



Technische Hochschule  
Ingolstadt

Fakultät Informatik

# *Einführung in die Informatik*

## *Grundlagen (2)*

Andreas Frey    WS 2021/22

Skript bereitgestellt von Prof. Volker Stiehl

## *Disclaimer*

**Diese Präsentation // der Inhalt dieses Dokuments ist rein zur  
persönlichen lernunterstützenden Verwendung hinsichtlich  
der entsprechenden Vorlesung an der TH Ingolstadt  
vorgesehen**

**Dieses Dokument NICHT verbreiten, NICHT „online“ stellen!**



- **Was haben wir schon gelernt?**
  - Darstellung von Zahlen und Zeichen
- **Was lernen wir heute?**
  - Verknüpfung binär codierter Information zu neuer Information
  - Rechnen mit Binärzahlen
  - Aufbau und Funktionsweise eines Computers
  - „Von-Neumann Rechnerarchitektur“
  - Maschinensprache und Assembler



# Grundlagen

## ■ Voraussetzungen

1. Es muss ein **vollständiges System** elementarer **binärer Operatoren** existieren
2. Zu jedem elementaren Operator muss es eine **entsprechende elektronische Schaltung** geben
3. Es müssen **systematische Verfahren** existieren, die die Analyse beliebiger Verknüpfungsschaltungen und ihre Synthese aus elementaren Operatoren erlauben

## ■ Schaltalgebra

- Ein Hilfsmittel zur Analyse und Synthese mit den elementaren Operatoren  
**UND ODER NICHT**  
ist die Schaltalgebra als spezielle **boolesche Algebra**

# Rechnen mit Binärzahlen

- Rechenregeln für Binärzahlen sind **analog** zu den Rechenregeln für Dezimalzahlen definiert
- Ausführung von Algorithmen mithilfe eines Computers: Unterteilung des Problems in Teilaufgaben, die unter Verwendung der **vier Grundrechenarten** und der **logischen Operationen** gelöst werden können
- → es genügt, sich auf die binäre **Addition**, **Subtraktion**, **Multiplikation**, **Division** und die **logischen Operationen** zu beschränken
- In Computersystemen werden logische Operationen grundsätzlich **bitweise** durchgeführt
- Wesentlich sind die beiden **zweistelligen Operationen** logisches **UND** (AND, Symbol:  $\wedge$ ), logisches **ODER** (OR, Symbol:  $\vee$ ) und die **einstellige Operation Inversion** oder **Negation** (NOT, Symbol:  $\neg$ )
- **Alle anderen logischen Operationen** können durch **Verknüpfung der Grundfunktionen** abgeleitet werden

## Rechnen mit Binärzahlen

- Logischen Grundfunktionen sind durch ihre **Wahrheitstafeln/-tabellen** definiert:

OR:	$1 \vee 1 = 1$	$0 \vee 1 = 1$	$1 \vee 0 = 1$	$0 \vee 0 = 0$
AND:	$1 \wedge 1 = 1$	$0 \wedge 1 = 0$	$1 \wedge 0 = 0$	$0 \wedge 0 = 0$
NOT:	$\neg 1 = 0$	$\neg 0 = 1$		

- Weitere **wichtige logische Funktion** ist das **exklusive Oder** (eXclusive OR, XOR), definiert durch  $a \text{ XOR } b = (a \wedge \neg b) \vee (\neg a \wedge b)$

Wahrheitstabelle:

$$1 \text{ XOR } 1 = 0, \quad 0 \text{ XOR } 1 = 1, \quad 1 \text{ XOR } 0 = 1, \quad 0 \text{ XOR } 0 = 0$$

- Binärzahlen mit mehr als einer Ziffer werden **stellenweise** durch logische Operationen verknüpft

$\begin{array}{r} 10011 \\ \vee 10101 \\ \hline = 10111 \end{array}$	$\begin{array}{r} 10011 \\ \wedge 10101 \\ \hline = 10001 \end{array}$	$\begin{array}{r} \neg 10101 \\ \hline = 01010 \end{array}$
--	--	---

# Rechnen mit Binärzahlen

- Rechenregeln für die **binäre Addition** zweier Binärziffern:

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1, \quad 1 + 1 = 0 \text{ Übertrag } 1$$

- Regeln für **binäre Addition** sind mit denen des **logischen XOR identisch**, es kommt lediglich der Übertrag hinzu!
- **Stellenweise Addition** (hier  $11 + 14$ ; funktioniert auch für Festkommazahlen):

1011	
+ 1110	
111	Übertrag
<hr/>	
= 11001	Ergebnis

- Rechenregeln für die **binäre Subtraktion** zweier Binärziffern

$$0 - 0 = 0, \quad 1 - 1 = 0, \quad 1 - 0 = 1, \quad 0 - 1 = 1 \text{ Übertrag } -1$$

- **Stellenweise Subtraktion** (hier  $13 - 11$ ):

1101	
- 1011	
-1	Übertrag
<hr/>	
= 0010	Ergebnis



## Rechnen mit Binärzahlen

- Für die praktische Ausführung mit Computern gibt es jedoch eine **geeignere Methode zur Subtraktion**, die sich **leichter als Hardware** realisieren lässt: die **Zweierkomplement-Methode**
- **Zahlen** in Computern werden als **Bitmuster** dargestellt
- **Wichtig:** auch das **Vorzeichen** (VZ) einer Zahl **durch ein Bit codieren!**
- Für VZ Bit mit dem **höchsten Stellenwert verwenden** (Most Significant Bit, **MSB**)
- **Beachte:** feste **Stellenzahl n** (i. d. R. 8 Bit oder ein Vielfaches davon) muss vorausgesetzt werden und **Zahlenbereich** umfasst das MSB **nicht**.
- **Möglichkeit 1** zur Darstellung einer negativen Zahl: die Binärdarstellung der positiven Zahlen nehmen und **nur das Vorzeichenbit ändern** (0 = positiv; 1 = negativ)
- Für n=8 ergeben sich für die Zahlen +5 und -5 die folgenden Binärdarstellungen:  
$$(5)_{10} = (0000\ 0101)_2 \text{ und } (-5)_{10} = (1000\ 0101)_2$$
- Was ergibt aber dann  $5 + (-5)$  in Binärdarstellung??? → **Darstellung unbrauchbar!**



## Rechnen mit Binärzahlen

- **Möglichkeit 2** über **bitweises Invertieren aller Stellen (Stellenkomplement oder Einerkomplement)** → ließe sich auch maschinell schnell über Inverter realisieren
- → **positive Zahlen direkt codieren** und für **negative Zahlen die Stellen invertieren** → Vorzeichenänderung hat nur Inversion zur Folge (umfasst automatisch auch VZ-Bit)
- Für  $n=8$  ergeben sich für die Zahlen +5 und -5 die folgenden Binärdarstellungen:

$$(5)_{10} = (00000101)_2 \text{ und } (-5)_{10} = (11111010)_2$$

- **MSB aller negativen Zahlen** hat den Wert 1
- Was ergibt diesmal  $5 + (-5)$  in Binärdarstellung? → positive und negative 0 ☹
- → **Möglichkeit 3: Zweierkomplement**

**Zweierkomplement einer binären Zahl erhält man durch Bildung des Stellenkomplements/Einerkomplements und Addieren von 1 zum Ergebnis**

- Für  $n=8$  ergeben sich für die Zahlen +5 und -5 die folgenden Binärdarstellungen:

$$(5)_{10} = (00000101)_2 \text{ und } (-5)_{10} = (11111011)_2$$

# Rechnen mit Binärzahlen

- → Der Rechner muss nicht subtrahieren können – Addition genügt!!!
- Subtrahieren ist Addieren des Zweierkomplements!
- Beispiele:

7 – 4

00000100    4

11111011    Stellenkomplement von 4

1    1 wird addiert

11111100    Zweierkomplement von 4 (entspricht –4)

00000111    7

---

10000011    Ergebnis der Addition: 9 Stellen → Überlauf streichen

00000011    Ergebnis ( $n = 8$ ):  $7 - 4 = 3$  (positiv, da MSB = 0)

12 – 17

00010001    17

11101110    Stellenkomplement von 17

1    1 wird addiert

11101111    Zweierkomplement von 17 (entspricht –17)

00001100    12

---

11111011    Ergebnis ( $n = 8$ ):  $12 - 17 = -5$  (negativ, da MSB = 1)

## Rechnen mit Binärzahlen

- Rechenregeln für die **binäre Multiplikation** entsprechen der logischen **UND-Verknüpfung** zweier Binärziffern
$$0 \cdot 0 = 0, 0 \cdot 1 = 0, 1 \cdot 0 = 0, 1 \cdot 1 = 1$$
- Multiplikation mehrstelliger Zahlen wird auf die **Multiplikation des Multiplikanden mit den einzelnen Stellen des Multiplikators** und **stellenrichtige Addition** der Zwischenergebnisse zurückgeführt
- Multiplikation wird durch fortgesetzte **Addition** ersetzt, da die Multiplikation mit den Grundziffern 0 und 1 keinen Aufwand erfordern:

Die Aufgabe  $10 \cdot 13 = 130$  ist in binärer Arithmetik zu lösen.

$$\begin{array}{r} 1010 \cdot 1101 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 1000010 \end{array} \quad \text{Ergebnis}$$

## Rechnen mit Binärzahlen

- **Binäre Division:** Ähnlich wie die Multiplikation lässt sich auch die binäre Division in Analogie zu dem im Zehnersystem gewohnten Verfahren durchführen:

$$\begin{array}{r} 10100 : 110 = 11,0101 \dots \\ -110 \\ \hline 1000 \\ -110 \\ \hline 1000 \\ -110 \\ \hline \dots \end{array}$$

- Tatsächlich führt man **Multiplikation** und **Division** in digitalen Rechenanlagen durch eine **Kombination** von **Verschieben** (Shift) und **Addieren** bzw. **Subtrahieren** aus.
- Wird eine Binärzahl mit einer **Zweierpotenz  $2^k$  multipliziert**, so entspricht dies – in Analogie zur Multiplikation mit einer Potenz von 10 im Zehnersystem – lediglich einer **Verschiebung dieser Zahl um  $k$  Stellen nach links**:  
$$13 \cdot 4 = 52 \rightarrow 1101 \cdot 100 = 110100 \rightarrow \text{Verschieben um 2 Stellen nach links}$$
- In analoger Weise ist die **Division durch Zweierpotenzen  $2^k$  einer Verschiebung nach rechts um  $k$  Stellen** äquivalent

# Rechnen mit Binärzahlen

## Boolesche Algebra als Grundlage der digitalen Elektronik

### ■ Definition (Boolesche Algebra):

- Die Menge  $B$  von Elementen, über der zwei zweistellige Operationen '+' und '•' erklärt sind, ist genau dann eine **Boolesche Algebra**, wenn für beliebige Elemente  $a, b, c \in B$  folgende Axiome (beweislos vorausgesetzt) gelten:

I	$a + b = b + a$	Die Operationen (+) und (•) sind kommutativ
II	$0 + a = a$ $1 \bullet a = a$	Für jede der Operationen (+) bzw. (•) existiert in $\mathcal{B}$ ein neutrales Element '0' bzw. '1'
III	$(a + b) \bullet c = (a \bullet c) + (b \bullet c)$ $(a \bullet b) + c = (a + c) \bullet (b + c)$	Jede der Operationen ist distributiv bezüglich der anderen
IV	$a + \bar{a} = 1$ $a \bullet \bar{a} = 0$	Zu jedem Element $a \in \mathcal{B}$ existiert ein inverses Element $\bar{a} \in \mathcal{B}$ .

### ■ Satz (Dualitätsprinzip):

- Zu jeder Aussage**, die sich aus diesen vier Axiomen ableiten lässt, **existiert eine duale Aussage**, die dadurch entsteht, dass man die Operationen '+' und '•' und gleichzeitig die Elemente '0' und '1' vertauscht.

### ■ Gesetzmäßigkeiten

- Von den Axiomen lassen sich viele weitere hilfreiche Gesetzmäßigkeiten ableiten.

## Rechnen mit Binärzahlen

Statt der Operatoren '+', '•' und '' werden meist '∨', '∧' sowie '¬' verwendet.

Seien  $a, b, c, \neg a, \neg b, \neg c \in B$ . Dann gelten folgende **Äquivalenzen**:

Idempotenz	$a \wedge a \equiv a$ $a \vee a \equiv a$
Kommutativität	$a \wedge b \equiv b \wedge a$ $a \vee b \equiv b \vee a$
Assoziativität	$(a \wedge b) \wedge c \equiv a \wedge (b \wedge c) \equiv a \wedge b \wedge c$ $(a \vee b) \vee c \equiv a \vee (b \vee c) \equiv a \vee b \vee c$
Absorption	$a \wedge (a \vee b) \equiv a$ $a \vee (a \wedge b) \equiv a$
Distributivität	$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$ $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$
Doppelnegation	$\neg \neg a \equiv a$
deMorgansche Regeln	$\neg (a \vee b) \equiv (\neg a) \wedge (\neg b)$ $\neg (a \wedge b) \equiv (\neg a) \vee (\neg b)$
Neutrales Element	$a \wedge 1 \equiv a$ $a \vee 0 \equiv a$

**Beweis:** Anwendung und Verknüpfung der Axiome (siehe nächste Folie).



# Rechnen mit Binärzahlen

## Beweis eines Satzes

**Axiom 5.1** (Kommutativgesetze).  $a \wedge b = b \wedge a$  und  $a \vee b = b \vee a$ .

**Axiom 5.2** (Distributivgesetze).  $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$  und  $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ .

**Axiom 5.3** (Existenz der neutralen Elemente).  $a \wedge 1 = a$  und  $a \vee 0 = a$ .

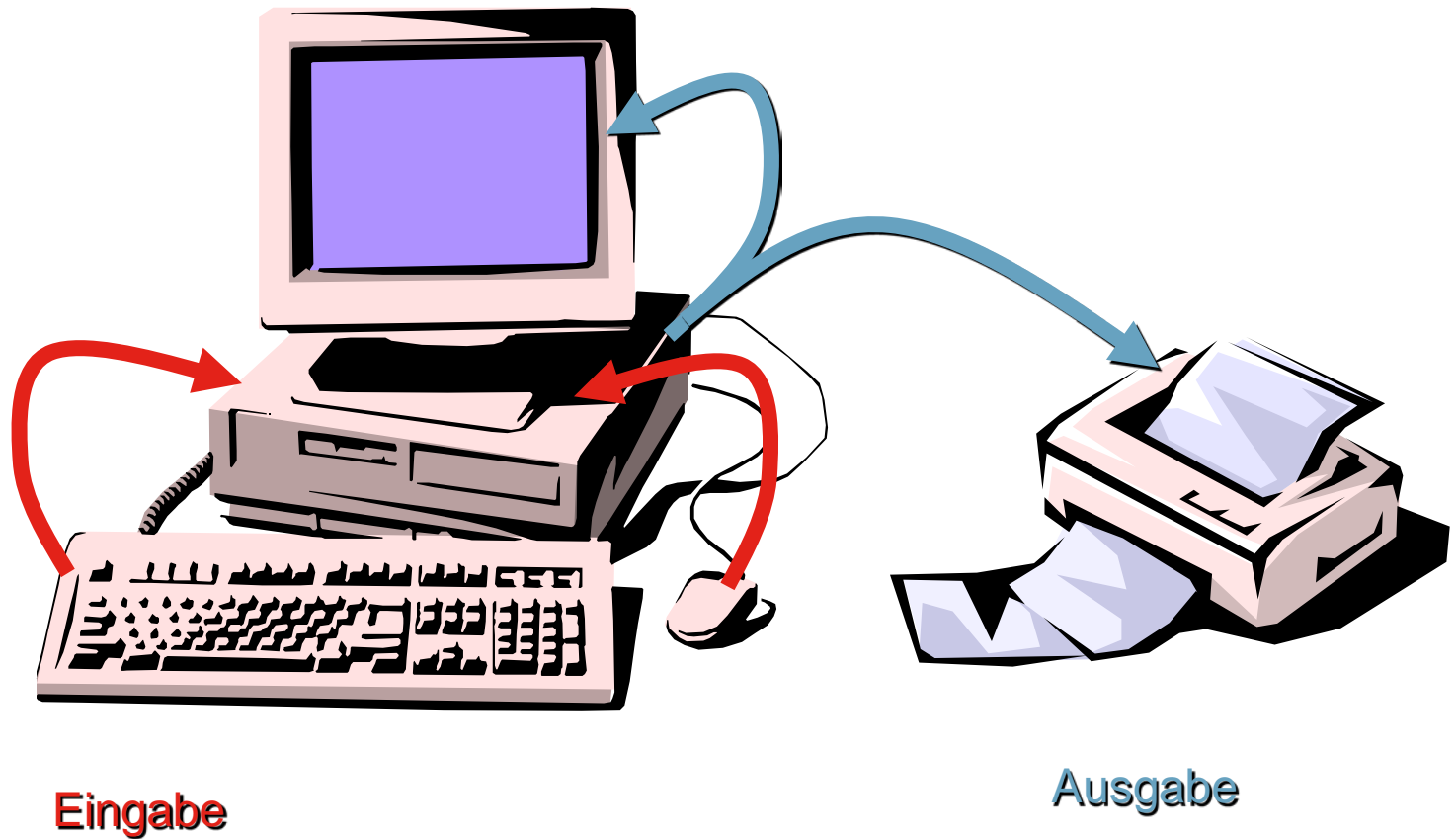
**Axiom 5.4** (Definition des komplementären (inversen) Elements).  $a \wedge \neg a = 0$  und  $a \vee \neg a = 1$ .

**Beispiel:** Es ist die Gültigkeit des Idempotenz-Gesetzes  $a \wedge a = a$  zu beweisen:

$$a \wedge a = (a \wedge a) \vee 0 = (a \wedge a) \vee (a \wedge \neg a) = a \wedge (a \vee \neg a) = a \wedge 1 = a.$$

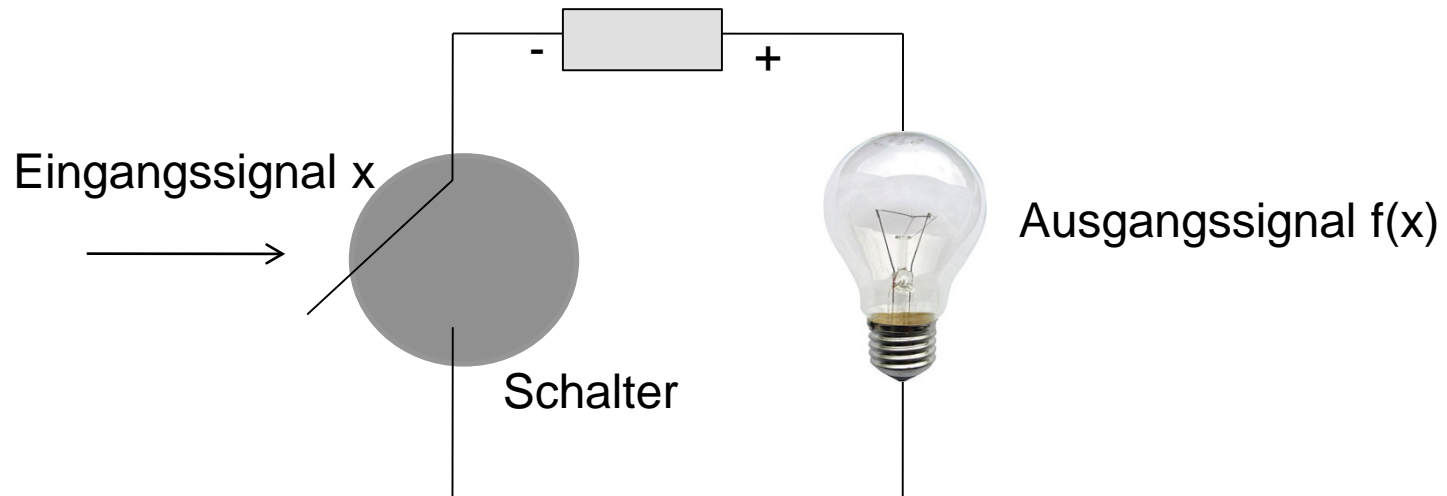
Die Axiome wurden in der Reihenfolge 5.3, 5.4, 5.2, 5.4, 5.3 angewendet.

- **Wie funktioniert ein Computer?**





## ■ Elementares Bauelement: Schalter



x	f(x)
1	1
0	0

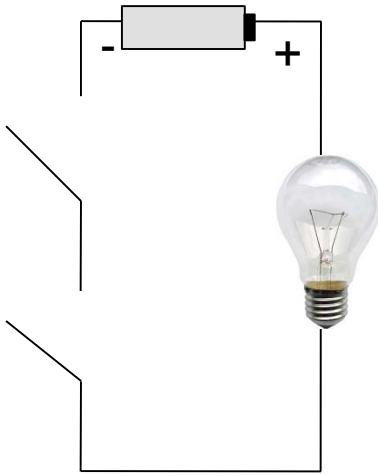
- **Umsetzung von AND und OR**



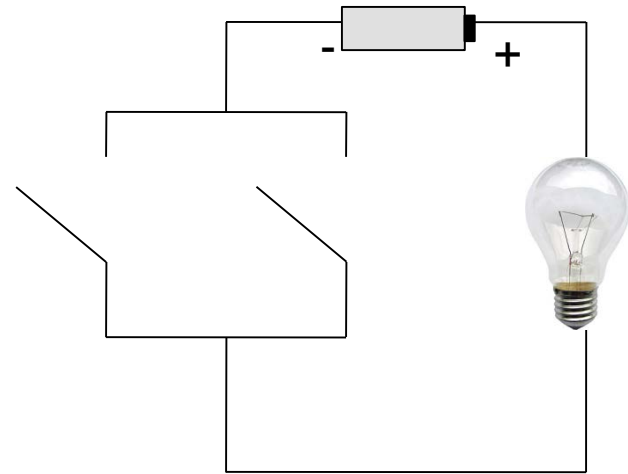
x	y	$f(x,y) = x \text{ AND } y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$f(x,y) = x \text{ OR } y$
0	0	0
0	1	1
1	0	1
1	1	1

## ■ Umsetzung von AND und OR



x	y	$f(x,y) = x \text{ AND } y$
0	0	0
0	1	0
1	0	0
1	1	1



x	y	$f(x,y) = x \text{ OR } y$
0	0	0
0	1	1
1	0	1
1	1	1

# Grundlagen



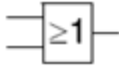

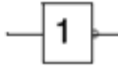
## ■ Definition (Schaltalgebra):

Die Schaltalgebra ist eine **spezielle boolesche Algebra** über  $B = \{0, 1\}$  mit den Operatoren ODER, UND, NEGATION

## ■ Warum ist die Schaltalgebra so wichtig?

- **Elektrische Schaltungen**, die nur zwei Zustände einnehmen können, lassen sich elegant **mit dieser Schaltalgebra beschreiben**.
- Bistabile Schalt- und Speicherelemente lassen sich **technisch leicht realisieren**.

## ■ Beschreibung mittels Schaltungssymbole

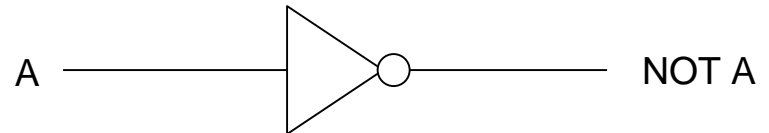
Operation	ODER	UND	NEGATION
Operationssymbole	$+, \vee$	$\bullet, \wedge$	$\neg, \neg$
Schaltungssymbole nach DIN			$\neg$ oder $\neg$
Schaltungssymbole nach IEEE/ANSI			

## ■ Grundbausteine

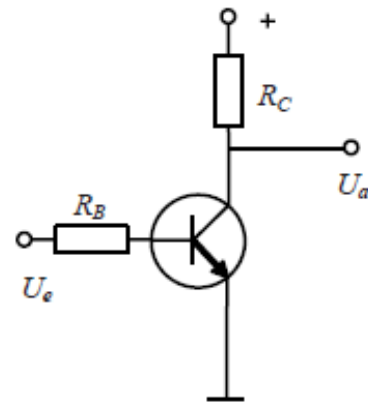
### ■ Logische **Gatter**

- Rechner kennt **ausschließlich** „Spannung (5V) oder Nicht-Spannung (0V)“
- Gatter ermöglichen die **logische Verknüpfung von Eingangssignalen** auf der Basis **Bool'scher Operatoren** zur Erzeugung entsprechender Ausgangssignale:
- NOT (*Inverter*)

Symbol

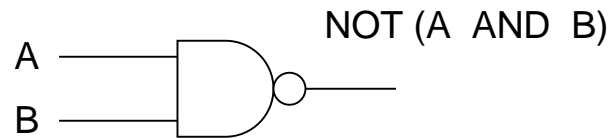


Technische Realisierung

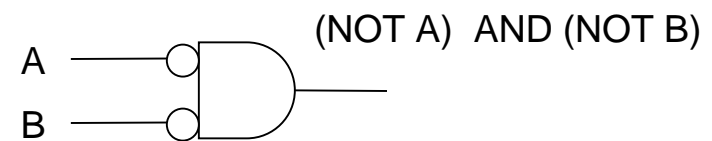


# Aufbau von Computersystemen

## ▪ NAND (NOT AND)

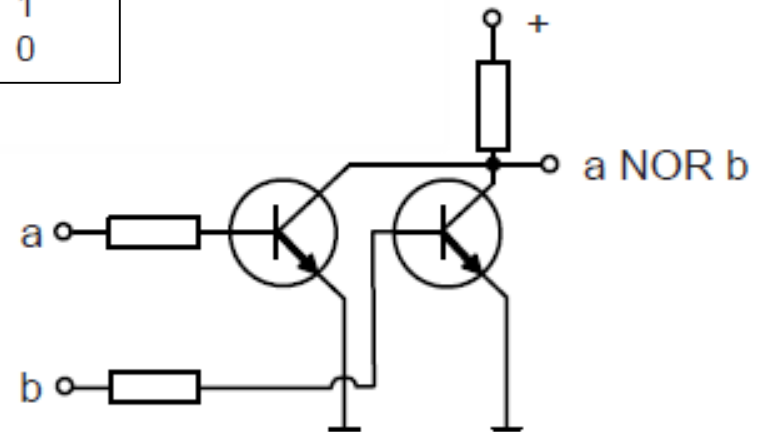
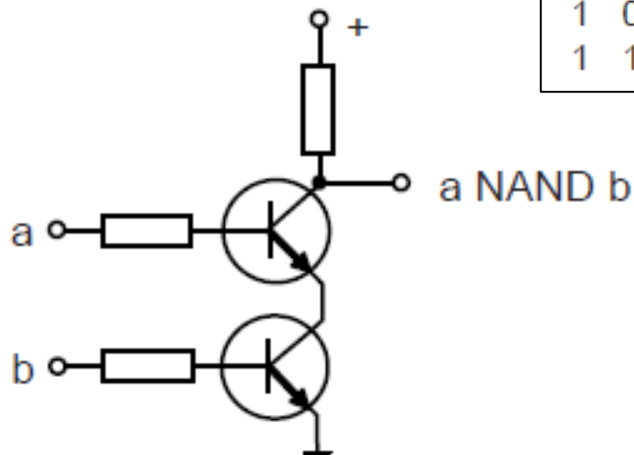


## NOR (NOT OR)



Technische Realisierung

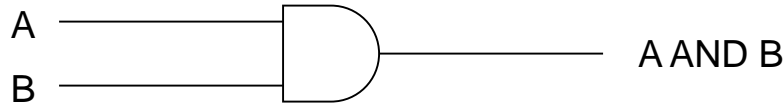
a	b	NOR	NAND
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0



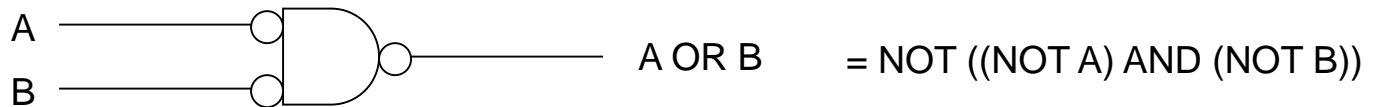
# Aufbau von Computersystemen

- Gatter

- AND

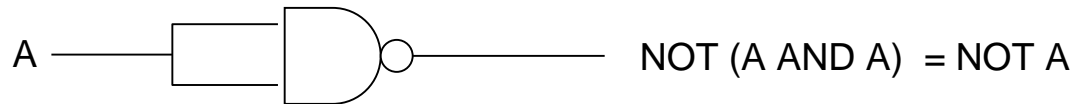


- OR



- **Alle Funktionen können durch NAND-Gatter nachgebildet werden**  
(siehe nächste Folie)

- Z.B. NOT:

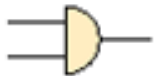
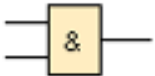
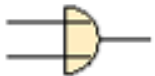


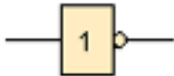
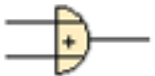
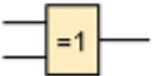
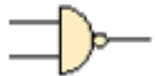
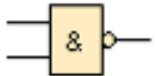
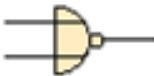

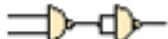
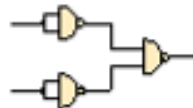

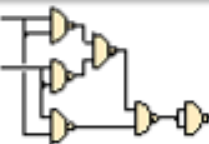

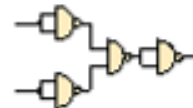


- **Logische Schaltungen**

- Hintereinanderschaltung von Gattern
  - Komplexe Funktionen
  - z.B. Addierer

# Aufbau von Computersystemen

## Zusammenfassung

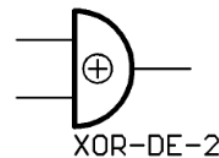
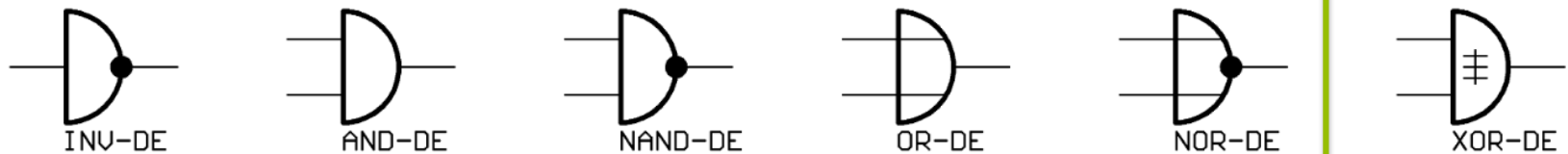
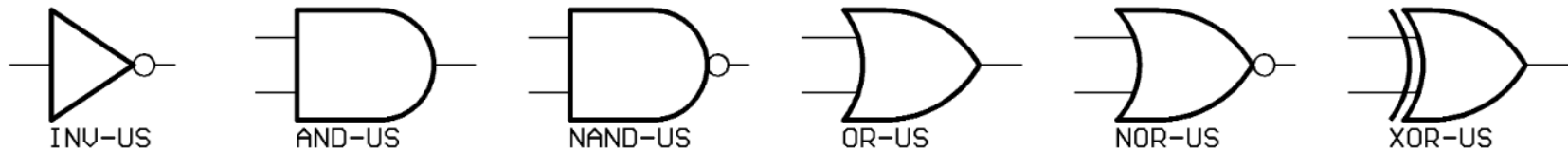
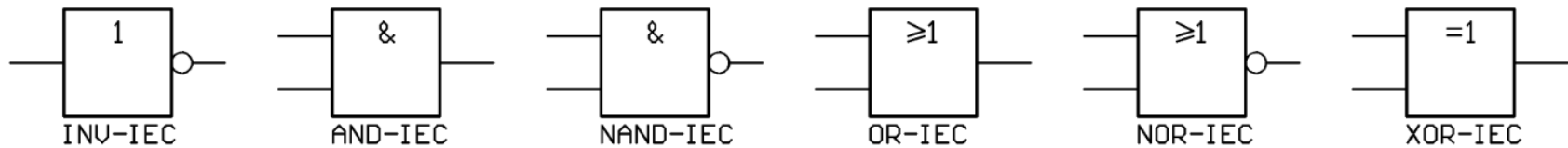
AND	OR	NOT	XOR	NAND	NOR																																																								
 	 	 	 	 	 																																																								
<table><tr><th>ein</th><th>aus</th></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>0</td></tr><tr><td>1 0</td><td>0</td></tr><tr><td>1 1</td><td>1</td></tr></table>	ein	aus	0 0	0	0 1	0	1 0	0	1 1	1	<table><tr><th>ein</th><th>aus</th></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>1</td></tr></table>	ein	aus	0 0	0	0 1	1	1 0	1	1 1	1	<table><tr><th>ein</th><th>aus</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	ein	aus	0	1	1	0	<table><tr><th>ein</th><th>aus</th></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>0</td></tr></table>	ein	aus	0 0	0	0 1	1	1 0	1	1 1	0	<table><tr><th>ein</th><th>aus</th></tr><tr><td>0 0</td><td>1</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>0</td></tr></table>	ein	aus	0 0	1	0 1	1	1 0	1	1 1	0	<table><tr><th>ein</th><th>aus</th></tr><tr><td>0 0</td><td>1</td></tr><tr><td>0 1</td><td>0</td></tr><tr><td>1 0</td><td>0</td></tr><tr><td>1 1</td><td>0</td></tr></table>	ein	aus	0 0	1	0 1	0	1 0	0	1 1	0
ein	aus																																																												
0 0	0																																																												
0 1	0																																																												
1 0	0																																																												
1 1	1																																																												
ein	aus																																																												
0 0	0																																																												
0 1	1																																																												
1 0	1																																																												
1 1	1																																																												
ein	aus																																																												
0	1																																																												
1	0																																																												
ein	aus																																																												
0 0	0																																																												
0 1	1																																																												
1 0	1																																																												
1 1	0																																																												
ein	aus																																																												
0 0	1																																																												
0 1	1																																																												
1 0	1																																																												
1 1	0																																																												
ein	aus																																																												
0 0	1																																																												
0 1	0																																																												
1 0	0																																																												
1 1	0																																																												
$a \wedge b$	$a \vee b$	$\bar{a}$	$(a \vee b) \wedge (\bar{a} \wedge \bar{b})$	$\overline{a \wedge b}$	$\overline{a \vee b}$																																																								
																																																													

Umsetzung nur mit NAND-Gattern



# Aufbau von Computersystemen

## Überblick über Standards für logische Gatter



**Bildnachweis:** <https://de.wikipedia.org/wiki/Datei:Logic-gate-index.png#file>  
<https://upload.wikimedia.org/wikipedia/commons/9/9f/Logic-gate-index.png>

IEC (International Electrotechnical Commission): Gemäß IEC 60617-12 Standard

US: Gemäß ANSI/IEEE Std 91-1984 und ANSI/IEEE Std 91a-1991 Standard

ANSI: American National Standards Institute

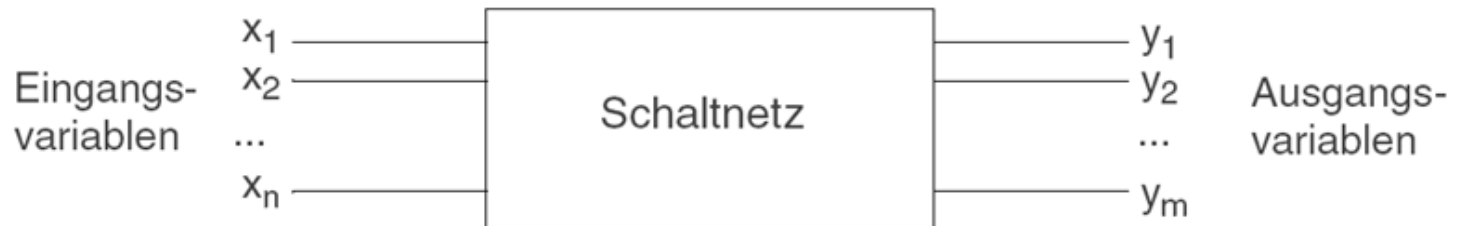
IEEE: Institute of Electrical and Electronics Engineers

DE: Gemäß DIN 40700

## ■ Ziel:

- Verknüpfung der Grundbausteine (UND, ODER, NEGATION) zu einem **Netzwerk (Schaltnetz)**, das **abhängig von Eingangswerten Ausgangswerte liefert**.

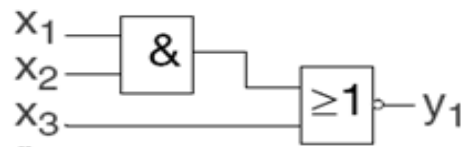
## ■ Prinzip:



## ■ Aufgabe:

- Realisierung von **Schaltfunktionen**  $y_i = f_i(x_1, x_2, \dots, x_n)$  ( $1 \leq i \leq m$ ) durch **Baumstrukturen** (keine Rückkopplung!) aus Verknüpfungsgliedern

## ■ Beispiel für $y_1$ :





# Aufbau von Computersystemen

## ■ Was ist eine Schaltfunktion?

Eine **Schaltfunktion** ist eine **Zuordnungsvorschrift**, die jeder der  $2^n$ -Wertekombinationen der Variablen  $x_1, x_2, \dots, x_n$  ( $x_j \in \{0, 1\}$ ) **eindeutig einen Wert**  $y_i = f_i(x_1, x_2, \dots, x_n)$  ( $1 \leq i \leq m$ ) ( $y_i \in \{0, 1\}$ ) zuordnet.

Sie ist also eine **zweiwertige Funktion** von **zweiwertigen Variablen**.

## ■ Wie kann die Schaltfunktion dargestellt werden?

### (1) Werte- oder Wahrheitstabelle (Werte- oder Wahrheitstafel)

In jeder der  $2^n$  Zeilen wird zu jeder der  $2^n$  möglichen Wertekombinationen der Variablen  $x_i$  der zugehörige Funktionswert  $y_i$  geschrieben.

Vorteil: Wertezuordnung direkt erkennbar

Nachteil: Unübersichtlichkeit bei größerer Anzahl an Variablen

### (2) Ausdruck

Aussagenlogischer Ausdruck basierend auf der Schaltalgebra

Vorteil: Rechenregeln der Booleschen Algebra gestatten es, Ausrücke zu vereinfachen oder miteinander zu vergleichen



# Aufbau von Computersystemen

## ■ Problembereiche

- **Gegebene Wahrheitstabelle in Gleichungsform bringen**
- Komplexe Ausdrücke **umformen** mit dem Ziel, den **Ausdruck** (und damit **die Schaltung**) zu **vereinfachen** und die **Zahl der Gatter zu reduzieren**

## ■ Normalformen

- **Konjunktive Normalform (KNF)**

Konjunktion von Disjunktionen:  $F = ( \bigwedge_i ( \bigvee_j L_{ij} ) )$

(mit  $L_{ij}$  als atomare Formel oder deren Negation)

Beispiel:  $F = a \wedge (b \vee c) \wedge (a \vee \neg d)$

- **Disjunktive Normalform (DNF)**

- Disjunktion von Konjunktionen:  $F = ( \bigvee_i ( \bigwedge_j L_{ij} ) )$

(mit  $L_{ij}$  als atomare Formel oder deren Negation)

Beispiel:  $F = u \vee (v \wedge w) \vee (u \wedge \neg z)$

- **Es gilt: Für jeden Ausdruck in DNF gibt es einen äquivalenten in KNF (und umgekehrt).**

- **Regel (zur Erzeugung eines Ausdruckes in DNF (KNF analog)):**
  - Jede Zeile der Wahrheitstafel mit **Wahrheitswert 1** trägt zu einem **Konjunktionsglied** bei. Die Konjunktionsglieder sind durch **Disjunktion** getrennt. Die Variablen der Konjunktionsglieder bestimmen sich wie folgt: Falls die Belegung der Variablen  $A_k$  in der betreffenden Zeile **1** ist, so wird  $A_k$  eingesetzt, sonst  $\neg A_k$ .

- **Beispiel:**

- Sei nebenstehende Wahrheitstafel gegeben.

Dann ergibt sich folgende DNF

$$(\neg A \wedge \neg B \wedge \neg C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge \neg B \wedge C)$$

bzw. folgende KNF

$$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (A \vee \neg B \vee \neg C)$$

$$\wedge (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

## ■ Vereinfachung von Ausdrücken:

- Auflösung einer gegebenen komplexen Gleichung von innen nach außen mit Hilfe der **Rechenregeln der booleschen Algebra**:

(1) Ersetze in einem Ausdruck jeden vorkommenden Teilausdruck

$$\begin{array}{lll} \neg\neg G & \text{durch} & G \\ \neg(G \wedge H) & \text{durch} & (\neg G \vee \neg H) \\ \neg(G \vee H) & \text{durch} & (\neg G \wedge \neg H) \end{array}$$

bis kein derartiger Ausdruck mehr vorkommt.

(2) Ersetze jedes Vorkommen eines Teilausdruckes

$$\begin{array}{lll} (F \vee (G \wedge H)) & \text{durch} & ((F \vee G) \wedge (F \vee H)) \\ ((F \wedge G) \vee H) & \text{durch} & ((F \vee H) \wedge (G \vee H)) \end{array}$$

bis kein derartiger Ausdruck mehr vorkommt.

## ■ Beispiel:

$$\begin{aligned} \neg(a \wedge \neg b) \vee (\neg\neg a \wedge b) &= (\neg a \vee \neg\neg b) \vee (a \wedge b) = \underbrace{(\neg a \vee b)}_{(F)} \vee \underbrace{(a \wedge b)}_{\vee (G \wedge H)} = \\ &= \underbrace{(\neg a \vee b \vee a)}_{\text{ist immer 1}} \wedge (\neg a \vee b \vee b) = (\neg a \vee b) \end{aligned}$$

ist immer 1

# Aufbau von Computersystemen

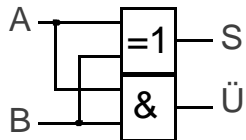
- **Addierglied = Schaltnetz zur Addition zweier Dualzahlen**
- **Mögliche Additionen einstelliger Dualzahlen im Dualsystem:**

A	+	B	Summe	Übertrag
0	+	0	0	0
0	+	1	1	0
1	+	0	1	0
1	+	1	0	1

- **Schaltfunktion in DNF:**

- $$S = (\neg A \wedge B) \vee (A \wedge \neg B)$$
$$\ddot{U} = (A \wedge B)$$

- **Schaltnetz:**

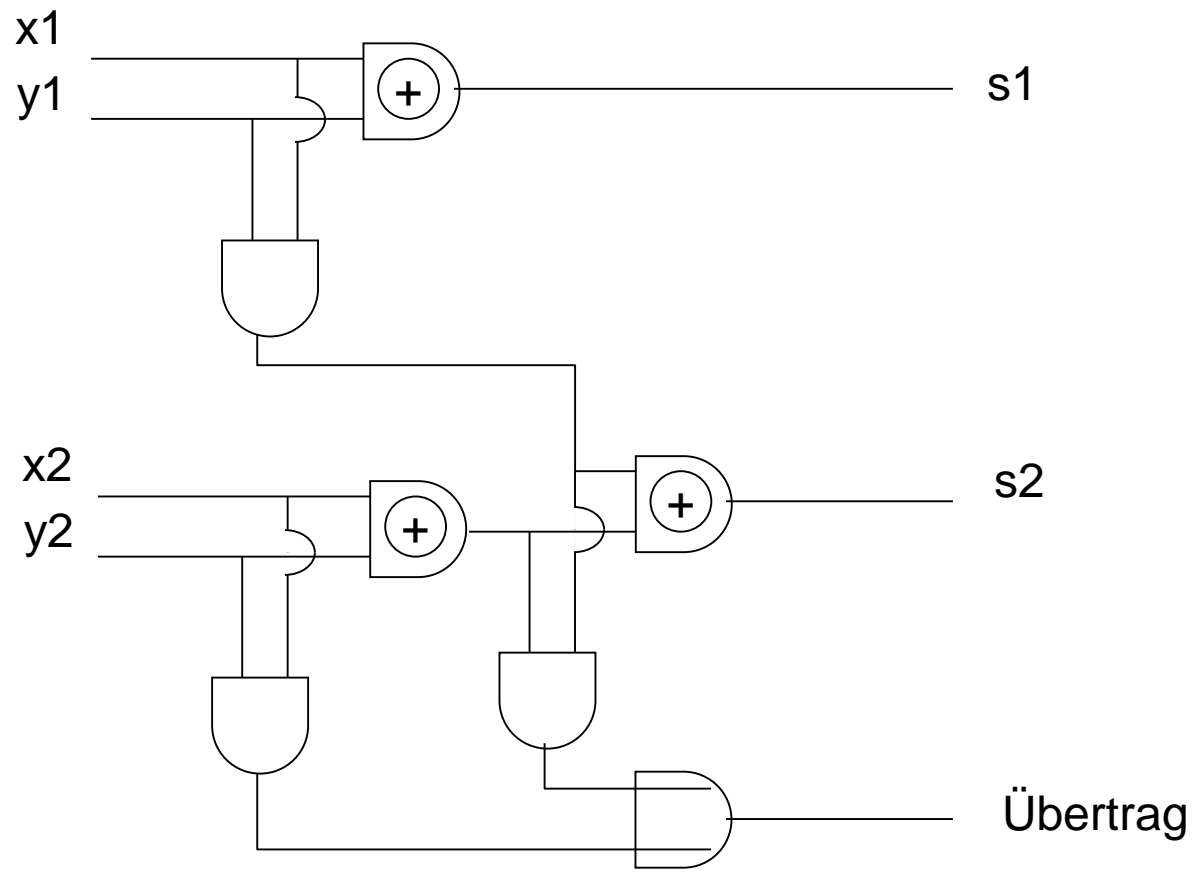


- **Schaltzeichen:**



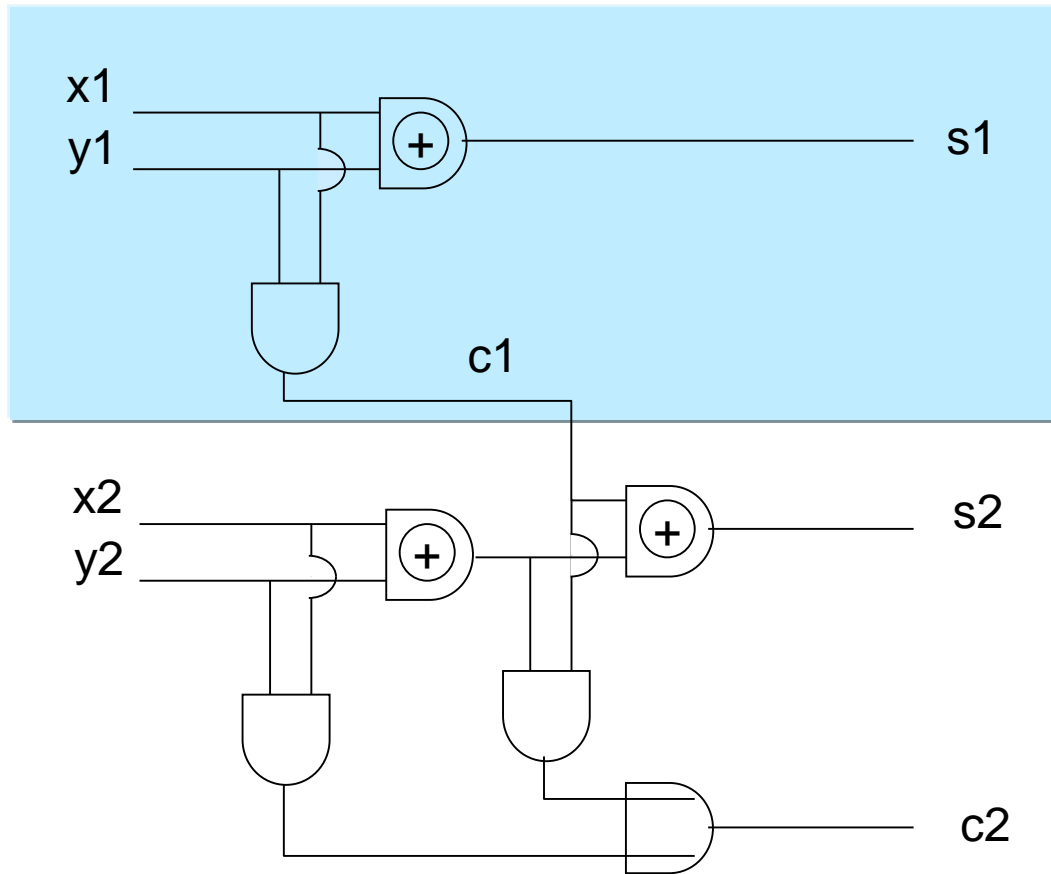
Legende:  
=1 : EXOR-Gatter  
HA: Halbaddierer

## ■ Addierer



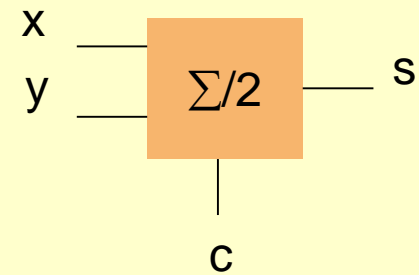


## ■ Addierer

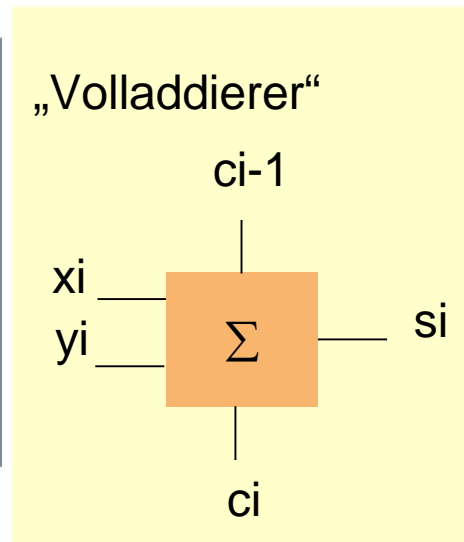
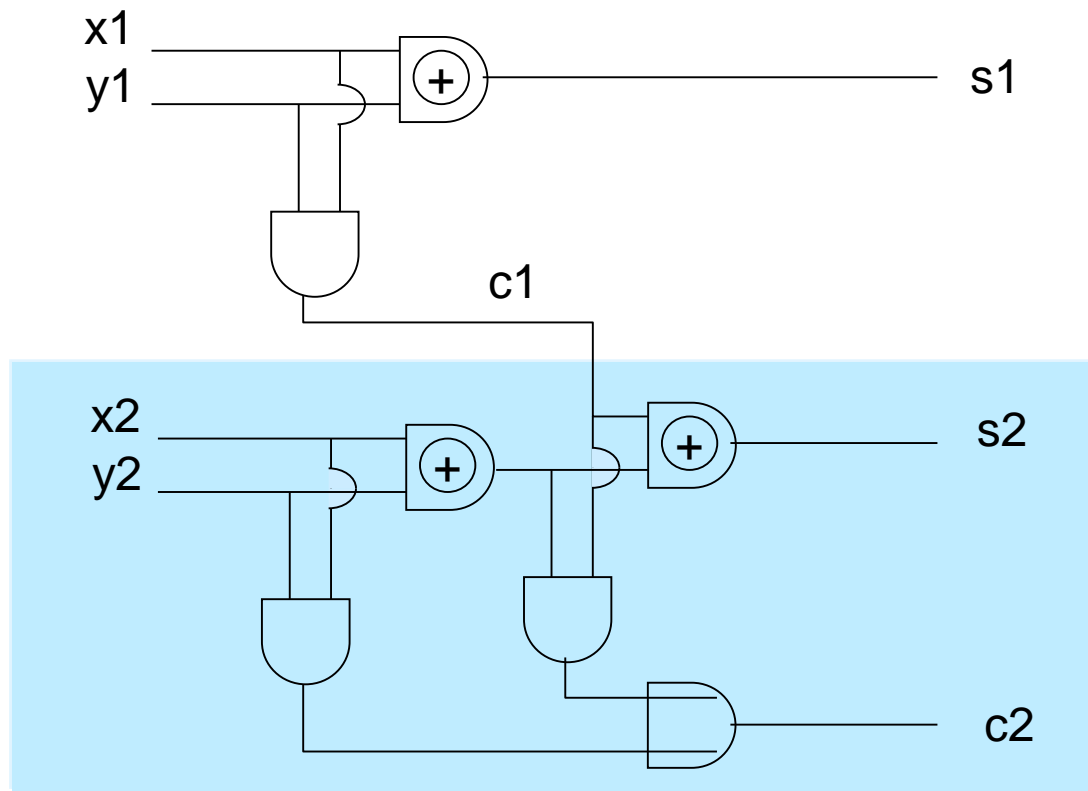


„Halbaddierer“

x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

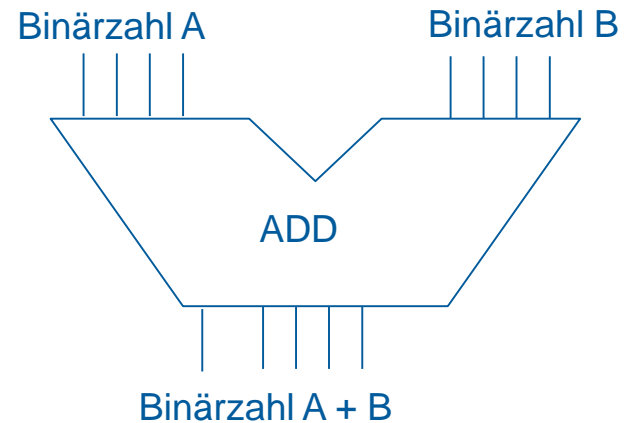
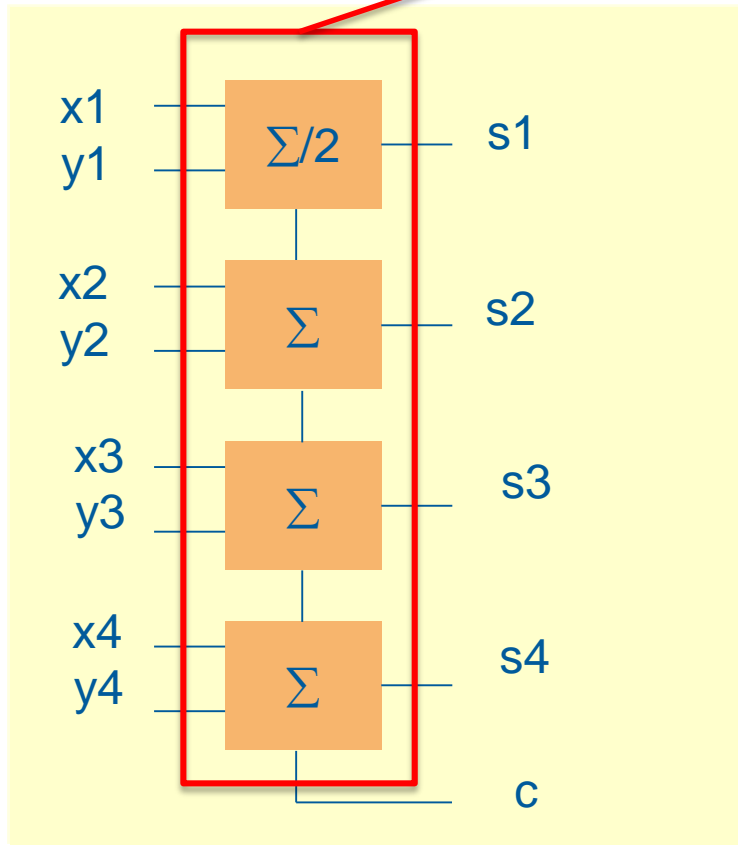


## ■ Addierer



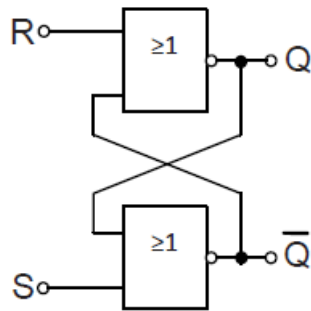
## ■ Addierwerk

Schaltnetz



- **Problematik der vorgestellten passiven Schaltnetze:**
  - Sie **behalten ihren Zustand nicht**, was aber zum Zwecke einer Speicherung notwendig ist!
- **Aktive Schaltnetze:**
  - Können Schaltvariablen **aufnehmen**, **speichern** und **abgeben** und heißen deshalb auch **Speicherglieder**
  - Speicherglieder mit dieser Eigenschaft sind die bistabilen Kippglieder, auch **Flipflops** genannt (**rückgekoppelte** Schaltungen).

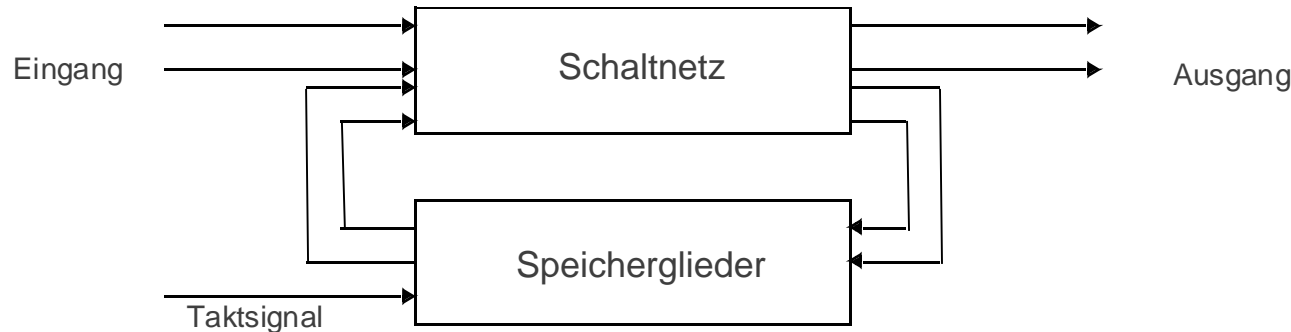
## ■ Basis-Flipflop aus NOR-Schaltgliedern (RS-Flipflop)



$Q_{\text{alt}}$	R	S	$Q_{\text{neu}}$
0	?	0	0
0	0	1	1
1	1	0	0
1	0	?	1
—	1	1	verboten

- **Ruhezustand:**  $R = S = 0$ ;  $Q = 0$
- Ein Impuls auf S (set) setzt Q auf 1
- Ein Impuls auf R (reset) setzt Q auf 0
- Fällt der Impuls (auf R oder S) wieder auf 0 ab, so bleibt der **vorherige Wert von Q erhalten** → Schaltung merkt sich, ob letzte Aktion ein *set* oder *reset* war → **ideale Schaltung für Register** innerhalb eines Prozessors

## ■ Schaltwerk = Schaltnetz + Speicherglieder



## ■ Charakterisierung:

### ■ Schaltnetz:

Wert der Ausgangsvariablen ist zu irgendeinem Zeitpunkt *nur vom Wert der Eingangsvariablen abhängig*.

### ■ Speicherglieder:

Nehmen taktabhängig einen *stabilen Zustand ein* und speichern ihn (*innerer Zustand des Schaltwerkes*)

- **Was ist ein Register?**

Ein Register R ist eine **geordnete Menge von Speicherelementen**.

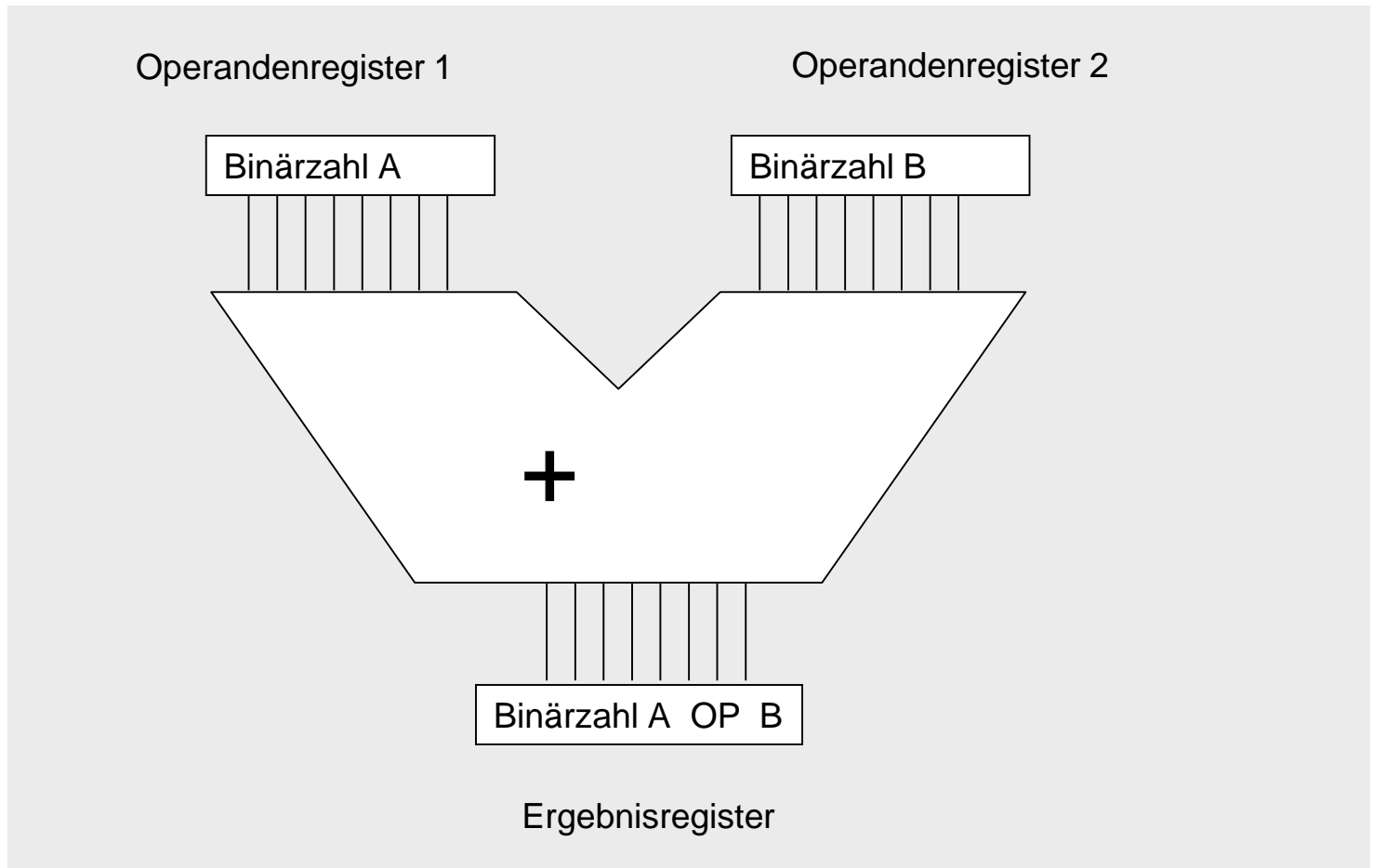
Die abgespeicherte Information nennt man Inhalt von R.

Technisch sind die Register aus **Flipflops** aufgebaut.

- **Was kann gespeichert werden?**

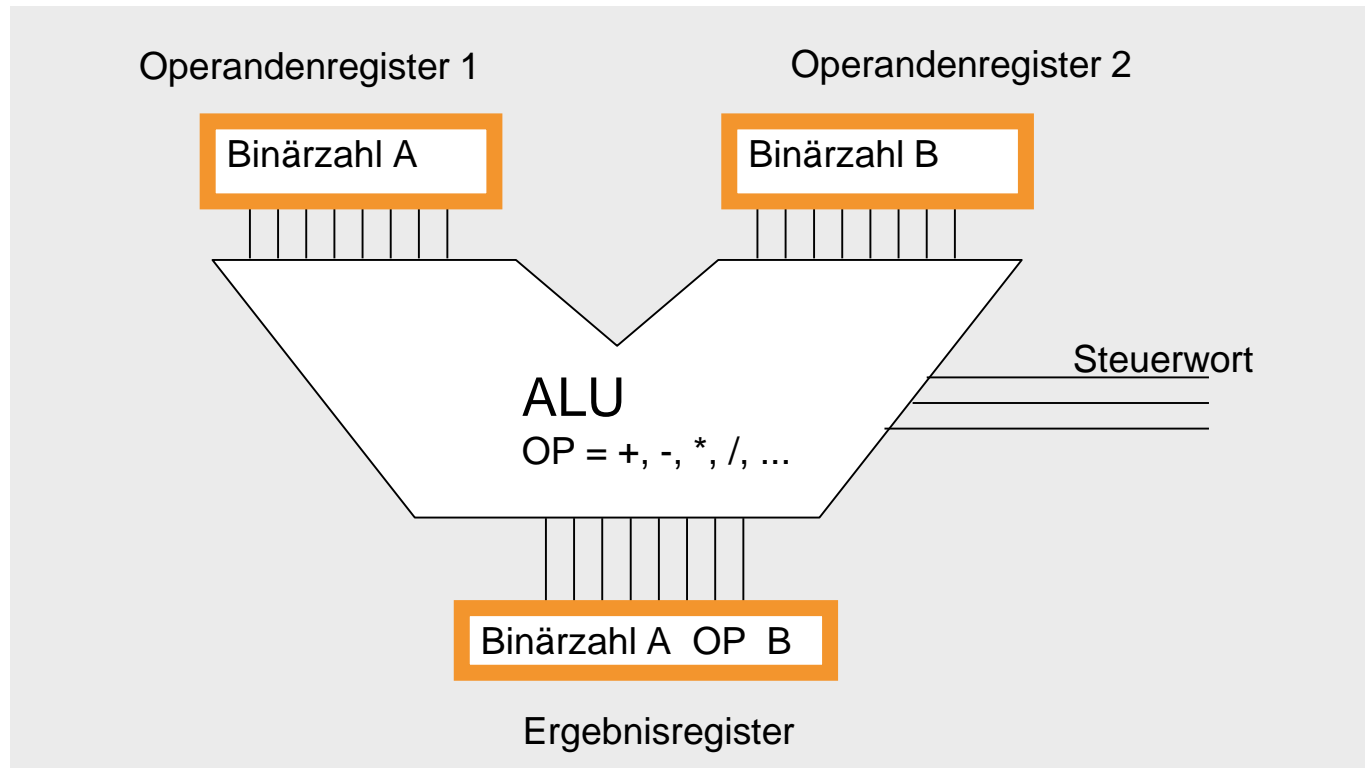
- **Befehle** (Anweisung zur Ausführung einer bestimmten Operation)
- **Adressen** (zur Bezeichnung von Speicherzellen, die die Operanden enthalten)
- **Daten** bei Rechenoperationen

## ■ Addierwerk



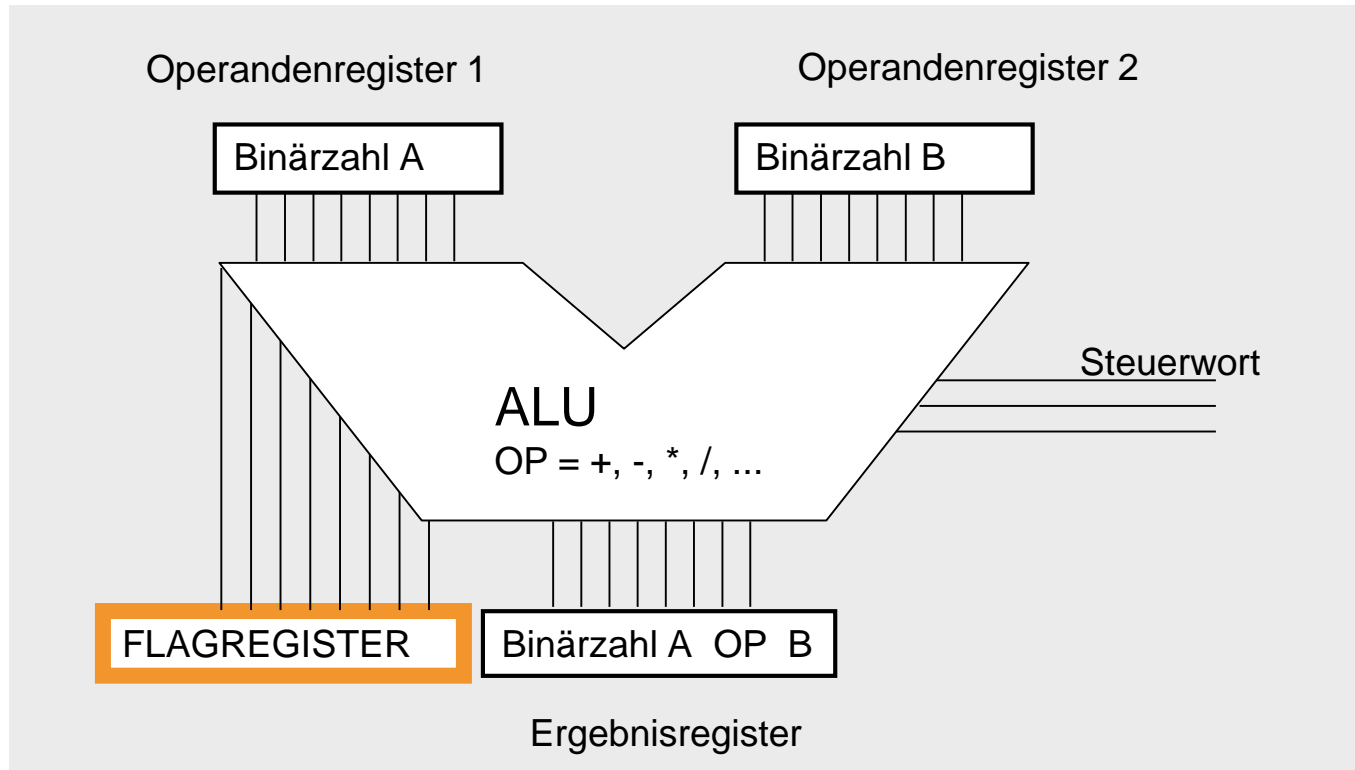


## ■ Rechenwerk (Arithmetisch Logische Einheit)



Register = Wortspeicher  
(Registerbreite = Wortlänge des Rechners, 8/16/32/64 Bit)

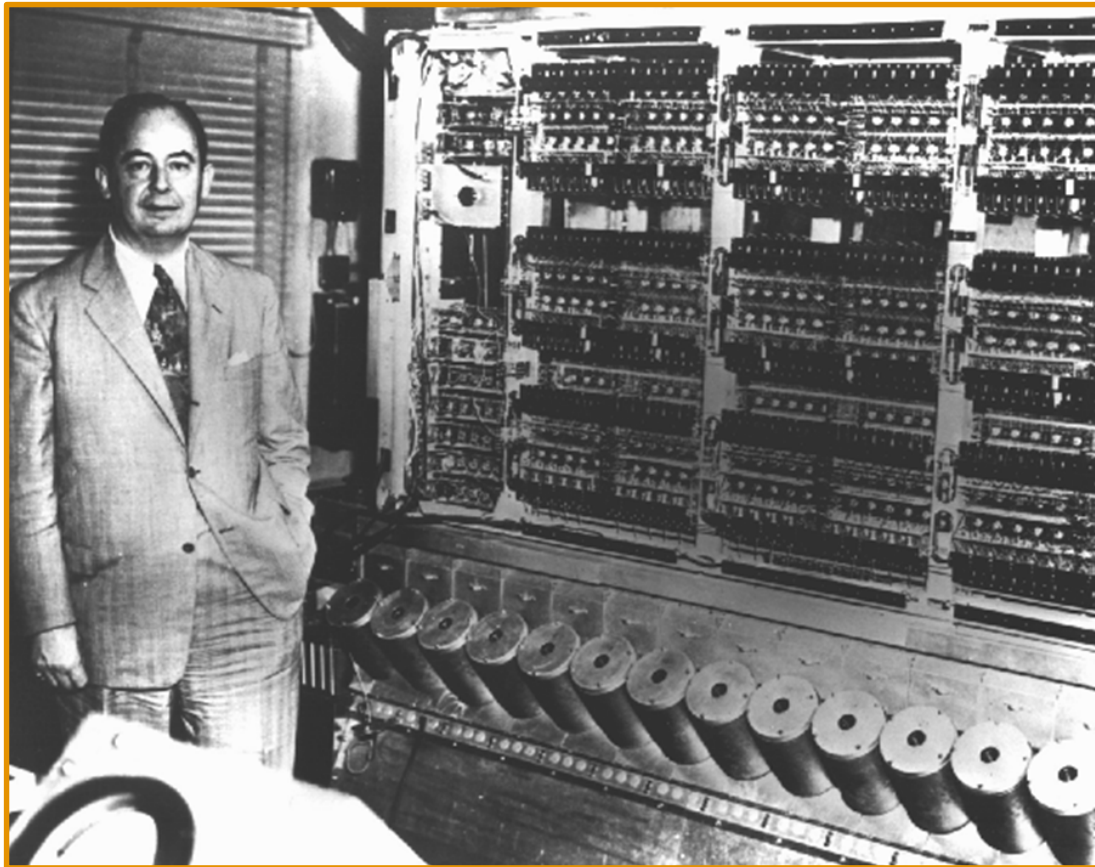
## ■ Rechenwerk (Arithmetisch Logische Einheit)



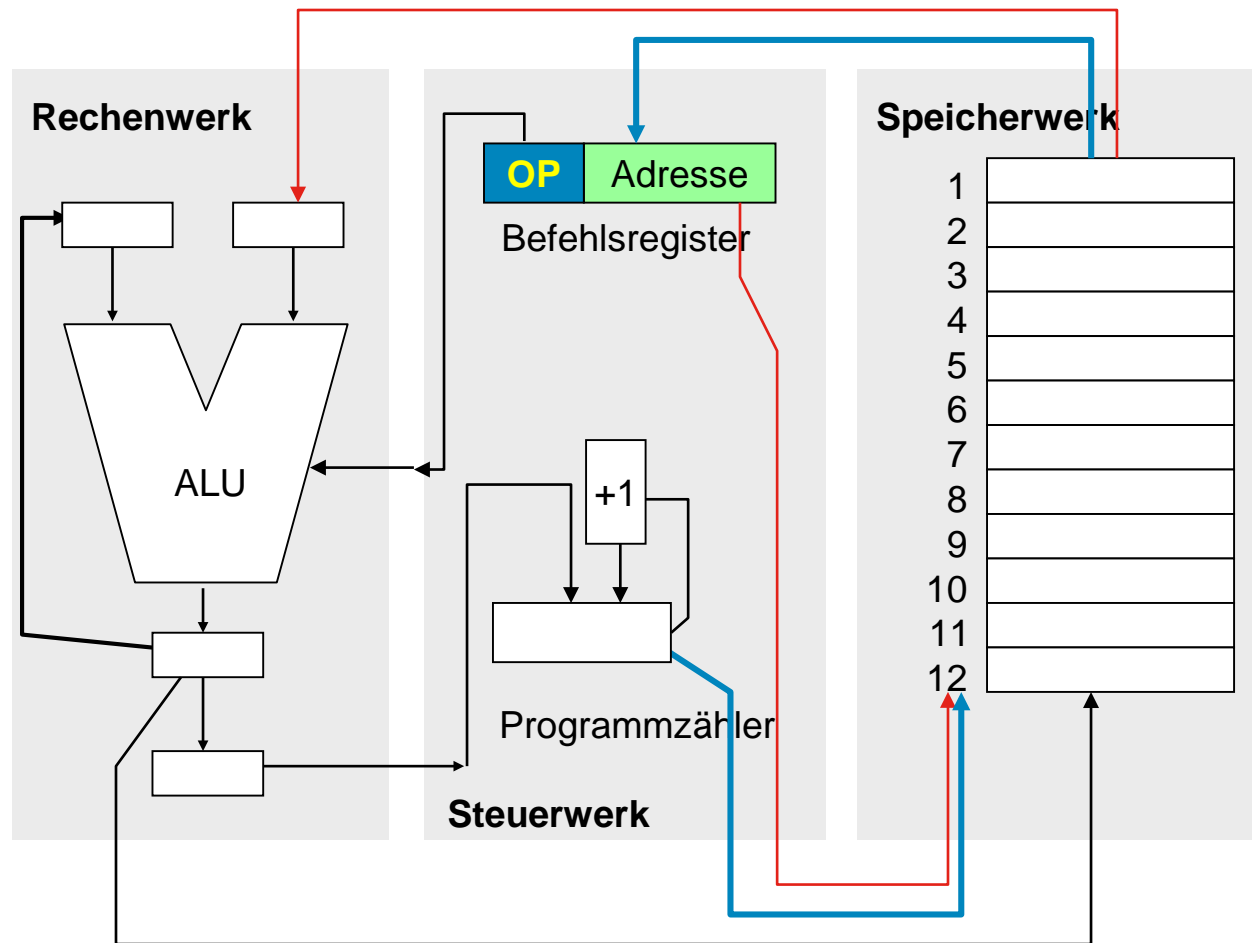
Flagregister für Ausnahmefälle  
z.B. Overflow, Vorzeichen, Ergebnis = 0, usw.

- **Von-Neumann Architektur**

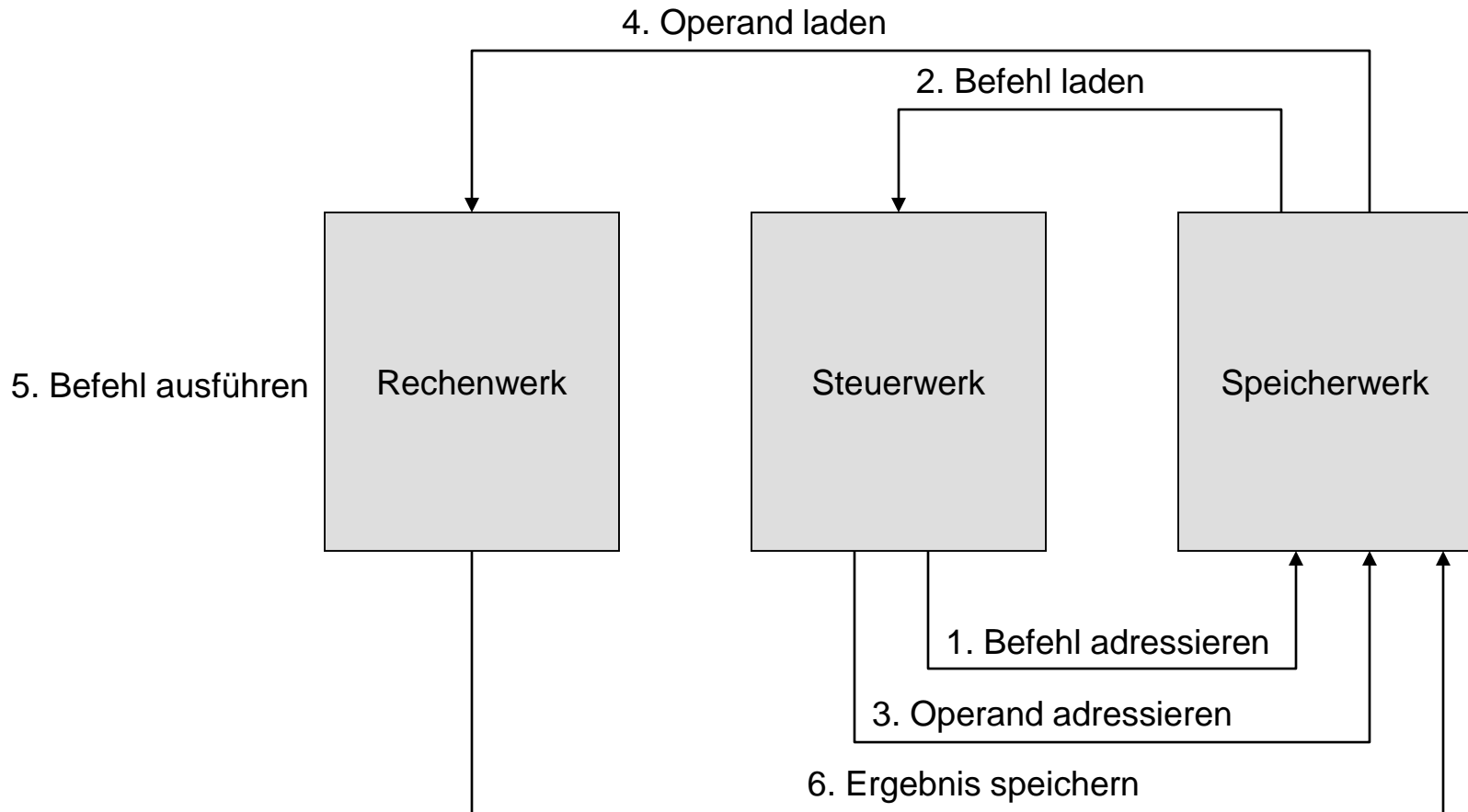
- John von Neumann, österreichisch-ungarischer Mathematiker (1903-1957)



### ■ Von-Neumann Architektur



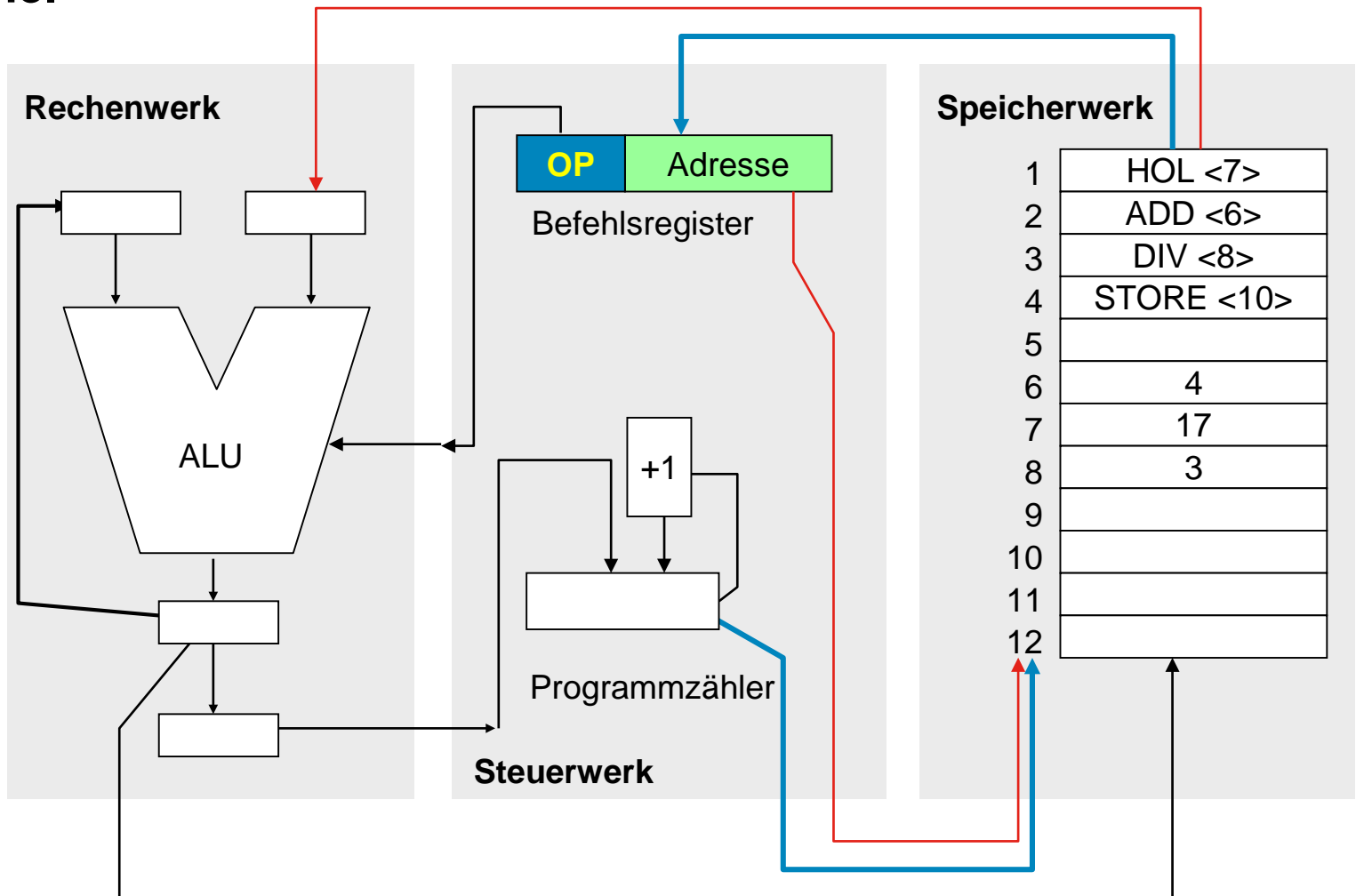
### ■ Von-Neumann Architektur - Befehlszyklus



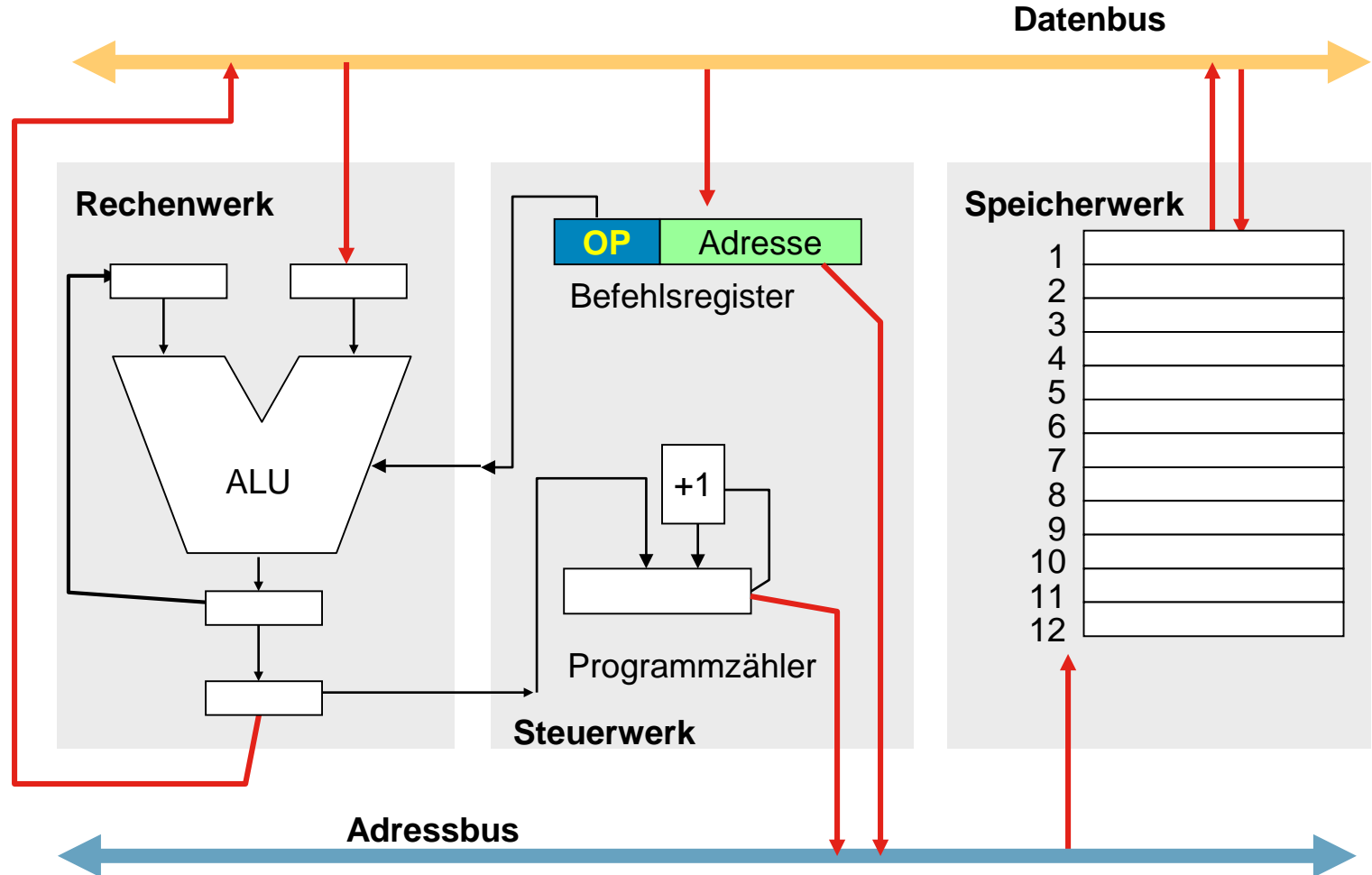
# Aufbau von Computersystemen

## Von-Neumann Architektur

### ■ Beispiel



### ■ Adressbus und Datenbus



### ■ Von-Neumann Prinzipien

- Der Rechner besteht aus **fünf Funktionseinheiten**
  - **Rechenwerk**
  - **Steuerwerk (Leitwerk)**
  - **Speicherwerk**
  - **Eingabewerk**
  - **Ausgabewerk**
- Die Struktur des Rechners ist **unabhängig von den zu bearbeitenden Problemen**
  - Zur Lösung eines Problems muss ein **Programm** von außen **in den Speicher** eingegeben werden
  - Ohne eine Programmeingabe ist die Maschine **nutzlos**
- Programme, Daten, Zwischen- und Endergebnisse werden **in dem selben Speicher** abgelegt



# Aufbau von Computersystemen

## Von-Neumann Architektur

- Der Speicher ist in **gleichgroße Speicherzellen** aufgeteilt
  - Die Speicherzellen sind **fortlaufend durchnummeriert**
  - Die Nummer einer Zelle ist die **Adresse der Zelle**
  - Über die Adresse einer Zelle kann **deren Inhalt** abgerufen oder verändert werden
- **Aufeinanderfolgende Befehle** eines Programms werden in **aufeinanderfolgenden Speicherzellen** abgelegt
  - Die Auswahl des **nächsten Befehls** geschieht vom Steuerwerk durch **Erhöhen des Befehlszählers** um eins
- Durch **Sprungbefehle** kann von der vorgegebenen Bearbeitungsreihenfolge abgewichen werden
  - Um einen Sprungbefehl durchzuführen wird der **Programmzähler** mit der **Adresse des nächsten Befehls** überschrieben

# Aufbau von Computersystemen

## Von-Neumann Architektur

- Es gibt zumindest folgende **Befehle**:
  - **Arithmetische** Befehle: Addieren, Subtrahieren, Multiplizieren, etc. ...
  - **Logische** Befehle: Vergleiche (z.B. > ; = ; < ; ...); AND; OR; NOT; ...
  - **Transportbefehle**
    - Vom Speicher ins Rechenwerk
    - Von der Eingabe in den Speicher
    - Vom Speicher in die Ausgabe
  - **Bedingte Sprünge**
    - Abhängig vom Ausgang einer Operation wird der Programmzähler überschrieben oder auch nicht
- Weitere Befehle:
  - **Schieben** (Shift): 00001110  $\xrightarrow{\text{shiftleft}}$  00011100
  - **Unterbrechen** (Interrupt): Unterbrechen eines Programms
  - **Warten**: Blockieren eines Programms
  - ...

# Aufbau von Computersystemen

## Von-Neumann Architektur

- Die CPU (Central Processing Unit):
  - Auch: **Prozessor**
  - **Die Einheit, die eine Befehlsfolge ausführt**
  - Im Prinzip **Rechenwerk und Steuerwerk zusammengekommen**
  - Die CPU kann **weitere Elemente** enthalten
  - Die CPU enthält **mindestens**:
    - Zwischenspeicher für Daten (sog. **Register**).
    - Die Adresse des gegenwärtigen bzw. nächsten Befehls (**Programmzähler**).
    - Speicher für einen Maschinenbefehl (**Befehlsregister**).
    - Die notwendige Logik um einen Befehl ausführen zu können (**ALU**).
  - Die CPU führt ständig den sogenannten **Befehlszyklus** aus:
    - Die einzelnen Bearbeitungsschritte werden **durch einen Takt** veranlasst

# Aufbau von Computersystemen

## Von-Neumann-Architektur

- → **Definition Rechnerarchitektur**

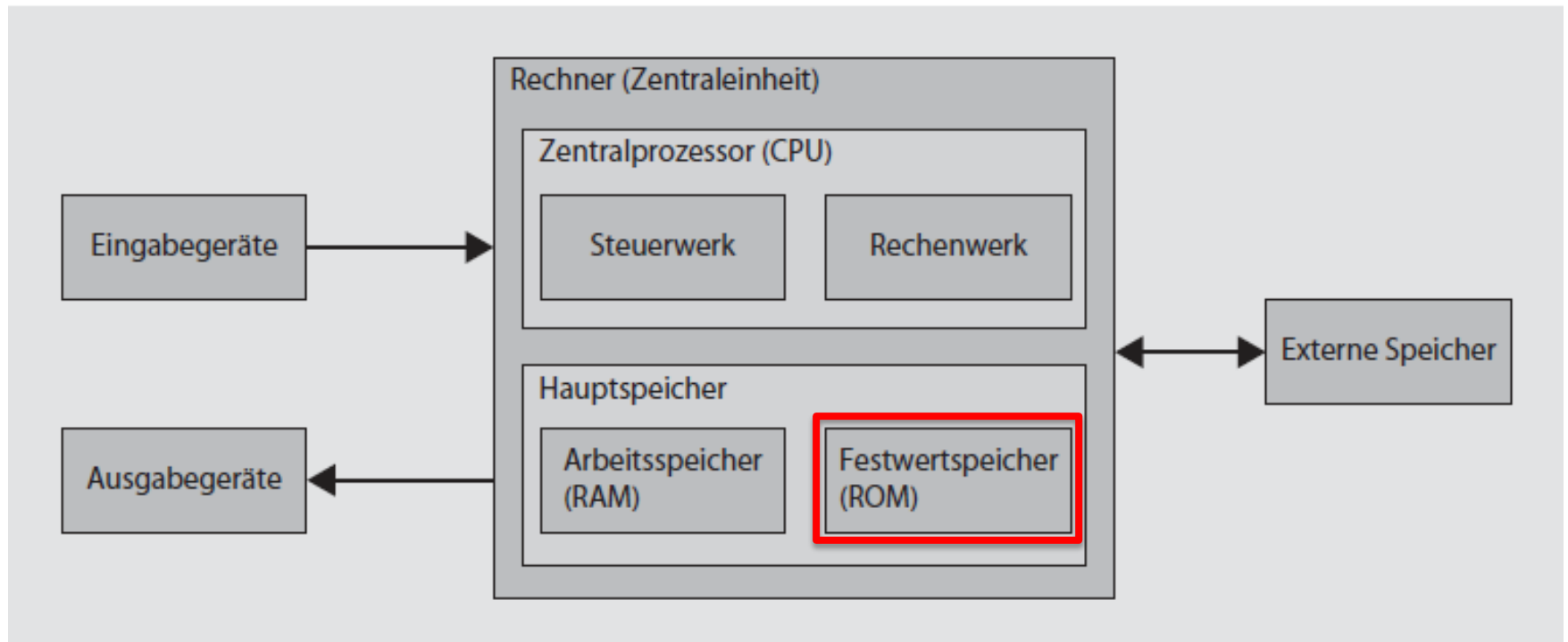
- **Interne Struktur eines Rechners**

- Aufbau aus verschiedenen **Komponenten**
    - Organisation der **Arbeitsabläufe**
    - → **von-Neumann-Architektur** bestehend aus (angewandt auf heutige Rechner):
      - **Zentralprozessor** (CPU – Central Processing Unit): Interpretiert und führt Befehle (Instruktionen) eines Programms einzeln nacheinander aus
      - **Hauptspeicher**: Speichert die zum Zeitpunkt der Verarbeitung auszuführenden Programme und die dafür benötigten Daten
      - **Datenwege (Busse)**: sind für den Datentransfer zwischen den Komponenten des Rechners (**interne Datenwege**) und zwischen dem Rechner und den peripheren Geräten (**periphere Datenwege oder Eingabe-/Ausgabesystem**) zuständig
  - CPU und Hauptspeicher befinden sich auf einer **Platine (Motherboard)**
  - Bei **Parallelrechnern** enthält die Zentraleinheit mehrere Prozessoren

Zentraleinheit

# Aufbau von Computersystemen

## Von-Neumann-Architektur





# Aufbau von Computersystemen

## Befehlssätze

- Erinnerung: **Arbeitsweise Prozessor**
  - **Steuerwerk** veranlasst das **Rechenwerk**, die im **Operationsteil des Befehls** enthaltene Operation mit den angegebenen Operanden auszuführen
  - **Rechenwerk** übernimmt die vom Steuerwerk entschlüsselten Befehle und **führt sie aus**
  - Die Operationen werden entweder durch **elektronische Schaltungen** oder durch **Mikroprogramme**, die in einem speziellen Festwertspeicher (**ROM**) enthalten sind, ausgeführt → **Befehlssatz = Maschinenbefehle**, die die CPU ausführen kann
  - Art und Zusammenstellung der Befehle wird als **Befehlssatzarchitektur (Instruction Set Architecture, ISA)** bezeichnet
    - Beispiel: **x86-CPU**: nicht ein 8086-Prozessor gemeint, sondern ein Prozessor, der **denselben Befehlssatz ausführen kann wie 8086-Prozessor aus dem Jahr 1978**
    - Prozessoren mit x86-Befehlssatz finden sich in **Desktop-Computern** und **Laptops**
    - **ARM (Advanced RISC Machines)** ist ein zweiter bekannter Befehlssatz: zu finden in **Smartphones** und **Tablet-Computern**



# Aufbau von Computersystemen

## Exkursion: x86-Befehlssatz

- **Arithmethische Befehle:**

- ADD, ADC (add with carry), DIV, IDIV, MUL, IMUL, SUB, SBB (Subtract with borrow - vorzeichenlos)

- **Logische Befehle:**

- AND, OR, NOT, XOR

- **Sprungbefehle:**

- JA, JAE, JB, JBE, JC, JCXZ, JE, JMP,...

- **Transport:**

- MOV, PUSH, POP, XCHG

- **Schieben:**

- SHL, SHR, ROL, ROR

- **Unterbrechen:**

- INT

- **Warten:**

- WAIT (...dass BUSY-Eingang nicht aktiv ist – BUSY dient der Kommunikation mit 8087-Floating Point-Koprozessor – wenn 8087 aktiv ist, ist BUSY gesetzt)

- **Unterprogrammaufrufe:**

- CALL, RET

# Aufbau von Computersystemen

## Befehlssätze

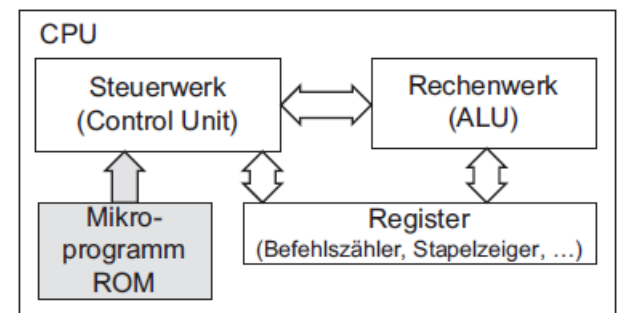
- **Optimierbarkeit von Hardware**

- Hängt mit den **Eigenschaften des Befehlssatzes** eng zusammen
- **Je einfacher** Befehlssatz einer CPU ist, desto **leichter** ist dessen **Ausführung** optimierbar
- Komplexe Befehle → komplexe Hardware → schwieriger optimierbar

- Ziel erster großer Mikroprozessoren der 60er Jahre (IBM System/360 Familie): sollten **universell** für verschiedene Anwendungszwecke **geeignet** sein

- **CISC (Complex Instruction Set Computer)**: Schwer in Hardware umsetzbar →  
Stattdessen erhielt CPU interne (für Programmierer unsichtbar) **Mikro-Maschinenbefehle** über welche die komplexeren außen sichtbaren Befehle der CPU in Form von **Mikroprogrammen** implementiert wurden

- **CISC → viele** verschiedene Befehle und Befehlsvarianten







# Aufbau von Computersystemen

## Befehlssätze

- Das Steuerwerk der CPU übersetzt die Maschinenbefehle eines Programms intern in **Aufrufe von Mikroprogrammen** (befinden sich im **Mikroprogramm-ROM**)
- **Bewertung Mikroprogramme:**
  - Architektur ist **sehr flexibel** (internen Mikroprogramme **leichter anpassbar, korrigierbar** oder **austauschbar** als die auf Silizium gebrannte Hardware-Architektur)
  - Prozessoren können **intern** sehr **verschieden strukturiert** sein, nach außen **denselben Befehlssatz** anbieten → Rechner mit **unterschiedlicher HW** können trotzdem **dieselben Programme ausführen**, ohne diese neu in Maschinensprache übersetzen zu müssen → Grundlage für **Abwärtskompatibilität** von Prozessoren
  - Problem der **schwankenden Ausführungszeiten** der verschiedenen Befehle (sind vom Mikroprogramm abhängig): ADD zweier Register in HW **2 Takte**, MUL in Mikroprog. **38 Takte**
  - Viele verschiedene **Adressierungsarten** erschweren Hardware-Optimierung zusätzlich: einige Befehle arbeiten beispielsweise mit **Registern**, andere Varianten derselben Befehle verwenden **Inhalte des Hauptspeichers** als Operanden



# Aufbau von Computersystemen

## Befehlssätze

- 1980: Start eines Projektes an der Berkeley Universität mit dem Ziel, **ohne Mikroprogramme** und mit **stark vereinfachten Befehlssätzen** auszukommen
- Konzept wurde unter dem Namen **RISC** (**Reduced Instruction Set Computer**) bekannt
- **Eigenschaften:**
  - Jeder Befehl **vollständig in Hardware** ausgeführt. Auf **Mikroprogramme** wird **verzichtet**.
  - Zahl benötigter Taktzyklen je Befehl kann **deutlich gesenkt** werden
  - Jeder Befehl muss innerhalb **eines Zyklus** geladen und dann möglichst **einheitlich** weiter verarbeitet werden können
  - **Viele Register (> 32 bis zu 256)** → Zwischenergebnisse in Registern speichern  
→ **Zugriffe auf den Hauptspeicher werden vermieden**
  - **Load und Store-Architektur:** (Rechen-)Befehle sind **nur auf Registern** erlaubt → Datum wird zuerst in eines der **Register geladen (Load)**, dort verarbeitet und das Ergebnis wird aus dem Ergebnis-Register zurück in den **Hauptspeicher geschrieben (Store)**  
→ schränkt die **Variantenvielfalt** der Befehle enorm ein (meisten Befehle beziehen sich **nur noch auf Registerinhalte**)



# Aufbau von Computersystemen

## Befehlssätze

- Die über diese Vereinfachungen erreichte einfachere Struktur ermöglicht den Bau **effizienterer Compiler und weiterer Hardware-Optimierungen**
- Die derzeit in Desktop-PCs eingesetzten x86-CPUs, etwa der Core i9 der Firma Intel sind **Mischformen zwischen RISC und CISC**
- Ein **RISC-Kern** wird um **Mikroprogramme für die komplexeren Befehle** ergänzt, so dass die **Grenze zwischen RISC und CISC fließend** ist
- **Abwärtskompatibilität:** Dank der Mischformen können auch moderne x86-Prozessoren daher noch die Befehle der 8086-CPU aus dem Jahr 1978 unterstützen, auch wenn diese Befehle mithilfe von Mikroprogrammen und intern über einen RISC-Kern ausgeführt werden



# Aufbau von Computersystemen

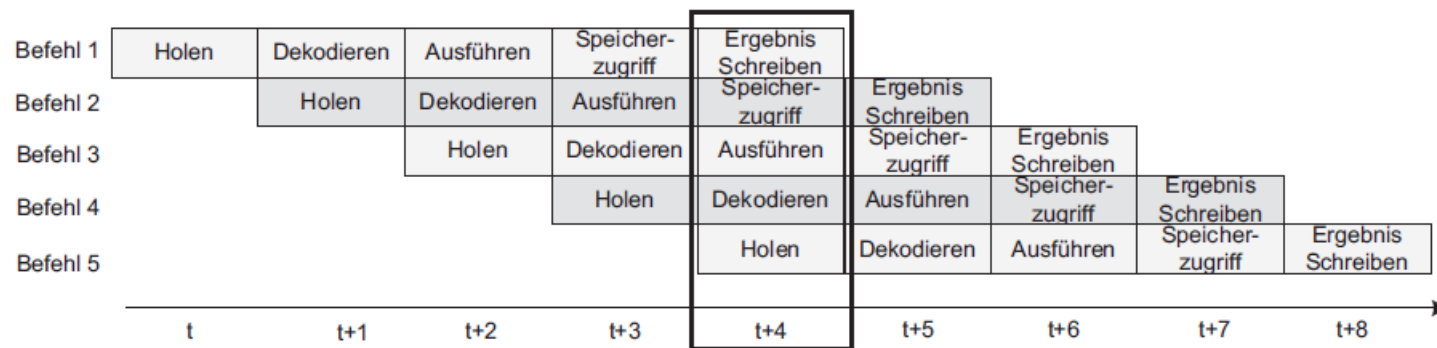
## Weitere Optimierungen

- Steuerwerk und Rechenwerk arbeiten nach dem sogenannten **Pipelineprinzip**
- **Ein Befehl** wird **nacheinander** (wie in einer Pipeline) zunächst vom Steuerwerk und anschließend vom Rechenwerk abgearbeitet
- Während das Rechenwerk einen Befehl ausführt, bereitet das Steuerwerk **zeitlich parallel** dazu (»überlappend«) schon die **nächsten Befehle** auf
- Das **vorsorgliche Holen** der sequenziell nachfolgenden Befehle in den Cache heißt »**Prefetching**«

# Aufbau von Computersystemen

## Weitere Optimierungen

- Beispiel RISC (konkret: MIPS-Prozessor)
  - In einem Fließband wird die **Ausführung eines Befehls** auf der CPU in **mehrere Schritte aufgeteilt**
  - Für **jeden Schritt** gibt es eine **Stufe** in dem Fließband
  - Die Stufen sind jeweils durch ein **Register als Zwischenspeicher** getrennt
  - Idealerweise kann **ein Schritt** innerhalb **eines Prozessortaktes** ausgeführt werden
  - **Befehlspipeline:**



# Aufbau von Computersystemen

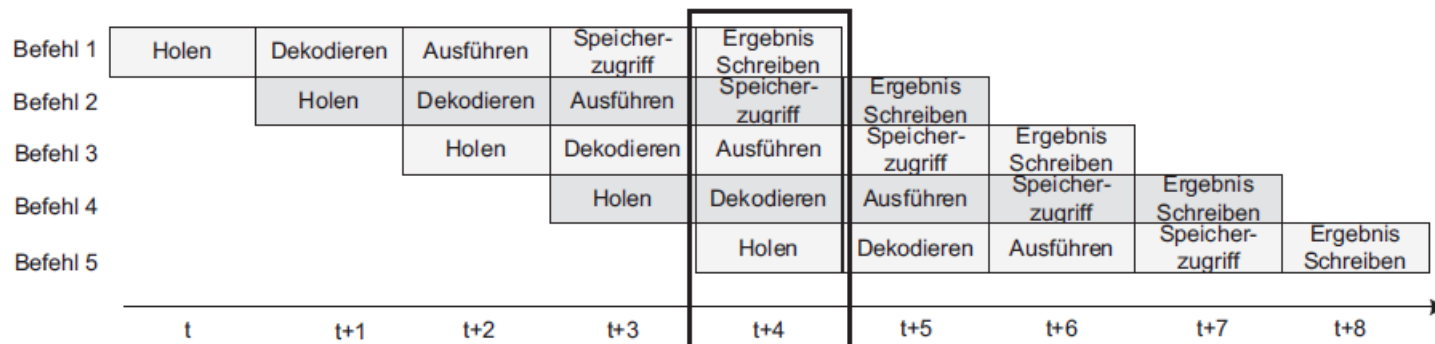
## Weitere Optimierungen

- Beispiel RISC (konkret: MIPS-Prozessor)
  - **Befehl holen** (Instruction Fetch): Befehlszähler zeigt auf eine Speicheradresse → Inhalt wird geladen und als Befehl interpretiert → Befehlszähler inkrementieren
  - **Befehl dekodieren** (Decode): Befehl dekodieren und notwendige Operanden aus Registern laden (Operand Fetch).
  - **Befehl ausführen** (Execute)
  - **Speicherzugriff** (Memory Access): Nur bei Befehlen, die auf den Hauptspeicher zugreifen (Load und Store), findet Speicherzugriff in diesem Schritt statt. Adresse, auf die zugegriffen wird, wurde im vorhergehenden Execute-Schritt berechnet.  
**Dieser Schritt muss nicht immer vorhanden sein!!!**
  - **Ergebnis zurückschreiben** (Write Back): Das Ergebnis des Befehls wird **in ein Register** zurückgeschrieben

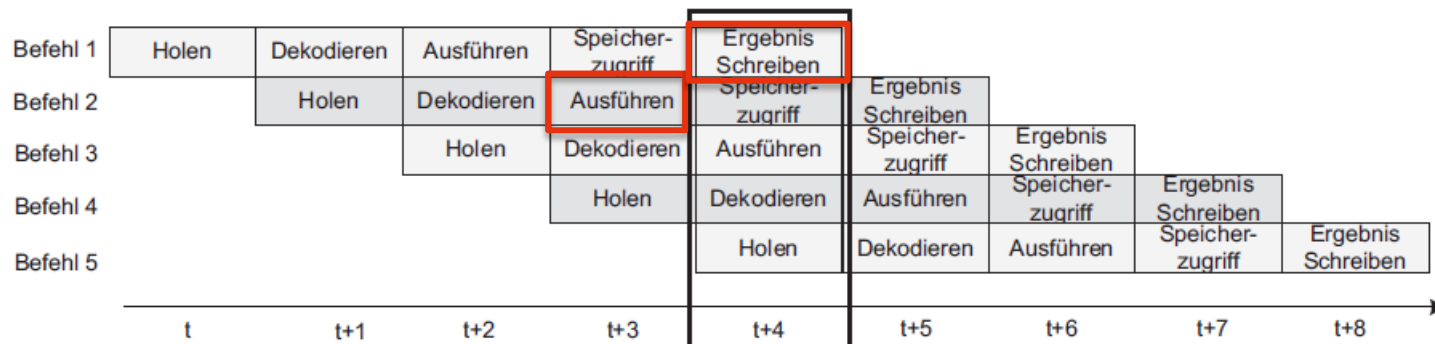
# Aufbau von Computersystemen

## Weitere Optimierungen

- Ideale Fließband-Architektur:
  - In **jedem Prozessortakt** kann ein Befehl am Anfang des Fließbands **gestartet** und ein anderer Befehl am Ende des Fließbands **beendet** werden
  - Zum Zeitpunkt  $t+4$  werden **fünf Befehle gleichzeitig** in verschiedenen Verarbeitungsschritten verarbeitet
  - Ohne Pipeline wären **fünf (oder mehr) Prozessortakte für jeden Befehl** erforderlich
  - Alle modernen Prozessoren enthalten daher **mehrere parallel arbeitende Pipelines**
  - Ist der Prozessor 5 mal schneller?

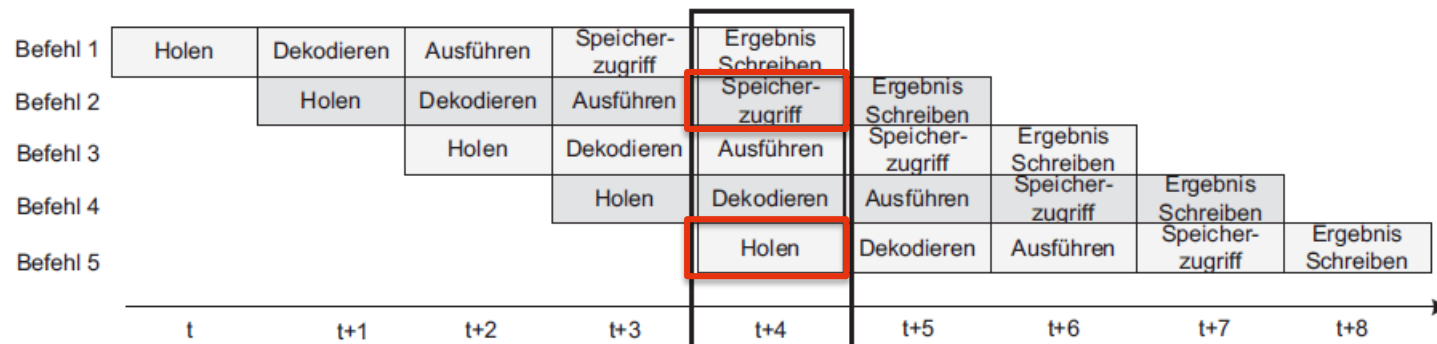


- Problem: Befehle **hängen** häufig **voneinander ab**, bzw. **stehen im Konflikt zueinander** → **Fließband-Konflikte** (Pipeline-Hazards)
  - **Daten-Konflikte** (Data Hazards):
    - Befehl schreibt Ergebnis in **Register x**
    - Nächster Befehl benötigt in der Pipeline den Inhalt dieses **Registers x** zum Weiterrechnen
    - Zweiter Befehl kann erst dann mit Schritt „Ausführen“ (Execute) fortsetzen, wenn der erste Befehl sein Ergebnis in dieses Register geschrieben hat (Ergebnis Schreiben - Write Back)
    - → zweite Befehl muss in der Pipeline **mindestens einen Takt** verzögert werden
    - Alle nachfolgenden Befehle werden ebenfalls verzögert.

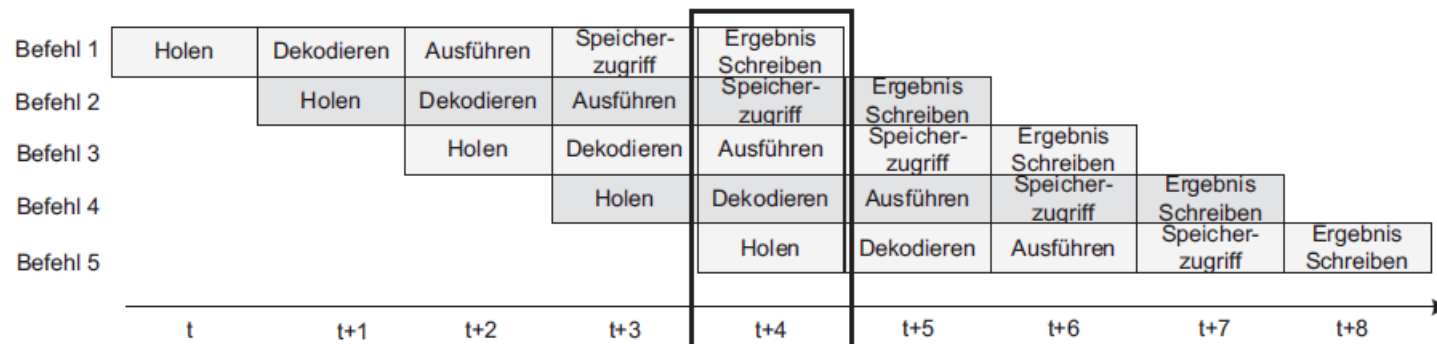




- Problem: Befehle **hängen** häufig **voneinander ab**, bzw. **stehen im Konflikt zueinander** → **Fließband-Konflikte** (Pipeline-Hazards)
  - **Struktur-Konflikte** (Structural Hazards):
    - Zwei Befehle in der Pipeline **brauchen gleichzeitig dasselbe Betriebsmittel** (z.B. Hauptspeicher), das aber nur einem Befehl exklusiv zur Verfügung stehen kann.
    - Wird ein Befehl aus **Hauptspeicher geladen** (Holen - Fetch), gleichzeitig versucht ein anderer Befehl in den **Hauptspeicher zu schreiben** (Speicherzugriff - Memory-Access).
    - Diese Zugriffskonflikte können z. B. durch **Verzögern des zweiten Befehls und der nachfolgenden Befehle** in der Pipeline aufgelöst werden.



- Problem: Befehle **hängen** häufig **voneinander ab**, bzw. **stehen im Konflikt zueinander** → **Fließband-Konflikte** (Pipeline-Hazards)
  - **Steuer-Konflikte** (Control Hazards):
    - Verzweigungen/Sprunganweisungen: hinter diesem Befehl stehenden Befehle müssen eventuell gelöscht werden, da jetzt angesprungenen Befehle ausgeführt werden sollen
    - Bei einfachen (unbedingten) Sprüngen: Dekodier-Einheit kann Löschung vermeiden
    - Bei bedingten Sprüngen hängt Sprungziel von einer vorab durchzuführenden Berechnung ab → moderne Prozessoren enthalten Einheit zur Vorhersage des Sprungziels (Branch Prediction) und laden richtige Befehle nach



# **PROGRAMMIERUNG - MASCHINENSPRACHE UND ASSEMBLER**

# Maschinensprache und Assembler

- **Programm**

Ein *Programm* ist eine Vorschrift, nach der vorgegebene Daten für die Lösung einer Aufgabenstellung verarbeitet werden. Es besteht aus einer **Folge von Befehlen**, die dem Rechenautomaten als **binäre** Information (Bitmuster) übergeben werden müssen.

- **Maschinensprache**

Die Darstellung von Befehlen, die **an eine Rechenmaschine angepasst** und für diese verständlich sind, heißt *Maschinensprache* oder *Maschinencode*.

- **Problematik der Maschinensprache**

Für Menschen als Benutzer sind Programme in Maschinensprache **schwer lesbar**.

- **Lösung**

Bequemere Befehlsdarstellung (*Maschinenorientierte* oder **Assembler-Sprache**) oder problemnahe Anweisungen (*Problemorientierte Sprache*). In beiden Fällen erfolgt mittels eines speziellen Programms (**Assembler** bzw. *Compiler*) eine **Übersetzung in Maschinensprache**.

## Maschinensprache

- **Charakterisierung**
  - **Elementaroperationen** mit bestimmtem Funktionsumfang (Erinnerung: **Befehlssatz**)
    - Datentransfer
    - Programmablaufsteuerung
    - Arithmetische und logische Operationen
    - Schiebe-Befehle
    - Unterbrechungsverarbeitung (Sprung)
- **Binärdarstellung der Befehle**
- Regelmäßiger Aufbau der einzelnen Befehle
- **Befehlsformate:**
  - 1-Adress-Befehl
  - 2-Adress-Befehl
  - 3-Adress-Befehl

## Maschinensprache

### 3-Adress-Befehl

- Prinzip:

F	Adr. a	Adr. b	Adr. c
Funktionsteil (Operationsteil, OP-Code)	Adressteil (bezeichnet Speicherzellen, die die Operanden und das Ergebnis enthalten)		

- Beispiel:

Operationsteil	Adressteil		
	Adresse 1	Adresse 2	Adresse 3
ADD	[100]	[104]	[110]
was?	wohin?	woher?	woher?

- Problematik des 3-Adress-Befehls:**

Die begrenzte Wortlänge heutiger Rechenanlagen **reicht meist für diese Befehlsdarstellung nicht aus**, weshalb 2- oder 1-Adressbefehle verwendet werden.

# Maschinensprache und Assembler

## Maschinensprache

### ■ 2-Adress-Befehl

- Prinzip:

F	Adr. b	Adr. c
Funktionsteil (Operationsteil, OP-Code)	Adressteil (bezeichnet Speicherzellen, die die Operanden enthalten)	

Bezüglich des Ergebnisses a  
gibt es eine implizite  
Vereinbarung.

- Beispiel:

Operationsteil	Adressteil	
	Adresse 1	Adresse 2
ADD	[100]	[104]
was?	woher? wohin?	woher?

# Maschinensprache und Assembler

## Maschinensprache

### 1-Adress-Befehl

- Prinzip:

F	Adr. b
Funktionsteil (Operationsteil, OP-Code)	Adressteil

Die Operation wird in mehrere Teiloperationen zerlegt; dies setzt einen zentralen Speicherplatz voraus, den **Akkumulator**.

- Beispiel:

Operationsteil	Adressteil
	Adresse 1
ADD	[110]
was?	woher?

Implizit: Akkumulator  
1. Operand und Ergebnis

- Häufig gibt es noch einen **Modifikationsteil** (Mod), der angibt, ob gewisse **Varianten der Operation** bzw. **Veränderungen der Adressierung** vorgenommen werden.



## Maschinensprache

### Ausschnitt einer hypothetischen Maschinensprache

OP-Code		Bedeutung der Operation	Mnemo. Bez.
dual	hex.		
0000	0	Halt, Ende der Programmbearbeitung	HLT
0001	1	Lade Operand in den Akkumulator	LAD
0010	2	Speichere Akku-Inhalt unter der angegebenen Adresse	SPI
0011	3	Addiere Operand zum Akkumulator	ADD
0100	4	Subtrahiere Operand vom Akkumulator	SUB
0101	5	Multipliziere Operand zum Akkumulator	MUL
...	...	...	...

### Beispiel: "Addiere 157 zum Akkumulator"

- Aufbau einer Speicherzelle

F	Mod	Adr. b
---	-----	--------

- Speicherwort: 0011 0100 1001 1101 ( **34 9D**<sub>16</sub> oder 13469<sub>10</sub> )

0011: Addiere Operand zum Akkumulator

0100: Operand ist Konstante ("Sofort-Operand")

10011101 157<sub>10</sub>

# Maschinensprache und Assembler

## Assembler

- **Nachteile der Maschinensprache:**
  - Binäre Darstellung der Operationen und Operanden
  - Verwendung fester Adressen
  - unkommentiert
- **Charakterisierung von Assembler:**
  - Mnemotechnischer Bezeichnung für Operationen
  - Verschiedene Zahlensysteme
  - Marken (als Ziel eines Sprungbefehls)
  - Kommentare
  - Makros (ein Befehl für eine Folge von anderen Befehlen)
- **Beispiel:** „Addiere 157 zum Akkumulator“

Maschinensprache: 0011 0100 1001 1101

Assembler: ADD, #157



# Maschinensprache und Assembler

## Beispielassembler (16-Bit-Maschine)

- Zwei **Allzweckregister**: R1, R2, ein **Flagregister** (FL) und einen **Befehlszähler** (IP=Instruction Pointer)
- In den folgenden Befehlen muss **Reg** ein **Register** (z.B. R2) sein. **Quelle** kann ein **Register**, eine **Speicheradresse** (z.B. [0x12AB] oder [R1]) oder eine **Konstante** (z.B. 12, 014, 0x0C, 0000 1100b) sein
- **Arithmetische Befehle**
  - ADD Reg, Quelle ; Reg = Reg + Quelle
  - SUB Reg, Quelle ; Reg = Reg - Quelle
  - DIV Reg, Quelle ; Reg = Reg / Quelle
  - MUL Reg, Quelle ; Reg = Reg \* Quelle
  - MOD Reg, Quelle ; Reg = Reg MOD Quelle (Modulo)
  - INC Reg ; Reg = Reg + 1
  - DEC Reg ; Reg = Reg - 1
  - CMP Reg, Quelle ; Vergleich Inhalt von Reg mit Inhalt von Quelle; keine Veränderung an Reg oder Quelle; Flagregister FL wird gemäß Vergleichsergebnis gesetzt  
; (FL kann anschließend von Sprungbefehlen ausgewertet werden)
- **Logische Befehle**
  - AND Reg, Quelle ; Reg = Reg AND Quelle
  - OR Reg, Quelle ; Reg = Reg OR Quelle
  - NOT Reg ; Reg = NOT Reg
  - XOR Reg, Quelle ; Reg = Reg EXOR Quelle



# Maschinensprache und Assembler

## Beispielassembler

- Sprungbefehle (**Ziel** ist eine **Konstante** = Speicheradresse)
  - JG Ziel ; Jump Greater
  - JGE Ziel ; Jump Greater or Equal
  - JL Ziel ; Jump Lower
  - JLE Ziel ; Jump Lower or Equal
  - JZ Ziel ; Jump Zero
  - JNZ Ziel ; Jump Not Zero
  - JMP Ziel ; Jump (unbedingter Sprung)
- Schiebebefehle (**Reg** muss ein **Register** sein)
  - SHL Reg ; Shift Left
  - SHR Reg ; Shift Right
  - ROL Reg ; Rotate Left
  - ROR Reg ; Rotate Right
- Laden/Speicher-Befehl (**Reg** muss ein **Register** sein; **Quelle** kann ein **Register**, eine **Speicheradresse** oder eine **Konstante** sein; **Mem** muss eine **Speicheradresse** sein)
  - MOV Reg, Quelle ; Lade Register mit Inhalt von Quelle
  - MOV Mem, Reg ; Speichere Inhalt des Registers in Speicher



# *Maschinensprache und Assembler*

## *Beispielassembler*

# Übung Assembler

- **Literatur**
  - GUMM, Heinz Peter; SOMMER, Manfred:  
Einführung in die Informatik. München; Wien:  
Oldenbourg Verlag, 10. Auflage
    - Kapitel 5

A solid yellow vertical bar with rounded top corners is positioned on the left side of the slide.

# Vielen Dank!