Assembler

Andreas Schärer

Kanti Romanshorn

Ziele Unterrichtseinheit

- **Hauptziel: Verstehen, was beim Programmieren im Hintergrund passiert.**
- Zusammenspiel von Software und Hardware verstehen.
- Wissen, welche Sprache die CPU eigentlich versteht.
- Wenig Assembler programmieren können.

Lektion heute

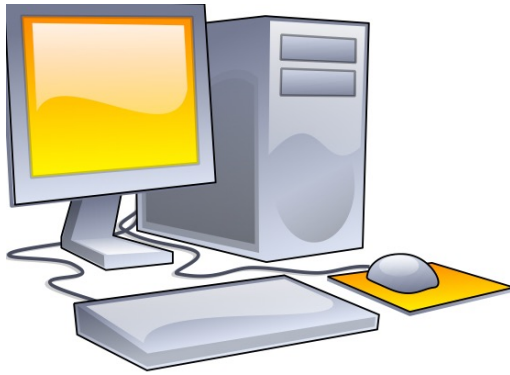
- **Ziele:**
  - Verstehen, wie Computer 30 + 12 berechnet
  - Beim **Kahoot** gewinnen!
- **Inhalt:**
  1. Aufbau von Computer.
  2. Sprache von Computer.
  3. Rechnung 30+12
  4. Kahoot

Teil 1

Aufbau eines Computers

Hardware von Computer





- Schauen uns drei wichtige **Bauteile** an …
- … die in *jedem* Computer vorkommen

1

Permanentspeicher
(Festplatte, SSD)

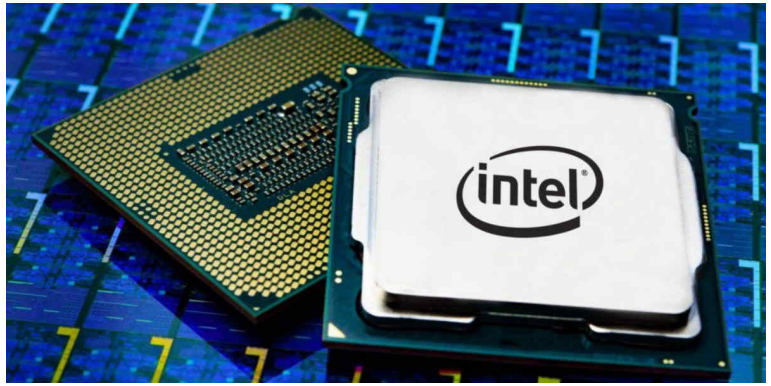
- Speichern von Filmen, Doks, …
- Installierte Programme
- Nach Herunterfahren noch vorhanden

2

Arbeitsspeicher
(RAM)

- Temporäre Speicherung
- Daten, die Programme im Moment benötigen
- Gelöscht nach Herunterfahren

3



Prozessor
(CPU)

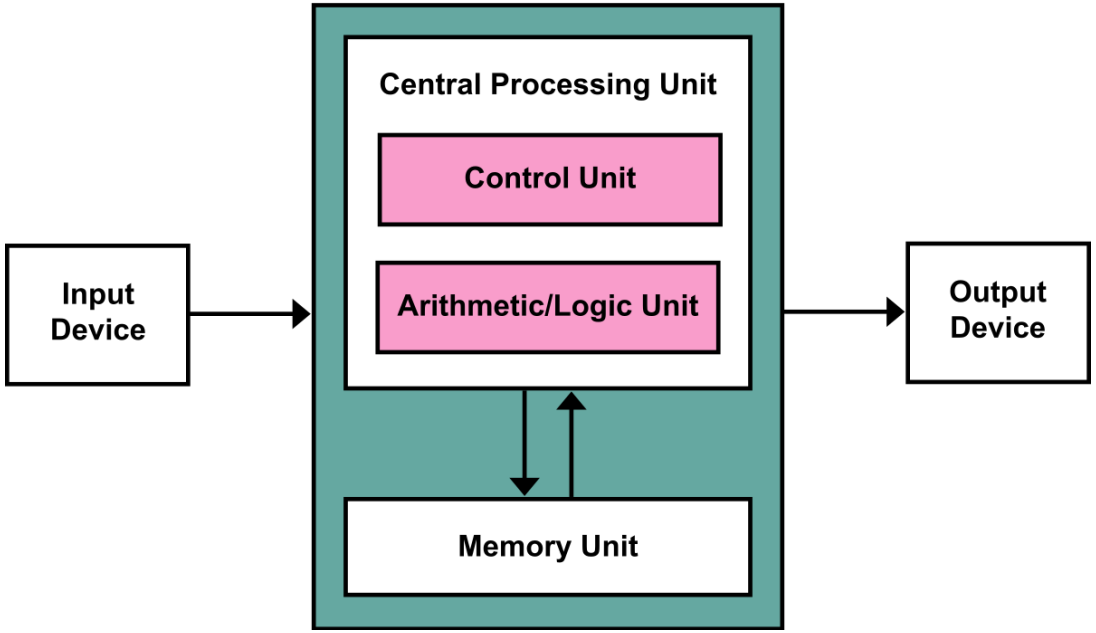
- «Gehirn» des Computers
- Rechnet, führt Befehle aus

Analogie Speicher

- Analogie: **Postfach**
- Jedes Postfach hat:
  - **Adresse/Nummer** (eindeutige Identifizierung)
  - **Inhalt** (leer, Brief, Paket)
- Vorgehen beim Leeren von Postfach:
  - Finde richtiges Postfach mithilfe von Adresse …
  - … und nimm dort Inhalt heraus
- Speicher: Aufgeteilt in Bereiche:
  - jeder hat eindeutige Adresse
  - Kann Werte Speichern
- Mithilfe von Adresse kann man dann auf Werte zugreifen

Von-Neumann-Architektur

- Grundidee zum Aufbau von Computer stammt von **John von Neumann** aus Jahr **1945**
- Hat **Von-Neumann-Architektur** von präsentiert
- Ein einfaches **Modell**, wie Computer *aufgebaut* ist und *funktioniert*:

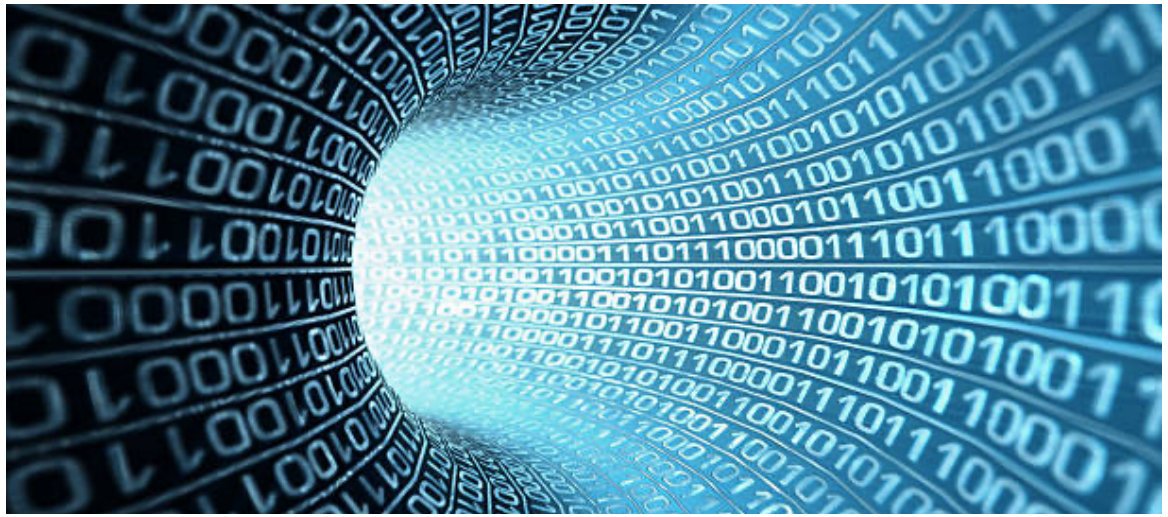


- Moderne Computer basieren (mehrheitlich) auf Von-Neumann-Architektur
- Später im Detail

Teil 2

Sprache des Computers (genauer: der CPU)

Sprache der CPU



- Was CPU macht: **Befehle** ausführen
- CPU **kennt nur Binärzahlen** (0 & 1)
- Befehle, die CPU ausführen soll, müssen also in Sprache sein, die CPU versteht
- Beispiel: 10010100111011010010100101001101010010110101001…
- Diese Sprache heisst **Maschinensprache**
- **Verschiedene CPUs** haben verschiedene Maschinensprachen (aber alle rein binär)

Sprache der CPU

- *Wie* können wir nun *Programmieren*?
- **Variante 1:** Code *direkt in Maschinensprache* schreiben!

- **Variante 2:** *Programmiersprache* wie Python wählen

- **Variante 3:** Programmieren in Assembler (Zwischending, später mehr)

Programmiersprachen

- Anstelle Befehle (Code) direkt in Maschinensprache zu schreiben, können eine **Programmiersprache** wählen …
- … wie Python, C#, Java, …
- Achtung: Diese versteht Prozessor *nicht direkt!*
- Muss diese zuerst deinem Computer *‘beibringen’*
- Mit Installation von TigerJython hast du deinem Computer Python ‘beigebracht’!
- Was passiert, wenn man **Python-Code ausführt**?
  1. Python-Code wird **in Maschinensprache übersetzt**:
     ’x = 10’ -> «10110010100101011001010011010100010110100 1»
  2. Code in **Maschinensprache** wird von CPU **ausgeführt**
- Achtung: Dies ist massiv vereinfacht dargestellt!

Ausführen von Code (F5)

Python-Code

```
print(30+12)
```

Wird umgewandelt in

Code in Maschinensprache

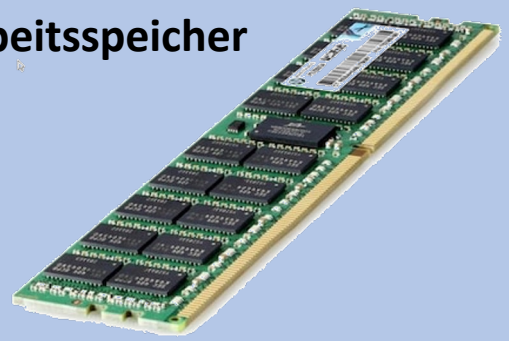
1001010011101101001010
0101001101010010110101
0010110101001101010100
1011010010101110101000
1010011100101001011001
0100

Wird ausgeführt von

Prozessor

Lesen und Schreiben von Werten

Arbeitsspeicher



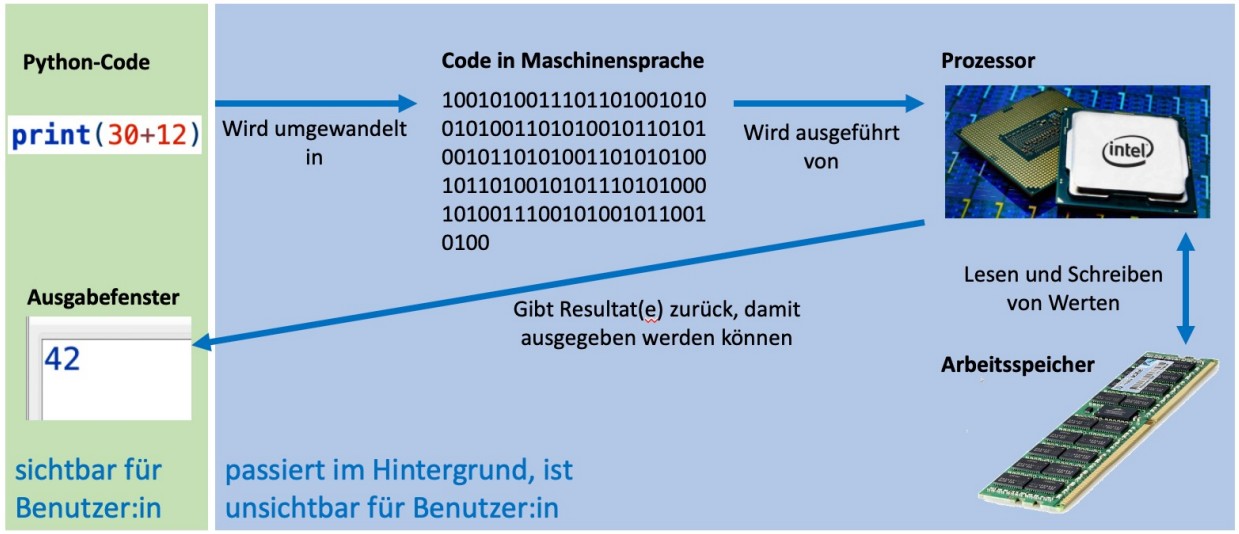
Gibt Resultat(e) zurück, damit ausgegeben werden können

Ausgabefenster

sichtbar für Benutzer:in

passiert im Hintergrund, ist unsichtbar für Benutzer:in

Assembler



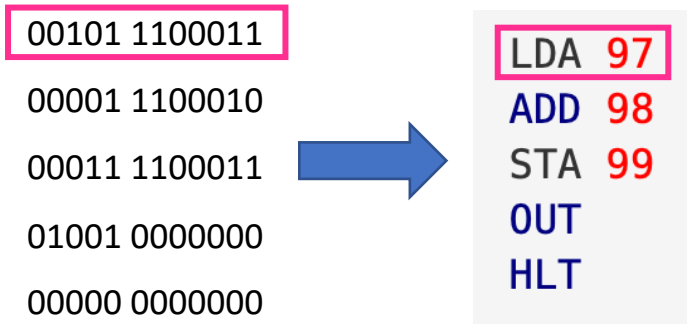
- Wollen jetzt blauen Teil, also **was im Hintergrund passiert**, genauer anschauen!
- Maschinensprache programmieren ist praktisch unmöglich für Menschen, deshalb …
- **Assembler**: *Übersetzung von Maschinensprache* (binär) in etwas, was für Menschen noch verständlich ist (Buchstaben, Zahlen).
- Genauer: **Assembly Language**
- Beispiel: Maschinensprache -> Assembly Language

```
00101 1100011
00001 1100010
00011 1100011
01001 0000000
00000 0000000
```

```
LDA 97
ADD 98
STA 99
OUT
HLT
```

Assembler

- Beispiel:



- Aus «Little Man Computer»: einfache Computer-Simulation
- Wie kommt man von «00101 1100011» auf «LDA 97»?
- Braucht ’Entzifferungsschlüssel’, genannt **Instruction Set**:
  - Erste 5 Bits: $00101_2 = 5_{10}$, Code von Instruction Set
  - Restliche 7 Bits: Binärzahl in Dezimalzahl umwandeln, gibt Speicherort an

Code	Name	Description
0	HLT	Stop (Little Man has a rest).
1	ADD	Add the contents of the memory address to the Accumulator
2	SUB	Subtract the contents of the memory address from the Accumulator
3	STA or STO	Store the value in the Accumulator in the memory address given.
4		This code is unused and gives an error.
5	LDA	Load the Accumulator with the contents of the memory address given
6	BRA	Branch - use the address given as the address of the next instruction
7	BRZ	Branch to the address given if the Accumulator is zero
8	BRP	Branch to the address given if the Accumulator is zero or positive
9	INP or OUT	Input or Output. Take from Input if address is 1, copy to Output if address is 2.
9	OTC	Output accumulator as a character if address is 22. (Non-standard instruction)
	DAT	Used to indicate a location that contains data.

- Kann mit Assembler **CPU direkt** programmieren -> maximal effizient
- Warum will man das?
  - Früher: einzige Möglichkeit
  - Heute: Code in Programmiersprache schreiben, dann in Assembler **optimieren**

Zusammenfassung

Besprecht in 2-3er Gruppen und macht Notizen:

1. Erkläre, was **Maschinensprache** ist.
2. Erkläre was **im Hintergrund passiert**, wenn man ein **Python**-Programm laufen lässt.
3. **Python vs. Assembler vs. Maschinensprache.** Fülle Tabelle aus:

	Python	Assembler	Maschinensprache
Vorteile			
Nachteile			
Beispiel			

Teil 3

Rechnung

Anspruchsvolle Rechnung

- Wollen mit Computer berechnen: 30 + 12 = ?
- Lösung:

```
print(30+12)
```

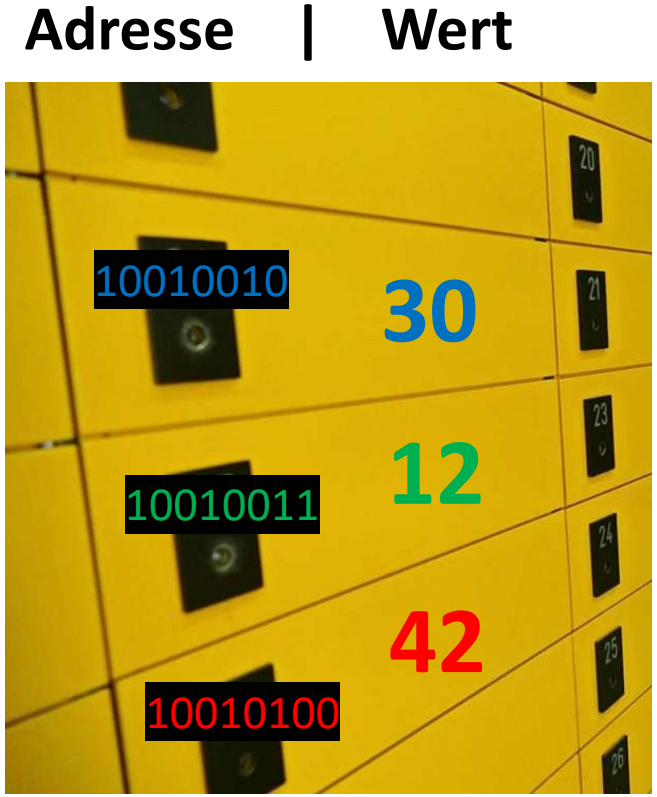
- Doch was passiert *im Hintergrund*?

Addition in Assembler/Maschinensprache

Müssen CPU dazu folgende Befehle geben:

1. Speichere Wert **30** in Arbeitsspeicher an Adresse 10010010
2. Speichere Wert **12** in Arbeitsspeicher an Adresse 10010011
3. Hohle Wert aus Arbeitsspeicher an Adresse 10010010 und merke dir diesen
4. Hole Wert an Adresse 10010011 …
5. … und addiere zum bisherigen Wert dazu
6. Speichere Wert (also **42**) an Adresse 10010100
7. Zeige Wert an Adresse 10010100 auf Bildschirm an

- Achtung: In Realität bestehen ganze Befehle nur aus Nullen und Einsen! Und Adressen sind viel länger.



Addition: Vergleich

In Python

```
print(30+12)
```

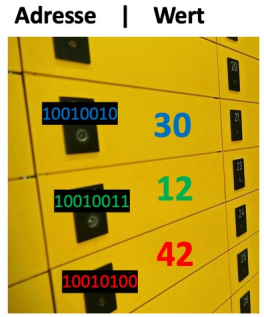
(was wir Coden)

In Assembler / Maschinensprache

Müssen CPU dazu folgende Befehle geben:

1. Speichere Wert **30** in Arbeitsspeicher an Adresse 10010010
2. Speichere Wert **12** in Arbeitsspeicher an Adresse 10010011
3. Hohle Wert aus Arbeitsspeicher an Adresse 10010010 und merke dir diesen
4. Hole Wert an Adresse 10010011 …
5. … und addiere zum bisherigen Wert dazu
6. Speichere Wert (also **42**) an Adresse 10010100
7. Zeige Wert an Adresse 10010100 auf Bildschirm an

- Achtung: In Realität bestehen ganze Befehle nur aus Nullen und Einsen! Und Adressen sind viel länger.



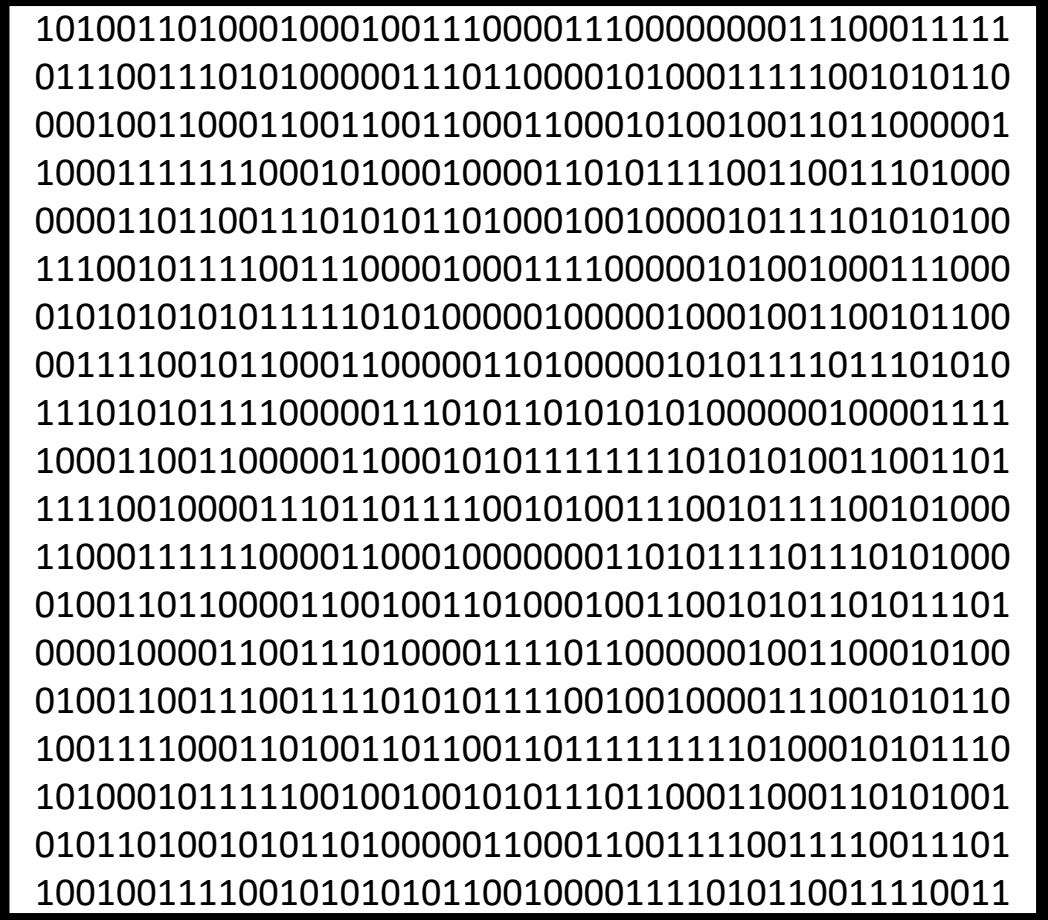
(was der Computer macht)

CPU

Was Leute denken, was ich tue.

```
print(30+12)
```

Was ich tue.



Teil 4

Kahoot