Zusammenfassung TGI

WS2021 by Stalex | Alex

Formeln

Kommutativität:	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
Distributivität:	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Verknüpfung 1/0:	$a \wedge 1 = a$	$a \vee 0 = a$
Kompl. Element:	$a \wedge !a = 0$	$a \vee !a = 1$
Assoziativität:	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
Absorption:	$(a \wedge b) \vee a = a$	$(a \vee b) \wedge a = a$
Auslöschung:	$a \wedge (b \vee !b) = a$	$a \vee (b \wedge !b) = a$
Idempotenz:	$a \wedge a = a$	$a \vee a = a$
Involutivität:	$!(!a) = a$	
Morgansche Regel:	$!(a \wedge b) = !a \vee !b$	$!(a \vee b) = !a \wedge !b$

- Amdahl's Law:
  - $S = \frac{ET_{alt}}{ET_{neu}} = \frac{ET_{alt}}{ET_{alt} * \left((1-F_e) + \frac{F_e}{S_e}\right)} = \frac{1}{(1-F_e)+F_e/S_e}$

Speedup durch Parallelisierung

$$S = \frac{n}{1+(n-1)f}$$

- $x = (-1)^V * M * B^{\pm E}$
- V=Vorzeichen, M=Mantisse, B=Basis, E=Exponent
- Bildung Single Precision 32 Bit:
  - **Vorzeichen** 1 Bit: 1 → negativ, 0 → positiv
  - **Exponent** e 8 Bit: Exzessdarstellung e = E+q, **q=127**, E=
  - **Mantisse** 23 Bit: genau eine Vorkommastelle ungleich 0 → Maschinenepsilon $\epsilon = 2^{-23}$

- $x = (-1)^V * M * B^{\pm E}$
- V=Vorzeichen, M=Mantisse, B=Basis, E=Exponent
- Bildung Single Precision 64 Bit:
  - **Vorzeichen** 1 Bit: 1 → negativ, 0 → positiv
  - **Exponent** e 11 Bit: Exzessdarstellung – e = E+q, **q=1023**
  - **Mantisse** 52 Bit: genau eine Vorkommastelle ungleich 0 → Maschinenepsilon $\epsilon = 2^{-52}$

CPU-Zeit

- $t_{CPU} = IC \cdot t_{cc} \cdot CPI$
- $t_{CPU} = \frac{IC \cdot CPI}{f}$

Formeln2

Caching

Memory Stall Cycles

$$MSC = Anzahl\ der\ Misses \cdot MP$$
$$= IC \cdot MPI \cdot MP = IC \cdot \frac{Misses}{Instruktion} \cdot MP$$
$$= IC \cdot \frac{Speicherzugriff}{Instruktion} \cdot \frac{Misses}{Speicherzugriff} \cdot MP$$

IC = Instruction Count, MPI = Misses per Instruction, MP = Miss Penalty

Misses Per Instruction (MPI)

$$MPI = \frac{Misses}{Instruktion} \qquad \frac{Speicherzugriffe}{Instruktion} \cdot MR$$

MR = Miss Rate

Miss Rate (MR)

$$MR = \frac{Misses}{Speicherzugriff}$$

Mittlere Speicherzugriffszeit

$$\bar{t}_{SZ} = t_H + MR \cdot MP$$

t_H = Hit Time, MR = Miss Rate, MP = Miss Penalty

$\bar{t}_{SZ}$ für mehrstufige Caches

$$\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot MP_{L1}$$
$$MP_{L1} = t_{H,L2} + MR_{L2} \cdot MP_{L2}$$
$$\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot (t_{H,L2} + MR_{L2} \cdot MP_{L2})$$

t_H = Hit Time, MR = Miss Rate, MP = Miss Penalty

Average Memory Stalls Per Instruction (AMSPI)

$$AMSPI = MPI_{L1} \cdot t_{H,L2} + MPI_{L2} \cdot MP_{L2}$$

MPI = Misses per Instruction, MP = Miss Penalty

Festplatten

Seek TIme

$$T_S = m \cdot n + s$$

m = Zeit pro Spur, n = Anzahl der überquerten Spuren, s = Startup-Zeit

Rotationsverzögerung

$$T_r = \frac{1}{2r}$$

r = Umdrehungen pro Minute (RPM)

Transferzeit

$$T = \frac{b}{r \cdot N}$$

b = Anzahl der Bytes, N = Anzahl an Bytes auf der Spur

Gesamte Transferzeit

$$T_a = T_s + T_r + T = T_S + \frac{1}{2r} + \frac{b}{rN}$$

T_s = Seek Time, T_r = Rotationsverzögerung, T = Transferzeit, r = RPM, N = Anzahl an Bytes auf der Spur

AMSPI → geht davon aus keine Miss time bei L1 Cache

Zahlendarstellung

- Dezimal – Binär:
  - dividiert durch 2 bis 0 erreicht wird, Reste = Binärzahl
  - Bruchzahlen: nur Näherungswerte: $0.5 = 0.1_2, 0.25 = 0.01_2, 0.125 = 0.001_2$
- Binär – Dezimal:
  - Potenzen von 2 summieren
- Dezimal – Hexadezimal:
  - dividiert durch 16 bis 0 erreicht wird, Reste = Hexadezimalzahl
- Hexadezimal – Dezimal:
  - Potenzen von 16 summieren

- Von Binär zu Oktal:
  - 3er Gruppen weil $2^3 = 8$
  - 0 111 101 110 100 011 = 75643_8
  - Von Oktal zu Binär: jede Ziffer ist Binär eine 3er Gruppe
- Von Binär zu Hexadezimal:
  - 4er Gruppen weil $2^4 = 16$
  - $0111\ 1011\ 1010\ 0011 = 7BA3_{16}$
  - Von Hex auf Binär: jede Ziffer ist eine Binär 4er Gruppe

Sowas kommt!

$\overline{B-1}$ Einerkompliment

$\overline{B}$ Zweierkompliment

Zahlensystem - Stellenwertsystem

Stellenwertsysteme

Binärsystem = Dualsystem → nur zwei verschiedene Zustände

Oktalsystem = zur Basis 8 wichtig da Byte 8 bit sind

Hexadezimalsystem = zur Basis 16 →

10 =	A
11 =	B
12 =	C
13 =	D
14 =	E
15 =	F

Zahlensystem - Konvertierung

Konvertierung

Basis 10 → 2

- (Ganz)Zahl so lange durch 2 dividieren, bis 0 erreicht
- Jeweilige Reste (rückwärts gelesen (von unten nach oben)) ergeben Binärzahl

131 : 2 = 65 1
65 : 2 = 32 1
32 : 2 = 16 0
16 : 2 = 8 0
8 : 2 = 4 0
4 : 2 = 2 0
2 : 2 = 1 0
0 : 2 = 0 1

↑ unten nach oben

Basis 2 → 10 ⇒ Werte über Zahlen schreiben am besten

- Potenzen von 2 summieren, entsprechen den gesetzten Bits der Binärzahl

Basis 10 → 16

- Zahl so lange durch 16 dividieren, bis 0 erreicht
- Jeweilige Reste ergeben Hexadezimalzahl
- Achtung: falls Rest > 9 → durch entspr. Buchstabenersetzen!

Basis 16 → 10

- Potenzen von 16 summieren
- Siehe Folien zu Stellenwertsysteme

Methode für gebrochene Zahlen

Basis 10 → 2

- Ganzzahlen (Zahlen links vom Komma) trennen → "normale" Umrechnung
- Ziffernfolge rechts vom Komma → wert z.b 0,2 * 2 immer und wenn über 1 → 1 auf der Seite notieren und -1 das Ergebnis und weitermachen bis 1 rauskommt und daher 0 → diese Kette dann von

0,4 · 2 = 0,8 0
0,8 · 2 = 1,6 1
0,6 · 2 = 1,2 1
0,2 · 2 = 0,4 0
0,4 · 2 = 0,8 0
0,8 · 2 = 1,6 1
0,6 · 2 = 1,2 1

↓ oben nach unten

Konvertierung zwischen Basen 2, 8, 16

Abkürzung

- Basis 2 ⇒ Basis 8: Dreiergruppen bilden ($2^3 = 8$)

0 111 101 110 100 011 → 7 5 6 4 3_8

- Basis 2 ⇒ Basis 16: Vierergruppen bilden ($2^4 = 16$)

wenn nicht genug für gruppe leer = 0

0111 1011 1010 0011 → 7 B A 3_{16}

10 .. A
11 .. B
12 .. C
13 .. D
14 .. E
15 .. F

Geht natürlich auch in die andere Richtung!

BSP REP 1-1

1. **Konvertierung (Rechenschritte angeben bzw. erklären!)**

   (a) **Vervollständigen Sie die folgenden Tabelle.**

   Hinweis: In der ersten Zeile gehen Sie von der Dezimalzahl d_{10} *aus und wandeln diese in Binär, Hexadezimal und Oktal um. In der zweiten Zeile starten Sie mit der gegebenen Binärzahl und wandeln diese in Dezimal, Hexadezimal und Oktal um usw.*

	Dezimal	Binär	Hexadezimal	Oktal
1)	d_{10} 1201_{10}	10010110001_2	$FB1_{16}$	2261_8
2)	31_{10}	11111_2	$1F_{16}$	37_8
3)	64 083	1111101001010011	h_{16} $FA53_{16}$	175123_8
4)	63_{10}	$111\ 111_2$	$3F_{16}$	77_8

$8 \to 2$ $2 \to 16$ $2 \to 10$

4) $77_8 = 111\ 111$ (7, 7) | 00111111 → 3 15 → 3F | 32 16 8 4 2 1: 111 111 = 63

a)

1234	2	= 0
617	2	= 1
308	2	= 0
154	2	= 0
77	2	= 1
38	2	= 0
19	2	= 1
9	2	= 1
4	2	= 0
2	2	= 0
1	2	= 1

10011010010_2] binär

$10\ 011\ 010\ 010_2$ → 3 3 2 2 → 3322_8] 8er

$100\ 1101\ 0010_2$ → 4 13 2 → 4D2 (A B C D = 10 11 12 13)] 16er

Bit und Byte

Bit und Byte

Byte = 8 bit(Oktett)

msb und lsb

- msb = most significant **bit** = höchstwertiges bit
- lsb = least significant **bit** = niedrigstwertiges bit

Dualsystem vs. Dezimalsystem

$10011_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

$17564_{10} = 1 \cdot 10^4 + 7 \cdot 10^3 + 5 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0$

→ lsb => kann 0 sein

Vielfache von Bit

Dezimalpräfix			Binärpräfix		
Name	**Symbol**	**Wert**	**Name**	**Symbol**	**Wert**
Kilobit	kbit	10^3	Kibibit	Kibit	2^{10}
Megabit	Mbit	10^6	Mebibit	Mibit	2^{20}
Gigabit	Gbit	10^9	Gibibit	Gibit	2^{30}

Vielfache von Byte

Dezimalpräfix			Binärpräfix		
Name	**Sym.**	**Wert**	**Name**	**Sym.**	**Wert**
Kilobyte	kB	10^3 Byte	Kibibyte	KiB	2^{10} Byte
Megabyte	MB	10^6 Byte	Mebibyte	MiB	2^{20} Byte
Gigabyte	GB	10^9 Byte	Gibibyte	GiB	2^{30} Byte

Rechnen-Addition

- *Addition:*

Additionstabelle

A	+	B	=	Übertrag	Summe(Σ)
0	+	0	=	0	0
0	+	1	=	0	1
1	+	0	=	0	1
1	+	1	=	1	0

Übertrag (engl. carry) bei nächsthöherer Stelle berücksichtigen!

Carry Bit → Ergebnis von zwei 8 Bit zahlen addition z.b kann auch 9 bit lang sein also vorne eine 1, welche in einem Spezialregister gespeichert wird.

Rechnen- Subtraktion

1.) Komplementbildung des Subtrahenden

NOT	0	0	0	1	1	0	1	0_2
=	1	1	1	0	0	1	0	1_2
+								1_2
	1	1	1	0	0	1	1	0_2

2.) Addition

Übertrag vorhanden ⇒ Ergebnis positiv!

	0	0	1	1	0	1	1	1_2	
+	1	1	1	0	0	1	1	0_2	
1	0	0	0	1	1	1	0	1_2	$= 29_{10}$

Als erstes bildet man von der Zahl das Kompliment also immer 0 → 1 und 1 →0, zu diesem wird 1 dazu addiert und diese Zahl wird dann zu der anderen zahl addiert und man bekommt das Ergebnis.

- **Falls kein Übertrag vorhanden: Rückkomplementieren**

⇒ Ergebnis ist negativ

- **Falls Übertrag vorhanden, diesen streichen**

⇒ Ergebnis ist positiv

➔ Heist bei einer positiven Zahl wird die erste 1 (Übertrag) einfach gelöscht

➔ Falls die Zahl negativ ist wie folgt vorgehen →

3.) Rückkomplementbildung

KEIN Übertrag ⇒ Ergebnis negativ!

Zwischenergebnis:	1	1	1	0	0	0	1	1_2
Stellenkomplement:	0	0	0	1	1	1	0	0_2
+1	0	0	0	0	0	0	0	1_2
=	0	0	0	1	1	1	0	1_2

- $11101_2 = 29_{10}$ ⇒ Minus davorschreiben: -29_{10}

Rechnen - *&/

Multiplikation mit 2^N

Verschieben nach links (bei Big Endian)

Verschieben einer Zahl um S Stellen nach **links** entspricht **Multiplikation** mit B^S

Beispiel: 01100111_2 = 8bit unsigned integer

Stelle	7	6	5	4	3	2	1	0
Vorher	0	1	1	0	0	1	1	1
Nachher	1	1	0	0	1	1	1	0

Vorsicht vor Überläufen (Overflows)!

Division durch 2^N

Verschieben nach rechts (bei Big Endian)

Verschieben einer Zahl um S Stellen nach **rechts** entspricht **Division** durch B^S

Beispiel: 1010_2 = 4bit unsigned integer

Stelle	3	2	1	0
Vorher	1	0	1	0
Nachher	0	1	0	1

Vorsicht vor Unterläufen (Underflows)!

BSP REP 1-2

2. **Addition und Subtraktion im Binärsystem**

Gegeben seien die beiden unsigned 8 Bit Binärzahlen $a = 0101\,1010_2$ und z, wobei Sie für z die Binärzahl aus Aufgabe 1, Zeile 1 verwenden sollen, d.h. die Zahl d_{10} als Binärzahl.

Wichtig: Stellen Sie z als 8-Bit Binärzahl dar – evtl. müssen Sie vorne 0er hinzufügen bzw. die führenden Stellen abschneiden – und setzen Sie das msb auf 0.

Beispiel: $d_{10} = 1234 \Rightarrow$ d in binär $= 100\,1\underline{101\ 0010} \Rightarrow z = \mathbf{0}\underline{101\ 0010}$

(a) Addieren Sie a und z und achten Sie dabei auf Überläufe!

(b) Subtrahieren Sie a von z (also $z-a$). Führen Sie die Subtraktion dabei auf eine Addition zurück (Zweierkomplement!).

(c) Kontrollieren Sie Ihre Ergebnisse indem Sie a, z, und alle Ergebnisse in das Dezimalsystem umwandeln.

a)

82 z 01010010
+ 90 a + 01011010
10101100

$\bar{a}$ = 10100101 -> +1
10100110

b)

82 01010010
− 90 10100110
−8 11111000 ← kein Überlauf -> sonst falsch ⇓ negativ

↓

00000111
+ 1 ← +1 nicht vergessen
00001000
= −8

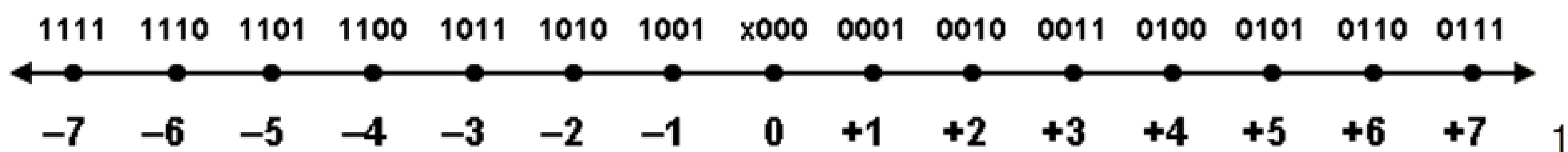
Negative Binärzahlen - Vorzeichen

1 = negativ
0 = positiv

Negative Binärzahlen

Vorzeichen und Betrag

Führendes Bit codiert Vorzeichen: (0/1 -> +/-)
Daher ist aber Zeichen belegt und man hat n-1Platz, weiters -1 weil 0 als +/-0 gespeichert wird → für den Bereich muss man weiters bedenken wieder -1 weil null zählt als Zahl in der Menge aber nicht im Bereich (0-177 z.b sind 178 Zahlen)



1

Negative Binärzahlen - Einerkompliment

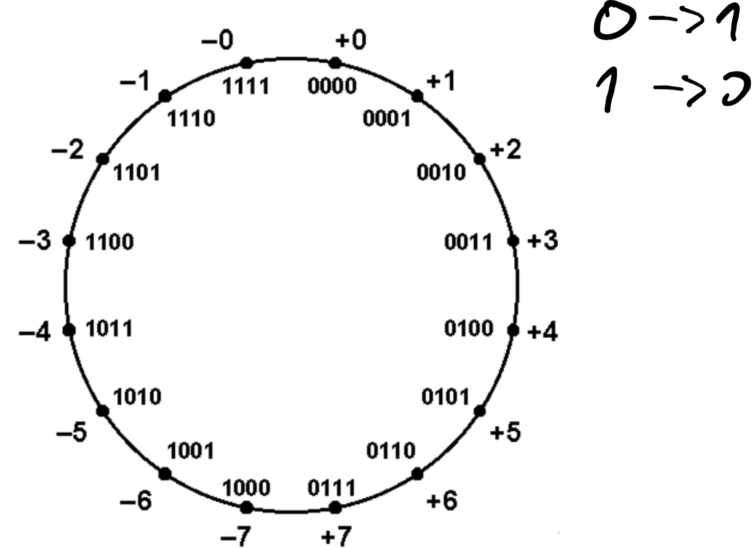
Einerkomplement

Bildung des Gegenteils der Zahl 0 wird zu 1; 1 wird zu 0 → Jedoch nicht +1 (Zweierkompliment)

$$B = 1\ 0\ 1\ 0_2$$

$$\overline{B - 1} = 0\ 1\ 0\ 1_2$$

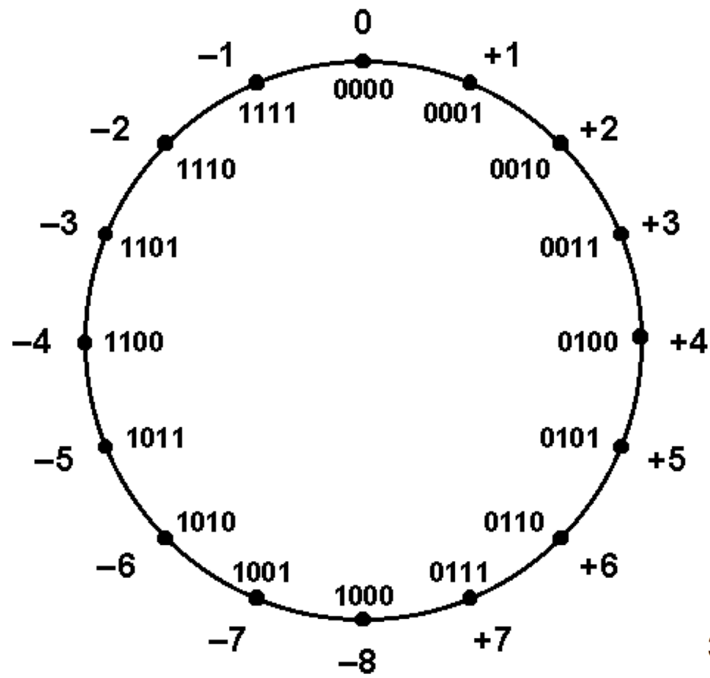
Sieht dann wie folgt aus:



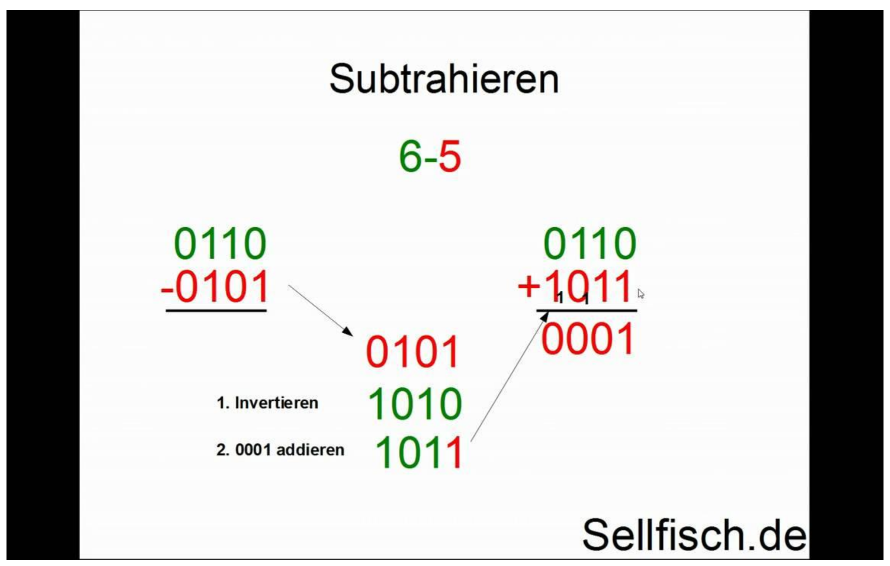
Negative Binärzahlen - Zweierkompliment

Zweierkompliment

= Zweierkomplement + 1 → Wenn man das Zweierkompliment zur normalen Zahl addiert kommt immer 1xxxx raus.



Daher nimmt 0 nur eine Belegung ein im Gegensatz zu Einer Kompliment und man kann eine Zahl mehr speichern.



BSP Rep 1-3

unsigned = $0 - (2^n - 1)$
signed = $-(2^{n-1} - 1) - (2^{n-1} - 1)$
Einerkompliment = $-(2^{n-1} - 1) - (2^{n-1} - 1)$
Zweierkompliment = $-(2^{n-1}) - (2^{n-1} - 1)$

3. **Darstellung negativer Zahlen**

(a) Welcher Bereich der Dezimalzahlen kann mit folgender Darstellung abgebildet werden?

i. **unsigned** 8 Bit Binärzahl, also ohne Vorzeichen:
Stellen Sie 129_{10} als **unsigned** 8 Bit Binärzahl dar.

ii. **signed** 8 Bit Binärzahl, also mit Vorzeichen und Betrag:
Stellen Sie 3_{10} und -3_{10} als **signed** 8 Bit Binärzahl dar.

iii. 8 Bit **Einer**komplement-Darstellung:
Stellen Sie 3_{10} und -3_{10} in **Einer**komplement-Darstellung dar.

iv. 8 Bit **Zweier**komplement-Darstellung:
Stellen Sie 3_{10} und -3_{10} in **Zweier**komplement-Darstellung dar.

v. Geben Sie für (a)-(d) den Bereich allgemein für n Bit Binärzahlen an

(b) Was passiert, wenn Sie bei einer **4-Bit** Architektur, die mit **Zweierkomplement-darstellung** arbeitet:

$-(2^{4-1}) - (2^{4-1}-1) = -8 - 7$

- die Zahlen $+7_{10}$ und $+3_{10}$ addieren? → 10 ⇒ problem ⇒ −6
- die Zahlen -5_{10} und -3_{10} addieren? → passt
- die Zahlen -6_{10} und -3_{10} addieren? → geht nicht

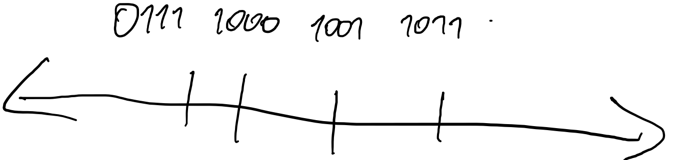
a) 129 = 10000001

b) −3 = 10000011
3 = 00000011

c) 3 = 00000011
−3 = 11111100

d) 3 = 00000011
−3 = 11111101

Exessdarstellung



Exessdarstellung

Beispiel: $n = 2^5 \Rightarrow q = 2^4 = 16$

- $z = -2^4$ bis -1 $\Rightarrow$ 00000 bis 01111
- $z = 0$ $\Rightarrow$ 10000$(= q)$
- $z = 1$ bis $+(2^4 - 1)$ $\Rightarrow$ 10001 bis 11111
- Führendes Bit codiert Vorzeichen: 0/1 $\Rightarrow$ -/+
- Null hat nur eine Codierung!
- **Ordnungsrelation bleibt erhalten!**

$\Rightarrow$ **Vergleiche zwischen Zahlen!**

Festkomma Darstellung

Festpunkt-Darstellung

VZ	Vorkomma					Nachkomma				
v	d_{n+g-1}	d_{n+g-2}	…	…	d_n	d_{n-1}	…	…	d_1	d_0

- n Nachkommastellen
- g Vorkommastellen
- 1 Vorzeichenbit (1 bei negativer Zahl)

$\Rightarrow$ Zahl x ist N = n + g + 1 Bit breit

- $x = (-1)^v \cdot 2^{-n} \cdot \sum_{j=0}^{N-2} d_j \cdot 2^j$

$= (-1)^v \, d_{N-2} \ldots d_n \, . \, d_{n-1} \ldots d_1 \, d_0$

Festpunkt-Zahlensystem

- N = 12 Bit, n = 3 Bit Nachkommastellen
- $(1001\ 1000\ 1110)_2 \Leftrightarrow (-49.75)_{10}$

VZ	Vorkomma								Nachkomma		
v	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
1	0	0	1	1	0	0	0	1	1	1	0

z.b 2,77 → sprich mit Nachommastellen

Festpunkt-Zahlensystem

- N = 12 Bit, n = 3 Bit Nachkommastellen
- $(-10.375)_{10} \Leftrightarrow (\ 1010.011\ )_2$

VZ	Vorkomma								Nachkomma		
v	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
1	0	0	0	0	1	0	1	0	0	1	1

Gleitpunkt/Fließkomma Darstellung

Fließkommadarstellung

$$x = (-1)^V \cdot M \cdot B^{\pm E}$$

- $V \ldots$ Vorzeichen (V=1: negative Zahl, V=0: positive Zahl)
- $M \ldots$ Mantisse: Für Genauigkeit entscheidend
- $B \ldots$ Basis
- $E \ldots$ Exponent: Für Bereich entscheidend

Beispiele

- $-0.0000123_{10} = -123 \cdot 10^{-7} = -12.3 \cdot 10^{-6}$
- $2016_{10} = 20.16 \cdot 10^2 = 0.2016 \cdot 10^4$
- ...

Mehrdeutigkeiten möglich

- Mögliche **Normalisierung** um Mehrdeutigkeiten zu vermeiden:

$\Rightarrow$ Mantisse hat genau eine Vorkommastelle, die ungleich 0 ist

- $-1.23 \cdot 10^{-5}$
- $2.016 \cdot 10^3$

Denormalisierte Gleipunkt Darstellung

Denormalisierte Gleitpunkt-Darstellung

Problem im vorherigen Beispiel: Lücke zwischen 0 und kleinsten positiven darstellbaren Zahl 0.5_{10}

- Grund: Normalisierungsbedingung ($m_0 \; ! = 0$)
- U.a. gilt nun die Eigenschaft $x = y \Leftrightarrow x - y = 0$ NICHT mehr

⇒ Bsp.: y = $1.11 \cdot 2^{-1} = 0.875_{10}$; x = $1.00 \cdot 2^{0} = 1.00_{10}$

⇒ x - y = $0.01 \cdot 2^{-1} = 0.125_{10}$ kann nicht als normalisierte Gleitpunktzahl dargestellt werden

⇒ Nächstliegende Zahl wäre Null. Durch eine Rundung auf Null kann ein folgenschwerer Laufzeit-Fehler passieren

⇒ Beispiel: `if (x!=y) then z = 1/(x-y);`

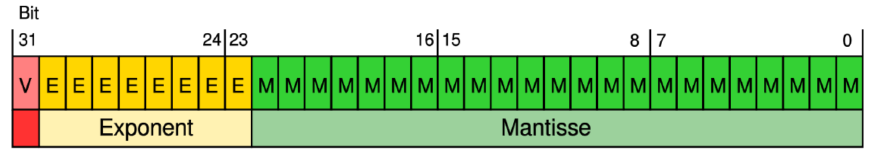
IEEE 754

IEEE 754 Standard (aktuell: 2008)

Verschiedene Formate definiert - die wichtigsten:

1. Einfache Genauigkeit: 32 Bits ("Single-precision")
2. Doppelte Genauigkeit: 64 Bits ("Double-precision")

Einfache Genauigkeit



Single-Precision: 32 bit

- VZ: 1 bit, Exponent: 8 bit, Mantisse: 23 Bit

$\Rightarrow$ Maschinenepsilon $\epsilon = 2^{-23} \Rightarrow$ dezimal $\approx 1.2 \cdot 10^{-7}$

$\Rightarrow$ Anzahl Dezimalstellen: ≈ 7

Double-Precision: 64 bit

- VZ: 1 bit, Exponent: 11 bit, Mantisse: 52 Bit

$\Rightarrow$ Maschinenepsilon $\epsilon = 2^{-52} \Rightarrow$ dezimal $\approx 2.2 \cdot 10^{-16}$

$\Rightarrow$ Anzahl Dezimalstellen: ≈ 16

Exponent in Exzessdarstellung

- Biased exponent e, $e = E + q$ ($\Rightarrow E = e - q$)
- E... rechnerisch wirkende Exponent, q... Exzess/bias

$\Rightarrow$ q = 127 bei 32 Bit (single-precision)

$\Rightarrow$ q = 1023 bei 64 Bit (double-precision)

Spezielle Codierung für

- Null
- Unendlich
- Ungültige Zahl (NaN, not a number)

Beispiel: -5.375_{10} in Single Precision

Umwandlung in Dualsystem

- $-5.375_{10} = -101.011_2$

Normierung der Mantisse $\pm 1. \cdots \cdot 2^E$

- $-1.01011_2 \cdot 2^2$
- Nur Bitfolge nach dem Komma wird gespeichert: 01011
- Restlichen Stellen werden mit 0-ern aufgefüllt

$\Rightarrow$ 01011000000000000000000 (23 bit)

Exponent in Exzessdarstellung

- $e = E + q \Rightarrow 129 = 2 + 127$
- e = 10000001

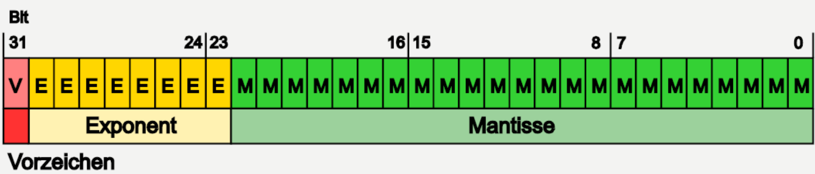
Ergebnis

- $-5.375_{10} = -101.011_2 = -1.01011_2 \cdot 2^{2 (entspricht\ E)} = -1.01011_2 \cdot 2^{(129-127)(entspricht\ e-q)}$

- Vorzeichenbit = 1
- Exponent e = 10000001
- Mantisse M = 01011000000000000000000 (23 bit)

IEEE 754

- $x = (-1)^V * M * B^{\pm E}$
- V=Vorzeichen, M=Mantisse, B=Basis, E=Exponent
- Bildung Single Precision 32 Bit:
  - **Vorzeichen** 1 Bit: 1 → negativ, 0 → positiv
  - **Exponent** e 8 Bit: Exzessdarstellung e = E+q, **q=127**, E=
  - **Mantisse** 23 Bit: genau eine Vorkommastelle ungleich 0 → Maschinenepsilon $\epsilon = 2^{-23}$



- $x = (-1)^V * M * B^{\pm E}$
- V=Vorzeichen, M=Mantisse, B=Basis, E=Exponent
- Bildung Single Precision 64 Bit:
  - **Vorzeichen** 1 Bit: 1 → negativ, 0 → positiv
  - **Exponent** e 11 Bit: Exzessdarstellung – e = E+q, **q=1023**
  - **Mantisse** 52 Bit: genau eine Vorkommastelle ungleich 0 → Maschinenepsilon $\epsilon = 2^{-52}$

BSP-Rep 1-4

4. Numerik – IEEE754

Stellen Sie die (negative) Dezimalzahl -23.125_{10} als binäre Fließkommazahl in Single Precision (IEEE754) dar.

(a) Wandeln Sie die Dezimalzahl in eine Binärzahl um

(b) Berechnen Sie die Mantisse M

(c) Berechnen Sie den Exponenten $e = E + q$ und geben Sie das Vorzeichenbit an (q siehe Vorlesungsfolien).

(d) Geben Sie die komplette Zahl

- als normalisierte **Dezimalzahl** in Fließkommadarstellung (siehe Formel VO Folie 73) bzw.
- als binäre Fließkommazahl in Single Precision nach IEEE 754 an

16 8 4 2 1
1 0 1 1 1

a) −23 --> 10111.001

$0{,}125 \cdot 2 = 0{,}25 \quad = 0$
$0{,}25 \cdot 2 = 0{,}5 \quad = 0$
$0{,}5 \cdot 2 = 1 \quad = 1$

b) $-1.0111001 \cdot 2^4$

← 2^n n = verschobene Stellen (Komma)

10110010000.... 23 bit lang (32 bit)

=> IEE754 1 stell danach => Komma

c) Vorzeichenbit = 1 => negativ

Exponent = $e = E + q$

Exzessdarstellung !

$e = 4 + 127$

$e = 131$ => 10000011

Rundungsfehler

Addition von Maschinenzahlen

Besonders kritisch: Endergebnis nahe bei Null (“Auslöschung”)

- Beispiel: Differenz zwischen $a = 3/5$ und $b = 4/7$ bei fünfstelliger Mantisse (hier 5 Stellen inkl. führender 1)
- Exaktes Ergebnis: $a - b = 1/35 \approx 0.11101_2 \cdot 2^{-5}$
- Rundung: $a = (1.0011001...)_2 \cdot 2^{-1} \approx 1.0011_2 \cdot 2^{-1}$ und $b = (1.001001...)_2 \cdot 2^{-1} \approx 1.0010_2 \cdot 2^{-1}$
- Also ergibt Rechnung: $1.0011_2 \cdot 2^{-1} - 1.0010_2 \cdot 2^{-1} = 0.0001_2 \cdot 2^{-1} = 1.0000_2 \cdot 2^{-5} = 1/32$
- Relativer Fehler: $|\frac{x-f}{x}| = |(1/35 - 1/32)/(1/35)| = 9.4\%$
- Vergleich: die Maschinengenauigkeit bei fünfstelliger Mantisse (inkl. führender 1) liegt bei ca. 3.1%

Beispiele [Bearbeiten | Quelltext bearbeiten]

Berechnung Dezimalzahl → IEEE754-Gleitkommazahl [Bearbeiten | Quelltext bearbeiten]

Die Zahl $18{,}4$ soll in eine Gleitkommazahl umgewandelt werden, dabei nutzen wir den Single IEEE-Standard.

1. Umwandlung der Dezimalzahl in eine duale Festkommazahl ohne Vorzeichen

$$\begin{aligned} 18 \div 2 &= 9 && \text{Rest } 0 && \text{(Least-Significant Bit)} \\ 9 \div 2 &= 4 && \text{Rest } 1 \\ 4 \div 2 &= 2 && \text{Rest } 0 \\ 2 \div 2 &= 1 && \text{Rest } 0 \\ 1 \div 2 &= 0 && \text{Rest } 1 && \text{(Most-Significant Bit)} \\ & 18 = 10010_2 \end{aligned}$$

$$\begin{aligned} 0{,}4 \cdot 2 &= 0{,}8 && -0 && \text{(Most-Significant Bit)} \\ 0{,}8 \cdot 2 &= 1{,}6 && -1 \\ 0{,}6 \cdot 2 &= 1{,}2 && -1 \\ 0{,}2 \cdot 2 &= 0{,}4 && -0 \\ 0{,}4 \cdot 2 &= 0{,}8 && -0 \\ 0{,}8 \cdot 2 &= 1{,}6 && -1 && \text{(Least-Significant Bit)} \\ & \dots \\ & 0{,}4 = 0{,}011001\dots_2 \end{aligned}$$

also $18{,}4 = 10010{,}011001\dots_2 = (1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) + (0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} + \dots)$

2. Normalisieren und bestimmen des Exponenten

Ausklammern der höchsten Zweierpotenz: $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 0 \cdot 2^{-8} + 0 \cdot 2^{-9} + 1 \cdot 2^{-10} + \dots) \cdot 2^4$

Der Bias-Wert für den Exponenten setzt sich aus einer Null und $r - 1$ Einsen zusammen. Für $r = 8$ gilt also: $B = 01111111_2 = 127_{10}$

Der Exponent der Zweierpotenz 2^4 wird mit dem Bias B damit als $4 + 127 = 131 = 10000011_2$ hinterlegt.

Die Normalisierung von $10010,011001\dots_2$ lässt sich genauso durch Kommaverschiebung im Binärsystem erreichen:

$$\begin{aligned} & 10010{,}011001\dots \cdot 2^{01111111-01111111} \\ = {} & 1001{,}0011001\dots \cdot 2^{10000000-01111111} \\ = {} & 100{,}10011001\dots \cdot 2^{10000001-01111111} \\ = {} & 10{,}010011001\dots \cdot 2^{10000010-01111111} \\ = {} & 1{,}0010011001\dots \cdot 2^{10000011-01111111} \end{aligned}$$

Die Mantisse ist also $1{,}0010011001\dots$ und der Exponent mit Bias 10000011.

3. Vorzeichen-Bit bestimmen

Hier positiv, also 0.

4. Die Gleitkommazahl bilden

1 Bit Vorzeichen	8 Bit Exponent	23 Bit Mantisse
0	10000011	00100110011001100110011

Die Vorkomma-Eins der Mantisse wird als Hidden Bit weggelassen.

BSP-Rep 1-5

5. Numerik – Fehlerfortpflanzung

Berechnen Sie den relativen Fehler, der bei der Berechnung der Differenz von zwei Maschinenzahlen auftritt. Gehen Sie von einer fünfstellige Mantisse wie auf Folie 79 (Kapitel 1 – "Auslöschung") aus.

Gegeben: $x = 2/5$; $y = 3/7$;

Gesucht: $x - y$

$x = 0{,}4$

$y = 0{,}4285$

$0{,}011001$

$x = \quad 0{,}4 \cdot 2 = 0{,}8 \quad 0$
$0{,}8 \cdot 2 = 1{,}6 \quad 1$
$0{,}6 \cdot 2 = 1{,}2 \quad 1$
$0{,}2 \cdot 2 = 0{,}4 \quad 0$
$0{,}4 \cdot 2 = 0{,}8 \quad 0$
$0{,}8 \cdot 2 = \quad 1$

$y = \quad 0{,}43 \cdot 2 = 0{,}86 \quad 0$
$0{,}86 \cdot 2 = 1{,}72 \quad 1$
$0{,}72 \cdot 2 =$
⋮

=> Mantisse

$1.10011 \cdot 2^{-2}$ => Komma 2 nach rechts

$1.11010 \cdot 2^{-2}$

$= -0.001 \cdot 2^{-2} = -2^{-5} = 2^5 = 32$ ✓

$-\frac{1}{32}$

$x = \frac{2}{5} = 0{,}4 = \frac{14}{35}$

$y = \frac{3}{7} = 0{,}4285\ldots \quad \frac{15}{35}$

$x - y = -\frac{1}{35}$ exakt

$x = 1.1001 \cdot 2^{-2}$ | umrechnen

$y = 1.1011 \cdot 2^{-2}$

$x - y = 0.0010 \cdot 2^{-2} = \frac{1}{32} = 2^{-5}$

relativer Fehler = ca 9,4 %

Basics

Unäre Operationen

- Negation („nicht“, „NOT“): $\neg$

Binäre Operationen

- Konjunktion („und“, „AND“): $\wedge$
- Disjunktion („inklusives oder“, „OR“): $\vee$
- Implikation[a]: $\Rightarrow$
- Äquivalenz: $\Leftrightarrow$

[a] VORSICHT: die Implikation (auch materiale Implikation, Subjunktion, Konditional genannt) drückt eine hinreichende Bedingung aus, aber keine Kausalität!

Wahr	Falsch
W (wahr)	F (falsch)
T (true)	F (false)
H (high)	L (low)
1 (Bit gesetzt)	0 (Bit nicht gesetzt)

Wahrheitswerte

a	b	$\neg$a	a $\vee$ b	a $\wedge$ b	a $\Rightarrow$ b	a $\Leftrightarrow$ b
0	0	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	1	0	0	0
1	1	0	1	1	1	1

Reihenfolge

Ordnungsrelation (Operatorrangfolge)

1. $\neg$
2. $\land$
3. $\lor$

Gesetze

Tautologie, Antilogie

- Tautologie: Ausdruck, der für jede Belegung **wahr** ist
- Antilogie (Kontradiktion): Ausdruck, der für jede Belegung **falsch** ist

Verknüpfung mit 1 bzw. 0

- $a \wedge 1 = a$
- $a \vee 0 = a$

Kommutativität

- $a \wedge b = b \wedge a$
- $a \vee b = b \vee a$

Idempotenz

- $a \wedge a = a$
- $a \vee a = a$

Komplementäres Element

- $a \wedge \neg a = 0$
- $a \vee \neg a = 1$

Assoziativität

- $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
- $(a \vee b) \vee c = a \vee (b \vee c)$

Involutivität

- $\neg(\neg a) = a$

Absorption

- $(a \wedge b) \vee a = a$
- $(a \vee b) \wedge a = a$

De Morgansche Regel

- $\neg(a \wedge b) = \neg a \vee \neg b$
- $\neg(a \vee b) = \neg a \wedge \neg b$

Distributivität

- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

Auslöschung

- $a \wedge (b \vee \neg b) = a$
- $a \vee (b \wedge \neg b) = a$

1. Ersetze $G \leftrightarrow H$ durch $(G \rightarrow H) \wedge (H \rightarrow G)$
2. Ersetze $G \rightarrow H$ durch $(\neg G \vee H)$

Alternative Folie dazu

Kommutativität

$$a \wedge b = b \wedge a$$
$$a \vee b = b \vee a$$

Assoziativität

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$
$$(a \vee b) \vee c = a \vee (b \vee c)$$

Involutivität

$$\neg(\neg a) = a$$

Distributivität

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$
$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

Absorption

$$(a \wedge b) \vee a = a$$
$$(a \vee b) \wedge a = a$$

De Morgan

$$\neg(a \wedge b) = \neg a \vee \neg b$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$

Verknüpfung

$$a \wedge 1 = a$$
$$a \vee 0 = a$$

Auslöschung

$$a \wedge (b \vee \neg b) = a$$
$$a \vee (b \wedge \neg b) = a$$

1. Ersetze $G \leftrightarrow H$ durch $(G \rightarrow H) \wedge (H \rightarrow G)$
2. Ersetze $G \rightarrow H$ durch $(\neg G \vee H)$

Komplementäres Element

$$a \wedge \neg a = 0$$
$$a \vee \neg a = 1$$

Idempotenz

$$a \wedge a = a$$
$$a \vee a = a$$

Auflösen → & =

Konjunktive und Disjunktive Normalform - Algorithmus

Diese Schritte lassen sich auch in einen Algorithmus fassen der eine Formel F in eine semantisch äquivalente KNF umzuwandeln.

Algorithmus

1. Ersetze $G \leftrightarrow H$ durch $(G \rightarrow H) \wedge (H \rightarrow G)$
2. Ersetze $G \rightarrow H$ durch $(\neg G \vee H)$
3. Iteriere das Folgende solange wie möglich
   1. Ersetze $\neg\neg G$ durch G
   2. Ersetze $\neg(G \wedge H)$ durch $(\neg G \vee \neg H)$
   3. Ersetze $\neg(G \vee H)$ durch $(\neg G \wedge \neg H)$
4. Iteriere das Folgende solange wie möglich
   1. Ersetze $(G \vee (H \wedge I))$ und $((H \wedge I) \vee G)$ durch $((G \vee H) \wedge (G \vee I))$

„Ersetze“ liest sich als „Ersetze alle Teilformeln der Form“

BSP-REP1/2

1. **Aussagenlogik**

   (a) Beweisen Sie mittels Wahrheitstabelle die De Morgan'sche Regel

   (b) Beweisen Sie mittels Wahrheitstabelle: $a \to b = \neg a \vee b$

   (c) Beweisen Sie mittels Wahrheitstabelle: $a \leftrightarrow b = (a \to b) \wedge (b \to a)$

   (d) Wenn Boolesche Funktionen überall den Wert 1 annehmen (also immer "wahr"sind), nennt man das eine Tautologie; wenn sie überall den Wert 0 annehmen (also immer "falsch" sind), handelt es sich um eine Antilogie (Widerspruch)

   ⇒ Handelt es sich beim folgenden Satz um eine Tautologie, eine Antilogie (Kontradiktion), oder keines von beiden? Erklären Sie ihre Antwort!

   $(b \leftrightarrow a) \vee [(c \wedge \neg b) \to (c \vee a)]$

a) $\neg a \vee \neg b = \neg(a \wedge b)$

a	b	$\neg a \vee \neg b$	$\neg(a \wedge b)$
0	0		
0	1	1	1
1	0	1	1
1	1	1	1
		0	0

d) tautologie ... $\vee \neg b \to 1$

↑ immer wahr

2. **Aussagenlogik**

   Vereinfachen Sie folgende Boole'sche Ausdrücke soweit wie möglich, jedoch OHNE KV-Diagramm. Verwenden Sie zur Vereinfachung die Gesetze und Axiome der Aussagelogik. Geben Sie bei jedem Schritt an, welches Axiom bzw. Gesetz Sie verwendet haben.

   (a) $f(x, y) = (y \vee \neg x) \wedge (y \vee x)$

   (b) $f(x, y) = y \vee (\neg x \wedge x)$

   (c) $f(x, y) = y \wedge (\neg x \vee x)$

   (d) $f(x) = x \vee 1 \wedge 0$ ← reihenfolge $\neg \to \wedge \to \vee$

   (e) $f(x) = (x \vee 1) \wedge 0$

   (f) $f(e_3, e_2, e_1, e_0) = (\neg e_3 \wedge e_2 \wedge e_1 \wedge e_0) \vee \neg(\neg e_2 \vee e_0 \vee e_3 \vee \neg e_1)$

   (g) $f(x, y, z) = (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z) \vee \neg(\neg x \vee y \vee \neg z)$

a) $(y \vee \neg x) \wedge (y \vee x)$ => Assoziativ

$y \vee (x \vee \neg x)$ => tautologie

y

b) $y \vee 0$ $(\neg x \wedge x)$ -> Analyse

y

c) $y \wedge 0$ => nicht möglich

0

d) $x \vee 0$

x

Disjunktive Normalform (DNF)

DNF

- **Vollkonjunktionen werden disjunktiv verknüpft**
- Vollkonjunktionen = alle Variablen sind konjunktiv verknüpft

$\Rightarrow$ Variablen können negiert sein

- Beispiel: $f(e_1, e_2, e_3) = (e_1 \Rightarrow e_2) \wedge (\neg e_1 \Leftrightarrow e_3)$

Negierungen erlaubt

1.: Wahrheitstabelle

#	e_1	e_2	e_3	$e_1 \Rightarrow e_2$	$\neg e_1 \Leftrightarrow e_3$	$(e_1 \Rightarrow e_2) \wedge (\neg e_1 \Leftrightarrow e_3)$
1	0	0	0	1	0	0
2	0	0	1	1	1	1
3	0	1	0	1	0	0
4	0	1	1	1	1	1
5	1	0	0	0	1	0
6	1	0	1	0	0	0
7	1	1	0	1	1	1
8	1	1	1	1	0	0

2.: Zeilen auswählen, Vollkonjunktionen bilden

- Auswahl der Zeilen mit **wahrem** Ergebnis (= 1)
  - Zeile 2
  - Zeile 4
  - Zeile 7
- Variablen mit Wert 0 negieren, andere direkt übernehmen, und alle konjunktiv verbinden
- $\neg e_1 \wedge \neg e_2 \wedge e_3$, $\neg e_1 \wedge e_2 \wedge e_3$, $e_1 \wedge e_2 \wedge \neg e_3$

3.: Disjunktive Normalform

Einzelne „Zeilen“ disjunktiv verbinden.

$$(\neg e_1 \wedge \neg e_2 \wedge e_3) \vee (\neg e_1 \wedge e_2 \wedge e_3) \vee (e_1 \wedge e_2 \wedge \neg e_3)$$

(und und) oder (und und) oder ...

Alle zeilen mit 1 wählen und die verbindungen der Lösungen konjuntiv (alle Lösungen dann disjunktiv zusammen)

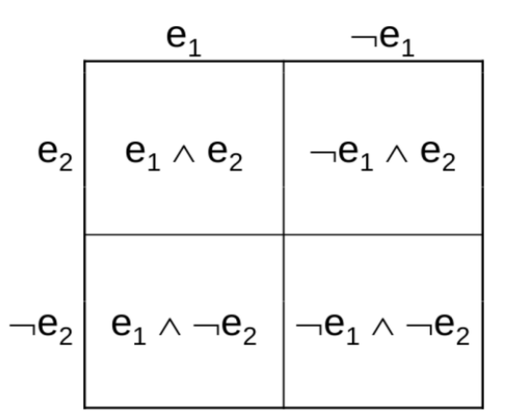
Konjunktive Normalform (KNF)

KNF

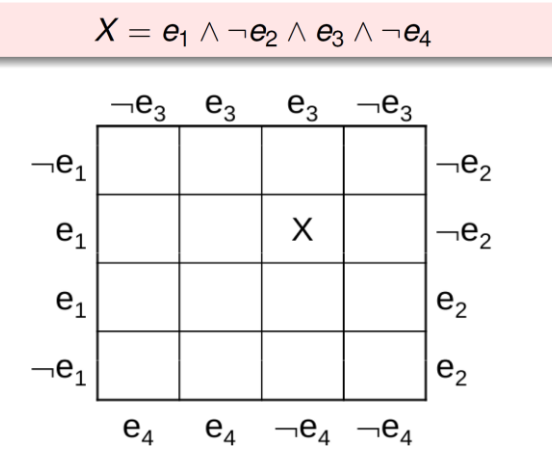
- **Volldisjunktionen werden konjunktiv verknüpft**
- Volldisjunktionen = alle Variablen sind disjunktiv verknüpft
- Beispiel: $f(e_1, e_2, e_3) = (e_1 \wedge e_2) \vee e_3$ Negierungen erlaubt

Hierbei werden alle Zeilen welche eine 0 ergeben aus der Wahrsheitstabelle herangezogen. Diese werden negiert und disjunktiv verbunden → Alle Zeilen dann konjunitv zusammen

Boolische Funktionen und Vereinfachung

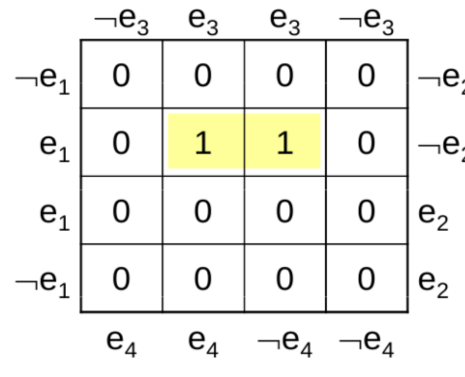


	e_1	$\neg e_1$
e_2	$e_1 \wedge e_2$	$\neg e_1 \wedge e_2$
$\neg e_2$	$e_1 \wedge \neg e_2$	$\neg e_1 \wedge \neg e_2$



$X = e_1 \wedge \neg e_2 \wedge e_3 \wedge \neg e_4$

	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$					$\neg e_2$
e_1			X		$\neg e_2$
e_1					e_2
$\neg e_1$					e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	



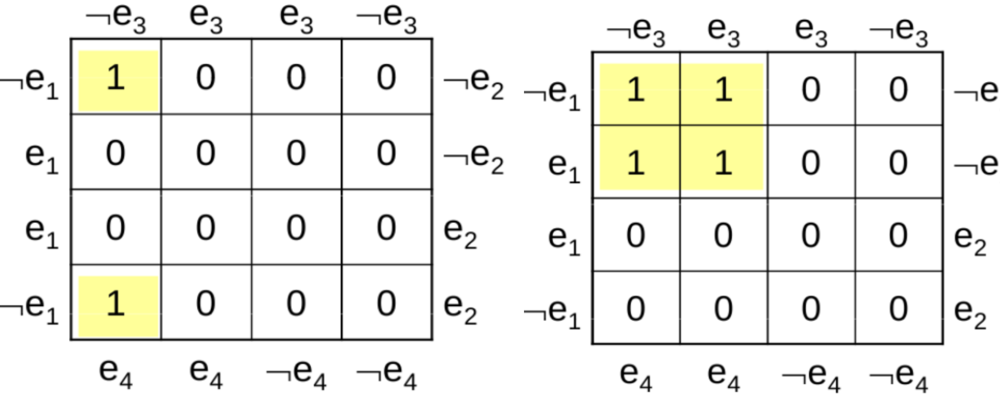
	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	0	0	0	0	$\neg e_2$
e_1	0	1	1	0	$\neg e_2$
e_1	0	0	0	0	e_2
$\neg e_1$	0	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $e_1 \wedge \neg e_2 \wedge e_3$

	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	0	0	0	0	$\neg e_2$
e_1	0	0	0	1	$\neg e_2$
e_1	0	0	0	1	e_2
$\neg e_1$	0	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $e_1 \wedge \neg e_3 \wedge \neg e_4$

Braucht eine DNF!! Zum bilden



	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	1	0	0	0	$\neg e_2$
e_1	0	0	0	0	$\neg e_2$
e_1	0	0	0	0	e_2
$\neg e_1$	1	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $\neg e_1 \wedge \neg e_3 \wedge e_4$

	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	1	1	0	0	$\neg e_2$
e_1	1	1	0	0	$\neg e_2$
e_1	0	0	0	0	e_2
$\neg e_1$	0	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $\neg e_2 \wedge e_4$

	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	0	0	0	1	$\neg e_2$
e_1	0	0	0	1	$\neg e_2$
e_1	0	0	0	1	e_2
$\neg e_1$	0	0	0	1	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

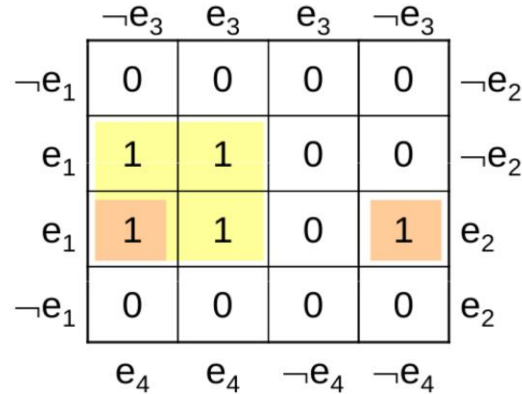
Abbildung: $\neg e_3 \wedge \neg e_4$

	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	1	0	0	1	$\neg e_2$
e_1	0	0	0	0	$\neg e_2$
e_1	0	0	0	0	e_2
$\neg e_1$	1	0	0	1	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $\neg e_1 \wedge \neg e_3$

Boolische Funktionen und Vereinfachung 2

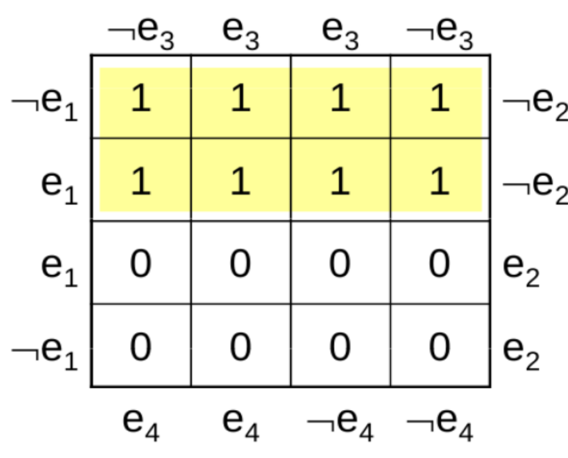
$f(e_1, e_2, e_3, e_4) = (e_1 \wedge \neg e_2 \wedge \neg e_3 \wedge e_4) \vee (e_1 \wedge \neg e_2 \wedge e_3 \wedge e_4) \vee (e_1 \wedge e_2 \wedge \neg e_3 \wedge e_4) \vee (e_1 \wedge e_2 \wedge e_3 \wedge e_4) \vee (e_1 \wedge e_2 \wedge \neg e_3 \wedge \neg e_4)$



	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	0	0	0	0	$\neg e_2$
e_1	1	1	0	0	$\neg e_2$
e_1	1	1	0	1	e_2
$\neg e_1$	0	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Minimiert: $(e_1 \wedge e_4) \vee (e_1 \wedge e_2 \wedge \neg e_3)$

42



	$\neg e_3$	e_3	e_3	$\neg e_3$	
$\neg e_1$	1	1	1	1	$\neg e_2$
e_1	1	1	1	1	$\neg e_2$
e_1	0	0	0	0	e_2
$\neg e_1$	0	0	0	0	e_2
	e_4	e_4	$\neg e_4$	$\neg e_4$	

Abbildung: $\neg e_2$

$(x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z) \vee \neg(\neg x \vee y \vee \neg z)$

	z	$\neg z$	
$\neg x$			$\neg y$
x	1	1	$\neg y$
x	1		y
$\neg x$			y

$(x \wedge \neg y) \vee (z \wedge x)$

$\Rightarrow x \wedge (z \vee \neg y) \Rightarrow$ Distributiv Gesetz

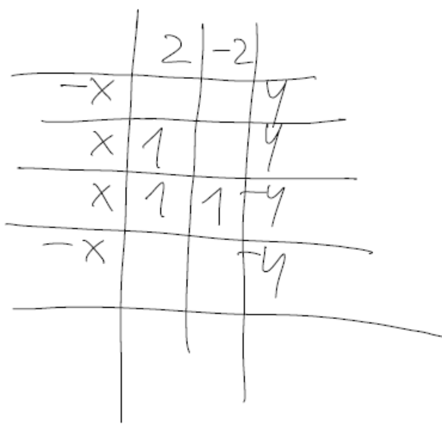
Hinweise zum erstellen → Gegenüberliegende Seiten nie gleich machen bei 3 → 4x2 Matrix
→ Gruppen bilden und Lösung von der Seite ablesen

BSP-Rep3/4

3. Normalformen und KV-Diagramm

Erstellen Sie mittels KV Diagramm die minimale DNF für Ausdruck aus Beispiel 2g) und vergleichen Sie die beiden Lösungen.

$$f(x, y, z) = (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z) \vee \neg(\neg x \vee y \vee \neg z)$$



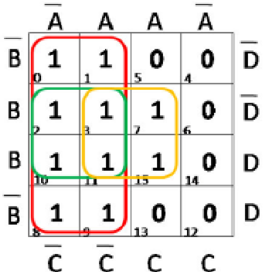
$\Rightarrow (x \wedge z) \vee (x \wedge \neg y)$

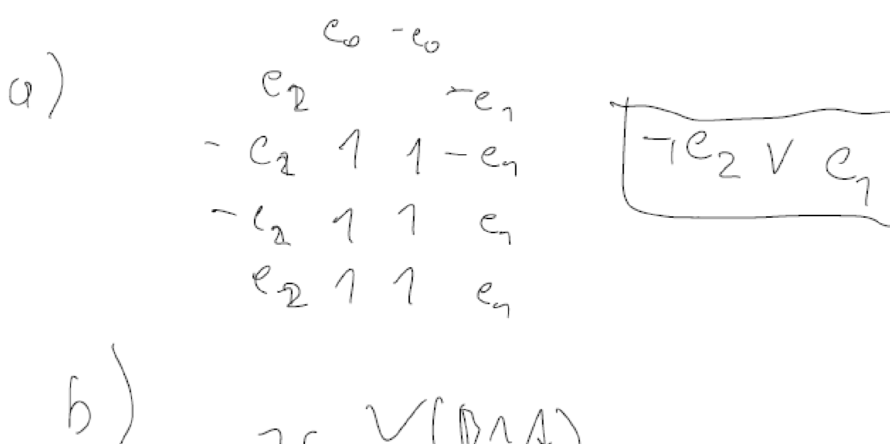
4. Wahrheitstabelle und KV-Diagramm

(a) Erstellen Sie für folgende Wahrheitstabelle ein KV Diagramm, lesen Sie die minimale DNF ab, und zeichnen Sie die Schaltung des minimalen Terms

e_2	e_1	e_0	$f(e_2, e_1, e_0)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(b) Erstellen Sie die **minimale DNF** für folgendes KV Diagramm und vergleichen Sie Ihre Lösung mit den eingezeichneten Gruppen. Was fällt Ihnen auf?

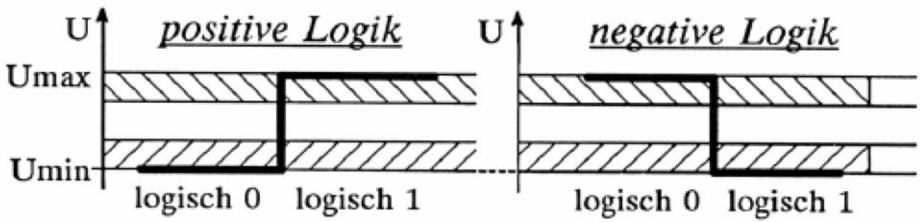




Logische Zustände

Wie realisiert man logische Zustände?

- unterschiedliche Spannungsniveaus
- Strom fließt/fließt nicht
- Kondensator geladen/ungeladen
- Position eines elektromagnetischen Relais'
- ...
- Also: viele Arten der Realisierung möglich
  $\rightarrow$ manche praktisch, manche weniger...



Spannung U = Potentialdifferenz

- Physik: elektrisches Feld = Coulombkraft auf Probeladung
- elektrisches Potential = potentielle Energie einer Einheitsladung
- Einheit: Volt; $1\,V = 1J/C$ (Energie pro Ladung!)

Wo elektrische Ladung ist, gibt es ein Feld und daher ein Potential.

- **Zwischen zwei unterschiedlichen Potentialen liegt Spannung.**
- **Es muss kein Strom fließen, damit Spannung da ist.**
- Je größer die Spannung, desto mehr Energie trägt jedes Coulomb der Ladungen.

Binär → 0/1

Halbleiter

Was ist ein Halbleiter?

- elektrischer Leiter: Kristallgitter + freie Elektronen
  - Ladungstransport (Beispiel: Silber, Kupfer, ...)

Ein freies valenzelektron

- Elektrische Leitfähigkeit von Halbleitern zwischen der von elektrischen Leitern und der von Nichtleitern
- Isolator: Elektronen fest an Atome gebunden
- Halbleiter: Kristallgitter ohne freie Elektronen
  - leitet besser als Isolator, aber schlechter als Leiter
  - Beispiel: Germanium, Silizium, ...

Dotierung

- Gezieltes Einschleusen von Fremdatomen in die Kristallstruktur
- Zwei Möglichkeiten:
  - *Donatoren* setzen ein Elektron frei
    → bewegliche negative Ladungen (*n-leitend*)
  - *Akzeptoren* binden jeweils ein Elektron
    → Löcher = "positive Ladungen" (*p-leitend*)

Anwendung: Diode

- Besonders interessant: p-n-Übergang
- Diffusion: Elektronen wandern von n-Zone zur p-Zone
  → elektrisches Feld, bis Gleichgewicht erreicht ist
  → Entstehung einer Sperrschicht = Verarmungszone

Anwendung: Diode

- Was passiert beim Anlegen einer äußeren Spannung?
  - Durchlassrichtung: Minus an n-Zone, Plus an p-Zone
  - Sperrrichtung: Plus an n-Zone, Minus an p-Zone
- Wichtige Anwendungen:
  - Gleichrichterdiode
  - Bipolartransistor (npn- bzw. pnp-Transistor)

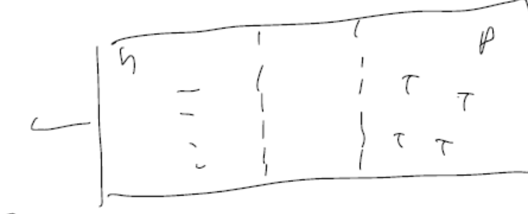
REP-Bsp 1/2

2. **Diode**

   Erläutern Sie, was passiert, wenn man am Eingang einer Halbleiter-Diode (mit pn-Übergang) einen Wechselstrom anlegt? Wozu könnte man so etwas verwenden?

↳ Wechsel zu gleichstrom umwandeln

=> Sperrichtung ->



\- -> leitet

\+ -> sperrt

-> nur + kommt raus

3. **Bipolartransistor**

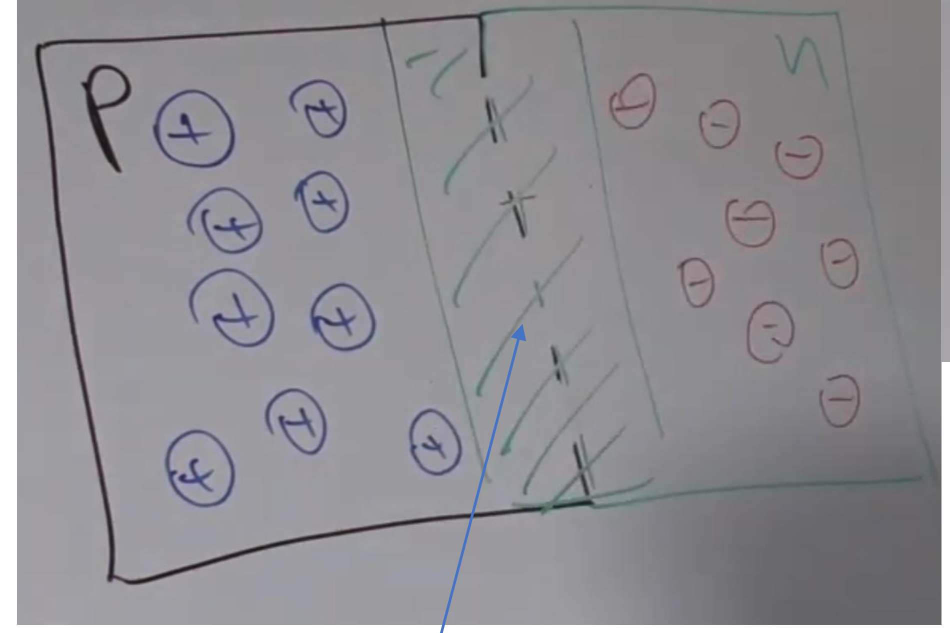
   Sie kennen aus der Vorlesung das Prinzip des npn-Bipolartransistors. Wäre auch ein pnp-Bipolartransistor denkbar?

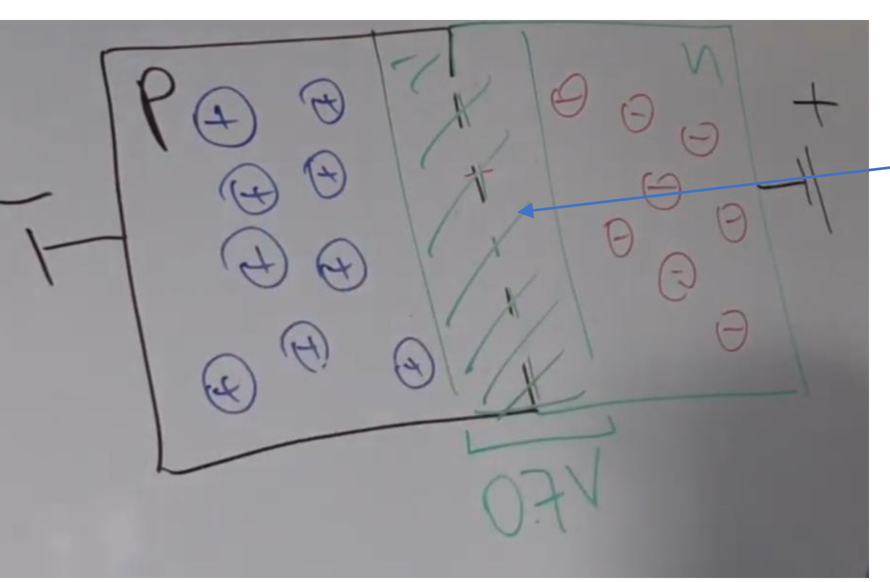
   Wenn ja, erläutern Sie dessen Aufbau und Funktion sowie allfällige Unterschiede zum npn-Bipolartransistor.

   Wenn nein, begründen Sie, warum nicht.

pnp => ident => aber langsamer => physik

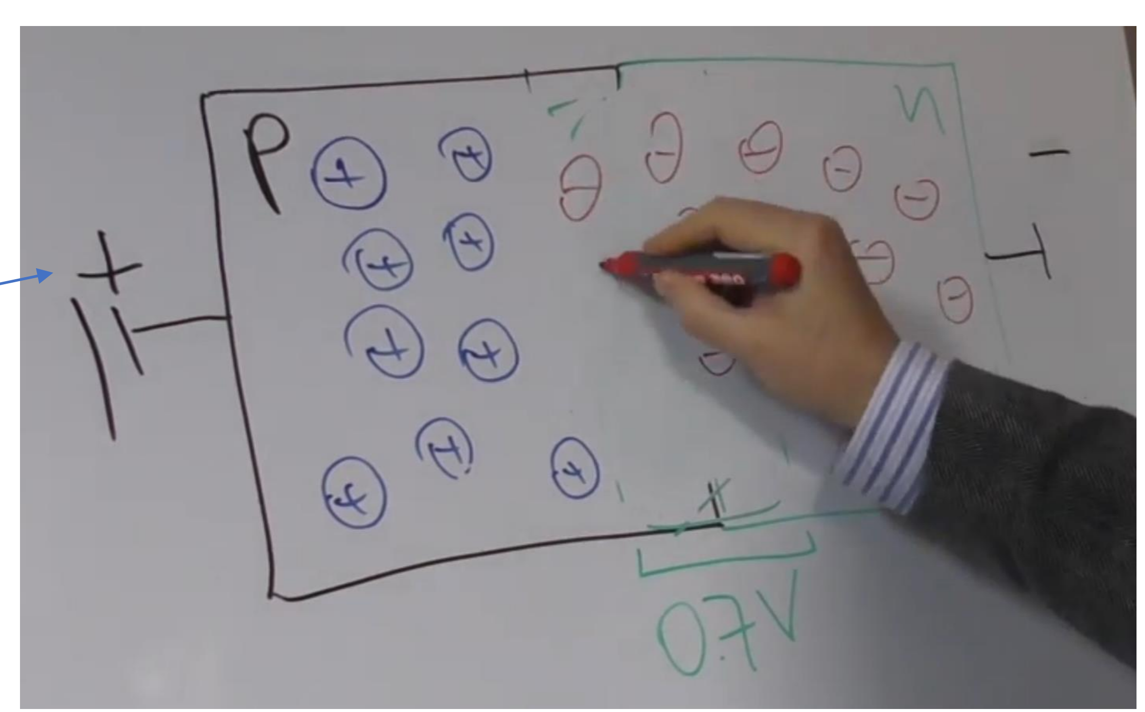
Halbleiter 2





Stabiler Zustand

Anlegen gleiche Spannung löst zone auf

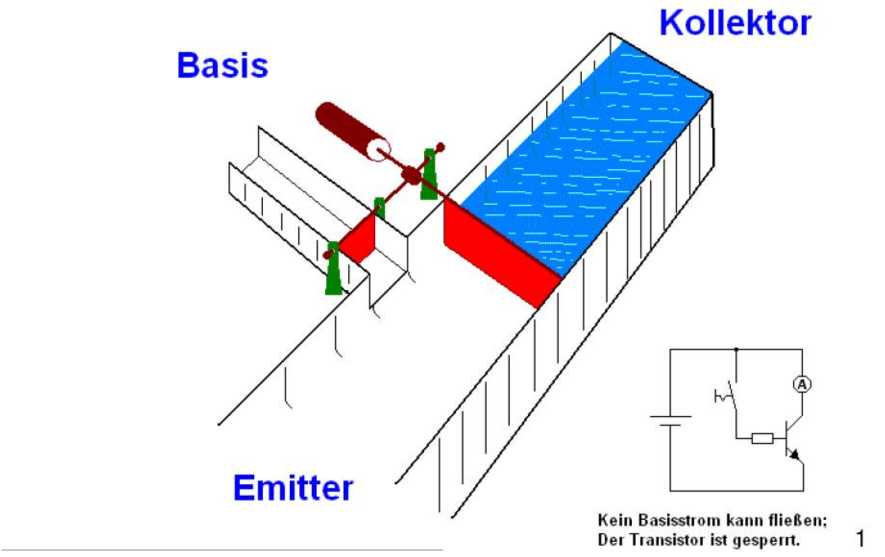


Verarmungszone

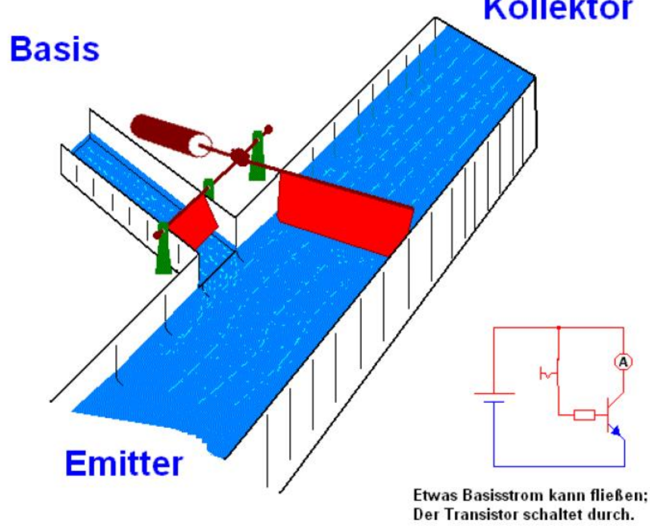
Siehe Anwendung Diode folie davor für details

Transistor

Prinzip des Transistors: sperrend



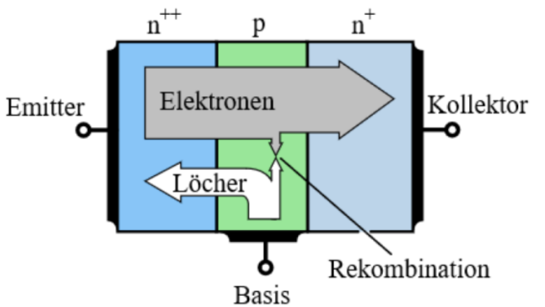
Prinzip des Transistors: geschalten



Transistor 2

npn-Transistor (selbstsperrend)

- Idee: n-p- und p-n-Übergang hintereinander
- linke Schicht stark n-dotiert, p-dotierte Schicht (Mitte) relativ dünn, rechte Schicht schwach n-dotiert
- kleiner Basis-Emitter-Strom → Emitter-Kollektor-Strom



2

Ziel → Verbindung Ermitter Kolletor → durch anlegen an Basis
Wenn das Basis + und Emitter - → **strom fliest basis – Emitter**
→ Verarmungszone löst sich auf → Verbindung gesamt

MOSFET - Metal Oxide Semiconductor Field Effect Transistor

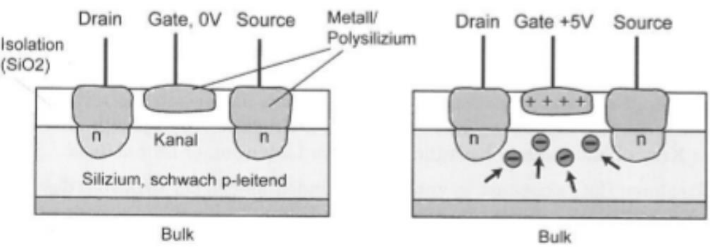


Abbildung 3.1: Selbstsperrender n-Kanal-MOSFET. Ein leitender Kanal bildet sich erst aus, wenn positive Gate-Spannung anliegt (rechts).

4

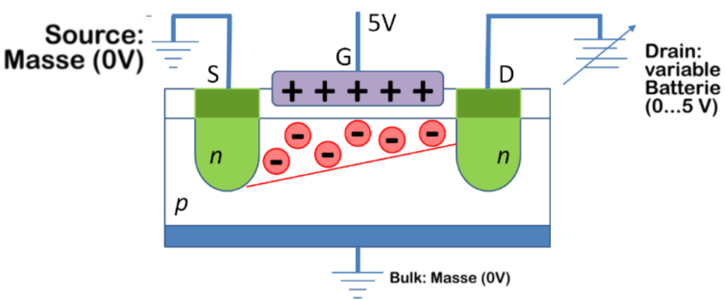
- Vier Anschlüsse: Source - Drain - Gate - Bulk
- Grundsätzliche Varianten: n-Kanal-MOSFET (NMOS) vs. p-Kanal-MOSFET (PMOS)

Hier kein Strom → wird heute verwendet
Gate zieht Elektronen an → Elektronen kette zwischen den beiden n bildet sich (drain, source)

MOS

Legende:
S = Source
D = Drain
G = Gate

Feldeffekt-Transistor: NMOS



- Gate nicht geladen: keine Ladungen im Kanal, kein Strom von Source nach Drain
- Gate geladen: Kanal leitet
- Aber: zusätzlich p-n-Übergang am Drain → Sperrichtung!
- Also: NMOS leitet niedriges U_{DS} falls hohes U_G

Leitet niedrige Spannung falls hohe anliegend

Zeichnen relevant!

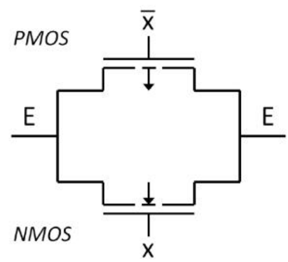
Feldeffekt-Transistor: PMOS

D
G
0V
5V
S
p
p
n
5V

- Im Vergleich zum NMOS alles umgedreht!
- Niedriges Potential am Gate:
  Kanal enthält Löcher, keine vollständige Rekombination
- Also: PMOS leitet hohes U_{DS} falls niedriges U_G

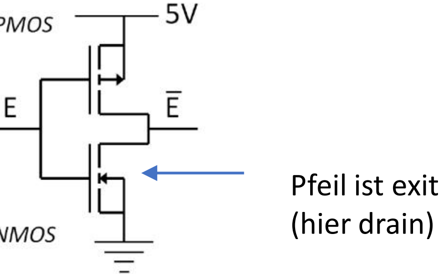
Leitet hohe Spannung falls niedrige anliegend

Feldeffekt-Transistor



- MOSFET allein noch kein Schalter (NMOS leitet nur niedriges U_{DS}, PMOS nur hohes U_{DS})
- Idee: kombiniere NMOS + PMOS in einer Schaltung → Transfer Gate (= Schalter, Transmission Gate)

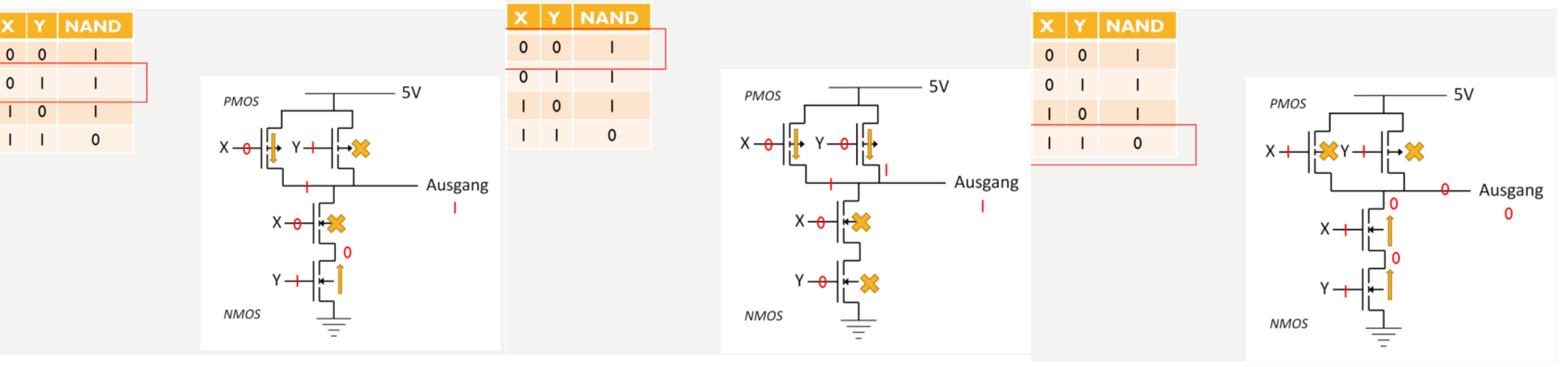
CMOS: Complementary MOS



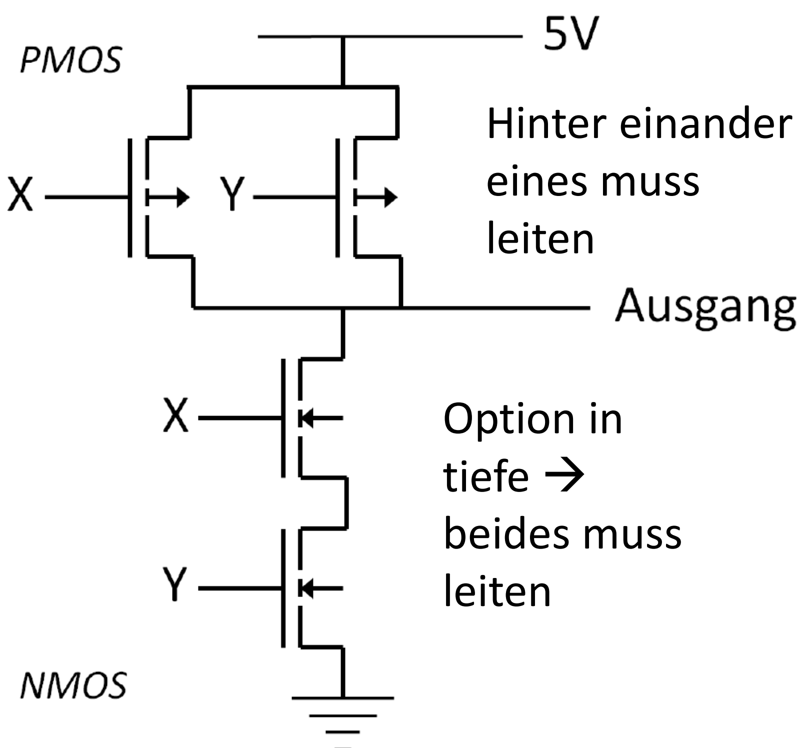
Pfeil ist exit (hier drain)

- Kombination von NMOS- (Pull-down-Pfad) und PMOS-Technologie (Pull-up-Pfad) auf gemeinsamem Substrat
- Einfachste CMOS-Schaltung: Inverter
- Viele komplexe Logikschaltungen benötigen Inverter an Ein- bzw. Ausgängen (z.B. XOR)

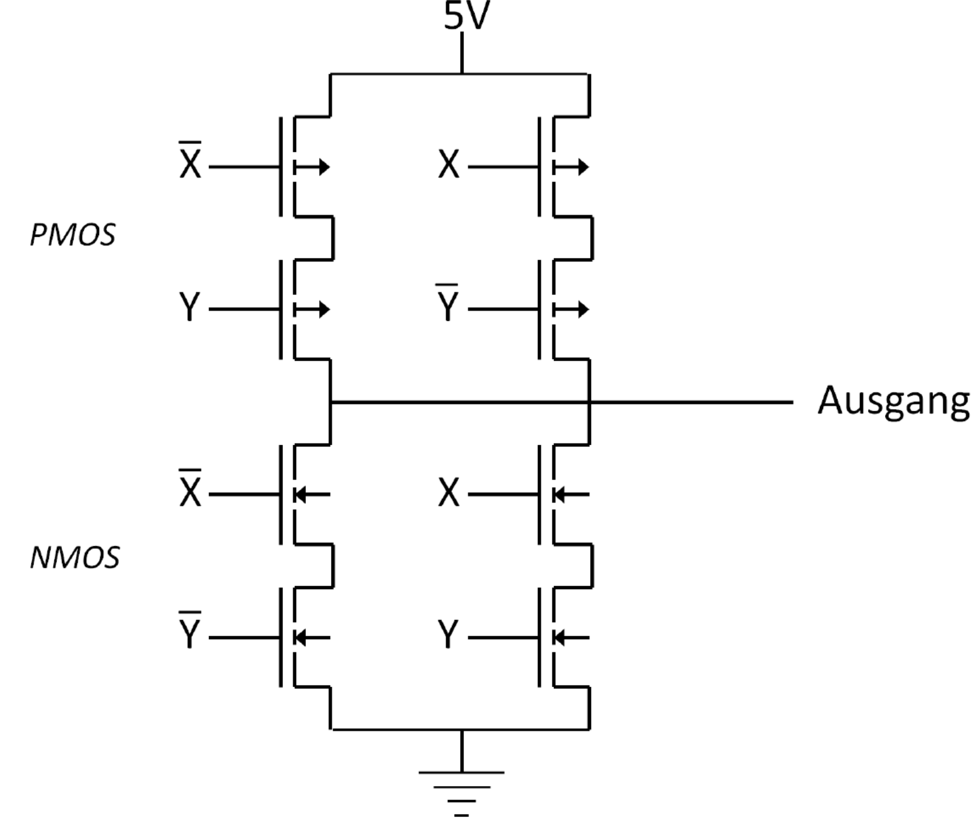
CMOS



CMOS: NAND



CMOS: XOR



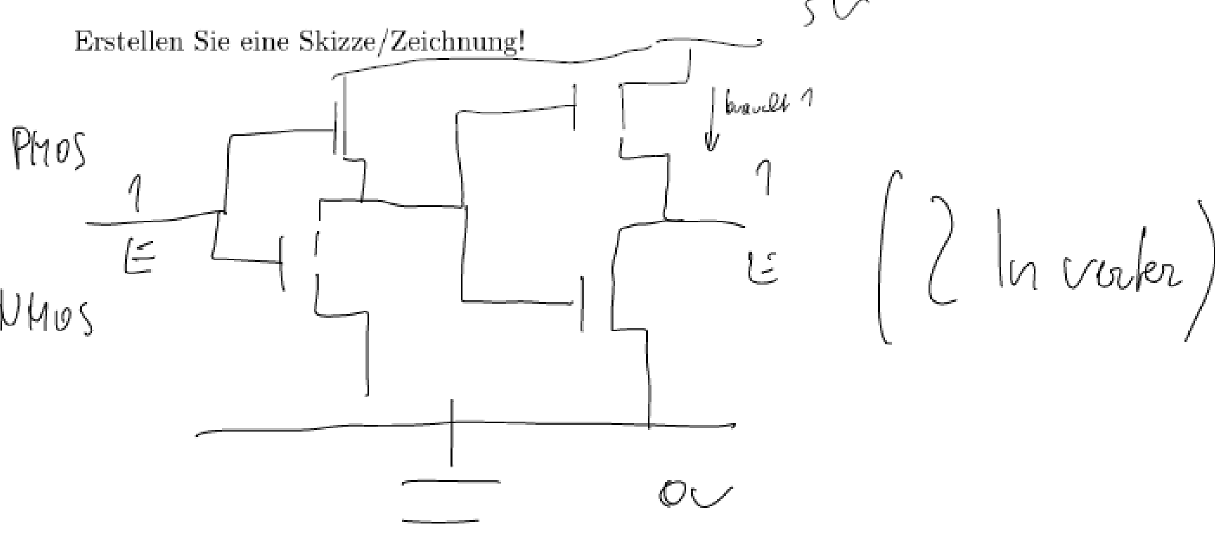
REP-BSP3/4

4. **Unäre Schaltung**

Unäre Schaltungen sind Schaltungen, die nur einen Eingang und einen Ausgang haben (Bsp.: Inverter).

Wie könnte man eine unäre Schaltung (d.h. ein Eingang, ein Ausgang) aus einem NMOS- und einem PMOS-Transistor bauen, die als Funktionalität die Identität realisiert (d.h. die Belegungen am Eingang und Ausgang sind identisch), aber im Gegensatz zum Transfergatter keine direkte Verbindung von Ein- und Ausgang vorsieht?

Erstellen Sie eine Skizze/Zeichnung!



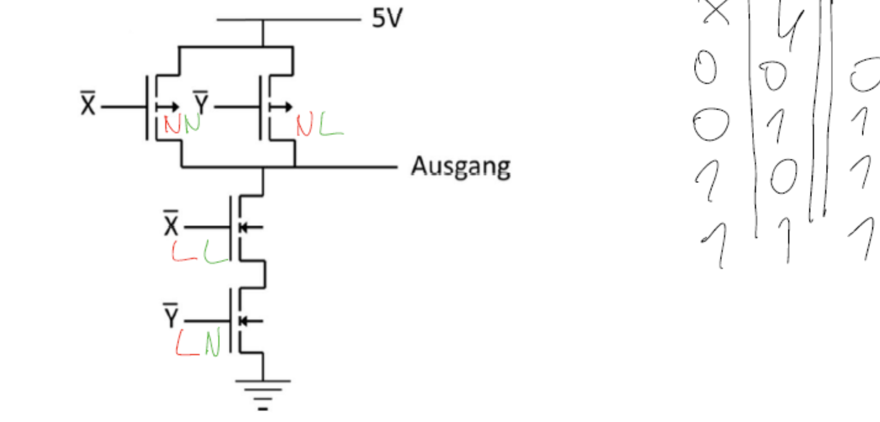
5. **CMOS-Schaltung**

(a) Welcher Boole'schen Relation entspricht die Schaltung? Begründen!

(b) Wo ist der "Pull-up-Pfad" und wo ist der "Pull-down-Pfad" und woran erkennen Sie das?

(c) Markieren Sie für die Eingangsbelegungen X=0, Y=0 bzw. X=0, Y=1 alle leitenden Transistoren mit einem "L" bzw. alle nicht leitenden mit einem "N";

(d) Welche Spannung liegt in diesen beiden Fällen jeweils am Ausgang an?



a) or (oder)

b) oben ist pull up
unten ist pulldown
weil pull up = PMOS
pull down = NMOS

d) bei X, Y = 0 => liegt 0 an
X = 0, Y = 1 => liegt 1 am Ende an

Gatter Übersicht

Gatter	Funktion	IEC-Symbol	ANSI-Symbol	DIN-Symbol	Wahrheitstabelle
Und-Gatter (AND)	$Y = A \wedge B$ $Y = A \cdot B$ $Y = A\,B$ $Y = A \& B$	A, B — & — Y	A, B — out	A, B — Y	A B Y 0 0 0 0 1 0 1 0 0 1 1 1
Oder-Gatter (OR)	$Y = A \vee B$ $Y = A + B$	A, B — ≥1 — Y	A, B — out	A, B — Y	A B Y 0 0 0 0 1 1 1 0 1 1 1 1
Nicht-Gatter (NOT)	$Y = \overline{A}$ $Y = \neg A$ $Y = \tilde{A}$	A — 1 — Y	A — out	A — Y	A Y 0 1 1 0
NAND-Gatter (NICHT UND) (NOT AND)	$Y = \overline{A \wedge B}$ $Y = A \overline{\wedge} B$ $Y = \overline{A\,B}$ $Y = A \mid B$	A, B — & — Y	A, B — out	A, B — Y	A B Y 0 0 1 0 1 1 1 0 1 1 1 0
NOR-Gatter (NICHT ODER) (NOT OR)	$Y = \overline{A \vee B}$ $Y = A \overline{\vee} B$ $Y = \overline{A + B}$ $Y = A - B$	A, B — ≥1 — Y	A, B — out	A, B — Y	A B Y 0 0 1 0 1 0 1 0 0 1 1 0
XOR-Gatter (Exklusiv-ODER, Antivalenz) (EXCLUSIVE OR)	$Y = A \underline{\vee} B$ $Y = A \oplus B$	A, B — =1 — Y	A, B — out	A, B — Y oder A, B — Y	A B Y 0 0 0 0 1 1 1 0 1 1 1 0
XNOR-Gatter (Exklusiv-Nicht-ODER, Äquivalenz) (EXCLUSIVE NOT OR)	$Y = \overline{A \underline{\vee} B}$ $Y = A \overline{\underline{\vee}} B$ $Y = \overline{A \oplus B}$ $Y = A \odot B$	A, B — =1 — Y	A, B — out	A, B — Y oder A, B — Y	A B Y 0 0 1 0 1 0 1 0 0 1 1 1

Gatter

Gatter verändert logischen Zustand

Gatter

- **Haben einen/mehrere Eingänge und einen Ausgang.**
- **Wert des Ausgangs hängt von den Werten der Eingänge und des Gattertyps ab.**

Gatter mit mehreren Eingängen

- **AND**: liefert '1' wenn alle Eingänge '1' sind
- **NAND**: liefert '0' wenn alle Eingänge '1' sind
- **OR**: liefert '1' wenn mindestens ein Eingang '1' ist
- **NOR**: liefert '0' wenn mindestens ein Eingang '1' ist
- **XOR ("Antivalenz")**: liefert '1' wenn eine ungerade Anzahl von Eingängen '1' ist

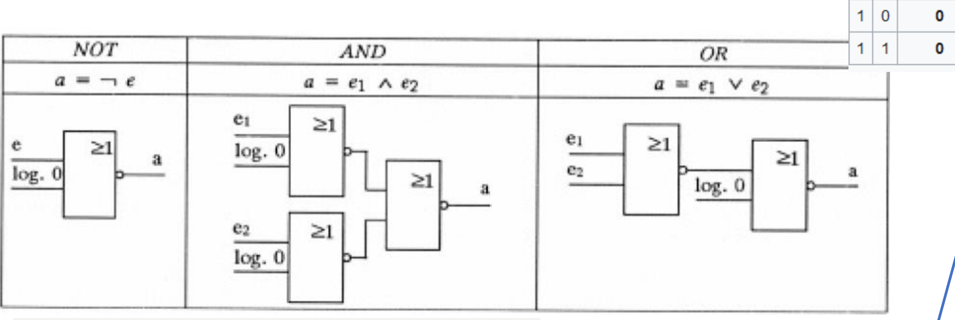
Gatter mit einem Eingang

- **NOT**: liefert '1' wenn der Eingang '0' ist
- Buffer: liefert '1' wenn der Eingang '1' ist

Funktion	Aktuelle Darstellung	DIN alt	USA alt
NOT	1		
AND	&		
OR	≥1		
Antivalenz	=1	⊕	

NOR: Logisch vollständig

A	B	Y = A ⊽ B
0	0	1
0	1	0
1	0	0
1	1	0



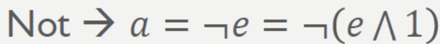
Not → $a = \neg e = \neg(e \vee 0)$

And → $a = e_1 \wedge e_2 = \neg(\neg(e_1 \vee 0) \vee \neg(e_2 \vee 0))$

Or → $a = e_1 \vee e_2 = \neg(\neg(e_1 \vee e_2) \vee \neg(e_2 \vee 0))$

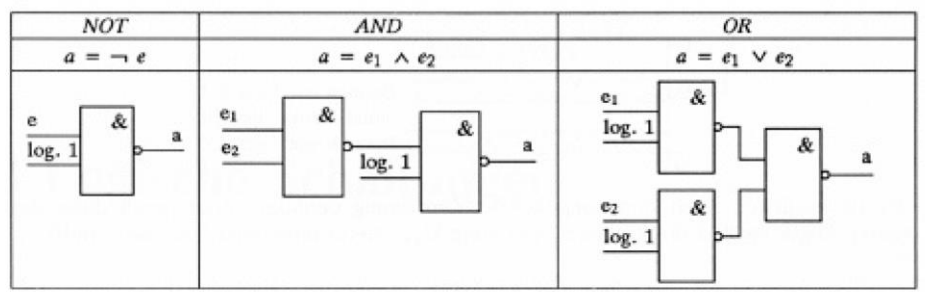
← wichtig →

NAND: Logisch vollständig



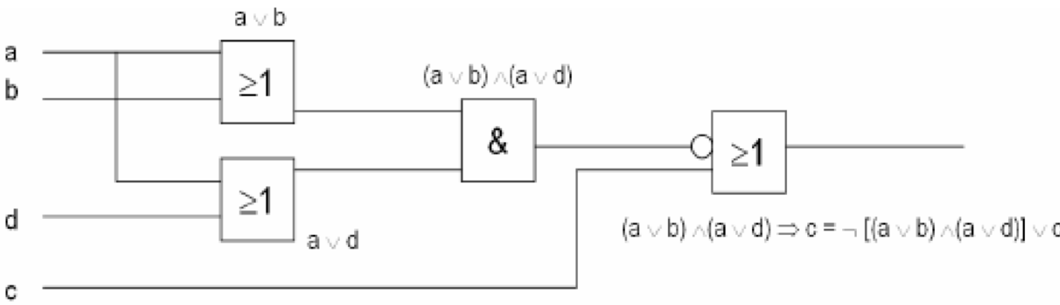
And → $a = e_1 \wedge e_2 = \neg(\neg(e_1 \wedge e_2) \wedge 1)$

Or → $a = e_1 \vee e_2 = \neg(\neg(e_1 \wedge 1) \wedge \neg(e_2 \wedge 1))$



A	B	Y = A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Gatterschaltung



DNF

$a = f(e_1, e_2, e_3) =$
$(\neg e_1 \wedge \neg e_2 \wedge \neg e_3) \vee (\neg e_1 \wedge \neg e_2 \wedge e_3) \vee (\neg e_1 \wedge e_2 \wedge \neg e_3)$

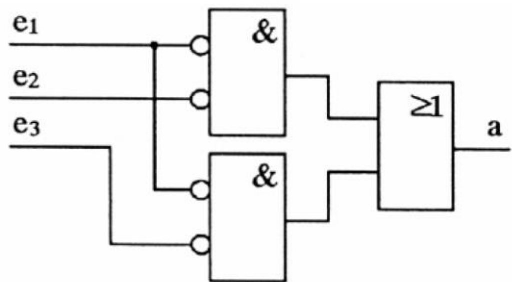
	e_1	e_1	$\neg e_1$	$\neg e_1$
e_3	0	0	1	0
$\neg e_3$	0	0	1	1
	e_2	$\neg e_2$	$\neg e_2$	e_2

Minimiert

$a = f(e_1, e_2, e_3) = (\neg e_1 \wedge \neg e_2) \vee (\neg e_1 \wedge \neg e_3)$

Minimiert

$a = f(e_1, e_2, e_3) = (\neg e_1 \wedge \neg e_2) \vee (\neg e_1 \wedge \neg e_3)$



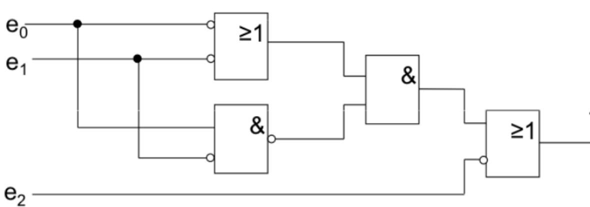
BSP-Rep1

1. **Gatterschaltung**

   (a) Für folgenden Ausdruck soll eine Schaltung erstellt werden. Formen Sie zuerst den Term so um, dass weder Äquivalenz noch Implikation vorkommen, und zeichnen Sie danach die Schaltung.

   $$f(a,b,c) = (a \wedge b) \Rightarrow (a \Leftrightarrow c)$$

   (b) Reduzieren Sie die Anzahl der Schaltelemente (d.h. lesen Sie zuerst den Ausdruck der Schaltung ab, vereinfachen Sie diesen Ausdruck und zeichnen Sie die vereinfachte Schaltung).



a	b	c	$a \wedge b$	$a = c$	$\rightarrow$
0	0	0	0	1	1
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	0	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

1.

$(a \wedge b) \rightarrow (a \Leftrightarrow c)$

$\neg(a \wedge b) \vee (a \Leftrightarrow c)$

$\neg(a \wedge b) \vee [a \rightarrow c \wedge c \rightarrow a]$

$\neg(a \wedge b) \vee [(\neg a \vee c) \wedge (\neg c \vee a)]$

2

$\neg(a \wedge b) \vee [a \rightarrow c \wedge c \rightarrow a]$

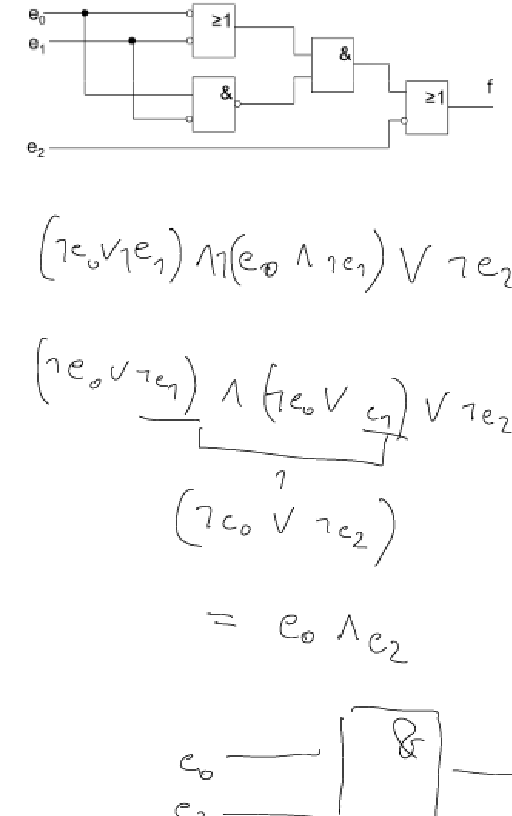
$\neg a \vee \neg b \vee ((\neg a \vee c) \wedge (\neg c \vee a))$

$\neg a \vee (\neg a \vee c) \wedge \neg a \vee (\neg c \vee a) \vee \neg b$

$(\neg a \vee c) \wedge 1 \vee \neg b$

$\neg a \vee c \vee \neg b$

(b) Reduzieren Sie die Anzahl der Schaltelemente (d.h. lesen Sie zuerst den Ausdruck der Schaltung ab, vereinfachen Sie diesen Ausdruck und zeichnen Sie die vereinfachte Schaltung).



Halbaddierer

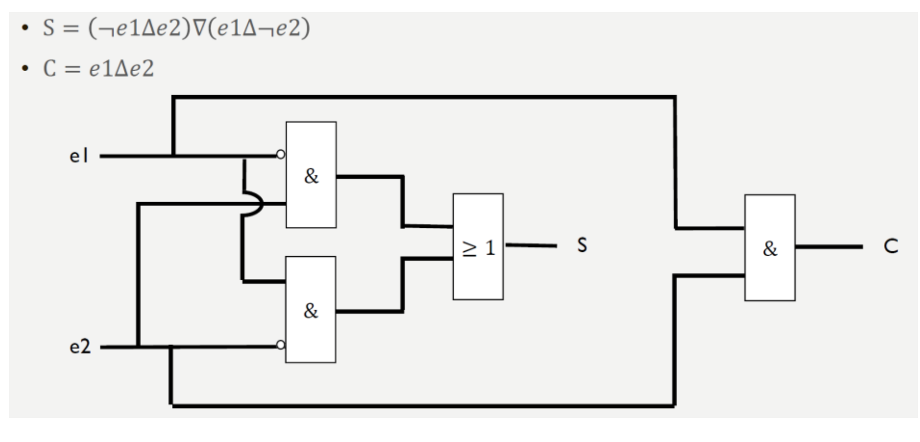
Halbaddierer

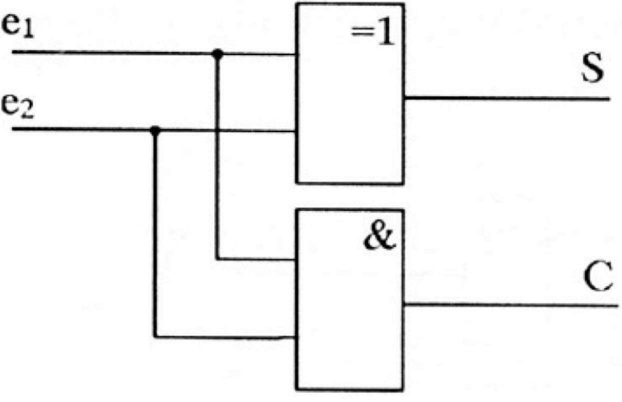
- Addiert zwei **einstellige** Binärzahlen $\Rightarrow$ 2 Eingänge
- Zwei Ausgänge: Summe (S), Carry (C)

Wahrheitstabelle

e_1	e_2	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- **S entspricht XOR**
- **C entspricht AND**



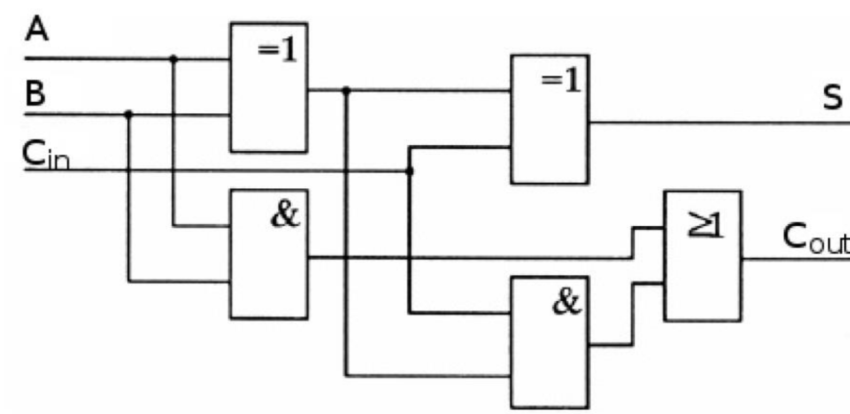


- geht auch nur mit AND und OR
- einfache Rechenzeitabschätzung für diesen Fall: Annahme 10*psec* pro Gatter $\rightarrow$ gesamter HA: 30*psec* (kritisch: XOR-Gatter, braucht 30*psec*)

Volladdierer

Minimiert

- $S = A \bigoplus B \bigoplus C_{in}$
- $C_{out} = (A \wedge B) \vee (C_{in} \wedge (A \bigoplus B))$

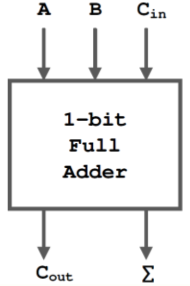


- Abschätzung: zwei HA + OR-Gatter $\rightarrow$ 70*psec* gesamt
- etwas genauer: zwei XOR-Gatter für S $\rightarrow$ 60*psec* gesamt

Evlt auch nur mit und / oder darstellen

1-bit Volladdierer

- EIN 1-bit Volladdierer addiert **drei einstellige** Binärzahlen $\Rightarrow$ 3 Eingänge
- drei Eingänge: *A*, *B*, und Carry in C_{in}
- C_{in} entspricht dem “Eingangsübertrag”
- zwei Ausgänge: Summe (S bzw. $\sum$), Carry out (C_{out})



Wahrheitstabelle eines 1-bit Volladdierers

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

BSP-REP2

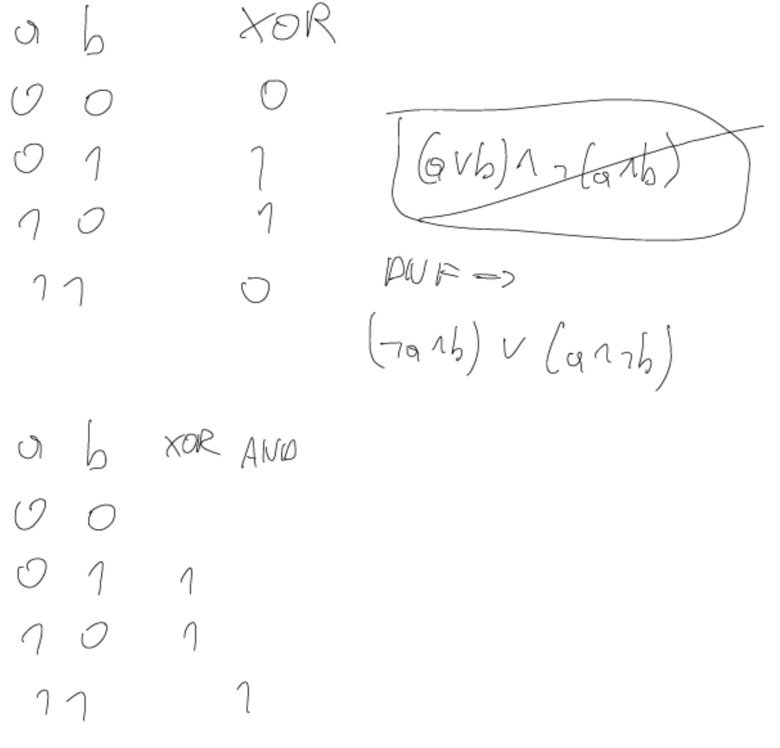
2. Addierer

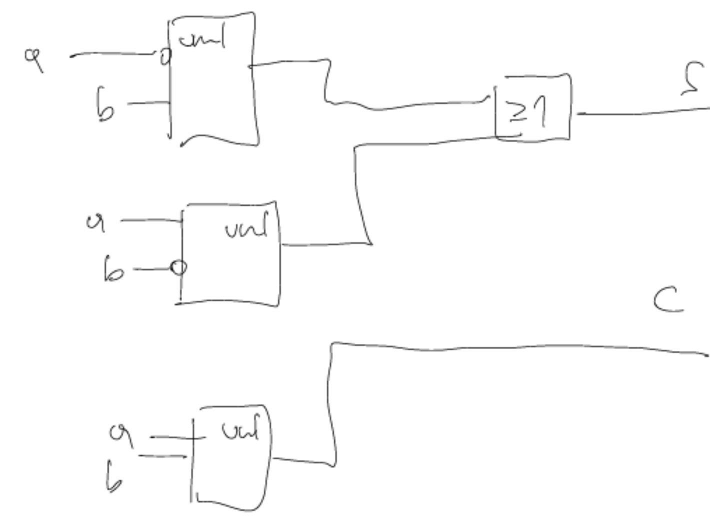
(a) XOR: Erstellen Sie die Wahrheitstabelle und ein KV Diagramm für die XOR-Relation mit den Eingangsvariablen a und b, und lesen Sie die minimale DNF ab.

(b) Halbaddierer: Zeichnen Sie die Schaltung eines Halbaddierers OHNE XOR Gatter

(c) Worin unterscheidet sich der Volladdierer vom Halbaddierer?

(d) Volladdierer: Erstellen Sie die Wahrheitstabelle für die Addition von 2 einstelligen Binärzahlen unter Berücksichtigung eines Carry_{in} (C_{in}). Erstellen Sie sowohl für Summe (S) als auch für Carry_{out} (C_{out}) jeweils die minimierte DNF und zeichnen Sie die dazugehörige Schaltung (Sie können auch jeweils eine Schaltung für C_{out} und eine für S zeichnen).





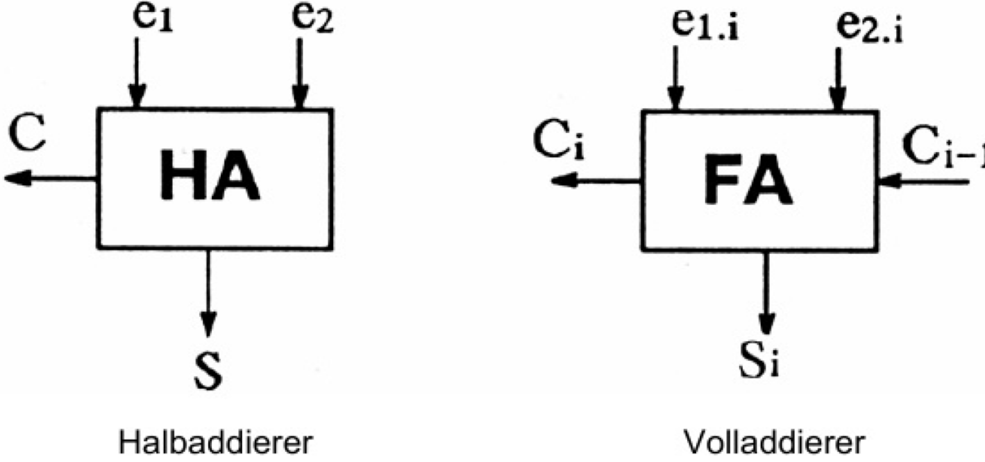
=> Volladdierer hat übertrag input

d) siehe Folien

Blockschaltbild

Blockschaltbilder

- Eingänge, Ausgänge, Funktion
- nur Funktionalität, nicht innere Struktur (Blackbox)
- enthalten Module, die wieder kombiniert werden können



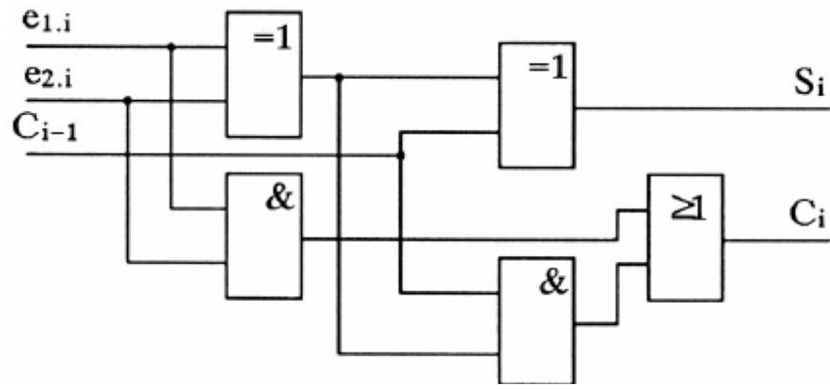
Halbaddierer

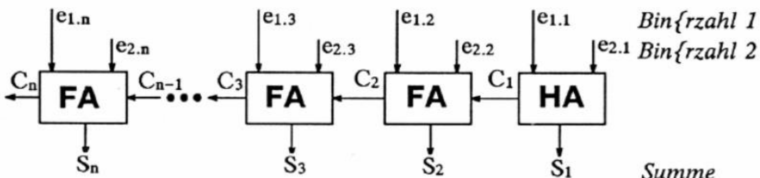
Volladdierer

Carry Ripple Addierer

Minimiert

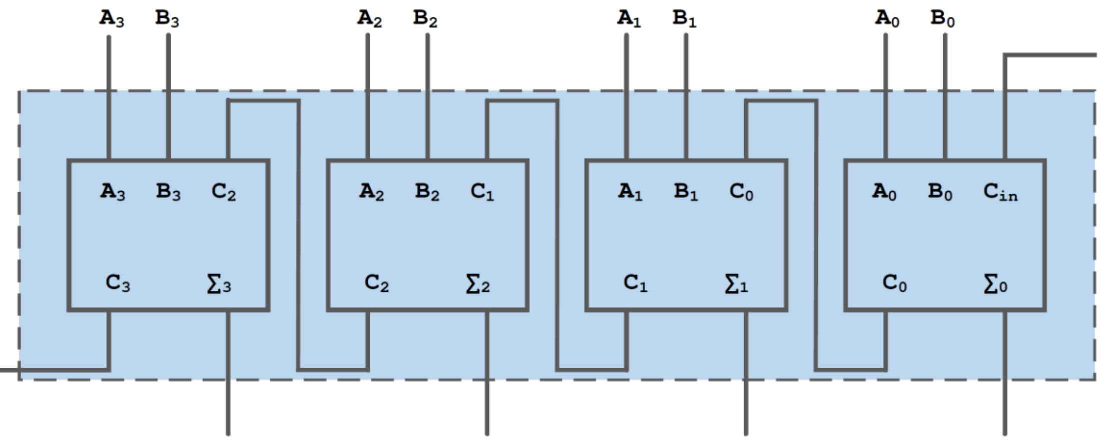
- $S_i = e_{1i} \oplus e_{2i} \oplus C_{i-1}$
- $C_i = (e_{1i} \land e_{2i}) \lor (C_{i-1} \land (e_{1i} \oplus e_{2i}))$





Paralleladdierer

- Jeder Addierer braucht für Berechnung Carry-Information von voriger Einheit.
- **Worst case: Carry-Bit muss die ganze Schaltung durchlaufen!** (*carry propagation*)
- Abschätzung: 3 FA + HA: $3 \cdot 60 psec + 30 psec = 210 psec$



Wahrheitstabelle eines Carry-Ripple-Addierers

e_{1i}	e_{2i}	C_{i-1}	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

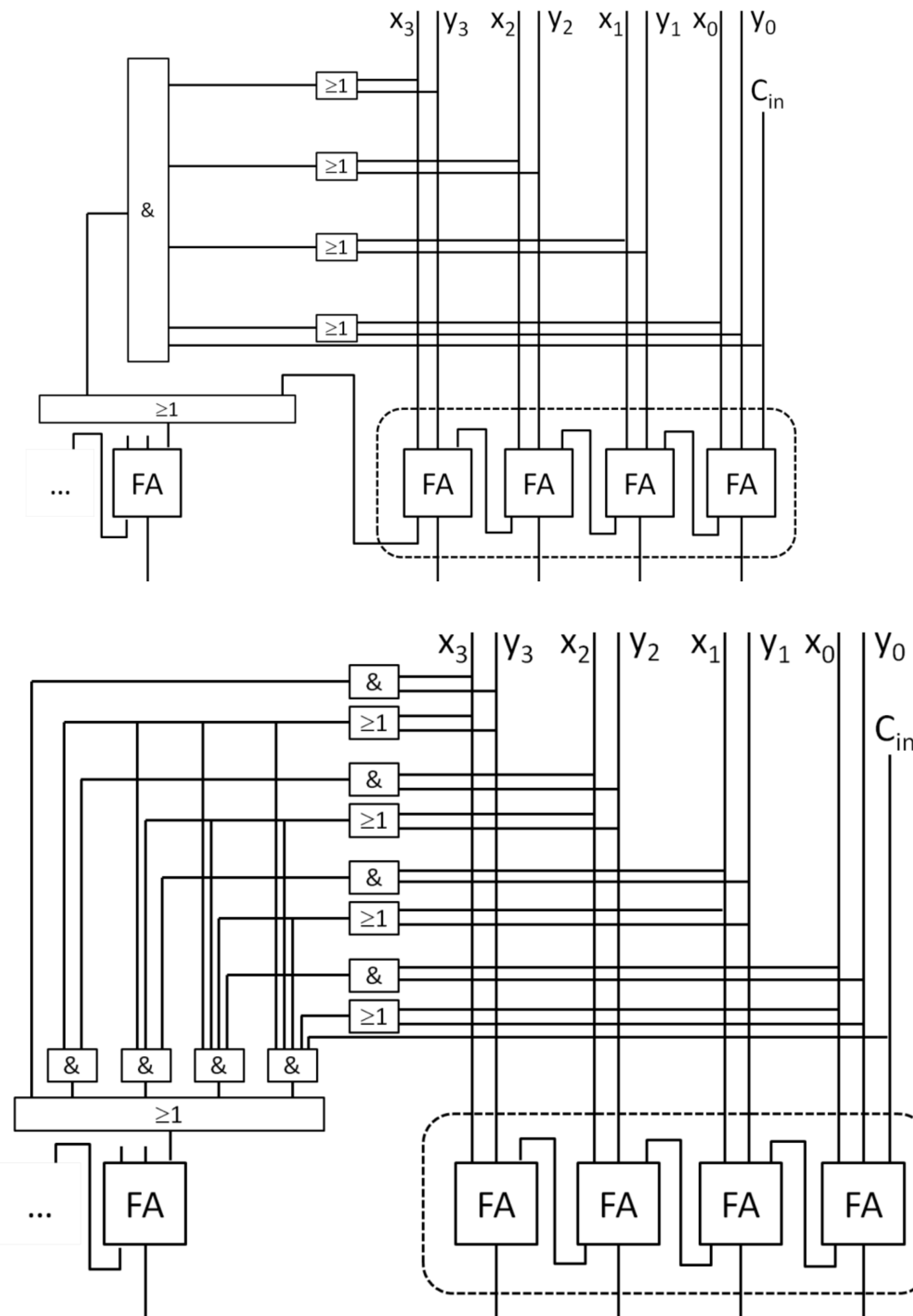
Optimierung

Carry-Skip-Addierer

- Volladdierer werden gruppiert
- Zusatzlogik untersucht, ob sich ein Übertrag durch gesamte Gruppe propagiert
- Wenn ja, wird dies der nächsten Gruppe gemeldet, damit diese ebenfalls mit der Berechnung anfangen kann.

Carry-Look-Ahead-Addierer

- Überträge werden bereits im ersten Additionsschritt ermittelt
- Größerer Schaltungsaufwand nötig
- Nur für kleine Wortbreiten effektiv



4-Bit-Matrixmultiplizierer

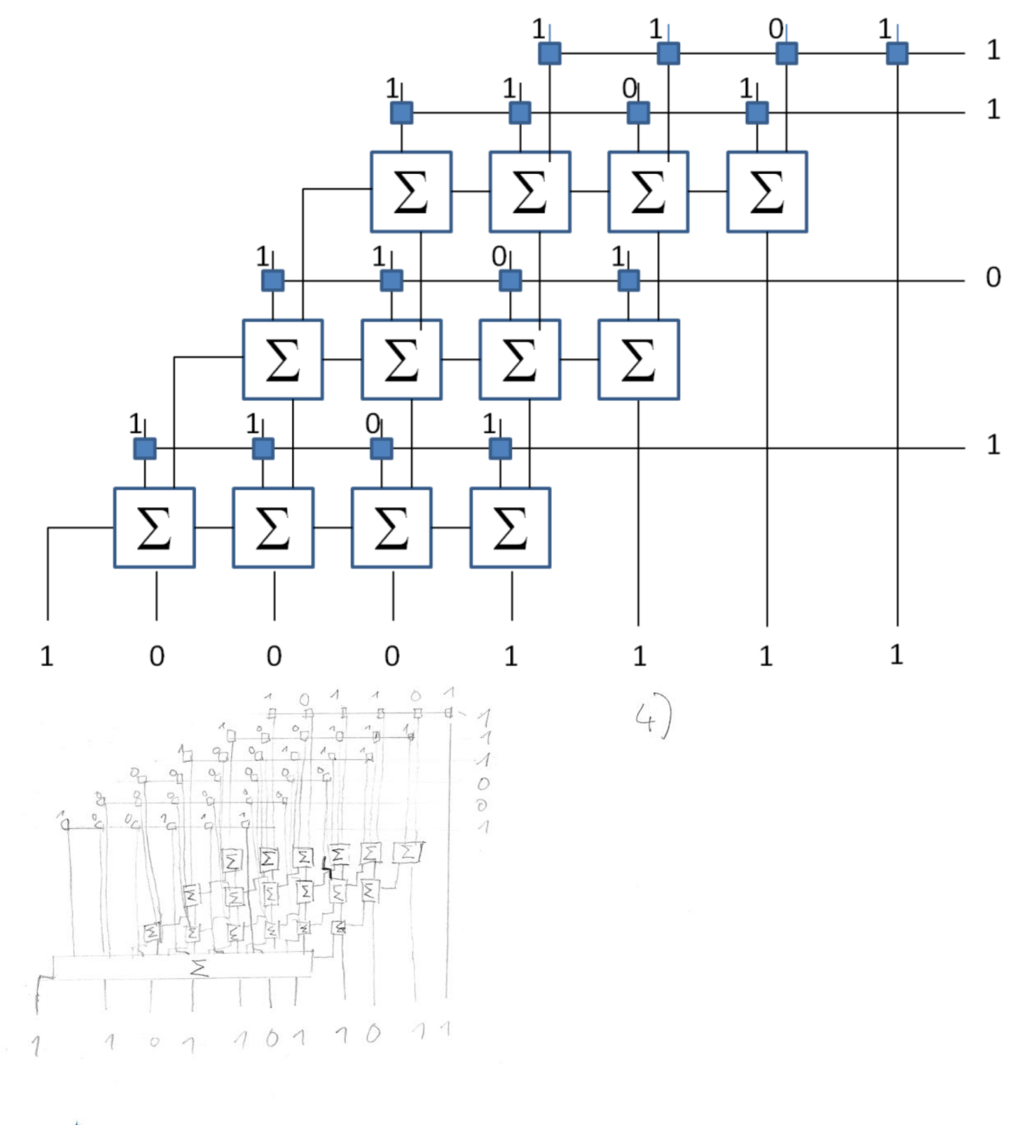
Multiplizierer

Idee: Standardmethode aus der Schule

- Beispiel mit Binärzahlen:

				1	1	0	1	x 1 0 1
				1	1	0	1	1
			1	1	0	1		1
		0	0	0	0			0
	1	1	0	1				1
1	0	0	0	1	1	1	1	

- "Multiplikation" von 1-Bit-Zahlen = AND-Gatter
- Matrixstruktur
- Matrixmultiplizierer ⇒ Schalttiefe = $O(n)$



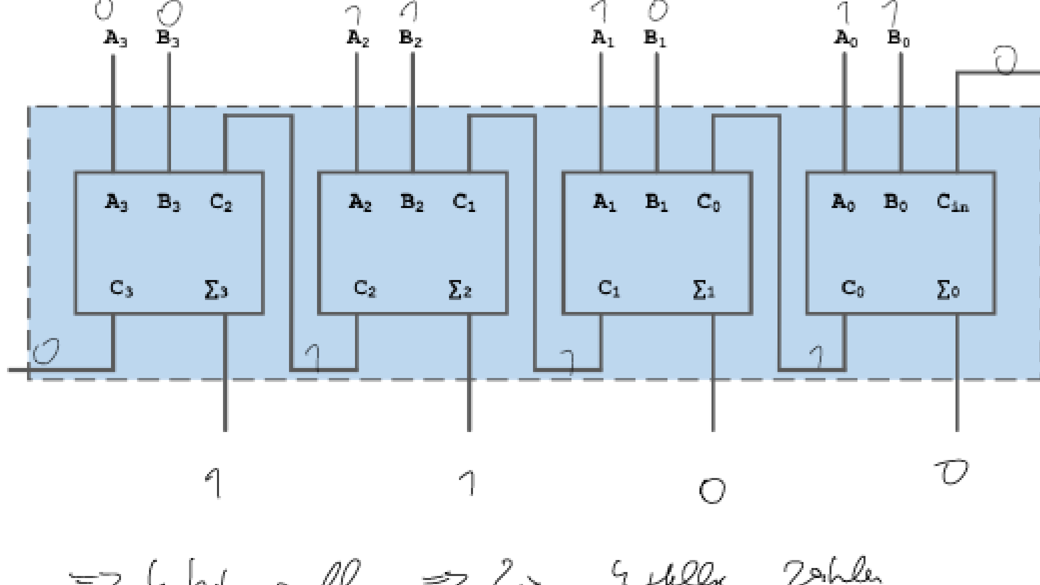
die Summe wird 3-mal mit Volladdierern und schließlich einem Carry-Ripple-Addierer berechnet → Tiefe 4

BSP-REP3/4

3. Carry-Ripple-Addierer

Erklären Sie anhand des folgenden Carry-Ripple-Addierers, wie dieser Addierer die beiden Binärzahlen $A = 0111_2$ ($A_3 = 0, A_2 = 1, A_1 = 1, A_0 = 1$) und $B = 0101_2$ addiert. C_{in} soll in diesem Fall 0 sein, es gibt also keinen eingehenden Übertrag.

Wie viele Zahlen kann dieser Carry-Ripple-Addierer in einem Durchlauf addieren? Wie viele Stellen können diese Zahlen haben?



=> 4 bit voll => 2x 4 stellige Zahlen

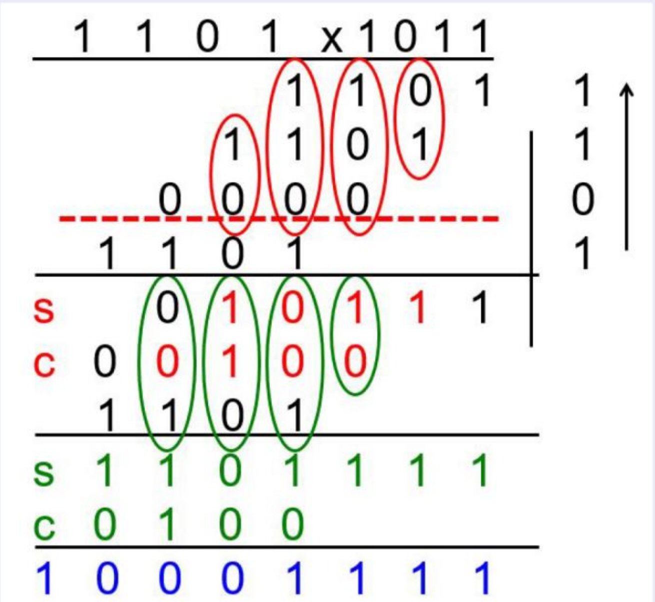
4. Multiplizierer

Ein Wallace-Tree-Multiplizierer für die Multiplikation zweier **vier**stelliger Binärzahlen hat die Schalttiefe 3 (zwei Verdichtungsschichten Volladdierer plus der abschließende Carry-Ripple-Addierer).

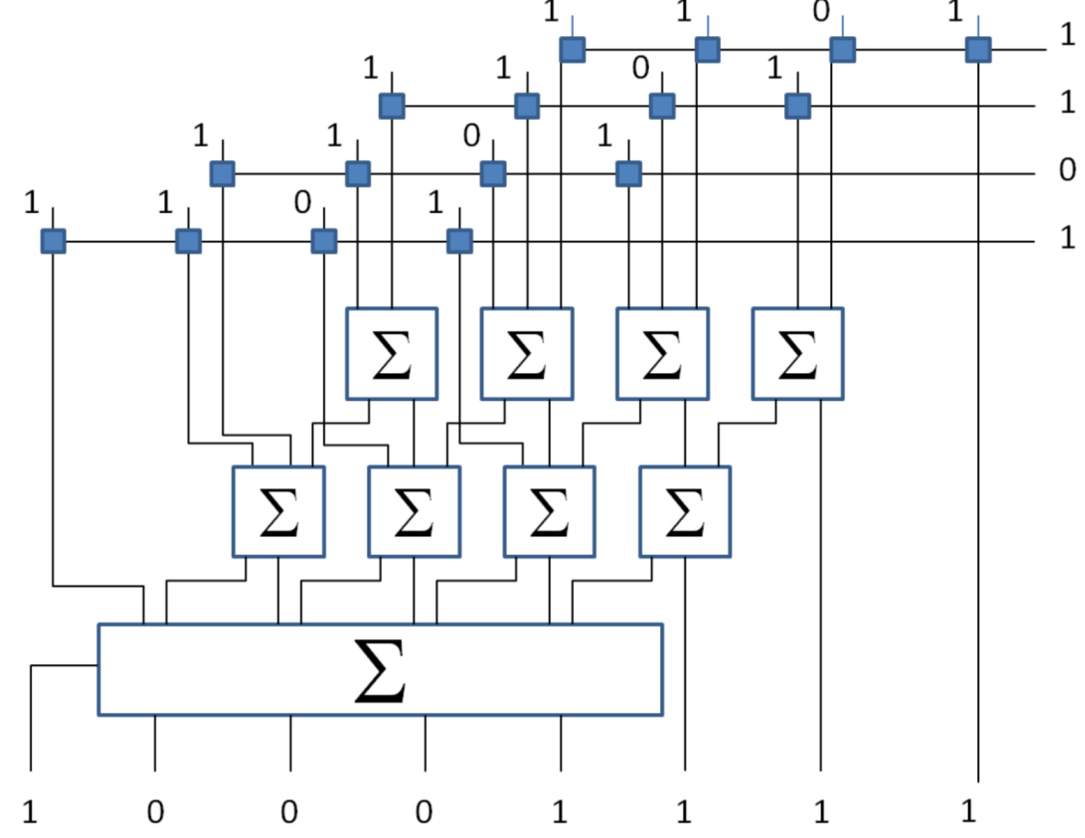
Erklären Sie anhand einer Skizze/Zeichnung, warum die Multiplikation von zwei **sechs**stelligen Binärzahlen nur eine Verdichtungsschicht mehr braucht, also eine Schalttiefe von 4 hat.

Wallace-Tree-Multiplizierer

Idee: Volladdierer addiert drei Bit ⇒ addiere jeweils drei Zeilen



- Ergebnis: drei Zeilen werden zu zwei Zeilen verdichtet
- Parallelität ⇒ Schalttiefe = $O(\log n)$



Multiplexer

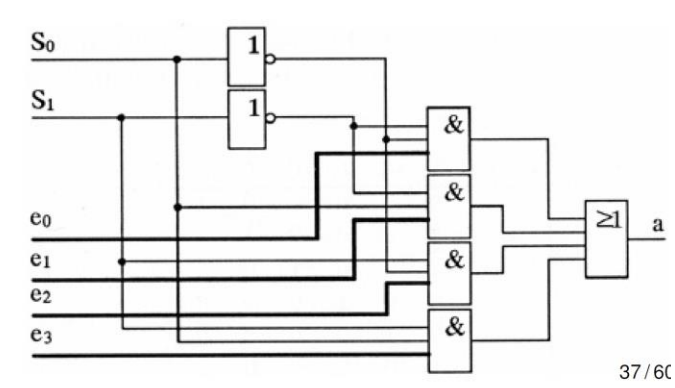
Multiplexer

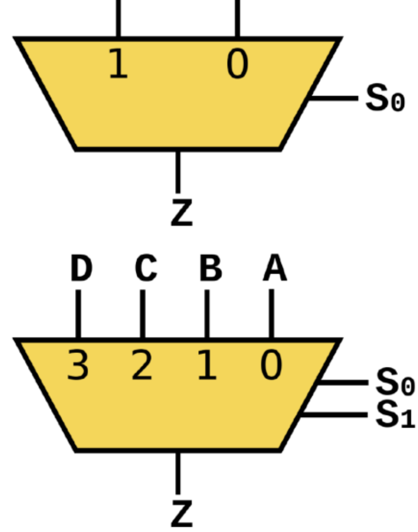
- $2^n = m$ **Dateneingänge**
- $n = log_2(m) = ld(m)$ Steuereingänge
- ***Ein*** **Datenausgang – Multiple input, single output!**
- **Der über die Steuereingänge gewählte Dateneingang wird unverändert zum Datenausgang geleitet**
- Umkehrung: **De**multiplexer

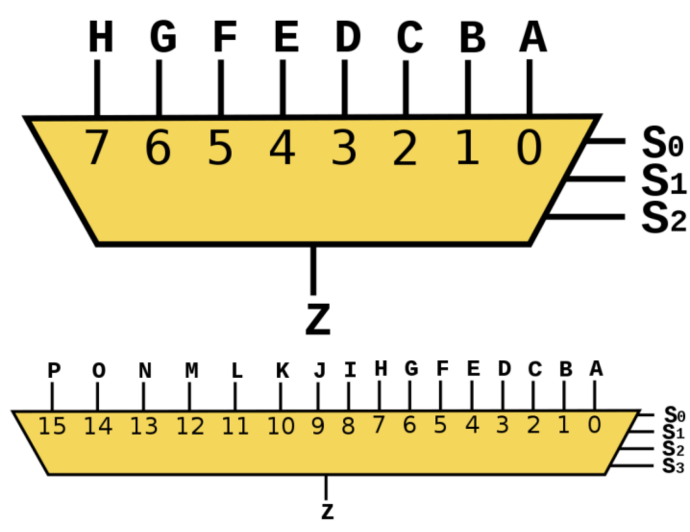
Bsp: 4 to 1 MUX

- Angenommen: $(S_1 S_0) = (10)$
- Nur das AND-Gatter mit e_2 kann geschalten sein
- $\Rightarrow e_2$ wird auf a durchgeschalten

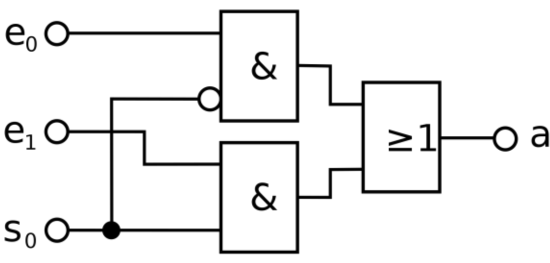
S_1	S_0	a
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

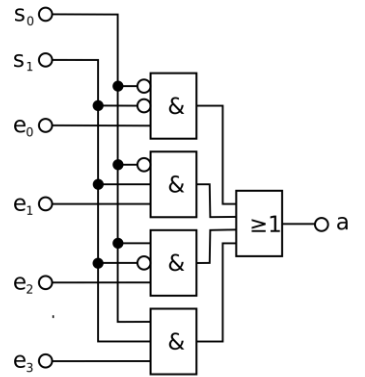






Multiplexer





Multiplexer: Anwendungen

Allgemeines Prinzip

- **Parallelen Datenstrom in seriellen Datenstrom umwandeln**
- Per Demultiplexer wieder rückwandeln
- Anstatt vielen Leitungen zwischen zwei Einheiten
- D.h.: Multiplexed + Steuerleitungen $<$ Leitungen

I/O

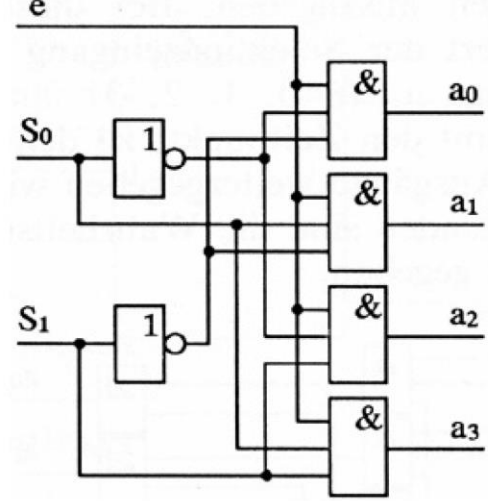
Z.B. Tastatur: Jede Taste wird per 7-bzw. 8-Bit codiert, bei einem Anschlag werden die Bits aber nicht parallel, sondern **seriell (hintereinander) über eine einzige Leitung übertragen.**

Demultiplexer

Prinzip

Steuereingänge S_i bestimmen, auf welchen der **Ausgänge** a_i der **Eingang** e durchgeschaltet wird

S_1	S_0	a_0	a_1	a_2	a_3
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e



Decoder

Wichtige Anwendung: Instruction Decoder (CPU)

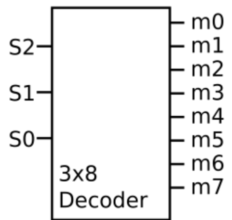
- Instruction Decoder (CPU)
- **Wandelt die Bits des Instruktionsregisters in Kontrollsignale um, die andere Teile der CPU steuern**
- Bei Mikrocode-basierenden CPUs wird die Mikroinstruktion umgewandelt (dekodiert)

Simples Beispiel

- CPU mit 8 Registern
- 3-zu-8 Logik Decoder
- 2 Quellregister als Input für die ALU
- 1 Zielregister als Output für die ALU

Decoder

- **Multiple input, multiple output**
- Wandelt kodierten Input in kodierten Output um, wobei Input-Code und Output-Code unterschiedlich sind!
- n-zu-2^n Decoder: n Eingänge, 2^n Ausgänge
- Zu jeder Zeit ist nur ein Ausgang aktiv
- Umkehrung: Encoder (Kodierer)



e_2	e_1	e_0	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Abbildung: 3-zu-8 Decoder

Wichtige Anwendung: Random-access memory

- Adresse, die über Adress-Bus hereinkommt, wird in Zeilen- und Spalten-Adressen umgewandelt

Simples Beispiel

- Speicher[a] aus 8 Chips mit je 1MiByte[b], Chip 0 = Adressen 0-1MiByte, Chip 1 = Adressen 1-2 MiByte, etc.
- 3 höchstwertigen Bits der Speicheradresse geben zu wählenden Chip an
- 3-zu-8 Logik Decoder

Enable-Eingang

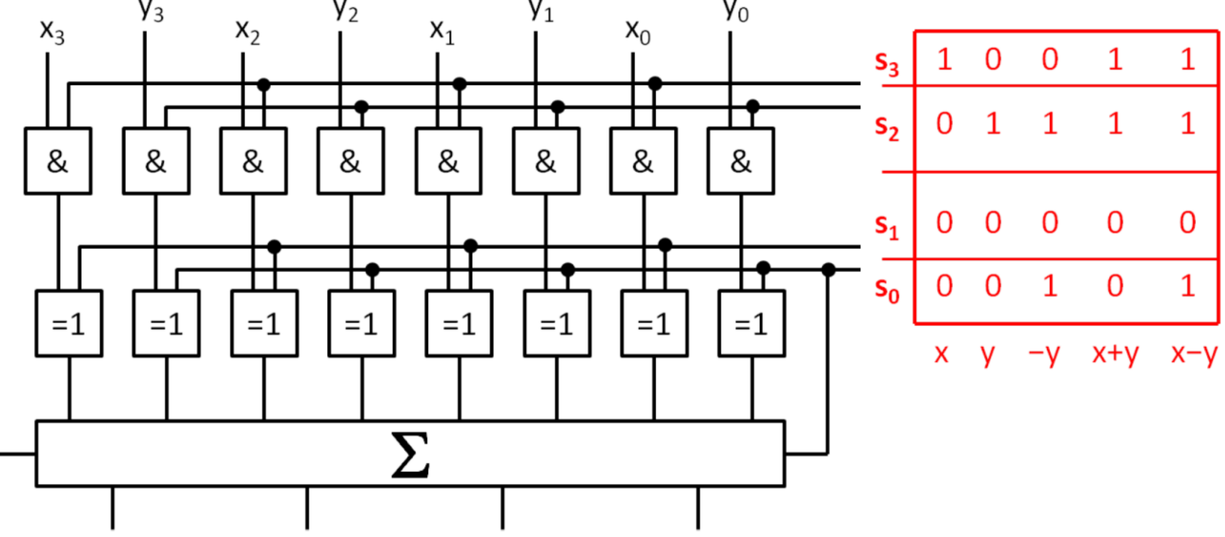
- Nur wenn enabled, kann ein Ausgang überhaupt aktiv sein
- Bsp: 2-zu-4 Decodierer mit Enable

E	e_1	e_0	a_3	a_2	a_1	a_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

ALU

ALU – Rechen-/Operationswerk

- Führt **logische und arithmetische** Operationen aus
- Wird vom Steuerwerk nach Dekodierung einer entspr. Instruktion angesprochen
- Verfügbare Operationen unterschiedlich, z.B.:
  - $a \wedge b$
  - $\neg b$
  - $a + b$
  - $a + 1$
  - $a - b$
  - bit shift left/right
  - ...



AND-Gatter:

S, D → &

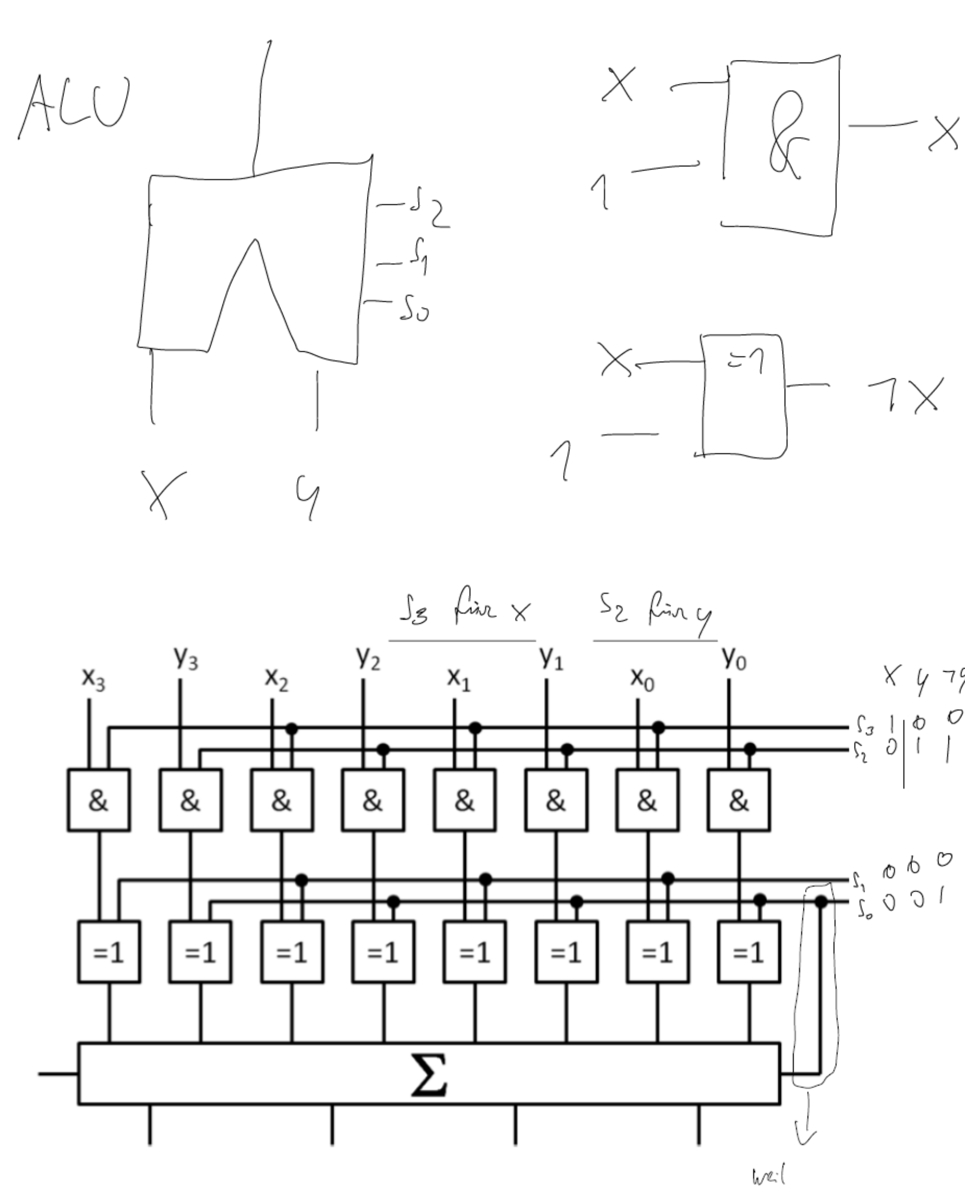
Steuer-leitung	Daten-leitung	Ausgang	Funktionalität
0	0	0	ausgeschaltet
0	1	0	ausgeschaltet
1	0	0	transparent
1	1	1	transparent

XOR-Gatter:

S, D → =1

Steuer-leitung	Daten-leitung	Ausgang	Funktionalität
0	0	0	transparent
0	1	1	transparent
1	0	1	invertiert
1	1	0	invertiert

ALU add



BSP-REP 5/6

5. **Multiplexer**

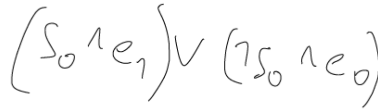
(a) Erklären Sie das allgemeine Funktionsprinzip eines Multiplexers, inkl. Eingangsleitungen (e_x), Steuerleitungen (s_x), Ausgangsleitung (a).

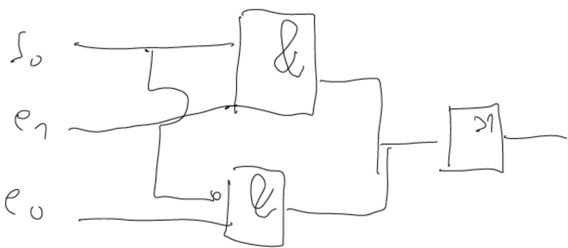
(b) Erstellen Sie mit Hilfe folgender Tabelle die minimale DNF eines 2 to 1 Multiplexers und zeichnen Sie die entsprechende logische Schaltung

s_0	e_1	e_0	a	Kommentar
0	0	0	0	Eingang 0 ist ausgewählt
0	0	1	1	Eingang 0 ist ausgewählt
0	1	0	0	Eingang 0 ist ausgewählt
0	1	1	1	Eingang 0 ist ausgewählt
1	0	0	0	Eingang 1 ist ausgewählt
1	0	1	0	Eingang 1 ist ausgewählt
1	1	0	1	Eingang 1 ist ausgewählt
1	1	1	1	Eingang 1 ist ausgewählt

leitet e_0

leitet e_1





6. **4-Bit-ALU**

Die einfache 4-Bit-ALU aus der Vorlesung (Folie 51) arbeitet mit Zweierkomplement. Welche Operationen führt diese ALU aus, wenn die Steuerleitungen wie folgt belegt sind (es genügt, wenn Sie die vier Summenbits am Ausgang betrachten)?

(a) $s_3 = 0, s_2 = 0, s_1 = 0, s_0 = 0$

(b) $s_3 = 1, s_2 = 0, s_1 = 0, s_0 = 1$

(c) $s_3 = 0, s_2 = 1, s_1 = 1, s_0 = 0$

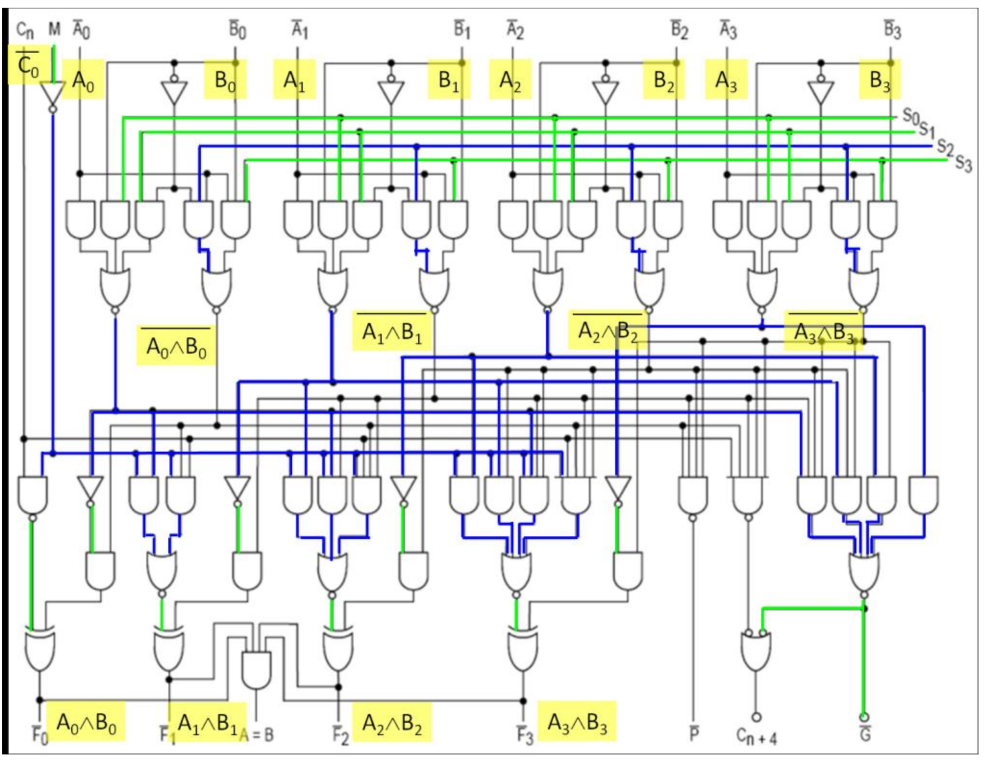
(d) $s_3 = 0, s_2 = 1, s_1 = 1, s_0 = 1$

a) Nichts (0)
b) X-1+1 (1111+x = x-1 (+1 von s0) = x)
c) y-1
d) -y-1 = -(y+1)

Beispiele dazu am besten einfach ausrechnen !

Bonus 74181

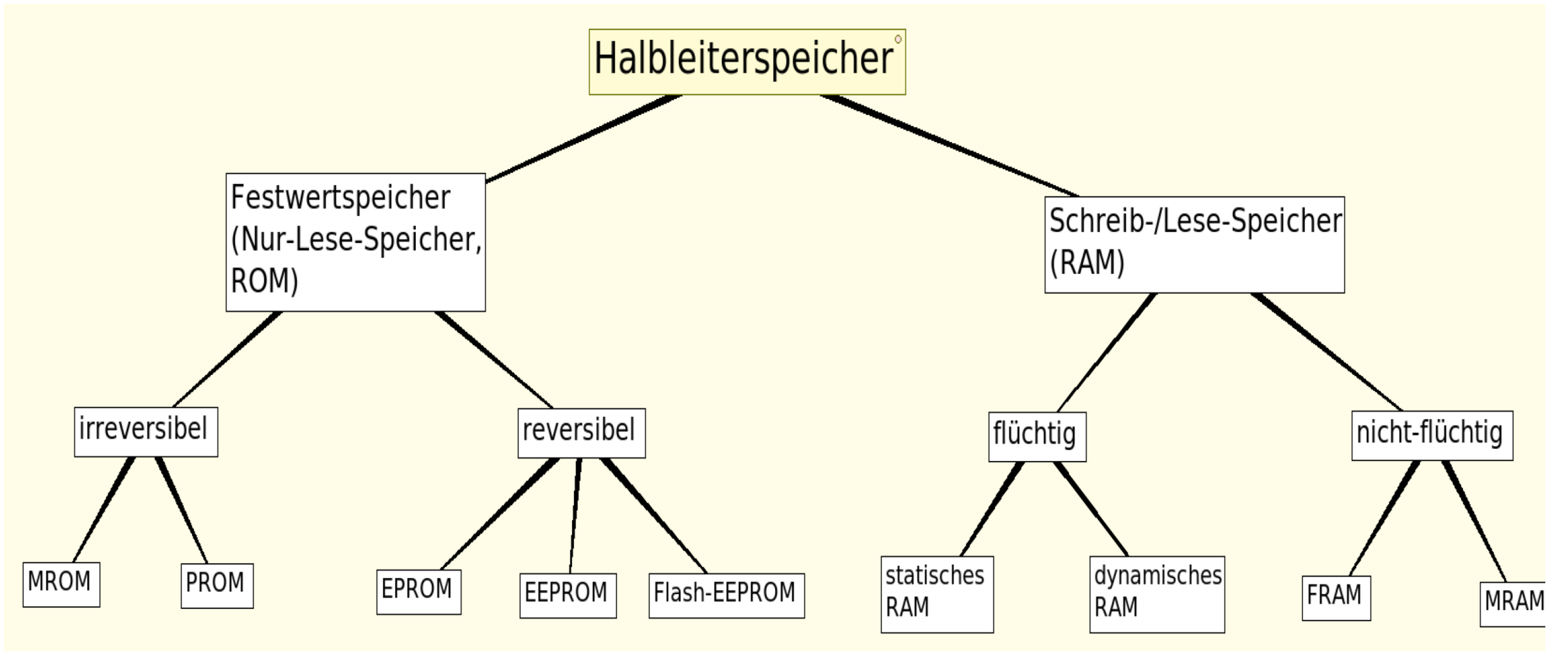
Steuersignale				Ergebnis am Ausgang der ALU		
				M=H logische Operationen	M=L arithmetische Operationen	
S_3	S_2	S_1	S_0		$C_n = H$ (ohne Carry)	$C_n = L$ (mit Carry)
L	L	L	L	$\overline{A}$	A	$A + 1$
L	L	L	H	$\overline{A \vee B}$	$A \vee B$	$(A \vee B) + 1$
L	L	H	L	$\overline{A} \wedge B$	$A \vee \overline{B}$	$(A \vee \overline{B}) + 1$
L	L	H	H	0	-1	0
L	H	L	L	$\overline{A \wedge B}$	$A + A \wedge \overline{B}$	$A + A \wedge \overline{B} + 1$
L	H	L	H	$\overline{B}$	$(A \vee B) + A \wedge \overline{B}$	$(A \vee B) + A \wedge \overline{B} + 1$
L	H	H	L	$A \neq B$	$A - B - 1$	$A - B$
L	H	H	H	$A \wedge \overline{B}$	$A \wedge \overline{B} - 1$	$A \wedge \overline{B}$
H	L	L	L	$\overline{A} \vee B$	$A + A \wedge B$	$A + A \wedge B + 1$
H	L	L	H	$\overline{A \neq B}$	$A + B$	$A + B + 1$
H	L	H	L	B	$(A \vee \overline{B}) + A \wedge B$	$(A \vee \overline{B}) + A \wedge B + 1$
H	L	H	H	$A \wedge B$	$A \wedge B - 1$	$A \wedge B$
H	H	L	L	1	$A + (A\ shl\ 1)$	$A + A + 1$
H	H	L	H	$A \vee \overline{B}$	$(A \vee B) + A$	$(A \vee B) + A + 1$
H	H	H	L	$A \vee B$	$A \vee \overline{B} + A$	$A \vee \overline{B} + A + 1$
H	H	H	H	A	$A - 1$	A



Halbleiter - Speichertypen

Eigenschaften von ROM Bausteinen

Typ	Dauer Schreibvorgang	max. # Schreibvorg.
MROM	Monate	1
(OT)PROM	Minuten	1
EPROM	Minuten	bis zu 100
EEPROM	Millisekunden	$10^4 - 10^6$
Flash	$10\ \mu s$ ($1 \mu s = 10^{-6}$ Sek)	$10^4 - 10^6$



Read Only Memory (ROM)

Festwertspeicher, Nur-Lese-Speicher

- Kann im normalen Betrieb nur gelesen werden
- **Nicht-flüchtig** (non-volatile), Inhalt bleibt erhalten
- Kommen zum Einsatz um **Daten dauerhaft und unabänderlich zu speichern**
- BIOS, Messgeräte, Unterhaltungselektronik, Steuerungen

Arten von ROMs

- MROM: Masken-ROM
- PROM: Programmable ROM
- EPROM: Erasable PROM
- EEPROM (Electrical EPROM) bzw. Flash-Speicher

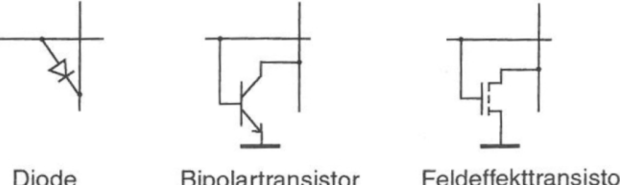
MROM

Masken ROM

- **Schon bei Herstellung des Speicherchips** werden an Kreuzungspunkten zw. Wort-/Bit-Leitung Brücken erstellt

⇒ **Brücke vorhanden: 1; Brücke fehlt: 0**

- Brücke: Diode oder Transistor
- Dateninhalt muss schon **vor** der Herstellung feststehen und wird in Belichtungsmaske eingearbeitet
- Belichtungsmaske **teuer**, lohnt sich nur in großen Stückzahlen



PROM

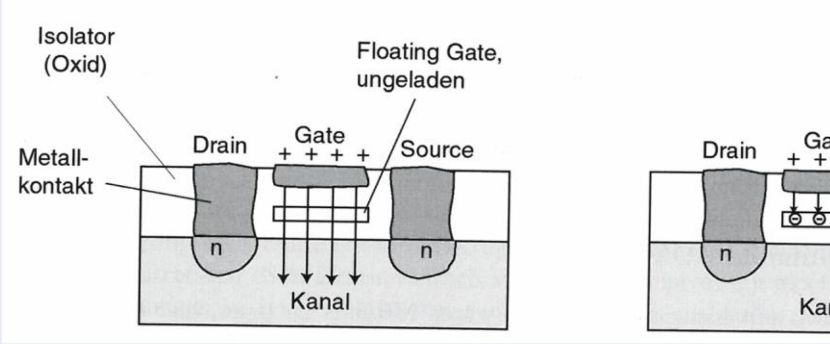
Programmable ROM

- Speicherzellen so gebaut, dass mit einem Programmierimpuls die Brückenfunktion **dauerhaft** hergestellt/unterbrochen wird
- Programmierimpuls: Einmaliger, kräftiger Stromstoß
- Im Originalzustand alle Verbindungen aufrecht
- Gezieltes Zerstören der Verbindungen $\Rightarrow$ **0/1**
- **Lässt sich nicht mehr rückgängig machen!**

$\Rightarrow$ "One Time Programmable ROM" (OTPROM)

- Bei kleinen Stückzahlen, billiger als MROMs

Floating Gate



EPROM

Erasable Programmable ROM

- **Unpraktisch jedes Mal PROM zu opfern, z.B. Testphase**
- **EPROM: Per Bestrahlung mit UV-Licht löschbar**
- Elektronen in Transistoren[a] werden durch Licht-Quanten in ursprüngliche Position gebracht
- Löschen dauert ca. 20min (100 bis 200 Mal, danach kaputt)
- Quarzfenster (teuer)
- Auch Tageslicht enthält UV-Anteile, gute Abdeckung nötig
- **Etwas umständliche Löschprozedur $\Rightarrow$ EEPROM**

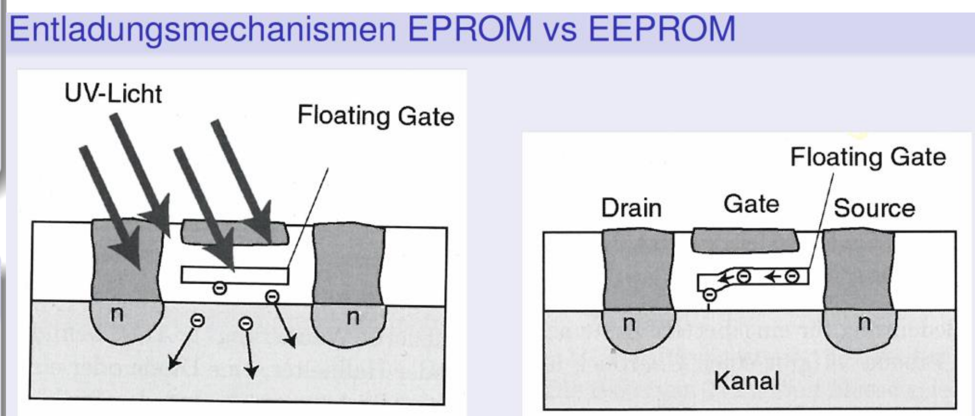
---

[a] *Floating Gate*

EEPROM

Electrical Erasable Programmable ROM

- Floating Gate des Transistors kann **elektrisch** entladen (und damit neu programmiert) werden
- Anzahl der Schreibzyklen ist begrenzt: ca. 10^4 bis 10^6 Zyklen



NVRAM (non-volatile RAM)

- Alle Speicherzellen doppelt vorhanden: Als RAM- und EEPROM-Zelle
- RAM für normalen Betrieb
- Vor Abschalten der Betriebsspannung in EEPROM sichern
- $\Rightarrow$ Nicht-flüchtiges RAM

Flash Speicher

Flash-Speicher

- **Spezielle EEPROM-Variante**
- Dünneres Tunneloxid
- Löschen/Beschreiben der Zellen erfordert geringere Spannungen/Ströme

$\Rightarrow$ **Nur ganze Blöcke von Zellen löschbar/beschreibbar**

- Arbeitet ähnlich wie RAM-Baustein, aber **nicht-flüchtig**
- CF, MMC, SD, SSD, USB-Memory-Stick, ...

SR-Latch (Set/Reset)

Unterschiedliche Bezeichnungen

- Auch RS-Latch oder RS-Flipflop bezeichnet

⇒ Vorsicht: Flipflop eigentlich taktflankengesteuert

- Stellt die einfachste Art eines Flipflops dar
- **Pegel- bzw. zustandsgesteuert, und nicht taktgesteuert** (da kein Taktsignal vorhanden)

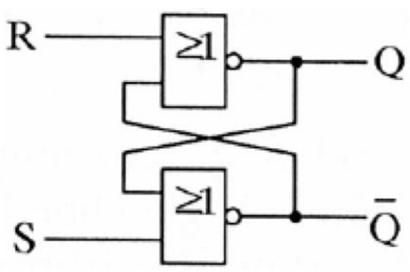
⇒ Taktsteuerung mittels Zusatzschaltungen möglich

Aufbau

- Üblicherweise aus 2 NOR Gattern aufgebaut
- Kann auch aus zwei NAND Gattern und (üblicherweise) negierten Eingängen $\overline{S}$, $\overline{R}$ aufgebaut werden!

Grundschaltung (kein Strom)

- $R = 0, S = 0$
- Ausgang (Q) und negierter Ausgang ($\overline{Q}$)



SR-Latch (**S**et / **R**eset) aus NOR Gattern

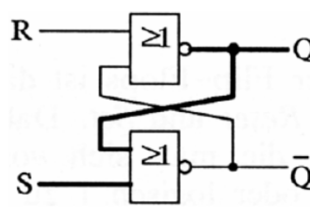
"Set" – $R = 0, S = 1$

- Setze S auf 1

⇒ Dicke Linien: Logisch 1

- Ausgänge ändern sich

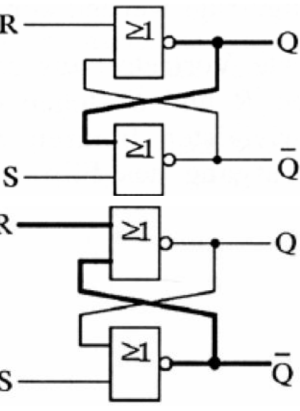
⇒ Q auf 1 gesetzt



$R = 0, S = 0$

- Liegt nachher an beiden Eingängen 0 an, bleibt der Zustand unverändert

⇒ Stabiler Zustand



"Reset" – $R = 1, S = 0$

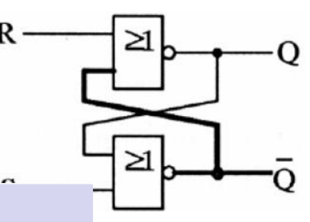
- Setze R auf 1
- Ausgänge ändern sich

⇒ Q auf 0 gesetzt

$R = 0, S = 0$

- Liegt nachher an beiden Eingängen 0 an, bleibt der Zustand unverändert

⇒ Stabiler Zustand



Wahrheitstabelle

S	R	Q	$\overline{Q}$	Bemerkung
0	0	Q	$\overline{Q}$	Speichern des Zustandes
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	potentielle race condition wenn R=S=0 folgt [a]

a-> Dieser Eingangszustand wird in der Literatur oft als instabil bezeichnet (was eigentlich nicht stimmt), bzw. als "verbotener Zustand", da er einen logischen Widerspruch ergibt (Q = Q)

Getaktetes SR-Latch (taktzustand)

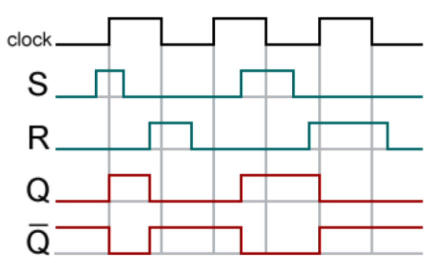
Getaktetes (“clocked”) SR-Latch

Getaktetes SR-Latch

- Latch kann Zustand nur wechseln wenn Takt/Clock-Eingang aktiv (logisch 1)
- Kann mehrmals während aktiver Taktphase den Zustand wechseln!

$\Rightarrow$ **Latch ist taktzustands-(taktpegel-)gesteuert**

- Zustand $R = S = 1$: weiterhin potentielle race condition



Bei Taktzustand gesteuert kann sich der Latch öfters während einem takt (1) ändern es gilt immer der letzte befehl welcher dann in der 0 Phase gespeichert wird

Bei input 0 von taktung ist zustand stabil

D-Latch

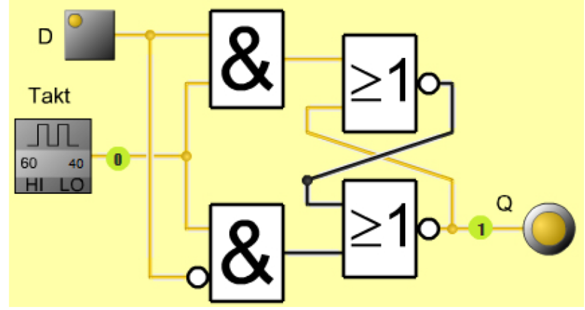
Data(Delay)-Latch

- R-Eingang wird durch das invertierte S-Signal definiert

$\Rightarrow$ $R = S = 1$ nicht mehr möglich

- Der Wert des Daten-Eingangs wird an den Datenausgang durchgeschleift so lange **Takt 1** ist. (**Taktzustand!**)
- Geht **Takt auf 0**, wird letzter Wert des Eingangs "eingesperrt" (*latch*)

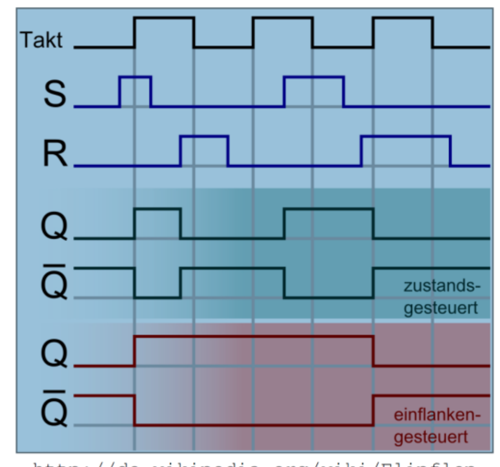
Version von SR- Latch um Race Kondition zu verhindern



Wenn ur auf enable(1) wird der befehl von d weitergeben → muss immer befehl sein

D-FlipFlop // Taktzustand vs Taktflanke

Taktzustands- vs. taktflankengesteuert



http://de.wikipedia.org/wiki/Flipflop

Taktflanken gesteuert → Zustand kann sich nur an der flanke ändern (genau dann wenn 0-1 springt) Und bleibt dann zwischen den flanken stabil → Flipflop

D-FlipFlop

Date(Delay)-FlipFlop

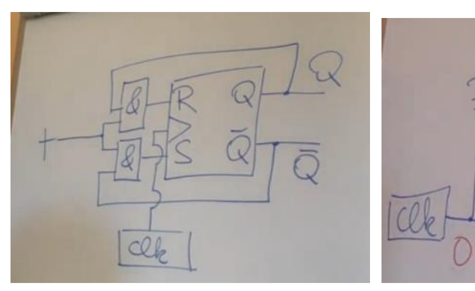
- Hat wie das D-Latch einen Daten-Eingang und einen Clock/Takt-Eingang
- Übernimmt den Eingangswert NUR zu dem Zeitpunkt, zu dem Takteingang von Low auf High wechselt

⇒ **FlipFlop ist taktflankengesteuert**

- Wird über Kaskade von zwei D-Latches realisiert

Notation

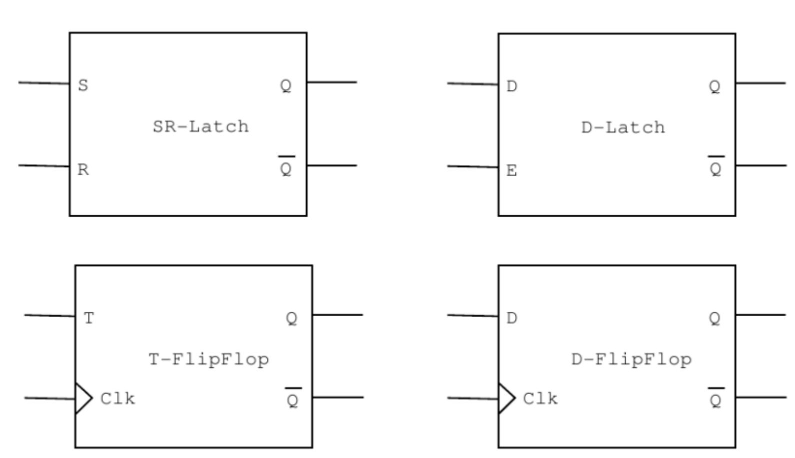
- In der (deutschen) Literatur wird das D-Latch manchmal als **taktzustandsgesteuertes** D-Flipflop bezeichnet



D-Flipflop

T-Flipflop

Speichersymbole



Latch- Taktzustand
FlipFlop- Taktflanken gesteuert

Random Access Memory

Schreib-/Lese-Speicher

- Für Register, Akkumulatoren, Caches, Hauptspeicher, ...
- **Random Access:** Es kann in beliebiger Reihenfolge zugegriffen werden, in theoretisch gleicher Zeit (vgl. Bandlaufwerk oder Disk)
- **Sind aus einzelnen Speicherzellen aufgebaut, die gruppiert werden**
- Verschiedene Arten der Organisation möglich

Arten von flüchtigen RAMs

- SRAM: Statisches RAM
- DRAM: Dynamisches RAM

SRAM

Ähnlich wie flip flop

Statisches RAM

- **Schnelle Lesezugriffe und Umschaltzeiten**
- **Kein Refresh nötig, dennoch flüchtig**
- **Teurer, größer als DRAM**
- Für Register, Akkumulatoren und Caches verwendet
- SRAM-Zelle besteht meist aus 6 Transistoren
- Funktionsprinzip wie **taktgesteuerte D-FlipFlops**

simpel

widerstand

5V

R

R

Q

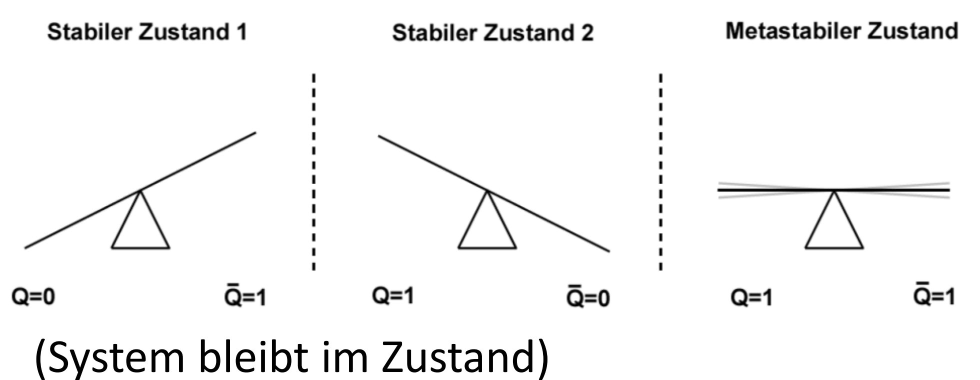
$\bar{Q}$

Bistabiler Schaltkreis - Bistabile Kippstufe

- **Zwei mögliche stabile Zustände**
- In stabilen Zustand gebracht, verbleibt Schaltkreis darin

⇒ **Schaltkreis hat „Gedächtnis"**

- Realisierung: z.B. mit zwei Inverter oder zwei NANDs/NORs, deren Ausgänge an den Input des jeweils anderen rückgekoppelt ist



(System bleibt im Zustand)

Kanal grundsätzlich da aber leitet gering (NMOS alle

NMOS->

Bitleitung BL

$+U_B$

Bitleitung $\overline{BL}$

Tr5

Tr6

Tr3

Q

$\bar{Q}$

Tr4

Tor transistor regelt zugang

Tr1

Tr2

GND

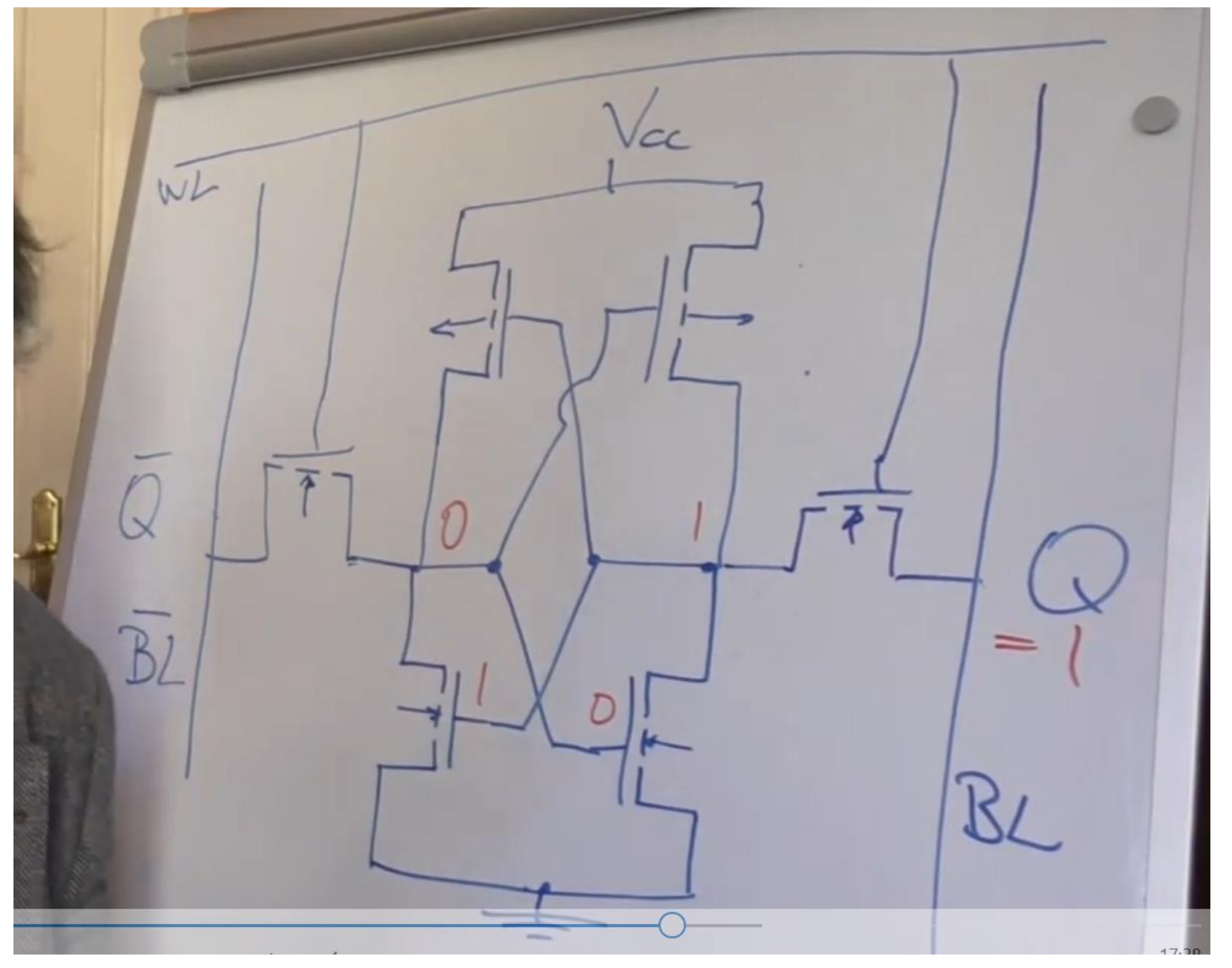
Wortleitung WL

Lesezugriff über
Wortleitung = 1
Bitleitung = Wert

Schreiben über
Wortleitung = 1
Bitleitung = Input Wert

Abbildung 4.13: Das Flipflop aus zwei NMOS-Transistoren ist der Kern der SRAM-Speicherzelle. Die Wortleitung schaltet über zwei weitere Transistoren die Ein-/Ausgänge auf die Bitleitungen BL und $\overline{BL}$ durch. Über diese erfolgt das Schreiben und Lesen von Daten.

SRAM PMOS



Lesezugriff über
Wortleitung = 1
Bitleitung = Wert

DRAM

DRAM-Zelle

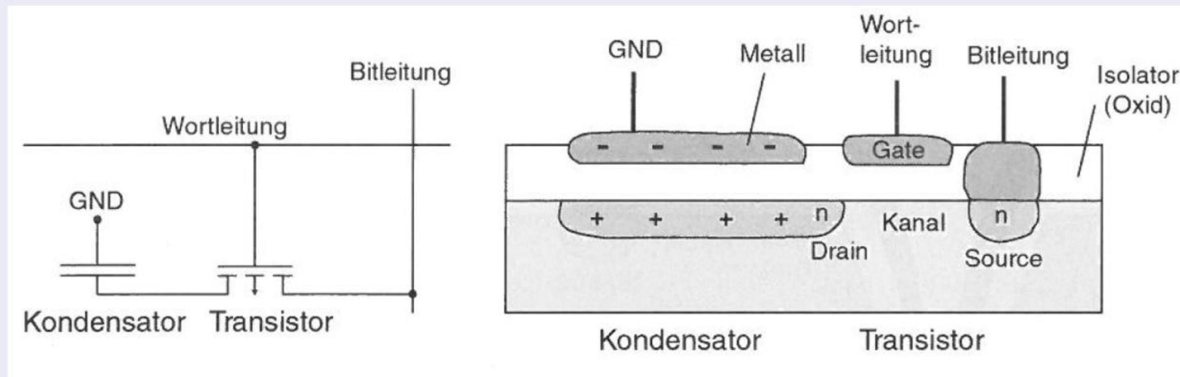
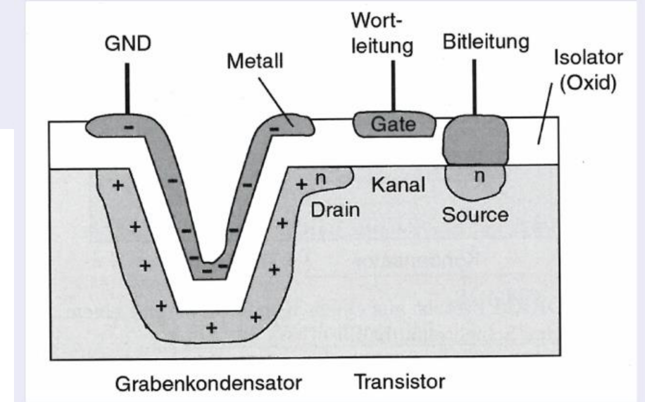


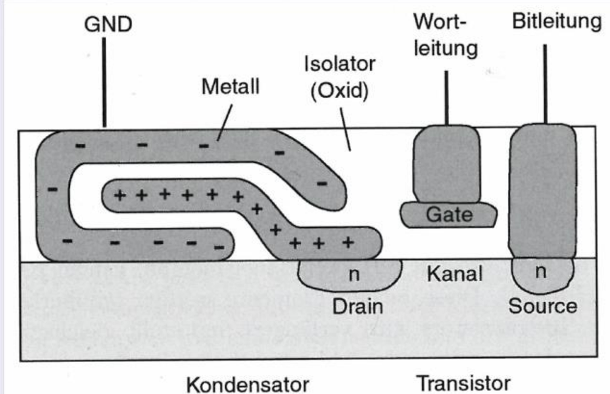
Abbildung 4.14: Die Zelle eines DRAM besteht aus einem Kondensator und einem Tortransistor. Links Schaltbild, rechts Schichtenaufbau.

Dynamic RAM

- Kleiner und billiger als SRAM ⇒ Hauptspeicher
- DRAM-Zelle besteht aus Kondensator und Transistor
- Kondensator = Ladungsspeicher = 0/1
- (Tor-)Transistor zum Ansteuern
- **Regelmäßiger Refresh nötig, sonst Datenverlust**
- Typische Refreshzeiten: alle 32ms oder 64ms (d.h. hält nur für Sekundenbruchteile!)
- In Array angeordnet - immer ganze Zeilen aktiviert!

Graben- vs Stapelkondensator





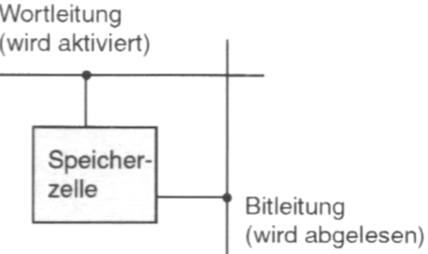
Speicheraufbau

Aufbau von Speicherbausteinen

Aufbau

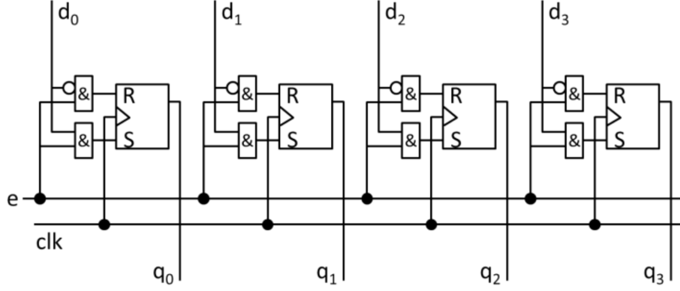
- **Gitterartiger Aufbau** (Matrix)
- **Waagrechte Wortleitungen**: Aktivieren alle Speicherzellen einer Zeile (→ Gates)
- **Senkrechte Bitleitungen**: Zum Lesen/Schreiben einer Speicherzelle (→ Drains)
- An Kreuzungspunkten sitzen Speicherzellen

⇒ **1 Bit pro Speicherzelle**



Reihe von taktflankengesteuerte R/S-FlipFlops

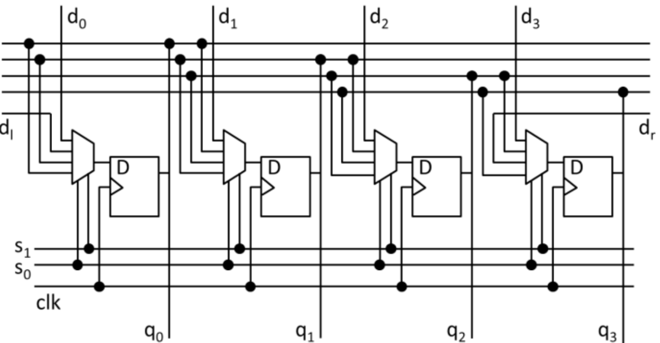
- Enabler = 1: Bits an Eingängen $d_0 \ldots d_3$ werden eingespeichert
- Enabler = 0: Registerinhalt kann an Ausgängen $q_0 \ldots q_3$ abgelesen werden



Universalregister

Alle Funktionalitäten auf einmal

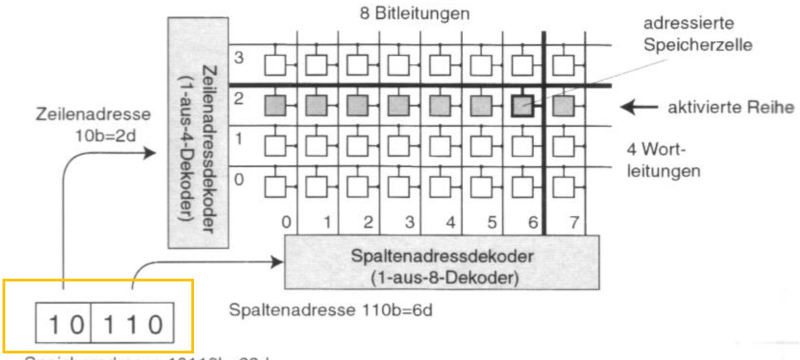
- 4:1-Multiplexer mit 2 Steuerleitungen: laden, speichern, schieben nach links/rechts



Adressierung

Aufbau von Speicherbausteinen

Beispiel
- 32-Bit-Speicherbaustein (5 Adress-Bits)
- Speicherzelle mit Adresse 22 wird selektiert



Ablauf
- Speicheradresse wird geteilt (Zeilen-/Spalten-Adresse)
- Erster Teil geht an Zeilenadressdekoder (Wortleitung)

⇒ Alle Speicherzeilen dieser Wortleitung werden aktiviert
- Spaltenadressdekoder wählt **eine** Bitleitung aus

⇒ Nur Signal dieser Bitleitung wird gelesen/geschrieben

Lese vs. Schreibzugriff
- Mittels zusätzlichem Signal “READ/WRITE” wird gesteuert, ob gelesen oder geschrieben werden soll

⇒ Typische Bezeichnung: $R/\overline{W}$; HIGH:lesen, LOW:schreiben

Organisation
- Bezeichnet Anzahl der Speicheradressen die unter jeder Speicheradresse gespeichert wird (*auch Speichertiefe genannt*)

Beispiel: 4kx1
- Baustein mit 4096 Speicherzellen mit je 1 Bit
- 12 Adressleitungen (2^{12}), eine Datenleitung, Gesamtkapazität 4KiBit

8kx4
- Zellenmatrix von 8192 Speicherzellen vierfach vorhanden
- 13 Adressleitungen (2^{13}), 4 Datenleitungen, Gesamtkapazität 32KiBit (2^{13} * 4 = 2^{15})

Speicher von Mikroprozessorsystemen
- Meist Byte- oder Wort-strukturiert ⇒ Kein einzelnes Bit ansprechbar, sondern immer nur Gruppen (8, 16, 32 Bit)
- Um diese Strukturierung zu erreichen, müssen je nach Organisation mehrere Speicherbausteine parallel geschaltet werden

⇒ z.B. für Byte-Struktur: ein Nx8 Speicherbaustein, oder zwei Nx4 Speicherbausteine in parallel

Schreib-Zugriff

Beispiel: Einzelner Schreibzyklus
- CS - Eingang zur Bausteinaktivierung (Chip Select)

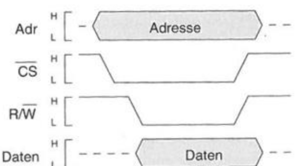


Abbildung 4.4: Typischer Ablauf eines Schreibzyklus. Die Daten werden dem Speicherbaustein schon früh im Zyklus zur Verfügung gestellt. Da es viele Adress- und Datenleitungen gibt, ist bei diesen immer durch eine Aufspreizung angedeutet, wann gültige Werte anliegen. $\overline{CS}$ = Chip Select, $R/\overline{W}$ = Read/Write.

K. Wüst, Mikroprozessortechnik, 4. Auflage

Schreib-Zugriff

Beispiel: Zwei aufeinanderfolgende Lesesyklen
- Zugriffszeit: Zeitdifferenz zwischen Beginn des ersten Lesezyklus und der Bereitstellung der Daten
- Zykluszeit: Zeitdifferenz zwischen zwei Lesezugriffen

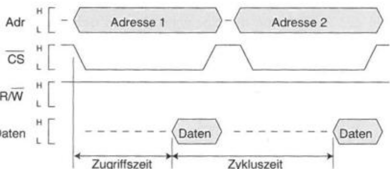


Abbildung 4.5: Typischer Ablauf zweier aufeinanderfolgender Lesezyklen. Der Speicher stellt die Daten gegen Ende der Zyklen zur Verfügung. Die Zykluszeit ist hier deutlich größer als die Zugriffszeit.

REP-BSP1

1. **Speicher.**

   Wählen Sie für jede der folgenden Aufgaben (a)-(d)einen passenden Speichertyp aus und **begründen** Sie Ihre Wahl!

   (a) Speicherung des Programms einer digitalen Armbanduhr.

   (b) Speicherung der Firmware auf einem Mainboard.

   (c) Datenspeicherung (für Bilder, Musik etc.) in einem Smartphone

   (d) Arbeitsspeicher eines Webservers.

   (e) Wir betrachten einen 32 Bit Baustein mit 5 Adress-Bits: Welche Wort- bzw. Bitleitung (i.e., Zeilen- bzw. Spaltenadresse) wird durch die Speicheradresse 11|100 angesprochen? Erklären Sie auch den Ablauf dieses Speicherzugriffs.

   (f) Wir betrachten einen größeren Baustein: Wie viele unterschiedliche Adressen können Sie mit einer Speicheradresse ansteuern, welche die ersten 4 Bit für Zeilenadressen, und die restlichen 3 Bit für Spaltenadressen reserviert hat, und warum?

a) eine PROM das man einmal mit der Software beschreibt. Billiger als MROM und kleinere Stückzahl möglich. Aber Speicher in Uhr muss nicht wieder beschreibbar sein => daher PROM

b) EEPROM muss evtl für Firmware Updates geändert werden aber nicht oft daher EEPROM alternativ auch Flash Speicher jedoch ist der (zu) leicht veränderbar (evtl. versehentlich)

c) Flash Speicher => schnell und nicht flüchtig und kann oft gelöscht und beschrieben werden.

d) DRAM => schnell und gut für Arbeitsspeicher oder dieser nur kurz Daten zwischenspeichern muss und ist billiger als SRAM

e) Es wird die 4. Zeile (Wortleitung) ausgewählt und in dieser Zeile die 4. Zahl (4. Bitleitung)

Durch Decoder wird 11 für Zeile und 100 in die zugehörige Spalte umgewandelt und Signal fließt über beide Kanäle (Wortleitung) und Bitline in jener eine Speicherzelle welche ausgelesen wird (1 Bit) -> output des Speichers ist dann dieses Bit

dieser Speicher wird gelesen

3
2
1
0

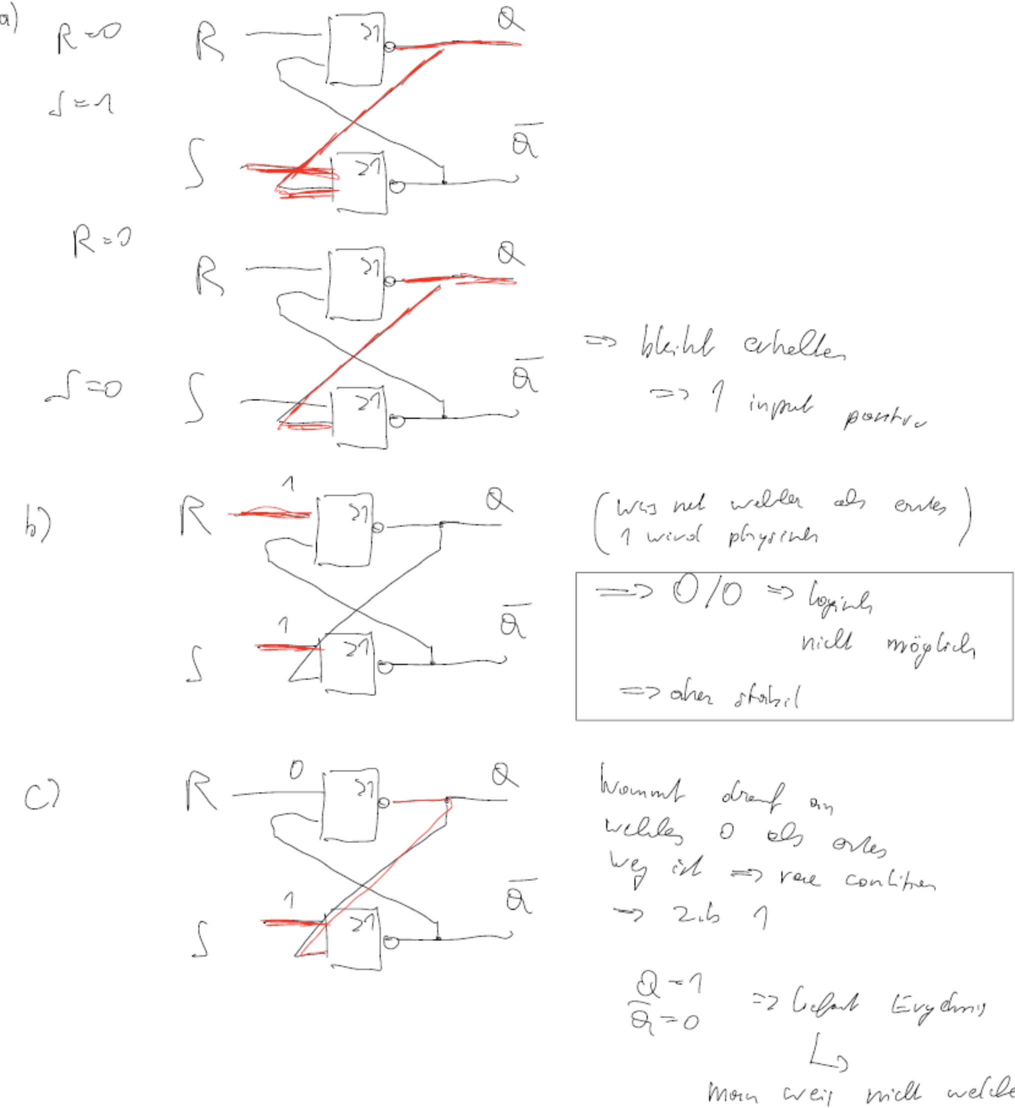
Decoder

0 1 2 3 4 5 6 7

11/100 - Decoder

f) *Man kann 2^4 * 2^3 adressen speichern → 128*

REP-BSP 2



2. **Analyse eines (nicht-taktgesteuerten) SR-Latches.**

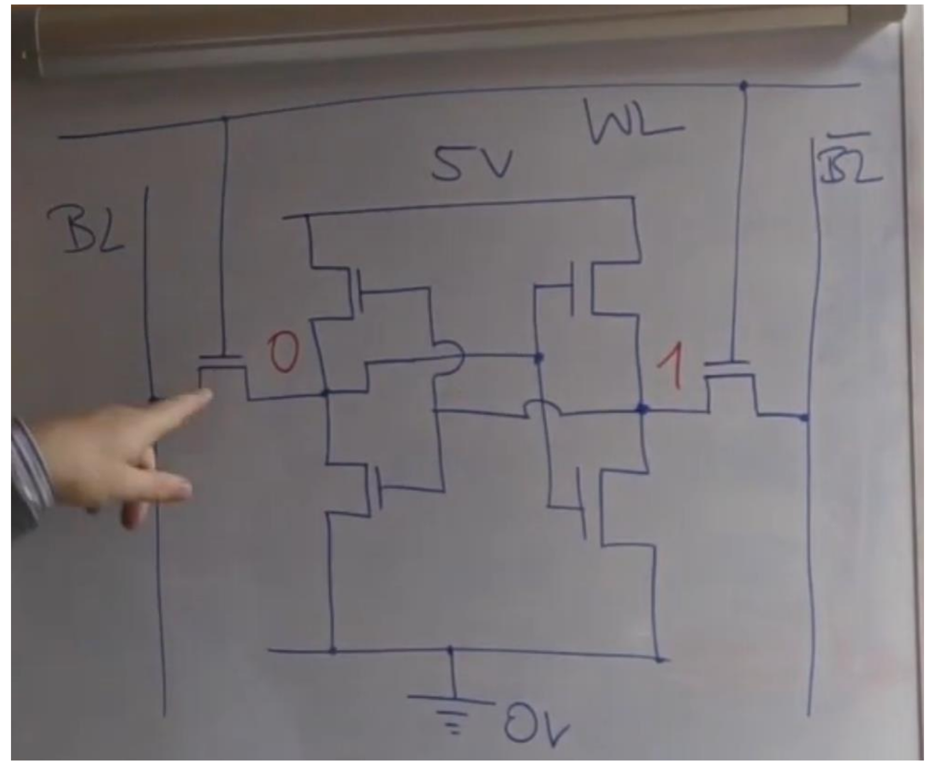
(a) Erklären Sie **warum** der Zustand eines SR-Latch aus NOR Gattern beim Wechsel von $(R = 0, S = 1)$ auf $(R = 0, S = 0)$ erhalten bleibt?

(b) Zeichnen Sie den Zustand (dicke Linie = logisch 1) für die Eingangsbeschaltung $(R = S = 1)$. Ist der Zustand stabil? (*Anmerkung: in der Literatur wird diese Eingangsbeschaltung oft als verboten oder instabil bezeichnet*)

(c) Was passiert, wenn nun direkt nach $(R = S = 1)$ die Eingangsbeschaltung $(R = S = 0)$ folgt? (*Anmerkung: Gehen Sie davon aus, dass S und R* ***nicht genau gleichzeitig*** *in den Zustand 0 wechseln, sondern es kurze Verzögerungen gibt.*)

(wenn wirklich genau gleichzeitig)

REP-BSP3

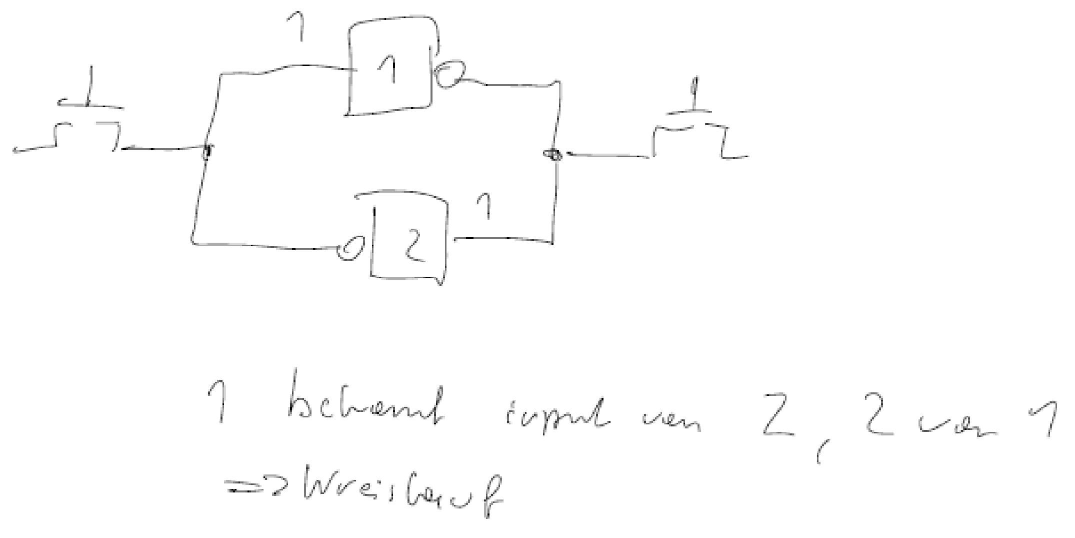


3. **Betrachten Sie eine in CMOS-Technologie realisierte SRAM-Zelle (Folie 35).**

   (a) Erläutern Sie, inwiefern sich diese SRAM-Zelle als gegenseitige Verschaltung zweier CMOS-Inverter charakterisieren lässt.

   (b) Welche Schritte sind für ein Auslesen des Speicherinhalts vorzunehmen?

   (c) Nehmen Sie an, die SRAM-Zelle speichert aktuell eine "0", Sie möchten aber eine "1" hineinschreiben. Wie gehen Sie vor?



b) transistor von Wortleitung öffnen => Bitleitung lesen (Q und $\overline{Q}$)

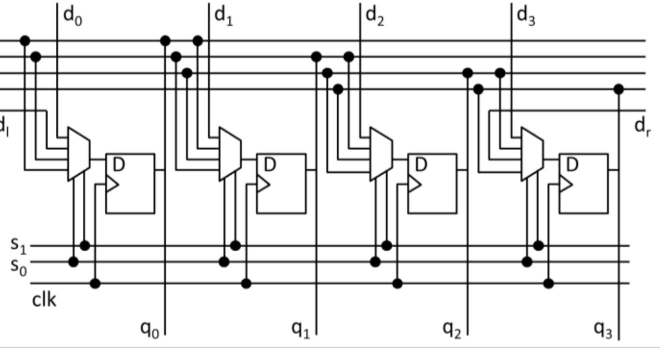
c) Wortleitung => offen => Bitleitung 1 übertragen

REP-BSP4

4. **Betrachten Sie das 4-Bit-Universalregister auf Folie 43.** Nehmen Sie an, die Eingänge sind konstant wie folgt belegt: $d_0 = d_1 = d_3 = d_r = 1, d_2 = d_l = 0$. Die beiden Steuereingänge sind jeweils zu Taktbeginn (d.h. zum Zeitpunkt der Taktflanke) wie folgt belegt:

0 0 laden 0 1 links 1 0 rechts 1 1 speichern

   (a) Takt 1: s0=s1=0
   (b) Takt 2: s0=s1=1
   (c) Takt 3: s0=0, s1=1
   (d) Takt 4: s0=0, s1=1
   (e) Takt 5: s0=s1=1
   (f) Takt 6: s0=1, s1=0
   (g) Takt 7: s0=1, s1=0
   (h) Takt 8: s0=s1=0



Frage: Wie lautet am Ende jedes Taktes jeweils die Belegung von $q_0 \ldots q_3$?

a) $q_0 = 1$ $q_1 = 1$ $q_2 = 0$ $q_3 = 1$ / read

b) $q_0 = 1$ $q_1 = 1$ $q_2 = 0$ $q_3 = 1$ / save / hold

c) $q_0 = 1$ $q_1 = 0$ $q_2 = 1$ $q_3 = 1$ left

d) $q_0 = 0$ $q_1 = 1$ $q_2 = 1$ $q_3 = 1$ left

e) $q_0 = 0$ $q_1 = 1$ $q_2 = 1$ $q_3 = 1$ / save / hold

f) $q_0 = 0$ $q_1 = 0$ $q_2 = 1$ $q_3 = 1$ / right

g) $q_0 =$ [illegible] $q_1 = 0$ $q_2 = 0$ $q_3 = 1$ / right

h) $q_0 = 1$ $q_1 = 1$ $q_2 = 0$ $q_3 = 1$ / read

Allgemein

Anwendung

Höhere Programmiersprache

Assembler

Betriebssystem

Maschinensprache

Mikroprogramm

Physikalischer Rechner

Übersetzer

Interpreter

Assembler-Ebene:

- Stellt symbolische Notation (Mnemonics) der auf Maschinensprache- und Betriebssystem-Ebene vorhandenen Befehle zur Verfügung
- Pseudoinstruktionen, die sich einfach durch wenige Maschinenbefehle realisieren lassen

Höhere Programmiersprache-Ebene (Fortran, C, Java,...):

- Rechner ist eine Black Box, tatsächlicher technischer Ablauf bzw. Hardware-Struktur bleiben verborgen

Anwendungs-Ebene:

- Nur Anwendung sichtbar

Maschinensprache-Ebene:

- Unterste für Programmierer frei zugängliche Ebene
- *Software/Hardware-Schnittstelle* eines Prozessors
- Satz der verfügbaren Maschinenbefehle eines Prozessors bestimmt die "Architektur" des Rechners (CISC vs RISC)

Betriebssystem-Ebene:

- Verwaltung von Speicher
- Ansteuerung der Ein-/Ausgabe-Geräte
- Ausführung von Programmen
- Kommunikation zwischen Mensch und Computer

Hardware-Ebene:

- Physikalische, elektrotechnische Ebene
- Wie ist Prozessor (i.e., Steuer- und Recheneinheiten) hardwaremäßig aufgebaut (Transistoren, Gatter, etc.)?

Mikroprogramm-Ebene:

- Wie werden einzelne Bausteine der Steuer- und Recheneinheit von Maschinenbefehlen angesteuert?
- Fetch–Decode–Execute
- Nur bei CISC-Architektur, bei RISC-Architektur kann diese Ebene entfallen

Compiler vs. Interpreter

Programmausführung:

Jede Ebene muss auf Hardware-Ebene abgebildet werden

Compiler

L_i-**Compiler**: Ein Maschinenprogramm, das ein Programm P_i der Sprache L_i in ein Programm P_j der Sprache L_j transformiert (**übersetzt**), das äquivalent zu P_i ist, d.h. das gleiche Ein-/Ausgabeverhalten wie P_i hat.

Interpreter

L_i-**Interpreter**: Ein Maschinenprogramm, das ein Programm P_i der Sprache L_i **Anweisung für Anweisung ausführt.**
Kein zu P_i äquivalentes Maschinenprogramm wird erzeugt.

Compiler vs. Interpreter

Anschaulich

Wir wollen ein Rezept nachkochen, das auf französisch verfasst ist, wir verstehen diese Sprache allerdings nicht.

Variante 1

Mit Rezept und Wörterbuch in den Supermarkt, jede Zutat einzeln nachschauen. Zuhause jeden Arbeitsschritt einzeln nachschauen. $\Rightarrow$ Interpretieren

Variante 2

Das Rezept vorab von jemandem, der der Sprache mächtig ist, in eine Sprache, der wir mächtig sind, übersetzen lassen, sodann die Einkaufsliste und Befehlsliste abarbeiten. $\Rightarrow$ Übersetzen (Kompilieren)

Maschienencode

Maschinenbefehlssatz

- Compiler übersetzt Programm von Hochsprache in Maschinenbefehle
- **Maschinenbefehlssatz:** Direktes Abbild aller vom Prozessor durchführbaren Operationen → Menge der Maschinenbefehle
- Transportbefehle: Speicher-Speicher, Register-Speicher, PUSH/POP, ...
- Arithmetische Befehle: Addition, Inkrement, Dekrement, ...
- Bitweise logische Befehle: AND, OR, XOR, NOT, ...
- Schiebe- und Rotierbefehle
- Sprungbefehle ...

- **Programm in Maschinenbefehlsdarstellung**
- **Sequenz von OpCodes und Operanden**
- Schwer merkbare Bitmuster
- Platzsparender: Hex-Darstellung
- Programme schwer zu lesen, unflexibel, schwer veränderlich; keine Kommentare möglich
- **Assemblersprache:** Jeder Maschinenbefehl durch leicht merkbare Abkürzung dargestellt, z.B. ADD, SHL, MOV, DEC, ...

Maschinencode / Maschinenprogramm

- **Mikroprozessor kann exakt bestimmte Menge von Aktionen ausführen**
- **Aktion = Maschinenbefehl**
- Bsp: Schreiben in Hauptspeicher, Bitmuster invertieren, ...
- Maschinenbefehl im ausführbaren Code **durch Bitmuster dargestellt**: Operationscode, **Opcode**
- Meist zusätzliche **Operanden** notwendig, z.B. Adresse
- **Maschinencode** = Sequenz von zusammenhängenden Maschinenbefehlen, die ablaufendes Programm darstellen
- **Programm = Abfolge von Instruktionen**

Ein Maschinenprogramm kann nicht mehr übersetzt werden

- Befehl für Befehl des Maschinenprogramms muss durch ein Mikroprogramm (Ebene 2) bzw. die Hardware (Ebene 1) abgearbeitet werden, d.h.
- aus dem Hauptspeicher in den Prozessor **geholt**,
- **dekodiert**, und letztendlich
- **ausgeführt** werden,

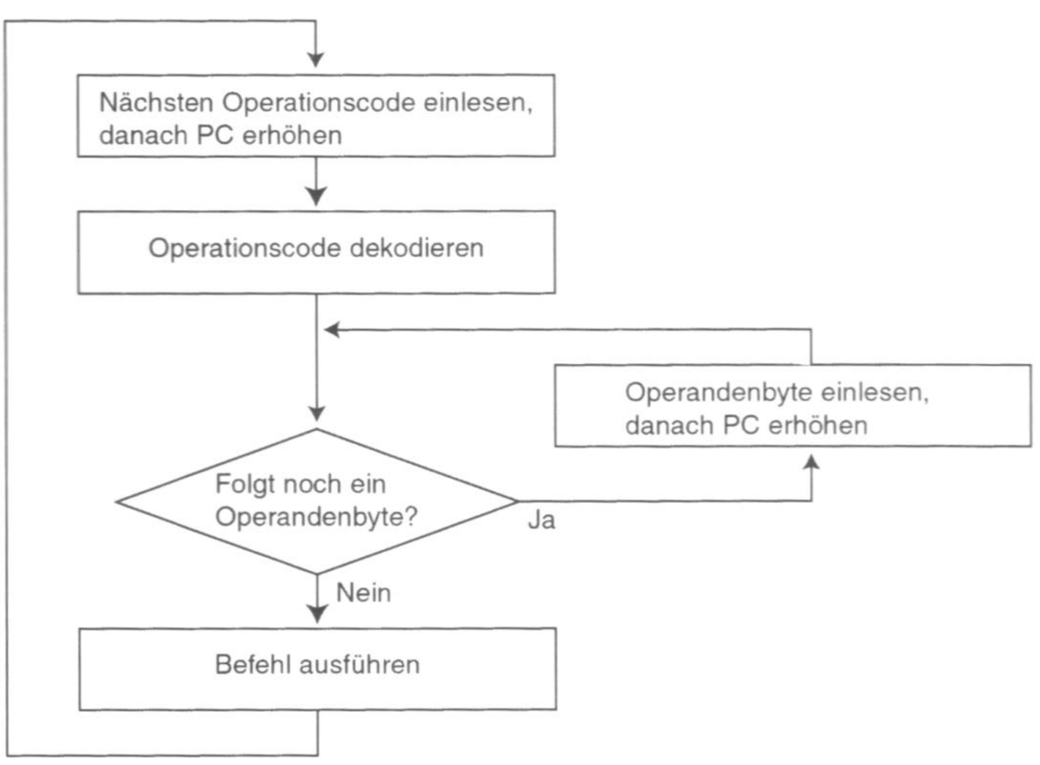
... bevor der nächste Maschinenbefehl durch den Prozessor betrachtet und verarbeitet werden kann

OpCode1	*Operand*	OpCode2	OpCode3	*Operand*	*Operand*	OpCode4	*Operand*
Maschinenbefehl 1		Maschinen-befehl 2	Maschinenbefehl 3			Maschinenbefehl 4	

Fetch-Decode-Execute

Programmausführung

- **Fetch**: Nächster auszuführender Opcode wird aus Programmspeicher gelesen
  - Program (auch Instruction) Counter (PC): Spezialregister, das die Adresse des nächsten Befehls enthält
  - PC wird inkrementiert, zeigt damit auf nächsten Befehl
- **Decode**: OpCode wird bitweise mit bekannten Mustern (**Befehlssatz**) verglichen, um Bedeutung herauszufinden
  - Falls OpCode mit Operand, wird PC inkrementiert, um Operanden auf nachfolgendem Speicherplatz zu lesen (bis alle Operanden gelesen)
- **Execute**: OpCode wird ausgeführt



REP-BSP1

1. **Abstraktionsebenen/Prozessorarchitektur**

   (a) Nennen Sie drei Beispiele für Programmiersprachen, die kompiliert werden, und drei, die interpretiert werden?
   Was sind jeweils die Vor-/Nachteile?
   Wo ist die Sprache Java einzuordnen?

   (b) Kann ein OpCode ohne Operand existieren bzw. umgekehrt? Erklären Sie Ihre Antwort!

   (c) Erklären Sie den Fetch – Decode – Execute Zyklus. Was passiert in den einzelnen Schritten? Gehen Sie auch auf Operanden und Befehlszähler (*program counter*) ein.

a) kompiliert => optimiert
C, C++, Fortran
interpretierer
Javascript, Phyton, basic
VM -> Java -> Mitte -> Laufzeit kompilierung

b) ja Opcode ohne Operand geht => z.b End (halt)
=> operand alleine geht nicht => was soll er tun?

c) Fetch = Opcode laden
Counter ++ (Adresse wo nächster Befehl)

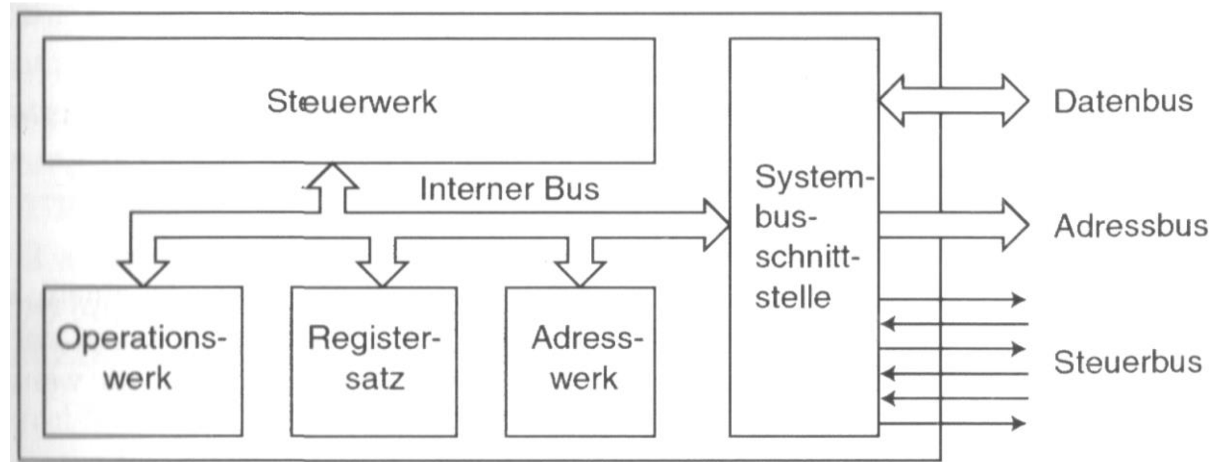
Decode = Opcode mit Befehl vergleichen => bedeutung
↳ evtl operand -> lesen Counter ++ ↻

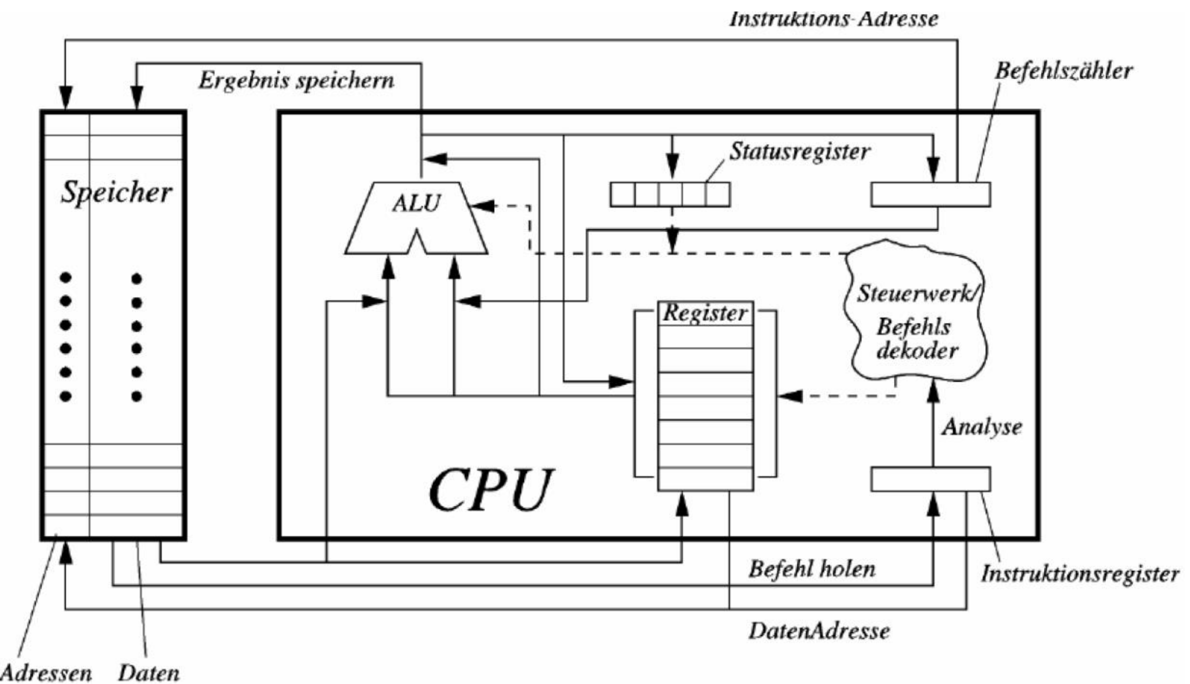
Execute = Ausführen Befehl

Prozessor

Interner Aufbau einer CPU

- **Registersatz**: Register, um Daten innerhalb des Prozessors speichern zu können
- **Steuerwerk**: Verantwortlich für Ablaufsteuerung
- **Operationswerk** (Rechenwerk): Eigentliche Datenverarbeitung
- **Adresswerk**: Um auf Daten und Code im Hauptspeicher zugreifen zu können
- **Systembus**-Schnittstelle: Datenverkehr mit Rest des Systems





Registersatz

- Speicherbereich in CPU
- Schnell
- Speichert Operanden für ALU und Resultate
- Speichert PC(Counter=
- Gibt Spezialregister z.b für Überlauf, Befehle, Befehlzähler…
- Oder Universal für verschiedene

Akkumulator = Datenregister für Operanden (für ALU)
PC = Programm Counter

Register

- Ein Speicherbereich innerhalb eines Prozessors
- Durch internen Datenbus meist direkt mit ALU verbunden
- Wesentlich schneller (und kleiner) als Hauptspeicher
- Gruppe von Flip-Flops mit gemeinsamer Steuerung
- 1 Flip-Flop kann 1 Bit speichern
- Register haben gewisse Breite, meist 8, 16, 32, 64... Bit
- **Registersatz**: Nach außen „sichtbare“ Register

Registertypen

- Datenregister (Akkumulator): speichern Operanden für die ALU und deren Resultate
- Adressregister: speichern Speicheradressen eines Operanden oder Befehls
- Universalregister: für verschiedene Inhalte verwendbar
- Spezialregister: für bestimmte Zwecke vorgesehen
  - Befehlszählregister: Speicheradresse des nächsten auszuführenden Befehls
  - Befehlsregister (Instruction register): (Zwischen)-Speicherung des aktuellen Befehls
  - Statusregister: z.B. Auftreten eines Überlaufs

Steuerwerk

- Decodiert den OpCode
- Koordination der anderen Elemente
- (Quasi der Chef der aufgaben verteilt)
- Sehr kompliziert muss z.b auf Daten warten oder warten bis ergebnise kommen → koordinieren

Steuerwerk – Control Unit (CU)

- Steuert den Ablauf der Befehlsverarbeitung
- **Decodierung** der OpCodes (Instruction Decoder)
- Ansteuerung und Koordination der anderen Elemente
- Opcode wird im Befehlsregister abgelegt und dekodiert
- Mittels Schaltwerk werden notwendige Signale für Ausführung erzeugt

Programmzähler zeigt auf Speicherplatz, an dem folgender OpCode liegt: `Kopiere Register 1 in Register 2`

- Programmzähler auf Adressbus legen
- Aktivierung ext. Steuerleitungen f. Lesezugriff im Speicher
- Einspeicherimpuls für Befehlsregister erzeugen, OpCode vom Datenbus entnehmen und im Befehlsregister einspeichern
- Dekodierung des Opcodes
- Register 1 auf Senden einstellen und auf internen Datenbus aufschalten
- Register 2 auf internen Datenbus aufschalten
- Einspeicherimpuls an Register 2 geben
- Programmzähler inkrementieren

Zu beachten

- Wartezeiten + Buszugriffe in Ausführungsphase
- Komplizierte Sequenzen von Signale zu erzeugen!
- Exakter zeitlicher Ablauf $\Rightarrow$ Synchronisierung!
- Statusflags von Prozessor zu berücksichtigen
- Externe Signale, z.B. Unterbrechungen (Interrupts)
- ...

Operationswerk

- Führt vom Steuerwerk verlangte Operationen aus
- z.B plus, minus, invertiert
- Ergebnis → Universalregister
- Statusregister. Z.b Übertrag, zero, oferlow, parity

Operations-/Rechenwerk, Arithmetic & Logic Unit (ALU)

- Führt die vom Steuerwerk verlangten **logischen und arithmetischen** Operationen aus
- Wird vom Steuerwerk nach Dekodierung einer entsprechenden Instruktion angesprochen
- Verfügbare Operationen unterschiedlich, z.B.:
  - $a \wedge b$
  - $\neg b$
  - $a + b$
  - $a + 1$
  - $a - b$
  - bit shift left/right
  - ...

Operations-/Rechenwerk, Arithmetic & Logic Unit (ALU)

- Über Steuereingänge auf bestimmte Anzahl von arithmetischen/logischen Operationen einstellbar
- Hängt vom Befehl ab, wird vom Steuerwerk nach dem OpCode-Decode gemacht
- Schaltwerk ohne eigene Speicherzellen: Operandenregister (Hilfsregister) auf Dateneingänge für Zeit der Berechnung aufgeschaltet
- Ergebnis wird über Datenbus z.B. in Universalregister geschrieben
- Kann aber auch erneut in Operandenregister geladen werden ⇒ komplexe Operationen in mehreren Schritten ausführen

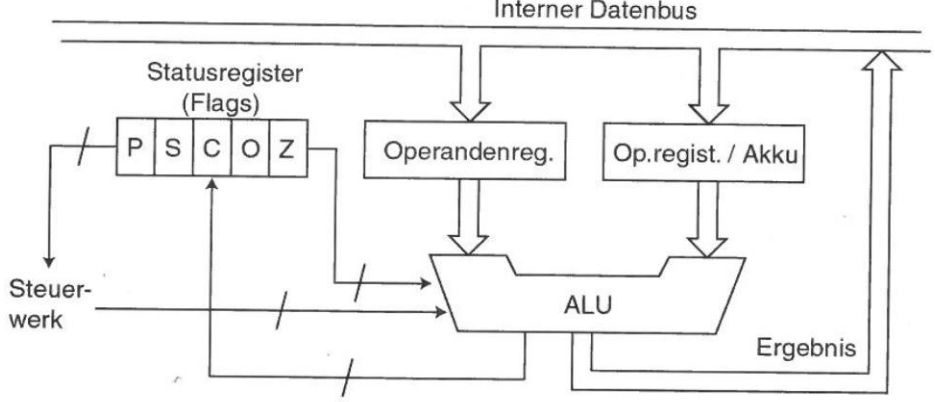


Abbildung 7.8: Zentraler Teil des Operationswerkes.

ALU selbst hat keinen Speicher operandenregister ist speicher
Resultat auf Bus → Register / oder zurück zu sich selber

Adresswerk

- Berechnet Adresse (mit eigenem ALU dafür)
- Viele Adressierung Arten

Aufgaben

- Berechnet nach den Vorschriften des Steuerwerks die Adresse eines Befehls oder eines Operanden

Adressierungsarten

- Unmittelbare Adressierung: Operand als Konstante im Befehl enthalten (folgt unmittelbar auf den OpCode) Erkennt Operanden
- Registeradressierung: Register **direkt** angesprochen Operant in Register in CPU

Adressierungsarten

- **Direkte** Adressierung: Speicheradresse beim Kompilieren bekannt und als Operand im Maschinencode
  Operand im Hauptspeicher → vorher bekannt wird im code mitgesendet
- Z.B. Sortieralgorithmen: Adressen müssen zur Laufzeit festlegbar sein
- Register-**indirekte** Addressierung: **Inhalt eines Registers** als Adresse interpretiert Adresse wo der Operand ist
- **Basisregister**: volle Adressbusbreite, für Anfangsadresse
- **Displacement** (aus Maschinencode) kann addiert werden
- Adresswerk hat eigenen Adressrechner

Für Vetkoren z.b → Daten durchforsten

Adressierungsarten

- **Indexregister**: Kann z.B. Autoinkrement, Autodekrement
- Für **indizierte Adressierung** ideal, um zusammenhängende Datenblöcke sequenziell zu adressieren Adresse im Speicher ist pointer
- **Speicherindirekte Adressierung:** Register verweist auf Speicherplatz, dieser wird gelesen und als Adresse verwendet
- Bis zu zwei Displacements möglich

Stackpointer Last in first out

- **Spezialregister, SP, Stapelzeiger**
- Verwendung eines Stacks in Hardware unterstützt
- Speicherstruktur mit **Last-In–First-Out** Prinzip (LIFO)
- Anschaulich: ein Stapel Teller oder Bücher, ...
- Temporäre Daten, z.B. Programmverlauf
- Wächst meist zu kleineren Speicheradressen hin
- Neues Wort im Stack ablegen: Stackpointer dekrementieren **(PUSH)**
- Wort von Stack holen: Stackpointer inkrementieren **(POP)**

Systembus

Systembus = schnittstelle interner Bus und externen bus

BSP-REP2

2. Prozessor

(a) Erklären Sie die Aufgaben von Adressbus, Kontrollbus und Datenbus. In welche Richtung kann jeweils kommuniziert werden und warum?

(b) Erklären Sie den Unterschied zwischen direkter und indirekter Adressierung!

(c) Erläutern Sie mindestens fünf der Standardflags (Zero, Overflow, Parity, ...)

2)
a) Adressbus = kann nur nach außen kommunizieren, da von außen keine Adressen im Prozessor aufgerufen werden können (der Hauptspeicher ist ja außerhalb). Daher geht er nur in eine Richtung um die Kapazität zu erhöhen
Kontrollbus = geht nur von außen nach innnen da es von außen die CPU steuert -->wird auch Steuerleitungen geannt und sind besondere Leitungen für besondere Aufgaben z.b für Signale vom Betriebssysstem für den Prozessor oder zum Resseten oder auch für die Lese und Schreibsteuerung
Datenbus = sind bidirketional. Transferiert Daten in die CPU und auch Ergebnise aus der CPU raus ist die Hauptroute von und in den Prozessor auf welcher auch Befehle und Operandten transportiert werden

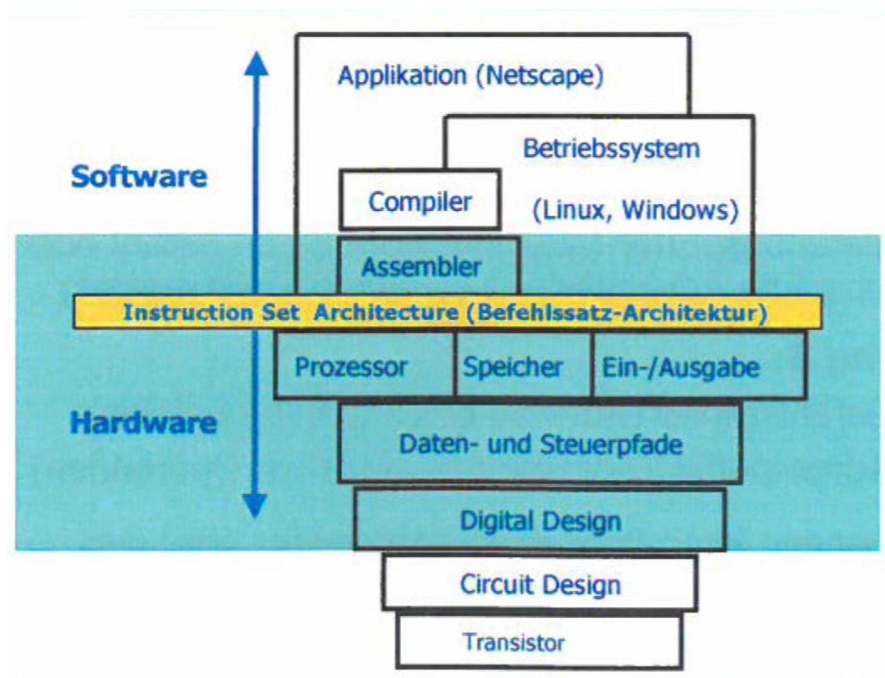
b) Bei der direkten Adressierung ist die Speicheradresse bekannt und ist direkt im maschienencode enthalten und ist somiet sehr schnell
bei der indirekten Adressierung dagegen muss der Inhalt eines Registers als Adresse interpretiert werden wo die Daten liegen daher muss man dann noch auf diesen zugreifen (und suchen) und kann dann erst auf die Daten zugreifen was noch mehr Zeit benötigt.

c)
Zero = beduetet das, dass Resultat einer operation null ist
carry flag = ermöglicht es additionen subtraktionen zu ermöglichen mit Überlauf
negative flag = bedeutet das die operation negativ ist
overflow flag = resultat einer berechnung ist zu groß um in den zugewiesenen Speicher zu passen
parity flag = zeigt ob Numer der bits in der letzten operation gerade oder ungerade sind
Interrupt flag = Zeigt ob unterbrechnungen aktiviert sind

ISA

ISA - Instruction Set Architecture / Befehlssatz

- **Gesamte nach außen hin sichtbare Architektur**
  1. Befehlssatz (s.o.)
  2. Registersatz (s.o.)
  3. Speichermodell (Größe des Adressraums, Breite der Busse, etc.)
- Muss zur Erstellung von Maschinenprogrammen für diesen Prozessor bekannt sein
- Für Assemblerprogrammierung, Aufbau von Compilern, ...
- **ISA = Schnittstelle zwischen Hard- und Software**



ComputerDesign

Computerdesign

- **Wir wollen einen neuen, *optimalen* Computer kaufen/bauen**
- **Oder wir wollen verschiedene existierende vergleichen**
- **Was aber ist eigentlich *optimal*?**
  - Welche Tasks?
  - Welche Programme?
  - Welcher Benutzer?
  - Schnelligkeit?
  - Zuverlässigkeit?
  - Physische Robustheit?
  - ...
- **Wir brauchen eine Metrik, um Vergleichbarkeit herzustellen**

Wichtige Kriterien

- **„Computer A ist schneller als Computer B“ muss quantifizierbar sein**
- **Durchsatz (Throughput, *X* [Aufträge/s]):** Mittlere Anzahl von erledigten Aufträgen pro Zeiteinheit. Je größer, desto besser.
- **Ausführungszeit (Execution Time, *ET*, [s]):** Zeit zwischen Beginn und Ende eines Auftrags. Je kleiner, desto besser.

Rechner *X* ist n-mal so schnell wie Rechner *Y*

$$\frac{ET_Y}{ET_X} = n$$

ET ist reziprok zu Performance

$$n = \frac{ET_Y}{ET_X} = \frac{Performance_X}{Performance_Y}$$

Beispiel

Durchsatz von X ist 1.3 mal so hoch wie von Y $\Rightarrow$ auf Rechner X können 1.3 mal so viele Tasks pro Zeiteinheit erledigt werden wie auf Rechner Y.

Designprinzipien

- „Make the common case fast“
- **Häufig auftretende Ereignisse schnell abarbeiten**
- Seltene Ereignisse nicht übermäßig beachten
- **Welche Gesamt-Verbesserung können wir erwarten?**

Execution Time

- **Verschiedene Definitionen möglich**
- **Response time** (wall clock time): gesamte Zeit vom Absenden der Anfrage bis zum Eintreffen der Antwort
- **CPU time**: nur Zeit, in der CPU aktiv ist (z.B. keine Wartezeiten)
- **User CPU time:** nur Zeit, die CPU für das Programm aufwendet (user mode)
- **System CPU time**: nur Zeit, die CPU für Betriebssystemaufgaben (für Programm) aufwendet (kernel mode)
- Linux/UNIX: `time` kommando, z.B.: `time firefox`

Arithmetisches Mittel

$$\overline{ET} = \frac{1}{n} \cdot \sum_{i=1}^{n} ET_i \quad [s]$$

Gewichtete Ausführungszeit (weighted ET)

- Für ungleich verteilte Workloads
- Z.B. 20% der Tasks betreffen P1, 80% P2, dann sind Gewichte $g_1 = 0.2$ bzw. $g_2 = 0.8$

$$WET = \sum_{i=1}^{n} g_i \cdot ET_i \quad [s]$$

Gesetz von Amdahl

Definiert den Gesamt-Speedup S, der mit einer Teil-Verbesserung erzielt werden kann!

$$S = \frac{ET_{alt}}{ET_{neu}}$$

$$ET_{neu} < ET_{alt}$$

$$S > 1$$

Fraction Enhanced F_E

- **Anteil** der Gesamt-ET, der verbessert (beschleunigt) ist
- Bsp: Von einer Gesamt-ET von 60s wurden 20s verbessert $\Rightarrow F_E = \frac{20}{60} = \frac{1}{3}$
- Es gilt: $F_E \leq 1$

Speedup Enhanced S_E

- **Faktor**, um den der erwähnte Anteil der Gesamt-ET (F_E) verbessert (beschleunigt) ist
- Bsp: Für das gleiche Teil-Problem statt 5s nur noch 2s benötigt: $\Rightarrow S_E = \frac{5}{2}$
- Es gilt: $S_E > 1$

Herleitung

$$S = \frac{ET_{alt}}{ET_{neu}} = \frac{ET_{alt}}{ET_{alt} \cdot \left(\underbrace{(1 - F_E)}_{unverbessert} + \underbrace{\frac{F_E}{S_E}}_{verbessert} \right)} \Rightarrow$$

Amdahl's Law

$$\Rightarrow \mathbf{S} = \frac{\mathbf{1}}{(\mathbf{1} - \mathbf{F_E}) + \frac{\mathbf{F_E}}{\mathbf{S_E}}}$$

Beispiel fernab der IT (?)

- Wir arbeiten alleine an einem Projekt
- 40% der Zeit für geht für Meetings, Koordination, etc. drauf
- 60% der Zeit kann produktiv gearbeitet werden
- Projekt verzögert sich, 5 neue Mitarbeiter eingestellt
- Einfache Annahme: 40:60-Verhältnis gilt weiterhin
- Mit welchem Speedup ist maximal zu rechnen?
- Warum wird dieser praktisch wahrscheinlich nicht halten?

$$S = \frac{1}{(1 - F_E) + \frac{F_E}{S_E}} = \frac{1}{(1 - 0.6) + \frac{0.6}{6}}$$

$$= \frac{1}{0.4 + 0.1} = \frac{1}{0.5} = \mathbf{2}$$

- **Mitarbeiterzahl versechsfacht** ($1 \Rightarrow 6$)**, aber nur Speedup von 2!**
- Absurdes Experiment: Mitarbeiterzahl unendlich groß machen: $S = \frac{1}{1 - F_E + 0} = 2.5$

Anwendung

- **Gibt Hinweise, inwieweit eine Verbesserung überhaupt die Gesamt-Performance steigert (steigern kann)**
- **Gibt eine Grenze des Speedups an, der vor allem von der F_E abhängt** ($\lim_{S_E \to \infty} S = \frac{1}{1 - F_E}$)
- Hilft auch bei der Beurteilung, welche von zwei Verbesserungsvarianten besser ist!

Beispiel aus der IT

- Gesamt-ET eines Programm: 30% CPU-Berechnungen, 70% Warten auf Daten aus Arbeitsspeicher
- Zwei alternative Varianten der Verbesserung:
  1. Neue CPU kaufen, die 1.5 mal so schnell rechnen kann
  2. Neues Mainboard/Speicher kaufen, Transfer zwischen Speicher/CPU 1.4 mal so schnell
- Zeigen Sie, welcher Ansatz besser ist, indem Sie den jeweiligen Speedup berechnen und dann vergleichen!

Variante 1, CPU-Verbesserung

$$S = \frac{1}{(1 - 0.3) + \frac{0.3}{1.5}} = 1.11\ldots$$

Variante 2, Speicherverbesserung

$$S = \frac{1}{(1 - 0.7) + \frac{0.7}{1.4}} = 1.25\ldots$$

Lösung

Variante 2 ist besser!
(solange man Anschaffungskosten nicht miteinbezieht...)

- Amdahl's Law:
  - $S = \frac{ET\,alt}{ET\,neu} = \frac{ET\,alt}{ET\,alt * \left((1 - F_e) + \frac{F_e}{S_e} \right)} = \frac{1}{(1 - F_e) + F_e / S_e}$

REP-BSP4

3. Gesetz von Amdahl

(a) Die Performance von Floating Point Square Root (FPSQR) Instruktionen variiert sehr stark bei verschiedenen (Grafik-)Prozessoren. Angenommen, die FPSQR Instruktionen sind verantwortlich für 20% der Ausführungszeit eines Grafik-Benchmarks. Vergleichen Sie zwei Verbesserungsansätze:

- die FPSQR Instruktionen werden um den Faktor 10 beschleunigt
- alle Floating Point (FP) Instruktionen werden direkt im Grafikprozessor um den Faktor 1.8 beschleunigt, wobei FP Instruktionen für 45% der Ausführungszeit verantwortlich sind.

Welcher der beiden Ansätze ist besser?

(b) Ein unbekannter Anteil x eines Programms konnte um den Faktor 7 beschleunigt werden, wodurch die Gesamtausführungszeit des Programmes um einen Speedup von 4 verbessert werden konnte.
Berechnen Sie den Anteil der beschleunigt bzw. nicht beschleunigt werden konnte.

$$\frac{n}{(1-n)\ \frac{n}{boost}}$$

1) $\frac{1}{(1-0,2)+\frac{0,2}{10}}$

$0,2 : 10 = 0,02$
20
20

$\frac{1}{0,8+}$

$0,82$ $\quad \frac{1}{0,82} = \underline{1,22}$

2) $\frac{1}{(1-0,45)+\frac{0,45}{1,8}}$

1,8
3,6

$0,45 : 1,8 = 0,25$
4,5
,90

$0,55 + 0,25 = 0,80$

$= \underline{1,25}$ => besser

b) $\frac{1}{(1-n)+\frac{n}{7}} = 4$

$1 = 4 \cdot (1-n+\frac{n}{7})$

$1 = 4 - 4n + \frac{4n}{7}$

$-3 = -4n + \frac{4n}{7}$

$-3 = -\frac{28n}{7} + \frac{4n}{7}$

$-3 = -\frac{24n}{7}$

$3 = \frac{24n}{7}$

$24 : 7 = 3,4$
30
20

$3 = 3,34n$

$3 / 3,34 = \underline{0,89}$

ca

CPU-Performance

Taktung

- **Alle Einheiten und Operationen in einer CPU müssen zeitlich synchronisiert werden**
- Taktgeber mit konstanter Frequenz erzeugt diskrete Zeitschritte
- Begriffe: *ticks, cycles, clock cycles, ...*
- **Zeitdauer eines Takts** t_{cc}, $[s]$
- **Taktfrequenz** (Taktrate) f, $[Hz]$ = Anzahl an Cycles pro Sekunde (entscheidend für die in einer Sekunde ausführbaren Instruktionen/Operationen)
- Jede Instruktion benötigt eine gewisse Anzahl an Cycles
- **Ein Programm ist eine Abfolge von Instruktionen**

CPU-Zeit t_{CPU}

- $t_{CPU} = (CPU\ clock\ cycles\ fuer\ ein\ Programm) \cdot t_{cc}$
- $t_{CPU} = \frac{(CPU\ clock\ cycles\ fuer\ ein\ Programm)}{f}$
- $t_{cc} \ldots$ Zeitdauer eines Takts [s]

Clock Cycles Per Instruction (CPI)

- Programm = Anzahl von Instruktionen ⇒ Instruction Count (IC)
- Wichtig: Mittlere Anzahl an **Zyklen pro Instruktion**
- Auch invers (Instructions per Cycle, IPC) verwendet

$$CPI = \frac{(CPU\ clock\ cycles\ pro\ Programm)}{IC}$$

CPU-Zeit

- $t_{CPU} = IC \cdot t_{cc} \cdot CPI$
- $t_{CPU} = \frac{IC \cdot CPI}{f}$

Überprüfung

$$\frac{Instruktionen}{Programm} \cdot \frac{ClockCycles}{Instruktion} \cdot \frac{Sekunden}{ClockCycle} = \frac{Sekunden}{Programm} = ET_{CPU}$$

Wichtige Kriterien für CPU-Performance

- Clock cycle time bzw. invers: Taktrate
- Cycles per Instruction bzw. invers: Instructions per Cycle
- Instruction count, also “Größe” des Programms

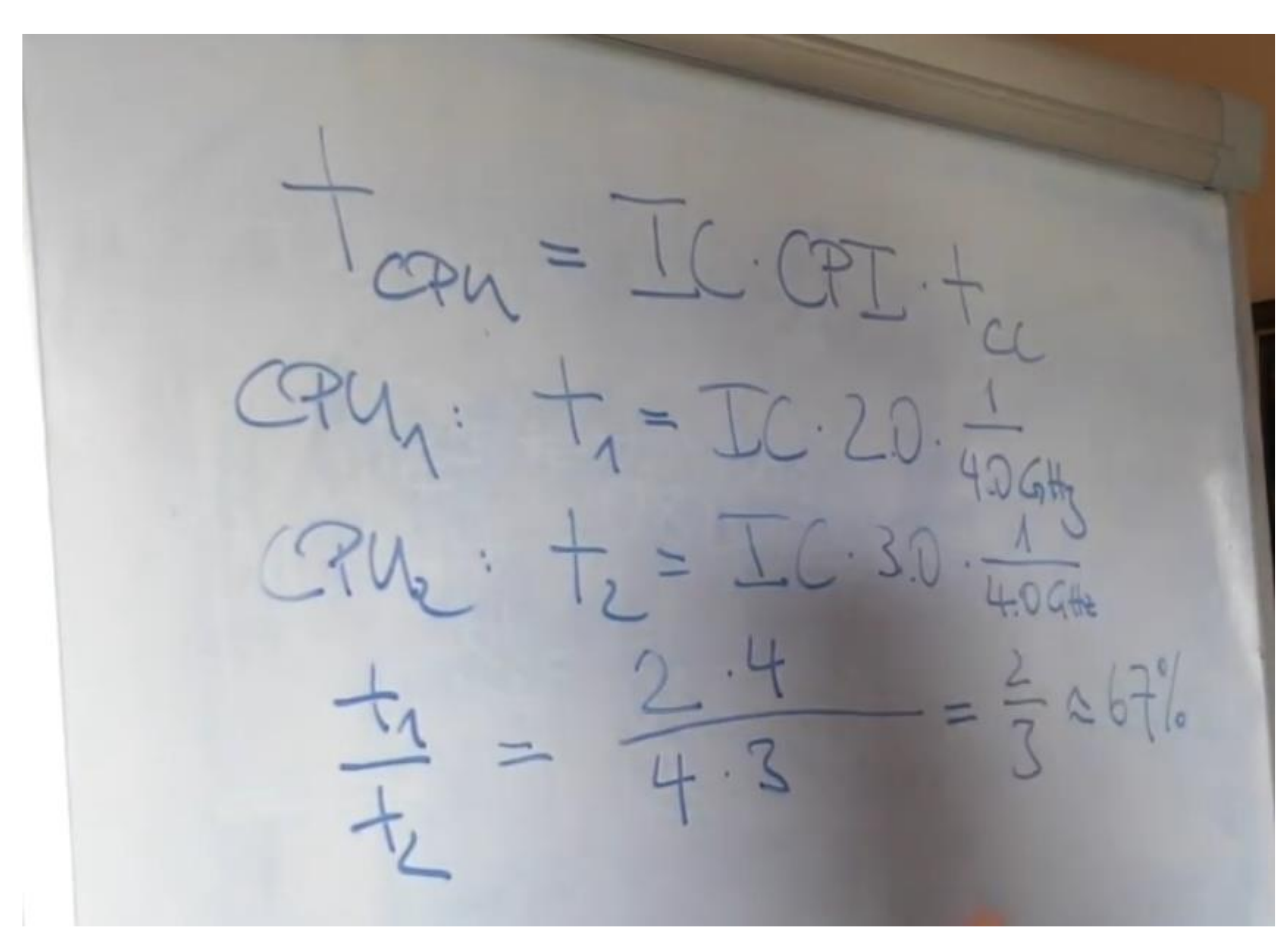
CPU-Zeit bei verschiedenen Instruktionen

- **Programme bestehen fast immer aus mehreren, verschiedenen Arten von Instruktionen**
- **Verschiedene Instruktionen benötigen oft unterschiedliche Mengen an Zyklen**
- IC_i: Anzahl der Instruktionen vom Typ i in einem Programm
- CPI_i: Mittlere Anzahl der Zyklen/Instruktion vom Typ i

CPU-Zeit

$$\mathbf{CpuZeit} = \left(\sum_{i=1}^{n} IC_i \cdot CPI_i \right) \cdot t_{cc}$$

Reichel Lösung



REP-BSP5

4. **Prozessorperformance**

Auf zwei unterschiedlichen Prozessoren läuft dasselbe Programm (z.B.: MP3-Encoder), d.h. die Prozessoren operieren mit dem gleichen Befehlssatz. Der eine Prozessor hat einen Prozessortakt von 3.0 GHz (CPU_1), der andere 4.0 GHz (CPU_2). Die CPIs liegen bei den beiden Prozessoren bei 2.0 (CPI_1) und 3.0 (CPI_2).

(a) Vergleichen Sie die CPU-Zeiten beider Prozessoren. Welche CPU ist schneller, und um wie viel?

PS: ein relativer Vergleich (zB. "... um X% schneller... ", "... um den Faktor X langsamer...") ist wohl sinnvoller als ein absoluter Vergleich.

(b) Berechnen Sie den Vergleich nochmals mit folgenden Zahlen: Beide Prozessoren haben einen Takt von 4.0 GHz, CPI bleiben unverändert.

Zeit pro Takt

$$t_{CPU} = \frac{IC \cdot CPI}{f} \rightarrow \text{Zyklen pro Instruktion}$$

1) $\frac{IC \cdot 2}{3} = \frac{2}{3}$ → $\frac{8}{12}$

2) $\frac{IC \cdot 3}{4} = \frac{3}{4}$ → $\frac{9}{12}$

$8/9 = 0{,}12 \Rightarrow 12\%$

b) $\frac{2 \cdot 4}{4 \cdot 3} = \frac{2}{3} = 67\%$

b) $\frac{IC \cdot 2}{4} = \frac{200 \cdot 2}{4} = 100$

$\frac{IC \cdot 3}{4} = \frac{600}{4} = 150$

a) 0,5
b 0,25

50% schneller → 66% von b

REP-BSP5 alternativ

4. **Prozessorperformance**

Auf zwei unterschiedlichen Prozessoren läuft dasselbe Programm (z.B.: MP3-Encoder), d.h. die Prozessoren operieren mit dem gleichen Befehlssatz. Der eine Prozessor hat einen Prozessortakt von 3.0 GHz (CPU_1), der andere 4.0 GHz (CPU_2). Die CPIs liegen bei den beiden Prozessoren bei 2.0 (CPI_1) und 3.0 (CPI_2).

(a) Vergleichen Sie die CPU-Zeiten beider Prozessoren. Welche CPU ist schneller, und um wie viel?

PS: ein relativer Vergleich (zB. "... um X% schneller... ", "... um den Faktor X langsamer...") ist wohl sinnvoller als ein absoluter Vergleich.

(b) Berechnen Sie den Vergleich nochmals mit folgenden Zahlen: Beide Prozessoren haben einen Takt von 4.0 GHz, CPI bleiben unverändert.

CPU 1 = 3 Ghz 2 CPI

CPU 2 = 4 Ghz 3 CPI

$$t_{CPU_1} = \frac{IC \cdot 2}{3} \Rightarrow \text{Bsp} \quad \frac{150 \cdot 2}{3} = 100$$

$$t_{CPU_2} = \frac{IC \cdot 3}{4} \Rightarrow \text{Bsp} \quad \frac{150 \cdot 3}{4} = \frac{450}{4} = 112{,}1$$

⟹ CPU 2 braucht 12 % länger als CPU 1 zum berechnen des Programms

b)

$$t_{CPU_1} = \frac{IC \cdot 2}{4} \Rightarrow \text{Bsp} \quad \frac{200 \cdot 2}{4} = 100$$

$$t_{CPU_2} = \frac{IC \cdot 3}{4} \Rightarrow \text{Bsp} \quad \frac{200 \cdot 3}{4} = \frac{600}{4} = 150$$

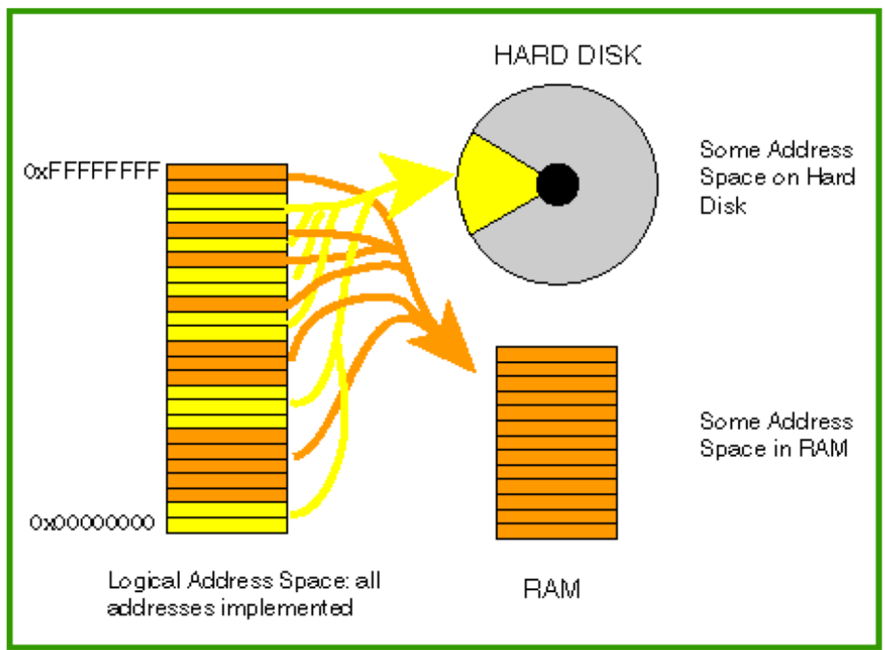
⟹ CPU2 braucht 50 % länger zum Berchnen des Programms als CPU1 zum berchnen des Programms

Virtueller Speicher

Mangel an Arbeitsspeicher – Mögliche Lösungen

- Früher: *Overlays*, Programmteile manuell ausgelagert
- Probleme: Manuell, ist für jede Maschine anders
- **Virtueller (logischer) Adressraum: unabhängig vom physikalisch vorhandenen Arbeitsspeicher**
- Betriebssystem lagert bei Speichermangel Bereiche (**Pages**) auf Massenspeicher aus (**Paging, Swapping**)
- Paging ist **transparent** für Programmierer (muss keine Rücksicht nehmen, ob Adressen im *real* vorhandenen *physikalischen Arbeitsspeicher* wirklich existieren)
- **Virtuelle Adresse $\Rightarrow$ physikalische Adresse**
- Hardware-Baustein: Memory Management Unit auf CPU

Entwickler → sieht unendlich ram → zu wenig platz → seiten auf festplatte laden



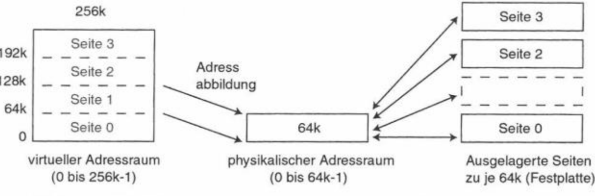


Abbildung 10.1: Die Seitenauslagerung ermöglicht einen beliebig großen virtuellen Adressraum. In diesem Beispiel befindet sich gerade Seite 1 im physikalischen Speicher, die anderen Seiten sind ausgelagert.

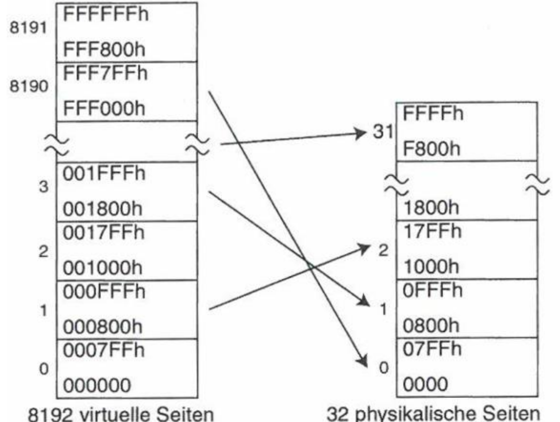
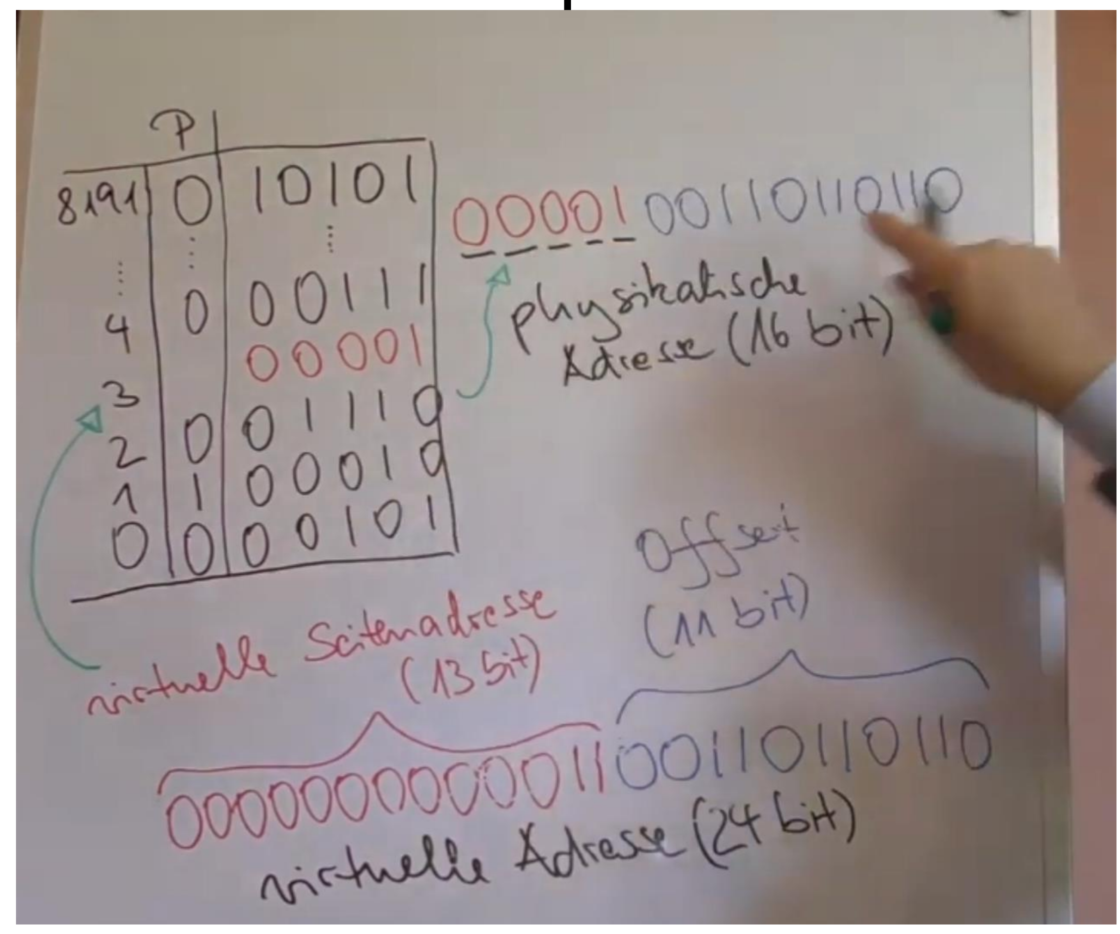


Abbildung 10.2: Beispiel zum Paging: Ein virtueller Adressraum von 16 MByte, wird mit einer Seitengröße von 2 KByte auf einem physikalischen Adressraum von 64 KByte abgebildet.

Virtueller Speicher 2



Virtuelle Adresse = 24 bit
Physikalisch aber nur 16 bit

Virtuelle Seitenadresse = index der Tabelle

Wert in der Tabelle = anfang der physischen Tabelle

→Tabelle verknüpft virtuell und physisch

0/1 in tabelle ob nicht im ram / im ram

Paging

Page size: 512 Byte bis 4 MiB, typisch sind 4 KiB

⇒ zu groß: Ein-/Auslagerung dauert zu lange

⇒ zu klein: HDD Seek Time problematisch

- **Page table**: Verwaltung der Pages
- **Page fault**: Page wird gebraucht, wurde aber ausgelagert
- **Demand paging**: Physikalischer Speicher wird sukzessive mit benötigten Speicherbereichen befüllt (per page faults). Liefert nach einiger Zeit **working set** (Arbeitsmenge) des Programms.
- **Dirty bit**: Wurde eine Seite überhaupt geändert, oder muss sie bei Auslagerung gar nicht auf die Disk zurückgeschrieben werden?

Was passiert wenn physikalischer Speicher voll ist?

- Wenn physikalischer Speicher voll ist, muss vor dem Einlagern einer angeforderten Page (Page-In) eine andere Page ausgelagert (Page-out) werden. **Welche**?
- Diejenige, die möglichst lange nicht mehr gebraucht wird
- Lässt sich aber nur per Wahrscheinlichkeit ausdrücken:
  - **LRU**-Ersetzung: die am längsten nicht gebraucht wurde
  - **LFU**-Ersetzung: die am seltensten gebraucht wurde
  - **FIFO**-Ersetzung: die am längsten im Speicher ist
- **Thrashing**: System ist (fast) nur mit Einlagerung und Auslagerung beschäftigt und kann kaum (gar nicht) Programm abarbeiten

Caches Allgemein

Motivation

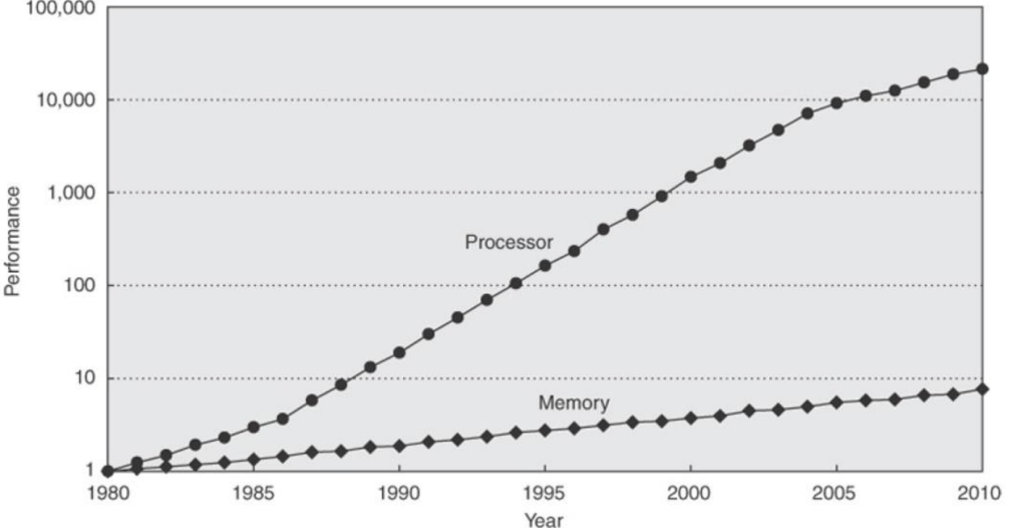
- CPUs in letzten 30 Jahren um Größenordnungen schneller
- (Haupt-)Speicher auch schneller, aber nicht in diesem Maße ⇒ Speicherzugriff dauert oft mehr als 10 CPU-Takte
- **CPU-Designziel: in jedem Takt Befehl abarbeiten**
- Bei superskalaren CPUs sogar noch mehr Befehle/Takt
- Daten und Code müssen schnell genug aus Speicher geliefert werden, sonst sind alle CPU-Fortschritte sinnlos!

⇒ **Speicher-Latenzzeit**: Wartezeit, bis Speicherzugriff anläuft

⇒ **Speicherbandbreite**: Übertragungsrate

... müssen beide verbessert werden

- ⇒ **Caches: kleine, schnelle Zwischenspeicher**



Caches Allgemein 2

Auf (Haupt-)Speicher wird meist nicht völlig zufällig zugegriffen

1. **Räumliche Lokalität**: Häufig Zugriffe auf Adressen, die in der Nähe kürzlich benutzter Adressen liegen
2. **Zeitliche Lokalität**: Folgezugriffe auf kürzliche benutzte Adressen

⇒ **Kürzlich benutzte Daten möglichst lange im Cache halten**

- Nach Hauptspeicherzugriff wird nicht nur Inhalt der adressierten Speicherzelle im Cache aufbewahrt (zeitliche Lokalität), sondern gleich ganzer Speicherblock (räumliche Lokalität), in dem die Speicherzelle liegt

"Datum": Eine kleine Datenmenge im Cache

- Cache zunächst leer, füllt sich bei Zugriffen: Kopie des Speicherblocks wird im Cache abgelegt
- Lesezugriff auf Hauptspeicher: **Cache auf Kopie prüfen**
- **Cache-Hit: Datum vorhanden, aus Cache lesen**
- **Cache-Miss: Datum nicht vorhanden, aus Hauptspeicher lesen – mehrfacher Aufwand**
- **Hit-Rate: Wahrscheinlichkeit für Cache Hit**

Look-Through-Cache vs. Look-Aside-Cache

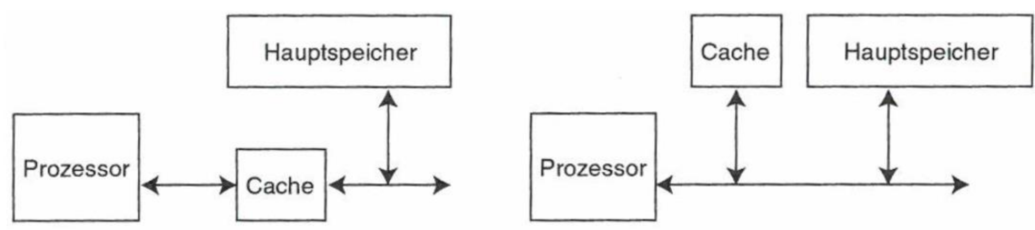


Abbildung 10.8: Ein Cache kann zwischen Hauptspeicher und Prozessor liegen (Look-Through-Cache, links) oder parallel zum Hauptspeicher (Look-aside-Cache,

Look-Through-Cache

- Zwischen CPU und Hauptspeicher
- Schnell an CPU angebunden (höherer Bustakt als Hauptspeicher)
- Cache und Hauptspeicher nicht gleichzeitig aktivierbar (Hauptspeicherzugriff erfolgt erst, wenn Daten im Cache nicht gefunden wurden)

⇒ Bei guter Hit-Rate fast nur Zugriffszeit des Caches entscheidend

⇒ Bei schlechter Hit-Rate hohe Wartezeiten

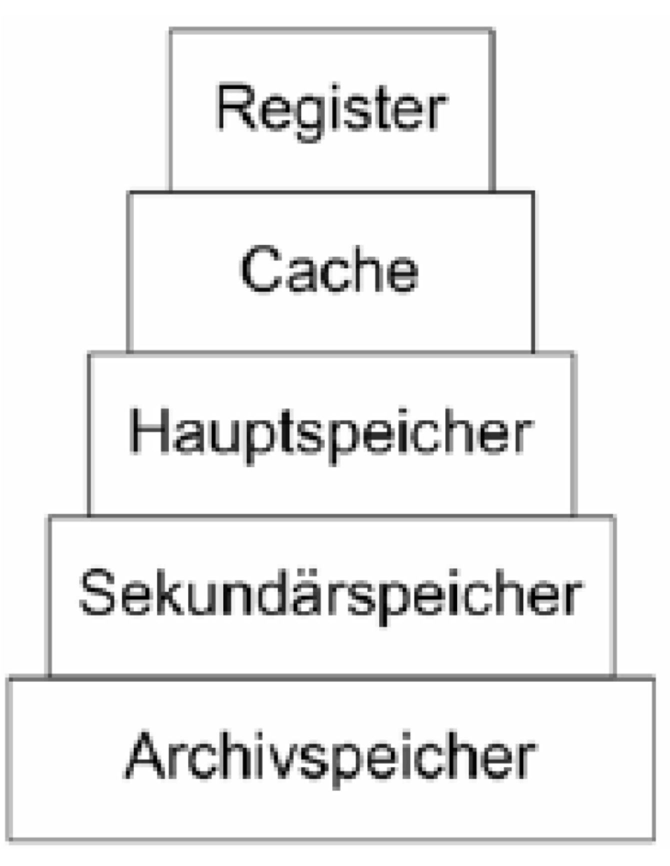
Look-Aside-Cache

- Parallel zum Hauptspeicher
- Cache-Zugriff und Hauptspeicher-Zugriff werden gleichzeitig aktiviert

⇒ Kein Zeitverlust bei Fehltreffer im Cache

⇒ Aber: Cache wird durch langsamen Speicherbus getaktet

Speicherhierachie



Einstufige Caches oft ineffizient ⇒ mehrstufige Caches

1. **First-Level (L1) Cache:**
   - Kleiner Cache, mit CPU-Takt betrieben; ca. 16-64 KiB
   - Meist in Prozessorchip integriert
   - *Split cache:* Daten / Code; gleichzeitiger Zugriff möglich
2. **L2-Cache**
   - Hinter L1-Cache; f<CPU-Takt; ca. 256-512 KiB
   - Meist in Prozessorchip integriert
   - *Unified cache:* Daten und Code gemeinsam
3. **L3-Cache**
   - Hinter L2-Cache; oft schnelles SRAM auf Systemplatine
   - Bei MultiCore-CPUs meist geteilt zwischen einzelnen Cores (L1 und L2 extra für jeden Core)
4. ***L4-Cache*** *Prinzip: Hauptspeicher als Cache für HDD ...*

Cache Performance

Noch anschauen

MSC, MP

- Memory Stall Cycles (MSC): Anzahl der Zyklen, die CPU auf Speicherzugriff warten muss
- Hängt von der Anzahl der Cache Misses und den Kosten pro Cache Miss (Miss Penalty) ab
- MSC = Anzahl der Misses · MP

$= IC \cdot MPI \cdot MP = IC \cdot \frac{Misses}{Instruktion} \cdot MP$

$= IC \cdot \frac{Speicherzugriff}{Instruktion} \cdot \frac{Misses}{Speicherzugriff} \cdot MP$

- $MPI = \frac{Misses}{Instruktion} = \frac{Speicherzugriffe}{Instruktion} \cdot MR$
- $MR = \frac{Misses}{Speicherzugriff}$ (Miss Rate)

Mittlere Speicherzugriffszeit

- $\bar{t}_{SZ} = t_H + MR \cdot MP$
- $t_H \dots$ **Hit Time**, Zeit um **Treffer** im Cache zu erzielen
- $MR \dots$ Miss Rate, Wahrscheinlichkeit für Cache Miss
- $MP \dots$ Extra-Aufwand bei Cache Miss (z.B. L2, RAM, ..)

Für mehrstufige Caches

- $\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot MP_{L1}$
- $MP_{L1} = t_{H,L2} + MR_{L2} \cdot MP_{L2}$
- $\bar{t}_{SZ} = t_{H,L1} + MR_{L1} \cdot (t_{H,L2} + MR_{L2} \cdot MP_{L2})$

Average Memory Stalls per Instruction (AMSPI)

- Durchschnittliche Anzahl an Zyklen pro Instruktion, in denen auf Daten aus dem Speicher gewartet wird
- Annahme: L1-Cache wird mit vollem Prozessortakt betrieben $\Rightarrow$ erfolgreicher L1-Cache Zugriff braucht also keinen Extra-Takt (Zyklus)
- $AMSPI = MPI_{L1} \cdot t_{H,L2} + MPI_{L2} \cdot MP_{L2}$

Globale Werte !!!!! Speicherzugriff per Instruktion z.b 1,5 → 1,5x MR = MPI

Grob: 3 Varianten

1. **Miss Penalty (MP) senken**
   - Mehrstufige Caches
2. **Miss Rate (MR) senken**
   - Blockgröße erhöhen, nutzt räumliche Lokalität, erhöht MP
3. **Hit Time (HT) senken**
   - Kleiner, einfacher, schneller Cache; On-chip, CPU-Takt, direct mapped

Cache Performance Notes

- MSC = Wartezeit in Zyklen von CPU (= wieviele Misses + Wartezeit)

Cache Organisation

Bestandteile eines Caches

1. **Datenbereich**
   - **Besteht aus Anzahl von Cache-Zeilen** (*Cache Lines*)
   - Alle Lines haben **einheitliche Blocklänge** (*block/line size*)
   - Immer ganze Cache Line laden/ersetzen
2. **Identifikationsbereich**
   - **Enthält *tags*** (Etikett): Aus welchem Abschnitt des Hauptspeichers wurde Cache-Zeile geladen
   - Stellt fest, ob ein Datum im Cache ist oder nicht
3. **Verwaltungsbereich**: Mindestens zwei Bits
   - **Valid** bit: Ist Cache-Line ungültig (veraltet, `0`) oder gültig (`1`)
   - **Dirty** bit: Wurde auf Cache-Line geschrieben (`1`) oder nicht (`0`) – für Copy-Back-Ersetzungsstrategie wichtig

Mögliche Organisationsformen

1. **Vollassoziativ** (fully associative)
2. **Direkt abbildend** (direct mapped)
3. **Mehrfach assoziativer Cache (n-way set associative, teilassoziativ)**, am häufigsten zu finden

Fully associative

- **Datenblock in beliebiger Cache-Zeile abspeicherbar**

  $\Rightarrow$ **alle** Tags mit angefragter Adresse zu vergleichen
- Muss aus Performance-Gründen gleichzeitig erfolgen

  $\Rightarrow$ Vergleichereinheit für **jede** Cache-Zeile

  $\Rightarrow$ **sehr hoher Hardwareaufwand**
- **Dafür keine Einschränkungen bei Ersetzung!**

  $\Rightarrow$ **höchste Trefferquote aller Organisationen**

Direct mapped

- **Teil der Speicheradresse wird als Index benutzt, um dem Datenblock eine Cache-Zeile zuzuordnen.**

  $\Rightarrow$ **Block kann nur dort gespeichert werden**

  $\Rightarrow$ **es gibt keine Ersetzungsstrategie**
- Kann zu Thrashing führen = schlechter als gar kein Cache
- Aus Index sofort klar, wo Block stehen muss (falls im Cache)
- Einfacher Aufbau, nur ein Vergleicher notwendig

  $\Rightarrow$ **schlechteste Trefferquote aller Organisationen**

n-way set associative

- **Kompromiss zw. fully associativ und direct mapped**
- Cache-Zeilen werden zu einem *Satz* zusammengefasst
- **Satznummer wird als Index benutzt:**

  $\Rightarrow$ aus Adresse ergibt sich nur Nummer des Satzes

  $\Rightarrow$ innerhalb des Satzes kann frei gewählt werden

  $\Rightarrow$ Ersetzungsstrategien möglich
- **Trefferbestimmung:**

  $\Rightarrow$ Index verweist auf Satz, wo Datum stehen könnte

  $\Rightarrow$ innerhalb von Satz kann Datum in n Einträgen liegen

  $\Rightarrow$ n Vergleicher: parallel Tags der n Einträge vergleichen

Cache Organisation 2

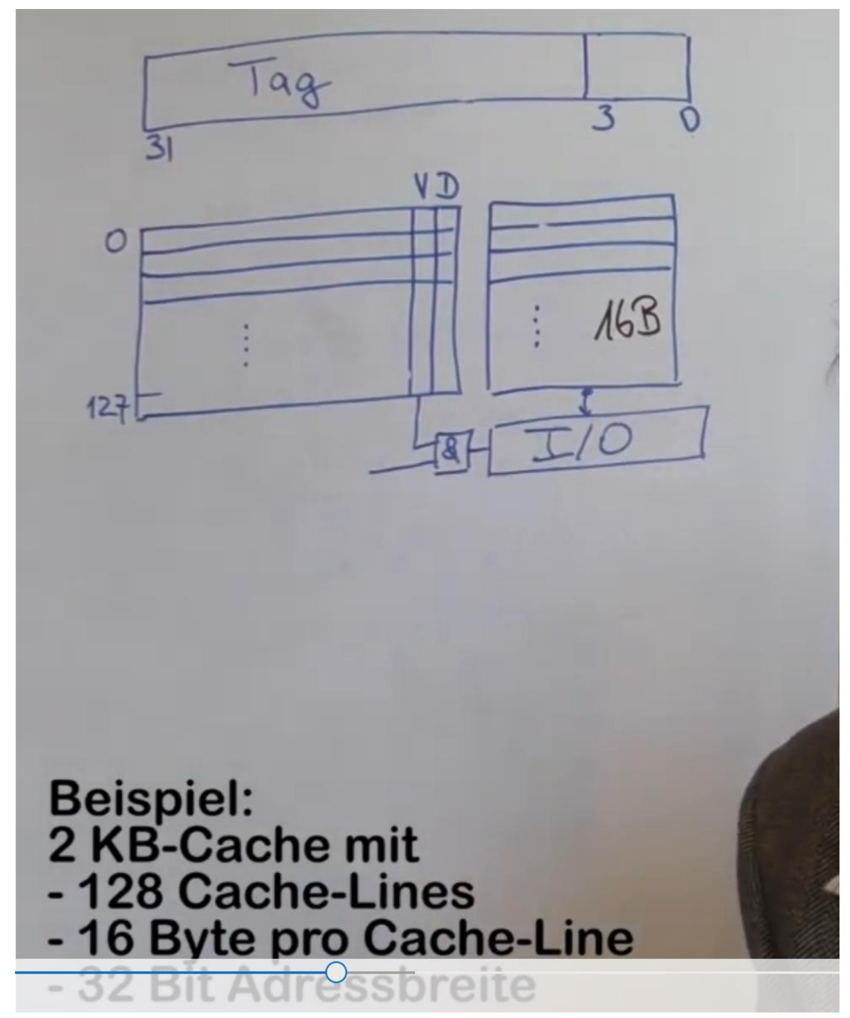
- Wie bei Caching Problem – was lade ich nach?
- Ersetzungsstrategien wie beim Paging
- → random gute idee zufälliger wird rausgeschmissen (simpel)
- Alle Datenblöcke gleich groß (16 byte)
- Assoziativ → index wird durchsucht (128 vergleicher benötigt)
- Direkt Assoziativ → 7 bit als Index auswählen → daher genau an einer stelle → Problem trashing 2x mal genauer wert z.b
- Teil Assoziativ → Mittelding → Index 6 bit (weniger) → Teil wo es reinkommen kann = Satz → hat n größe (n vergleicher)
- = n-way set associative

Cache Organisation 3

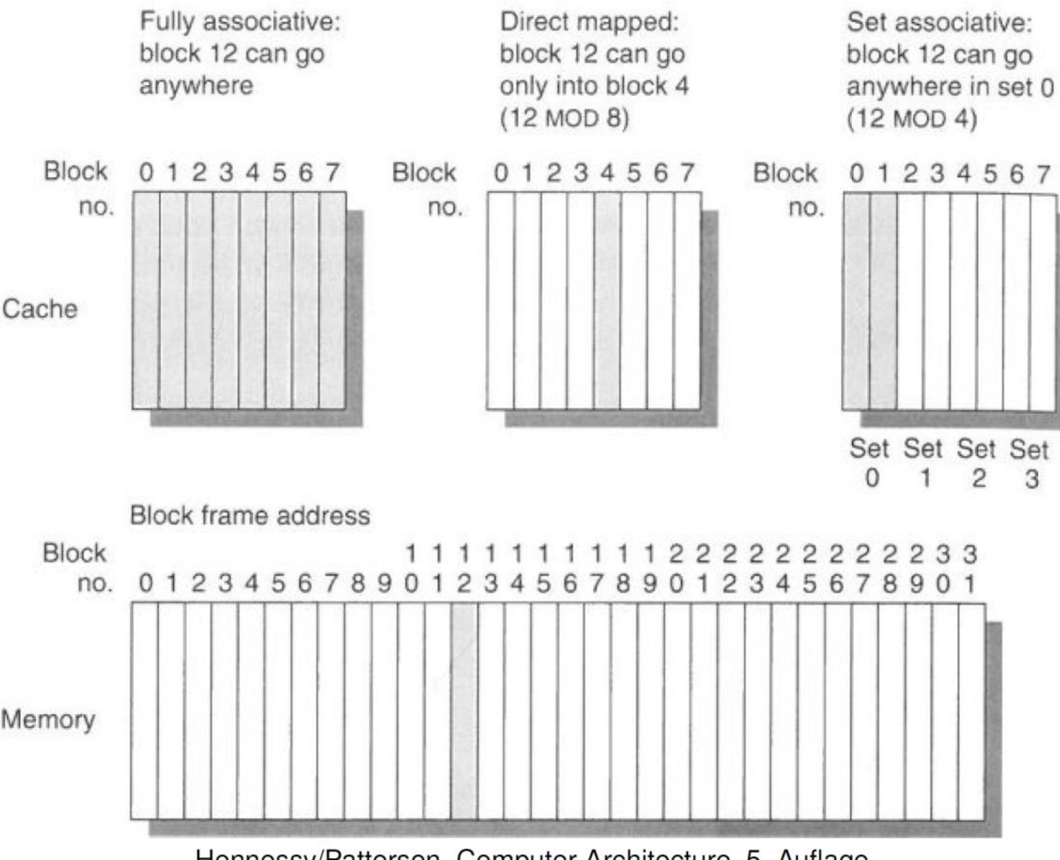
Problem → man hat selben Speicher 2x auf hdd und cache beides zusammen → sollen aber alle gleich sein → nur eine Version

Schreib vs Lesezugriff →
Lesezugriff – hit oder miss(laden cache)(räumliche lokalität)
Schreibzugriff →

- Write through –> sofort auch in hauptspeicher (hoher aufwand) aber konsistent (bzw alle caches updaten dahinter)
- Copy Back(Write later) → valid bit 1 = aktueller stand
  valid bit 0 = veraltet → wenn ganzer block ersetzt wird wird die valide line zurückgeschrieben
  (Zwischen durch inkonsistent)



Cache Organisation 2

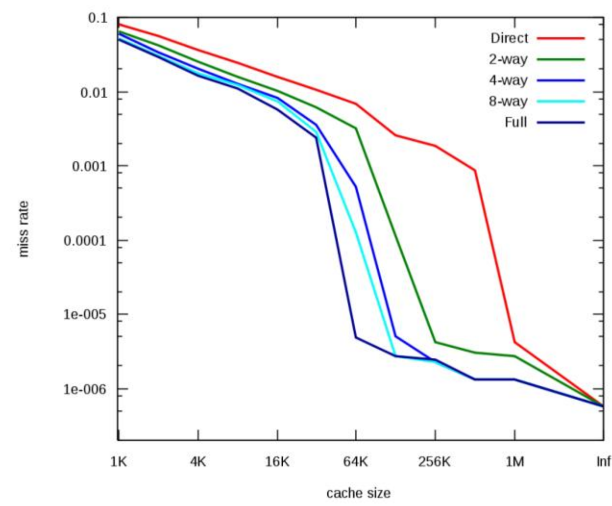


Hennessy/Patterson, Computer Architecture, 5. Auflage

AMD K10

1. L1
   - 64 KB 2-way set associative instruction cache
   - 64 KB 2-way set associative data cache
2. 512 KB 16-way set associative cache
3. 32-way set associative L3 shared cache

Vergleich per Benchmark



Ersetzung/Aktualisierung

Ersetzungsstrategien

- Analog zu virtuellem Speicher
- LRU, LFU, FIFO, ...
- Random: Zufallsprinzip $\rightarrow$ erstaunlich gute Ergebnisse

Aktualisierungsstrategien

Wichtige Begriffe

- **Konsistenz**: **Alle** Kopien eines Datums inhaltsgleich
- **Kohärenz**: Korrektes Fortschreiten des Systemzustands
- Inkohärenz, wenn Dateninkonsistenz nicht bereinigt wird
- Lesen vs. **Schreiben**

Lesen

- Lese-Treffer: Daten zeitsparend aus Cache nehmen; keine weiteren Aktionen
- Lese-Fehltreffer: Block, der Datum enthält, in Cache laden; angefordertes Datum in CPU laden

Schreib-Treffer

Zwei Möglichkeiten:

1. **Write-Through-Strategie**: Datum wird in Cache und in Hauptspeicher geschrieben: Einfach, garantiert Konsistenz
2. **Copy-Back (Write-Later)-Strategie**: Datum wird nur im Cache aktualisiert und betroffene Cache-Zeile als verändert markiert (Dirty Bit wird auf `1` gesetzt).
   - Spätestens vor Ersetzung muss Datum in Hauptspeicher zurückgeschrieben werden (write back, copy back)
   - Vorteilhaft z.B. bei viel benutzter Variable (z.B. Schleifencounter)
   - System jedoch teilweise inkonsistent

Schreib-Fehltreffer

Auch zwei Möglichkeiten:

- **No-Write-Allocate-Strategie:** Datum nur in Hauptspeicher schreiben, liegt danach nicht im Cache vor
- **Write-Allocate-Strategie:** Datum in Hauptspeicher schreiben und zusätzlich den entsprechenden Block in Cache laden

Sekundärspeicher

Sekundärspeicher

- **Speichermedien mit persistenter (dauerhafter) Speicherung**
- Optische Medien: CD, DVD, BluRay, ...
- **Magnetische Medien**: Festplatte (HDD), Floppy, ...
- Elektronische Medien: v.a. Flash-Speicher (USB-Stick, SSD, ...)
  - Keine mechanischen Verzögerungen
  - Dafür meist unterschiedliche Zeiten für Lesen/Schreiben
  - Vor dem Schreiben muss meist gelöscht werden

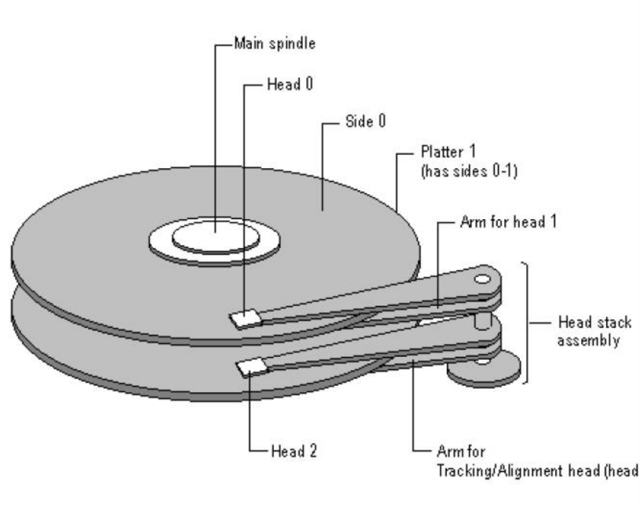
Aufbau

- Durch Halbleiterbausteine realisiertes, nichtflüchtiges Speichermedium (zB. EEPROM, FLASH)
- Üblicherweise teurer als HDDs (per GB)
- Aber auch schneller! **(Seek time entfällt)**
- Geringer Stromverbrauch, geräuschfrei
- Unempfindlich gegen Erschütterungen, sehr leicht
- **Limitierte Anzahl an Schreibvorgängen**
  - Nur ca. $10^4 - 10^6$ Schreibvorgänge pro Speicherzelle
  - "Wear Leveling"-Algorithmen: Daten werden gleichmäßig auf alle Speicherzellen verteilt

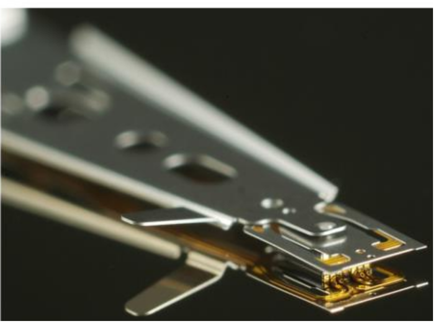
Festplatte

Aufbau

- Mehrere Scheiben (Platter) aus nicht magnetisierbarem Material (Substrate)
- Magnetisierbares Material wird aufgebracht
- Für jeden Platter: Schreib/Leseköpfe auf beweglichem Arm



Disk head



Schreiben

- **Bewegte Ladungen (=Strom) erzeugen ein Magnetfeld**
- Definierter Bereich der Oberfläche wird magnetisiert
- Richtung des Stroms bestimmt Richtung der Magnetisierung (logisch 0/1)

Lesen

- **Veränderliches Magnetfeld induziert Strom in einem Leiter**
- Magnetoresistiver (MR) Sensor, Widerstand hängt von Richtung der Magnetisierung ab. Per Lesestrom wird Widerstand (über Spannungsabfall) gemessen

Fragmentierung

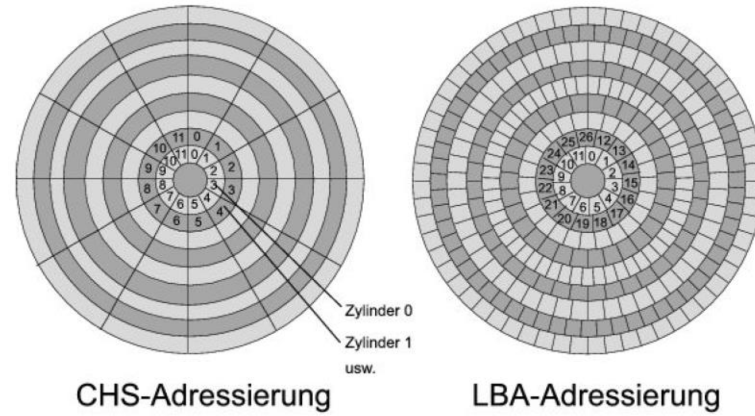
- Verstreute Speicherung logisch zusammengehörender Datenblöcke des Dateisystems auf einem Datenträger
- Führt speziell bei hohen Zugriffszeiten (seek time, HDD) zu spürbarer Verlangsamung der Lese- und Schreibvorgänge

Defragmentierung

- Neuordnung von fragmentierten Datenblöcken
- Logisch zusammengehörende Datenblcke sollten aufeinander folgen
- Kann den sequentiellen Zugriff beschleunigen

Festplatte- Speichern

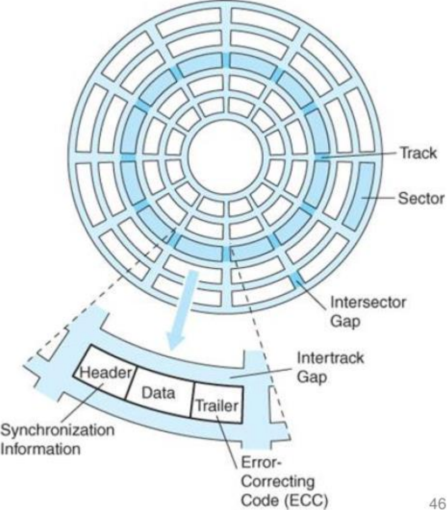
Aufbau – CHS (Cylinder, Head, Sector) vs. LBA (Logical Block Addressing)



Datenorganisation bei CHS-Adressierung

Vereinfacht:

- Jeder Platter in viele Spuren (Tracks, konzentrische Kreise) aufgeteilt
- Jede Spur in viele Sektoren unterteilt, Einheit für Datentransfer
- Sektorgröße früher meist 512 Byte, heute auch oft 4 KiB



Performance bei CHS-Adressierung

Seek Time T_S

- **Aktuator (Lese/Schreibkopf) muss auf jeweilige Spur bewegt werden**
- Abschätzung $T_S = m \cdot n + s$
- $n \ldots$ Anzahl der überquerten Spuren
- $m \ldots$ Zeit pro Spur (Bruchteile einer ms)
- $s \ldots$ Startup-Zeit (wenige ms)
- **Meist wird durchschnittliche Seek Time angegeben, ca. 8-15ms**

Rotationsverzögerung T_r

- **Ist Aktuator auf richtiger Spur, dauert es, bis sich entsprechender Sektor unter dem Aktuator befindet**
- Abschätzung $T_r = \frac{1}{2r}$
- $r \ldots$ Umdrehungen pro Minute (RPM)
- **Im Durchschnitt ist man eine halbe Rotation von Sektor entfernt**
- Bsp r=10 000 RPM; $T_r = \frac{1}{2 \cdot 10\,000} min = 5 \cdot 10^{-5} min = 5 \cdot 10^{-5} \cdot 60s = 3 \cdot 10^{-3} s = 3ms$

Transferzeit T

- **Reine Übertragungszeit der Daten, Aktuator ist schon an richtiger Position**
- Hängt v.a. von RPMs und Datendichte ab
- $T = \frac{b}{r \cdot N}$
- $b \ldots$ Anzahl Bytes, die transferiert werden müssen
- $N \ldots$ Anzahl an Bytes auf der Spur

Gesamte Transferzeit T_a

$$T_a = T_s + T_r + T = T_S + \frac{1}{2r} + \frac{b}{rN}$$

Disk Scheduling

Disk Scheduling

- Leistungssteigerung nicht nur durch mechanische Verbesserungen oder Defragmentierung erzielbar
- Scheduling: Gezieltes Umreihen der Anfragen, mechanische Verzögerungen möglichst minimieren
- Strategien: FIFO/FCFS, SSTF, (C-)SCAN, (C)-LOOK

FIFO/FCFS

- First–In–First–Out; First–Come–First–Served
- Anfragen in Reihenfolge des Eintreffens bearbeiten
- Vorteil: Faire Abarbeitung
- Nachteil: Gesamtanzahl an überquerten Spuren sehr hoch

Das selbe

SSTF

- Shortest Seek Time First
- Immer Anfrage bearbeiten, deren Spur der momentanen Aktuator-Position am nächsten ist
- Vorteil: Jeweilige Seek Time wird verringert
- Nachteil: Durchschnittliche Seek Time nicht garantiert minimal

SCAN

- Aktuator läuft immer zwischen Endpositionen auf und ab
- Verhindert, dass Aufträge „verhungern“ (*Starvation*)

C-SCAN

- **C**: Circular
- Wie SCAN, beim Zurückkehren an Anfangsposition wird nichts abgearbeitet
- Reduziert Verzögerung für neu eintreffende Anfragen

LOOK

- Ähnlich zu SCAN, aber intelligenter
- Sobald letzte angefragte Spur einer Richtung erreicht wurde, ändert Aktuator die Laufrichtung
- Restliche, unnötige Spuren müssen also nicht überquert werden
- C-LOOK: Analog

Disk Scheduling Bsp.

Beispiel

- **Disk hat insgesamt 200 Spuren**
- In der Warteschlange (Queue) sind mehrere Anfragen
- Angefragte Daten belegen jeweils maximal eine Spur
- Aktuator befindet sich gerade auf Spur 100, war davor auf Spur 85 ⇒ **aufsteigende** Laufrichtung bei LOOK/SCAN
- Folgende Spuren sind zu Lesen:
- `55, 58, 39, 18, 90, 160, 150, 38, 184`

SSTF

Nächste Spur	Differenz
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
Gesamt	**248**
Schnitt	**27,6**

FIFO

Nächste Spur	Differenz
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
Gesamt	**498**
Schnitt	**55,3**

SCAN

Nächste Spur	Differenz
150	50
160	10
184	24
200	16
90	110
58	32
55	3
39	16
38	1
18	20
Gesamt	**282**
Schnitt	**31,3**

C-SCAN

Nächste Spur	Differenz
150	50
160	10
184	24
200	16
1	199
18	17
38	20
39	1
55	16
58	3
90	32
Gesamt	**388**
Schnitt	**43,1**

C-LOOK

Nächste Spur	Differenz
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
Gesamt	**322**
Schnitt	**35,8**

LOOK

Nächste Spur	Differenz
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
Gesamt	**250**
Schnitt	**27,8**

Bespiel: Vergleich

	FCFS	SSTF	SCAN	C-SCAN	LOOK	C-LOOK
Spuren	498	248	282	388	250	322
Schnitt	55,3	27,6	31,3	43,1	27,8	35,8

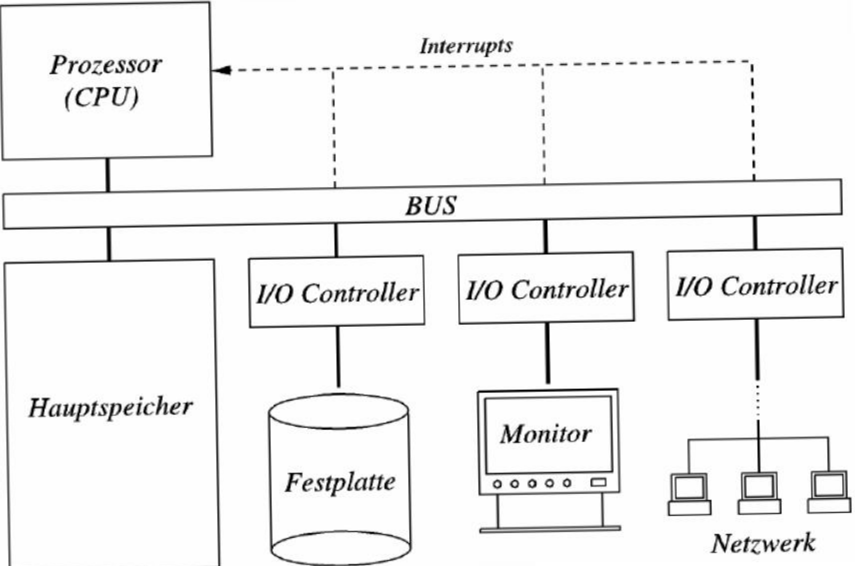
Nota bene

Es muss immer durch die Anzahl der zu bearbeitenden Spuren (hier 9) dividiert werden. Wenn ein Verfahren (hier SCAN und C-SCAN) **Wege** zurücklegt, **bei denen keine Arbeit erledigt wird, müssen** die **in die Summe eingerechnet werden**, der **Divisor wird aber NICHT erhöht!**

Von Neumann Architektur

Grundlegende Prinzipien

- Programmgesteuerter Rechner
- Kein fundamentaler Unterschied zwischen Daten und Programmcode
  - ⇒ gemeinsame Form (Bitmuster)
  - ⇒ gemeinsamer Speicher
  - ⇒ gemeinsamer Bus
- Sequentielle Abarbeitung eines Programms



Rechner Klassifikation nach Flynn

Anzahl der vorhandenen Datenströme

- SD: Single Data Stream
- MD: Multiple Data Streams

Anzahl der vorhandenen Befehlsströme

- SI: Single Instruction Stream
- MI: Multiple Instruction Streams

- **SISD**
- **SIMD**
- MISD
- **MIMD**

SISD - Single Instruction, Single Data

- **Sequenzielle Programmausführung (ein Prozessor)**
- **Princeton-Architektur („von Neumann-Rechner“)**
  - **Gemeinsamer Speicher für Instruktionen und Daten**
  - Streng sequentieller (deterministischer) Programmablauf garantiert
  - Bei wechselndem Daten-/Programmspeicher-Bedarf gute Auslastung: “Von Neumann Bottleneck”
- **Harvard-Architektur:**
  - **Getrennter Speicher für Instruktionen und Daten**
  - Zwei unabhängige (unterschiedlich breite) Busse
  - Daten und Befehle können gleichzeitig geladen werden
  - Aber: Race conditions, nicht-deterministischer Programmablauf

CISC

- **Complex** Instruction Set Computer
- Viele, verhältnismäßig mächtige Einzelbefehle
- **Mikroprogrammierung:** Sequenzen für Steuerung der CPU werden aus Mikrocode-ROM abgerufen
- Befehlssatz wurde immer größer (oft mehrere Hundert)
- Immer kompliziertere Befehle kamen dazu

CISC: Vorteile

- Flexibilität: Befehlssatz auf Software-Ebene erweiterbar
- Fehlerbehebung: Neuer Mikrocode auch beim Kunden einspielbar
- **Kompatibilität** – Emulation: Befehlssatz von Vorgängern auf SW-Ebene nachbildbar

CISC: Nachteile

- **Dekodierung der vielen komplexen Befehle aufwändig**
- Dekodierungseinheit brauchte mehr Zeit und Platz auf Chip

$\Rightarrow$ Bsp. Intel 386: Addition: 2 Takte; Multiplikation: bis zu 38 Takte

RISC

- **Reduced** Instruction Set Computer
- Wenige, einfache Maschinenbefehle

RISC

- **Kein Mikrocode, keine algorithmische Abarbeitung**
- **Befehl muss in Hardware implementiert sein**
- Befehlssatz ist kleiner, enthält keine komplizierten Befehle
- IBM, Mitte 70er: 10 einfache Befehle machen 2/3 des Programmcodes aus
- Rest durch Folge einfacher RISC-Befehle ersetzen
- **Skalarität**: Möglichst in jedem Takt einen Befehl bearbeiten. ($> 1 \Rightarrow$ superskalar)

CISC

- **Flexibilitat** Dem Prozessorbefehlssatz können auf Software-Ebene neue Befehle hinzugefügt
- werden. Das macht es leichter, den Prozessor weiter zu entwickeln und an die Bedürfnisse des
- Marktes anzupassen.
- **Fehlerbeseitigung** Design-Fehler können durch Einspielen eines neuen Mikrocodes sogar
- noch beim Kunden behoben werden.
- **Kompatibilit¨at und Emulation** Bei neuen Prozessorkonzepten kann auf Software-Ebene

RICS

Skalarität Es soll möglichst mit jedem Takt ein Befehl bearbeitet werden. Dieses Ziel ist sehr weitreichend und erfordert aufwändige konstruktive Maßnahmen nach sich. Prozessoren, die mehr als einen Befehl pro Takt bearbeiten, heißen *superskalar*. Skalare und superskalare Architekturen werden im Kap. 11 behandelt. Einfache RISC-Prozessoren erreichen nicht unbedingt Skalarität.

Verzicht auf Mikroprogrammierung Alle Befehle sind einer Hardwareeinheit zugeordnet („fest verdrahtet"). Dadurch sind nur einfache möglich, die schnell ausgeführt und dekodiert werden können.

Load/Store-Architektur Die Kommunikation mit dem Hauptspeicher wird nur über die Befehle Laden und Speichern (LOAD und STORE) abgewickelt. Dadurch wird der zeitkritische Transport zwischen Prozessor und Speicher auf ein Minimum beschränkt. ALU-Operationen können dementsprechend nur auf Register angewendet werden. Es gibt keine Befehle, die einen Speicheroperanden laden, bearbeiten und wieder speichern (Read-Modify-Write-Befehle), dies sind typische mikroprogrammierte Befehle.

Großer Registersatz Viele Register ermöglichen es, viele Variablen in Registern zu halten und damit zeitraubende Hauptspeicherzugriffe einzusparen. Üblich sind mindestens 16 Allzweck-Register, meistens deutlich mehr.

Feste Befehlswortlänge Alle Maschinenbefehle haben einheitliche Länge, das vereinfacht das Laden und Dekodieren der Befehle. Die Verlängerung des Codes, die durchaus 50% betragen kann, nimmt man in Kauf.

Horizontales Befehlsformat In den Maschinenbefehlen haben Bits an fester Position eine feste und direkte (uncodierte) Bedeutung; auch dies beschleunigt die Dekodierung.

Orthogonaler Befehlssatz Jeder Befehl arbeitet auch mit jedem Register zusammen.

Als Folge aus diesen Forderungen ergibt sich zwangsläufig ein **einfacher Befehlssatz** (Reduced Instruction Set). Es gibt nur wenige und einfache Befehle, die jeweils aus wenigen Teilschritten bestehen und schnell abgearbeitet werden können.

REP-BSP1

1. **Rechnerarchitekturen**

   (a) Welche der folgenden Operationen/Eigenschaften ist typisch für RISC bzw. für CISC?

   Erklären Sie Ihre Antwort.

   - Speicher / Register Operationen
   - Register / Register Operationen
   - Wenige einfache Arten der Adressierung
   - Viele aufwendige Arten der Adressierung
   - Eher wenige Register
   - Typischerweise sehr viele Register

   (b) Nennen Sie Vor- bzw. Nachteile der Harvard-Architektur gegenüber der Von Neumann-Architektur!

CISC -> viele komplizierte Befehle

RISC -> wenige gut optimierte

RISC -> load-store architektur => kein Speicherzugriff => Zugriff nur über Register => viele Register

-> Pipelining RISC => Superskalarität

-> Adressierung -> wenige Speicheradressierungen

CISC -> Zugriff Speicher

-> weniger Register

-> Mikroprogrammieren -> komplexe Befehle zerlegen

Harvard vs Neumann

↳ Daten -> eigener Speicher

Code -> eigener Speicher -> Code oder Daten laden -> dauert länger

-> Zugriffsrecht einfach -> kein Code überschreiben

-> Speicherverwaltung schwer

Neumann -> Bottleneck Bus -> Daten und Code

Pipelining

Ohne Pipelining

- Jede Stufte nur 1 von 5 Takten beschäftigt
- Auslastung 20%

Pipelining Prinzip

- Fließbandverarbeitung
- **Befehle überlappend und parallel durch verschiedene Bearbeitungsstufen schleusen**
- **In jedem Takt:**
  1. Fertigen Befehl herausnehmen
  2. Jeden andren Befehl in nächste Bearbeitungsstufe
  3. Neuen Befehl in erste Stufe

Praktische Probleme

Interlocks

- Z.B. Hauptspeicherzugriffe, kann nicht immer in einem Takt erfolgen
- Nur dann, wenn Datum in L1-Cache und dieser mit vollem Prozessortakt betrieben!

Data hazards (Datenabhängigkeiten)

- Z.B.: Befehl 1 schreibt in Register, das von Befehl 2 gelesen werden muss.
- Befehl 2 muss warten, bis Befehl 1 fertig ist
- Read-After-Write-Hazard (RAW-Hazard)

Ohne Pipelining

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	Befehl 1	×	×	×	×
2	×	Befehl 1	×	×	×
3	×	×	Befehl 1	×	×
4	×	×	×	Befehl 1	×
5	×	×	×	×	Befehl 1
6	Befehl 2	×	×	×	×
7	×	Befehl 2	×	×	×
8	×	×	Befehl 2	×	×
9	×	×	×	Befehl 2	×
10	×	×	×	×	Befehl 2

Pipelining

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	Befehl 1	×	×	×	×
2	Befehl 2	Befehl 1	×	×	×
3	Befehl 3	Befehl 2	Befehl 1	×	×
4	Befehl 4	Befehl 3	Befehl 2	Befehl 1	×
5	Befehl 5	Befehl 4	Befehl 3	Befehl 2	Befehl 1
6	Befehl 6	Befehl 5	Befehl 4	Befehl 3	Befehl 2
7	Befehl 7	Befehl 6	Befehl 5	Befehl 4	Befehl 3
8	Befehl 8	Befehl 7	Befehl 6	Befehl 5	Befehl 4

- **Takt 1-5: Latenzzeit - nur beim Befüllen relevant**
- **Takt 6-8: Ein Befehl pro Takt!**
- Nach Befüllen **theoretisch** 100% Auslastung!

17 / 5

RAW-Hazard

Bsp Code

```
1 ADD R1,R1,R2 ; R1=R1+R2
2 SUB R3,R3,R1 ; R3=R3-R1
3 XOR R2,R2,R4 ; R2=R2 xor R4
```

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	SUB	ADD	×	×	×
3		SUB	ADD	×	×
4			SUB	ADD	×
5				SUB	ADD
6					SUB

SUB würde veraltete Daten verwenden!

Lösung über Leerbefehl

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	NOP	ADD	×	×	×
3	SUB	NOP	ADD	×	×
4		SUB	NOP	ADD	×
5			SUB	NOP	ADD
6				SUB	NOP
7					SUB

- Wenn SUB ausgeführt wird, liegt Ergebnis von ADD bereits vor
- Leertakte: Leistungseinbußen

Lösung über Compiler

- **Compiler versucht, Datenabhängigkeiten zu erkennen, ordnet Instruktionen falls möglich entsprechend um**
- **In diesem, und *nur* diesem konkreten Beispiel: XOR vor SUB, da XOR nicht datenabhängig ist!**
- **Keine Leistungseinbußen!**
- Zusätzlich auf CPU-Ebene: **Out of Order Execution**

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	ADD	×	×	×	×
2	XOR	ADD	×	×	×
3	SUB	XOR	ADD	×	×
4		SUB	XOR	ADD	×
5			SUB	XOR	ADD
6				SUB	XOR
7					SUB

Pipelining 2

Pipelining: Bedingte Sprungbefehle

Bedingter Sprungbefehl (`goto`, `jmp`, Schleifenabbruch)

- Eine Anweisung, die dazu dient, die Ausführung des Programms an anderer Stelle fortzusetzen

⇒ Programmteile können übersprungen werden

Probleme

- Sprungziel (Program Counter!) erst nach Auswertung (Ausführung) bekannt
- Nicht-sequenzielle Ordnung der Befehle

⇒ Je länger Pipeline, desto mehr zu verwerfen

Pipelines: Bedingte Sprungbefehle

Bsp: BREQ (Branch if equal) verzweigt zu Befehl 80. Befehle 2-5 zu verwerfen, Pipeline neu füllen (Dauer: Latenzzeit)

Takt	Instr-Fetch	Decode	Opnd-Fetch	Execute	Write
1	**BREQ**	×	×	×	×
2	Befehl 2	**BREQ**	×	×	×
3	Befehl 3	Befehl 2	**BREQ**	×	×
4	Befehl 4	Befehl 3	Befehl 2	**BREQ**	×
5	Befehl 5	Befehl 4	Befehl 3	Befehl 2	**BREQ**
6	Befehl 80	×	×	×	×
7	Befehl 81	Befehl 80	×	×	×
8	Befehl 82	Befehl 81	Befehl 80	×	×
9	Befehl 83	Befehl 82	Befehl 81	Befehl 80	×
10	Befehl 84	Befehl 83	Befehl 82	Befehl 81	Befehl 80

Sprungvorhersage und spekulative Ausführung

Branch Prediction

- Macht Annahme, ob Sprung genommen wird oder nicht
- Setzt dementsprechend Ausführung fort
- ⇒ Speculative Execution
- Wenn Annahme richtig: kein Zeitverlust
- ⇒ Idee: möglichst gute Annahmen treffen

Branch History Table

- Sprungbefehle im aktuellen Programmlauf beobachten und Statistik aufstellen
- Aufwändige Implementierungen erreichen Trefferquoten bis zu 99%

REP-BSP2

2. **Pipelining / Hazards.**

   Betrachten Sie die fünfstufige Pipeline auf VO-Folie 17. Identifizieren Sie in der folgenden Abfolge von Instruktionen mögliche Data Hazards. Wie könnten diese umgangen werden?

```
I0:  MUL F2,F2,F3  (Syntax: F2=F2*F3)
I1:  ADD F5,F4,F2
I2:  ADD F2,F2,F4
I3:  OR  F6,F6,F2
I4:  SUB F5,F3,F2
I5:  AND F7,F2,F5
```

0
NOP
1
2
NOP
4
3
5

Optimal

	Fetch	Decode	Operand Fetch	Ex	Write
1	MUL				
2	NOP	MUL			
3	ADD	NOP	MUL		
4	ADD	ADD	NOP	MUL	
5	NOP	ADD	ADD	NOP	MUL F2
6	SUB	NOP	ADD	ADD	NOP
7	OR	SUB	NOP	ADD	ADD F5
8	AND	OR	SUB	NOP	ADD F2
9		AND	OR	SUB	NOP
			AND	OR	SUB
				AND	OR
					AND

REP-BSP3

3. **Parallele Architekturen – Effizienz**

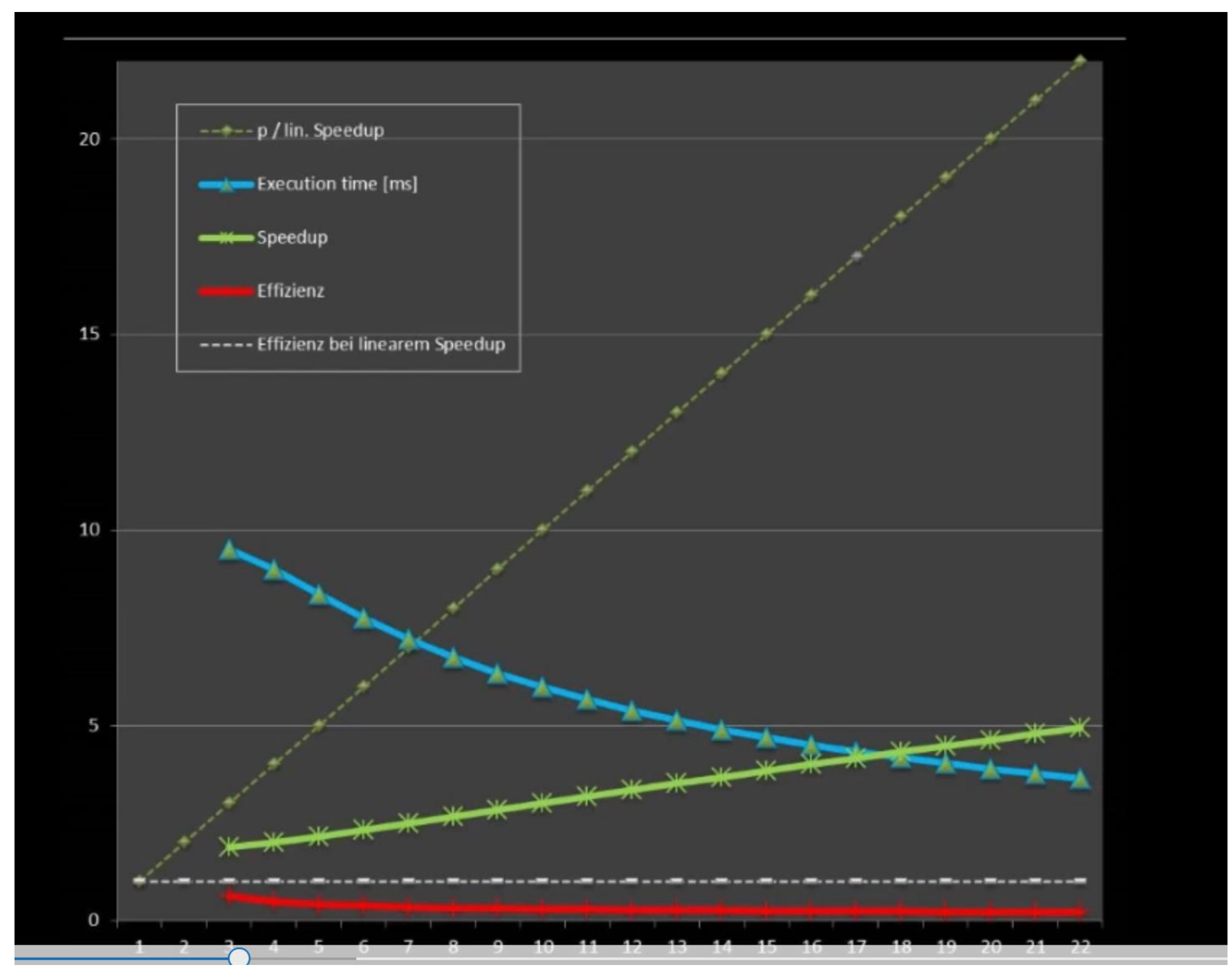
Ein Programm wird von einem Prozessor in 18 ms abgearbeitet. Das gleiche Programm kann auch auf p Prozessoren gleichzeitig ausgeführt werden. Die neue Execution Time ET_p beträgt dann:

$$ET_p = \frac{ET_1}{p} \times \log 2(p)$$

(a) Gesucht sind der Speedup S_p und die Effizienz E_p bei vier Prozessoren – plotten Sie hierfür (z.B. in Excel, Libre office calc, C++, Java, ...) eine Funktion für den Verlauf aller wichtigen Kennzahlen (p, ET, Speedup, Effizienz).

(b) Argumentieren Sie, warum das Programm nicht einfach p Mal so schnell ausgeführt werden kann!

Hinweis: Die Effizienz E_p bezieht sich auf das Verhältnis zwischen erreichtem Speedup und verwendeten Prozessoren.



Superskalare Architekturen

Superskalare Architekturen

Skalarität

⇒ Fähigkeit, in jedem Takt eine Instruktion ausführen

Superskalarität

⇒ Fähigkeit, Durchsätze von **mehr als einem Befehl** pro Takt zu erreichen. *"Instruction-level parallelism (ILP)"*

- Nur durch echte Parallelisierung erreichbar, die über Befehlspipelining hinausgeht
- Superskalare Architekturen besitzen mehrere Dekodierungsstufen und mehrere Ausführungseinheiten, die parallel arbeiten können

Ansätze

Zusätzlich (zu den parallelen Dekodierungsstufen und Ausführungseinheiten) verwendet man oft:

- **Out-of-Order-Execution**: Reihenfolge der Abarbeitung zwecks Beschleunigung verändern
- **Register Renaming**: Datenabhängigkeiten können durch Verwendung anderer Register (die keine Datenabhängigkeiten aufweisen) teilweise vermieden werden
- Verschiedene Pipeline Längen
- Mehrfache spekulative Ausführung
- **VLIW** (Very Long Instruction Word)
- Hyper-Threading

... bis hin zu mehreren Kernen/Prozessoren

Infos

Informationen zum Beispiel der nachfolgenden Folien:[a]

[a] Alle Details siehe Wüst, Mikroprozessortechnik, 4. Auflage

- Die folgende Architektur besitzt zwei Dekoder, zwei Ganzzahl-Einheiten, eine Gleitkomma-Einheit und eine Lade-/Speichereinheit.
- Die beiden Dekoder begrenzen die Leistung der Architektur auf maximal zwei Instruktionen pro Takt.
- Es stehen getrennte Ganzzahlregister R0-R31 und Gleitkommaregister F0-F15 zur Verfügung.
- Die beiden Dekoder dekodierten die Instruktionen und speichern sie in einem kleinen Pufferspeicher mit zwei Platzen.
- Die Befehle werden von dort an Ausführungseinheiten ausgegeben, sobald diese verfügbar sind.

Informationen zum Beispiel der nachfolgenden Folien:[a]

[a] Alle Details siehe Wüst, Mikroprozessortechnik, 4. Auflage

- Mögliche Datenabhängigkeit müssen berücksichtigt werden.
- Wenn im Pufferspeicher Plätze frei werden, werden im gleichen Takt neue Befehle dekodiert und im Pufferspeicher abgelegt.
- Wir nehmen an, dass durch Voreinholung (Prefetching) immer genug Code vorausschauend eingelesen und an die Dekoder weitergegeben wird.
- Die Instruktionen werden abgeschlossen durch die Rückordnungs- und Abschlusseinheit.
- Bei STORE-Befehlen werden durch diese Einheit die Ergebnisse aus den Registern entnommen und an die LOAD/STORE Einheit übergeben.

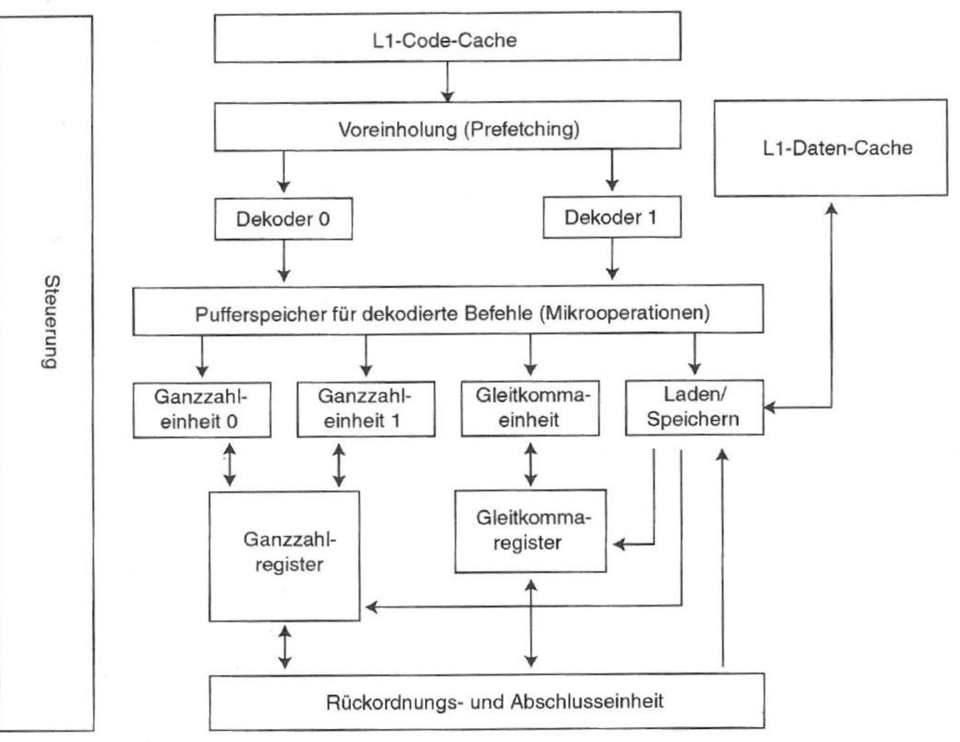


Abbildung 11.9: Beispiel für eine superskalare Architektur.

Informationen zum Beispiel der nachfolgenden Folien:[a]

[a] Alle Details siehe Wüst, Mikroprozessortechnik, 4. Auflage

- Ganzzahl-Befehle brauchen nach der Ausgabe an die Ganzzahl-Einheit drei Takte zur Ausführung .
- Gleitkomma-Befehle brauche nach der Ausgabe an die Gleitkomma-Einheit vier Takte zur Ausführung.
- Die Ausführungszeit der Lade- und Speicherbefehle hängt vom Inhalt des L1-Cache ab, soll aber mindestens vier Takte betragen (L1-Cache-Treffer).

⇒ Bitte beachten Sie, dass alle Angaben/Werte der letzten 3 Folien ...

... nur beispielhafte Angaben/Werte für diese spezielle exemplarische Architektur darstellen und nicht allgemein gültig sind.

Beispiel - strikt

```
;Beispiel-Befehlsfolge für die superskalare Architektur von Abb. 11.9
            AND R1,R2,R3      ; R1=R2 AND R3
            INC R0            ; Increment R0
            SUB R5,R4,R1      ; R5=R4-R1
            ADD R6,R5,R1      ; R6=R5+R1
            COPY R5, 1200h    ; R5=1200h
            LOAD R7,[R5]      ; Laden mit registerindirekter Adressierung
            FADD F0,F1,F2     ; F0=F1+F2
            FSUB F0,F0,F3     ; F0=F0-F3
```

Abbildung: R0-R31: Ganzzahlregister; F0-F15: Gleitkommaregister; Datenabhängigkeiten beachten

Beispiel der Abarbeitung in strikter Reihenfolge

Annahmen:

- Wir nehmen zunächst an, dass alle Instruktionen in richtiger Reihenfolge ausgegeben werden müssen (in-order-issue).
- Es ist also eine parallele Ausgabe aufeinanderfolgender Befehle erlaubt, nicht aber die Ausgabe eines Befehls, der im Code nach Befehlen steht, die noch nicht ausgegeben wurden.
- Außerdem wollen wir annehmen, dass es kein Bypassing gibt und bei Datenabhängigkeit abgewartet wird, bis das betreffende Register beschrieben ist.

Takt-zyklus	Dekoder0	Dekoder1	Ganzzahl-Einheit0	Ganzzahl-Einheit1	Gleit-komma-einheit	Load/Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3			AND	INC		
4			AND	INC		
5	COPY		SUB			
6			SUB			
7			SUB			
8	LOAD		ADD			
9			ADD			
10			ADD			
11	FADD		COPY			
12			COPY			
13			COPY			
14	FSUB	...			FADD	LOAD
15					FADD	LOAD
16					FADD	LOAD
17					FADD	LOAD
18	...				FSUB	
19					FSUB	
20					FSUB	
21					FSUB	

Abbildung 11.10: Ausführung der Beispiel-Befehlsfolge, wenn der Prozessor die Instruktionen in der richtigen Reihenfolge ausgibt. „...“ steht für einen nachfolgenden Befehl.

Abbildung: FADD(FloatADD) und FSUB sind unabhängig von den vorhergehenden Befehlen, werden aber aufgrund der Abarbeitung in strikter Reihenfolge erst spät ausgeführt

Out of Order

```
;Beispiel-Befehlsfolge für die superskalare Architektur von Abb. 11.9
            AND R1,R2,R3     ; R1=R2 AND R3
            INC R0           ; Increment R0
            SUB R5,R4,R1     ; R5=R4-R1
            ADD R6,R5,R1     ; R6=R5+R1
            COPY R5, 1200h   ; R5=1200h
            LOAD R7,[R5]     ; Laden mit registerindirekter Adressierung
            FADD F0,F1,F2    ; F0=F1+F2
            FSUB F0,F0,F3    ; F0=F0-F3
```

Abbildung: R0-R31: Ganzzahlregister; F0-F15: Gleitkommaregister; Datenabhängigkeiten beachten

Annahmen:

- Ausführung in geänderter Reihenfolge soll erlaubt werden.
- Damit sich dazu überhaupt die Möglichkeiten ergeben, vergrößern wir den Pufferspeicher, so dass er sechs dekodierte Befehle fasst (es gibt aber weiterhin nur 2 Dekoder).
- Datenabhängigkeiten müssen natürlich weiterhin beachtet werden.

Bsp. der Abarbeitung mit Out-of-Order Execution

Takt-zyklus	Dekoder0	Dekoder1	Ganzzahl-Einheit0	Ganzzahl-Einheit1	Gleit-komma-einheit	Load/ Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3	COPY	LOAD	AND	INC		
4	FADD	FSUB	AND	INC		
5	...	...	SUB		FADD	
6			SUB		FADD	
7			SUB		FADD	
8	...		ADD		FADD	
9	...		ADD		FSUB	
10			ADD		FSUB	
11	...		COPY		FSUB	
12			COPY		FSUB	
13			COPY			
14	...					LOAD
15						LOAD
16						LOAD
17						LOAD

Abbildung 11.11: Ausführung der Beispiel-Befehlsfolge, wenn der Prozessor die Instruktionen in veränderter Reihenfolge ausgibt.

Abbildung: Vier Takte schneller, aber erheblicher Hardwareaufwand

Rest

Abarbeitung mit OoO Exec. + Register Renaming

Register Renaming:

- Der COPY-Befehl lädt die Adresse 1200h nach R5, damit der nachfolgende LOAD-Befehl mit R5 adressieren kann.
- Genau so gut hätte in diesen beiden Befehlen jedes andere Register benutzt werden können.
- Wahl von R5 war unglücklich ⇒ produziert unnötige Datenabhängigkeit zu vorausgehendem ADD-Befehl.

⇒ Ein optimierter Complier hätte eine andere Wahl getroffen.

⇒ *Register Renaming*: Die im Code referenzierten ISA-Register werden über eine Zuordnungstabelle auf eine größere Zahl von Hintergrundregistern abgebildet ⇒ mehr Möglichkeiten bei der Registerwahl

Abarbeitung mit OoO Exec. + Register Renaming

Takt-zyklus	Dekoder0	Dekoder1	Ganzzahl-Einheit0	Ganzzahl-Einheit1	Gleit-komma-einheit	Load/ Store
1	AND	INC				
2	SUB	ADD	AND	INC		
3	COPY	LOAD	AND	INC		
4	FADD	FSUB	AND	INC		
5	...	...	SUB	COPY	FADD	
6	...		SUB	COPY	FADD	
7			SUB	COPY	FADD	
8	...	...	ADD		FADD	LOAD
9	...		ADD		FSUB	LOAD
10			ADD		FSUB	LOAD
11					FSUB	LOAD
12					FSUB	

Abbildung 11.12: Ausführung der Beispiel-Befehlsfolge, bei veränderter Reihenfolge und Benutzung von Register-Umbenennung.

Abbildung: Verwendung eines anderen Registers als R5 im COPY und im folgenden LOAD Befehl (siehe Beispiel-Befehlsfolge) bringt weitere fünf Takte Performance-Gewinn, da Datenabhängigkeiten vermieden werden können

Verschiedene Pipeline-Längen und mehrfache spekulative Ausführung

Super-pipelined Prozessoren ⇒ mehr als 10 Pipeline-Stufen

- Höherer Durchsatz, da höherer Prozessortakt möglich
- Problem 1: Zugriffe auf Speicherbausteine lassen sich nicht weiter unterteilen
- Problem 2: Latenzzeit wächst mit Pipeline-Länge

Spekulative Ausführung bei superskalaren Architekturen

- Weiterhin ein Problem (evtl. höhere Latenzzeit!)
- Bei ausreichend Systemressourcen wird spekulative Ausführung in mehreren Zweigen durchgeführt
- Nach der Sprungentscheidung wird ein Zweig, der nicht gebraucht, wird ungültig gemacht

VLIW-Prozessoren

Very Long Instruction Word

- VLIW enthält mehrere Befehle, die zur Übersetzungszeit gebündelt wurden
- Schon bei Übersetzung wird der Code analysiert und unter Kenntnis der Hardware entschieden, welche Maschinenbefehle parallel ausgeführt werden können
- Übersetztes Programm enthält dann schon die Anweisungen zur Parallelausführung (muss nicht mehr vom Prozessor übernommen werden)

⇒ “explicit parallelism” (control over the parallel execution)

⇒ Gegenstück “implicit parallelism” (*automatic* parallel execution)

Ausblick

Hyper-Threading (Ausblick VO “Betriebssysteme”)

Thread (“Faden”): eigenständiger Teil eines Prozesses

- Hat seinen eigenen Programmfluss samt Programmzähler, Registerinhalten, und Stack
- Aber: Alle Threads eines Prozesses teilen sich denselben Adressraum

***Hyper*-Threading: schnelleres Umschalten zwischen 2 Threads**

- Hintergrundregister und Registerzuordnungstabelle doppelt vorhanden
- Zweiter Registersatz enthält Daten des zweiten Threads

$\Rightarrow$ Umschalten zwischen zwei Threads geht damit schneller

Multicore, Multicpu

Taktfrequenz der Prozessoren ist limitiert

Max. Taktfrequenz liegt seit Jahren bei ca. 3.5 GHz und kann kaum noch gesteigert werden

- Gründe: Wärmeentwicklung, Abstrahlung, ...

– Weitere Verbreiterung der Register bring keine große Leistungssteigerung – schon 64-Bit-Reg. kaum ausgenutzt

– Ebenso bringen mehrere parallele Ausführungseinheiten keine weiteren Vorteile, da zu viele Datenabhängigkeiten

Trend zu Parallelrechnern

- Prozessoren mit mehreren Kernen, oder
- Rechner mit mehreren Prozessoren im System
- Damit auch *“task-parallelism”* möglich

Mehrprozessorsystem vs. Mehrkernsystem

Mehrprozessorsystem

- Für ein echtes Mehrprozessorsystem braucht man
  - mehrere Sockel (“Steckplatz” für einen Prozessor) und
  - einen dementsprechenden Chipsatz (mehrere zusammengehörende integrierte Schaltkreise)

  ⇒ Aufwändig und teuer

Mehrkernsystem – SMP

- Günstiger ist es, mehrere Prozessorkerne in einem Chip unterzubringen
- Ein Chip ist ein integrierter Schaltkreis auf einem einzigen Halbleiterplättchen
- Bei mehreren gleichartigen Kernen spricht man von “Symmetrischen Multiprocessing” SMP

Mehrkernprozessor

Mehrkernprozessoren (Multi-Core)

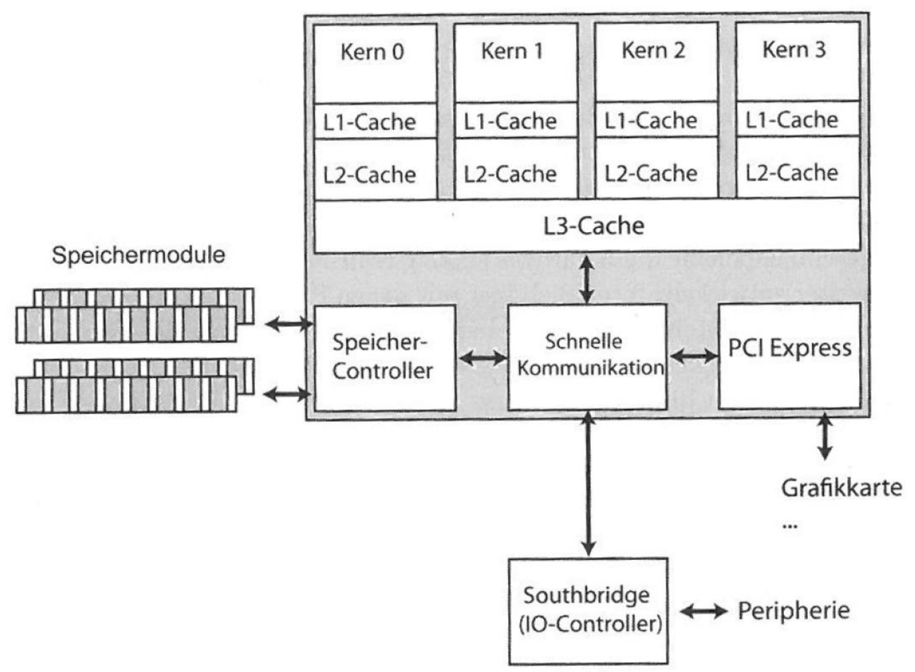


Abbildung 11.16: Vierkernprozessor mit integrierter PCI-Express-Schnittstelle und Southbridge

K. Wüst, Mikroprozessortechnik, 4. Auflage

Mehrkernprozessoren (Multi-Core)

Cache-Designfragen und Cache-Organisation

- Gibt es private Caches für die Kerne?
  - Bei einem gemeinsamen Cache wäre Datentransfer sehr groß und langsam
  - Besser: jeder Prozessor(-kern) hat seinen eigenen Cache
- Wie werden die Caches konsistent gehalten?
  - Problem: In zwei Caches von verschiedenen Kernen liegt eine Kopie des gleichen Speicherblocks, eine davon wird durch Schreibzugriff verändert
  - “Cache coherency”, “bus snooping”

Shared Memory / Uniform Memory Access (UMA)

- Prozessoren der Multi-Prozessor-Hauptplatine können auf den vorhandenen Arbeitsspeicher “gleichwertig” zugreifen

REP-BSP4

4. **Multicore Rechner.**

Zwei Programme sind zu unterschiedlichen Teilen parallelisierbar. Programm 1 ist zu 20% parallelisierbar, Programm 2 zu 90%

Das Programm soll auf zwei verschiedenen Multi-Core PCs mit jeweils 2 bzw. 16 Cores ausgeführt werden. Die zwei schlauesten TGI Studierenden bekommen außerdem eine Reise ins ferne Wuxi, China, geschenkt, um dort das Programm auf einem der schnellsten Supercomputer der Welt, dem Sunway TaihuLight mit 10 649 600 (>10 Mio) Cores, berechnen zu lassen. [1]

Berechnen Sie den theoretisch möglichen Speedup für beide Programme auf den drei verschiedenen Rechnern mittels F_E und S_E bzw. mittels der Formel $S = \frac{n}{1+(n-1)f}$

R1 20 %
R2 90 %

1:0,55 = 1,8   1:0,9 = 1,11
450   10
50   10

P1 2: $\frac{1}{0{,}8 + 0{,}1} = \frac{1}{0{,}9} = 1{,}11$ ca

P2 2: $\frac{1}{0{,}1 + 0{,}45} = \frac{1}{0{,}55} = 1{,}8$ ca

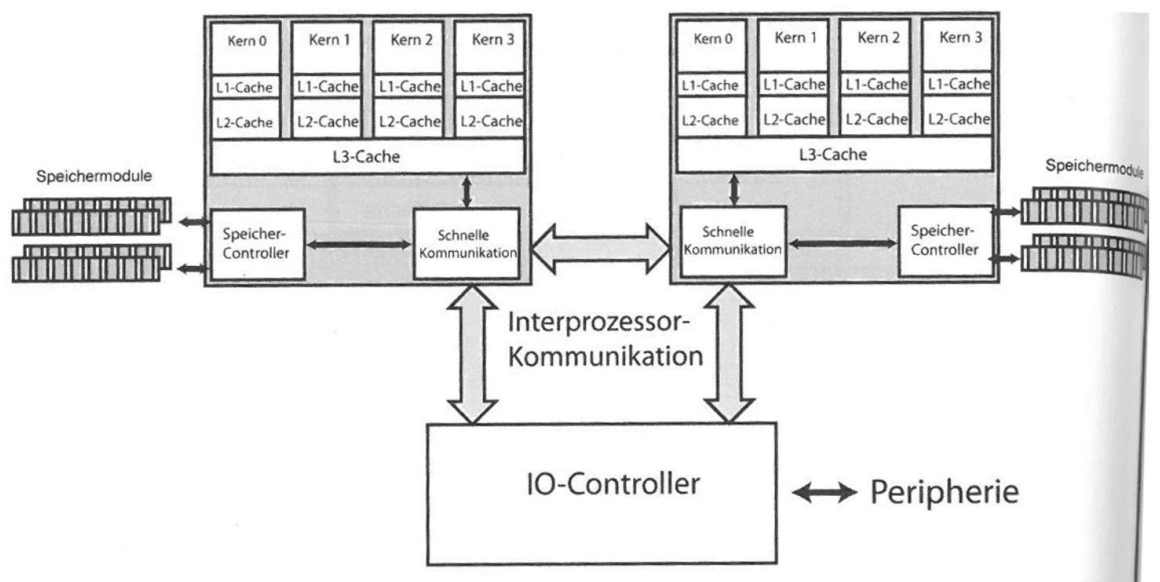
P1 16: $\frac{1}{0{,}8 + \frac{0{,}2}{16}} = 1{,}23...$

P2 16: $\frac{1}{0{,}1 + \frac{0{,}9}{16}} =$

$$S = \frac{1}{1-F_E + \frac{F_E}{S_E}} = \frac{1}{f + \frac{1-f}{n}} = \boxed{\frac{n}{n-1 \cdot f + 1}}$$

Mehrprozessoren

Mehrprozessorsystem (Multi-CPU)



K. Wüst, Mikroprozessortechnik, 4. Auflage

Mehrprozessorsystem (Multi-CPU)

Inter-Prozessor-Kommunikation (IPC/IPK)

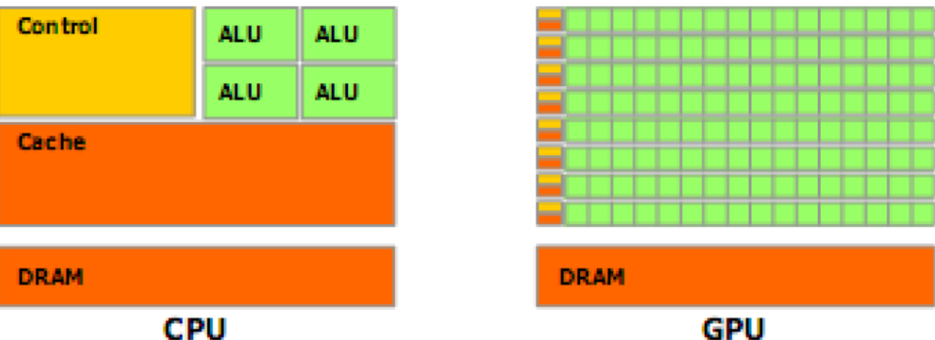
- Leistungsstarke Schnittstelle um untereinander (zwischen Prozessoren) schnell Daten auszutauschen
- zB: *Quick Path Interconnect* (Intel); *Hypertransport* (AMD)

Distributed (Shared) Memory / Non-Uniform Memory Access (NUMA)

- Jeder Prozessor besitzt eigenen Arbeitsspeicher
- Zugriff auf “fremde” Arbeitsspeicher nur mittels Arbeitsspeicher-Verwalter (Bsp. MESI-/MOESI-Protokoll)

GPU

Manycore Prozessor



Relative Allocation of Transitors for Computation, Data Caching, and Flow Control - NVIDIA CUDA C Programming Guide - Version 4.2 - 4/5/2012 - copyright NVIDIA Corporation 2012

- Große Anzahl an einfachen, unabhängigen Kernen
- Beispiel: nvidia GPGPU (General Purpose Computation on Graphics Processing Unit)
- Programmier-APIs: OpenCL, CUDA, C++ AMP
- Typisch: *"Data-parallel"* job execution

Speedup

Speedup durch Parallelisierung

Amdahl's Law

- Nur gewisser Anteil des Codes kann parallel ausgeführt werden (Vergleich F_E von Kapitel Computerdesign)
- Der Rest muss sequenziell abgearbeitet werden
  - Mögliche Gründe: Vorbereitung der Parallelausführung, Erfassen der Ergebnisse, Datenabhängigkeiten, nicht parallelisierbarer Code, IPC, ...
- Nur der **parallele** Teil kann statt auf einem Prozessor auf n Prozessoren (Vergleich S_E) ausgeführt werden
- Sequentieller Anteil f, parallelisierbarer Anteil $(1-f)$
- *Theoretisch* erreichbarer Speedup: $S = \frac{n}{1+(n-1)f}$
- Erfolgt hängt u.a. von Caches und Speicherlatenzen ab

Speedup durch Parallelisierung

Beispiel zu Amdahl's Law bei System mit 4 Prozessoren (n=4)

- Achtung! Tippfehler in K. Wüst, Mikroprozessortechnik, 4. Aufl. p218. Wenn f gleich 0.1 ist, dann muss der parallelisierbare Anteil 0.9 sein

Bsp. 1: Speedup bei einem parallelisierbaren Anteil von 90%

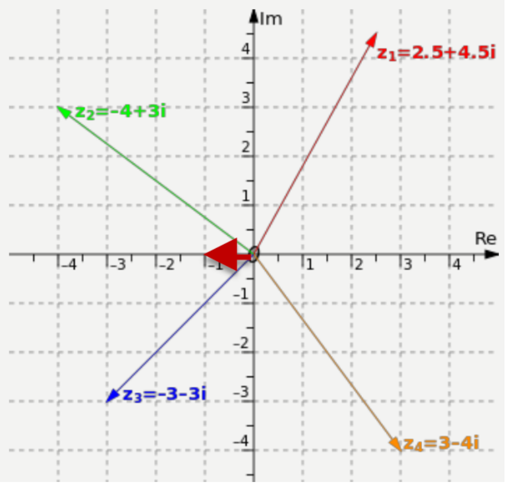
- $(1-f) = 0.9 \Rightarrow f = 0.1 \Rightarrow S = \frac{4}{1+(4-1)0.1} \Rightarrow S = 3.077$
- Alternative Berechnung mittels F_E und S_E: $S = \frac{1}{(1-F_E)+F_E/S_E} = \frac{1}{(1-0.9)+0.9/4} = \frac{1}{(0.1+0.225)} = 3.077$

Bsp. 2: Speedup bei einem parallelisierbaren Anteil von 10%

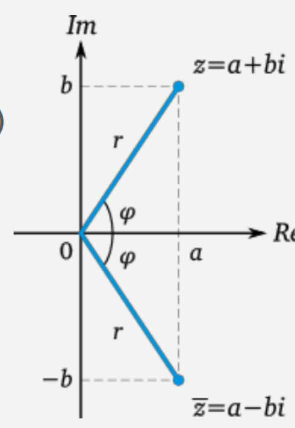
- $(1-f) = 0.1 \Rightarrow f = 0.9 \Rightarrow S = \frac{4}{1+(4-1)0.9} \Rightarrow S = 1.08$

Mathematische Grundlagen

- Komplexe Zahlen
  - $i * i = i^2 = -1$
  - $i = 0 + 1 * i$ ← i ist Rotation 90° um Ursprung
  - $i * i \triangleq zwei\ Rotationen\ um\ Urpsrung$ = -1



- Addition = Komponentenweise (3+2i)+(1+i) = (4+3i)
- Multiplikation Variante 1
  - Als Polarkoordinaten anschreiben → Länge des Vektors (r) und Winkel (α)
  - Multipliziere Länge und **ADDIERE** Winkel
  - (3+2i)*(1+i)=1+5i
- Multiplikation Variante 2
  - Ausmultiplizieren
  - 3*1+2i*1+3*I+2i*i= 3+2i+3i-2 = 1+5i



Mathematische Grundlagen 2

- Tensorprodukt

$$x \otimes y = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_1 y_2 \\ x_2 y_1 \\ x_2 y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Tensorprodukt ist linear

- Komplexe Konjugation: 3+4i → 3-4i
- Inverse Matrix: Matrix M mit inversen Matrix von M multipliziert ergibt Einheitsmatrix
  - Reelle Matrix nicht symmetrisch → transponieren entlang Hauptachse spiegeln
- Unitäre Matrix: Wenn Inverse von M gleich M ist nennt man das unitäre Matrix

Quantencomputing

- Zwei Zustände $|0\rangle$ und $|1\rangle$
- Superposition: $|x\rangle = \alpha|0\rangle + \beta|1\rangle$
  - $|\alpha|^2$ und $|\beta|^2$ $Wahrscheinlichkeiten\ für\ den\ Zustand$
  - Durch Messung kollabiert Superposition zu bestimmten Zustand
  - Variante Schreibeweise: $|x\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \alpha|0\rangle + \beta|1\rangle$
- Prüfung ob Bloch Kugel: $|\alpha|^2 + |\beta|^2 = 1$

Quantengatter:

- $X = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$ ← „Invertiert" Spiegelung X-Achse
- $Z = \begin{pmatrix}1 & 0\\0 & -1\end{pmatrix}$ ← Spiegelung Z-Achse
- $H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$ ← Hadarmar, erzeugt Superposition mit CNOT Verschränkung

- Quantenregister

$|\psi\rangle = |\phi_1\rangle|\phi_2\rangle = |\phi_1\phi_2\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$ ← Tensorprodukt =

$=\left[\alpha_1\begin{pmatrix}1\\0\end{pmatrix} + \beta_1\begin{pmatrix}0\\1\end{pmatrix}\right] \otimes \left[\alpha_2\begin{pmatrix}1\\0\end{pmatrix} + \beta_2\begin{pmatrix}0\\1\end{pmatrix}\right]=$

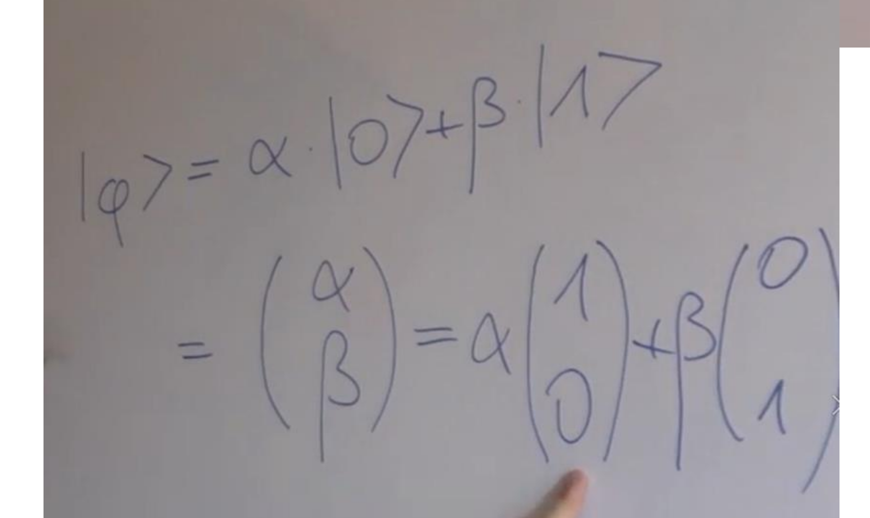
$=\alpha_1\alpha_2\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \alpha_1\beta_2\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \alpha_2\ \beta_1\begin{pmatrix}0\\0\\1\\0\end{pmatrix} + \beta_1\beta_2\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$ ← neu anschreiben

$=\alpha_{00}\ |00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$

$\alpha_1\alpha_2 = \alpha_{00}$
$\alpha_2\beta_1 = \alpha_{10}\ \dots$
$\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = |0\rangle \otimes |0\rangle = |00\rangle$

Quantencomputer

- Superposition = Elektron ist an zwei orten bis man misst (Schrödinger Katze Überlagerung von zwei zuständen)
- Kopenhagener Interpretation = Zustandswahrscheinlichkeit
- No cloning → man darf quantenbits nicht kopieren
- Notation für 0 und 1 ---→
- Reversibel → jede Operation umkehrbar
- Alpha und beta komplexe zahlen



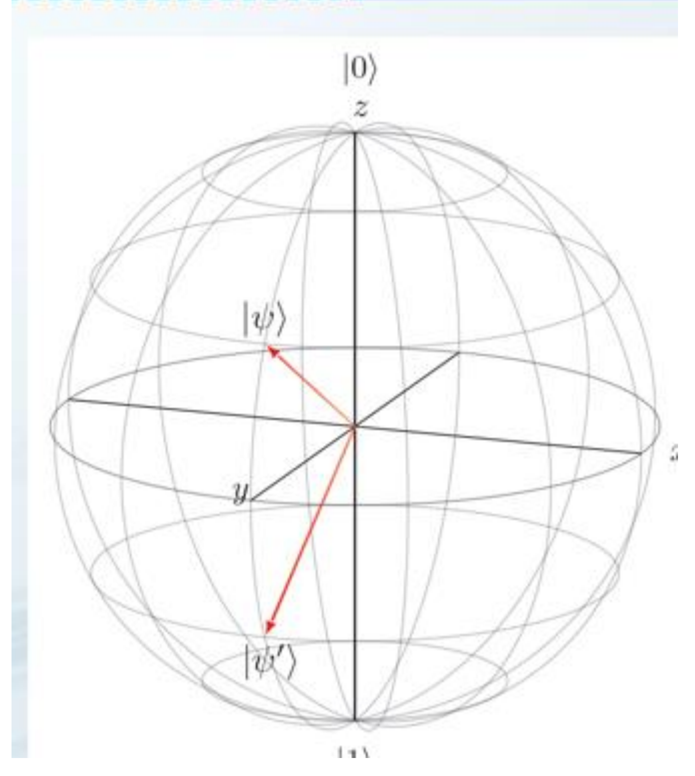
- Blochkugel → 3 dimensionen
- Wahrscheinlichkeit in Kugel drehen → schiebt alle wahrscheinlichkeiten → jede drehung lässt sich rückgängig machen
- Keine hidden variable → erst beim messen wird wirklich der Wert festgelegt

X Gatter → tauscht wahrscheinlichkeit → dreht kugel 180 grad

Pauli Matrizen

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- Die 3 Gatter drehen jeweils laut namen die Kugel
- x dreht auf x Achse (tauscht oben und unten)
- Y dreht kugel (dreht um y achse)
- Z dreht links nach rechts (vorne hinten)



- Messen = kein Gatter → nicht irrevasibel

- Hamada Gatter = 0/1 wird zu 50/50 → Überlagerung

- Quantenregister

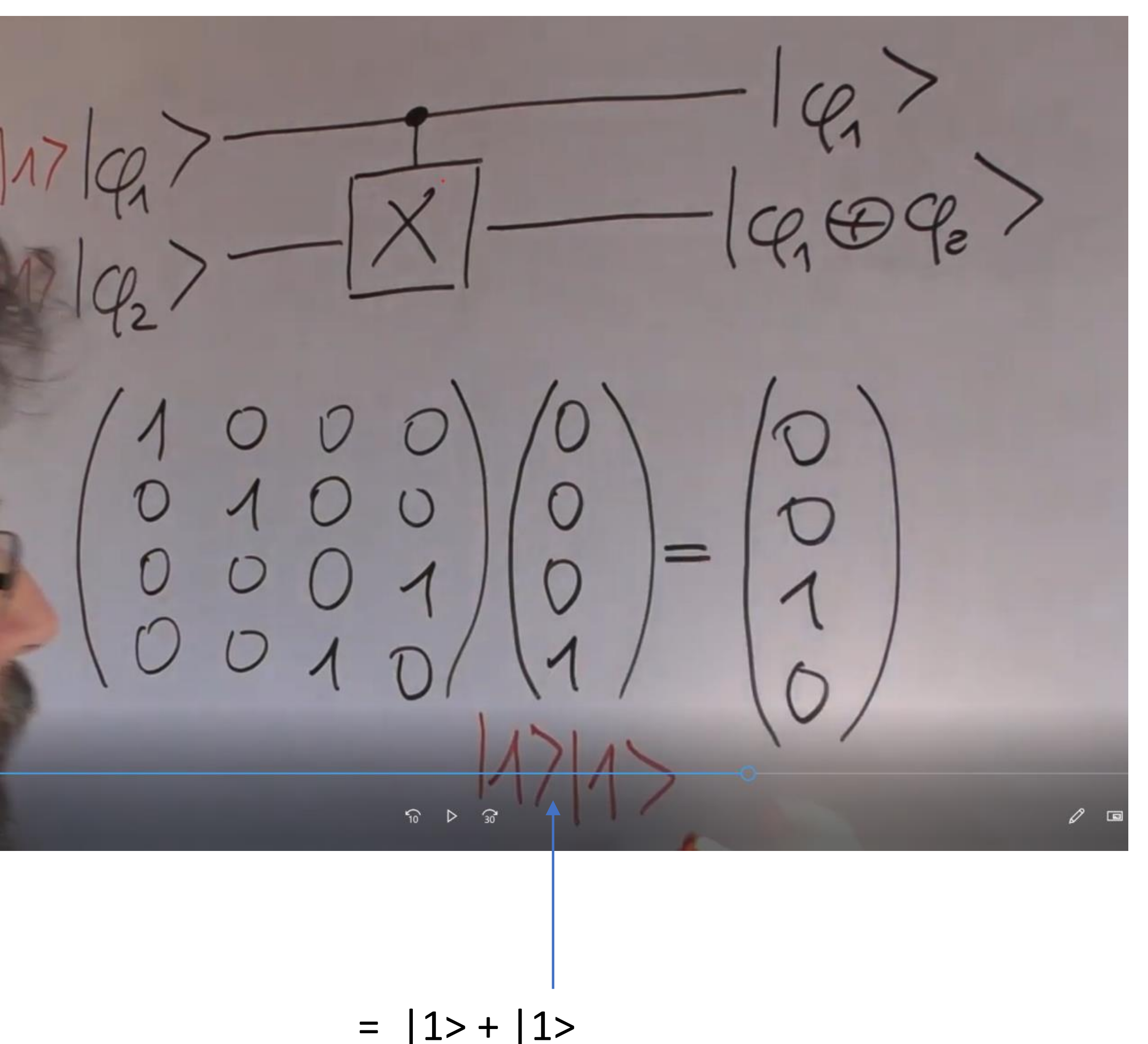
$$|\psi\rangle = |\varphi_1\rangle|\varphi_2\rangle = |\varphi_1\varphi_2\rangle = |\varphi_1\rangle \otimes |\varphi_2\rangle$$

$$= \left[\alpha_1\begin{pmatrix}1\\0\end{pmatrix} + \beta_1\begin{pmatrix}0\\1\end{pmatrix}\right] \otimes \left[\alpha_2\begin{pmatrix}1\\0\end{pmatrix} + \beta_2\begin{pmatrix}0\\1\end{pmatrix}\right]$$

$$= \underbrace{\alpha_1\alpha_2}_{\alpha_{00}}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \underbrace{\alpha_1\beta_2}_{\alpha_{01}}\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \underbrace{\alpha_2\beta_1}_{\alpha_{10}}\begin{pmatrix}0\\0\\1\\0\end{pmatrix} + \underbrace{\beta_1\beta_2}_{\alpha_{11}}\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

$$= \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

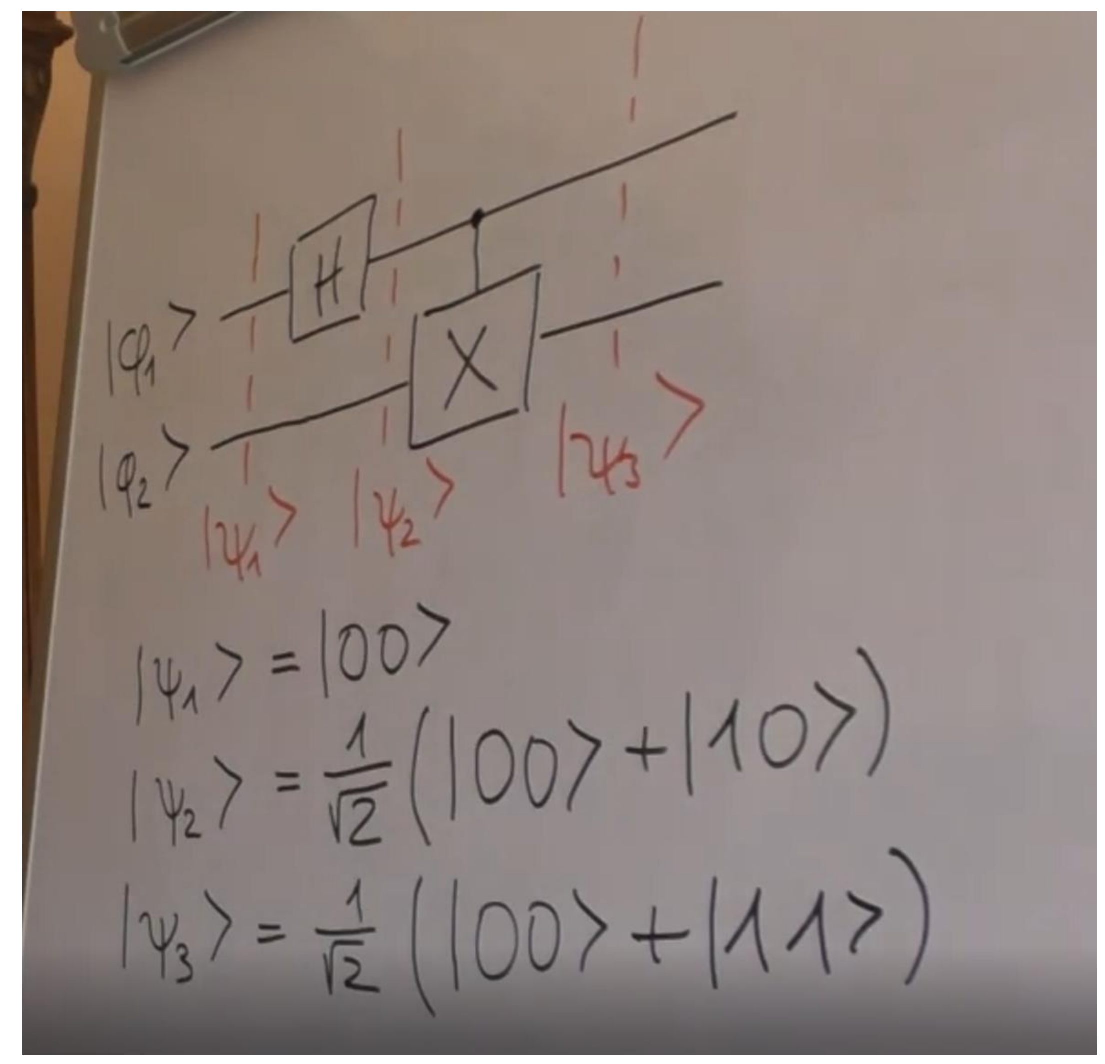
CNOT Gatter (XOR-Gatter)



= |1> + |1>

- 00 wird zu 00 weil CNOT nur schaltet wenn oben 1
- 1 oben lässt schaltung zu → invertiert unten

- → oben muss gleich unten sein
- → kannst oben jemanden geben
- Selber messen muss der ander selbes Ergebnis machen



REP-BSP1

1. **Quantenbits**

Gegeben sei ein Quantenbit im Überlagerungszustand $|\varphi\rangle = \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle$.

(a) Zeigen Sie, dass dieser Überlagerungszustand auf der Blochkugel liegt.

(b) Mit welcher Wahrscheinlichkeit erhalten Sie $|1\rangle$ als Ergebnis einer Messung von $|\varphi\rangle$?

(c) In welchem Zustand $|\varphi'\rangle$ befindet sich das Quantenbit nach Anwendung einer NOT-Operation? Stellen Sie hierzu $|\varphi\rangle$ als zweidimensionalen Zustandsvektor dar und wenden Sie darauf die zugehörige Paulimatrix X an.

(d) Mit welcher Wahrscheinlichkeit erhalten Sie $|1\rangle$ als Ergebnis einer Messung von $|\varphi'\rangle$?

a) $\left|\frac{1}{2}\right|^2 + \left|\frac{\sqrt{3}}{2}\right|^2 = \frac{1}{4} + \frac{3}{4} = 1$

b) $\Rightarrow$ 75%

c) X Gatter $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

$\Rightarrow \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$

d = 25%

REP-BSP2

2. **Quantengatter**

Betrachten Sie das Qubit $|\varphi\rangle$ aus der vorigen Aufgabe und folgende Schaltung:

$H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

$|\varphi\rangle$ — H — X — H — [Messung]   ($|\varphi_1\rangle$, $|\varphi_2\rangle$, $|\varphi_3\rangle$)

(a) Berechnen Sie die Zwischenzustände $|\varphi_1\rangle$, $|\varphi_2\rangle$, $|\varphi_3\rangle$.

(b) Lässt sich die aus drei Quantengattern bestehende Schaltung durch eine einzige 2×2-Matrix ausdrücken? Wenn ja, wie lässt sich diese Matrix berechnen und wie lautet sie, wenn nein, warum ist das nicht möglich?

(c) Was ergibt die abschließende Messung?

$\alpha \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \beta \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

I) $\frac{1}{\sqrt{2}} \cdot \alpha \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \alpha \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix}$

$\frac{1}{\sqrt{2}} \cdot \beta \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \beta \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

$\frac{1}{\sqrt{2}} \alpha \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \frac{1}{\sqrt{2}} \cdot \beta \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

II) $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

$\frac{1}{\sqrt{2}} \alpha \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} + \frac{1}{\sqrt{2}} \cdot \beta \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

III) $\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -2 \end{pmatrix}$

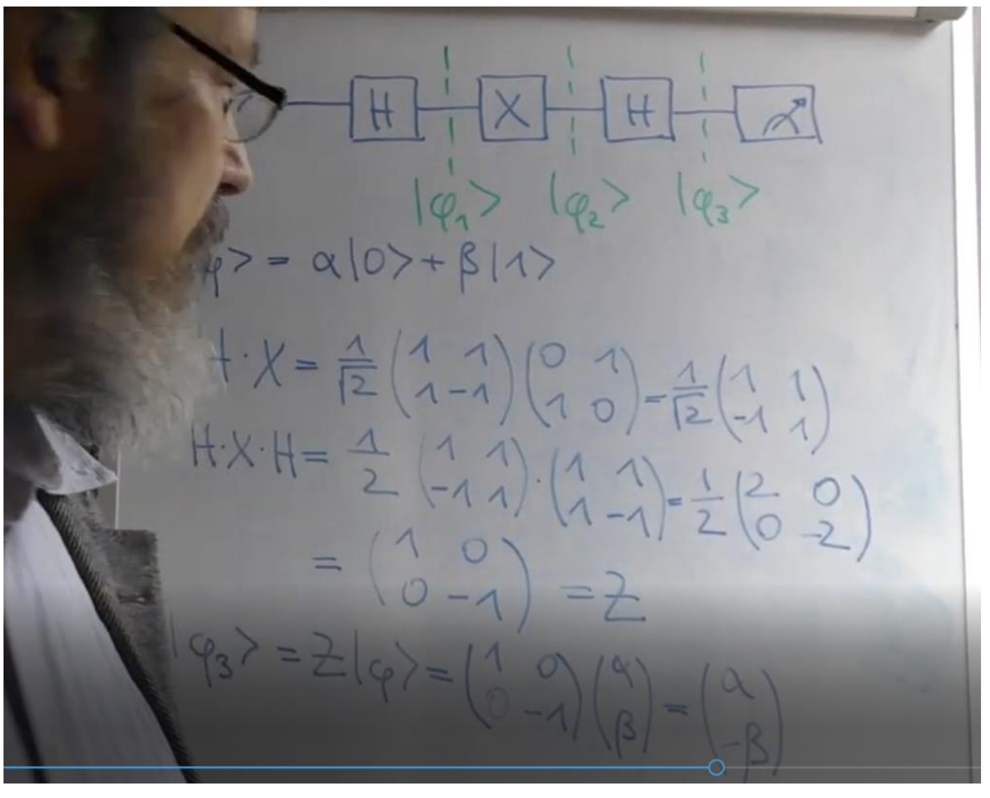
$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$

$\frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} = 0{,}5$

$0{,}5 \cdot \alpha \cdot 2 \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix} + 0{,}5 \cdot \beta \cdot 2 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

$= \alpha \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix} + \beta \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

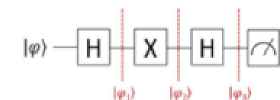
$\Rightarrow = \alpha \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \beta \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$



Version 2

2. Quantengatter

Betrachten Sie das Qubit $|\varphi\rangle$ aus der vorigen Aufgabe und folgende Schaltung:



(a) Berechnen Sie die Zwischenzustände $|\varphi_1\rangle$, $|\varphi_2\rangle$, $|\varphi_3\rangle$.

(b) Lässt sich die aus drei Quantengattern bestehende Schaltung durch eine einzige 2×2-Matrix ausdrücken? Wenn ja, wie lässt sich diese Matrix berechnen und wie lautet sie, wenn nein, warum ist das nicht möglich?

(c) Was ergibt die abschließende Messung?

a) Input $= \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle \;/\; \frac{\sqrt{3}}{2}\begin{pmatrix}0\\1\end{pmatrix} + \frac{1}{2}\cdot\begin{pmatrix}1\\0\end{pmatrix}$

beides überlegen

$\varphi_1 = \frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\frac{\sqrt{3}}{2}\cdot\begin{pmatrix}0\\1\end{pmatrix} = \frac{\sqrt{3}}{2}\cdot\frac{1}{2}\cdot\begin{pmatrix}1\\-1\end{pmatrix} = \frac{\sqrt{3}}{2\cdot\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}$

$\frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\frac{1}{2}\cdot\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\cdot\frac{1}{\sqrt{2}}\cdot\begin{pmatrix}1\\1\end{pmatrix} = \frac{1}{\sqrt{2}\cdot 2}\begin{pmatrix}1\\1\end{pmatrix}$

$\frac{\sqrt{3}}{2\cdot\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} + \frac{1}{\sqrt{2}\cdot 2}\cdot\begin{pmatrix}1\\1\end{pmatrix}$

$\varphi_2 : \frac{\sqrt{3}}{2\cdot\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} + \frac{1}{\sqrt{2}\cdot 2}\cdot\begin{pmatrix}1\\1\end{pmatrix}$

$\begin{pmatrix}0&1\\1&0\end{pmatrix}\cdot\begin{pmatrix}1\\-1\end{pmatrix} = \begin{pmatrix}-1\\1\end{pmatrix} \;\Big|\; \begin{pmatrix}0&1\\1&0\end{pmatrix}\cdot\begin{pmatrix}1\\1\end{pmatrix} = \begin{pmatrix}1\\1\end{pmatrix}$

$\varphi_2 = \frac{\sqrt{3}}{2\cdot\sqrt{2}}\cdot\begin{pmatrix}-1\\1\end{pmatrix} + \frac{1}{\sqrt{2}\cdot 2}\cdot\begin{pmatrix}1\\1\end{pmatrix}$

$\varphi_3 \sim \begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\begin{pmatrix}-1\\1\end{pmatrix} = \begin{pmatrix}0\\-2\end{pmatrix}$

$\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\begin{pmatrix}1\\1\end{pmatrix} = \begin{pmatrix}2\\0\end{pmatrix}$

$\varphi_3 = \frac{\sqrt{3}}{\sqrt{2}\cdot\sqrt{2}\cdot 2}\cdot -2\begin{pmatrix}0\\1\end{pmatrix} = -\frac{\sqrt{3}}{2}\cdot\begin{pmatrix}0\\1\end{pmatrix}$

$\frac{1}{\sqrt{2}\cdot 2\cdot\sqrt{2}}\cdot 2\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{2}\cdot\begin{pmatrix}1\\0\end{pmatrix}$

$= -\frac{\sqrt{3}}{2}\cdot\begin{pmatrix}0\\1\end{pmatrix} + \frac{1}{2}\cdot\begin{pmatrix}1\\0\end{pmatrix}$

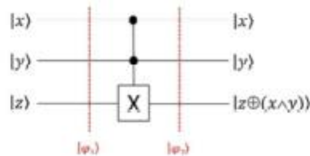
b) in dem Fall wäre die 2x2 Matrix die einheitsmatrix $\begin{pmatrix}1&0\\0&1\end{pmatrix}$ da sich nichts verändert (Ich glaube mal das - vor der wahrscheinlichkeit) ändert nix am (absoluten) wert der wahrscheinlichkeit
Alternativ $\begin{pmatrix}1&0\\0&-1\end{pmatrix}$ um Ergebnis mit - wahrscheinlichkeit zu erhalten

c) zu 75 % 1 und zu 25 % 0

REP-BSP3

3. Toffoli-Gatter

Das sogenannte Toffoli-Gatter besteht aus drei Quantenbits $|x\rangle$, $|y\rangle$, $|z\rangle$. Hierbei kontrollieren die beiden Qubits $|x\rangle$ und $|y\rangle$ das dritte Qubit $|z\rangle$: wenn $|x\rangle$ und $|y\rangle$ *beide* im Zustand $|1\rangle$ sind, dann wird $|z\rangle$ durch Anwendung der X-Matrix invertiert, ansonsten bleibt $|z\rangle$ unverändert.



(a) Das Toffoli-Gatter wird durch die 8×8-Matrix T ausgedrückt, die Sie aus Aufgabe 0 kennen. Zeigen Sie mit Hilfe dieser Matrix, dass das Toffoli-Gatter reversibel ist!

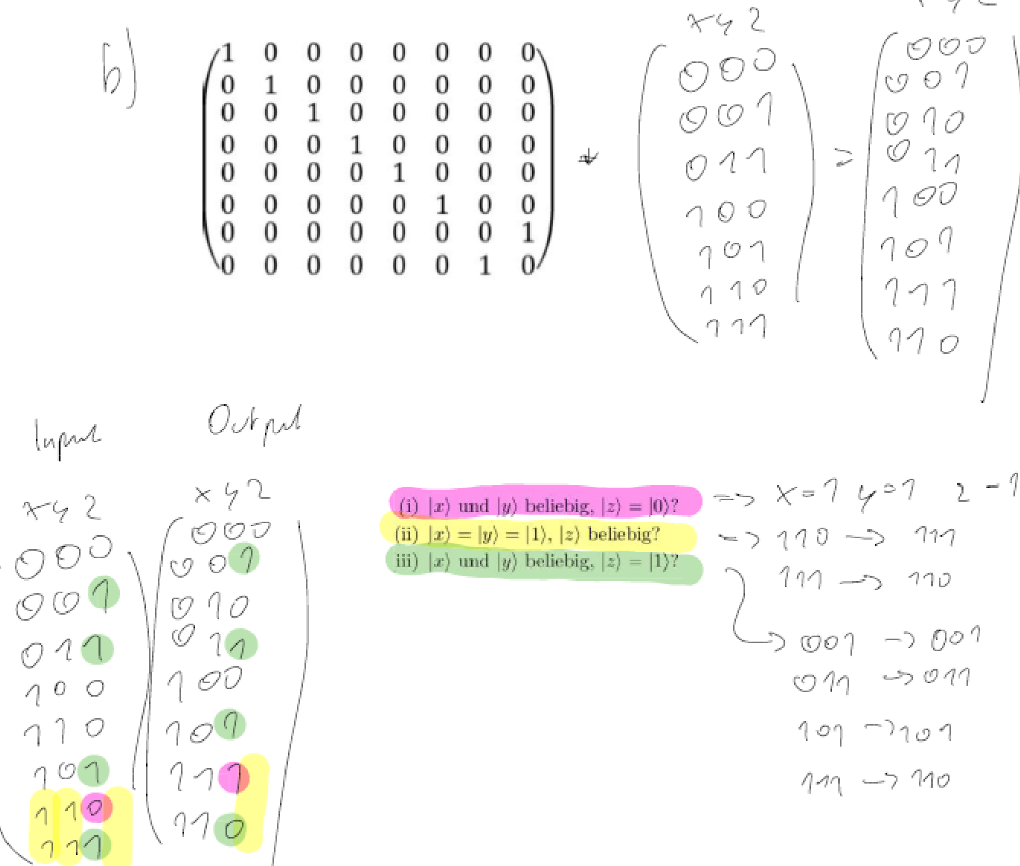
(b) Was erhält man am untersten Ausgang des Toffoli-Gatters für die nachfolgenden Eingangsbelegungen.

(i) $|x\rangle$ und $|y\rangle$ beliebig, $|z\rangle = |0\rangle$?

(ii) $|x\rangle = |y\rangle = |1\rangle$, $|z\rangle$ beliebig?

(iii) $|x\rangle$ und $|y\rangle$ beliebig, $|z\rangle = |1\rangle$?

(c) Was bedeutet das für den Zusammenhang zwischen traditionellen Digitalrechnern und Quantencomputern?



c)

Es ist ein gutes Beispiel dafür das man komplizierte normale Schaltungen auch einfach an Quantencomputern darstellen kann. und dies nur anhand einer Matrix

Allgemein

Die Grundidee

- Herkömmliche Rechner: Universal Turing Machine
  - abstraktes Konzept + materiale Realisierung
  - Moore's Law und seine Grenzen
- Feynmann's Idee (1982): Rechnen wie die Natur funktioniert
  - Quantenmechanik
- Die „Waffen" der Quantenmechanik
  - Superposition
  - Verschränkung
  - Interferenz
- Besonderheiten
  - umkehrbare Berechnungen
  - No-cloning Theorem

Quantenbits

- Bisher: Bit als fundamentales Konzept
  → jetzt: Quantenbit (Qubit)
  - mathematisches Konzept
  - physikalische Realisierung
- Idee: zwei Zustände $|0\rangle$ und $|1\rangle$
- Unterschied zu klassischen Bits: Überlagerung (Superposition) ergibt weitere Zustände
  → $|x\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ (mit α, β komplexe Zahlen = „Amplituden" und $|\alpha|^2 + |\beta|^2 = 1$)
  - Vektor in zweidimensionalem komplexen Vektorraum
  - $\{|0\rangle, |1\rangle\}$ = orthonormale Basis dieses Vektorraums, damit:

$$|x\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \alpha\begin{pmatrix}1\\ 0\end{pmatrix} + \beta\begin{pmatrix}0\\ 1\end{pmatrix} = \alpha\,|0\rangle + \beta\,|1\rangle$$

- Grundlegender Unterschied zum klassischem Fall: Zustand des Qubit ist unbekannt
  - Messung ergibt 0 mit Wahrscheinlichkeit $|\alpha|^2$ und 1 mit Wahrscheinlichkeit $|\beta|^2$
  - keine direkte Korrespondenz zwischen makroskopischer Welt und Qubit

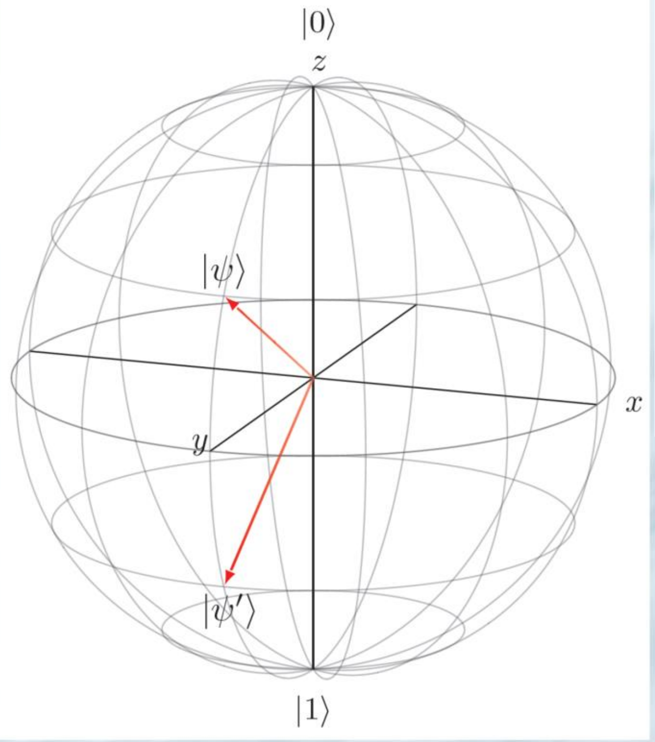
Allgemein

Bloch-Kugel

- Nochmal: $|\alpha|^2 + |\beta|^2 = 1$

→ Satz des Pythagoras (mehrdimensionaler Fall)

→ geometrische Interpretation von Quantenzuständen: „Bloch-Kugel"
  - unendlich große Information
  - Gatter = Rotation auf Kugel
  - $|0\rangle$ → „Nordpol", $|1\rangle$ → „Südpol"
- Messung eines Qubits ergibt entweder „0" oder „1"
  - ändert Zustand des Qubits (kollabiert zu einem klassischen Bit)
- Frage: wieviel Information steckt in einem Qubit, ohne dass man es misst?
  - „hidden information"



Mehrere Qubits

- System aus zwei Qubits → vier Basiszustände $|00\rangle$, $|01\rangle$, $|10\rangle$ und $|11\rangle$
- Möglicher Zustand: Linearkombination dieser vier Zustände

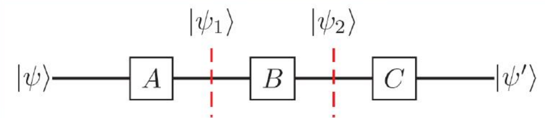
  $|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$
- Beispiel (Einstein-Podolsky-Rosen → „EPR-Paar"): $|\psi\rangle = (|00\rangle + |11\rangle) / \sqrt{2}$
  - Messung des zweiten Qubits führt immer zum selben Ergebnis wie Messung des ersten Qubits
  - starke Korrelation zwischen beiden Qubits
- System aus n Qbits spezifiziert durch 2^n komplexe Amplituden
  - Sycamore: n = 53 → 2^{53} verschiedene Zustände „gleichzeitig"
  - Zum Vergleich 1: klassischer Rechner mit n Bits → einer von 2^{53} Zuständen
  - Zum Vergleich 2: Anzahl der Atome im Universum: ca. $10^{90} = 2^{300}$

Mehrere Qubits

- System aus zwei Qubits → vier Basiszustände $|00\rangle$, $|01\rangle$, $|10\rangle$ und $|11\rangle$
- Möglicher Zustand: Linearkombination dieser vier Zustände
  $|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$
- Beispiel (Einstein-Podolsky-Rosen → „EPR-Paar"): $|\psi\rangle = (|00\rangle + |11\rangle) / \sqrt{2}$
  - ➢ Messung des zweiten Qubits führt immer zum selben Ergebnis wie Messung des ersten Qubits
  - ➢ starke Korrelation zwischen beiden Qubits
- System aus n Qbits spezifiziert durch 2^n komplexe Amplituden
  - ➢ Sycamore: n = 53 → 2^{53} verschiedene Zustände „gleichzeitig"
  - ➢ Zum Vergleich 1: klassischer Rechner mit n Bits → einer von 2^{53} Zuständen
  - ➢ Zum Vergleich 2: Anzahl der Atome im Universum: ca. $10^{90} = 2^{300}$

Quantengatter

- Klassische Computer: Leitungen leiten, Gatter realisieren Wahrheitstabelle
- Quantencomputer:
  - ➢ Leitungen nicht notwendig physikalische Verbindungen
  - ➢ Gatter = Multiplikation eines Zustandes mit unitärer Matrix

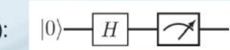


- Beispiel: NOT-Gatter für Qubits (d.h. $|0\rangle \to |1\rangle$, $|0\rangle \to |1\rangle$)
  - ➢ Idee 1: Linearität (als grundlegende Eigenschaft der Quantenmechanik!):
    $\alpha\,|0\rangle + \beta\,|1\rangle \to \alpha\,|1\rangle + \beta\,|0\rangle$
  - ➢ Idee 2: Gatter als 2x2-Matrix $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ $|\psi\rangle$—[X]—$|\psi'\rangle$
- Anwendung X-Gatter: $X\,|1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $X\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

Zufallszahlengenerator

- Weitere wichtige Gatter:
  - Z-Gatter: $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ $\quad |\psi\rangle - \boxed{Z} - |\psi'\rangle$
  - Hadamard-Gatter $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ $\quad |\psi\rangle - \boxed{H} - |\psi'\rangle$

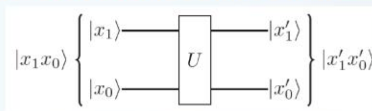
    → $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle); H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ (beide Zustände gleichwahrsch.)
  - Messung: $|\psi\rangle$ — Messung — $\{0, 1\}$
- Einfache Gatterschaltung (Hadamard-Gatter + Messung):



- Ergebnis: echter Zufallszahlengenerator, da beide Zustände jeweils genau 50% !!

Controlled-NOT

- Gatter für mehrere Qubits:



- Beispiel: CNOT-Gatter (vgl. klassisches XOR)

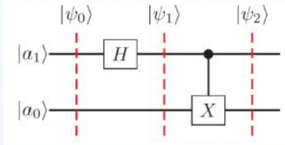
$|x_1\rangle \to |x_1\rangle$, $|x_0\rangle - \boxed{X} - |x_1 \oplus x_0\rangle$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- Beispiel: $CNOT|11\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$

Verschränkung

- Etwas komplizierter: Hadamard + CNOT



- Beispiel für |00⟩:

$$|\psi_0\rangle = |00\rangle$$
$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$
$$|\psi_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

- Ergebnis: erster „Bell-Zustand"

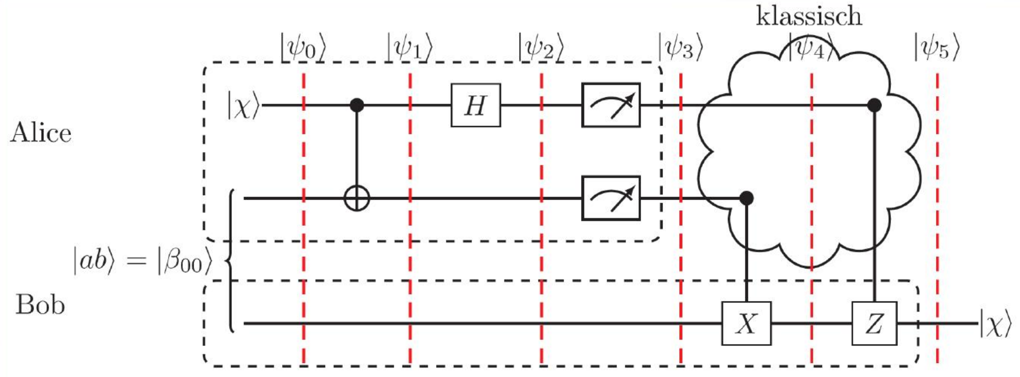
  → Verschränkung von $|a_0\rangle$ und $|a_1\rangle$
- Je nach Eingangskombination gibt es insgesamt vier Bell-Zustände

$$|a_1 a_0\rangle = |00\rangle \Rightarrow |\beta_{00}\rangle = \mathbf{\Phi}^+ = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
$$|a_1 a_0\rangle = |10\rangle \Rightarrow |\beta_{10}\rangle = \mathbf{\Phi}^- = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$
$$|a_1 a_0\rangle = |01\rangle \Rightarrow |\beta_{01}\rangle = \mathbf{\Psi}^+ = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$
$$|a_1 a_0\rangle = |11\rangle \Rightarrow |\beta_{11}\rangle = \mathbf{\Psi}^- = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

Teleportation

Teleportation

- Transport eines Quantenzustands $|\chi\rangle$ von A nach B



- Grundsätzliche Idee der Schaltung:
  - ➢ Alice und Bob haben jeweils eines der verschränkten Qubits ($|a\rangle$ bzw. $|b\rangle$)
  - ➢ Alice führt CNOT, Hadamard und Messung an $|\chi\rangle$ und/oder. $|a\rangle$ durch und schickt das Messergebnis (2 Bits) über klassischen Kanal an Bob
  - ➢ Mit Hilfe dieser Bits und seinem verschränkten Qubit $|b\rangle$ rekonstruiert Bob das gesuchte Qubit $|\chi\rangle$ (ggf. durch Anwendung eines X- und/oder Z-Gatters)

- Schritt 1: $$|\psi_0\rangle = |\chi\rangle\,|\beta_{00}\rangle = (\chi_0\,|0\rangle + \chi_1\,|1\rangle)\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$ $$= \frac{1}{\sqrt{2}}(\chi_0\,|000\rangle + \chi_0\,|011\rangle + \chi_1\,|100\rangle + \chi_1\,|111\rangle)$$

- Schritt 2: $$|\psi_1\rangle = \frac{1}{\sqrt{2}}(\chi_0\,|000\rangle + \chi_0\,|011\rangle + \chi_1\,|110\rangle + \chi_1\,|101\rangle)$$

- Schritt 3: $$|\psi_2\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right)\frac{\chi_0}{\sqrt{2}}(|00\rangle + |11\rangle) + \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)\frac{\chi_1}{\sqrt{2}}(|10\rangle + |01\rangle)$$ $$= \frac{\chi_0}{2}(|000\rangle + |011\rangle + |100\rangle + |111\rangle) + \frac{\chi_1}{2}(|010\rangle + |001\rangle - |110\rangle - |101\rangle)$$

- Messung (+ Skalierung):

Messung	$\lvert b\rangle$ zum Zeitpunkt $\lvert\psi_3\rangle$	Operation	Ergebnis $\lvert\psi_5\rangle$ ($\lvert b\rangle$)
$\lvert 00\rangle$	$\chi_0\lvert 0\rangle + \chi_1\lvert 1\rangle$	(keine)	$\chi_0\lvert 0\rangle + \chi_1\lvert 1\rangle$
$\lvert 01\rangle$	$\chi_0\lvert 1\rangle + \chi_1\lvert 0\rangle$	X	$\chi_0\lvert 0\rangle + \chi_1\lvert 1\rangle$
$\lvert 10\rangle$	$\chi_0\lvert 0\rangle - \chi_1\lvert 1\rangle$	Z	$\chi_0\lvert 0\rangle + \chi_1\lvert 1\rangle$
$\lvert 11\rangle$	$\chi_0\lvert 1\rangle - \chi_1\lvert 0\rangle$	$Z \cdot X$	$\chi_0\lvert 0\rangle + \chi_1\lvert 1\rangle$

- $|b\rangle$ befindet sich im selben Zustand wie ursprünglich $|\chi\rangle$
- Non-cloning Theorem ist nicht verletzt, da der Zustand von $|\chi\rangle$ durch die Messung verlorengegangen ist
- Auch $|a\rangle$ ist durch die Messung kollabiert
- Die Übertragung der beiden Signalisierungsbits erfolgen auf einem klassischen Kanal, also höchstens mit Lichtgeschwindigkeit (kein Widerspruch zur speziellen Relativitätstheorie)

Interferenz

Interferenz

- Beobachtung: Amplituden eines Zustands addieren sich bzw. löschen sich aus (konstruktive bzw. destruktive Interferenz)

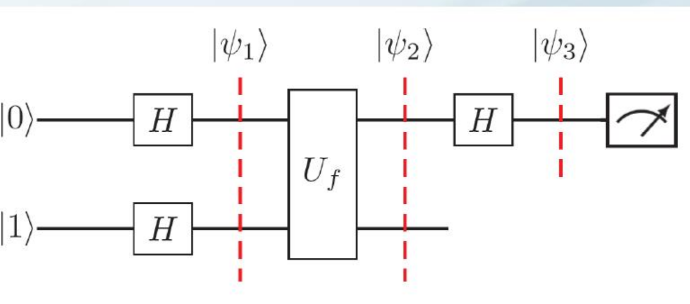
  → das lässt sich ausnutzen, um zusätzliche Informationen über ein Quantengatter U_f (Black Box, Orakel) zu erhalten

- Anwendung: Problem von Deutsch
  - Annahme: es gibt zwei Arten von Münzen, echte (Vorder- und Rückseite unterschiedlich) und gefälschte (Vorder- und Rückseite identisch)
  - Frage: ist eine vorliegende Münze echt oder gefälscht?
  - Klassisch: zwei Berechnungsschritte (Vorder- und Rückseite anschauen)
  - Quantencomputing: nur ein Berechnungsschritt nötig
  - Idee: Qubit in Superposition über beide mögliche Seiten

Algorithmus von Deutsch

- Etwas formaler:
  - Gegeben sei ein „Orakel“ f
  - Frage: ist f mit f : {0; 1} → {0; 1} konstant oder balanciert?
    - konstant: f(0) = f(1)
    - balanciert: f(0) ≠ f(1)
  - Annahme: Orakel ist teuer (d.h. Auswertung von f ist schwierig)
  - Idee: Qubit in Superposition über beide mögliche Eingaben
  - Achtung: Orakel muss in reversibler Form vorliegen!

- Schaltung:



- Orakel (formal): $U_f : |x, y\rangle \mapsto |x, y \oplus f(x)\rangle$ (Addition modulo 2 = XOR)

$$|\psi_0\rangle = |01\rangle$$

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{2}(|0\rangle|0\rangle - |0\rangle|1\rangle + |1\rangle|0\rangle - |1\rangle|1\rangle)$$

$$|\psi_1\rangle \xrightarrow{U_f} \frac{1}{2}(|0\rangle|0 \oplus f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|0 \oplus f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle)$$
$$= \frac{1}{2}(|0\rangle|f(0)\rangle - |0\rangle|1 \oplus f(0)\rangle + |1\rangle|f(1)\rangle - |1\rangle|1 \oplus f(1)\rangle) = |\psi_2\rangle$$

- Es gilt: $|f(x)\rangle - |1 \oplus f(x)\rangle = (-1)^{f(x)}(|0\rangle - |1\rangle)$

$$|\psi_2\rangle = \frac{1}{2}((-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle))$$
$$= \frac{1}{2}((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle) \cdot (|0\rangle - |1\rangle)$$

- Konstanter Fall:

$$|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \xrightarrow{H} |0\rangle \text{ oder}$$
$$|x\rangle = -\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \xrightarrow{H} -|0\rangle$$

- Balancierter Fall:

$$|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \xrightarrow{H} |1\rangle \text{ oder}$$
$$|x\rangle = -\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \xrightarrow{H} -|1\rangle$$

- Ergebnis: es genügt Auswertung des ersten Qubits !!
- Trick: Verlagerung des Ergebnisses in die Amplituden

Zusammenfassung

Zusammenfassung

- Quantencomputing als neues Rechner-Paradigma
- Besonderheiten von Quantenbits
  - Superposition
  - Verschränkung
  - Interferenz
- Jede Schaltung mit Pauli-Gattern + CNOT realisierbar
- Einfache Schaltungen:
  - Zufallszahlen
  - Teleportation
- Algorithmus von Deutsch
- Weitere bekannte Algorithmen
  - Shor-Algorithmus: Faktorisierung grosser Primzahlen (→ Kryptographie!)
  - Grover-Algorithmus: effiziente Suche in einem unsortierten Array
  - Quantensimulation