

---

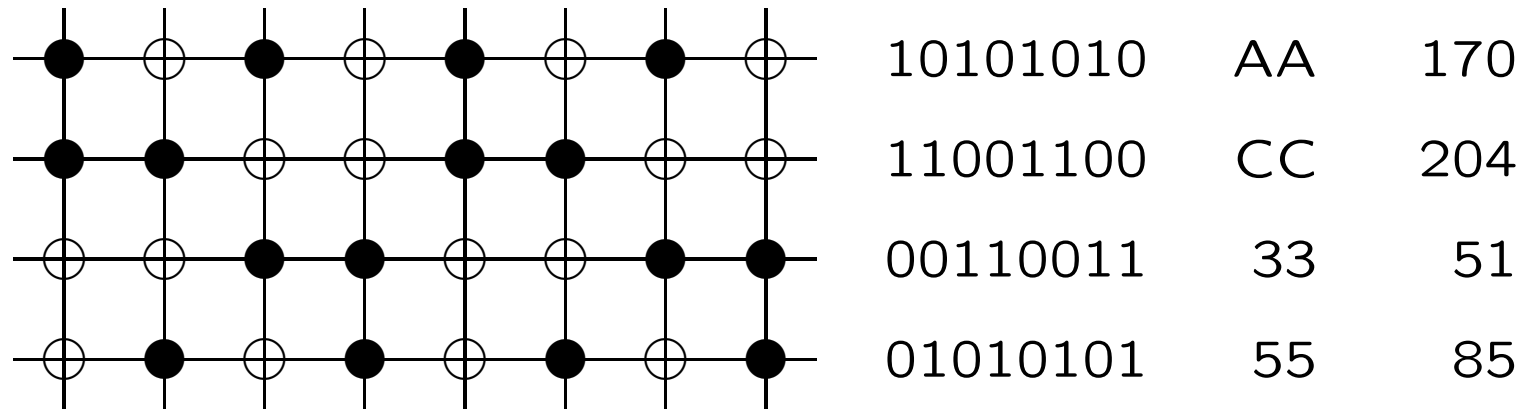
# Herzlich Willkommen zur Programmkonstruktion

---

# Digital & Binär

---

# Vom Webstuhl ins digitale Zeitalter



- analog = kontinuierlicher Wert = physikalische Größe, aber Fehler bei Verarbeitung und Speicherung unvermeidlich
- digital = diskreter Wert = Digitalisierungsfehler bei Messung, aber fehlerfrei verarbeit- und speicherbar

---

# Binäre Worte – Darstellungsformen

binär / oktäl / hexadezimal / dezimal / Zweierkomplement

0000 = 00 = 0 = 00 = 0	1000 = 10 = 8 = 08 = -8
0001 = 01 = 1 = 01 = 1	1001 = 11 = 9 = 09 = -7
0010 = 02 = 2 = 02 = 2	1010 = 12 = A = 10 = -6
0011 = 03 = 3 = 03 = 3	1011 = 13 = B = 11 = -5
0100 = 04 = 4 = 04 = 4	1100 = 14 = C = 12 = -4
0101 = 05 = 5 = 05 = 5	1101 = 15 = D = 13 = -3
0110 = 06 = 6 = 06 = 6	1110 = 16 = E = 14 = -2
0111 = 07 = 7 = 07 = 7	1111 = 17 = F = 15 = -1

- ein  $n$ -Bit-Wort kann  $2^n$  Werte darstellen
- Dezimalzahlen:  $0 \dots 2^n - 1$  oder  $-2^{n-1} \dots 2^{n-1} - 1$

---

# Bytes

- früher: 1 Byte = Wortlänge für Darstellung von Zeichen
- heute: 1 Byte = 8 Bit (auch als Teil eines Wortes)
- noch immer: Größe in Byte  $\approx$  Textlänge in Zeichen

1 kB	=	$2^{10}$ Byte =	1.024 Byte $\approx 10^3$ Byte
1 MB	=	$2^{20}$ Byte =	1.048.576 Byte $\approx 10^6$ Byte
1 GB	=	$2^{30}$ Byte =	1.073.741.824 Byte $\approx 10^9$ Byte
1 TB	=	$2^{40}$ Byte =	1.099.511.627.776 Byte $\approx 10^{12}$ Byte

---

# Bit-Operationen

- Logische Bit-Operationen:

AND:	$00 \rightarrow 0$	$01 \rightarrow 0$	$10 \rightarrow 0$	$11 \rightarrow 1$
OR:	$00 \rightarrow 0$	$01 \rightarrow 1$	$10 \rightarrow 1$	$11 \rightarrow 1$
XOR:	$00 \rightarrow 0$	$01 \rightarrow 1$	$10 \rightarrow 1$	$11 \rightarrow 0$
NOT:	$0 \rightarrow 1$	$1 \rightarrow 0$		

- Bit-Multiplikation entspricht UND:

$$0 \cdot 0 = 0 \quad 0 \cdot 1 = 0 \quad 1 \cdot 0 = 0 \quad 1 \cdot 1 = 1$$

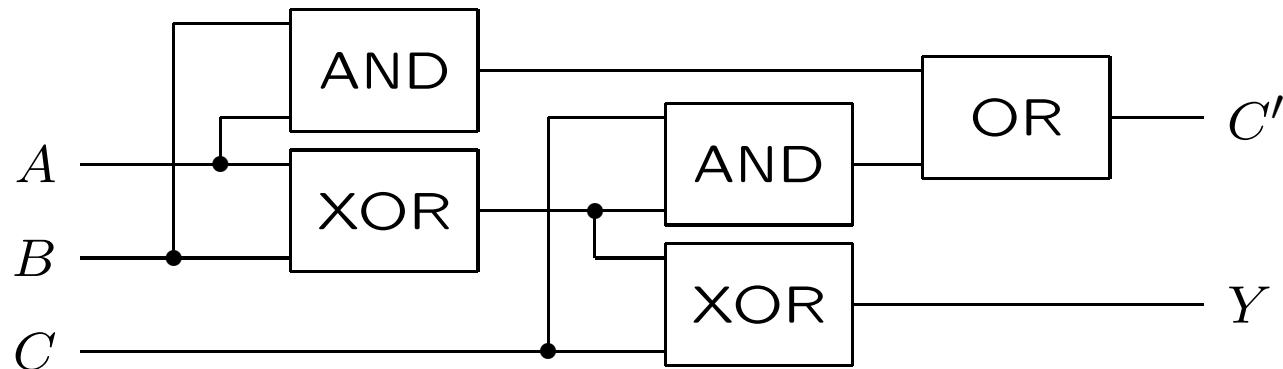
- Bit-Addition durch XOR, Übertrag durch AND:

$$0 + 0 = 00 \quad 0 + 1 = 01 \quad 1 + 0 = 01 \quad 1 + 1 = 10$$

---

# Logische Schaltung

Beispiel: Volladdierer



- Auch komplexe Schaltungen über einfache Gatter realisiert
- Gatterlaufzeiten ca. 0,1 ns bis über 100 ns
- Quelle und Ziel ist Speicher, Berechnung im fixen Takt

---

# VHDL-Programm für Hardware

```
library IEEE;                -- aus der Bibliothek IEEE:
use IEEE.std_logic_1164.all; -- Form der Bit-Darstellung

entity ANDGATE is            -- ein ANDGATE hat
  port (IN1 : in std_logic;  -- zwei Eingaenge (IN1, IN2)
        IN2 : in std_logic;  -- und einen Ausgang (OUT1)
        OUT1: out std_logic); -- als einfache Bits
end ANDGATE;

architecture RTL of ANDGATE is
begin
  OUT1 <= IN1 and IN2;      -- Berechnung des Ergebnisses
end RTL;
```

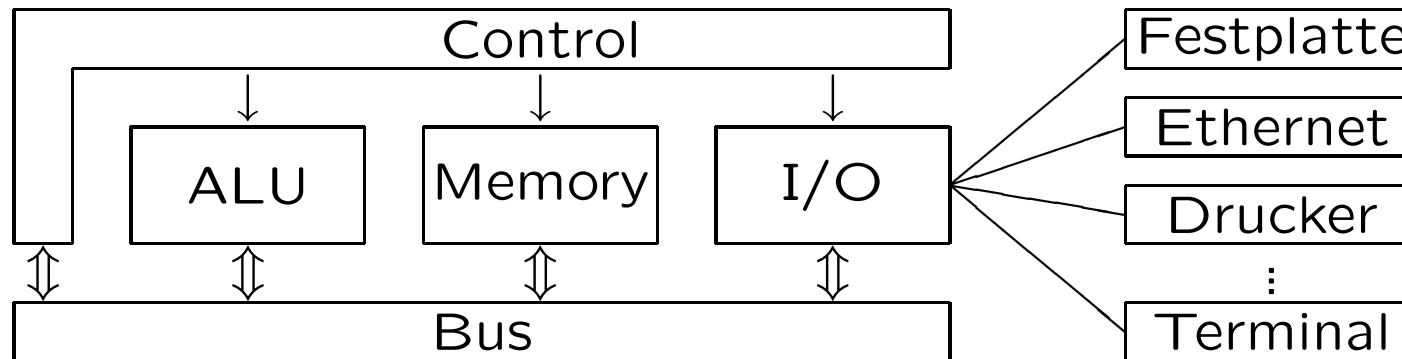


---

# Maschinen und Architekturen

---

# Von Neumann-Architektur



Zur **Ausführung** ständig wiederholt:

1. hole nächsten Befehl aus Speicher (durch PC adressiert)
2. erhöhe PC um 1
3. interpretiere Befehl und führe ihn aus

---

# Aktuelle Computer-Architekturen

- *Harward-Architektur* trennt Befehls- von Datenspeicher
- Hierarchie von Speicherebenen: Register und Caches
- virtuelle Adressierung
- Spezialregister und Spezialbefehle für Zugriffe darauf
- Prozessor fasst zentrale Teile auf einem Chip zusammen
- mehrere Prozessorkerne

---

# Assembler-Programm (i386, Linux)

```
section .text
global _start
_start:
    mov ecx, hello
    mov edx, length
    mov ebx, 1
    mov eax, 4
    int 80h
    mov ebx, 0
    mov eax, 1
    int 80h
section .data
    hello db 'Hello World!'
    length equ $ - hello;
```

---

# Architektur versus Implementierung

- *Architektur* beschreibt Maschine in dem Detailliertheitsgrad, den Programmierer bzw. Benutzer kennen muss
- *Implementierung* der Architektur = tatsächliche Maschine
- Alle Implementierungen derselben Architektur verstehen dieselben Programme (Portierbarkeit)

---

# Abstrakte Maschine

- abstrakte Beschreibung aller Implementierungen
- niedriger bis hoher Abstraktionsgrad
- häufig in Software implementiert (keine Hardware)
- Beispiel: *Java Virtual Machine (JVM)* – Bytecode
- höherer Abstraktionsgrad erhöht Portabilität
- niedriger Abstraktionsgrad erhöht Effizienz

---

# JVM-Code für Hello

```
public class Hello extends java.lang.Object{
public Hello();           //Konstruktor (nicht verwendet)
    Code:
        0: aload_0
        1: invokespecial #1; //Method java/lang/Object."<init>"
        4: return

public static void main(java.lang.String[]);
    Code:
        0: getstatic #2;       //Field java/lang/System.out
        3: ldc #3;            //String Hello World!
        5: invokevirtual #4; //java/io/PrintStream.println
        8: return
}
```