Kapitel 3: Rechenanlagen

Anmerkung: Dieses Kapitel wird zum Selbststudium und der Vollständigkeit halber zur Verfügung gestellt. Inhaltlich ist es durch die beiden Exkursionen, zum Deutschen Technikmuseum und zum Potsdamer Platz, abgedeckt.

Um Programme auszuführen, ist ein *Prozessor* erforderlich, der die einzelnen Schritte tätigt. Das kann ein Mensch oder eine Maschine (auf mechanischer, elektronischer oder biochemischer Basis) sein, oder sogar ein anderes Programm, welches eine Ausführungsmaschine nur simuliert.

3.1 Historische Entwicklung

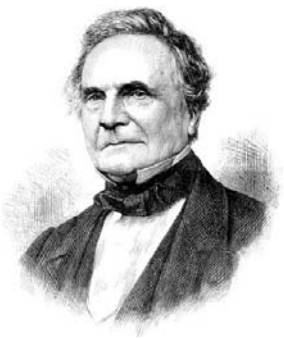
Die Entwicklung und den Aufbau moderner Rechner begreift man besser, wenn man sich ihre historischen Wurzeln betrachtet.

- 1842 Charles Babbage / Ada Lovelace: „Die analytische Maschine“; Konzept einer programmierbaren mechanischen Rechenanlage zur Lösung von Differentialgleichungen. Dieser „erste Computer der Weltgeschichte“ wurde jedoch nie realisiert, da Kosten, Machbarkeit und Haltbarkeit nicht einschätzbar waren
- 1936: Alonzo Church (1903-1995): „lambda-Kalkül“, Begriff der berechenbaren Funktion
- 1936: Alan Turing: Computer als universelle Maschine; Äquivalenz von Programm und Daten („Turing-Maschine“)
- 1941 Konrad Zuse: Z3: vollautomatischer, programmierbarer, in binärer Gleitkommarechnung arbeitender Rechner mit Speicher und einer Zentralrecheneinheit aus Telefonrelais
- 1946 Johan von Neumann (EDVAC-Report): konkrete Vorschläge für Aufbau („von-Neumann-Computer“)

Programmierparadigmen:

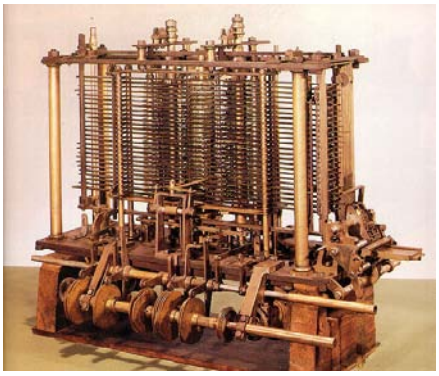
- Analytische Maschine: Rechnen als Durchführung arithmetischer Operationen
- lambda-Kalkül, Lisp-Maschine: Rechnen als Termersetzung
- Turing-Maschine (vgl. ThI): Rechnen als Schreiben von Zeichen auf ein Band
- von Neumann: Rechnen als Modifikation von Wörtern im Speicher.

Die analytische Maschine



Babbage’s „analytische Maschine“ (*) war das „Nachfolgemodell“ der „Differenzmaschine“ (1821-1833), die arithmetische Berechnungen durchführen können sollte, aber nie funktionierte. Zitat eines Zeitgenossen (L. F. Menabrea, *): “Mr. Babbage has devoted some years to the realization of a gigantic idea. He proposed to himself nothing less than the construction of a machine capable of executing not merely arithmetical calculations, but even all those of analysis, if their laws are known.” Historisches Vorbild waren mit so genannten Jaquard-Lochkarten „programmierbare“ Webstühle (mit bis zu 24000 Karten). Die Rechenmaschine sollte ein „Mill“ genanntes Rechenwerk, ein „Store“ genanntes Speicherwerk (1000 fünfzigstellige Zahlen), Lochkartenleser und -stanzer als Ein- und Ausgabe und einen Drucker als Ausgabe enthalten. Die Maschine sollte

mit Dampf angetrieben und frei programmierbar sein. Ein Programm sollte drei Kartentypen enthalten:



- **Operationskarten**
  enthalten mögliche Operationen: Addition, Subtraktion, Multiplikation und Division. Die Maschine hat einen Schalter für den auszuführenden Operationstyp, der in seiner Stellung bleibt bis er durch eine Operationskarte umgestellt wird.
- **Zahlenkarten**
  enthalten numerische Konstanten und dienen als externer Speicher, damit nicht alle benötigten Zahlen im (teuren) Speicherwerk bereit gehalten werden müssen. Auf einer Zahlenkarte steht jeweils neben dem Wert auch die Nummer des Speichers, in welchen dieser Wert geschrieben werden soll. Zwischenergebnisse können von der Maschine auf Karten gestanzt und später wieder eingelesen werden.
- **Variablenkarten**
  steuern den Transfer von Werten aus dem Store zur Mill und zurück („Adressierung“). Die Maschine besitzt zwei Operandenregister („Ingress-Achsen“, je zweimal 50 Stellen: I_1 und $I_1´$, I_2 und $I_2´$) und ein Resultatregister („Egress-Achse“, zweimal 50 Stellen: E und E´); es gibt Karten zum Transport einer Variable (eines Speicherwerts) in die Ingress-Achsen und zum Transport der Egress-Achse in den Speicher.

Spezielle Karten sind kombinatorische und Indexkarten, die im Kartenstapel vor- und zurückblättern können und somit Sprünge realisieren. Für Verzweigungen gibt es einen „Alarmhebel“, der hochgesetzt wird, falls

- bei einer arithmetischen Operation ein Überlauf oder eine Division durch Null auftritt
- das Ergebnis einer arithmetischen Operation ein anderes Vorzeichen hat als das erste Argument (d.h. Egress-Achse E hat ein anderes Vorzeichen als die Ingress-Achse I_1)

Ferner gibt es Kontrollkarten wie „Stopp“ und „Pause“.
Aus den vorhandenen Dokumenten lässt sich rekonstruieren, dass für die analytischen Maschine folgende Befehle vorgesehen waren (Ausschnitt):

```
<Programm> ::= {Karte}
<Karte> ::= <Zahlkarte> | <Opkarte> | <Varkarte>
<Zahlkarte> ::= “N”[z]_1^3 _ [“+|“-][z]_0^50
```

Die Zahl wird an der bezeichneten Stelle in den Speicher eingetragen

```
<Opkarte> ::= “+” | “-” | “*” | “/”
```

Die Operation wird für nachfolgende Befehle eingestellt

```
<Varkarte> ::= <Transferkarte> | <Kartenkarte> | <Atkarte>
<Transferkarte> ::= (“L”|“Z”|“S”)[z]_1^3[“´”]
```

Lzzz: Transfer des Inhalts der Variable zzz in die Mill Ingress Achse
Zzzz: Wie Lzzz, wobei Variable zzz gleichzeitig auf Null gesetzt wird
Szzz: Transfer der Egress-Achse in Variable zzz
Falls ein ´ nach der Adresse steht, sind die zweiten 50 Stellen betroffen
Transfer in I_2 löst die Ausführung der eingestellten Operation aus

```
<Kartenkarte> ::= “C”(“F”|“B”)(“+”|“?”)[z]_0^50
```

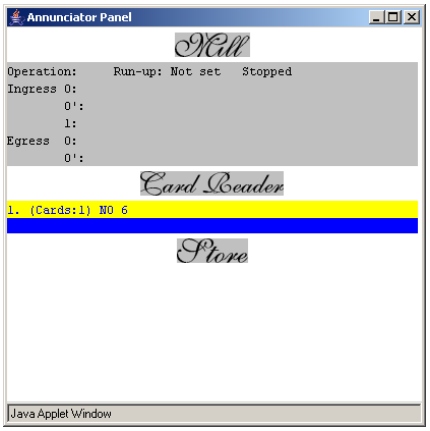
blättert die angegebene Zahl von Karten vor (F) oder zurück (B)
+ bedeutes unbedingtes, ? bedingtes Blättern (falls Alarmhebel hochgesetzt)

<Atkarte> ::= “B” | “H” | “P”

B läutet eine Glocke um den Operateur zu verständigen

H hält die Maschine an (keine weiteren Karten werden gelesen)

P druckt den Wert der Egress-Achse auf dem Druckapparat



Beispiel: drucke 17 + 4

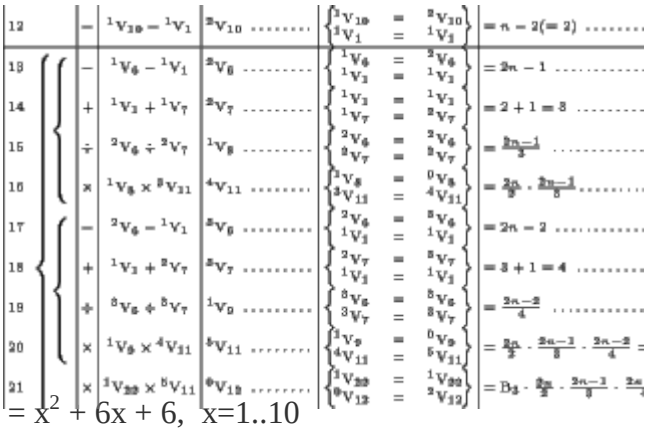
```
N001
+0000000000000000000000000000000000000000000000000017
N002
+0000000000000000000000000000000000000000000000000004
+
L001
L002
P
```

Beispiel: Variable in Speicher 003 := 10000 div 28, Variable 004 := 10000 mod 28,

```
N1 10000
N2 28
/
L1
L2
S3'
S4
```

Beispiel: Fakultätsfunktion S2 := S1!

```
N1 6
N2 1
N3 1
*
L2
L1
S2
-
L1
L3
S1
L3
L1
CB?11
```



Beispiel: drucke Tabelle für $f(x) = x^2 + 6x + 6$, x=1..10

V1 = x

V2 = x^2

V3 = 6

V4 = x^2, x^2+6x, x^2+6x+6

V5 = 6x

```
N1 10
N3 6
*
L1
L1
S4
L1
L3
S5
+
L4
```

```
L5
S4
L4
L3
S4
…
```

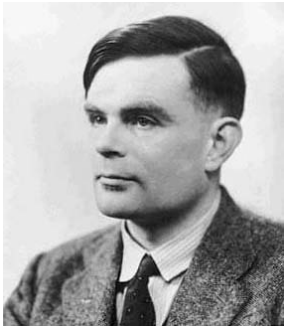
Umformung als $f(x) = (x+3)^2 - 3$ bringt eine Verbesserung:
V1 = x
V2 = 3
V3 = x+3, $(x+3)^2$, $(x+3)^2 - 3$

```
N1 10
N2 3
+
L1
L2
S3
*
L3
L3
S3
-
L3
L2
S3
```

Ada Lovelace schlägt etliche solcher Verbesserungs-Transformationen vor.
Ähnliche (einfachere) Optimierungen heute im Code-Generator guter Compiler enthalten.

Ada Lovelace schreibt: “the Analytical Engine does not occupy common ground with mere `calculating machines´”… “on the contrary, (it) is not merely adapted for *tabulating* the results of one particular function and of no other, but for *developing and tabulating* any function whatever. In fact the engine may be described as being the material expression of any indefinite function of any degree of generality and complexity.“ …
“It may be desirable to explain, that by the word *operation*, we mean *any process which alters the mutual relation of two or more things*, be this relation of what kind it may. This is the most general definition, and would include all subjects in the universe.” …
“Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.“ …

Die Turingmaschine



Nahezu 100 Jahre später (1936) untersuchte Alan Turing die Grenzen des Berechenbaren. Die Arbeit „On Computable Numbers, with an Application to the Entscheidungsproblem“ kann als der Beginn der modernen Informatik betrachtet werden. Turing definierte darin eine hypothetische Maschine, die die „Essenz des Rechnens“ durchführen können soll. Eine „berechenbare“ reelle Zahl ist eine, deren (unendliche Folge von) Nachkommastellen endliche Mittel (d.h. durch einen Algorithmus) berechnet werden kann. Turing schreibt:

„Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. …
The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused.“

Eine Turingmaschine ist also gegeben durch

- einen endlichen Automaten zur Programmkontrolle, und
- ein (potentiell unbegrenztes) Band auf dem die Maschine Zeichen über einem gegebenen Alphabet notieren kann.

Zu jedem Zeitpunkt kann die Maschine genau eines der Felder des Bandes lesen (abtasten, „scannen“), und, ggf.. abhängig von der Inschrift dieses Feldes, das Feld neu beschreiben, zum linken oder rechten Nachbarfeld übergehen, und einen neuen Zustand einstellen.
Hier ist eine Syntax, die Turings Originalschreibweise nahe kommt.

```
<Turingtabelle> ::= {<Zeile>}
<Zeile> ::= <Zustand> <Abtastzeichen> <Operation> <Zustand>
<Zustand> ::= <Identifier>
<Abtastzeichen> ::= <Zeichen>
<Operation> ::= {R | L | P<Symbol>}
```

Dabei wird angenommen, dass für jeden Zustand und jedes Abtastzeichen des Alphabets genau eine Zeile der Tabelle existiert, welche Operation und Nachfolgezustand festlegt. (Turing nennt solche Maschinen “automatisch”, wir nennen sie heute “deterministisch”. In Turings Worten: „If at any stage the motion of a machine is completely determined by the

configuration, we shall call the machine an ‚automatic machine' ... For some purposes we might use machines (choice machines or c-machimes) whose motion is only partially determined by the configuration… When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator… In this paper I deal only with automatic machines.'')
Da die Tabellen für deterministische Turingmaschinen oft sehr groß werden, darf man Zeilen, die nicht benötigte werden, weglassen, und Zeilen, die sich nur durch das Abtastzeichen unterscheiden, zusammenfassen. In der entsprechenden Zeile sind beliebige Mengen von Abtastzeichen zugelassen. „any" steht dann für ein beliebiges Abtastzeichen, d.h. für das gesamte Alphabet. Ferner fordert Turing, dass das Alphabet immer ein spezielles Leerzeichen „none" enthält, und „E" eine Abkürzung für „P none" ist..

Beispiel für eine Maschine, die das Muster „0 11 0 11 0 11…" auf ein leeres Band schreibt:

```
s0  any  P0,R,R  s1
s1  any  P1,R,P1,R,R  s0
```

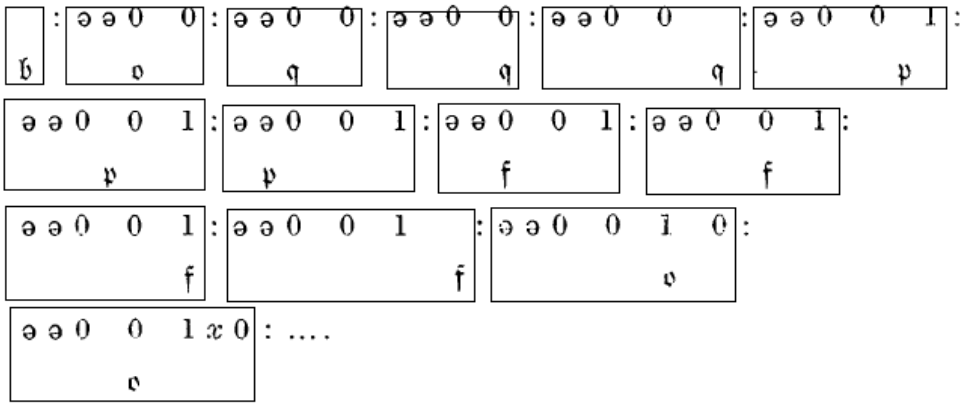
Ein äquivalentes Programm mit nur einem Zustand s0 ist

```
s0 none P0 s0
s0 0 R,R,P1,R,P1 s0
s0 1 R,R,P0 s0
```

Beispiel für eine Maschine, die die Sequenz 0 01 011 0111 01111 … erzeugt:

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
b		Pə, R, Pə, R, $P0$, R, R, $P0$, L, L	o
o	1	R, Px, L, L, L	o
	0		q
q	Any (0 or 1)	R, R	q
	None	$P1$, L	p
p	x	E, R	q
	ə	R	f
	None	L, L	p
f	Any	R, R	f
	None	$P0$, L, L	o

Arbeitsweise dieser Maschine:



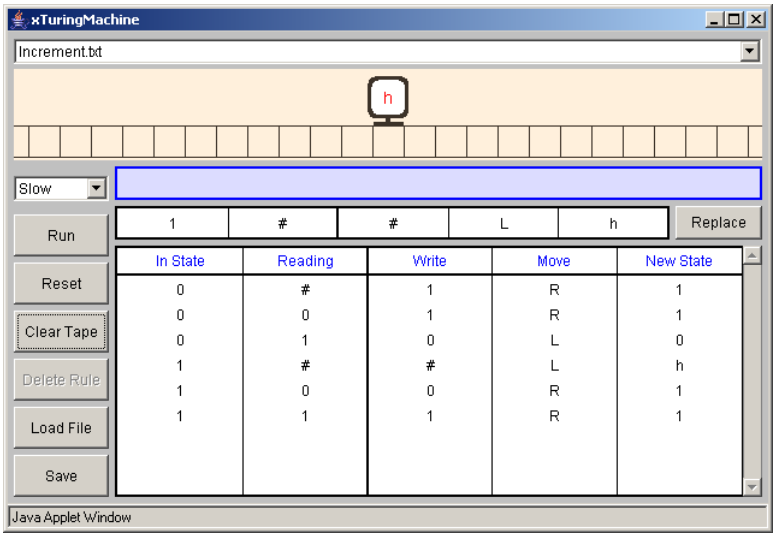
Die von Turing verwendete Tabellenschreibweise für Programme betrachten wir heute als unleserlich. Eine moderne Variante („Turing-Assembler“) wäre etwa:

```
<Turingprogram> ::= {<statement>}
<statement> ::= <label>“:” | “print” <symbol> “;” |
                “left;” | “right;” | “goto” <label>“;”
<label> ::= <Identifier>
```

In dieser Notation sähe unser Beispielprogramm etwa so aus:

```
label0:
print 0;
right;
right;
print 1;
right;
print 1;
right;
right;
goto s0;
```

Im Internet sind viele Turingmaschinen-Simulatoren verfügbar, versuchen Sie z.B. http://math.hws.edu/TMCM/java/labs/xTuringMachineLab.html



Andere empfohlene Beispiele:
http://www.matheprisma.uni-wuppertal.de/Module/Turing/
http://ais.informatik.uni-freiburg.de/turing-applet/turing/TuringMachineHtml.html

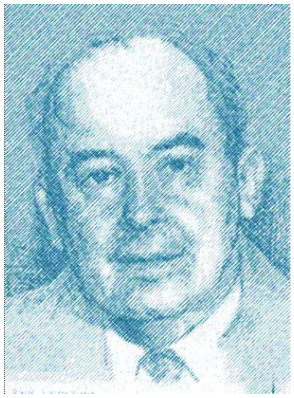
Zuse Z3

erster voll funktionsfähiger programmierbarer Digitalrechner
viele Merkmale moderner Rechner:

- Relais-Gleitkommaarithmetikeinheit für Arithmetik
- einem Relais-Speicher aus 64 Wörtern, je 22 bit
- einem Lochstreifenleser für Programme auf Filmstreifen
- eine Tastatur mit Lampenfeld für Ein- und Ausgabe von Zahlen und der manuellen Steuerung von Berechnungen.
- Taktung durch Elektromotor, der Taktwalze antreibt (5rps)

Programmiersprache: Plankalkül
http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Compiler/plankalk.html

Von-Neumann-Rechner, EDVAC & ENIAC



John von Neumann wurde vor hundert Jahren (im Dezember 1903) in Budapest geboren. 1929 wurde er als jüngster Privatdozent in der Geschichte der Berliner Universität habilitiert. Von Neumann wurde binnen kurzer Zeit weltberühmt durch seine vielfältigen Interessen auf dem Gebieten Mathematik, Physik und Ökonomie. 1930 emigrierte er wegen der Nazis nach USA. In Princeton schuf von Neumann dann mit dem von ihm erdachten Rechnerkonzept die Grundlagen für den Aufbau elektronischer Rechenanlagen, die noch bis heute gültig sind. Er gilt daher als einer der Begründer der Informatik

EDVAC, ENIAC (Electronic Numerical Integrator and Computer)

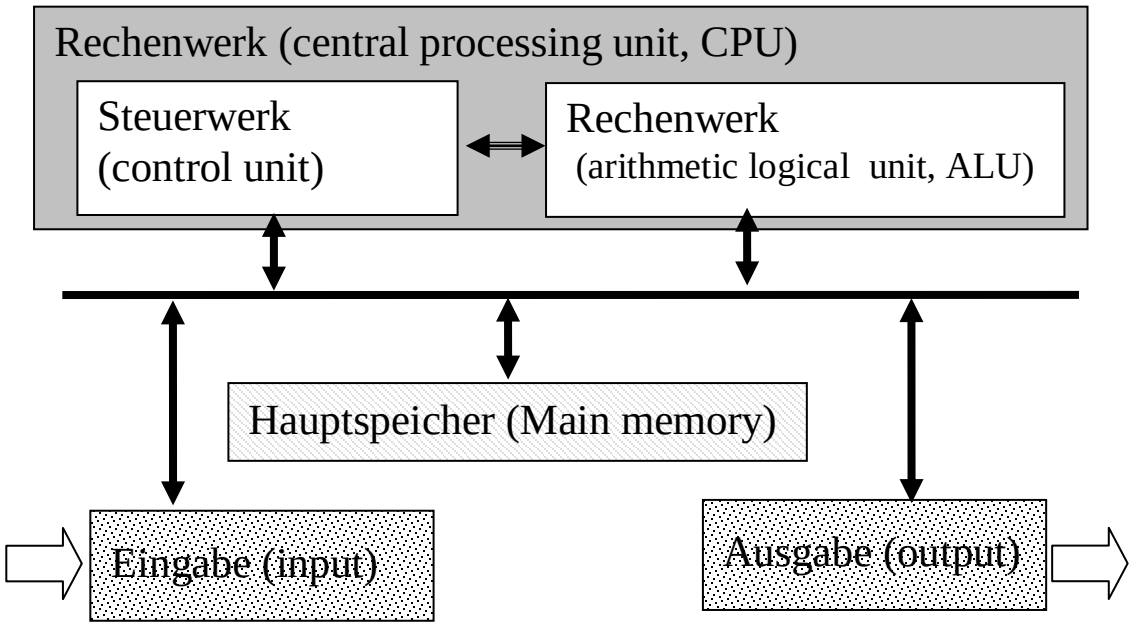
- Elektronenröhren zur Repräsentation von Zahlen
- elektrische Pulse für deren Übertragung
- Dezimalsystem
- Anwendung: H-Bomben-Entwicklung
- Programmierung durch Kabel und Drehschalter

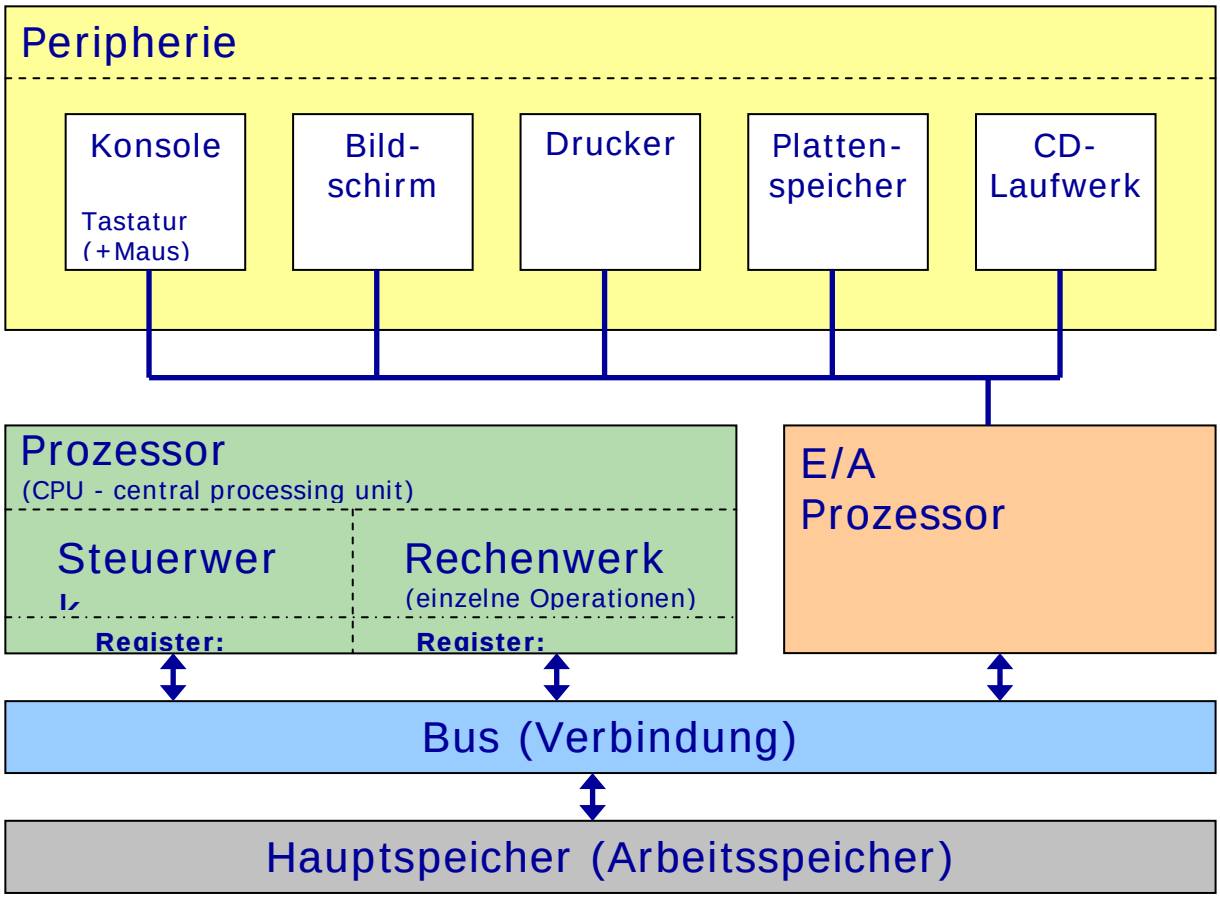
3.2 von-Neumann-Architektur

1945 *First Draft of a Report on the EDVAC* (Electronic Discrete Variable Automatic Computer): Befehle des Programms werden wie zu verarbeitenden Daten behandelt, binär kodiert und im internen Speicher verarbeitet (vgl. Zuse, Turing)

Ein von-Neumann-Computer enthält mindestens die folgenden fünf Bestandteile:

1. ***Input unit*** (kommuniziert mit der Umgebung)
2. ***Main memory*** (Speicher für Programme und Daten)
3. ***Control unit*** (führt die Programme aus)
4. ***Arithmetic logical unit*** (für arithmetische Berechnungsschritte)
5. ***Output unit*** (kommuniziert mit der Umgebung)



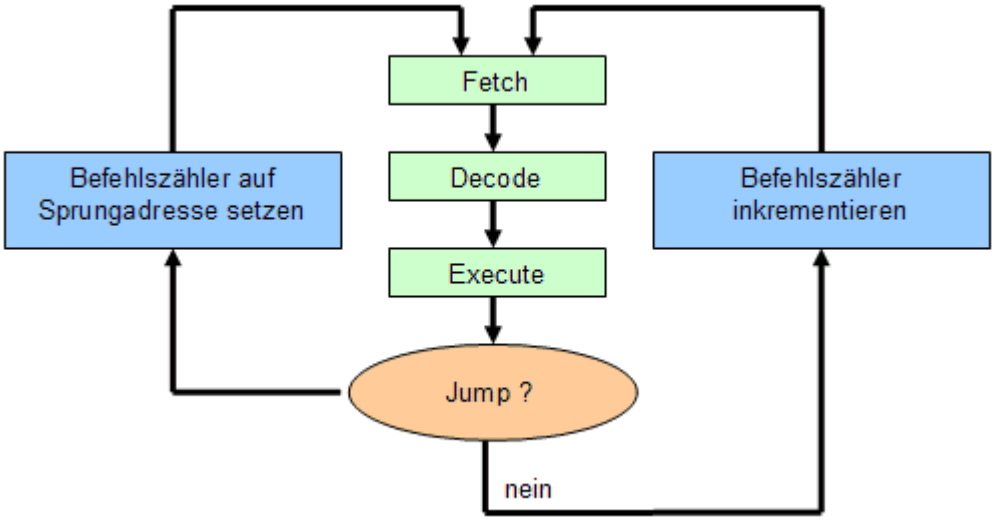


Prinzipien:

- Der Rechner enthält zumindest Speicher, Rechenwerk, Steuerwerk und Ein/Ausgabegeräte. „***EVA-Prinzip***“: Eingabe – Verarbeitung – Ausgabe
- Der Rechner ist ***frei programierbar***, d.h., nicht speziell auf ein zu bearbeitendes Problem zugeschnitten; zur Lösung eines Problems wird ein Programm im Speicher ablegt. Dadurch ist jede nach der Theorie der Berechenbarkeit mögliche Berechnung programmierbar.
    - Programmbefehle und Datenworte liegen im selben Speicher und werden je nach Bedarf gelesen oder geschrieben.
    - Der Speicher ist unstrukturiert; alle Daten und Befehle sind binär codiert.
    - Der Speicher wird linear (fortlaufend) adressiert. Er besteht aus einer Folge von Plätzen fester Wortlänge, die über eine bestimmte Adresse einzeln angesprochen werden können und bit-parallel verarbeitet werden.
    - Die Interpretation eines Speicherinhalts hängt nur vom aktuellen Kontext des laufenden Programms ab. Insbesondere: Befehle können Operanden anderer Befehle sein (Selbstmodifikation)!
- Der Befehlsablauf wird vom ***Steuerwerk*** bestimmt. Er folgt einer sequentiellen Befehlsfolge, streng seriell und taktgesteuert.
    - Zu jedem Zeitpunkt führt die CPU nur einen einzigen Befehl aus, und dieser kann (höchstens) einen Datenwert verändern (single-instruction-single-data).
    - Die normale Verarbeitung der Programmbefehle geschieht fortlaufend in der Reihenfolge der Speicherung der Programmbefehle. Diese sequentielle Programmabarbeitung kann durch Sprungbefehle oder datenbedingte Verzweigungen verändert werden.

- Die ***ALU*** führt arithmetische Berechnungen durch, indem sie ein oder zwei Datenwerte gemäß eines Befehls verknüpft und das Ergebnis in ein vorgegebenes Register schreibt. Zwei-Phasen-Konzept der Befehlsverarbeitung:
  - In der Befehlsbereitstellungs- und Decodierphase-Phase wird, adressiert durch den Befehlszähler, der Inhalt einer Speicherzelle geholt und als Befehl interpretiert.
  - In der Ausführungs-Phase werden die Inhalte von einer oder zwei Speicherzellen bereitgestellt und entsprechend den Opcode als Datenwerte verarbeitet.
- Datenbreite, Adressierungsbreite, Registeranzahl und Befehlssatz als Parameter der Architektur
- ***Ein- und Ausgabegeräte*** sind z.B. Schalter und Lämpchen, aber auch entfernte Speichermedien (Magnetbänder, Lochkarten, Platten, …). Sie sind mit der CPU prinzipiell auf die selbe Art wie der Hauptspeicher verbunden.

Zentrale Befehlsschleife (aus *):



Vergleiche: Assemblersprache, Ausführung eines Befehles

Vor- und Nachteile der von-Neumann-Architektur:

+ minimaler Hardware-Aufwand, Wiederverwendung von Speicher
+ Konzentration auf wesentliche Kennzahlen: Speichergröße, Taktfrequenz
- Verbindungseinrichtung CPU – Speicher stellt einen Engpass dar („von-Neumann-Flaschenhals“)
- keine Strukturierung der Daten, Maschinenbefehl bestimmt Operandentyp

„von-Neumann-bottleneck“ John Backus, Turing-Award-Vorlesung 1978:

> When von Neumann and others conceived it [the von Neumann computer] over thirty years ago, it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed

radically, we nevertheless still identify the notion of "computer" with this thirty [jetzt fast sechzig] year old concept.
In its simplest form, a von Neumann computer has three parts" a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the store in a major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Wie oben erwähnt, sind auch heute noch die meisten Computer nach der von-Neumann-Architektur konstruiert. **Beispiel:** Architektur des Intel Pentium-Prozessors (*):

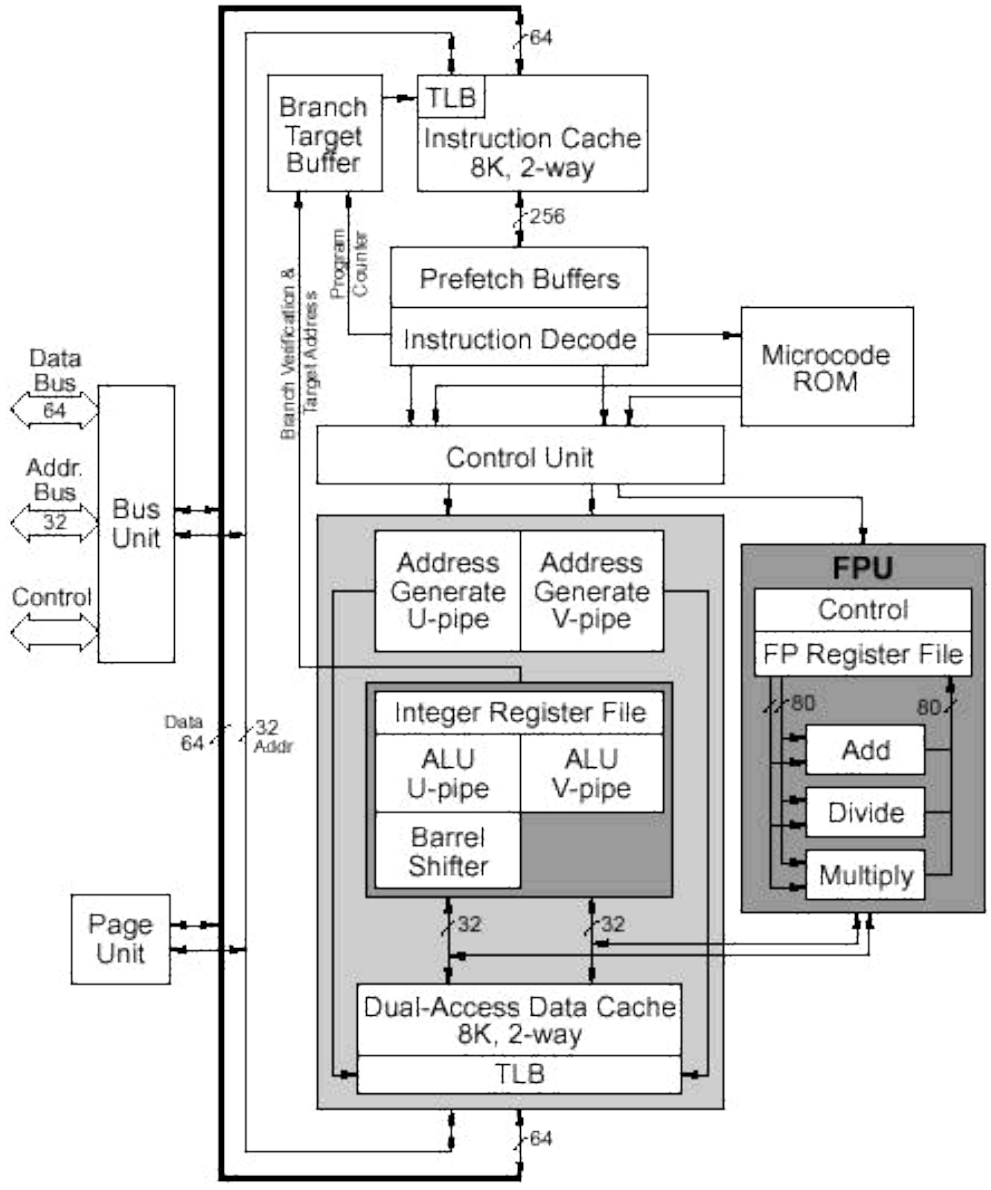
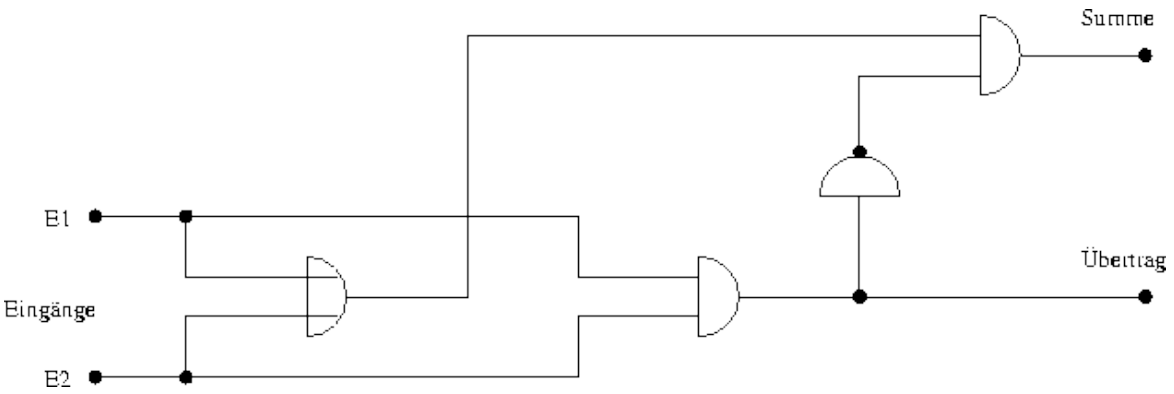


Figure 1. Pentium block diagram.

Realisierung der einzelnen Baugruppen durch Mengen von Halbleiterschaltern; z.B. eines Halbaddierers (Summe= E1 XOR E2, Übertrag=E1 AND E2):



Beschreibung der Hardware auf verschiedenen Ebenen: Physikalische Ebene, Transistor-Ebene, Gatter-Ebene (s.o.), Register-Ebene, Funktionsebene (siehe TI)

Abweichungen und Varianten der von-Neumann-Architektur:

- Spezialisierte Eingabe-Ausgabe-Prozessoren
  - z.B. Grafikkarte mit 3D-Rendering
  - z.B. Modem oder Soundkarte zur Erzeugung von Tönen
  - z.B. Tastatur-, Netzwerk- oder USB-Controller, die auf externe Signale warten
- Parallelität zwischen/innerhalb von Funktionseinheiten, z.B.
  - Blocktransfer von Daten
  - Pipelining in CPU
- Duplikation von Funktionseinheiten
  - z.B. Mehrprozessorrechner, **Mehrkern-Architektur:** Zwei oder mehr CPU auf einem Chip, Kopplung durch speziellen Memory-Control-Bus; z.B. Pentium Extreme Edition 840 (April 2005), Preis 999 Dollar, „dürfte allerdings nur wenig Käufer finden"; Aktuell: Quadcore, Octocore (Sun UltraSparc, Intel Nehalem 2008);
- komplexere Speicherstrukturen, z.B.
  - Register (einzelne Speicherwörter direkt in der CPU)
  - *Caches* (schnelle Pufferspeicher) und *Bus-Hierarchien* für Verbreiterung des Flaschenhalses
  - Harvard-Architektur (Trennung von Daten- und Befehlsspeicher)
- komplexere Verbindungsstrukturen
  - externe Standardschnittstellen: IDE, SCSI, USB, IEEE1394/Firewire/iLink
  - ISA/PCI/AGP: hierarchischer bzw. spezialisierter Aufbau des Bussystems
- komplexere Befehle, z.B.
  - mehrere Operanden
  - indirekte Adressierung („Adresse von Adresse")
  - CISC versus RISC
- Programmunterbrechung durch externe Signale
  - *Interrupt*-Konzept
  - Mehrbenutzer-Prozesskonzept

3.3 Aufbau PC/embedded system, Speicher

Heutige PCs sind meist prinzipiell nach der von-Neumann-Architektur , mit o.g. Erweiterungen, konstruiert. Beispiel (Gumm/Sommer p55)

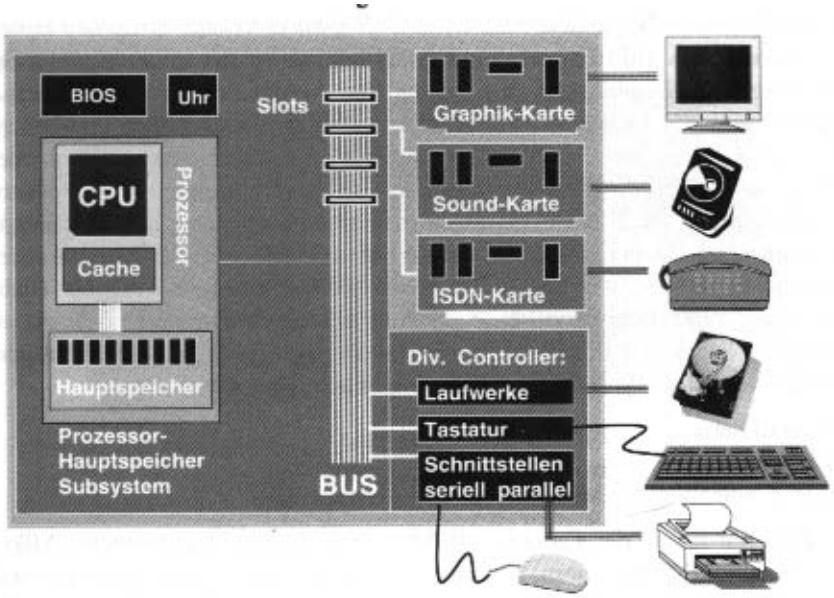


Abb. 1-11: Überblick über die Komponenten eines Rechners

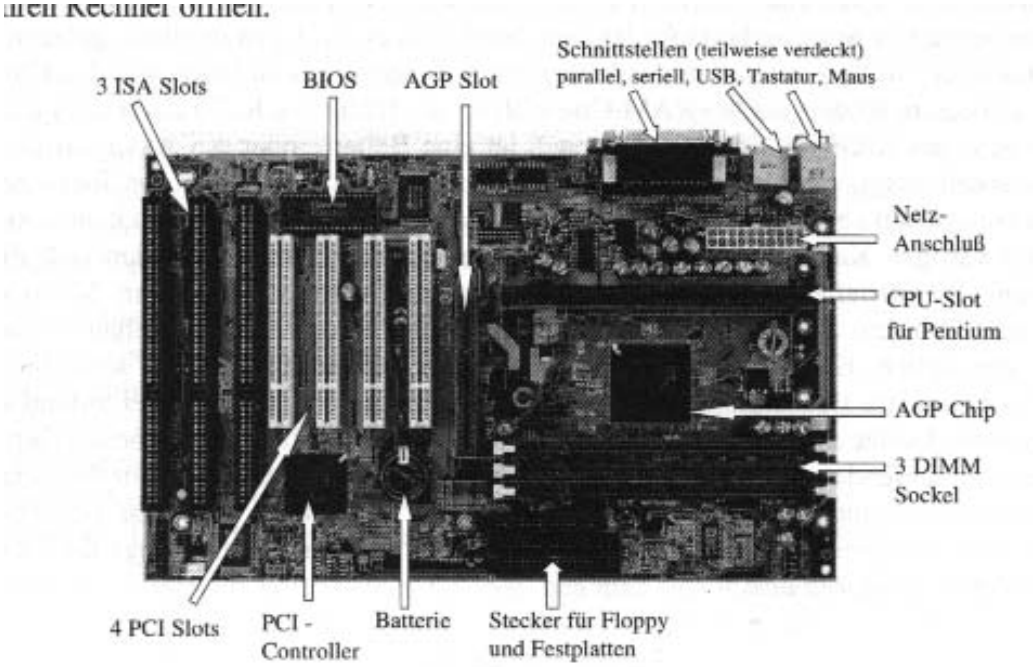
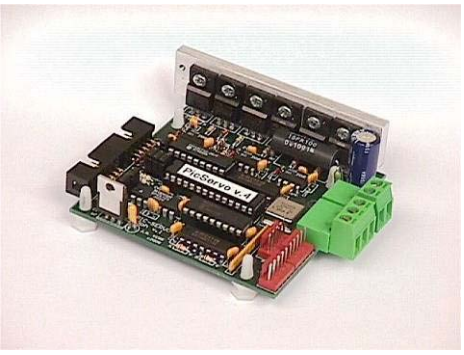


Abb. 1-12: Ein Motherboard

Prinzipiell ist dieser Aufbau auch in den meisten eingebetteten Steuergeräten zu finden. Beispiel: ein Steuergerät im Audi quattro, welches zwei separate Prozessoren für Zündung und Ladung enthält, und ein seriell einstellbares Drehzahlsteuergerät für Elektromotoren.



Unterschiede zur „normalen“ Rechnerarchitektur:

- Ein- und Ausgabegeräte sind Sensoren und Aktuatoren
- Wandlung analoger in digitale Signale und umgekehrt auf dem Chip (A/D D/A)
- Der Speicher ist oft nichtflüchtig und manchmal mit dem Prozessor integriert
- Meist wird nur wenig Datenspeicher benötigt, der Programmspeicher wird nur bei der Produktion oder Wartung neu beschrieben (➔ andere Speicherkonzepte)
- Hohe Stückzahlen verursachen Ressourcenprobleme (Speicherplatz)
- Oft komplizierte Berechnungen mit reellen Zahlen (DSP, digitale Signalprozessoren), spezialisierte ALUs für bestimmte numerische Algorithmen.

Speicher

So groß wie ein Zweimarkstück: Auf die 16 Gramm leichte Mini-Festplatte von IBM passt 1 Gigabyte an Daten.

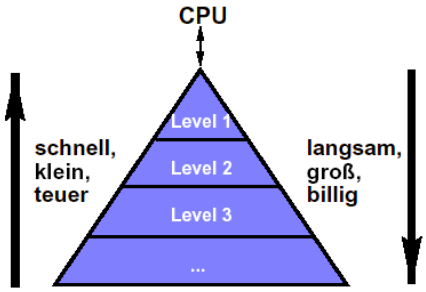
Speicher dienen zur temporären oder permanenten Aufbewahrung von (binär codierten) Daten. Sie können nach verschiedenen Kriterien klassifiziert werden

- *Permanenz*: flüchtig – nichtflüchtig (bei Ausfall der elektrischen Spannung)
  - Halbleiterspeicher sind meist flüchtig, magnetische / optische Speichermedien nicht.
  - So genannte Flash Speicher sind nichtflüchtige Halbleitermedien, die elektrisch beschrieben und gelöscht werden können. Bei Flash-Speichern ist nicht jedes einzelne Bit adressierbar, die Zugriffe erfolgen auf Sektorebene ähnlich wie bei Festplatten. Vorteil: keine mechanisch bewegten Teile.
- *Geschwindigkeit*: Zugriffszeit (Taktzyklen oder ns) pro Wort (mittlere, maximale)
  - Gängige Zugriffszeiten liegen bei 1-2 Taktzyklen für Register in der ALU, 2-50 ns für einen schnellen Cache, 100-300 für einen Hauptspeicherzugriff, 10-50ms für einen Plattenzugriff, 90 ms für eine CD-ROM, Sekundenbereich für Floppy Disk, Minutenbereich für Magnetbänder
- *Preis*: Cent pro Byte

- Ein 512 MB Speicherbaustein oder CompactFlash kostet 100 Euro ($2*10^{-5}$ c/B=20c/MB), eine 120GB Festplatte etwa genauso viel ($8.3*10^{-8}$ c/B =83c/GB), ein 700MB CD-Rohling 25 cent ($3.5*10^{-8}$c/B=35c/GB)

- *Größe*: gemessen in cm^2 oder cm^3 pro Bit
  - Papier: 6000 Zeichen/630cm²=10B/cm²; Floppy: 1.44MB/ 80cm^2=18 KB/cm^2; Festplatte: 10-20 GB/cm^2, physikalisch machbar 1000GB/cm^2 m
  - für mechanisch bewegte Speicher muss der Platz für Motor und Bewegungsraum mit berücksichtigt werden.

Wie man sieht, sind Geschwindigkeit und Preis umgekehrt proportional. Üblicherweise wird der verfügbare Speicherplatz daher hierarchisch strukturiert. Das Vorhalten von Teilen einer niedrigeren Hierarchieebene in einer höheren nennt man ***Caching***.



Cache („Geheimlager“): kleiner, schneller Vordergrundspeicher, der Teile der Daten des großen, langsamen Hintergrundspeichers abbildet („spiegelt“).

Konzept: Hauptspeicher = Folge von Tupeln (Adresse, Inhalt)

Cache-Speicher = Folge von Quadrupeln *Cache-Zeilen:*

(Index, Statusbit, Adresse, Inhalt)

Index: Adresse im Cache

Statusbits: modifiziert?, gültig?, exklusiv?, …

Adresse: Speicherzelle die gespiegelt wird

Falls die CPU ein Datum dat einer bestimmten Adresse adr benötigt, wird zunächst geprüft, ob es im Cache ist (d.h. (idx, sbt, adr, dat) im Cache). Zwei Fälle:

Cache Hit: D.h. (idx, sbt, adr, dat) im Cache gespiegelt: dat wird

Cache Miss: kein idx mit (idx, adr, …) im Cache. Die Speicherzelle adr mit Inhalt dat muss aus dem Hauptspeicher nachgeladen werden; ggf. muss dafür ein bereits belegter Platz im Cache geräumt werden (Verdrängungsstrategie)

SPEICHER

	…
3786:	17
3787:	"c"
3788:	3.1415
3789:	3786
3790:	"x"
3791:	123456
3792:	3790
3793:	NIL
	…

CACHE

c1:	0	1	0	3787	"c"
c2:	1	1	0	3788	3.1415
c3:	0	1	0	3792	3790

c1:	0	1	0	3790	"x"
c2:	1	1	0	3788	3.1415
c3:	0	1	0	3792	3790

Falls jede Hintergrundadresse prinzipiell in jede Cache-Zelle geladen werden kann, ist der Cache *assoziativ (fully associative)*.

Vorteil: Flexibilität.

Nachteil: gesamter Cache muss durchsucht werden, ob Hit oder Miss.

Falls eine Cache-Zelle nur bestimmte Hintergrundadressen abbilden kann, sagen wir der Cache ist *direkt abgebildet (direct mapped)*

Beispiel: C0 für H00, H10, H20, …, C1 für H01, H11, H21, …Suche H63 nur in C3!

Vorteil: schnelle Bestimmung ob Hit oder Miss; Nachteil: Unflexibilität

Hitrate ist entscheidend für Leistungssteigerung durch Cache; Verdrängungsstrategie beeinflusst Hitrate entscheidend

- LRU: Least recently used
- FIFO: First-In, First-Out
- LFU: Least frequently used

Cache Coherency Problem: Mehrere Prozesse greifen auf Speicher zu, jeder Prozess hat einen eigenen Cache: Wie wird die Konsistenz sichergestellt?