IICM

TU Graz

706.088 INFORMATIK 1

Fehlerbehandlung, Funktionsweise eines Computers

- Wiederholung:
  - Binärzahlen
  - Binär rechnen
- Fehler-behandlung in Python
- Funktionsweise eines Computers

BINÄRZAHLEN

BINÄR RECHNEN

- addieren
- subtrahieren
  - 2er Komplement
- multiplizieren
- dividieren

BINÄR RECHNEN

- Bit Operatoren
  - shiften
  - AND
  - OR
  - XOR

FEHLER-BEHANDLUNG IN PYTHON

FEHLERBEHANDLUNG IN PYTHON

- Ausnahmebehandlung, engl: **Exception Handling**
- Vereinfacht Fehlerbehandlung durch speziellen Mechanismus
- Rückgabewerte von Funktionen können für ordentlichen Programmablauf verwendet werden
- Fehler können strukturiert behandelt werden

EXCEPTION HANDLING

- Fehler 'wirft' eine *Exception* (Objekt) nach 'oben', Funktion ist beendet.
- Übergeordnete Funktion kann:
  - fangen, fortfahren
  - fangen, weiterwerfen, Funktion ist beendet
  - lässt passieren, Funktion ist beendet

EXCEPTION OBJEKT

Enthält Atribute und Methoden (Funktionen) zur Klassifizierung des Fehlers

```
>>> e = Exception("My custom error")
>>> e.args
('My custom error',)
>>> e = Exception("My custom error","test", 1,2)
>>> e.args
('My custom error', 'test', 1, 2)
```

EXCEPTION WERFEN

```
>>> raise Exception("My Exception")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: My Exception
```

EXCEPTION WERFEN

Nur BaseException und davon Abgeleitete dürfen geworfen werden

```
>>> raise "test"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must derive from BaseException
```

EINGEBAUTE EXCEPTIONS

- BaseException
- Exception (Basisklasse für Benutzer)
- SyntaxError
- NameError
- TypeError
- ImportError
- ...

EXCEPTION BEHANDLUNG

- *try* öffnet den Try-Block
- Exceptions aus dem Try-Block werden im Except-Block gefangen
- *except* definiert welche Exceptions behandelt werden

EXCEPTION FANGEN

```
>>> open("/tmp/non_existing_file",'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/non_ex
```

```
try:
    open("/tmp/non_existing_file")
except OSError as e:
    print("Caught:", e)

Caught: [Errno 2] No such file or directory: '/tmp/non_existing_file
```

TRY - EXCEPT - ELSE

```
try:
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")
```

EXCEPT - ELSE

```
try:
    print("all good")
    open("/tmp/non_existing_file")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
all good
Don't know this error!
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/non_ex
normal program flow
```

EXCEPT - ELSE

```
try:
    print(a) # undefined!
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
Undefined vars found
normal program flow
```

EXCEPT - ELSE

```
try:
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
all good
everything is fine
normal program flow
```

FINALLY

```
try:
  open("/tmp/non_existing_file",'r')
except FileNotFoundError:
  print("file does not exist")
except:
  print("don't know this error")
  raise
finally:
  print("cleaning up")
```

```
file does not exist
cleaning up
```

ASSERT

- Setzt Bedingung, die, wenn falsch, zu einer Exception führt.
- Nur zur Entwicklung sinnvoll.
- Nur mit `__debug__ == True` aktiv
- Wird mit `python3 -O` deaktiviert (`__debug__` = False)

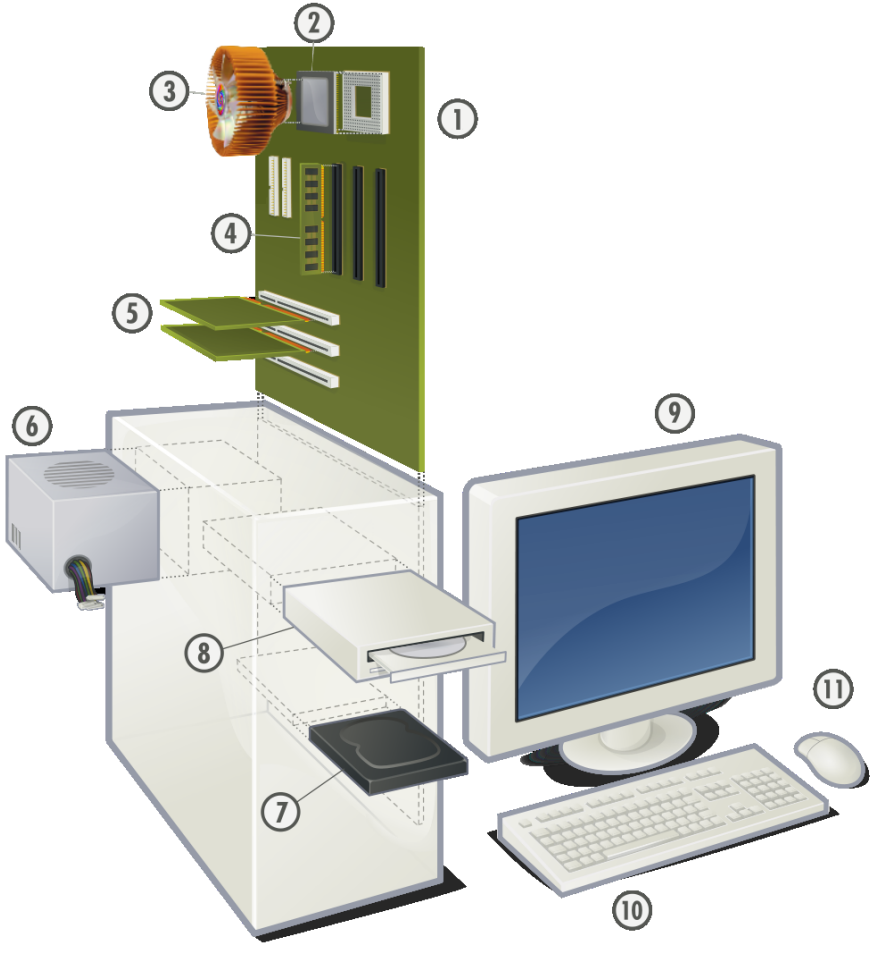
ASSERT

```
a = [1,2]
a[0] = 17
assert a == [17,2]
```

```
a[1] = a[1] + 3
assert a == [17,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

AUFBAU EINES COMPUTERS

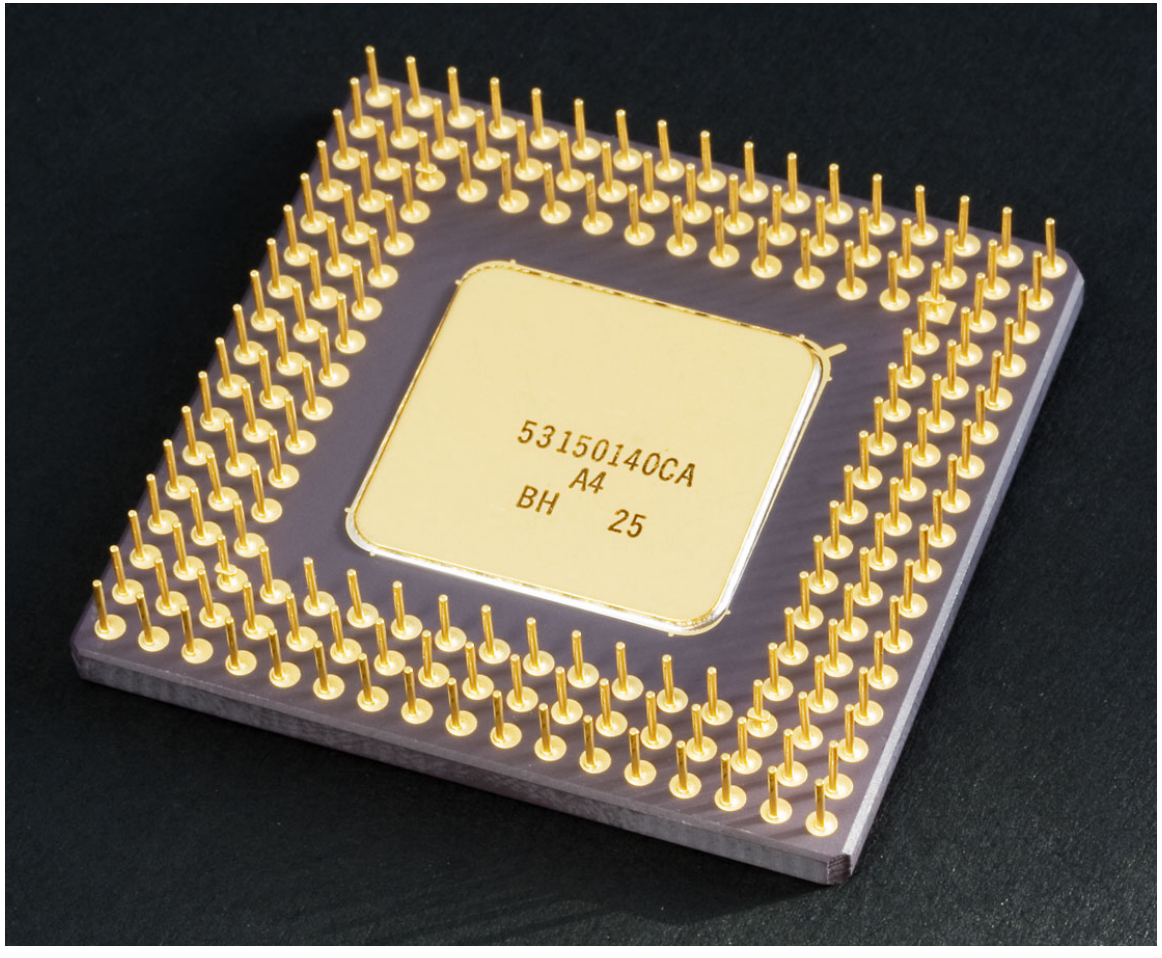
AUFBAU



Symbolbild, CC BY 2.5, Quelle

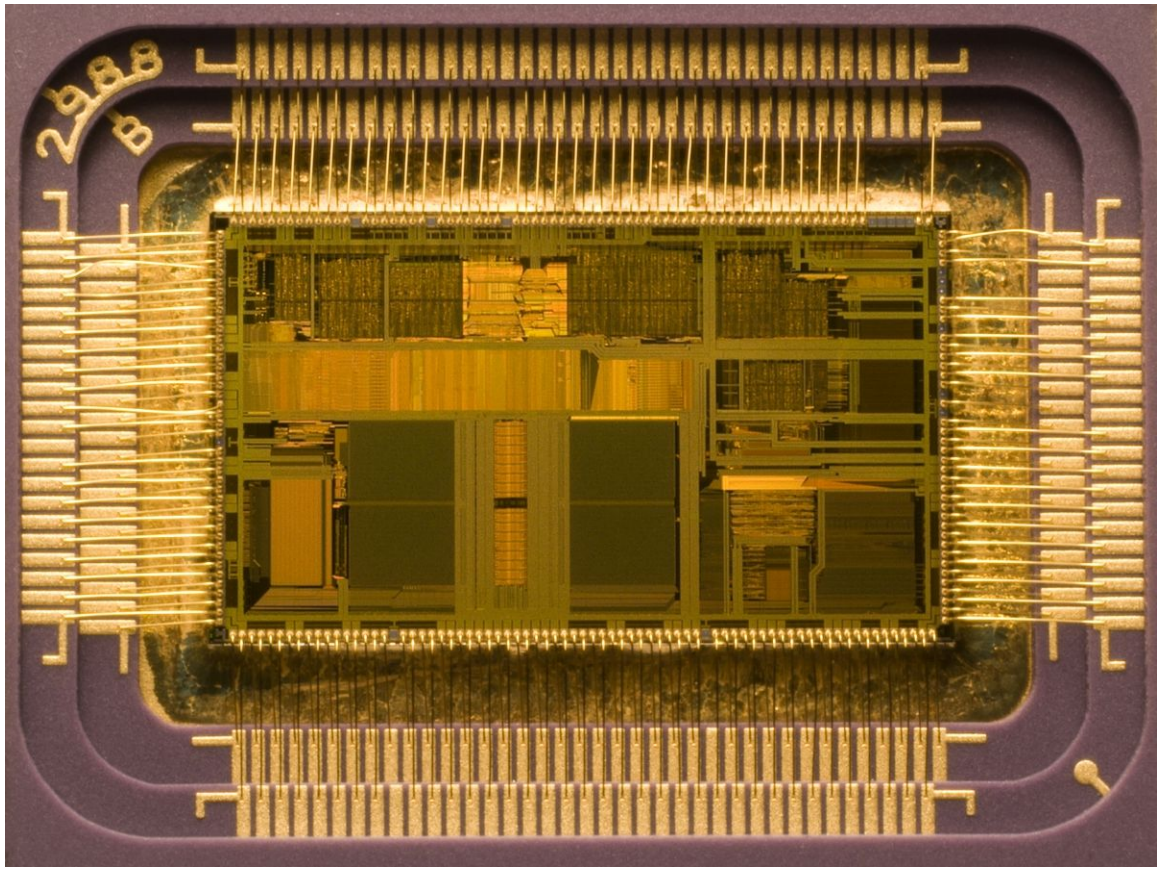
PROZESSOR - CPU

Central Processing Unit



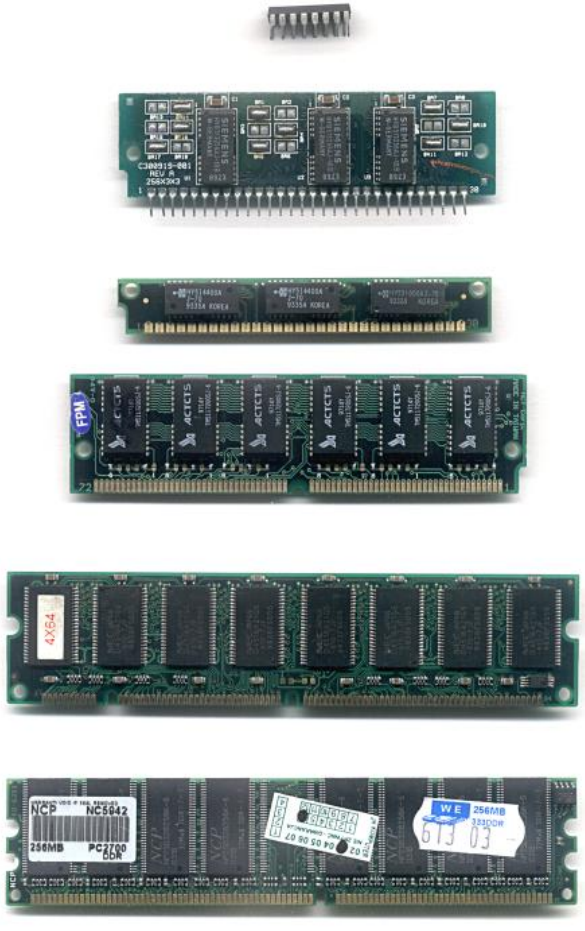
Intel 80486DX2, CC BY-SA 2.0, Link

Innenleben Intel 80486DX2



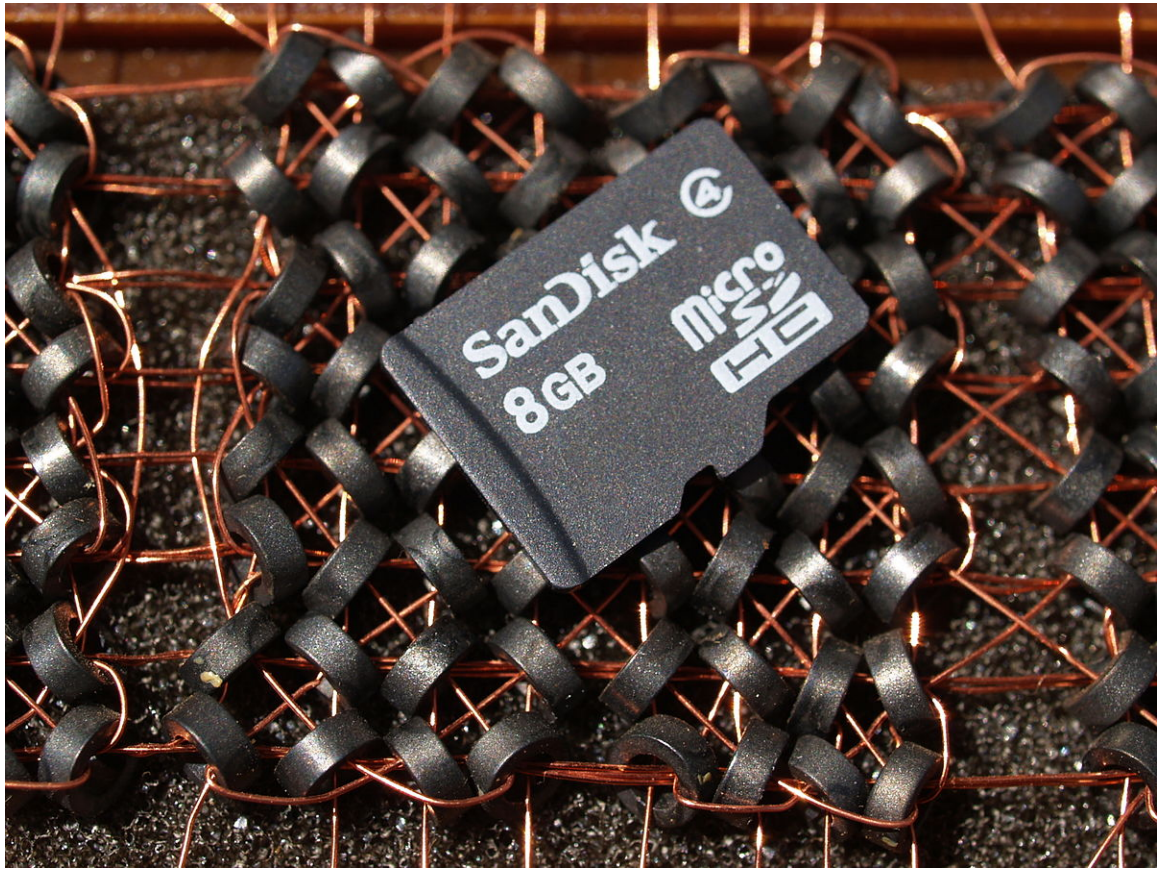
Von Uberpenguin aus der englischsprachigen Wikipedia, CC BY-SA 3.0, Link

ARBEITSSPEICHER - RAM



ARBEITSSPEICHER - RAM

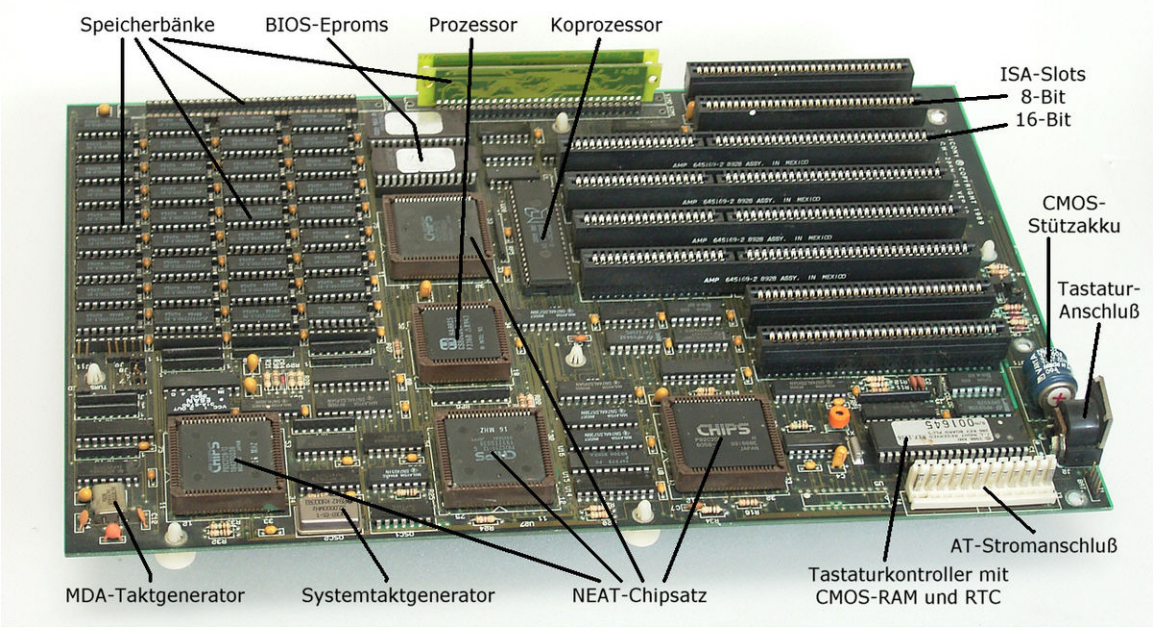
Magnetic Core Memory, 1947



By Daniel Sancho from Málaga, Spain - 8 bytes vs. 8Gbytes, CC BY 2.0, Quelle

MAINBOARD - HAUPTPLATINE

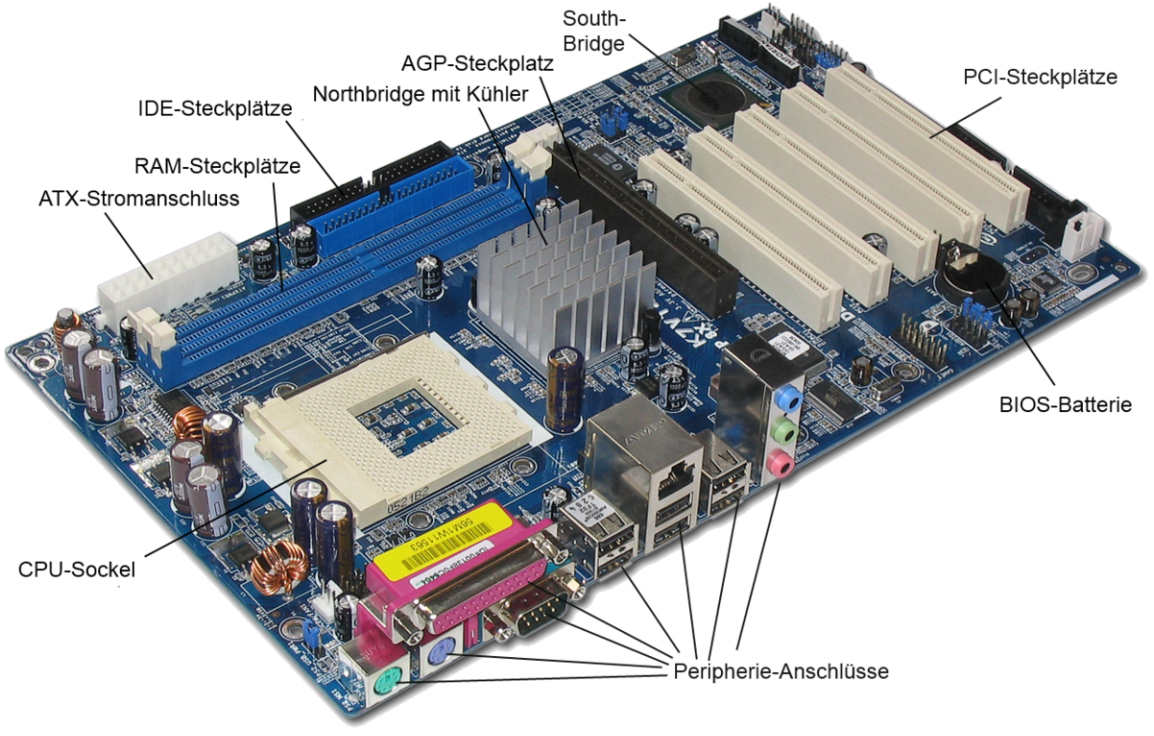
1980



Von User Smial on de.wikipedia - Eigenes Werk, CC BY-SA 2.0 de, Link

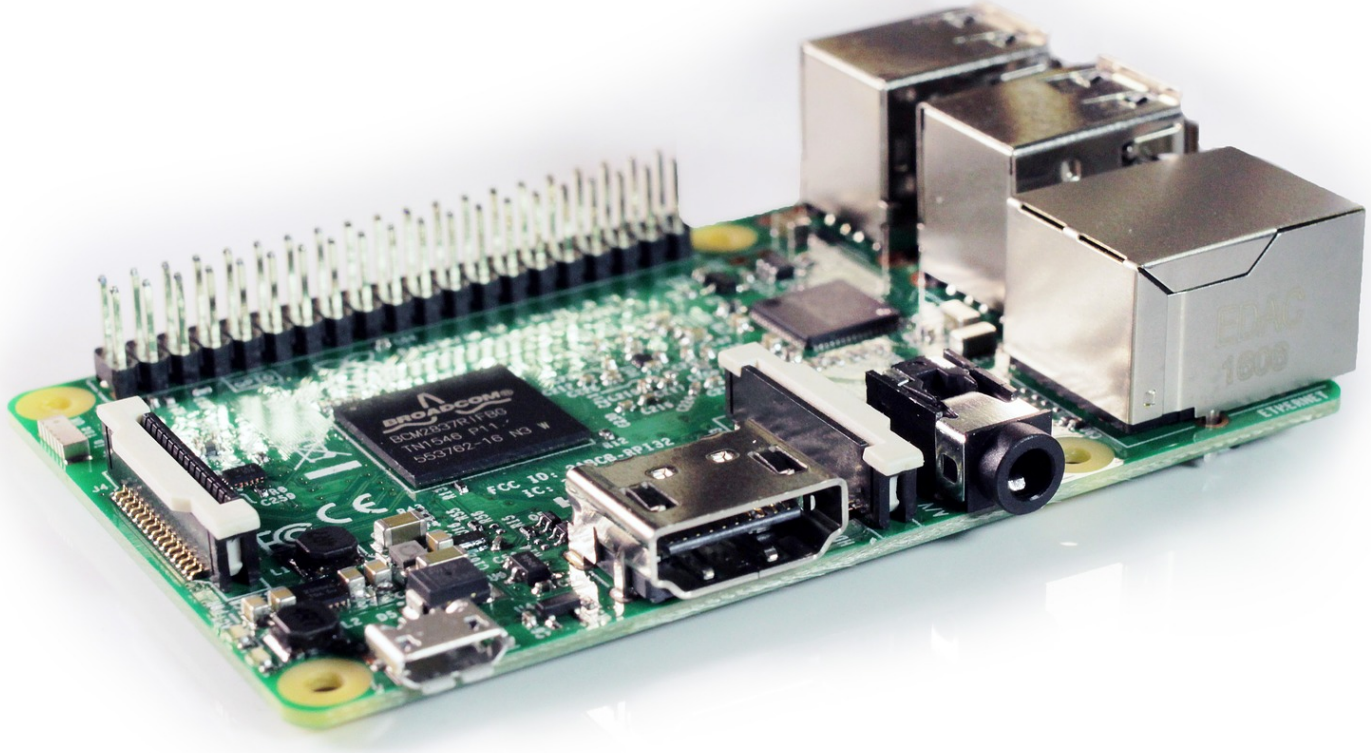
MAINBOARD - HAUPTPLATINE

2004



Von Freddy2001 Description by User:leipnizkeks – released under same license. - Eigenes Werk, CC BY-SA 2.5, Link

MAINBOARD - HAUPTPLATINE



FESTPLATTEN

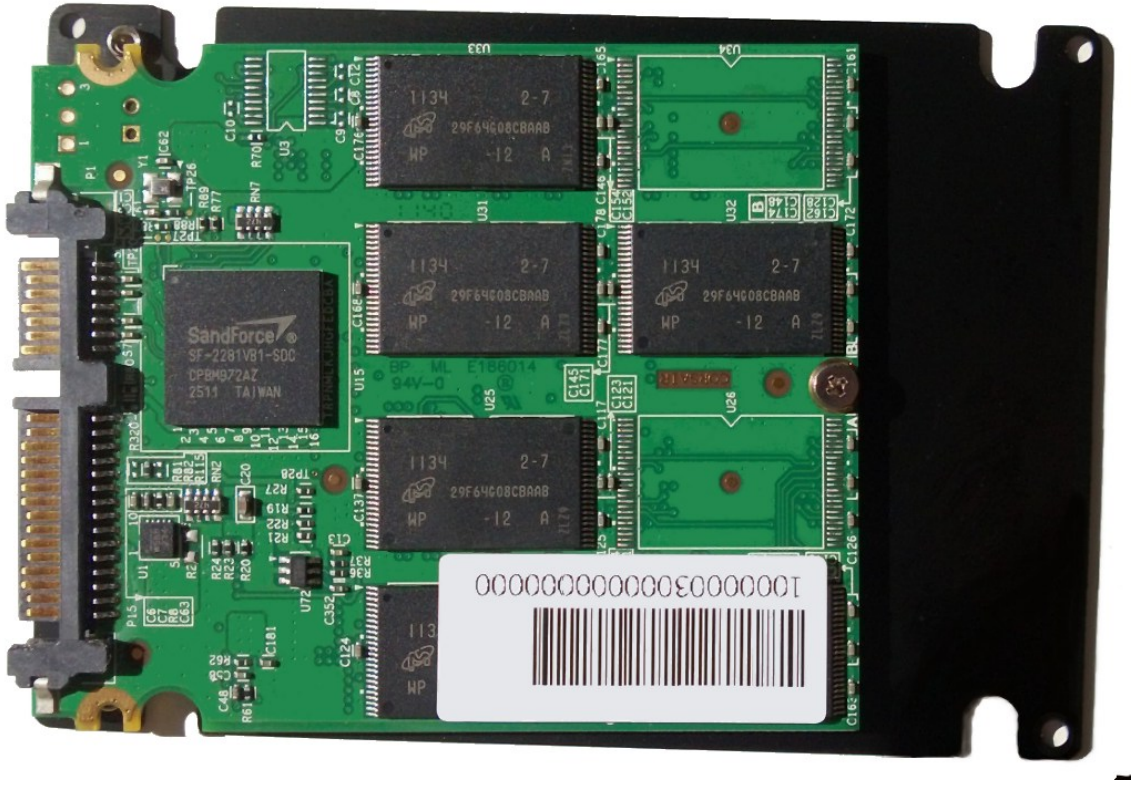
nichtflüchtiger Speicher, billiger, langsamer.

- Klassische Festplatte:
  - magnetisierbare, rotierende Platten

Von Eric Gaba, Wikimedia Commons user Sting, CC BY-SA 3.0, Link

FESTPLATTEN

Solid State Drive (SSD), Halbleiterlaufwerk



Von Hans Haase - Eigenes Werk, CC BY-SA 3.0, Link

ARBEITSWEISE

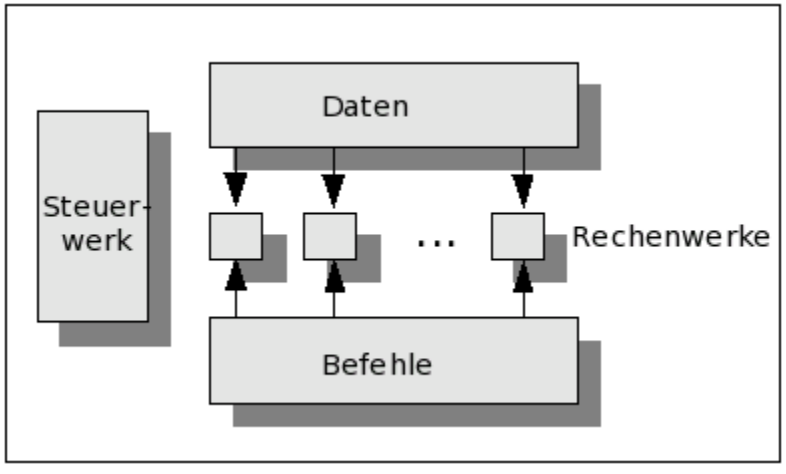
EVA-Prinzip

- **E**: Eingabe
  - » über Maus, Tastatur, Speichermedien gelangen Daten in den Computer
- **V**: Verarbeitung
  - » Der Prozessor (CPU) verarbeitet diese Daten
- **A**: Ausgabe
  - » Verarbeitete Daten werden über Ausgabegerät ausgegeben (Bildschirm, Drucker, Festplatte)

HARVARD ARCHITEKTUR

Strikte Trennung von Daten und Befehlen

› Zugriff erfolgt über je einen eigenen Bus. Entwickelt 1944 (Mark I) von IBM und der Harvard-University

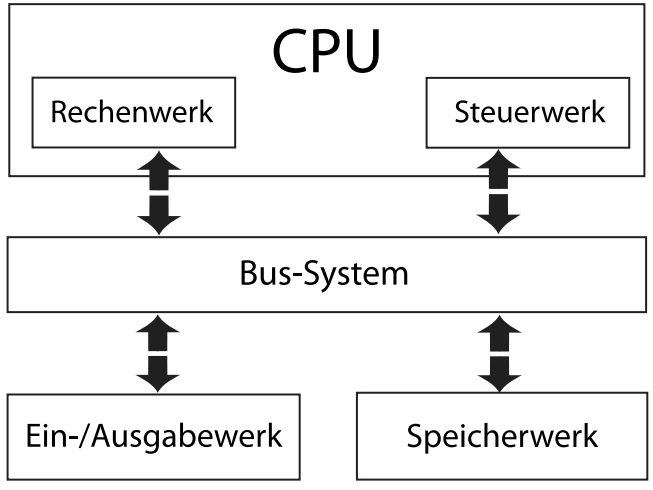


Von Matthias Kleine (April 2005) - Matthias Kleine, CC BY-SA 3.0, Link

HARVARD ARCHITEKTUR

- **Steuerwerk**: ist für das Einlesen der Befehle zuständig
- **Rechenwerk(e)**: führt entsprechende arithmetische und/oder logische Befehle aus
- **Daten**: enthält gespeicherte oder zu verarbeitende Daten
- **Befehle**: enthalten die einzelenen Befehle eines Programms
- **Bussystem** (Pfeile): transportiert Daten zwischen Einheiten

VON NEUMANN ARCHITEKTUR



VON NEUMANN ARCHITEKTUR

- **CPU**: Besteht aus Rechen- und Steuerwerk
  - **Steuerwerk**: ist für das Einlesen der Befehle zuständig
  - **Rechenwerk**: führt entsprechende arithmetische und/oder logische Befehle aus
- **Arbeitsspeicher**: enthält das Programm sowie alle dafür notwendigen Daten
- **Bussystem**: transportiert Daten zwischen Einheiten
- **Ein-/Ausgabe**: kommuniziert mit der Umwelt

HARVARD VS. VON NEUMANN ARCHITEKTUR

VON NEUMANN ARCHITEKTUR

- ✚ Einfacher da Programm und Daten im Speicher liegen, erlaubt einheitliche Routinen des Betriebssystems
- ✚ Programmcode kann sich selbst modifizieren; leichter zu 'debuggen'
- ━ Selbstmodifikation ist Risiko für Stabilität
- ━ Es gibt keinen Speicherschutz
- ━ Langsamer: eine Leitung für Befehle und Daten

HARVARD VS. VON NEUMANN ARCHITEKTUR

HARVARD ARCHITEKTUR

- ✚ Schnellerer Zugriff auf Daten und Programme, durch getrenntes Ansteuern
- ✚ Speicherschutz einfach umsetzbar
- ━ Parallele Zugriffe können zu **Race Conditions** führen
  - Ungewolltes Verhalten von Programmen

RACE CONDITIONS

› Bei **Race condition** hängt das Ergebnis einer Operation vom zeitlichen Verhalten der Einzeloperationen ab

Beispiel: 2 Systeme wollen Wert einer Zahl erhöhen

Schritt	System 1		System 2	
0	Lesen	0		
1			Lesen	0
2	Erhöhen	1		
3	Schreiben	1		
4			Erhöhen	1
5			Schreiben	1

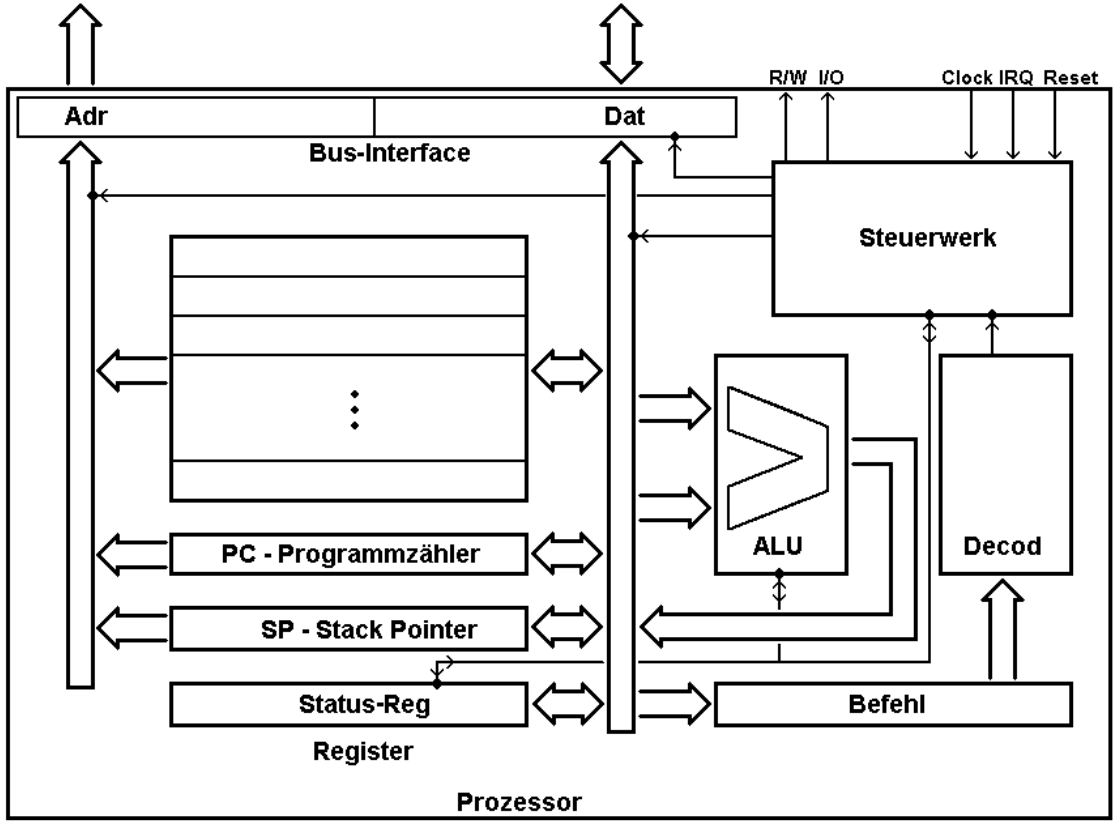
MODERNE COMPUTER

Basieren auf der Von Neumann Architektur

CPU, Arbeitsspeicher und Ein-/Ausgabe Hardware werden durch eine Hauptplatine (**Mainboard**) via **Bussystem** verbunden

Integrierte Hardware im Mainboard (Sound, Netzwerk, Grafik) zählt weiter als Peripherie.

FUNKTIONSWEISE CPU

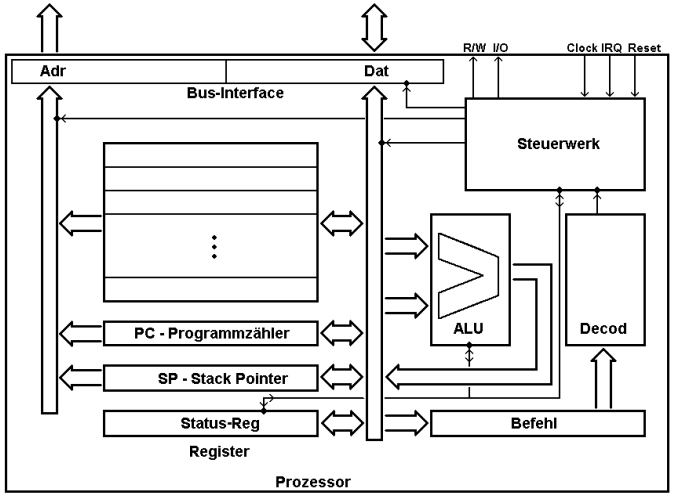


Von PeterFrankfurt - Eigenes Werk, CC BY-SA 3.0, Link

ABSTRAKTE FUNKTIONSWEISE CPU

1. **FETCH**
   - › Befehlsadresse lesen und aus Arbeitsspeicher in Register laden
2. **DECODE**
   - › Befehl in Register wird dekodiert und entsprechende Schritte für Verarbeitung werden vorbereitet
3. **EXECUTE**
   - › Der Befehl wird ausgeführt und das Ergebnis in den Arbeitsspeicher zurück geschrieben
4. **UPDATE** Instruction Pointer
   - › Die nächste Befehlsadresse wird eingestellt

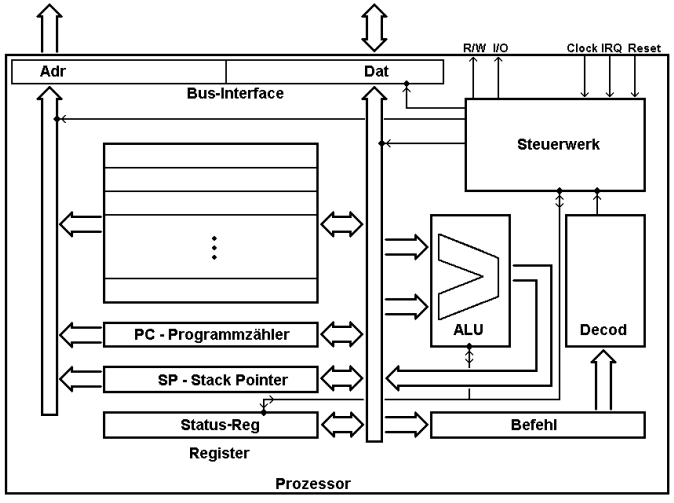
FUNKTIONSWEISE CPU



1. Befehlszähler (**PC**) zeigt auf Adresse im Speicher
2. Steuerwerk legt Adresse auf Bus und startet Lesebefehl
3. wenn RAM bereit, legt Inhalt an Datenleitung an
4. Steuerwerk kopiert den Inhalt des Befelsregisters
5. Befehl wird dekodiert und geprüft

   ...

FUNKTIONSWEISE CPU



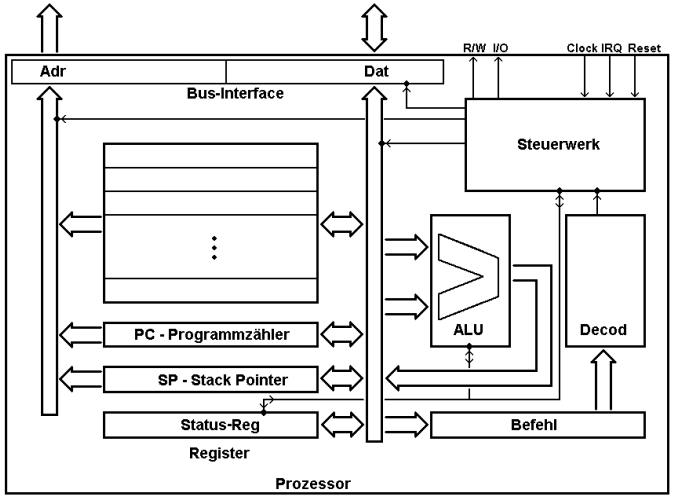
Von PeterFrankfurt - Eigenes Werk, CC BY-SA 3.0, Link

Nach Dekodieren:

- Wenn Befehl grösser oder benötigt Befehl weitere Daten aus Speicher: Schritte 1-3 erneut, und ins entsprechende Prozessorregister geladen.

6. Steuerwerk involviert benötigte Ressourcen (z.B. ALU)
7. ALU führt Befehl aus (z.B: Addition 2er Registerinhalte)

FUNKTIONSWEISE CPU



Von PeterFrankfurt - Eigenes Werk, CC BY-SA 3.0, Link

8. Ergebnis wird in ein Register geschrieben
9. Falls nötig wird das Ergebnis vom Register in RAM gespeichert
10. Programmzähler wird erhöht
11. Befehl ist abgearbeitet (start bei 1)

CPU BEFEHLSSÄTZE

2 generelle Gruppen:

- **RISC**
  - Reduced Instruction Set Computing
  - Sehr einfache Befehle (z.B: "ADD")
  - Kleine Anzahl an Befehlen
- **CISC**
  - Complex Instruction Set Computing
  - Komplexe Befehle direkt durchführbar
    - Bsp: Gleitkommazahl-Operationen
  - Große Anzahl an Befehlen

ARBEITSSPEICHER - CPU REGISTER

Register in Prozessoren sind sehr klein (aber schnell). Programme benötigen viel mehr Speicherplatz als im Register vorhanden ist.

- Intelligentes Zwischenspeichern von oft gebrauchten Daten im Arbeitsspeicher (RAM)
- Schneller als Laden von Festplatte, aber langsamer als direkt im Prozessor
- Register im Arbeitsspeicher sind nicht persistent
  - Ohne Strom ist Inhalt verloren
- Heute Arbeitsspeicher in "GByte"

FRAGEN?

NÄCHSTES MAL

2016-11-30 16:00

SOFTWAREENTWICKLUNGSPROZESS

mit Johanna Pirker