Digitale Rechenanlagen

Rade Kutil, 9. Oktober 2023

Inhaltsverzeichnis

Blaue Striche links bedeuten, dass hier Grundlagen wiederholt werden, die bekannt sein sollten. Es handelt sich nicht um expliziten Prüfungsstoff. Das Verständnis davon ist aber nötig.

Orange Striche links bedeuten, dass es sich um ein Beispiel handelt. Das Verstehen der Beispiele ist nötig, Auswendiglernen aber sinnlos.

Literatur:

- J. L. Hennessy, D. A. Patterson. *Rechnerarchitektur*. Vieweg Verlag, 1996
- K. Beuth. *Digitaltechnik*. Vogel Buchverlag, 2006
- L. Borucki. *Digitaltechnik*. Teubner Verlag, 1996
- A. S. Tanenbaum, T. Austin. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. Pearson, 2014

1 Informationstheorie

1.1 Codes

Eine **Menge** wird aufzählend so geschrieben:

$$A = \{p, q, a, x\}$$

In einer Menge ist die Reihenfolge egal. Doppelte Nennungen gelten nur einmal.

$$\{m, e, n, g, e\} = \{e, n, g, e, m\} = \{e, g, m, n\}$$

Mengen können eine Ordnung besitzen:

$$e < g < m < n$$

Ein **Zeichenvorrat** ist eine Menge von Zeichen. Ein **Alphabet** ist eine geordnete Menge von Zeichen.

Beispiele:

- Dezimalziffern: $\{0, 1, \ldots, 9\}$
- Binärziffern: $B = \{0, 1\}$
- Hexadezimalziffern: $\{0, 1, \ldots, 9, A, B, C, D, E, F\}$

Ein n-**Tupel** ist eine geordnete Menge der Länge n, in der Elemente auch doppelt auftreten können:

$$(t, u, p, e, l) \neq (t, u, l, p, e) \neq (t, u, l, p, e, l)$$

Üblicherweise steht Tupel für 2-Tupel, Tripel für 3-Tupel, usw. n-Tupel können **konkateniert**, also „zusammengehängt“ werden. Und zwar mit dem Operator $\circ$ oder $\cdot$. Wie bei der Multiplikation wird der Operator oft weggelassen.

$$A = (t, u), \quad B = (p, e, l), \quad A \circ B = A \cdot B = AB = (t, u, p, e, l)$$

Potenzieren kann man natürlich auch:

$$(l, a)^3 = (l, a, l, a, l, a)$$

Ein **Wort** ist ein n-Tupel aus Zeichen eines Zeichenvorrats. Wörter werden der Einfachheit halber oft ohne Klammern und Kommas geschrieben.

$$(w,o) \circ (r,t) = wo \cdot rt = wort$$
$$al^2e^2 = allee, \qquad ba(na)^2 = banana$$

Die Konkatenierung zweier Zeichenvorräte oder Wörtermengen ist definiert durch die Menge der Konkatenierungen aller Elemente der ersten mit allen Elementen der zweiten Menge, wobei einzelne Zeichen als 1-Tupel interpretiert werden. Die Potenzierung einer Menge entspricht dann einer (mehrfachen) Konkatenierung der Menge mit sich selbst.

$$\{a,b\}^3 = \{a,b\} \circ \{a,b\} \circ \{a,b\} = \{aa,ab,ba,bb\} \circ \{a,b\}$$
$$= \{aaa,aab,aba,abb,baa,bab,bba,bbb\}$$

Beispiele:

- $\{0,1\}^4 = \{0000,0001,\ldots,1111\}$: 4-Bit-Binärwörter, Nibbles
- $\{0,1\}^8$: 8-Bit-Binärwörter, Bytes
- $\{0,1\}^{16}$: 16-Bit-Binärwörter, Half-Words (früher Words)
- $\{0,1\}^{32}$: 32-Bit-Binärwörter, Words
- $\{0,1\}^{64}$: 64-Bit-Binärwörter, Double-Words

Element, Vereinigung, Schnittmenge:

$$e \in \{e,g,m,n\}$$
$$\{e,g,m,n\} \cup \{m,e,h,r\} = \{e,g,h,m,n,r\}$$
$$\{e,g,m,n\} \cap \{m,e,h,r\} = \{e,m\}$$

Der $*$-Operator produziert die Vereinigung aller Wörtermengen A^n mit beliebiger Länge n.

$$A = \{a,b\}$$
$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots$$
$$= \{\lambda,a,b,aa,ab,ba,bb,aaa,aab,\ldots,aabbababbbab,\ldots\}$$

λ steht für das leere Wort mit der Länge 0.

- $\{0,1\}^* = \{\lambda,0,1,00,01,10,11,001,\ldots\}$: Binärwörter

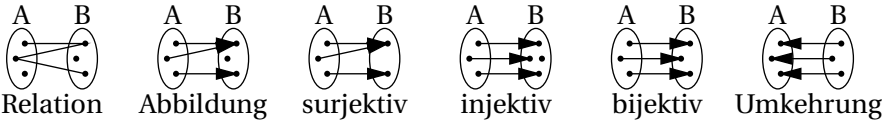


Abbildung 1: Abbildung und Umkehrabbildung

Mengen können auch **intensional** definiert werden, also über eine Bedingung, die alle Elemente erfüllen müssen.

$$\{2n \mid n \in \mathbb{N} \wedge n < 5\} = \{m \in \mathbb{N} \mid 2\lfloor m/2 \rfloor = m \wedge m \leq 8\} = \{2,4,6,8\}$$

$\mathbb{N}$ sind die natürlichen Zahlen $\{1,2,3,\ldots\}$, das $\mid$ steht für „für die gilt, dass“, $\wedge$ steht für „und“, $\vee$ steht für „oder“, $\lfloor\ \rfloor$ steht für Abrunden, $\lceil\ \rceil$ für Aufrunden.

$$\{a \circ b \mid a = 0^n \circ 1 \wedge n \geq 0 \wedge b \in \{0,1\}^2\}$$
$$= \{100, 101, 110, 111, 0100, 0101, 0110, 0111, 00100, \ldots\}$$

$$\{10000, 01000, 00100, 00010, 00001\} = \{0^m \cdot 1 \cdot 0^n \mid m+n = 4 \wedge m \geq 0 \wedge n \geq 0\}$$

Eine **Relation** zwischen zwei Mengen A und B ist eine Teilmenge von $A \circ B$. Sie ordnet also *manchen* Elementen von A *manche* Elemente von B zu. Siehe Abbildung 1.

Eine **Abbildung** f von A auf B, geschrieben $f : A \longrightarrow B$, $f(x) = y$, ist eine Relation, die *jedem* Element von A *genau ein* Element von B zuordnet. A heißt Definitionsmenge, B Zielmenge.

Die Abbildung ist **surjektiv**, wenn jedes Element von B *mindestens* einem Element von A zugeordnet ist. Die Abbildung ist **injektiv**, wenn jedes Element von B *maximal* einem Element von A zugeordnet ist. Die Abbildung ist **bijektiv**, wenn sie surjektiv *und* injektiv ist, wenn also jedes Element von B *genau einem* Element von A zugeordnet ist.

Eine bijektive Abbildung ist also auch bei Vertauschung von A und B eine Abbildung. Diese Abbildung nennt man **Umkehrabbildung** $f^{-1} : B \longrightarrow A$. Klarerweise ist $f^{-1}(f(x)) = x$ und $f(f^{-1}(y)) = y$.

Ein **Code** (oder **Codierung**) ist eine Abbildung c einer Menge A auf die Wörtermenge B^* eines Zeichenvorrats B, $c : A \longrightarrow B^*$. Die Elemente der Definitionsmenge A heißen **Symbole**, die Elemente der Zielmenge B^* heißen auch Codes.

Ein Code sei wie folgt definiert:

$$c : \{0,\ldots,7\} \longrightarrow \{0,1\}^3, \quad c(k) := a_2 a_1 a_0$$

$$a_0 := k \bmod 2, \qquad a_1 := \lfloor k/2 \rfloor \bmod 2, \qquad a_2 := \lfloor k/4 \rfloor$$

mod berechnet den Divisionsrest ($8 \bmod 3 = 2$). Aus dieser Definition folgt z.B.:

$$c(5) = \lfloor 5/4 \rfloor \circ (\lfloor 5/2 \rfloor \bmod 2) \circ (5 \bmod 2) = 1 \circ (2 \bmod 2) \circ 1 = 101$$

Der gesamte Code ist dann der 3-Bit-Binärcode:

$$c(0) = 000,\ c(1) = 001,\ c(2) = 010,\ c(3) = 011,$$
$$c(4) = 100,\ c(5) = 101,\ c(6) = 110,\ c(7) = 111$$

:= bedeutet „ist definiert als".

Die **Decodierung** eines Codes c ist die Umkehrfunktion c^{-1}.

$$c^{-1}(000) = 0,\ c^{-1}(001) = 1,\ c^{-1}(010) = 2,\ c^{-1}(011) = 3,$$
$$c^{-1}(100) = 4,\ c^{-1}(101) = 5,\ c^{-1}(110) = 6,\ c^{-1}(111) = 7$$

Die **Bildmenge** einer Abbildung $f : A \longrightarrow B$ ist die Teilmenge von B, deren Elemente alle einem Element von A zugeordnet sind. Man schreibt die Bildmenge $f(A)$.

$$f(A) = \{y \in B \mid \exists x \in A : f(x) = y\} = \{f(x) \mid x \in A\} \subseteq B$$

$\exists$ steht für „es gibt ein".

Eine injektive aber nicht surjektive Abbildung hat eigentlich keine Umkehrabbildung, weil es Elemente in B gibt, die keinem Element von A zugeordnet sind. Wenn man aber B auf die Bildmenge beschränkt, wird die Abbildung surjektiv, und daher bijektiv, ist daher umkehrbar.

$$f^{-1} : f(A) \longrightarrow A, \quad f^{-1}(f(x)) := x$$

Die **gültigen** Codewörter eines Codes sind jene aus der Bildmenge des Codes.

Ein Code ist **decodierbar**, wenn die Umkehrfunktion existiert, d.h. wenn der Code injektiv ist.

$$c : \{0,\ldots,n\} \longrightarrow \{0,1\}^3, \qquad c(k) := a_2 a_1 a_0$$

$$a_0 := k \bmod 2, \quad a_1 := (k \bmod 3) \bmod 2, \quad a_2 := ((k+1) \bmod 3) \bmod 2$$

Bis zu welchem n ist dieser Code decodierbar?

$c(0) = 100, c(1) = 011, c(2) = 000, c(3) = 101, c(4) = 010, c(5) = 001, c(6) = 100 = c(0)$

Antwort: Bis zu $n = 5$.

1.1.1 Codes fester Länge

Bei Codes fester Länge ist jedes Codewort gleich lang, d.h. $c : A \longrightarrow \{0,1\}^n$.

Der einfachste Code ist der n-Bit-**Binärcode**. Er bildet 2^n Symbole, z.B. $\{0,1,\ldots,2^n-1\}$, auf die 2^n Wörter aus $\{0,1\}^n$ ab, und zwar auf folgende Weise. Es sei c_n der n-Bit-Binärcode. $c_1(0) = 0$, $c_1(1) = 1$, und

$$c_n(k) = \begin{cases} 0 \circ c_{n-1}(k) & k < 2^{n-1} \\ 1 \circ c_{n-1}(k-2^{n-1}) & k \geq 2^{n-1}. \end{cases}$$

Daraus ergibt sich:

$$\begin{aligned} c_2(0) &= 0 \circ c_1(0) = 00 \\ c_2(1) &= 0 \circ c_1(1) = 01 \\ c_2(2) &= 1 \circ c_1(0) = 10 \\ c_2(3) &= 1 \circ c_1(1) = 11 \end{aligned}$$

$$\begin{aligned} c_3(0) &= 0 \circ c_2(0) = 000 \\ c_3(1) &= 0 \circ c_2(1) = 001 \\ c_3(2) &= 0 \circ c_2(2) = 010 \\ c_3(3) &= 0 \circ c_2(3) = 011 \\ c_3(4) &= 1 \circ c_2(0) = 100 \\ c_3(5) &= 1 \circ c_2(1) = 101 \\ c_3(6) &= 1 \circ c_2(2) = 110 \\ c_3(7) &= 1 \circ c_2(3) = 111 \end{aligned}$$

Ein **Übertragungsfehler** entsteht, wenn in einem Codewort ein oder mehrere Bits kippen, d.h. aus 0 wird 1 oder umgekehrt. Um solche Fehler zu erkennen, ist es von Vorteil, wenn durch den Fehler kein gültiges Codewort entsteht. Damit das nicht passiert, müssen die Codewörter möglichst unterschiedlich sein, also eine große Distanz aufweisen.

Die **Hammingdistanz** zweier Codewörter ist die Anzahl der Positionen, an denen die Zeichen unterschiedlich sind.

$$\mathrm{Hd}(a_{n-1}\ldots a_1 a_0, b_{n-1}\ldots b_1 b_0) := |\{k \mid 0 \leq k < n \wedge a_k \neq b_k\}|$$

Die Betragsstriche $|\{\ldots\}|$ bedeuten hier die Anzahl der Elemente in der Menge, oft auch als $\#\{\ldots\}$ geschrieben.

$$\mathrm{Hd}(1\overline{1}01\overline{0}, 1\overline{0}11\overline{1}) = 3, \qquad \mathrm{Hd}(01\overline{01}, 01\overline{10}) = 2$$

Für einen ganzen Code kann man die minimale und maximale Hammingdistanz zwischen allen Paaren von Codewörtern ermitteln.

$$\min \mathrm{Hd}(c) := \min_{i \neq j} \mathrm{Hd}(c(i), c(j))$$
$$\max \mathrm{Hd}(c) := \max_{i \neq j} \mathrm{Hd}(c(i), c(j))$$

$$c : \{0, \ldots, 4\} \longrightarrow \{0,1\}^5$$

$$c(0) = 10110, c(1) = 00001, c(2) = 01010, c(3) = 11111, c(4) = 10000$$

Hd	0	1	2	3	4
0		4	3	2	2
1	4		3	4	2
2	3	3		3	3
3	2	4	3		4
4	2	2	3	4	

$\min \mathrm{Hd}(c) = 2$
$\max \mathrm{Hd}(c) = 4$

Ein k-**Bit-Fehler** entsteht, indem bei der Übertragung eines Codeworts a ein Codewort a' entsteht, in dem k Bits gekippt sind, also $\mathrm{Hd}(a, a') = k$. Der Fehler kann **erkannt** werden, wenn a' kein gültiges Codewort ist; wenn a' ein gültiges Codewort ist, bleibt der Fehler **unerkannt** und wird falsch decodiert.

$$c(0) = 10110, c(1) = 00001, c(2) = 01010, c(3) = 11111, c(4) = 10000$$

Gibt es in diesem Code einen 2-Bit-Fehler, der erkannt bzw. nicht erkannt wird?

Ja: Z.B. $c(0)' = 10110' = \overline{0}0\overline{0}10 \notin c(A)$, $\mathrm{Hd}(c(0)', c(0)) = 2$, wird erkannt.

Und: $c(0)' = 1\overline{1}11\overline{1} = c(3)$ wird nicht erkannt.

Ein k-Bit-Fehler wird jedenfalls erkannt, wenn $k < \min \mathrm{Hd}(c)$ oder $k > \max \mathrm{Hd}(c)$ ist. Andernfalls wird der Fehler möglicherweise nicht erkannt.

Beweis Wenn $k < \min \mathrm{Hd}(c)$ oder $k > \max \mathrm{Hd}(c)$ ist, kann nicht $c(m)' \in c(A)$ sein (also $c(m)' = c(m')$ für ein m'), weil sonst $\mathrm{Hd}(c(m), c(m')) < \min \mathrm{Hd}(c)$ oder $> \max \mathrm{Hd}(c)$ wäre. Daher wird der Fehler erkannt.

$$c(0) = 0000, c(1) = 0011, c(2) = 0101, c(3) = 1110$$
$$\min \mathrm{Hd}(c) = 2, \max \mathrm{Hd}(c) = 3$$

Zeige (bestätige), dass alle durch 1-Bit-Fehler bzw. 4-Bit-Fehler entstehenden Codewörter keine Überschneidung mit den gültigen Codewörtern haben.

1-Bit-Fehler: $c(A)' = \{0001, 0010, 0100, 1000, \cancel{0010}, \cancel{0001}, 0111, 1011, \cancel{0100}, 0110, \cancel{0001}, 1101, \ 1111, 1100, 1010, \cancel{0110}\}$, $c(A)' \cap c(A) = \emptyset$.

4-Bit-Fehler: $c(A)' = \{1111, 1100, 1010, 0001\}$, $c(A)' \cap c(A) = \emptyset$.

Umgekehrt, z.B. welcher 2- bzw. 3-Bit-Fehler von $c(0)$ wäre nicht erkennbar?

$c(0)' = 0000' = 00\overline{11} = c(1)$, $c(0)' = 0000' = \overline{1110} = c(3)$.

Das **Paritätsbit** ist eine einfache Möglichkeit zur Erzeugung eines fehlererkennenden Codes. Dabei wird ein zusätzliches Bit an einen n-Bit-Binärcode gehängt. Es gibt zwei Versionen, die **gerade** und **ungerade** Parität (parity). Bei der geraden Parität (even parity) wird ein 1-Bit angehängt, wenn die Anzahl der 1-en im Codewort *ungerade* ist, ansonsten ein 0-Bit, sodass die Anzahl der 1-en im neuen Codewort *gerade* wird. Bei der ungeraden Parität (odd parity) ist es umgekehrt.

$$c_n(k) = a_{n-1} \cdots a_1 a_0, \quad m = |\{i \mid a_i = 1\}|$$
$$\Rightarrow \quad c_{\text{even parity}}(k) = c_n(k) \circ \begin{cases} 0 & m \bmod 2 = 0 \\ 1 & m \bmod 2 = 1\,. \end{cases}$$

Der Code mit 3 Bit plus 1 gerades Paritätsbit sieht so aus:

$c(0) = 0000$	$c(4) = 1001$
$c(1) = 0011$	$c(5) = 1010$
$c(2) = 0101$	$c(6) = 1100$
$c(3) = 0110$	$c(7) = 1111$

Die minimale Hammingdistanz eines Codes mit Paritätsbit ist 2. Es können also 1-Bit-Übertragungsfehler erkannt werden.

Beweis Wir betrachten zwei Codewörter $a \circ p_a$ und $b \circ p_b$, wobei p_a und p_b die jeweiligen Paritätsbits sind. Ist $a \circ p_a \neq b \circ p_b$, dann ist auch $a \neq b$, weil sonst $p_a = p_b$ wäre. Daher ist $\mathrm{Hd}(a, b) \geq 1$. Ist $\mathrm{Hd}(a, b) \geq 2$, dann ist auch $\mathrm{Hd}(a \circ p_a, b \circ p_b) \geq 2$, und die Behauptung ist erfüllt. Ist $\mathrm{Hd}(a, b) = 1$, dann unterscheiden sich a und b in genau einem Bit. Daher hat eines eine gerade und das andere eine ungerade Anzahl von 1-en. Daher unterscheiden sich auch die Paritätsbits p_a und p_b. Und daher ist $\mathrm{Hd}(a \circ p_a, b \circ p_b) = 2$.

Codewort mit 7 Bits + 1 Paritätsbit: 0011011 0 (4 1-en, gerade Parität)

- Wenn ein 0-Bit kippt, entsteht z.B. 1011011 0
  (5 1-en, ungerade, Fehler erkannt)
- Wenn ein 1-Bit kippt, entsteht z.B. 0001011 0
  (3 1-en, ungerade, Fehler erkannt)
- Wenn das Parity-Bit kippt, entsteht 0011011 1
  (5 1-en, ungerade, Fehler erkannt)

Der Binomialkoeffizient $\binom{n}{k}$ gibt an, wieviele Möglichkeiten es gibt, k Elemente aus n auszuwählen.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots 1}{k\cdots 1\cdot(n-k)(n-k-1)\cdots 1} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$$

Beispiel:

$$|\{\{a,b\} \mid a,b \in \{1,2,3,4\} \wedge a \neq b\}| = |\{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}|$$
$$= \binom{4}{2} = \frac{4\cdot 3}{2\cdot 1} = 6$$

Ein **k-aus-n-Code** ist ein Code, in dem die Codewörter genau k 1-en besitzen. Daher gibt es $\binom{n}{k}$ Codewörter.

$$c : \left\{0,\ldots,\binom{n}{k}-1\right\} \longrightarrow \{a \in \{0,1\}^n \mid \mathrm{Hd}(a,0\ldots 0) = k\}$$

2-aus-4-Code:

$$c : \{0,\ldots,5\} \longrightarrow \{1100,1010,1001,0110,0101,0011\}$$

In einem k-aus-n-Code ist $\min \mathrm{Hd}(c) = 2$ und $\max \mathrm{Hd}(c) = 2\min(k, n-k)$. Alle Hamming-Distanzen sind gerade.

Beweis Die kleinste Hamming-Distanz entsteht, wenn sich zwei Codes nur in einer 1-Position unterscheiden. Dadurch wird eine 1 zu einer 0 und eine 0 zu einer 1, und die Hammingdistanz ist 2. Die größte Hamming-Distanz entsteht, wenn sich alle 1-Positionen unterscheiden. Für jede der k 1-Positionen wird eine 1 zu einer 0 und eine 0 zu einer 1, die Hammingdistanz ist dann $2k$. Wenn allerdings $k \geq n/2$ ist, können sich nicht alle 1-Positionen unterscheiden. In diesem

Fall gilt Obiges für die 0-Positionen, von denen es $n-k$ gibt. Daher $\min(k, n-k)$. Alle Hammingdistanzen sind gerade, weil sich für jede sich unterscheidende 1-Position zwei Bits unterscheiden.

2-aus-6-Code:
$a = 010100$, 1-Positionen: {2,4}
$b = 000110$, 1-Positionen: {4,5}
1 Position unterschiedlich $(2 \mapsto 5) \quad \Rightarrow \quad \mathrm{Hd}(a,b) = 2 \cdot 1 = 2$.

In einem 1-aus-n-Code ist $\min \mathrm{Hd}(c) = \max \mathrm{Hd}(c) = 2$.

Falls es nicht möglich ist, Fehler zu erkennen und/oder zu korrigieren, sollte ein Fehler zumindest nicht starke Auswirkungen haben. Oft ist ein fehlerhaft decodiertes Symbol weniger schlimm, wenn es im Alphabet weniger weit vom richtigen Symbol entfernt ist. Z.B. fehlerhaft decodierte Audio-Signale produzieren ein deutlich leiseres Knacksen, wenn der falsch decodierte Signalwert nahe dem richtigen liegt. Das wird erreicht, indem sich benachbarte Codes durch möglichst wenige Bits unterscheiden.

Ein **Gray-Code** (nach Frank Gray) ist ein Code, in dem sich benachbarte Codes nur in einem Zeichen unterscheiden.

$$c : \{0, \ldots, m-1\} \longrightarrow \{0,1\}^n, \quad \mathrm{Hd}(c(k), c(k+1)) = 1$$

Die **Transitionssequenz** t eines Gray-Codes ist das $m-1$-Tupel der Positionen, an denen sich aufeinander folgende Codewörter unterscheiden.

$$t = (t_1, \ldots, t_{m-1}), \quad c(i-1)_{t_i} \neq c(i)_{t_i}$$

$$\begin{array}{ll} c(0) = 000 & \\ c(1) = 00\overline{1} & t_1 = 0 \\ c(2) = 0\overline{1}1 & t_2 = 1 \\ c(3) = \overline{1}11 & t_3 = 2 \\ c(4) = 1\overline{0}1 & t_4 = 1 \\ c(5) = 10\overline{0} & t_5 = 0 \\ c(6) = 1\overline{1}0 & t_6 = 1 \\ c(7) = \overline{0}10 & t_7 = 2 \end{array} \qquad t = (0,1,2,1,0,1,2)$$

Bei einem **zyklischen Gray-Code** unterscheiden sich auch das letzte und das erste Codewort nur an einer Position. Die Transitionssequenz wird dann um diese Position erweitert.

Der Gray-Code im letzten Beispiel ist zyklisch.
$\mathrm{Hd}(c(7), c(0)) = 1, \quad t = (0,1,2,1,0,2,(1))$

Ein nicht-zyklischer Gray-Code ist auch möglich:
$c((0,\ldots,7)) = (000, 00\overline{1}, 0\overline{1}1, 01\overline{0}, \overline{1}10, 1\overline{0}0, 10\overline{1}, 1\overline{1}1), \quad t = (0,1,0,2,1,0,1),$
$\mathrm{Hd}(c(7), c(0)) = 3.$

Eine strukturierte Möglichkeit, einen Gray-Code mit 2^n Codewörtern zu erzeugen, ist folgendes rekursives Schema. $T(k)$ ist die Transitionssequenz für einen k-Bit-Gray-Code, rekursiv definiert durch:

$$T(1) := (0)$$
$$T(k+1) := T(k) \circ k \circ T(k)$$

$$T(2) = T(1) \circ 1 \circ T(1) = (0) \circ 1 \circ (0) = (0,1,0)$$
$$T(3) = T(2) \circ 2 \circ T(2) = (0,1,0) \circ 2 \circ (0,1,0) = (0,1,0,2,0,1,0)$$
$$\Rightarrow \quad c((0,\ldots,7)) = (000, 00\overline{1}, 0\overline{1}1, 01\overline{0}, \overline{1}10, 11\overline{1}, 1\overline{0}1, 100)$$

Diese Transitionssequenz führt auch zu einem zyklischen Gray-Code.

1.1.2 Codes variabler Länge

Die Codierung einer ganzen Sequenz von Symbolen ist die Konkatenierung aller Codewörter, ein „Zeichenstrom“, im Binärfall auch **Bitstream** genannt.

$$c((a_1, a_2, \ldots, a_m)) = c(a_1 a_2 \cdots a_m) := c(a_1) \circ c(a_2) \circ \cdots \circ c(a_m)$$

Die Decodierung eines solchen Zeichenstroms ist die Umkehrabbildung.

$$c^{-1}(c(a_1) \circ c(a_2) \circ \cdots \circ c(a_m)) = a_1 a_2 \cdots a_m$$

Für einen Code fester Länge $c(A) = 00, c(B) = 01, c(C) = 10, c(D) = 11$:

$$c(DABCCA) = 110001101000$$

Und die Umkehrung:
$$c^{-1}(00101110) = ACDC$$

Bei generellen Codes variabler Länge $c : A \longrightarrow \{0,1\}^*$ gibt es das Problem, dass ein zusätzliches Trennzeichen notwendig sein kann, damit ein Zeichenstrom decodierbar ist.

$$c(A) = 0, c(B) = 1, c(C) = 01, c(D) = 10$$

Decodiere 010. Es gibt zwei Möglichkeiten:

- $c^{-1}(0 \cdot 10) = (A, D)$
- $c^{-1}(01 \cdot 0) = (C, A)$

Morsecode: $c(E) = \cdot, c(T) = -, c(K) = - \cdot -$
Decodiere $- \cdot -$.

- $c^{-1}(- \cdot -) = K$
- $c^{-1}(- \cdot -) = TET$

Man muss also eine kleine Pause als Trennzeichen einfügen.

Das Problem tritt genau dann auf, wenn ein Codewort Präfix eines anderen ist. Ein Codewort a ist Präfix eines Codeworts b, wenn es ein Wort c gibt, sodass $a \circ c = b$ ist.

$$a \,\text{prefix}\, b :\Leftrightarrow \exists c : b = a \circ c$$

$\Leftrightarrow$ steht für „genau dann, wenn“. $:\Leftrightarrow$ steht für „ist definiert als genau dann, wenn“. $\exists \ldots$: steht für „es gibt ein …, sodass“. $\nexists \ldots$: steht für „es gibt *kein* …, sodass“. $\forall \ldots$: steht für „für alle … gilt“.

$a = 01$, $b = 0110$. Es gilt $a \,\text{prefix}\, b$, weil für $c = 10$ gilt: $b = a \circ c$.

Die **Fano-Bedingung** formuliert nun die Bedingung, dass kein Codewort Präfix eines anderen ist:

$$\forall k \nexists l \neq k : c(k) \,\text{prefix}\, c(l)\,.$$

Es sei $c(A) = 1, c(B) = 01, c(C) = 001, c(D) = 011$

- Ist die Fano-Bedingung erfüllt?
  Wir müssen nur kürzere mit längeren Codewörtern vergleichen.
  $1 \,\text{prefix}\, 01$? Nein. $1 \,\text{prefix}\, 001$? Nein. $1 \,\text{prefix}\, 011$? Nein. $01 \,\text{prefix}\, 001$? Nein.
  $01 \,\text{prefix}\, 011$? Ja, weil $011 = 01 \circ 1$. Daher ist die Fano-Bedingung nicht erfüllt.
- Was wäre das kürzeste Wort für $c(D)$, sodass die Fano-Bedingung erfüllt ist?

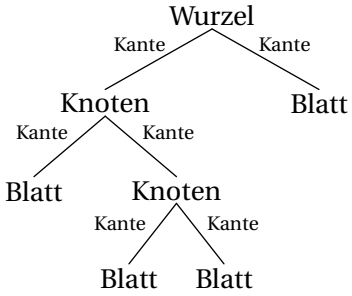


Abbildung 2: Binärer Baum

$c(D)$ darf nicht mit 1 beginnen, weil sonst $c(A)$ Präfix wäre.
$c(D)$ darf nicht 0 sein, weil es sonst Präfix von $c(B)$ wäre.
$c(D)$ darf nicht 00 sein, weil es sonst Präfix von $c(C)$ wäre.
$c(D)$ darf nicht mit 01 beginnen, weil sonst $c(B)$ Präfix wäre.
$c(D) = 000$ ist die Lösung, weil kein anderes Codewort Präfix davon ist, und 000 kein Präfix eines anderen Codeworts ist.

Wenn die Fano-Bedingung erfüllt ist, ist jeder korrekt codierte Zeichenstrom korrekt decodierbar.

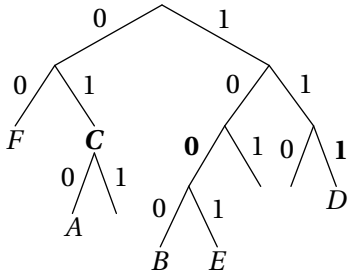
Beweis Wenn der Zeichenstrom aus $c(a_1) \circ c(a_2) \circ c(a_3) \cdots$ besteht, dann wird als erstes Symbol a_1 decodiert. Würde stattdessen ein anderes Symbol b decodiert, wäre entweder $c(b)$ kürzer, dann wäre $c(b)$ prefix $c(a_1)$, und die Fano-Bedingung verletzt, oder es wäre $c(b)$ länger, dann wäre $c(a_1)$ prefix $c(b)$, und wiederum die Fano-Bedingung verletzt, oder es wäre $c(b)$ gleich lang, dann wäre $c(b) = c(a_1)$ und daher $b = a_1$. Nach der Decodierung von a_1 kann das Codewort $c(a_1)$ vom Beginn entfernt werden, und der verbleibende Zeichenstrom $c(a_2) \circ c(a_3) \cdots$ wird auf gleiche Weise weiter decodiert.

$$c(A) = 1, c(B) = 01, c(C) = 001, c(D) = 000$$

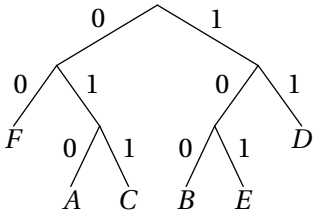
$$c^{-1}(100101000) = c^{-1}(1) \circ c^{-1}(001) \circ c^{-1}(01) \circ c^{-1}(000) = ACBD$$

Ein **binärer Baum** ist ein zusammenhängender Graph mit Wurzel, in dem jeder Knoten außer der Wurzel genau einen Elternknoten hat, und jeder Knoten maximal zwei Kinderknoten hat. Siehe Abbildung 2. Knoten ohne Kinderknoten heißen Blätter, die anderen innere Knoten. Die Verbindungen zwischen den Knoten heißen Kanten. Diese können mit einem Kantenwert belegt sein. Ein Pfad ist eine Folge von Knoten, die aufeinanderfolgend durch Kanten verbunden sind.

Codes können in einem **Codebaum** dargestellt werden. Ein Codebaum ist ein bi-



(a) $c(A) = 010$, $c(B) = 1000$, $c(C) = 01$, $c(D) = 111$, $c(E) = 1001$, $c(F) = 00$



(b) $c(A) = 010$, $c(B) = 100$, $c(C) = 011$, $c(D) = 11$, $c(E) = 101$, $c(F) = 00$

Abbildung 3: Codebäume

närer Baum, in dem in jeder Verzweigung die zwei Kanten mit jeweils einer 0 und einer 1 belegt sind. Symbole sind einem der Knoten so zugeordnet, dass der Pfad von der Wurzel zum Knoten das Codewort ergibt, wenn man die Kantenwerte konkateniert. Siehe Abbildung 3.

Die Verletzung der Fano-Bedingung erkennt man in einem Baum daran, dass ein Symbol einem inneren Knoten zugeordnet ist. In Abbildung 3 (a) betrifft das das Symbol C. Wenn ein Knoten nur einen Kindknoten hat, bedeutet das, dass der Code an der Stelle unnötige Bits beinhaltet. In Abbildung 3 (a) betrifft das die 0 über BE und die 1 über D. In Abbildung 3 (b) sind die Mängel dieses Codes behoben.

Die Decodierung eines Bitstreams kann auch mit Hilfe eines Codebaums erfolgen. Dabei bewegt man sich im Codebaum nach links oder rechts, je nachdem welches Bit vom Bitstream gelesen wurde, bis man ein Blatt erreicht. Das Symbol auf dem Blattknoten gilt dann als decodiert, und man springt zum Wurzelknoten zurück.

Ein **Ereignis** e hat eine **Wahrscheinlichkeit** $P(e)$, auch p_e geschrieben. Es gilt $0 \leq P(e) \leq 1$.

Wenn sich zwei Ereignisse d und e gegenseitig ausschließen (wie z.B. d = „3 gewürfelt“ und e = „4 gewürfelt“), dann ist die Wahrscheinlichkeit, dass d *oder* e eintritt, $P(d \vee e) = P(d) + P(e)$.

Wenn eine Menge von Ereignissen $\{e_1, \ldots, e_n\}$ so gestaltet ist, dass sich die Ereignisse gegenseitig ausschließen und sicher eines der Ereignisse eintreten *muss*

(wie z.B. e_k = „k gewürfelt“ für $k = 1,\ldots,6$), dann gilt: $\sum_{k=1}^{n} P(e_k) = 1$.

Wenn jedem Ereignis ein Wert $l(e_k)$ zugeordnet ist (z.B. 10 Punkte für „6 gewürfelt“, also $l(e_6) = 10$), dann ist der **Erwartungswert** von l definiert als

$$\mathrm{E}(l) := \sum_{k=1}^{n} P(e_k) l(e_k) .$$

Beispiel: 10 Punkte für „6 gewürfelt“, 5 Punkte für „3 bis 5 gewürfelt“, 0 Punkte für „1 bis 2 gewürfelt”. Die mittlere Punktezahl ist dann:

$$\mathrm{E}(l) = \frac{1}{6} \cdot 10 + \frac{3}{6} \cdot 5 + \frac{2}{6} \cdot 0 \approx 4.1667$$

Manchmal sind statt Wahrscheinlichkeiten **absolute Häufigkeiten** H_k gegeben. Z.B. $H_1 = 10$ grüne Äpfel, $H_2 = 15$ rote Äpfel. Indem man durch die Gesamtanzahl dividiert, erhält man **relative Häufigkeiten** h_k, z.B. $h_1 = 10/25 = 0.4$ und $h_2 = 15/25 = 0.6$. Die relativen Häufigkeiten dienen dann als Approximation (Schätzer) der Wahrscheinlichkeiten p_k und können stattdessen verwendet werden.

Die **Codelänge** $l(a)$ eines Codeworts a ist die Anzahl der Bits (oder Zeichen), also z.B. $l(11010) = 5$. Wenn jedes zu codierende Symbol A_k eine Auftrittswahrscheinlichkeit von $P(A_k)$ hat, dann ist die **mittlere Codelänge** L definiert als der Erwartungswert der Codelängen:

$$L = \mathrm{E}(l(c)) = \sum_{k=1}^{n} P(A_k) l(c(A_k)) .$$

Die mittlere Codelänge gibt also an, wie viele Bits ein einzelnes Symbol im Schnitt benötigt.

$$c(A) = 01,\ c(B) = 001,\ c(C) = 000,\ c(D) = 1,$$
$$P(A) = 0.25,\ P(B) = 0.125,\ P(C) = 0.125,\ P(D) = 0.5$$
$$L = 0.25 \cdot 2 + 0.125 \cdot 3 + 0.125 \cdot 3 + 0.5 \cdot 1 = 1.75$$

Im obigen Beispiel sind wahrscheinlichere Codewörter kürzer. Das führt zu einer kleineren mittleren Codelänge, was besser ist, weil damit Bitstreams komprimiert werden können.

Es gibt nun für jede Wahrscheinlichkeitsbelegung eine optimale Codierung, die Bitstreams im Mittel am kürzesten werden lässt. Diese Codierung kann mit dem **Huffman-Algorithmus** ermittelt werden. Dazu werden zuerst die Symbole ihrer

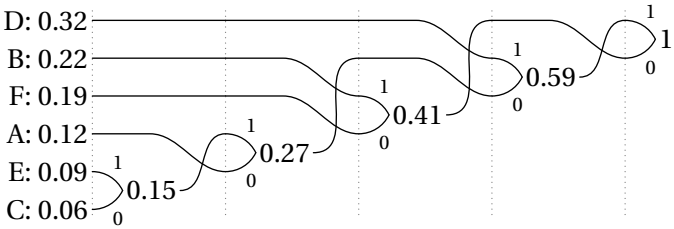


Abbildung 4: Huffman-Algorithmus

Wahrscheinlichkeit nach absteigend sortiert. Dann werden die zwei Symbole mit geringster Wahrscheinlichkeit zusammengefasst. Dass das eine *oder* das andere dieser zwei Symbole auftritt, hat eine Wahrscheinlichkeit, die der Summe der beiden einzelnen Wahrscheinlichkeiten entspricht. Diese muss nun anstatt der beiden zusammengefassten Symbole in die Symbolliste einsortiert werden. Danach fährt man fort, bis alle Symbole zusammengefasst wurden. Dadurch entsteht ein Codebaum, aus dem man die Codes ablesen kann.

$$P(A) = 0.12,\ P(B) = 0.22,\ P(C) = 0.06,$$
$$P(D) = 0.32,\ P(E) = 0.09,\ P(F) = 0.19$$

Abbildung 4 zeigt den Algorithmus. Links sind die Symbole absteigend nach Wahrscheinlichkeit sortiert. Zuerst wird C und E zusammengefasst. Die Summenwahrscheinlichkeit 0.15 wird dann einsortiert, und zwar auf die Höhe von A, welches um eine Stelle nach unten rückt. Danach wird 0.15 und A zusammengefasst, und die Summenwahrscheinlichkeit 0.27 an der Stelle von B einsortiert, wodurch B und F nach unten rücken. Und so weiter.

Jede Zusammenfassung entspricht einer Verzweigung im (quasi nach rechts gekippten) Codebaum. Die Kanten der Verzweigung werden mit 0 und 1 belegt, und zwar nach Konvention immer zur niedrigeren Wahrscheinlichkeit hin mit 0. Die Wurzel ist ganz rechts bei Wahrscheinlichkeit 1. Daraus können nun die Codes von der Wurzel aus nach links den Verzweigungen folgend abgelesen werden.

$$c(A) = 100,\ c(B) = 01,\ c(C) = 1010,$$
$$c(D) = 11,\ c(E) = 1011,\ c(F) = 00$$

Der Huffman-Baum in aufrechter Darstellung ist in Abbildung 5 zu sehen. Die mittlere Codelänge ist

$$L = 0.12 \cdot 3 + 0.22 \cdot 2 + \ldots = 2.42\,.$$

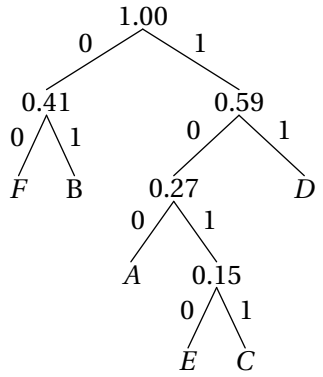


Abbildung 5: Huffman-Baum

1.2 Informationsgehalt

Wenn zwei Ereignisse d und e **unabhängig** sind (wie z.B. d = „die Sonne scheint“, $P(d) = 1/3$, und e = „4 gewürfelt“, $P(e) = 1/6$), dann ist die Wahrscheinlichkeit, dass d *und* e eintritt $P(d \wedge e) = P(d) \cdot P(e)$ (wie z.B. $P(d \wedge e) = 1/3 \cdot 1/6 = 1/18$).

Der Logarithmus zur Basis B einer Zahl X ist jene Zahl x, mit der man die Basis potenzieren muss, um X zu erhalten.

$$\log_B(X) = x \quad :\Leftrightarrow \quad B^x = X$$

Es gelten folgende Regeln:

- $\log_B XY = \log_B X + \log_B Y$ (da $B^x B^y = B^{x+y}$).
- $\log_B(X/Y) = \log_B X - \log_B Y$ (da $B^x / B^y = B^{x-y}$).
- $\log_B 1 = 0$.
- $\log_B 1/X = -\log_B X$ (da $1/B^x = B^{-x}$).
- $\log_B B = 1$ (da $B^1 = B$).
- $\ln X = \log_e X$. (Oft steht auch log für ln.)
- $\log_B X = \ln X / \ln B$ (generell $\log_B X = \log_A X / \log_A B$).

Wenn ein Codewort übertragen wird, ist das eine **Nachricht**. Eine Nachricht A (aus einer Nachrichtenquelle) hat eine gewisse Wahrscheinlichkeit $P(A)$. Eine

Nachricht hat auch einen **Informationsgehalt** $I(A)$, der von der Wahrscheinlichkeit abhängt. Denn Nachrichten, die mit großer Wahrscheinlichkeit, also sehr häufig, auftreten, haben keinen großen Informationsgehalt. Z.B. „morgen geht die Sonne auf“ hat keinen Informationsgehalt (Neuigkeitswert), weil das ohnehin mit 100%-iger Wahrscheinlichkeit eintritt.

Der Informationsgehalt $I(A)$ sollte folgende Bedingungen erfüllen:

1. $I(A) \geq 0$.

   Negative Informationsgehalte sollten nicht möglich sein, denn nach Erhalt einer Nachricht weniger zu wissen als vorher, macht keinen Sinn.

2. Wenn $P(A) = 1$, dann $I(A) = 0$.

   Eine Nachricht, die zu 100% vorhersehbar ist, stellt keine zusätzliche Information dar.

3. $I(AB) = I(A) + I(B)$, sofern die Wahrscheinlichkeiten aufeinanderfolgender Symbole unabhängig sind.

   Die Übertragung zweier Symbole sollte den Informationsgehalt des ersten plus des zweiten Symbols haben. Man beachte, dass die Wahrscheinlichkeit für das erste Symbol $P(A)$ ist und für das zweite Symbol $P(B)$. Die Wahrscheinlichkeit, dass beides eintritt (1. Symbol ist A, 2. B), ist dann $P(AB) = P(A) \cdot P(B)$. Also: Wenn sich Wahrscheinlichkeiten multiplizieren, addieren sich die Informationsgehalte.

4. Wenn $P(A) = 1/2$, dann $I(A) = 1$.

   Ein Münzwurf sollte den Informationsgehalt 1 haben.

Die Funktion, die diese Bedingungen erfüllt, ist:

$$I(A) := -\log_2 P(A)$$

Beweis

1. Für $0 \leq x \leq 1$ ist $\log x \leq 0$. Da $0 \leq P(A) \leq 1$ gilt, ist $-\log_2 P(A) \geq 0$.

2. $\log 1 = 0$.

3. $I(AB) = -\log_2 P(AB) = -\log_2 (P(A) \cdot P(B)) = -\log_2 P(A) - \log_2 P(B) = I(A) + I(B)$.

4. $-\log_2 \frac{1}{2} = \log_2 2 = 1$.

Die Einheit des Informationsgehalts ist ebenso „Bit“. Genaugenommen sollte man aber Codelänge 1 Bit und Informationsgehalt 1 Bit unterscheiden, denn ein Codewort der Länge 1 hat nur dann auch Informationsgehalt 1, wenn die Wahrscheinlichkeit für das Code-Bit 50% ist ($P(0) = P(1) = 1/2$), denn dann ist ($I(0) = I(1) = -\log_2 \frac{1}{2} = 1$. Ansonsten kann ein Bit Code auch einen Informationsgehalt kleiner oder größer als 1 Bit haben.

Ähnlich der Codelänge $l(c(A))$ eines Symbols A ordnet auch der Informationsgehalt $I(A)$ einem Symbol einen Wert zu. In Analogie zur mittleren Codelänge gibt es auch einen **mittleren Informationsgehalt** H, auch **Entropie** genannt, definiert als Erwartungswert des Informationsgehalts.

$$H = \mathrm{E}(I) = \sum_{k=1}^{n} P(A_k) I(A_k) = -\sum_{k=1}^{n} P(A_k) \log_2(P(A_k))$$

Für die gleichen Wahrscheinlichkeiten wie oben:

$$P(A) = 0.12,\ P(B) = 0.22,\ P(C) = 0.06, P(D) = 0.32,\ P(E) = 0.09,\ P(F) = 0.19$$

ergibt sich:

$$H = -0.12 \log_2 0.12 - 0.22 \log_2 0.22 - \ldots \approx 2.385$$

Die mittlere Codelänge des Huffman-Codes war zum Vergleich $L = 2.42$.

Man sieht, dass die mittlere Codelänge knapp größer als die Entropie ist. Dass das immer so ist, zeigt das folgende Theorem (nach Claude Shannon):

$$H \leq L$$

Die Differenz zwischen L und H ist die **Redundanz** R. Entropie und Redundanz kann man bezüglich L normieren (durch L dividieren) und erhält die **Codeeffizienz** η (Eta) und die **relative Redundanz** r.

$$R := L - H, \qquad \eta := \frac{H}{L}, \qquad r := \frac{R}{L} = \frac{L-H}{L} = 1 - \frac{H}{L} = 1 - \eta$$

Die Werte bewegen sich in den Bereichen $R \geq 0$, $0 \leq r \leq 1$, $0 \leq \eta \leq 1$. Erwünscht sind natürlich kleine R bzw. r und ein großes η.

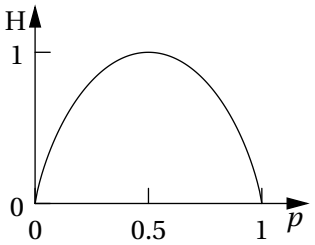


Abbildung 6: Entropie für $P(A) = p$, $P(B) = 1 - p$.

Das Beispiel von oben weitergeführt:

$$R \approx 2.385 - 2.42 \approx 0.0349, \qquad \eta = \frac{2.385}{2.42} \approx 0.986, \qquad r = \frac{0.0349}{2.42} \approx 0.0144$$

Die Redundanz ist für den optimalen Huffman-Code also relativ klein. Man kann sogar zeigen, dass für diesen $H \leq L \leq H + 1$ gilt.

Wenn die Wahrscheinlichkeiten passende Werte haben ($P(A_k) = 2^{-n_k}$), kann die Redundanz 0 werden.

$$P(A) = 1/2,\ P(B) = 1/4,\ P(C) = 1/4, \qquad c(A) = 0,\ c(B) = 10,\ c(C) = 11$$

$$\begin{aligned} L &= \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{4}2 = 1.5, \\ H &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4} \\ &= -\frac{1}{2}(-1) - \frac{1}{4}(-2) - \frac{1}{2}(-2) = 1.5, \\ R &= 1.5 - 1.5 = 0, \qquad r = 0, \qquad \eta = 1 \end{aligned}$$

Für nur zwei Symbole A, B ist die Nachrichtenquelle durch eine Wahrscheinlichkeit $P(A) = p$ charakterisiert, $P(B) = 1 - p$. Abhängig von p ergibt sich die Entropie wie in Abbildung 6. Man sieht, dass das Maximum bei $p = 0.5$ liegt. Generell ist die Entropie maximal, wenn alle Symbole die gleiche Wahrscheinlichkeit $1/n$ haben.

Man kann die Wahrscheinlichkeiten $P(A_k)$ durch die relative Häufigkeit annähern. Diese kann aus den absoluten Häufigkeiten aus der zu codierenden Sequenz selbst berechnet werden.

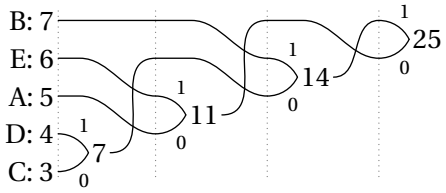


Abbildung 7: Huffman-Algorithmus mit absoluten Häufigkeiten

Zu codierende Sequenz: BACEABAEBBCBDADBEDDEAECEB
Absolute Häufigkeiten: $H(A) = 5$, $H(B) = 7$, $H(C) = 3$, $H(D) = 4$, $H(E) = 6$.
Den Huffman-Algorithmus kann man auch mit absoluten Häufigkeiten ausführen, siehe Abbildung 7. Man beachte übrigens, dass man hier zwei Möglichkeiten hat, 7 einzuordnen. Auf diese Weise können andere Codes mit sogar anderen Codelängen entstehen. Die mittlere Codelänge ist jedoch immer gleich.
Codierung: $c(A) = 00$, $c(B) = 11$, $c(C) = 100$, $c(D) = 101$, $c(E) = 01$.
Wahrscheinlichkeiten/relative Häufigkeiten: $P(A) \approx h(A) = H(A)/25 = 5/25 = 0.2$, $P(B) \approx 0.28$, $P(C) \approx 0.12$, $P(D) \approx 0.16$, $P(E) \approx 0.24$.
$H \approx 2.263$, $L = 2.28$, $R \approx 0.017$, $r \approx 0.75\%$, $\eta \approx 99.2\%$.

Es gibt auch die Möglichkeit, zwei oder mehrere Symbole zusammenzufassen. Dadurch wird die Anzahl der Symbole erhöht, was es dem Huffman-Code erlaubt, sich besser an die Symbolwahrscheinlichkeiten anzupassen. Das verringert die Redundanz. Wenn k Symbole zusammengefasst werden, gilt die mittlere Codelänge L_k für je k Symbole. Umgelegt auf Einzelsymbole ist dann $L_1 = L_k/k$. Und L_1 ist meist kleiner als das ursprüngliche L.

$P(A) = 0.2$, $P(B) = 0.8$, $H = -0.2\log_2 0.2 - 0.8\log_2 0.8 \approx 0.722$.
Einzige Möglichkeit: $c(A) = 0$, $c(B) = 1$, $L = 1$, $R = r \approx 0.278$.
Je zwei Symbole zu einem zusammenfassen: AA, AB, BA, BB.
$P(AA) = P(A)P(A) = 0.2^2 = 0.04$, $P(AB) = 0.2 \cdot 0.8 = 16$,
$P(BA) = 0.16$, $P(BB) = 0.64$.
Huffman-Code: $c_2(AA) = 010$, $c_2(AB) = 011$, $c_2(BA) = 00$, $c_2(BB) = 1$.
$L_2 = 0.04 \cdot 3 + 0.16 \cdot 3 + 0.16 \cdot 2 + 0.64 \cdot 1 = 1.56$.
Die Codelänge pro Einzelsymbol ist dann die Hälfte: $L_1 = L_2/2 = 0.78$.
$R_1 = L_1 - H \approx 0.78 - 0.722 \approx 0.058$, $r_1 = 0.075 (\ll r \approx 0.278)$.

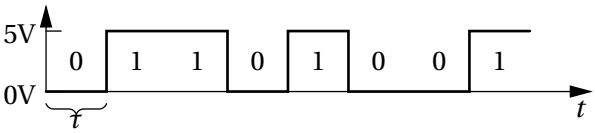


Abbildung 8: Simple Leitungscodierung

1.3 Leitungscodierung

Um Bitstreams über einen Kanal (über eine Leitung) zu übertragen, muss man die Folge von Bits in ein Analogsignal verwandeln. Im einfachsten Fall wird jedes Bit für eine bestimme Zeit τ als konstantes Signal dargestellt, z.B. als Pegel 5 V für 1 und 0 V für 0. Siehe Abbildung 8.

Die **Bandbreite** B einer Leitungscodierung ist die Differenz zwischen höchster Frequenz f_H und kleinster Frequenz f_L.

$$B = f_H - f_L$$

Die höchste Frequenz erreicht man in diesem Fall, indem man abwechselnd 0 und 1 codiert. Dadurch ergibt sich eine Periode von 2τ, und $f_H = \frac{1}{2\tau}$. Die niedrigste Frequenz ergibt sich, wenn alle Bits 1 sind. Dann ist die Periode unendlich und $f_L = 0$. Daher ist

$$B = f_H - f_L = \frac{1}{2\tau} - 0 = \frac{1}{2\tau}\,.$$

Also: Je höher die Bandbreite, desto kürzer ist τ und desto mehr Bits können pro Sekunde übertragen werden.

Nachteile der simplen Leitungscodierung sind:

1. Der Takt (also τ und die Synchronisierung mit dem Beginn der Bits) kann nicht immer aus dem Signal abgeleitet werden. Wenn z.B. sehr viele 1-Bits hintereindander kommen, gibt es keine Signalflanke (Wechsel von 0 auf 1 oder umgekehrt).

2. Es gibt einen **Gleichanteil**, also einen gemittelten Signalwert ungleich 0. Dieser Gleichanteil geht in vielen Übertragungskanälen verloren, was die Decodierung erschwert.

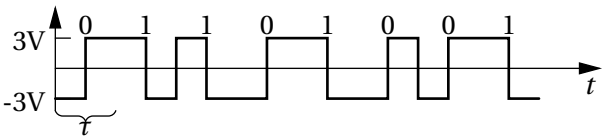


Abbildung 9: Manchester-Code

Bei einem Bitstrom von 0110100111 und Signalpegeln von 0 V und 5 V wäre der Gleichanteil

$$\frac{1}{10}(6 \cdot 5\,\mathrm{V} + 4 \cdot 0\,\mathrm{V}) = 3\,\mathrm{V}.$$

Die Lösung für diese Probleme ist z.B. der **Manchester-Code**. Dabei wird 0 als steigende Flanke von einem negativen Pegel (z.B. −3 V) auf einen positiven Pegel (z.B. +3 V) dargestellt, und 1 als fallende Flanke. Siehe Abbildung 9.

Die höchste Signalfrequenz wird hier z.B. bei einer Abfolge mehrerer 1-en erzeugt, nämlich eine Periode von τ und daher $f_H = \frac{1}{\tau}$. Die niedrigste Frequenz ergibt sich, wenn sich 0 und 1 abwechseln. Dann ist die Periode 2τ und $f_L = \frac{1}{2\tau}$. Die Bandbreite bleibt allerdings gleich:

$$B = f_H - f_L = \frac{1}{\tau} - \frac{1}{2\tau} = \frac{1}{2\tau}.$$

Die **Basisband-Bandbreite**, bei der $f_L = 0$ angenommen wird, ist allerdings doppelt so groß wie bei der simplen Leitungscodierung.

Da jedes codierte Bit zur Hälfte aus $-a$V und $+a$V besteht, ist der Gleichanteil immer 0.

Bei einem Bitstrom von 0110100111 und Signalpegeln von −3 V und +3 V wäre der Gleichanteil

$$\frac{1}{10}\left(\frac{6}{2} \cdot 3\,\mathrm{V} + \frac{6}{2} \cdot (-3)\,\mathrm{V} + \frac{4}{2} \cdot (-3)\,\mathrm{V} + \frac{4}{2} \cdot 3\,\mathrm{V}\right) = 0\,\mathrm{V}.$$

2 Zahlendarstellung

Um in Hard- und Software Zahlen darzustellen, müssen Zahlen als Code abgebildet werden. Die Zeichen des Alphabets, aus dem die Codewörter bestehen, sind als Ziffern zu interpretieren. Meist sind das nur 0 und 1. Damit mit solchen Codes sinnvoll gerechnet werden kann, braucht man ein Zahlensystem, das wie das

Dezimalsystem funktioniert, aber eine andere Anzahl von Ziffern benutzt. Ein solches Zahlensystem ist das **Stellenwertsystem**, oder die **polyadische Darstellung**.

2.1 Polyadische Darstellung

Die polyadische Darstellung einer Zahl $a \geq 0$ zur **Basis** B ist gegeben durch:

$$a = \sum_{i=0}^{n-1} a_i B^i = a_0 + a_1 B + a_2 B^2 + \ldots + a_{n-1} B^{n-1}, \qquad a < B^n, \quad 0 \leq a_i < B$$

Die a_i sind die B Ziffern von 0 bis $B-1$. Sie werden zusammen als Codewort geschrieben:

$$a = (a_{n-1} \cdots a_2 a_1 a_0)_B$$

1234 zur Basis $B = 10$: Es ist $n = 4$, da $1234 < B^n = 10^4 = 10000$.

$$\begin{aligned} 1234 &= a_0 + a_1 \cdot B + a_2 \cdot B^2 + a_3 \cdot B^3 \\ &= 4 + 3 \cdot 10 + 2 \cdot 10^2 + 1 \cdot 10^3 \\ &= 1234_B = 1234_{10} \end{aligned}$$

Nun zur Basis $B = 4$: $n = 6$, da $1234 < B^n = 4^6 = 4096$.

$$\begin{aligned} 1234 &= a_0 + a_1 \cdot B + a_2 \cdot B^2 + a_3 \cdot B^3 + a_4 \cdot B^4 + a_5 \cdot B^5 \\ &= 2 + 0 \cdot 4 + 1 \cdot 4^2 + 3 \cdot 4^3 + 0 \cdot 4^4 + 1 \cdot 4^5 \\ &= 103102_B = 103102_4 \end{aligned}$$

Ein **Beweis durch Induktion** beweist eine Aussage $P(n)$ für alle n, indem zuerst $P(1)$ (oder $P(0)$) (**Induktionsanfang**) bewiesen wird, und dann gezeigt wird, dass aus $P(n)$ (**Induktionsvoraussetzung**) auch $P(n+1)$ folgt (**Induktionsschluss**). Dadurch gilt dann automatisch auch $P(2)$, $P(3)$, $P(4)$, ...

Beispiel: $P(n) :\Leftrightarrow \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ soll für alle n gezeigt werden.

Induktionsanfang: $P(1) \Leftrightarrow 0 + 1 = 1 = \frac{1 \cdot (1+1)}{2}$. ✓

Induktionsschluss: $P(n+1) \Leftrightarrow \sum_{k=0}^{n+1} k = n + 1 + \sum_{k=0}^{n} k \overset{P(n)}{=\!=} n + 1 + \frac{n(n+1)}{2} = \frac{2(n+1)+n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$. ✓

An der Stelle $\overset{P(n)}{=\!=}$ wird die Induktionsvoraussetzung angewendet.

Die polyadische Darstellung ist eindeutig (bis auf führende 0-en) und immer möglich.

Beweis Induktionsanfang: Für $n = 1$ gilt $a < B^1$ und daher $a = (a_0)_B = a_0 \cdot B^0$ für $0 \le a = a_0 < B$. ✓
Induktionsschluss: Es wird angenommen, dass die Darstellung für $a < B^n$ möglich und eindeutig ist. Für $a < B^{n+1}$ ist $\lfloor \frac{a}{B} \rfloor < B^n$ und kann als $\lfloor \frac{a}{B} \rfloor = (a_n \cdots a_1)_B$ dargestellt werden (nur etwas anders nummeriert). Für die Abrundung gilt $0 \le \frac{a}{B} - \lfloor \frac{a}{B} \rfloor < 1$. Mit B multipliziert, bekommt man $0 \le a - \lfloor \frac{a}{B} \rfloor B < B$. Wir wählen $a_0 := a - \lfloor \frac{a}{B} \rfloor B$, sodass $0 \le a_0 < B$ gilt. Umgeformt ist auch

$$a = \left\lfloor \frac{a}{B} \right\rfloor B + a_0 \overset{\text{Ind.-Vo.}}{=\!=\!=} \left(\sum_{i=0}^{n} a_{i+1} B^i \right) B + a_0 = \sum_{i=0}^{n} a_{i+1} B^{i+1} + a_0 = \sum_{i=0}^{n+1} a_i B^i \,. \checkmark$$

Wir wollen nun eine Methode finden, mit der man möglichst einfach die Ziffern a_i ermitteln kann. In Anlehnung an obigen Beweis ergibt sich die Methode der **sukzessiven Division**. Dazu definieren wir die Division mit Rest:

$$a / B = q \,\mathrm{R}\, r \quad :\Leftrightarrow \quad a = qB + r,\ 0 \le r < B$$

$q = \lfloor \frac{a}{B} \rfloor$ ist der abgerundete Quotient, $r = a - qB$ ist der Divisionsrest. Der Algorithmus ist dann:

$i = 0$, $c_0 = a$
Wiederhole bis $c_i = 0$
  $c_{i+1} \,\mathrm{R}\, a_i = c_i / B$
  $i = i + 1$

1234 zur Basis $B = 10$:

i	c_i	$/10 =$	$c_{i+1} \,\mathrm{R}\, a_i$
0	1234	$/10 =$	123 R 4
1	123	$/10 =$	12 R 3
2	12	$/10 =$	1 R 2
3	1	$/10 =$	0 R 1
4	0	$/10 =$	

$a = 1234_{10}$

Zur Basis $B = 4$:

c_i	$c_{i+1}\,\mathrm{R}\,a_i$
1234	308 R 2
308	77 R 0
77	19 R 1
19	4 R 3
4	1 R 0
1	0 R 1
0	

$a = 103102_4$

Zur Basis $B = 16$, Ziffern $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F$:

c_i	a_i
1234	2
77	$13 = D$
4	4
0	

$a = 4D2_{16}$

Wichtige Basen sind:

B	Name
2	binär
8	oktal
10	dezimal
16	hexadezimal

Wenn von zwei Basen eine eine Potenz der anderen ist, gibt es eine einfachere Möglichkeit, eine Darstellung in die andere umzurechnen. Sei $C = B^m$. Dann ist für $j = i\,m + k$ mit $0 \le k < m$ (Quotient i und Rest k):

$$B^j = B^{im+k} = B^{im} B^k = C^i B^k,$$

$$a = \sum_{i=0}^{n} a_i C^i = \sum_{i=0}^{n} (a'_{im} + a'_{im+1} B + a'_{im+2} B^2 + \ldots) C^i = \sum_{j=0}^{nm-1} a'_j B^j.$$

Die a_i können also einzeln in m Ziffern a'_{im+k} umgewandelt werden.

$$1234 = 4D2_{16} = \underbrace{0100}_{4}\,\underbrace{1101}_{D}\,\underbrace{0010}_{2}{}_2 = \underbrace{010}_{2}\,\underbrace{011}_{3}\,\underbrace{010}_{2}\,\underbrace{010}_{2}{}_2 = 2322_8$$

Die **Addition** von nicht-negativen Zahlen in polyadischer Darstellung funktioniert ganz gleich wie im Dezimalsystem.

Addend	5234_6	11001110_2
Addend	3153_6	11011101_2
Übertrag	1011	11011100
Summe	12431_6	110101011_2

Z.B. ist $4_6 + 3_6 = 7 = 11_6$ $(1 \cdot 6 + 1)$, und daher 1 Übertrag.

2.2 Negative Zahlen

Negative Zahlen könnte man naiv darstellen, indem man das höchstwertige Bit als Vorzeichen verwendet. Zum Beispiel:

$$3 = 0011_2 \quad \Rightarrow \quad -3 = 1011_2$$

Das Problem dabei ist, dass man Addition in Abhängigkeit des Vorzeichens als Subtraktion ausführen muss. Außerdem muss man die Zahlen dann so ordnen, dass man die kleinere von der größeren subtrahiert. Zum Beispiel:

$$3 + (-5) = 0011_2 + 1101_2 = 0(011 - 101) = 1(101 - 011) = 1010_2 = -2$$

Um die Berechnungen für ein Rechenwerk einfacher zu machen, hat man bessere Darstellungen gefunden.

Die **Modulo-Arithmetik** setzt alle Zahlen gleich, für die der Divisionsrest bei Division durch eine positive Zahl m gleich ist.

$$a = b \pmod{m} \quad :\Leftrightarrow \quad a - m \cdot \lfloor a/m \rfloor = b - m \cdot \lfloor b/m \rfloor$$

Beispiel:

$$12 = 7 = 2 = -3 = -8 \pmod{5}$$

Die Darstellung negativer Zahlen geht zurück auf eine Methode, die Subtraktion durch Addition zu implementieren. Dabei nutzt man aus, dass durch Abschneiden aller Stellen a_k für $k > n$ eine Modulo-Arithmetik zum Divisor 2^n entsteht. Zum Beispiel:

$$10000_2 = 0000_2 \pmod{2^4}, \qquad 111111_2 = 1111_2 \pmod{2^4}$$

Da nun für $s = 2^n - 1$ gilt, dass $s + 1 = 2^n = 0 \pmod{2^n}$, und daher auch $a + s + 1 =$

$a \pmod{2^n}$ kann man die Subtraktion durch Addition ausführen.

$$x - y = x - y + s + 1 = x + \underbrace{\underbrace{s - y}_{=:K_1(y)} + 1}_{=:K_2(y)} \pmod{2^n}$$

Man nennt $s - y$ das **Einerkomplement** K_1 und $s - y + 1$ das **Zweierkomplement** K_2. Diese Komplemente dienen als Darstellung negativer Zahlen. Das Einerkomplement erhält man dabei einfach durch Invertieren aller Stellen, beim Zweierkomplement muss man noch 1 addieren.

$$K_1(0010) = 1101, \qquad K_2(0010) = K_1(0010) + 1 = 1101 + 1 = 1110$$

Das Zweierkomplement kann man auch ermitteln, indem man die Bits von rechts beginnend kopiert bis inklusive des ersten 1-Bits, und alle Bits links davon invertiert.

$$K_2(\underbrace{0110}_{\text{invert.}}\underbrace{1000}_{\text{kop.}}) = 10011000$$

Beispiel für Subtraktion durch Addition $\pmod{16}$:

$+5$	0101	$\mapsto$	$+5$	0101	$\mapsto$	$+5$	0101	$\mapsto$	$+5$	0101
-3	0011		$+s$	1111		$+K_1$	1100		$+K_2$	1101
			-3	0011		$+1$	1		$=2$	0010
			$+1$	1		$=2$	0010			

Das Übertragsbit mit der Wertigkeit $2^4 = 16$ wird wegen $\pmod{16}$ abgeschnitten.

Den Wert einer Zahl in einer Komplement-Darstellung erhält man, indem man das höchstwertige (linke) Bit nicht mit Wertigkeit 2^{n-1} wie üblich belegt, sondern mit $-(2^{n-1}-1)$ für das Einerkomplement bzw. mit -2^{n-1} für das Zweierkomplement.

$$\begin{aligned} K_1: &\quad (a_{n-1}\dots a_0)_2 = a_{n-1}(-(2^{n-1}-1)) + \textstyle\sum_{k=0}^{n-2} a_k 2^k \\ K_2: &\quad (a_{n-1}\dots a_0)_2 = a_{n-1}(-2^{n-1}) + \textstyle\sum_{k=0}^{n-2} a_k 2^k \end{aligned}$$

$$\begin{aligned} K_1: &\quad 1101 = 1\cdot(-(2^{n-1}-1)) + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = -7 + 4 + 1 = -2 \\ K_2: &\quad 1101 = 1\cdot(-2^{n-1}) + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = -8 + 4 + 1 = -3 \end{aligned}$$

Da das Komplement eines Komplements wieder die ursprüngliche Zahl ergibt, kann man die Werte auch darüber ermitteln.

$$K_1: \quad 1101 = -K_1(1101) = -0010 = -1 \cdot 2^1 = -2$$
$$K_2: \quad 1101 = -K_2(1101) = -0011 = -(1 \cdot 2^1 + 1 \cdot 2^0) = -(2+1) = -3$$

Als negative Zahlen gelten nun alle, bei denen das höchstwertige Bit gleich 1 ist.

Alle darstellbaren Zahlen für $n = 4$:

	K_1	K_2
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
0	0000	0000

	K_1	K_2
−0	1111	
−1	1110	1111
−2	1101	1110
−3	1100	1101
−4	1011	1100
−5	1010	1011
−6	1001	1010
−7	1000	1001
−8		1000

Der darstellbare Bereich ist also für das Einerkomplement $-(2^{n-1}-1)\ldots 2^{n-1}-1$ und für das Zweierkomplement $-2^{n-1}\ldots 2^{n-1}-1$. Das Zweierkomplement kann also eine Zahl mehr darstellen als das Einerkomplement, weil letzteres zwei Darstellungen für 0 hat (+0 und −0).

Zur Umwandlung von negativen Dezimalzahlen in die Komplement-Darstellung gibt es nun mehrere Methoden:

Komplementbildung $-(a)_{10} \mapsto K_i((a)_2)$

$$-4_{10} \mapsto K_1(0100_2) = 1011 \qquad -4_{10} \mapsto K_2(0100_2) = 1100$$

Umwandlung im Dezimalbereich

$K_1: -a_{10} \mapsto (2^n - 1 - a)_2, \quad K_2: -a_{10} \mapsto (2^n - a)_2$

$$K_1: \quad -4_{10} \mapsto 15 - 4 = 11_{10} = 1011_2$$
$$K_2: \quad -4_{10} \mapsto 16 - 4 = 12_{10} = 1100_2$$

Sukzessive Division Für das Zweierkomplement ist es möglich, auch negative Zahlen mittels Division mit Rest zu dividieren (z.B. $-5/2 = -3\,\mathrm{R}\,1$, da $-5 = -3 \cdot 2 + 1$).

$$
\begin{array}{rll}
-5 & /2 = & -3\,\mathrm{R}\,1 \\
-3 & /2 = & -2\,\mathrm{R}\,1 \\
-2 & /2 = & -1\,\mathrm{R}\,0 \\
-1 & /2 = & -1\,\mathrm{R}\,1 \qquad a \mapsto 111011_2 \\
-1 & /2 = & -1\,\mathrm{R}\,1 \\
-1 & /2 = & -1\,\mathrm{R}\,1 \\
 & \dots &
\end{array}
$$

Bei der Addition von potentiell negativen Zahlen in Komplementdarstellung gibt es noch zwei Probleme. Das erste ist, dass im Einerkomplement möglicherweise (aber nicht immer) eine 1 addiert werden muss. Es stellt sich heraus, dass man das am (üblicherweise abgeschnittenen) Übertragsbit an der Stelle n erkennt. Dieses entsteht (ohne +1), wenn die Summe $\geq s$ wird.

Rechnungsart	Einerkomplement	+1?	Übertrag an Stelle n
pos + pos = pos	$a+b=c$	nein	$a+b<s$
pos + neg = pos	$a+b+s+1=c$	ja	$a+b+s\geq s$
pos + neg = neg	$a+b+s+\not{1}=c+s+\not{1}$	nein	$a+b+s<s$
neg + neg = neg	$a+s+1+b+s+\not{1}=c+s+\not{1}$	ja	$a+s+b+s\geq s$

Daher muss man diese Additionen erst ohne 1 durchführen, und, wenn sich ein Übertragsbit ergibt, nochmals mit +1 durchführen.

+3	0011	+5	0101	+5	0101
−5	1010	−3	1100	−3	1100
−2	**0**1101	?	**1**0001	+Ü	1
	kein Ü.		Übertrag	2	0010

Im Zweierkomplement muss man *nicht* auf einen Übertrag achten, da kein +1 notwendig ist.

Das zweite Problem ist die Erkennung eines **Überlaufs**. Dazu benötigt man drei Erkenntnisse:

- Eine Darstellung mit n Bits kann man in eine mit $n+1$ Bits umwandeln, indem man einfach das Vorzeichenbit verdoppelt.

  Beweis Für $a_n = a_{n-1}$ vergleichen wir die Werte der Darstellungen mit n und $n+1$ Bits:
  K_1: $a_{n-1}\underline{(-(2^{n-1}-1))} + \sum_{k=0}^{n-2} a_k 2^k = a_n\underline{(-(2^n-1))} + a_{n-1}\underline{(2^{n-1})} + \sum_{k=0}^{n-2} a_k 2^k$.

K_2: $a_{n-1}\underline{(-2^{n-1})} + \sum_{k=0}^{n-2} a_k 2^k = a_n\underline{(-2^n)} + a_{n-1}\underline{(2^{n-1})} + \sum_{k=0}^{n-2} a_k 2^k$.
Die unterstrichenen Terme sind jeweils gleich.

- Wenn bei einer Addition in n Bits die Summe den Wertebereich überschreitet (im positiven oder negativen), dann überschreitet diese Summe den Wertebereich in $n+1$ Bits nicht, da die Summe z.B. für K_2 im Bereich

$$-2^{n-1} - 2^{n-1} \ldots 2^{n-1} - 1 + 2^{n-1} - 1 = -2^n \ldots 2^n - 2$$

  ist, was in der $n+1$-Bit-Darstellung darstellbar ist.

- Bei einem Überlauf ist das Vorzeichenbit der Summe falsch, da für $a + b > 2^{n-1} - 1$ die Wertigkeit $+2^{n-1}$ Teil der Binärdarstellung sein müsste. Für den negativen Überlauf gilt Ähnliches.

Ein Überlauf kann also erkannt werden, indem in der $n+1$-Bit-Darstellung die Bits a_n und a_{n-1} unterschiedlich sind. Dazu müssen bei der Addition die **Vorzeichenbits verdoppelt** werden.

Für das Einerkomplement brauchen wir zwei Schritte, einen für den Übertrag (ohne Vorzeichenverdoppelung), einen für den Überlauf (mit Vorzeichenverdoppelung).

−12	11110011	
−77	10110010	
−89	**1**10100101	Übertrag

−12	1|11110011	
−77	1|10110010	
	1	
−89	1**1**|10100110	kein Überlauf

Für das Zweierkomplement braucht es nur einen Schritt (mit Vorzeichenverdoppelung).

−111	1|10010001	
−66	1|10111110	
−177	1**1**|01001111	Überlauf

2.3 Multiplikation, Division

Die **Multiplikation** funktioniert ganz ähnlich wie im Dezimalsystem. Der Vorteil im Binärsystem ist, dass die Multiplikation des Multiplikanden mit einer Multiplikatorziffer entweder gleich dem Multiplikanden ist, wenn die Ziffer 1 ist, oder

komplett 0, wenn die Ziffer 0 ist, was recht einfach zu implementieren ist. Wenn Multiplikand und Multiplikator jeweils m bzw. n Bits haben, hat das Produkt $m+n$ Bits.

$$
\begin{array}{r}
1110\times0111 \\
\hline
0000 \\
1110 \\
1110 \\
1110 \\
0122100 \\
\hline
01100010
\end{array}
\qquad 14\times 7 = 98
$$

Man sieht, dass hier mehrere Zahlen addiert werden müssen, und dass dabei die Überträge größer als 1 werden. In Rechenwerken gibt es dazu Verfahren, die solche Überträge vermeiden, wie z.B. immer jeweils 3 Bits zu addieren, und als Zwischenergebnis 2 Bits zu produzieren.

Auch die **Division** funktioniert ganz ähnlich wie im Dezimalsystem. Auch hier gibt es den Vorteil, dass die Quotientenziffern nur 0 und 1 sein können, je nachdem, ob die Divisorbits kleiner oder gleich den aktuellen Dividendenbits sind. Die Division involviert also n Vergleiche zweier Zahlen, was jeweils einer Subtraktion entspricht. Das invertierte Vorzeichenbit ist dann ein Quotientenbit.

$$
\begin{array}{r}
110110:101=1010 \\
-101 \\
\hline
111 \\
-101 \\
\hline
100\mathrm{R}
\end{array}
\qquad 54/5 = 10\,\mathrm{R}\,4
$$

Die Division ist die aufwendigste Operation eines Rechenwerks. Der Grund ist, dass man für jedes Quotientenbit eine volle Subtraktion durchführen muss. Es gibt daher komplexe Methoden (z.B. SRT), um ein Quotientenbit aus ein paar führenden Bits zu schätzen und Schätzfehler später in der Berechnung zu korrigieren. Das führt u.a. zu ternären Quotientenziffern, d.h. es ist 1, 0 und -1 erlaubt.

Ein einfacher Spezialfall von Multiplikation und Division auf Binärzahlen ist der Multiplikator bzw. Divisor 2. Eine Multiplikation mit 2 führt dazu, dass jedes Bit an die Stelle mit der nächst-höheren Wertigkeit verschoben wird ($2^k \mapsto 2^{k+1}$), oder anders ausgedrückt: Es wird rechts eine 0 eingefügt, analog zur Multiplikation mit 10 in der Dezimaldarstellung. Gleicherweise entspricht eine Division

durch 2 der Verschiebung nach rechts. Es wird also rechts eine Stelle abgeschnitten, was einer Abrundung entspricht.

Diese Operationen werden üblicherweise als „left-shift“ bzw. „right-shift“ bezeichnet. Prozessoren besitzen dafür Maschinenbefehle SHL und SHR, oder Ähnliches. Eine Verschiebung um mehr als eine Stelle entspricht dabei der Multiplikation (oder Division) mit 2, 4, 8, usw.

$$13_{10} \cdot 2 = 1101_2 \cdot 2 = 11010_2\,, \qquad \lfloor 13_{10}/2 \rfloor = \lfloor 1101_2/2 \rfloor = 110_2 = 6_{10}$$

Eine Variante von shift-right berücksichtigt negative Zahlen (im Zweierkomplement). Dabei wird von links nicht immer 0 nachgeschoben, sondern das Vorzeichen dupliziert.

$$\lfloor -5/2 \rfloor = \lfloor 1011_2/2 \rfloor = 1101_2 = -3$$

2.4 Rationale Zahlen

Rationale Zahlen werden mit Stellen hinter dem Komma dargestellt. Bei der **Festkommadarstellung** verwendet man eine fixe Anzahl an Bits, wobei das Komma an einer vorgegebenen Stelle steht. Bei n Vorkommastellen und m Nachkommastellen zur Basis B stellt die Zahlendarstellung

$$(a_{n-1} \ldots a_1 a_0).(a_{-1} a_{-2} \ldots a_{-m})$$

die Zahl

$$a = \sum_{i=-m}^{n-1} a_k B^k = a_{-m} B^{-m} + \ldots + a_{-1} B^{-1} + a_0 B^0 + a_1 B^1 + \ldots + a_{n-1} B^{n-1}$$

dar. Wir schreiben das „Komma“ als „.“ statt „,“.

Für $n = 3$ und $m = 2$:

$$\underbrace{101}_{n}.\underbrace{11}_{m}{}_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 0 + 1 + 0.5 + 0.25 = 5.75_{10}$$

Zur Umwandlung kann man auch das Komma weglassen und die Zahl mit 2^{-m} multiplizieren.

$$101.11_2 = 10111_2 \cdot 2^{-2} = 23/4 = 5.75$$

Zur Darstellung negativer Zahlen kann man wie bei ganzen Zahlen die Komplementdarstellung verwenden. Das höchstwertige Bit hat im Zweierkomplement mit $n+1$ Vorkommastellen den Wert -2^n.

4 + 1 Vorkommastellen, 3 Nachkommastellen, Zweierkomplement:

$$\begin{aligned} 11010.010_2 &= -2^4 + 2^3 + 2^1 + 2^{-2} = -16 + 8 + 2 + 0.25 = -5.75 \\ &= -K_2(11010010) \cdot 2^{-3} = -101110_2/8 = -46/8 = -5.75 \end{aligned}$$

Zur Umwandlung vom Dezimalsystem in die Binärdarstellung betrachtet man die Vorkommastellen und die Nachkommastellen separat. Während die Vorkommastellen per sukzessiver Division ermittelt werden, wendet man bei den Nachkommastellen die **sukzessive Multiplikation** an. Die Idee ist, dass man bei einer Zahl zwischen 0 und 1

$$\lfloor 0.(a_{-1}a_{-2}a_{-3}\ldots) \cdot B \rfloor = \lfloor (a_{-1}).(a_{-2}a_{-3}\ldots) \rfloor \overset{0 \le c < 1}{=\!=\!=} \lfloor a_{-1} + c \rfloor = a_{-1}$$

durch Multiplikation mit der Basis die erste Stelle hinter dem Komma vor das Komma schiebt und durch Abrunden einfach ablesen kann. Auf diese Weise fährt man fort, bis der Nachkommaanteil c 0 wird.

$i = 1$, $c_0 = a - \lfloor a \rfloor$
Wiederhole bis $c_i = 0$:
$\quad a_{-i} + c_i = c_{i-1} \cdot B \qquad (a_{-i} \in \{0, \ldots, B-1\}, 0 \le c_i < 1)$
$\quad i = i + 1$

6.6875 ist in die Binärdarstellung umzuwandeln. Die Vorkommastellen wandelt man wie bekannt um: $\lfloor 6.6875 \rfloor = 6 = 110_2$. Die Nachkommastellen $c_0 = 6.6875 - \lfloor 6.6875 \rfloor = 6.6875 - 6 = 0.6875$ so:

i	c_{i-1}	$\cdot 2 =$	$\boldsymbol{a_{-i}} + c_i$
1	0.6875	$\cdot 2 =$	**1**.375
2	0.375	$\cdot 2 =$	**0**.75
3	0.75	$\cdot 2 =$	**1**.5
4	0.5	$\cdot 2 =$	**1**.0

Die Nachkommastellen werden von oben nach unten abgelesen. Das Ergebnis ist $110.\mathbf{1011}_2$

Eine endliche Anzahl an Nachkommastellen gibt es nur, wenn die Zahl durch ein B^{-m} teilbar ist (für m groß genug), also wenn sie mit B^m multipliziert ganzzahlig wird. Das gilt auch im Dezimalsystem, wo z.B. $1/3 = 0.3333\ldots = 0.\overline{3}$ unendlich viele Nachkommastellen hat. Die Nachkommaziffern werden dabei periodisch, was für die sukzessive Multiplikation bedeutet, dass sich der Algorithmus zu wiederholen beginnt. Das hängt auch von der Basis ab.

$$9/11 = 0.\overline{81}_{10}$$

$$\begin{array}{llll} 9/11 & \cdot 10 = & 90/11 = \mathbf{8} + 2/11 \\ 2/11 & \cdot 10 = & 20/11 = \mathbf{1} + 9/11 \\ 9/11 & & \textit{(Wiederholung)} \end{array}$$

$$9/11 = 0.\overline{1101000101}_2$$

$$\begin{array}{llll} 9/11 & \cdot 2 = & 18/11 = \mathbf{1} + 7/11 \\ 7/11 & \cdot 2 = & 14/11 = \mathbf{1} + 3/11 \\ 3/11 & \cdot 2 = & 6/11 = \mathbf{0} + 6/11 \\ 6/11 & \cdot 2 = & 12/11 = \mathbf{1} + 1/11 \\ 1/11 & \cdot 2 = & 2/11 = \mathbf{0} + 2/11 \\ 2/11 & \cdot 2 = & 4/11 = \mathbf{0} + 4/11 \\ 4/11 & \cdot 2 = & 8/11 = \mathbf{0} + 8/11 \\ 8/11 & \cdot 2 = & 16/11 = \mathbf{1} + 5/11 \\ 5/11 & \cdot 2 = & 10/11 = \mathbf{0} + 10/11 \\ 10/11 & \cdot 2 = & 20/11 = \mathbf{1} + 9/11 \\ 9/11 & & \textit{(Wiederholung)} \end{array}$$

$$4/5 = 0.8_{10}$$

$$4/5 \quad \cdot 10 = \quad 40/5 = \mathbf{8} + 0/5$$

$$4/5 = 0.8 = 0.\overline{1100}_2$$

$$\begin{array}{llll} 0.8 & \cdot 2 = & \mathbf{1}.6 \\ 0.6 & \cdot 2 = & \mathbf{1}.2 \\ 0.2 & \cdot 2 = & \mathbf{0}.4 \\ 0.4 & \cdot 2 = & \mathbf{0}.8 \\ 0.8 & & \textit{(Wiederholung)} \end{array}$$

Das letzte Beispiel zeigt, dass eine Zahl, die in der Dezimaldarstellung endlich viele Nachkommastellen hat, in der Binärdarstellung unendlich viele haben kann, also mit endlich vielen Nachkommastellen möglicherweise nicht fehlerfrei dargestellt werden kann. Beim Abschneiden von Stellen sollte man runden, d.h., wenn nach a_{-n} abgeschnitten wird, dann sollte aufgerundet werden, wenn $a_{-(n+1)} = 1$ ist, sonst abgerundet.

0.8 mit 8 Nachkommastellen ergibt 0.11001101_2. Die letzte Ziffer a_{-8} ist aufgerundet, weil $a_{-9} = 1$ ist. Zurückumgewandelt ergibt sich 0.80078125. (Mit Abrundung ergäbe sich 0.796875.)

Aufgrund dieser Problematik wird in Datenbanken meist nicht die Binärdarstellung für Zahlendarstellung verwendet, sondern so etwas wie BCD (Binary Coded

Decimal), wo Dezimalziffern mit je 4 Bit dargestellt werden. Das verhindert Rundungsprobleme z.B. in Buchhaltungsanwendungen.

Bei Rechenoperationen ist darauf zu achten, dass es keinen Überlauf gibt. Dazu kann man die Anzahl der Vorkommastellen dementsprechend erhöhen und die Nachkommastellen dementsprechend verringern. Das schränkt zwar die Genauigkeit der Berechnung ein, verhindert aber Überläufe. Im Folgenden bezeichnet $n.m$ eine Darstellung mit n Vor- und m Nachkommastellen. Es sei $n+m = n_1+m_1 = n_2+m_2$. Für die Operation $\square \in \{+, \times, /\}$ braucht $a \square b$ folgende Vor- und Nachkommastellen (eigentlich und gekürzt auf insgesamt $n+m$ Stellen).

a	$\square$	b	$a \square b$	$n+m$ Stellen
$n.m$	$+$	$n.m$	$(n+1).m$	$(n+1).(m-1)$
11.111	$+$	11.111	111.110	111.11
$n_1.m_1$	$\times$	$n_2.m_2$	$(n_1+n_2).(m_1+m_2)$	$(n_1+n_2).(m_1-n_2)$
11.111	$\times$	11.111	1111.000001	1111.0
$n_1.m_1$	$/$	$n_2.m_2$	$(n_1+m_2).\infty$	$(n_1+m_2).(m_1-m_2)$
11.111	$/$	00.001	11111.00	11111.
11.111	$/$	00.011	$01010.\overline{01}$	01010.

Falls die Anzahl der Nachkommastellen negativ wird, heißt das, dass auch vor dem Komma einige niederwertige Bits abgeschnitten werden. Z.B. wäre die Darstellung von 24 nach dem Schema $6.(-2)$ die Zahl $0110\_\_._2$, wobei _ das Fehlen einer Ziffer anzeigt.

Da im Voraus die Zahlen nicht bekannt sind, sondern nur deren Bereich, müssen Systeme (Hard- oder Software) immer auf das Maximum ausgelegt sein. Wenn die meisten Werte aber viel kleiner als das Maximum sind, führt das zu vielen „verschwendeten“ Bits bei den Vorkommastellen und schlechter Genauigkeit bei den Nachkommastellen.

Daher ist es von Vorteil, die Position des Kommas zur Laufzeit des Systems dynamisch verschieben zu können. Das macht man mit der **Gleitkommadarstellung**. Dazu verwendet man ein Schema, das an die wissenschaftliche Notation (z.B. $1.23\text{E}{-45} = 1.23 \cdot 10^{-45}$) angelehnt ist:

$$a = (-1)^v M \cdot B^C, \qquad M = 1.m, \qquad C = c - K$$

$$+12.5 = (-1)^0 1.1001_2 \cdot 2^3, \qquad m = 1001, \qquad 3 = 10 - 7 = 1010_2 - 7$$

- M ist die **Mantisse**, die den Wert der Zahlenbits repräsentiert (1.1001 = 1.5625). M hat (fast) immer nur 1 als Vorkommaziffer.
- m sind die Nachkommastellen der Mantisse (1001).
- C ist der Exponent, mit dem das Komma an die richtige Stelle verschoben wird (3).
- c ist die **Charakteristik**, eine nicht-negative Binärzahl, aus der der Exponent berechnet wird ($1010_2 = 10$).
- K ist der Bias-Wert, der den Wertebereich der Charakteristik verschiebt (Exzess-K-Code) (7).
- v ist das Vorzeichenbit, das das Vorzeichen der Zahl bestimmt, 0 für +, 1 für − (0).

Die Darstellung als Code ist

$$v \circ c \circ m \qquad (0\,1010\,10010000)\,.$$

Der am weitesten verbreitete Standard für Gleitkommazahlen ist IEEE-754 (Institute of Electrical and Electronics Engineers, „Ei-Trippl-Ih“). Dort gibt es folgende Definitionen:

#Bits	precision	Typ	m-Bits	c-Bits	K
32	single	`float`	23	8	127
64	double	`double`	52	11	1023
128	quadruple		112	15	16383

Ein Problem bei dieser Darstellung ist, dass die Zahl 0 nicht genau dargestellt werden kann, weil die Mantisse immer mit 1 beginnt. Daher gibt es noch einige Ausnahmen in diesen Standard-Gleitkommadarstellungen. Für single-precision sind das:

- c = 00000000, m = 00000000000000000000000: $a = 0$ (exakte Darstellung von 0)

- $c = 00000000$, $m \neq 0$: $C = -126$, $M = 0.m$ (denormalisierte Zahl, genauere Darstellung für Zahlen nahe 0)
- $c = 11111111 = 255$, $m = 00000000000000000000000$: $a = (-1)^v \infty$ (Darstellung von unendlich, z.B. für $1.0/0.0$)
- $c = 11111111 = 255$, $m \neq 0$: NaN (Not a Number) (ungültige Zahl, z.B. für $0/0$, $\sqrt{-1}$)

$1\,10000111\,10100000000000000000000 = (-1)^1 \cdot 1.101_2 \cdot 2^{10000111_2 - 127}$
$= -1 \cdot 1.625 \cdot 2^{135-127} = -416$

$1\,01111000\,10100000000000000000000 = (-1)^1 \cdot 1.101_2 \cdot 2^{01111000_2 - 127}$
$= -1 \cdot 1.6252^{120-127} = -0.012695313$

$1\,00000000\,10100000000000000000000 = (-1)^1 \cdot 0.101_2 \cdot 2^{-126}$
$= -1 \cdot 0.625 \cdot 2^{-126} \approx -7.34 \cdot 10^{-39}$

$1\,00000000\,00000000000000000000000 = -0$

$0\,11111111\,00000000000000000000000 = +\infty$

$0\,11111111\,10100000000000000000000 = \text{NaN}$

Zur Umwandlung von der Dezimaldarstellung in die Gleitkommadarstellung muss man zuerst die Zahl normieren, d.h. mit 2^{-C} multiplizieren, sodass $1 \leq a \cdot 2^{-C} < 2$ ist.

$13.375 = 1101.011_2 = 1.101011_2 \cdot 2^3 = (-1)^0 1.101011_2 \cdot 2^{130-127}$
$= (-1)^0 1.101011_2 \cdot 2^{10000010_2 - 127} = 0\,10000010\,10101100000000000000000$

Die größte positive darstellbare Zahl ist

$$0\,11111110\,11111111111111111111111 = (2 - 2^{-23}) 2^{254-127} \approx 3.4028 \cdot 10^{38}.$$

Die kleinste positive darstellbare Zahl ist

$$0\,00000000\,00000000000000000000001 = 2^{-23} \cdot 2^{-126} \approx 1.401 \cdot 10^{-45}$$

3 Logische Schaltungen

Computer verarbeiten Codes und Zahlen, indem sie aus Bits neue Bits erzeugen. All diese Operationen kann man auf Operationen zurückführen, die aus einem

oder zwei Input-Bits ein Output-Bit erzeugen. Das ist äquivalent zur **Aussagenlogik**, in der Aussagen die Wahrheitswerte „wahr“ oder „falsch“ zugeordnet werden und durch Junktoren (logische Verknüpfungen) verknüpft werden. Die Zuordnung ist $1 \mapsto$„wahr“, $0 \mapsto$„falsch“.

3.1 Aussagenlogik

Die Grundelemente der Aussagenlogik sind **Aussagenvariablen** $a, b, c, \ldots$, die die Wahrheitswerte 0 und 1 annehmen. Sie werden mittels **Junktoren** $\wedge, \vee, \rightarrow, \leftrightarrow, \ldots$ zu **Aussageformen** verknüpft.

Die Aussageform $\alpha := a \wedge b$ hat zwei Aussagenvariablen a und b, die durch den Junktor $\wedge$ verknüpft sind.

Jede Aussageform besitzt eine **Wahrheitstabelle**, die für jede mögliche Zuordnung von Wahrheitswerten zu den Aussagenvariablen auch der Aussageform einen Wahrheitswert zuordnet.

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Es gibt folgende Junktoren:

Name	geschrieben	gesprochen
Negation	$\bar{a}$, $\neg a$, a', `!a`	nicht a, a-nicht
Konjunktion	$a \wedge b$, $a \cdot b$, ab, `a&&b`	a und b
Disjunktion	$a \vee b$, $a + b$, `a||b`	a oder b
Subjunktion	$a \rightarrow b$	wenn a dann b, aus a folgt b
Bijunktion	$a \leftrightarrow b$	a genau dann wenn b
Exklusives Oder	$a \oplus b$, $a \dot{\vee} b$, $a \nleftrightarrow b$	a XOR b
NAND	$a \uparrow b$, $a \bar{\wedge} b$	a NAND b
NOR	$a \downarrow b$, $a \bar{\vee} b$	a NOR b

Die Junktoren werden über folgende Wahrheitstabellen definiert:

a	$\bar{a}$
0	1
1	0

a	b	$a \wedge b$	$a \vee b$	$a \to b$	$a \leftrightarrow b$	$a \oplus b$	$a \uparrow b$	$a \downarrow b$
0	0	0	0	1	1	0	1	1
0	1	0	1	1	0	1	1	0
1	0	0	1	0	0	1	1	0
1	1	1	1	1	1	0	0	0

Die Verknüpfung von Wahrheitstabellen wird nun folgendermaßen definiert. Wir schreiben einen Eintrag in der Wahrheitstabelle einer Aussageform α als

$$\alpha[a : x_a, b : x_b, \ldots],$$

wobei x_a der Wahrheitswert von a ist. So ist z.B. $(a \wedge b)[a : 0, b : 1] = 0$. Einzelne Aussagenvariablen haben Wahrheitstabellen der Länge 2: $a[a : 0] = 0$, $a[a : 1] = 1$. Wenn nun ein Junktor $\square$ zwei Aussageformen α und β verknüpft, bekommt man eine neue Wahrheitstabelle durch

$$(\alpha \square \beta)[a : x_a, b : x_b, \ldots] = (p \square q)[p : \alpha[a : x_a, b : x_b, \ldots], q : \beta[a : x_a, b : x_b, \ldots]].$$

Die Wahrheitstabelle von $(a \vee b) \to a$ bekommt man mit:

$$\begin{aligned}((a \vee b) \to a)[a : 0, b : 0] &= (p \to q)[p : (a \vee b)[a : 0, b : 0], q : a[a : 0, b : 0]] \\ &= (p \to q)[p : 0, q : 0] = 1 \\ ((a \vee b) \to a)[a : 0, b : 1] &= (p \to q)[p : (a \vee b)[a : 0, b : 1], q : a[a : 0, b : 1]] \\ &= (p \to q)[p : 1, q : 0] = 0\end{aligned}$$

Es ergibt sich:

a	b	$a \vee b$	$(a \vee b) \to a$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	1

Dabei können Wahrheitstabellen natürlich beliebig um zusätzliche Variablen erweitert werden. So wäre $(a \wedge b)[a : x_a, b : x_b, c : 0] = (a \wedge b)[a : x_a, b : x_b, c : 1] = (a \wedge b)[a : x_a, b : x_b]$. Oben wird das z.B. bei $a[a : 0, b : 1] = 0$ angewendet.

Eine **Tautologie** ist eine Aussageform, deren Wahrheitstabelle nur 1-Einträge hat. Sie ist also immer wahr. Man schreibt:

$$\models \alpha \quad \text{genau dann, wenn} \quad \forall x_a \forall x_b \ldots : \alpha[a : x_a, b : x_b, \ldots] = 1$$

$\models a \vee \bar{a}$ weil

a	$\bar{a}$	$a \vee \bar{a}$.
0	1	1
1	0	1

Eine **Kontradiktion** ist eine Aussageform mit nur 0-Einträgen. Sie ist also immer falsch: $\models \bar{\alpha}$.

Die **Implikation** ist eine Tautologie mit zentraler Subjunktion. Man schreibt dann $\Rightarrow$ statt $\rightarrow$ (und sagt „impliziert“).

$$\alpha \Rightarrow \beta \quad \text{genau dann, wenn} \quad \models \alpha \rightarrow \beta$$

$a \Rightarrow a \vee b$ weil

a	b	$a \vee b$	$a \rightarrow (a \vee b)$
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	1

Implikation und Subjunktion werden oft gleich als „daraus folgt“, etc. gesprochen. Man muss dabei aus dem Kontext verstehen, ob es *objektsprachlich* (Subjunktion) oder *metasprachlich* (Implikation) gemeint ist. Ersteres ist eine Aussage *innerhalb* der Aussagenlogik (also eine Aussageform), letzteres eine Aussage *über* Aussageformen.

Um den Unterschied zwischen Subjunktion und Implikation zu verdeutlichen, betrachten wir die Aussage: „Wenn die Sonne scheint, bin ich Milliardär.“ Diese Subjunktion stimmt an einem Regentag (weil $0 \rightarrow 0$), aber nicht an einem Sonnentag (weil $\overline{1 \rightarrow 0}$). Ersteres wirkt etwas unlogisch. Aus diesem Grund unterscheidet man die Subjunktion von der Implikation und nennt erstere auch **materiale Implikation**. Obige Aussage ist *keine* Implikation, weil sie nicht immer wahr ist.

Eine Kette von Implikationen ist definiert als:

$$\alpha \Rightarrow \beta \Rightarrow \gamma \Rightarrow \delta \ldots \quad \text{genau dann, wenn} \quad \models (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma) \wedge (\gamma \rightarrow \delta) \ldots$$

also $\alpha \Rightarrow \beta$ und $\beta \Rightarrow \gamma$ und $\gamma \Rightarrow \delta$ … Beachte, dass hingegen $a \to b \to c \to d$ gleichbedeutend ist mit $((a \to b) \to c) \to d$ (und dabei sollte man aus Eindeutigkeitsgründen ohnehin die Klammern setzen).

Zur Bijunktion gibt es ein ähnliches Konstrukt: die **Äquivalenz**. Man schreibt sie als $\Leftrightarrow$ oder $\equiv$ und sagt „ist äquivalent zu“.

$$\alpha \Leftrightarrow \beta \quad \text{genau dann, wenn} \quad \models \alpha \leftrightarrow \beta$$

$a \to b \Leftrightarrow \bar{b} \to \bar{a}$ weil

a	b	$a \to b$	$\bar{a}$	$\bar{b}$	$\bar{b} \to \bar{a}$	$(a \to b) \leftrightarrow (\bar{b} \to \bar{a})$
0	0	1	1	1	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	1
1	1	1	0	0	1	1

Äquivalenz $\alpha \Leftrightarrow \beta$ gilt also, wenn beide Aussageformen α und β die gleiche Wahrheitstabelle haben.

Auch hier wird die Bijunktion und die Äquivalenz oft gleich als „genau dann, wenn“ oder „ist äquivalent“ gesprochen. Es gilt die gleiche Unterscheidung zwischen objektsprachlicher und metasprachlicher Bedeutung. Die Bijunktion wird daher auch als **materiale Äquivalenz** bezeichnet.

Eine Kette von Äquivalenzen ist definiert als:

$$\alpha \Leftrightarrow \beta \Leftrightarrow \gamma \Leftrightarrow \delta \ldots \quad \text{genau dann, wenn} \quad \models (\alpha \leftrightarrow \beta) \wedge (\beta \leftrightarrow \gamma) \wedge (\gamma \leftrightarrow \delta) \ldots$$

also $\alpha \Leftrightarrow \beta$ und $\beta \Leftrightarrow \gamma$ und $\gamma \Leftrightarrow \delta$ … Beachte, dass hingegen $a \leftrightarrow b \leftrightarrow c \leftrightarrow d \Leftrightarrow ((a \leftrightarrow b) \leftrightarrow c) \leftrightarrow d$ (und dabei sollte man aus Eindeutigkeitsgründen ohnehin die Klammern setzen).

Die Wahrheitswerte 0 und 1 kommen bis jetzt noch nicht in Aussageformen vor. Dazu definieren wir die Aussageformen 0 und 1, deren Wahrheitstabellen die Länge 1 haben mit den Werten 0 bzw. 1. Formal: $0[] = 0$, $1[] = 1$.

$$a \leftrightarrow 1 \Leftrightarrow a\,, \qquad a \leftrightarrow 0 \Leftrightarrow \bar{a}$$

weil

a	1	$a \leftrightarrow 1$	$(a \leftrightarrow 1) \leftrightarrow a$	0	$a \leftrightarrow 0$	$\bar{a}$	$(a \leftrightarrow 0) \leftrightarrow \bar{a}$
0	1	0	1	0	1	1	1
1	1	1	1	0	0	0	1

In einer gekürzten Schreibweise für (die Herleitung von) Wahrheitstabellen kann man die Wahrheitswerte direkt unter die Junktoren schreiben.

$(a$	$\leftrightarrow$	$1)$	$\leftrightarrow$	a
0	0	1	**1**	0
1	1	1	**1**	1
1	3	2	5	4

Die letzte Zeile gibt die Reihenfolge an, in der die Spalten ausgefüllt werden.

Manchmal scheint nicht klar zu sein, wo Klammern gedacht werden müssen, wenn diese weggelassen werden. Um das zu vereinheitlichen, gibt es die **Junktorpräzedenz** (**Operatorpräzedenz**, **Operatorrangfolge**), ein Konzept, das auch in Programmiersprachen zum Einsatz kommt. Was hier weiter links steht, „bindet stärker“, muss also zuerst eingeklammert werden:

$$\begin{array}{ccccc} & \wedge \cdots \oplus \cdots \vee & & \rightarrow & \\ \neg \cdots & & \cdots & & \cdots & \Rightarrow \\ & \uparrow \cdots \downarrow & & \leftrightarrow & & \Leftrightarrow \end{array}$$

Falls Unklarheit besteht, wie z.B. bei $a \uparrow b \vee c$ oder $a \rightarrow b \leftrightarrow c$, sollte man immer Klammern setzen.

$$\neg a \wedge b \vee c \wedge d \rightarrow \neg e \wedge f \vee \neg(g \wedge h) \Leftrightarrow ((\bar{a}b) \vee (cd)) \rightarrow ((\bar{e}f) \vee \overline{gh})$$

Es wird oft auch „=“ als Junktor verwendet. Die Verwendung ist aber nicht eindeutig. Z.B. wäre $a = 0$ als $a \leftrightarrow 0$ zu verstehen, was äquivalent zu $\bar{a}$ ist. Hingegen wäre $a \vee b = b \vee a$ als Formulierung der Kommutativität der Disjunktion zu verstehen, also $a \vee b \Leftrightarrow b \vee a$. Daher wollen wir hier die =-Schreibweise vermeiden. Ansonsten ist immer auf den Kontext zu achten.

Ein **Satz** (**Theorem**) ist in der Aussagenlogik einfach eine Tautologie. Meist haben Sätze die Form von Äquivalenzen. Wichtige Sätze sind:

Name	Satz
Kommutativität	$a \wedge b \Leftrightarrow b \wedge a$
	$a \vee b \Leftrightarrow b \vee a$
Assoziativität	$(a \wedge b) \wedge c \Leftrightarrow a \wedge (b \wedge c)$

Name	Satz
	$(a \vee b) \vee c \Leftrightarrow a \vee (b \vee c)$
Distributivität	$a \wedge (b \vee c) \Leftrightarrow (a \wedge b) \vee (a \wedge c)$
	$a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$
Neutralität	$a \wedge 1 \Leftrightarrow a$
	$a \vee 0 \Leftrightarrow a$
Extremalität	$a \wedge 0 \Leftrightarrow 0$ (bzw. $\models \overline{a \wedge 0}$)
	$a \vee 1 \Leftrightarrow 1$ (bzw. $\models a \vee 1$)
Dualität	$\bar{0} \Leftrightarrow 1$ (bzw. $\models \bar{0}$)
	$\bar{1} \Leftrightarrow 0$
Doppelte Negation	$\bar{\bar{a}} \Leftrightarrow a$
Komplementarität	$a \wedge \bar{a} \Leftrightarrow 0$ (bzw. $\models \overline{a \wedge \bar{a}}$)
	$a \vee \bar{a} \Leftrightarrow 1$ (bzw. $\models a \vee \bar{a}$)
Idempotenz	$a \wedge a \Leftrightarrow a$
	$a \vee a \Leftrightarrow a$
Absorption	$a \wedge (a \vee b) \Leftrightarrow a$
	$a \vee (a \wedge b) \Leftrightarrow a$
De Morgan	$\overline{a \wedge b} \Leftrightarrow \bar{a} \vee \bar{b}$
	$\overline{a \vee b} \Leftrightarrow \bar{a} \wedge \bar{b}$
Subjunktion	$a \rightarrow b \Leftrightarrow \bar{a} \vee b$
Bijunktion	$a \leftrightarrow b \Leftrightarrow (a \wedge b) \vee (\bar{a} \wedge \bar{b})$
Bijunktion 2	$a \leftrightarrow b \Leftrightarrow (a \rightarrow b) \wedge (b \rightarrow a)$
Umkehrschluss	$a \rightarrow b \Leftrightarrow \bar{b} \rightarrow \bar{a}$
XOR	$a \oplus b \Leftrightarrow (a \wedge \bar{b}) \vee (\bar{a} \wedge b)$
NAND	$a \uparrow b \Leftrightarrow \overline{a \wedge b}$
NOR	$a \downarrow b \Leftrightarrow \overline{a \vee b}$
Wahrheitswert	$\models 1$

All diese Sätze lassen sich leicht über deren Wahrheitstabellen beweisen.

Wir haben also die Aussagenlogik eingeführt, indem wir im Prinzip zuerst eine Menge $\{0,1\}$ und dann Operationen (Junktoren) auf dieser Menge definiert haben. Daraus haben wir dann Sätze abgeleitet, deren Wahrheitswert unabhängig von der Variablenbelegung immer 1 ist. Man kann aber auch den umgekehrten Weg gehen und zuerst einige Sätze als wahr postulieren (**Axiome**), dann **Ableitungsregeln** definieren, woraus dann alle wahren Sätze (Tautologien) ableitbar sein sollten. Dieses Schema nennt man **Boolesche Algebra** (nach George Boole, 1815–1864). Sätze werden in der Booleschen Algebra als $\vdash \alpha$ geschrieben, wobei

$\alpha \Leftrightarrow \beta$ für $\vdash \alpha \leftrightarrow \beta$ geschrieben werden kann.

In der Booleschen Algebra gibt es (mindestens) folgende Ableitungsregeln:

- In einem Satz $\vdash \alpha$ kann eine Aussagenvariable a (überall!) durch eine beliebige Aussageform β ersetzt werden. Das Ergebnis $\vdash \alpha \mid b : \beta$ ist wiederum ein Satz.
- Wenn in einem Satz $\vdash \alpha$ eine Aussageform β als Teilausdruck vorkommt, und ein anderer Satz $\beta \Leftrightarrow \gamma$ lautet, dann kann man in α den Teilausdruck β durch γ ersetzen. Das Ergebnis $\vdash \alpha \mid \beta : \gamma$ ist wiederum ein Satz.

Nach Huntington (1874–1952) genügen als Axiome die Kommutativität, Distributivität, Neutralität und Komplementarität. Durch die Ableitungsregeln kann man dann alle Sätze ableiten. Mittlerweile (1973, 2002) wurden die nötigen Axiome allerdings auf ein einziges unschönes Axiom reduziert.

Wir wollen $\vdash a \to (a \vee b)$ ableiten.

Satz	Ersetzung	abgeleiteter Satz
$a \to b \Leftrightarrow \bar{a} \vee b$	$\mid b : (a \vee b)$	$a \to (a \vee b) \Leftrightarrow \bar{a} \vee (a \vee b)$
$a \vee (b \vee c) \Leftrightarrow (a \vee b) \vee c$	$\mid a : \bar{a}, b : a, c : b$	$\bar{a} \vee (a \vee b) \Leftrightarrow (\bar{a} \vee a) \vee b$
$a \to (a \vee b) \Leftrightarrow \bar{a} \vee (a \vee b)$	$\mid \bar{a} \vee (a \vee b) : (\bar{a} \vee a) \vee b$	$a \to (a \vee b) \Leftrightarrow (\bar{a} \vee a) \vee b$
$a \to (a \vee b) \Leftrightarrow (\bar{a} \vee a) \vee b$	$\mid \bar{a} \vee a : 1$	$a \to (a \vee b) \Leftrightarrow 1 \vee b$
$\vdash 1 \vee b$	$\mid 1 \vee b : a \to (a \vee b)$	$\vdash a \to (a \vee b)$

In der ersten Zeile steht die Definition von $a \to b$, und wir ersetzen darin die Aussagenvariable b durch den beliebigen Ausdruck $a \vee b$. In der zweiten Zeile bringen wir die Assoziativität in die passende Form und wenden sie in Zeile 3 auf das Ergebnis von Zeile 1 an. In Zeile 4 wird die Komplementarität angewendet. In Zeile 5 wird schließlich die daraus entstandene Äquivalenz zur Ersetzung des gesamten Extremalitätsaxioms verwendet.

Die Wahrheitstabellen sind ein **semantisches Modell** für die Sätze der Booleschen Algebra und weisen den Aussageformen eine Bedeutung (wahr, falsch, 1, 0) zu. Man kann nun zeigen, dass die Boolesche Algebra

korrekt ist, da sie *nur* Tautologien produziert: $(\vdash \alpha) \Rightarrow (\models \alpha)$, und

vollständig ist, da sie *alle* Tautologien produziert: $(\models \alpha) \Rightarrow (\vdash \alpha)$.

Neben der Menge $\{0,1\}$ gibt es noch weitere gültige Modelle für die Booleschen Axiome, z.B. ist jede Menge mit den Operatoren $\cap$, $\cup$ und Komplement als Junktoren (für $\wedge$, $\vee$, $\neg$) ein gültiges Modell, das die Axiome erfüllt. Es wird üblicherweise gefordert, dass in der Modellmenge $0 \neq 1$ gilt (Quasiaxiom).

Man kann jetzt noch eine weitere und praktischere kombinierte Ableitungsregel einführen:

- Wenn man einen Satz $\beta \Leftrightarrow \gamma$ durch Aussagenvariablenersetzungen $(\beta' \Leftrightarrow \gamma') := (\beta \Leftrightarrow \gamma) \mid a : \delta_a, b : \delta_b, \ldots$ so modifizieren kann, dass ein Äquivalenzterm β' in einer Aussageform α als Teilausdruck vorkommt, dann kann man in α den Teilausdruck durch den anderen Äquivalenzterm γ' ersetzen und erhält eine zu α äquivalente Aussageform: $(\alpha \mid \beta' : \gamma') \Leftrightarrow \alpha$.

Wir wollen wieder $a \Rightarrow (a \vee b)$ beweisen, und beginnen dieses Mal von hinten.

	Aussageform	Satz mit Ersetzungen
	$a \rightarrow (a \vee b)$	$a \rightarrow b \Leftrightarrow \bar{a} \vee b \mid b : (a \vee b)$
$\Leftrightarrow$	$\bar{a} \vee (a \vee b)$	$a \vee (b \vee c) \Leftrightarrow (a \vee b) \vee c \mid a : \bar{a}, b : a, c : b$
$\Leftrightarrow$	$(\bar{a} \vee a) \vee b$	$\bar{a} \vee a \Leftrightarrow 1$
$\Leftrightarrow$	$1 \vee b$	$1 \vee a \Leftrightarrow 1 \mid a : b$
$\Leftrightarrow$	1	

Die Äquivalenzkette (von unten nach oben) vom Satz 1 zu unserer Aussageform beweist diese als Satz.

Diese Regel kann man nicht nur zum Beweisen von Sätzen sondern auch zum Umformen und Vereinfachen von Aussageformen verwenden.

	$(\bar{a} \vee b) \rightarrow b(a \vee c)$	Subjunktion $\mid a : (\bar{a} \vee b), b : b(a \vee c)$
$\Leftrightarrow$	$\overline{\bar{a} \vee b} \vee b(a \vee c)$	De Morgan $\mid a : \bar{a}$; Distributivität
$\Leftrightarrow$	$a\bar{b} \vee ab \vee bc$	Distributivität $\mid c : \bar{b}$
$\Leftrightarrow$	$a(\bar{b} \vee b) \vee bc$	Komplementarität $\mid a : b$
$\Leftrightarrow$	$a1 \vee bc$	Neutralität
$\Leftrightarrow$	$a \vee bc$	

Eine **Verknüpfungsbasis** ist eine Menge von Junktoren, mit denen alle Sätze dargestellt bzw. alle anderen Junktoren ausgedrückt werden können. Es stellt sich die Frage nach einer minimalen Verknüpfungsbasis. Ein Blick auf die Liste der Sätze zeigt, dass $\{\wedge, \vee, \neg\}$ offenbar ausreicht. Die Frage, ob es noch kleinere Mengen gibt, kann mit Ja beantwortet werden:

- Die Sheffer-Basis (NAND-Basis) benutzt nur $\uparrow$. (Sheffer selbst benutzte dafür das Sheffer-Symbol $|$.) Um zu zeigen, dass damit alle Sätze darstellbar sind, müssen wir nur zeigen, dass $\wedge$, $\vee$ und $\neg$ ausdrückbar sind:

$$\begin{aligned} \bar{a} &\Leftrightarrow \overline{a \wedge a} \Leftrightarrow a \uparrow a \\ a \wedge b &\Leftrightarrow \overline{\overline{ab}} \Leftrightarrow \overline{\overline{ab}\,\overline{ab}} \Leftrightarrow (a \uparrow b) \uparrow (a \uparrow b) \\ a \vee b &\Leftrightarrow \overline{\overline{a \vee b}} \Leftrightarrow \bar{a}\bar{b} \Leftrightarrow \overline{\overline{aa}\,\overline{bb}} \Leftrightarrow (a \uparrow a) \uparrow (b \uparrow b) \end{aligned}$$

- Die Peirce-Basis (NOR-Basis) benutzt nur $\downarrow$. Der Beweis ist analog.

3.2 Normalformen, Minimalformen

Jetzt soll es darum gehen, möglichst einfache Aussageformen zu finden, die äquivalent zu vorgegebenen Aussageformen oder Wahrheitstabellen sind. Der erste Schritt dazu sind Normalformen.

Ein **Konjunktionsterm** ist eine Aussageform, die aus einer Konjunktion von verschiedenen Aussagenvariablen in möglicherweise negierter Form besteht. Ein **Disjunktionsterm** ist analog dazu eine Disjunktion. Bei diesen Termen ist die Reihenfolge der Aussagenvariablen egal, zwei Konjunktionsterme (bzw. Disjunktionsterme) werden als gleich betrachtet, wenn sie äquivalent sind.

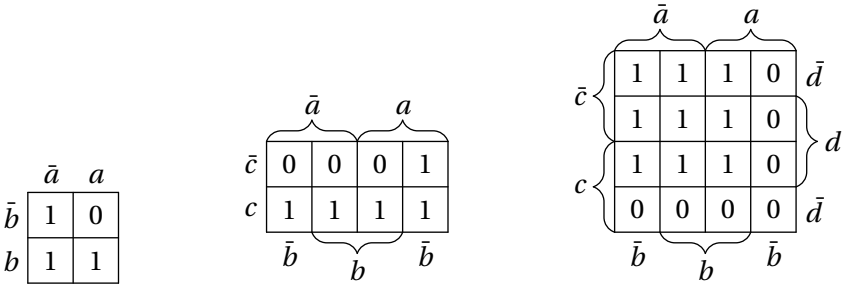
Konjunktionsterme: $a\bar{b}c\bar{d}$, $\bar{a}bc$, b, $\bar{c}$
Disjunktionsterme: $a \vee \bar{b} \vee c \vee \bar{d}$, $\bar{c} \vee d$
Weder noch: $ab \vee c$, $(a \vee b)c$, $\overline{a \vee c}$, $a \to b$

Man kann auch die Aussageform 1 als Konjunktion von 0 Aussagenvariablen und somit als Konjunktionsterm betrachten. Genauso die Aussageform 0 als Disjunktion von 0 Aussagenvariablen und somit als Disjunktionsterm. Das wird jedoch uneinheitlich gesehen.

Eine **disjunktive Normalform** (**DNF**) ist eine Aussageform, die aus einer Disjunktion von verschiedenen Konjunktionstermen besteht. Eine **konjunktive Normalform** (**KNF**) besteht analog dazu aus einer Konjunktion von verschiedenen Disjunktionstermen.

DNF: $a\bar{b}c \vee a\bar{d} \vee \bar{a}\bar{c}d$
KNF: $(a \vee \bar{b} \vee c)(a \vee \bar{d})(\bar{a} \vee \bar{c} \vee d)$

Ein **K-Diagramm** (Karnaugh-, Karnaugh-Veitch-, KV-Diagramm) ist eine alternative Darstellung von Wahrheitstabellen als zweidimensionale Tabelle. Dabei



(a) $a \to b$ (b) $(a \to b) \to c$ (c) $(a \to b)(c \to d)$

Abbildung 10: K-Diagramme für 2, 3 und 4 Aussagenvariablen. An der Stelle $a\bar{b}$ steht z.B. der Eintrag $(a \to b)[a:1, b:0] = 0$.

werden die Wahrheitswerte von ein bis zwei Variablen pro Dimension (horizontal, vertikal) dargestellt. Zusätzlich sind die Wahrheitswerte so angeordnet, dass sich zwei benachbarte Spalten (bzw. Zeilen) in nur einer Variable unterscheiden. Wenn z.B. in einer Spalte $a\bar{b}$ gilt (also $\alpha[a:1, b:0, \ldots]$ dargestellt wird), dann gilt für die benachbarten Spalten $\bar{a}\bar{b}$ ($\alpha[a:0, b:0, \ldots]$) oder ab ($\alpha[a:1, b:1, \ldots]$). Die erste und letzte Spalte (bzw. Zeile) gilt auch als benachbart. Das führt im Prinzip zu einem zyklischen Gray-Code.

Abbildung 10 zeigt K-Diagramme für 2, 3 und 4 Aussagenvariablen.

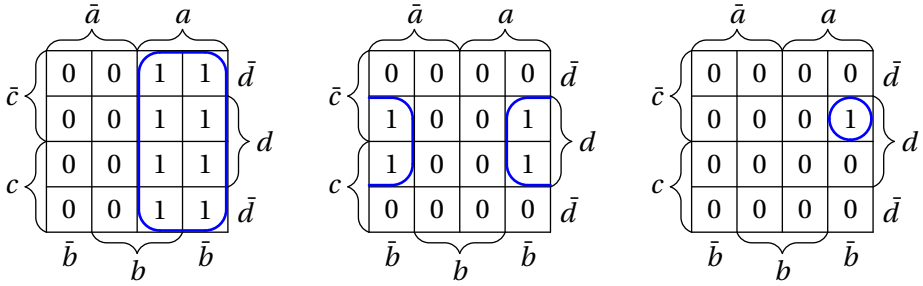
Die Anordnung der Variablen in Spalten und Zeilen ist nicht eindeutig. Die Position und Negiertheit der Variablen kann beliebig vertauscht werden.

Konjunktionsterme erscheinen in K-Diagrammen als zusammenhängende Blöcke mit Seitenabmessungen 1, 2 oder 4. Das funktioniert allerdings nur wirklich für bis zu 4 Variablen, daher sind K-Diagramme für mehr als 4 Variablen unüblich.

Abbildung 11 zeigt K-Diagramme für verschiedene Konjunktionsterme.

Man sieht, dass längere Konjunktionsterme kleineren Blöcken entsprechen, und umgekehrt.

Für eine vorgegebene Menge von Aussagenvariablen bezeichnet man einen Konjunktionsterm, in dem alle Aussagenvariablen vorkommen als **Minterm**. Sie erscheinen in K-Diagrammen als 1×1-Block, wie z.B. in Abbildung 11(c). Das analoge Konzept für Disjunktionsterme heißt **Maxterm**.



(a) a (b) $\bar{b}d$ (c) $a\bar{b}\bar{c}d$

Abbildung 11: Konjunktionsterme in K-Diagrammen.

Eine DNF, die eine Disjunktion von nur Mintermen ist, nennt man eine **vollständige DNF**. Eine vollständige KNF besteht nur aus Maxtermen.

Für die Aussagenvariablenmenge $\{a, b, c\}$ ist

- $abc \vee ab\bar{c} \vee a\bar{b}c \vee \bar{a}\bar{b}\bar{c}$ eine vollständige DNF, aber
- $ab \vee a\bar{b}c \vee \bar{a}\bar{b}\bar{c}$ nicht, weil im ersten Konjunktionsterm die Variable c fehlt,

obwohl die zwei Aussageformen äquivalent sind.

Theorem Jede Aussageform hat eine eindeutige äquivalente vollständige DNF (und auch eine vollständige KNF).

Zum Beweis zeigen wir, wie man die vollständige DNF aus der Wahrheitstabelle ableitet. Für jede Zeile, in der die Wahrheitstabelle einen 1-Eintrag hat, erzeugen wir einen Minterm, in dem jede Variable genau dann negiert ist, wenn die Variable in dieser Zeile den Werte 0 hat. Jeder dieser Minterme erzeugt genau diesen einen 1-Eintrag, und deren Disjunktion damit die gewünschte Wahrheitstabelle.

a	b	c	α	Minterm
0	0	0	0	
0	0	1	1	$\bar{a}\bar{b}c$
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	$a\bar{b}c$
1	1	0	0	
1	1	1	1	abc

$$\alpha \Leftrightarrow \bar{a}\bar{b}c \vee a\bar{b}c \vee abc$$

Für die KNF wenden wir eine Strategie mit zweifacher Anwendung der De-Morgan-Regel an:

$$\alpha \Leftrightarrow \overline{\mathrm{DNF}(\bar{\alpha})}$$

$\bar{\alpha}$ bedeutet, dass wir die Minterme der Wahrheitstabellen-Zeilen mit 0-Eintrag (statt 1-Eintrag) erzeugen. Danach produziert die zweistufige Anwendung von De-Morgan eine vollständige KNF dadurch, dass sich Disjunktionen in Konjunktionen verwandeln, und umgekehrt. Die Negation der Aussagenvariablen wird dadurch invertiert.

a	b	c	α	Minterm
0	0	0	0	$\bar{a}\bar{b}\bar{c}$
0	0	1	1	
0	1	0	1	
0	1	1	0	$\bar{a}bc$
1	0	0	0	$a\bar{b}\bar{c}$
1	0	1	1	
1	1	0	1	
1	1	1	1	

$$\begin{aligned}\alpha &\Leftrightarrow \overline{\bar{a}\bar{b}\bar{c} \vee \bar{a}bc \vee a\bar{b}\bar{c}} \\ &\Leftrightarrow \overline{\bar{a}\bar{b}\bar{c}}\,\overline{\bar{a}bc}\,\overline{a\bar{b}\bar{c}} \\ &\Leftrightarrow (a \vee b \vee c)(a \vee \bar{b} \vee \bar{c})(\bar{a} \vee b \vee c)\end{aligned}$$

Wenn eine Aussageform nicht als Wahrheitstabelle sondern als Formel gegeben ist, kann man die vollständige DNF auch durch Äquivalenz-Umformungen ermittelt. Der erste Schritt ist die DNF durch „Ausmultiplizieren“ (Distributivität, z.B. $a(b \vee c) \Leftrightarrow ab \vee ac$). Der zweite Schritt ist die vollständige DNF durch „Erweitern“ (Konjunktion mit z.B. $1 \Leftrightarrow c \vee \bar{c}$, also $ab \vee ac \Leftrightarrow ab(c \vee \bar{c}) \vee a(b \vee \bar{b})c$) und weiteres Ausmultiplizieren.

$$\begin{aligned}(a \to b) \to c &\Leftrightarrow \overline{\bar{a} \vee b} \vee c \\ &\Leftrightarrow a\bar{b} \vee c\end{aligned}$$

$$\Leftrightarrow a\bar{b}\underbrace{(c \vee \bar{c})}_{\Leftrightarrow 1} \vee \underbrace{(a \vee \bar{a})}_{\Leftrightarrow 1}\underbrace{(b \vee \bar{b})}_{\Leftrightarrow 1} c$$
$$\Leftrightarrow a\bar{b}c \vee a\bar{b}\bar{c} \vee abc \vee \cancel{a\bar{b}c} \vee \bar{a}bc \vee \bar{a}\bar{b}c$$

Ein **Implikant** einer Aussageform α ist ein Konjunktionsterm β, der α impliziert: $\beta \Rightarrow \alpha$. Ein **Primimplikant** ist ein Implikant, aus dem keine Aussagenvariable gestrichen werden kann, d.h. kein Teilausdruck wäre ein Implikant.

Dazu ist folgende Methode hilfreich: Es sei α eine Aussageform und β ein Konjunktionsterm. Wenn β wahr ist, dann ist auch jede Aussagenvariable, die in β un-negiert vorkommt, wahr (1), bzw. falsch (0), wenn sie negiert vorkommt. Man kann dann in α alle Vorkommen dieser Variablen durch diese Wahrheitswerte ersetzen: $\alpha \mid \beta : 1$. Es gilt:

$$\beta \Rightarrow \alpha \quad \text{genau dann, wenn} \quad \models \alpha \mid \beta : 1$$

Weiters gilt $\models \gamma$ natürlich nicht, wenn eine Variablenbelegung gefunden werden kann, für die γ 0 ist.

$$\alpha := a \vee c \rightarrow (\bar{a} \rightarrow b)c$$

Ist $\bar{a}bc$ Implikant? ($\bar{a}bc \Rightarrow \alpha$?)
$\alpha \mid \bar{a}bc : 1 \Leftrightarrow 0 \vee 1 \rightarrow (1 \rightarrow 1)1 \Leftrightarrow 1 \rightarrow 1 \Leftrightarrow 1$ Ja.

Ist $\bar{a}bc$ Primimplikant? Wenn ja, dann dürften $\bar{a}b$, $\bar{a}c$ und bc keine Implikanten sein.
Ist also $\bar{a}b$ Implikant? ($\bar{a}b \Rightarrow \alpha$?)
$\alpha \mid \bar{a}b : 1 \Leftrightarrow 0 \vee c \rightarrow (1 \rightarrow 1)c \Leftrightarrow c \rightarrow c \Leftrightarrow 1$ Ja.
Daher ist $\bar{a}bc$ *kein* Primimplikant.

Ist $\bar{a}b$ Primimplikant? Wenn ja, dann dürften $\bar{a}$ und b keine Implikanten sein.
Ist $\bar{a}$ Implikant? ($\bar{a} \Rightarrow \alpha$?)
$\alpha \mid \bar{a} : 1 \Leftrightarrow 0 \vee c \rightarrow (1 \rightarrow b)c \Leftrightarrow c \rightarrow bc$. Und da $c \rightarrow bc[b : 0, c : 1] = 0$: Nein.
Ist b Implikant? ($b \Rightarrow \alpha$?)
$\alpha \mid b : 1 \Leftrightarrow a \vee c \rightarrow (\bar{a} \rightarrow 1)c \Leftrightarrow a \vee c \rightarrow c$. Und da $a \vee c \rightarrow c[a : 1, c : 0] = 0$: Nein.
Daher ist $\bar{a}b$ Primimplikant.

Implikanten und Primimplikanten kann man auch aus dem K-Diagramm ablesen. Man sucht nach möglichst großen Blöcken (mit Seitenabmessungen 1, 2 oder 4), die nur 1-en umschließen. Ein Block, den man in irgend eine Richtung erweitern könnte, ist nur ein Implikant. Wenn man den Block nicht mehr erweitern kann, ist es ein Primimplikant.

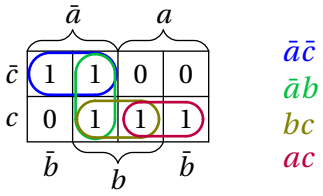


Abbildung 12: K-Diagramm von $a \vee c \to (\bar{a} \to b)c$ mit Primimplikanten

Abbildung 12 zeigt alle Primimplikanten zu $a \vee c \to (\bar{a} \to b)c$. So ist z.B. ac ein Primimplikant, weil man ihn weder nach links auf Größe 4 erweitern kann, wegen der 0 in $\bar{a}\bar{b}c$, noch nach oben, wegen der 0-en in $a\bar{c}$.

Die Primimplikanten überdecken den 1-Bereich vollständig. Das heißt, dass deren Disjunktion äquivalent zur Aussageform ist.

$$a \vee c \to (\bar{a} \to b)c \Leftrightarrow \bar{a}\bar{c} \vee \bar{a}b \vee bc \vee ac$$

Eine Menge von Primimplikanten heißt **wesentliche Primimplikanten**, wenn man keinen Primimplikanten weglassen kann, ohne die Äquivalenz zur Aussageform zu verlieren. Die Disjunktion von wesentlichen Primimplikanten nennt man **disjunktive Minimalform** (**DMF**).

In Abbildung 12 sieht man, dass man $\bar{a}b$ weglassen kann, ohne dass 1-en unüberdeckt bleiben. Also sind $\{\bar{a}\bar{c}, bc, ac\}$ wesentliche Primimplikanten.

Allerdings könnte man alternativ auch bc weglassen. Also sind auch $\{\bar{a}\bar{c}, \bar{a}b, ac\}$ wesentliche Primimplikanten.

Es gibt daher zwei DMFs:

$$a \vee c \to (\bar{a} \to b)c \Leftrightarrow \bar{a}\bar{c} \vee bc \vee ac \Leftrightarrow \bar{a}\bar{c} \vee \bar{a}b \vee ac$$

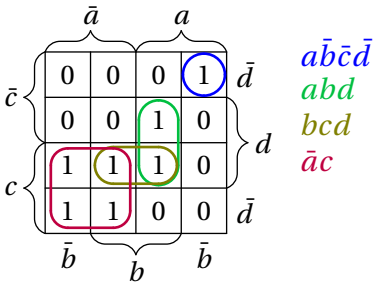
Man sieht, dass die DMF nicht immer eindeutig ist.

Da K-Diagramme erstens nur bis zu 4 Variablen wirklich funktionieren, und zweitens nur dem Menschen nutzen, aber nicht für computergestützte Schaltungsminimierung geeignet sind, braucht man einen Algorithmus, um die (wesentlichen) Primimplikanten zu finden. Dafür gibt es das Verfahren von **Quine-McCluskey**.

Dabei beginnt man mit den Mintermen und ordnet sie in Klassen nach der Anzahl der Negationen. Danach versucht man, alle Paare von Mintermen, die sich in nur einer Variable bezüglich Negation unterscheiden, zusammenzufassen. Dazu

a	b	c	d	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

(a) Wahrheitstabelle



(b) K-Diagramm (zum Vergleich)

$abcd$ ①	1,3 ***abd***	
	1,2 ***bcd***	
$\bar{a}bcd$ ②	2,5 $\bar{a}bc$ ⑦	9,10 7,8 ***$\bar{a}c$***
$ab\bar{c}d$ ③	2,4 $\bar{a}cd$ ⑨	
$\bar{a}b\bar{c}d$ ④	4,6 $\bar{a}b\bar{c}$ ⑧	
$\bar{a}bcd\bar{}$ ⑤	5,6 $\bar{a}cd\bar{}$ ⑩	
$\bar{a}b\bar{c}d\bar{}$ ⑥		
$a\bar{b}\bar{c}\bar{d}$		

(c) Primimplikanten ermitteln

	$abcd$	$\bar{a}bcd$	$ab\bar{c}d$	$\bar{a}b\bar{c}d$	$\bar{a}bcd\bar{}$	$\bar{a}b\bar{c}d\bar{}$	$a\bar{b}\bar{c}\bar{d}$
$a\bar{b}\bar{c}\bar{d}$							$\otimes^5$
abd	$\boxtimes^2$		$\otimes^1$				
bcd	×	×					
$\bar{a}c$		$\boxtimes^4$		$\otimes^3$	$\boxtimes^4$	$\boxtimes^4$	

(d) Wesentliche Primimplikanten

Abbildung 13: Der Quine-McCluskey-Algorithmus

muss man immer nur eine Klasse (mit m Negationen) mit der nächsten (mit $m+1$ Negationen vergleichen), deswegen werden diese Klassen gebildet. Zusammenfassen bedeutet hier, dass man die Variable, in der sich die Minterme unterscheiden, weglassen kann, weil z.B. $a\bar{b}\bar{c} \vee ab\bar{c} \Leftrightarrow a\bar{c}$ ist.

Dabei werden Terme mit einer Variable weniger produziert. Diese kommen wiederum in entsprechende Klassen und werden auf gleiche Weise weiter zusammengefasst. Es dürfen aber nur Terme zusammengefasst werden, die aus den gleichen Variablen bestehen. Abbildung 13(c) zeigt diesen Teil des Algorithmus. Beim Zusammenfassen bekommen die zusammenzufassenden Terme eine Nummer (rechts), falls sie noch keine haben. Beim zusammengefassten Term wird links notiert, aus welchen Term-Nummern er sich zusammensetzt. Das wird wiederholt, bis nichts mehr zusammengefasst werden kann. Am Schluss erkennt man die Primimplikanten daran, dass rechts nicht mit einer Nummer markiert sind.

Der zweite Teil des Quine-McCluskey-Algorithmus ist das Ermitteln der wesentlichen Primimplikanten. Dazu erzeugt man eine Tabelle wie in Abbildung 13(d). Oben stehen die Minterme und links die Primimplikanten. Man macht an jeder Stelle der Tabelle ein Kreuz, für die der Minterm α den Primimplikanten impliziert (z.B. $\bar{a}\bar{b}cd \Rightarrow \bar{a}c$), also wenn die Negationen übereinstimmen. Dann führt man folgenden Algorithmus aus:

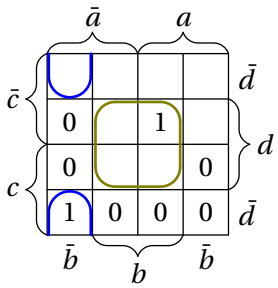
> Solange nicht jede Spalte markiert ist:
> Finde unmarkierte Spalte mit möglichst wenigen Kreuzen.
> Markiere dort ein Kreuz mit einem Kreis.
> Markiere alle Kreuze in der selben Zeile mit einem Quadrat.

Unmarkierte Zeilen zeigen dann unwesentliche Primimplikanten an. In Abbildung 13(d) wählt man als erstes die dritte Spalte (mit nur einem Kreuz), markiert dieses mit einem Kreis$^{(1)}$, und dann alle anderen Kreuze in Zeile 2 mit einem Quadrat$^{(2)}$. Es folgen Spalte 4$^{(3)}$ (Zeile 4$^{(4)}$) und Spalte 7$^{(5)}$ (Zeile 1). Danach sind alle Spalten markiert. Zeile 3 bleibt unmarkiert, d.h. bcd ist unwesentlich, wie man auch zum Vergleich im K-Diagramm in Abbildung 13(b) sieht. Die DMF ist also $a\bar{b}\bar{c}\bar{d} \vee abd \vee \bar{a}c$.

Für mehr als 10 Variablen wird aber der Rechenaufwand des Quine-McCluskey-Algorithmus auch für Computer meist zu groß, das Problem ist nämlich NP-komplett. Daher weicht man dann auf heuristische Methoden (wie ESPRESSO) aus, die gute Lösungen, aber nicht unbedingt die beste Lösung finden.

a	b	c	d	y
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	1	0	0
1	0	1	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

(a) Wahrheitstabelle



(b) K-Diagramm

$(abcd)$ ①	1,3 abd ⑪	13,14 11,12 $\boldsymbol{bd}$
	1,2 (bcd) ⑬	
$(\bar{a}bcd)$ ②	2,4 $(\bar{a}bd)$ ⑫	18,19 15,17 $\boldsymbol{a\bar{c}}$
$ab\bar{c}d$ ③	3,6 $ab\bar{c}$ ⑮	14,20 15,16 $\boldsymbol{b\bar{c}}$
	3,5 $a\bar{c}d$ ⑱	
	3,4 $b\bar{c}d$ ⑭	
$(\bar{a}b\bar{c}d)$ ④	4,7 $(\bar{a}b\bar{c})$ ⑯	20,22 19,21 $(\bar{c}\bar{d})$
$(a\bar{b}\bar{c}d)$ ⑤	5,8 $(a\bar{b}\bar{c})$ ⑰	
$(ab\bar{c}\bar{d})$ ⑥	6,8 $(a\bar{c}\bar{d})$ ⑲	
	6,7 $(b\bar{c}\bar{d})$ ⑳	
$\bar{a}\bar{b}cd$ ⑨	9,10 $\boldsymbol{\bar{a}\bar{b}d}$	
$(\bar{a}b\bar{c}\bar{d})$ ⑦	7,10 $(\bar{a}\bar{c}\bar{d})$ ㉑	
$(a\bar{b}\bar{c}\bar{d})$ ⑧	8,10 $(\bar{b}\bar{c}\bar{d})$ ㉒	
$(\bar{a}\bar{b}\bar{c}\bar{d})$ ⑩		

(c) Primimplikanten

	$ab\bar{c}d$	$\bar{a}\bar{b}c\bar{d}$
$\boldsymbol{\bar{a}\bar{b}\bar{d}}$		⊗
$\boldsymbol{bd}$	⊗	
$a\bar{c}$	×	
$b\bar{c}$	×	

(d) Wesentliche Primimplikanten

Abbildung 14: Quine-McCluskey-Algorithmus für unvollständige Wahrheitstabellen

Eine verallgemeinerte Logik-Minimierung lässt unvollständige Wahrheitstabellen zu, in denen manche Einträge weder 0 noch 1 sondern „egal“ sind. Das kann der Algorithmus ausnutzen, um noch kürzere Darstellungen zu finden, indem die Einträge nach Belieben auf 0 oder 1 gesetzt werden.

Abbildung 14(a) zeigt so eine unvollständige Wahrheitstabelle. Die Wahrheitstabelle kann auch komplett ausgeschrieben werden, mit X-Einträgen, wo y egal ist. Im K-Diagram in Abbildung 14 werden Blöcke so weit wie möglich auch auf leere Felder ausgedehnt. Falls es aber mehrere maximale Blöcke gibt, um die selben 1-en abzudecken, wie im Beispiel bd, $b\bar{c}$ und $a\bar{c}$, dann wird nur einer davon ausgewählt.

Auch im Quine-McCluskey-Algorithmus können „Egal“-Einträge berücksichtigt

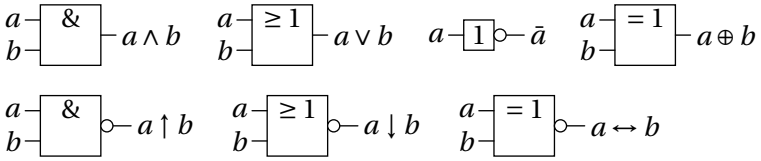


Abbildung 15: Logikgatter

werden. Deren Minterme werden, wie in Abbildung 14(c), in Klammern geschrieben. Beim Zusammenfassen zweier Terme ist der zusammengefasste Term genau dann in Klammern zu schreiben, wenn beide zusammengefassten Terme in Klammern stehen (z.B. ⑬ aus ① und ②). Als Primimplikanten zählen am Schluss nur Terme, die nicht eingeklammert sind. So wird $(\bar{c}\bar{d})$ nicht berücksichtigt. Das wäre die obere leere Zeile im K-Diagramm.

In der Tabelle zum Bestimmen der wesentlichen Primimplikanten werden nur tatsächliche Minterme, also keine „Egal"-Einträge berücksichtigt. Abbildung 14(d) zeigt, dass es in der ersten Spalte drei Möglichkeiten gibt, was exakt den drei Möglichkeiten im K-Diagramm entspricht.

3.3 Schaltnetze

Schaltnetze sind Graphen aus **Logikgattern** und **Verbindungen** (Leitungen). Logikgatter repräsentieren Junktoren, wie in Abbildung 15 ersichtlich. Sie haben Eingänge, die links oder oben zu finden sind, und Ausgänge rechts oder unten. Verbindungen verbinden jeweils einen Ausgang mit einem oder mehreren Eingängen. Eine Verbindung darf also nicht zwei Ausgänge miteinander verbinden. Es darf in Schaltnetzen auch keine Zyklen geben, d.h. es darf keinen Pfad aus Ausgang-zu-Eingang-Verbindungen geben, der wieder zum selben Gattern zurückführt.

Abbildung 16 zeigt ein beispielhaftes Schaltnetz. Links stehen die Eingänge a, b, c und d. Sie sind verbunden mit den Eingängen der Und-Gatter, wobei a mit beiden Gattern verbunden ist (Abzweigung). Wenn sich zwei Linien kreuzen, muss man einen Punkt an die Kreuzung setzen, wie bei der Verbindung acd, damit die Linien als verbunden gelten. Ohne Punkt sind Kreuzungen nicht verbunden, wie z.B. die Kreuzung von a und b. Rechts stehen die Ausgänge, die Aussageformen entsprechen.

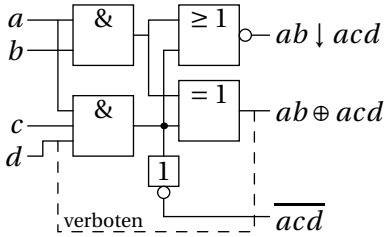


Abbildung 16: Schaltnetz

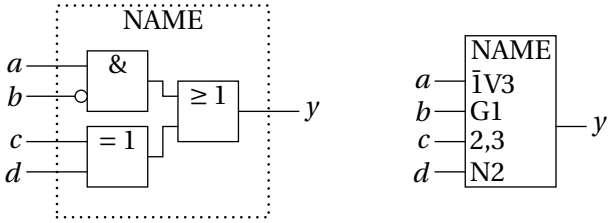


Abbildung 17: Baustein-Gruppierung, Abhängigkeitsnotation

Das zweite Und-Gatter hat mehr als zwei Eingänge und steht für acd. Das Nicht-Gatter hat einen Signalfluss von oben nach unten. Signalflüsse von rechts nach links oder unten nach oben sind unüblich. Die strichlierte Verbindung wäre eine in Schaltnetzen verbotene Rückkopplung, die einen Zyklus mit einem Und-Gatter und dem XOR-Gatter produzieren würde.

Ein wichtiger Kennwert von Verbindungen ist der **Fanout**: die Anzahl der Eingänge, die mit dem Ausgang verbunden sind. In Abbildung 16 hat z.B. das untere Und-Gattern einen Fanout von 3. Ist der Fanout zu groß (technologieabhängig), müssen eventuell Treiber-Gatter zwischengeschaltet werden.

Ein weiterer Kennwert ist das **Delay** (oder Latency). Es bezeichnet die Zeitverzögerung, mit der der Ausgang eines Gatters auf eine Änderung der Eingänge reagiert. Wir nehmen hier an, dass jedes Gatter ein Delay von Δ aufweist. Eingänge gelten als unverzögert, haben also 0 Delay. Bei Gattern, die in Serie, also hintereinander geschaltet sind, addiert sich das Delay; bei parallel geschalteten Gattern nicht. In Abbildung 16 haben die Ausgänge der Und-Gatter ein Delay von Δ, die Ausgänge des Schaltnetzes ein Delay von 2Δ.

Teilschaltungen können zu neuen wiederverwendbaren Bausteinen gruppiert werden. Abbildung 17 zeigt ein Beispiel. Der Baustein bekommt einen Namen. Die

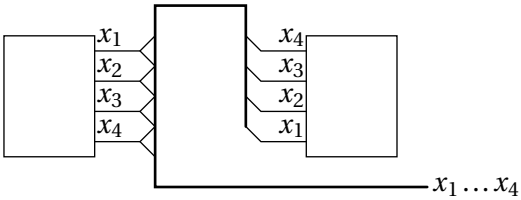


Abbildung 18: Bus

Eingänge und Ausgänge können geeignet bezeichnet werden. Dazu gibt es einen DIN/IEC-Standard, genannt „**Abhängigkeitsnotation**“. Die funktioniert auf folgende Weise: G1 bedeutet, dass alle Eingänge, die mit 1 markiert sind, mit diesem Eingang (b) Und-verknüpft werden. Das trifft auf Eingang a zu. Dass dort $\bar{1}$ steht, bedeutet, dass der Eingang nicht mit b sondern mit $\bar{b}$ Und-verknüpft wird. Das Resultat mündet in V3, was bedeutet, dass alle Eingänge, die mit 3 markiert sind, mit diesem Zwischenergebnis Oder-verknüpft werden. N2 bei Eingang d bedeutet, dass alle Eingänge, die mit 2 markiert sind, negiert werden, wenn d 1 ist. Das entspricht einer XOR-Operation. Eingang c ist schließlich mit 2,3 markiert, was bedeutet, dass c zuerst mit d XOR verknüpft wird, und dann mit obigem Zwischenergebnis Oder-verknüpft wird. Es ergibt sich die selbe logische Funktion wie für das Schaltnetz links. Außer G, V und N (für Und, Oder, Negation) gibt es noch eine Menge weiterer Symbole, von denen manche weiter unten eingeführt werden.

Auch Verbindungen können gruppiert werden zu einem sogenannten **Bus**. Abbildung 18 zeigt ein Beispiel für einen Bus aus vier Verbindungen. Die schrägen Linien zeigen an, in welche Richtung die Verbindung eingespeist bzw. entnommen wird. Die Verbindungen selbst müssen benannt werden, um sie richtig zuordnen zu können. Busse haben in der Praxis üblicherweise noch Kontroll-Leitungen für einen Bus-Takt, Schreib-/Lese-Auswahl, etc.

Abbildung 19 zeigt ein Beispiel für Multiplexer. Ein **Multiplexer** ist eine Schaltung, die auf Basis von m Selektionsleitungen s, s_0, s_1, ... einen aus 2^m Eingängen auswählt und auf den Ausgang durchschaltet (m-MUX). Ein 1-MUX kann durch vier Gatter implementiert werden. Wenn $s \leftrightarrow 0$ ist, wird x_0 mit 1 und x_1 mit 0 geundet, und die Ergebnisse geodert. Dadurch wird $y \leftrightarrow x_0$. Bei $s \leftrightarrow 1$ ist es umgekehrt, und $y \leftrightarrow x_1$. Das Delay zwischen einem x-Eingang und dem y-Ausgang ist 2Δ. Das Schaltsymbol dafür hat den Namen MUX, was bedeutet, dass der ausgewählte Eingang auf den Ausgang geschaltet wird. Die Auswahl passiert durch

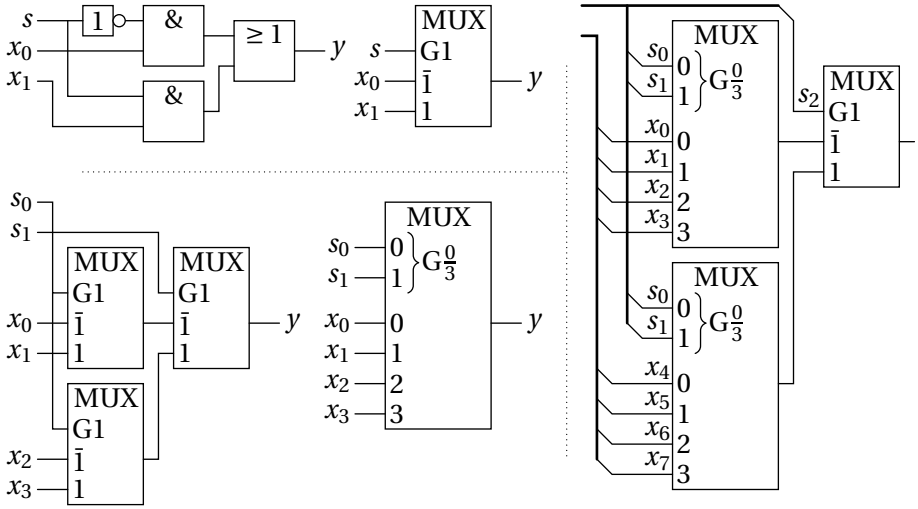


Abbildung 19: Kaskadierte Multiplexer (1-MUX, 2-MUX, 3-MUX)

das G1 auf Eingang s.

Der 2-MUX kann durch drei 1-MUX implementiert werden (Kaskade), wobei zwei auf Basis von s_0 zwischen x_0 und x_1 bzw. x_2 und x_3 auswählen, und deren Ausgänge mit einem dritten 1-MUX auf Basis von s_1 ausgewählt werden. Das Schaltsymbol fasst die Selektionseingänge s_0, s_1 mit einer Klammer zusammen und erzeugt mit $\mathrm{G}\frac{0}{3}$ vier Und-Labels, von denen genau eines aktiv wird und einen der vier x-Eingänge auswählt. Diese Spezialnotation stellt im Prinzip eine Übersetzung einer Binärzahl $(s_1 s_0)_2$ dar. Ein 3-MUX kann dementsprechend mit zwei 2-MUX und einem 1-MUX implementiert werden. Die Gatteranzahlen und Delays sind:

MUX	#Gatter	Delay $x \mapsto y$
1-MUX	4	2Δ
2-MUX	12	4Δ
3-MUX	28	6Δ
4-MUX	60	8Δ

Da logische Funktionen immer in disjunktive Normalform gebracht werden können, hat man zur Implementierung spezielle **Programmable-Logic-Arrays** (**PLA**s) entwickelt. Abbildung 20 zeigt den Aufbau. Die Eingänge werden zuerst in negier-

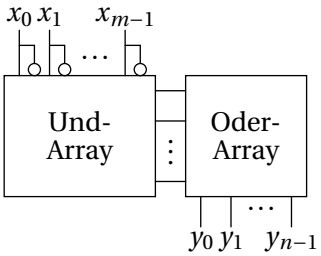


Abbildung 20: Programmable-Logic-Array (PLA)

te und nicht-negierte Leitungen dupliziert. Diese Leitungen durchziehen vertikal ein Und-Array, das programmierbare Verbindungen zu horizontalen Leitungen herstellt, in denen Konjunktionen berechnet werden. Jede Verbindungsstelle steht für eine Aussagenvariable der Konjunktion. Die horizontalen Leitungen werden dann auf ähnliche Weise in einem Oder-Array zu DNFs zusammengeführt. Viele solche PLAs haben auch Speicherbausteine integriert, um mit Rückkopplung iterative Berechnungen implementieren zu können.

Abbildung 21 zeigt ein Beispiel zur Schaltungsminimierung. Dabei soll eine 7-Segment-Anzeige die Buchstaben A,...,F anzeigen. Die Eingänge sind die drei Bits der Zahl $x = (x_2x_1x_0)_2 \in \{0,\dots,5\}$. Für $x \in \{6,7\}$ ist die Anzeige undefiniert, also „egal“. Man sieht, dass zuerst der undefinierte Primimplikant x_2x_1 ausgeschieden wird. Beim Bestimmen der wesentlichen Primimplikanten hat man die Wahl zwischen x_1x_0 und $\bar{x}_2x_1$, das Resultat ist also nicht eindeutig.

Arithmetische Schaltnetze sind Schaltnetze, die arithmetische Operationen wie z.B. Addition auf Bit-Repräsentationen von Zahlen durchführen. Der einfachste arithmetische Bauteil ist ein **Halbaddierer**. Er addiert zwei Bits a, b. Die Ausgänge sind das Summen-Bit s und ein Übertrags-Bit (Carry-Bit) c. Im Folgenden die Wahrheitstabelle und Schalftfunktion des Halbaddierers, und in Abbildung 22 die Schaltung.

$$(c,s)_2 = a+b$$
$$(c,s) := \mathrm{HA}(a,b)$$

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

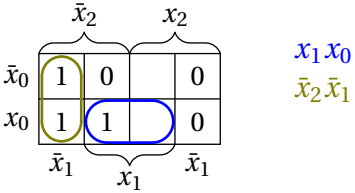
$$c \Leftrightarrow ab$$
$$s \Leftrightarrow a \oplus b$$
$$\Leftrightarrow \bar{c}(a \vee b)$$

Ein **Volladdierer** erhält zusätzlich zu den zwei Eingängen noch ein hereinkommendes Carry-Bit c_{in}. Wir addieren zuerst a und b, dann das Ergebnis und c_{in}. Es

(a) 7-Segment

x	x_2	x_1	x_0	y
0	0	0	0	1
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0

(b) Wahrheitstabelle



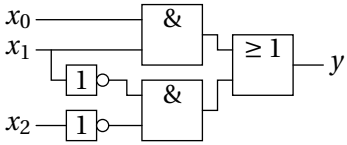
(c) K-Diagramm

$(x_2 x_1 x_0)$ ①	1,3	$(x_2 x_1)$
	1,2	$\boldsymbol{x_1 x_0}$
$\bar{x}_2 x_1 x_0$ ②	2,4	$\boldsymbol{\bar{x}_2 x_0}$
$(x_2 x_1 \bar{x}_0)$ ③		
$\bar{x}_2 \bar{x}_1 x_0$ ④	4,5	$\boldsymbol{\bar{x}_2 \bar{x}_1}$
$\bar{x}_2 \bar{x}_1 \bar{x}_0$ ⑤		

(d) Quine-McCluskey

	$\bar{x}_2 x_1 x_0$	$\bar{x}_2 \bar{x}_1 x_0$	$\bar{x}_2 \bar{x}_1 \bar{x}_0$
$\boldsymbol{x_1 x_0}$	⊗		
$\bar{x}_2 x_0$	×	×	
$\boldsymbol{\bar{x}_2 \bar{x}_1}$		⊠	⊗

(e) Wesentliche Primimplikanten



(f) Schaltung

Abbildung 21: Beispiel 7-Segment-Anzeige

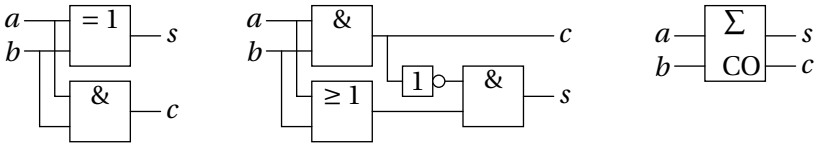


Abbildung 22: Halbaddierer

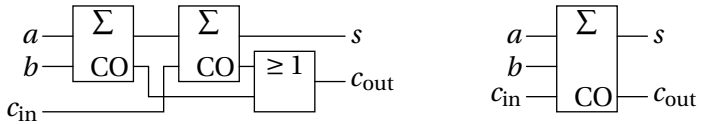


Abbildung 23: Volladdierer

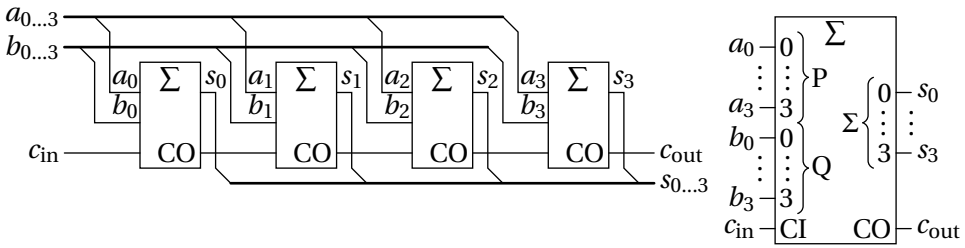


Abbildung 24: 4-Bit-Addierer

ergibt sich:

$$(c_1, s_1) := \mathrm{HA}(a, b)$$
$$(c_2, s_2) := \mathrm{HA}(s_1, c_{\mathrm{in}})$$
$$(c_{\mathrm{out}}, s) := \mathrm{VA}(a, b, c_{\mathrm{in}})$$

$$s \Leftrightarrow s_2$$
$$c_{\mathrm{out}} \Leftrightarrow c_1 \vee c_2$$

a	b	c_{in}	c_1	s_1	c_2	s_2	c_{out}	s
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1
0	1	0	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	0	1	0	1
1	0	1	0	1	1	0	1	0
1	1	0	1	0	0	0	1	0
1	1	1	1	0	0	1	1	1

Abbildung 23 zeigt die Schaltung.

Damit kann man jetzt einen Mehrfach-Bit-Addierer aufbauen. Die Schaltung in Abbildung 24 nennt man **Ripple-Carry-Addierer**, da die Schaltung wegen der Carry-Weitergabe eine Weile braucht, bis sich ein stabiler Zustand einstellt. Da ein Halb-Addierer ein Delay von Δ hat (in der XOR-Ausführung), hat der Volladdierer ein Delay von 2Δ zwischen c_{in} und c_{out}. Ein m-Bit-Addierer hat daher ein lineares Delay von $2m\Delta$ zwischen c_{in} und c_{out}. Es gibt verbesserte Schaltungen, die ein Delay von $\mathcal{O}(\log m)$ erreichen, z.B. die **Carry-Lookahead-Addition**.

3.4 Schaltwerke

In einem Schaltnetz ändern sich die Ausgänge, sobald sich die Eingänge ändern, abgesehen vom Delay, das sich durch das Zeitverhalten der Gatter ergibt. Grob sind dann die Ausgänge $y = (y_0, \ldots, y_{n-1})$ zu jedem Zeitpunkt t eine Funktion der Eingänge $x = (x_0, \ldots, x_{m-1})$:

$$y[t] := f(x[t])$$

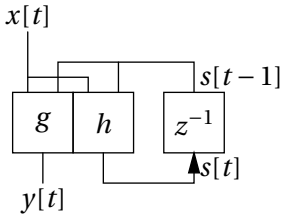


Abbildung 25: Huffman-Modell für Schaltwerke. z^{-1} ist (in der Signalverarbeitung) ein Verzögerungsglied um eine Zeiteinheit.

Wir messen die Zeit im Folgenden diskret, also $t = 0, 1, 2, \ldots$. Die eckigen Klammern deuten wie in der Signalverarbeitung auf ganzzahlige Funktionsargumente t hin.

In **Schaltwerken** soll nun auch ein Zeitverhalten abgebildet werden. Das heißt, dass $y[t]$ nicht nur eine Funktion der aktuellen ($x[t]$), sondern auch vergangener Eingangszustände $x[t-k]$ ist.

$$y[t] := f(x[t], x[t-1], x[t-2], \ldots)$$

Um vergangene Zustände in die Gegenwart zu holen, benötigt man einen Speicher $s = (s_0, \ldots, s_{l-1})$ (Zustand, State, Memory). So ergibt sich das **Huffman-Modell** für Schaltwerke:

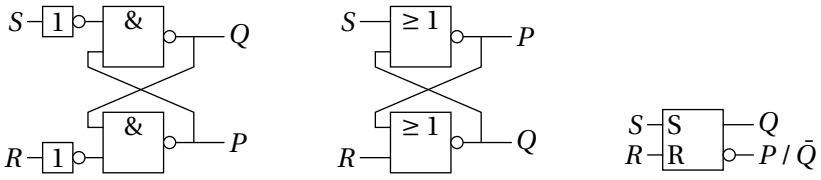
$$y[t] := g(x[t], s[t-1]), \qquad s[t] := h(x[t], s[t-1])$$

Abbildung 25 zeigt dieses Schema. Dass diese Form immer möglich ist, sieht man, wenn man

$$s[t-1] := (x[t-1], x[t-2], \ldots) \quad \Rightarrow \quad s[t] = h(x[t], s[t-1]) := (x[t]) \circ s[t-1]$$

setzt. Das entspricht einem Schieberegister, sofern man jedes Mal das älteste Element $s[t-l]$ entfernt. Die Endlichkeit des Speichers schränkt die Allgemeinheit des Modells ein. Für praktische Anwendungen ist das aber üblicherweise kein großes Problem.

In einem Schaltwerk sind, im Gegensatz zu Schaltwerken, also Zyklen erlaubt und sogar notwendig. Es gibt auch Speicherbausteine. Diese basieren auf Rückkopplungen, also Zyklen. Der einfachste Baustein, der einen Zustand behalten kann, ist das **RS-Latch**. Es wird oft auch RS-Flipflop genannt. Zum Unterschied zwischen Latches und Flipflops später. Abbildung 26 zeigt die Schaltungen in NAND- und NOR-Version.



(a) NAND-Version (b) NOR-Version (c) Schaltsymbol

Abbildung 26: RS-Latches

Da das RS-Latch ein Zeitverhalten hat, betrachten wir den Zustand der Ausgänge zum Zeitpunkt t, $t+\Delta$, $t+2\Delta$, ..., wobei Δ das Delay zwischen Ein- und Ausgang eines NAND-Gatters ist. Die Eingänge betrachten wir vorerst als konstant. Aus der Schaltung (NAND-Version) ergibt sich:

$$Q[t+\Delta] \Leftrightarrow \bar{S} \uparrow P[t] \Leftrightarrow S \vee \overline{P[t]}, \qquad P[t+\Delta] \Leftrightarrow \bar{R} \uparrow Q[t] \Leftrightarrow R \vee \overline{Q[t]}$$

Weiter eingesetzt erhält man:

$$Q[t+2\Delta] \Leftrightarrow S \vee \overline{P[t+\Delta]} \Leftrightarrow S \vee \bar{R}Q[t]$$

Das bedeutet, dass $S \Rightarrow Q$ impliziert (set, Q wird auf 1 gesetzt), $\bar{S}R \Rightarrow \bar{Q}$ (reset, Q wird auf 0 gesetzt), und $\bar{S}\bar{R} \Rightarrow Q[t+2\Delta] \leftrightarrow Q[t]$ (hold, der Zustand von Q wird gehalten). Außerdem sieht man, dass sich ein eventuelles instabiles Verhalten über maximal zwei Δ erstrecken kann, bevor es sich wiederholt. Daher definieren wir Ausgangswerte Q_1, P_1 für gerade Δ-Zeitpunkte, und Q_2, P_2 für ungerade: $Q_1 := Q[t+2k\Delta]$, $P_1 := P[t+2k\Delta]$, $Q_2 := Q[t+(2k+1)\Delta]$, $P_2 := P[t+(2k+1)\Delta]$. Damit können wir die vier Übergangsbedingungen laut Schaltung auf folgende Weise umformen:

$$\begin{aligned}
&(Q_1 \leftrightarrow \bar{S} \uparrow P_2)(P_1 \leftrightarrow \bar{R} \uparrow Q_2)(Q_2 \leftrightarrow \bar{S} \uparrow P_1)(P_2 \leftrightarrow \bar{R} \uparrow Q_1) \\
\Leftrightarrow\ &\ldots \text{mühsame Umformungen} \ldots \\
\Leftrightarrow\ &\underbrace{Q_1\bar{P}_1Q_2\bar{P}_2\bar{R}}_{\text{set}(S)/\text{hold}(\bar{S})} \vee \underbrace{\bar{Q}_1P_1\bar{Q}_2P_2\bar{S}}_{\text{reset}(R)/\text{hold}(\bar{R})} \vee \underbrace{Q_1P_1Q_2P_2RS}_{\text{vermeiden}} \vee \underbrace{(Q_1P_1\bar{Q}_2\bar{P}_2 \vee \bar{Q}_1\bar{P}_1Q_2P_2)\bar{R}\bar{S}}_{\text{instabil (Oszillation)}}
\end{aligned}$$

Wir erhalten wieder den set-Fall ($\bar{R}S$) und den reset-Fall ($R\bar{S}$), und sehen, dass für $\bar{R}\bar{S}$ der Zustand gehalten werden kann. In diesen Fällen gilt immer $P \leftrightarrow \bar{Q}$, weshalb statt P oft einfach $\bar{Q}$ für diesen Ausgang geschrieben wird. Für $\bar{R}\bar{S}$ kann sich aber auch ein instabiler Zustand ergeben, in dem sich QP und $\bar{Q}\bar{P}$ abwechseln (Oszillation). In diesen Zustand gerät das RS-Latch, wenn beide Ausgänge

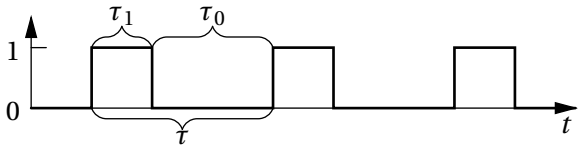
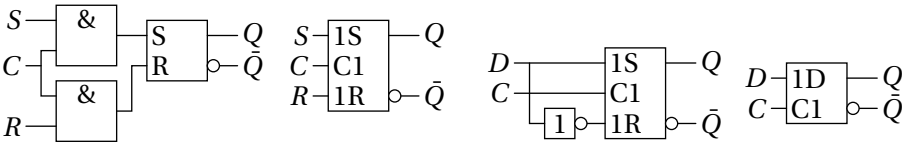


Abbildung 27: Taktsignal



(a) Gated RS-Latch

(b) D-Latch

Abbildung 28: Gated Latches

auf den selben Zustand gesetzt werden. Das passiert bei RS, weshalb diese Eingangsbelegung vermieden werden muss.

Ein Speicherbaustein ist **transparent**, wenn Änderungen der Eingänge sofort zu einer Änderung der Ausgänge führen. Transparenz ist für das Huffman-Modell unbrauchbar, weil sich die Phasen $s[t-1]$ und $s[t]$ unweigerlich vermischen würden, und so chaotische Ergebnisse produzieren würden. Man braucht also einen **Takt** (Clock), also eine Leitung, die in regelmäßigen Abständen eine neue Phase einleitet. Abbildung 27 zeigt ein Taktsignal. Es hat die **Periodendauer** τ, die **Taktfrequenz** $f = 1/\tau$ in Hz und den **Tastgrad** τ_1/τ („Tastverhältnis“ wird uneinheitlich für τ_1/τ oder τ_1/τ_0 verwendet). Wir stellen Schaltungszustände hier in diskreten Zeitzuständen $t = 0,1,2,3,\ldots$ dar, was einer Periodendauer von 1 entspricht. In der Praxis liegt die Periodendauer natürlich meist im Nanosekundenbereich. Beachte auch, dass die Gatter-Delays noch viel kleiner sind (Picosekundenbereich), d.h. $\Delta \ll 1$. Der Takt muss/soll in einem Schaltwerk mit allen Speicherelementen verbunden sein. Dann ist die Schaltung **synchron**. Es gibt aber auch asynchrone Schaltungen.

Das RS-Latch ist leider volltransparent. Ein erster Ansatz wäre, einen Takteingang C (Control, Clock) zu benutzen, um die Eingänge R und S auf 0 (hold) zu zwingen, also $R' := RC$, $S' := SC$. Das führt zu einem **Gated RS-Latch** und zu einem **D-Latch** wie in Abbildung 28. Das D-Latch setzt immer $S \leftrightarrow D$ und $R \leftrightarrow \bar{D}$, d.h. es speichert D, wenn $C \leftrightarrow 1$ ist, und hält sonst den Zustand. Dadurch wird auch

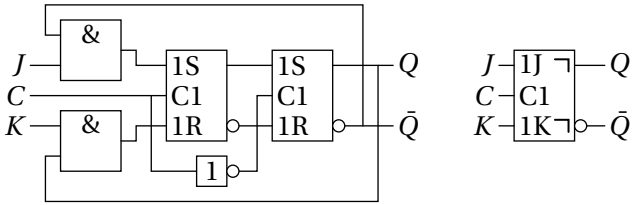


Abbildung 29: Master-Slave JK-Flipflop

automatisch die unerwünschte Eingangsbelegung RS vermieden.

Diese Latches sind in der 0-Phase des Takts nicht mehr transparent, in der 1-Phase aber noch immer. Obwohl es dafür durchaus Anwendungen gibt, braucht man für eine rückkopplungsfähige Schaltung echte **Flipflops**. Hier passiert die Änderung am Ausgang nur zu einem bestimmten Zeitpunkt (Taktflanke). Sie sind also nie transparent. Leider werden vor allem in der deutschen Literatur Latches auch als Flipflops bezeichnet.

Ein Konstruktionsschema für Flipflops ist das Master-Slave-Prinzip. Dabei werden zwei Gated RS-Latches hintereinander geschalten, und das zweite mit einem invertierten Clock-Signal beschaltet. Siehe Abbildung 29. Dadurch wird in der 1-Phase des Takts das erste Latch, das Master-Latch, in den richtigen Zustand gebracht, während das zweite, das Slave-Latch, weiter den alten Zustand hält. Beim Wechsel in die 0-Phase des Takts schaltet das Master-Latch in den Haltemodus. Das Slave-Latch übernimmt den neuen Zustand und lässt ihn auf den Ausgang durch. Durch diese Konstruktion wird wie in einer Schleuse die Einlassphase von der Auslassphase getrennt.

Das **JK-Flipflop** in Abbildung 29 funktioniert ansonsten ähnlich wie das RS-Flipflop, wobei das J (Jump, Jack) dem S und das K (Kill, Kilby) dem R entspricht. Es hat außerdem noch eine Rückkopplung von $\bar{Q}$ zu J und von Q zu K. Das verwandelt die verbotene RS-Eingangsbelegung in einen Toggle-Modus, der den aktuellen

Zustand invertiert, wie man in folgender Tabelle sieht:

$J[t]$	$K[t]$	$Q[t]$	$S[t]$	$R[t]$	$Q[t+1]$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	0	1
1	1	1	0	1	0

Das ergibt folgende Transitionstabelle:

$J[t]$	$K[t]$	$Q[t+1]$
0	0	$Q[t]$
0	1	0
1	0	1
1	1	$\overline{Q[t]}$

Zur Entwicklung von Schaltungen kann man diese Tabelle auch umkehren, um herauszufinden, wie man die Eingänge belegen muss, um erwünschte Zustandsübergänge zu gewährleisten. Die **Ansteuerungstabelle** für JK-Flipflops ist:

$Q[t]$	$Q[t+1]$	$J[t]$	$K[t]$
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Ein zweites Konstruktionsschema für JK-Flipflops sieht man in Abbildung 30. Hier ist der Ansatz, das transparente Zeitfenster so kurz wie möglich zu halten, um den neuen Zustand zu halten, bevor sich die Eingänge wieder ändern. Dazu wird mit Hilfe eines Verzögerungsgliedes (3Δ) ein kurzer Impuls erzeugt, da der invertierte Ausgang des Verzögerungsgliedes 3Δ nach der steigenden Taktflanke 0 wird und den 1-Level des Takts durch das Und-Gatter wieder auf 0 bringt. So ein Verzögerungsglied kann man z.B. durch drei in Serie geschaltete Invertierer (Nicht-Gatter) implementieren. Der Rest ist gleich wie beim Master-Slave-JK-Flipflop, also auch die Transitionstabelle. Diese Variante nennt man **flankengesteuertes JK-Flipflop**.

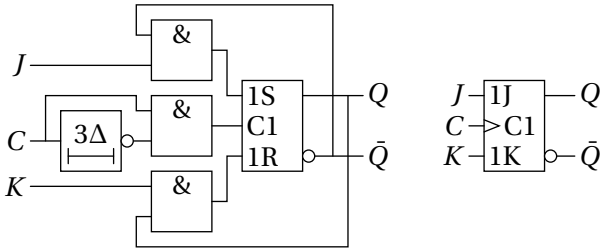


Abbildung 30: Flankengesteuertes JK-Flipflop

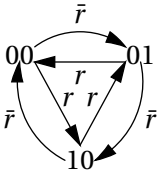
Beachte auch, dass beim Master-Slave-JK-Flipflop die Zustandsänderung am Ausgang bei der fallenden Taktflanke passiert, was durch die Verzögerungs-Markierungen im Schaltsymbol angezeigt wird. Möchte man die Zustandsänderung bei der steigenden Taktflanke haben, muss man das Taktsignal mit einem zusätzlichen Nicht-Gatter invertieren. Das flankengesteuerte JK-Flipflop reagiert auf die steigende Taktflanke. Die Taktflankensteuerung wird durch das zusätzliche Dreieckssymbol im Schaltsymbol dargestellt.

Es gibt übrigens auch RS-Flipflops und D-Flipflops, die rückkopplungsfähig sind.

Um nun mit Hilfe von RS-Flipflops Schaltungen zu entwickeln, überlegt man sich zuerst, welche Zustände die Schaltung haben soll, und bildet jedes Bit dieser Zustände als JK-Flipflop ab. Dann werden die Zustandsübergänge als Wahrheitstabelle in Abhängigkeit der Zustandsbits und weiterer Eingabgsbits dargestellt (d.h. der neue Zustand in Abhängigkeit des alten Zustands). Mit Hilfe der Ansteuerungstabelle entwickelt man die Wahrheitstabelle der J- und K-Eingänge der Flipflops, minimiert diese mittels K-Diagramm oder Quine-McCluskey-Algorithmus, und kann dann die Schaltung angeben.

Zwei Lampen sollen durch Blinken die Richtung links bzw. rechts anzeigen. Die Lampen werden durch zwei JK-Flipflops und deren Ausgänge q_1, q_0 abgebildet. Die Richtung soll durch einen Eingang r (für rechts) ausgewählt werden. Bei Richtung rechts (r) soll zuerst die linke Lampe (q_1) leuchten, dann die rechte, und dann beide dunkel sein, bevor es wieder von vorne beginnt. Bei Richtung links ($\bar{r}$) ist es umgekehrt. Das ergibt das Zustandsdiagramm in Abbildung 31 (a). Der Zustand $q_1 \leftrightarrow q_0$ tritt nicht auf und gilt in der nachfolgenden Schaltungsminimierung als „Egal“-Fall.

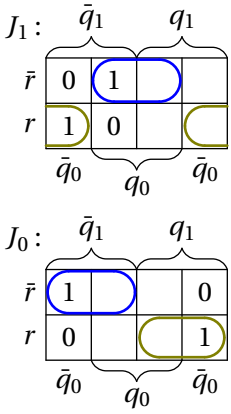
Nun erstellen wir die Transitionstabelle (Abbildung 31 (b)), d.h. zuerst die Wahr-



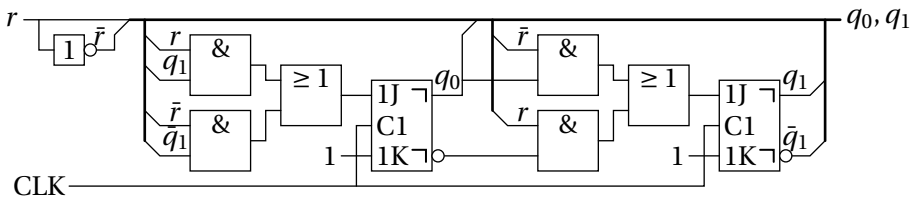
(a) Zustandsübergänge für $q_1 q_0$

q_1	q_0	r	q_1'	q_0'	J_1	K_1	J_0	K_0
0	0	0	0	1	0	X	1	X
0	0	1	1	0	1	X	0	X
0	1	0	1	0	1	X	X	1
0	1	1	0	0	0	X	X	1
1	0	0	0	0	X	1	0	X
1	0	1	0	1	X	1	1	X

(b) Transitionstabelle



(c) K-Diagramme



(d) Schaltung

Abbildung 31: Schaltungsminimierung für Links-Rechts-Blinkanlage

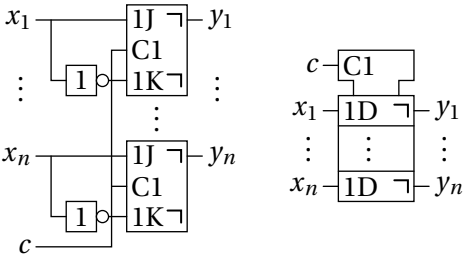


Abbildung 32: Register

heitstabelle für den jeweils nachfolgenden Zustand, der mit q_1', q_0' bezeichnet wird. So ist z.B. in der ersten Zeile $(q_1' q_0') = 01$, weil $r \leftrightarrow 0$ die Richtung links anzeigt, weshalb nach der Dunkelphase $(q_1 q_0) = 00$ die rechte Lampe an sein soll. Die J- und K-Eingänge J_1, K_1, J_0, K_0 ergeben sich dann aufgrund der Ansteuerungstabelle. Z.B. ist in der ersten Zeile der Übergang von $q_1 \mapsto q_1'$ gleich $0 \mapsto 0$. Daher wird J_1, K_1 auf $0, X$ gesetzt.

Der nächste Schritt ist die Minimierung mittels K-Diagramm in Abbildung 31 (c). Sowohl die X-Einträge aus der Tabelle als auch die fehlenden Zeilen bleiben im K-Diagramm leer. Es ergibt sich $J_1 := q_0 \bar{r} \vee \bar{q} r$ und $J_0 := \bar{q}_1 \bar{r} \vee q_1 r$. Für K_1 und K_0 kann man sich das K-Diagramm sparen, denn es ist aus der Wahrheitstabelle direkt ersichtlich, dass $K_1 := K_0 := 1$ die Lösung ist. Die Schaltung sieht man in Abbildung 31 (d).

Ein Register ist ein Baustein, der mehrere Bits speichert. Abbildung 32 zeigt Schaltung und Schaltsymbol. Durch die Belegung $J := x, K := \bar{x}$ wird x in den Speicherzustand übernommen (set wenn x, reset wenn $\bar{x}$). Das entspricht einem D-Flipflop. Die horizontalen Linien im Schaltsymbol trennen unzusammenhängende Blöcke. Der oberste Block mit der Einkerbung ist ein Kontroll-Block. Dessen Labels (in diesem Fall 1) beschalten alle abhängigen Blöcke. Register spielen in CPUs eine bedeutende Rolle. Sie können auch zusätzliche Eingänge haben, die kontrollieren, wann (statt in jedem Takt) ein neuer Wert gespeichert werden soll.

Eine spezielle Art von Register ist das Schieberegister (Shift-Register). Dabei wird ein einzelnes Input-Bit in einem Flipflop gespeichert und bei jedem Taktimpuls an ein nächstes Flipflop weitergegeben. Siehe Abbildung 33. Dadurch sind die Bitzustände der letzten n Taktzyklen verfügbar.

Eine Anwendung von Schieberegistern ist das Codieren von Bitstreams. Sobald etwa 8 Bits an das Schieberegister übergeben wurden, können diese als Byte in

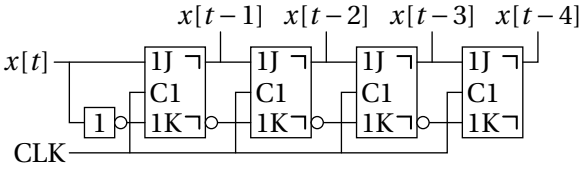
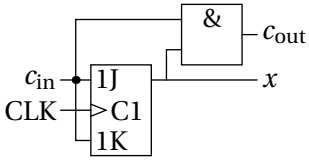


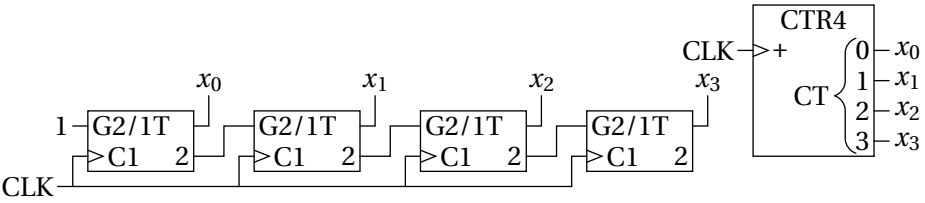
Abbildung 33: Schieberegister

c_{in}	x	c_{out}	x'	J_x	K_x
0	0	0	0	0	X
0	1	0	1	X	0
1	0	0	1	1	X
1	1	1	0	X	1

(a) Transitionstabelle



(b) Zähler-Element



(c) 4-Bit-Zähler

Abbildung 34: Synchroner Binärzähler

einem Speicher (RAM) abgelegt werden. Ein anderes Beispiel ist die Erzeugung pseudozufälliger Bitstreams (Maximum-Length-Sequences). So erzeugt z.B. $x[t] := x[t-3] \oplus [t-4]$ eine Sequenz von Bits, die sich erst nach 15 Bits zu wiederholen beginnt (ausprobieren!). Für längere Schieberegister (Länge n) gibt es komplexere Schaltungen, die Sequenzen der Länge $2^n - 1$ erzeugen.

Ein n-Bit-**Binärzähler** zählt bei jedem Takt einen n-Bit-Wert x um 1 hoch. Das entspricht im Prinzip einer Addition von nur einer Zahl, wobei aber jeweils ein Carry-Bit an die nächste Stelle weitergegeben werden kann. An der Stelle x_0 wird ein konstantes 1-Bit als Carry-Bit addiert. Die Wahrheitstabelle entspricht also einem Halbaddierer, wobei aber das Summenbit der neue Zustand des Bits werden soll. Siehe Transitionstabelle in Abbildung 34 (a). Man sieht auch ohne K-Diagramm schnell, dass $J_x := K_x := c_{\text{in}}$ ist. Demgemäß ergibt sich die Schaltung in Abbildung 34 (b). Im Schaltsymbol bedeutet der Schrägstrich, dass der Ein-

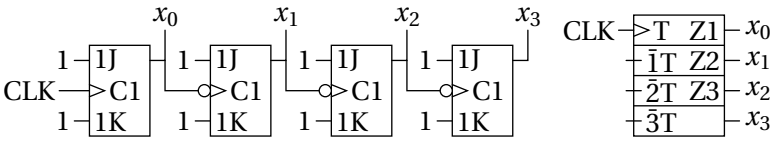


Abbildung 35: Asynchroner Zähler

gang sowohl G2 als auch 1T bewirkt. Das T steht für „Toggle“. Ein T-Flipflop hat nur die Operationen „Hold“ (Eingang auf 0) und „Toggle“ (Eingang auf 1), was einem J-K-Flipflop mit zusammengeschalteten J- und K-Eingängen entspricht.

Die Zähler-Elemente werden dann wie in Abbildung 34 zusammengeschaltet. Das c_{out} einer Stufe wird mit dem c_{in} der nächsten Stufe verbunden. Für das Schaltsymbol gibt es einen eigenen „CTR“-Typ. Der (flankengesteuerte) Eingang „+“ erhöht den internen Zählerzustand „CT“ um 1, der mit Hilfe der geschwungenen Klammer als Binärwert ausgegeben wird. Zähler können diverse zusätzliche Eingänge haben, wie z.B. einen „CT=0“-Eingang, der den Zähler auf 0 zurücksetzt. In der Schaltung wäre das z.B. mit einer Oder-Verknüpfung bei allen K-Eingängen machbar. Es gibt aber auch J-K-Flipflops mit extra Reset-Eingängen.

Neben dem synchronen Zähler gibt es auch einen **asynchronen Zähler**, der etwas einfacher im Aufbau ist, aber ein schlechteres Zeitverhalten hat. Siehe Abbildung 35. Hier sind die Clock-Eingänge der Folgestufen mit den negierten x-Ausgängen der vorherigen Stufen verbunden. Die J-K-Eingänge sind fix auf „Toggle“ gelegt. Das bedeutet, dass bei fallender Taktflanke der Vorstufe die nachfolgende Stufe den Zustand ändert. Das ist korrekt, weil, wie in Abbildung 34 (a) zu sehen, c_{out} in der Vorstufe genau dann 1 ist, wenn $x \mapsto x'$ von $1 \mapsto 0$ wechselt, und c_{in} der Folgestufe dann auch 1 wird und das „Toggle“ auslöst.

Der asynchrone Zähler heißt oft auch Ripple-Counter. Der Grund für beide Benennungen ist, dass Folgestufen erst schalten, wenn die Vorstufe geschaltet hat, also asynchron mit mehreren Δ Verzögerung. Falls also mehrere oder alle Flipflops im 1-Zustand sind, dann fallen diese nacheinander wie Dominosteine auf 0. Das bezeichnet man als „Ripple“. Die Verzögerungen können sich negativ auf die restliche Schaltung, in die der Counter integriert ist, auswirken und potentiell die Taktrate verringern.

Im Schaltsymbol findet der Operator Z Anwendung, der einfach eine Schaltverbindung darstellt, d.h. Eingänge, die mit dem jeweiligen Label (z.B. 1) versehen sind, werden mit dem Ein-/Ausgang verbunden, der mit Z gekennzeichnet ist

(z.B. Z1). Der Überstrich $\bar{1}$ bewirkt wie üblich die Negation. Weiters zeigt der kurze Strich durch den Eingang eines Blocks eine interne Verbindung an.

Index