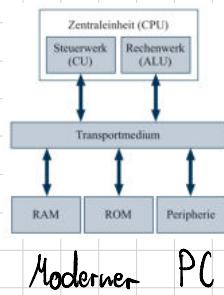


# Einführung

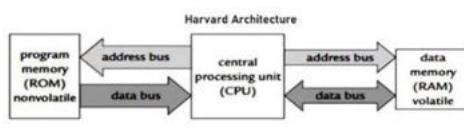
## Architektur

### Von-Neumann-Architektur



Moderner PC

### Harvard-Architektur



→ Trennung

↳ Üblich für Geräte, deren Maschinenbefehle nicht überschrieben werden müssen  
- Wassermaschine, Router

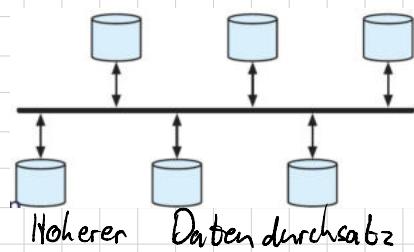
⇒ Vorteil anhand von höheren Übertragungsraten

### Kommunikation (BUS-System)

<https://de.wikipedia.org/wiki/Systembus>

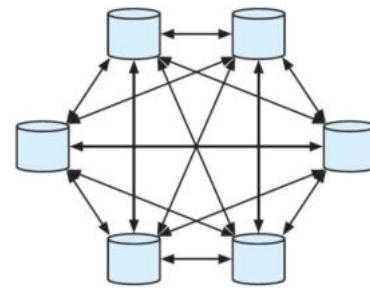
Mit Speicher, RAM, ROM, I/O

#### Indirekte Kommunikation



beides wird angewandt

#### Direkte Kommunikation



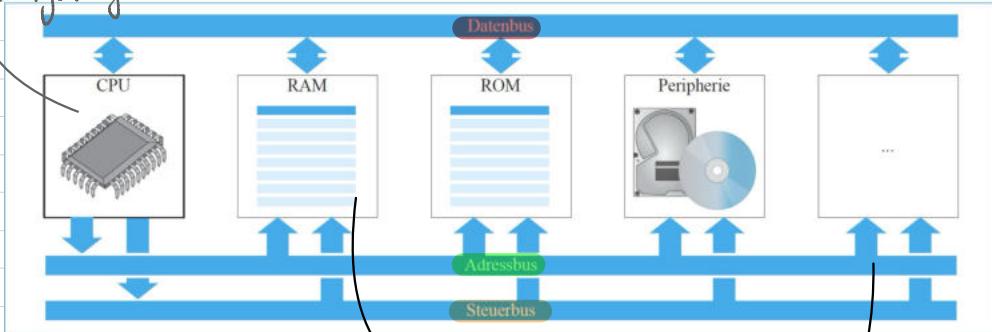
flexible Erweiterbarkeit

Für moderne Rechner:

CPU kann als Diktator gesehen werden

#### Datenbus:

Datenaustausch zwischen CPU und untergeordneten Modulen (Bidirektional)



#### Adressbus:

Zuweisung einer Speicher- oder I/O-Adresse. (Unidirektional)

Jede Speicherzelle ist einzeln  
zuordnbar.  
Jedoch sind sie nur über mehrere  
Geräte einzigartig.

#### Steuerbus:

Steuerung und Synchronisation des gesamten Systems. → Leitung für Memory (R/W) / I/O (R/W)

- Reset Leitung
- Buszugriffsbewertung

Jeder Teilnehmer sieht, welche Adresse von der CPU bekannt gegeben wird.  
Steuerbus regelt, welches Gerät den Speicher aktivieren soll.

LDA : (Adressbus) Speicheradresse freigeben → (Steuerbus) We

SDA:

# TODO

Lukas Mensch

[2] Back Panel Unit Sockel

[1] (lat. unus eins) in eine Richtung

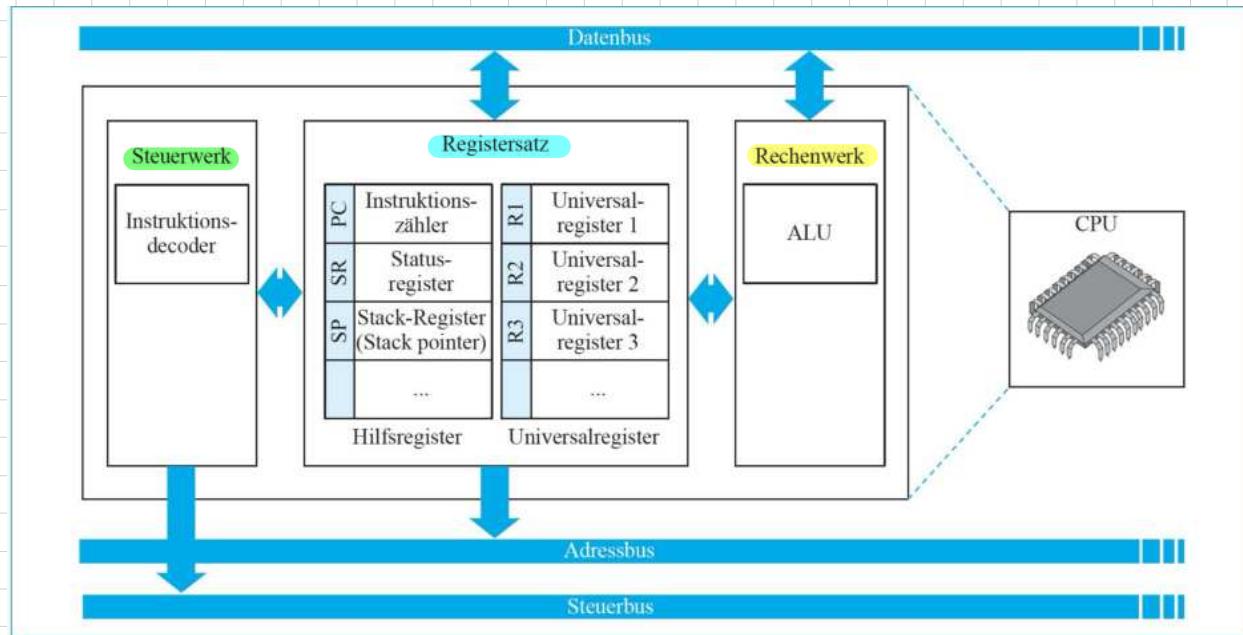
<https://de.wikipedia.org/wiki/Unidirektional>

Beispiel der BUS-Aktivitäten mit den Maschinenbefehl:

LDA (14) // Operand laden (x)

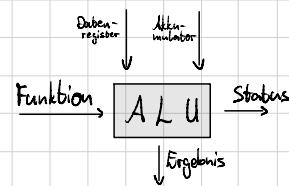
1. CPU fragt über den Adress-BUS die Adresse 14 an
2. RAM aktiviert den Chip mit der Speicherzelle unter Adresse 14
3. RAM stellt die Adresse zur Verfügung
4. CPU übernimmt den Inhalt aus dem Datenbus und schreibt diesen in den Spezialregister (Akkumulator)
5. CPU deaktiviert das Memory Read Signal sowie die Adresse
6. RAM deaktiviert sein Signal auf dem Datenbus

### Struktur eines typischen Mikroprozessors



Steuerwerk kann als Kommandobrücke einer CPU gesehen werden, die als Aufgabe hat, den Kontroll- und Datenfluss zu überwachen.  
Maschinenbefehle werden hier mit Hilfe des Instruction decoders und den Befehlsatz (Bsp. x86) in eine Reihe von Steuersignalen übersetzt

In den **Rechenwerken** entsteht mit Hilfe der ALU die eigentliche Verarbeitung der Daten.



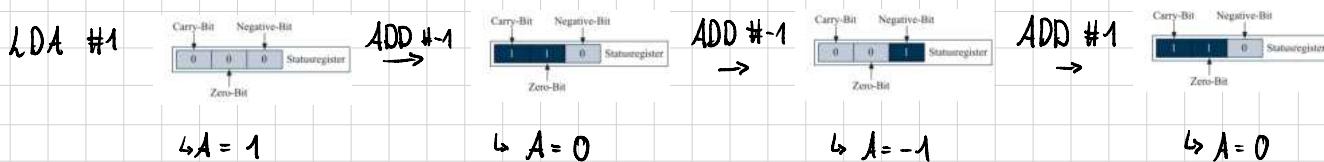
Der **Registersatz** einer CPU lässt sich grob in Universal- und Hilfsregister einteilen.

- Universal register können mit beliebigen Werten befüllt werden und währe aus theoretischer Sicht völlig ausreichend.
- Hilfsregister werden bestimmte Rollen zugewiesen:
  - Instruktionszähler (program counter (PC)):
 

Enthält die Speicheradressen des nächsten auszuführenden Befehls. Um den nächsten Befehl zu laden werden die Adressen um eins erhöht oder überschrieben.
  - Statusregister (status register (SR))

Dieses Register wird vom Rechenwerk direkt beschrieben.

Es enthält Statusbits, die weitere Infos über das Ergebnis der zuletzt durchgeführten Operation geben.



- **Carry-Bit (C):**  $1 \Rightarrow$  „Ergebnis hat einen Überlauf“
- **Zero-Bit (Z):**  $1 \Rightarrow$  „Ergebnis ist 0“
- **Negative-Bit (N):**  $1 \Rightarrow$  „Ergebnis ist negativ“

Wird auch zur Realisierung von bedingten Sprüngen eingesetzt, somit kann ein Sprung von den Statusbits abhängen

- Stackregister (stack pointer (SP)): verwaltet Unterprogrammaufrufe.
  - Ausstatt den Programzzähler direkt zu überschreiben wird zuerst die aktuelle Adresse vor dem Sprung gesichert.
  - Nachdem das Unterprogramm vollständig ausgeführt worden ist, erfolgt ein Rücksprung zu der gespeicherten Adresse
  - Verschachtelte (mehrere) Unterprogrammaufrufe werden auf einem Stack gespeichert

## - Befehlsregister (Instruction Register (IR)):

Beinhaltet (Verwaltet) den vom Speicherwerk eingegebenen aktuellen Befehl

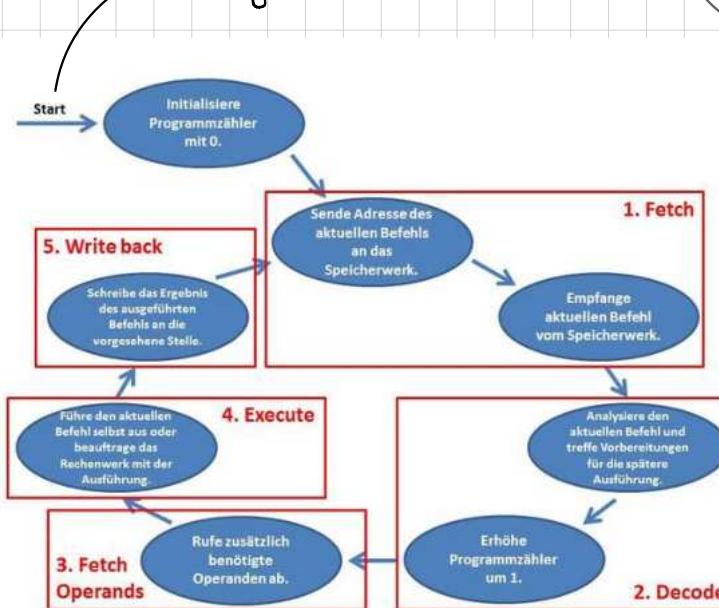
## - Akkumulator (ACC) / Datenregister:

Wert von vorherigen Befehl (Berechnungen) / Aktueller Operanden

## Funktionsweise eines Prozessors

- Bios: 1. Test der Hardware
- 2.

Reset signal #



## Koch Beispiel:

→ Fetch:

Lesen die zuvor notierte Anweisung aus dem Kochbuch aus  
Notiere die nächste Anweisung

↓  
Decode:

Interpretiere die Anweisung.

↓  
Fetch Operands:

Lege Zutat und Werkzeug bereit.

↓  
Execute:

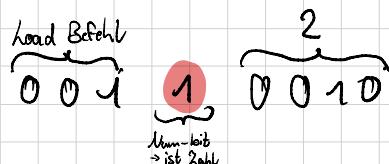
Bearbeitung der Zutat.

↓  
Write Back:

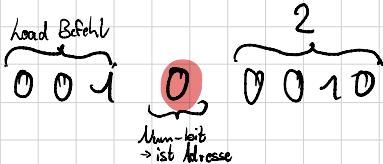
Zubereitete Zutat in ein Behältnis (Schüssel, Töpf, Teller, Schneidebrett, ...) geben.

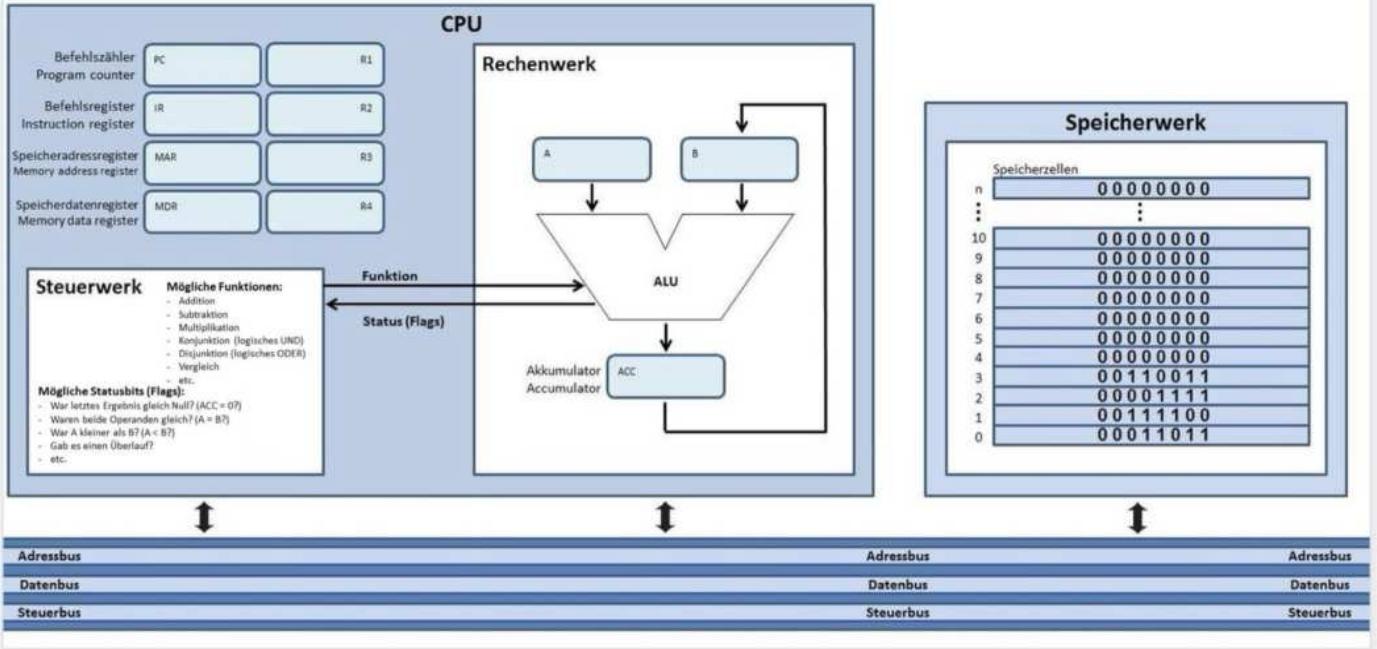
Num-Bit:

load #2 =>

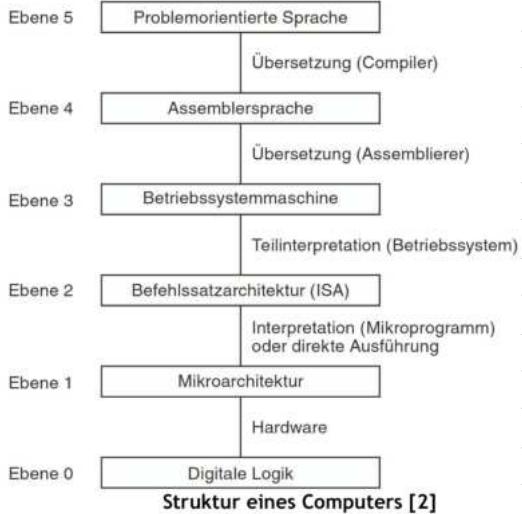


load 2 =>





## Konzepte/Begriffe

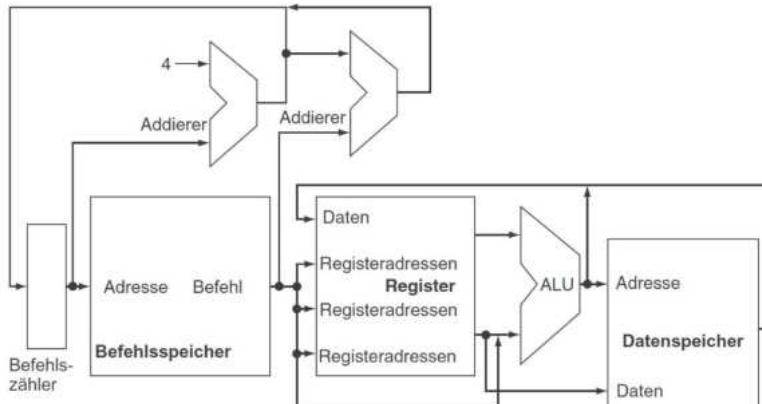


## Kodierung, ALU, Kalkulation

- Zahlendarstellung
- Codes
- Binärarithmetik
- Multiplikationshardware
  - Schaltnetze
  - Schaltwerke

## Rechen-, Steuer-, Schaltwerke/-netze

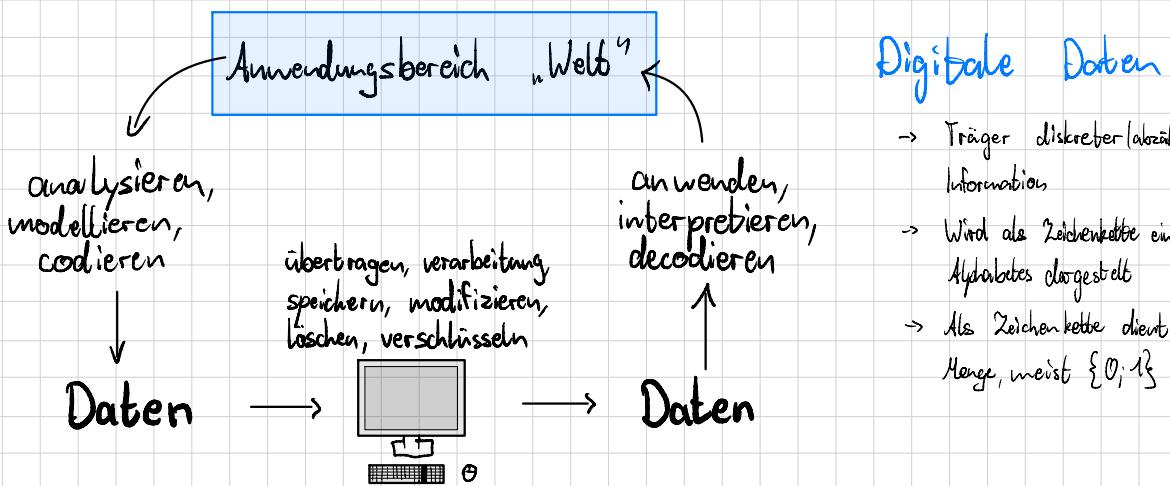
- Boolesche Algebra
- Grundlagen der Digitaltechnik
- Schaltungssynthese
- Minimierung
- Standardschaltnetze
- Schaltwerke
- Flipflop
- Endliche Automaten
- Schaltwerksynthese
- Standardschaltwerke



Eine abstrakte Implementierung eines Prozessors für MIPS-Befehlssatz mit den wichtigsten Funktionseinheiten und den wichtigsten Verbindungen der Funktionseinheiten untereinander.

# Kodierung

Analog	Digital
Werkzeug kann im best. Bereich beliebig verstellt werden	Werkzeug lässt sich stufenweise verstellen
Sensor gibt eine kontinuierliche Größe aus	Sensor gibt eine diskrete Größe aus
Das Signal ändert sich kontinuierlich mit der Zeit	Das Signal ändert sich stufenweise mit der Zeit



Um Daten codieren zu können, wird ein Zeichenvorrat (Alphabet) benötigt, welches eine endliche Menge aus unterschiedlichen Zeichen ist

Bsp.:  $\{0; 1; \dots; 8; 9\}$ ;  $\{a; b; \dots; x; y; z\}$ ;  $\{w; f\}$

Dies lassen gemäß einer Ordnungsrelation  $\leq$  anordnen:

Bsp.:  $\{0; 1; \dots; 8; 9\} \rightarrow 0 \leq 1 \leq 2 \leq \dots \leq 9$   
 $\{a; b; \dots; z\} \rightarrow a \leq b \leq \dots \leq z$

Grundbegriffe:

- $\Sigma \rightarrow$  Zeichenvorrat  $\oplus$
- $w = z_1 z_2 \dots z_n \rightarrow$  Zeichenwort, -kette, String
- $\lambda \rightarrow$  leeres Wort
- $n \rightarrow$  Wortlänge von  $w$ ,  $n = |w|, |\lambda| = 0$
- $\Sigma^n \rightarrow$  die Menge aller Wörter der Länge  $n$   
 $\Sigma^n = \{w \mid w = z_1 \dots z_n \in \Sigma\}$
- $\Sigma^+ \rightarrow$  die Menge aller Wörter, die aus  $\Sigma$  gebildet werden können  
 $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
- $\Sigma^* = \Sigma^+ \cup \Sigma^0$  Das leere Wort

Lukas Mensch

## → Konkatenation (Verkettung)

Bezeichnet eine Operation die ein Wort  $u$  mit einem anderen Wort  $v$  oder Zeichen  $z$  verbindet, sodass ein längeres Wort  $w$  entsteht

$$u \circ v = w \Rightarrow |w| \geq u \cup v$$

## Codierung

**Codierung** sind die Zeichen/Wörter über ein Zeichenverat auf Wörter über einem anderen Zeichenverat abbilden.

→ Abbildungen können variieren (Tabelle, Rechenvorschrift)

→ Zeichencodierungen  $c$  können auf eine Wortcodierung  $c^*$  erweitert werden

$$c^*(z_1 z_2 \dots z_n) := c(z_1) c(z_2) \dots c(z_n)$$

$$c: \Sigma_1 \rightarrow \Sigma_2$$

„ $c$  für das gilt, dass sich  $\Sigma_1$  auf  $\Sigma_2$  abbildet“

## Kodierungsklassen

Chiffrierung:

$$A = B = \{0; 1; \dots; 9\} \quad \text{jedoch: } A: 0 \leq 1 \leq \dots \leq 9$$

$$B: 5 \leq 6 \leq \dots \leq 3 \leq 0 \leq \dots \leq 4$$

$$c(0) = 5$$

Sei  $c: A \rightarrow B$



$$c(1) = 6$$

Neue Ordnungsrelation

:

$$c(42) = 97$$

$$c(8) = 4$$

Blockcodierung: ( $c: A \rightarrow B^n$ )

Sei  $c: \{0; 1; \dots; 8\} \rightarrow \{a, b, c\}^2$  - 3<sup>2</sup> Kombinationen

$$c(0) = aa$$

$$c(1) = ab$$

$$c(2) = ac$$

$$c(3) = ba$$

$$c(4) = bb$$

:

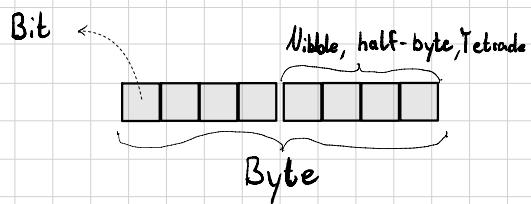
$$c(8) = cc$$

Weitere: • Binärcodierung ( $c: A \rightarrow \text{Bool}$ )

•  $n$ -Bit-Codierung ( $c: A \rightarrow \text{Bool}^n$ )

# Grundbegriffe bei Binärcodierung

- Bitstring → ein binäres Wort



# Zahlen

Die Darstellung von Zahlen kann stark variieren je nach:

- **Zahlbereich** ( $\mathbb{N}; \mathbb{Z}; \mathbb{Q}; \mathbb{R}; \mathbb{C}$ )
- **Genauigkeit**, Rundung (sfehler?)
- Realisierung von **arithmetischen Operationen**

Des Weiteren stellt auch der Code **Anforderungen**:

- Konvertierbarkeit in/aus Dezimalsystem
- einfache Arithmetik  $\rightarrow$  durch geringe Schaltungsaufwand hohe Rechengeschwindigkeit
- Anwendung von:
  - Fehlererkennbarkeit (große Hammingdistanz)
  - Zählwerks (kleine Hammingdistanz nötig)

## Codierung: Zahlen vs Ziffern

Ziffern sind die Bausteine von Zahlen

Beispiel: 123  $\rightarrow$  1, 2 und 3 sind Ziffern

von der Zahl 123 (<sup>Hunder-</sup>  
<sup>drei und</sup>  
<sup>zwanzig</sup>)

Eine Ziffer lässt sich mit Hilfe einer **Codierung** (ASCII, UTF-8,...) erstellen:

$$c_{\text{ASCII}}('1') = 49; \quad c_{\text{ASCII}}('4') = 52; \quad c_{\text{ASCII}}('7') = 55$$

↗ Ziffer  
 ↗ ASCII-Codierung

Somit kann man aus mehreren Ziffern eine **Ziffernfolge (Wort)** erstellen:

$$c_{\text{ASCII}}^*("4711") = 52 \bullet 55 \bullet 49 \bullet 49$$

↗ Natürliche Fol-  
sezung von...  
 ↗ Konkatenation

=> Berechnung mit ASCII-Darstellung ist **unpraktisch** und **verschwendet unnötig Platz**

## Codierung bei Zahlen mit fester Länge

Gegeben sei:

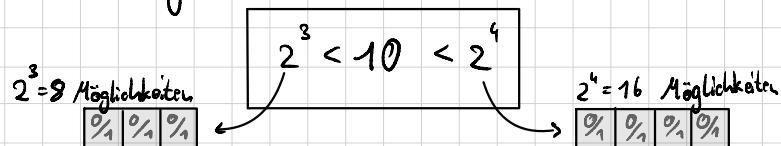
- Zahl  $z \in \mathbb{Z}$
  - Binärwort  $b = 2^n \quad n \in \mathbb{N}^*$
- länge der  
Binärzahl

Gesucht ist eine Codierung  $c$ , welche die Zahl  $z$  auf ein binäres Codewort  $c(z)$  abbildet.

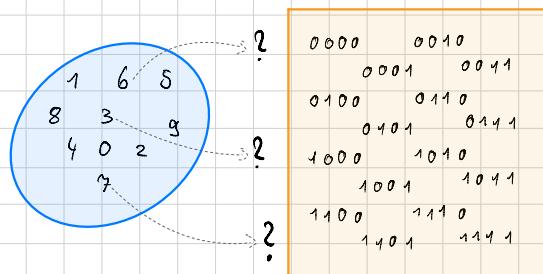
Überlegung der Binärdarstellung für die Ziffern 0 bis 9

↪ Für diese Darstellung wird pro Ziffer eine Darstellung benötigt

⇒ somit benötigt man für alle 10 Ziffern 10 Codewörter in Binär



Es stellt sich heraus, dass ein Binärbreite von  $2^4$  benötigt wird. Somit stellt sich jedoch die Frage, welchen Codewort man welcher Ziffer zuordnet?



Durch schriftliches Hochzählen einer Binärzahl nach den Regeln aus dem Dezimalsystem lässt sich eine Art der Codierung ermitteln:

Dezimal	Binär
0	0
+ 1	1
-----	1
1	1
+ 1	1
-----	10
2	10
+ 1	11
-----	11
3	11
+ 1	111
-----	1000
4	1000
⋮	⋮
⋮	⋮
8	1000
+ 1	1010
-----	1010

⇒ Diese Codierung nennt sich BCD-Code  
(binary-coded decimal)

Sowie gibt es auch weitere Codierungen

BCD-Code	Exzess-3-Code	Aiken-Code	2-aus-5-Code
0	0000	0000	11000
1	0001	0100	00011
2	0010	0101	00101
3	0011	0110	00110
4	0100	0111	01001
5	0101	1000	01010
6	0110	1001	01100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

# Zahlensysteme

Um mit Zahlen rechnen zu können, werden diese ins Binärsystem überführt um dann im Binärsystem zu rechnen

Basis	Bezeichnung	Ziffernbereich
2	binär, dual	$a_i \in \{0; 1\}$
8	oktal	$a_i \in \{0; 1; \dots; 8\}$
10	dezimal	$a_i \in \{0; 1; \dots; 9\}$
16	hexadezimal	$a_i \in \{0; 1; \dots; 9; A; B; \dots; F\}$

Schreibweise einer Zahl aus einem Zahlensystem:

$$(a_1 \cdot a_2 \cdot \dots \cdot a_n)_b$$

bzw:  $(a_1 a_2 a_3 \dots a_n)_b$  Bezeichnung für Basis

## Umrechnung in verschiedenen Zahlensystemen:

Mit Hilfe der Definition eines Stellenwertsystems

$$(a_n a_{n-1} \dots a_0)_b = (a_n \cdot b_x^n + a_{n-1} \cdot b_x^{n-1} + \dots + a_0 \cdot b_x^0)_b = \sum_{-m \leq i \leq n} a_i \cdot b^i$$

Beispiel:

$$(1101)_2 = (\underbrace{1 \cdot 2^3}_{(10)_8} + \underbrace{1 \cdot 2^2}_{(4)_8} + \underbrace{0 \cdot 2^1}_{(0)_8} + \underbrace{1 \cdot 2^0}_{(1)_8})_8 = (15)_8 = (\underbrace{1 \cdot 8^1}_{(8)_10} + \underbrace{5 \cdot 8^0}_{(5)_10})_{10} = (13)_{10}$$

$$(42)_{10} = \left( (\underbrace{100}_{} \cdot \underbrace{(1010)}_{(101000)_2})^1 + (10) \cdot (1010)^0 \right)_2 = (101010)_2$$

Das Umrechnen von der Basis 10 zu einer anderen Basis erweist sich als unständlich, da hier unüblich mit einem nicht Dezimalsystem gerechnet werden muss.

$$\begin{aligned} \left(\frac{1}{2}\right)_{\frac{1}{2}} &= \left(E_n \cdot \left(\frac{1}{2}\right)^n + E_{n-1} \cdot \left(\frac{1}{2}\right)^{n-1} + \dots + E_0\right)_b = \left(\frac{E_n}{2^n} + \frac{E_{n-1}}{2^{n-1}} + \dots + E_0\right)_b \\ \left(\frac{1}{2} \cdot 0 \cdot \frac{1}{2}\right)_{\frac{1}{2}} &= \left(\frac{1}{4} + \frac{1}{2}\right)_{10} = \left(\frac{3}{2}\right)_{10} \quad \left(\frac{3}{2} : \frac{1}{2}\right)_{10} = \left(\frac{4}{3}\right)_{10} = \left(1 + \frac{\frac{1}{2} \cdot 1}{\frac{3}{2} - 1}\right)_{10} = \left(1 + \frac{1}{\frac{1}{2}}\right)_{10} \end{aligned}$$



$\Rightarrow$  Vereinfachung

$$(\tilde{E})_{b_x} = (E_n \cdot b_x^n + E_{n-1} \cdot b_x^{n-1} + \dots + E_0 \cdot b_x^0)_{b_y} = (a_n \cdot a_{n-1} \cdot \dots \cdot a_0)_{b_y} = (a)_{b_y} = (a_n b_y^n + a_{n-1} b_y^{n-1} + \dots + a_0 b_y^0)_{b_x}$$

Um die letzte Ziffer der Codierung  $b_y$  von der Zahl abzusplittern, wird durch  $b_y$  geteilt

$$\begin{aligned} \left( \frac{\tilde{E}}{b_y} \right)_{b_x} &= \left( \frac{a_n b_y^n + a_{n-1} b_y^{n-1} + \dots + a_0 b_y^0}{b_y} \right)_{b_x} \\ &= \left( \frac{a_n b_y^{n-1} + a_{n-1} b_y^{n-2} + \dots + a_1 b_y^1}{b_y} + \frac{a_0}{b_y} \right)_{b_x} \end{aligned}$$

Dadurch lassen ein  $b_y$  bei allen Summanden, bis auf den Summanden  $a_0$ , streichen.

$$= \left( (a_n \cdot b_y^{n-1} + a_{n-1} \cdot b_y^{n-2} + \dots + a_1) + \frac{a_0}{b_y} \right)_{b_x}$$

Man erhält einen Bruch der die letzte Ziffer der Zahl  $(a)_{b_y}$  enthält.

$\rightarrow a_0$  ist schließlich auch der ganzzahlige Rest der Division  $\left( \frac{\tilde{E}}{b_y} \right)_{b_x}$ ,

$$\text{da } \frac{a \cdot n + r}{n} = a + \frac{r}{n} \quad \begin{array}{l} r \text{ ist der} \\ \text{Teil, der sich nicht} \\ \text{durch } n \text{ teilen kann} \end{array} \quad a \in \mathbb{R}, n \in \mathbb{N}, r \in \{x \mid x \text{ ist teilerfremd mit } n\}$$

Um die nächsten Ziffern zu ermitteln, teilt man Quotienten erneut durch  $b_y$ , um ein weiteres  $a$  abzusplittern.

Somit ergibt sich folgender Algorithmus:

$$\left. \begin{aligned} (\tilde{E} : b_y)_{b_x} &= \left( \tilde{E} + \frac{a_0}{b_y} \right)_{b_x} = \left( \tilde{E} \text{ Rest } a_0 \right)_{b_x} \\ (\tilde{E} : b_y)_{b_x} &= \left( \tilde{E} + \frac{a_1}{b_y} \right)_{b_x} = \left( \tilde{E} \text{ Rest } a_1 \right)_{b_x} \\ &\vdots \\ (\tilde{E} : b_y)_{b_x} &= \left( 0 + \frac{a_n}{b_y} \right)_{b_x} = \left( 0 \text{ Rest } a_n \right)_{b_x} \end{aligned} \right\} (a_n \cdot \dots \cdot a_1 \cdot a_0)_{b_y}$$

Des Weiteren ist es auch hilfreich, besonders für Zahlensysteme der Basis 2, das Zahlensystem ins Binärsystem umzurechnen, da durch Ziffern abgelesen werden können:

$$\begin{aligned}
 (4711)_8 &= (\underbrace{100}_{(4)_8} \quad \underbrace{111}_{(7)_8} \quad \underbrace{001}_{(1)_8} \quad \underbrace{001}_{(1)_8})_2 \\
 &= (1001 \cdot (2^3)_{10} + 1100 \cdot (2^4)_{10} + 1001)_2 \\
 &= (9)_{16} \cdot (16^2)_{10} + (C)_{16} \cdot (16^1)_{10} + (9)_{16} \\
 &= (9 \cdot 10^2 + C \cdot 10^1 + 9 \cdot 10^0)_{16} \\
 &= (9C9)_{16}
 \end{aligned}$$

$$\begin{aligned}
 (4711)_8 &= (100) \cdot (1000) + (111) \cdot (100^2) + (4) \cdot (100^1) + (1) \cdot (100^0) \\
 &= (100) \cdot (10^3) + (111) \cdot (10^2) + (4) \cdot (10^1) + 1
 \end{aligned}$$

#TODO  
Verallgemeinerung

Noch eine Methode wäre das Horner-Schema:

$$723 = 1 \cdot 10^7 + 2 \cdot 10^4 + 2 \cdot 10^0$$

$$\begin{aligned}
 (a_n \cdot a_{n-1} \cdots \cdot a_0)_{b_x} &= (a_n \cdot b_x^n + a_{n-1} \cdot b_x^{n-1} + \dots + a_0 \cdot b_x^0)_{by} \\
 &= \left[ (a_n \cdot b_x^{n-1} + a_{n-1} \cdot b_x^{n-2} + \dots + a_1) b_x + a_0 \right]_{by} \\
 &= \left[ \left[ (a_n \cdot b_x^{n-2} + a_{n-1} \cdot b_x^{n-3} + \dots + a_2) b_x + a_1 \right] b_x + a_0 \right]_{by} \\
 &= \left[ \left[ \left[ (a_n \cdot b_x^{n-3} + a_{n-1} \cdot b_x^{n-2} + \dots + a_3) b_x + a_2 \right] b_x + a_1 \right] b_x + a_0 \right]_{by} \\
 &= \left[ \left[ \left[ \cdots (a_n \cdot b_x + a_{n-1}) b_x + a_{n-2} \right] b_x + \dots + a_2 \right] b_x + a_1 \right] b_x + a_0
 \end{aligned}$$

Interpretation:

Anfangen mit der ersten Ziffer wird dazu die vorherige Basis  $by$  davon multipliziert und mit der nächsten Ziffer addiert.  
 (... anschließend wieder mit der Basis multiplizieren und mit der nächsten Ziffer addieren)

## Rechnen im Binärsystem

Rechnen in anderen Zahlensystemen können analog zum Dezimalsystem durchgeführt werden.

### Multiplikation/Addition:

$$\begin{array}{r} 9 - 11 \\ \hline + 9 \\ \hline 99 \end{array}$$

$\Leftrightarrow$

$$\begin{array}{r} 1001 \cdot 1011 \\ \hline 1001 \\ 0000 \\ 1001 \\ + 1001 \\ \hline 1100011 \end{array}$$

### Division/Subtraktion:

$$\begin{array}{r} \overbrace{299}^{\text{13}} : 13 = 23 \\ - 26 \\ \hline 39 \\ - 39 \\ \hline 0 \end{array}$$

$$\begin{array}{r} \overbrace{10010101011}^{\text{1101}} : 1101 = 10111 \\ - 1101 \\ \hline 1011 \\ - 1011 \\ \hline 0 \\ - 1101 \\ \hline 1101 \\ - 1101 \\ \hline 0 \end{array}$$

Bei Multiplikation / Division mit 2er Potenz kann eine Bitverschiebung vorgenommen werden.  
(Wie im Dezimalsys. mit einer 10er Potenz.)

Big-Endian vs. little-Endian

0x 12345678

Adresse	Data		Adresse	Data	LSB (least significant byte)
00	0x12	MSB (most significant byte)	00	0x78	an der niedrig Adresse
01	0x34	an der niedrigsten Adresse	01	0x56	→ starts little
02	0x56	→ starts big	02	0x34	
03	0x78		03	0x12	

→ wird von der Speicherordnung d. Computers festgelegt.

→ muss bei vernetzten Computers u. Debuggen berücksichtigt werden

## Ganze Zahlen mit Vorzeichen

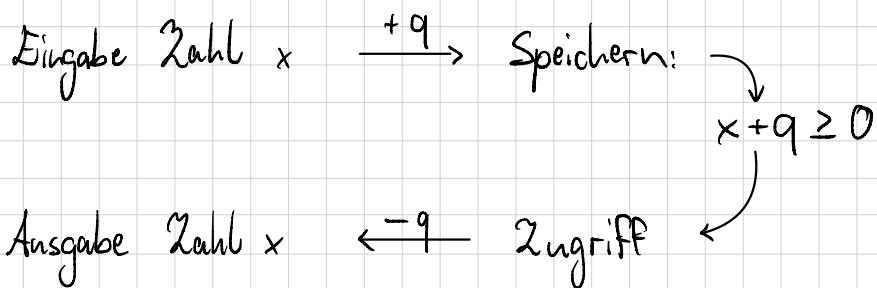
### Exzess-q-Darstellung

- Vorzeichen + Betrag
- Exzess-q-Darstellung
- Komplement-Darstellung

Ein q (meist  $q = 2^m$  bei  $m+1$  verfügbaren Stellen)

↳ erzeugt den Zahlenbereich  $[-2^{m-1} \dots 2^{m-1}]$

welches vor Eingabe und Ausgabe Subtrahiert oder Addiert wird damit im Speicherer eine positive Zahl gespeichert wird



Beispiel:  $(-5)_{10}$  als  $(x)_{2ex}$  mit 4 Bits,  $q = 2^3 = 8$

$$(-5)_{10} \longrightarrow \underbrace{(-0101)_2}_{-5} + \underbrace{(1000)_2}_8 = \underbrace{(0011)_2}_3$$

$$\downarrow$$

$$(-5)_{10} \longleftarrow (-0101) = (0011)_2 - (1000)_2$$

Gespeicherter Wert | Dezimale Interpretation

Bin	Hex	Am V2	Betrag u. V2	Einerkomp.	Zweierkomp.
0000	0	0	0	0	0
0001	1	1	1	1	1
0010	2	2	2	2	2
0011	3	3	3	3	3
0100	4	4	4	4	4
0101	5	5	5	5	5
0110	6	6	6	6	6
0111	7	7	7	7	7
1000	8	8	-0	-7	-8
1001	9	9	-1	-6	-7
1010	A	10	-2	-5	-6
1011	B	11	-3	-4	-5
1100	C	12	-4	-3	-4
1101	D	13	-5	-2	-3
1110	E	14	-6	-1	-2
1111	F	15	-7	0	-1

### V2-Betrag Darstellung

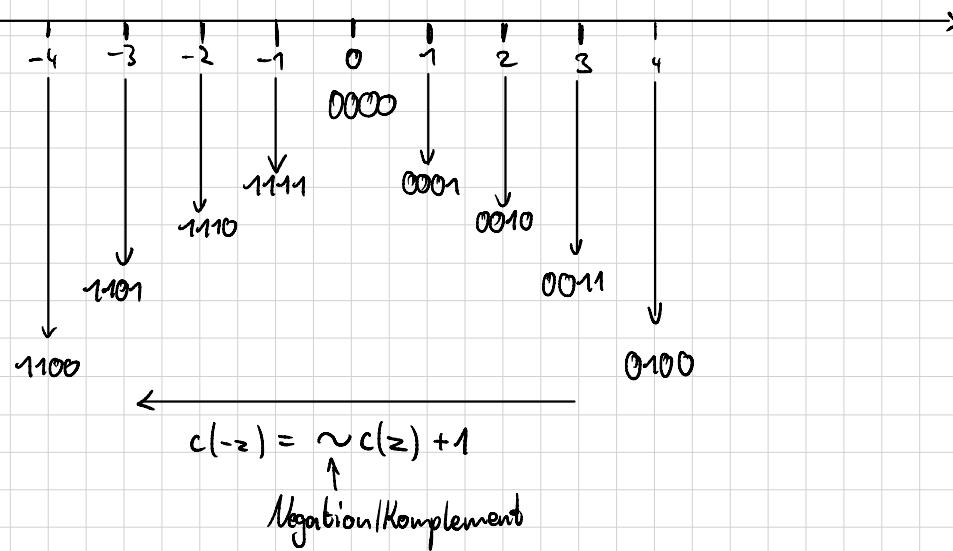
↳ Null gibt es zweimal

(0000, 1000)

↳ macht rechnen komplizierter

mit  
+1

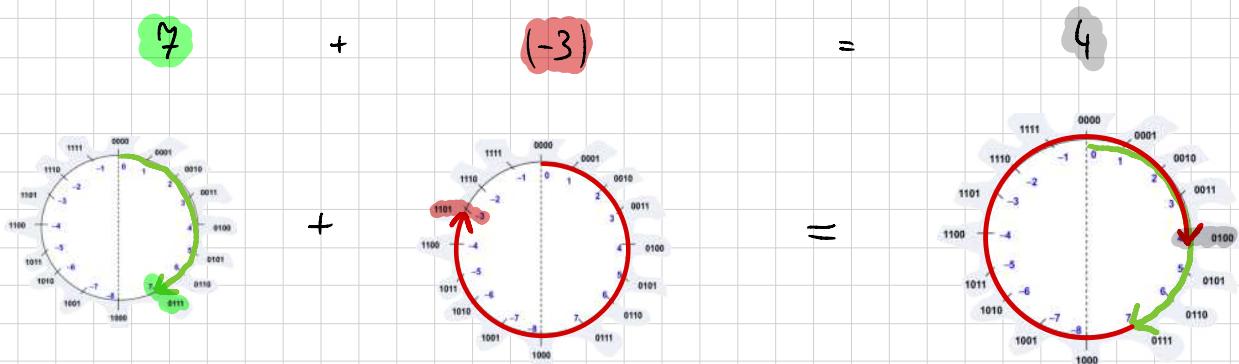
## Zweierkomplement



## Berechnungen im Zweierkomplement:

$$\begin{array}{r}
 7_{10} \\
 + -3_{10} \\
 \hline
 4_{10}
 \end{array}
 \quad
 \begin{array}{r}
 0111_2 \\
 + 1101_2 \\
 \hline
 1010_2
 \end{array}
 \quad
 \text{Da Überlauf}$$

Verständlicher kann die Subtraktion in einer Ringdarstellung demonstriert werden:

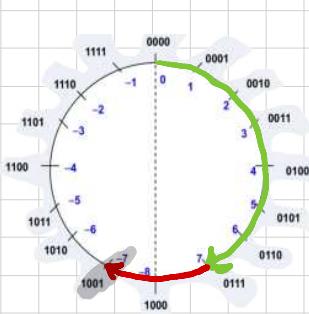


**Wichtig!**

Dadurch entsteht auch die Gefahr eines Überlaufs

Beispiel:

"7 + 2" wird zu "-7", da:



=> Die Berechnung mit Multiplikation und Division kann auf die Addition zurückgeführt werden, jedoch gibt es auch bessere Methoden.

### Berechnung in der Praxis

Zero-Flag(ZF) : 1, wenn Ergebnis 0 ist

Overflow-Flag(OF) : 1, wenn es zu einem Overflow kommt

Carry-Flag(CF) : 1, wenn Speicher überschritten wird

2 8 9 2 8

Betrachte wird folgende Berechnung  $\frac{+ 16385}{45313}$  in der ...

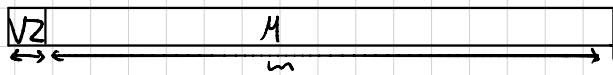
.., 16 Bit Arithmetik:

$$\begin{array}{r} 0111 \quad 0000 \quad 1111 \quad 1111 \\ + 0100 \quad 0000 \quad 0000 \quad 0001 \\ \hline 1011 \quad 0001 \quad 0000 \quad 0000 = (-2022)_10 \end{array} \quad \begin{array}{l} CF=0 \\ ZF=0 \\ OF=1 \end{array}$$

.., 8 Bit Arithmetik:

$$\begin{array}{r} 0111 \quad 0000 \quad 1111 \quad 1111 \\ + 0100 \quad 0000 \quad 0000 \quad 0001 \\ \hline 1011 \quad 0001 \quad 0000 \quad 0000 \end{array} \quad \begin{array}{l} CF=0 \quad ZF=0 \quad OF=1 \\ CF=1 \quad ZF=1 \quad OF=0 \end{array}$$

## Festkommaformat



VZ: Vorzeichen (ob pos. oder neg.)

M: Mantisse (Nachkommastellen)

⇒ Damit lässt der Zahlenbereich darstellen

$$z = (-1)^{V_2} \cdot 0, M$$

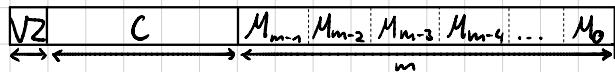
$$= (-1)^{V_2} \cdot \sum_{i=1}^m \frac{M_{m-i}}{2^i}$$

→ m: Bitbreite der Mantisse

⇒ Findet insbesondere im Bereich der Signalverarbeitung Einsatz

⇒ CPUs und GPUs setzen stattdessen auf Gleitkomma

## Gleitkommaformat (Erweiterung der Festkommandarstellung um einen Exponenten E)



C: Charakteristik

$$z = (-1)^{V_2} \cdot 0, M \cdot 2^E$$

$$= (-1)^{V_2} \cdot \left( \sum_{i=1}^m \frac{M_{m-i}}{2^i} \right) \cdot 2^E$$

→ E: Exponent, der das Komma verschiebt

• E < 0: nach links

• E > 0: nach rechts

↳ Dieses Zahlenformat ist nicht eindeutig, da ...

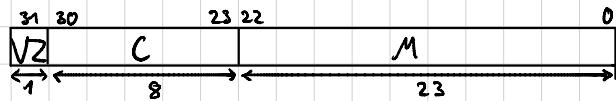
$$\begin{aligned} z.B. \quad 0,001101 &= 000000,1101 \cdot 2^{-2} \\ &= 0000,01101 \cdot 2^{-1} \\ &= 000,001101 \cdot 2^0 \\ &= 0,00001101 \cdot 2^1 \\ &= \dots \end{aligned}$$

Zahl kann auf mehrere Art und Weisen dargestellt werden

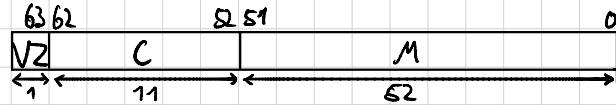
⇒ somit wird dies mit der IEEE-754-Norm genannt

welche folgendes vorgibt

- Single-precision-Format (Bitbreite von 32)



- Double-precision-Format (Bitbreite von 64)



- Vorkommarnormalisiert:

Bedeutet, dass stets 1-Bit sich vor dem Komma befindet

- Mantisse wird gepackt angelegt:

Bedeutet, erste gespeicherte Mantissenbit ist nicht das Vorkomma-Eins, sondern die erste Nachkommastelle.

- Speicherung des Exponenten:

→ Verwendet wird hier die Excess-q-Darstellung um neg. Zahlen zu realisieren.

$$\text{Single: } q = 2^7 - 1 = 127$$

$$\text{Double: } q = 2^{10} - 1 = 1023 \quad \text{als } q$$

Somit ist der Exponent:  $E = C - q$

... und die Charakteristik:  $C = E + q$

- Charakteristik gleich Null:

→ Im Falle, dass die Charakteristik 0 ist, gelten andere Normregeln:

- Exponent ist  $E = 1 - q$  ( $= -126$ )  
Für Singelbit-Pers.

- Gepackte Ziffer vor Komma wird von 1 zu 0

Bsp.: aus 1,00101... 

V2	C	M
----	---	---

wird 0,00101... 

V2	C	M
----	---	---

⇒ Somit lassen sich die kleinstmöglichen Zahlen  $Z_{\min}$  darstellen, sowie die Null, wenn  $M=0$

$$Z_{\min} \in \left\{ a_i \in \{0,1\} \mid \pm 2^{1-q} \cdot \sum_{i=1}^n a_i \cdot \frac{1}{2^i} \right\}$$

und  $Z_{\min}$  befinden sich zwischen:

$$-M_{\max} \cdot 2^{1-9} \quad \text{und} \quad M_{\max} \cdot 2^{1-9}$$

mit:  $M_{\max} = \underbrace{0,1111\dots11}_m_2 = \sum_{i=1}^m \frac{1}{2^i}$   
 m mal  
 Single-Bit: 23 mal

Berechnung von  $M_{\max}$  bei Single-Precision:

$$\begin{aligned} M_{\max} &= \sum_{i=1}^{23} \frac{1}{2^i} = \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 + \dots + \left(\frac{1}{2}\right)^{23} = \\ &= \frac{1}{2} \cdot \left(1 + \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \dots + \left(\frac{1}{2}\right)^{22}\right) \quad \left| \begin{array}{l} \text{Anwendung} \\ \text{Geometrische} \\ \text{Summenformel: } \frac{1-x^{n+1}}{1-x} \end{array} \right. \\ &= \cancel{\frac{1}{2}} \cdot \frac{1 - \left(\frac{1}{2}\right)^{23}}{1 - \frac{1}{2}} = 1 - \frac{1}{2^{23}} \quad \rightarrow \text{Analog Schriftlich: } \begin{array}{r} \overbrace{1,000\dots00}^{x^{23}} \\ - 0,000\dots01 \\ \hline 0,111\dots11 \end{array} \checkmark \end{aligned}$$

Weitere Beispiele für Fließ (Single Precision ( $q=127$ ):

→ 0:

0,0000 0000,0000 0000 0000 0000 0000 0000 0000 0000 0000,

→ -0:

1,0000 0000,0000 0000 0000 0000 0000 0000 0000 0000 0000,

$$\rightarrow M_{\max} \cdot 2^{-126} = \left(1 - \frac{1}{2^{23}}\right) \cdot 2^{-126}$$

0,0000 0000,1111 1111 1111 1111 1111 1111 1111 1111,

$$\rightarrow 1 = 1 \left(0 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + \dots\right) \cdot 2^{\frac{127-127}{2}}$$

0,0111 1111,0000 0000 0000 0000 0000 0000 0000 0000 0000,

→ Infinity (Wenn Exponent den Maximalwert hat)

0,1111 1111,0000 0000 0000 0000 0000 0000 0000 0000 0000,

→ NaN = Not a Number (Wenn Mantisse > 0)

0,1111 1111,1000 1010 0000 1000 0000 0000,

# Speichern einer Dezimalzahl in Singel-Precision Format:

Beispiel: 5,1       $5 : 2 = 2 \text{ r } 1$        $0,1 \cdot 2 = 0,2 \rightarrow 0$   
 $2 : 2 = 1 \text{ r } 0$        $0,2 \cdot 2 = 0,4 \rightarrow 0$   
 $1 : 2 = 0 \text{ r } 1$        $0,4 \cdot 2 = 0,8 \rightarrow 0$   
 $\downarrow$        $0,8 \cdot 2 = 1,6 \rightarrow 1$   
 $5,1 = (101,0\overline{0011})_2$        $0,6 \cdot 2 = 1,2 \rightarrow 1$

$$= (1,0\overline{100011})_2 \cdot 2^2$$

$\sim$  Mantisse

$$\lfloor 2 \rfloor = 0$$

$$C = E + q = 2 + 127 = 129 = (100000001)_2$$

Nebenrechnung:  
 $129 : 2 = 64 \text{ r } 1$   
 $64 : 2 = 32 \text{ r } 0$   
 $32 : 2 = 16 \text{ r } 0$   
 $16 : 2 = 8 \text{ r } 0$   
 $8 : 2 = 4 \text{ r } 0$   
 $4 : 2 = 2 \text{ r } 0$   
 $2 : 2 = 1 \text{ r } 0$   
 $1 : 2 = 0 \text{ r } 1$

$$\Rightarrow 0.\underline{100000001}, \underline{010001100110011001100110110}$$

Probe mit C:

```
uint32_t x = 0b0100000101000110011001100110011;
a = bit2float(x);
show_float(a);
```

5.0999999e+00  
Variable in BIN: 0100|0000 1010|0011 0011|0011 0011|0011

## Aritmetik mit Gleitkommazahlen

### Addition

simples Beispiel mit Dezimalzahlen:

Um die Addition zu erleichtern wird die Zehnerpotenz ausgedrückt

$$0,1234 \cdot 10^1 + 0,9921 \cdot 10^2 =$$

$$= 0,01234 \cdot 10^2 + 0,9921 \cdot 10^2 =$$

→ Fakten müssen jedoch zuerst die Exponenten angepasst werden

$$= (0,01234 + 0,9921) \cdot 10^2 =$$

$$\begin{array}{r} 0,01234 \\ + 0,99210 \\ \hline \end{array}$$

$$= 1,00444 \cdot 10^2$$

⇒ Dadurch verschiebt sich jedoch die Nachkommazahl eins nach rechts, was in der Binärarithmetik zu Verlusten kommen kann

Analog im Binären

$$\begin{aligned} \text{sum} = z_1 + z_2 &= m_1 \cdot 2^{e_1} + m_2 \cdot 2^{e_2} \\ &= m_1 \cdot 2^{e_1} + \frac{m_2 \cdot 2^{e_2}}{2^{e_1}} \cdot 2^{e_1} \quad | \text{Exponentenausgleich} \\ &= \left( m_1 + m_2 \cdot 2^{e_2 - e_1} \right) 2^{e_1} \quad | \text{Ausklammer} \end{aligned}$$

Entspricht einer Bitverschiebung der Mantisse

(mit Vorzeichenbit) um  $e_2 - e_1$  nach rechts

→ zu beachten ist jedoch:

- Excess-q Darstellung des Exponenten
- Das gepackte Vorzeichenbit bei einer Bitverschiebung der Mantisse
- Die Normalisierung der berechneten Summe

→ Eine Subtraktion/Multiplikation wird nach dem selben Prinzip durchgeführt

## Probleme mit Gleitpunktzahlen

• Überlauf („Overflow“) des Exponenten

`print(0.9e25f * 0.9e25f)` → infinity

• Unterlauf („Underflow“) des Exponenten

`print(0.9e25f * 0.9e25f)` → 0.0

• Große Rundungsfehler

- durch Exponentenausgleich bei Addition sehr unterschiedlich großer Zahlen

$$\text{Beispiel: } 10^{23} \left( \underbrace{\left( 10^{-9} + 1 \right)}_{\approx 1} - 1 \right) \approx 10^{23} (1 - 1) = 0$$

$$10^{23} \left( 10^{-9} + (1 - 1) \right) = 10^{23-9} = 10^{14}$$

! Dass Assoziativgesetz gilt somit nicht!

- durch Stellenausschaltung bei Subtraktion gleich großer Zahlen

⇒ Ergebnisse können unter Umständen erheblich vom exakten Wert abweichen

## Vorsichtsmaßnahmen:

• Statt:

- $x = 0 \rightarrow |x| < d$
  - $x = y \rightarrow |x-y| < d$
- $\left. \begin{array}{l} \\ \end{array} \right\} d \stackrel{\text{!}}{=} \text{"Toleranzdifferenz"}$

• Terme auf weniger kritische Rechenoperationen umstellen:

$$\sqrt{1+x} - \sqrt{1-x} = \dots = \frac{2x}{\sqrt{1+x} + \sqrt{1-x}}$$

2 Substitutionen    1 Subtraktion

# Schaltnetze

## Entwurf und Realisierung einer Schaltfunktion

### Logische Ebene

- Wertetabelle der Funktion aufstellen
- Minterme bestimmen
- vereinfachen  
↳ Verwendung von anderen Operatoren

### Physische Ebene

- Erstellung der Grundgatter.  
→ AND, OR, NOT
- Minimierung
- Aufbau anderer Gatter

→ Implementierung der Schaltung

⇒ Grundsätzliche Vorgehensweise um aus einer Wertetabelle eine Schaltung zu erstellen

### Decoder

Ein Decoder ist ein Konverter der digitalen Signale in Analoge umhandelt.

Bsp.: Eine binäre Zahl (0-3) als Dezimalzahl darstellen

$$(D(s_1, s_0) = z \rightarrow D(\underbrace{1, 0}_{(10)_2}) = 2)$$

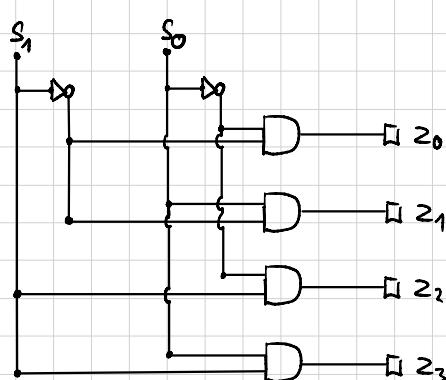
Wertetabelle:

$s_1, s_0$	$z_3 z_2 z_1 z_0$
00	0001
01	0010
10	0100
11	1000

Minterme:

$$\left. \begin{array}{l} z_0(s_1, s_0) = \bar{s}_1 \cdot \bar{s}_0 \\ z_1(s_1, s_0) = \bar{s}_1 \cdot s_0 \\ z_2(s_1, s_0) = s_1 \cdot \bar{s}_0 \\ z_3(s_1, s_0) = s_1 \cdot s_0 \end{array} \right\}$$

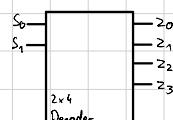
$$z(s_1, s_0) = \begin{cases} \bar{s}_1 \cdot \bar{s}_0 & \bar{s}_0 \text{ für } z_0 \\ \bar{s}_1 \cdot s_0 & s_0 \text{ für } z_1 \\ s_1 \cdot \bar{s}_0 & \bar{s}_0 \text{ für } z_2 \\ s_1 \cdot s_0 & s_0 \text{ für } z_3 \end{cases}$$



Als Schaltkreis

So ein Decoder würde sich 2x4 Decoder nennen, da er aus 2 Eingangssignalen 4 Zustände repräsentieren kann

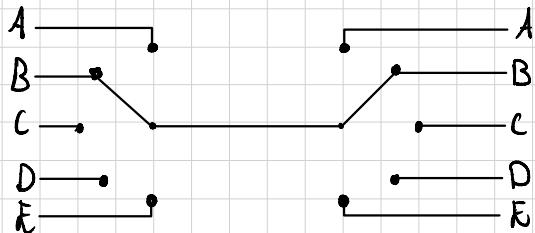
Vereinfachte  
Abbildung:



Lukas Mensch

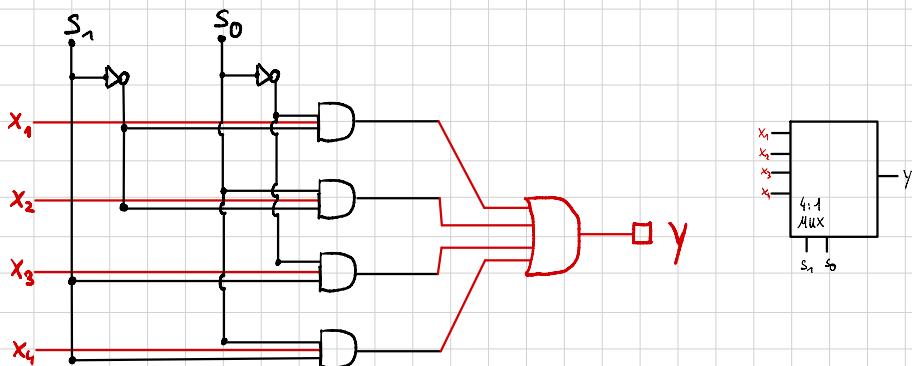
# Multiplexer und Demultiplexer

Ein Multiplexer (MUX) überträgt ein Signal von mehreren Inputs über eine Leitung zu einem Demultiplexer (DEMUX)

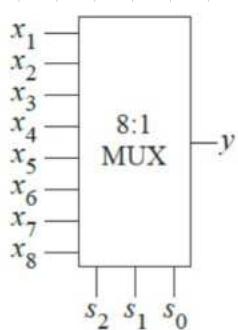


Dieser Unterschiede sich dabei nicht stark von einem Decoder, womit durch ein paar Modifikationen aus einem Decoder ein MUX erstellt werden kann.

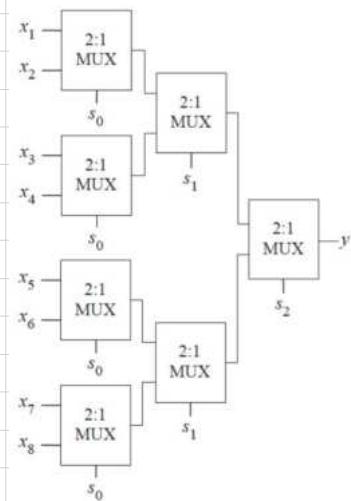
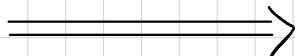
Beispiel 4:1 MUX



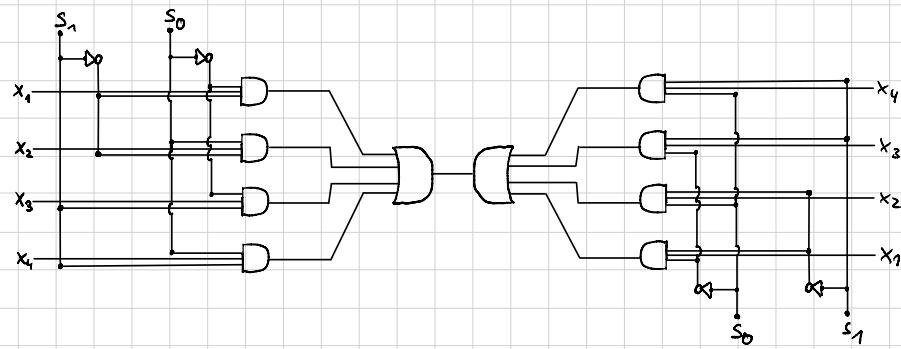
$$y(s_1, s_0) = \bar{s}_1 \bar{s}_0 x_1 + \bar{s}_1 s_0 x_2 + s_1 \bar{s}_0 x_3 + s_1 s_0 x_4 = \bar{s}_1 (\bar{s}_0 x_1 + s_0 x_2) + s_1 (\bar{s}_0 x_3 + s_0 x_4)$$



MUX kann auf kleinere verteilt werden



## 4 Bit - Kanal



Mit Multiplexoren lassen sich auch beliebige Funktionen erstellen, da der Aufbau eines Min-Term der selbe ist.

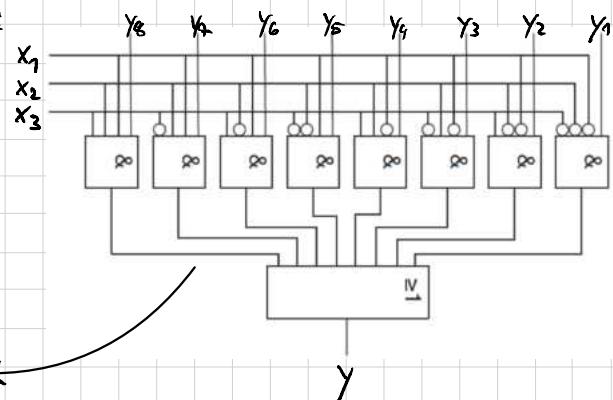
Beispiel:

$x_1$	$x_2$	$x_3$	$y$
0	0	0	$y_1$
0	0	1	$y_2$
0	1	0	$y_3$
0	1	1	$y_4$
1	0	0	$y_5$
1	0	1	$y_6$
1	1	0	$y_7$
1	1	1	$y_8$

Als Min-Term:

$$y(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 y_1 + \bar{x}_1 \bar{x}_2 x_3 y_2 + \bar{x}_1 x_2 \bar{x}_3 y_3 + \bar{x}_1 x_2 x_3 y_4 + \\ x_1 \bar{x}_2 \bar{x}_3 y_5 + x_1 \bar{x}_2 x_3 y_6 + \bar{x}_1 \bar{x}_2 x_3 y_7 + x_1 x_2 \bar{x}_3 y_8$$

Als Schaltnetz:

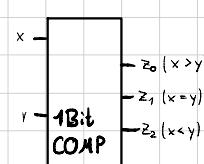


## Rechnerische Module

Komparatoren  $>, <, =$

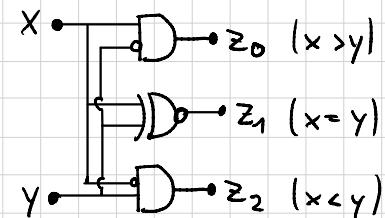
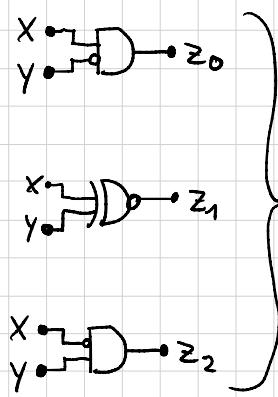
Ein Komparator, welcher nur 1-Bit große Zahlen vergleichen soll, lässt sich durch eine Wahrheitstabelle einfach realisieren

$x$	$y$	$z_0$	$z_1$	$z_2$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0



## Min-Terme

$$x > y (x_0, y_0) = x_0 \bar{y}_0$$



$$\begin{aligned} x = y (x_0, y_0) &= \bar{x}_0 \bar{y}_0 + x_0 y_0 \\ &= x_0 \text{XOR } y_0 \end{aligned}$$

$$x < y (x_0, y_0) = \bar{x}_0 y_0$$

Bei einem Vergleich mit größeren Zahlen wird beim höchstwertigen Bit angefangen zu vergleichen und ggf. bei Gleichheit der Vergleich bis zum niederwertigsten Bit weitergeführt.

Vergleich zwischen  $X_{MSB}$  u.  $Y_{MSB}$

$$\text{Wenn } \begin{cases} X_{MSB} > Y_{MSB} \rightarrow X > Y \\ X_{MSB} < Y_{MSB} \rightarrow X < Y \end{cases}$$

Sonst wenn  $X_{MSB} = Y_{MSB}$

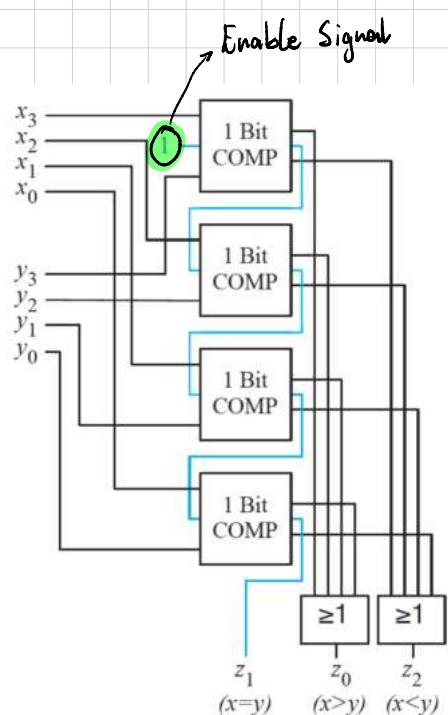
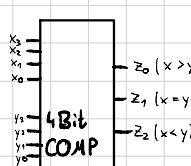
$\hookrightarrow$  Vergleich zwischen  $X_{MSB-1}$  u.  $Y_{MSB-1}$

[...]

Sonst wenn  $X_0 = Y_0 \rightarrow X = Y$

Beispiel 4-Bit Komparator:

$x_3 x_2 x_1 x_0$	$y_3 y_2 y_1 y_0$	$x > y$	$x = y$	$x < y$
1 - - -	0 - - -	1	0	0
0 - - -	1 - - -	0	0	1
=y <sub>3</sub> 1 - -	=y <sub>3</sub> 0 - -	1	0	0
=y <sub>3</sub> 0 - -	=x <sub>3</sub> 1 - -	0	0	1
=y <sub>3</sub> =y <sub>2</sub> 1 -	=x <sub>3</sub> =x <sub>2</sub> 0 -	1	0	0
=y <sub>3</sub> =y <sub>2</sub> 0 -	=x <sub>3</sub> =x <sub>2</sub> 1 -	0	0	1
=y <sub>3</sub> =y <sub>2</sub> =y <sub>1</sub> 1	=x <sub>3</sub> =x <sub>2</sub> =x <sub>1</sub> 0	1	0	0
=y <sub>3</sub> =y <sub>2</sub> =y <sub>1</sub> 0	=x <sub>3</sub> =x <sub>2</sub> =x <sub>1</sub> 1	0	0	1
=y <sub>3</sub> =y <sub>2</sub> =y <sub>1</sub> =y <sub>0</sub>	=x <sub>3</sub> =x <sub>2</sub> =x <sub>1</sub> =y <sub>0</sub>	0	1	0



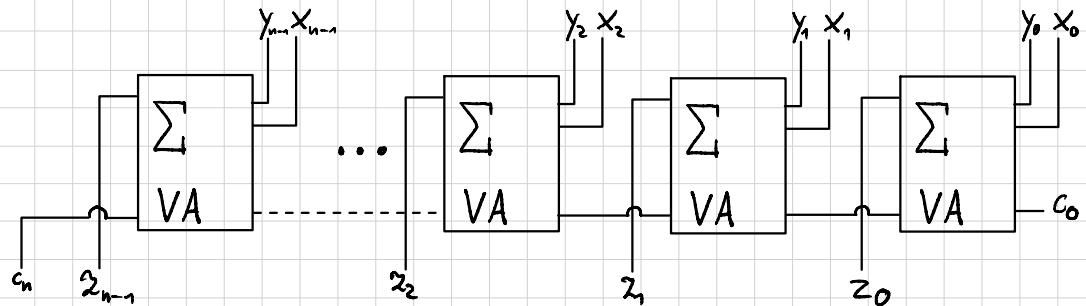
- Das Enable-Signal schaltet weitere Komponenten frei bei Gleichheit zweier Bits

- Der Fall  $>$  bzw.  $<$  wird mit einem "oder" zu einem  $z_0 / z_2$  Signal zusammengeschlossen

- Wenn alle Bits gleich sind, ist die Zahl gleich

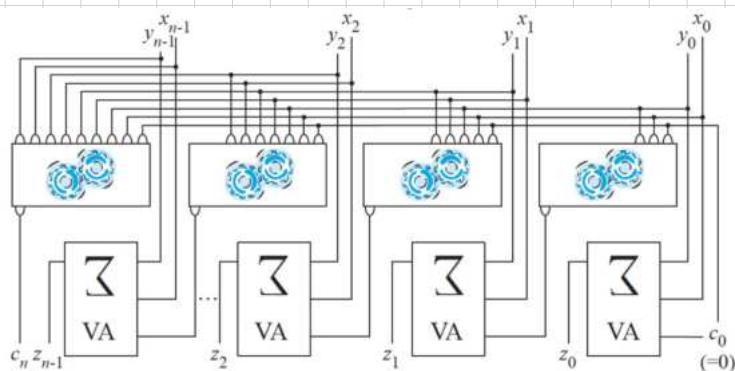
Verschiedene Arten eines Addierers:

- n-Bit-Carry-ripple-Addierer:



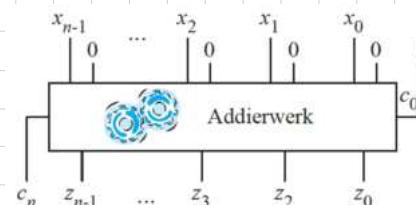
- Carry-look-ahead-Addierer:

(Parallelierung durch Vorausberechnung des Trags)



- Inkrementierer

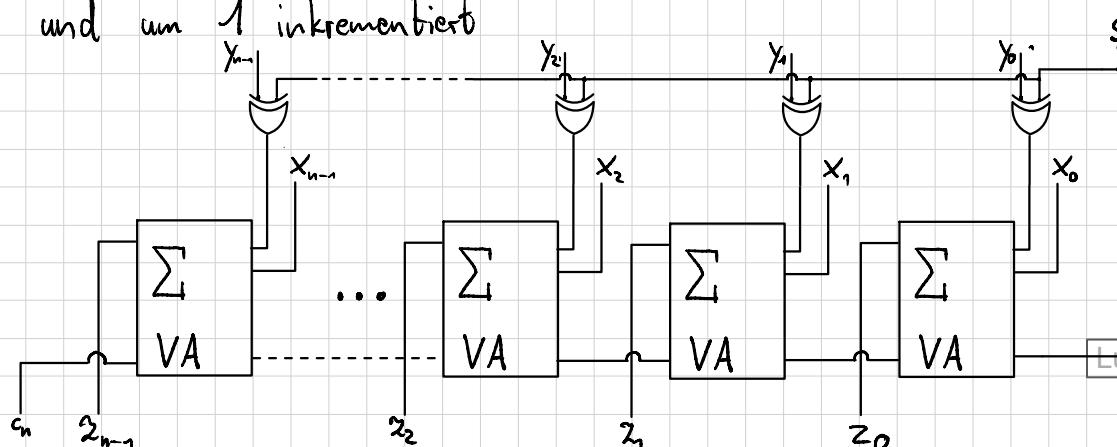
→ Ein Spezialfall des Addierers, wo der angelegte Wert um 1 erhöht wird



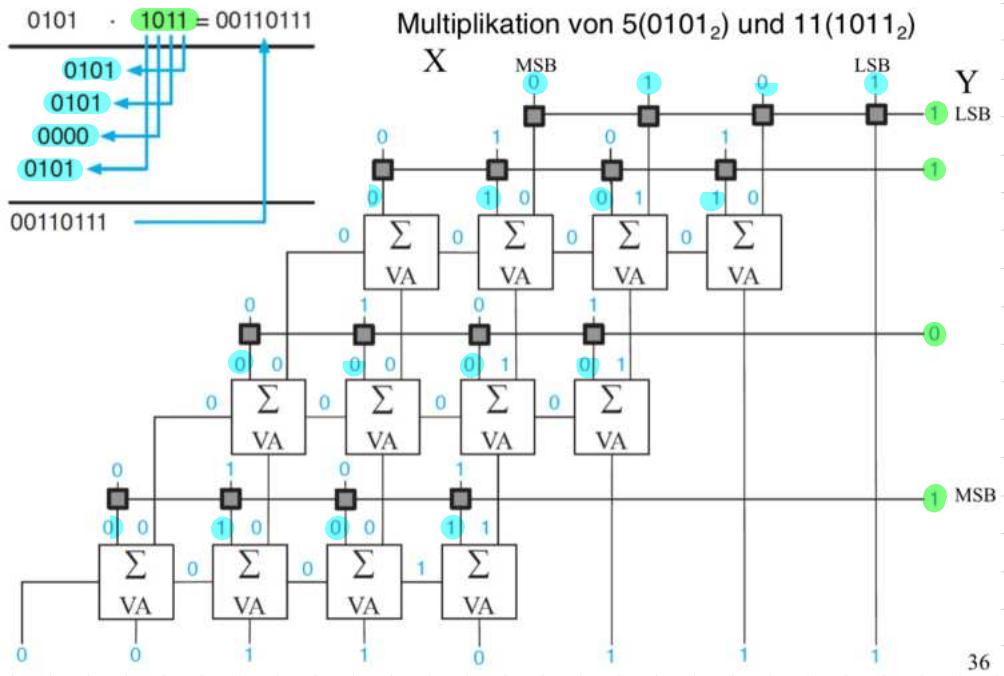
- Subtrahierer

→ Subtraktion ist die Addition mit einer negierten Zahl  
 $\hookrightarrow x - y = x + (-y)$

→ Um y zu negieren werden alle Bits mit einem XOR geflippert und um 1 inkrementiert



## • Multiplizierer



## • Bitshift

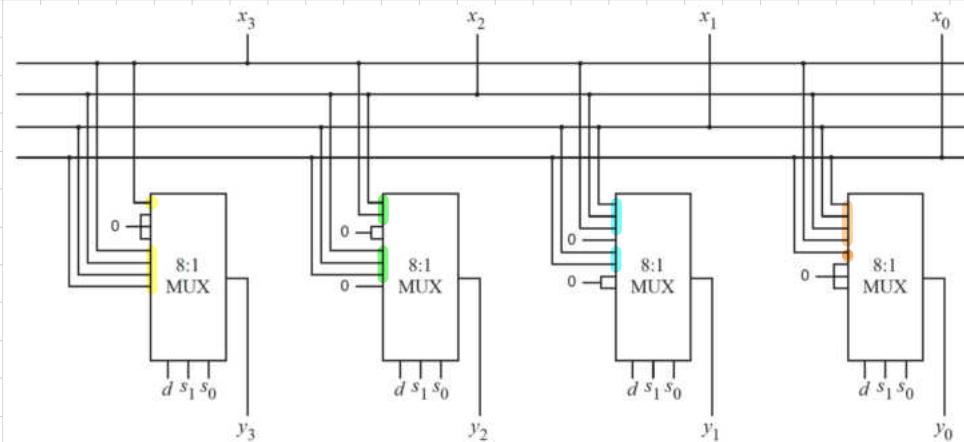
→ Barrel-Shifter

d s <sub>1</sub> s <sub>0</sub>	y <sub>3</sub> y <sub>2</sub> y <sub>1</sub> y <sub>0</sub>
0 0 0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> x <sub>0</sub>
0 0 1	0 x <sub>3</sub> x <sub>2</sub> x <sub>1</sub>
0 1 0	0 0 x <sub>3</sub> x <sub>2</sub>
0 1 1	0 0 0 x <sub>3</sub>
1 0 0	x <sub>3</sub> x <sub>2</sub> x <sub>1</sub> x <sub>0</sub>
1 0 1	x <sub>2</sub> x <sub>1</sub> x <sub>0</sub> 0
1 1 0	x <sub>1</sub> x <sub>0</sub> 0 0
1 1 1	x <sub>0</sub> 0 0 0

Als Min-Terme:

$$B(d, s_1, s_0) = \begin{cases} \bar{d} \bar{s}_1 \bar{s}_0 \bar{x}_3 + \bar{d} \bar{s}_1 \bar{s}_0 \bar{x}_3 + \bar{d} \bar{s}_1 s_0 \bar{x}_2 + \bar{d} s_1 \bar{s}_0 x_1 + d s_1 s_0 x_0 & \text{für } y_3 \\ \bar{d} \bar{s}_1 \bar{s}_0 \bar{x}_2 + \bar{d} \bar{s}_1 s_0 \bar{x}_3 + \bar{d} s_1 \bar{s}_0 x_2 + \bar{d} s_1 s_0 x_1 + d s_1 \bar{s}_0 x_0 & \text{für } y_1 \\ \bar{d} \bar{s}_1 \bar{s}_0 x_1 + \bar{d} \bar{s}_1 s_0 \bar{x}_2 + \bar{d} s_1 \bar{s}_0 x_3 + \bar{d} s_1 \bar{s}_0 x_1 + \bar{d} \bar{s}_1 s_0 x_0 & \text{für } y_2 \\ \bar{d} \bar{s}_1 \bar{s}_0 x_0 + \bar{d} \bar{s}_1 s_0 \bar{x}_1 + d s_1 \bar{s}_0 \bar{x}_2 + \bar{d} s_1 s_0 \bar{x}_3 + \bar{d} \bar{s}_1 s_0 x_0 & \text{für } y_0 \end{cases}$$

Anstatt die Funktion  $B(d, s_1, s_0)$  weiter zu minimieren, lässt sich diese mit Multiplexoren sofort realisieren (siehe Multiplexoren):



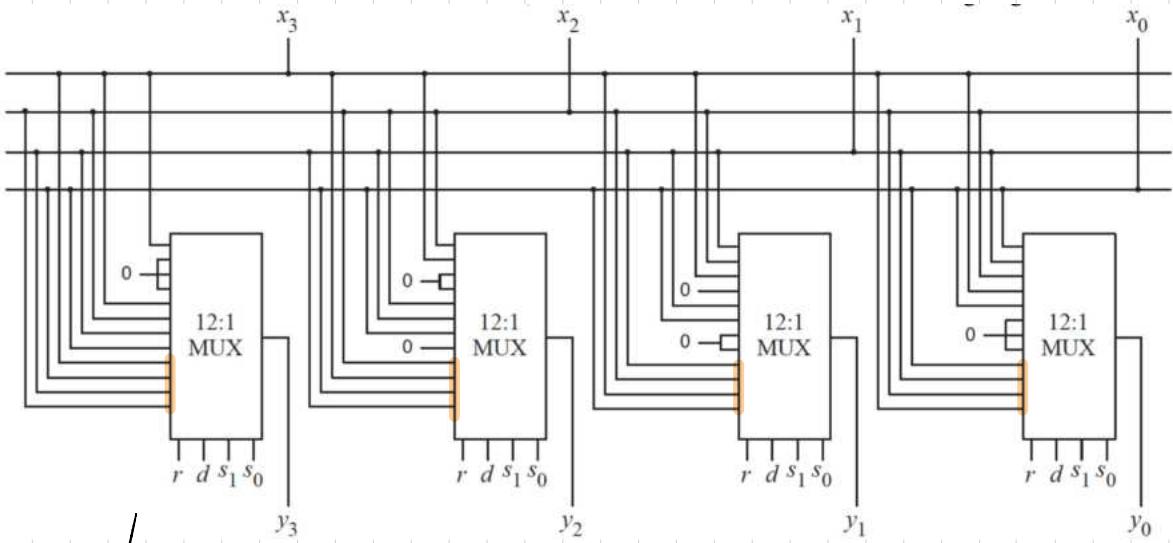
## → Rotierender Barrel-Shifter

(hierbei wird das wegfallende Bit am anderem Ende wieder angefügt)

Wahrheitstabelle für  $d=0, r=1$ :

→ Rechtsrotation

$s_1 s_0$	$y_3$	$y_2$	$y_1$	$y_0$
0 0	$x_3$	$x_2$	$x_1$	$x_0$
0 1	$x_0$	$x_3$	$x_2$	$x_1$
1 0	$x_1$	$x_0$	$x_3$	$x_2$
1 1	$x_2$	$x_1$	$x_0$	$x_3$

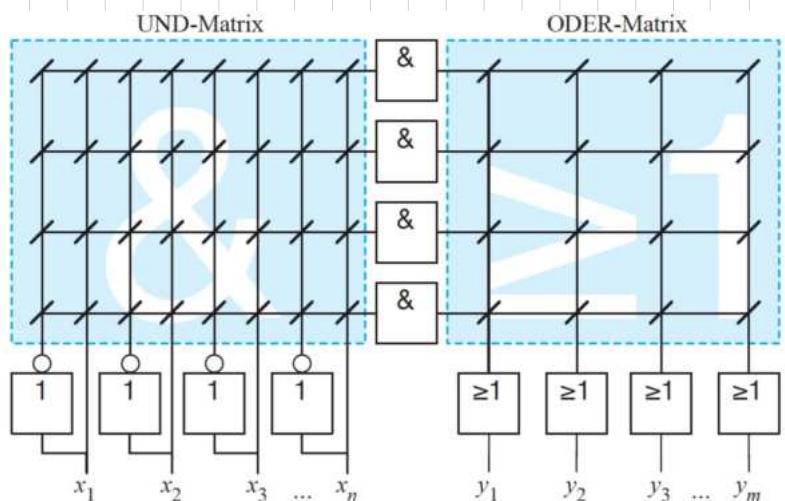


↳ Ich geh hier nicht weiter drauf ein >w<

## Programmierbare Logikbausteine

Funktionen können auch durch ein Programmable logic Array (PLA) implementiert werden.

Dabei lassen sich in der Konjunktions-Matrix durch setzen von Verbindungs punkten beliebig viele Min-Terme erstellen, welche in der Konjunktions-Matrix kombiniert werden.



Beispiel für folgende Wahrheitstabelle:

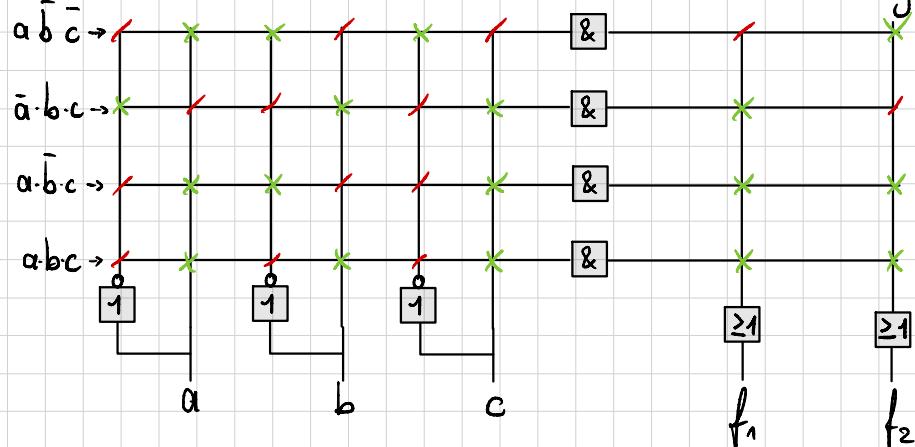
a	b	c	$f_1$	$f_2$
0	0	0	00	
0	0	1	00	
0	1	0	00	
0	1	1	10	
1	0	0	01	
1	0	1	11	
1	1	0	00	
1	1	1	11	

$$F(a,b,c) = \begin{cases} a \cdot b \cdot c + a \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot c & \text{für } f_1 \\ a \cdot b \cdot c + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c} & \text{für } f_2 \end{cases}$$

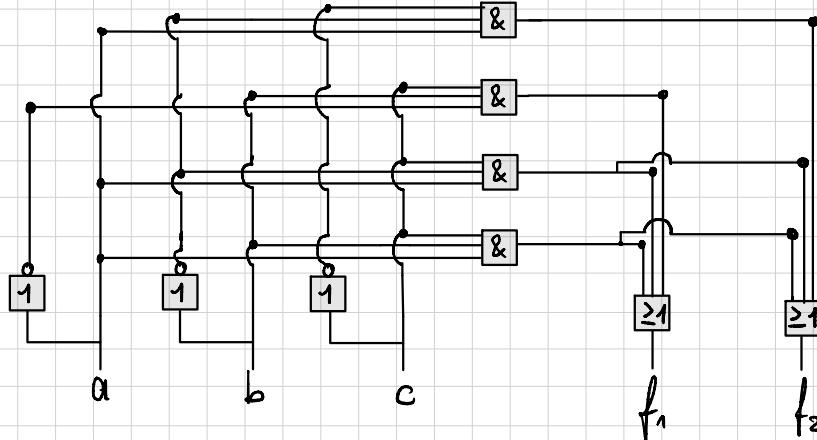
Legende:

- / nicht angeschlossen
- ✗ angeschlossen

Realisierung in einem PLA:



Vereinfacht:



→ Sehr flexibel

→ kann auf Basis reprogrammierbaren Funktionen erstellt werden

→ Gut geeignet von mehreren Schaltfunktionen eines Gerätes

# Parametrisierbare Schaltnetze

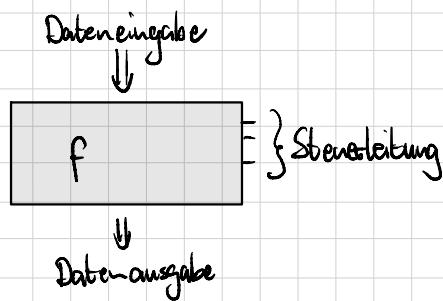
Motivation:

$$\text{Aus } f(e_1 \dots e_n) = a \rightarrow f_{e_2 e_3 \dots e_n}(e_1) = a \text{ machen}$$

$\Rightarrow e_2, e_3, \dots, e_n$  werden als Parameter betrachtet

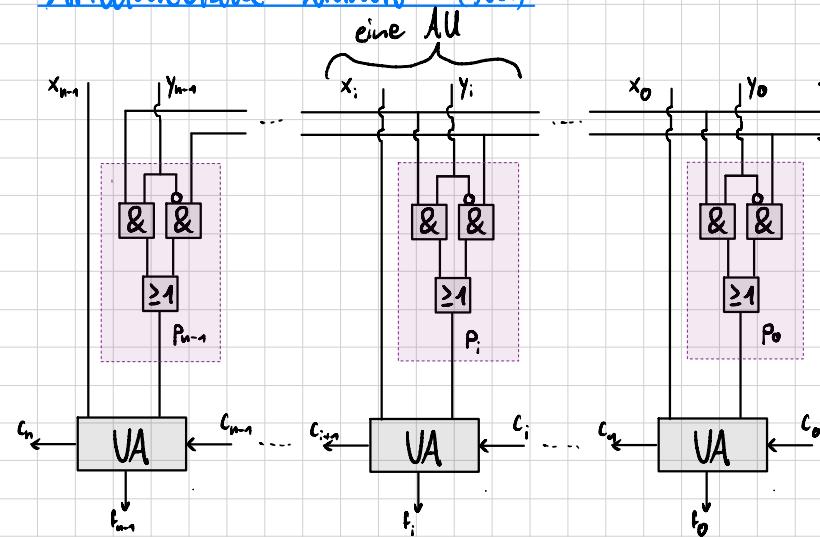
$\Rightarrow$  das Verhalten der einstelligen Funktion  $f_{e_2, \dots, e_n}(e_1) \Rightarrow a$  lässt sich nun steuern

$\Rightarrow$  dient als Grundlage von Befehlen einer programmierbaren Steuereinheit.



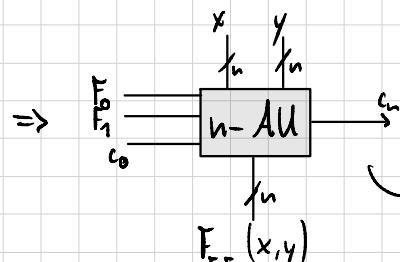
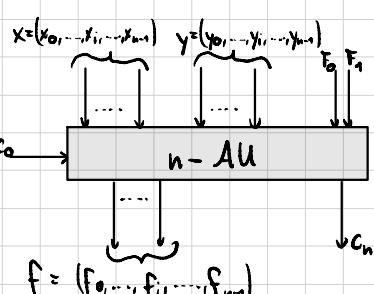
$\Rightarrow$  Vorteil: Schaltfunktion kann geändert werden, ohne den Schaltanlagen zu ändern.

## Arithmetische Einheit (AU)



$F_0 F_n$	$c_i$	$p_{F_0 F_n}(y_i)$	interpretation von $f_{F_0 F_n c_i}(x_i, y_i)$
0 0	0	0	$x_i$
0 0	1	0	$x_i + 1$ (increment)
0 1	0	$\bar{y}_i$	$x_i - y_i$ im 1er-Komplement
0 1	1	$\bar{y}_i$	$x_i - y_i$ im 2er-Komplement (also $(x_i - y_i) + 1$ )
1 0	0	$y_i$	$x_i + y_i$
1 0	1	$y_i$	$x_i + y_i + 1$
1 1	0	1	$x_i + 1$
1 1	1	1	$x_i + 2$

Somit erhalten wir eine n-Bit AU



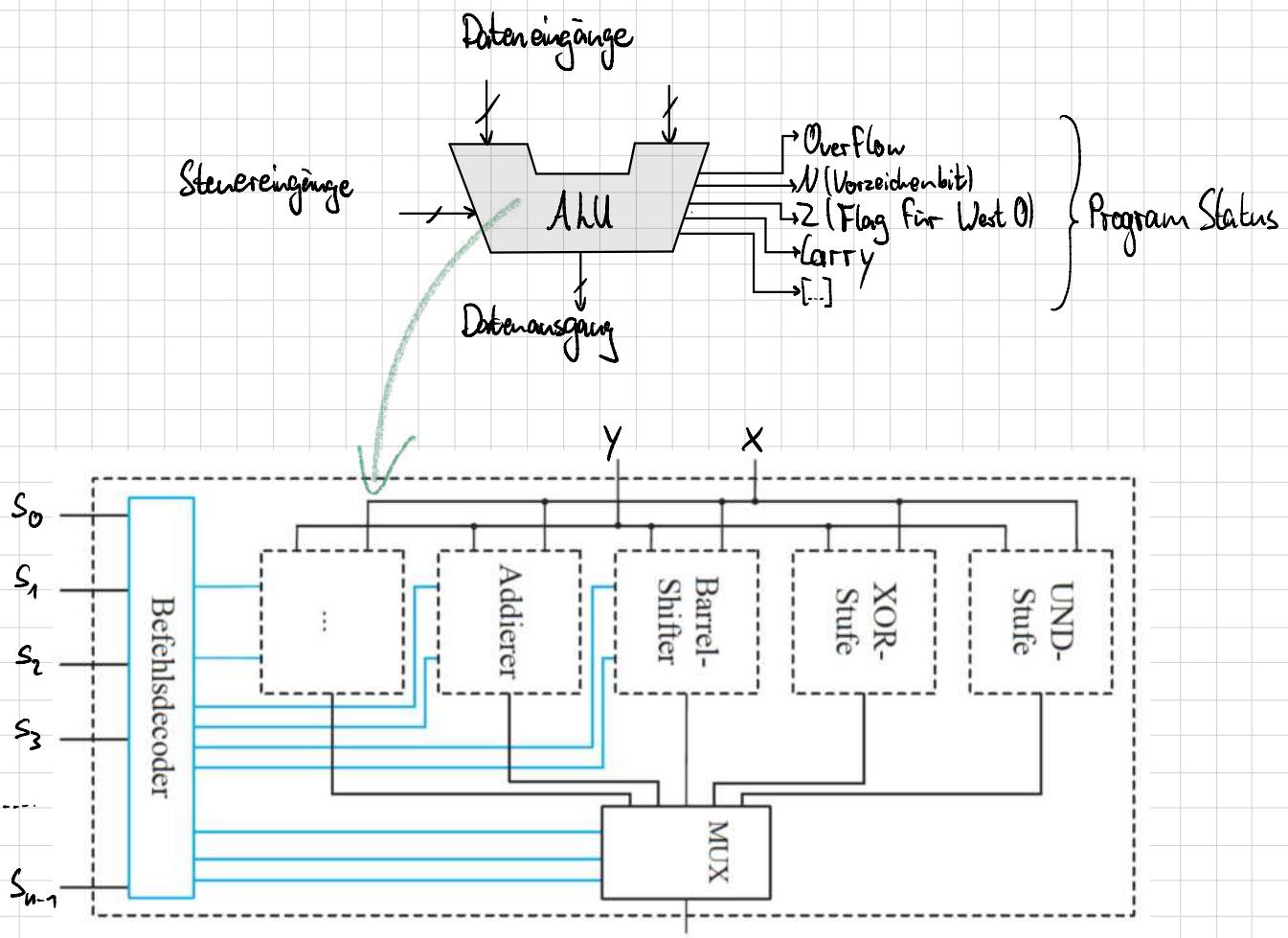
$\hookrightarrow$  AU muss mind. die Addition realisieren

Lukas Mensch

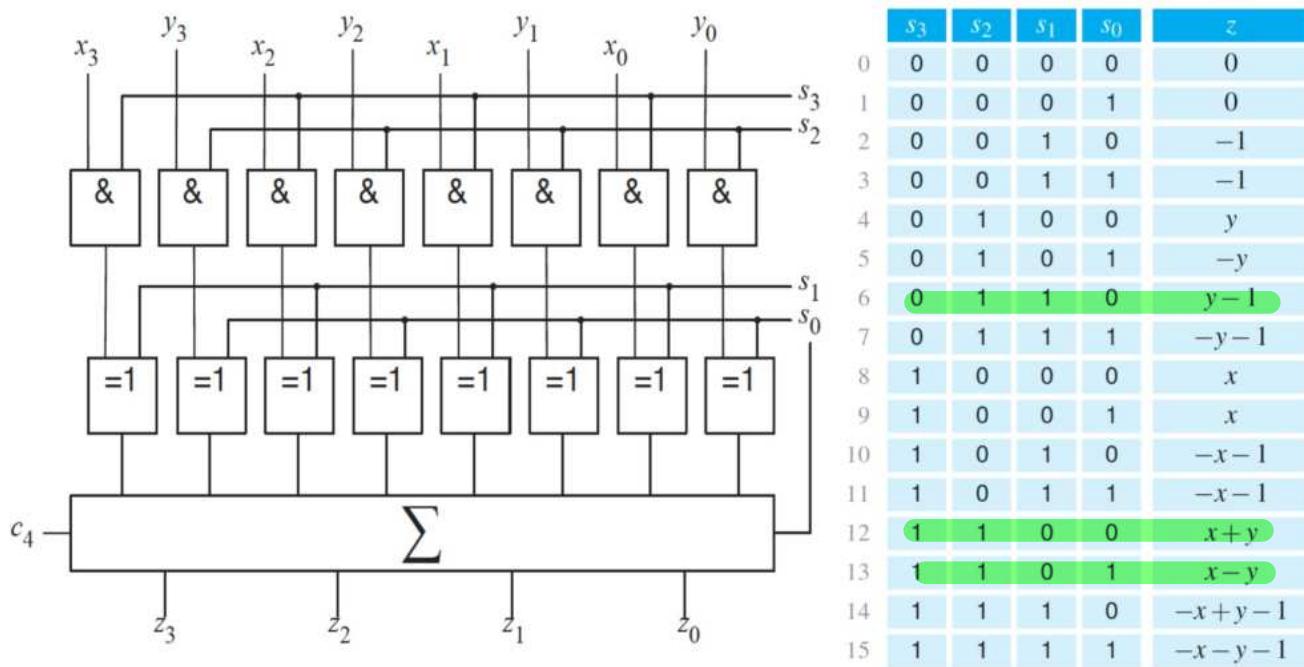
# Arithmetisch-logische Einheit (ALU)

vollständiges  
realisierbar

→ parametrisierbare Einheit, welche den Minimalzettel von logischen und arithmetischen Funktionen realisiert



Beispiel: 4-Bit - Addierer / Subtrahierer / Inkrementer / Decrementer



# Schaltwerke Teil 1

Ein Schaltwerk gibt es nur einen gerichteten Signalfluss

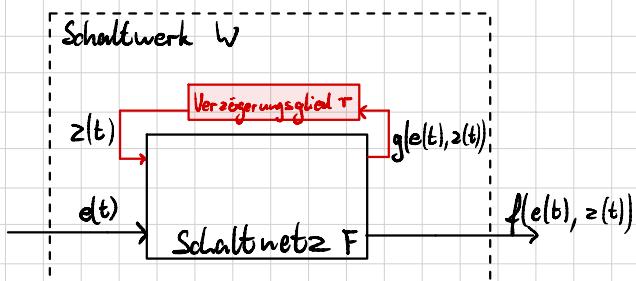


von Eingang bis Ausgang

→ Gatter-Schaltzeit spielt hierbei keine Rolle

→ Die Ausgabe von einem Schaltnetz sind von der Eingabe abhängig

Bei einem Schaltwerk ist eine Rückkopplung eingebettet.

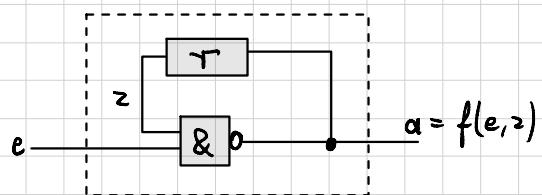


⇒ Ein Schaltwerk besitzt eine Art Gedächtnis

⇒ Die Ausgabe ist von vorangegangenen Eingaben abhängig

## Stabile vs. instabile Schaltwerke

Betrachte wird folgendes Schaltwerk:



Diese ist:

Instabil bei einer ständigen Eingabe von  $e=1$ :

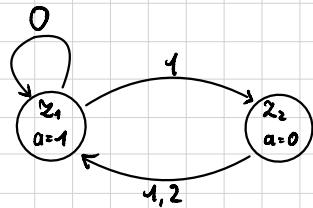
$$\Rightarrow f(e=1, z) = \dots 0 | 1 | 0 | 1 | 0 \dots$$

Stabil bei einer ständig Eingabe von  $e=0$ :

$$\Rightarrow f(e=0, z) = 1$$

Modellierung dieses Schaltwerkes

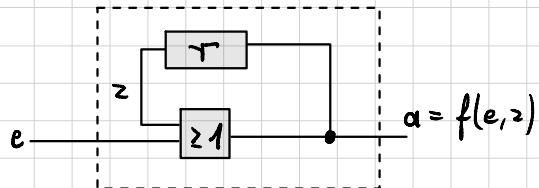
- Als Automat (Moore-Automat)



Übergänge in Tabelle:

$z_1 e$	0	1
$z_1$	$z_1$	$z_2$
$z_2$	$z_1$	$z_1$

### Merkeln von Zuständen (Fanregister)



Bei einer einmaligen Eingabe von  $e=1$  erhält man immer

$$\Rightarrow f(e, z) = 1$$

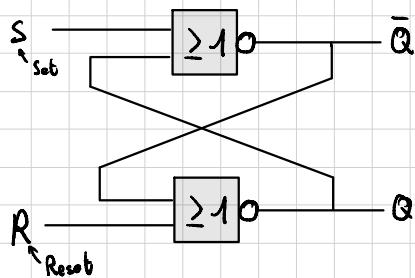
### Trigger-Schaltungen Latch u. Flipflop

Es wird ein elementares Speicherelement als Bauteil gesucht, welches zwischen zwei verschiedenen Zuständen  $z_1$  u.  $z_2$  unterscheiden kann, so wie sich merkt in welchem Zustand er sich befindet. ( $\Rightarrow$  Speicherelement kann 1Bit speichern )

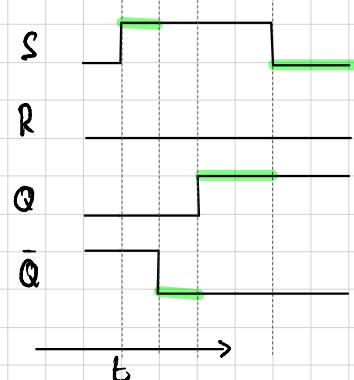
Aufforderungen:

- Speicherung der Information
- Einschreiben (Zustand auf 1 oder 0)
- Auslesen der Information

Implementierung mit Flip Flop



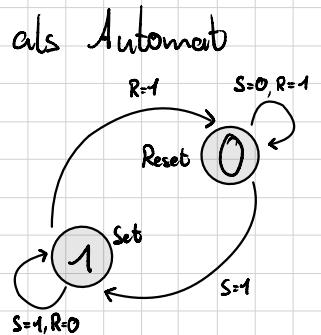
Zeitdiagramm



Wertetabelle:

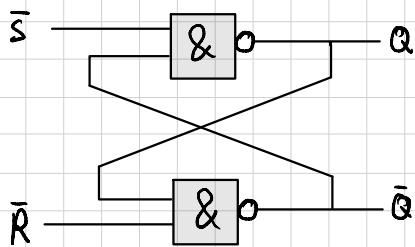
S	R	$Q_n$	$Q_{n+1}$
0	0	0	0
0	1	1	1
1	0	0	0
1	1	0	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?

} Speicherfall  
} Rücksetzfall  
} Setzfall  
} undefiniert [darf nicht auftreten]

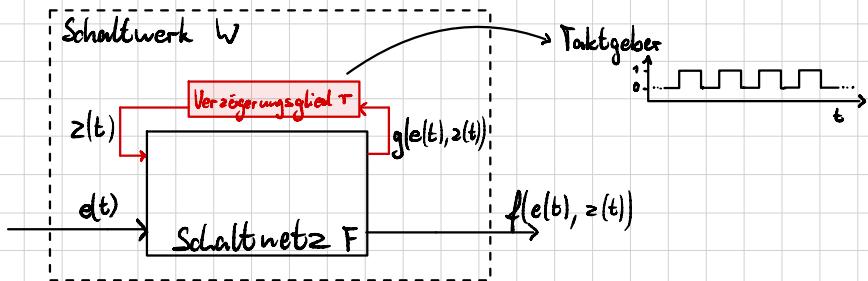


Lukas Mensch

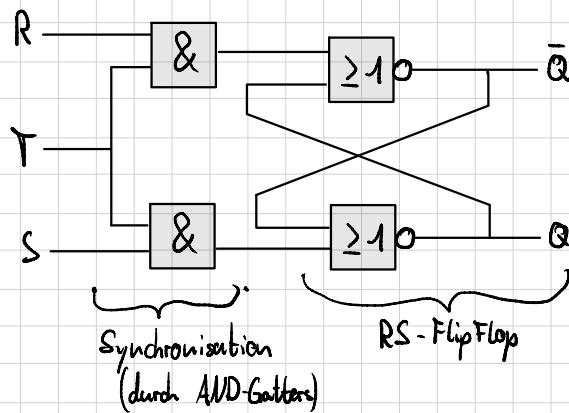
Auf Basis von AND-Gatter



### Gekoppeltes Schaltwerk

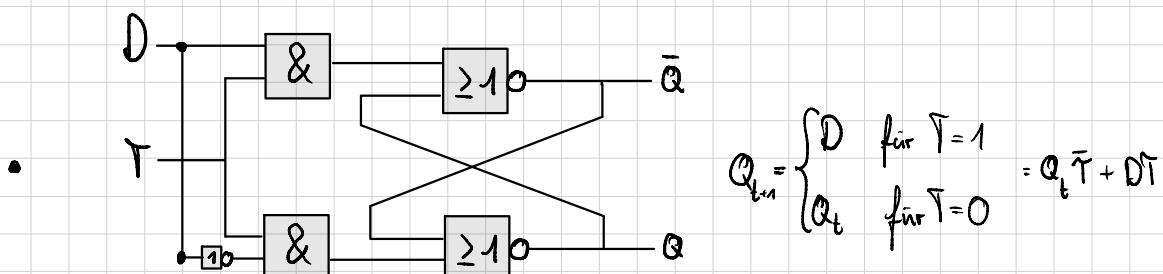


Motivation: Speicherstände sollen zu definierten Zeitpunkten  $t$  gesetzt werden.



$$Q_{t+n} = \begin{cases} S + \bar{R} \cdot Q_t & \text{für } T=1 \\ Q_t & \text{für } T=0 \end{cases}$$

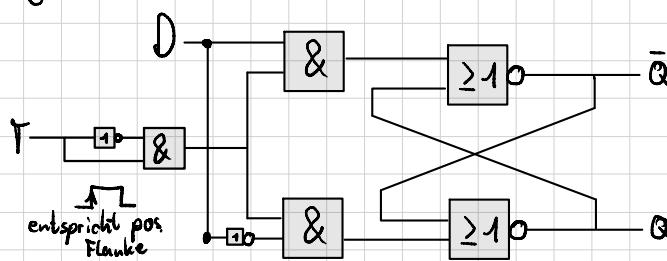
Weitere Verbesserungen:



$$Q_{t+n} = \begin{cases} D & \text{für } T=1 \\ Q_t & \text{für } T=0 \end{cases} = Q_t \bar{T} + D T$$

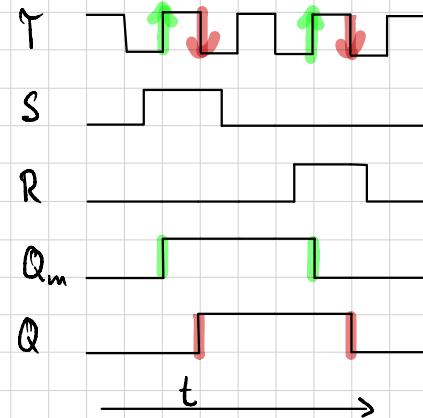
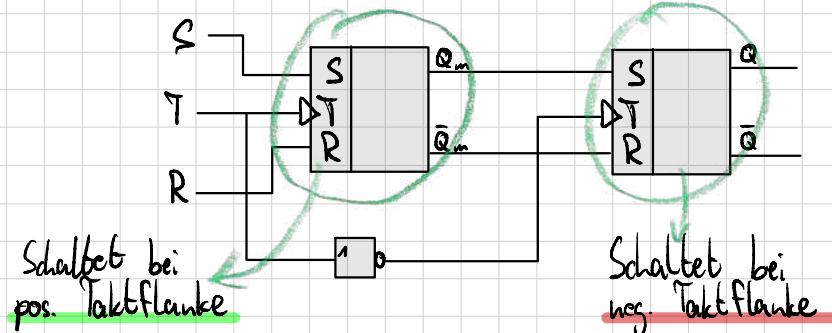
↳ Das Problem  $S=R=1$  gleichzeitig ist  
wird so verhindert

- Aktivierung des RS-FF nur durch am Taktanfang (Takt flankengesteuert)



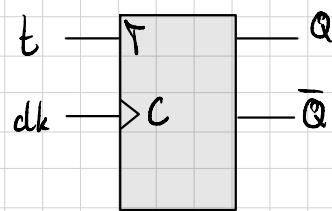
# Master-Slave Flipflop

- Durch serielle Hintereinanderschaltung wird ein Bit erst nach einer neg. Taktflanke geflippert



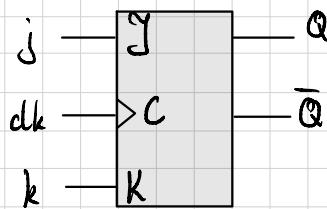
## Weitere FlipFlops

T - Element:



clk	t	$q^{t+1}$
0/1/1/1	-	$q^t$
↓/↑	0	$q^t$
↓/↑	1	$\overline{q^t}$

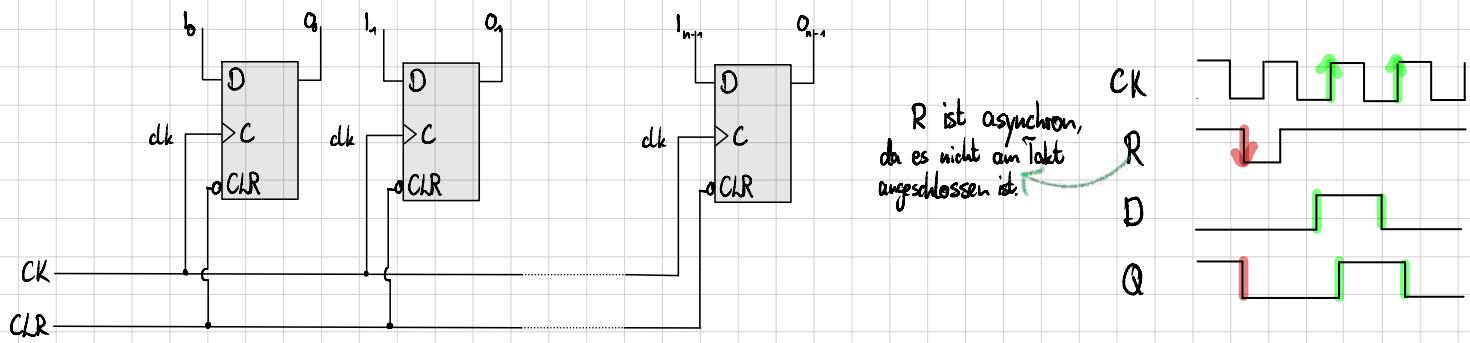
JK-Element:



clk	j	k	$q^{t+1}$
0/1/1/1	-	-	$q^t$
↓/↑	0	0	$q^t$
↓/↑	0	1	0
↓/↑	1	0	1
↓/↑	1	1	$\overline{q^t}$

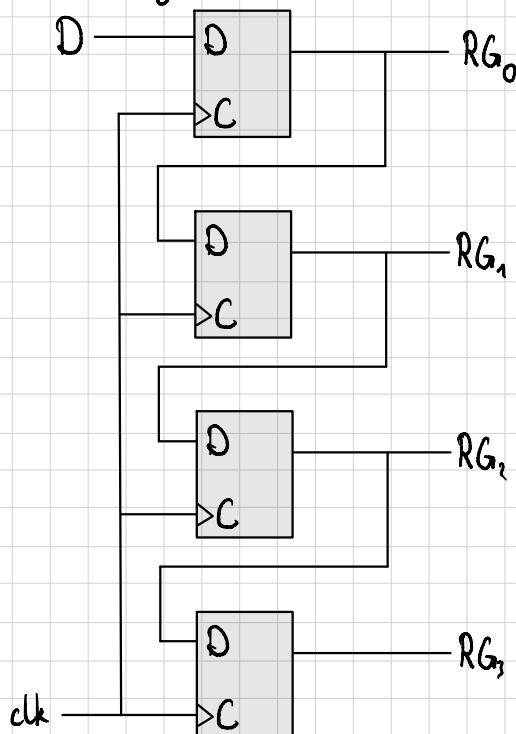
# Register

n-Bit-Register aus Einzelbit-FlipFlops

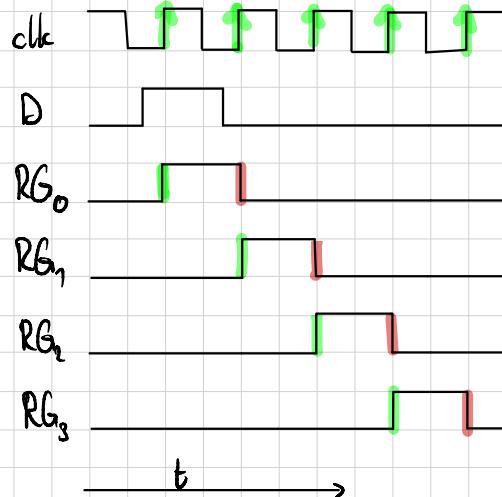


R ist asynchron,  
da es nicht am Takt  
angeschlossen ist.

## Schieberegister



Schieberegister sind eine Verkettung von D-FlipFlops, welche den D-Eingang von Vorfänger zum Nachfolger weiterleiten.

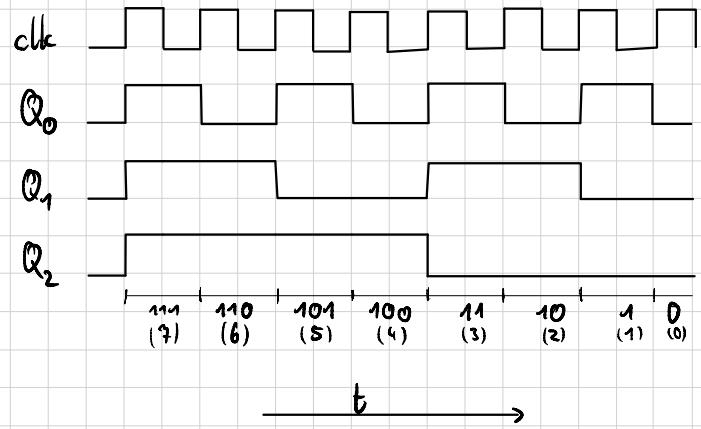
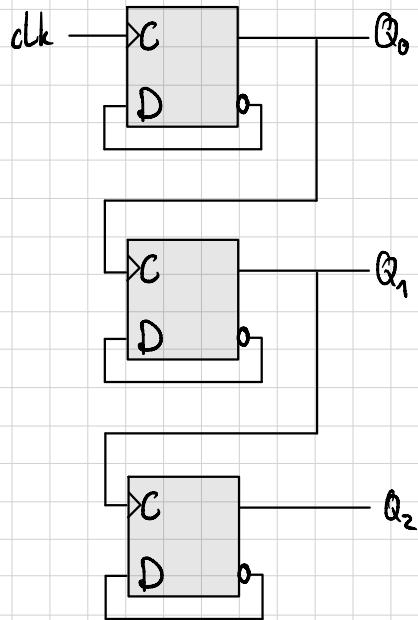


Signal von D geht bei:

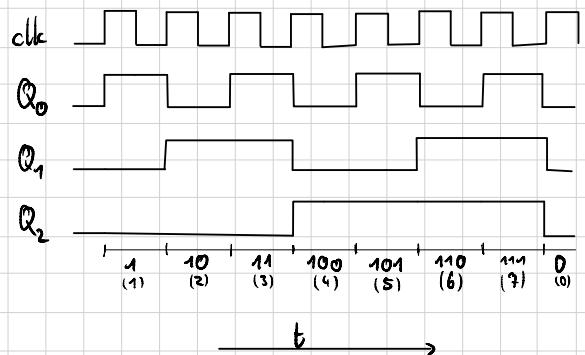
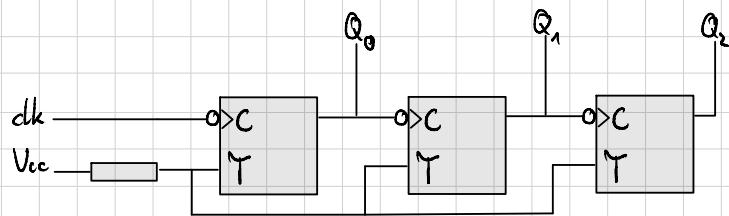
- 1. pos. Taktflanke  $\rightarrow RG_0$
- 2. pos. Taktflanke  $\rightarrow RG_1$
- 3. pos. Taktflanke  $\rightarrow RG_2$
- 4. pos. Taktflanke  $\rightarrow RG_3$

# Zähler

Subtrahierender Binärzähler (D-latch serial über Clk verlaufen)



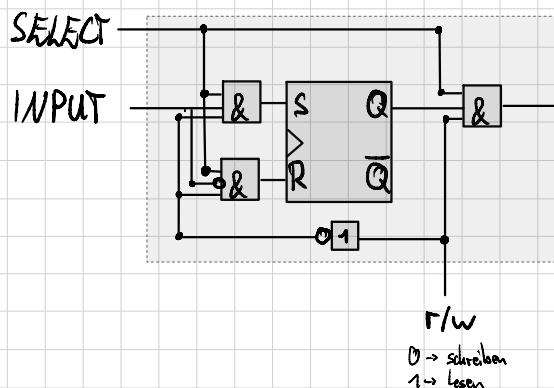
Binärzähler (Serrische Schaltung von T-FlipFlops neg. Tackfunkengesteuert)



# Schaltwerke Teil 2

## Elementare Speicherzelle (Binary Cell (BC))

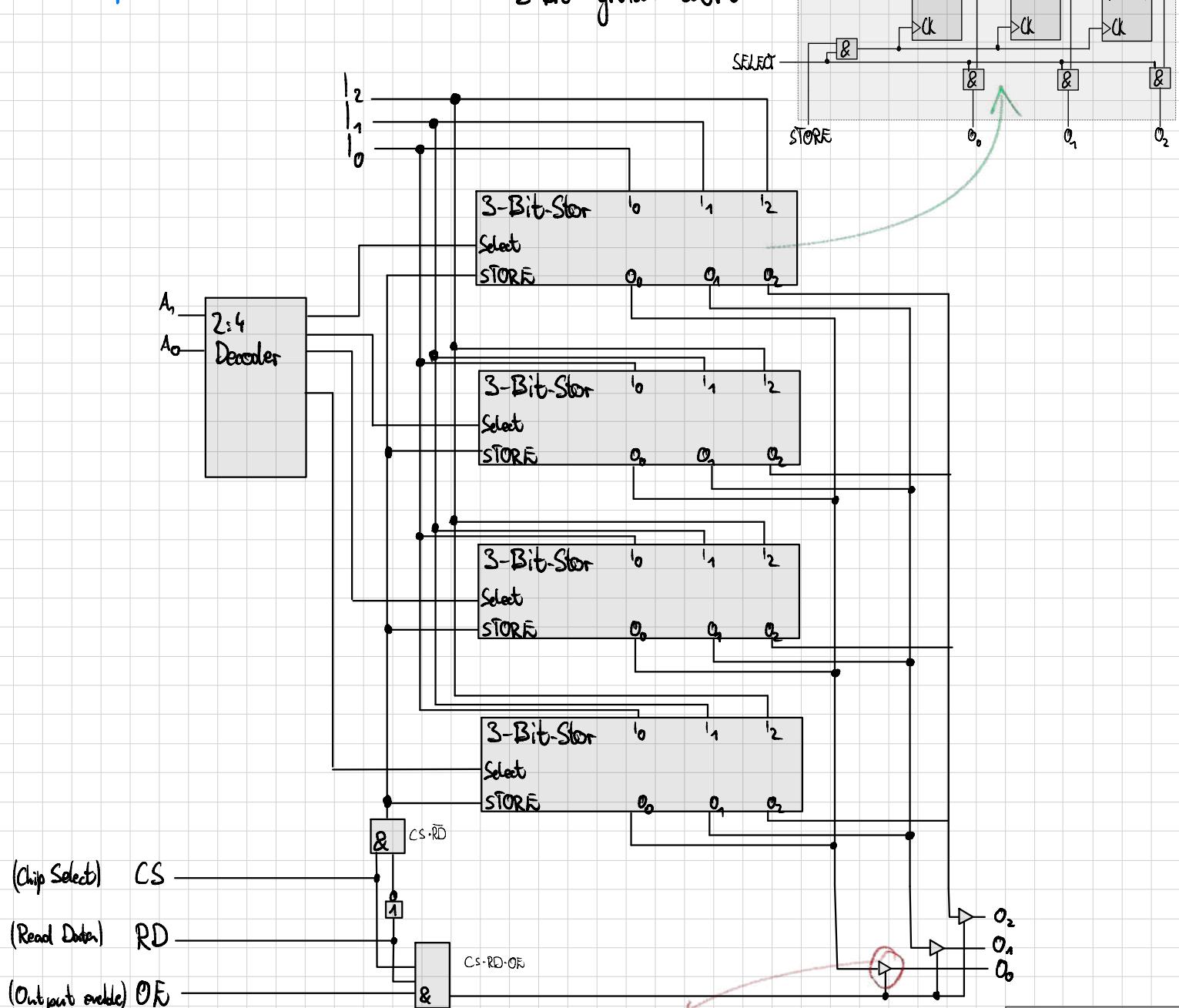
→ Speicher werden aus elementaren Speicherzellen aufgebaut



SELECT	R/W	in	Q	OUT
0	0/1	0/1	q	0 (deaktiviert)
1	1	0/1	q	q (q wird ausgelesen)
1	0	A	A	0 (A wird gespeichert)

## RAM Speicher aus BCs

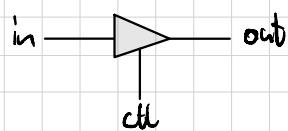
Speicherkomponente für ein  
3-Bit großes Wort



Ist wichtig, da bei einem AND-Gatter zum ein- und ausschalten, beim ausgeschalteten Zustand  $0 = 0$  wäre.

Lukas Mensch

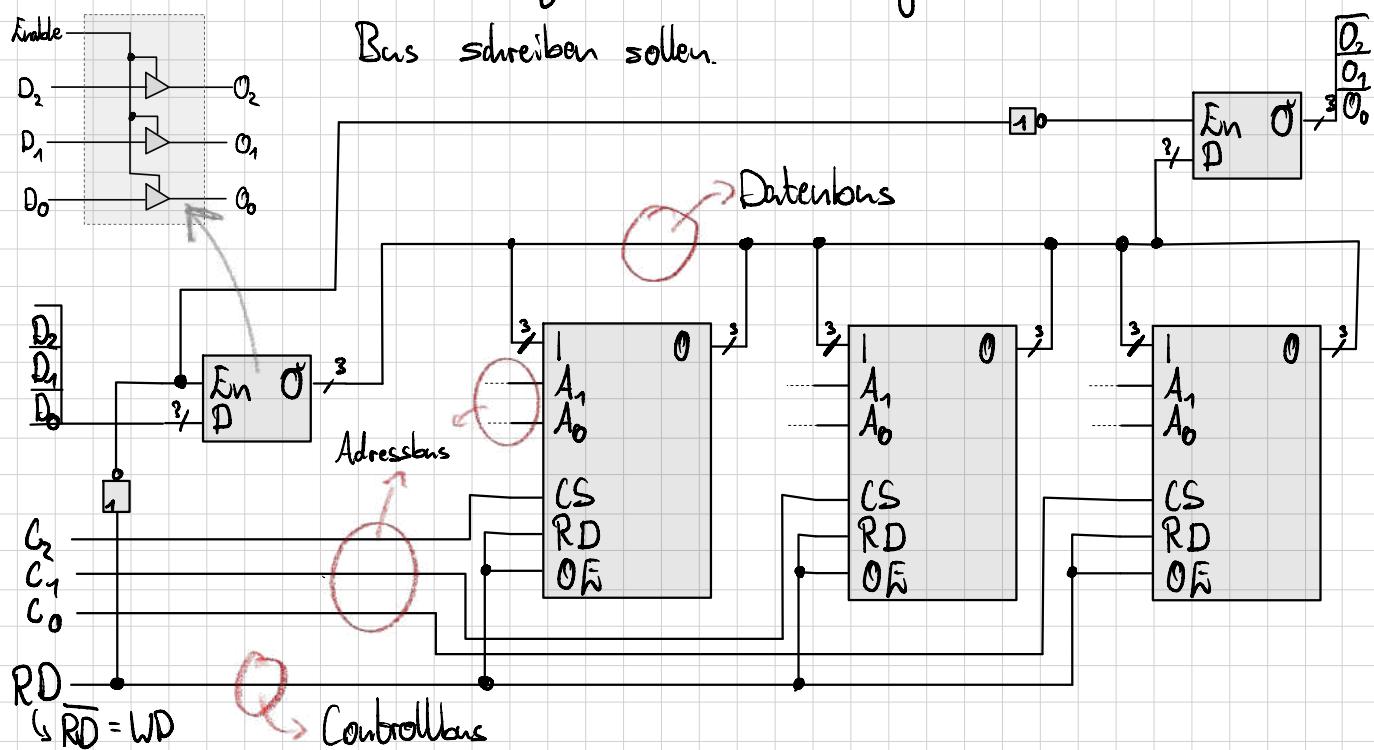
## Tristate Puffer



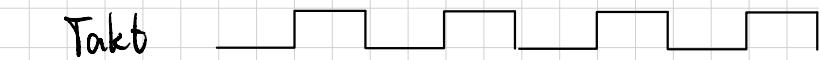
Dienen dazu elektronische Verbindungen herzustellen

- oder zu trennen:
  - $\text{ctl} = 1$  verbunden:  $\text{out}$  kann 0 o. 1 sein
  - $\text{ctl} = 0$  getrennt:  $\text{out}$  ist weder 0 noch 1

→ Wird benötigt wenn mehrere Schaltglieder auf einem Bus schreiben sollen.



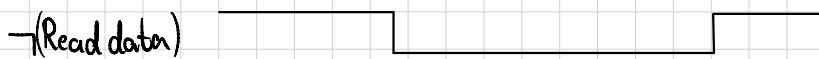
## Prozessor BUS



Wird der Pegel bei Readdata



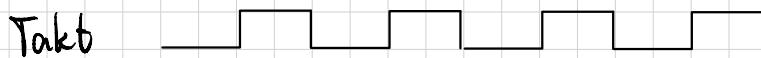
auf niedrig gesetzt, werden die



Daten beim nächsten Takt über



den Datenbus gesendet



Wird der Pegel bei Write



auf niedrig gesetzt, werden die



Daten beim nächsten Takt über

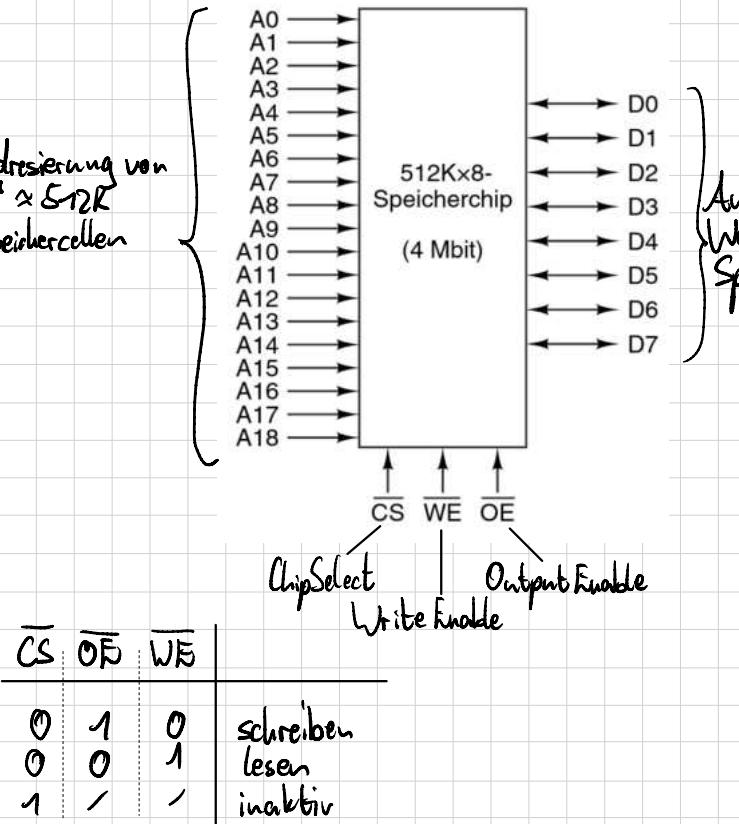


den Datenbus gesendet

⇒ Des Weiteren gibt es auch ein  $\neg(\text{Ready})$ , der darauf hinweist wann gelesen/geschrieben werden kann.

# SRAM (static random-access memory)

Adressierung von  
 $2^{18} \approx 512K$   
Speicherzellen



hese

Adr.

CS

WE

OE

Data

Schreibe

Adr.

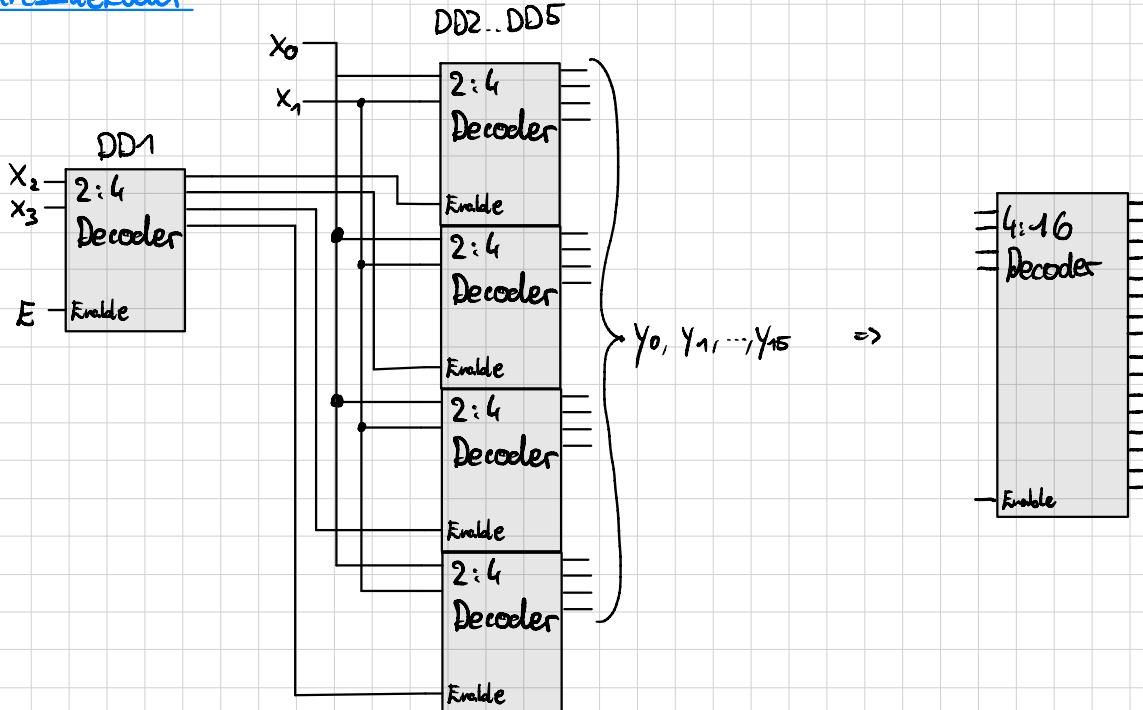
CS

WE

OE

Data

## Adressdekoder



Der erste Dekoder DD1 erlaubt die nächsten Dekoder DD2 - DD5.

## Zustandsautomaten

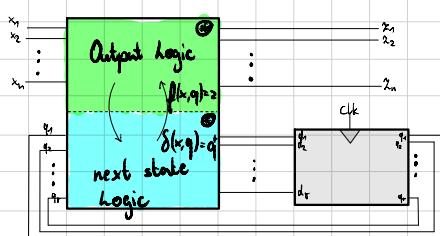
Entgegengesatz von einer normalen logischen Funktion  $f(x_1, \dots, x_n) = (z_1, \dots, z_n)$  im Schaltkreis, haben wir in einem Schaltwerk noch Zustände  $q_1, \dots, q_r$ , welche beispielsweise durch Register gegeben sind.

$$\Rightarrow f(x_1, \dots, x_n, q_1, \dots, q_r) = (z_1, \dots, z_n)$$

Des Weiteren wird bei jedem Takt ein neuer Zustand  $q_1^+, \dots, q_r^+$  berechnet

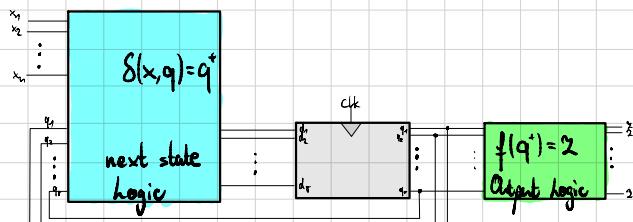
$$\Rightarrow \delta(x_1, \dots, x_n, q_1, \dots, q_r) = (q_1^+, \dots, q_r^+)$$

- Mealy-Automat



Ausgänge hängen von Zustand und Eingangssignal ab

- Moore-Automat



Ausgänge hängen nur von den Zustand ab.

Beispiel Fußgängerampel (Moore-Automat)

Eine Fußgängerampel soll bei Bedarf auf Rot schalten und wieder bei Bedarf auf Grün schalten;

Geg.:

Eingabe s:

- 1,
- 0

Zustände  $q_1, q_0$ :

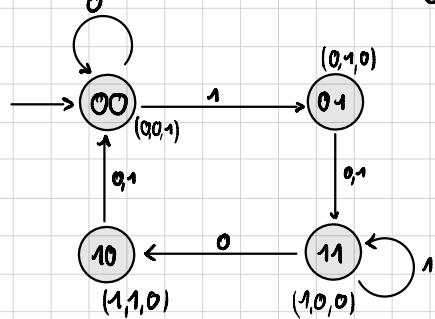
- 00 = grün,
- 01 = gelb,
- 10 = gelb-rot,
- 11 = rot

Ausgabe (R, Y, G):

- (00, 1)
- (0, 1, 0)
- (1, 1, 0)
- (1, 0, 0)

1. Schritt  
codieren der Zustände

2. Schritt: Ablauf mit Zuständen und Übergängen modellieren



3. Schritt: Schaltfunktion aufstellen und minimieren

- Output-logic: (wie soll ein Zustand repräsentiert werden)

$q_1 q_0$	R	Y	G
0 0	0 0	1	
0 1	0 1	0	
1 0	1 1	0	
1 1	1 0	0	

$R(q_1, q_0) = q_1$

$\Rightarrow Y(q_1, q_0) = q_0 \oplus q_1$

$G(q_1, q_0) = \bar{q}_1 \cdot \bar{q}_0 = \overline{q_1 + q_0}$

- next-state-logic: (welcher Zustand  $q$  kommt mit  $s$  zu  $d$ )

$s q_1 q_0$	$d_1 d_0$	Für $d_1(s, q_1, q_0)$ :
0 0 0	0 0	
0 0 1	1 1	
0 1 0	0 0	
0 1 1	1 0	
1 0 0	0 1	
1 0 1	1 1	
1 1 0	0 0	
1 1 1	1 1	

$d_1(s, q_1, q_0) = q_0$

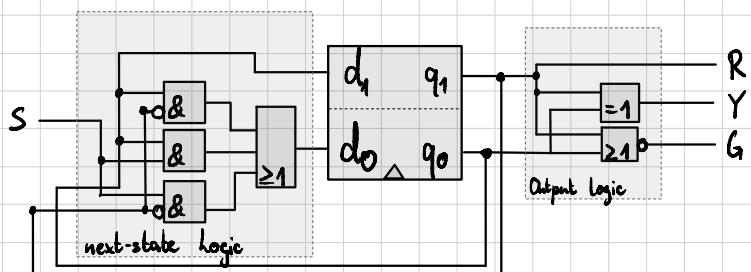
  

$s$	$q_1$	$q_0$	$d_1$	$d_0$	
1	1	0	0	0	
1	1	0	1	0	$\Rightarrow d_1(s, q_1, q_0) = q_0$
1	0	0	0	1	
0	1	0	0	1	
0	1	1	1	1	
1	0	1	1	0	
1	1	1	1	0	

Für  $d_0(s, q_1, q_0)$ :

$d_0(s, q_1, q_0) = q_0 \bar{q}_1 + s q_0 + s \bar{q}_1$

4. Schritt: Schaltfunktion realisieren

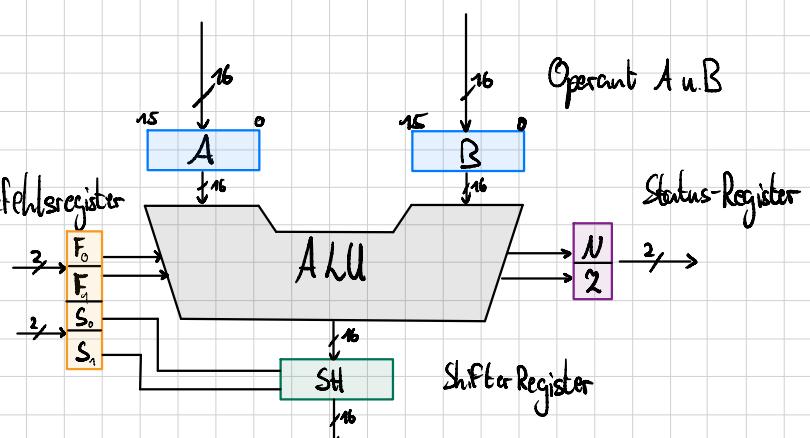


# Vom Rechenwerk zum Computer

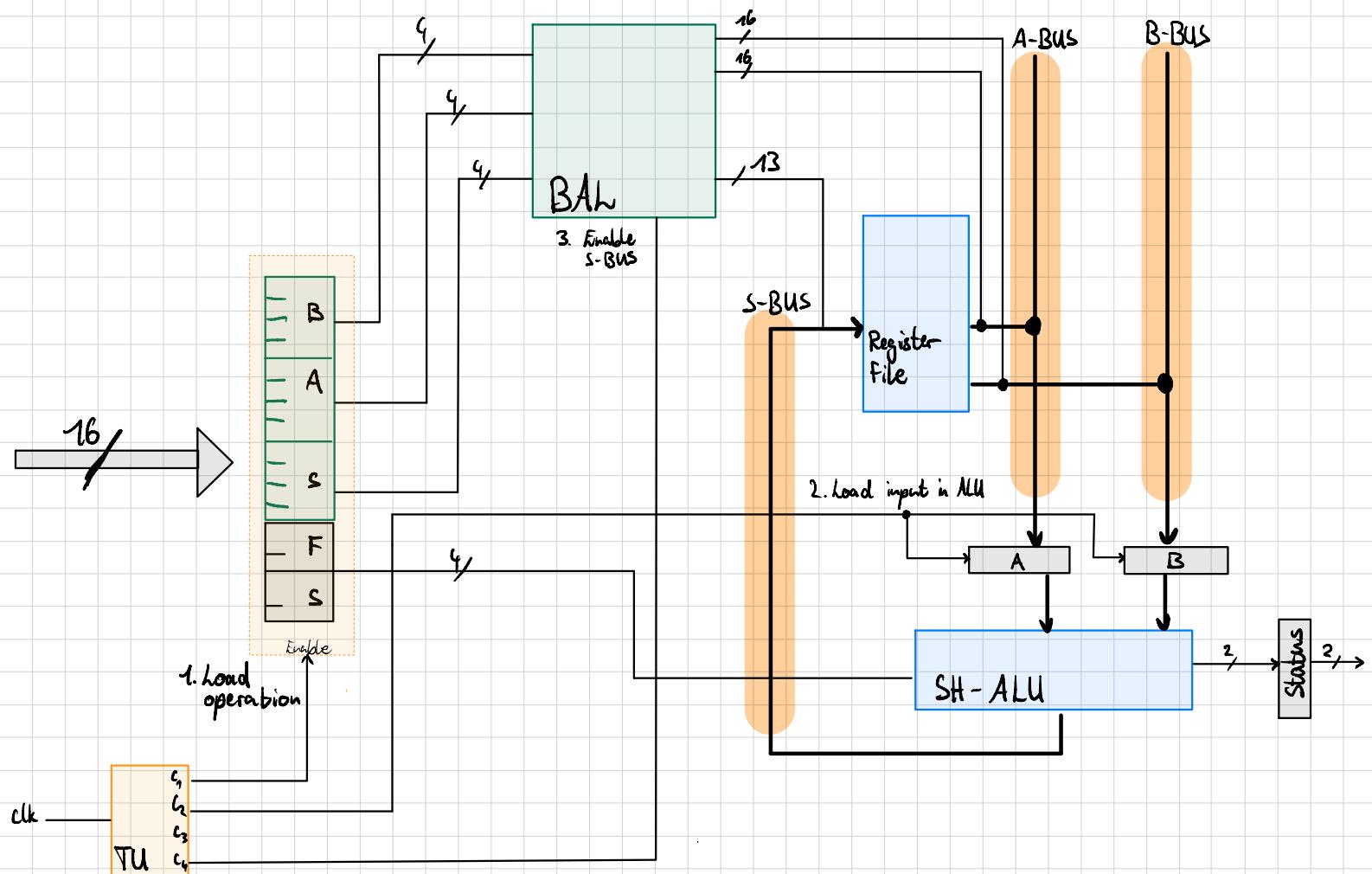
## Aufbau einer ALU

„Für eine gleichzeitige Operation werden die Schnittstellen gepuffert.“

„Des Weiteren wird der Ausgang mit einem Schieberegister (für Multi. oder Div) erweitert.“

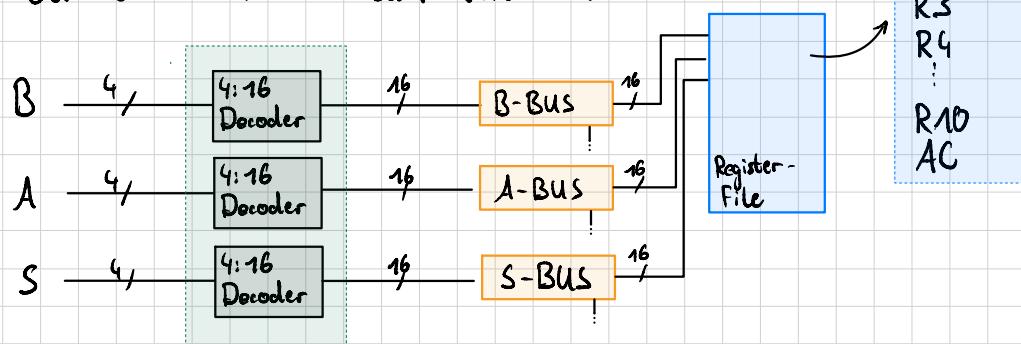


## Rechenwerk mit zeitlicher Steuerung (CPU)



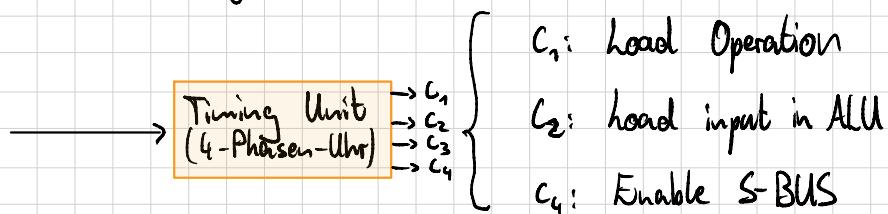
## - Die Bus Arbitration Logic (BAL)

↳ Zur Auswahl der Register wird für jeden der 3 BUSSE ein 4:16 Decoder verwendet



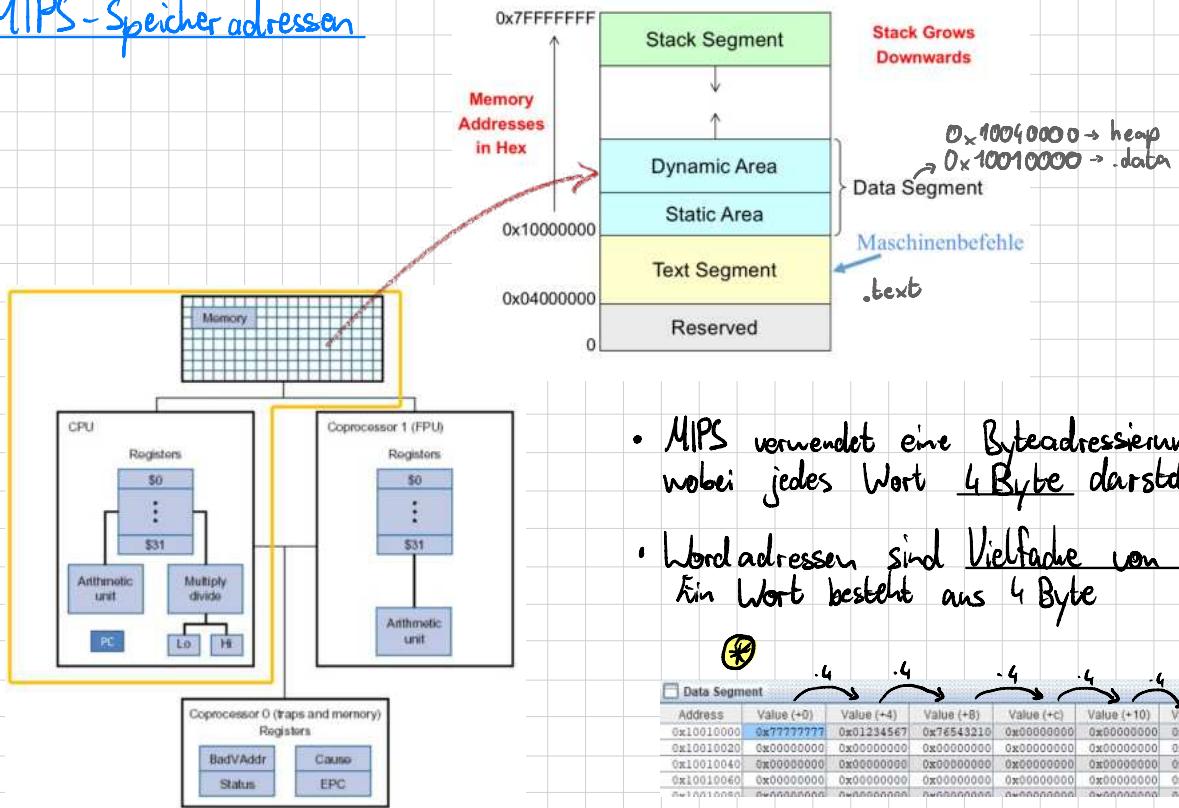
## - Zeitliche Koordination (Timing Unit)

↳ regelt die zeitliche Reihenfolge indem die Komponenten der CPU nach und nach angesteuert werden



# Maschinensprache

## MIPS-Speicheradressen



- MIPS verwendet eine Byteadressierung, wobei jedes Wort 4 Byte darstellt

- Wordadressen sind Vielfache von 4. Ein Wort besteht aus 4 Byte

\$

Address	Value (+0)	Value (-4)	Value (+8)	Value (+C)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x77777777	0x01234567	0x7E543210	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

## Datentransfer-Befehle

# Register kopieren

move \$t1, \$2 # \$t1 = \$t1

# Konstanten in Register laden

li \$t1, 42 # \$t1 = 42

li \$t1, 'k' # \$t1 = 0x6B

# Adressen in Register laden

la \$t1, var # \$t1 = var\_addr // Adresse von var

# Daten vom Speicher lesen

lw \$t1, 4(\$t2) # \$t1 = Memory[\$t2 + 4]  
 offset      Speicheradresse

lw \$t1, var + 1 # \$t1 = Memory[var + 1]

# Daten auf den Speicher speichern

sw \$t1, 4(\$t2) # Memory[\$t2+4] = \$t1

## Arithmetik und Logik Befehle

add \$t1, \$t2, \$t3 # \$t1 = \$t2 + \$t3

sub \$t1, \$t2, \$t3 # \$t1 = \$t2 - \$t3

and \$t1, \$t2, \$t3 # \$t1 = \$t2 & \$t3

or \$t1, \$t2, \$t3 # \$t1 = \$t2 | \$t3

# set if equal

seq \$t1, \$t2, \$t3 # \$t1 = (\$t2 == \$t3) ? 1 : 0

# set if less than

sLt \$t1, \$t2, \$t3 # \$t1 = (\$t2 < \$t3) ? 1 : 0

# set if less than or equal

sLe \$t1, \$t2, \$t3 # \$t1 = (\$t2 <= \$t3) ? 1 : 0

# Mit Konstanten

addi \$t1, \$t2, 4 # \$t1 = \$t2 + 4

subi \$t1, \$t2, -15 # \$t1 = \$t2 - 15

andi \$t1, \$t2, 0x00FF # \$t1 = \$t2 & 0x00FF

sLti \$t1, \$t2, 42 # \$t1 = (\$t2 < 42) ? 1 : 0

# Mult und Div

mult \$t1, \$t2 # hi,lo = \$t1 \* \$t2

mflo \$t3 # \$t3 = lo (Ergebnis)

mfhi \$t4 # \$t4 = hi (Überschuss)

mul \$t0, \$t1, \$t2 # hi,lo = \$t1 \* \$t2; \$t0 = lo

div \$t1, \$t2 # lo = \$t1 / \$t2; hi = \$t1 % \$t2

## Branches und Jumps

### # Bedingter Sprung

b eq	\$b1, \$b2, 100	# if (\$1 == \$2) goto pc+4+100
bgtz	\$b1, 100	# if (\$1 > 0) goto pc+4+100
bgez	\$b1, 100	# if (\$1 >= 0) goto pc+4+100

### # Unbedingter Sprung

j	\$t1, 1000	# goto 1000
jr	\$t1, \$t0	# goto \$t0
jal	\$b1, 1000	# \$ra = pc+4 ; goto 1000

Beispiel:

if ( i == j ) then	main:	bne	\$s3, \$s4, Else	# if (\$s3 != \$s4) goto Else
F = g + h;		add	\$s0, \$s1, \$s2	
else	Else:	j	Exit	# goto Exit
F = g - h;		sub	\$s0, \$s1, \$s2	
	Exit:			

Beispiel \$s2 < 42:

main:	slti	\$t0, \$s2, 42	# \$t0 = (\$s2 < 42) ? 1 : 0
	bne	\$t0, \$zero, Exit	
	[...]		

Exit:

# MIPS CPU

## R-, I-, J-Befehle

Register-type:



add \$t0, \$t1, \$t2

(→ Mnemonic gefolgt von 3 Registern)

↳ op und funct werden gemeinsam benutzt um eine arithmetische Funktion auszuwählen.

Immediate-type:



addi \$t0, \$t1, 100

(→ Mnemonic gefolgt von 2 Registern und einer Konstante)

Jump-type:



j 1000

(→ Mnemonic gefolgt von einer Konstante)

Decodierung der Mnemonic zur Operation Code und ggf. Funktion

[https://en.wikibooks.org/wiki/MIPS\\_Assembly/Instruction\\_Formats#Opcodes](https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats#Opcodes)

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x2
and	Bitwise AND	R	0x00	0x24
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
mult	Multiply	R	0x00	0x18
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
sub	Subtract	R	0x00	0x22
...				
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
addi	Add Immediate	I	0x08	NA
beq	Branch if Equal	I	0x04	NA
lw	Load Word	I	0x23	NA
sw	Store Word	I	0x2B	NA
...				

op: operation code  
rs: register source  
rt: register target  
rd: register destination  
shamt: shift amount  
funtct: function

# Implementieren von AhU-Operationen im R-Format

Ablauf dieser Instruktion immer gleich

1. Lese Register **rs** und **rt**
2. Führe eine AhU-Operation darauf aus
3. Schreibe das Ergebnis in das Register **rd**

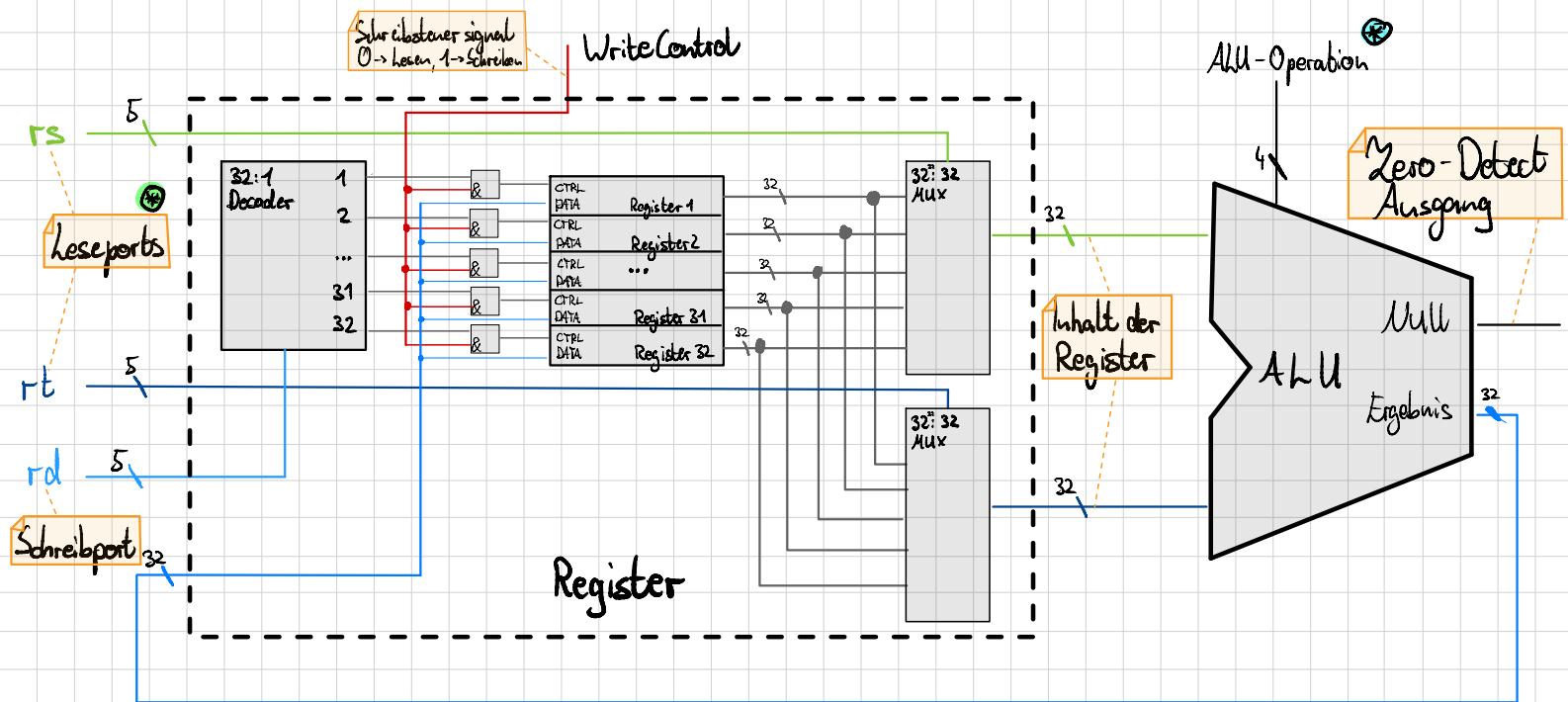
<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>func</b>
-----------	-----------	-----------	-----------	--------------	-------------

→ sind add, sub, and, or, ...

Beispiel:

add \$4, \$3, \$2

000000	00011	00010	00100	00000	100000
--------	-------	-------	-------	-------	--------

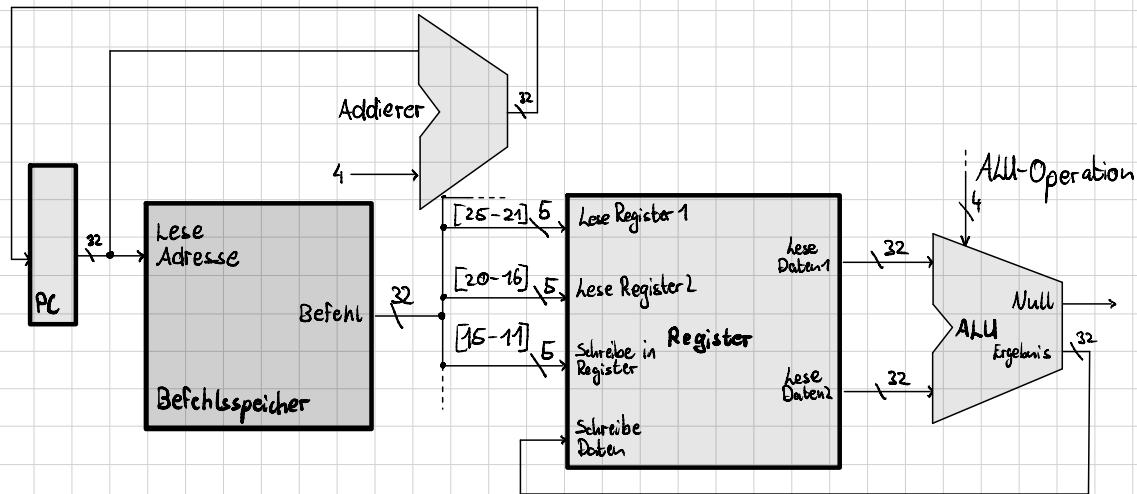


Datenwert des Ergebnis

- ↳ Der Registersatz gibt den Inhalt der Register, abhängig vom Leseregister-Eingangssignal, an. (keine weiteren Steuereingänge erforderlich)
- ↳ Bei einem Lesevorgang muss WriteControl auf 0 gesetzt sein.  
→ Es wird dabei ein Wert gelesen, welcher zuvor mit lw, li, add, ... in den Register geschrieben worden ist.
- ↳ Die AhU-Operation <sup>②</sup> setzt sich aus opb u. func zusammen  
opb immer 0 bei

R-Type Instructions

=> Vereinfacht mit Befehlszähler und -speicher



## Implementierung von Branch

Genauere Betrachtung der Branch-Instruction `beq`:

ELSE ist 4 Zeilen weiter

`bne $1, $2, ELSE`

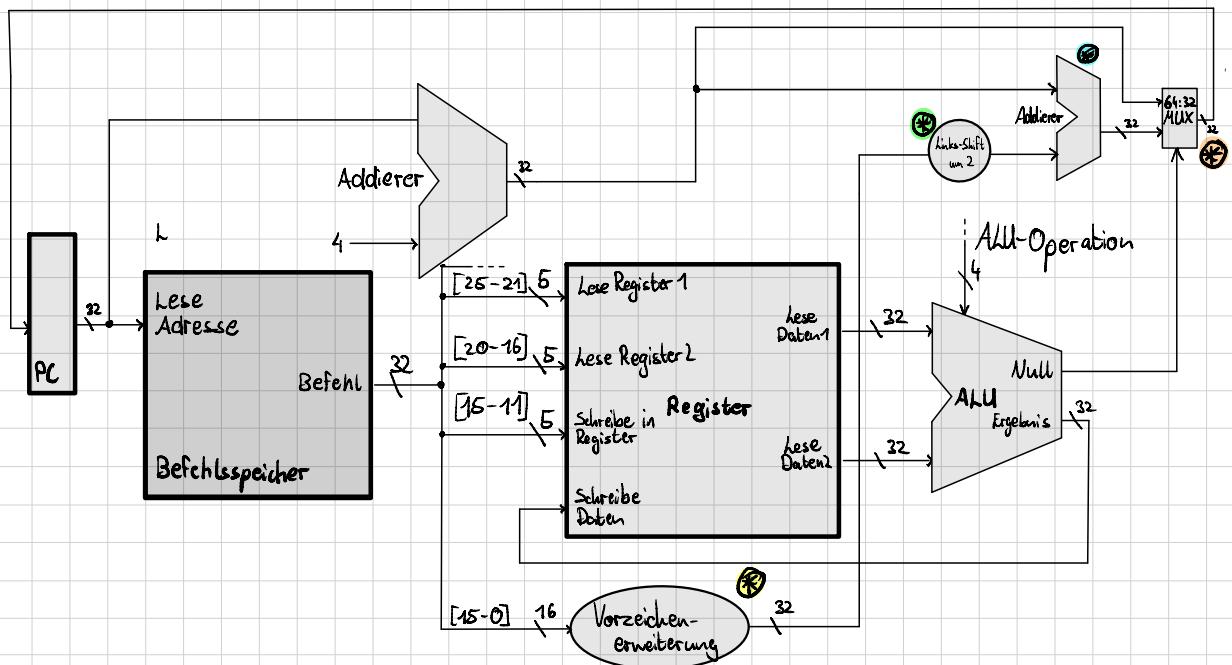
000101	00001	00010	0000000000000000100
--------	-------	-------	---------------------

Ablauf einer `beq`-Instruction

1. Ziehe `rs` von `rt` ab.
2. Vergleiche Ergebnis mit 0:
  - Ergebnis  $\neq 0$ :  
↳ nächste Instruction bei  $PC + 4$
  - Ergebnis  $= 0$ :  
↳ Zeichenverweiterung des `offset` von 16-Bits auf 32-Bit
  - ↳ Offset muss  $\downarrow$  nehmen, da Befehlszähler in 4er-Schritten zählt.
  - $offset \times 4 = offset * 4$  (Links-Shift um 2)
  - ↳ nächste Instruction ist bei  $PC + 4 + offset \times 4$

Um den Offset wie gewünscht zu verarbeiten, werden zwei neue Komponenten benötigt:  
 → Vorzeichenerweiterung   → Links-Shift um 2

Des Weiteren wird ein zusätzlicher Addierer, der den Offset auf den PC+4 addiert, sowie auch ein Multiplexor, welcher zwischen PC+4 und PC+4+Offset schaltet



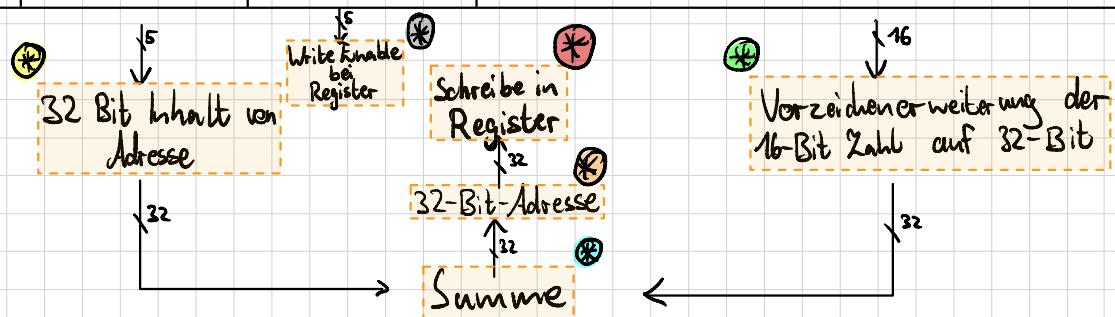
## Implementierung von Speicherbefehlen

Bei lw wird der Speicherinhalt in das Register geschrieben:

lw      \$t1, 4(\$t2)    # \$t1 = Memory[\$t2+4]

op      rs      rt      offset

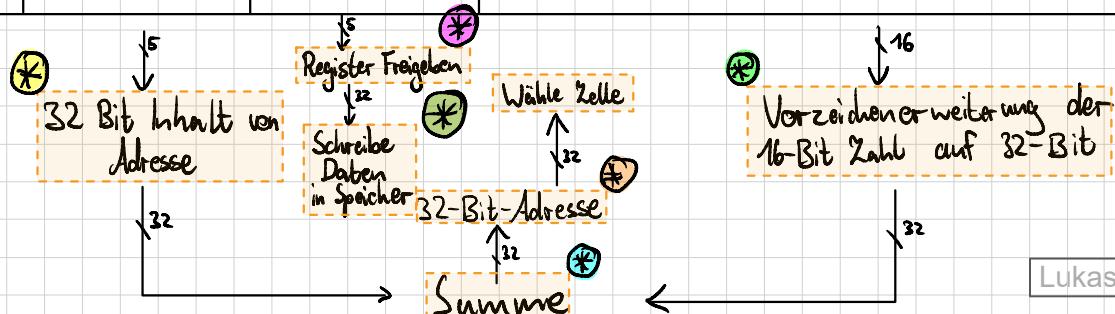
100011	01010	01001	00000000000000000100
--------	-------	-------	----------------------



Bei sw wird der Registerinhalt in den Speicher geschrieben:

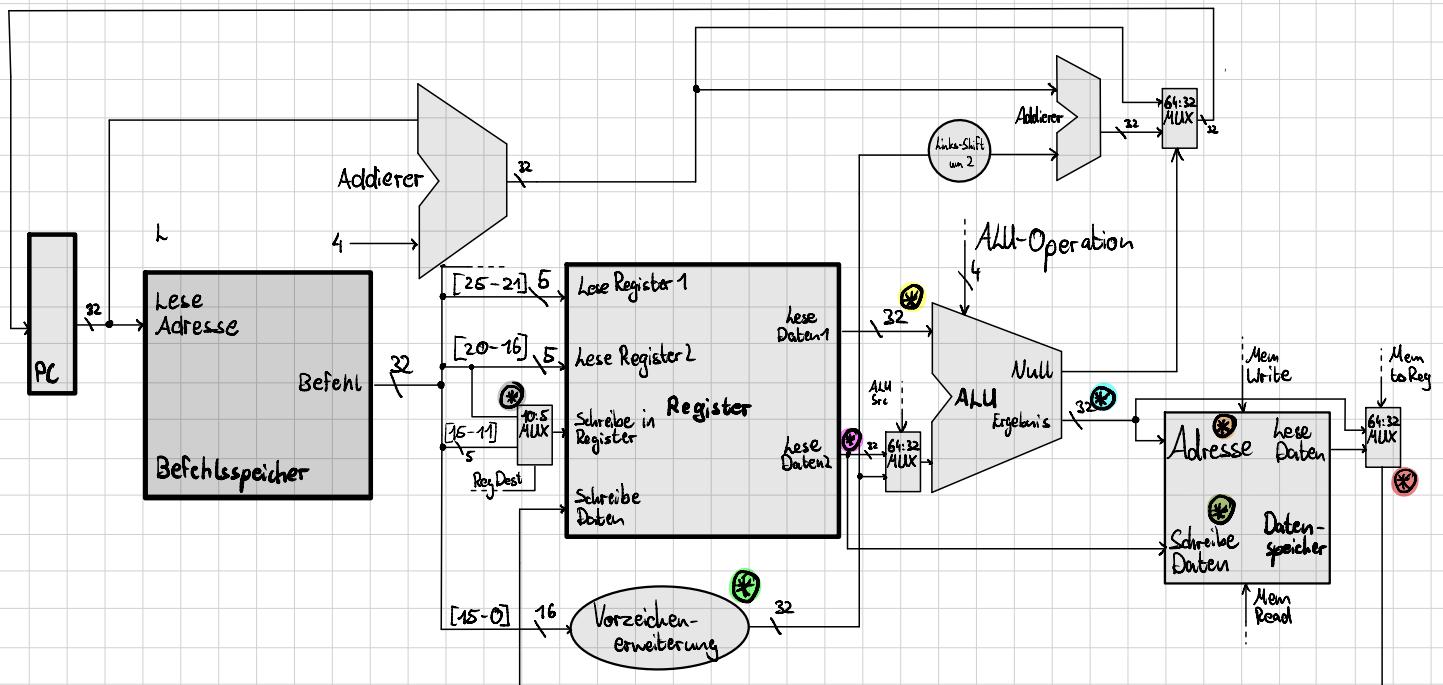
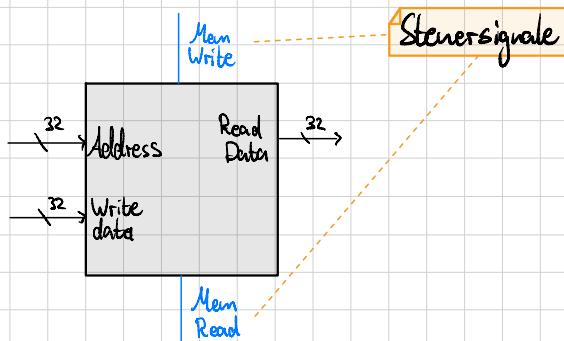
sw      \$t1, 4(\$t2)    # Memory[\$t2+4] = \$t1

101011	01010	01001	00000000000000000100
--------	-------	-------	----------------------



↳ Hierzu werden vorhandene ALU und Vorzeichenverweiterung mitbenutzt

Es wird zur Vereinfachung ein extra Speicher für die Daten verwendet, anstatt den Befehlspeicher zu verwenden.



## ALU Control

Die MIPS-ALU wird mit 4 Steuer-eingängen angesprochen, wobei sie dann mit 6 Kombinationen angesprochen wird:

Bei Load- und store-word Befehlen wird die ALU zum Berechnen der Speicheradresse durch Addition verwendet.

Bei R-Befehlen wird je nach Wert des Funkt-Feldes eine der ersten 5 Funktionen ausgeführt.

Kombination	Funktion
0 0 0 0	and
0 0 0 1	or
0 0 1 0	add
0 1 1 0	subtract
0 1 1 1	set on less than
1 1 0 0	nor

Bei Branch-on-Equal wird an der ALU eine Subtraktion durchgeführt.

⇒ Realisierung des 4-Bit breiten ALU-Steuereingang anhand einer Steureeinheit.

Nun ist entgegen:

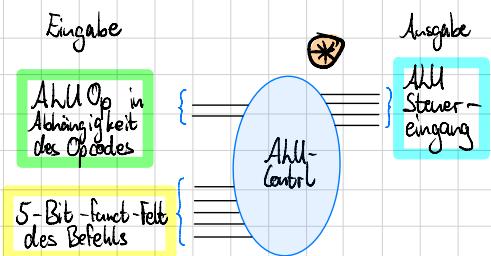
↳ 6-Bit funct-Feld bei R-Befehlen

↳ 2-Bit breites Steuerfeld

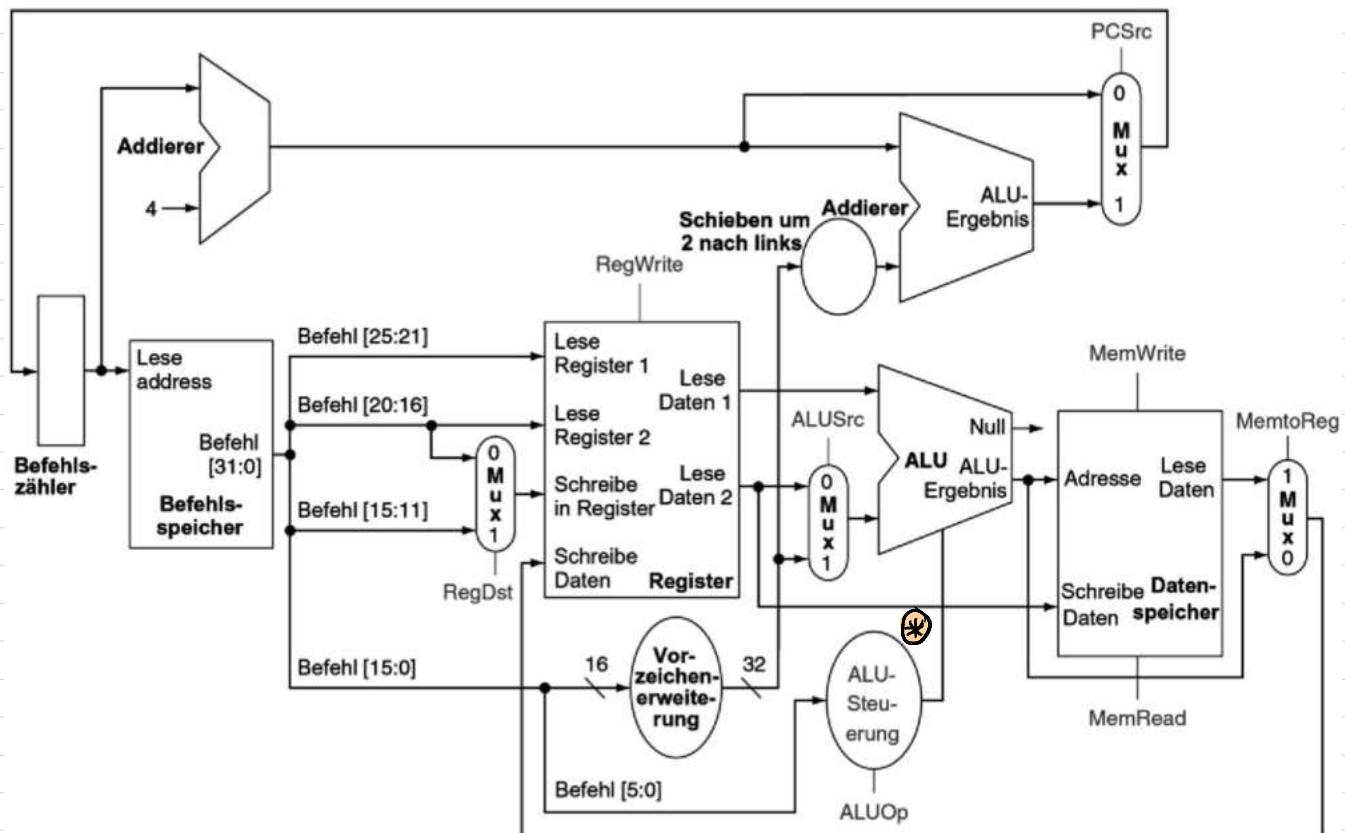
Das Steuerfeld (ALUOp) gibt an, um welche Operation es sich handelt:

- Bei 00 Addition für load- und store-word
- Bei 01 Subtraktion für den beq Befehl
- Bei 10 funct-Feld-Operation

Opcode des Befehls	ALUOp	Befehlsoperation	funct-Feld	Gewünschte ALU-Aktion	ALU-Steuereingang
LW	00	load word	XXXXXX	Addition	0010
SW	00	store word	XXXXXX	Addition	0010
Branch on equal	01	branch on equal	XXXXXX	Subtraktion	0110
R-Befehl	10	add	100000	Addition	0010
R-Befehl	10	subtract	100010	Subtraktion	0110
R-Befehl	10	and	100100	UND	0000
R-Befehl	10	or	100101	ODER	0001
R-Befehl	10	set on less than	101010	kleiner als	0111



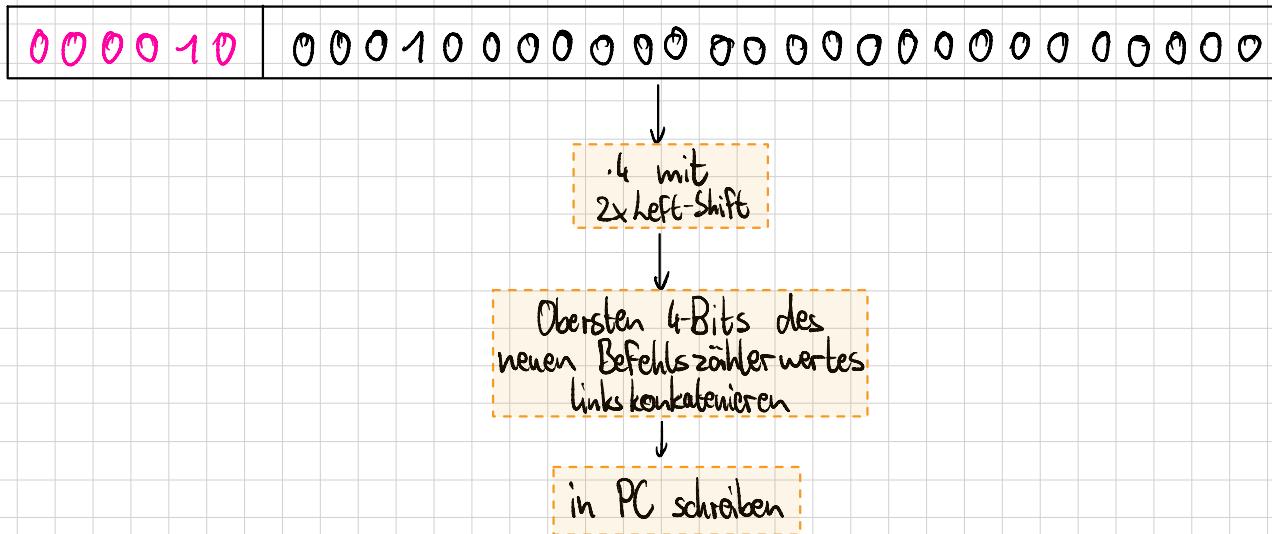
Implementiert im MIPS-Prozessor:



## J-Befehl

j main

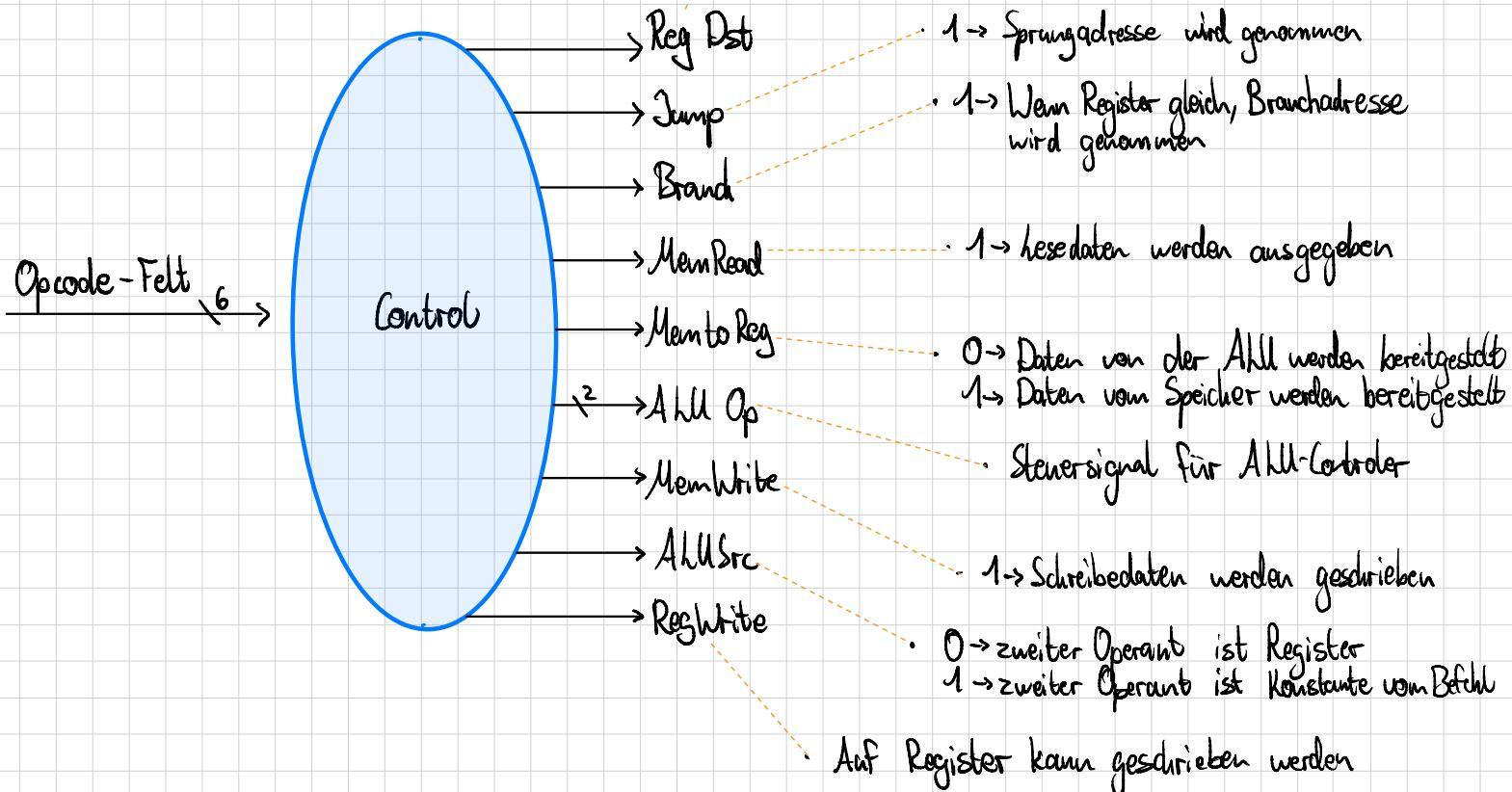
# Sprung zum gegebenen Label



## Steureinheit

Mit den 6-Bit großen Opcode-Feld wird die Steureinheit angesteuert:

- 0 → rt-Feld [20-16] ist Schreibregister
- 1 → rd-Feld [15-11] ist Schreibregister



Durch Hinzufügen der Steuerungseinheit und Implementierung des J-Befehles, erhält man schließlich den fertigen MIPS-Prozessor

