Einfürhung in die Informatik

Jan

Inhaltsverzeichnis

1 Zahlensysteme

Zahlensysteme zeichnen sich durch den so genannten *Radix* oder auch die *Basis* aus. Das für die meisten Menschen geläufige Zahlensystem ist das Dekadische. Diese hat einen Radix von 10.

1.1 Konvertierungen

Beim konvertieren einer Zahl in eine neue Zielbasis kann stets der Weg über das 10er System gewählt werden. Hierzu wird das *Horner-Schema* genutzt.

Eine Ausnahme bilden die Basen, die ein vielfalches von 2 sind, hierbei kann direkt und ohne Umweg über das 10er System umgerechnet werden.

1.1.1 Beliebig in das $10er$ System

Eine beliebige Basis in das Zehnersystem funktioniert **immer** wie folgt:

(1) Indizies anschreiben (Zahlen durchnummerieren)

(2) Basis (Radix) hoch Index anschreiben

(3) Aufsummieren

Beispiel 1

$$101,1_{[2]}$$

Konvertieren vom 2er ins 10er System.

(1) Indizies anschreiben: $\overset{2}{1}\overset{1}{0}\overset{0}{1},\overset{-1}{1}_{[2]\leftarrow \text{Basis}}$

(2) Basis (Radix) hoch Index anschreiben: $\overbrace{\mathbf{1} * \underbrace{2}_{\text{Basis}}{}^{2} + \mathbf{0} * \underbrace{2}_{\text{Basis}}{}^{1} + \mathbf{1} * \underbrace{2}_{\text{Basis}}{}^{0}}^{\text{Vorkomma}} + \overbrace{\mathbf{1} * \underbrace{2}_{\text{Basis}}{}^{-1}}^{\text{Nachkomma}}$

(3) Aufsummieren: $5,5_{10}$

1.1.2 $10er$ System in beliebiges System

Die Konvertierung erfolgt durch Restwert-Division mit der jeweiligen Basis. Anschließend wird der Restwert von unten nach oben gelesen bei **Vorkommastellen** und von oben nach unten bei **Nachkommastellen**.

Beispiel 2 Vorkommastellen

$$91_{[10]}$$

Konvertieren vom 10er ins 2er System.

$$
\begin{array}{lcccrl}
91 & \div & \overbrace{2}^{\text{Basis}} & = & 45 & \mid \; 91 \bmod 2 = \; 1 \quad \text{Rest} \\
45 & \div & 2 & = & 22 & \mid \; 45 \bmod 2 = \; 1 \quad \text{Rest} \\
22 & \div & 2 & = & 11 & \mid \; 22 \bmod 2 = \; 0 \quad \text{Rest} \\
11 & \div & 2 & = & 5 & \mid \; 22 \bmod 2 = \; 1 \quad \text{Rest} \\
5 & \div & 2 & = & 2 & \mid \; 5 \bmod 2 = \; 1 \quad \text{Rest} \\
2 & \div & 2 & = & 1 & \mid \; 2 \bmod 2 = \; 0 \quad \text{Rest} \\
1 & \div & 2 & = & 0 & \mid \; 1 \bmod 2 = \; 1 \quad \text{Rest} \\
 & & & & & \mid \qquad\qquad\quad \uparrow \; \text{von unten nach oben}
\end{array}
$$

$\rightarrow 1011011_{[2]}$

Beispiel 3 Nachkommastellen

$$0,375_{[10]}$$

Konvertieren vom 10er ins 2er System.

$$
\begin{array}{lcccrlc}
 & & & & & & \downarrow \\
0,375 & * & \overbrace{2}^{\text{Basis}} & = & 0,75 & \mid \text{Vorkomma} & 0 \\
0,75 & * & 2 & = & 1,5 & \mid \text{Vorkomma} & 1 \\
0,5 & * & 2 & = & 1,0 & \mid \text{Vorkomma} & 1 \\
 & & & & & \text{ab hier bleibt} & 0
\end{array}
$$

$\rightarrow$ von oben nach unten ablesen 0,**011**

Beispiel 4 Vor- und Nachkommastellen

$$91,375_{[10]}$$

Konvertieren vom 10er ins 2er System.

Vorkommastellen wie oben angeführt berechnen, Nachkommastellen wie oben angeführt berechnet. Anschließend lautet das Ergebnis: **1011011,011**$_{[2]}$

1.1.3 Beliebig in das $10er$ System mit Tabelle

Für das Horner Schema gibt es eine *Schema F* Variante die sich einer Tabelle bedient.

Als Beispiel hierfür dient uns $5423_{[8]}$. Diese Zahl wollen wir aus dem Oktal (8er)-System ins Dezimal-System konvertieren.

Erster Schritt. Als erstes Stellen wir Tabelle auf. Hierzu geben wir bei x unsere Basis (Radix) an. In diesem Fall 8. Anschließend werden die Spalten mit den Ziffern der Zahl ausgefüllt

x = 8	5	4	2	3

Zweiter Schritt. Im zweiten Schritt notieren wir die 0 unter dem 5er in der ersten Spalte, da die Zahl links neben dem 5er eine Null ist (**0**5423).

$x = 8$	5	4	2	3
	0			

Dritter Schritt. Im dritten Schritt addieren wir die Spalte ($5 + 0 = 5$).

$x = 8$	5	4	2	3
	0			
	5			

Vierter Schritt. Nun benötigen wir unser x, also unsere Basis (Radix). Wir multiplizieren das Ergebnis mit unserer Basis ($5 * 8 = 40$) und tragen es unter der nächsten Ziffer (4) ein.

$x = 8$	5	4	2	3
	0	40		
	5			

Weitere Schritte. Nun addieren wir wieder die 4 mit der Zahl darunter. Diese Schritte wiederholen wir für den Rest der Tabelle bis diese ausgefüllt ist.

$x = 8$	5	4	2	3
	0	40		
	5	44		

Finale Tabelle. In der finalen Tabellle finden wir in der rechten Spalte in der letzten Zeile das Ergebnis.

$x = 8$	5	4	2	3
	0	40	352	2832
	5	44	354	**2835**

1.1.4 Hexadezimal

Bei der Konviertierung zu Hexadezimal ist es günstig sich eine Tabelle anzufertigen mit der leichter Konvertiert werden kann.

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

Anschließend kann man wenn nun die 11 rauskommt aus der Tabelle ablesen das dies mit B ersetzt gehört.

1.1.5 Vielfache von 2 (Binär, Oktal, Hexadezimal)

Systeme die als Basis ein Vielfaches von 2 haben lassen sich ohne Umweg über das 10er System rechnen. Diese Technik heißt „Gruppieren“.

Hierfür erfasst man die Ziffern in Gruppen und konvertiert direkt.

Binär Gruppe	Hex	Binär Gruppe	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Beispiel 5

$$1010101101001_{[2]}$$

Konvertieren in Hexadezimal$_{[16]}$

(1) Gruppen auffüllen das jede Gruppe 4 Breit ist (links). $\overbracket{{\color{green}000}1}\,\overbracket{0101}\,\overbracket{0110}\,\overbracket{1001}$

(2) Jede Gruppe ersetzen. $\overbracket{1}\,\overbracket{5}\,\overbracket{6}\,\overbracket{9}$

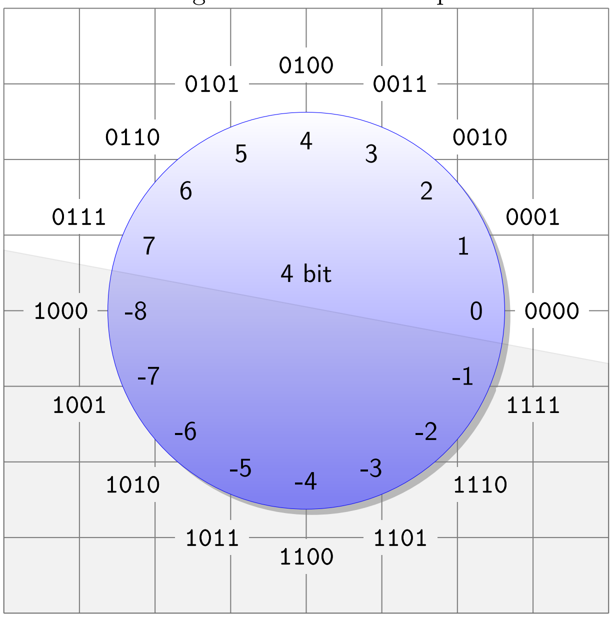
(3) $1010101101001_{[2]} = 1569_{[16]}$

1.2 Zweierkomplement 2_k

Das Zweierkomplement teilt eine Zahl fixer Breite (gleichviele Stellen) in eine positive und eine negative Seite.

Nachfolgendes Besipiel illustriert dies mit einer 4 Bit breiten Zahl, sprich eine Binärzahl die fix 4 Stellen hat.

Abbildung 1: 4 Bit Zweierkomplement



Der Vorteil diese Verfahrens ist das man negative Zahlen ohne extra Vorzeichenbit darstellen kann (auch die +/- 0 Thematik fällt weg). Des Weiteren benötigt man für Addition und Subtraktion nur mehr eine Operation. Um Zahlen zu subtrahieren addiert man lediglich die negative Variante der Zahl (das Komplement).

1.2.1 2er System in Zweierkomplement 2_k

Sollte die Zahl im 10er System vorliegen ist diese zuerst in das 2er System umzuwandenln.

Um vom Dualsystem in das Zweierkomplement zu konvertieren müssen wir zuerste die Zahl auf eine fixe Breite (auch „Wortbreite“, fixe Anzahl von Stellen) bringen.
Die Breite ist durch das Zweierkomplement vorgegeben (z.b. 8 Bit 2k heißt 8 Stellen, 4 Bit 2k heißt 4 Stellen).
Hat die Zahl bereits mehr Stellen als das Zweierkomplement verlangt so können wir sie nicht ins Zweierkomplement konvertieren.
Hat die Zahl weniger Stellen so ergänzen wir links mit 0en.

Folgende Schritte werden benötigt um ins Zweierkomplement zu konvertieren:

(1) In Dualzahl umrechnen

(2) Links mit Nullen auffüllen (auf Breite)

(3) Bitweise Komplement bilden

(4) Binär 1 addieren

Beispiel 6 Konvertieren in 2_k

$$-6_{[10]}$$

Konvertieren vom 10er ins 4-Bit 2k System.

(1) Zuerst ins 2er System umrechnen: $6_{10} = \overbrace{6 \bmod 2}^{0}, \overbrace{3 \bmod 2}^{1}, \overbrace{1 \bmod 2}^{1\leftarrow} = 110_2$.

(2) Auf 4-Bit auffüllen: **0**110

(3) Komplement bilden (alle Bits umdrehen): 0110 wird zu 1001

(4) Anschließend +1 rechnen: 1001 + 0001 = 1010

$-6_{10} = -110_2 = 1010_{2k}$

Schnellverfahren. Es gibt einen kleinen Trick um schneller zu konvertieren. Hierfür sucht man sich von links weg den ersten 1er. Anschließend dreht man alle nachfolgenden Bits um.

Beispiel 7 Schnellverfahren: Bitflip

$$10_{[10]}$$

Konvertieren vom −10er ins 6-Bit 2k System.

(1) Zuerst ins 2er System umrechnen: $-10_{10} = -1010_2$.

(2) Stellen ergänzen auf 6-Bit 001010.

(3) „Bitflip“

$$\begin{array}{cccccc}
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & \overbrace{1}^{\text{erste 1 von links}} & 0 \\
0 & 0 & 1 & \mathbf{1} & 1 & 0 \\
0 & 0 & \mathbf{0} & 1 & 1 & 0 \\
0 & \mathbf{1} & 0 & 1 & 1 & 0 \\
\mathbf{1} & 1 & 0 & 1 & 1 & 0
\end{array}$$

$-001010_2 = 110110_{2k}$

1.2.2 Zweierkomplement 2_k zurückrechnen

Nachdem bei dem Zweierkomplement stets das höchstwertige Bit negativ ist so kann man es enstprechend mit dem Hornerschema berechnen in dem man das höchste Bit negativ rechnet.

Beispiel 8

$$11111100_{2k}$$

Konvertiere vom 8-Bit 2_k ins 10er System.

$$11111100_{2k} = \overbrace{1 * (-2^7)}^{\text{höchstwertiges Bit}} + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = -4_{10}$$

Alternativ kann man auch wieder „Bitflippen“ (inverterieren oder auch Komplement bilden) und binär 1 addieren.

$1111100_{2k} \rightarrow 00000011 \rightarrow 00000011 + 00000001 = 00000\overset{210}{100}_{[2]} = 1 * 2^2 = 4_{[10]}$
höchstes Bit war 1 daraus folgt $-4_{[10]}$

Es funktioniert auch wieder der Trick mit dem ersten 1er suchen und danach alle Bits umdrehen.

$$\overset{-}{\downarrow}11111100_{2k} = 11111\overbrace{1}^{\text{erster 1er von links}}00_{2k} = \overbrace{11111}^{\text{umdrehen}}100_{2k} = 00000100_{[2]} = 1 * 2^2 = -4_{[10]}$$

1.3 Rechenoperationen

1.3.1 Dualsystem

Addition Addition im Dualsystem funktioniert gleich wie im dekadischen System.

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 0$$
$$1 + 1 = 10$$

Beispiel 9 Addition im Dualsystem

Berechne $201 + 255$ im 10er und im 2er System.

$$\begin{array}{r} 201 \\ +\,255 \\ \hline 456 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\,1111\,111} \\ 0\,1100\,1001 \\ +\,0\,1111\,1111 \\ \hline 1\,1100\,1000 \end{array}$$

Subtraktion Bei Subtraktion muss unterschieden werden ob die resultierende Zahl negativ wird, sollte dies nicht der Fall sein so kann wie gewohnt aus dem dekadischen System subtrahiert werden.
Ist dies **nicht** der Fall, so muss der Umweg über das Zweierkomplement genommen werden.

Subtraktion in $_2$

$$0 - 0 = 0$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$
$$0 - 1 = 1*$$

* geborgt von nächst höherer Stelle (von links nach rechts), das heißt 0-1 führt zu einem Übertrag von 1.

Beispiel 10 Subtraktion im Dualsystem

Berechne $249 - 38$ im 10er und im 2er System.

$$\begin{array}{r} 249 \\ -\,38 \\ \hline 211 \end{array} \qquad\qquad \begin{array}{r} {\scriptstyle 1\,1} \\ 11111001 \\ -\,100110 \\ \hline 11010011 \end{array}$$

Subtraktion in $_2$k Im Zweierkomplement muss nur die negative Nummer zur positiven Nummer addiert werden. Sinnbildlich funktioniert das Zweierkomplement beim subtrahieren so: $3 + (-2) = 1$

Beispiel 11 Subtraktion in $_2$k

Berechne $5_{10} - 11_{10}$ im Zweierkomplement.

(1) Zuerst ins 2er System umrechnen: $5_{10} = 101_2$ und $-11_{10} = -1011_2$.

(2) Auf feste Breite ergänzen (eins mehr für - *Sign Extension*) $101 \longrightarrow 00101$ $-1011 \longrightarrow -01011$.

(3) „Bitflip“ nach erstem einer von Links $-\overline{0101}1_2 = \overline{1010}1_{2k}$

(4) Addieren

$$\begin{array}{r} {\scriptstyle 1\;\;1} \\ 00101 \\ +\,10101 \\ \hline 11010 \end{array}$$

(5) 11010_{2k} höchstes Bit 1 also negativ, $-\overset{\text{Flip}}{\overbrace{001}^{110}}\,10$

(6) $-00110_2 = -6_{10}$

Beispiel 12 Subtraktion in $_2$k

Berechne $-1_{10} - 15_{10}$ im Zweierkomplement.

(1) Zuerst ins 2er System umrechnen: $-1_{10} = -1_2$ und $-15_{10} = -1111_2$.

(2) Auf feste Breite ergänzen $-1111 \longrightarrow -01111$ $1 \longrightarrow 00001$.

(3) „Bitflip“ nach erstem einer von Links $-\overline{0111}1_2 = \overline{1000}1_{2k}$ $-\overline{0000}1_2 = \overline{1111}1_{2k}$

(4) Addieren

$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1} \\ {\scriptstyle 1}11111 \\ +\,10001 \\ \hline 110000 \end{array}$$

(5) Achtung 1 Übertrag: Bei 5 Bits müssten wir den Übertrag wegschneiden (Overflow/Underflow).

(6) Diesmal direkt mit Horner: $\overbrace{1}^{-}\overset{43210}{10000} = 1 * \overbrace{-}^{\text{VZ}}(2^4) + 0 * 2^3 + 0 * 2^2 0 * 2^1 + 0 * 2^0 = -16_{[10]}$

Multiplikation Multiplikation im Dualsystem funktioniert gleich wie im dekadischen System.

Beispiel 13 Multiplaktion im Dualsystem

Berechne $24 * 11$ im 10er und im 2er System.

```
  24*11
 ------
  24
 +24
 ------
  264
```

```
   11000*1011
  ------------
   11000
   ¹11000
   ¹00000
 + ¹11000
  ------------
   100001000
```

Division

Beispiel 14 Division im Dualsystem

Berechne 200/8 im 10er und im 2er System.

```
 →
 200÷8 = 25
 ----------
 16
 ----------
  40
  40
 ----------
   0
```

```
 11001000÷1000 = 11001
 ---------------------
 1000
 ---------------------
  1001
  1000
 ---------------------
     1000
     1000
 ---------------------
        0
```

1.3.2 andere Systeme

Info: Bei anderen Systemen einfach aufs 10er System konvertieren, berechnen und wieder zurück.

1.4 Gleitkommadarstellung

IEEE 754 (*IEEE Standard for Floating-Point Arithmetic*) ist ein Standard um Gleitkommadarstellung in Binär zu repräsentieren.

Es gibt zwei ausführung wobei die *Single Precision* Ausführung 32 Bit hat. Die *Double Precision* Ausführung 64 Bit.

Das höchste Bit ist stets für das Vorzeichen reserviert. Danach folgt der Exponent (oder auch *Charakteristik* genannt) und die Mantisse (*fraction*).

Abbildung 2: Single Precision - 32 Bit

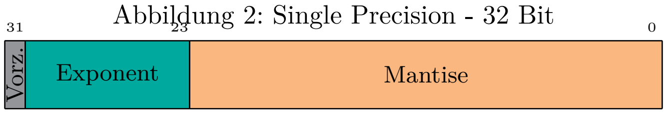
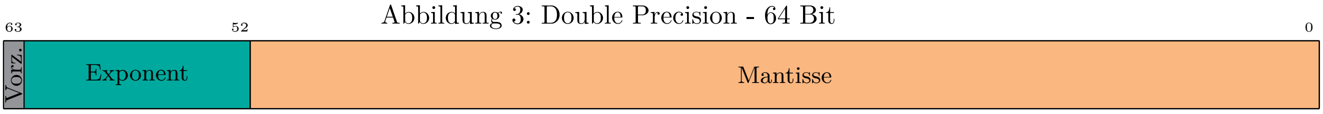


Abbildung 3: Double Precision - 64 Bit



1.4.1 In duale Gleitkommadarstellung umrechnen

Für uns derzeit am relevantesten ist die Umrechnung vom dekadischen (10er System) in duale Gleitkommadarstellung.

Der Ablauf der Umrechnung ist wie folgt:

(1) Vorkommastellen umrechnen ($_{[10]}$ auf $_{[2]}$)

(2) Nachkommastellen umrechnen ($_{[10]}$ auf $_{[2]}$)

(3) Normieren (manchmal auch *Normalisieren* genannt) - also die Mantisse ermitteln. Beim normieren verschiebt man die Zahl so das vor dem Komma nur mehr 1 steht.

(4) Exponent (Charakteristik) errechnen.

(5) Vorzeichen-Bit richtig setzen (Positiv = 0, Negativ = 1)

Beispiel 15

Berechne die duale Gleitkommadarstellung von $18,4$ mit einer Genauigkeit von 32-Bit.

(1) Vorkommastellen umrechnen $18_{[10]} = 10010_{[2]}$)

(2) Nachkommastellen umrechnen $0,4_{[10]} = 0,01100110011_{[2]}$

(3) Normieren

 (1) $10010,0110011 * 2^0$ Komma verschieben durch Hochzahl

 (2) $1,00100110011 * 2^4$ nun nur mehr die 1 vorne - normiert

(4) Exponent (Charakteristik) errechnen.

 (1) Wir haben einfache Genauigkeit (32 Bit) das heißt wir haben einen Bias von 127

 (2) Charakteristik = Exponent (oben beim normieren ermittelte Hochzahl) + Bias = $4 + 127 = 131$

 (3) $131_{[10]} = 10000011_{[2]}$

(5) Vorzeichen-Bit Positiv = 0

(6) In „Bitfeld“ eintragen

 (1) Vorzeichen = 0

 (2) Exponent = 10000011

 (3) Mantisse = $1,\overbrace{00100110011}*2^4$

31	23	0
Vorz.	Exponent	Mantisse
0	10000011	00100110011

1.5 BCD (Binary Coded Decimal)

BCD (Bianry Coded Decimal) ist eine Repräsentation in der jede Ziffer als Zahl dargesteltl wird. Man kann sich das so verstellen das es einfach eine Tabelle gibt wo für jede Ziffer die zugehörige Zahl steht.

Aus dieser Tabelle wird anschließend jede Ziffer gelesen um die Zahl zu konstruieren.

$$12,3_{[10]} = (\overbracket{0001}\overbracket{0010},\overbracket{0011})_{\mathrm{BCD}}$$

Zahl	Ziffer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

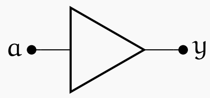
2 Schaltalgebra

Gleich

Symbole $=$
Bauteil *Buffer*

$$y = a$$

a	y
0	0
1	1

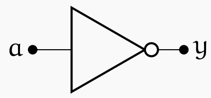


Nicht

Symbole $\neg$
Bauteil *Not*

$$y = \overline{a}$$

a	y
0	1
1	0

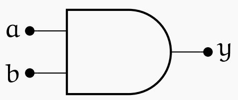


Und

Symbole $\wedge, \&, *$
Bauteil *And*

$$y = a * b$$

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

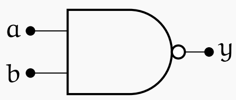


Nicht Und

Symbole $\neg\wedge, \overline{*}$
Bauteil *Nand*

$$y = \overline{a * b}$$

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

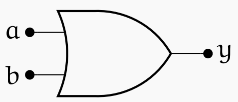


Oder

Symbole $\vee, |, +$
Bauteil *Or*

$$y = a + b$$

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

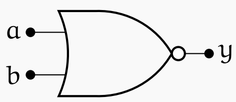


Nicht Oder

Symbole $\neg\vee, \overline{+}$
Bauteil *Nor*

$$y = \overline{a + b}$$

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0

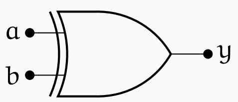


Exklusives Oder

Symbole $\oplus, \mathsf{XOR}$
Bauteil *Xor*

$$y = a \oplus b$$

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

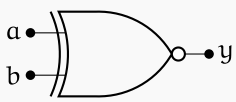


Nicht Exklusives Oder

Symbole $\neg\mathsf{XOR}, \overline{\oplus}$
Bauteil *Xnor*

$$y = \overline{a \oplus b}$$

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1



2.1 Boolsche Algebra

Boolsche Algebra definiert sicht auf Wahrheitswerten, anders als bei Zahlen kann eine Variable hier nur den Wert *Wahr* (oder auch 1) oder *Falsch* (oder auch 0) annehmen.

Die Operationen auf diese Werte sind:

UND: $*$ bzw. $\wedge$ *Und* hat wie bereits oben illustriert die Wahrheitstabelle

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

und wird nur dann wahr wenn beide Seiten wahr sind.

Und ist ein **binärer** Operator das heißt das es zwei Eingaben braucht. Um dies Anhand eines Beispieles zu illustrieren: $\overbrace{A}^{\text{erstes Argument}} \wedge \overbrace{B}^{\text{zweites Argument}}$.

ODER: $+$ bzw. $\vee$ *Oder* ist ebenfalls ein **binärer** Operator und wird dann wahr wenn entweder eine Variable oder beide Variablen wahr sind wie die Wahrheitstabelle zeigt:

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

NICHT: $\overline{x}$ bzw. $\neg$ *Nicht* ist ein **unärer** Operator, das heißt es braucht nur einen Wert, z.B. $\neg \overset{\text{ein Argument}}{A}$. Oft wird das *Nicht* in elektrotechnik und verwandten Feldern oft durch einen Strich dargestellt. Dies ist so wie eine Klammer mit dem $\neg$ zu verstehen. Als Beispiel $\neg(A \wedge B) = \overline{A \wedge B}$.

Vorrang (*Präzedenz*) Wie in der „normalen“ Algebra gibt es auch hier Vorrangregeln (Punkt-Vor-Strich). Die Reihenfolge ist hier $\neg$ (*Nicht*) hat den höchsten Vorrang (Klammerwirkung!) danach $\wedge$ (*Und*) (Quasi Punkt vor Strich) und danach $\vee$ (*Oder*).

Info: Die Erklärung von Wahrheitstabellen ist ausgelassen.

2.2 Rechenregeln

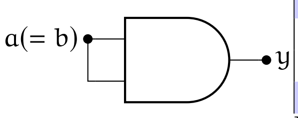
2.2.1 Idempotenz

Idempotenz kann man so verstehen das egal was man macht das Gleiche rauskommt. In dem Kontext von Schaltalgebra ist Idempotenz der Regelsatz für die Kombination der selben Variable.

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Markierte Zeilen wo $a = a$

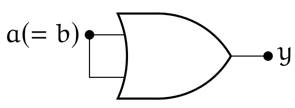
$y = a * a$



a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

Markierte Zeilen wo $a = a$

$y = a + a$



Info: Daraus können wir schließen: wenn $y = a * a$ oder $y = a + a$ dann ist $y = a$

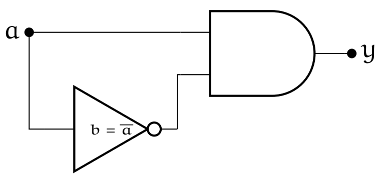
2.2.2 Komplementär

Die Komplementär Regel folgt dem selben Prinzip wie die Idempotenz-Regel nur das die Variable mit ihrer negierten Version kombiniert wird.

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Markiert $a \wedge \overline{a}$

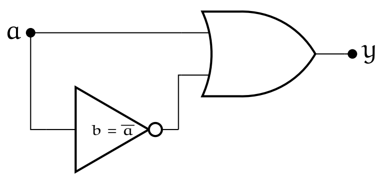
$y = a * \overline{a}$



a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

Markiert $a \vee \overline{a}$

$y = a + \overline{a}$



Info: Daraus können wir schließen:

$$y = a * \overline{a} = 0 \longrightarrow a \wedge \overline{a} = false$$

und

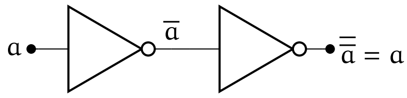
$$y = a + \overline{a} = 1 \longrightarrow a \vee \overline{a} = true$$

2.2.3 Involution

Bei der *Involution* geht es daraum das sich zwei *Nicht* aufheben.

$$a = \overline{\overline{a}}$$

a	$\overline{a}$	$\overline{\overline{a}}$
0	1	0
1	0	1

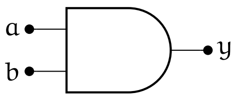


Info: Daraus können wir schließen: $a = \overline{\overline{a}} = \overline{\overline{\overline{\overline{a}}}}$.
Eine **gerade** Anzahl von Verneinungen hebt sich immer auf.

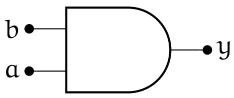
2.2.4 Kommutativ

Und und *Oder* sind kommutativ. Das heißt die Reihenfolge der Argumente ist egal, es führt zu dem selben Ergebnis.

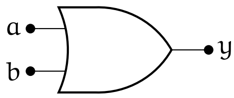
$y = a * b = b * a$



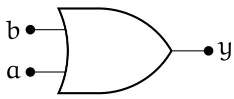
selbes Ergebnis wie



$y = a + b = b + a$



selbes Ergebnis wie



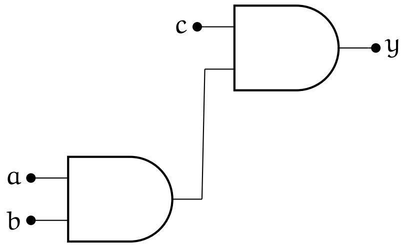
Info: Daraus können wir schließen: $a + b = b + a$ und $a * b = b * a$

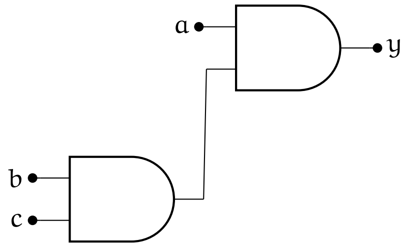
2.2.5 Assoziativ

Assoziativ sagt das die Reihenfolge in der ein Ausdruck ausgewertet wird das Ergebnis nicht beeinflußt. Sowohl *Und* als auch *Oder* sind *Assoziativ*.

$$y = (a * b) * c \qquad y = a * (b * c)$$

Und

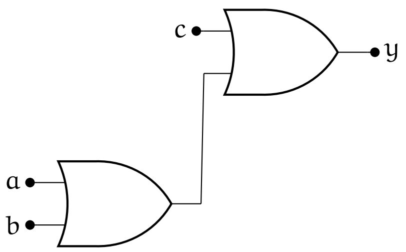


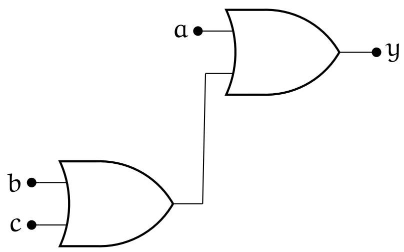


a	b	c	$a \wedge b$	$b \wedge c$	$a \wedge c$	$(a \wedge b) \wedge c$	$(b \wedge c) \wedge a$	$(a \wedge c) \wedge b$
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0
1	0	1	0	0	1	0	0	0
1	1	1	1	1	1	1	1	1

$$y = (a + b) + c \qquad y = a + (b + c)$$

Oder





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

Info: Daraus können wir schließen:

$$a * b * c = (a * b) * c = a * (b * c)$$

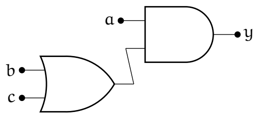
und

$$a + b + c = (a + b) + c = a + (b + c)$$

2.2.6 Distributivität

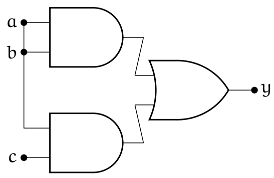
Erste Regel Die erste Distributivitätsregel kennt man ebenfalls aus der „normalen“ Algebra. Distributivität ist die Eigenschaft einer Operation sich auf andere zu „Verteilen“ (distrubte), dies ist am besten mit einem Beispiel zu verstehen: $6*(5+2) = (6*5)+(6*2)$. Die Multiplikation lässt sich über die Addition „verteilen“, sie ist *distributiv*.
In der booleschen Algebra lässt sich das *Und* über das *Oder* „verteilen“ (das erste distributive Gesetz).

$$y = a * (b + c)$$



a	b	c	$b \vee c$	y
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

$$y = (a * b) + (a * c)$$



a	b	c	$a \wedge b$	$a \wedge c$	y
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

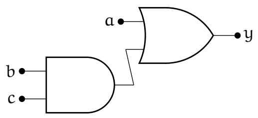
Wir sehen die Spalte y von beiden Tabellen ist gleich, damit ist die Schaltung gleichwertig.

Info: Daraus können wir schließen:

$$a * (b + c) = (a * b) + (a * c)$$

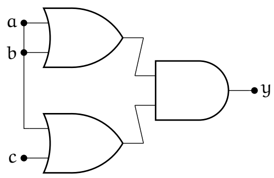
Zweite Regel Das zweite Distributivitätgesetzt kennen wir nicht aus der „normalen“ Algebra, es besagt das wir auch *Oder* über *Und* verteilen können. Dieses Gesetzt funktioniert nicht in der „normalen“ Algebra da: $6 + (5 * 2) \neq (6 + 5) * (6 + 2)$
Dies funktioniert aber in der boolschen Algebra, obwohl *Oder* eine niedrigere Präzedenz hat.

$$y = a + (b * c)$$



a	b	c	$b \wedge c$	y
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$y = (a + b) * (a + c)$$



a	b	c	$a \vee b$	$a \vee c$	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Wir sehen die Spalte y von beiden Tabellen ist gleich, damit ist die Schaltung gleichwertig.

Info: Daraus können wir schließen:

$$a + (b * c) = (a + b) * (a + c)$$

2.2.7 Vereinfachungsregeln

Diese Regeln sind nicht mehr Schaltungen visualisiert.

Folgende Regeln führen zur Reduktion auf ein *Buffer*-Bauteil, sprich der Wert ist nur a

$$a + (a * b) = a$$
$$a * (a + b) = a$$
$$(a * b) + (a * \overline{b}) = a$$
$$(a + b) * (a + \overline{b}) = a$$

Folgende Regeln führen zur Reduktion auf ein *Oder*-Bauteil, sprich der Wert ist nur $a + b$

$$a + (\overline{a} * b) = a + b$$

Folgende Regeln führen zur Reduktion auf ein *Und*-Bauteil, sprich der Wert ist nur $a * b$

$$a * (\overline{a} + b) = a * b$$

2.2.8 De-Morgan

De-Morgan Transformation

(1) Tausche alle $*, \wedge, \&$ Operatoren gegen $+, \vee, \|$ Operatoren und umgekehrt

(2) Inverte alle Variablen (auch 0, $\mathtt{false}$ wird 1, $\mathtt{true}$ und umgekehrt)

(3) Inverte die gesamte Funktion

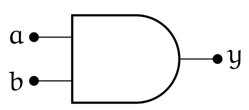
(4) Wende die *Involution* an, sprich doppelte Negierungen heben sich auf

De-Morgan $y = a * b$

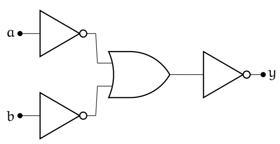
$$a * b$$
$$a + b$$
$$\overline{a} + \overline{b}$$
$$\overline{\overline{a} + \overline{b}}$$

De-Morgan $y = \overline{\overline{a} + \overline{b}}$

$$\overline{\overline{a} + \overline{b}}$$
$$\overline{\overline{a}} * \overline{\overline{b}}$$
$$a * b$$



a	b	y
0	0	0
0	1	0
1	0	0
1	1	1



a	b	$\overline{a}$	$\overline{b}$	$\overline{a} + \overline{b}$	y
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

Die y Spalten sind ident, damit sind auch die Schaltungen gleichwertig.

Info: Diese Transformation ist vorallem für realisiere ausschließlich mit NAND oder NOR-Gattern Aufgaben besonders wichtig.

2.2.9 Zusammenfassung

$$a = \overline{\overline{a}}$$

$$a * a = a$$
$$a * \overline{a} = 0$$
$$a * b = b * a$$
$$a * b * c = (a * b) * c = a * (b * c)$$
$$a * (b + c) = (a * b) + (a * c)$$

$$a + a = a$$
$$a + b = b + a$$
$$a + \overline{a} = 1$$
$$a + b + c = (a + b) + c = a + (b + c)$$
$$a + (b * c) = (a + b) * (a + c)$$

$$a + (a * b) = a$$
$$a * (a + b) = a$$
$$(a * b) + (a * \overline{b}) = a$$
$$(a + b) * (a + \overline{b}) = a$$

$$a + (\overline{a} * b) = a + b$$
$$a * (\overline{a} + b) = a * b$$

2.3 Min und Maxterme

Für jede Zeile aus einer Wahrheitstabeller eine Schaltfunktion gibt es sogenannte *Min* und *Max* Terme.
Der *Min* Term verküpft die Variablen jeder Zeile mit einem *Und*.
Der *Max* Term verküpft die Variablen jeder Zeile mit einem *Oder*.

a	b	c	Min	Max
0	0	0	$(\overline{a} * \overline{b} * \overline{c})$	$(a + b + c)$
0	0	1	$(\overline{a} * \overline{b} * c)$	$(a + b + \overline{c})$
0	1	0	$(\overline{a} * b * \overline{c})$	$(a + \overline{b} + c)$
0	1	1	$(\overline{a} * b * c)$	$(a + \overline{b} + \overline{c})$
1	0	0	$(a * \overline{b} * \overline{c})$	$(\overline{a} + b + c)$
1	0	1	$(a * \overline{b} * c)$	$(\overline{a} + b + \overline{c})$
1	1	0	$(a * b * \overline{c})$	$(\overline{a} + \overline{b} + c)$
1	1	1	$(a * b * c)$	$(\overline{a} + \overline{b} + \overline{c})$

2.4 Normalformen

Mit dem Wissen über Min und Max Terme können wir nun die Normalformen aufstellen.

Angenommen wir haben nun die Wahrheitstabelle

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

So können wir nun Entweder unter zu Hilfe name der *Max* Terme oder der *Min* Terme daraus eine Schaltfunktion extrahieren.

2.4.1 KNF

Konjunktive Normalform
„Produkte von Summen"

a	b	c	y	Max Term
0	0	0	0	$(a + b + c)$
0	0	1	1	
0	1	0	0	$(a + \overline{b} + c)$
0	1	1	1	
1	0	0	1	
1	0	1	1	
1	1	0	0	$(\overline{a} + \overline{b} + c)$
1	1	1	0	$(\overline{a} + \overline{b} + \overline{c})$

$y = (a + b + c) * (a + \overline{b} + c) * (\overline{a} + \overline{b} + c) * (\overline{a} + \overline{b} + \overline{c})$

2.4.2 DNF

Disjunktive Normalform
„Summe von Produkten"

a	b	c	y	Min Term
0	0	0	0	
0	0	1	1	$(a * \overline{b} * c)$
0	1	0	0	
0	1	1	1	$(\overline{a} * b * c)$
1	0	0	1	$(a * \overline{b} * \overline{c})$
1	0	1	1	$(a * \overline{b} * c)$
1	1	0	0	
1	1	1	0	

$y = (a * \overline{b} * c) + (\overline{a} * b * c) + (a * \overline{b} * \overline{c}) + (a * \overline{b} * c)$

$(a * \overline{b} * c) + (\overline{a} * b * c) + (a * \overline{b} * \overline{c}) + (a * \overline{b} * c)$ und $(a + b + c) * (a + \overline{b} + c) * (\overline{a} + \overline{b} + c) * (\overline{a} + \overline{b} + \overline{c})$ sind gleichwertige Schaltfunktionen.

Info: Es ist natürlich günstig immer sich die Variante mit zweniger Zeilen zu suchen. (Weniger 0er oder weniger 1er)

Beide Varianten sind *kanonische* Formen. Wenn man Schaltfunktionen vergleichem möchte so ist es wichtig das sie beide in der selben kanonischen Form vorliegen (sonst vergleicht man Äpfel mit Birnen).

2.5 KV-Diagramm

KV-Diagramme sind Diagramme von Wahrheitstabellen die es ermöglichen die resultierende Schaltung zu vereinfachen und zu minimimieren.

Unser KV Diagramm für 4 Variablen hat die folgende Form:

cd \ ab	00	01	11	10
00				
01				
11				
10				

Info: Wichtig ist das die seitlichen Nummerierungen einen *Gray Code* bilden.
Ein *Gray Code* hat die Eigenschaft das sich pro Zeile nur maximal 1 Bit ändert.
Das würde für das KV Diagram heißen das unter 00 nicht 11 stehen darf weil sich zwei Bits ändern.

Diese Diagram wird nun mit den Werten der Wahrheitstabelle befüllt, zum Beispiel:

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

cd \ ab	00	01	11	10
00	1 Zeile 0	5 Zeile 1	1	0
01	2 Zeile 0	6 Zeile 1	1	0
11	4 Zeile 1	1	1	1
10	3 Zeile 1	0	0	1

Nun suchen wir die *Primimplikanten*. *Implikanten* sind alle möglichen Kombinationen die man einkreisen kann. Wird ein Implikant von einem Anderen gänzlich überdeckt so ist er **kein** *Primimplikant*. In unserem Beispiel währen zum Beispiel die zwei 1er in der oberen Reihe Implikaten aber keine Primimplikanten da sie von dem gelben Kasten voll überdeckt sind.

DNF Variante - 1er - „Summe von Produkten“

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

cd \ ab	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	1	1	1
10	1	0	0	1

Die Primimplikanten lauten:

$B\overline{C}$, BD, CD, $\overline{B}C$

Es ergibt sich folgende minimierte Schaltfunktion:

$y = (B \ast \overline{C}) + (B \ast D) + (\overline{B} \ast C)$

... des Weiteren ergibt sich folgende gleich gute Schaltfunktion:

$y = (B \ast \overline{C}) + (C \ast D) + (\overline{B} \ast C)$

KNF Variante - 0er - „Produkte von Summen“

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

cd \ ab	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	1	1	1
10	1	0	0	1

Die Primimplikanten lauten:

Achtung hier Variablen umdrehen wenn man mit 0ern arbeitet.

$\overline{B} + \overline{C} + D$, $B + C$

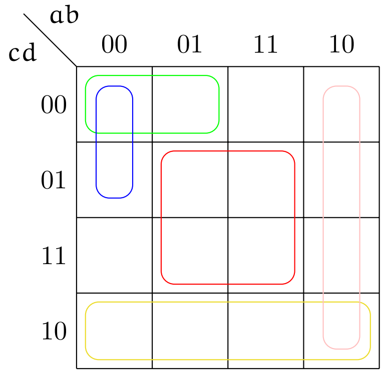
Es ergibt sich folgende minimierte Schaltfunktion:

$y = (\overline{B} + \overline{C} + D) \ast (B + C)$

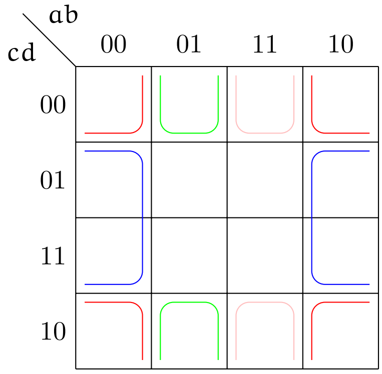
Info: Auch hier gilt: Immer schauen wovon hab ich weniger: 1er oder 0er!

2.5.1 KV-Diagramm - was darf ich einkreisen

Zweier- und Vierergruppen.



Auch über den Rand!



2.6 NAND und NOR Realisierung

Um anschließend mit NAND- oder NOR-Gattern zu realisieren nutzen wir einerseits die tatsache das sich zwei Negationen aufheben anderseits das Gesetz von DeMorgan.

Beispiel 16

Realisieren Sie die Schaltfunktion

$$y = \overline{A} + B + C + D$$

ausschließlich mit NAND-Gattern.

Im ersten Schritt wenden wir eine doppelte Negationen auf unsere Schaltfunktion an. Diese verändert das Verhalten unserer Schaltfunktion bekanntlich nicht.

$$\overline{A} + B + C + D \equiv \overline{\overline{\overline{A} + B + C + D}} \tag{1}$$

Nun können wir das Gesetz von DeMorgan anwenden:

$$\overline{\overline{\overline{A}} * \overline{B} * \overline{C} * \overline{D}} \tag{2}$$

Hier können wir auch schon aufhören, wir haben lauter *Und* ($*$) und diese sind alle verneint, wir haben die Schaltung nun ausschließlich in NAND-Gattern.

3 Endliche Automaten

Endliche Automaten haben endlich viele Zustände, endlich viele Inputs und endlich viele Outputs - deshalb endliche Automaten (FSM - Finite State Maschine).

„Der Zustand eines endlichen Automaten M enthält zu jeder Zeit sämtliche Informationen über die Vergangenheit von M, die für das zukünftige Verhalten von M relevant sind“.

Heißt im Endeffekt das wir uns immer den „Ist-Zustand“ merken müssen, was das heißt sehen wir gleich anhand der Beispiele.

Wir brauchen für unsere Automaten immer: Eingabe(/Input)-Menge, Ausgabe(/Output)-Menge und Zustands(/State)-Menge.

3.1 Moore Automat

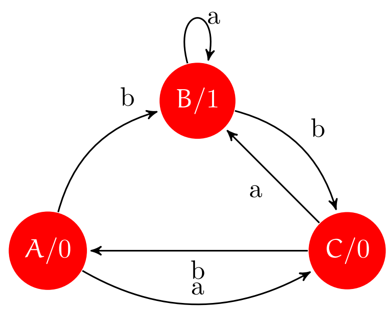
Beim Moore Automaten ist ausgabe ausschließlich vom Zustand abhängig.

Wir haben folgenden Moore Automaten:

Eingabemenge $I = \{a, b\}$

Ausgabemenge $O = \{0, 1\}$

Zustandsmenge $S = \{A, B, C\}$



Zustand	Eingabe a	Eingabe b	Ausgabe
A	C	B	0
B	B	C	1
C	B	A	0

So lang der Zustand B ist wird 1 ausgegeben ansonsten 0. Die Ausgabe ist nur vom derzeitigen Zustand abhängig.

3.2 Mealy Automat

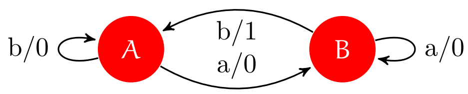
Beim Mealy Automat, anders als beim Moore-Automat, findet die Ausgabe beim Zustandwechsel statt.

Zum Vergleichen nehmen wir einen Mealy Automat der die Zeichenfolge ”ab”erkennt und 1 ausgib wenn diese gefunden wurde.

Eingabemenge $I = \{a, b\}$

Ausgabemenge $O = \{0, 1\}$

Zustandsmenge $S = \{A, B\}$



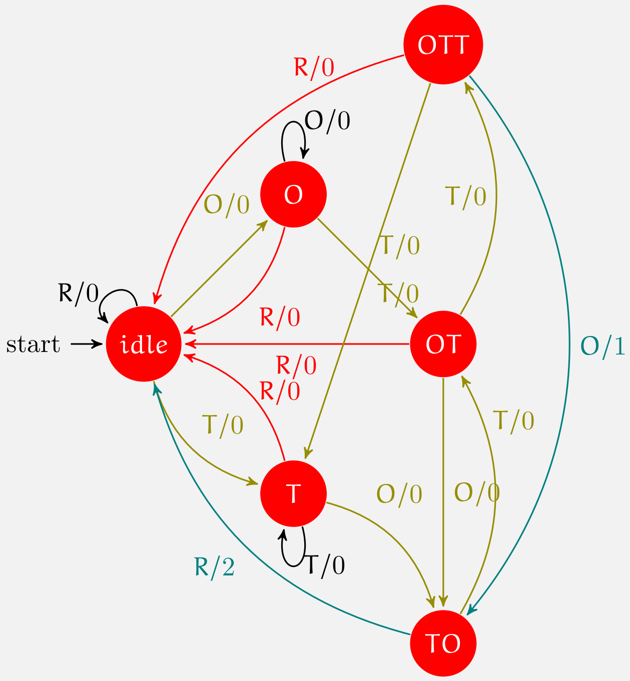
Wir sehen das hier die Ausgaben, anders als beim Moore Automaten beim Zustandwechseln notiert sind.

Beispiel 17

Realisieren Sie einen Mealy Automaten der OTTO (Ausgabe 1) und TOR (Ausgabe 2) überlappend erkennt und geben Sie die Zustandmenge an.

Eingabemenge $I = \{O, T, R\}$ (Alle Buchstaben die vorkommen können)

Ausgabemenge $O = \{0, 1, 2\}$ (Aus der Angabe alle Möglichkeiten)



Für die Überlappung müssen wir uns wenn bei „OTT“ noch ein O kommt merken das wir damit auch „TO“ bereits gelesen haben und uns damit nur mehr ein „R“ auf „TOR“ fehlt.

Wenn wir **keine** Überlappung wollen würden müssten wir direkt von OTT bei einem „O“ zurück auf den „idle“ state gehen.

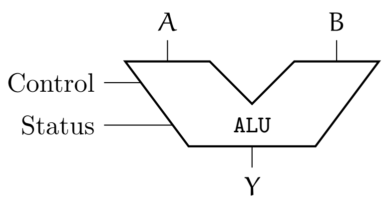
Zustandsmenge $S = \{\mathtt{idle}, O, T, OT, TO, OTT\}$ (Alle Zustände abschreiben)

4 Rechnerarchitektur

4.1 Komponenten

4.1.1 ALU

Die *ALU* kurz für *arithmetic logic unit* kann sowohl logische als auch ariemtische Operationen ausführen.



Durch Control wird die Operation gewählt die, die ALU ausführen soll. In A und B werden die Argumente für die Operation geladen und in Y das Ergebnis der Operation geschrieben. Mit `Status` werden zusätzliche Flags markiert die Aufschluss über das letzte Ergebnis geben können.

4.2 Von-Neumann Architektur

Die Von-Neumann Architektur ist bis heute die Referenz-Architektur für Computer.

Grundsätzlich besteht ein Von-Neumann Computer aus einer *CPU* diese beinhaltet die *ALU* und das *Steuerwerk*. Die CPU ist über *Busse* mit dem *Speicherwerk* und dem *Ein- und Ausgabewerk* verbunden.

CPU Prozessor / *central processing unit* bestehend aus:

- **ALU** arithmetisch-logische Einheit / *arithmetic logic unit* , kann sowohl arithmetische ($+, -, *$,...) als auch logische Operationen ($\wedge, \vee, \neg$...)
- **Steuerwerk** *control unit*, regelt die Befehlsfolge für die ALU, das Programm dirigiert das Steuerwerk, dieses die ALU.

Speicherwerk Arbeitsspeicher / *RAM* hält sämtliche Daten für das Rechenwerk

IO Eingabe- und Ausgabewerk *Input / Output* bestehend aus:

- **Eingabe** Tastatur, Maus, etc
- **Ausgabe** Bildschirm, Drucker, etc

Busse Die Busse verbinden die einzelnen Komponenten