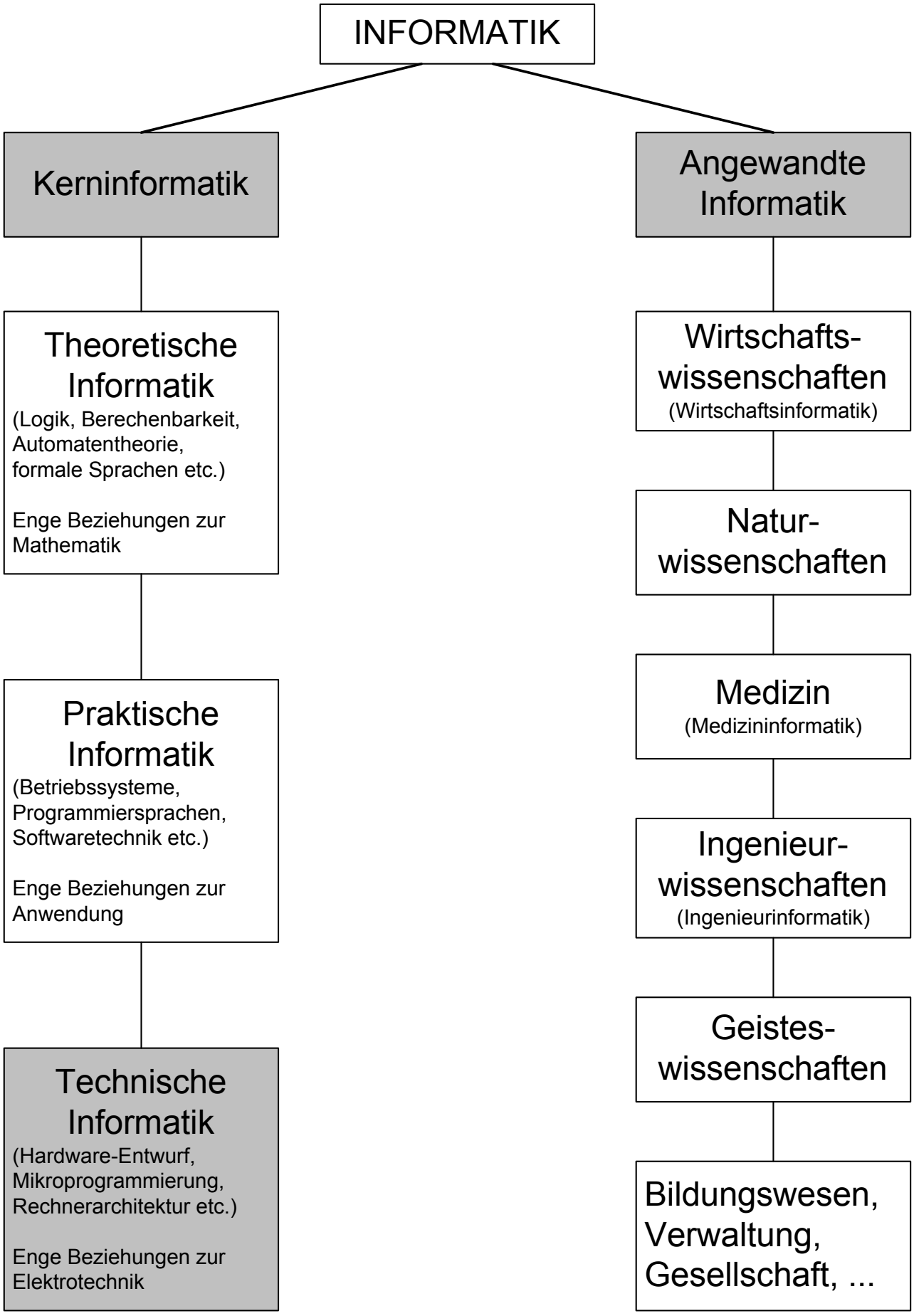
Gegenstände der Technischen Informatik

- *Hardware*
- hardwarenahe Software
- Organisationsstrukturen von Rechenanlagen
- Entwicklung und Einsatz von Rechenanlagen



„Neue Hardware?“

Quelle: Computer Cartoons von Helmut Schreiner, Verlagsgesellschaft Rudolf Müller 1979


INFORMATIK
Kerninformatik
Angewandte Informatik
Theoretische Informatik
(Logik, Berechenbarkeit, Automatentheorie, formale Sprachen etc.)
Enge Beziehungen zur Mathematik
Praktische Informatik
(Betriebssysteme, Programmiersprachen, Softwaretechnik etc.)
Enge Beziehungen zur Anwendung
Technische Informatik
(Hardware-Entwurf, Mikroprogrammierung, Rechnerarchitektur etc.)
Enge Beziehungen zur Elektrotechnik
Wirtschafts-wissenschaften
(Wirtschaftsinformatik)
Natur-wissenschaften
Medizin
(Medizininformatik)
Ingenieur-wissenschaften
(Ingenieurinformatik)
Geistes-wissenschaften
Bildungswesen, Verwaltung, Gesellschaft, ...

Hauptaufgabengebiet der Technischen Informatik:

Entwurf und Realisierung von informationsverarbeitenden *Systemen* mit besonderer Betonung der *Hardware-Aspekte* (oberhalb der Ebene der Schaltungstechnik und Halbleitertechnologie)

Schnittstellen zu Gebieten der Elektrotechnik wie Schaltungstechnik, Automatisierungstechnik, Messtechnik, Energietechnik usw. sowie zur praktischen, theoretischen und angewandten Informatik

Hardwarenahe Anwendungen wie beispielsweise eingebettete Echtzeitsysteme, Echtzeitbetriebssysteme, fehlertolerante Rechensysteme, Robotik, …

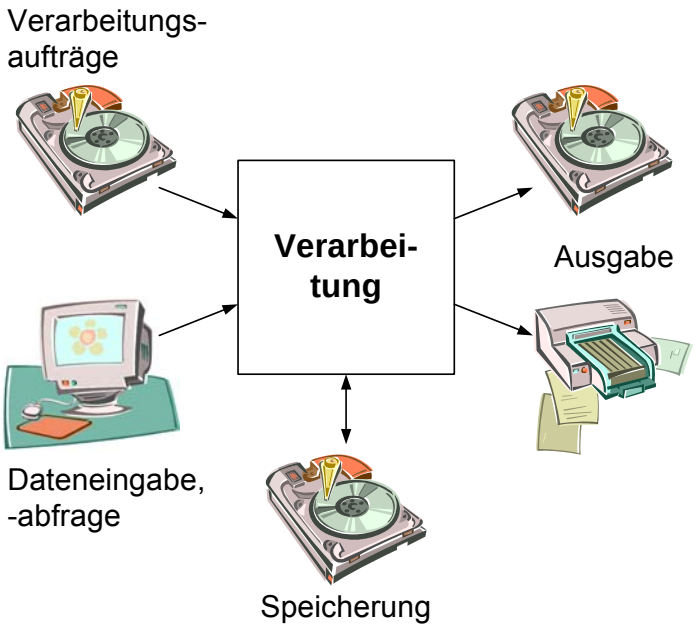
Entwurfswerkzeuge („Tools“) zur effektiveren Ausnutzung von Hardware, speziellen Methoden oder Rechnerarchitekturen

1.2 Anwendungsaspekte

EDV, kommerzielle Datenverarbeitung
(traditionell Großrechner):

- regelmäßige oder sofortige Verarbeitung, z.B.:
  Auskunftssysteme, Suchmaschinen, ...
- kaufmännische und betriebliche Daten:
  Bestellungen, Rechnungen, Mahnungen, Lagerbestände, Produktionsplanung, Lohnabrechnungen, ...

-> hohe Anforderungen, teure Maschinen.



Technisch-wissenschaftliche Anwendungen
(Workstations, Parallelrechner):

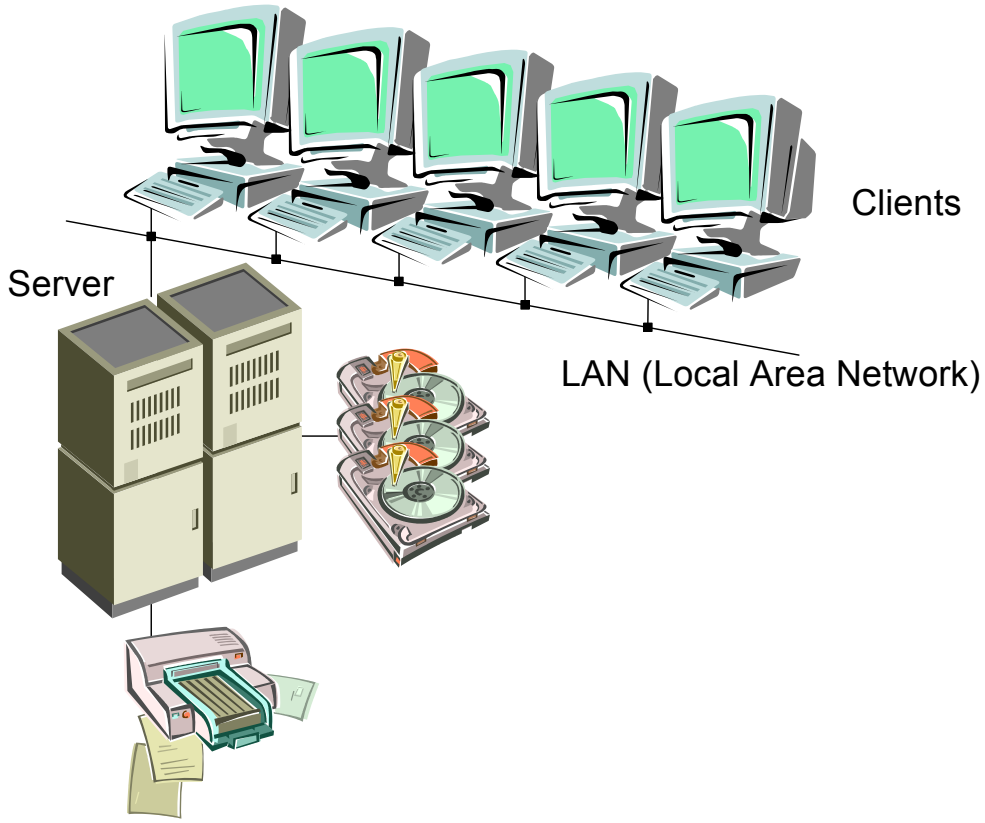
- Chemische und physikalische Modelle, Simulationen (Wetter, Crash, Strömung, …), Optimierung.
- Hoher Anteil der Programmentwicklung.
- Rechnergestützte Konstruktion und Fabrikation (CAD = Computer Aided Design, CIM = Computer Aided Manufacturing)
- Bioinformatik (Gensequenzierung, Drug Design)

-> interessante Probleme, aber kleiner Markt.

Bürokommunikation
(Vernetzte PCs, Client/Server-Systeme):

Textverarbeitung, Briefe, Berichte, Bücher, elektronische Post (E-Mail), Intranet, Vorlesungsunterlagen etc.

-> riesiger Markt, hoher Kostendruck.



<u>Öffentliche Kommunikationsdienste:</u>

ISDN, DSL, Datenbanken, Zahlungsverkehr

Weltweite E-Mail- und Informationsdienste wie Telekommunikation und Internet

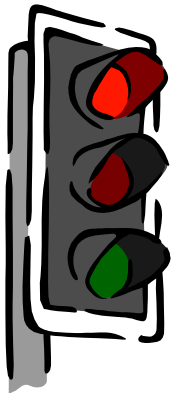
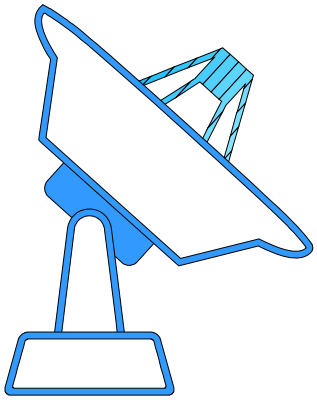
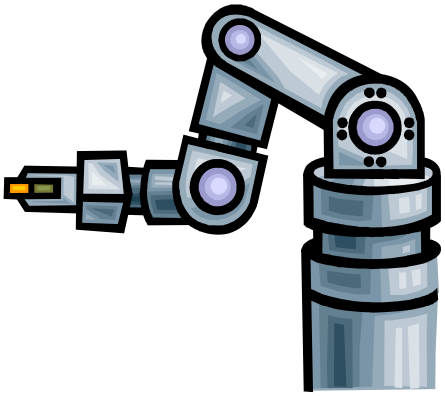
Mobile Kommunikation über Handies (GSM, WAP, UMTS) oder Smartphones und Notebooks (GPRS, WLAN, UMTS)

-> weit gefächerte Probleme und Markt.

Automatisierungstechnik (Prozesssteuerungen, Echtzeitsysteme, Eingebettete Systeme)
(Mikrocontroller, Speicherprogrammierbare Steuerungen, Industrie-PCs, Prozessrechner, oft hierarchisch vernetzte Systeme)

Fertigungsanlagen, Hochregallager, Kraftwerke, Erdölraffinerien, Marschflugkörper, Frühwarnsysteme, Signalanlagen für den Verkehr, Roboter, ...

→ sehr hohe Anforderungen an das Zeitverhalten und die Betriebssicherheit.

Aufgrund der *Echtzeitanforderungen* in der Regel spezielle Betriebssysteme und Programmierumgebungen.

Mikrocontroller bzw. Prozesssteuerungen sind heute in fast allen fortgeschrittenen technischen Systemen zu finden (Waschmaschinen, Videorecordern, Autos, Flugzeugen, Lokomotiven, Aufzügen, Heizungsanlagen, ...).

Aktuell: Ubiquitäre Rechner, pervasive computing, d. h. als solche nicht sichtbare, i. Allg. vernetzte Computer, die in allen möglichen Gegenständen eingebaut sind (Haushaltsgeräte, Möbel, Kleidung (Smart Clothes), Internet der Dinge, Smart Home etc.)

Eingebettete Systeme

Die Mehrzahl der heute eingesetzten Mikroprozessoren befindet sich in eingebetteten Systemen, also in Anwendungen wie

- KFZ (bis über 100 Stück/Fahrzeug),

Beispiel: Bordnetz des VW Phaeton

- Avionik,

- Medizintechnik,

- mobile Geräte,

- Geräte für: Haushalt

Entertainment

Kommunikation

Aufzüge

Motorsteuerungen

Fabrikanlagen

. . .

1.3 Geschichtliches

Historische Entwicklung

Ca. 5000 v. Ch.	Grundlage des Rechnens ist das Zählen; Benutzen der zehn Finger; größere Zahlen mit Steinen, Perlen, Holzstäbchen.
1100 v. Ch.	Abakus (Suan Pan)
82 v. Ch.	Räderwerk von Antikythera (astronomisches Gerät und nautisches Hilfsmittel)
500 n. Ch.	Hindu-arabisches Zahlensystem mit den Ziffern 0 bis 9, ab ca. 1150 im Abendland
1623	Schickard konstruiert für Kepler eine Maschine, die mit 6-stelligen Zahlen +, -, x, / rechnen kann.
1624	Gunter entwickelt den Rechenschieber (logarith.)
1641-45	Pascal baut für seinen Vater (Steuerpächter) eine Addiermaschine mit 6-Stellen.
	... bei nahezu gleich bleibender Konzeption werden *mechanisch* arbeitende Rechenanlagen bis ins 20. Jahrhundert hinein stetig verbessert.
1679	Leibniz beschäftigt sich mit dem Dualsystem, das zur Grundlage heutiger Datenverarbeitungsanlagen (DVA) wurde.
1805	Jacquard setzt Kartons mit eingestanzten Webmustern zur automatischen Steuerung von Webstühlen ein („Lochkarten“).

1833 Babbage, Mathematik-Professor aus Cambridge, baut eine mechanische Rechenanlage (*difference engine*) zur Überprüfung von Tabellen.
Die Konzeption der geplanten "*analytic engine*" (= digitaler Rechenautomat) enthält alle Elemente moderner Datenverarbeitungsanlagen, Realisierung scheiterte aber am Stand der Technik.

- Speicher (1000 Worte à 50 Stellen)
- Rechenwerk
- Steuerwerk
- Ein- und Ausgabewerk
- Programm, gespeichert in Lochkarten (Flexibilität)

1886 Hollerith, USA, entwickelt elektrisch arbeitende Zählmaschinen für Lochkarten und benutzt sie für Statistiken (Volkszählung, …)

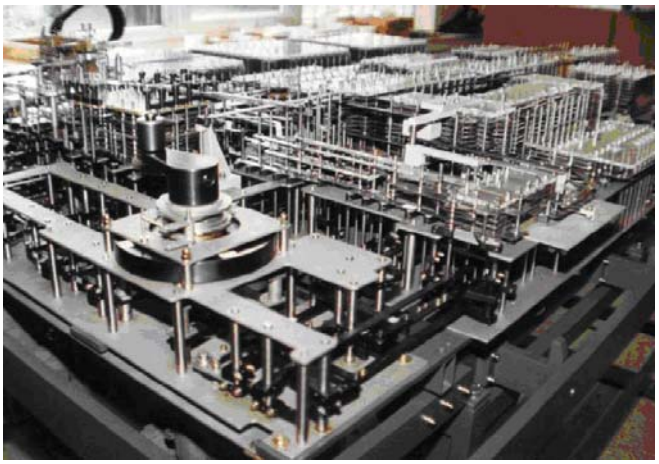
... Datenverarbeitung im kaufmännischen und Verwaltungsbereich werden als Markt erkannt

... Lochkartenmaschinen werden bis in die 1950er Jahre verfeinert und erfolgreich eingesetzt.

Parallel dazu Entwicklung von (programmierbaren) Automaten / Ablaufsteuerungen

Moderne Entwicklung

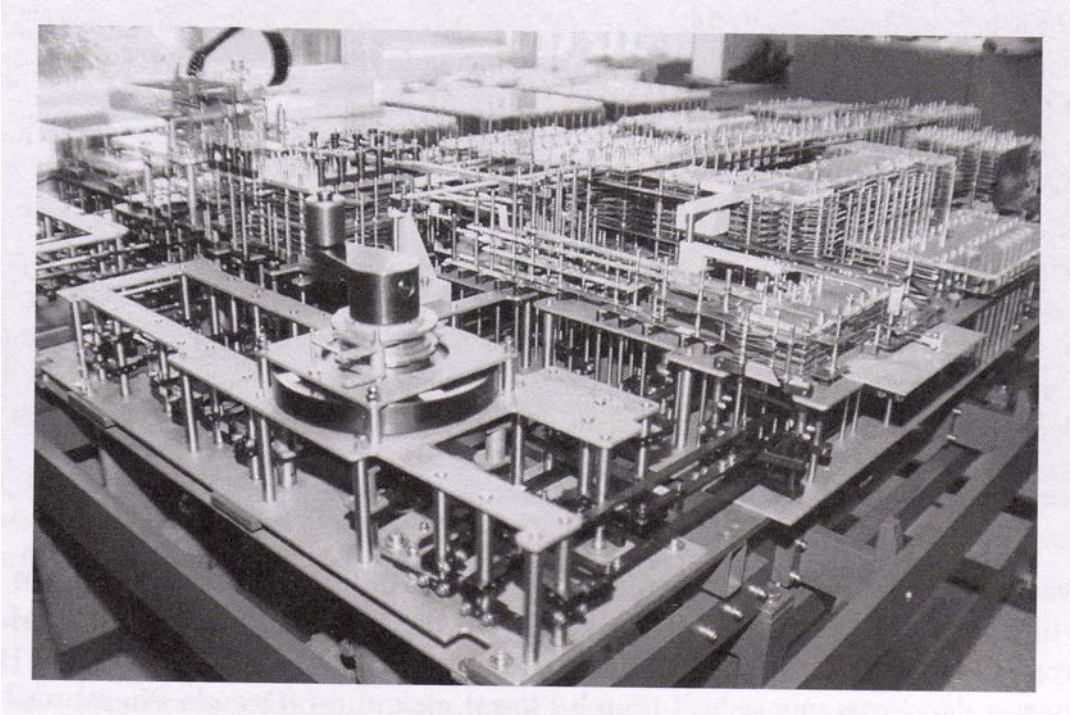
1936-1939	Konrad Zuse, Bauingenieur, beginnt noch während des Studiums mit dem Bau einer Datenverarbeitungsanlage, der *Z1* (elektro-mech. Rechner).
1941	Zuse baut die *Z3* als erste funktionsfähige, industriell nutzbare programmgesteuerte Rechenmaschine (Relaistechnik, 9 Instruktionen, kein bedingter Sprung; Nachbau im Deutschen Museum).
1944	Aiken, Harvard Universität, erstellt in Zusammenarbeit mit IBM die Großrechenanlage *Mark I* (Multiplikation 0,6 s).
1946/1947	Theoretische Arbeiten von Burks, Goldstine, von Neumann in Princeton bilden das grundlegende Konzept für elektronische Rechenanlagen
	... heutige moderne Rechner (vom Mikroprozessor bis zum Großrechner) arbeiten noch fast ausschließlich nach dem *von-Neumann-Prinzip.*



Nachbau der Zuse *Z1*

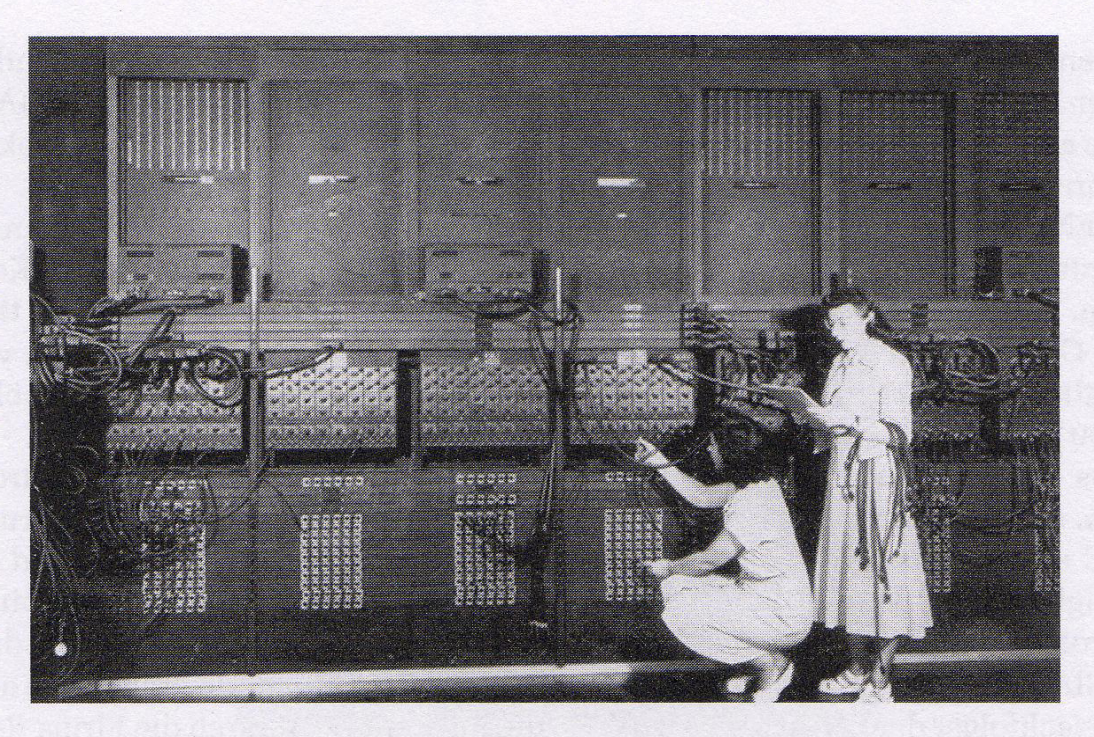
(im rechten Bild: vorne die Drehkurbel zur manuellen Taktung, links die Programmsteuerung mit 35 mm-Normalfilm, rechts das Gleitkommarechenwerk, links hinten die drei Speicherbänke;
aus Zusammenstellung von Horst Zuse über das Werk seines Vaters Konrad Zuse)

1.4 Technologischer Fortschritt



Zuse Z1 – 1938

(Elektromech. Rechner, 30.000 Einzelteile, 1 Hz, 1000 W)



ENIAC - 1946

(Röhrenrechner, 17468 Röhren, Addition in 0,2 ms, > 150 kW)

Generationen von elektronischen Rechnern

1946-1957 1. Generation der Datenverarbeitungsanlagen mit Elektronenröhren als Schaltelemente (Operationszeiten im ms-Bereich)

- ENIAC (30 Tonnen, 17.000 Röhren + 1500 Relais, zu 45 % der Zeit verfügbar)
- Z22, IBM 650

1957-1964 2. Generation der Datenverarbeitungsanlagen mit Transistoren (Operationszeiten im 100 µs-Bereich)

- IBM 1400er Serie, Siemens 2002, ...

1964-1974 3. Generation der Datenverarbeitungsanlagen mit Integrierten Schaltkreisen, Betriebssystemen, allgemeinen Dienstprogramme, Zentralrechner-, Familienkonzept (Operationszeiten im µs-Bereich))

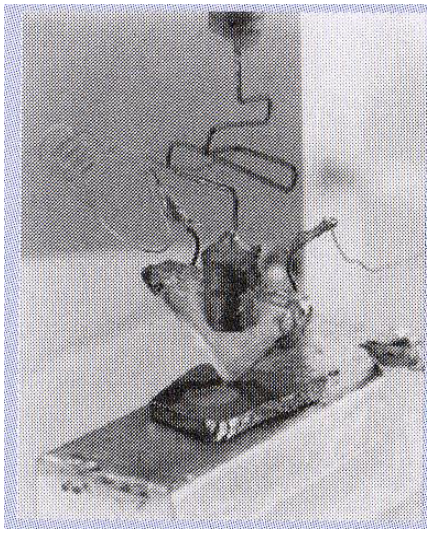
CDC - 3000, IBM 360, Siemens 4004, Univac 9000, ...

1975-... 4. Generation der Datenverarbeitungsanlagen mit Großintegration, Massenspeicher, Mehrprozessor-Architektur, Terminal Orientierung, … (Operationszeiten im ns-Bereich))

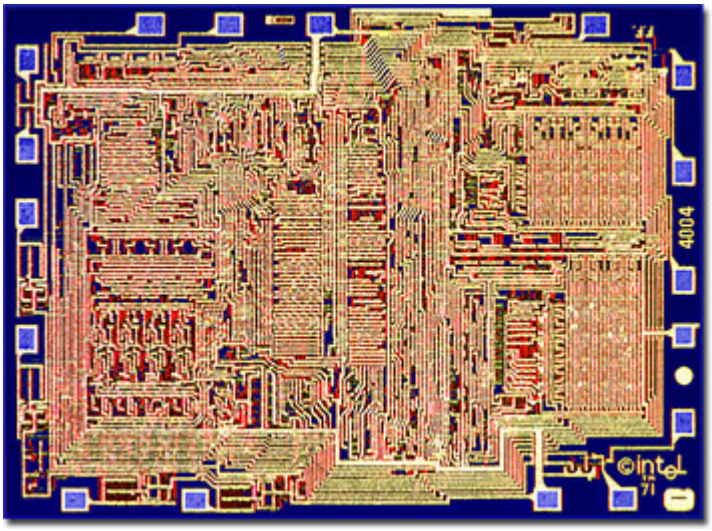
Borroughs, CDC Cyber, IBM 370, 3300, Siemens 7700, …

Parallel dazu Entwicklung von:
Personal Computing (PCs), eingebetteten Systemen, Parallel Computing, Mobile Computing, Grid Computing, Cloud Computing, Organic Computing, ubiquitäre Rechner, Bio-Computing, Quanten-Computer, Optical Computing…

??? 5. Generation der Datenverarbeitungsanlagen (Künstliche Intelligenz, kognitive Fähigkeiten, Benutzerkommunikation in natürlicher Sprache, …)

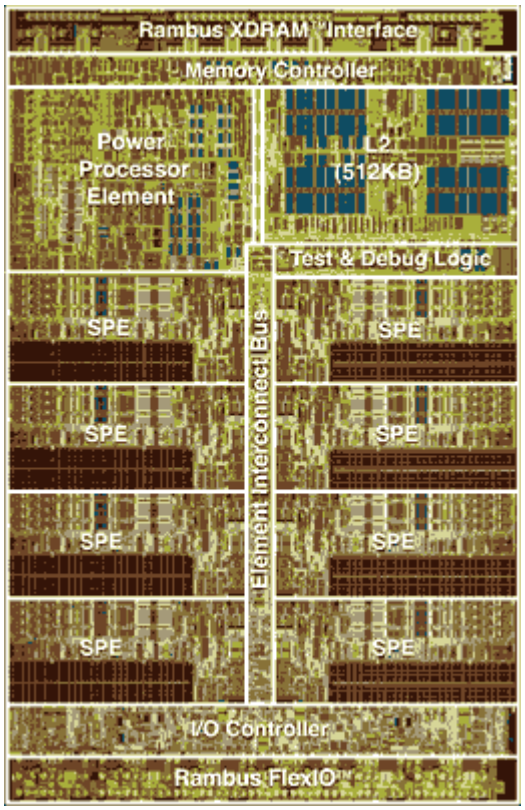


Erster Transistor
(1947)



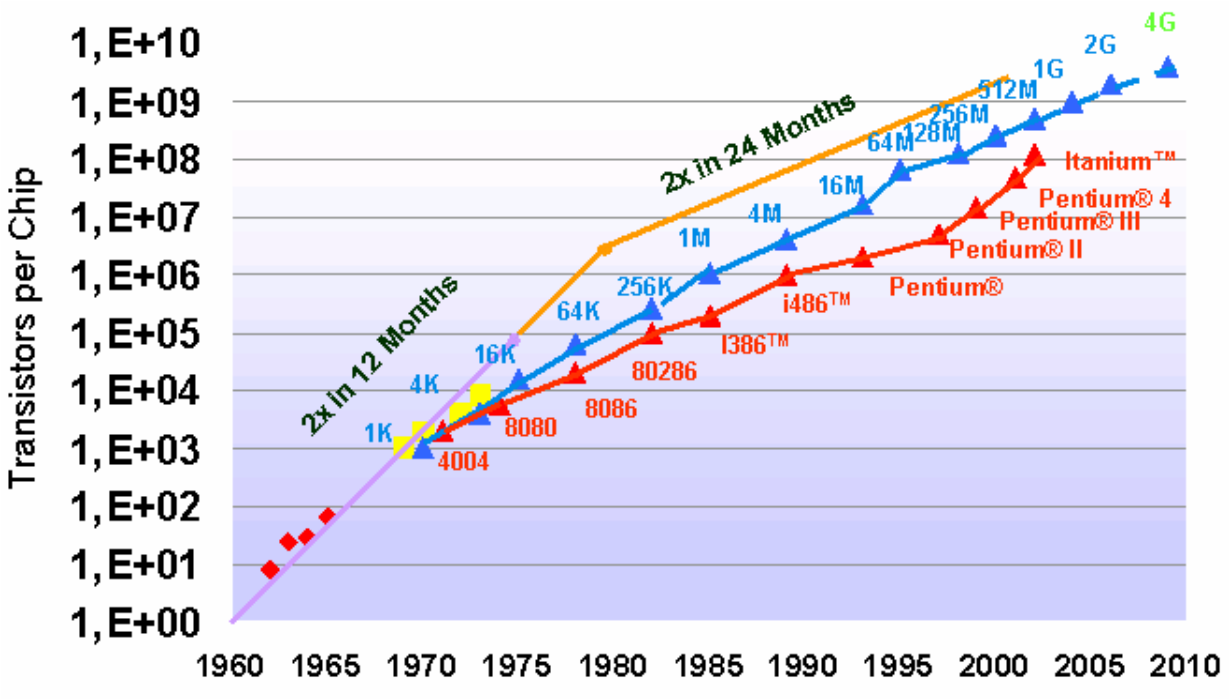
Erster Mikroprozessor Intel 4004
(1975, 4 bit, 2300 Trans., 108 kHz)

Intel Pentium 4
(2000, 32 bit,
42 Mio. Trans.,
1,5 GHz)



Cell-Prozessor
(2005, Hochleistungsmikroprozessor
und acht Grafik-Prozessoren;
235 Mio. Tr., 4 GHz, 256 GFLOPS)

Moore's Law:
Verdopplung der Transistoren pro Chip alle 18-24 Monate

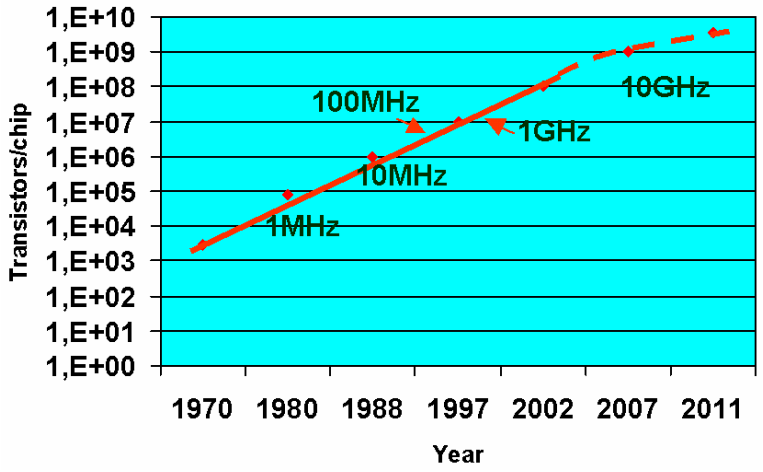


Beispielhafte Entwicklung einer Mikroprozessor-Familie:

Mikroprozessor (Beispiel Intel)	**Markteinführung**	**Anzahl Transistoren**
4004	1971	2.300
8008	1972	2.500
8080	1974	4.500
8086	1978	29.000
Intel´286	1982	134.000
Intel´386	1984	275.000
Intel´486	1989	1.200.000
Pentium	1993	3.100.000
Pentium II	1997	7.500.000
Pentium 4	2000	42.000.000
Itanium 2	2002	220.000.000
Itanium 2 mit 9 MB Cache	2004	592.000.000
Xeon MP X7460	2008	1.900.000.000
Paulson Itanium	2011	3.400.000.000

Entwicklung der Taktfrequenz

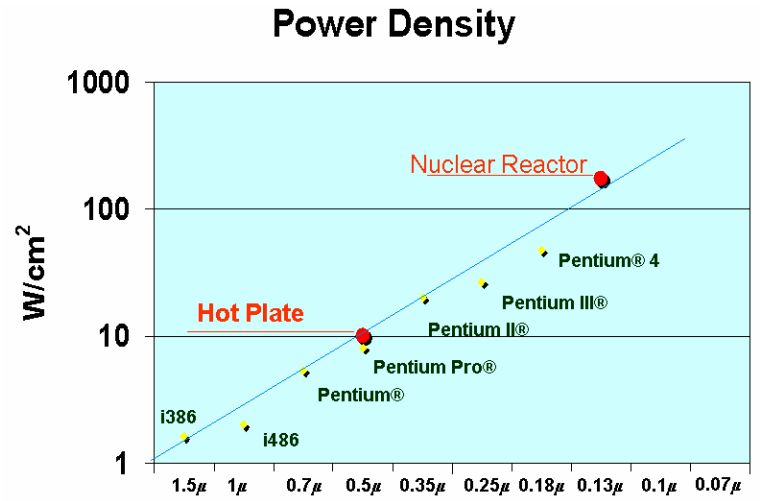
(kleinere Strukturen ⇒ höhere Frequenzen)



Zusammen mit Fortschritten bei den Rechnerarchitekturen Verdopplung der Rechenleistung alle 18 Monate (Moore's Law)

Wärmeproblem

(kleinere Strukturen ⇒ höhere Leistungsdichte)



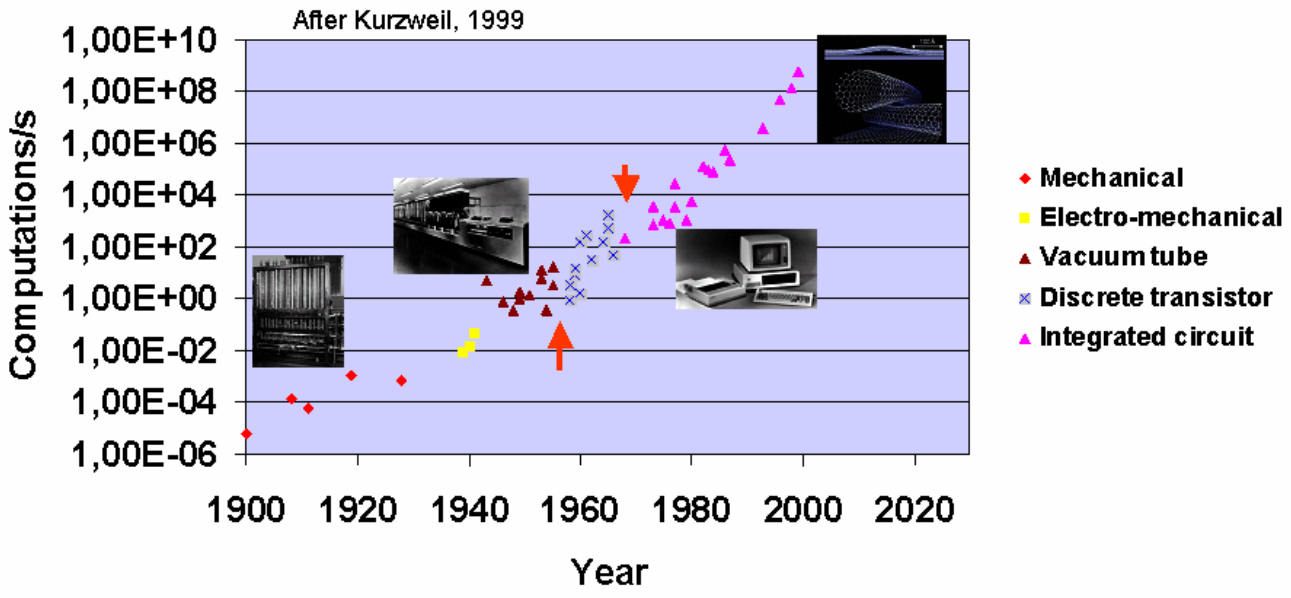
Grenzen von Moore's Law absehbar, aber noch längst nicht erreicht:

- Leistungsdichte
- Größe der Transistoren (Lithografie, Quanteneffekte)
- Lichtgeschwindigkeit

Fortschritte durch alternative Technologien wie Nanotubes, Quantencomputer, DNA-Computing, …?

Kostenentwicklung

Was man für 1000 $ kaufen kann ...



Wie viele TFLOPS (10^{12} Floating Point Operationen pro Sekunde) kann man für 1 Millionen $ kaufen …

Rechner	Jahr	Anzahl Prozessoren	TFLOPS
Illiac IV	1976	64	0,00000048
Cray-1	1976	1	0,00001778
Cray Y-MP	1988	8 (Vektor)	0,000115
ASCI RED	1997	4510	0,01818182
Earth Simulator	2002	5120	0,0175
Blue Gene/L	2004	65.536	2,8
Roadrunner	2008	19440	8,3
Jaguar - Cray XT5-HE	2010	18.688	2.300
Playstation-3 Cluster	2007	8 Playstations	375
Nvidia Tesla	2008	960 Cores	439,1

1.5 Relevante Aspekte für den Bau von Rechnern

Allgemein zu unterstützende Softwarekonstrukte

Daten:

- Konstanten
- Variablen, -werte
- Zahlendarstellungen
- Datentypen
    - Basistypen
    - zusammengesetzte Typen (Arrays, …)
    - dynamische Datenstrukturen (Keller, …)

Operationen:

- arithmetische und logische Operationen
- Abfragen
- logische Formeln

Programmausführung:

- Ablauf
- Verzweigungen
- Schleifen
- Unterprogramme
    - Aufruf, Rücksprung
    - Parameterübergabe
    - Rekursion
- Eingabe, Ausgabe
- externe Ereignisse und Reaktionen darauf

Relevante Fragestellungen:

Hochsprachenunterstützung

- durch die Rechnerhardware bzw. ISA-Schnittstelle
  - → zur Verfügung gestellte
    - Operationen, Befehlssatz, Maschinensprache
    - Adressierungsarten für Operandenzugriff
    - Programmiermodell

- durch die Abbildung auf Maschinensprache
  - → Kompiler → siehe *Kompilerbau*

Struktur und Funktion von Rechnern („Architektur“)

Struktur ist die Art und Weise wie die Komponenten zu einander in Beziehung stehen.

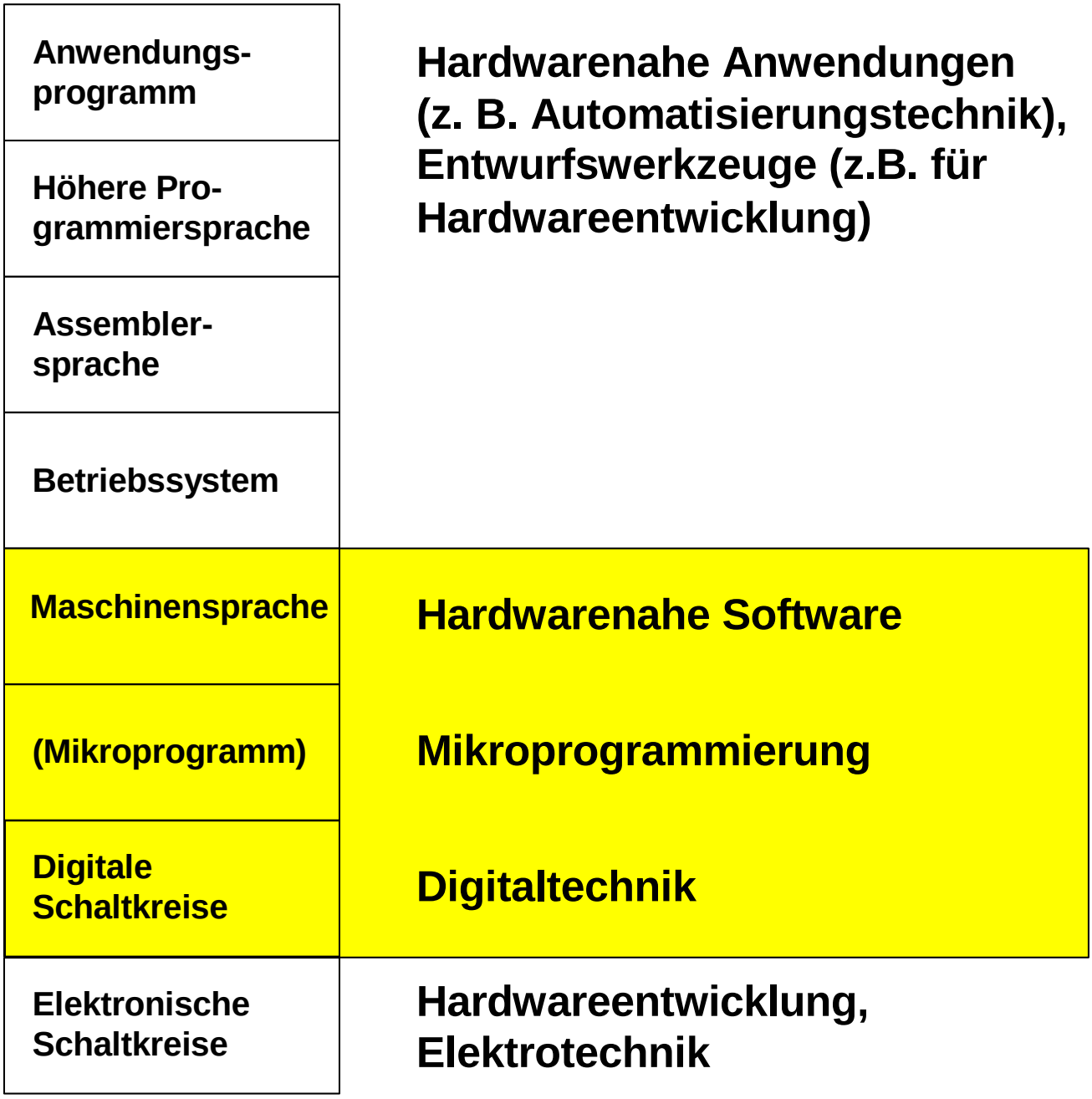
Funktion ist Verarbeitung in den einzelnen Komponenten als Teil der Struktur.

Das ***Schichtenmodell*** eines Rechners spiegelt verschiedene Sichtweisen und Abstraktionsebenen wider.

Die einzelnen Ebenen können wie eine Hierarchie virtueller Rechner betrachtet werden, die auf jeder Ebenen (bis auf die Hardwareebene) eine andere Sprache zur Verfügung stellen. Sie stellen eine **Abstraktion** darunter liegender Schichten dar.

Die Umsetzung der Sprache in den oberen vier Schichten erfolgt in der Regel durch Übersetzer, in den unteren durch Interpretation.

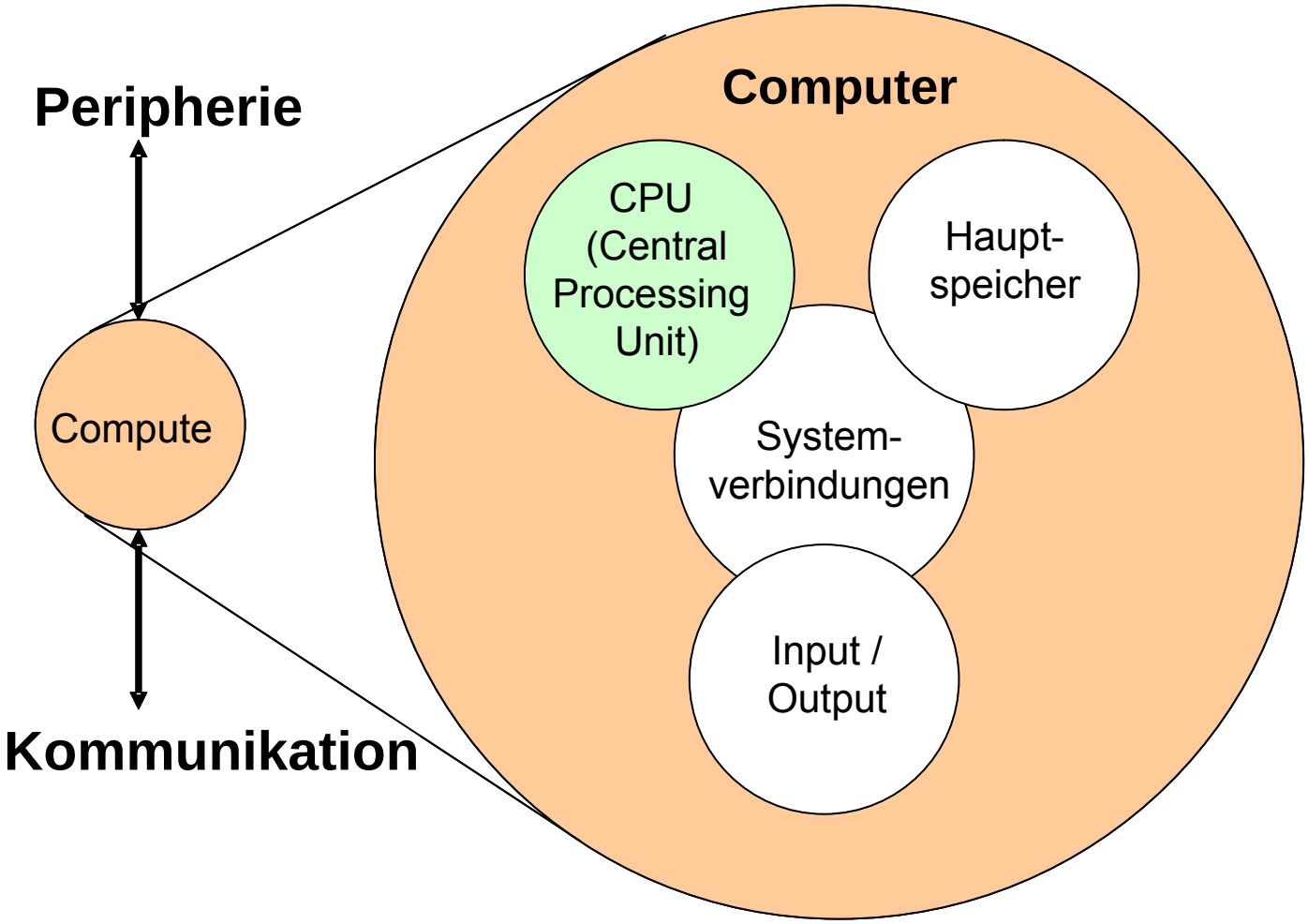
Schichtenmodell eines Rechners vs. Technische Informatik allgemein



Informatik A, B, D: Obere vier Schichten

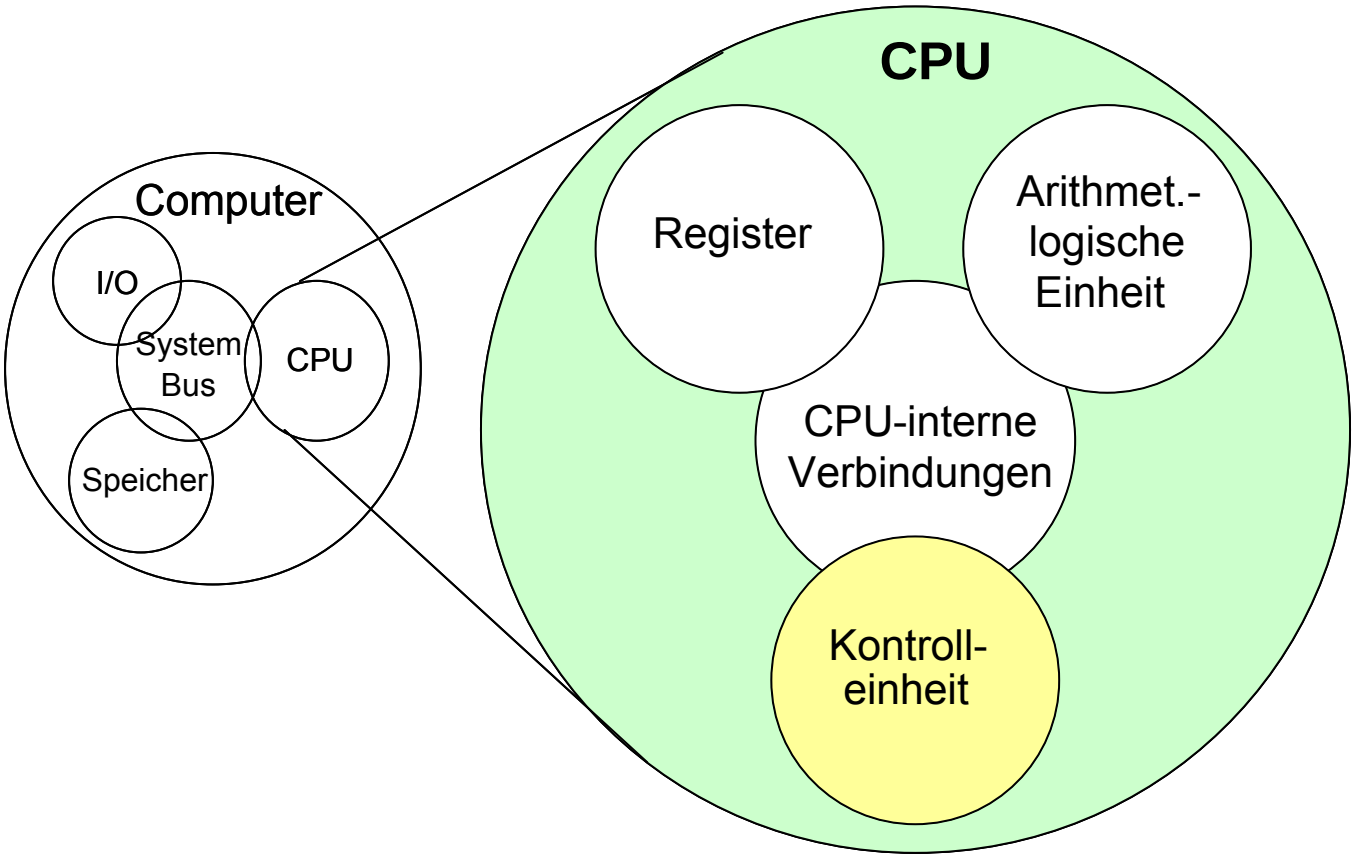
Technische Informatik: Untere Schichten (hardwarenahe Informatik oberhalb der Elektrotechnik)

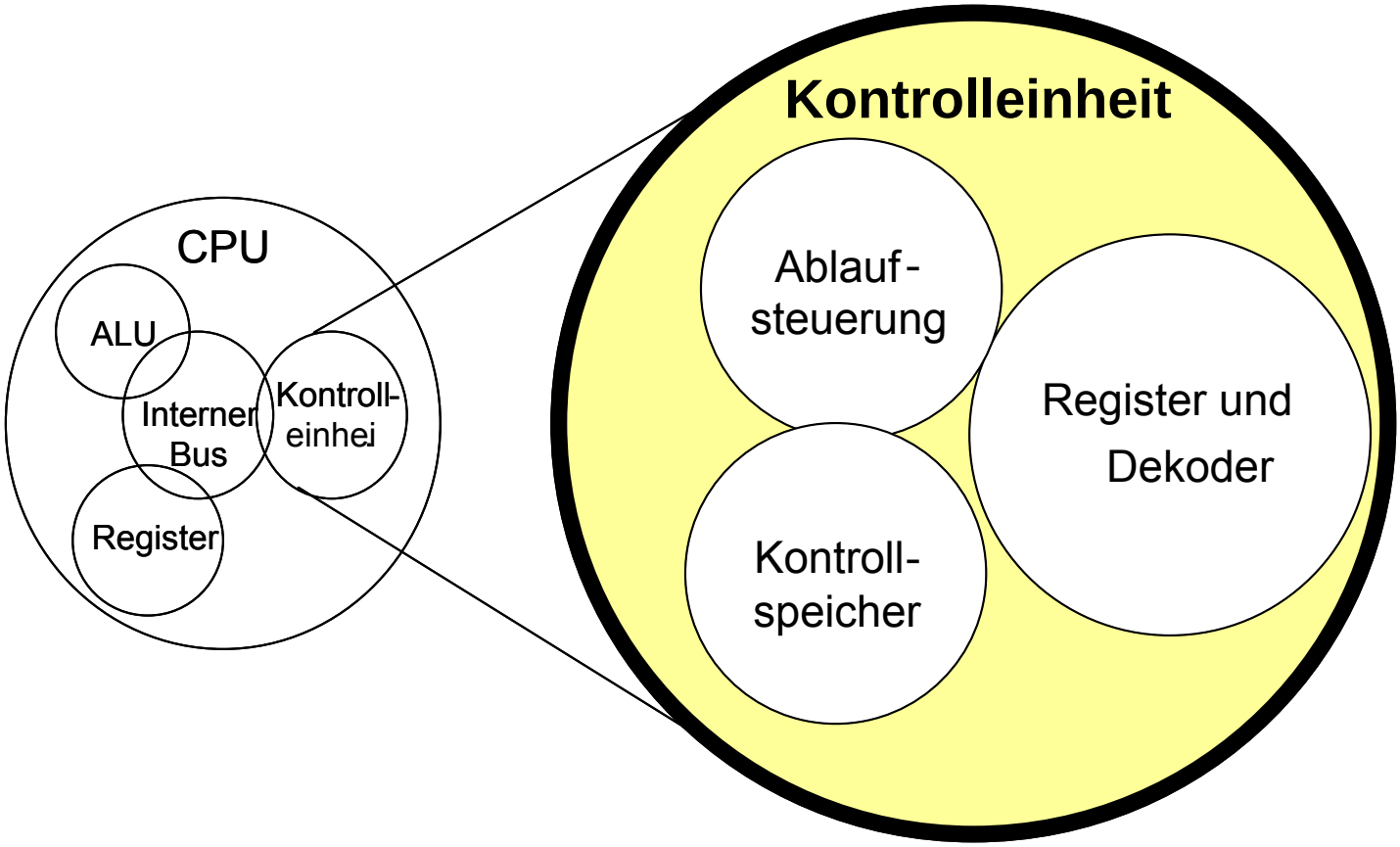
Typische Struktur von Rechnern



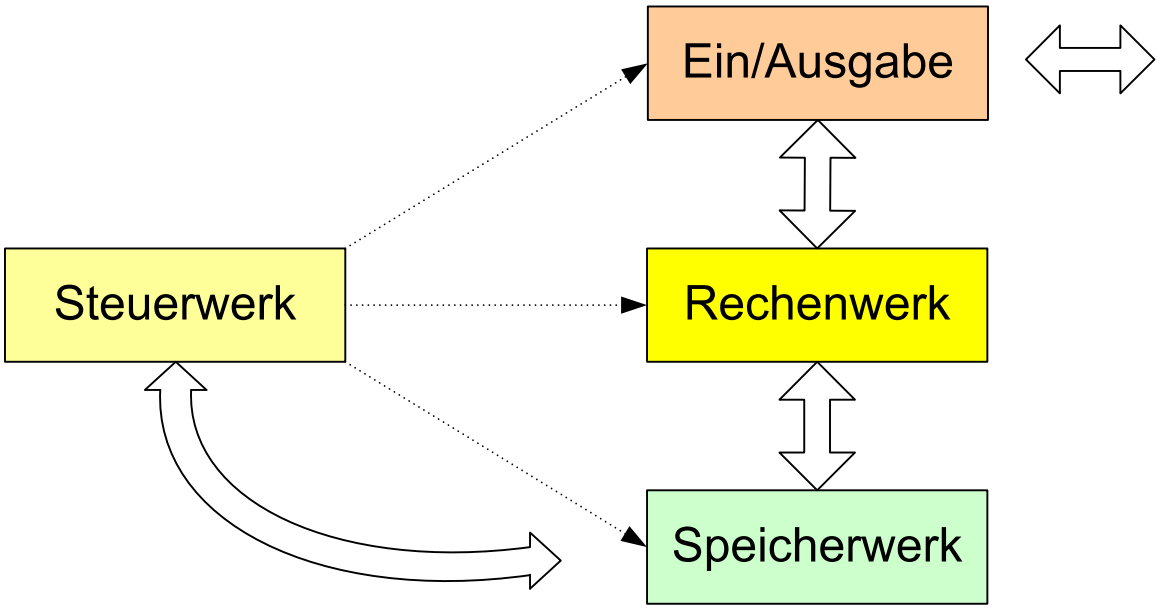
Ein Computer (mehr dazu in Kap. 8 und 9) besteht also immer aus:

- Prozessor (CPU)
- Speicher (Programm- und Arbeitsspeicher)
- Ein-/Ausgabeeinheit (E/A; Input/Output)


Computer
I/O
System Bus
CPU
Speicher
CPU
Register
Arithmet.-logische Einheit
CPU-interne Verbindungen
Kontroll-einheit


CPU
ALU
Interner Bus
Kontroll-einhei
Register
Kontrolleinheit
Ablauf-steuerung
Register und Dekoder
Kontroll-speicher

Von-Neumann-Rechner (Princeton-Architektur)



Daten und Instruktionen

Steuerimpulse

Kennzeichen:

(1) Der Rechner wird räumlich und logisch in folgende Teile gegliedert:

   (a) Rechenwerk (Rechenoperationen und logische Verknüpfungen).

   (b) Speicherwerk (Speicherung von Programmen und Daten).

   (c) Steuerwerk (Leitwerk) zur Steuerung des Programmablaufs.

   (d) Ein/Ausgabewerk zur Kommunikation mit der Außenwelt.

(2) Programmsteuerung durch von außen eingebbare Programme (Universalität).

(3) Programm und Daten werden in einem einheitlichen Speicher abgelegt (von Neumann-Architektur).

(4) Jeder Speicherplatz hat eine Adresse, über die sein Inhalt aufrufbar und ggf. ladbar ist.

(5) Befehle eines Programms werden i. Allg. aus aufeinanderfolgenden Speicherplätzen geholt (d. h. Erhöhen der Adresse um eins).

(6) Fetch-Decode-Execute-Arbeitszyklus

(7) Sprungbefehle (d. h. nach der Ausführung des Befehls mit Adresse s wird ein Befehl mit Adresse $t \neq s + 1$ ausgeführt).

(8) Bedingte Sprungbefehle (Sprung zu Befehl mit Adresse $t \neq s + 1$ nur, wenn eine Bedingung erfüllt ist, sonst Fortsetzung mit s+1)

(9) Verwendung des Dualzahlensystems.

Die Hardware wird also so gestaltet, dass sie selbsttätig nach der Abarbeitung eines Kommandos (Maschinenbefehl) das nächste aus einem Speicher holt. Dadurch lässt sich ein (Mikro-)Prozessor, der nach dem von-Neumann-Prinzip arbeitet, recht einfach realisieren.

Heutige Rechner verwenden fast alle noch das Grundprinzip nach von Neumann unverändert!!!

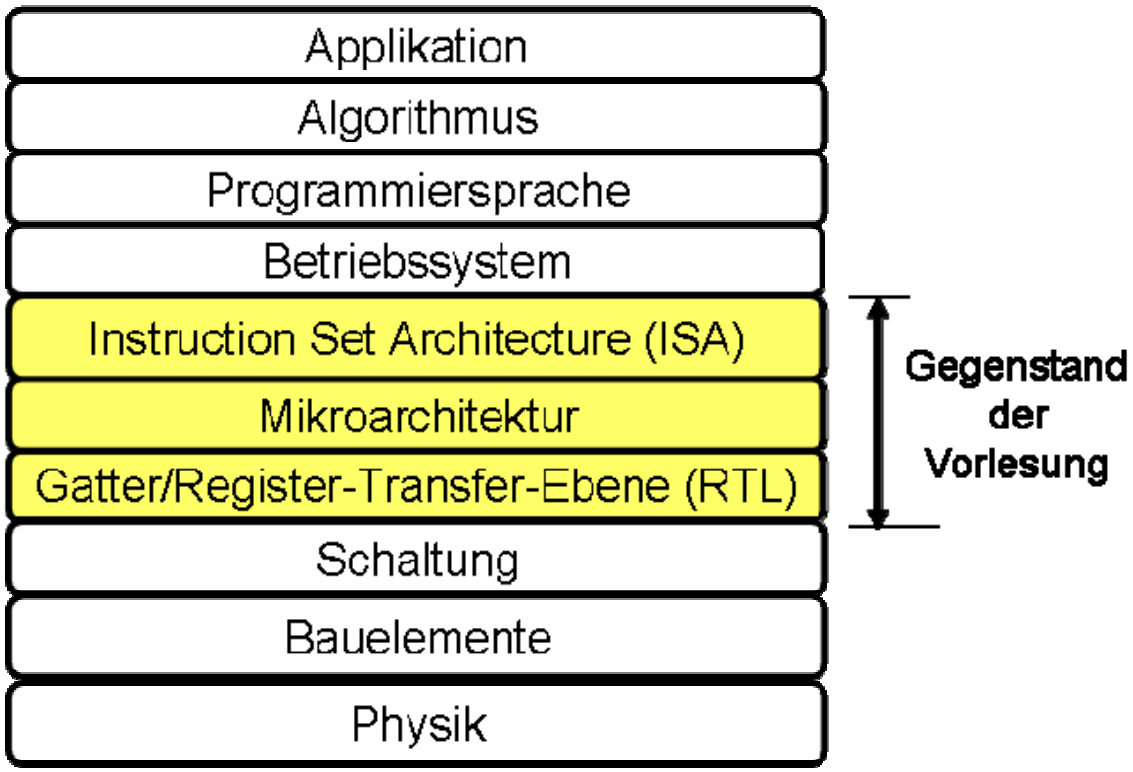
Befehlsatz-Architekturen

Die (Mikro-)Prozessoren stellen einen fest vorgegebenen Befehlssatz und eine Menge von Arbeitsregistern für die Programmierung bereit. Diese Hardware-Software-Schnittstelle wird ***Befehlssatzarchitektur*** genannt (ISA-Architektur, Instruction Set Architecture). Sie ist so gestaltet, dass durch sie beliebige Algorithmen abgearbeitet werden können (Universalität).

Jeder Prozessortyp (-familie) hat einen eigenen Maschinenbefehlssatz. Die Programme müssen daher in dem richtigen Maschinencode (Objektcode) im Speicher liegen.

Die Maschinenbefehlssatz steht direkt in Form einer Assemblersprache für die Programmierung zur Verfügung.

Einordnung in das Schichtenmodell eines Computers



1.6 Akkumulatormaschine H6809

1.6.1 Vorbemerkung

Dieses Kapitel betrachtet nun Mikroprozessoren als eine Universalhardware, die durch den Austausch von Programmen flexibel für die Implementierung der verschiedensten, auch komplexeren Abläufe/Algorithmen eingesetzt werden kann, und gibt auch eine Einführung in die Assemblerprogrammierung.

Es bildet damit eine Brücke zwischen der Hochsprachenprogrammierung und der technischen Realisierung.

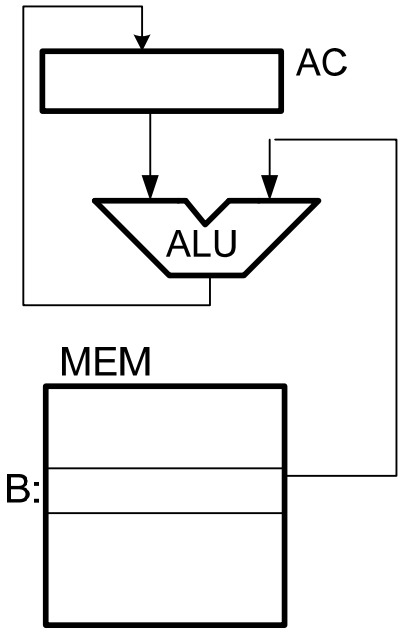
Als Beispiel wird der Einfachheit halber ein hypothetischer Prozessor, der **H6809**, verwendet, der eine abgespeckte Version des Motorola 6809-Mikroprozessors ist.

Bei dem Programmiermodell des H6809 handelt es sich um eine so genannte Akkumulatormaschine.

Grundprinzip einer Akkumulatormaschine

Eine *Akkumulatormaschine* ist eine einfache Prozessorarchitektur mit einem oder zwei Arbeitsregistern *(Akkumulatoren),* die zur Durchführung aller arithmetischen und logischen Befehle sowie als Quelle und Ziel für Transferbefehle von/zum Speicher bzw. Ein-/Ausgabe dienen. D. h., die Befehle haben als impliziten Operanden den Akkumulator und ggf. als weiteren Operanden eine Speicheradresse (siehe auch Beispiel-CPU aus Kap. 7.7).

Blockdiagramm einer Akkumulatormaschine



Beispiel: ADD B

AC ← AC + MEM (B)

Beliebte Architektur bei älteren Maschinen und einfachen, kostengünstigen (8-Bit-)Mikroprozessoren (insbesondere Mikrocontrollern).

Beispiele:		
	Intel 8080, 8085, 8051	(1 Akku)
	Zilog Z80	(2 Akkus)
	Motorola 6800, 68HC11*, 68HC12	(2 Akkus)
	MOS Technology 6510, 650X	(1 Akku)
	Motorola **6809**, 68HC08, 68HC11*	(1 Akku)

*: 2 8-Bit- auch als 1 16-Bit-Akkumulator nutzbar

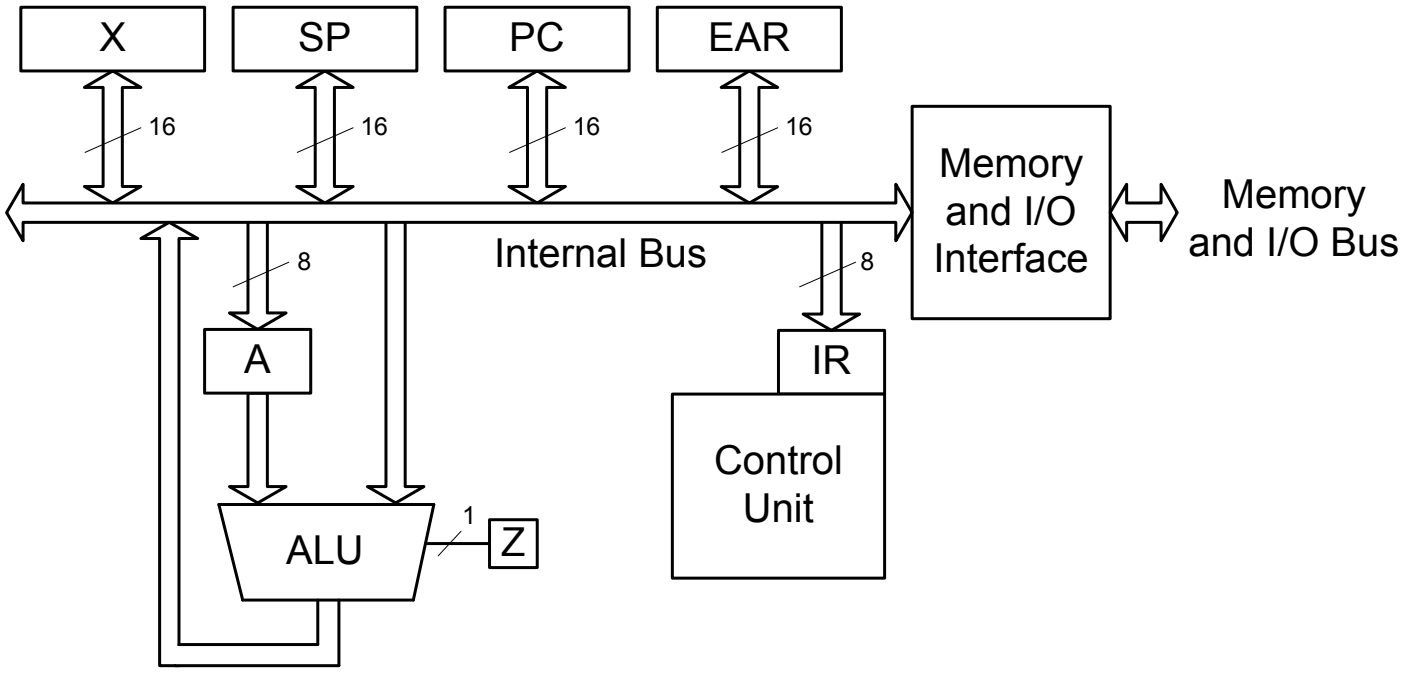
1.6.2 Organisation der hypothetischen Akkumulatormaschine H6809

Der H6809 ist eine Untermenge des Mikroprozessors *Motorola 6809*[1]. Er weist typische Kennzeichen modernen Mikroprozessoren auf.

Charakteristika:

- Wortlänge: 8 Bit (1 Byte)
- 16-Bit-Adressen, d. h. Adressraum: 2^{16} = 64 kB
- variables Befehlsformat: 1-, 2-, 3-Byte-Befehle
- Adressregister X für indirekte Adressierung
- Stapel für Unterprogramm-Rückkehradressen

Blockdiagramm des H6809



[1] nach John F. Wakerly: *Microcomputer Architecture and Programming*; John Wiley & Sons, New York, 1981

ALU *Arithmetisch-logische Einheit:*
Verknüpfung von 8-Bit-Operanden

Control Unit (Steuerwerk):
Decodierung des Befehls im IR-Register und Erzeugung der Steuersignale

Mem & I/O-Interface:
Schnittstelle zum Speicher- und E/A-Bus

IR *Instruktions-Register (Befehlsregister):*
Operationscode des auszuführenden Befehls (8 Bit)

EAR *Effektiv-Adress-Register:*
Adressteil des ausgeführten Befehls für Speicherzugriff in der Ausführungsphase (16 Bit)

PC *Program Counter (Befehlszähler):*
Zeigt auf die Adresse des nächsten auszuführenden Befehls (16 Bit)

A *Akkumulator:* Arbeitsregister für Daten (8 Bit)

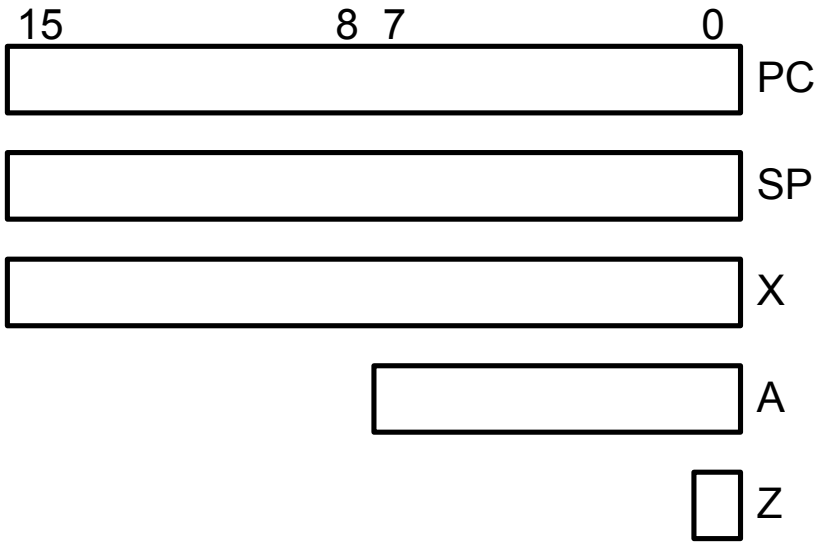
Z *Zero-Flag:*
1-Bit-Register; gesetzt, wenn Resultat = 0

X *´Index´-Register:*
Adressregister für indirekte Adressierung (16 Bit) (erlaubt auch einfache Operationen auf 16 Bit-Operanden)

SP *Stack Pointer (Stapelzeiger):*
zeigt auf die erste freie Speicherstelle vor der Spitze des Stapels (TOS) im Hauptspeicher (16 Bit)

Programmiermodell

(für Programmierer sichtbare und beeinflussbare Register, deren Inhalt dem Prozessorstatus entspricht)



Elementare Befehlszyklen

- Befehlsholphase (Fetch-Zyklus):

  IR ← MEM[PC],
  PC ← PC + 1;

- Befehlsausführungsphase (Execute-Zyklus):

  Ausführung des im IR kodierten Befehls,
  z. B. für LDA *addr*

  {Adressteil *addr* ins EAR holen}
  EAR[15..8] ← MEM[PC],
  PC ← PC +1;
  EAR[7..0] ← MEM[PC],
  PC ← PC +1;

  {Operand laden}
  A ← MEM[EAR],
  If A = 0 then Z ← 1 else Z ← 0;

1.6.3 Befehlssatz des H6809

Mnem.	Operand	Z	Length (bytes)	Opcode (hex)	Description	Number of Clock Cycles
NOP			1	12	No operation	1
CLRA		*	1	4F	Clear A	1
COMA		*	1	43	One´s complement bits of A	1
NEGA		*	1	40	Negate A (two´s complement)	1
LDA	#data	*	2	86	Load A with data	2
LDA	@X	*	1	A6	Load A with MEM[X]	2
LDA	addr	*	3	B6	Load A with MEM[addr]	4
STA	@X	*	1	A7	Store A into MEM[X]	2
STA	addr	*	3	B7	Store A into MEM[addr]	4
ADDA	#data	*	2	8B	Add data to A	2
ADDA	addr	*	3	BB	Add MEM[addr] to A	3
ANDA	#data	*	2	84	Logical AND data to A	2
ANDA	addr	*	3	B4	Logical AND MEM[addr] to A	3
CMPA	#data	*	2	81	Set Z according to A-data	2
CMPA	addr	*	3	B1	Set Z according to A-MEM[addr]	3
LDX	#addr	*	3	8E	Load X with addr	3
LDX	addr	*	3	BE	Load X with MEMW[addr]	5
STX	addr	*	3	BF	Store X into MEMW[addr]	5
CMPX	#addr	*	3	8C	Set Z according to X-addr	3
CMPX	addr	*	3	BC	Set Z according to X-MEM[addr]	5
ADDX	#addr	*	3	30	Add addr to X	3
ADDX	addr	*	3	31	Add MEMW[addr] to X	5
LDS	#addr	*	3	8F	Load SP with addr	3
BNE	offset		2	26	Branch if result is nonzero (Z=0)	2
BEQ	offset		2	27	Branch if result is zero (Z=1)	2
BRA	offset		2	20	Branch unconditionally	2
JMP	addr		3	7E	Jump to addr	3
JSR	addr		3	BD	Jump to subroutine at addr	5
RTS			1	39	Return from subroutine	3

Anmerkungen:

Mnem. = mnemonic; data = 8 Bit-Datum; addr = 16 Bit-Adresse/Datum; offset = 8 Bit signed Integer, wird bei einer Verzweigung zum PC addiert

MEM[i] Speicherbyte an Adresse i; MEMW[i] Speicherwort an Adresse i, also die Konkatenation von MEM[i] und MEM[i+1]

Die mit * gekennzeichneten Befehle beeinflussen das Z-Flag.

Die Anzahl Takte wird im Wesentlichen durch die Anzahl Speicherzugriffe bestimmt (z.B. 2-Phasen-Timing).

Befehlsgruppen

- Akkumulator-Befehle:
  CLRA … CMPA

- Adressregister-Befehle: LDX … LDS
  (X-, SP-Register)

- Programmfluss-Steuerung: BNE … RTS (NOP)
  (Sprünge etc.)

Befehlsformate des H6809

Der Op-Code des H6809 hat 8 Bit. Es sind also maximal 256 Befehle codierbar.

Der mnemonische Code (*Mnemocode*) ist in Anlehnung an die Semantik des Op-Codes gewählt.

Die Adressierungsarten sind hier als Teil des Op-Codes codiert (Kennzeichnung durch # bzw. @ im Mnemocode).

Variables Befehlsformat

Op-Code

1 Byte-Befehle, z. B. CLRA

Op-Code
Datum

2 Byte-Befehle, z. B. LDA #data

Op-Code
Addr (high)
Addr (low)

3 Byte-Befehle, z. B. LDA addr

1.6.4 Adressierungsarten des H6809

Die Adressierungsarten beschreiben die Berechnung der **effektiven Adresse** (EA) im EAR-Register nach Angaben im Op-Code.

Eine mögliche Einteilung ist nach der Anzahl der Komponenten zur Adressberechnung.

0-Komponenten-Adressierung

Implizite (inhärente) Adressierung

Spezifizierung des Operanden implizit im Op-Code

z. B. CLRA $A \leftarrow 0$

Unmittelbare Adressierung (Immediate)

Der Operand ist *unmittelbar* Bestandteil des Befehls (2. Byte).

z. B. LDA #$10

Anmerkung: Operanden und Argumente werden als Dezimalzahlen interpretiert, wenn sie nicht explizit durch ein vorgestelltes „$“ als hexadezimal gekennzeichnet sind.
(Es sind auch andere Kennzeichnungen für Hex-Zahlen gebräuchlich, z.B. „…H“ oder „0X…“.)

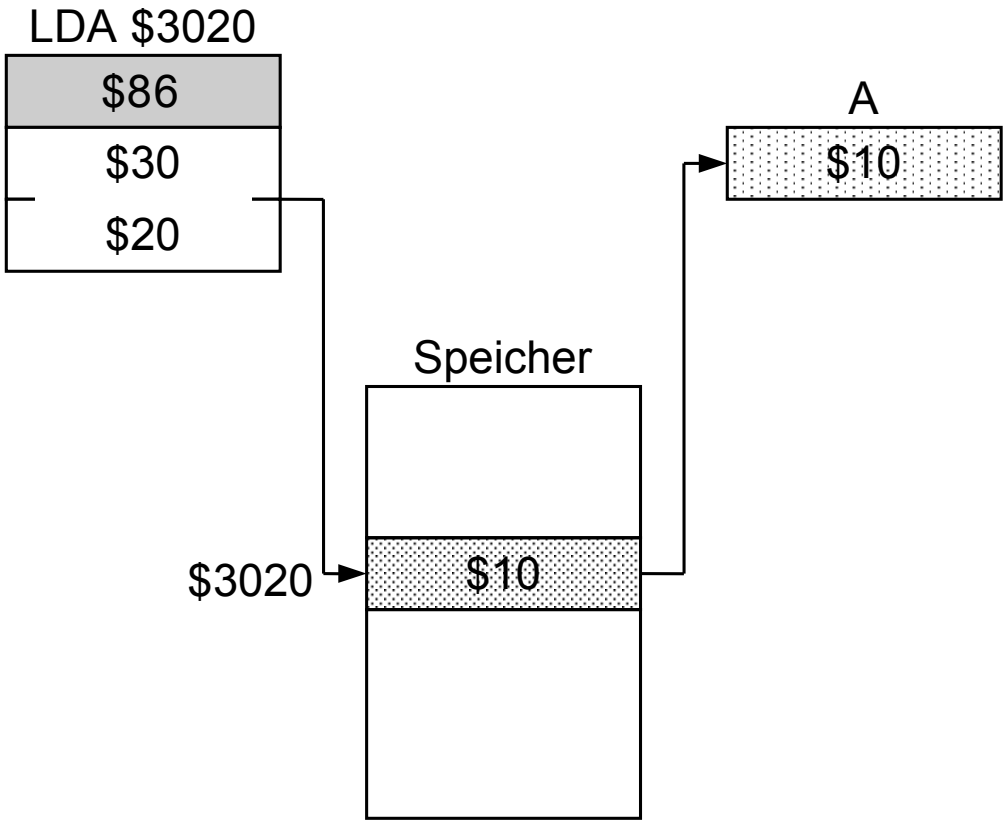
1-Komponenten-Adressierung

Ein Teil des Befehls oder ein Register bestimmen den Operanden bzw. seine Adresse.

Absolute Adressierung

Die Adresse des Operanden wird direkt im Befehl angegeben (2. und 3. Byte)

z. B. LDA $3020



Dies ist die einfachste Adressierungsart für Variablen.

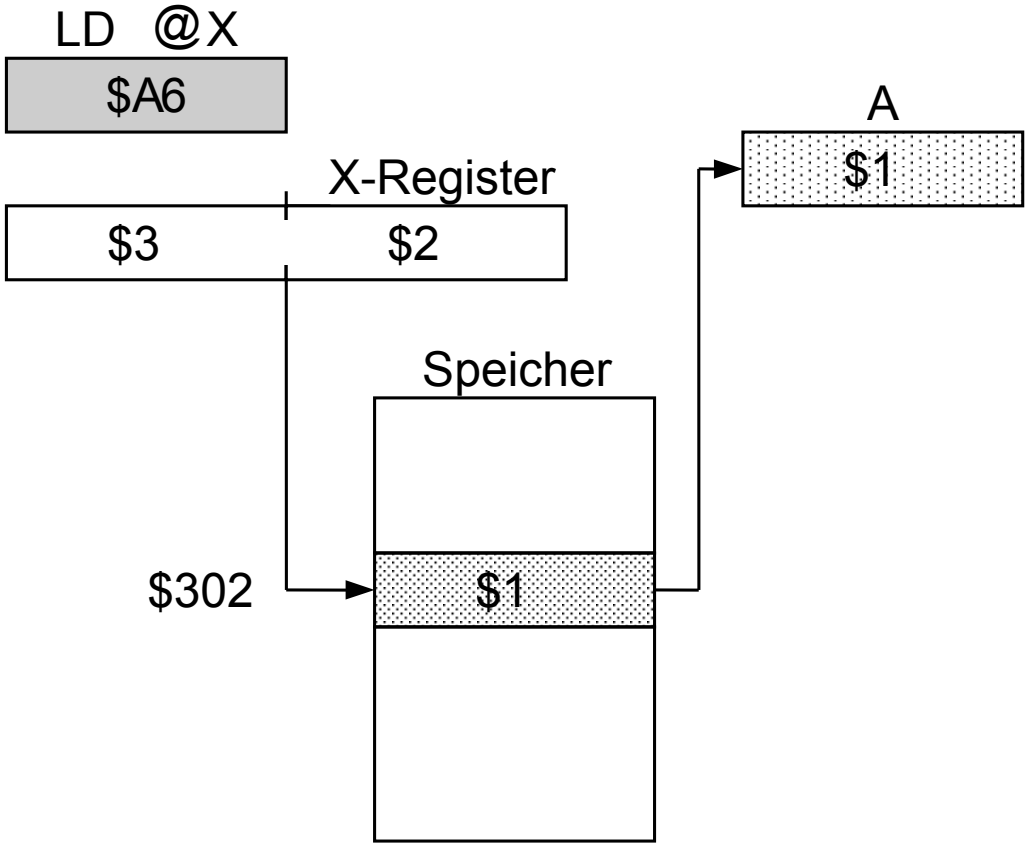
Register-indirekte Adressierung

Bei der indirekten Adressierung wird nicht die Adresse des Operanden direkt, sondern der Ort angegeben, an dem die effektive Adresse steht.
Hier steht die Adresse des Operanden im Adressregister X.

Die effektive Adresse kann also noch zur Laufzeit des Programms (im X-Register) geändert werden.

Das unterstützt (neben der echten indizierten Adressierung) z. B. die Implementierung komplexer Datenstrukturen wie Felder, Stapel, Schlangen …

LDA @X



Anmerkung: Das X-Register im H6809 ist kein echtes *Index*register, weil es hier nur für die indirekte Adressierung verwendet werden kann.

Beispiel: Initialisierung eines Feldes

var Q: array[0..4] **of** Byte; …

. . .

for i := 0 **to** 4 **do** Q[i] := 0;

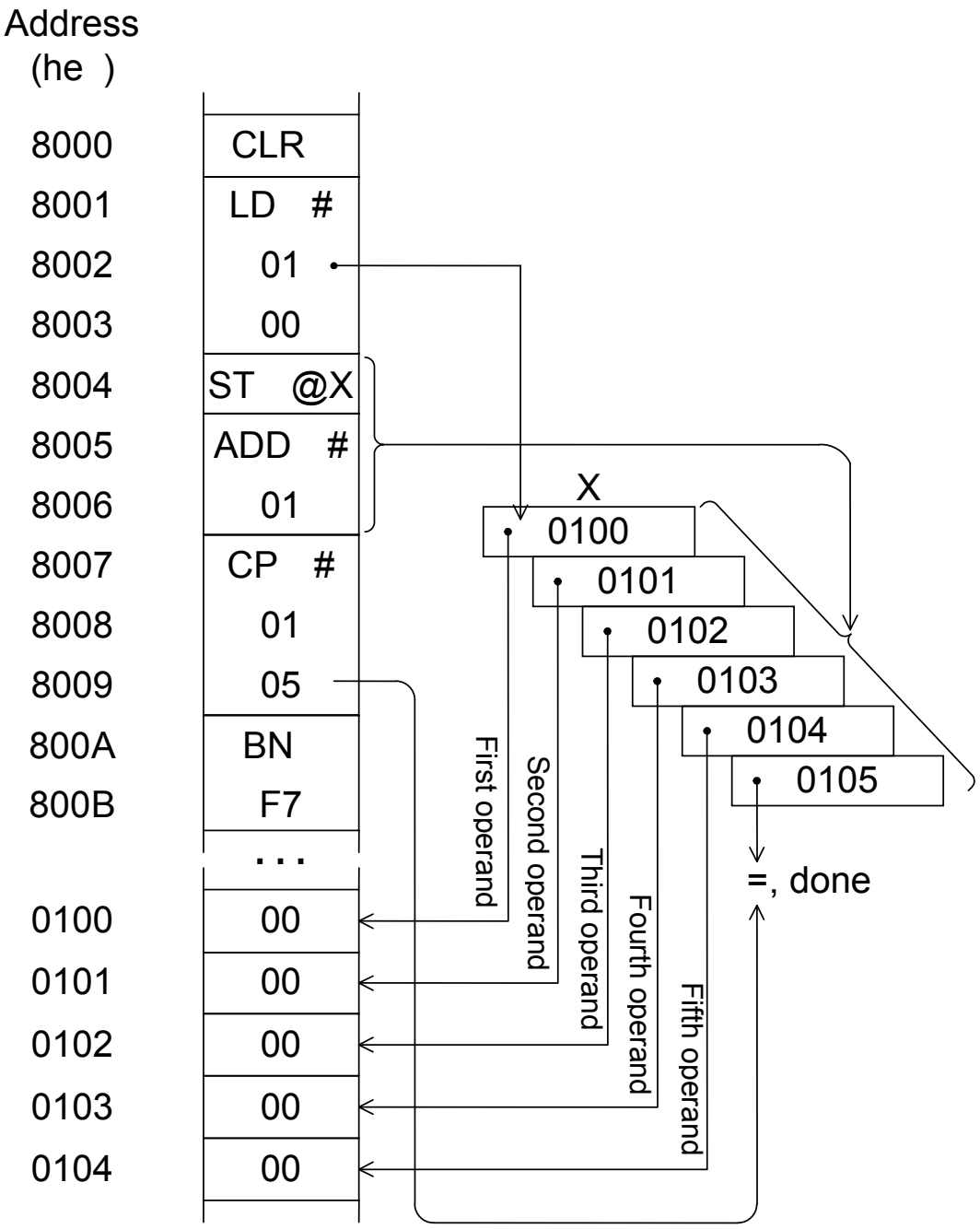
Mit absoluter Adressierung:

```
Addr  Contents  Label   Op-code  Operand   Comments
                        .ORG     $8000
8000  4F        INIT:   CLRA               ; Set components of Q to 0
8001  B7  0100          STA      Q         ; First component
8004  B7  0101          STA      Q+1       ; Second component
8007  B7  0102          STA      Q+2       ; Third component
800A  B7  0103          STA      Q+3       ; Fourth component
800D  B7  0104          STA      Q+4       ; Fifth component
8010  7E  1000          JMP      $1000     ; Return to operating syst.
...                     .ORG     $0100
0100  ??        Q       .BYTE    5         ; Reserve 5 bytes for array
...                     .EXIT    INIT
```

Mit register-indirekter Adressierung:

```
Addr  Contents  Label   Op-code  Operand   Comments
                        .ORG     $8000
8000  4F        INIT:   CLRA               ; Set components of Q to 0
8001  8E  0100          LDX      #Q        ; Address of first compon.
8004  A7        LOOP:   STA      @X        ; Set MEM[X] to 0
8005  30  0001          ADDX     #1        ; Point to next component
8008  8C  0105          CMPX     #Q+5      ; Past last component?
800B  26  F7            BNE      LOOP      ; If not, go do some more
800D  7E  1000          JMP      $1000     ; Return to operating syst.
...                     .ORG     $0100
0100  ??        Q       .BYTE    5         ; Reserve 5 bytes for array
...                     .EXIT    INIT
```

Arbeitsweise der indirekten Adressierung



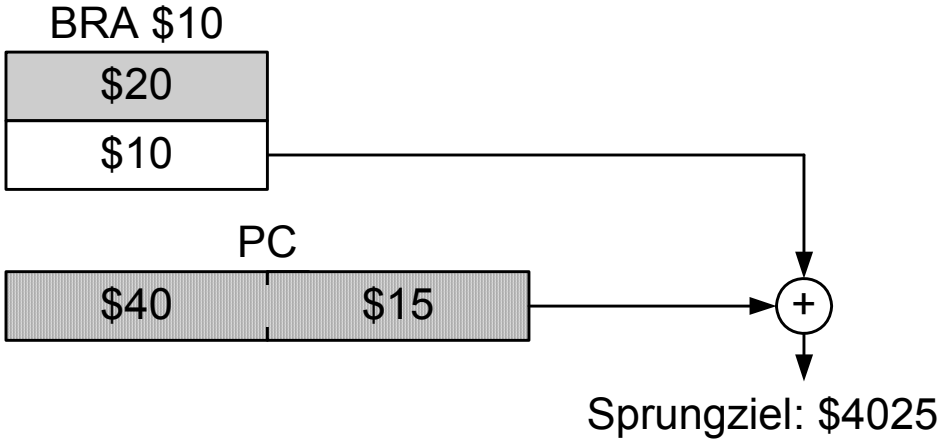
Die Anfangsadresse des Feldes ($0100) wird in das X-Register geladen und zur Laufzeit bei jeder Iteration um eins erhöht.

Der Zugriff auf die Feldvariablen (Operand) erfolgt indirekt über das X-Register (STA @X).

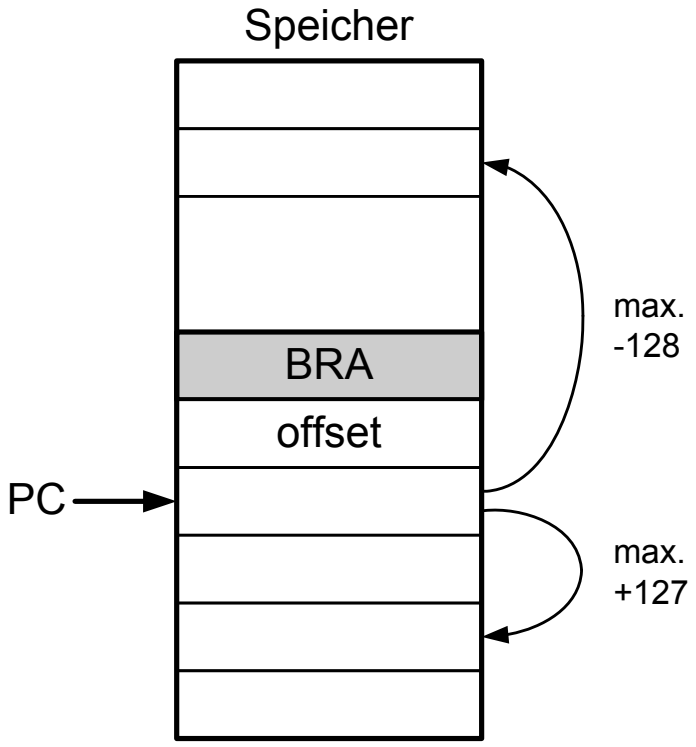
So sind auch große Felder leicht handhabbar.

PC-relative Adressierung

Die effektive Adresse (des nächsten auszuführenden Befehls) wird als Summe des aktuellen Programmzählers PC und eines Offsets (pos. oder neg. im Zweierkomplement), der Teil des Befehls ist (2. Byte), in einer eigenen Hardware für die Adressberechung gebildet.



Die PC-relative Adressierungsart wird für relative Sprünge / Verzweigungen genutzt.



Das ist günstig für verschieblichen Code, aber nur für kurze Sprungdistanzen geeignet.

1.6.5 Assemblerprogrammierung am Beispiel des H6809

Der „Assembler“

Der Assembler ist ein Programm (auf dem Entwicklungssystem), das ein Quellprogramm (Source-Code) aus der Assemblersprache mit symbolischer, mnemonischer Notation eins–zu–eins in ein binäres Maschinenprogramm übersetzt (Objektcode aus Folgen von Nullen und Einsen). D.h., eine Assembleranweisung entspricht einem Maschinenbefehl. Man spricht von einem Makroassembler, wenn der Assembler auch Makros (s.u.) unterstützt.

Moderne Assembler sind in Entwicklungsumgebungen integriert, die außerdem noch einen Editor und verschiedene Debugging-Hilfsmittel sowie einen Lader enthalten.

Der Lader lädt den Objektcode in den Speicher des Zielsystems und startet die Ausführung. Alternativ wird eine Programmierdatei z.B. für einen Flash- oder EEPROM-Speicher erzeugt (Hex-Format).

Während der Übersetzung wird für die Dokumentation und das Debugging i.d.R. auch ein List-File erzeugt, dem u.a. die Adressen der Assembleranweisungen zu entnehmen sind.

Dadurch stehen dem Entwickler von Assemblerprogrammen praktisch die gleichen Hilfsmittel wie bei der Hochsprachenentwicklung zur Verfügung. D.h., es können **und sollten** die gleichen Programmiertechniken angewendet werden:

- Strukturierung (Unterprogramme, Makros bzw. Dateien)
- aussagekräftige Label für Konstanten und Sprungziele
- Kommentare
- Debugging (z.B. **Single-Stepping, Breakpoints**)

Programmierhinweise

Bei der Programmierung in Assembler steht ein unmittelbarer Zugriff auf die Hardware(-Software-Schnittstelle) zur Verfügung. Dadurch können und müssen aber auch (im Vergleich zur Hochsprachenprogrammierung) mehr Details berücksichtigt werden.

Daher bietet sich für die Assemblerprogrammierung folgende Vorgehensweise an:

1) Programmablaufplan erstellen
2) Bestimmung der erforderlichen Konstanten und Variablen
3) Festlegung der Register- und Speicherbelegung (für Programm und Daten)
   (dabei entscheiden, ob Daten global im Speicher oder lokal beim (Unter)Programm gehalten werden)
4) Label vergeben für Daten (Variablen und Konstanten) und Programmabschnitte (mindestens Sprungziele)
5) explizites Initialisieren **aller Variablen** und des **Stack Pointers** nicht vergessen !!!
6) dann zunächst Programmfunktionalität schrittweise als Kommentar hinschreiben
7) erst danach ausprogrammieren

Tipp: Konstanten wegen Wartbarkeit explizit möglichst global und zentral anlegen
(Bei größeren Projekten wird in der Praxis mit *Include-Dateien* gearbeitet.)

Format eines Assemblerbefehls

Label	*Op-Code*	*Operand(en)*	*Kommentar (nach ";")*
LOOP:	ADDA	addr	; Add variable at [addr] to A

Assembler-Direktiven (Pseudobefehle)

sind Anweisungen an den Assembler, die nicht in Maschinenbefehle übersetzt werden. Üblich sind Anweisungen wie:

.ORG (Programm-)Ladeadresse. Gibt an, ab welcher Adresse der nachfolgende Programmcode bzw. Datenbereich im Speicher liegen soll, nachdem das Programm geladen wurde.

.BYTE Reservierung von Speicherplatz (Anzahl Bytes) für Variablen. Die **Variable** erhält die Adresse des ersten reservierten Speicherplatzes. Der Speicherinhalt ist undefiniert.

.DB Spezifiziert 8 Bit-**Konstante**, die im Speicher abgelegt wird.

.DW Spezifiziert 16 Bit-**Konstante**, der in zwei auf einander folgenden Adressen abgelegt wird.

.DB und .DW können auch mehrere Parameter oder Ausdrücke haben (Trennung durch Komma).

.EQU Dem Bezeichner im Label-Feld wird der Wert im Operandenfeld zugewiesen. D. h. der Bezeichner kann im Programm anstelle des Wertes verwendet werden (vgl. Konstantendeklaration in Hochsprachen).

.EXIT Ende des zu assemblierenden Programmtextes, oft mit Angabe der Startadresse bzw. –marke; sonst Ende am File-Ende.

; Kommentar(zeile)

Bei manchen Assemblern:

***** in einem Ausdruck: aktuelle Adresse, die gerade assembliert wird (*Program location counter*), d. h. Festlegung zur Assemblierzeit

Beispielprogramm für den H6809

Multiplikation durch fortgesetzte Addition

Algorithmus im Pseudocode:

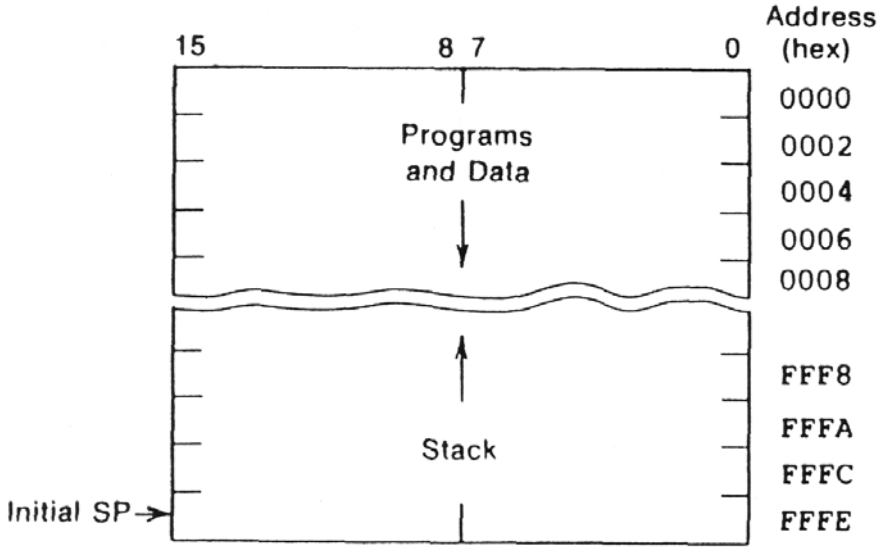
```
MCND  aus Speicherstelle holen;
MPY   aus Speicherstelle holen;
PROD  = 0;
CNT   = MPY;
WHILE CNT <> 0  DO {
    PROD = PROD +MCND;
    CNT - - ; }
```

```
Addr  Contents  Label    Op-code  Operand   Comments
                         .ORG     $2A40   ; Multiply MCND by MPY.
2A40  4F        START:   CLRA             ; Init.
2A41  B7 2C00            STA      PROD    ; Set PROD to 0
2A44  B6 2C02            LDA      MPY     ; Set CNT equal to MPY
2A47  B7 2C01            STA      CNT     ; and do loop MPY times
2A4A  B6 2C01  LOOP:     LDA      CNT     ; Done if CNT = 0
2A4D  27 10              BEQ      OUT
2A4F  8B FF              ADDA     #-1     ; Else decrement CNT
2A51  B7 2C01            STA      CNT
2A54  B6 2C00            LDA      PROD    ; Add MCND to PROD
2A57  BB 2C03            ADDA     MCND    ; only 8 bit result
2A5A  B7 2C00            STA      PROD
2A5D  20 EB              BRA      LOOP    ; Repeat the loop again
2A5F  B6 2C00  OUT:      LDA      PROD    ; Put PROD in A when done
2A62  7E 1000            JMP      $1000   ; Return to operating syst.

...                      .ORG     $2C00
2C00  ??       PROD:     .BYTE    1       ; Storage for PROD
2C01  ??       CNT:      .BYTE    1       ; Storage for CNT
2C02  05       MPY:      .DB      5       ; Multiplier value
2C03  17       MCND:     .DB      23      ; Multiplicand value
...                      .EXIT    START
```

Übliche Speicherorganisation

Aufteilung des Speichers in einen Programm- und Datenbereich, der von den niedrigeren zu den höheren Adressen wächst, sowie einen Stack (Stapel), der von einer höheren Speicheradresse zu niedrigeren Adressen hin wächst.



Im Beispiel hier: Speicher mit 16 Bit-Adressraum mit Wortorganisation von 16 Bit und Byteadressierung (*little endian*).

1.6.6 Unterprogramme (Subroutines)

Unterprogramme entsprechen PROCEDUREs, FUNCTIONs in Hochsprachen. Sie bilden eine Anweisungssequenz, die nur einmal geschrieben und beliebig häufig (von verschiedenen Stellen) aufgerufen werden kann. Es wird also nur einmal der entsprechende Code im Speicher abgelegt.

Probleme:

- Aufruf des Unterprogramms („*gerufenes Programm*“) erfordert Retten des **Prozessorstatus**, zumindest der Rückkehradresse
- Verlassen des Unterprogramms durch Rücksprung an gerettete Rückkehradresse im „*rufenden Programm*“
- Parameterübergaben an das Unterprogramm und zurück an das rufende Programm
- geschachtelte und ggf. rekursive/wiedereintrittsfähige Unterprogramme.

Oft ist die Schachtelungstiefe und damit auch die Anzahl an zu übergebenden Parametern zur Assemblier-(Kompilier) Zeit nicht bekannt (z.B. bei Rekursionen). Deshalb muss eine dynamische Datenstruktur verwendet werden.

Üblicherweise werden Stapel (**Stack**) für die Rückkehradresse (Return Stack) verwendet.

Der Stapelzeiger (**Stack Pointer**) wird vom Hauptprogramm bzw. Betriebssystem auf freien Bereich im RAM initialisiert.

Der Stack Pointer zeigt beim H6809 auf die erste freie Speicherzelle vor dem „**Top of Stack**“ (TOS). (Bei anderen Prozessoren kann er auch auf den TOS selbst zeigen.)

- Unterprogrammaufruf: **JSR *Sub***

  ***Back*:** **...** ; ab hier weiter

  Push der Adresse der nächsten Anweisung (*Back*) auf den Stapel; d.h., Schreiben des aktuellen PC auf den Stack und Dekrementieren des SP per Hardware durch die Kontrolleinheit.
  Sprung an Adresse *Sub* durch Laden des Program Counters.

- Verlassen des Unterprogramms:

  ***Sub*:** **...** ; Unterprogr.anweisungen

  RTS

  Pop (*Back*) vom Stapel; d.h., Inkrementieren des SP und Laden des PC mit der Rückkehradresse per Hardware, dadurch Sprung an Adresse *Back*.

- Parameterübergabe an Unterprogramme:

  hier keine explizite Unterstützung (sonst meist über Register oder Stack)

Beispielprogramm mit Unterprogramm

Zählt die Anzahl Einsen in einem 16-Bit-Wort

```
Addr  Contents   Label     Opc.   Operand      Comments
                SYSRET:   .EQU   $1000     ; Operating system address
                          .ORG   $0100     ; init. small stack
0100  ??        STK:      .BYTE  7*2        ; Space for 7 return addr.
0100            STKE:     .EQU   *-1        ; Init. of address for SP

          5B29  TWORD:    .DW    $5B29      ; Test word to count 1s

                          .ORG   $2000
2000  8F  201A  MAIN:     LDS    #STKE      ; Initialize SP
2003  BE  201A            LDX    TWORD      ; Get test word
2006  BD  201C            JSR    WORDCT     ; Count number of 1s in it
2009  7E  1000            JMP    SYSRET     ; Return to operating system
201C                                        ; Count the number of '1'
201C                                        ; bits in a word.
201C                                        ; Enter with word in X.
201C                                        ; Exit with count in A.
201C  BF  2032  WORDCT:   STX    CWORD      ; Save input word
201F  B6  2032            LDA    CWORD      ; Get high-order byte
2022  BD  2035            JSR    BYTECT     ; Count 1s
2025  B7  2034            STA    W1CNT      ; Save '1' count
2028  B6  2033            LDA    CWORD+1    ; Get low-order byte
202B  BD  2035            JSR    BYTECT     ; Count 1s
202E  BB  2034            ADDA   W1CNT      ; Add high-order count
2031  39                  RTS               ; Done, return
2032  ??        CWORD:    .BYTE  1*2        ; Save word being counted
2034  ??        W1CNT:    .BYTE  1          ; Save number of 1s
2035                                        ; Count the number of '1'
2035                                        ; bits in a byte.
2035                                        ; Enter with byte in A.
2035                                        ; Exit with count in A.
2035  B7  2061  BYTECT:   STA    CBYTE      ; Save input byte
2038  4F                  CLRA              ; Initialize '1' count
2039  B7  2062            STA    B1CNT
203C  8E  2059            LDX    #MASKS     ; Point to 1-bit masks
203F  A6        BLOOP:    LDA    @X         ; Get next bit mask
2040  B4  2061            ANDA   CBYTE      ; Is there a '1' there?
2043  27  08              BEQ    BNO1       ; Skip if not
2045  B6  2062            LDA    B1CNT      ; Otherwise increment
2048  8B  01              ADDA   #1         ;    '1' count
204A  B7  2062            STA    B1CNT
204D  30  0001  BNO1:     ADDX   #1         ; Point to next mask
2050  8C  2061            CMPX   #MASKE     ; Past last mask?
2053  26  EA              BNE    BLOOP      ; Continue if not
2055  B6  2062            LDA    B1CNT      ; Put total count in A
2058  39                  RTS               ; Return
2059                                        ; Define 1-bit masks to test
2059                                        ; bits of byte
2059  80402010  MASKS:    .DB    $80,$40,$20,$10,$8,$4,$2,$1
205D  08040201                              ; local const. and var.
2061            MASKE:    .EQU   *          ; Address just after table
2061  ??        CBYTE:    .BYTE  1          ; Save byte being counted
2062  ??        B1CNT:    .BYTE  1          ; Save '1' count
2063                        .EXIT  MAIN
```

Hauptprogramm:

- initialisiert den Stapelzeiger und reserviert Platz für den Stapel (7 Worte)

- bereitet Testwort für Parameterübergabe im X-Register vor

- Sprung in Unterprogramm WORDCT

- Rückkehr ins Betriebssystem

Unterprogramm WORDCT:

- zählt die Anzahl von Einsen in dem als Parameter (Adresse im X-Register) übergebenen Wort

- ruft dazu das Unterprogramm BYTECT zweimal auf (Parameterübergabe jeweils im Akkumulator)

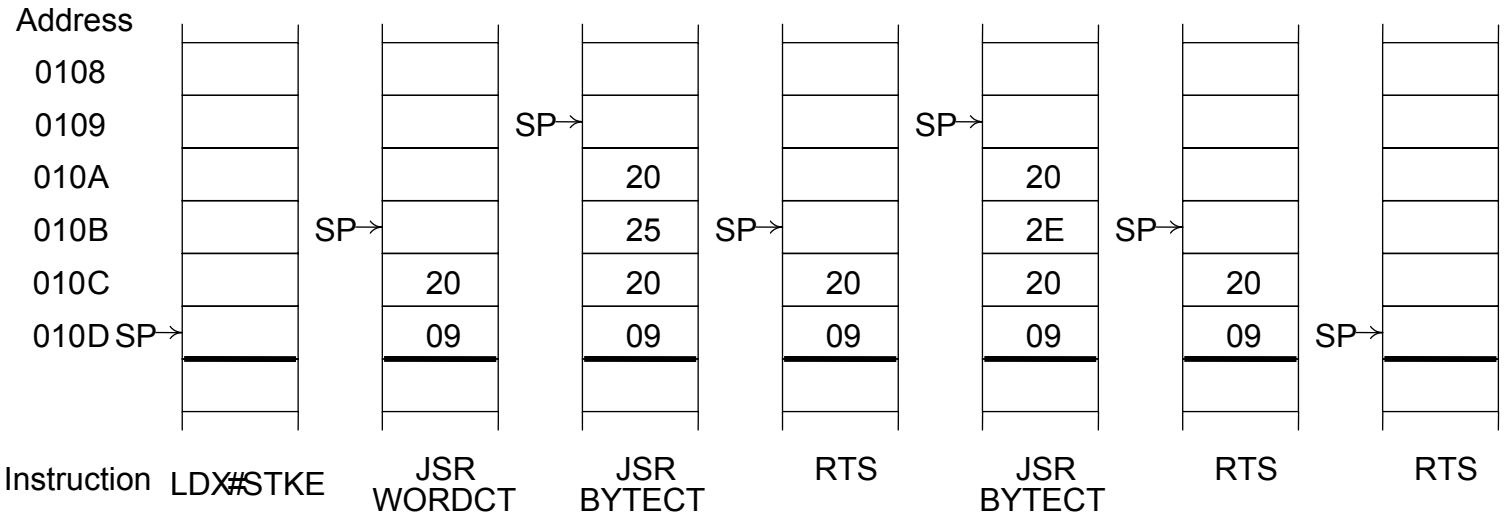
- addiert die Anzahl gezählter Einsen in beiden Bytes und kehrt ins Hauptprogramm zurück
(Parameterübergabe ebenfalls im Akkumulator)

Unterprogramm BYTECT:

- zählt die Einsen des im Akkumulator übergebenen Bytes

- gibt Ergebnis im Akkumulator an das rufende Programm zurück

Stapelinhalt im Verlauf der Programmausführung



Die Verwaltung der Unterprogramm-Rückkehradressen mittels eines Stapels als dynamische Datenstruktur ist heute Standard bei Mikroprozessoren.

Vorteile: Beliebig geschachtelte und rekursive Unterprogramme sind leicht implementierbar.

Der Stapel ist auch für andere Zwecke wie Parameterübergabe an Unterprogramme und die Auswertung arithmetischer Ausdrücke sehr gut geeignet.

Es gibt auch Prozessoren, die gar kein(e) Arbeitsregister oder Akkumulator(en) besitzen und alle Operationen nur auf dem Stapel abwickeln (Stack-Maschinen, s.u.).

1.6.7 Makros

Um Assemblerprogramme übersichtlicher zu gestalten und um für wiederkehrende Programmstücke nicht immer den gleichen Code schreiben zu müssen, werden Makros verwendet.

Format:

```
.MACRO  macroname
        …
        Anweisungsliste
        …
.ENDMACRO
```

Im Gegensatz zu Unterprogrammen wird an jeder Stelle des Makroaufrufs der entsprechende Code eingefügt. Dadurch wird der mit einem Unterprogrammaufruf verbundene (Zeit-) Aufwand eingespart, aber mehr Speicher gebraucht.

Nichtsdestotrotz können (eine beschränkte Anzahl) formale Parameter an Makros übergeben werden, um den Code für die jeweilige Verwendung zu spezialisieren.

Immer wenn der Makroname im Programm auftaucht, wird das Makro expandiert, indem an diese Stelle die Anweisungen eingetragen werden.

Die Aufrufparameter sind im Makro der Reihe nach beginnend mit @0 zugreifbar.

Beim Aufruf des Makros ersetzt der Assembler (zur Assemblierzeit) die formalen Parameter durch die aktuellen Parameter.

Beispiele:

Emulation des Befehls INCA

```
.MACRO  INCA  ; increment accu
ADDA    #1    ; by simplified notation
              ; needs still two bytes of
              ; memory and a number of
              ; clock cycles
.ENDMACRO     ; end macro INCA
```

Aufruf mit: `INCA`

Emulation des Befehls SUBA #data

```
.MACRO  SUBA  ; subtract immediately
STA     temp  ;  assumes  a  single  auxil-
iary
              ;  memory  cell  for  all  mac-
ros
LDA     #@0   ; put in actual constant
              ; parameter labelled @0
NEGA          ; build 2´s-complement
ADDA    temp  ; perform subtraction
.ENDMACRO     ; end macro SUBA
```

Aufruf z.B. mit: `SUBA $12`

Fragen, um die es also in dieser Vorlesung geht:

Wie werden digitale Systeme allgemein und Rechner prinzipiell gebaut?

Wie werden solche Rechner auf Maschinensprachebene programmiert?